Pattern Matching in Compressed Texts and Images

By Don Adjeroh, Tim Bell
and Amar Mukherjee

Contents

1 Introduction
1.1 Data Compression Methods
1.2 Compressed Pattern Matching
1.3 Compressed Domain versus Transform Domain Analysis
1.4 Organization

2 Search Strategies
2.1 The Pattern Matching Problem and its Variants
2.2 Search Strategies for Text
2.3 Search Strategies for Images

3 Relationship Between Searching and Compression
3.1 Pattern Matching for Compression
3.2 Compression for Pattern Matching

4 Searching Compressed Data:
Performance Measurement
4.1 Performance Measures for Compression Algorithms
Pattern Matching in Compressed Texts and Images

Don Adjeroh\(^1\),* Tim Bell\(^2\),† and Amar Mukherjee\(^3\),‡

1 Lane Department of Computer Science and Electrical Engineering, West Virginia University, Morgantown, WV 26506-6109, USA, don@csee.wvu.edu
2 Department of Computer Science and Software Engineering, University of Canterbury, Christchurch, New Zealand, tim.bell@canterbury.ac.nz
3 School of Electrical Engineering and Computer Science, University of Central Florida, Orlando, FL 32816, USA, amar@eecs.ucf.edu

Abstract

This review provides a survey of techniques for pattern matching in compressed text and images. Normally compressed data needs to be decompressed before it is processed, but if the compression has been done in the right way, it is often possible to search the data without having to decompress it, or at least only partially decompress it. The problem can be divided into lossless and lossy compression methods, and then in each of these cases the pattern matching can be either exact.

*This work was partially supported by grants from the US National Science Foundation IIS-0228370 and IIS-0312484.
†Supported by a grant from the Hitachinaka Techno Center Inc, Japan.
‡This work has been partially supported by grants from the US National Science Foundation IIS-9977336 and IIS-0207819.
or inexact. Much work has been reported in the literature on techniques for all of these cases, including algorithms that are suitable for pattern matching for various compression methods, and compression methods designed specifically for pattern matching. This work is surveyed in this review. The review also exposes the important relationship between pattern matching and compression, and proposes some performance measures for compressed pattern matching algorithms. Ideas and directions for future work are also described.

*Keywords*: Compressed pattern matching; text compression; image compression; performance measures; searching.
Given a text sequence (the database) and a pattern sequence (the query), the pattern matching problem is to search in the text to determine all the locations (if any) where the pattern occurs. Searching for a pattern is an important activity that is performed on a daily basis, with significant applications in a diverse range of areas, including signal processing, computer vision, robotics, pattern recognition, data mining, computational biology, and bioinformatics, to name a few. Today computers are increasingly being used to process text, digitized images, digital video, and various other types of data. However, representing the data can require large amounts of storage space, and operations on the data, such as searching for a pattern, are often time consuming. Even the amount of image or textual data typically available to the ordinary user has witnessed a tremendous growth due to a number of factors, such as improvements in storage and communication technologies (for example, the web, electronic mail, smartphones, general wireless mobile devices), widespread deployment of digital libraries, improved document processing techniques, and the availability of different types of sensors, including cameras and scanners. Apart from
the problem of sheer size, the huge amounts of data involved also pose problems for efficient search and retrieval of the required information from the stored data.

Since the digitized data are usually stored using compression techniques, and because of the problem of efficiency (in terms of storage space, computational time, and power consumption), the trend now is to keep the compressed data in its compressed form for as much time as possible. This means that operations such as search and analysis on the data (be it text or images) is ideally performed directly on the compressed representation, without decompression, or at least, with minimal decompression. Intuitively, compared to working on the original uncompressed data, operating directly on the compressed data will require the manipulation of less data, and hence should be more efficient. This also avoids the often time-consuming process of decompression, and the problem of temporary storage space that may be required to keep the decompressed data. The need to search data directly in its compressed form has been recognized by international compression standards such as MPEG-4, MPEG-7 [1, 311] and H.264 [373], where part of the requirement is the ability to search for objects directly in the compressed video.

This review surveys techniques that solve the two basic problems of efficiency (in storage and computation) at the same time. That is, the digitized image or text is stored and searched in a compressed format. Without addressing both problems together, compression and searching work against each other, since a simple system would have to decompress a file before searching it, thus slowing down the pattern matching process. However, there is a strong relationship between compression and pattern matching, and this can be exploited to enable both tasks to be performed efficiently at the same time.

In fact, pattern matching can be regarded as the basis of compression. For example, a dictionary compression system might identify the occurrences of an English word in a text, and replaces these with a reference to the word in a lexicon. The main task of the compression system is thus to identify patterns (in this example, words), which are then represented using a compact code. The identified pattern need not be an exact match every time (for example, in lossy compression).
Thus, the strong connection between searching and compression can be traced way back to the early days of lossy compression of signals. For instance, Shannon [303, 304] proposed block source coding with a fidelity criterion (essentially the basis for vector quantization), whereby the encoder uses a reproduction codebook (akin to the dictionary above), and searches in the codebook for the codeword with a minimum distortion to the input vector, and transmits the index for the codeword. If the type of pattern used for compression is the same as the type being used during a later search of the text, then the compression system can be exploited directly to perform a fast search. In the example of the dictionary system, if users wish to search the compressed text for words, then they could look up the word in the lexicon, which would immediately establish whether a search will be successful. If the word is found, then its code could be determined, and the compressed text searched for the code. This will considerably reduce the amount of data to be searched, and the search will be matching whole words rather than a character at a time. In one sense, much of the searching has already been performed off-line at the time of compression.

The potential savings are significant. Text can typically be compressed to less than one-third of its original size, and images are routinely compressed to a tenth or even a hundredth of the size of the raw data. These factors indicate that there is considerable potential to speed up searching, and indeed, systems exist that are able to achieve much of this potential saving. For instance, compressed domain indexing and retrieval is the preferred approach to multimedia information management [210, 234] where orders of magnitude speedup have been recorded over operations on uncompressed data [7, 384].

There are various surveys on the general problem of content-based image and video retrieval [100, 315, 318]. None has focused on the restricted problem of compressed domain image retrieval. Similarly, pattern matching and compression were considered in the survey paper [97]. However, compressed pattern matching was not considered, and various techniques have been developed since then. More importantly, we try to compare and contrast methods that have been proposed for compression, and for compressed pattern matching in text and images. Thus we consider compressed pattern matching for both lossy
compression (used for images) and for lossless compression (used for text and images).

1.1 Data Compression Methods

We begin with a brief introduction to data compression methods. More complete introductions to the topic of compression are available [50, 91], and the reader is referred to textbooks for more details [3, 51, 264, 294, 298]. There is also an IEEE conference series on data compression from 1991 to the present — see http://www.cs.brandeis.edu/~dcc/.

Compression methods are generally classed as lossless or lossy. Lossless methods enable the original data to be recovered exactly, without any error. Lossless compression, sometimes called text compression, is typically used for text, and to a lesser extent on images, such as in medical imaging where exact reconstruction of the original image is important. In contrast, lossy methods allow some deterioration in the original data, and are generally applied in situations where the data have been digitized from an analog source (such as images, video and audio). Usually the level of deterioration is near-imperceptible, yet considerable compression improvement can be achieved because the system is not storing unnecessary detail. Many lossy methods include a lossless method as a sub-component. For example, an image or video compression system might transform the image to a frequency domain, quantize some of the frequency domain coefficients (which is a lossy step), and then encode the coefficients using a lossless method.

Figure 1.1 shows a general model of what happens in data compression. The data transformation stage transforms the input data into a form that will make it easier to compress, for instance by exposing the redundancies or repetitions in the data. Some transformation schemes convert the input data or signal into the frequency domain, with the aim of packing most of the energy in the signal into only a few transform coefficients. The specific transformation performed depends on the type of input data. For text, the transformation could be a simple cyclic permutation of the text sequence (for example, the Burrows–Wheeler transform, or BWT [3, 70]); for images, we could have spatial prediction where the original image is essentially represented as a sequence
Fig. 1.1 General model for data compression. The triangle markers show the points where pattern matching on the transformed or compressed data can take place.

of prediction errors; for audio signals we could have temporal prediction on the signals; and for video, spatio-temporal prediction using motion vectors can also be viewed as a form of transformation. Standard linear transforms that pack the energy into a few coefficients, such as the Fourier transform or the wavelet transform, are quite common in image, video and audio compression. The particular transformation applied will have an impact on the compression performance, and on the ability to search the compressed data.

The quantization stage is used to reduce the number of distinct values in a signal to a much smaller number. Input data or a signal may contain a large number of distinct values. For analog signals (without digitization), the signal is continuous and the digitization process must quantize it into a set of distinct values, so a digital video has already had some quantization applied. Transforming the digitized signal puts the representation into a domain where the effect of further quantization will be less perceptible. Quantization of the transformed signal reduces the representation to a smaller set of distinct values, each of which can be represented using fewer bits, thus requiring reduced data storage. Quantization however leads to reduced accuracy in the data representation. In fact, the quantization stage represents the major source of compression, but also the major source of data loss (error) in the reconstructed data. In the case of a channel with limited bandwidth, the data rate available will determine the level of quantization needed, and in this case the quantization can be seen as maintaining the best possible fidelity for the capacity available. In some cases for images, video, or audio, it may be possible to keep the loss of accuracy to a level where
it isn’t observable by a human, given the limitations of human perception. Thus, quantization is common in such applications. Quantization could be performed on either scalar or on vector values, leading to the respective notions of scalar quantization and vector quantization. The encoding stage (also called the coding stage) codes the data to remove further redundancies, often based on the probability distribution of symbols in the data, or symbols in the quantized data for lossy compression.

Decompression involves performing the reverse operations of decoding, de-quantization and inverse transformation. The operations before and after the quantization stage are generally reversible and hence do not introduce any loss or artefacts in the compression. (Here, we ignore errors due to limited arithmetic precision, such as round-off errors during the transformation stage). Quantization, however, is not reversible and thus introduces some error in the compression process. In effect, from the viewpoint of compression models, the major difference between lossless and lossy compression is the quantization stage: lossless compression does not involve any quantization. The quantization stage also accounts for the huge compression ratios often achievable in lossy data compression.

1.2 Compressed Pattern Matching

In traditional pattern matching one is given a text and a pattern, and the problem is to determine whether the pattern occurs in the text. The search could be exact, whereby the pattern matches a substring of the text with no errors, or could be inexact or approximate, whereby some mismatches or errors could be allowed in the match. The result could be simply a binary decision on whether the pattern occurs in the text, a count of the number of occurrences, or possibly a listing of all the locations of occurrences (if any). Variations of the pattern matching problem have been studied, including multiple pattern matching, parameterized pattern matching, and multi-dimensional pattern matching. Compressed pattern matching involves one or more of the pattern matching variants, with the constraint that either the text, the pattern, or both are in compressed form [15, 133]. In general,
exact pattern matching is a natural fit for lossless compression, though inexact pattern matching can also be performed on lossless compressed data. On the other hand, given their nature, one can only hope for approximate matches for lossy compressed data. We identify two major categories for compressed pattern matching problems. *Fully compressed pattern matching* is when the text and the pattern are both compressed, and matching involves no form of decompression. Let $C$ be a compression scheme. Then, given $T_c = C(T)$, the compressed version of the text $T$, and $P_c = C(P)$, the compressed form of the pattern $P$, the problem of fully compressed pattern matching is to determine all the positions in $T$, where the pattern $P$ occurred, without first decompressing $T_c$ and $P_c$. The more general term *compressed pattern matching* is used for the following less restricted problem: Given $T_c = C(T)$, and the (uncompressed) pattern $P$, determine all the positions in $T$, where the pattern $P$ occurred, without first decompressing $T_c$. Most efficient pattern matching algorithms perform some type of preprocessing (either on the prefixes or the suffixes) of the pattern, $P$. Thus fully compressed pattern matching is made more difficult, since the usual preprocessing on the pattern is no longer easy to achieve without decompressing $P$.

For text, various algorithms have been proposed for both variants of the compressed pattern matching problem. For compressed-domain image retrieval, the usual assumption is that the query image is not compressed. However, some methods on 2D image pattern matching also consider fully-compressed pattern matching.

## 1.3 Compressed Domain versus Transform Domain Analysis

Our primary focus in this review is on compressed pattern matching for text and images. Compressed pattern matching is one activity under the general area of compressed domain analysis. In image and video analysis, the terms “compressed domain” and “transform domain” are often used interchangeably. Here we distinguish between the two. With compressed domain analysis, the required analysis is performed on the compressed data with minimal or no decoding, and before the stage of inverse quantization. For transform domain analysis, the required analysis is performed on the transform coefficients, typically
Fig. 1.2 Transform domain versus compressed domain analysis. Whether operating in the transform domain or compressed domain depends on the point(s) in the compression pipeline where the analysis is performed. The type(s) of analyses are characterized as follows: transform domain: points (1), (2), (4), and (5); compressed domain: point (3); compressed domain with partial decoding: points (4) and (5).

after the transformation or quantization stages in the compression pipeline, but before the final encoding stage. Here the objective of the transformation could be simply for efficient analysis, and not necessarily for compression or data storage. This is typical in some signal analysis or image analysis applications. If the data is already compressed, transform domain analysis could be performed after decoding the compressed stream, but before the inverse transformation stage. Figure 1.2 uses the compression pipeline of Figure 1.1 to explain the difference between compressed domain and transform domain analysis. For images or video, compressed (or transform) domain analysis could include general operations, such as image enhancement, noise removal, shape, color or texture-based image retrieval, etc. Recent surveys on general compressed or transform domain analysis of images and video are provided in [254, 343]. From the description above, most video and image retrieval and search operations, especially those for lossy compression, are performed in the transform domain (mainly after the stages of transformation, or quantization in a few cases, but generally before the encoding stage). Some text pattern matching methods operate in the transform domain (with partial decoding), while others operate in the compressed domain, that is, after final encoding, such as using Huffman codes.

1.4 Organization

In this review, we survey compression methods for text and images, especially identifying the search techniques that they use, and how they
could be exploited for searching the compressed data later. First, we consider search strategies and pattern matching methods for uncompressed data, to set the scene for more sophisticated systems. Next we explore the interesting relationship between searching and data compression. This is followed by a discussion on performance measurement for compressed pattern matching. The next two sections survey techniques that have been developed for searching compressed data, which is sometimes called \textit{compressed-domain pattern matching}. The first section looks at methods for lossless data compression (as used for text). The next section then surveys methods that have been proposed for pattern matching on compressed images. This is presented in two parts. The first part considers methods for pattern matching on lossless compressed images, which often borrow a lot from methods for searching compressed text. The second part surveys methods where the image has been compressed using lossy methods. The review concludes with a speculation on the likely directions of future work in the area.

Given the breadth of the topics involved, we will focus only on compressed pattern matching for text and images. Video and audio will be mentioned at times, but without much detail. Also, we will only briefly describe other types of signal processing activities (different from pattern matching) that are often performed in the compressed domain. For lossless compression (for text and images), we focus mainly on exact pattern matching. For lossy compression, we focus mainly on compressed domain or transform domain retrieval. Throughout this work, we assume that data from analog sources, such as images, audio and video, have already been digitized. Thus we ignore the potential data loss due to the digitization process.
Search Strategies

Searching for patterns is an important function in many applications, for both humans and machine. The pattern matching problem can be stated as follows: given a query string (the pattern), and a database string (the text), find one or all of the occurrences of the query in the database. The problem then is to search the entire text for the requested pattern, producing a list of the positions in the text where a match starts (or ends). In this section, we describe the pattern matching problem, and the key techniques that have been proposed for its efficient solution. We also briefly describe the image pattern-matching problem, which is more usually considered as a problem of image retrieval. Although this section deals with searching uncompressed data, the techniques described are often used as the foundation for compressed pattern matching methods.

2.1 The Pattern Matching Problem and its Variants

The problem of searching for patterns in a text is usually approached as a string pattern matching problem: given two strings, determine whether they are matches or not. Pattern matching is a basic task in
text retrieval. As noted by Grossi and Vitter [144], a pattern matching system is expected to be able to support three basic types of queries, namely, existential query: returns a binary value (true or false) indicating whether a pattern, \( P \) occurs in the text, \( T \); counting query: returns \( \eta_{\text{occ}}, (0 \leq \eta_{\text{occ}} \leq n) \), the number of occurrences of \( P \) in \( T \); and enumerative query: returns \( \eta_{\text{occ}} \) numbers, each indicating the starting position in \( T \), of an occurrence of \( P \).

Matches could be exact or inexact (approximate), independent of the specific type of query. Given a text string \( T \), and a pattern string \( P \), each with symbols from an alphabet \( \Sigma \), the exact string matching problem, is to check for the existence of a substring of the text that is an exact replica of the pattern string. Exact pattern matching is an old problem, and various algorithms have been proposed. Faro and Lecroq [111] provide a recent survey; see also the books [3, 96, 146, 317]. For inexact matching, some error may be allowed in the matches between the text and the pattern. In this case matches between two strings are determined based on a distance measure between them. The distance is traditionally calculated using the string edit distance (also called the Levenshtein distance). Given two strings \( T = t_1 \ldots t_n \), and \( P = p_1 \ldots p_m \), over an alphabet \( \Sigma \), and a set of allowed edit operations, the edit distance indicates the minimum number of edit operations required to transform one string into the other. Three basic types of edit operations are used: insertion of a symbol, \((\epsilon \rightarrow a)\); deletion of a symbol, \((a \rightarrow \epsilon)\); and substitution of one symbol with another \((a \rightarrow b)\); (\(\epsilon\) represents the zero-length empty symbol, and \(x \rightarrow y\) indicates that \(x\) is transformed into \(y\)). The edit operations could be assigned different costs, using suitable weighting functions. The edit distance is a generalization of the Hamming distance, which considers only strings of the same length and allows only substitution operations. In a sense, exact pattern matching is the special case where the edit distance between the substring of the text and the matching pattern is zero. Computing the edit distance usually involves dynamic programming, and requires an \(O(mn)\) computational time.

A variant of the inexact pattern matching problem is the \(k\)-difference problem, also called approximate string matching. The problem is to check if there exists a substring \( T_s \) of \( T \), such that the edit distance
between $T_s$ and $P$ is less than $k$. Another form of approximate matching, the *k-mismatch problem*, checks for a substring of $T$ having only a maximum of $k$ mismatches with $P$. That is, only the substitution operation is allowed. The parameter $k$ thus acts as a form of threshold to determine the correctness of a match. As with the exact matching problem, different algorithms have also been proposed for the case of approximate matching. Navarro [258] provides a comprehensive review. Other variants of the pattern matching problem have also been identified, usually for specific applications. Examples include

- pattern matching with swaps [206]: a transposition of two symbols (or symbol blocks) in one of the strings is treated specially by using different weights; a variation is circular pattern matching, whereby valid matches could include circular rotations of the pattern [143, 160, 220];
- pattern matching with fusion [10, 347]: consecutive symbols of the same character can be merged into one symbol, and one symbol can be split into different symbols of the same character;
- pattern matching with don’t cares [13, 146]: a more general form of pattern matching in which wild-card characters can be allowed in either the text, the pattern, or both. This is different from the usual approximate pattern matching in that now the don’t care characters can occupy fixed or arbitrary positions in the pattern (or text) as opposed to the usual case where we have the flexibility of alignment to obtain a maximal match;
- matching with scaling [20]: matching when a potential match is a scaled version of the pattern, similar to matching with fusion;
- matching with rotation [19, 119, 121]: matching when a potential match is a rotated version of the query pattern;
- multiple pattern matching [12, 18]: a generalization of the pattern matching problem, in which various patterns can be searched for in parallel. This is related to another variation called dictionary matching, whereby a pattern is to be
searched for simultaneously in a set of text sequences (the dictionary) [188, 190];

- super-pattern matching: finding a pattern of patterns [193];
- multidimensional pattern matching [136, 201]: matching when the text and pattern are multidimensional — typically used for images (2D pattern matching), or video (2D/3D pattern matching);
- parameterized pattern matching [38, 43, 44, 120]: when pattern matching is performed on parameterized strings (a generalized form of a string produced from two alphabets, the constant alphabet, and the parameter alphabet). The parameterized pattern matching (p-match) problem is to identify an equivalence between a pair of p-strings when (1) the individual constant symbols in the two strings match, and (2) there exists a bijection between the parameter symbols of the two strings.

2.1.1 Compressed Pattern Matching

A general problem in compressed pattern matching is that the format used to represent the compressed data is usually different from that of uncompressed data. More seriously, applying the same compression algorithm to two identical patterns that have different contexts could lead to completely different representations. That is, the same pattern located in two different text or image regions could result in different representations. Matching in such an environment will then have to consider the specific compression scheme used, and how the context could affect the compression. For lossy compression, the effect of the introduced error would also have to be considered, and once again, the error introduced could depend on the context. Rytter et al. [55, 133, 290] studied computational complexity issues involved in compressed pattern matching and for string embeddings.

2.1.2 Applications of Pattern Matching

Although pattern matching is sometimes pursued for its theoretical algorithmic significance, it also has applications in various real life
problems. Traditional areas where pattern matching has been used include simple spell checkers, comparing files and text segments, protein and DNA sequence alignments [146, 368], automatic speech recognition [267, 293], character recognition [69], handprint recognition [126], shape analysis [6, 347], and general computer vision [69, 347].

Other applications of pattern matching have been reported, including image and video compression [395], audio compression [14], video sequence analysis [10], searching in large shape databases [6], music sequence comparison [248], music retrieval [87, 90], and IP lookup in data communications and intrusion detection [62, 349]. A book about the BWT [3] listed several applications of the BWT, most of which are related to pattern matching.

2.2 Search Strategies for Text

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

Fast methods for string pattern matching is an area that has long been investigated, especially for exact pattern matching [60, 111, 174, 194]. The methods can be broadly grouped as either pre-indexing, pre-filtering, or a combination of the two. Pre-indexing (or preprocessing) usually involves the description of the database strings using a predefined index. The indices are typically generated by use of a hashing function or a scoring scheme [85, 256, 286]. Pre-filtering methods divide the matching problem into two stages: the filtering stage and the verification stage [85, 271, 378]. In the first stage, an initial filtering is performed to select candidate regions of the database sequence that are likely to be matches to the query sequence. In the second stage, a detailed analysis is made on only the selected regions, to verify if they are true matches.
The performance (in both efficiency and reliability of results) depends critically on the pre-filtering stage: if the filter is not effective in selecting only the text regions that are potentially similar to the pattern, the verification stage will end up comparing all parts of the text. Conversely, any region missed during the filtering stage can no longer be considered in the verification, and hence any false misses incurred at the first stage will be carried over to the final results. Pattern matching algorithms with sub-linear complexity have been reported \cite{85, 256, 258}. They generally combine both pre-indexing and pre-filtering methods. For \cite{85}, sub-linearity was defined in the sense of Boyer-Moore \cite{60}: on average, fewer than $n$ symbols are compared for a text of length $n$. That is, the matching time is in $O(n^p)$ for some $0 < p < 1$.

Fast algorithms have also been proposed for approximate string matching. Ukkonen \cite{355} suggested the use of a cut-off, which avoids calculating portions of a column if the entries in the edit distance table can be inferred to be more than the required $k$-distance. Galil and Park \cite{125} proposed a method based on the observation that the diagonal of the edit distance matrix is non-decreasing, and that adjacent entries along the row or columns differ by at most one (when equal weights of unity are used for each edit operation). Chang and Lawler \cite{85} proposed a column partitioning of the edit distance matrix, based on local longest exact matches. A review on approximate pattern matching algorithms is presented in \cite{258}.

