Compressed Pattern Matching For Text And Images

2005

Tao Tao
University of Central Florida

Find similar works at: https://stars.library.ucf.edu/etd

University of Central Florida Libraries http://library.ucf.edu

Part of the Computer Sciences Commons, and the Engineering Commons

STARS Citation

Tao, Tao, "Compressed Pattern Matching For Text And Images" (2005). Electronic Theses and Dissertations. 402.
https://stars.library.ucf.edu/etd/402

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
ABSTRACT

The amount of information that we are dealing with today is being generated at an ever-increasing rate. On one hand, data compression is needed to efficiently store, organize the data and transport the data over the limited-bandwidth network. On the other hand, efficient information retrieval is needed to speedily find the relevant information from this huge mass of data using available resources.

The compressed pattern matching problem can be stated as: given the compressed format of a text or an image and a pattern string or a pattern image, report the occurrence(s) of the pattern in the text or image with minimal (or no) decompression. The main advantages of compressed pattern matching versus the naïve decompress-then-search approach are: First, reduced storage cost. Since there is no need to decompress the data or there is only minimal decompression required, the disk space and the memory cost is reduced. Second, less search time. Since the size of the compressed data is smaller than that of the original data, a searching performed on the compressed data will result in a shorter search time.

The challenge of efficient compressed pattern matching can be met from two inseparable aspects: First, to utilize effectively the full potential of compression for the information retrieval systems, there is a need to develop search-aware compression algorithms. Second, for data that is compressed using a particular compression technique, regardless whether the compression is search-aware or not, we need to develop efficient searching techniques. This means that techniques must be developed to search the compressed data with no or minimal decompression and with not too much extra cost.
Compressed pattern matching algorithms can be categorized as either for text compression or for image compression. Although compressed pattern matching for text compression has been studied for a few years and many publications are available in the literature, there is still room to improve the efficiency in terms of both compression and searching. None of the search engines available today make explicit use of compressed pattern matching. Compressed pattern matching for image compression, on the other hand, has been relatively unexplored. However, it is getting more attention because lossless compression has become more important for the ever-increasing large amount of medical images, satellite images and aerospace photos, which requires the data to be losslessly stored. Developing efficient information retrieval techniques from the losslessly compressed data is therefore a fundamental research challenge.

In this dissertation, we have studied compressed pattern matching problem for both text and images. We present a series of novel compressed pattern matching algorithms, which are divided into two major parts. The first major work is done for the popular LZW compression algorithm. The second major work is done for the current lossless image compression standard JPEG-LS. Specifically, our contributions from the first major work are:

1. We have developed an “almost-optimal” compressed pattern matching algorithm that reports all pattern occurrences. An earlier “almost-optimal” algorithm reported in the literature is only capable of detecting the first occurrence of the pattern and the practical performance of the algorithm is not clear. We have implemented our algorithm and provide extensive experimental results measuring the speed of our algorithm. We also developed a faster implementation for so-called “simple patterns”. 
The simple patterns are patterns that no unique symbol appears more than once. The algorithm takes advantage of this property and runs in optimal time.

2. We have developed a novel compressed pattern matching algorithm for multiple patterns using the Aho-Corasick algorithm. The algorithm takes $O(mt+n+r)$ time with $O(mt)$ extra space, where $n$ is the size of the compressed file, $m$ is the total size of all patterns, $t$ is the size of the LZW trie and $r$ is the number of occurrences of the patterns. The algorithm is particularly efficient when being applied on archival search if the archives are compressed with a common LZW trie.

All the above algorithms have been implemented and extensive experiments have been conducted to test the performance of our algorithms and to compare with the best existing algorithms. The experimental results show that our compressed pattern matching algorithm for multiple patterns is competitive among the best algorithms and is practically the fastest among all approaches when the number of patterns is not very large. Therefore, our algorithm is preferable for general string matching applications. LZW is one of the most efficient and popular compression algorithms used extensively and both of our algorithms require no modification on the compression algorithm. Our work, therefore, has great economical and market potential.

Our contributions from the second major work are:

1. We have developed a new global context variation of the JPEG-LS compression algorithm and the corresponding compressed pattern matching algorithm. Comparing to the original JPEG-LS, the global context variation is search-aware and has faster encoding and decoding speeds. The searching algorithm based on the global-context variation requires partial decompression of the compressed image. The experimental results show that it improves the search speed by about 30% comparing to the
decompress-then-search approach. Based on our best knowledge, this is the first two-dimensional compressed pattern matching work for the JPEG-LS standard.

We have developed a two-pass variation of the JPEG-LS algorithm and the corresponding compressed pattern matching algorithm. The two-pass variation achieves search-awareness through a common compression technique called semi-static dictionary. Comparing to the original algorithm, the compression of the new algorithm is equally well but the encoding takes slightly longer. The searching algorithm based on the two-pass variation requires no decompression at all and therefore works in the fully compressed domain. It runs in time $O(n_c + m_c + nm + m^2)$ with extra space $O(n + m + m_c)$, where $n$ is the number of columns of the image, $m$ is the number of rows and columns of the pattern, $n_c$ is the compressed image size and $m_c$ is the compressed pattern size. The algorithm is the first known two-dimensional algorithm that works in the fully compressed domain.
Dedicated to Chan B. Huynh
ACKNOWLEDGEMENTS

I would like to express my deep and sincere gratitude to my academic advisor Dr. Mukherjee for his guidance, encouragement and support during the seven years of my graduate study. Without his encouragement and direction, this work would not have been possible. I would like to thank my committee members Dr. Kien A. Hua, Dr. Ratan Guha and Dr. Guy Schiavone for taking their precious time to review my dissertation and for giving insightful comments on my work.

I would like to take this opportunity to thank all the members of the M5 research group at the University of Central Florida namely, Nan Zhang, Ravi Vijaya Satya and Weifeng Sun for their help and support. It is both pleasant and encouraging to work with these talented people. I would also like to thank all my friends in UCF, in particular, Ning Jiang, Jun Li, Dr. Zhiguang Xu, Rong Wang, Ping Wu, Yi Ma, Yixiao Yang, Yue Rong, Hongliang Gao and Chun Huang for their consistent support and for making my life more colorful. A special acknowledgement goes to Miss Chan B. Huynh for her constant moral support throughout these years. It has been quite a time and I would not trade anything for it.

Finally, I am deeply grateful to my parents and brother for their encouragement, support and love.
# TABLE OF CONTENTS

LIST OF FIGURES ...................................................................................................................... xii
LIST OF TABLES .......................................................................................................................... xiv

CHAPTER 1.  INTRODUCTION .............................................................................................. 1
  1.1.  Background ................................................................................................................ ..... 1
  1.2.  Problem Definition ......................................................................................................... 6
  1.3.  Contributions ............................................................................................................... 9
  1.4.  Dissertation Outline ..................................................................................................... 11

CHAPTER 2.  BASIC CONCEPTS ......................................................................................... 13
  2.1.  Data Compression ......................................................................................................... 13
  2.1.1  Information Theory ................................................................................................... 14
  2.1.2  Text Compression Algorithms .................................................................................. 15
  2.1.3  Lossless Image Compression .................................................................................... 24
  2.2.  Pattern Matching .......................................................................................................... 29
  2.2.1  Single Pattern Matching Algorithms ....................................................................... 29
  2.2.2  Multiple Pattern Matching ....................................................................................... 34
  2.2.3  Two Dimensional Pattern Matching ......................................................................... 36

CHAPTER 3.  COMPRESSED PATTERN MATCHING....................................................... 40
  3.1.  Introduction ............................................................................................................... 40
  3.2.  CPM Algorithms ......................................................................................................... 46
  3.2.1  CPM For Non-adaptive Compressions .................................................................... 47
  3.2.2  CPM For Adaptive Compressions .......................................................................... 52
3.3. Performance Measurement ........................................................................................... 57
   3.3.1 Data Compression ................................................................................................... 57
   3.3.2 Pattern Matching ................................................................................................. 59
   3.3.3 Compressed Pattern Matching .............................................................................. 59

CHAPTER 4. CPM IN LZW COMPRESSED FILES ............................................................ 61

4.1. Extension Of Amir’s Algorithm ............................................................................... 62
   4.1.1 LZW Compression And Amir’s Algorithm ......................................................... 62
   4.1.2 Reporting All Pattern Occurrences .................................................................... 67
   4.1.3 A Simple Implementation For “Simple Patterns” .............................................. 69
   4.1.4 The Proposed Algorithm .................................................................................... 70
   4.1.5 A Discussion On Multiple Pattern Matching .................................................... 72

4.2. Multiple-Pattern Matching ....................................................................................... 73
   4.2.1 Aho-Corasick Algorithm .................................................................................... 73
   4.2.2 Searching Multiple Patterns Using Aho-Corasick Algorithm ............................ 75
   4.2.3 Analysis And Comparison ................................................................................... 80
   4.2.4 Applying On Archival Search ............................................................................ 82

4.3. Experimental Results ............................................................................................... 83
   4.3.1 Compression Performance .................................................................................. 83
   4.3.2 Encoding And Decoding Time ............................................................................ 85
   4.3.3 Search Performance For a Single File ............................................................... 86
   4.3.4 Search performance For Archives .................................................................... 88
   4.3.5 Comparison With Kida’s Algorithm ................................................................. 89

4.4. Conclusion And Future Work .................................................................................. 90
LIST OF FIGURES

Figure 1-1 Information explosion ................................................................................................... 2
Figure 2-1 LZW trie example ....................................................................................................... 21
Figure 2-2 Casual template used for context modeling and prediction ......................................... 24
Figure 2-3 KMP algorithm illustration ........................................................................................ 30
Figure 2-4 Pattern shifting in KMP algorithm ............................................................................. 31
Figure 2-5 Boyer-Moore algorithm illustration .......................................................................... 32
Figure 2-6 Good-suffix illustration .............................................................................................. 32
Figure 2-7 Pattern shifting in good-suffix .................................................................................... 33
Figure 2-8 Pattern shifting in good-suffix .................................................................................... 33
Figure 2-9 AC automaton example ............................................................................................. 35
Figure 2-10 Bird's algorithm ....................................................................................................... 37
Figure 2-11 Primary rows and secondary rows .......................................................................... 39
Figure 3-1 False match in Huffman code .................................................................................... 51
Figure 4-1 LZW trie structure ..................................................................................................... 63
Figure 4-2 Pattern inside a node ................................................................................................ 67
Figure 4-3 PIC flag update ......................................................................................................... 68
Figure 4-4 An Aho-Corasick automaton ..................................................................................... 73
Figure 4-5 The AC automaton for patterns $aa$, $ab$ and $abc$ ..................................................... 76
Figure 4-6 The initial LZW trie with state-transition lists ............................................................ 77
Figure 4-7 The LZW trie with state-transition lists ..................................................................... 78
Figure 4-8 Search performance .................................................................................................. 86
### LIST OF TABLES

Table 2-1 Lossless JPEG prediction modes .................................................................................. 25  
Table 4-1 State transition table of the initial LZW trie ................................................................. 78  
Table 4-2 State transition example ........................................................................................... 79  
Table 4-3 Comparison with Kida's algorithm ............................................................................. 82  
Table 4-4 Compression performance ......................................................................................... 84  
Table 4-5 Encoding and decoding time ...................................................................................... 85
CHAPTER 1. INTRODUCTION

In this chapter, the concept of compressed pattern matching for lossless data compression is introduced. The problem is formally defined and variations of the problem are listed. Applications where this research may apply are introduced and the main contributions of this work are clearly stated. Finally, the outline of the dissertation is given.

1.1. Background

We are in a time of “information explosion”. A team of researchers from the School of Information Management and Systems, University of California, Berkeley released a new study [Lyman 2003] that chronicles the information explosion over the past several years. According to the team, during the period of 1999 to 2002, "new stored information grew about 30% a year." Additional facts: "Print, film, magnetic, and optical storage media produced about 5 exabytes (1 exabyte= 10^{18} bytes which is approximately equal to all words spoken by human beings) of new information in 2002. Ninety-two percent of the new information was stored on magnetic media, mostly in hard disks … The World Wide Web contains about 170 terabytes (1,024 gigabytes or 2^{20} bytes) of information on its surface, in volume this is seventeen times the size of the Library of Congress print collections….Email generates about 400,000 terabytes of new information each year worldwide." Figure 1-1 illustrates the rapid growth of information over time.
While a good fraction of this information is of transient interest, useful information of archival value continues to accumulate. For instance, the TREC database (Text Retrieval Conference database, co-sponsored by the National Institute of Standards and Technology and U.S. Department of Defense. Its purpose was to support research within the information retrieval community and provides the infrastructure for large-scale evaluation of text retrieval methodologies) alone holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. Therefore, efficient storage, management, organization, and transmission of the data from one point to the other on data communications links with limited bandwidth are a big challenge.

Data compression is the general term for the various algorithms and programs developed to address this problem. Data compression can be defined as: given a set of data, data compression is to transform the data into a different set of data. The transformed data size is smaller than the original set of data and the original data should be able to reconstructed from the transformed data, losslessly or in an acceptable lossy manner. The basic strategy data compression utilizes is the elimination or reduction of redundancy in the information. A compression program is used to convert data from its
There are many different ways that data compression algorithms can be categorized. For example, a data compression algorithm can be classified as either lossless or lossy. A lossless compression algorithm means that the restored data file is identical to the original. This is absolutely necessary for many types of data such as executable code, word processing files, tabulated numbers, medical images, astronomical images, fingerprint data and data bases containing mostly vital numerical data, tables and text information etc. It is not affordable to misplace even a single bit of this sort of information. In comparison, a lossy compression algorithm means that the restored data file has reasonable degradation to the original file. The idea behind lossy compression is that some type of data does not have to be kept in perfect condition for storage or transmission. A good example of this type of data is most general-purpose photos. Like many other acquired signals, when in its original format, a photograph inherently contains a certain amount of noise (in this case, a noise means redundant information that is not perceptible by Human Visual System). If any change made to the photo during the compression/decompression process eliminates or reduces the noise, no harm is done. Not surprisingly, lossy techniques are much more effective at compression than lossless methods. In another popular classification of data compression algorithms, an algorithm is classified based on the target data it is designated to compress. For example, if the targets are textural information, a data compression algorithm is called a text compression algorithm; if the targets are images, a data compression algorithm is called an image compression algorithm; if the targets are graphical geometric mesh, a data compression algorithm is called a mesh compression algorithm; and so on.
compression algorithm is called an mesh compression algorithm. Text compression can be further classified as dictionary based methods, statistical methods and sorted-contexts methods while image compression can be further classified as predictive methods, block-transform methods, fractal based methods, sub-band methods etc.

To understand why data compression is important and to show the effectiveness of lossless versus lossy compression, consider image transmission problem over the world wide web (WWW) as an example. Suppose we need to download a high-resolution color image over a computer's 33.6 kbps modem and the image in its un-compressed format contains about 600 kilobytes of data. If it has been compressed using a lossless technique (For instance a common technique on the WWW called GIF), it will be about one-half the original size, or 300 kilobytes. If the image is compressed by a lossy compression algorithm, such as JPEG, it will be about 50 kilobytes. The meaning of this is, for the end user, the download times for these three equivalent files are 142 seconds, 71 seconds, and 12 seconds, respectively. That's a big difference! Today, data compression has been extensively used in data storage, transmission over the network, database management, spreadsheet applications, backup utilities etc. and it impacts many people’s daily activities. Given the continued increase in the amount of data that needs to be transferred and/or archived on personal computers or servers nowadays, the importance of data compression is likely to increase in the foreseeable future.

On the other hand, the problem of efficient retrieval of information from the huge mass of data poses another big challenge. We must have means to speedily find the information we need from this huge mass of data using available resources (such as memory). Today, search engines have become the most popular and powerful tools for
hunting relevant documents on the Internet. According to the statistics, search engines such as Google contribute about 50% of all traffic on the Internet and approximately 30% of all Internet users in North America search on Google. For many people, search engines have become an essential part of their everyday life. Meanwhile, a single site such as a library database may also contain large collections of data and thus an efficient search mechanism is also necessary. Even for users for work only on their personal computers, the importance of information retrieval cannot be neglected. The most popular text-searching algorithm has been implemented into almost every commercial product of text editor, compiler etc. 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.

Pattern matching (or string searching) algorithms constitute the core part of the searching mechanism. The pattern matching problem may be formulated as follows [Stephen94]: given pattern string x, with |x|=m, and text string y, with |y|=n, where m, n>0 and m≤n, if x occurs as a sub-string of y then determine the position within y of the first occurrence of x, i.e., return the least value of i such that y(i, i+m-1)=x(1,m). The problem is generally extended to find all of the occurrences of x in y instead of only the first. There are also many variations of this problem, such as the exact pattern matching problem, the approximate pattern matching problem, the wildcard pattern matching problem, the multiple-pattern matching problem and the two-dimensional (or even higher dimension) pattern matching problem. The pattern matching techniques are of important in many application areas such as data processing, information retrieval, text editing and
word processing, linguistic analysis, and areas of molecular biology such as DNA sequence analysis.

Sometimes, the need of compression and the need of fast information retrieval appear to be conflicting each other. Since most string searching algorithms operate on the “raw” text strings, for fast searching, it is easier to keep the data in its uncompressed format. If string searching has to be done for compressed data, the intuitive way is to decompress the data first and then search. However, 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, even when it is being searched. This brings up a new problem called the compressed pattern matching and this new problem is our research subject in this dissertation.

1.2. Problem Definition

The compressed pattern matching (CPM) problem is generally defined as: given the compressed format S.Z of a text string (or an image) S and a pattern string (or a sub-image) P, report the occurrences of P in S with minimal (or no) decompression of S.Z.

Not surprisingly, the main advantages of compressed pattern matching are:

1. Reduced storage cost. Since there is no need to decompress the data or there is only minimal decompression required, the disk space and the memory cost is reduced.

2. Less search time. Since the size of the compressed data is smaller than that of the original data, a search performed on the compressed data will result in a shorter search time. Ideally, if the size of the compressed data is $m$ times smaller than that
of the original data, the search time in the compressed domain is expected to be $m$ times faster than that in the uncompressed domain.

Similar to its uncompressed counterpart, the CPM problem also has many variations such as the approximate CPM, wildcard CPM, CPM with “don’t cares”, multiple-pattern CPM and two-dimensional CPM problems. Besides, since the problem of CPM is closely related to data compression, CPM methodologies may also be categorized with respect to compression. For example, if no decompression is required at all during the search process of a CPM, the CPM is called a full CPM. If a CPM is to search in the text/image (where the text/image is compressed by a text/image compression algorithm) then the CPM is called a text/image-based CPM. Generally, a text-based CPM is a one-dimensional CPM problem while an image-based CPM is a two-dimensional CPM problem. The full CPM problem and the two-dimensional CPM problem are generally considered to be more challenging and are comparatively less developed.

The efficiency of a compressed pattern matching algorithm is primarily evaluated based on both the compression ratio and the search speed. The challenge of efficient CPM can be met from two inseparable aspects: First, to utilize effectively the full potential of compression for the information retrieval systems, there is a need to develop search-aware compression algorithms. A search-aware compression algorithm is one that the compression is performed in such a way that later searching on the compressed data is supported. One way to make the compression search-aware is to use the random-access property, which refers to being able to randomly access and partially decode the compressed data, starting from any location in the compressed text. The random-access
property is highly desirable for efficient retrieval and is also required in many applications. Second, for data that is compressed using a particular compression technique, regardless whether the compression is search-aware or not, we need to develop efficient searching techniques. This means that techniques must be developed to search the compressed data with no or minimal allowed decompression and with not too much extra cost. The measurements of CPM algorithms will be further addressed in Chapter 3.

The application areas of CPM include information retrieval and database management. Lossless image compression based CPM is particularly useful for searching in medical imaging, aerospace photos and satellite images where the size of these data usually are extremely large and need to be kept in the compressed format.

Our motivations for this research are: first, although the text-based CPM has been studied for a few years and many publications are available in the literature, there is still room to improve the efficiency in terms of both compression and searching. A major portion of the research on compressed pattern matching came out of research conducted at UCF. Among the current CPM algorithms, some have good compression but the search performance is not satisfactory; some have pleasing search speed but the high search performance is gained over the sacrifice of compression. Some present both excellent compression and search performance, but are practically infeasible. Because of the above reasons, none of the search engines available today make explicit use of compressed pattern matching. Second, lossless compression has become more important since it is used for the large amount of medical images, satellite images and aerospace photos, which requires the data to be losslessly stored. The need of efficient compressed pattern
matching in these kinds of data and the relative poor development of two-dimensional compressed pattern matching points a new research direction for us.

### 1.3. Contributions

In this dissertation, we present two major works of compressed pattern matching, which will enrich the family of compressed pattern matching algorithms. The first major work is done for the popular LZW compression algorithm and it includes a series of algorithms, developed under different assumptions and/or for different objectives. The work includes an extension of Amir’s well-known “almost-optimal” algorithm [Amir96]. The original algorithm has been improved to search not only the first occurrence of the pattern but also all other occurrences. A faster implementation for so-called “simple patterns” is also proposed. The work also includes a novel multiple-pattern matching algorithm using the Aho-Corasick algorithm. The algorithm takes $O(mt+n+r)$ time with $O(mt)$ extra space, where $n$ is the size of the compressed file, $m$ is the total size of all patterns, $t$ is the size of the LZW trie and $r$ is the number of occurrences of the patterns.