In this section, we discuss methods for pattern matching in uncompressed text. The methods used for approximate pattern matching generally make use of techniques for exact pattern matching. Furthermore, the various proposed fast algorithms for exact pattern matching can be traced to one or more of the three basic fast algorithms — the Karp–Rabin (KR), Knuth–Morris–Pratt (KMP), and Boyer–Moore (BM) algorithms. All three algorithms use some form of pre-processing. BM and KR also used pre-indexing and verification. We discuss the three algorithms in more detail in this section. We then briefly introduce methods that use preprocessing on the text (i.e., some form of
indexing) to achieve faster searching on the text. The suffix tree and suffix array are key data structures for this type of searching.

2.2.1 Linear Search

Although a simple linear search is often regarded as the least efficient method for searching, it has some interesting variants that can perform surprisingly well. In particular, if the access pattern to the text is known, then more frequently accessed records can be placed nearer the front, and if the probability distribution of access is skewed this can result in very efficient searching. “Self-adjusting” lists [314] exploit this by using various heuristics to move items toward the front when they are used.

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

2.2.2 The Karp–Rabin Algorithm

The Karp–Rabin (KR) algorithm [174] is based on the concept of hashing, by considering the equivalence of two numbers modulo another number. Hashing is a method to map long sequences of potentially variable length to shorter sequences (typically numeric sequences) having a fixed length. For instance, an arbitrary-length string of characters can be hashed to a fixed-length short binary sequence. Given a pattern \( P \), the \( m \) consecutive symbols of \( P \) are viewed as a length-\( m \) \( d \)-ary number, say \( P_d \). Typically, \( d \) is the size of the alphabet, \( d = |\Sigma| \). Similarly, \( m \)-length segments of the text \( T \) are also converted into the same \( d \)-ary number representation. Suppose the numeric encoding of the \( i \)th such segment is \( T_d(i) \). Then we can conclude that the pattern occurs in the text if \( P_d = T_d(i) \), for some \( i \) — that is, if the encoding for the pattern is the same as that of a segment of the text.

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

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

The basic KR algorithm has been extended and has been used for 2-D pattern matching [58, 396] and more recently for parameterized pattern matching [43].

### 2.2.3 The Knuth–Morris–Pratt Algorithm

The KMP algorithm [194] simulates a pattern-matching automaton. It analyzes the pattern, for example, by considering how the pattern matches against shifts of itself, to determine which subsequent positions in the text can be skipped without missing possible matches.

The information is precomputed by use of a prefix function. In general, when the pattern is being matched against a text segment, it is possible that a prefix of the pattern will match a corresponding prefix of the text. Suppose we denote such prefix of the pattern as $P_p$. The prefix function determines which prefix of the pattern $P$ is a suffix of the matching prefix $P_p$. The prefix function is pre-computed from the $m$-length pattern in $O(m)$ time using an iterative enumeration of all the prefixes of $p_1p_2...p_m$ that are also suffixes of $p_1p_2...p_q$, for any $q, q = 1, 2, \ldots, m$. 
By observing that a certain prefix of the pattern has already matched a segment of the text, the algorithm uses the prefix function to determine which further symbol comparisons cannot result in a potential exact match for the pattern, and hence skips them. The average matching time is in $O(m + n)$.

The KMP algorithm is one of the more frequently cited pattern-matching algorithms. It has also been used for multidimensional pattern matching [39] and for compressed domain matching (see Table 5.7).

### 2.2.4 The Boyer–Moore Algorithm

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

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

Matching is performed by sliding the pattern over the text, and by comparing the characters right to left, starting with the last character in the pattern. When a mismatch is found, the mismatching character in the text is called the “bad character.” The part of the text that has so far matched some suffix of the pattern is called the “good suffix.” The bad-character heuristic proposes to move the pattern to the right by an amount that guarantees that the bad character in the text will match the rightmost occurrence of the bad character in the pattern. Therefore, if the bad character does not occur in the pattern, the pattern may be moved completely past the bad character in the text. The good-suffix heuristic proposes to move the pattern to the right by the minimum amount that guarantees that some pattern characters will match the good suffix characters previously found in the text. The BM algorithm then takes the larger of the two proposals.
It is possible that the bad-character heuristic might propose a negative shift (i.e., moving back to the already matched text area). However, the good-suffix heuristic always proposes a positive number, guaranteeing progress in the matching.

The preprocessing for the bad-character heuristic requires \(O(m + |\Sigma|)\) time, while the good-suffix heuristic requires \(O(m)\). The original BM algorithm has a worst case running time of \(O((n - m + 1)m + |\Sigma|)\). A modified version of the algorithm using the concept of memorization runs in \(O(n + m)\) time [27, 123]. The average running time is typically \(\leq O(n + m)\) using \(O(n)\) space. Overall, the BM algorithm generally produces better performance than the KMP and the KR algorithms for long patterns (large \(m\)), and relatively large alphabet sizes. See [95, 155] for further improvements on the BM algorithm.

### 2.2.5 Other Search Strategies

The \texttt{SHIFT-OR} and \texttt{SHIFT-AND} algorithms [35] are another family of algorithms that have been proposed to improve the efficiency of string pattern matching. These produce speed-ups by exploiting the parallelism in the bit level representation of the characters in the symbol alphabet. Navarro and Raffinot [260] proposed methods that combine suffix automata and bit-parallel algorithms. Bit-parallel methods for parameterized pattern matching are described in [120]. The bit-parallel algorithms have also been used in compressed pattern matching [181, 250, 261].

Various other algorithms have also been proposed for both exact and approximate pattern matching, most of them being a modification or combination of the above methods [111, 258]. Crochemore and Lecroq [97] provide a brief overview of pattern-matching methods, including the BM and KMP algorithms. They also discussed text compression, but not the relationship between compression and pattern matching.

The basic pattern matching algorithms have been extended to two dimensional pattern matching [58, 396]. Baker [39] applied string matching algorithms to character arrays. The basic algorithms also represent the primary building blocks for compressed domain pattern
2.2.6 Index-based Methods

In this work, our focus is on non-index based methods, that is, methods that perform little or no preprocessing on the text before pattern matching can begin. In such methods, if any preprocessing is involved, it will be performed on the pattern rather than the text. For the index-based methods, the required preprocessing is used to build indices or index data structures, such as suffix trees and suffix arrays, which can be used for later rapid pattern search and retrieval on the database. As might be expected, the preprocessing time is typically long, but this allows for a quick response time (usually in $O(m)$ time). This approach is valuable if a single text is to be searched many times, and thus extensive preparation will pay off. Suffix trees are discussed in detail in [146], while suffix arrays are discussed in [3, 282]. A detailed treatment on compressed full-text indexing can be found in [110, 115]. We do not include index-based pattern matching, whether for compressed or uncompressed text, in this survey.

2.3 Search Strategies for Images

Recognizing patterns in static or moving images is important for many applications, including tracking moving objects, character recognition, and face recognition. When exact matching is needed, it is possible to define simple functions or statistics on the image, based on what sort of matching is to be performed. This could be useful when the results of image matching is to be used by a machine — for instance for further processing (such as in robot navigation in computer vision).

In practice, however, the result of image matching is often used by humans, which means that human subjectivity will have to be considered. In addition, images typically use lossy representation. Image matching is therefore typically based on similarity rather than exact matches. Issues such as distance or similarity measures between images thus become important. The measures used are generally similar to the distortion measures used to determine the quantitative fidelity for
compressed images (see Section 4). Other problems include the large amount of data often involved, and the huge computation that is generally needed. Vasconcelos [357, 358] proposed probabilistic similarity functions based on the minimum probability of error as a basis for comparing images for retrieval. In [360] a multi-resolution manifold distance was proposed for computing image similarity. The motivation was the observation that images span manifolds in high dimensional spaces, and thus an appropriate similarity measure between images should capture the distance between the manifolds, rather than the general practice of focusing on distances between other properties or features extracted from the images. They then proposed the similarity (distance) measure between two image manifolds as the Euclidean distance between their closest points. Computing all the possible manifolds for a given image is obviously a time consuming process. Methods for efficient evaluation of image similarity functions have been described in [358]. An occupancy model for fast image similarity evaluation was described earlier in [8]. Surveys on general image retrieval can be found in [100, 315].

The image similarity-matching problem is similar to the pattern-matching problem. Given an image database \( D_I \) and a target query image \( Q \), the similarity matching problem is to report all images in \( D_I \) that are similar to \( Q \) (i.e., all images with a similarity distance not exceeding a given threshold). The matching is on a whole-image basis and the retrieved images are then ranked according to their similarity to the target image. When the threshold is zero we have exact matching. There is also a variant of this problem, object-level image search: given a target image \( Q \), search within each image in \( D_I \) and report all subimages that match \( Q \). In this case, the searching requires a more careful description of object shapes or contours, and also requires matching within each image in the database. This distinction is often important, as it affects the time required for the match, the method to be used in the search, and the quality of the results. Below, we briefly describe the general methods that are used in image matching.

### 2.3.1 Template Matching

Template matching [142, 281] is the simplest approach to image matching. It does a point by point comparison of the pixel positions in
the image, using the target image as a template. After choosing a suitable starting point, corresponding positions on the template and the database image are compared point by point. The overall difference is then added to determine the match distance between the images.

Choosing a starting point requires solving the difficult problem of establishing a correspondence between a point in the target image and the same (or similar) point in the database image. Because of this difficulty, template matching often involves an exhaustive consideration of each pixel position in the target image as a candidate starting point. The match distance is then chosen as the minimum obtained over all the starting points. To reduce the time required for template matching, some methods partition the images into subimages, and the comparison is performed block by block, rather than point by point. This block-based approach has a direct application for searching on compressed images, especially for transform-coded images. For approximate matching, block-based methods can use image statistics, such as average color or the variance of the pixel values, to estimate the similarity between images.

### 2.3.2 Feature-based Image Matching

Unlike template matching, where images are matched by checking for correspondence between pixel locations or key points in the image, in the feature-based approach important features in the image are extracted based on which different images (or subimages) are compared for possible matches. The features (also called descriptors) are usually based on shape, color, texture, or spatial information in the image. Under this approach images are compared for similarity by using the features to construct a multidimensional feature space, whereby a given image will be a point in the feature space. Distance or similarity functions are then applied to the points in the multidimensional space to determine images that are similar. The feature-based approach is the primary method used in image retrieval [100, 278, 315]. Shape matching methods are reviewed in [336, 362], while texture analysis methods are reviewed in [104, 172]. Berretti et al. [56, 57] studied methods that perform image retrieval by exploiting the spatial relations between objects in the image or scene.
To further reduce the time required, some methods try to detect important areas or points in the image, such as corners or interest points, and use these to speed up the matching. For instance, this could be done by matching only at the interest points or regions, or by using them to prune the search for candidate matches. Various descriptors based on interest points have been proposed. Popular methods that have been used to generate interest points or descriptors for image matching include Shift-Invariant Feature Transform (SIFT) [226], Rotational-Invariant Feature Transform (RIFT) [204], Speeded-Up Robust Features (SURF) [42], and Gradient Location and Orientation Histogram (GLOH) [241]. Other feature-based methods for image comparison make use of specific image descriptors, such as shape contexts [52], and local texture descriptors [204]. A performance review on image descriptors can be found in [241].

2.3.3 Other Search Strategies for Images

Conceptually, methods for 2D string pattern matching could be applied to the problem of image matching. For instance, methods for 2D approximate pattern matching can be well suited for comparing images. However, often the huge alphabet size involved can be a problem. For images, the alphabet size is typically the number of color levels in the image (which might be 256 for a lower quality image, or even millions of color levels for high quality images).

Other techniques for transform domain image matching have also been explored. The basic idea is to define some functions using the transform coefficients, based on which two images (or image subparts) can be compared for approximate matching. The matching is often in terms of the shape, texture, or general intensity characteristics in the images. Generally approximate image matching is performed as a basic process in image retrieval. Methods for feature-based image retrieval have been reported for DCT-based compression [268, 369], vector quantization (VQ) based methods [159], fractal-coding [391], and wavelet-based compression [235]. A comprehensive survey on image retrieval can be found in [100, 315]. General compressed domain analysis for both images and video are discussed in the book [254].
As was seen in the previous sections, there is a strong relationship between searching and compression. This connection can be explored by considering the issues of pattern matching during compression, compression as pattern matching, and searching using compressed structures.

3.1 Pattern Matching for Compression

Some authors have considered compression as basically a problem of pattern matching [14, 380] or pattern recognition [176]. More generally, most compression methods require some sort of searching:

- The Lempel–Ziv family of methods search the previously coded text for matches [51, 371, 397, 398].
- PPM (prediction by partial matching) methods search for previous occurrences of a context, typically using a trie data structure, to predict what will happen in the current one [88, 89].
- DMC (dynamic Markov compression) uses a finite state machine to establish a context that turns out to have a
3.1 Pattern Matching for Compression

- BWT [49] does not perform direct searching, however, it performs an initial lexicographic sorting as part of the compression process. The resulting sorted contexts for symbols in the sequence provide an excellent opportunity for later search on the compressed data. The BWT is also strongly connected with important search data structures such as suffix trees and suffix arrays [3], and hence can be exploited for rapid pattern matching.

- PPAM (prediction by partial approximate matches) [394] performs lossless image compression using inexact or approximate contexts. Efficient and effective techniques for quickly locating matching similar contexts is a key issue in using the approach.

- Inexact pattern matching is intimately related to lossy compression (i.e., source coding with a fidelity criterion). In lossy compression, the goal is often to minimize the average distortion between the original signal and reconstructed signal, or in some cases, to minimize the maximum distortion between them. The amount of searching performed has a direct bearing on the compression performance. For instance, vector quantization must search the codebook for the nearest match to the input vector or pattern being coded. The performance in terms of bit rate, distortion introduced, and encoding time is dependent on the overall amount of searching involved. Hence, there has been various proposals for intelligent search data structures and algorithms for vector quantization [283]. Further, in the vector quantization community, there is significant interest in designing codes such that small/large squared error distortion in the reproduction codewords correspond to small/large Hamming distances between the binary indices (channel codewords) that correspond to the reproduction codewords. See for example “pseudo-Gray codes” [390] or tree-structured vector quantization [170]. Thus these codes
can be used in lossy compression schemes where matching binary codewords correspond to similar (or an approximate matching of the) decompressed signals.

- In Section 2.1, the Hamming distance was introduced for inexact matching for text strings. This is related to the concept of rate-distortion coding with an $\ell_\infty$ distortion used in lossy image compression, which results from applying a constraint on the maximum allowed Hamming distance (see [279]).

- PMIC (pattern matching image compression) [14, 34] used in lossy image and video compression has to search for approximate repetitions on a prefix of the uncompressed image in the compressed part of the image. Here, matches are defined only in an approximate sense, based on a specific distortion criterion.

- MPEG requires searching as part of its motion estimation and motion compensation, which are key aspects of the MPEG standard, as they affect both the compression ratio and compression time. Motion estimation requires a fast method to determine the motion vectors, and always involves searching for the matching blocks within a spatio-temporal neighborhood. While the quality of the compression improves with a wider search area, the compression time increases. The same applies to H.26X video compression series, which also depends on motion estimation and motion compensation.

In [176, 177], lossless data compression was viewed as a pattern recognition problem. They generated a codebook using the LZW algorithm, and then used it to compress the text by applying rule-based logic from pattern recognition. Clearly, this idea has long been used in lossy data compression, such as vector quantization, and dates back to Shannon’s earlier work on lossy coding [303, 304]. Explicit considerations on the data structures used in searching as a way of improving the compression performance have been made in [46, 92, 330].

The relationship between pattern matching and compression for images have been studied in [14, 34]. More theoretical studies on data
3.2 Compression for Pattern Matching

As discussed above, pattern matching is a fundamental operation for performing compression, but it is also possible to use compression to perform pattern matching or pattern recognition. For example, in [164] a compression method was used as the basis for searching, and the results of the search were in turn used as matches for a textual image compression system. The amount of compression achieved is used to
determine whether the target data fits the compression model (which was trained on a template or another object of interest). This is similar to the work in [231], where bar codes in an image are recognized because of the way they compress. However, in the case of the bar codes, the pattern being recognized just happens to induce observable behavior in a general purpose compressor.

In another study [169], the matches used by an LZ coder were used to infer which class an object in the image belongs to. The method was then used for hand-written character recognition, and recognition of graphic patterns in printed text. More recently, Heidemann and Ritter [148] viewed data compression as the basis for pattern recognition, and showed how standard compression schemes such as gzip and bzip2 can be applied to solve complex recognition tasks, such as visual object recognition and texture classification, without any data preprocessing, feature extraction, or special modification of the compression algorithms. Otu and Sayood [270] showed how the phylogenetic tree between various biological species can be constructed using distance measures computed from the LZ-decomposition of their respective protein sequences. The recent review [257] provides a more detailed treatment of the use of data compression in pattern recognition and discovery in biological sequences.

*Cross-entropy* [288] is a generalization of this approach, where the probability distribution for one pattern is used to generate a coding scheme that is applied to other patterns; the pattern that the coding scheme compresses into the minimum number of bits per symbol can be considered to be the best match.
To discuss performance issues in compressed pattern matching, it is appropriate to consider performance from the viewpoint of the two main aspects of the problem, namely, compression and pattern matching.

### 4.1 Performance Measures for Compression Algorithms

Compression algorithms are usually evaluated in terms of the amount of compression they can achieve, coding complexity, decoding complexity, and the extra space required during the compression. The amount of compression measures how good the compression algorithm is in reducing the size of the given input data. There are several measures that can be used. In lossless compression it is common to measure the number of bits per input sample (e.g., bits per character, or bits per pixel), although it is also sometimes measured as a percentage (which can be ambiguous as it may be the amount of data removed, or the amount left). In lossy compression the amount of compression is less meaningful since very high compression ratios can be obtained by allowing low quality, and thus a measure of the quality of the data after compression can be more important.
Depending on the type of modeling or transformation used by the algorithm, the compression ratio may not necessarily vary linearly with the original data size. For instance, with simple character-by-character Huffman coding, the size of the vocabulary grows slower than the file size, since after some point, most of the codewords (characters) that are encountered would have appeared earlier in the text. Thus the amount of compression can improve with the file size. See Figure 2 in [250]. On the other hand, with the Lempel–Ziv methods that stop updating the dictionary after it is full (such as the LZW-based Compress), or those that use a fixed window for matching (such as the LZ77-based Gzip), the compression ratio is practically independent of the original data size, since the size of the dictionary will be constant after a while. If there isn’t a restriction on the window size, the compression will improve with data size, but at the expense of higher coding complexity.

The coding and decoding complexity describe the amount of computation required at the coding or decoding stage, as the case may be. In practice, these relate to the speed of compression and decompression respectively. Typically, more coding complexity will be due to more extensive search during the compression (for instance, larger search windows for LZ compression), or other kinds of computation-intensive operations (such as lexicographic sort in BWT). More computation and hence higher coding complexity usually leads to better compression ratios.

Some algorithms provide symmetric complexity, in that both the coding and decoding complexity are the same. This is often important in on-line algorithms, such as those used in streaming network communications, where the current data stream must be processed (compressed or decompressed) before the next data stream arrives. For other applications, for example images in a web page, where data could be compressed just once, but might be decompressed many times, the symmetry may not be required and off-line algorithms may be used. Here lower decompression times might be more beneficial, even at the expense of a higher compression time.

The extra space criterion [16] indicates the space-complexity of the algorithm. It shows how much memory the algorithm will require to store temporary data during compression. For instance, this may be
used to store the Huffman tables, dictionaries, or statistics about the data, and will typically increase with increasing data size. The block size for BWT and the size of the priming text in PPMD are examples where the space complexity can be chosen, and providing more space can give better compression [16].

More recently, with the increasing need to access data in its compressed form without decompression, the capability of random access to the compressed data and the ability to support search have emerged as new criteria for evaluating compression techniques. Table 4.1 (expanded from [36]) shows a comparison of these factors for general lossless (text) compression methods.

Lossless image compression methods are evaluated similarly based on their compression ratio, computational time, and memory space. For lossy compression, an important extra performance measure is the quality of the reconstructed data. The quality is measured both qualitatively and quantitatively. Qualitatively, subjective perceptual criteria are used, for instance, by empirically comparing the reconstructed data.

Table 4.1. Comparison of text compression methods.

<table>
<thead>
<tr>
<th>Performance measure</th>
<th>AC</th>
<th>HC</th>
<th>HC (word)</th>
<th>LZ</th>
<th>PPM</th>
<th>BWT</th>
<th>DMC</th>
<th>TC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compression ratio</td>
<td>good</td>
<td>poor</td>
<td>good</td>
<td>good</td>
<td>very good</td>
<td>good</td>
<td>good</td>
<td>good</td>
</tr>
<tr>
<td>Compression speed</td>
<td>slow</td>
<td>fast</td>
<td>fast</td>
<td>very fast</td>
<td>very fast</td>
<td>very fast</td>
<td>very slow</td>
<td>slow</td>
</tr>
<tr>
<td>Decompression speed</td>
<td>slow</td>
<td>fast</td>
<td>very fast</td>
<td>very fast</td>
<td>very fast</td>
<td>very fast</td>
<td>very fast</td>
<td>very fast</td>
</tr>
<tr>
<td>Memory space</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>medium</td>
<td>high</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Compressed pat. matching</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Random access</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

Expanded from [36], p. 187. Note: for compression ratio (c.r.), the grouping is as follows: very good: c.r. < 30%; good: 30% < c.r. < 45%; poor: c.r. > 45%. The following abbreviations are used: AC — arithmetic coding, HC — character based Huffman codes, HC(word) — word based Huffman codes, TC — Tunstall codes.
with the original data using a human observer [165]. Quantitative performance measures are based on different fidelity criteria, such as the distortion between the original and reconstructed data. Two common measures of distortion often used for images and video are the mean square error (MSE) and the peak signal-to-noise ratio (PSNR).

For two images $I_1$ and $I_2$ each of dimensions $M \times N$, the MSE and PSNR are related, and are defined as follows:

$$\text{MSE} = \frac{1}{MN} \sum_{x=1}^{M} \sum_{y=1}^{N} [I_1(x,y) - I_2(x,y)]^2$$  \hspace{1cm} (4.1)$$

$$\text{PSNR} = 10 \log_{10} \frac{(2^b - 1)^2}{\text{MSE}},$$  \hspace{1cm} (4.2)

where $b$ is the number of bits used to represent each pixel without compression. Thus, $2^b - 1$ will be the maximum gray-scale value. In general, for lossy compression (as used for images and video) higher compression ratios often result in lower compression quality. Thus, the performance is often indicated as a pair of values, indicating both the MSE or PSNR and the compression ratio (or bit rate) at which the given distortion or compression quality was obtained.

The interesting question of the connection between rate distortion, bit rate, retrieval quality, and retrieval speed was studied in [350] for general content-based retrieval, in [163] for similarity queries, and in [197] for facial image databases.

### 4.2 Performance Measures for Pattern Matching Algorithms

For traditional (text) pattern matching, the major performance measure is the complexity of the matching algorithm — in terms of both time and space. In general, with $n$ as the text size and $m$ as the pattern size, the worst case complexity is in $O(nm)$, while some linear-time algorithms that run in $O(n + m)$ are available. Algorithms with sub-linear time complexity have also been reported [85, 256]. In practice, the complexity directly affects the system response time to user queries.

With image retrieval, the pattern matching is necessarily approximate. However, the approximation required here cannot easily be
4.3 Performance Measures for Compressed Pattern Matching

described in terms of the usual $k$-distance or $k$-mismatches. Other measures of effectiveness, such as precision, recall and ranking are therefore needed. The precision shows how good the algorithm is in retrieving only the correct (or similar) matches by giving the proportion of matches returned that are relevant, while recall indicates how well the algorithm can retrieve all correct matches by giving the proportion of relevant matches that are retrieved. Once again, for certain data types, such as images, what is correct or similar could be quite subjective, and will depend on the application [100, 315]. For conventional exact pattern matching algorithms on text, perfect precision and recall are usual. However, under compressed pattern matching, the context of the pattern in the text could lead to possible mis-detection of the pattern in the text. Precision and recall could therefore be relevant in compressed pattern matching for both lossy and lossless compression.

4.3 Performance Measures for Compressed Pattern Matching

Based on the considerations above, we can enumerate key considerations in measuring the performance of compressed domain pattern matching algorithms as follows:

- **Time complexity and speed**: The time complexity is related to the speed of the algorithm and shows how fast the algorithm could be, at least, in theory. The complexity here is only the theoretical complexity, but should reflect the speed of the algorithm in practice.
- **Space complexity**: This indicates the amount of extra storage space (primary or secondary) that is needed temporarily during processing. It can have a bearing on the amount of resources the algorithm will require, especially for huge-volume applications, when the text or even the pattern is quite large. Table 5.7 (Section 5) gives the theoretical performance (in terms of time and space complexity) of various proposed text-based compressed-pattern matching algorithms. Table 6.1 shows the corresponding performance
figures for the few algorithms for pattern matching on lossless compressed images.

- **Optimality:** Let $T_c$ be the compressed version of the text $T$, and let $P_c$ be the compressed version of the pattern, $P$. Denote $n = |T|, m = |P|, n_c = |T_c|, m_c = |P_c|$. We assume that $n_c \leq n$, and $m_c \leq m$. In [15, 16, 17], the concept of optimality was introduced to the problem of compressed domain pattern matching. They classify a compressed pattern matching algorithm as follows:

\begin{enumerate}
  \item \textit{efficient}: if the time required is in $o(n)$
  \item \textit{almost-optimal}: if the time complexity is in $O(n_c \text{ polylog } m + m)$
  \item \textit{optimal}: if the time complexity is in $O(n_c + m)$
\end{enumerate}

where polylog$(x) = (\log x)^k$ for some $k \geq 1$. For a fully compressed pattern matching algorithm, the classification is as follows:

\begin{enumerate}
  \item \textit{efficient}: if the time required is in $o(n)$
  \item \textit{almost-optimal}: if the time complexity is in $O((n_c + m_c) \text{ polylog } (n_c + m_c))$
  \item \textit{optimal}: if the time complexity is in $O(n_c + m_c)$
\end{enumerate}

Generally, compressed-domain searching is regarded as optimal if the factor by which the search is sped up is no less than the factor by which the text has been compressed (i.e., the inverse of the compression ratio). Schmalz [300] provides a discussion on how the speedup can be measured. One question that arises is whether the speedup can be better than the compression ratio.

- **Comparison with decompress-then-search:** Another way the performance can be measured is by considering how fast the compressed pattern matching is compared with performing the same searching operation on the uncompressed data using existing text matching algorithms — the so-called decompress-then-search technique. In [259] it was argued that, in practice, compressed pattern matching cannot be
faster than uncompressed pattern matching for some compression methods, such as LZ77. A stricter criteria would be to remove the decompression time, and compare how fast the compressed-pattern matching algorithm could be when compared with the fastest traditional (i.e., uncompressed) pattern matching algorithm. If we can achieve fast searching by using compressed domain methods for any compression scheme, one can then take advantage of this to compress the data with the primary objective of improving search time. Shibata et al. [308] suggest using compression for this type of purpose. These issues are related to the issue of optimality. In general, since a compressed pattern matching algorithm will typically consider a smaller amount of data, using an optimal compressed pattern-matching algorithm should be faster than doing the same search on uncompressed data. But this may not always be the case in practice.

- **Precision and recall**: This will be important for searching on data that is compressed using lossy-compression methods. Since the retrieved results may no longer be exact matches to the original pattern, we will require a way of knowing the effectiveness of the matching. Taken together, precision and recall can provide us with an objective way to evaluate a search algorithm. We note that this is not a problem in traditional $k$-approximate pattern matching used in text retrieval, because the effectiveness of the match is implicitly defined by the distance parameter $k$. With compressed pattern matching, however, depending on the compression method and the context of a given pattern in the text, it is possible that a search algorithm could miss out an occurrence of a query pattern in the text — even for exact matching. Therefore, precision and recall will also be relevant in considering compressed pattern matching for text.

- **Ranking**: For lossy compression, this is one other way of evaluating the effectiveness of compressed domain pattern matching. The precision and recall usually do not provide adequate information on the effectiveness of the retrieval,
especially for perceptual data, such as multimedia data (images, video and audio). The ranking measure shows how well the ranking of the results produced by the algorithm compares with those produced by a non-compressed domain matching algorithm or with the ranking produced by a human observer. Ranking is typically performed using a distortion measure between the query input and the retrieved samples (or using some other similarity measures, based for example on the image features or descriptors).

- Others: For particular applications there may be other ways to evaluate the performance. For instance, we may be interested in how the method performs in related activities, such as data mining and discovery, which involves a series of pattern matching operations.
Text compression is generally lossless, but results in only a modest reduction in the data size. The general approach is to exploit different forms of redundancy that may occur in the text, such as repetitions, correlations, and so on. The methods can be grouped into four general classes: dictionary methods (also called pattern substitution), statistical methods, variable to fixed length codes, and methods based on sorted contexts. In this section we will consider methods from these different classes, and ways that they have been adapted to allow compressed pattern matching on text.

The compressed domain pattern matching problem is to find the occurrence(s) of a given pattern in the compressed text without decompression. Lossless compression algorithms can typically compress files to within the range of 50% to 20% of the original uncompressed file size, depending on which algorithm is being used and the type of text. We have already pointed out that the relative size of the compressed file will usually decrease with increasing input file size (see Figure ??). In practice, for large text files, the size of the compressed file is almost linearly proportional to the original size. Thus, a linear pattern matching algorithm on the compressed file with a large proportionality
constant could actually perform worse than searching on the original file [181, 259].

We have noted earlier that pattern matching can be regarded as the basis of compression. The process can be reversed, in the sense that the compression algorithms can be designed to aid pattern search. For example, Manber [233] proposed such an algorithm for a class of files that are searched often such as catalogs, bibliographic files, and address books. The basic idea was to substitute common bigrams of characters with special symbols that can still be encoded in one byte. The scheme allows the basic pattern matching algorithms to be used for fast pattern search, although the reduction of the file size was only to about 70% of the original size. Shibata et al. [309] adopted a similar approach in byte-pair encoding (BPE), and used a faster Boyer–Moore algorithm for pattern search.

5.1 Dictionary Methods

Currently the most widely used lossless compression methods are based on Lempel–Ziv (LZ) coding, which represents a variety of methods from the general class known as dictionary coders. A dictionary method simply maintains some dictionary (also known as a codebook or lexicon), and data is compressed by replacing strings in the text with a reference to the string in the dictionary. The many variants of dictionary coding arise from different choices for how the dictionary is constructed, the algorithm used to search the dictionary, the rules for which substrings in the text are replaced with a reference, and what bit pattern will be used to represent the reference.

Static dictionary coders work with a pre-constructed dictionary. The problem of constructing an optimal dictionary is NP-complete [324], although suitable heuristics are available. However, a more elegant solution to the problem is to construct the dictionary adaptively, and this is the main idea in Lempel–Ziv coding.

There are two main variants of Lempel–Ziv coding, those based on a 1977 paper [397] and those based on a 1978 paper [398], sometimes called LZ77 (or LZ1) and LZ78 (or LZ2) respectively. LZ77 methods use the simple idea that text prior to the current coding point is the
5.1 Dictionary Methods

dictionary, so a greedy search is performed of recent text to see if the next few characters have occurred before, and they are then replaced with a reference that defines how far back the match is, and how long it is. LZ78 methods break the prior text up into phrases which are stored in a dictionary and are typically referred to by a sequence number.

Lempel–Ziv coding often involves some form of factorization on the input text and the use of a suffix tree. Here, factorization refers to identifying repeating substrings in an input string. With respect to the block diagram of Figure 1.1, the transformation stage could involve the initial construction of the suffix trees or the factorization of the input text to expose the repetitions in the text. The coding stage essentially involves the replacement of a factor that has appeared previously in the text with a pointer to where it appeared.

An LZ77 decoder is very fast; it maintains an array of recent text, and simply looks up the reference in the array and copies the characters. The encoder tends to be slower, as the previous text (usually just the last few thousand characters) must be searched for a match. This is usually done using a hash table to index the text, or sometimes a binary tree, linked list, or lookup table. Normally LZ77 coders use a greedy algorithm and match the longest string that they find. The searching for encoding could be used to save work in compressed-domain matching by forcing the units being matched to be of relevance to later pattern searches. For example, the matches could be forced to start or finish on word boundaries to simplify word-based searches. This only affects the encoding algorithm; the same decoder could be used without modification, which has the nice property that such files could still be viewed and processed without any changes to the viewing software.

The LZ77 approach has been used by a number of popular compression methods, including the ZIP methods (a general purpose compression system used in products like PKZIP, WINZIP, and GZIP), and the system has been adapted for an image graphics standard called PNG (portable network graphics). The better LZ77 methods use a form of Huffman coding (see Section 5.5) to encode the pointers in their output.
The other branch of the Lempel–Ziv family is the LZ78 type methods. These methods also base the dictionary on previously coded text, but the text is broken up into substrings according to a simple heuristic (usually by concatenating one character to a previously coded substring). This limits the range of strings available for substitution, but makes encoding simpler. LZ78 methods are waning in popularity because the LZ77 methods are now both faster and give better compression, and also there have been patent issues surrounding the use of some LZ78 variants. However, it may be possible to adapt LZ78 methods for compressed-domain searching again by restricting pattern matches during compression to correspond to the kind of match that will be required later. Also, the dictionary contains a lot of implicit information about patterns, and pointers to similar patterns, which could be exploited.

One of the best known variants of the LZ78 family is the LZW method [371], which developed into a popular public domain system called COMPRESS, which for many years was the de-facto standard for lossless compression. The LZW method is also the basis of the compression component of the GIF (Graphics Interchange Format) image compression standard, which is widely used on the web, amongst other places. While variants of the general LZ methods (i.e., LZ77 and LZ78) can allow exponential decrease in the size of the compressed text for highly repetitive text, for LZW we have that \( n_c \geq \sqrt{n} \), where \( n \) is the original data size, and \( n_c \) is the compressed size. For compressed pattern matching, however, LZW algorithm, is much easier to work with, compared to using the original LZ77 or LZ78 algorithms.

There are many other proposed variants of the LZ78 family, such as the LZMW [242], the LZAP [322], RAY and XRAY [74, 75] methods. (See [294] for more variants of the LZ77 and LZ78 compression family). A hybrid approach between the LZ77 and LZ78 which has specific properties particularly tuned for searching on the compressed data has also been proposed [259, 261]. The variants differ mainly in how they factor the input text, how they reference the previous occurrence of the text segments, or how they limit the amount of memory that may be taken up by the dictionary.
5.2 Searching on LZ-encoded Text

The challenge in compressed pattern matching for the LZ family is the adaptive nature of the dictionary used. This means that a given symbol in the input string could have a different encoding, depending on when it is coded or the symbols around it. This is different from methods such as static Huffman codes, where the same symbol is always coded with the same codeword. The general approach follows the standard two-step approach used in traditional pattern matching, namely pattern preprocessing, and text scanning using results from the preprocessing. The pattern preprocessing often uses methods similar to those for searching on uncompressed text, such as the KMP, BM, or KR methods. The scanning stage usually involves three major steps:

1. Reconstructing the dictionary from the LZ-encoded string;
2. Identifying pattern substrings, including pattern prefixes and pattern suffixes in the dictionary entries; and
3. Verifying possible pattern occurrences within the results from step (2), or using their combination.

Methods have been proposed for compressed pattern matching for different variants of the LZ family, including LZ77, LZ78, and LZW. Below, we describe the original methods that were proposed for searching on text compressed with LZW and LZ77, respectively. Most other methods extended these original methods or used similar ideas to provide improved performance.

5.2.1 Searching on LZW-compressed Text

The LZW variant of LZ78 performs compression of the input text by simply encoding the identification number of entries in the dictionary. The dictionary is initialized to contain the alphabet, and then additional strings are added based on the patterns found in the text as it is encoded. For a text string \( T = t_1t_2\ldots t_n \) with symbols from the alphabet \( \Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|} \} \), the LZW-compressed version \( T_c \) of \( T \) is a string of integers \( T_c[1]T_c[2]\ldots T_c[n_c] \), where the numbers are in the range \( 1 \leq T_c[i] \leq n_c + |\Sigma| \).
Amir et al. [16] were the first to propose a method for compressed pattern matching on LZW-compressed files. They reported the first occurrence of a pattern \( P = p_1p_2\ldots p_m \) in a compressed text \( T_c \). Here, we use an example to explain how the algorithm works. Our discussion in this section is essentially taken from [338], where the method was extended to report all occurrences, and also to support multiple pattern matching.

To understand Amir et al.’s algorithm for searching on LZW-compressed strings, we first consider how the encoded strings are generated. We use the sample text \( T = \text{abaababaabab} \), with \( \Sigma = \{a, b\} \). Table 5.1 shows the steps involved, including the dictionary entries (sometimes called phrases or chunks) generated at each step. In practice, the LZW encoding is performed using a tree-like search data structure called a *dictionary trie*. This stores the dictionary entries generated as the algorithm processes the text. Each node in the trie corresponds to a dictionary entry, and is described by three parameters: (1) a *node number*: a unique ID in the range \([0, n + |\Sigma|]\); (2) a *label*: a symbol from the symbol alphabet \( \Sigma \); and (3) a *chunk*: the string that the node represents, i.e., the string of labels spelt out by following the path from the root to the node.

### Table 5.1. Sample LZW encoding using the text \( T = \text{abaababaabab} \).

| Step | Phrase ID | Phrase added | Output code | \( S_i \) | \( E_i \) | \( |PP_i| \) | \( |SS_i| \) | \( P=bab \) | \( P=aba \) |
|------|-----------|--------------|-------------|--------|--------|----------|----------|----------|----------|
| 1    | 3         | ab           | 1 1 1 1 2   | 1      | 2      | 2*       | 1        |
| 2    | 4         | ba           | 2 2 2 1 1   | 2*     | 1      | 2*       | 1        |
| 3    | 5         | aa           | 1 3 3      | 1      | 1      | 1*       | 1        |
| 4    | 6         | aba          | 3 4 5      | 2*     | 2      | 3*       | 3*       |
| 5    | 7         | abaa         | 6 6 8      | 2*     | 2      | 1*       | 3*       |
| 6    | 8         | abaab        | 7 9 12 1   | 2      | 2      | 3*       | 3*       |
| 7    | 9         | b$           | 2 13 13 1  | 1      |        |          |          |

The first few columns show the phrase and output code generated at each coding step. The table also shows the prefix numbers (\(|PP_i|\), the length of longest prefix chunks) and suffix numbers (\(|SS_i|\), the length of the longest suffix chunks) identified at different steps in the LZW encoding, using two patterns, \( P = \text{bab} \) and \( P = \text{aba} \). Bold and asterisk show positions where there is a pattern occurrence in \( PP_i \), \( SS_i \), \( PP_i \cdot SS_i \), or \( PP_i \cdot SS_{i+1} \). Note that concatenation here removes possible overlapping positions in \( PP_i \) and \( SS_i \). We use \( S_i \) and \( E_i \) to denote the respective starting and ending positions in \( T \) that is encoded at step \( i \).
5.2 Searching on LZ-encoded Text

Figure 5.1 shows the corresponding LZW encoding trie for the sample input $T = \text{abaababaabaab}$. Amir et al.'s algorithm performs the pattern matching directly on the trie. To facilitate the pattern matching, the following terms related to a node in the trie are defined with respect to the pattern. A chunk is a prefix chunk if it ends with a nonempty 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 nonempty 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 substring of the pattern, i.e., the chunk is $p_i \ldots p_j$ for $i > 1$ and $j \leq m$. If $j = m$, the internal chunk is also a suffix chunk.

In this work, we use $X \cdot Y$ to denote the concatenation of strings $X$ and $Y$. For a given chunk (or node), at step $i$, we use $PP_i$ and $SS_i$ to denote the representative prefix and representative suffix, respectively. We use the terms prefix number and suffix number to refer to their respective lengths. These quantities are computed for a given node as the node is added to the trie. Using the example text, Table 5.1 also shows the prefix number ($|PP_i|$) and suffix number ($|SS_i|$) computed at each step of compressed pattern matching, for two sample patterns. At a given scanning step $i, i = 1, 2, \ldots, n_c$, we can observe how the $PP_i$ and $SS_i$ combine to determine possible matches with the pattern. For instance, we only need to check for possible matches around the text regions where one of the following conditions holds: $|PP_i| = m,$
$|SS_i| = m$, $|PP_i| + |SS_i| \geq m$, or $|PP_i| + |SS_{i+1}| \geq m$. Essentially, this is when $|PP_i \cdot SS_i| \geq m$, or $|PP_i \cdot SS_{i+1}| \geq m$. Other constraints can be obtained based on the internal chunks. Thus, after obtaining $PP_i, SS_i$, and the internal chunks at a given step, compressed pattern matching only needs to be performed at steps where any of the conditions holds.

The method makes use of three key functions which are computed as part of preprocessing for the pattern. Given a pattern prefix $S_1$, and an internal substring $S_2$, function $Q_1(S_1, S_2)$ returns the length of the longest pattern prefix that is a suffix of $S_1 \cdot S_2$. For a pattern prefix $S_1$ and a pattern suffix $S_2$, function $Q_2(S_1, S_2)$ checks if the pattern occurs in $S_1 \cdot S_2$, and returns the smallest index of $S_1 \cdot S_2$ where there is an occurrence. It returns a zero otherwise. The last function $Q_3(S_1, \sigma)$, takes an internal substring $S_1 = p_i \ldots p_j$, and a symbol $\sigma \in \Sigma$, and returns the index pair $<i,j>$ if $S_1\sigma = p_i \ldots p_j$ is an internal substring, or $<0,0>$ otherwise.

Pattern matching is then performed as follows using the general two-phase procedure of preprocessing and scanning [16, 338]:

**Preprocessing:** This basically computes the functions $Q_1()$, $Q_2()$, and $Q_3()$. $Q_1()$ can be performed based on standard KMP preprocessing as used in pattern matching on uncompressed texts.

**Text scanning:** Initialize trie and set global variable $Prefix = NULL$.

For $i = 2$ to $n$, obtain $T_c[i]$, the code for the current node. Then, perform the following steps:

- **Step 1.** Dictionary construction and update:
  Add a new node in the trie and compute its prefix number, suffix number, and internal range.

- **Step 2.** Pattern matching:
  1. If $Prefix = NULL$, set $Prefix$ as current node’s representative prefix.
  2. If $Prefix \neq NULL$ and the current node is a suffix node, check the pattern occurrence from $Prefix$ and the current node’s representing suffix $SS_i$; this
5.2 Searching on LZ-encoded Text

Checking is performed using $Q_2(\text{Prefix}, SS_i)$. Pattern occurs if $Q_2(\text{Prefix}, SS_i) \neq 0$.

3. If $\text{Prefix} \neq \text{NULL}$ and the current node’s chunk is an internal chunk, compute $\text{Prefix}$ as $Q_1(\text{Prefix}, IP)$, where $IP$ is the current node’s internal range.

4. If $\text{Prefix} \neq \text{NULL}$ and the current node’s chunk is not an internal chunk, set $\text{Prefix}$ as the current node’s prefix number.

The total time and space complexity of the algorithm is $O(n + m^2)$. Amir et al. [16] also described various possible tradeoffs between time and space, leading to an algorithm that runs in $O(n_c \log m + m)$ time using $O(n_c + m)$ space. Kosaraju [196] improved the pattern preprocessing stage used in [16] to obtain an overall improved time for compressed pattern matching, reducing the required time to $O(n_c + m^{1+\epsilon})$, for some small value $\epsilon > 0$. In [41], Amir et al.’s approach [16] was extended to work on general LZ compressed schemes using the so-called ID heuristic [242]. The ID heuristic is also known as LZMW. This heuristic grows the phrases in the dictionary by concatenating pairs of adjacently parsed phrases, rather than just adding one character to an existing phrase. The search method is able to exploit these large components to keep track of whether or not they contain the target pattern. Their algorithm requires $O(m + t)$ space, where $m$ is the pattern size and $t$ is the maximum target length. This is optimal. However, the search time is $O(n_c(m + t))$, where $n_c$ is the size of the compressed file. This is not as good as the $O(n_c + m)$ “optimal” time established by Amir et al. [17].

The original method reported only the first occurrence of the pattern. The basic approach was modified by Tao and Mukherjee [338] to report all pattern occurrences and also for multiple pattern matching. Their method for multiple pattern matching requires $O(n_c + mt + n_{occ})$ time using extra space in $O(mt)$, where $t$ is the size of the LZW trie, $m$ here is the total length of all the patterns, and $n_{occ}$ is the number of occurrences found. We observe that $t$ could be small relative to $n_c$, for a very large text.
5.2.2 Fully-compressed Pattern Matching on LZW-text

Methods for fully compressed pattern matching have also been proposed for LZW compressed strings. Gąsieniec and Rytter [133] proposed an almost-optimal algorithm by observing that, for the LZW-compression, each codeword (dictionary entry) for compressed text can be stored in the dictionary trie, whereby each dictionary entry corresponds to a path from the root to some node in the trie. The key to the method is the idea that both complete codewords and factors of a codeword (subwords of a codeword) can be treated as single comparison elements. This is achieved by representing only factors with lengths in the form $2^k$, $k = 0, 1, 2, \ldots$, called basic factors. Given the basic factors, testing for equality between two arbitrary factors can be performed by decomposing them into a sequence of basic factors. This leads to a logarithmic time and space cost, since we can have at most $O(\log n_c)$ basic factors for a sequence of length $n_c$. Furthermore, two separate tables are maintained to keep the end of codewords in $P_c$ and $T_c$.