Extensive experiments have been conducted to test the performance of our algorithms and to compare with other well-known compressed pattern matching algorithms, particularly the BWT-based algorithms and another similar multiple-pattern matching algorithm by Kida et. al. [Kida98] [Kida00] that also uses the Aho-Corasick algorithm on the LZW compressed data. The results showed that our multiple-pattern matching algorithm is competitive among the best compressed pattern-matching algorithms and is practically the fastest among all approaches when the number of patterns is not very large. Therefore, our algorithm is preferable for general string matching applications. The
proposed algorithm is efficient for large files and it is particularly efficient when being applied on archive search if the archives are compressed with a common LZW trie. LZW is one of the most efficient and popular compression algorithms used extensively and our method requires no modification on the compression algorithm. The work reported here, therefore, has great economical and market potential.

The second major work is done for the current lossless image compression standard JPEG-LS. Based on our best knowledge, this is the first two-dimensional compressed pattern matching work based on the JPEG-LS standard. A series of algorithms are proposed, including: a global context variation of the JPEG-LS algorithm and the corresponding searching algorithm, a two-pass variation of the JPEG-LS algorithm and the corresponding searching algorithm. Among them, the global context algorithm can be regarded as a simplified version of the original algorithm, which achieves search-awareness by removing the global-context dependency from each pixel’s prediction. As the result, the compression of the algorithm is slightly worsened but the encoding and decoding speeds are both improved. The searching algorithm based on the global-context variation requires partial decompression of the compressed image. The two-pass variation, on the other hand, achieves search-awareness through a common compression technique called the semi-static dictionary. Comparing to the original algorithm, the compression of the new algorithm is equally well but the encoding takes slightly longer. The searching algorithm based on the two-pass variation requires no decompression at all and therefore works in the fully compressed domain. It runs in time $O(n_c + m_c + nm + m^2)$ with extra space $O(n + m + m_c)$, where $n$ is the number of columns of the image, $m$ is the number of rows and columns of the pattern, $n_c$ is the compressed image
size and $m_c$ is the compressed pattern size. The algorithm is the first known two-dimensional CPM algorithm that works in the fully compressed domain. Several important problems in the two-dimensional CPM algorithm are addressed: the multiple-row searching problem, the vertical alignment problem and the boundary identification problem.

1.4. Dissertation Outline

The organization of the rest of this document is described in the following. In chapter 2, we briefly discuss basic concepts that are essential to our research. First, some of the most important text and lossless image compression algorithms are introduced. The purpose of introducing these algorithms is to understand how each particular compression algorithm works and the internal structures/organizations of the compressed data. Understanding of the compression mechanism and the internal representation of the compressed data are crucial to the development of both search-aware compression algorithms and compressed pattern matching algorithms. Second, pattern matching algorithms, as in the raw data domain, are introduced as well in this chapter. Understanding of how searching algorithms work will help us to better apply the algorithms in the compressed domain or design search mechanism that are suitable for the compressed data. In chapter 3, a survey of the most important pattern searching algorithms is given. The survey will help us to understand the overall situation of this relative new area and the existing works will be the base of our research work. Also in Chapter 3, general performance measurements of the searching algorithms in compressed domain are discussed. Chapter 4 and chapter 5 form the core part of this dissertation as
our original works are presented in these two chapters. In Chapter 4, a series of compressed pattern matching algorithms are developed for the LZW compression while in chapter 5, a full compressed pattern matching system is developed for the current lossless compression standard JPEG-LS. Chapter 6 concludes the dissertation.
CHAPTER 2. BASIC CONCEPTS

In this chapter, we introduce basic concepts that are essential to our research. We first give a brief introduction to the theory of compression - the Information Theory. After that, some of the most important text compression algorithms and lossless image compression algorithms are presented. The understanding of the compression algorithms is not only essential for developing search-aware compression algorithms, but also important for understanding and development of compressed pattern matching algorithms because nearly all compressed pattern matching algorithms are dependent on specific compression algorithms. We then introduce some of the most important pattern matching algorithms in the uncompressed domain.

2.1. Data Compression

In this section, we first give an introduction to Information Theory, which is then followed by brief introduction to some of the most important text compression algorithms and lossless image compression algorithms. These algorithms include: the LZ family algorithms, BWT algorithm, lossless JPEG, CALIC algorithm and JPEG-LS (LOCO-I). Among them, the LZ algorithms are dictionary-based compressions, the BWT algorithm is a sorted-contexts compression while the rest are predictive image compressions. We are not giving an exhausted list of all important compression algorithms. These algorithms are chosen both because of their importance in the data compression family and their relevance to this research.
2.1.1 Information Theory

The Information Theory was introduced by Shannon [Shannon48]. It provides the quantitative analysis of data compression, specifically, the quantitative analysis of how much the data can be compressed.

- **Self-information**

  We define $-\log(P_A)$ as the *self-information* of event $A$ if $P_A$ is the probability of $A$.

  The above definition tells us that if the probability of $A$ decreases, then the self-information of $A$ increases.

- **Entropy**

  For a general source $S$ with alphabet $A = \{1, 2, \ldots, m\}$ that generates a sequence $\{X_1, X_2, \ldots, X_n\}$, $X_i \in A$ for $1 \leq i \leq n$, the entropy of $S$ is:

  $$H(S) = \lim_{n \to \infty} \frac{1}{n} G_n$$

  Where

  $$G_n = - \sum_{i_1=1}^{i=m} \sum_{i_2=1}^{i=m} \cdots \sum_{i_n=1}^{i=m} P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n) \log P(X_1 = i_1, X_2 = i_2, \ldots, X_n = i_n)$$

  If each element in the sequence is independent and identically distributed (IID), we can show that

  $$G_n = -n \sum_{i_1=1}^{i=m} P(X_1 = i_1) \log P(X_1 = i_1)$$

  And thus
\[ H(S) = - \sum_{i=1}^{m} P(i) \log P(i) \]  

(2.2)

The H(S) in Equation 2.1 is called \textit{entropy} and the H(S) in Equation 2.2 is called the \textit{first-order entropy} of the source.

Entropy (first-order entropy) can be interpreted as the average information associated with S. The formula of entropy is not defined randomly – it was founded to meet several requirements of the “average information”.

In the above equations, if we use log base 2, the unit is \textit{bits}; if we use log base \(e\), the unit is \textit{nats}. The concept of entropy is very significant in the context of data compression because of the following property:

The best a lossless compression algorithm can do is to encode the output of a source such that the average number of bits equals the entropy of the source.

It is easy to prove from the above equations that a source that has a more “skewed” set of probabilities – certain events occur with much higher probability than others, has lower entropy than a source that has a less “skewed” set of probabilities. Thus, changing the probability distribution of the source to make it more “skewed” has been a very important way to achieve better compression.

\subsection*{2.1.2 Text Compression Algorithms}

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.

- **LZ77 Algorithm**

  The LZ77 [Ziv77] algorithm is the first text compression algorithm proposed in the LZ (Zev-Lempel) family compression algorithms. It is incorporated by most commercial compression utility such as ZIP and ARJ. LZ77 algorithm is a dictionary-based algorithm and the dictionary is both implicit and dynamic, as can be seen from the followings. The algorithm is very simple: *if the current symbol(s) has appeared somewhere before, then output a reference to that position; otherwise output a null-reference and the first mismatched symbol*. Therefore, a LZ77 code is denoted as \((b, l) c\), which tells the decoder to “go back \(b\) characters, copy \(l\) characters from there and then append a \(c\) character”.

  For example, string “AABCBBABC”, will be encoded as shown below:

<table>
<thead>
<tr>
<th>Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Code</td>
<td>(0,0)A</td>
<td>(1,1)B</td>
<td>(0,0)C</td>
<td>(2,1)B</td>
<td>(5,2)C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  The encoder starts at position 1. Since at the beginning there is no “dictionary” (because no data has been coded yet), a null-reference \((0,0)\) and the first mismatched symbol ‘A’ are set. Therefore, the code is \((0,0) A\). At position 2, since character ‘A’ has occurred previously but string “AB” has not, the encoder sends a code \((1,1) B\) to tell the decoder to “go back 1 position, copy 1 symbol and append a ‘B’ symbol”. The encoder then moves to position 4 (position 3 is skipped since symbol ‘B’ has been sent) and continues until all characters have been encoded.
The decoding process is very simple since the triple \((b, l)\) \(C\) tells the decoder exactly what to do.

LZ77 algorithm provides good compression performance but the encoding takes long time since a large number of comparisons need to be done. In contrast, the decoder is very fast since it only does simple “copy and paste”.

Compressed pattern matching for LZ77 compressed text is not straightforward. It is not possible to decode the sub-string starting from a randomly selected point of the compressed text without knowing the symbols appearing before that point. On the other hand, if we want to compress the pattern and compare it with the compressed text, we are not able to do that because we will have no knowledge of the “dictionary” being used by the text and, worse than that, the dictionary changes at every position in the compressed text.

One way to solve the above problem is to put some restrictions on the encoding process. For example, when word-based searching is preferred, we may force LZ77 encoder to start/finish the match on word boundaries such that the word-based search will be simplified. The same decoder can be used to decode such compressed file without any modification. An alternative is to use a static dictionary. In this way, when the compressed text is being scanned to search the pattern in the same direction as it was being encoded, we may be able to collect the information we need to encode our pattern or decode the sub-string being compared, without decoding or fully decoding all the symbols that have been scanned.
• **LZSS Algorithm**

LZSS algorithm [Ziv77] is a modified version of LZ77 algorithm. The following problems may affect the efficiency of the LZ77 algorithm:

1. When there is no match or only one match, it is too expensive to represent the characters (one character when no match and two characters when one match is found) using code \((b,l)\)

2. Using code \(\langle b, l \rangle c\) is probably not efficient since \(c\) might be the first symbol of the next match.

To solve the above problems, the LZSS algorithm simply sends a reference \(\langle b, l \rangle\) when a match is found and it sends the original symbol when no match is found. Furthermore, a minimal matching length \(M\) is applied. When encoder looks for match symbols in the “dictionary”, only the match that has a length larger than \(M\) is considered.

For the same example, if the minimal matching length is 2. the encoding of LZSS is shown below.

<table>
<thead>
<tr>
<th>Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Code</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>(5,2)</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
• **LZ78 Algorithm**

Different than LZ77 and LZSS algorithms, LZ78 algorithm [Ziv78] uses an explicit dictionary. The idea is very simple: “*Why not store the strings in a dictionary and outputs its index in the dictionary when a match is found. Anyway an index such as 7 is shorter than a reference such as (5, 2)*”. At the beginning of encoding, the dictionary is empty. Every time when a mismatch happens, i.e. no match is found in the dictionary for the current string, the encoder inserts the current string into the dictionary and outputs the index of the matched part in the dictionary, followed by the mismatched symbol.

The encoding is illustrated in the following table.

<table>
<thead>
<tr>
<th>Position</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>Code</td>
<td>0A</td>
<td>1B</td>
<td>0C</td>
<td>0B</td>
<td>4A</td>
<td>4C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dictionary</td>
<td>A</td>
<td>AB</td>
<td>C</td>
<td>B</td>
<td>BA</td>
<td>BC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The dictionary will not be sent to the decoder since the decoder can re-construct the same dictionary in the same fashion.

The compression performance of LZ78 is close to LZ77 and LZSS. However, since a tree structure is utilized in LZ78 to store the dictionary, the search speed is improved.

Although LZ78 constructs the dictionary explicitly, the way it constructs the dictionary is similar to that of LZ77 in that they both use the symbols that have been coded. Therefore, randomly accessing the compressed text is still impossible for LZ78.

• **LZW Algorithm**

LZ78 algorithm presents the following problems:
1. Since the alphabet is a small finite set and the symbols from the alphabet frequently appear in the text, it might be better to store the alphabet in the dictionary at the very beginning.

2. Using code “index, c” may not be efficient if c is frequently the first symbol of the next match.

Let $S=c_1c_2c_3 \ldots c_u$ be the uncompressed text of length $u$ over alphabet $\Sigma=\{a_1, a_2, a_3, \ldots, a_q\}$, where $q$ is the size of the alphabet. We denote the LZW compressed format of $S$ as $S.Z$ and each code in $S.Z$ as $S.Z[i]$, where $1 \leq i \leq n$.

The LZW compression algorithm [Welch84] uses a tree-like data structure called a “trie” to store the dictionary generated during the compression processes. Each node on the trie contains:

- A node number, which is a unique ID in the range of $[0, n+q]$ and,
- A label, which is a symbol from the alphabet $\Sigma$.
- A chunk, which is defined as the string on the path from the root to the node.

At the beginning of the compression, the trie has $q+1$ nodes, including a root node with node number 0 and a NULL label and $q$ child nodes each labeled with a unique symbol from the alphabet. During the compression, LZW algorithm scans the text and finds the longest sub-string that appears in the trie as the chunk of some node $N$ and outputs $N$ to $S.Z$. The trie then grows by adding a new node under $N$ and the new node’s label is the next un-encoded symbol in the text. Obviously, the new node’s chunk is node $N$’s chunk appended by the new node’s label. Figure 2-1 illustrates the trie structure.
The decoder constructs the same trie and uses it to decode $S.Z$. Both the compression and decompression (and thus and trie construction) can be done in time $O(u)$.

The LZW trie can be reconstructed from $S.Z$ in time $O(n)$ without explicitly decoding $S.Z$ in the following manner [5]: When the decoder receives a code $S.Z[i]$, assuming $S.Z[i-1]$ has already been received in the previous step ($2 \leq i \leq n$), a new node is created and added as a child of node $S.Z[i-1]$. The node number of the new node is $i-1+q$ and the label of the new node is the first symbol of node $S.Z[i]$’s chunk.

A popular compression utility that is based on LZW is COMPRESS.

Amir [Amir96] proposed an algorithm to implicitly decode the text and build the dictionary for compressed pattern matching. We will discuss the problem further in Chapter 3 and 4.

- **BWT Algorithm**

  BWT (Burrow-Wheeler-Transform) [Burrows94] [Balkenhol00] [Fenwick96], is a very interesting transform that re-orders the symbols in the text by sorting symbols based on their contexts. For instance, in string “abccdabcd”, both the first ‘a’ and the second ‘a’ have “bc” as their right-hand neighbors, therefore a “sorted-context” transform
may put them together in the re-ordered string. Since symbols that have similar context are very likely to be the same symbol (such as the two ‘a’ symbols in the previous example), the output from a “sorted-context” transform is suitable for RLE (Run Length Encoding). The following example illustrates how BWT works on string “AABCBBABC”.

The forward transform include the following steps:

1: Cyclically rotate the original string and build a matrix that consists of the original string and all possible rotated strings.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. Sort the strings lexicographically.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

3: Output the last column and the row index of the original string in the new matrix. In the example, the output is string ”CBABCAABB” and row 1.
BWT is reversible. In the inverse transform, the symbols in the output are sorted to obtain a new string, which is the first column in sorted matrix. In this example, “CBABCAABB” is sorted to “AAABBBBCC”. The algorithm then computes a transformation vector V that provides a one-to-one mapping between the symbols in the two strings. If L is the last column and F is the first column of the sorted matrix, we have \( F[V[j]] = L[j] \) and for a given symbol \( c \) in the alphabet, if \( L[j] \) is the \( i \)-th occurrence of \( c \) in \( L \), then \( F[V[j]] \) is the \( i \)-th occurrence of \( c \) in \( F \). The original string then can be easily computed from \( L \), \( F \), \( V \) and the index of \( L \) in the sorted matrix since \( L[V[j]] \) cyclically precedes \( L[j] \) in the original string.

In practical implementation, \( V \) can be computed as \( V[i] = R[i] + C[L[i]] \) for \( 0 \leq i < n \) where \( n \) is the length of the string, \( |C| = |\Sigma| \) and \( C[i] \) stores the number of occurrences in \( L \) of any character preceding the \( i \)-th symbol in \( \Sigma \), \( R[i] \) keeps count of the number of occurrences of \( L[i] \) in the prefix \( L[0, 1, \ldots, j] \) of \( L \).

In a typical BWT based compression, BWT is followed by a transform called MTF (Move-To-Front), which re-orders the string and makes it more suitable for RLE. The output from MTF is encoded by RLE and is finally entropy encoded by Arithmetic coding [Langdon84] [Witten87] or Huffman coding [Huffman52].

BWT transform is suitable for compressed pattern matching since a binary search can be performed on \( F \) (a completely ordered string) and \( F \) has direct mapping to \( L \). Random access in \( L \) is supported and the sub-strings decoded from subsequent positions in \( L \) are lexicographically sorted. A discussion on BWT based CPM will be presented in Chapter 3 and the performance comparisons of BWT-based CPM algorithm are shown in Chapter 4.
2.1.3 Lossless Image Compression

The lossless image compression algorithms introduced below are all predictive coding. As we have introduced in section 2.1.1, one important way to achieve better compression is to change the probability distribution of the source. The basic strategy of predictive coding employs this principle. Predictive encoding is very efficient for lossless image compression because it employs the local characteristics of the natural images. Generally, a predictive encoder scans the image in a specific order (usually in raster scan order) and predicts a pixel’s value from the pixels that have been encoded, particularly those pixels that are spatially close to the current pixel. If the prediction is accurate enough, the prediction errors will present a probability distribution that is very efficient for entropy encoding. An entropy encoder on the prediction errors will produce very good compression. The various predictive coding schemes differ in their ways to predict the errors.

Throughout the following discussion, we refer Figure 2-2 as the casual template of the pixel $x$. The symbols $a$, $b$, $c$ and $d$ denote the pixels at the west, north, northwest and northeast with respect to $x$. In the followings, each of these symbols is used interchangeably to represent a pixel itself and its actual value.

```
<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 2-2 Casual template used for context modeling and prediction
• Lossless JPEG

Lossless JPEG [Sayood03] is the first lossless still image compression international standard. There are seven predictive mode in Lossless JPEG, as shown in Table 2-1, where $x'$ is the predicted value of $x$.

Table 2-1 Lossless JPEG prediction modes

<table>
<thead>
<tr>
<th></th>
<th>$x' = b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$x' = a$</td>
</tr>
<tr>
<td>3</td>
<td>$x' = c$</td>
</tr>
<tr>
<td>4</td>
<td>$x' = a+b-c$</td>
</tr>
<tr>
<td>5</td>
<td>$x' = b+(a-c)/2$</td>
</tr>
<tr>
<td>6</td>
<td>$x' = a+(b-c)/2$</td>
</tr>
<tr>
<td>7</td>
<td>$x' = (a+b)/2$</td>
</tr>
</tbody>
</table>

Different images have different structures. Therefore, An image should choose from the seven prediction models the one that is best suitable for it. For example, an image that has many vertical lines may consider mode 1 while an image that has many horizontal lines may consider mode 2. The prediction mode can be pre-determined by a pre-processing of the image. Some compression implementations use adaptive prediction mode, where the prediction algorithm is dynamically changed, depending on the context of the pixel.

The prediction errors, $\delta = x - x'$ are then entropy encoded using Huffman or Arithmetic encoding.
JPEG-LS Algorithm

JPEG-LS [weinberger00] is the current international standard on lossless still image compression. Referring Figure 2-2, JPEG-LS compression algorithm performs the following operation for pixel $x$ in the scan line:

1. Find the initial prediction $x'$

The prediction algorithm is in the following:

$$\begin{align*}
\text{If } c &\geq \max(a, b) \\
x' &= \min(a, b) \\
\text{else} &\{ \\
\text{If } c &\leq \min(a, b) \\
x' &= \max(a, b) \\
\text{else} & \\
x' &= a + b - c
\}
\end{align*}$$

2. Determine the current pixel’s context $Q$.

To be able to predict a pixel’s value accurately, in JPEG-LS, each pixel is categorized as in one of 365 contexts. A context is used to condition the coding of the present pixel based on a small number of neighboring pixels. The context determines the values of the coding variables used in the rest of the coding process for the current pixel.

The context $Q$ is indicated by an integer number. To compute $Q$, three values $Q_1$, $Q_2$, $Q_3$ are first computed. $Q_i$ ($i=1,2,3$) is determined by $D_i$ ($i=1,2,3$) as shown in the following table:

<table>
<thead>
<tr>
<th>$D_i$</th>
<th>$Q_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_i \leq -T3$</td>
<td>-4</td>
</tr>
<tr>
<td>$-T3 &lt; D_i \leq -T2$</td>
<td>-3</td>
</tr>
<tr>
<td>$-T2 &lt; D_i \leq -T1$</td>
<td>-2</td>
</tr>
<tr>
<td>$-T1 &lt; D_i \leq 0$</td>
<td>-1</td>
</tr>
<tr>
<td>$D_i == 0$</td>
<td>0</td>
</tr>
<tr>
<td>$0 &lt; D_i \leq T1$</td>
<td>1</td>
</tr>
<tr>
<td>$T1 &lt; D_i \leq T2$</td>
<td>2</td>
</tr>
</tbody>
</table>
where $D_1 = d - b$, $D_2 = b - c$, $D_3 = c - a$. $T_1$, $T_2$ and $T_3$ are user predefined positive numbers. Finally $Q$ is computed as: $Q = 81 \times Q_1 + 9 \times Q_2 + Q_3$. The computation of $Q$ captures the level of activity (smoothness, edginess) surrounding pixel $x$.

3. Refine the prediction by subtracting or adding the bias of context $Q$. Each context maintains a prediction bias value, which is computed based on all previous predictions and it reflects how good the prediction is for that particular context. The bias of a context is updated every time a pixel belonging to that context is being encoded.

4. Compute the prediction error (residual) $\text{Errval}$ by subtracting the refined prediction from the actual value of $x$. Therefore, $\text{Errval} = x - x'$.

5. Update the bias of the current pixel’s context. For a context $Q$, the following values are maintained: $N[Q]$ is increased by 1 each time a pixel belongs to $Q$ is encoded; $C[Q]$ is the bias of context $Q$; $B[Q]$ accumulates the pixel errors (positive or negative) and it is used to increment or decrement the bias as in the following algorithm:

```c
if (B[Q] <= -N[Q]) {
    B[Q] = B[Q] + N[Q];
    if (C[Q] > MIN_C)
        C[Q] = C[Q] - 1;
    if (B[Q] <= -N[Q])
        B[Q] = -N[Q] + 1;
} else if (B[Q] > 0) {
    B[Q] = B[Q] - N[Q];
    if (C[Q] < MAX_C)
        C[Q] = C[Q] + 1;
    if (B[Q] > 0)
        B[Q] = 0;
}
```
when N[Q] exceeds a pre-determined threshold (usually 64), B and N are right-shifted by 1 bit to halve their values.

6. Re-map the residuals. The original pixel values lie in range [0, RANGE-1], where the variable RANGE is decided by the number of bits used to represent the pixels. For example, if the pixels are represented by 8 bits, RANGE = $2^8 = 256$. The residuals should be re-mapped to the same range so that they have one-sided geometric distributions (OSGDs) [Weinberger00]. The re-mapping is necessary for the entropy encoder – Golomb encoder since it requires the input value to be non-negative and it is optimal for OSGD.

7. Encode the remapped residual using limited length Golomb code. A parameter $k$ must be computed before using the Golomb code. The value of $k$ is computed based on the geometric distribution of prediction errors from pixels that belong to the same context $Q$. An integer $A[Q]$ is also maintained in $Q$ to track the absolute prediction errors and $k$ is computed as:

$$k = \min(k' \mid 2^{k'} \cdot N(Q) \geq A(Q))$$

where $A[Q]$ is also right-shifted by 1 bit when the threshold is exceeded.

JPEG-LS preserves the best characteristics of CALIC and has a much simpler implementation. Although the compression is slightly worse than CALIC, the improved speed and memory usage make JPEG-LS out-perform CALIC and has become the current lossless image compression international standard.
2.2. Pattern Matching

The pattern matching problem may be formulated as follows [Knuth77]: given pattern string $x$, with $|x|=m$, and text string $y$, with $|y|=n$, where $m, n>0$ and $m \leq n$, if $x$ occurs as a sub-string of $y$ then determine the position within $y$ of the first occurrence of $x$, i.e., return the least value of $i$ such that $y(i, i+m-1)=x(1,m)$. The problem is generally extended to find all of the occurrences of $x$ in $y$ instead of only the first. There are also many variations of this problem, such as the approximate pattern matching problem, the wildcard pattern matching problem, the multiple-pattern matching problem and the two-dimensional (or even higher dimensional) pattern matching problem. The pattern matching techniques are important in many application areas such as data processing, information retrieval, text editing and word processing, linguistic analysis, and areas of molecular biology such as genetic sequence analysis. Pattern matching can be applied on either text strings or images.

In situations where a fixed text string is to be searched repeatedly for the occurrences of many different patterns, it is worthwhile to build an auxiliary index table for the text. This strategy is particularly useful for searching large files where the file contents of the files are relatively stable. A typical example is the dictionary search. Dictionary search is not the focus of this research.

2.2.1 Single Pattern Matching Algorithms

- **Brute Force Method**

  The brute force method scans the text $T$ from left to right. At every position of $T$, it compares $P$ and the sub-string of $T$ starting at that position.
The C code of this algorithm is shown below:

```c
for (i = 0; i < n; i++)
{
    for (j = 0; j < m; j++)
    {
        if (T[i+j] != P[j]) break;
    }

    if (j == m) printf("A match found at position %d\n", i);
}
```

The brute force method is the most expensive searching algorithm. It takes $O(mn)$ time in the worst case. When applied to image, which in nature is two dimensional, the algorithm is called template matching.

- **Kunth-Morris-Pratt Algorithm**

In the brute force method, every time when a mismatch happens, the algorithm moves to the next immediate symbol of text and checks for a match. The Knuth-Morris-Pratt (KMP) algorithm [Knuth77], in contrast, moves more than one position.

![KMP algorithm illustration](image)

As shown in Figure 2-3, suppose pattern P and text T are being compared at position $i$ and the first mismatch happens at position $i+j$, i.e. $P[0..j-1] = T[i...i+j-1]$ and $P[j] \neq T[i+j]$. 

30
Figure 2-4 Pattern shifting in KMP algorithm

To check the next possible match, we shift P to the right along T. To avoid missing any possible match, we need to check if the situation shown in Figure 2-4, where \( v \) is both a prefix and a suffix of \( u \) and \( ? \) represents any symbol, happens before \( P[0] \) is aligned with \( T[i+j] \). It is possible that there are more than one such prefix/suffix of \( u \). However, we are only interested in the longest one \( \overline{v} \), or the one that starts at the leftmost position in \( u \). Once such \( \overline{v} \) is found, P can be “safely” shifted to the right \( j-|\overline{v}| \) positions and the algorithm resumes the comparison at position \( i+j \).

However, if \( ? \) is still an \( a \), an immediate mismatch will happen after the comparison resumes. To avoid this, only \( \overline{v} \) followed by a symbol that is different than \( P[j] \) (\( a \) in this case) is interested. Compared to one shift in the brute force method, the \( j-|\overline{v}| \) shifts in KMP algorithm may greatly improve the speed of searching.

The description about \( \overline{v} \) can be generalized as: *The longest prefix of \( P[0...j-1] \) which is also a suffix of \( P[0...j-1] \) and is followed by a symbol that is different than \( P[j] \).* A careful examination of this description reveals that \( \overline{v} \) is all about pattern P and is irrelevant to T. Therefore, \( \overline{v} \) can be pre-computed at \( j \) where \( 0 \leq j < m \).

The pre-processing of the pattern can be done using KMP automaton. It takes \( O(m) \) time and space to compute the value of \( \overline{v} \) for all \( j \) (\( 0 \leq j < m \)). The result from the
KMP automaton is also called the failure function of the pattern. The expected theoretical behavior of the algorithm is \(O(n+m)\).

- **Boyer-Moore Algorithm**

  Boyer-Moore algorithm is the most efficient pattern-searching algorithm in practice [Aoe94]. It has been implemented in most editors for string searching [Boyer77] [El-Mabrouk96]. It scans the text from left to right, as any other searching algorithm, but starts the comparison from the rightmost position, as shown in Figure 2-5. The algorithm takes two shifting strategies when mismatch happens: *good-suffix* shift and *bad-character* shift, at any time of the execution of the algorithm, whichever gives the most shifts will be taken.

![Figure 2-5 Boyer-Moore algorithm illustration](image)

The *good-suffix* shift is explained below. In Figure 2-6, suppose the first mismatch happens at position \(i+j\), i.e. \(P[j+1, \ldots, m-1] = T[i+j+1, \ldots, i+m-1]\) and \(P[j] \neq T[i+j]\).

![Figure 2-6 Good-suffix illustration](image)
To check the next possible match, we shift P to the right along T. If \( u \) re-occurs somewhere else in P, as shown in Figure 2-7, there is no need to compare the symbols in \( u \) when the comparison resumes. Again, only \( \bar{u} \), the rightmost re-occurrence of \( u \) is interested (because it gives us the shortest shift) and \( \bar{u} \) should preceded by a symbol different than \( P[j] \) (\( a \) in this case) to avoid an immediate mismatch.

![Figure 2-7 Pattern shifting in good-suffix](image)

When \( \bar{u} \) does not exist, P can be shifted to the right until \( P[0] \) is aligned with \( T[i+j+2] \) without worrying missing a match. P can be further shifted to the first place where a prefix of P and a suffix of \( u \) (or P) match, as shown in Figure 2-8. In other words, P can be shifted until \( T[0] \) is aligned with \( T[j+m-|\bar{v}|] \), where \( \bar{v} \) is the longest prefix of P that is also a suffix of P.

![Figure 2-8 Pattern shifting in good-suffix](image)

Again, \( \bar{u} \) and \( \bar{v} \) can be pre-computed for \( j \) where \( 0 \leq j < m \).

The bad-character shift follows the following simple idea: when mismatch happens, P should be shifted to the right until \( T[i+j] \) is aligned with its rightmost match in
P[0, ..., j-1]. In practice, the pre-computation computes the rightmost occurrence in P[0, ..., m-2] for each symbol in the alphabet Σ.

Finally, when the comparison resumes, the algorithm does not need to compare symbols \( \overline{u} \) or \( \overline{v} \). The searching speed is therefore improved. The theoretical behavior of the Boyer-Moore algorithm is equal to that of the KMP algorithm. However, experimental results show that it is faster than the KMP algorithm.

### 2.2.2 Multiple Pattern Matching

A multiple pattern-matching algorithm considers searching the text string for occurrences of any of a set of N patterns, \( P = \{P_1, P_2, ..., P_N\} \). The repeated application of any of the single pattern matching algorithms introduced above would lead to inefficient search. Taking the BM algorithm for instance, it would take \( O(m+Nn) \) time where \( m \) is the total length of the patterns. To accomplish a better search process such as one runs in \( O(n+m) \) time, most multiple-pattern matching algorithm, such as the ones introduced below, construct an automaton from the patterns to search for pattern string simultaneously.

- **Aho-Corasick Algorithm**

  Aho-Corasick algorithm [10] is a classic solution for multiple-pattern matching. The algorithm is a generalization of the KMP algorithm. In The AC algorithm, pattern matching is performed using an automaton called \textit{AC automaton}, which is constructed from the patterns. The AC automaton for a set of patterns \( \{aa, ab, abc\} \) is shown in Figure 2-9.
In an AC automaton, each edge is labeled with a symbol and edges coming out from a node (state) have different labels. If we define $R(v)$ of a state $v$ as the concatenation of labels along the path from the root to state $v$, the following is true: for each pattern $P$ in the pattern set, there is a state $v$ that $R(v) = P$, and this state is called a final state; each state represents a prefix of some pattern; for each leaf state $v$, there is some pattern $P$ in the pattern set so that $R(v) = P$. For instance, in Figure 2-9, pattern “aa” is represented by state 2 and leaf state 4 represents pattern “abc”. Both states 2 and states 4 are final states.

![Figure 2-9 AC automaton example](image)

For each state, a $\text{goto}(v, a)$ function is defined which gives the state to enter from state $v$ by matching symbol $a$. For instance, $\text{goto}(1, a) = 2$ means that the state entered is 2 if we match symbol $a$ from state 1. A failure link $f(v)$ (indicated as dotted line in Figure 2-9) is also defined for each state and it gives the state to enter when mismatch happens. The failure link $f(v)$ points to a state that represents the longest proper suffix of $R(v)$. Thus, when mismatch happens, by following the failure link, we will be able to continue the matching process since the state to enter also corresponds to a prefix of some pattern. Finally, an $\text{out}(v)$ function is defined for state $v$ that gives the patterns recognized when entering that state.
When searching the patterns, the AC automaton starts from the root of the automaton and processes one symbol from the input text $T$ at a time. Through the goto functions and the failure links, the automaton changes its current state from one to another. The automaton reports the pattern occurrence if a final state is entered.

The construction of the automaton takes time and space $O(m)$ where $m$ is the total length of the patterns. The search takes time $O(u)$ where $u$ is the size of the input text. Thus, the overall computational time of Aho-Corasick algorithm is $O(u+m)$.

2.2.3 Two Dimensional Pattern Matching

Two dimensional pattern matching is a natural generalization of the one dimensional pattern matching problem. The Exact Two Dimensional Matching problem is defined in [Amir92] as: let $\Sigma$ be an alphabet, given a text array $T[n \times n]$ and a pattern array $P[m \times m]$, report all locations $[i,j]$ in $T$ where there is an occurrence of $P$, i.e. $T[i+k, j+l] = P[k, l]$ for $0 \leq k, l \leq n$.

- Bird’s Algorithm

The first linear two dimensional pattern matching algorithm was developed by Bird [Bird77] and also independently, by Baker [Baker78].

Bird’s algorithm performs a linear scan on the text. At each location of the linear scan, the examination of the text consists of two distinct parts: row-matching and column-matching.

The row-matching step is performed to determine which row of the pattern, if any, ends at the current location. This can be done easily using the Aho-Corasick algorithm by treating each row of the pattern as a separate keyword.
At a particular location \([i, j]\), if the row-matching step determines a row \(P_i\) of the pattern occurs at that place, we also need to know if the rows \(P_0, P_1, \ldots, P_{i-1}\) occur immediately above \(P_i\) in order to determine if the complete pattern array is a sub-array of the text. This is done in the column-matching step. The algorithm maintains a vector \(a[1…n]\) such that at the beginning of the examination at location \([j,k]\), \(a[k] = i\) means that rows \(P_0, P_1, \ldots, P_{i-1}\) have been found to occur immediately above location \([j,k]\). This situation is illustrated in Figure 2-10. Therefore, if \(a[k] = m\), all rows of the pattern are matched and we shall report a pattern occurrence.

![Figure 2-10 Bird's algorithm](image)

At the beginning of the algorithm, \(a[k]\) for all \(k (1\leq k \leq n)\), is set to 0. The value \(a[k] = i\) is then updated as follows: at a particular location \([j, k]\), if the row-matching stage fails to yield a match for any row of the pattern, \(a[k]\) is reset to 0; if it is found that row \(P_r\) occurs at location \([j, k]\), then \(a[k] = s+1\), where \(s\) is the maximum value\(\leq i\) such that:

\[
P_r = P_s,
\]

\[
P_{r-1} = P_{s-1},
\]

\[
\ldots
\]

\[
P_{r-s} = P_0
\]
If we consider each row of the pattern as one single symbol, the rows \( P_0P_1\ldots P_{i-1}P_i \) can then be considered as a vertical string \( u \) consists of such symbols. Therefore, the problem of finding \( s \) is equivalent to finding the longest prefix of \( u \) that is also a suffix of \( u \). The exact same problem has been encountered in the KMP algorithm, from which we know that the problem can be solved through a pre-processing of the “vertical pattern” \( P_0P_1\ldots P_m \).

Bird’s algorithm takes time \( O(n^2+m^2) \) on average and in the worst-case. The algorithm needs \( O(n+m^2) \) space.

- **Baeza-Yates Algorithm**

Baeza-Yates [Baeza-Yates93] proposed another linear-time algorithm which works faster than Bird’s algorithm. In Baeza-Yates’s algorithm, the \( m^{th} \), \( 2m^{th} \), \( 3m^{th} \) … rows are called the primary rows while the rest are called the secondary rows. The algorithm linearly scans ONLY the primary rows and at each location, it performs a row-matching step just as in Bird’s algorithm. If the row-matching stage yields a match for any row of the pattern, further check is done in the nearby area to determine if the pattern occurs. The basic idea of the algorithm is illustrated in Figure 2-11 and the algorithm is shown below, where OUT represents the row-matching results.