The preprocessing step is essentially an extension of the standard KMP preprocessing routine. While the KMP algorithm computes the failure table with information about each position in the pattern, here a restricted failure function is computed, whereby only the end position of codewords are considered. In [133], a recursive algorithm is given for computing the restricted failure table for the compressed pattern $P_c$ in $O((n_c + m_c)\log(n_c + m_c))$, by considering consecutive codeword end positions in the pattern. Given the restricted failure table ($RFT$), the scanning stage is simple, again considering only the codeword end positions in the text. The procedure is as follows: (1) Concatenate $P_c$ and $T_c$ to get a new sequence $PT = P_c \cdot \$ \cdot T_c$; (2) Compute the restricted failure table for $PT$; (3) Check if a prefix of $PT$ can be extended using the restricted failure table. The pattern $P$ occurs in the text $T$ if there is codeword end position $k$ in $PT$, such that $k \geq m + 1$ and $P$ occurs in the text $P[1 \ldots j] \cdot B$, where $j = RFT[k]$, and $B$ is the codeword in $PT$ following position $k$. The case of periodic prefixes or periodic patterns is a little more involved, requiring a call to the function $Q3()$ used earlier in [16] (see also Section 5.2.1). However, all together, no more than $O(n_c + m_c)$ factor comparisons will be required.
Thus, the overall complexity is in $O((n_c + m_c)\log(n_c + m_c))$ time and $O((n_c + m_c)\log(n_c + m_c))$ space. Using the optimality classification for compressed pattern matching algorithms, this is almost-optimal.

Recently, Gawrychowski [129] improved the basic approach described above to an optimal linear time algorithm, using partial decompression on the pattern. Building on his recent $o(n_c + m)$ time algorithms for LZW-compressed pattern matching [127], he extended his method to develop an improved algorithm for the fully compressed pattern matching problem. He improved the almost-optimal complexity results in [133] to an optimal time of $O(n_c + m_c)$. A major difference in the new approach was that rather than using the LZW-coding tries as was done in [133], he used suffix trees to represent the LZW dictionary entries for both the compressed text and compressed pattern. Then, he used the LCA (longest common ancestor) data structures [53] on the suffix trees, which supports constant time look-up to determine the longest common substring between any two substrings (leaves in the suffix tree). This removed the $O(\log(n_c + m_c))$ factor in the search time reported in [133], leading to an optimal $O(n_c + m_c)$ time and $O(n_c + m_c)$ space algorithm.

5.2.3 Searching on LZ77-compressed Text

LZ77 compresses the text by using the previously observed part of the text as the dictionary. To code the symbol at coding step $i$, it outputs the triple $<B_i,L_i,C_i>$, where $B_i$ indicates how far back we need to go in the previous text, $L_i$ indicates how many symbols we need to copy, and $C_i$ is the next symbol from the input text. Thus, for a text string $T = t_1t_2...t_n$ with symbols from the alphabet $\Sigma$, the LZ77 compressed text $T_c$ of $T$ is a string of triples of the form: $T_c[1]T_c[2]...T_c[n_c] = <B_1,L_1,C_1> \cdots <B_i,L_i,C_i> \cdots <B_{n_c},L_{n_c},C_{n_c}>$, where $B_i$ and $L_i$ are integers, and $C_i \in \Sigma$. Using the example text $T = \text{abaababaabaab}$, with $\Sigma = \{a,b\}$, the LZ77-encoded sequence will be the string $T_c = <0,0,a>,<0,0,b>,<2,1,a>,<3,2,b>,<5,4,a>,<3,1,\$>$. Here the symbol $\$ is used to indicate the end of the string (see Table 5.2).

Define $S_1 = 0$, and let $S_i = S_{i-1} + L_{i-1} + 1$. Also let $E_i = S_i + L_i$. Equivalently, $S_i = E_{i-1} + 1$. The parameters $S_i$ and $E_i$ indicate the
Table 5.2. LZ77 encoding steps for input $T = \text{abaababaabab}$.

<table>
<thead>
<tr>
<th>Step i</th>
<th>Code $&lt;B_i,L_i,C_i&gt;$</th>
<th>Start $S_i$</th>
<th>End $E_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$&lt;0,0,a&gt;$</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>$&lt;0,0,b&gt;$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>$&lt;2,1,a&gt;$</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>$&lt;3,2,b&gt;$</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>$&lt;5,4,a&gt;$</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>$&lt;3,1,$ $&gt;$</td>
<td>13</td>
<td>13</td>
</tr>
</tbody>
</table>

respective starting and ending positions in $T$, the uncompressed text, that is encoded at step $i$. That is, the triple $<B_i,L_i,C_i>$ at coding step $i$ encodes the text fragment $T[S_i...E_i]$. A key observation important for both decoding and searching the encoded text is the following relationship:

$$T[S_i...S_i+L_i] = T[S_i - B_i...S_i - B_i + L_i];$$

$$T[E_i] = T[S_i + L_i] = C_i.$$  \hspace{1cm}(5.1)$$

In fact, LZ77 decompression is performed by essentially iterating on $i$ using the above relationship. Farach and Thorup [109] were the first to propose a method for searching on LZ77-text. They observed that, given that $t$ is the least position such that the pattern $P$ of length $m$ occurs at position $t$ in the text, (i.e., $T[t...t+m-1]$), then there must be some $i$, such that $S_i \in \{t, t+1, \ldots, t+m\}$. Based on this observation, for a given encoding step $i = 1, 2, \ldots, n_c$, they computed the quantities $P^+_i$ and $P^-_i$, where $P^+_i$ is defined as the longest pattern substring that is also a prefix of $T[S_i...n]$, and $P^-_i$ is the longest pattern substring that is also a suffix of $T[1:S_i + L_i]$. Using the above, the compressed pattern matching problem is then reduced to that of checking whether the pattern $P$ occurs as a substring of the concatenation $P^-_i \cdot P^+_i+1$, for some $i = 1, 2, \ldots, n_c$.

Using the example encoded text, Table 5.3 shows the longest prefix patterns and longest suffix patterns computed at each step of compressed pattern matching, for two sample patterns. Similar to the case of searching on LZW-text, at a given scanning step $i, i = 1, 2, \ldots, n_c$, we can observe how the $P^+_i$ and $P^-_i$ combine to determine possible
matches with the pattern. For instance, we only need to check for possible matches around the text regions where $|P_i^- \cdot P_i^+| \geq m$.

Farach and Thorup thus performed compressed pattern matching by introducing informers to indicate the regions that are likely to lead to pattern occurrences, as captured by $P_i^+$ and $P_i^-$, and thus decompress and analyze only the areas identified by the informers. They discussed various methods to reduce the computational requirements, for instance, representing $P_i^+$ and $P_i^-$ in constant space using beginning and ending pointers into the pattern, rather than as symbol sequences. They preprocessed $P$ in linear time using their earlier algorithm [145], such that given a concatenation of two substrings in $P$, it can be determined whether $P$ occurs in the concatenation in $O(\log m)$ time. Thus, they showed a compressed matching algorithm that runs in $O((m + n_c \log \frac{n}{n_c})(\log \frac{n}{n_c} + \log m))$. Using a randomized algorithm based on the Karp–Rabin fingerprint method, they improved the time bound to $O(m + n_c \log^2(n/n_c))$.

Navarro and Raffinot [259, 261] proposed a hybrid compression scheme between LZ77 and LZ78 which can be searched in
$O(\min(n, n_c \log m) + \eta_{occ})$ average time, where $\eta_{occ}$ is the total number of matches. More recently, Gawrychowski [127] introduced a fully deterministic algorithm that achieves optimal performance (i.e., $O(n_c + m)$ time) for LZW-compressed strings. The algorithm built on ideas similar to the KMP algorithm combining several known tools, such as longest common ancestor queries, borders, and periodicity in strings. In [128], the method was extended to tackle the relatively more difficult problem of compressed pattern matching on the general LZ compressed strings. He improved the (randomized) $O(n \log^2 \frac{n}{n_c} + m)$ time reported in [109] to a deterministic almost-optimal time of $O(n \log \frac{n}{n_c} + m)$, using the same $O(n \log \frac{n}{n_c} + m)$ space. As discussed in [127], the *collage system* reported in [180] can be applied for compressed pattern matching on LZ77 or LZ78 (with appropriate modifications on the input string), leading to an $O(n \log \frac{n}{n_c} + m^2)$ time and space algorithm. A search tool for LZ-compressed text using the BM algorithm was reported in [263].

Other methods based on conversion of the LZ-compressed text to straight-line programs (SLPs) (see below), and then performing compressed matching on the SLP have also been proposed. Recent examples can be found in [128] where they reported an $O(n_c \log \frac{n}{n_c} \log m_c + m_c)$ time algorithm for compressed pattern matching on LZ77-text, which was improved to $O(n_c \log \frac{n}{n_c} + m_c \log m_c)$ time using properties of balanced SLPs. Another is the work of Jež [167], who performed fully compressed pattern matching using local recompression on SLPs. He showed how the method can be generalized for LZW-text, leading to $O(q \log q \log m)$ search time, where $q = n_c \log \frac{n}{n_c} + m_c \log \frac{m}{m_c}$.

### 5.3 Grammar-based Compression

Grammar-based coding, studied by Kieffer and Yang [183, 184, 380] is another text compression scheme that is closely related to the dictionary based methods. It compresses data by a recursive decomposition of the original sequence, and subsequent substitution of repeating patterns by generating a grammar that represents the patterns. Essentially, the original sequence is transformed into a context free grammar, and the grammar is then compressed. Grammar-based codes use the idea of multilevel pattern matching (MPM) to decompose the sequence, and
can be viewed as a generalization of traditional LZ-based compres-
sion methods. An earlier example of a grammar-based coding system is
SEQUITUR, introduced by Nevill-Manning and Witten [265, 266]. The
results of the production rules can be the basis for pattern matching.
A special type of grammar-based coding is the *straight-line program*,
which have been found suitable for fully-compressed pattern matching.

A straight-line program (SLP) is a context free grammar in the
Chomsky normal sense that derives a single string. Thus, with SLPs,
the text is compressed by representing it using a context free grammar.
The SLPs have a strong connection with the LZ family, since SLPs can
also be viewed as a form of dictionary-based compression where, like
the LZ family, the previously encountered text is used as a dictionary.
In fact, an LZ-compressed text can be tuned into an equivalent SLP
in $O(n \log \frac{n}{m})$ time, with just an $O(\log \frac{n}{m})$ overhead [291]. Similarly,
an SLP can be turned into an equivalent LZ-text in linear time and
$O(1)$ overhead. Thus methods for searching on SLPs can be applied on
LZ-text. Not surprisingly, most methods for fully compressed pattern
matching on LZ-text have first converted the LZ-encoded pattern and
text into their respective SLPs, and then perform matching on the
SLPs.

More specifically, following Karpinski et al. [175], the SLP $S$ can be
described as a sequence of assignments:

$$X_1 = expr_1; X_2 = expr_2; \ldots; X_n = expr_n,$$

(5.2) where the $X_i$ and $expr_i$ are variables and expressions, respectively. An
expression is either a constant (i.e., symbol in the alphabet), or a con-
catenation of previous expressions. That is, we have (1) $expr_i \in \Sigma$, or
(2) $expr_i = X_j \cdot X_k$, for some $j < i, k < i$, where “$\cdot$” denotes concate-
nation. Let $val(X_i)$ denote the resulting string when the expression $X_i$
is uncompressed. Thus, when the SLP $S$ is executed, the output $S$
will be the value of the last variable $X_{ne}$, that is, $S = val(X_{ne})$. The size of
the SLP $S$ is given by $n_c$. Thus, in terms of compression, for a given
text $T$, we will have $T_c = SLP(T)$, and $n_c = |T_c|$. We can observe that
for certain inputs, for example $T = a^n$, or highly compressive strings
such as Fibonacci strings, the size of the uncompressed text may be
exponentially large when compared with the size of the corresponding
SLP. The SLP provides very good compression in such instances. Similarly, the number of occurrences could also be exponential with respect to \( n \) — consider the case with \( P = a^{2^m} \) and \( T = a^{2^n} \). Thus, we may not be able to list all the positions of occurrences explicitly, but would rather want a compact representation of all such positions. This also means that initially decompressing the text before searching may not be feasible. Similarly, the pattern could be exponentially long. These problems motivate the need for fully compressed pattern matching in general, and for straight line programs in particular.

5.3.1 Fully Compressed Pattern Matching on SLPs

Using the example with the Fibonacci string \( T = \text{abaababa} \), we have the following SLP representation:

\[
\begin{align*}
X_1 &= b; \\
X_2 &= a; \\
X_3 &= X_2 \cdot X_1; \\
X_4 &= X_3 \cdot X_2; \\
X_5 &= X_4 \cdot X_3; \\
X_6 &= X_5 \cdot X_4;
\end{align*}
\]

Now, given the pattern \( P = \text{aba} \), we have the corresponding SLP:

\[
\begin{align*}
X_1 &= b; \\
X_2 &= a; \\
X_3 &= X_2 \cdot X_1; \\
X_4 &= X_3 \cdot X_2;
\end{align*}
\]

Thus, we have the compressed text \( T_c = X_6 \) and compressed pattern \( P_c = X_4 \). Our challenge in fully compressed pattern matching is to decide if \( P \) occurs in \( T \) using their SLPs, without decoding. This problem has been studied by many authors [175, 215, 244]. Below, we describe the method proposed by Karpinski et al. [175]. As with general compressed pattern matching, the two basic steps of preprocessing, and scanning based on the results of the preprocessing are also followed. The major procedure in the preprocessing is the construction of the overlap data structure. To understand fully compressed pattern matching using
SLPs, we consider the following aspects:

1. The overlap data structure
2. Space-efficient encoding of the overlap structure
3. Computing the overlap table using the compressed (SLP) codes
4. Finding pattern matches using the overlap structure.
5. Complexity results

Given two strings $X$ and $Y$, the set of overlaps between $X$ and $Y$ is defined as the set of positions $k$, where the $k$th prefix of $Y$ equals the $k$th suffix of $X$. More formally,

$$\text{Overlap}(X,Y) = \{k > 0 : X[|X| - k + 1...|X|] = Y[1...k]\} \quad (5.3)$$

For example, $\text{Overlap}(X_4,X_5) = \text{Overlap}(aba,abaab) = \{1,3\}$, while $\text{Overlap}(X_5,X_4) = \text{Overlap}(abaab,aba) = \{2\}$. For the straight-line program $S$, of size $n_c$, the overlap data structure for $S$ is an $n_c \times n_c$ table $OV$, where $OV(i,j) = \text{Overlap}(X_i,X_j)$. Using the example SLP for $T$ above, the corresponding overlap table is given in Table 5.4.

Each element in the overlap table is a set, with potential cardinality that is exponential with respect to $n_c$. In [175] it was observed that each element can be represented as a set of at most $n_c$ arithmetic progressions. Since each progression can be represented in constant space (e.g., using three numbers, namely, the starting number, difference between

<table>
<thead>
<tr>
<th></th>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$X_3$</th>
<th>$X_4$</th>
<th>$X_5$</th>
<th>$X_6$</th>
<th>$X_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>{1}</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>$X_2$</td>
<td>\emptyset</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td></td>
</tr>
<tr>
<td>$X_3$</td>
<td>{1}</td>
<td>\emptyset</td>
<td>{2}</td>
<td>{2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_4$</td>
<td>\emptyset</td>
<td>{1}</td>
<td>{1}</td>
<td>{1,3}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_5$</td>
<td>{1}</td>
<td>\emptyset</td>
<td>{2}</td>
<td>{2}</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_6$</td>
<td>\emptyset</td>
<td>{1}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$X_7$</td>
<td>{1}</td>
<td>\emptyset</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See text for the SLP encoding of $T$. Only the first few elements in the matrix are shown.
successive numbers in the progression, and the ending number), each element in the overlap table can be represented in $O(n_c)$ space, for a total of $O(n_c^3)$ storage space. Further, the set of progressions at each element are kept in a sorted form, such that queries of the form: “is $q \in \text{Overlap}(X,Y)$?” can be answered in $O(\log n_c)$ time.

Computing the overlap table is performed by observing the dependence of the values at a given position $(i,j)$ in the table on previously computed entries at positions $(i',j')$, $i' < i, j' < j$. The procedure is as follows:

1. Compute $\text{Overlap}(X_i, X_j)$, where $X_i$ is a terminal symbol
2. Compute $\text{Overlap}(X_i, X_j)$, where $X_j$ is a terminal symbol
3. Compute $\text{Overlap}(X_i, X_j)$ for non-terminal symbols:
   for ($i = 3$ to $n_c$, $j = 3$ to $n_c$)
   {
     Let $X_i = X_l \cdot X_r$
     $U_1 = \text{Overlap}(X_l, X_j)$;
     $U_2 = \text{prefixExtension}(U_1, X_j, X_r)$;
     $W = \text{Overlap}(X_j, X_r)$;
     $\text{Overlap}(X_i, X_j) = \text{compress}(U \cup W)$;
   }

Given the set $U \subset \{1, 2, \ldots, n\}$, and two strings $A$ and $B$, the function $\text{prefixExtension}()$ is defined as follows:

$$\text{prefixExtension}(U, A, B) = \{k + |B| : k \in U \text{ and } A[1..k] \cdot B \text{ is a prefix of } A\}.$$  

(5.4)

The function $\text{compress}()$ essentially encodes the set of numbers at the given $(i,j)$ position in a succinct manner using $O(n_c)$ space, as described above. Steps 1 and 2 require only constant time operations, when both $X_i$ and $X_j$ are constant symbols (simply compare the two symbols). For the cases when $X_i$ is a constant symbol, but $X_j$ is not, we still need to compare with one symbol. The element will be either $\{1\}$ or $\emptyset$: $\{1\}$ if the first symbol in $\text{val}(X_j)$ equals $X_i$, and $\emptyset$ otherwise. Similarly, when $X_j$ is a constant symbol, but $X_i$
is not, the element will be either \(\{1\}\) or \(\emptyset\), depending on whether the last symbol in \(val(X_i)\) is equal to \(X_j\) or not. Each case will require at most \(O(n_c)\) time to determine the last symbol, or the first symbol in the uncompressed value of the given variable. Thus, the algorithm first fills in the first two rows and columns in \(O(n_c^2)\) time, before proceeding with subsequent entries. These entries are computed by decomposing each variable (e.g., \(X_i = X_l \cdot X_r\), with \(l < i, r < i\)), and using the overlap procedure involving these constituent variables, since these must have been processed at an earlier step. Using the example, \(\text{Overlap}(X_3, X_4)\) will be computed as follows:

\[
X_i = X_3 = X_2 \cdot X_1; \quad X_j = X_4; \quad U_1 = \text{Overlap}(X_2, X_4) = \{1\}; \quad U_2 = \text{prefixExtension}(U_1, X_4, X_1) = \{2\}; \quad W = \text{Overlap}(X_4, X_1) = \{0\}.
\]

This gives \(\text{Overlap}(X_3, X_4) = \{2\}\). Similarly, for \(\text{Overlap}(X_1, X_4)\) we obtain \(U_1 = \{1\}; U_2 = \{3\}; W = \{1\}\), or \(\text{Overlap}(X_4, X_4) = \{1, 3\}\). Thus, by re-using previously computed elements, we can fill-up all the elements in the table. In [175], it was shown how the \text{prefixExtension()}\ function can be computed in \(O(n_c \log n_c)\) time using the compressed elements in the overlap table. This operation is performed for each arithmetic progression at a given element. Since we have potentially \(O(n_c)\) arithmetic progressions at each element in the table, the operation will be applied an \(O(n_c^3)\) number of times, resulting in an overall time of \(O(n_c^4 \log n_c)\) to compute the overlap table.

Now given the pattern \(P\) also represented as an SLP, compressed pattern matching is performed by scanning the SLP encoding the text, using the overlap table. This is a two step procedure:

1. **Preprocessing:** For each variable \(X_k\) in the SLP of \(T\), compute \(\text{Overlap}(X_k, P)\) and \(\text{Overlap}(P, X_k)\); Tables 5.5 and 5.6 show the tables for \(\text{Overlap}(T, P)\) and \(\text{Overlap}(P, T)\), using \(P = \text{bab}\). For this example, the SLP for \(P\) is given as: \(\{Y_1 = b; Y_2 = a; Y_3 = Y_1 \cdot Y_2; Y_4 = Y_3 \cdot Y_1\}\). Thus, \(P_c = Y_4 = Y_3 \cdot Y_1\).

2. **Scanning:** Define the match function:

\[
f(p, A, B) = \begin{cases} 
  i, & \text{if } (i \in A), (j \in B), \text{ and } (i + j) = p \\
  0, & \text{otherwise}
\end{cases} \quad (5.5)
\]

Scanning is then performed as follows: Find an \(X_k\), such that \(X_k = X_I \cdot X_r\) and \(f(|P|, \text{Overlap}(X_I, P), \text{Overlap}(P, X_r)) \neq 0\).
Whenever a nonzero value is returned (i.e., a match is found), the exact location of the match can also be derived from the returned value, \( i \). That is, the match starts in \( X_k \) using the length-\( i \) suffix of \( X_l \), and finishes with the length-(\( m - i + 1 \)) suffix of \( X_r \). We can see that our case of \( P = \text{aba} \) with \( SLP(P) = P_a = X_4 = X_3 \cdot X_3 \) now looks trivial. From Tables 5.5 and 5.6, we see that for the case of \( P = \text{bab} \), with \( P_a = Y_4 \), a matching will be found in \( X_6 = X_5 \cdot X_4 \), straddling \( X_5 \) and \( X_4 \).

After computing the overlap tables, scanning for matching patterns is a much simpler problem, requiring \( O(n_c^3) \) time for the \( f() \) function, or \( O(n_c^4) \) total time. Thus, the overall time for compressed pattern matching on SLPs is dominated by the time for preprocessing, which is in \( O(n_c^4 \log n_c) \). The overall space is in \( O(n_c^3) \). Following a similar approach, Miyazaki et al. [244] proposed an improved method that runs in \( O(n_c^2 m_c^2) \) time using \( O(n_c m_c) \) space. This was further improved to \( O(n_c^2 m_c) \) by Lifshits [215].

**Table 5.5.** Structure for Overlap\((T, P)\), using the SLP encoding of \( T = \text{abaababa} \) and \( P = \text{bab} \). See text for the SLP encoding of \( T \) and \( P \).

<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( Y_1 )</th>
<th>( Y_2 )</th>
<th>( Y_3 )</th>
<th>( Y_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( X_2 )</td>
<td>{1}</td>
<td>0</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>( X_3 )</td>
<td>0</td>
<td>{1}</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( X_4 )</td>
<td>{1}</td>
<td>0</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>( X_5 )</td>
<td>0</td>
<td>{1}</td>
<td>0</td>
<td>{2}</td>
</tr>
<tr>
<td>( X_6 )</td>
<td>{1}</td>
<td>0</td>
<td>{1}</td>
<td>{1}</td>
</tr>
</tbody>
</table>

**Table 5.6.** Structure for Overlap\((P, T)\), using the SLP encoding of \( T = \text{abaababa} \) and \( P = \text{bab} \). See text for the SLP encoding of \( T \) and \( P \).

<table>
<thead>
<tr>
<th>( X_1 )</th>
<th>( X_2 )</th>
<th>( X_3 )</th>
<th>( X_4 )</th>
<th>( X_5 )</th>
<th>( X_6 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( Y_1 )</td>
<td>{1}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( Y_2 )</td>
<td>0</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>( Y_3 )</td>
<td>0</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>( Y_4 )</td>
<td>{1}</td>
<td>0</td>
<td>{2}</td>
<td>{2}</td>
<td>{2}</td>
</tr>
</tbody>
</table>
5.3 Grammar-based Compression

The above approaches exploited various combinatorial properties of strings (such as periodicity) in order to perform fully compressed pattern matching on SLPs. In a significant departure from the above methods, Jeż [167] proposed quite a different approach, using the idea of re-compression. Here, the SLPs for both the text and the pattern are re-factored using information from the pattern, such that substrings that appear in both the pattern and the text are now represented in a uniform manner in the resulting SLPs. Essentially, this involves some local decompression and then local recompression of the SLPs encoded in $T_c$ and $P_c$. Each pair of symbols, say $ab$, that appear in the pattern (called crossing pairs) are replaced by a new symbol, say $c$, in both the pattern SLP and text SLP. This idea is similar to the use of bigrams in [233] and in byte pair encoding (BPE) [307] to enhance the ability for compressed pattern matching on the encoded text. Let $val(X_i)$ denote the resulting string when the expression $X_i$ is uncompressed. A crossing pair $ab$ is recompressed as follows: if symbol $a$ is to the left of $X_i$ and $val(X_i) = bz$, modify the production rule $X_i$ such that $val(X_i) = z$, then replace $X_i$ by $bX_i$ in every rule where $X_i$ appears. A similar transformation is performed for the case when the non-terminal $X_j$ is to the left of $b$, such that $val(X_j) = za$. Maximal runs of the same symbol are also replaced with a shorter code word (e.g., $a^k$ is replaced with $a_k$).

Each round of replacement typically would lead to a shorter SLP. The procedure is applied iteratively on the two SLPs, until the pattern is replaced by only one symbol. That is, the pattern SLP $P_c = X_{m_c}$ now points to one new terminal symbol. At this point, for the simple cases, the pattern occurrences in the text (if any) will be explicit, and can be found easily.

Consider the example with $T = abaababa$, $P = bab$. The first round of replacements (replace $ba$ with $c$) will yield $P = cb$ and $T = acacc$. A second round of replacements (replace $cb$ with $d$) will yield $P = d$ and $T = acada$ (after some further processing). Using $P = aba$, the first round of replacements (replace $ab$ with $c$) will yield $P = ca$ and $T = cacca$. A second round of replacements (replace $ca$ with $d$) will yield $P = d$ and $T = dcd$.

In the examples, the pattern occurrences are now explicit, and can be found using a simple scan. We can also observe the difficulty when
an occurrence is embedded within the new symbols, for example, in the second example, there is a third occurrence of $P = \text{d}$ inside the text $\text{dcd}$. Also, there could be a problem with runs of the same symbol, e.g. $P = \text{b}a^4\text{b}$. Jež [167] described how these problems can be handled to ensure correct results in the pattern matching. He showed that at most an $O(\log m)$ rounds of replacement are required for the recompression, and that at most $O((n_c + m_c) \log (n_c + m_c))$ new symbols are added at each round. Thus, the overall algorithm requires an $O((n_c + m_c) \log m \log (n_c + m_c))$ time and $O((n_c + m_c) \log (n_c + m_c))$ space to return an $O((n_c + m_c) \log (n_c + m_c))$ representation of all pattern occurrences in the text.

Interestingly, computing various other related quantities directly on the compressed data (i.e., on the SLPs) lead to NP-complete problems, for instance, the Hamming distance [215] and regular expression matching [175]. See also [55, 290] for more discussions on complexity issues in fully compressed pattern matching.

### 5.4 Run-length Encoding

Run-length encoding is a simple compression method which replaces runs of a symbol with a count of how many times it is repeated. This is particularly effective for a black and white image where it isn’t necessary to identify the symbol since it will always be alternating runs, and this approach is the foundation of coding used for fax machines.

Methods for pattern matching relating to run-length encoding have been considered for some time. Eilam and Vishkin [106] described the problem of pattern matching after various transformations on the input string. Initial ideas on pattern matching on RLE sequences were given in [28, 67, 68]. In [10] it was observed that when video sequences are represented as strings, nearby symbols in the string are often the same or similar, resulting in long runs of repeated (or similar) symbols. To accommodate the symbol repetition in matching the video sequences, a special type of edit operation was defined which works on a representation similar to the RLE. In [232] the authors studied methods for pattern matching on RLE-compressed sequences, including images. Below we describe one basic approach.
Consider the RLE for the text $T$ and pattern $P$, given as follows:

$$T_c = \text{RLE}(T) = a_1^{h_1} a_2^{h_2} \ldots a_u^{h_u} \quad \text{and} \quad P_c = \text{RLE}(P) = b_1^{g_1} b_2^{g_2} \ldots b_v^{g_v},$$

where $a_i \neq a_{i+1}, 1 \leq i < u$, and $b_j \neq b_{j+1}, 1 \leq j < v$, and $h_i$s and $g_j$s are positive integers, with $\sum_i h_i = |T| = n$ and $\sum_j g_j = |P| = m$. Then, there is a simple algorithm to search for $P$ in $T$ via their RLE representation [15, 290]: Construct auxiliary sequences:

$$T_1 = a_2 a_3 \ldots a_{u-1}, \quad P_1 = b_2 b_3 \ldots b_{v-1},$$
$$T_2 = h_2 h_3 \ldots h_{u-1}, \quad P_2 = g_2 g_3 \ldots g_{v-1}.$$  

Using linear time search algorithms, such as the KMP or BM algorithms, simultaneously, search for all occurrences of $P_1$ in $T_1$, and of $P_2$ in $T_2$. For each starting occurrence, say $i$, verify if $P$ actually occurs at position $i$ in $T$. This verification can be done in constant time for each $i$. Thus, using $n_c = |\text{RLE}(T)| = u$, $m_c = |\text{RLE}(P)| = v$, the overall time needed for pattern matching will be in $O(n_c + m_c)$. This is optimal.

In a recent paper [333], methods were proposed for converting RLE-encoded text to LZ78, and vice versa, without any initial decompression.

### 5.5 Statistical (symbolwise) Compression

Statistical methods take a different approach to compression. Each symbol in the input (typically a character, byte, or pixel) is coded based on its probability of occurrence. According to Shannon’s noiseless source coding theorem, a symbol with probability $p$ is optimally coded in approximately $-\log_2 p$ bits. The classic method for symbolwise coding is Huffman coding [153], a method for optimally allocating short integer codes to more probable strings, and longer codes to the less probable. Huffman coding is known to be optimal for a given block length. However, it uses a whole number of bits to code the input symbols when the optimal coding could share one bit with more than one symbol, and so it isn’t necessarily optimal. This can be partially addressed by coding blocks of symbols using extensions of the source,
although another method called arithmetic coding can be more effective for this (see below).

The task of finding an appropriate probability distribution for a given text to which the Huffman algorithm can be applied is something of an art, and is discussed below. The probability distribution is often referred to as the model, since it abstracts important features of the data. In a sense, the models used to determine/generate the probability distributions perform a kind of transformation, for instance transforming the input symbols into some simple frequency counts. Hence, for statistical methods, the transformation stage in Figure 1.1 captures the modeling part, while the encoding stage determines the appropriate codes for the data using the symbol probabilities. First we describe Huffman and arithmetic coding, which serve as the back-end for most symbolwise and lossy compression methods. Then we look at how models can be constructed to provide good probability distributions, and in turn, how pattern matching can be performed for the various combinations of models and coding methods.

Huffman’s method for coding is based on a binary tree. An example is given in Figure 5.2. This tree describes the encoder for an alphabet of four characters, each with some estimated probability of occurrence. The codeword for a particular character is given by the path from the root to the corresponding leaf node. For example, an “a” is coded as 000, and a “d” as “01”. The tree is generated using a simple algorithm: the two least probable leaves are paired to form an aggregate node (or symbol) which replaces the two leaves. The probability of the new node equals the sum of the probability of the two leaves.

![Huffman coding tree](image)

Fig. 5.2 Huffman coding tree for a source with symbol alphabet $\Sigma = \{a, b, c, d\}$ and symbol probabilities: $p(a) = 0.13, p(b) = 0.44, p(c) = 0.21, p(d) = 0.22$. 
The pairing process is applied on the reduced alphabet (now of size 3 in our example). This pairing process is applied until there is just one aggregate node, the root. This greedy algorithm is usually implemented using a heap to find the node with the smallest value. The codewords are given by the path from the root to the leaves; there will be many possible ways to label this, although in practice, a variation of the code called canonical Huffman codes [150] is common. Canonical Huffman codes have the same length as the traditional Huffman code, but have their bit patterns chosen in a way that makes them very fast to encode and decode using a lookup table rather than a tree.

The set of codewords thus forms a kind of vocabulary for the symbols in the text. Clearly, such a vocabulary can be exploited in performing compressed domain search on the text.

Other variants of the Huffman approach with special relevance to searching have also appeared in the literature. The byte-oriented word-based Huffman coding scheme [250, 399] uses words and text separators (rather than individual characters) as the basic symbols in the alphabet. Then a sequence of bytes (rather than the usual bits) are assigned to each word or separator. The plain Huffman code uses all the bits in the bytes, while the tagged Huffman code uses the highest order bit to tag the first byte of each codeword. The tagging was found to be useful in searching directly on the compressed data, because it provides random access. Although other statistical compression methods have been developed, Huffman coding is still very important because in many situations it produces very good compression, with high compression speed.

Another statistical method, called arithmetic coding improved compression by “splitting the bit.” Like Huffman coding, arithmetic coding also represents the input symbols based on a given probability distribution, assigning shorter descriptions to more probable symbol blocks, and longer descriptors to less probable blocks. It, however, allows coding arbitrarily close to the Shannon limit without incurring an undue computational cost. In arithmetic coding the representation of consecutive symbols may overlap. Thus one bit in the output may contain information about two different input symbols, and the boundary between two symbols is not likely to have a corresponding boundary
between two output bits. Arithmetic coding is the basis for many high performance compression schemes for both text and images. For more details on arithmetic coding, see [51, 246, 375].

With arithmetic coding, the same symbol can have quite different representations each time it is coded. Further, because of the possible overlap in the coding of two different symbols, decompression cannot normally start in the middle of the compressed file. This means that it will be difficult to provide random access to the compressed data. Therefore, arithmetic coding is generally viewed as not being very suitable for compressed-domain searching [36]. However, recent results on the use of arithmetic coding for construction of suffix arrays [11, 43], a key data structure for efficient pattern matching, may raise the hope for the possibility of pattern matching on arithmetic coded sequences. This is an area that needs to be investigated further.

We now turn to the models that are used for symbolwise methods. These models provide the probability distribution that Huffman and arithmetic coders use to create a bit-stream. One of the simplest models is one that counts symbol frequencies and uses the relative frequency of a symbol to estimate its probability. For example, if the symbol “u” accounts for 4 out of 200 characters then we might estimate its probability to be 2%, and using Shannon’s formula this should be coded in about $-\log_2 0.02 \approx 5.6$ bits. This kind of model is often used in conjunction with Huffman coding, and is particularly effective when the symbols being used are English words. For example, a word-level Huffman code is used in the mg full-text retrieval system [376], as it gives very good compression, and also the unit of coding (words) matches the unit that will be used for searching the text later.

The simple model just described can be improved by observing that the probability of a symbol is influenced by its context. For example, suppose whenever the letter “q” has occurred in a text that it is followed by the letter “u” 98 times out of 100. We can then estimate the probability of a “u” in this context to be 98%, which should be coded in about $-\log_2 0.98 \approx 0.03$ bits — considerably less than the 5.6 bits with the simple model. This is a particularly extreme example, but considerable gains can be made by using contextual information. This type of model is called a finite context model, and has a strong
5.5 Statistical (symbolwise) Compression

connection to Markov models. The order of the model is the number of symbols used as context; the example above used an order-1 finite context model. Even better compression is achieved using higher order models, but if the context size is too large then too much memory is required to keep track of all the contexts, and also large contexts will not have occurred very many times, and so probability estimates will be based on unreliable samples (if any is available!)

The probabilities can either be estimated statically (by determining them in advance, possibly from the text to be encoded), or adaptively (by keeping count of the character frequencies during encoding, and basing probability estimates on the counts of previously coded characters). In an adaptive coding situation, the decoder must also keep track of the counts so that it is using the same model as the encoder. This is not difficult because the counts are taken only from data that has already been decoded. For adaptive models, in the limiting case when the sequence length tends to infinity, the probabilities they generate will asymptotically approach the ones generated by the static models. Of course, adaptive models will make pattern matching particularly challenging because the codes can potentially change at every step of encoding.

The methods that achieve the best compression are based on adaptive finite context models. One of the most widely known is the “prediction by partial matching” (PPM) system [88, 89], which uses a variety of context sizes. To encode each symbol, PPM looks for the longest context available to code a character. Shorter contexts are needed early in the coding when few contexts have been seen. Arithmetic coding is used to efficiently code the probabilities, which are normally quite low. The PPM system usually uses a trie-like data structure [51] to search for previous occurrences of the context, looking for the longest match possible. This structure must also be maintained by the decoder, which raises the possibility of using the structure for compressed-domain searching. The decoding process would still have to be performed in full (particularly because arithmetic coding is used), but the output need not be stored, and it may be useful to keep the search trie in memory after decoding to use as an index to perform multiple pattern matching operations on the text. The PPM idea has been
extended to the concept of prediction by partial approximate matching (PPAM) for lossless image compression [394].

Another kind of model used for text compression is Dynamic Markov Compression (DMC), which uses a finite state automaton to make probability estimates for symbols [93]. Bell and Moffat [47] have shown that these models are equivalent to finite context models, but are a lot simpler to implement (although they use a lot of resources at runtime). As with PPM, there may be some possibility of using the finite state machine data structure to search for patterns, possibly after a whole text has been decoded.

Klein [185] has suggested improving the decompression time for static compression schemes by extending the symbol alphabet. The idea is to construct a meta-alphabet, which extends the standard alphabet by including frequent words, phrases, or word fragments. The main advantage here is the improved decompression time, even at the possible expense of longer compression time.

5.5.1 Searching on Huffman Codes

Mukherjee and Acharya [251] describe techniques for searching Huffman compressed files. We first observe that a simple search of the Huffman coded file to find a compressed pattern using a fast search algorithm such as KMP may not always produce correct results. For example, consider the following Huffman codes: \( a = 0 \), \( b = 1 \), \( c = 110 \), and \( d = 111 \). If the text is \( T = \text{abbacdacca} \), the compressed text will be \( C(T) = 0101001101101101100 \). If the pattern is \( P = \text{ab} \), the compressed pattern is \( C(P) = 010 \). If we now match the compressed pattern against the compressed text, we will get three matches of which the second match beginning at the third bit position from left is a false match. Furthermore, if the pattern is of the form \( P = aXbYc \), where \( X \) is a wildcard character and \( Y \) is a variable length wildcard character (that is, any sequence of characters of finite length), the representation of \( P \) becomes ambiguous in compressed form and the pattern matching algorithms fail to work.

The basic idea used in [251] is to address this by using a data structure that will determine the byte boundaries in the variable length
5.6 Text Compression by Sorted-contexts

Some compression methods make use of contexts, but do this by permuting the text so that characters that appear in similar contexts occur adjacent to each other in the permuted text. This general technique is called block sorting, or the Burrows–Wheeler transform (BWT) [3, 70, 112]. A large block of text is coded at a time. Essentially a list is constructed of every character and its (arbitrarily long) context, and then the list is sorted lexically by contexts. The characters are then transmitted in this sorted order. This permutation of the characters has the desirable property that characters in lexically similar contexts will be near each other, and this can be exploited by using

coded compressed text to initiate pattern search with respect to the compressed pattern. Related VLSI algorithms were also discussed in [252]. Their method raises the possibility of searching for part of a variable-length compressed string, even if the compressed file is only searched on byte boundaries. This is achieved by searching for all variations of the search strings generated by starting at different points in the string. Only eight starting points need to be considered to cover every possible way the coded string could cross a byte boundary. It seems that this idea could also be applicable to the Burrows–Wheeler Transform [70], as part of BWT involves sorting the data according to context. Mukherjee and Acharya’s method [251] can also be extended to handle patterns with fixed or variable length wildcard characters and to search data that has been compressed with an adaptive Huffman code.

Moura et al. [250, 399] proposed a semi-static word-based modeling scheme for Huffman coded text files which can be searched directly at word boundaries using any fast sequential pattern search algorithm. The coding alphabet is byte oriented and not bit oriented. The first bit of the byte is used to mark the beginning of a word. The authors report a factor of 2 improvement in the search time of the compressed file in comparison to searching using the uncompressed files, and a 33% compression ratio for the TREC 3 collection [345]. Other methods for pattern matching on Huffman coded text are reported in [186, 188].

5.6 Text Compression by Sorted-contexts

Some compression methods make use of contexts, but do this by permuting the text so that characters that appear in similar contexts occur adjacent to each other in the permuted text. This general technique is called block sorting, or the Burrows–Wheeler transform (BWT) [3, 70, 112]. A large block of text is coded at a time. Essentially a list is constructed of every character and its (arbitrarily long) context, and then the list is sorted lexically by contexts. The characters are then transmitted in this sorted order. This permutation of the characters has the desirable property that characters in lexically similar contexts will be near each other, and this can be exploited by using
a coder that assigns short codes to recently seen symbols. From the viewpoint of the general compression model of Figure 1.1, the permutation and subsequent sorting correspond to the transformation stage. Remarkably, the permuted stream of characters can be used to reconstruct the original stream. The inverse transformation is based on the observation that the decoder can sort the sequence of characters to get the last character of the sorted context. This provides enough information to reconstruct the order of the original characters.

The block sorted contexts in the encoder are an excellent index for a pattern matching system, since only a binary search is required to locate a given pattern, and similar patterns will be adjacent. The decoder only has limited information about the sorted context, but, for example, it may be possible to exploit this to perform an initial match on two symbols (a character and its context), and then decode only that part of the text to see if the pattern match continues.

Various methods have been proposed for searching on BWT compressed text [5, 48, 113, 114, 116, 392]. In fact, the BWT has led to various methods for compressed full text retrieval, and perhaps, equally significantly, to methods for compressed representation of search index structures, such as compressed suffix arrays. The BWT-based FM-index of Ferragina and Manzini [113, 114] was a precursor to many more recent methods for compressed suffix arrays and for compressed full text retrieval. Several other methods have been proposed, especially for compressed suffix trees and compressed suffix arrays. These are outside the scope of the current review. See [144] for more details.

### 5.6.1 Searching on BWT-text

There are a number of ways to get fast pattern matching by taking advantage of the transform capturing the lexical ordering of substrings. As observed above, it is possible to reconstruct the block sorted matrix at the decoder side and then perform arbitrary length pattern searches with it, although reconstructing the block isn’t required for decoding and so some extra processing and storage will be required.
Furthermore, because the text is broken up into blocks, the process has to be repeated for each block to complete the pattern search. An inverted index file giving the blocks where the pattern may possibly occur could be used to expedite the search process. This observation has been exploited in [48] for pattern matching on BWT text. Ferragina and Manzini [113, 114] earlier proposed the FM-index for indexing and rapid search on BWT compressed sequences. Adjeroh et al. [3, 116] provide a detailed comparison on methods for BWT-based pattern matching. Figure 5.3 (taken from [3]) shows a performance comparison for BWT-based pattern matching algorithms. As the figure shows, different algorithms perform better on different performance metrics. While suffix array and binary search [48] seem to do well with multiple patterns, the FM-index outperformed all the others in counting pattern occurrences.

5.7 Variable-to-fixed Length Codes

Another class of statistical codes is the variable-to-fixed length codes, whereby fixed-length codewords are used to encode variable-length segments of the original sequence. This class of compression schemes has the advantage of efficient decoding, though the performance in compression may be worse than standard schemes, such as fixed to variable-length codes (such as Huffman codes), or the variable to variable-length codes (which is the case for most implementations of the LZ family). Tunstall codes [351] are a typical example of a variable-length to fixed-length encoding scheme.

The major problem is how to divide up the original sequence so that fixed-length codewords can provide a compression of the text. The next problem is how to construct the dictionary, based on which segments of the text can be coded. Each dictionary element will be coded with an equal-length codeword, even though the segments can be of different lengths. Tunstall [351] introduced a method to construct the dictionary \( D = \{d_1, d_2, \ldots, d_D\} \) by generating a tree (now called a Tunstall tree) whose leaf nodes correspond to the dictionary strings (the partitions needed for the original text). Let \( \Sigma \) be the source alphabet,
Fig. 5.3 Performance comparison of algorithms for pattern matching on BWT-compressed text. (a) Mean number of comparisons by pattern length for “bible.txt”; (b) Search times for multiple patterns; (c) Times for just counting occurrences of multiple patterns. Figures taken from [3].
with $|\Sigma|$ symbols. Let $D = |\mathcal{D}|$ be the number of unique codewords for the Tunstall code. The procedure is as follows:

1. Start with the root node, and treat this as an intermediate node. Grow $|\Sigma|$ leaves on the root node (each edge from the root to a leaf corresponds to one source symbol).
2. If the number of leaf nodes is $\geq D$, goto step (4), else goto step (3).
3. Select the leaf node with the highest probability, treat this as an intermediate node, and grow $|\Sigma|$ leaves from this node; goto step (2).
4. Assign fixed $k$-length codewords to each leaf node. The path label from the root to the $i$th leaf node specifies $d_i$, the $i$th entry in $\mathcal{D}$. Given $k$, the fixed length of the codewords, the number of codewords is constrained to be $D = |\mathcal{D}| \leq 2^k$.

Figure 5.4 shows an example Tunstall tree for three different values of $k$.

(a) (b) (c)

Fig. 5.4 Corresponding Tunstall tree for the source with four symbols used in Figure 5.2 with symbol probabilities: $p(a) = 0.13, p(b) = 0.44, p(c) = 0.21, p(d) = 0.22$. Trees are shown for different values of $k$, the fixed codeword length: (a) $k = 2$; (b) $k = 3$; (c) $k = 4$. Codewords are assigned by using $k$-bit codes for each leaf node. Note that we have omitted some leaf probabilities for clarity of presentation.
The optimality and theoretical performance of Tunstall codes and its variants are studied in [297]. More recently, Tunstall codes have become the focus of attention given their utility in efficient compressed text retrieval [190, 386]. Suffix tree based variations for constructing Tunstall codes have been proposed [189, 353], showing significant improvement in compression, at times comparable with state-of-the-art text compression methods (see also [354, 387]). The dictionary generated as part of the encoding process provides an excellent basis for later pattern matching on the compressed data. This has been exploited by recent methods for text retrieval on variable-to-fixed length codes [190, 386].

5.7.1 Searching on Tunstall Codes

The original motivation for the variable-to-fixed length encoding methods such as Tunstall codes was their very fast decoding capability. More recently, there is a growing interest in their use for compressed pattern matching [45, 190, 387]. The fixed length of their codewords imply that we already know the code boundaries, and hence can have an easy random access to any given position in the compressed stream. Further, since we have the dictionary available in the compressed bit stream, we can perform simple prefix matching of the dictionary elements on the pattern $P$. This could produce an initial list of positions where a match could occur, which can then be be partially decompressed for verification. Depending on the pattern, this might identify some complete matches of $P$ in the text, without needing the partial decompression. Like other dictionary-based methods, another limited form of pattern matching can be performed by using only the dictionary (which is sent as part of the compressed data), without touching the compressed sequence itself. That is, we can answer existential queries about the pattern in the compressed text by solving the string-to-dictionary matching problem: Given a dictionary $D$, and a pattern $P$, find all the occurrences of $P$ in any string which can be formed from a concatenation of the elements of $D$. The concatenation of elements can be in any order, allowing possible repetitions. Extending the results to answer enumerative queries will require further processing. In [190],
a method was proposed for the string-to-dictionary matching problem on Tunstall compressed text, based on suffix trees. Then to verify all matches and enumerate the location of the matches, they used multiple pattern matching and Sunday’s variation \([155]\) of the Boyer–Moore algorithm \([60]\). Interestingly, fully compressed pattern matching (even where both \(P_c = C(P)\) and \(T_c = C(T)\) are based on the same dictionary) is still a challenge with Tunstall codes, since there could be different ways to partition \(P\) and \(T\) into codewords.