```java
For (j=m-1; j<n; j+=m)
{
    for (k=0; k<n; k++)
    {
        OUT = row-matching
        if (row_index not empty)
            check_match(j, k-m+1, OUT)
    }
}
```

Suppose row \( P_i \) is found at \([j,k]\), the check_match operation needs to check the \( i-1 \) rows immediately about row \( i \) and the \( m-i \) rows immediately below row \( i \). The cost of this
checking is $m(m-1)$. However, if there are repeated rows of the pattern, the checking needs to be done for every repeated row. In the worst case ($P_0=P_1=\ldots=P_{m-1}$), the cost of $check\_match$ is $m^2(m-1)$.

Figure 2-11 Primary rows and secondary rows

However, since there are at most $m-1$ secondary rows immediately above the current primary row and at most $m-1$ secondary rows below the current primary row need to be checked, we should be able to finish $check\_match$ at cost $2m(m-1)$. In fact, each of the $2m-1$ rows (including all the $2(m-1)$ secondary rows and the primary row) is computed at most once using the Aho-Corasick automaton. The results are stored and $check\_match$ uses the results to check the pattern match.

Overall, the algorithm runs at time $O(n^2/m+m^2)$ on average and $O(mn^2)$ in the worst case. It needs $O(m^2)$ space. The algorithm can be further improved. As shown in Figure 2-11, on the $m$th row, if there is horizontal overlapping, the overlapping area will be checked more than once. To avoid the redundancy, checking on the secondary rows can be held until the end of the primary row. The improved algorithm needs time $O(\alpha n^2/m+m^3+q)$ on average and $O(2n^2+m^3+q)$ in the worst case, with space cost $O(m^2+q)$, where $\alpha<1$ and $q$ is the size of the alphabet.
CHAPTER 3. COMPRESSED PATTERN MATCHING

In this chapter, the current status of compressed pattern matching is reported first. The CPM algorithms introduced in this chapter include: CPM algorithms for non-adaptive compression, for adaptive compression, for entropy encoding, multiple-pattern CPM algorithms and two-dimensional CPM algorithms. A discussion on the performance measurements of CPM algorithms is also introduced in this chapter. The classification of CPM algorithms is also discussed.

3.1. Introduction

Pattern matching in compressed domain started as a side effect of Eilam-Tsoreff and Vishkin’s work in [Eilam-Tsoreff 88] and Amir, Landau and Vishkin’s work in [Amir92-2]. In [Eilam-Tsoreff 88], an algorithm is developed to search the given pattern in RLE (Run Length Encoding) compressed stream while in [Amir92-2] the two-dimensional version of the same problem is addressed. Their works were among the earliest works of CPM.

In Amir and Benson’s paper [Amir92-3, Amir92-4], the CPM problem is formally defined as:

Let $\sigma = s_1 \ldots s_u$ be a text string of length $u$ over alphabet $\Sigma = \{a_1, \ldots, a_q\}$. Let $\sigma.c = t_1 \ldots t_n$ be a compression of $\sigma$ of length $n \leq u$. The inputs are the compressed text $\sigma.c$ and a pattern $p_1 \ldots p_m$ ($p_i \in \Sigma$ for $i=1, \ldots, m$), and the output is the first text location $i$ such that there is a pattern occurrence at $s_i$, i.e., $s_{i+j-1} = p_j$ for $j=1, \ldots, m$. 
Amir and Benson also defined that a compressed matching algorithm is efficient if its time complexity is \(O(u)\), sub-optimal if its time complexity is \(O(n \log m + m)\) and optimal if it runs in time \(O(n+m)\).

A survey on earlier works on compressed pattern matching can be found in [Bell01]. Most compressed pattern matching algorithms are for text compression. Multiple-pattern CPM algorithms and two-dimensional CPM algorithms are also getting more attentions.

CPM algorithms based on the LZ-family compression have been conducted in the last decade. The research of searching LZ-compressed files is very important because the LZ compression algorithms are among the most efficient and popular compression algorithms nowadays. Their excellent time/compression efficiency and easy implementation have gained them a large popularity in the commercial world (e.g. the ZIP utilities and COMPRESS utility). Among the LZ-family based CPM algorithms: Farach and Thorup proposed a randomized algorithm [Farach98] to determine whether a pattern is present or not in LZ77 compressed text in time \(O(m + n \log^2 (u/n))\), where \(u\) is the raw file size, \(n\) is the compressed file size and \(m\) is the length of the pattern; Navarro and Raffinot [Navarro99] proposed a hybrid compression between LZ77 and LZ78 that can be searched in \(O(\min(u, \log m) + r)\) average time, where \(r\) is the total number of matches; Amir [Amir96] proposed an algorithm which runs in \(O(n \log m + m)\) -- “sub-optimal” or in \(O(n+m^2)\) – “almost optimal”, depending on how much “extra space” is used to search the first occurrence of a pattern in the LZW encoded files. Barcaccia [Barcaccia98] extended Amir’s work to an LZ compression method that uses the so-called “ID heuristic”. Among the LZ-based CPM algorithms, Amir’s algorithm has been well-recognized, not only
because of its “almost-optimal” or near “optimal” performance, but also because it works directly with the LZW compression without having to modify it. This is a great advantage because keeping the popular implementations of the LZW and avoiding the re-compression of the LZW-compressed files are highly desirable. But, unfortunately, Amir’s original algorithm has never been implemented [Personal communication with Amir, 2003] and thus no experimental results are available to show the practical performance of the algorithm. To make Amir’s algorithm practical it is also necessary to enhance it by having it report all pattern occurrences and allow for multiple pattern matching. In [Kida98] and [Kida00], Amir’s algorithm has been extended by Kida for multiple-pattern matching by using Aho-Corasick algorithm and it takes $O(n+m^2+t+r)$ time and $O(m^2+t)$ extra space for the algorithm to report the pattern occurrences where $t$ is the size of the dictionary. Experimental results from [Kida98] and [Kida00] show that the algorithm is nearly twice as fast as the naïve decompress-and-search approach.

Recently, [Adjeroh02], [Bell02] and [Ferragina01] have developed a series of Burrows-Wheeler Transform (BWT) based approaches to CPM including: Compressed-Domain Boyer-Moore, Binary Search, Suffix Arrays, q-grams and FM-Index. These approaches have been implemented and compared in [Firth02] and the experimental results show that they are the most competitive CPM algorithms among all reported CPM works at the time. However, among these approaches, FM-Index is made search-aware at the price of sacrificing the compression performance. All other approaches cannot be applied directly on Bzip2 (an efficient commercialized BWT compression utility) compressed files in that they require the entire file to be compressed as one block, which is referred as the \textit{bsmp} compression. As can be seen from [Firth02], the \textit{bsmp}
Compressed pattern matching algorithms [Bell01] for other compressions such as Huffman encoding [Mukherjee94] [Mukheree95] [Moura00], arithmetic coding [Bell01], anti-dictionary [Shibata99-2], PPM [Cleary84] [Cleary97] RLE [Bunke93c] [Bunke95] and Byte-Pair-Encoding [Mandal99][Manber97] [Shibata99-2] etc. are also reported. Among them:

- Manber [Manber97] proposed an algorithm and the basic idea is to replace common bi-grams with special symbols. Although the compression ratio is only about 30%, this compression algorithm allows the basic pattern matching algorithms to be used on the compressed file. A similar approach is proposed by Shibata [Shibata99], who uses a KMP automaton to search the pattern.

- In [Shibata99-2], Shibata developed an algorithm to search patterns in the text compressed using anti-dictionary, which is a compression algorithm developed in [Crochemore00]. The algorithm preprocesses a pattern of length \( m \) and the anti-dictionary \( A \) in \( O(m^2+a) \) time, where \( a \) is the total length of strings in \( A \). The algorithm can determine all occurrences of the pattern by a linear scan of the compressed text of length \( n \) in time \( O(n+r) \) where \( r \) is the number of the pattern occurrences.
• Mukherjee and Acharya [Mukherjee94] [Mukheree95] studied searching on Huffman coded files. It was concluded that a simple search of the Huffman coded files to find a compressed pattern using a fast searching algorithm such as KMP will not produce correct result.

• Moura [Moura00] proposed a semi-static word-based modeling scheme for Huffman coded compressed text files. The decompression can be done for arbitrary portions of the text very efficiently and exact search can be done directly in the compressed text using any known sequential pattern-matching algorithm. For simple words, the search time is twice as fast as running the best existing search tool on the uncompressed files. The compression ratio is about 30% for typical English texts.

• Arithmetic coding (in its standard formulation) is considered not suitable for compressed pattern matching [Mukherjee94] [Bell01]. An example from [Mukherjee94] is given below to show the difficulty of searching patterns in arithmetic coded strings. Let $\Sigma = \{a, b, c, d\}$ and assume the probabilities of the four symbols are 0.5, 0.25, 0.125 and 0.125, respectively. Given a text $T = ababacabdbabbacabacac$, it is encoded to $[0.2874, 0.9425, 0.7838]$. Given a pattern $P = baca$, it is encoded to $[0.5950]$. The difficulty of identifying the occurrence of $P$ in $T$ from the encoded strings is obvious.

CPM algorithms on two-dimensional data have also been reported in the literature. Most of the two-dimensional CPM algorithms are for RLE compressed files. One reason is that RLE can be applied on image very efficiently and are used in fax. In one of the earliest work on CPM [Amir92-2], Amir and his colleagues showed an
algorithm for two-dimensional compressed pattern matching for RLE compression. However, the algorithm is far from optimal. Later Amir etc. developed an almost-optimal two-dimensional algorithm [Amir92-3] [Amir92-4] and an optimal two-dimensional [Amir97] algorithm for RLE. The two-dimensional CPM problem is formally defined in [Amir92-4]. In [Amir00-2], the first “inplace” two-dimensional CPM algorithm is reported and a searching algorithm is also presented on RLE compressed files. The algorithm is inplace because it uses $O(c(P))$ extra space for pattern searching where $P$ is the pattern and $c(P)$ is the result of compressing $P$. In one of his most recent works, Amir proposed an “inplace” 2D CPM algorithm on LZW compressed files [Amir03]. The algorithm in [Amir03] performs the search in time $O(n^2)$ and it takes $O(m^3)$ time to preprocess the pattern, assuming the pattern size is $m^2$ and the image size is $n^2$. However, the run-length encoding is only efficient for binary images while the LZW compression is not very efficient for images. Therefore, the work from [Amir97] [Amir00-2]-[Amir03] will not have strong impact on applications that work on gray-scale images or color images. Two-dimensional CPM algorithms based on other compression algorithms are also reported. In [Berman97], a 2D CPM algorithm on hierarchical compression is reported. In [Pajarola96] [Pajarola98], an image compression algorithm that is similar to lossless JPEG was proposed and the related compressed pattern matching approach was studied. It was proposed in [Pajarola98] to use Bird's two dimensional pattern matching algorithm on Huffman encoded files. The algorithm works well for satellite images except for the case when parts of the pattern matches while the whole does not. It was suggested in [Pajarola98] to replace Aho-Corasick algorithm in Bird's algorithm with Commentz-Walter algorithm and it was also suggested to replace Bird's algorithm with
that from [Baeza-Yates93]. The work from [Pajarola98] is very constructive. However, as will be discussed in Chapter 5, it is more feasible to work on the existing image compression standard, but rather than to propose a brand new compression algorithm. In fact, in [Bell01], it is pointed out that the current international standard on lossless image compression, JPEG-LS, has the potential to be searched in the compressed domain. We will present such an approach in this dissertation.

3.2. CPM Algorithms

As mentioned in Chapter 1, the data compression algorithm can be classified in many ways. One way to look at the compression algorithms is to classify them as either “adaptive” or “non-adaptive”. In an adaptive compression algorithm, the compression code that represents a string in the text is determined dynamically by the data. The same string that appears in different locations in the text may have different encoding in the compressed file. In contrast, in a non-adaptive compression, the same string, regardless where it is located in the text, always has the same compressed representation. For example, the run-length encoding and Huffman encoding are non-adaptive while the LZ compressions are adaptive. For non-adaptive compression, one can simply compress the pattern, as how the text is compressed, and run any pattern-matching algorithm on the compressed text and pattern because the pattern and all its matches would have the same encoding. However, as pointed in [Amir96], there may be a need for some extra work. For example, the first and last pattern elements have to be handled separately in the run-length encoding; the starting bit of each encoded symbol (the word boundary) needs to be identified in the Huffman encoding. For adaptive compression, compressed pattern
matching is more challenging. However, it is also more valuable because adaptive compression is generally more effective and thus more popular.

3.2.1 CPM For Non-adaptive Compressions

A "non-adaptive" compression algorithm compresses the symbols without having to consider the context of the symbols. For non-adaptive compressions, we only need to compress the pattern and then run any pattern-searching algorithm on the compressed text for searching of the compressed pattern. This kind of compression algorithms include RLE, BPE, Huffman encoding etc. and they usually need some extra work to make the compressed text "searchable". We now study some of the existing compressed domain searching algorithms based on these compression algorithms.

- Shibata's Approach on BPE

Byte Pair Encoding (BPE) is a substitution compression approach. The idea is to search the most frequent byte-pair (considering one character is generally represented by one byte, the byte-pair is two characters in the text) and to replace them with a special byte representation (or a special symbol). The process is repeated till all possible special symbols are exhausted or no "most-frequent" byte-pair can be found. Since one byte can represent 256 different symbols and only 128 of them are used in current ASCII code, we have 128 special symbols available for encoding.

We now show an example of BPE. However, since we don't have a way to write down the special symbol in this document, we assume that the alphabet of our text being encoded is {A - E} and the rest English letters {G -Z} are our special symbols. Thus, to encode "ABABCDEBDEFABDEABC", we first replace the most frequent byte-pair
"AB" with our first special symbol 'G' to obtain a new string: "GGCDEBDEFGDEGC". We then replace the most frequent byte-pair "DE" in the new string the second special symbol 'H' to obtain a new string "GGCHBHFGHGC". We eventually have string "GIHBHFGHI" as the final encoding.

To be able to decompress the code, the substitution table has to be sent along with the compressed code. For the above example, the table is: \{AB⇔G, DE⇔H, GC⇔I\}.

Practically, a file to be encoded is divided into many blocks and each block is BPE coded separately. To speed up the encoding, Shibata [Shibata99] proposed to build a substitution table based on the first block and universally apply this table for all other blocks. The coding speed therefore is increased but the compression is worse.

Obviously, if we use the above compression approach to encode a pattern, the compressed format of the pattern and the compressed format of its match in the text will unlikely be the same since the “most-frequent byte-pair” is different if the context is different. For example, pattern “ABABC”, which appears in the text given above, will be encoded “HC” while its match in the text is encoded as “GI”. Even we use the same transition table to encode the pattern, we still may not be able to get the same encoded data. For example, when we use the same transition table for the above text, pattern “BABC”, which is a sub-string of given text, will be encoded as “BI”, which does not appear in the compressed text.

Shibata proposed two searching approaches. The first approach is simple and brute force -- simply try all possible encoding of the pattern. The second approach is to use KMP automata. Since each code in the compressed data may represent more than one actual symbol and thus may cause a series of transitions, Shibata proposed to substitute a
series of successive transitions with one transition. The actual idea, is implicitly decoding the compressed text while searching the uncompressed pattern in the compressed text.

- **Manber's Search-aware BPE**

  In [Manber97], Manber proposed a search-aware compression that is based on BPE. Different than Shibata’s BPE, Manber’s algorithm puts restrictions on the encoding so that the pattern will be encoded exactly the same as their matches in the text.

  In Manber’s approach, the alphabet is divided into two non-overlap sets, V1 and V2 so that: the first byte of any byte-pair is from V1 and the second byte of that byte-pair is from V2; the sum of the frequencies of the byte-pairs is maximal. The compression ensures that the byte-pairs in the text is non-overlapped, and therefore guarantees the unique encoding of any sub-string of the text. To accomplish this, a directed graph is constructed and each node in this graph corresponds to a unique symbol appears in the text. If two nodes consist a pair, a directed edge is connected between them. The weight of the directed edge is the number of times the pair appears in the text. After the graph has been constructed, the nodes are then grouped into two sets, V1 and V2. V1 and V2 are divided so that the sum of the total weights of all edges directed from V1 to V2 is maximal.

  Comparing with Shibata’s approach, Manber’s approach has the following difference: the byte-pair is not recursive; a symbol can be only the first or the second in all pairs, but not both for a text. The approach makes a pattern uniquely encoded and the searching of the pattern can be directly performed in the compressed domain.

  The searching has the following steps:
1. The transition table is uncompressed from the compressed data. Inverse transition table is also built.

2. The pattern is encoded using the same table. If either the first symbol in the pattern belongs to V2 in the partition or the last symbol in the pattern belongs to V1 in the partition, they will be removed from the pattern.

3. The searching is performed directly on the compressed data since the encoding of the pattern is unique.

4. If the first or last symbol was removed from the pattern, a filtering process is applied on the outputs from step 3.

Manber’s searching approach works very efficiently. However, because of the restriction of the compression, the compression rate is only near 30% for English text.

- Huffman Coding

As one of the two most popular entropy-encoders (another one is the Arithmetic coding), Huffman coding plays a very important role in compression for both text and image. Many compression algorithms use Huffman encoding as the final stage of the compression to achieve optimal compression. Thus, to be able to search in Huffman coded data is very critical for fully compressed pattern matching.

Compressed pattern matching on Huffman code may seem straightforward. The pattern can be compressed using the same Huffman table that compresses the text and then any pattern searching algorithm should apply directly on the compressed text. However, a close examination reveals that the above simple idea will lead to false matches. The problem of false match can be illustrated in Figure 3-1.
P = ab, T = abbacdabca

C(P) = 010, C(T) = 01010011011101101100

Correct match

False match

Figure 3-1 False match in Huffman code

As can be seen from Figure 3-1, both the pattern P and the text T are encoded using the Huffman tree shown in the diagram. When a bit to bit match is performed for the compressed pattern C(P) on the compressed text C(T), false matches occur where the starting bit of the match is not the starting bit of a valid code. Because Huffman code is a variable length code, it is not easy to keep track of the code boundaries as for the fixed length code. The false match problem is therefore a challenge for any variable length code.

Mukherjee and Acharya [Mukherjee94] [Mukherjee95] studied the false match problem for Huffman code and proposed a method to keep track of the code boundaries. Their idea is illustrated in Figure 3-1. Each node of the Huffman tree is assigned a unique number, which is calculated as $2^p + b$ where $p$ is the number assigned to the node’s parent and $b$ is 0 if the node is a left child of its parent or 1 if a right child. The root node of the tree is numbered with 0. A bit vector CRBS (Character Boundary Signal) is then constructed so that the CRBS bits indexed by the numbers associated with the leaf nodes
are 1’s while the rest of the bits are 0’s. For example, for the Huffman tree shown in Figure 3-1, the CRBS vector is 10100011 as the numbers associated with the leaf nodes are 0, 2, 6, 7, respectively. The significance of CRBS vector is, when a node is given as its associated number, we can immediately tell whether or not the given node is a leaf node (and therefore a valid Huffman code) by checking the bit indexed by that number.

During the compressed pattern searching, an index I is generated to test the code boundary. The index I is initialized to 0. The searching algorithm linearly scans the compressed text and updates on the fly. At every bit, the index I is updated as: \( I = 2I + b \) where \( b \) is the current bit. If the CRBS bit indexed by I is 1, a boundary is detected and I is reset to 0.

The hardware implementation of the algorithm is also given in [Mukherjee94] [Mukherjee95].

### 3.2.2 CPM For Adaptive Compressions

Searching in the compressed domain from adaptive encoding is more challenging. Adaptive compression algorithms include the LZ family compressions, anti-dictionary compression, BWT compression, predictive lossless image compression algorithms and many other compression algorithms. In these methods, the coding highly depends on the context of the symbols so that the same string might be encoded differently if it appears at different locations. Thus, a simple scheme that compresses the pattern and then directly searches the compressed pattern in the compressed text will not work.

The LZ family compressions have gained more attentions than other adaptive compressions in the compressed pattern-matching world. In one algorithm that is based
on the LZ77 algorithm, restrictions have been put on the compression to force the matching starts/ends at the word-boundaries. Therefore, word-based searching will be easier. The advantage of this approach is, no separate decoder is needed to decode the data compressed by the modified encoder. Among the algorithms introduced below, some do not need such restrictions or modifications but work directly with the original compression algorithms while others need the compression to be modified so that the compressed texts are searchable.

- **Amir's Approach on LZW**

  Amir etc. proposed to search on LZW compressed texts [Amir96]. The algorithm is claimed to be “the first almost-optimal” algorithm for adaptive compressions. In [Amir96], it is proved that the LZW trie can be rebuilt from the compressed text without explicit decompression in time proportional to the size of the compressed text. The trie is used to search pattern in the compressed text. A new criterion of the compressed pattern matching algorithms called the “extra space” is introduced in [Amir96] to describe the extra memory used for searching. Details of the searching algorithm will be discussed in Chapter 4, where a modification is made on Amir’s original algorithm.

- **Kida’s Shift-And Approach on LZW**

  In [Kida99], the Shift-And searching algorithm [Abrahamson87] [Baeza-Yates92] [Wu92] is used to search in the LZW compressed file. The algorithm is elegant and fast. The experimental results show that it is about 1.5 times faster than a decompression followed by a search using the Shift-And algorithm. However, as with the Shift-And algorithm, the algorithm is fast when the pattern length is at most 32 (the typical word length).
In the Shift-And algorithm, a state $R$ is calculated for every symbol it scans. The state $R$ can be represented as a bit-vector and it takes constant time to calculate $R$ for a symbol by using the state of the preceding symbol. The algorithm is called “Shift-And” because the state of the current symbol is obtained by bit-shifting the state of the preceding symbol and then a bit-wise logical product. If the most significant bit of $R$ is 1, a pattern occurs at the current symbol.

Let $S.Z[i] \ (1 \leq i \leq n)$ be a LZW code where $n$ is the length of the compressed text. Let $u[i]$ be the sub-string decoded from $S.Z[i]$ and $|u[i]|$ be the length of $u[i]$. Let $k_i = \sum_{j=1}^{i} |S.Z[j]|$, i.e. $k_i$ is the length of the text decoded up to $S.Z[i]$ and $k_i$ is in the range $[0, u]$ where $u$ is the length of the raw text. We also denote $R_k$ to be the state of the $k^{th}$ symbol in the original text. To search the patterns efficiently by giving only $S.Z$ without decompression of $S.Z$, the algorithm computes only $R_{k_i}$ for $i=1...n$. thus the total time will be proportional to the size of the compressed text. There are two problems that need to be solved. The first problem is to compute $R_{k_{i+1}}$ from $R_{k_i}$ and $S.Z[i+1]$ in constant time. The second problem is to enumerate all pattern occurrences giving $R_{k_i}$.

For the first problem, it is proved in [Kida99] that $R_{k_{i+1}}$ can be obtained from $R_{k_i}$ and $\hat{M}(u)$ in constant-time, where $\hat{M}(u)$ is a value obtained from a constant bit-vector by a series bit-shifting an bit-wise operations with the symbols in $u$. Therefore, $R_{k_{i+1}}$ can be obtained from $R_{k_i}$ and $\hat{M}(u[i+1])$ in constant time. The quantity $\hat{M}(u[i])$ for arbitrary LZW trie node can be computed from its parent node and its label with ease and
is stored at the node. The overall computational cost on \( \hat{M}(u[i]) \) \( (i=1…n) \) is \( O(|D|+m) \) time and \( O(|D|) \) space where \( D \) is the size of the LZW trie.

The second problem can be solved in a similar fashion where information on the pattern occurrences is computed and stored for each node.

The algorithm takes \( O(n+r) \) time where \( r \) is the number of pattern occurrences.

- **Navarro's LZ77/LZ78 hybrid compression**

  Navarro [Navarro99] used an approach that is almost the same as that of Amir. He generalized the tree-rebuilding process in Amir’s approach using mathematical representations. Navarro directly applied the generalized approach on both LZ78 and LZ77 algorithms. Of course, it works better for LZ78 than LZ77.

  The main contribution from Navarro’s work was the design of a hybrid compression that combines the advantages of LZ77 (better compression) and LZ78 (better suitable for searching). Navarro also used bit-parallel searching algorithm to search the pattern.

- **Pattern Searching for BWT compression**

  Bell, Adjeroh and Mukherjee [Adjeroh02] [Bell02] have studied compressed pattern matching in BWT compressed texts. Their research has given very promising results. In practice, BWT compression has the following steps: BWT, MTF, RLE and Arithmetic Encoding. Their research work has focused on searching the pattern in the output produced from BWT. They have proposed two approaches:

  1. In the first approach, an array \( H_r \) is built and \( H_r \) will relate a symbol in the permuted string with its position in the original text. This is to say:

  \[
  F[H_r[i]] = T[i], 1 \leq i \leq n
  \]
where \( n \) is the length of the string; \( F \) is the first column in the transformation matrix and 
\( T \) is the original text.

They then apply Boyer-Moore searching algorithm to search the pattern in the text. In the searching process, when a symbol \( T[i] \) is needed, the algorithm access symbol 
\( F[Hr[i]] \).

The nature of the algorithm is implicitly decoding the string when searching is performed. The searching algorithm used in the above approach does not have to be 
Boyer-Moore. Any other sequential string-searching algorithm should work.

2. Their second approach is more interesting and shows the unique characteristics 
of BWT. It is observed that, in the transformation matrix, not only the symbols in the first 
column are sorted, the suffixes these symbols are leading, are also sorted. More 
importantly, random access to these sub-strings can be done easily. The following 
diagram shows the sorted suffixes for “AABCBBABC”.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Thus, if we are comparing the pattern with the sub-string in the text, we need just 
a binary search to locate the first symbol of the pattern in the first column (sorted 
column) of the matrix. This will give us some candidate sub-strings. Since these sub-
strings are also sorted, we then perform a variation of binary search to locate the pattern 
we are searching for.
The second approach works very good. This approach may be very helpful for lossless image compression domain pattern searching as well.

We conclude the status of the current CPM works in the followings:

1. CPM algorithms for the LZ-family compressions are dominant because of the popularity of the LZ compressions and also because their efficiencies. However, the practical performances of the algorithms are not clear except for the algorithm reported in [Kida99], where the pattern length is at most 32.

2. CPM algorithms for images have been reported but the amount of works have done for images are much less than that for texts.

3. The best reported work is “almost-optimal”. Optimal, even sub-optimal algorithms are still the goal of CPM.

4. Most compression algorithms use entropy coding as the last step. Thus the CPM algorithms based on these algorithms require partial decompression. A pattern matching algorithm fully in the compressed domain is yet a challenge.

3.3. Performance Measurement

3.3.1 Data Compression

The performance measurement of the data compression algorithms is introduced in this section. Different than traditional performance measurement, a new criterion for evaluating the search-awareness of the compression algorithm is included in the measurement. The criteria are:

- Compression ratio.
- Encoding/Decoding speed.
- Memory usage.
- Random access.

The compression ratio is the most important criterion for any compression algorithm and is the primary goal of compression. In CPM, compression ratio is sometime sacrificed to achieve the search-awareness. For example, in Shibata’s BPE compression, restrictions on the byte-pairs are applied so that the pattern can be searched directly in the compressed text but the compression ratio is degraded.

The encoding and decoding speed are very important parameters as well. Depending on the applications, the requirements of encoding speed and decoding speed are different. For on-line applications, the encoding and decoding speed needs to be symmetric. For off-line applications where the data is compressed once and decompressed many times, the decoding speed is required to be fast and the encoding/decoding time could be non-symmetric.

Memory usage of the compression may sometimes be critical in some applications.

Random access is a new criterion and is added for evaluating the search-awareness of the compression algorithms. Random access means the ability to randomly access the compressed data; in another words, it means the ability to decode arbitrary portions of the data. If the compressed data can be randomly accessed at any particular point without decoding the preceding data, pattern matching can be performed directly at that point thus the searching speed will be significantly improved.
3.3.2 Pattern Matching

The performance measurement for pattern matching algorithm include:

- Searching speed.
- Memory usage.
- Precision: How good the algorithm retrieves only the correct matches.
- Recall: How well the algorithm can retrieve all correct matches.
- Ranking (for approximate search): How well the ranking of the results produced by the algorithm compares with that produced by a human observer.

The compressed pattern-matching problem is one special case of the pattern-matching problem. The performance measurement of the pattern matching algorithms is therefore applied on the compressed pattern matching algorithms as well.

3.3.3 Compressed Pattern Matching

The following parameters are usually considered for compressed pattern matching algorithms to evaluate their performances:

- Searching speed. The compressed CPM algorithm can be classified based on its speed. Both Amir [Amir96] and Farach [Farach98] have defined similar categorizations. In this document, we adopt Amir’s definition. Let \( u \) be the size of the uncompressed data, let \( n \) be the size of the compressed data, let \( m \) be the size of the (raw) pattern, the compressed pattern matching algorithms are classified as:
  - **Optimal**, which takes time \( O(n+m) \) time; **Sub-Optimal**, which takes \( O(n\log m+m) \); and **Competitive** or **Efficient**, which takes time \( O(u+m) \). An
optimal algorithm takes time proportional to the size of the compressed data. A competitive algorithm can never be worse than the naïve “decompress-then-search” approach. Otherwise, performing the compressed pattern matching will make no sense.

- Memory usage and Extra Space. The concept of “extra space” is defined in [Amir96] to measure the amount of “extra” memory space used for the compressed pattern matching other than the storage of the compressed data. The introduction of extra space is very important because, in some applications, the amount of extra memory available is limited. For some applications, an optimal algorithm using O(n) extra space, in addition to the compressed data itself, may not be feasible. One would preferably to achieve an optimal algorithm that uses O(m) extra space.

- Precision: How good the algorithm retrieves only the correct matches.

- Recall: How well the algorithm can retrieve all correct matches.

- Ranking (for approximate search): How well the ranking of the results produced by the algorithm compares with that given by a human observer.
CHAPTER 4.  CPM IN LZW COMPRESSED FILES

In this chapter, we report our work on compressed pattern matching in LZW compressed files. The work includes an extension of Amir’s well-known “almost-optimal” algorithm [Amir96]. The original algorithm has been improved to search not only the first occurrence of the pattern but also all other occurrences. A faster implementation for so-called “simple patterns” is also proposed. The work also includes a novel multiple-pattern matching algorithm using the Aho-Corasick algorithm. The algorithm takes $O(mt+n+r)$ time with $O(mt)$ extra space, where $n$ is the size of the compressed file, $m$ is the total size of all patterns, $t$ is the size of the LZW trie and $r$ is the number of occurrences of the patterns. Extensive experiments have been conducted to test the performance of our algorithms and to compare with other well-known compressed pattern matching algorithms, particularly the BWT-based algorithms and another similar multiple-pattern matching algorithm by Kida et. al. [Kida98] [Kida00] that also uses the Aho-Corasick algorithm on the LZW compressed data. The results showed that our multiple-pattern matching algorithm is competitive among the best compressed pattern-matching algorithms and is practically the fastest among all approaches when the number of patterns is not very large. Therefore, our algorithm is preferable for general string matching applications. The proposed algorithm is efficient for large files and it is particularly efficient when being applied on archive search if the archives are compressed with a common LZW trie. LZW is one of the most efficient and popular compression algorithms used extensively and our method requires no modification on the compression algorithm. The work reported here, therefore, has great economical and market potential.
The plan for the remainder of the chapter is as follows. In section 4.1, we introduce Amir’s compressed pattern matching algorithm and then we present our work that is considered as an enhancement of Amir’s work. In section 4.2, we report a novel multiple-pattern matching algorithm using Aho-Corasick algorithm and compare it with the algorithm in [Kida98] and [Kida00]. In Section 4.3, we report the experimental results. Section 4.4 concludes the chapter.

4.1. Extension Of Amir’s Algorithm

In the rest of this chapter, we will use notations similar to those used in [Amir96]. Let $S=c_1c_2c_3…c_u$ be the uncompressed text of length $u$ over alphabet $\Sigma=\{a_1, a_2, a_3, …, a_q\}$, where $q$ is the size of the alphabet. We denote the LZW compressed format of $S$ as $S.Z$ and each code in $S.Z$ as $S.Z[i]$, where $1 \leq i \leq n$. We also denote the pattern as $P=p_1p_2p_3…p_m$, where $p_i \in \Sigma$ for $1 \leq i \leq m$, $m$ is the length of pattern $P$.

4.1.1 LZW Compression And Amir’s Algorithm

The LZW compression algorithm uses a tree-like data structure called a “trie” to store the dictionary generated during the compression processes. Each node on the trie contains:

- A node number: a unique ID in the range $[0, n+q]$; thus, “a node with node number ‘N’” and “node N” are sometimes used interchangeably in this paper.
- A label: a symbol from the alphabet $\Sigma$.
- A chunk: the string that the node represents. It is simply the string consisting of the
labels on the path from the root to this node.

For example, in Figure 4-1, the leftmost leaf node’s node number is 8; its label is ‘b’; and its chunk is “aab”.

At the beginning of the trie construction, the trie has \( q+1 \) nodes, including a root node with node number 0 and a NULL label and \( q \) child nodes each labeled with a unique symbol from the alphabet. During compression, LZW algorithm scans the text and finds the longest sub-string that appears in the trie as the chunk of some node \( N \) and outputs \( N \) to S.Z. The trie then grows by adding a new node under \( N \) and the new node’s label is the next un-encoded symbol in the next. Obviously, the new node’s chunk is node \( N \)’s chunk appended by the new node’s label. At the end of the compression, there are \( n+q \) nodes in the trie.

An example that was used in [Amir96] is presented in Figure 4-1 to illustrate the trie structure. The decoder constructs the same trie and uses it to decode S.Z. Both the compression and decompression (and thus the trie construction) can be done in time \( O(u) \).

\[
S = \text{aabbaabbbccccccc} \\
S.Z = 1,1,2,2,4,6,5,3,11,12;
\]

![Figure 4-1 LZW trie structure](image)

The following important observation makes it possible to construct the trie from S.Z in time \( O(n) \) without explicitly decoding S.Z:
**Observation:** When the decoder receives code S.Z[i], assuming S.Z[i-1] has already been received in the previous step (2 ≤ i ≤ n), a new node is created and added as a child of node S.Z[i-1]. The node number of the new node is i-1+q and the label of the new node is the first symbol of node S.Z[i]’s chunk. For S.Z[1], no new node is created.

In the practical implementation of the LZW algorithm, the trie size t is limited and does not depend on n. Thus, with the above observation, the LZW trie can be constructed in time $O(t)$.

Amir’s algorithm performs the pattern matching directly on the trie. To facilitate the pattern matching, the following terms of a node in the trie are defined with respect to the pattern:

- A chunk is a **prefix chunk** if it ends with a non-empty pattern prefix; the representing prefix of a prefix chunk is the longest pattern prefix it ends with.

- A chunk is a **suffix chunk** if it begins with a non-empty pattern suffix; the representing suffix of a suffix chunk is the longest pattern suffix it begins with.

- A chunk is an **internal chunk** if it is an internal sub-string of the pattern, i.e. the chunk is $p_i...p_j$ for $i>1$ and $j≤m$. If $j=m$, the internal chunk also becomes a suffix chunk.

If a node’s chunk is prefix chunk, suffix chunk or internal chunk, the node is called a **prefix node**, **suffix node** or **internal node**, respectively. To represent a node’s representing prefix, a **prefix number** is defined for the node to indicate the length of the representing prefix, a value of 0 means that the node is not a prefix node; Similarly, to represent a node’s representing suffix, a **suffix number** is defined for the node to indicate
the length of the representing suffix, a value of 0 means that the node is not a suffix node; To represent a node’s internal chunk status, an internal range \([i, j]\) is defined to indicate that the node’s chunk is an internal chunk \(p_i...p_j\); a internal range \([0, 0]\) means that the node is not an internal node. The prefix number, suffix number and internal range are computed for a node when the node is being added to the trie:

(a) The new node’s internal range is computed as function \(Q_3(I_P, a)\), where \(I_P\) is the internal range of its parent and \(a\) is its label.

(b) If the result from step (a) tells that the new node is not only an internal node, but also a suffix node (i.e. \(j=m\)), set its suffix number as \(m-i+1\). Otherwise the new node’s suffix number is set to its parent’s suffix number.

(c) The new node’s prefix number is computed as function \(Q_1(P_P, a)\), where \(P_P\) is the prefix number of its parent and \(a\) is its label.

When the new node’s label is not in the pattern, the above computations can be easily done; when the new node’s label is in the pattern, the operands of function \(Q_1\) and \(Q_3\) are all sub-strings of the pattern. Since the number of the sub-strings of a given pattern is finite, we can pre-compute the results for all possible combinations of the operands. In [Amir96], this pre-processing of the pattern is done by Knuth-Morris-Pratt automaton [Knuth77] and the suffix-trie [McCreight76] [Weiner73]. The pre-processing takes time \(O(m^2)\). Once the preprocessing is done, (a)-(c) can be computed in constant time.

The pattern matching is performed simultaneously as the trie is growing, as described in the following algorithm:

- Pre-process the pattern.
• Initialize trie and set global variable Prefix =NULL.

• For \(i=2\) to \(n\), perform the following after receiving code \(S.Z[i]\) (we will refer it as the current node)

  Step 1. Add a new node in the trie and compute the new node’s prefix number, suffix number and internal range.

  Step 2. Pattern Matching:

  (a) If Prefix =NULL, set variable Prefix as current node’s prefix number.

  (b) If Prefix !=NULL and the current node is a suffix node, check the pattern occurrence from Prefix and the current node’s representing suffix \(S_{P}\); this checking is defined as function \(Q2(Prefix, S_{P})\).

  (c) If Prefix !=NULL and the current node’s chunk is an internal chunk, compute Prefix as \(Q1(Prefix, I_{P})\) where \(I_{P}\) is the current node’s internal range.

  (d) If Prefix !=NULL and the current node’s chunk is not an internal chunk, set Prefix as the current node’s prefix number.

Function \(Q2\) can also be pre-processed by using the KMP automata and the suffix trie constructed from the pattern since both of its two operands are sub-strings of the pattern. The algorithm has a total of \(O(n+m^2)\) time and space complexity. Therefore, the algorithm is optimal when \(m<<n\) and it is sometimes called an “almost-optimal” algorithm. A tradeoff between the time and space alternatively gives \(O(n \log m+m)\) time and \(O(n+m)\) space algorithm.

Based on the work presented above, we propose a CPM method that improves the original algorithm by reporting all occurrences of the patterns. We also propose a fast implementation for so called “simple patterns”, as will be described in the following sections.
4.1.2 Reporting All Pattern Occurrences

In the pattern-matching algorithm presented in section 4.1.1, the pattern occurrence checking is performed only at step 2(b). It is obvious that the algorithm assumes that the detected pattern crosses the boundary of the nodes. Thus, if the pattern occurs inside a node, the above algorithm will not be able to find it. All three cases when the pattern occurs inside a node are shown in the Figure 4-2,

![Figure 4-2 Pattern inside a node](image)

The third case shown in Figure 4-2 can be easily fixed by checking the value of variable `Prefix`: if it is equal to the length of the pattern, a pattern occurrence is found. For the first occurrence of a pattern, the first two cases shown in Figure 4-2 never happen. We can prove it by contradiction:

Suppose $P$’s fist occurrence is in node $A$ and it is shown as either of the first two cases in Figure 4-2, if node $B$ is the parent of node $A$, it must have the same chunk as node $A$ except the last symbol from node $A$. This is to say that $P$ also occurs in node $B$. Since node $B$ is added before node $A$, an earlier occurrence of $P$ happens, this contracts with the assumption that $P$’s first occurrence is in node $A$. 

The above discussion explains why Amir’s algorithm reports only the first occurrence of $P$. To report all the occurrences of the pattern, we propose that, for each sub-pattern, a PIC (stands for “pattern is contained”) flag is maintained to indicate that the pattern is a sub-string of the node’s chunk. For example, if the chunk of a node is “bcdef”, the PIC flag of the node is set to true with respect to pattern “bcd” while it is set to false with respect to pattern “abc”.

Similar to prefix number, suffix number and internal range, each time a new node is added to the trie, its PIC flag is updated, after its prefix number and suffix number have been computed:

*If its parent’s PIC flag is on, a node’s PIC flag is also on; otherwise, if the prefix number of the node equals to the length of the pattern, set the PIC flag on;*

The reasoning of the above operation is illustrated in Figure 4-3.

The above operation turns on the PIC flag whenever it detects that the prefix number of a node equals to the length of the pattern. Once a node’s PIC flag is on, all of its offspring’s PIC flag is also on. By using the PIC flag, it is easy to identify the pattern.
occurrences inside the nodes. We will discuss pattern matching using PIC flag further in section 4.1.4.

### 4.1.3 A Simple Implementation For “Simple Patterns”

**Definition**: A simple pattern is a pattern in which no symbol appears more than once.

The examples of simple pattern are: *result, thus, world* and the examples of non-simple patterns are: *telephone, hello, pattern*. The following theorem is true for any simple pattern:

**Theorem**: Query $Q_1$, $Q_2$ and $Q_3$ can be computed in constant time for a simple pattern by using the following equations:

$$Q_1(P, I) = \begin{cases} j & \text{if } h+1 = i \\ 0 & \text{else} \end{cases} \quad (4.1)$$

$$Q_2(P, S) = \begin{cases} 1 & \text{if } h = m \\ h+1 & \text{elseif } k = m \\ 1 & \text{elseif } h+k = m \\ 0 & \text{else} \end{cases} \quad (4.2)$$

$$Q_3(I, a) = \begin{cases} [i, j+1] & \text{if } a = p_{j+1} \\ [0,0] & \text{else} \end{cases} \quad (4.3)$$

where $P_P = p_1...p_h$ ($h \leq m$), $I_P = p_i...p_j$ ($i > 1$ and $j \leq m$), $S_P = p_{(m-k+1)}...p_m$ ($m-k+1 \geq 1$), for any simple pattern $P$.

All the above equations can be easily proved.

*Proof of Q1*: if $h+1 = i$, $P_1 I_P = p_1...p_i p_j...p_j$ is obviously a prefix of the pattern and is the longest prefix of $P_P I_P$; In case $h+1 \neq i$, we can prove by contradiction: suppose there
is a pattern prefix $p_1...p_t \ (t \leq m)$, which is also a suffix of $P \downarrow P$. If $p_1$ occurs somewhere in $P \downarrow$ except the first position or somewhere in $I_P$, we have $p_r = p_1$ for some $r \ (1 < r \leq m)$ and it contradicts the definition of the simple pattern; if $p_1$ occurs at the first position of $P \downarrow$, we will have $h+1 = i$. Thus, there is no pattern prefix that is also a suffix of $P \downarrow P$.

Proof of Q2: The first two cases are obvious. In the third case, $h+k = m$, then $h+1 = m-k+1$ and $P \downarrow S_P = p_1...p_h p_{m-k+1}...p_m = p_1...p_h p_{h+1}...p_m = P$; If all the first three cases are not true and we suppose a pattern occurs, we must have some $r \ (1 < r \leq m)$ thus that $p_r = p_1$. By definition of simple pattern it cannot be true.

Proof of Q3: if $a = p_{j+1}$, $I_P a = p_1...p_j p_{(j+1)}$, the new internal sub-string of $P$ have position $[i, j+1]$; otherwise $I_P a$ is not an internal sub-string.

Thus, to search simple patterns, we simply plug the above implementations of Q1, Q2 and Q3 in the CPM algorithm presented in section 4.1.1. No pre-processing of the pattern is needed and thus the algorithm runs in time and space $O(n+m)$.

4.1.4 The Proposed Algorithm

In [Amir96], internal number is represented as $[i, j]$ where $i$ and $j$ represent the starting position and ending position of the internal chunk in a pattern, respectively. However, this representation has a problem in that the internal sub-string may appear in a pattern more than once. For example, if the internal chunk is “abc” and the pattern is “aabcaabcd”, we will not be able to represent the internal chunk by using only one set of starting and ending positions. For simple patterns, this is never the case. However, for non-simple pattern cases, a more reasonable representation is needed. Fortunately, recall that in Section 4.1.1, a suffix trie is constructed as part of the pattern pre-processing and a
sub-string of the pattern can be represented by a unique node in the suffix trie, we then represent the internal chunk as a node number in the suffix trie.

We now give our compressed pattern-matching algorithm as below:

If (pattern is a simple pattern) Use the queries and pattern matching functions for simple patterns; Else, Preprocess the pattern using suffix trie and KMP automaton and use Amir’s queries and pattern matching functions.

Initialize trie, set Prefix =NULL for each pattern

For i=2 to n Do:

1. Trie Updating:
   (a) When receive a code S.Z[i], add a new node to the trie as described before.
   (b) The new node’s suffix number for pattern P is set to its parent’s suffix number for P.
   (c) If the new node’s parent is an internal node with chunk I and the new node’s label is a, the new node’s internal number is set to Q3(I, a); If the new node’s chunk is also a suffix of P, the suffix number for P needs to be reset. The detection of the suffix chunk is easy: if the suffix trie is built, simply check if the internal number is a leaf node of the suffix trie; If no suffix trie (for simple patterns), check the right position of the internal chunk.
   (d) If the new node’s parent is a prefix node with chunk Pr and the new node’s label is a, the new node’s prefix number is set to Q1(Pr, a); Otherwise check if a is a prefix of P and set the new node’s prefix number correspondingly.
   (e) The new node’s PIC flag is set to what we discussed in section 4.1.2

2. Pattern Matching:
   (a) If Prefix =NULL
Set Prefix as the representing prefix of current node (the parent node of the new node).

If the PIC flag of the current node is set, a pattern occurrence is found.

(b) If Prefix != NULL

If one of the following two conditions is true, a pattern occurrence is found:

(1) If the current node is a suffix node and Q2(Prefix, the representing suffix of the current node) is true.

(2) If the PIC flag of the current node is set.

In either case, we need to set Prefix as the representing prefix of the current node to prepare for searching the next pattern occurrence.

If none of the above two conditions are true: if the current node is an internal node, set Prefix = Q1(Prefix, internal chunk); Otherwise set Prefix as the current node’s representing prefix.

4.1.5 A Discussion On Multiple Pattern Matching

Multiple pattern matching is often a useful feature in many applications such as when a Boolean query with many terms is performed or when interactive pattern searching is desired. Once the common “overhead” of searching different patterns is done, such as pre-processing of the compressed data or computation of an auxiliary array, it can be used to search many patterns. Thus, multiple pattern matching is not equivalent to performing pattern matching multiple times.

Amir’s algorithm can be used to perform multiple patterns matching by searching the multiple patterns simultaneously. The method requires each node to maintain a set of
\{prefix number, suffix number, internal range, PIC flag\} for each pattern being searched. Obviously, this method is rather naïve and is not expected to give good performance. In section 4.2, we report a novel multiple-pattern matching algorithm that uses Aho-Corasick algorithm and searches multiple patterns much more efficiently.

4.2. Multiple-Pattern Matching

4.2.1 Aho-Corasick Algorithm

Aho-Corasick algorithm [Aho75] is a classic solution for multiple-pattern matching. Pattern matching is performed using a trie structure, which is also called an Aho-Corasick automaton. To avoid confusion with the LZW trie, we will use the name \textit{AC automaton} throughout the remainder of this paper. The AC automaton for patterns \{he, she, his, hers\} is shown in Figure 4-4.

![Figure 4-4 An Aho-Corasick automaton](image-url)
It can be seen that in the automaton, each edge is labeled with a symbol and edges coming out from a node (we will refer the nodes as states in the remaining of the document) have different labels. If we define $R(v)$ of a state $v$ as the concatenation of labels along the path from the root to state $v$, the following is true: for each pattern $P$ in the pattern set, there is a state $v$ such that $R(v) = P$, and this state is called a final state; each state represents a prefix of some pattern; for each leaf state $v$, there is some pattern $P$ in the pattern set so that $R(v) = P$. For instance, pattern “he” is represented by state 2 and leaf state 9 represents pattern “hers”. Both states 2 and states 9 are final states.

For each state except the leaf states, a $\text{goto}(v, a)$ function is defined which gives the state to enter from state $v$ by matching symbol $a$. For instance, $\text{goto}(1, e) = 2$ means that the state to enter is 2 if we match symbol $e$ from state 1. A failure link $f(v)$ (indicated as dotted line in Figure 4-4) is also defined for each state and it gives the state to enter when a mismatch happens. The failure link $f(v)$ points to a state that represents the longest proper suffix of $R(v)$. Thus, when mismatch happens, by following the failure link, we will be able to continue the matching process since the state to enter also corresponds to a prefix of some pattern. Finally, a $\text{out}(v)$ function is defined for state $v$ that gives the patterns recognized when entering that state. Thus, for a non-final state $v$, its $\text{out}(v)$ function will be empty.

When searching the patterns, the AC automaton starts from the root of the automaton and processes one symbol from the input text $S$ at a time. Through the $\text{goto}$ functions and the failure links, the automaton changes its current state from one to another. The automaton reports the pattern occurrence if a final state is entered. The searching algorithm is given below, where !$\text{goto}(v, a)$ indicates the $\text{goto}$() function is not
defined for state \( v \) and symbol \( a \). The construction of the automaton takes time and space \( O(m) \) where \( m \) is the total length of the patterns. The search takes time \( O(u) \) where \( u \) is the size of the input text. Thus, the overall computational time of Aho-Corasick algorithm is \( O(u+m) \).

\[
\begin{align*}
\nu &= \text{root}; \\
\text{WHILE (not end of input file S)} & \{ \\
\text{get next symbol } a \text{ from S;} \\
\text{WHILE (!goto}(v, a)) & \{ \\
\nu &= f(v); \\
\nu &= \text{goto}(v, a); \\
\text{IF out}(v) \text{ is not empty THEN report patterns in out}(v) \\
\} \\
\}
\end{align*}
\]

4.2.2 Searching Multiple Patterns Using Aho-Corasick Algorithm

The basic idea of the proposed algorithm is: to be able to search the patterns in the compressed files, the AC automaton should be able to process the compressed symbols. Specifically, for LZW compression, we need an AC automaton that is still able to perform the state transitions by taking LZW trie node numbers.

The above objective can be achieved by constructing a LZW trie with a state-transition list constructed for each node, as described in this section. Consider the same example shown in Fig. 1, where the text is: \texttt{aabbaabbabcccccc} and the compressed data is \{1,1,2,2,4,6,5,3,11,12\} and we assume the patterns are: \{ \texttt{aa}, \texttt{ab}, \texttt{abc} \}. In the proposed method, the AC automaton for the patterns is first constructed, as shown in Figure 4-5.
Figure 4-5 The AC automaton for patterns \textit{aa, ab and abc}

After the AC automaton has been constructed, we then use it to create the state-transition list for each node in the initial LZW trie. Each entry in the state-transition list of a node is in the form \(v_1-v_2\), which indicates that for each state \(v_1\) in the AC automaton, the next state is \(v_2\) if the chunk of the current trie node is applied to the AC automaton. For the initial LZW trie, since each node’s chunk is simply its label, we may immediately create the state-transition list for a node by applying its label to the AC automaton. Figure 4-6 shows the initial LZW trie with the state-transition list. For example, the state-transition list for node 1 is \{0-1, 1-2, 2-2, 3-1, 4-1\}. It means that when node 1 (of label “a”) is received, if the current state is 0, the next state will be 1; if the current state is 1, the next state will be 2. For state 2, 3 and 4, there are no direct transitions with input \(a\), therefore the failure links are followed and the input \(a\) is then applied. Since the “from” state \(v_1\) exhausts all possible states and they are ordered from the lowest to the highest numbered state, the state-transition list can be written as \{1, 2, 2, 1, 1\} for node 1.
The algorithm then linearly scans the compressed data. Each time a code is received, a new node is added to the LZW trie, until there is no more space for new node. As have been discussed early in this chapter, the LZW trie can be reconstructed without explicitly decoding the data. The new node’s state-transition list, instead of being obtained by feeding its chunk to the AC automaton, can be computed directly from its label and its parent only. For example, when node 4 (whose label is “a”) is added under node 1, we do not have to construct the state-transition list by applying its chunk “aa” to the AC automaton. Instead, we could simply apply its label “a” to the AC automaton starting from the corresponding v2 state in its parent’s state-transition list. Thus, we obtain the new state-transition list: {2; 2; 2; 2; 2} because the states 1, 2, 2, 1 and 1, when receiving symbol “a”, will be changed to states 2, 2, 2, 2 and 2, respectively.

Finally, the complete trie (we ignored the state-transition information for some nodes because those nodes are not referenced during the compression) is shown in Figure 4-7.
In Figure 4-6 and Figure 4-7, the state-transition lists have been written directly under the LZW trie nodes. In actual implementation, instead of having a list for each node, a single table is constructed for the whole trie and we call this table the *state transition table*. Each row of the state transition table corresponds to one state in the AC automaton; each column of the state transition table corresponds to one node in the LZW trie. The state transition table of the initial trie in Figure 4-6 is shown below:

![Figure 4-7 The LZW trie with state-transition lists](image)

<table>
<thead>
<tr>
<th>State\Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Pattern matching can be performed using the state transition table. For the same example, at the beginning of the search, the current state is set to 0. When $S.Z[1]=1$ is received, using the state transition table, the current state is changed to 1; when the second node 1 is received, the state transition table indicates that the current state is changed to 2. The complete transition is shown in the following table, where the bold numbers in the table indicate the final states of the AC automaton.

<table>
<thead>
<tr>
<th>Node Sequence</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Sequence</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

A pattern occurs when a final state is entered.

However, the above table only shows five final states while there are six pattern occurrences in the above example. An occurrence of “$ab$” is missing from the above table. The reason is that, when we compute the third entry ($v1$ is 2) of the state-transition list of node 6, by matching the label “$b$” from the parent node’s corresponding $v2$ state, state 3, the computed state is 0, which is not a final state. However, the intermediate state, state 3, is a final state. Thus, a final state is “skipped” during the transitions and this is why the second occurrence “$ab$” is not reported. It can be shown that a final-state skipping happens only if the corresponding entry in a node’s ancestor node’s transition list is a final state. Thus, the problem can be fixed by adding a flag for each entry in the state transition list of a node and it is set to $true$ if the corresponding entry of the node’s ancestor is a final state. This flag is inheritable by a node’s offspring. During the search, if the flag is on, the current node’s chunk has to be processed symbol by symbol by the Aho-Corasick automaton so we will not miss any pattern occurrence.

79
4.2.3 Analysis And Comparison

The state transition table construction takes time and space $O(mt)$ where $t$ is the LZW trie size and $m$ is the total length of the patterns. The search time depends on how many final states are skipped during the search process and is proportional to $r$, the number of occurrences of the pattern. Thus, the search times is $O(n+r)$ and the total processing time of our algorithm is $O(n+mt+r)$. The extra space used in our algorithm is solely the cost on the state transition table, i.e. $O(mt)$, which is independent of the file size.

We now compare our algorithm with an algorithm developed by Kida et. al. [Kida98] [Kida00] as the latter has the same general idea as ours. Kida’s algorithm is also able to perform the state transition by taking the compressed data. Their algorithm first constructs the GST (General Suffix Trie) and the AC automaton of the patterns. The algorithm then relies on two main functions. The first function $\text{Next}(q, s)$ computes the next AC state from state $q$ by taking string $s$. The second function $\text{Output}(q, s)$ outputs all patterns that ends in $q.s$. Note that string $s$ is limited to only those that are represented as LZW trie nodes.

Function $\text{Next}(q, s)$ is computed as:

$$\text{Next}(q, s) = \begin{cases} N_1(q, s) & \text{if } s \text{ is substring of any pattern} \\ \delta(e, s) & \text{otherwise} \end{cases}$$

where $N_1$ is a two-dimensional table that gives the result in constant time. Since the number of AC states is $O(m)$ and the number of sub-strings of the patterns is $O(m^2)$, the
table can be constructed in time and space $O(m^3)$ using the AC automaton and the GST. The time and space can be further improved to $O(m^2)$. The state transition function $\delta(\varepsilon, s)$ computes the state from the initial state $\varepsilon$ by taking string $s$ and it can be incrementally computed in constant time when a new node is added to the LZW trie. Thus, the overall time and space complexity for $\delta$ is $O(t)$ where $t$ is the LZW trie size.