### 5.8 Others

A dictionary-related compression method based on an anti-dictionary (the words that do not appear in the text) has been proposed \([98]\). In \([309]\), this idea was used to develop an algorithm that preprocesses a pattern of length \(m\) and an anti-dictionary of size \(a\) in \(O(m^2 + a)\) time and then determines all occurrences of the pattern by a linear scan of the compressed text of length \(n_c\). The LZ family of algorithms have also been used in the context of PMIC for approximate pattern matching for lossy compression (See Section 6.3.7). Manber \([233]\) has described a compression system that allows for approximate pattern matching in the presence of errors, although the compression is lossless. More recently, Brisaboa et al. \([64]\) proposed methods for variable-length to variable-length codes that can support search directly on the compressed data, using words and phrases as the basic elements. In another related paper \([65]\), they considered methods for dynamic compression of natural language texts, and compressed text retrieval under such dynamic compression schemes.

The PPM compression methods use a trie-like data structure to search for the longest context in the portion of the text already encoded. This structure is also built and maintained by the decoder, which raises the possibility of using the structure for compressed domain searching. If arithmetic coding is used, the decoding has to be completed in full but the decoded output can be ignored. On the other hand, if Huffman coding is used (to the detriment of compression) or the contexts are stored in the trie corresponding to word boundaries, there is the possibility of doing a direct search on the compressed data. Further, the
<table>
<thead>
<tr>
<th>Compression method</th>
<th>Search strategy</th>
<th>Approximate match</th>
<th>Time complexity</th>
<th>Space complexity</th>
<th>Refs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>naïve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>RLE</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>2</td>
<td>RLE</td>
<td></td>
<td></td>
<td></td>
<td>[67, 68]</td>
</tr>
<tr>
<td>3</td>
<td>LZ77</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>4</td>
<td>LZ77</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>5</td>
<td>LZ77</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>6</td>
<td>LZ78, LZW</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>7</td>
<td>KMP</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>8</td>
<td>LCS</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>9</td>
<td>SLP</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>10</td>
<td>KMP</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>11</td>
<td>LZ78</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>12</td>
<td>LZW</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>13</td>
<td>LZ77, LZW</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>14</td>
<td>word Huffman</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
<tr>
<td>15</td>
<td>LZ78, LZW</td>
<td></td>
<td></td>
<td></td>
<td>[15]</td>
</tr>
</tbody>
</table>

Continued
<table>
<thead>
<tr>
<th>s.n.</th>
<th>Compression method</th>
<th>Search strategy</th>
<th>Year</th>
<th>Approx. match</th>
<th>Time complexity</th>
<th>Space complexity</th>
<th>Refs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>LZ78, LZW</td>
<td>BM</td>
<td>2000</td>
<td></td>
<td>$O(n_c), O(m n)$ w.c.</td>
<td>$O(n_c + r)$</td>
<td>[262]</td>
</tr>
<tr>
<td>17</td>
<td>gen. dictionary</td>
<td>BM</td>
<td>2000</td>
<td></td>
<td>$O(f(d) d + n_c) + n_c, m + m^2 + r$</td>
<td>$O(d + m^2)$</td>
<td>[308]</td>
</tr>
<tr>
<td>18</td>
<td>LZW</td>
<td></td>
<td>2000</td>
<td></td>
<td>$O(n_c + M^2 + t + r)$</td>
<td>$O(M^2 + t)$</td>
<td>[182]</td>
</tr>
<tr>
<td>19</td>
<td>LZ77, LZ78</td>
<td></td>
<td>2003</td>
<td></td>
<td>$O(n_c, \log(\frac{m}{m_c}) + m^2)$</td>
<td>$O(n_c, \log(\frac{m}{m_c}) + m^2)$</td>
<td>[180]</td>
</tr>
<tr>
<td>20</td>
<td>MPM</td>
<td></td>
<td>2004</td>
<td></td>
<td>$O(m_c n_c^2)$</td>
<td>$O(n_c m_c)$</td>
<td>[162]</td>
</tr>
<tr>
<td>21</td>
<td>LZW</td>
<td></td>
<td>2005</td>
<td></td>
<td>$O(n_c + M t + r)$</td>
<td>$O(M t)$</td>
<td>[338]</td>
</tr>
<tr>
<td>22</td>
<td>SLP</td>
<td></td>
<td>2007</td>
<td></td>
<td>$O(n_c^2 m_c)$</td>
<td>$O(n_c m_c)$</td>
<td>[215]</td>
</tr>
<tr>
<td>23</td>
<td>LZW</td>
<td>KMP, ST</td>
<td>2011</td>
<td></td>
<td>$O(n_c + m)$</td>
<td>$O(n_c + m)$</td>
<td>[127]</td>
</tr>
<tr>
<td>24</td>
<td>LZ77, LZ78</td>
<td></td>
<td>2011</td>
<td></td>
<td>$O(n_c, \log(\frac{m}{m_c}) + m)$</td>
<td>$O(n_c, \log(\frac{m}{m_c}) + m)$</td>
<td>[128]</td>
</tr>
<tr>
<td>25</td>
<td>LZW</td>
<td>LCA, ST</td>
<td>2012</td>
<td></td>
<td>$O(n_c + m_c)$</td>
<td>$O(n_c + m_c)$</td>
<td>[129]</td>
</tr>
<tr>
<td>26</td>
<td>SLP</td>
<td>recompression</td>
<td>2012</td>
<td></td>
<td>$O((n_c + m_c) \log m (\log(n_c + m_c))$</td>
<td>$O((n_c + m_c) \log(n_c + m_c))$</td>
<td>[167]</td>
</tr>
<tr>
<td>27</td>
<td>Tunstal</td>
<td>p-matching</td>
<td>2013</td>
<td></td>
<td>$O(n + m)$</td>
<td>$O(m)$</td>
<td>[45]</td>
</tr>
</tbody>
</table>

Key: $P =$ original pattern, $P_c =$ compressed pattern, $T =$ original text, $T_c =$ compressed text, $a =$ size of antidictionary, $d =$ size of dictionary, $n =$ $|T| =$ length of uncompressed text, $n_c =$ $|T_c| =$ length of compressed text, $m =$ $|P|$ = length of uncompressed pattern, $m_c =$ $|P_c| =$ length of compressed pattern, $k =$ number of differences allowed, $\Sigma =$ alphabet size, $r =$ number of occurences of pattern in text, $t =$ size of LZW tree, $M =$ total length for multiple patterns ($M = m$ for single pattern), $f(d)$ = function of the dictionary, depends on the tokens in $d$, d.p.: dynamic programming, LCS: longest common subsequence, MPM: multilevel pattern matching, ST: suffix tree, p-matching: parameterized pattern matching, LCA: longest common ancestors.
trie could be stored in memory after decoding to use as an index to perform multiple pattern matching operations on the text. If the file is compressed using the DMC algorithm, as with PPM, there may be a possibility of using the finite state machine data structure to search for patterns after the whole text has been decoded.

Table 5.7 shows the theoretical performance for various proposed algorithms for pattern matching on compressed text. Further results and challenges in lossless compression can be found in a special issue of the Proceedings of the IEEE, edited by Storer [325], and two recent special issues of Algorithms, one edited by Solomon [295], the other by Carpentieri [77]. See also the review paper on textual data compression and its use in biological sequence analysis [137].
In this section, we consider methods proposed for searching compressed images. We start with methods for lossless compressed images, which tend to be extensions of methods used for text pattern matching, such as methods for compressed 2D pattern matching. We then turn our attention to methods for compressed search on lossy compressed images. These tend to be more focused on approximate matches than exact matches. Given the nature of images, the notion of approximate matches in this context is quite different from traditional approximate matches such as the $k$-mismatch or $k$-approximate matches used in text pattern matching.

As with text, compression for images also tries to store the image using a smaller space by eliminating some redundancies in the image. For instance, there is always the problem of coding redundancy due to the basic scheme used to describe the pixels, such as the usual color or gray-level representation. Since the image represents a physical surface, neighboring points on the image are usually similar. Images therefore exhibit some level of spatial redundancy, which often appears in the form of correlation or similarity between nearby pixels. The image could also contain very fine details that a human is not likely to observe, due to the limitations of the human visual system. This
is sometimes called *psycho-visual redundancy* [142]. The image itself might also exhibit a limited form of self-similarity, where some part of an image could be similar to some other parts in some way. Image compression schemes aim to expose and then remove one or more of these types of redundancies.

We should note that the decorrelation of signals and removal of redundancy as described above are not the only goals in image compression. Depending on the type of compression, there could be several other goals, for instance, removing linear or nonlinear dependencies. In particular, various lossy compression algorithms have aimed at exploiting the ability of certain transformations to compact the energy in a signal into only a few transform coefficients.

For lossy compression, the reconstructed image is usually not exactly the same as the original image. Measures are therefore required to indicate the quality of the compression. Both qualitative and quantitative measures can be used. Qualitative measures are based on subjective human evaluation of the reconstructed image. Quantitative measures usually depend on some measure of distortion or similarity between the original image (before compression) and the reconstructed image (after decompression). One approach is using the general Minkowski ($\ell^p$) distance: Given two images, $I_1$ and $I_2$, the general Minkowski distance between them is given by:

$$\ell^p(I_1, I_2) = \left[ \sum_{xy} |I_1(x, y) - I_2(x, y)|^p \right]^{\frac{1}{p}}, \quad (6.1)$$

where $p \geq 1$, and $I_j(x, y)$ denotes the pixel value at position $(x, y)$ in image $I_j$. With $p = 1$, we have the simple city block distance, $p = 2$ gives the Euclidean distance.

A more popular approach is a squared-error distortion measure, such as the *mean square error* (MSE) and the *peak signal-to-noise ratio* (PSNR) (see Section 4). Lossy compression algorithms must find a balance between the rate (compression ratio — usually expressed as bit rate, the average number of bits per symbol) and quality or distortion. This balance is often captured in terms of the rate-distortion function, $R(D)$. Essentially, given a distortion measure, $R(D)$ indicates the lowest rate at which the source data can be compressed while still
maintaining a maximum allowed distortion of $D$. Clearly, at $D = 0$ (no distortion, implying lossless coding), $R(D)$ will be the entropy of the source, assuming the distortion measure is a metric.

6.1 Searching Lossless Compressed Images

In lossless image compression the image can be recovered exactly from the compressed version. The compression achieved is usually modest compared with lossy image compression. However, lossless compression is important in various signal processing applications, especially those involving medical images.

Initial efforts on compressed pattern matching on images were mainly based on extending the methods that have worked for 1D sequences to 2D sequences (images). Most of these were based on run-length encodings and the LZ family. Although LZ was the basis of some earlier lossless image compression formats, such as GIF and PNG, direct application of the LZ algorithms do not exploit all the characteristics of natural images, and hence may not always lead to effective compression. A few methods have been proposed for limited search on context-based predictive coding methods, such as the standard JPEG and JPEG-LS algorithms. Although none of the methods have been adopted by the mainstream image analysis or multimedia information systems communities, it is still important to review what has been done so far. This helps identify the key challenges in this area, and hence sets the stage for further work. Some recently introduced approaches to lossless image compression, such as the BWT-based lossless image coding [4] could be a stepping stone toward compressed pattern matching for images, where the compression scheme is competitive with state-of-the-art lossless coding schemes.

6.1.1 Run Length Encoding for Images

Run-length encoding simply involves replacing consecutive occurrences of the same symbol with a code that indicates what the symbol is, and how many times it has been repeated. Although this can be used to compress text, it produces better results when used on images. Image compression is achieved by exploiting the spatial redundancy by using
run-length pairs, for instance, along a given path in the image. It is particularly suitable for monochrome images; for example, a scanned page tends to contain large runs of white pixels that can be represented as just one run.

A variant of run-length coding is used in the CCITT fax standards [156], which are widely used for fax machines. The runs are assumed to alternate between black and white, and the length of runs are encoded using a static Huffman code that was designed from some sample documents. Such codes can be used directly for compressed-domain searching because it is a lot faster to compare the length of runs than to compare the individual pixels in a run.

There are also two-dimensional extensions of the basic RLE scheme. For example, for binary images (such as faxes), the relative address coding method [142] tracks the binary transitions that begin and end each black or white run. The distances between different types of transitions on different rows are calculated and then coded using a variable length code. This often requires the adoption of a convention to determine the run values. In general, 2D RLE coding involves an initial step of converting the 2D image data to a 1D sequence. The conversion could be based on simple horizontal (row-wise) scans, vertical scans, zig–zag scan, or other more complicated scans, such as the Hilbert scan, or Peano scan. The 1D run-length encoding is then applied to the transformed sequence to achieve compression.

RLE is a lossless compression scheme. The output from RLE can be compressed further by passing it as the input to a variable-length coding scheme, such as Huffman coding. Most practical image/video compression schemes such as JPEG and MPEG use RLE (and other forms of Huffman coding) during the encoding stage. Some text compression schemes, such as the BWT [3, 70] can also include run length encoding as a step in the overall compression pipeline.

6.1.2 Searching on RLE-coded Images

The first few methods of pattern matching in lossless compressed images were based on run-length encoding (RLE). This was primarily motivated by the need to search fax images, since the standard
compression scheme used for fax transmission is based on 2D run length encoding. Using the basic method described in Section 5.4, and the notion of 2D periodicity analysis, Amir and Benson [15] proposed an algorithm for the 2D run-length pattern matching problem. Here, for a 2D text (or image) $T$, the 2D run-length encoding, denoted 2D-RLE, is simply defined as the concatenation of run-length encodings from each row of $T$. This is similar to case (a) in Figure 6.1 of Section 6.1.3. The algorithm is almost optimal, requiring a running time in $O(n_c \log m + m)$. This was further improved to $O(n_c + m_c)$ optimal time in [17]. Space efficient variations of the algorithm were proposed in [22], requiring $O(n_c + m \log(\min\{m,|\Sigma|\}))$ time and $O(m_c)$ space.

Maa [231] considers a special case where the pattern to be located is a bar-code. Maa observes that for the CCITT fax standard, which uses both vertical and horizontal run-length coding, bar-codes create distinctive coding patterns, and can be detected reliably. It may be possible to extend this idea to other types of images; for example, half tone images will compress very poorly using run-length coding; text will have many short runs; and line drawings will have many long runs of the same color.

### 6.1.3 Dictionary-based Image Coding

The initial success of the LZ-family in text compression motivated the use of LZ and similar dictionary-based approaches on other types of data, such as images. Like the case of 2D RLE above, the basic approach involves an initial step of conversion from 2D to 1D, and the application of a text compression algorithm on the 1D sequences. This approach was originally proposed by Lempel and Ziv [208] (the same authors of the original papers on the LZ method), whereby compression of the resulting 1D sequence was performed using the LZ algorithm. Some popular lossless image coding programs (such as GIF, TIFF, and PNG) are all based on the LZ algorithm, after a raster scan of the image. These work well for images with a limited number of colors (such as an icon), but are unlikely to find runs in more complex images such as photographs, where two adjacent pixels are unlikely to have exactly the same value.
Different dictionary-based image compression methods differ in the way they perform the 2D-to-1D conversion. For instance, the Hilbert scan was proposed in [208], while the snake scan was suggested in [21, 290]. Pajarola and Widmayer 1996 [272] earlier considered scanning and compressing each row independently. The different models of dictionary-based image coding are shown in Figure 6.1. The question of which scanning method to use was laid to rest by Memon et al. [238], who showed that, with respect to data compaction, no single scanning method outperforms all the others on most practical images. Yet when the objective is to support later searching on the compressed image, it is clear that certain scanning methods, such as the simple row-wise or column-wise scans, may be easier to use.

A variation of the dictionary-based method is the family of image compression methods that use 2D-block matching (see case (c) in Figure 6.1). Here, the initial search required during the compression process is performed by matching 2D blocks of the image being compressed. The objective is to capture the 2D spatial correlation often observed in natural images, which may not be effectively captured
using the 2D to 1D approaches. Originally introduced by Storer et al. [287, 326] based on exact matching of 2D image blocks, this method has been extended to support inexact matches of the image blocks [66], since exact matches for image blocks may be difficult for increasing block sizes. For lossless image compression, such approaches that use inexact matching must provide a feedback mechanism to correct any errors when a matching block is replaced with a similar block (rather than an exact match). The grammar-based multilevel pattern matching approach has also been used for direct 2D dictionary matching for lossless coding of images [168].

Although the LZ-based image coding methods or general dictionary-based image compression schemes do not necessarily produce the best results in terms of compaction ability, they support relatively fast compression and decompression. Perhaps, more significantly, most methods for pattern matching on lossless compressed images have focused on the dictionary-based approaches. This is not completely unexpected, given the relative success of compressed pattern matching on LZ-compressed text.

### 6.1.4 Searching on Dictionary-compressed Images

From Figure 6.1, different basic models can be used to scan the image to form the compressed 2D-LZ representation, namely: (a) apply the LZ on each row or column, then concatenate the LZ sequences; (b) concatenate rows or columns to form one large string, and apply LZ once on this large string; (c) use search for repeating 2D patterns, and perform pattern substitutions based on the 2D patterns. For models (a) and (b), methods proposed for LZ compressed 1D sequences could be applied, with appropriate modifications. Rytter [290] described a method that uses model (b) using the Hilbert scan as the basic method for 2D to 1D conversion. Amir et al. [21] proposed a method that performs compressed matching on LZ-compressed sequences generated using an LZ78 model (b), by simply concatenating rows of the image. Their algorithm requires \(O(m\sqrt{m} + n)\) time and \(O(m)\) space, where the size (dimension) of \(T\) is \(n_1 \times n_2 = n\), and the size (dimension) of \(P\) is \(m_1 \times m_2 = m\). In principle it should be possible to use the repeating
2D patterns which form the dictionary when using model (c) above, for compressed pattern matching. However, there is not much reported on their use for compressed pattern matching on lossless compressed images.

The general case of pattern matching for a class of “highly compressed” two-dimensional texts is explored by [54, 55] for both cases of fully compressed and non-fully compressed pattern matching.

6.1.5 Huffman Coding for Images

Pajarola and Widmayer [273] considered the use of Huffman codes for 2D compression, with the aim of supporting later search. Using model (a), each row was scanned and compressed using Huffman codes. The concatenation of the compressed rows then formed the compressed image. To search for an image, \( P \), they applied the same procedure to each row of \( P \), using the same set of Huffman codes. Then they considered the problem as a 1D multiple pattern matching problem, where each compressed row from \( P \) was treated as a pattern. Thus, multiple pattern matching was applied to search for the set of compressed rows from \( P \) in \( T \), and the results are later verified. Their method required time and space in \( O(n_c + m_c + r_k) \) and \( O(m_c + n) \), respectively, where \( r_k \) is the total number of occurrences of any row, say row \( i \) of the pattern image \( P \) in any row of \( T \).

6.1.6 Context-based Predictive Image Coding

The dictionary-based methods are conceptually simple and generally fast. However, with respect to data compaction ability, the most successful lossless image compression algorithms are generally context-based and they exploit the 2-D spatial redundancy in natural images. Examples include LJPG (lossless JPEG), FELICS [151], CALIC [239], JPEG-LS [370], TMW [240], SPIHT [292], and PPAM [394]. These methods usually involve four basic components:

1. an initial prediction scheme to remove the spatial redundancy between neighboring pixels;
2. a context selection strategy for a given position in the image;
3. a modeling method for the estimation of the conditional probability distribution of the prediction error given the context in which it occurs; and
4. a final encoding step based on the estimated conditional probabilities.

Different context-based lossless image compression schemes vary in the details of one or more of the basic components.

The primary motivation for context-based approaches is the promise of improved compression. Consider an image of size $N_1 \times N_2$. Let the image be represented by a sequence $S = \{s_i, i = 1, 2, \ldots, |S|\}$, with symbols taken from a fixed alphabet $\Sigma = \{\sigma_i, i = 1, 2, \ldots, |\Sigma|\}$, where $|S| = N_1 \times N_2$ is the image size. The symbol alphabet $\Sigma$ is typically the set of distinct pixel gray levels in the image, or the set of distinct prediction errors, after applying a prediction scheme. Let the corresponding probability of the symbols in the image be $p(\sigma_i), i = 1, 2, \ldots, |\Sigma|$, with $\sum_i p(\sigma_i) = 1$. Assuming we code one pixel at a time, then the average number of bits per symbol required to encode the image without context modeling is no less than the first-order entropy:

$$H(S) = -\sum_{\sigma_i \in \Sigma} p(\sigma_i) \log_2 p(\sigma_i). \quad (6.2)$$

If contexts are considered, the conditional probability distribution for the set of symbols $S_j$ with the context $C_j$ will be $p(s_i|C_j), i = 1, 2, \ldots, |S_j|$. Then, the average number of bits per symbol needed to encode the image will be no less than the conditional entropy:

$$H(S|C) = -\sum_{j=1}^{M} p(C_j) \sum_{s_i \in S_j} \log_2 (p(s_i|C_j))$$

$$= -\sum_j \sum_{s_i} p(s_i, C_j) \log_2 (p(s_i|C_j)),$$  \quad (6.3)

where $M$ is the total number of contexts. Since conditional entropy is never greater than the unconstrained entropy [94], the number of bits/symbol needed to encode the image is also reduced using context modeling: $H(S|C) \leq H(S)$. 

Another way for improved compaction will be to consider a simple extension of the source (i.e., coding blocks of symbol at a time, rather than individual symbols). The entropy of the $k$th extension (or block entropy) is given by:

$$H(S^k) = - \sum_{\sigma^k_i \in \Sigma^k} p(\sigma^k_i) \log_2 p(\sigma^k_i).$$

(6.4)

where $\sigma^k_i$ is the $i$th symbol in $\Sigma^k$, the extended alphabet. The $k$th order entropy is then given by $H^{(k)}(S) = \frac{1}{k} H(S^k)$. In general, the $k$th order entropy is a monotonically decreasing function of $k$: $H^{(k+1)}(S) \leq H^{(k)}(S), \forall k > 0$. This leads to the notion of entropy rate, which represents the best achievable lossless data compression for a given source:

$$h(S) = \lim_{k \to \infty} H^{(k)}(S) = \lim_{k \to \infty} \frac{1}{k} H(S^k).$$

(6.5)

Correlation between pixels implies possible predictability of the pixel values using information from some other pixels. Spatial redundancy can be exploited by representing the image in terms of pixel differences, rather than explicit pixel values. Predictive lossless coding then codes only the new information using a symbol-encoding method. Practically, the new information (also called prediction error or residual error) is the difference between the actual and the predicted value of the pixel. The major difference between the different methods of predictive coding is in how they determine the prediction error, for instance, the size of the neighborhoods (i.e., prediction context) they consider, or the weights they assign to the neighbors, based for example on their distance from the pixel under consideration. Example popular image prediction schemes range from the simple autoregressive models that use a weighted combination of the pixel values within a prediction context and the median-edge detection (MED) predictor used in JPEG-LS [370], to the relatively more complicated gradient-adjusted prediction (GAP) used in CALIC [239], and the edge-directed prediction (EDP) [213].

Considering the data compression model of Figure 1.1, image prediction can be seen as a form of transformation on the image data, from the original pixel values to prediction errors, or residues. For natural
images, the resulting residues often have a highly skewed distribution, resulting in improved compaction. The amount of compression achievable with predictive coding thus depends on how skewed the distribution of the error residuals is, when compared with that of the original pixel values. Because there is usually a significant amount of inter-pixel redundancy in the image, this often leads to significant compression, with no loss. When lossy compression is acceptable, further compression can be achieved by quantizing the prediction error before it is coded, or by passing the prediction residues through a further transformation stage, such as the discrete cosine transform (DCT) before quantization. This reduces the psycho-visual redundancy, but also reduces the accuracy of the representation, and hence introduces some error in the compression process. Generally, with more quantization we achieve more compression, but we also introduce more error. The amount of acceptable error is usually application dependent, and can be specified using the rate distortion function.

The prediction of bi-level images has been explored by several authors [202, 245]. Predictive compression is used in the JBIG international facsimile standard [78]. The predictions are coded using an arithmetic coder, which does not bode well for compressed domain searching. However, as with PPM, it may be possible to exploit prediction information for searching. Also, JBIG includes a resolution-reduction method [388] that is used to transmit low quality versions of the image that can then be improved as more data is transmitted. This raises the possibility of searching the low resolution images for approximate matches, and then decoding the full image if it looks promising. Similarly, JPEG-LS [370], the international standard for lossless image compression, is a context-based predictive scheme.

In general, for context-based predictive models the prediction and context modeling stages are the key to improved data compaction. However, these two stages also create the most difficult challenges for direct pattern matching on the compressed data. Different sequences of original pixel values could generate the same prediction error sequence. On the other hand, both the modeling and prediction depend on the context, which varies in the image. The same pixel value could generate a different prediction error depending on its neighboring context.
Thus, typically, compressed matching on images compressed with these approaches will need to perform an initial preprocessing to undo the effect of these stages before pattern matching can be performed. (See for instance, [339] for searching on JPEG-LS, and [187] for Baseline JPEG). More recently, it was shown that the BWT, which was thought to be inappropriate for image compression, could in fact be effective in lossless image coding [4]. This requires a modification of the basic BWT compression pipeline, or changes in the encoding stage by exploiting the sorted context partitions induced by the BWT. Given the success of compressed pattern matching on BWT compressed text [3, 48, 115], it may be possible to exploit similar methods for compressed matching on BWT-compressed images.

6.1.7 Searching Lossless Predictive-coded Images

Few methods have been proposed for compressed matching on lossless image compression schemes that are based on predictive coding. Yet these tend to produce the best compression results, and form the basis of the most popular lossless image compression standards, such as lossless JPEG and JPEG-LS. The major difficulty is that the prediction stage, which is the primary source of redundancy reduction, is usually based on local contexts at a given pixel position in the image. Such contexts will vary from one position to the other, and hence the prediction error sequences which is what is sent to the encoder could vary dramatically between two image blocks that may be quite similar, but occur in locations with different surrounding contexts in T. Mukherjee et al. [337, 339] attempted to address this problem by proposing a modification of the JPEG-LS standard. They proposed a two-stage variation of the JPEG-LS compression method. In the first stage they perform a scan of the image, and compute information about the contexts for each position. This stage does not perform any compression. In the second stage, the image is scanned again in the same order as before, using the pre-computed contexts (from the first stage) to refine the prediction at a given pixel location, and then compress the pixel. The context data computed is then stored as part of the compressed bit-stream, along with the usual image data.
The context-data essentially provides a data structure that makes JPEG-LS search-aware. For pattern matching, the given query image $P$ is compressed using the same context used to compress the database image. By doing this, identical patterns will thus have identical contexts, hence the bit-stream from the compressed pattern will have the same binary representation as its match in $T$ (if there is a match), except for the first row and first column in $P$. The matching regions can then be verified to check if they are true matches. This approach provides search awareness to JPEG-LS, at a cost of potentially reduced compression (to store the context data). In [339], the authors discuss improvements to the method, leading to less loss in compression. Klein and Shapira [187] consider compressed pattern matching on Baseline JPEG, which is based on Huffman codes. They also introduced an additional index structure that stores information about the DC positions in the compressed bit stream, based on which they could perform pattern matching on the compressed data.

While several methods have been developed for compressed-domain or transform-domain searching on lossy compressed images, there is relatively little that has been done on the more effective lossless image compression schemes, especially the class of context-based predictive methods, which are known to produce superior results in lossless image compression. The methods described in this section can be seen as initial approaches that have shown that pattern matching on such schemes is possible, though with the need for auxiliary data. Further work is needed on this problem.

Table 6.1 shows the theoretical performance for various proposed algorithms for pattern matching on lossless compressed images.

6.2 Searching Compressed Document Images

The methods for RLE-compressed sequences can be applied for pattern matching on compressed data, such as documents compressed using the CCITT fax standard. Other file formats, such as the portable document format (PDF) can also use the CCITT Group 4 compression format. CCITT Group 3/4 are also used as the basis for compressed representation of document images, such as scanned texts. Searching
<table>
<thead>
<tr>
<th>s.n.</th>
<th>Compression method</th>
<th>Search strategy</th>
<th>Year</th>
<th>Exact match</th>
<th>Time complexity</th>
<th>Space complexity</th>
<th>Refs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RLE</td>
<td>2D-RLE</td>
<td>1992</td>
<td>✓</td>
<td>$O(n_c \log m + m)$</td>
<td>$o(n_c^2)$</td>
<td>[15]</td>
</tr>
<tr>
<td>2</td>
<td>Huffman codes</td>
<td>KMP Multiple PM</td>
<td>1996</td>
<td>✓</td>
<td>$O(n_c^2 + r_k + m_c^2)$</td>
<td>$O(m_c^2 + n)$</td>
<td>[272]</td>
</tr>
<tr>
<td>3</td>
<td>RLE</td>
<td>In place 2D</td>
<td>2003</td>
<td>✓</td>
<td>$O(m^3 + n^2)$</td>
<td>$O(m)$</td>
<td>[17]</td>
</tr>
<tr>
<td>4</td>
<td>LZ78</td>
<td>(2D to 1D) extra space</td>
<td>2003</td>
<td>✓</td>
<td>$O(n_c + m \log (\min{m_c, n}))$</td>
<td>$O(m_c)$</td>
<td>[22]</td>
</tr>
<tr>
<td>5</td>
<td>RLE</td>
<td>2D-RLE</td>
<td>2003</td>
<td>✓</td>
<td>$O(n_c + m \log (\min{m_c, n}))$</td>
<td>$O(m_c)$ extra space</td>
<td>[339]</td>
</tr>
<tr>
<td>6</td>
<td>JPEG-LS</td>
<td>partial decoding</td>
<td>2004</td>
<td>✓</td>
<td>$O(r_i \times m + mn)$</td>
<td>$O(m^2 n_c \log n_c)$</td>
<td>[186]</td>
</tr>
<tr>
<td>7**</td>
<td>Baseline</td>
<td>aux. index partial encoding</td>
<td>2005</td>
<td>✓</td>
<td>$O(r_i \times m + mn)$</td>
<td>$O(m^2 n_c \log n_c)$</td>
<td>[186]</td>
</tr>
</tbody>
</table>

**Supports only approximate matching, since the basic compression scheme is lossy. Key: $P$ = uncompressed query image pattern of size $m \times m$, $P_c$ = compressed image, $T$ = uncompressed database image of size $n \times n$, $T_c$ = compressed database image, $d = \text{spacing between indexed blocks of size } b \times b$, $|T_c| = n_c \times n_c = \text{length of compressed database image}$, $|P_c| = m_c \times m_c = \text{length of compressed query image}$, $r_k = \text{number of occurrences of any row, say } P_i \text{ of } P$, in a row of $T$; thus we have $r \leq r_k$, where $r = \text{number of occurrences of image query pattern } P \text{ in database image } T$. 


on such document images has become a major issue, given the amount of old text collections that are available only as scanned data. One approach to searching such images is to first apply OCR (optical character recognition) to the scanned documents, and then use traditional text retrieval methods. Current OCR technology, however, still has a lot of errors, which will have a direct impact on retrieval. Manual correction of OCR errors is very laborious and prone to further human errors.

Methods have thus been proposed for document analysis and retrieval directly on the document images, without the need for OCR — see for example [103, 228, 255]. Some methods have also been proposed for performing the required analysis directly on compressed document images. Hull and Lee [154, 205] studied duplication detection on document images compressed with CCITT. Imura and Tanaka [161] proposed a substitution-based compression scheme for document images, and showed how to perform string matching on the compressed data.

Lu and Tan [229] proposed an approach based on the three coding modes used in CCITT Group 4 document image compression. From the compressed document images they extracted feature pixels that correspond to the changing elements. A changing element is basically an element whose color is different from the previous element along the same line. Here, the color of a pixel is either black or white. Based on the detected changing elements, they identify connected components in the image. The connected components tend to correspond to individual characters or symbols in the document image. Using these, word objects are identified by combining nearby connected components (considering their sizes and relative positions) to determine word boundaries. These are further classified into word object groups. Each compressed document is then represented as a feature vector constructed based on the occurrence frequency of its word object groups. Similarity matching between two document images is then performed using standard distance measures, such as the Hausdorff distance [157], or the cosine distance between the vectors. It is simple to extend the approach to search for one document image in a database of such images, or to perform region-based matching on the document images.
6.3 Searching on Lossy Compressed Images

Most methods that perform compressed domain image search/retrieval, (or more generally, compressed domain image analysis) do so on lossy compressed images. If the data being searched has been compressed with a lossy method then the pattern matching needs to be approximate to accommodate the possibility that the target of the search may have been changed slightly by the compression process. Although lossy compression can be used for different types of multimedia data (such as video, images and audio), here our emphasis is on images. In general, approximate pattern matching for images has been studied in the context of content-based image retrieval. Content-based retrieval is concerned with retrieving images based on their true (visual) content, rather than by a textual description of such content.

Like the usual text pattern matching problem, the image retrieval problem can be stated as follows: given a database of images, and a query image, find all the images in the database that are similar to the query image. Object-level image retrieval is a variant of this, where the objective includes finding all the database images that contain image subparts that are similar to the query image. In general, the image retrieval problem is mainly concerned with determining the following:

- **The features to be used:** Because of the subjective (visual) nature of image contents, different features of the image (such as color, shape, texture, spatial information, etc.) could be used in matching images. The appropriate features to be used will depend on the particular application.
- **The representations to be used:** When we know the features to be used, we will need to find appropriate representations for the features that will be suitable for retrieval. Chang et al. [83] provides an overview of techniques for finding images in large archives. They point out the importance of representing the information in a form suitable for searching. Different features require different representations, and at times, the same feature can be represented in different ways. The representation also needs to accommodate various...
possible transformations of the image, such as simple affine transformations.

• The similarity criteria: The choice of similarity criteria will depend on the particular feature that is being used. Sometimes, a combination of the features (and hence various similarity metrics) will be required for effective matching of the images. Tverski [352] discussed general issues in evaluating similarity between objects. Vasconcelos and Lippman [357, 358, 360] propose probabilistic models for image similarity comparison. Adjeroh and Lee [8] proposed methods for efficient image similarity evaluation.

For each feature, appropriate methods are required for their extraction, efficient representation, and proper similarity matching. Surveys on image retrieval can be found in [100, 315]. There are also methods for shape matching [336], texture analysis [104, 172], and concept-based retrieval [319]. An empirical comparison of the various features used in image retrieval is presented in [101]. More recent work on the use of compression in content-based image retrieval can be found in [216, 217].

Because of the promise of efficiency in compressed domain operations, and since images are now typically stored in compressed form, compressed domain image retrieval has attracted considerable attention and various methods have been proposed [254, 343]. The general problem of image matching involves some form of image processing and analysis. Schmalz [300] presents a general introduction to processing compressed data. He also provides an overview of the problem of recognizing patterns in compressed data [302] and gives optical processing methods for the problem [301].

Below we use the common lossy compression methodologies (such as block-transform, wavelets, and vector quantization) as a guide and provide a brief survey of the reported work on compressed pattern matching on lossy-encoded images.

6.3.1 The Quadtree Representation

The quadtree representation [296] relies on the fact that natural images tend to contain large areas of uniform color or gray-levels. A quadtree
represents an image as a tree, whereby each node corresponds to some square portion of the image. The input image (the root) is divided into four quadrants. Each quadrant becomes a child of the root. If the quadrant is homogeneous, i.e., all of its pixels are identical, the quadrant is represented as a single color, and saved as a leaf node. Otherwise, the quadrant is saved as a child node. Each child node is further decomposed into quadrants, and the process continues recursively until every part of the image has been coded, or a threshold (say in terms of the tree depth) is reached.

The result is a tree structure in which a node is either a leaf node or contains exactly four children. The size of the quadtree depends on the complexity of the image. For images with large areas of identical pixels, compression is achieved since most of the quadrants will be represented by a single pixel value. However, for degenerate images (for instance, an image where each pixel has a different value), expansion rather than compression will result, since this may lead to a large number of nodes. This can be avoided by using a threshold on the height of the tree.

The quadtree and its 3-D extension (the octree) have been used for lossless data compression, for instance in medical imaging. For lossy compression, we can use an approximation to the quadrants. That is, homogeneity of the quadrants need not be based on exact equality of the pixel values. If no suitable approximation is found, the quadrant is partitioned further.

The quadtree representation is related to the general ideas of vector quantization, run-length encoding, and constant area coding [142]. It has also been used in fractal-based coding [117]. Vaisey and Gersho [356] applied quadtrees for lossy image compression by using quadtrees to segment an image into regions based on the activity in the image. Then, depending on the activity level in a given region, they choose either transform coding (for low activity) or vector quantization (for high activity) to code the image region. Block sizes for the transform coding were also decided based on the activity, while vector quantization used a fixed block size of $4 \times 4$. More recent work on quadtrees is reported in [25, 107, 249, 364, 385]. Verma et al. [364] proposed image compression using a combination of quadtrees and the DCT,
while Moreno et al. [249] proposed methods to combine the DCT with
wavelets using the Hilbert scan, rather than the usual Peano scan.

The quadtree partitioning could be suitable for compressed domain
search, especially if we can determine how to split the query image
to correspond with the quadrants used for the compressed images. Its
tree structure could also be exploited for fast searching and for progres-
sive image transmission. However, there are only a few reports on the
use of quadtrees for compressed domain matching (see [222] for exam-
ple). With the use of sequential symbol encoding for the blocks in the
quadtree, it might become possible to compare two compressed images
directly, using the encodings, by applying general pattern matching
methods. The quadtree is also related to popular spatial index struc-
tures used in nearest neighbor search, such as $kd$-trees. This connection
also points to the possibility of its use for compressed domain searching
and classification of images.

### 6.3.2 Block Transform Coding for Images

The most widely used methods and standards for image and video com-
pression are based on block transform codes. Particular examples here
are the JPEG standard [277, 365] for images, and the moving picture
equivalents for video, which are the MPEG series [207, 311] and the
H.26X series [373]. Block transform codes divide an image into blocks
(e.g., $8 \times 8$ as used in JPEG), and code these by transforming them into
a set of frequency domain coefficients, quantizing the coefficients, and
coding the quantized coefficients using Huffman or arithmetic coding.
These steps correspond to the three basic stages of image compression
as shown in Figure 1.1.

The transformation is typically a linear transform, and is usually
reversible (ignoring round-off errors). The transformation itself does
not produce any compression, but merely exposes the redundancies
in the image, which are then exploited by subsequent stages of the
compression. For most image blocks, the energy will be packed into only
a few coefficients, usually those with larger magnitudes. The remaining
coefficients (often the majority) will have small values, which can be
ignored, or quantized with little visual distortion in the reconstructed image.

Various compression methods based on the block-transform have been proposed [138]. The major differences are in the particular transformation they use, the way they quantize the coefficients (for instance, perceptually-adaptive schemes try to quantize based on the limitations of the human visual system), how the coefficients are chosen and traversed, and the bit allocation policy used to assign bits to the different coefficients. Most lossy compression standards, such as JPEG, MPEG, and H.26X, use the Discrete Cosine Transform (DCT), which is relatively fast and effective for images. Some other transforms such as wavelets (see below), Fast Fourier Transform (FFT), and Karhunen–Loève Transform (KLT) are also used.

6.3.3 DCT-based Transform-domain Image Retrieval

Although block-transform coding is a common approach to lossy image compression, transform coding poses some difficulty for compressed domain searching. The first problem comes with the choice of block positions and block boundaries. The value of the transform domain coefficients depends strongly on both the spatial position of the pixels within the transform block, and the actual value of the pixels. Two images that are similar, but whose blocks are selected in a slightly different manner (for instance, one image is slightly rotated, or translated) could produce different coefficient values. This creates a problem for matching two different blocks in the transform domain, since we cannot guarantee that the images would be perfectly registered before the transform blocks are chosen.

We have assumed above that the images are of the same size, or only differ by a small affine transformation. More difficult problems arise if the images are of different sizes, or if the image blocks are from different images. This is one of the reasons why, unlike in text pattern-matching, exact matching is usually inappropriate for images, or for multimedia data in general. This is related to the two basic distinctions in image search and retrieval, namely, whether we are to search for a given object or image block within another image, or whether we wish
to compare the images on a whole-picture basis (i.e., a small image can be compared with a much larger image block for similarity). The choice will depend on the application.

We can perform approximate (i.e., similarity) matching using the transform coefficients by doing further operations on the coefficients. In general, for image retrieval, searching is performed by use of image features extracted from the pixels, or from their transform coefficients, rather than the exact pixel values (or the exact coefficient values). For compressed domain search, the features are usually computed from the transform coefficients, based on the properties of the transformation used.

As with compressed domain text pattern matching, the reduced nature of the data in the compressed domain has also been exploited for image and video retrieval. For instance, the dc-image (an average image formed using only the dc coefficients of the DCT) has been used to speed up image searching in video sequences, and is the predominant method for fast image and video browsing [9, 320, 340].

In general, the techniques use the statistics of the compressed domain coefficients to form a feature vector for the database images, which are then used for later matching with those from a query image. For instance, Chang and his colleagues have studied various issues in compressed domain image and video manipulations, including methods for searching, especially for DCT and wavelet-based coding [82]. In [316], they computed the mean and variance of the coefficients from the DCT blocks, and used these to form the feature vector for the images. Reeves et al. [285] used a similar method, but observed that not all the DCT coefficients are very useful in discriminating between images, and hence used the first few significant DCT coefficients.

In [310] a method was proposed for searching on JPEG images after partial decompression. An image is divided into several windows, and a key is generated for the DCT (8 × 8) sub-blocks in each window. The key is computed based on the average of the DCT coefficients in a given window, and then used to compare images. Here, the operation could be at point (3) or (4) in Figure 1.2, that is, after the decoding stage, or after de-quantization.
Lew and Huang [209] investigated matching in transform block encoded images, specifically using the KLT and DCT. Li et al. [211] also investigated searching DCT coded data, but in a progressive transmission situation where the low-frequency DCT information is searched first. More recent approaches are investigated in [37, 101]. Zhang et al. [393] detected salient points based on the DCT, and used these for image retrieval on DCT-coded images. Bracamonte et al. [61] constructed the DCT-phase images by computing the phase of the DCT coefficients, requiring only a partial decoding and some basic mapping operations. Arnia et al. [32] used the simple sign of DCT coefficients for retrieval of JPEG compressed images, and later for content-based detection of near-duplicate image copies [30]. A similar method has also been used to analyze JPEG2000 images by considering the sign of the discrete wavelet transform (DWT) coefficients [367].

6.3.4 Vector Quantization

Vector quantization (VQ) [135, 149] is based on the concept of compression by pattern substitution. Instead of quantizing individual pixels (scalar quantization), a VQ system creates a limited codebook (a kind of dictionary) containing vectors, that is, blocks of an image. During encoding, blocks in the image are replaced with their respective nearest vector in the codebook. Nearness here is determined based on certain predefined distance or fidelity criteria. An image block that does not appear in the codebook/vocabulary will be replaced by its best approximation in the codebook. This explains why vector quantization is a lossy compression scheme.

VQ decoding is very fast, since the decoder need only look up an entry in the codebook. If the codebook is prepared in advance and fixed, encoding time is relatively fast since the main step is finding a match in the codebook. If the compression is adaptive, considerable time is needed to determine a good codebook. It may be possible to exploit the searching required during VQ coding to achieve fast compressed-domain matching, since the codebook provides a list of blocks that can be matched with the search pattern, and then only the locations of those blocks need be extracted in the compressed domain.
Indexing and retrieval for VQ-coded images have been explored in [159]. Here the indices of the codewords were defined as labels for the images, and a histogram of the labels was then used for image retrieval. In [361], the VQ codewords were used as image content descriptors suitable for retrieval. Oehler and Gray [269] use the information available in VQ-based coding to identify tumors in computerized tomography (CT) images, and to identify roads and buildings in aerial images. Schaefer [299] performed compressed domain image retrieval by observing that the VQ codebook for a given image should contain the important information in an image. Thus, image retrieval can be performed on the VQ-compressed data by simply comparing the VQ codebooks from the images. He proposed the use of the modified Hausdorff distance for similarity measurement. Jeong et al. [166] observed that VQ provides a simple and effective means to capture spatial information in an image, and thus explored the use of Gauss mixture vector quantization (GMVQ) for image retrieval. They showed that for color images, using the GMVQ with a log-likelihood distortion measure produced better retrieval results than the traditional VQ with a squared error distortion measure. In [178] a reduced complexity approach was proposed for image retrieval using VQ image thumbnails. Storer and colleagues performed color-based image retrieval on VQ-compressed images in [99], and VQ-based shape retrieval in [218]. In [280] compressed domain image retrieval on JPEG images was performed by building VQ codebooks on DCT coefficients after partial decompression, and then comparing images based on the VQ codebooks. A comparative performance evaluation on the use of VQ for image retrieval for both compressed and uncompressed images is presented in [179].

6.3.5 Subband and Wavelet-based Coding

Transform coding is just one type of the general frequency-domain coding techniques. The general approach is to decompose the original image into different (spatial) frequency components, with the aim of reducing the correlation in the original data, and to pack the most important information into fewer coefficients to facilitate compression. Subband decomposition and the wavelet transform are two other
methods that decompose the image into different components. In sub-band coding [138, 377] the image is decomposed into different subbands using different digital filters or band-pass filters. The components in each band are then quantized and encoded differently. Usually more bits are allocated to those bands that contain information that are more sensitive to human sensory perception.

Wavelet-based compression can be viewed as a type of subband coding, in which the image is decomposed into different bands at different resolutions [102]. The idea is related to the method of pyramidal image representation [71]. At each resolution, the image is represented as some averages and differences, called *detail coefficients*. The differences provide information about important details needed for exact reconstruction of the original image. Figure 6.2 shows a schematic diagram for the wavelet transformation using two and three levels of decomposition respectively. Figure 6.3 shows the corresponding decompositions on a sample image.