Function $Output(q,s)$ is broken into two parts:

$$Output(q,s) = Output(q,\bar{s}) \cup A(s)$$

where $\bar{s}$ is the longest prefix of $s$ such that $\bar{s}$ is a suffix of any pattern. It can be computed incrementally when constructing the trie. The first part $Output(q,\bar{s})$ can be computed by combing all outputs of the states traversed from $q$ by taking $\bar{s}$ and it takes time proportional to the number of pattern occurrences. A two-dimensional table $N_2$ is used to answer the states traversed. Since $\bar{s}$ is a suffix of a pattern and the total number of suffixes is $O(m)$, the table can be pre-computed in time and space $O(m^2)$. The second part $A(s)$ is basically the set of patterns that also begin in $q.s$ and it can be represented by $\bar{s}$ and $\hat{s}$, which is the longest proper prefix of $s$ whose suffix is a suffix of any pattern. Like $\bar{s}$, $\hat{s}$ can be computed incrementally when the trie is constructed. Therefore, it takes $O(t)$ time and space for $A(s)$ and it takes $O(m^2+t)$ time and space for the construction of the output function. As we mentioned above, the output function enumerates the patterns in time proportional to the number of pattern occurrences, i.e. $O(r)$.

Overall, Kida’s algorithm takes time $O(n+m^2+t+r)$ and extra space of $O(m^2+t)$. A comparison of the complexity between our algorithm and Kida’s algorithm is given in the following table.
Table 4-3 Comparison with Kida's algorithm

<table>
<thead>
<tr>
<th></th>
<th>Total time</th>
<th>Extra space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kida’s algorithm</td>
<td>$O(n+m^2+t+r)$</td>
<td>$O(m^2+t)$</td>
</tr>
<tr>
<td>The proposed algorithm</td>
<td>$O(n+mt+r)$</td>
<td>$O(mt)$</td>
</tr>
</tbody>
</table>

Although the size of the dictionary is proportional to the size of the alphabet, in a practical implementation, the size of the dictionary is limited by a parameter and can be regarded as a constant for large files. Therefore, for large files, the time and space complexity of our algorithm will be $O(n+m+r)$ and $O(m)$, respectively, contrasting to Kida’s $O(n+m^2+r)$ time and $O(m^2+t)$ space algorithm. The experimental results in section 4.3 will give further comparison of the two algorithms from the practical performance point of view.

4.2.4 Applying On Archival Search

When the file size is large, the extra space used in our algorithm takes only a small percentage of the total memory usage. For example, if each LZW trie node is encoded by 12-bits and the file size is 4MB, assuming the total length of the patterns is 50, then in the worst case, the extra space used in our algorithm is only 200k, which is only less than 5% of the file size.

The proposed algorithm works particularly efficient for archival search if the archives are compressed using a common LZW trie. In [Zhang05], a two-pass public trie scheme is proposed for archive compression where the archives, because of their similar characteristics, are compressed using a common LZW trie, which is pre-computed based
on a small set of the data from the archives. The common trie is stored together with the compressed data and does not need to be reconstructed during the decompression. Experimental results have shown that the compression performance in [Zhang05] is better than gzip. If we apply the compressed pattern matching algorithm on the archives, we will only need to pre-process the common LZW trie once before the search is started for the entire collection of the records. The pre-processing takes the pre-computed LZW trie and computes the state transition table, by scanning the LZW trie and applying the AC automaton for each node. In comparison to the size of the archives, which might be of hundreds of megabytes, the cost on the state transition table construction is very small.

4.3. Experimental Results

All experiments were conducted on a PC with the following configuration: CPU: Intel(R) Pentium(R) 4 1.80GHz; cache size: 512KB; total memory: 756MB. The OS is Linux 2.4.20. All data were obtained as the average of 50 runs. For pattern matching, each run uses a different set of patterns. Patterns are English words randomly selected from the file being searched. All file sizes are in bytes.

4.3.1 Compression Performance

Table 4-4 lists the compression rate for all 11 test benchmarks. Where LZW14 indicates that the LZW code is 14 bits. The compression results from the UNIX COMPRESS utility are also listed. COMPRESS uses adaptive bits encoding by first using 9 bits for the output code, when 9 bits codes are exhausted it uses 10 bits, and so on. In our experiments we used 16 bits, which is the upper limit of COMPRESS. It is
worthwhile to point out that our CPM method can be applied on COMPRESS compressed files as well. Two BWT-based compressions, \textit{bsmp}, which is used by Bell [Bell02] for the BWT-based CPM algorithms and FM-Index are also listed in the table.

Table 4-4 Compression performance

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>LZW</th>
<th>COMPRESS</th>
<th>BWT</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{alice29.txt}</td>
<td>148481</td>
<td>3.52</td>
<td>3.32</td>
<td>2.56</td>
</tr>
<tr>
<td>\textit{asyoulik.txt}</td>
<td>125179</td>
<td>3.66</td>
<td>3.51</td>
<td>2.85</td>
</tr>
<tr>
<td>\textit{bible.txt}</td>
<td>407775</td>
<td>3.15</td>
<td>2.76</td>
<td>1.79</td>
</tr>
<tr>
<td>\textit{cp.html}</td>
<td>24603</td>
<td>4.25</td>
<td>3.68</td>
<td>2.72</td>
</tr>
<tr>
<td>\textit{E.coli}</td>
<td>4638690</td>
<td>2.11</td>
<td>2.17</td>
<td>2.12</td>
</tr>
<tr>
<td>\textit{fields.c}</td>
<td>11150</td>
<td>4.44</td>
<td>3.56</td>
<td>2.43</td>
</tr>
<tr>
<td>\textit{grammar.lisp}</td>
<td>3721</td>
<td>5.30</td>
<td>3.90</td>
<td>2.92</td>
</tr>
<tr>
<td>\textit{icet10.txt}</td>
<td>419235</td>
<td>3.43</td>
<td>3.17</td>
<td>2.3</td>
</tr>
<tr>
<td>\textit{plrabn12.txt}</td>
<td>471162</td>
<td>3.57</td>
<td>3.41</td>
<td>2.74</td>
</tr>
<tr>
<td>\textit{world192.txt}</td>
<td>2473400</td>
<td>3.69</td>
<td>3.19</td>
<td>1.6</td>
</tr>
<tr>
<td>\textit{xargs.1}</td>
<td>4227</td>
<td>5.94</td>
<td>4.43</td>
<td>3.54</td>
</tr>
<tr>
<td>\textit{MEAN}</td>
<td></td>
<td>3.91</td>
<td>3.37</td>
<td>2.51</td>
</tr>
</tbody>
</table>

One may notice that the compression rate of LZW is worse than that of BWT in general. However, a follow-up entropy encoding can easily compress the LZW coded data to a compression rate close to that of BWT. It is also the case that the BWT-based CPM algorithms reported in [Adjeroh02], [Bell02] and [Ferragina01] require the compressed files to be partially decompressed and the partially decompressed data, on which the BWT-based CPM algorithms can be applied, have the same size as the uncompressed data. In contrast, our method requires no decompression thus the storage cost is much less.
### 4.3.2 Encoding And Decoding Time

Our searching method requires no modification on the LZW encode and decode algorithms. Thus the encode/decode time is very fast, as shown table V. The unit of time is second. Obviously LZW is not only much faster than bsmp and FM-I, but also faster than bzip2.

<table>
<thead>
<tr>
<th>File</th>
<th>lzw</th>
<th>bsmp</th>
<th>FM-I</th>
<th>Bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bible.txt</td>
<td>0.76</td>
<td>47.67</td>
<td>6.84</td>
<td>3.23</td>
</tr>
<tr>
<td>E.coli</td>
<td>0.71</td>
<td>62.67</td>
<td>6.76</td>
<td>3.89</td>
</tr>
<tr>
<td>World192.txt</td>
<td>0.52</td>
<td>31.63</td>
<td>4.12</td>
<td>2.00</td>
</tr>
</tbody>
</table>

(A) Encode Time

<table>
<thead>
<tr>
<th>File</th>
<th>Lzw</th>
<th>bsmp</th>
<th>FM-I</th>
<th>Bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bible.txt</td>
<td>0.46</td>
<td>3.85</td>
<td>1.64</td>
<td>0.96</td>
</tr>
<tr>
<td>E.coli</td>
<td>0.46</td>
<td>5.2</td>
<td>2.10</td>
<td>1.36</td>
</tr>
<tr>
<td>World192.txt</td>
<td>0.30</td>
<td>2.23</td>
<td>0.92</td>
<td>0.59</td>
</tr>
</tbody>
</table>

(B) Decode Time
4.3.3 Search Performance For a Single File

In Figure 4-8, the search performances of six different searching algorithms are plotted. Three of them are the BWT-based CPM algorithms including bwt-binary, bwt-qgram and bwt-suffix. (Since bwt-binary and bwt-qgram have very close performances, it is hard to differentiate them in Figure 4-8.). Three of them are LZW-based algorithms: lzw-ac, which is the multiple-pattern matching algorithm we proposed in section III; lzw-amir, which is the implementation of Amir’s original algorithm with the enhancement from section II; lzw-decompress-then-ac, which decompresses the LZW compressed data to a temporary file first and then applies the Aho-Corasick algorithm. A magnified view
of Figure 4-8 is shown as the left picture in Figure 4-9. The benchmark is *bible.txt* and the uncompressed file size is 3.88MB.

As can be seen from Figure 4-8 and its magnified view, when only a few patterns are to be searched, the proposed approach *lzw-ac* reports the results almost instantly. The search time of *lzw-ac* slowly increases with the number of search patterns increases. When the number of patterns to be searched is not very large (less than 140 in the left picture in Figure 4-9), *lzw-ac* has the best performance among all; when the number of the patterns is larger than that threshold, only the *decompress-then-ac* approach is better than *lzw-ac*. The reason that *decompress-then-ac* outperforms *lzw-ac* for large number of patterns is that the state transition table construction time in *lzw-ac* increases very fast when the number of patterns increases. The *lzw-amir* algorithm, when searching just a few patterns, has the performance next to the *lzw-ac* algorithm. However, when the number of patterns gets large, its performance dramatically decreases because of its naïve
multiple-pattern matching implementation. The BWT-based approaches are slow in general because their huge overhead of partial decompression.

The \textit{lzw-decompress-then-ac} approach can be improved by combining the decompression and the search together. Instead of waiting for the whole file to be decompressed, we can feed the characters to the AC automaton immediately after they are decoded. We call this approach \textit{lzw-decompress-ac} and a magnified view of its performance is shown as the right picture in Figure 4-9. It can be seen that the new implementation improves the search speed quite a lot. However, even with this improved version, the \textit{lzw-ac} algorithm still works the best when the number of patterns is approximately under 20.

4.3.4 Search performance For Archives

In Figure 4-10, the search performances for archives, which are compressed using a common LZW trie are shown. Only the performances of two algorithms are plotted: \textit{lzw-ac} algorithm and the \textit{lzw-decompress-ac} approach. The benchmarks are Wall Street Journal (1987-1989) and the uncompressed data size is about 270 MB. The trie is built based on a small set of the collection and is used for compressing the whole collection. It can be seen that the proposed algorithm is faster when the number of patterns is less than approximately 90. Comparing with the performances shown in the right picture in Figure 4-9, the proposed algorithm works very efficiently for archive search if the archives are compressed by the common trie.
4.3.5 Comparison With Kida’s Algorithm

Figure 4-11 compares the search performances of our *lzw-ac* algorithm and Kida’s algorithm. It can be seen that, when the length of the patterns is under approximately 40, our algorithm works faster than Kida’s algorithm. However, when the length of the patterns is larger than 40, Kida’s algorithm is faster. We think the competition of the two algorithms is highly affected by the pre-processing time of the patterns. In our algorithm, the pre-processing time is $O(mt)$ while in Kida’s algorithm the pre-processing takes time $O(m^2 + t)$. The hidden constant in the big $O$ notation and the size of the LZW trie $t$ will decide the performances of the two algorithms.
4.4. Conclusion And Future Work

The highlights of the work presented in this chapter are given below:

- We give the first implementation of Amir’s original algorithm and we enhance the original algorithm by reporting all pattern occurrences.
- The enhancement of Amir’s algorithm automatically detects patterns that we define as “simple patterns” and applies a faster implementation of the algorithm.
- A novel multi-pattern matching algorithm using Aho-Corasick algorithm is reported and the algorithm works in $O(n+mt+r)$ time and $O(mt)$ extra space.
- Extensive experiments have been conducted on the performance of our method and comparisons have made with other existing CPM algorithms. The results show that our work is competitive among the best CPM algorithms and is preferable for general string matching applications.
Further research can be conducted for the approximate pattern-matching algorithm.
CHAPTER 5.  TWO-DIMENSIONAL CPM WITH JPEG-LS

In this chapter, we report our work on compressed pattern matching for JPEG-LS compressed images. Based on our best knowledge, this is the first two-dimensional compressed pattern matching work based on the JPEG-LS standard. A series of algorithms are proposed, including: a global context variation of the JPEG-LS algorithm and the corresponding searching algorithm, a two-pass variation of the JPEG-LS algorithm and the corresponding searching algorithm. Among them, the global context algorithm is directly modified from the original JPEG-LS algorithm to achieve search-awareness. As a result, the compression of the global context algorithm is slightly worsened but the encoding and decoding speeds are both improved. The searching algorithm based on the global-context variation requires partial decompression of the compressed image. The two-pass variation, on the other hand, achieves search-awareness through a common compression technique called the semi-static dictionary. Compared to the original algorithm, the compression of the new algorithm is equally well but the encoding takes slightly longer. The searching algorithm based on the two-pass variation requires no decompression at all and therefore works in the fully compressed domain. It runs in time $O(n_c + m_c + nm + m^2)$ with extra space $O(n + m + m_c)$, where $n$ is the number of columns of the image, $m$ is the number of rows and columns of the pattern, $n_c$ is the compressed image size and $m_c$ is the compressed pattern size. The algorithm is the first known two-dimensional CPM algorithm that works in the fully compressed domain. Experimental results have shown that our approach improves the search speed significantly, comparing to the naïve “decompress-them-search” approach.
5.1. Introduction

As we noted in Chapter 1, the amount of information that we are dealing with is being generated at an ever-increasing rate. Images from sources such as defense and civilian satellites, military reconnaissance and surveillance flights, fingerprinting devices, biomedical imaging and scientific experiments are a major part of the data. For example, NASA’s Earth Observing System will generate about 1 terabyte of image data per day when fully operational [Hurson04]. What’s more, the satellite images and aerial photographs usually consist of several spectral channels and each channel uses 8 bits to encode the spectral intensity [Pajarola98]. Therefore, the data size of a typical satellite image can easily range up to several hundred megabytes.

The importance of data compression for efficiently and effectively storing this enormous amount of image data and transferring it over the limited-bandwidth network has been addressed in previous chapters. However, two practical issues must be taken into consideration:

The first concern is to choose between lossless and lossy compressions. Much of the recent research in compression has focused on lossy compression, which deliberately discards information that is not important (visually, diagnostically, or scientifically) at the time of compression. Unfortunately, lossy compression schemes may only achieve modest compression before significant information is lost. The role of lossy compression for particular applications is controversial and, if only mild levels of compression can be achieved from lossy compression, then it might be more appropriate to use lossless compression techniques [Clunie00]. For places where even a single dot or shade of color may be important such as medical imaging, fingerprinting and satellite imaging, lossless
data compression is strictly required [Amir04]. For example, in one of NASA’s recent program solicitations, it is stated that “NASA is interested in algorithms that provide lossless data compression and efficient error correction …” [NASA04].

The second concern is to decide the particular compression algorithm. When making the decision, the cost of the compression must be taken into account. The use of unusual or proprietary compression schemes has a risk associated with the end of life of equipment on which the algorithm is developed and it may also compromise interoperability with other equipment. The use of industry standards can reduce the cost and risk.

Having considered all the above factors, the JPEG-LS compression algorithm is therefore the best choice for satellite image and medical image compression. The JPEG-LS compression is preferred not only because it is the current international standard of lossless compression for continuous-tone image, but also because it offers greater benefit over other choices.

JPEG-LS offers nearly the best compression at affordable cost. To test the performance of various lossless compressions on medical images, Clunie compared the performances of lossless JPEG, JPEG-LS, lossless mode in JPEG2000, CALIC and several dictionary schemes. The benchmarks are 3679 medical images from multiple anatomical regions, modalities and vendors. It is reported that “JPEG-LS and JPEG 2000 performed equally well, almost as well as CALIC… both out-performed lossless JPEG”. It is also reported that dictionary schemes (gzip and PNG) both “performed poorly”. Clunie concluded that “JPEG-LS is simple, easy to implement, consumes less memory,
and is faster than JPEG 2000…” and recommend JPEG-LS to be incorporated into Digital Image and Communication in Medicine (DICOM) standards.

JPEG-LS offers real-time compression. In some applications, such as high-speed scanning and satellite image transmission, large image volumes need to be processed in near real time [Savakis02]. For instance, in [NASA04], NASA requires an “onboard (real-time) pulse compression …”. JPEG-LS provides real-time compression yet not scarifying the compression ratio.

Besides, JPEG-LS also offers the applications more flexibility by having the choice to be used in a lossless or near lossless manner. Because all the above benefits, JPEG-LS is incorporated into DICOM [DICOM] and a variation of JPEG-LS is incorporated in NASA’s Consultative Committee for Space Data Systems standard (CCSDS). Various commercial medical image compression products utilizing JPEG-LS are also available.

On the other hand, for this enormous amount of image data, efficient information management and retrieval must be taken into account. There are different approaches to manage and retrieve the images.

Many existing systems store textural information associated with the images, such as the comments and diagnoses made by a physician. However, the textual description is highly dependent on the knowledge on specific topics of the person who writes the descriptions, and is dependent on his purpose at the time of analysis [Traina97]. Therefore, textual data cannot substitute the actual information contained in an image.

There are also many systems called the content-based image retrieval (CBIR) system, which retrieves relevant images based on their contents. In most CBIR systems,
image contents are modeled as a set of attributes, which are extracted manually or automatically. The images are then managed within the framework of conventional database systems and queries are made using these attributes. This approach entails a high level of image abstraction. Generally, the higher the level of abstraction, the lesser is the scope for posing ad hoc queries to the image database [Gudivada95]. Therefore, meta information cannot substitute the actual information contained in an image either.

In a new direction of the content-based image retrieval, queries are made by matching a given pattern image with the images in the database. This pattern matching approach is useful in many practical applications, including applications where a certain specific type of feature in a Magnetic Resonance Imaging (MRI) is searched; applications where an aerial photograph and the template of an object are given while all locations in the aerial photograph where the object appears need to be returned, applications where the finger print of a certain person (a criminal, for instance) is given and the record of this person needs to be located in the database by searching the finger print. A typical application would be faxed documents management. If one needs to look for a logo in a faxed image of a document, one convenient way is to scan the logo and check if any portion of the fax image matches the logo image. It is easier to scan and match, for example, an IBM logo, than trying to describe it [Amir04]. A similar application would be digital library management. Due to various reasons, many documents are directly scanned from outdated books, magazines and periodicals. Although Optical Character Recognition (OCR) techniques can be used to convert these image documents to texts before searching them, OCR is still not perfect hence most of these documents are still
stored in the image format [Lu03]. In this case, pattern matching will be a great tool for hunting the relevant information.

Compared to the traditional CBIR approaches, one of the main advantages of the pattern matching approach is its ability to rank-order database images by the degree of similarity with the query image (that is, similarity-based retrieval) [Gudivada95].

Motivated from the popularity of JPEG-LS in various applications and the need to efficiently and effectively retrieve relevant images through pattern-matching approach, we study the compressed pattern matching problem for JPEG-LS compressed images in this chapter. This is a challenging problem, not only because we need perform the search on the compressed, two-dimensional data, but also because the following difficulties:

- Rotation: the pattern may appear in an image as a rotated version.
- Scaling: the pattern may appear in an image as having been scaled.
- Errors: the pattern may differently due to various reasons such as the transmission noise, atmospheric distortions etc.

The ultimate goal is to locate the rotated, scaled, approximate pattern, which may have any shape, in a compressed image. Clearly, achieving this goal is not an easy task and the research we present addresses the exact compressed pattern matching problem.

In the remaining of this chapter, we present several compressed pattern matching algorithms that are based on the JPEG-LS algorithm.

5.2. A Case Study On Lossless JPEG

Before we look at the compressed pattern matching problem for JPEG-LS, we would like to study a simpler case: compressed pattern matching on lossless JPEG
compressed files. The compression scheme of lossless JPEG is very similar to that of JPEG-LS, except that the latter is more complex. Studying compressed pattern matching problem for lossless JPEG would help us understand the problems with a typical predictive compression algorithm when it is applied in the CPM area and help us develop efficient CPM algorithms for JPEG-LS.

The prediction modes used in lossless JPEG are shown in Table 2-1. Taking the second lossless JPEG prediction mode, which takes the left pixel of the current pixel as the predicted value, as an example, and assuming the height of the pattern is one-pixel, we illustrate how compressed pattern matching works for lossless JPEG and we show the problem with the algorithm.

Assume that an integer number represents a pixel’s value, and the pattern to be searched is: \{100, 101, 102, 100, 102, 104\} in a row of the image such as: \{…, 99, 105, 100, 101, 102, 100 102, 104, 110, 105 …\}.

The prediction errors generated from the pattern is:

\{100, 1, 1, -2, 2, 2\}

The prediction errors generated from that row of image is:

\{…, ?, 6, -5, 1, 1, -2, 2, 2, 6, -5,…\}

It can be seen that, except for the first prediction error, the prediction errors of the pattern and its match in the image are identical. Therefore, to search a pattern, one can simply convert the pattern to prediction errors, partially decode the compressed image to prediction errors, and then search the prediction errors from the pattern into the prediction errors from the image, without comparing the first prediction errors of the pattern.
**Definition:** The boundary of a pattern contains the pixels that do not have a complete context to predict its value, under a given prediction mode.

For example, for prediction mode 1, which uses the pixel immediately above the current pixel as the prediction, the boundary of a pattern is the first row of the pattern. Similarly, for prediction mode 2, the boundary of a pattern is the first column of the pattern. The boundaries under prediction mode 1 and mode 2 are illustrated in Figure 5-1.

![Boundary under prediction mode 1](image1)

![Boundary under prediction mode 2](image2)

**Figure 5-1 Boundaries**

**Observation:** Under any particular lossless JPEG prediction model, the prediction errors produced from a pattern and the prediction errors produced from its matches in the image are identical except at the boundary.

The following algorithm gives the basic idea of compressed pattern matching on images compressed with lossless JPEG.

---

**Algorithm 1:** Under any particular lossless JPEG prediction mode, the pattern can be searched through the following steps:

1. Convert the pattern to prediction errors;
2. Partially decode the compressed image to prediction errors;
3. Search the prediction errors produced from step 1 into the prediction errors produced from step 3 using any existing two-dimensional pattern matching algorithm.

---

99
The above searching algorithm requires the compressed image to be partially decoded therefore it is a partial compressed pattern matching algorithm. One problem with the algorithm is that it may lead to false matches since two different patterns may have the same prediction errors except at the boundary. For example, using prediction mode 1, sub-image \{200, 201, 202, 200, 202, 204\} will have the following prediction errors: \{200, 1, 1, -2, 2, 2\}. Except at the boundary, the sub-image has the same prediction errors as the pattern previously given in this section. Therefore, Algorithm 1 will report occurrence of the pattern when it compares the sub-image with the pattern. The false match problem can be solved through a verification stage. The verification can be done by comparing any pixel value of the pattern with that of the matched sub-images.

However, it needs to be noted that the verification may not be necessary in many cases. Our experiments on satellite images and natural images have shown that it is rarely a false match when the prediction errors of the pattern and the image match each other. This observation is true even for small patterns. Besides, even when false match happens, we may consider the false matched sub-image as the “shifted” version of the original pattern, i.e., a sub-image whose pixels are shifted from the pixels in the pattern by a constant number. For many applications, the shifted version of the pattern and the original pattern present the same feature and is therefore also interested. Thus, the “shifted” matches may also need to be reported.

5.3. **Global-context JPEG-LS Variation**

In this section, we propose a modified JPEG-LS algorithm called the global context JPEG-LS algorithm. The new algorithm allows the searching of the pattern to be
performed directly in the partially compressed data. The experimental results show that the modified algorithm has an improvement of about 12-15% on encoding speed and an improvement of about 8-12% on decoding speed over the original JPEG-LS algorithm. Our results also show that the searching time in the compressed domain is about 30% faster than “decompress-then-search” method. The tradeoff is only 2-8% degradation of the compression ratio.

5.3.1 Compressed Pattern Matching For JPEG-LS

For convenience, Figure 2-2, which is used to describe the JPEG-LS algorithm, is reproduced in Figure 5-2. Throughout the remaining of this chapter, we use the same notation we used in Chapter 2 for JPEG-LS to describe our algorithms. Unless otherwise stated, $x$ denotes the “current pixel” and its value, $x'$ denotes the predicted value of $x$. $Q$ denotes the context of $x$, $Errval$ denotes the prediction error for $x$, $MErrval$ denotes the remapped prediction error and $k$ denotes the Golomb coding parameter. Again, we assume the pixel values are in the range $[0, \text{RANGE}-1]$.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$c$</td>
<td>$b$</td>
<td>$d$</td>
</tr>
<tr>
<td>$a$</td>
<td>$x$</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5-2 Casual template in JPEG-LS

The encoding steps in the original JPEG-LS algorithm are repeated in the following but with a focus on data involved in the computation.
1. Predict the value of $x$. As can be seen from the prediction algorithm presented in section 2.1.3, only pixels $a$, $b$ and $c$ are used for prediction.

2. Compute the context $Q$ of $x$. Again from section 2.1.3, only pixels $a$, $b$, $c$ and $d$ are used to calculate the context of $x$.

3. Refine the prediction by subtracting or adding the bias of $Q$ from the initial prediction.

4. Compute the prediction error (residual) by subtracting the refined prediction from the actual value of $x$.

5. Update the bias of $Q$. As can be seen from section 2.1.3, the bias is entirely decided by the accumulated prediction errors of $Q$ and a count of encoded pixels that belong to $Q$.

6. Re-map the residual.

7. Encode the remapped residual using limited length Golomb code. The coding parameter $k$ is decided by the accumulated absolute prediction errors of $Q$ and a count of encoded pixels that belong to $Q$.

We now have the following observation.

**Observation**: the original JPEG-LS encoding algorithm is not search-aware.

In step 3, bias of $Q$ is used to refine the prediction. The bias of $Q$ is computed based on the accumulated prediction errors of all preceding pixels that belong to $Q$. In this sense, the bias is an “adaptive” variable. In step 4, the parameter $k$ is used to encode the prediction errors. The value of $k$ is computed based on the accumulated absolute prediction errors of all preceding pixels that belong to $Q$. Therefore, the Golomb coding parameter $k$ is also an “adaptive” variable. The adaptive nature of the bias and the
Golomb coding parameter makes the original JPEG-LS encoding algorithm a non search-aware algorithm. For instance, if we are given a pattern P and an image I and we need to search the compressed format of P in the compressed format of I, since for a pixel $x$ in P, its preceding pixels are very unlikely to be the same as its match in I, the bias used for $x$ will be different from that for its match in I. Therefore, although the initial prediction will be the same for them, the prediction errors are different. As a result, P and its match in I will have different representation in the compressed domain.

Another problem of the original JPEG-LS is the Golomb code, which is a variable length code. Therefore, to search in the fully compressed data, we must overcome the false match problem brought from the variable length code.

In this section, we present an approach that avoids the problems brought by the variable length code. The idea is to convert both the pattern and the image to the “residual domain” and then search, as shown in Figure 5-3:

```
Uncompressed Domain
(Pattern)

Partially encode

Residual Domain
(Searching is performed here)

Partially decode

Compressed Domain
(image)
```

Figure 5-3 The general idea of JPEG-LS based CPM
To describe the details of our algorithm, Figure 5-4 shows a pattern $P$ and an image $I$, of which a sub-image $P_M$ matches exactly with the pattern $P$. For any pixel $x$ in $P$, the pixel at the same relative location in $P_M$ is named as $x_m$. Suppose the pattern $P$ and the compressed format of $I$, namely $I_c$, are both given and we need to search $P$ in $I_c$, we need to perform the procedures described below, following the above compressed pattern matching idea.

First, we need to partially compress $P$ so that $P$ is converted to residuals, i.e. we need to perform step 1 through step 6 of the compression algorithm. Ideally, the residuals produced from $P$ should be identical to those are produced from $P_M$. However, as have discussed previously, because of the bias’s adaptive nature, the residuals produced from $P$ will not be the same as those are produced from $P_M$.

One way to solve the problem is to remove the bias refinement step from the compression algorithm if it does not degrade the compression performance too much. Fortunately, the experimental results show that removing of the refinement process only have minor impact on the compression ratio. The decrease of the compression ratio is about 2% for most images that have been tested. An alternative is to pre-process the image before compression and generate a static (instead of adaptive) bias for every
context. Both the pattern and the image are compressed using the same bias set. Since the prediction errors $\varepsilon$ present a symmetric two-sided geometric distributions (TSGDs) [Weinberger00] centered at around 0, as shown in Figure 5-5, the pre-generated bias will approach to 0 when the number of pixels gets substantially large. In Figure 5-5, $P_{(\theta,s)}(\varepsilon) = C(\theta,s)\theta^{\varepsilon+s}$ where $C(\theta,s) = (1 - \theta)/(\theta^{1-s} + \theta^s)$ is a normalization factor, $0 < \theta < 1$ and $0 \leq s < 1$.

![Two-sided geometric distribution](image)

Figure 5-5 Two-sided geometric distribution

Second, we need to partially decompress $I_c$ to residuals. To decode the data from the Golomb codes, the coding parameter $k$ is needed. The computation of $k$ requires the context $Q$ to be determined. However, from step 2 of the compression algorithm, to compute $Q$, the pixels to the north, west, northwest, and northeast of the current pixel should be known to the decoder. Without full decompression, these pixels are not available and thus $Q$ cannot be determined.

This problem can be solved by merging all prediction contexts to one global context and forcing all pixels to use the global context. Therefore, the parameter $k$ can be
computed since there is no need to compute $Q$ and the accumulated absolute prediction errors can be obtained from the partially decoded data. Obviously, the global context approach may further degrade the compression since the statistical behavior of prediction errors is considered in one global context, regardless of the local characteristics of the present pixel. This has been confirmed by the experimental results, where the compression ratio is degraded by about 4 to 5%.

5.3.2 Global-context JPEG Variation

We now give our encoding and decoding algorithms based on the discussion above. Since our encoding algorithm is directly modified from the original JPEG-LS encoding algorithm and it achieves search-awareness by using one global-context instead of 365 contexts as in the original algorithm, our algorithm is called the global-context JPEG variation.

The following variables are used in our global-context compression algorithm: $N$ is the number of pixels encoded; $A$ is used to track the absolute prediction errors.

The global-context encoding algorithm is described below, referring to Figure 5-2.

**Global-Context Encoding Algorithm:**

**Initialize the variables:**

$$N = 1; A = \text{MAX}(2, \text{int}((\text{RANGE} + 32)/64))$$

**Encode the image in linear scan order.** For every pixel $x$:

1. Find the initial prediction $x’$

The prediction algorithm is in the following:
If $c \geq \max(a, b)$
   
   $x' = \min(a, b)$

else

   { 
   
   If $c \leq \min(a, b)$
      
      $x' = \max(a, b)$
   
   else
   
      $x' = a + b - c$
   
   }

2. Compute the prediction error (residual) $Errval$ by subtracting the prediction from the actual value of $x$.

   $$Errval = x - x'$$

3. Perform modulo reduction of the prediction error as shown below. After this, the prediction error $Errval$ will be in the range $[0, RANGE-1]$.

   If $Errval < 0$
   
   $Errval = Errval + RANGE$

   If $Errval > (RANGE+1)/2$
   
   $Errval = Errval - RANGE$

4. Re-map $Errval$ so that the remapped residual $MErrval$ will present a one-sided geometric distribution. The remapping algorithm is shown below.

   If $Errval \geq 0$
   
   $MErrval = 2 \cdot Errval$

   else
   
   $MErrval = -2 \cdot Errval - 1$

5. Encode the remapped residual $MErrval$ using limited length Golomb code. The encoding parameter $k$ is computed as:

   $$k = \min(k' | 2^{k'} \cdot N \geq A)$$
6. Update variables N and A:

\[
A = A + \text{ABS(Errval)};
\]

if (N == RESET)
{
    A = A>>1;
    N = N>>1;
}
N = N+1;

where RESET is a pre-defined value, usually 64. ABS represents the absolute function.

The global-context decoding algorithm is described below, referring to Figure 5-2.

**Global-Context Decoding Algorithm:**

**Initialize the variables:**

\[
N = 1; \ A = \text{MAX}(2, (\text{int})((\text{RANGE} + 32)/64))
\]

**Decode the compressed image. For every pixel x:**

1. Find the initial prediction \(x'\)

The prediction algorithm is in the following.

\[
\text{If } c \geq \text{max}(a, b) \\
\quad x' = \text{min}(a, b) \\
\text{else} \\
\quad \{ \\
\quad \quad \text{If } c \leq \text{min}(a, b) \\
\quad \quad \quad x' = \text{max}(a, b) \\
\quad \quad \text{else} \\
\quad \quad \quad x' = a + b - c \\
\quad \}
\]

Since the image was encoded (therefore was also decoded) in linear scan order, the neighboring pixels \(a, b\) and \(c\) are known to the decoder.
2. Decode the Golomb codes. The decoding parameter $k$ should be the same as the encoding parameter, and is computed as:

$$k = \min(k' \mid 2^{k'} \cdot N \geq A)$$

The decoded value is the remapped residual $MErrval$ from step 4 in the encoding algorithm.

3. The remapped residual $MErrval$ is inversed to $Errval$ as shown below.

```plaintext
if((MErrval&1))
    Errval = - ((MErrval+1)>>1);
else
    Errval = MErrval>>1;
```

4. Update variables N and A.

```plaintext
A = A + ABS(Errval);
if (N == RESET)
    { A = A>>1;
    N = N>>1;
    }
N = N+1;
```

5. Compute the pixel value $x$.

$$x = (Errval + x') \% \text{RANGE}$$

Compared to the original JPEG-LS, the above encoding and decoding algorithm only change the operations at the pixel level and these operations, like in the original algorithms, can be done in constant time. Therefore, the time complexities of the above algorithms are the same as the original JPEG-LS algorithm.
5.3.3 Pattern Matching Algorithm

The compressed pattern matching algorithm based on the global-context variation is proposed in this section. There are three main steps of the compressed pattern matching algorithm.

Step 1: partially encode the pattern. For every pixel $x$ in the pattern, step 1 through step 3 in the global-context encoding algorithm are applied, namely:

1. Find the initial prediction $x'$.
2. Compute the prediction error (residual) $Errval$.
3. Perform modulo reduction of the prediction error.

Step 2: partially decode the image. For every pixel $x$ in the image, step 2 through step 4 in the global-context decoding algorithm are applied, namely:

1. Decode re-mapped residual $MErrval$ from the Golomb codes.
2. The remapped residual $MErrval$ is inversed to $Errval$.
3. Update variables N and A.

Step 3: Search the residuals $Errval$ produced from the partially encoded pattern and those from the partially decoded image. Since each residual in the partially encoded pattern (or partially decoded image) corresponds to one pixel in the pattern (or image), any two-dimensional pattern matching algorithm in the raw data domain can be used.

5.3.4 Analysis

From our discussion in section 5.2, if some of the pixels in the pattern do not have complete neighboring information to predict their value, the prediction errors
produced from the pattern and the prediction errors produced from its matches in the image are identical except for those pixels. Those pixels are defined as boundary in section 5.2. For JPEG-LS, the boundary is defined as:

**Definition:** For JPEG-LS, the boundary of a pattern includes all pixels on the top row and on the left-most column of the pattern.

**Theorem 2:** In JPEG-LS, the prediction errors produced from a pattern and the prediction errors produced from its matches in the image are identical except at the boundary.

The theorem is obvious since the initial prediction of pixel \( x \) depends on the \( x \)'s west, north and northwest neighbors. For pixels on the boundary, these neighboring pixels are not available. Figure 5-6 illustrates this situation.

\[
\begin{array}{cccc}
3 & 1 & 2 & 2 \\
3 & 1 & 4 & 5 \\
6 & 3 & 2 & 5 \\
6 & 3 & 2 & 3 \\
\end{array}
\quad
\begin{array}{cccc}
4 & 3 & 2 & 5 \\
4 & 3 & 1 & 2 \\
5 & 3 & 1 & 4 \\
2 & 6 & 3 & 2 \\
1 & 6 & 3 & 2 \\
3 & 3 & 2 & 2 \\
\end{array}
\]

**Figure 5-6 Pattern and its match in image**

Therefore, in the compressed pattern matching algorithm proposed above, we should not compare pixels on the boundary of the pattern. The false match problem introduced by skipping comparing the boundary can be solved by a verification step. Again, practically the false match rarely occurs and sometimes may also need to be reported depending on the applications.
The compressed pattern matching algorithm proposed above can be easily adapted by lossless JPEG.

5.4. Two-pass JPEG-LS Variation

The algorithms introduced in the previous section, although allow searching the patterns without fully decompressing the image, have worse compression ratio comparing to the original JPEG-LS algorithms. In addition, the algorithms do not allow the patterns to be searched in the fully compressed domain and the searching algorithm requires partial decoding of the image. In this section, we propose a two-pass variation of JPEG-LS and the corresponding pattern matching algorithm. The two-pass compression algorithm does not only perform equally well with the original JPEG-LS in terms of compression ratio, but also has the potential to allow the patterns to be searched in the fully compressed domain.

5.4.1 Two-pass JPEG-LS Variation

The two-pass JPEG-LS compression algorithm is a semi-static dictionary approach. As can be told from the name of the algorithm, there are two passes of the encoding algorithm. The first pass scans the image in the raster scan order and computes the context data (bias of the contexts and other context-dependent information). No compression is done in this pass. The second pass scans the image again in the same order and takes the pre-computed contexts to compress the pixels. Refined prediction, prediction error mapping and entropy encoding are all done in the second pass. Figure 5-7 illustrates the two-pass compression scheme.
Same as in the original JPEG-LS algorithm, there are 365 contexts used in the algorithm and each context maintains a set of variables. The two-pass encoding algorithm is described below.

**Pass 1**: Pre-processing of the image and computation of the contexts. For every pixel $x$ in the image:

- Compute the initial prediction $x'$.
- Determine pixel $x$’s context $Q$.
- Refine the prediction by subtracting or adding the bias of $Q$.
- Compute the prediction error (residual) by subtracting the refined prediction from the actual value of $x$.
- Perform modulo reduction.

\[
\text{if } (B[Q] \leq -N[Q]) \{ \\
\quad B[Q] = B[Q] + N[Q]; \\
\}
\]
if (C[Q] > MIN_C)
    C[Q] = C[Q] - 1;
if (B[Q] <= -N[Q])
    B[Q] = -N[Q] +1;
} else if (B[Q] > 0) {
    B[Q] = B[Q] - N[Q];
    if (C[Q] < MAX_C)
    C[Q] = C[Q] + 1;
    if (B[Q] > 0)
    B[Q] = 0;
}

In the original JPEG-LS compression algorithm, A[Q], B[Q] and N[Q] are right-shifted (halved) when N[Q] exceeds a pre-defined threshold. By halving these values, the pixels that are spatially close to a pixel will have more impact on the encoding of that pixel than other pixels. By doing this, the original algorithm takes the advantage of the local characteristics. However, in the two-pass encoding algorithm, A[Q], B[Q] and N[Q] are not right-shifted because the encoding is not adaptive.

No error mapping done and encoding is performed in Pass 1. After all pixels have been pre-processed, the Golomb encoding parameter $k$ for context $Q$ is computed and stored as K[Q]. Both C[Q], the bias of $Q$, and K[Q] are stored and will be used in Pass 2.

**Pass 2**: Image Encoding. For every pixel $x$ in the image:
• Compute the initial prediction \( x' \).
• Determine pixel \( x \)'s context \( Q \).
• Refine the prediction by subtracting or adding \( C[Q] \). The bias \( C[Q] \) is the final bias of \( Q \) after Pass 1 is finished.
• Compute the prediction error (residual) \( \text{Errval} \) by subtracting the refined prediction from the actual value of \( x \).
• Perform modulo reduction.
• Re-map \( \text{Errval} \) to \( MErrval \).
• Encode the remapped residual \( MErrval \) using limited length Golomb code with parameter \( K[Q] \).

**Output the contexts:** the context data \( C[Q] \) and \( K[Q] \) are also output as part of the compressed data. The decoder needs the same context data to decode the compressed data.

In the original algorithm, the computations of both encoding parameter \( k \) and the bias are adaptive since the algorithm takes the cumulative errors and computes \( k \) and the bias adaptively. However, in the two-pass encoding algorithm, the computations of \( k \) and the bias are based the overall cumulative errors. Since \( k \) should be computed based on the statistical behavior of the prediction errors, the parameter \( k \) computed after Pass 1 in the two-pass encoding algorithm is optimal and therefore should give better compression than the original algorithm. On the other hand, the bias of \( Q \) is used to adjust the prediction for pixels belonging to \( Q \). The adjustment should be based on the local
characteristic of the pixels. Therefore, by using a final bias that is computed after Pass 1, the two-pass encoding algorithm takes no advantage of the local characteristics.

The decoding algorithm is described in the following. Since the image was encoded (therefore was also decoded) in linear scan order, the neighboring pixels $a$, $b$ and $c$ and $d$ are know to the decoder.

**Load the contexts.** Load the Golomb parameters $K[Q]$ and biases $C[Q]$ from the compressed data.

**Decode the image.** For every pixel $x$ in the image:

- Compute the initial prediction $x'$.
- Determine pixel $x$’s context $Q$.
- Refine the prediction by subtracting or adding the pre-loaded bias $C[Q]$.
- Decode the Golomb codes using the pre-loaded decoding parameter $K[Q]$. The decoded value is the remapped prediction error $M_{Errval}$.
- The remapped residual $M_{Errval}$ is inversed to $Errval$.
- Compute the pixel value $x$.

$$x = (Errval + x') \% \text{RANGE}$$

Comparing to the original decoding algorithm in JPEG-LS, the above decoding algorithm does not need to compute the context data and is therefore faster.

The two-pass compression algorithm is analyzed in the followings.
First, it will be very interesting to compare the compression performance of the two-pass algorithm with that of the original algorithm in JPEG-LS. As have discussed in the algorithm, the bias computation after Pass 1 takes no advantage of the local characteristics of the pixels and may cause worse prediction. Meanwhile, the entropy encoding parameter computed after Pass 1 should be able to somewhat compensate the loss of compression performance since the encoding parameter is computed based on the statistical behavior of the prediction errors of the entire image. Therefore, we need to know, in a practical implementation of the two-pass algorithm, how much the two factors affect the final compression.

Second, the two-pass algorithm is search-aware, even in the fully compressed domain. Given a pattern, we encode the pattern using the same contexts used for compressing the image (Keep in mind that the contexts are part of the bit stream and can be loaded for pattern compression). Except at the boundary of the pattern, the encoded pattern will have identical binary representation as its match (if there is any) in the compressed image since the Golomb encoding parameter is the same for a pixel in the pattern and for a pixel in the image as long as the two pixels belong to the same context $Q$.

5.4.2 Pattern Matching Algorithm

We now discuss how a pattern can be searched in the compressed domain under the two-pass variation of the JPEG-LS algorithm. If pattern matching to be performed in the fully compressed domain, we need to search the entropy encoded prediction errors of the pattern in that of the image. However, like Huffman code, the Golomb code using in
JPEG-LS is also variable length code. The variable length of codes means the false match problem illustrated in Figure 3-1 also exists for Golomb code. Besides, the variable length code also makes it difficult for the vertical alignment in a two-dimensional pattern matching algorithm.

For pattern matching in the fully compressed domain under the two-pass JPEG-LS variation, the following two problems need to be taken into account:

- **Row-matching**

  In a particular two-dimensional pattern matching algorithm, to search the two-dimensional pattern, each row of the pattern needs to be searched first. The rows of the pattern can be searched simultaneously by using Aho-Corasick automaton. In the compressed domain, the compressed rows of the pattern can also be searched in the same way. However, searching the Golomb codes may lead to false matches, as have said previously. Therefore, an approach to identify the boundaries of the Golomb codes is needed since the false match can be detected by checking if a match occurs on the boundary of the codes. The idea is illustrated in Figure 5-8.

![Figure 5-8 Code boundaries](image.png)
• **Vertical Alignment**

The vertical alignment, also called the column matching, is important in two-dimensional pattern matching since it checks the matched rows to make sure they are aligned vertically in the image. To align the rows, we need to know the locations where the matched rows occur in the image. Specifically, we need to know the relative row numbers and column numbers of the matches in the image. Since the Golomb code is variable length, it is not possible to get the location of a match by a simple computation such as dividing the total number of bits by the number of bits per code for a fixed length code. The vertical alignment problem in the compressed domain is illustrated in Figure 5-9.

![Figure 5-9 Vertical alignment](image)

The location of the matches, however, can be computed easily from the number of pixels have been scanned, provided that the dimension of the image is known. To get a count of the pixels when scanning the compressed image, we may check the boundaries of the Golomb codes and increase the count by one each time a code boundary is encountered. This also implies that the fast two-dimensional pattern matching algorithm introduced in [Baeza-Yates93] is not suitable for us since the algorithm “jumps” from
rows to rows. On the contrary, the two-dimensional pattern matching algorithm introduced in [Bird77] is more feasible since it also scans the image in the linear scan order.

Obviously, to solve the row-matching problem and the vertical alignment problem in the compressed domain, the identification of code boundaries is the key.

We now take a look at the Golomb code, which is used in JPEG-LS for entropy encoding. When an integer is encoded by Golomb code, it is firstly divided by $2^k$, where $k$ is the Golomb encoding parameter. The integer division result is represented by a unary code that terminated in a 1. The remainder of the division is simply a $k$-bit unsigned binary representation. For example, to encode 18 with encoding parameter $k=2$, we divide 18 by $2^2 = 4$. As a result, the integer division is 4 while the remainder of the division is 2. Therefore, the unary part of the code is 00001 and the $k$-bit part of the code is 10. The entire code is obtained as 0000110.

The unary part of a Golomb code can be easily identified whenever a bit 1 is detected. The other part of the Golomb code, however, cannot be identified since the value of $k$ varies for different codes. If we let $k$ to be a fixed number, the code boundaries can be easily checked. As a matter of fact, based on [Wu2004], the satellite images in NASA are divided into blocks and a fixed Golomb encoding parameter is used to encode all pixels in a block.

To use fixed encoding parameter, the two-pass JPEG-LS variation needs be slightly modified. In the encoding algorithm, instead of using $K[Q]$ computed from Pass 1 to encode the prediction errors, a fixed Golomb encoding parameter $k$ is used for a
block of image if the image is divided into blocks or for the entire image. Similarly, in
the decoding algorithm, the fixed parameter $k$ is used instead of $K[Q]$.

The fixed Golomb parameter $k$ can be obtained in either of the following two
ways:

- To obtain from the empirical studies. For each different category of images, or for
  images with different features, an optimal $k$ is empirically obtained.
- To obtain from Pass 1 of the encoding algorithm. After Pass 1 of the encoding
  algorithm, the fixed Golomb encoding parameter $k$ is computed as:

$$N = \sum_{Q=1}^{364} N[Q]$$
$$A = \sum_{Q=1}^{364} A[Q]$$
$$k = \min(k' \mid 2^{k'} \cdot N \geq A)$$

Finally, we propose the following compressed pattern matching algorithm based
on the two-pass JPEG-LS variation. The two-dimensional pattern matching idea from
[Bird97] is used in the proposed algorithm for searching the pattern. We assume the
number of rows and the number of columns of the pattern is $m$, the size of the
compressed pattern is $m_c$, the number of pixels on a row of the original image is $n$, and
the size of the compressed image is $n_c$.

Variables used in the algorithm are:
• Variable \textit{count} is used to count the number of pixels (Golomb codes) have been scanned;

• Variable \textit{state} is the current state of the AC automaton built from the compressed rows of the pattern;

• Variable \textit{a}[i] (1 \leq i \leq n) is used for vertical alignment on column \textit{i}.

• Variable \textit{t}[a] (1 \leq a \leq m) is the failure function computed from the KMP automaton by taking each row of the pattern as one symbol.

The algorithm is describe below:

Compress the pattern: The pattern is compressed using the biases \textit{C}[Q] pre-loaded from the compressed image and the fixed Golomb parameter \textit{k}.

Set \textit{a}[i] = 1 for 1 \leq i \leq n.

Build KMP automaton from the rows of the pattern.

Build AC automaton from the compressed rows of the pattern.

\textit{count} = 0;

For \textit{j} = 1 to \textit{n_c} do

{ 
  \textit{row} = \textit{count} / \textit{n}; \textit{col} = \textit{count} \% \textit{n}.

  If \textit{col} = 1, then \textit{state} = 0;

  Read the next bit \textit{b} from the compressed image.

  \textit{state} = the state returned by the AC automaton after bit \textit{b} is processed.

  Check bit \textit{b}, if the current Golomb code ends with \textit{b}, then

  \textit{count} = \textit{count} + 1;
If state is a final state, then

\[ VAlignment(row, \ col, state) \]

Else

\[ a[\col] = 1; \]

}\}

where the function \( VAlignment() \) is defined as:

\[ VAlignment(row, \ col, state) \]

\{

\[ a = a[\col] \]

While \( a > 0 \) and the \( a \)\(^{th} \) row of the pattern is not reported in state do:

\[ a = t[a] \]

\[ a[\col] = a + 1; \]

If \( a = m \), then report pattern found at \((row, \ col)\)

}\}

The compression of the pattern takes time proportional to the size of the pattern. Therefore, the compression of the pattern takes time \( O(m^2) \). The KMP automaton can be built from the pattern in time \( O(m + m^2) \) since there are \( m \) rows of the pattern and we need to process all pixels on the rows before the KMP construction can take place. The AC automaton construction takes time proportional to the size of the compressed pattern, i.e. \( O(m_c) \). The scanning of the compressed image takes time \( O(n_c + nm) \). Overall, the time complexity of the algorithm is \( O(n_c + m_c + nm + m^2) \). In comparison, the naïve “decompress-
then-search” approach takes time $O(n^2 + m^2)$ when using the algorithm from [Bird97] to search.

The extra space used in the above algorithm is $O(n + m + m_c)$, for storing vertical alignment variables $a[i]$ (1 ≤ $i$ ≤ $n$), KMP failure function $f[a]$ (1 ≤ $a$ ≤ $m$) and AC automaton.

5.5. Experimental Results

All experiments were conducted on a PC with the following configuration: Intel(R) Pentium(R) 4 3.2GHz; cache size: 512KB; total memory: 2GB. The OS is Linux

The 27 test images are from the DOI-10M - National Imagery and Mapping Agency. The images cover the region from 35 N, 34 E (northwest corner) to 29 N, 40 E (southeast corner). Most images cover an area that is one degree east-west and one-half degree north-south. The total image size is about 1.35 gigabytes. One test image is shown in Figure 5-10.

![Figure 5-10 A satellite image](image.jpg)
• Compressed Performances

The compression performances, including the compression ratio, the encoding and decoding time are given in Figure 5-11, Figure 5-12 and Figure 5-13, respectively, where "jpegls" indicates the original algorithm, "global" indicate the global-context JPEG-LS variation, "twopass_v" indicates the two-pass JPEG-LS variation.

It can be seen that the compression ratio of two-pass algorithm is almost equal to the original JPEG-LS algorithm. This is very impressive comparing with the compression ration achieved by the global context JPEG-LS variation global, where the compression ratio is slightly worse than the original JPEG-LS algorithm. The encoding time of the two-pass algorithm, as shown in Figure 5-12, is the longest among all algorithms because it takes two passes while the original algorithm and the global-context variation take only one pass to compress the images. This is a cost to pay for bring the search-awareness to the algorithm. Both the encoding time and the decoding time of the global context variation are the fastest among all. The decoding time of the two-pass algorithm is also faster than the original JPEG-LS decoding algorithm because the context data is pre-loaded and need not to be computed during decoding.
Figure 5-11 Compression

Figure 5-12 Encoding time
Searching Performance

The pattern searching performance is given in Figure 5-14. The improvement of the search speed of the global context algorithm over the naïve decompress-then-search approach is plotted. It can be seen that, the search speed is significantly improved by applying compressed pattern matching.
In this chapter, we proposed two variations of the JPEG-LS compression algorithm: a global-context algorithm and a two-pass algorithm. Comparing to the original algorithm, the new algorithms are search-aware. The pattern-matching algorithms based on the two variations are also presented. The global-context based searching algorithm requires the compressed image to be partially decoded before the search can start while the two-pass compression based searching algorithm requires no decompression at all.

The experimental results have shown that the two-pass algorithm, besides having the property of search-awareness, can achieve the same compression ratio with the

5.6. Conclusion And Future Work
original JPEG-LS algorithm. The decoding times of all our algorithms are also faster than the original algorithm. The encoding time of the two-pass algorithms are longer than the original algorithm because the algorithm takes two passes to compress the image to achieve search-awareness. The search performance showed from the experimental results also indicated that the compressed pattern matching significantly improves the searching speed.

The above algorithms can be modified for lossless JPEG and even CALIC algorithm since both of the two predictive compression algorithms work similarly as JPEG-LS. Lossless JPEG may be easier to be modified while CALIC may give a better compression.

The algorithms proposed in this chapter are, to our best knowledge, the first reported work on JPEG-LS based compressed pattern matching.

It is worth to point out that, although the fixed coding parameter approach does allow the code boundaries to be identified, it will degrade the compression ration comparing to the original JPEG-LS algorithm. In our future works, a better scheme is needed to solve the boundary identification problem while maintain the compression performance of the two-pass algorithm. The practical performance of the CPM algorithm based on the two-pass compression algorithm also needs to be obtained.
CHAPTER 6. CONCLUSIONS

In this dissertation, we have studied compressed pattern matching problem for both text compression and image compression. For text compression, the popular LZW compression has been examined and compressed pattern matching algorithms based on the LZW compression algorithm are proposed. We have extended Amir’s almost-optimal algorithm by reporting all pattern occurrences instead of reporting only the first occurrence. We have developed a novel multiple-pattern matching algorithm, which works equivalently well with the best existing multiple-pattern matching algorithm. All algorithms have been implemented and extensive experimental results have been conducted to test the practical performances of the algorithms and to compare with other existing algorithms. The experimental results have shown that our multiple pattern matching algorithm is practically the fastest among all algorithms, when the number of patterns is not very large. Therefore, our compressed pattern matching algorithm is preferable for general pattern matching applications. For image compression, the current still image lossless compression standard JPEG-LS has been examined. Two variations of the original JPEG-LS algorithm have been developed that are search-aware. The first variation, called the global-context variation, requires the compressed image to be partially decompressed. The second variation, called the two-pass variation, requires no decompression at all for the pattern matching to be done in the compressed domain. The experimental data have shown promising results of compressed pattern matching for JPEG-LS.

The future works include developing approximate pattern matching algorithms for LZW compressed files, further performance improvement of the developed algorithms,
implement of the two-pass variation based CPM algorithm and adapting the compressed pattern matching algorithms for other predictive coding algorithms such as CALIC.
LIST OF REFERENCES


136