The differences can be compressed using any of the lossless compression schemes, such as RLE or Huffman coding. Also, there is usually

![Fig. 6.2 Schematic diagram of the wavelet transformation: (a) original image, (b) 2-level decomposition; (c) 3-level decomposition.](image)

![Fig. 6.3 Using the wavelet transformation on a sample image: (a) original image, (b) 2-level decomposition; (c) 3-level decomposition.](image)
prounced correlation across subbands and this can be exploited for further compression (see EZW below). Lossy compression (and hence higher compression) can be realized by quantizing small differences to zero. JPEG2000, a popular image compression standard, is based on the wavelet transform.

Modern wavelet-based compression is based on the idea of zero trees and embedding. Although transformations such as wavelets pack the energy into a few coefficients, sending the significance map (i.e., information for identifying which coefficients actually have high values) to the decoder is usually an expensive process, and could account for a major part of the bit budget at low bit rates. An important development in wavelet-based coding was the introduction of EZW (embedded zerotree wavelet) coding algorithm by Shapiro [305]. A key observation used in the EZW algorithm is that the value of the wavelet coefficient at a given scale can be related to the values of the coefficients at the corresponding spatial locations in the next higher (finer) scale. This is the basis of the EZW hypothesis [305]: Define a wavelet coefficient \( x \) as insignificant with respect to a threshold \( \tau \) if \(|x| < \tau\). Then, if a coefficient at a lower scale is insignificant with respect to a threshold \( \tau \), then all coefficients at the same orientation and same spatial location in higher (finer) scales are also likely to be insignificant with respect to \( \tau \). Such insignificant coefficients can then be quantized to zero, and their locations at higher scales can easily be determined. This means that we do not need to waste further bits to code them, nor do we need to spend part of the bit budget on the significance map. Given that many higher frequency coefficients will be quantized to zero, this leads to a significant improvement in compaction performance, with little or no impact on the quality of the reconstructed image.

EZW produces a fully embedded code, such that the bits in the compressed bit streams are prioritized in order of their importance. Thus, any given prefix of the compressed bit stream is itself equally a compressed bit stream, but at a lower quality. As more bits are received, the quality of the reconstructed image continues to improve. This capability is supported using the idea of successive approximation quantization. Thus, with bit-stream embedding, the EZW easily supports progressive image transmission (see Section 6.3.8). In particular,
image searching under progressive transmission becomes akin to finding a pattern on an improving approximation of an image as more bits arrive. This underscores the significance of approximate matches (rather than exact matches) for embedded coding or progressive transmission in particular, and lossy compression in general.

Various generalizations and extensions of the EZW algorithm have been proposed. Perhaps, the most popular is the set partitioning in hierarchical trees (SPIHT) algorithm [292], which produced improved compression performance without the need for the relatively complicated arithmetic coding procedure used in the original EZW algorithm. SPIHT has also been used for video compression (see [275, 276]). Another approach similar to EZW is Embedded Block Coding with Optimal Truncation (EBCOT) [341] which was chosen as the basis for coding and ordering in the JPEG2000 standard (see [342]).

Wavelets and subband coding are established methods for data compression, but they generally require more computations than other methods. Improved implementations of the DWT operate on the entire image, rather than on individual blocks, which is done with the block coding methods such as the DCT. Given the nature of the wavelet transform, the computation is typically performed using a sliding window on the image, without needing to load the entire image into memory. This capability is very important for very large images, such as satellite images or certain types of biomedical images. Also, their performance (for instance in terms of quality of the reconstructed image) is not much better. However, they provide avenues for significantly higher compression ratios, which make them good candidates for archival applications, where searching and retrieval are important activities. As with transform coded streams, the average image can be used for a rough query and fast browsing of the database. More precise image searching on the compressed stream is still difficult.

6.3.6 Compressed Image Retrieval Using Wavelets

Because of its multiscale image decomposition capability, the subband/wavelet-based compression schemes have been quite popular
for texture-based image retrieval. In [84], images were decomposed into different subbands, and feature vectors derived from the most significant coefficients in the middle subbands were used for texture matching. A similar approach was taken in [316], where they used the energy of the subbands to determine the texture features to be used for image matching. In [235], a histogram of the wavelet coefficients from different subbands was used for image retrieval. They observed that while different images could have similar overall histograms, the statistics of the different bands are likely to be different, and hence can be used to discriminate between images. Manjunath and Ma [236] considered Gabor wavelets, and used the mean and standard deviation of the coefficients in each subband to differentiate between different images. In [372], information from different wavelet bases was used to investigate ways to recognize and classify images. Sebe et al. [225, 344] used the notion of wavelet-based salient points for image retrieval. The salient points primarily correspond to positions of global or local variations in the image. Content-based features, such as color and texture, are then extracted from the salient points for image representation and analysis.

As with coding methods such as the DCT, the average image generated using wavelets can also be used for a rough query and for fast browsing of a database. This, however, will require some partial decompression to determine the coefficient values. Recent methods for image search and retrieval using JPEG2000 compressed images have been reported in [31, 335, 366, 389]. Zargari et al. [389] computed features from the header information in a compressed packet, without necessarily decoding the packet itself. In [335], histogram features were extracted based on the MSB and LSB of the wavelet coefficients, based on which image retrieval was performed directly on compressed JPEG2000 images. The method needed to perform only partial decompression to compute the required features. In [366], fast image comparison for JPEG2000 images was performed by analyzing only the compressed bitstream. They extracted the number of zero-bit planes in the compressed image, which is stored as part of the header information in JPEG2000. They thus avoided the time consuming stage of embedded block coding and optimized truncation (EBCOT) decoding.
Using the extracted information, they then compared a query image to database images.

In other related work [29], image retrieval and browsing was performed for a database with images compressed using the SPIHT algorithm. Images are segmented into regions or objects, and standard content-based features such as color, shape, texture or spatial information are extracted from the segmented regions. The attributes are then stored along with the SPIHT-compressed images. At query time, image comparison is performed based on the stored attributes, while image browsing is accomplished based on the multi-layered bit-streams (base layer plus enhancement layers) from SPIHT. The base layer is used to generate an initial version of the images (like thumbnails) which are displayed to the user. The user selects candidate images that are of interest, and the enhancement bit streams are transmitted, and then combined with the base layer for a final image reconstruction. In [59], wavelet-based compression of digital elevation maps based on SPIHT was used as the basis for fast search and retrieval of such maps. They used partial bit-stream decoding, made possible by the embedded nature of SPIHT bit-streams to support interactive multiresolution elevation searches, while region-based coding was used to allow parts of a map to be coded and decoded independently.

We observe that the exact performance of the compressed domain image retrieval schemes often depends on the specific features used, and on the application. There is still no one compression technique that has proved to be better than all the others for the different features we might want to use to retrieve images. Some earlier techniques have thus attempted to retrieve images when the underlying compression is based on a hybrid of two or more compression schemes. In [158], wavelet-based vector-quantized images were considered. The original image is decomposed using the wavelet transform and the wavelet coefficients are then coded using vector quantization. The codebook used in the VQ was then used as the index for later image retrieval. Swanson et al. [329] also considered retrieving images compressed with a hybrid of wavelets and VQ, but with some image regions coded with the DCT (the JPEG algorithm).
6.3.7 Pattern Matching Image Compression

Dictionary-based methods such as the LZ family exploit repetitions in the text, and have traditionally been used for lossless compression. Pattern matching image compression (PMIC) is a natural extension of the LZ approach to lossy compression. The basis is the idea of approximate repetitiveness [14] which is detected by approximate pattern matching. PMIC is related to grammar-based image coding in that both depend heavily on pattern matching and later substitution. However, while grammar-based multi-level pattern matching codes are used for lossless compression, PMIC is essentially for lossy image compression.

If a part of the already coded data re-occurs in an approximate sense, subsequent occurrences can be coded as a direct or indirect reference to the first occurrence. The approximate re-occurrence may or may not be continuous, and the nature of the re-occurrence may be different for different types of data (say images and text). Hence different distortion measures may be required for different data types. The squared-error distortion measure (MSE) is often used for images.

In [92], a lossy extension of the LZ78 algorithm was proposed for use with vector quantization. The concept of waiting time was used in [321] to extend the Lempel Ziv algorithm to lossy compression. Here, the waiting time is modeled as the number of symbols before an approximate match to a string of a given length re-occurs for the first time in the text (or equivalently, the length of the shortest string that contains an approximate match to a string of a given length). Szpankowski et al. [14, 34] focused on the approximate prefix analysis for the LZ family, and their use in image compression.

The general approach to pattern matching image compression (PMIC) is as follows [34]:

1. Choose an appropriate distortion criteria (such as the squared error or the absolute error) to determine the match distance between strings or image blocks;
2. Select some known part of the image as the database (for instance, the last few rows or subimages);
3. Search for the longest prefix in the uncompressed image that approximately matches a substring in the database (that is,
the prefix should be within a pre-defined distance threshold from the matching substring in the database);
4. Instead of storing the entire prefix, store a pointer to the occurrence of a match, and the match difference.

The database and the uncompressed image can be considered either as a 1D sequence, (for example, traversing the image in row-order), or as a sequence of 2D subimages. Thus, we could have one dimensional-PMIC or two-dimensional PMIC (based on 2D pattern matching). Since the LZ algorithms have been found suitable for compressed pattern matching, it is expected that image searching could be performed directly on the lossy extensions, if we take cognizance of the distortion measures used during the compression stage.

It has been reported that, in terms of compression ratio and reconstruction quality, the PMIC methods are generally comparable with transform-based methods (such as JPEG), and methods based on wavelets. However, they require more compression time, but much less decompression time [34].

6.3.8 Progressive Image Transmission

Progressive image transmission [24, 306, 374] involves sending a low quality image followed by successively more detail, so the picture quality gradually improves as more data is being received. This method is used on the web, for example, so that the user can cancel a page early in the transmission if it is not of interest.

Some image compression standards (e.g., JPEG and JPEG2000) include options for progressive image transmission. For instance, with transform-based schemes (block-transform, subband, and wavelet decomposition schemes), progressive transmission can be achieved by first sending the first few coefficients (or frequency bands) for all blocks, and then transmitting information from more bands progressively. Another approach is to send the most significant bits from all the frequency coefficients first, and progressively send the remaining bits. Progressive coding (also called successive approximation quantization) is an important procedure in modern wavelet compression methods, such as those based on EZW [305], SPIHT [292], and EBCOT [341].
Progressive image transmission raises the possibility of searching a low resolution version of an image, and then focusing in on “interesting” parts of the image. For instance, if the coding scheme is based on the DCT, the dc components can be sent initially, which will be used to create an image with only the dc component of each block, which can in turn be used to perform a fast appraisal or search of the image, before all the data components arrive.

6.3.9 Others

Some methods for compressed-pattern matching have also been proposed for images compressed with nontraditional coding techniques. Vasconcelos and Lippman [359] argue that a library of relevant objects can be developed during the compression of a file, so the search component at the coding stage does double duty by also supporting later search of the data. This principle is similar to some compressed full-text retrieval systems where a lexicon of words is constructed which is used as a library for compression and also as an index for full-text searching [376]. Chen and Bovik [86] proposed a system that attempts to code sub-images that are “meaningful to a normal human observer.” Their system is called visual pattern image coding (VPIC). A similar approach was taken by Gerek et al. [134], who developed a textual image compression system that can facilitate searching the compressed data.

A more unusual form of lossy compression is the signature file [147], which has been popular in text information retrieval systems. A “signature” is created for each document or record using hashing, and a simple probabilistic test can be used to find possible matches of a search key with a signature. This is a specialist system, and allows complex queries including boolean combinations of terms. The signatures are not intended to be decompressed (in fact, the system is not really touted as a compression scheme), but they serve as a quick method for retrieving potentially similar text regions to a given query using less space than uncompressed alternatives.

A related topic is the concept of image hashing [247, 331, 382], which has found applications in various areas, from image/video retrieval to
watermarking and cryptography to duplicate detection. An image hash is a short sequence (usually a binary sequence) computed from an image using a hashing function. An important requirement for an image hash function is that perceptually similar images should have similar hash values, while perceptually dissimilar images should have less similar hash values. Image hashes were generated based on DCT coefficients in [122, 227] and in [219], where they used image hashes to identify the effect of malicious changes on JPEG compressed images. Image hashes based on wavelet coefficients were proposed in [363, 383]. In particular, Yang and Chen [383] proposed a method for robust image hashing based on the significance maps used in SPIHT coding. The significance map indicates the positions of significant wavelet coefficients (relative to a given threshold), and hence the binary sequence from the significance map can be used as a compact representation of the image. Yang and Chen used 108 significance maps, each map being a 16 \times 16 array of binary digits, where a “1” represents a significant pixel (or set). To reduce space and improve speed of comparison, the significance maps are reduced to autocorrelograms. For a given distance between two coefficients, the autocorrelogram essentially counts the number of significant coefficients that are separated by that distance. The autocorrelogram therefore captures important spatial relationships between the significant wavelet coefficients. Two images are then compared for possible similarity based on their hash values.
So far we have surveyed research on different aspects of the problem of searching compressed text and images. In this section, we speculate on what is likely to occupy researchers in this field in the short term and in the long run. Below, we describe these research activities under different subheadings: new compressed pattern matching algorithms, new search-aware compression algorithms, compressed image descriptors, performance measures, integration and adaptation, hardware implementation, and new applications.

7.1 New Compressed Pattern Matching Algorithms

Various methods have been proposed for compressed pattern matching for both text and images. Although the focus has been mainly on a few compression methods such as Huffman coding, the LZ family, and BWT for text, or the RLE, VQ, and transform-based methods (DCT, wavelets) for images, algorithms are still required for pattern matching on other compression algorithms and new ones that might appear. For instance, a few methods have been proposed [309] for searching text compressed using schemes such as antidictionaries [98]. Grammar-based methods, such as MPM and SLP have only received a little
attention with respect to compressed matching [132, 167, 243, 214]. But we are yet to see algorithms that search data compressed with high-performance context-based compression schemes, such as the PPM even though, in terms of data compaction, the PPM approach is one of the best performing methods for text sequences, although at the expense of a high computational cost. We note that because the performance measures could be different when searching is considered, algorithms that may not be very good on compaction and coding complexity could provide better performance for searching, especially if decoding can be made fast.

In the long run, researchers will still be looking for possible answers to some currently hard questions. For instance, can there be variations on arithmetic coding that will make it possible to perform compressed pattern matching? How can we perform precise image searching directly on compressed data? How can we make popular compression algorithms (such as wavelets) faster, which will make them more generally attractive, and hence more important for compressed pattern matching in images or video?

Furthermore, current algorithms have focused mainly on the basic pattern-matching problem (exact or approximate pattern matching). However, in the short term, there is a need to start looking into compressed solutions to the other variants of the compressed pattern-matching problem, especially those with immediate applications. For instance, compressed multidimensional pattern matching will find uses in new applications, such as multimedia data analysis and signal processing, while compressed solutions to the dictionary matching and super pattern-matching problems will be of interest to researchers in data mining and pattern discovery. In the long term we expect compressed solutions to the more theoretical variants of the pattern-matching problem, such as pattern matching with don’t cares, pattern matching with rotations, and parameterized pattern matching (see [26, 45] for initial ideas in this direction).

The use of parallel algorithms has the potential for further efficiency improvements. Various parallel algorithms have been proposed for the traditional pattern matching problem [124, 136, 198, 379], but there has been little or no attention paid to the problem of parallel
7.1 New Compressed Pattern Matching Algorithms

Compressed pattern matching. A few studies on this problem are reported in [130, 289]. We envisage more interest in developing parallel compressed pattern matching algorithms for both text and images. There are two obvious ways to do this: either develop parallel versions of existing compressed pattern matching algorithms, or to develop analogous compressed pattern matching versions for currently available parallel pattern matching algorithms. This issue is also related to hardware platforms for compressed pattern matching, discussed later in this section.

For images, the main attention has been on VQ, wavelets and DCT-based compression schemes, perhaps due to the availability of compression standards such as JPEG and MPEG (DCT-based), and JPEG2000 (wavelet-based). Little has been done on searching on images compressed with other techniques, yet some of these image compression methods (e.g., wavelets using EZW or SPIHT) have shown promise for significant compression ratio improvements over DCT-based methods, which make them candidates for archival applications, or typical multimedia environments. Further, other schemes (such as quadtrees) could provide ideas for addressing the difficult problem of object-level searching in compressed images. Already some of these have been found to be particularly suitable for searching based on particular image features. With the success of whole-image similarity searching using DCT based features, we anticipate more effort in developing equivalent retrieval algorithms for images that are compressed with these other schemes.

A similar observation can be made on the lack of compressed-domain methods for context-based predictive schemes used for lossless compression, yet these provide the best performance in terms of data compaction ability. Perhaps, the harder question is the problem of precise search on images — whether on compressed or uncompressed images. For compressed images, matching has generally been approximate, since the compression is usually lossy. Thus, initial attempts to solve this problem will have to look at developing pattern-matching algorithms for lossless image compression schemes, such as JPEG-LS [370]. Furthermore, with the increasing importance of applications for lossless image compression (such as in medical imaging), researchers will start to look at the problem of developing equivalent algorithms.
that can search directly on lossless compressed images. The recent report on the use of BWT for effective lossless compression of images [4] (with compression ratios comparable with those from popular lossless image encoding schemes such as CALIC [239], JPEG-LS, or PPAM [394]), could open up the possibility for effective compressed pattern matching on lossless compressed images, given the success of compressed pattern matching on BWT-text.

A related problem is searching on compressed images with gray-scale recognition. Methods have been proposed for the case of uncompressed images [23, 223, 332]. Developing equivalent methods that can work directly on compressed images (be it lossy or lossless) represents an interesting challenge.

7.2 New Search-aware Compression Algorithms

Already there have been proposals for search-aware compression algorithms — that is, compression schemes designed with search in mind [64, 118, 200, 233, 309]. Given the success of the BWT in supporting pattern matching on compressed text, we envisage that this trend will continue, especially with the increased demand for “compress-once, search-many” applications, such as digital libraries, multimedia databases, and web-based systems. In this situation the ability to support later searching becomes a performance criterion used to measure the algorithms.

Image compression schemes have, however, not generally considered the problem of search at the compression stage. This is bound to change. Already, as part of the compression process, some image compression methods provide useful by-products (such as code vocabulary in VQ or data structures in PMIC) that can be exploited later to search the compressed data. Also, for images, object-level search (i.e., within-image search for objects) has posed a serious problem and has been largely ignored. The major issue has been how to describe the shape boundaries in the transform domain. Coding methods, such as shape-adaptive coding schemes [212, 312] have made it possible to describe object boundaries as part of the compression. Compression standards such as MPEG-4 and MPEG-7, and the H.26X series have
also considered issues related to object-level access in the compressed data [1, 373, 395]. Pajarola and Widmayer [273] provide an example image compression scheme that supports spatial search.

In general, new image and video compression standards are taking the issue of searching directly on the compressed data more seriously. New search-aware compression algorithms that are tuned for special applications where searching is a key issue (such as web-based applications, digital libraries, multimedia databases) are expected in the near term for lossy compression. In the long term, we expect a similar development for lossless image compression.

7.3 Compressed Image Descriptors

We have presented previous work from the viewpoint of pattern matching on compressed data, and to some extent compression schemes that are designed with pattern matching in mind. In the case of images, features or image descriptors are computed from the compressed data, for instance, using the transform coefficients, based on which transform domain pattern matching is then performed. A more recent research thrust is on compressing the image features themselves, and then performing compressed domain image search based on these compressed features [80, 81]. That is, image descriptors or image features are designed with compression and later searching in mind. The basic approach follows the general compression model, except that here the inputs are the image features or descriptors, rather than the original image. For instance, in [81], traditional gradient histogram features are compressed by first applying vector quantization on the features, and then compressing the results using lossless fixed or variable length codes.

Compressing features or descriptors is motivated by the recent interest in mobile visual search [140], whereby cheap mobile handheld devices, such as smartphones and cellphones are used to capture images on the go, and the captured images are then used to search a database of images, typically remotely over a network. The use

---

of feature compression in this case has three advantages: reduced amount of data to be transmitted, reduced network latency, and reduced computation. Already there is increasing attention being paid to the problem of image feature compression. Calonder et al. [73] proposed BRIEF (Binary Robust Independent Elementary Features) while Johnson [171] described different methods to compress image descriptors. Chandrasekhar et al. [79] provided a survey of recent methods for compression of image descriptors. With the explosion in the number of users of resource-constrained devices, such as cellphones, smartphones, and other mobile handheld devices, we envisage that there will be more interest in this emerging area of compression for image features or descriptors.

7.4 Performance Measures

The performance measure for new search-aware compression algorithms will include the capacity for explicit search support, along with the traditional measures of data compaction, complexity, and reconstruction quality (in the case of lossy compression). Short-term problems here include the development of benchmark databases for testing the various algorithms. There are already a number of standard databases for testing text compression algorithms [72, 76], for text-based retrieval systems [345], or video retrieval systems [346]. None of these was designed specifically for measuring pattern matching performance. There is a need for a similar benchmark for evaluating algorithms that search directly on compressed images. Furthermore, we have enumerated a number of parameters based on which compressed pattern-matching algorithms can be evaluated. In the long run, researchers might also need to re-examine these and the current measures, possibly with a view to developing application-specific measures of performance.

7.5 Integration and Adaptation

Many environments will involve data corpora that are compressed with different compression schemes. This is currently typical of web-based applications, and the number of such applications is bound to increase
in the future. One problem will be how to perform search transparently on the different data formats or data types. We envisage that, over time, new compression methods will be introduced, and hence different methods will be used to search them directly. There will also be a proliferation of different search techniques on the same compression algorithm. We are already witnessing this proliferation of compressed pattern matching schemes, such as those for the LZ family, BWT, and RLE (for text), the LZ family and RLE for lossless compressed images, and DCT, VQ and wavelets for lossy compressed images. Moreover, for image matching algorithms, there is the possibility of missing the occurrence of the query image in the database, or returning wrong results altogether. When we consider all these, along with the different performance criteria (some of them opposing each other), the obvious question will be how to make compressed pattern matching to be adaptive.

Adaptation could be in different forms. Examples include:

- the choice of the compression scheme, which might be based on the perceived future search activities on the data or the matching algorithm that will be adopted;
- the choice of the particular compressed pattern matching algorithm to use, for instance, based on the weights attached to the different performance measures; and
- the use of a hierarchy of search algorithms (especially for image matching algorithms) — akin to the idea of metasearch used for web search engines [141, 203, 313].

A related issue is the increasing interest in transcoding between text compression systems. While transcoding has long been used in video coding, this has not been the case in text compression. However, certain operations could be easier under one compression system than some others. For instance, conversion from the LZ family to SLPs has been one major approach for fully compressed pattern matching on LZ-coded text [128]. Since fully compressed pattern matching is significantly easier on RLE, recent reports on conversion between LZ78 and RLE [333] could point toward simpler methods for compressed pattern matching for the LZ family. One can expect this trend to continue, with conversions involving algorithms other than SLPs or the LZ family.
7.6 Hardware Implementation

In parallel with the development of compression algorithms, very large scale integrated (VLSI) circuit technology has made tremendous strides and opens up the possibility of implementing a system-on-chip (SoC) to perform real-time data compression functions over mobile and network communication channels. Several papers have appeared in the literature (we mention only a small subset of these here) to perform hardware data compression. The basic operation of the LZ algorithm is a string matching operation. Hardware architectures for the LZ family have been proposed based on content-addressable memory (CAM) [221], linear systolic arrays [284], and reconfigurable field programmable gate arrays (FPGAs) [152]. A parallel algorithm for LZ compression was presented in [191]. Hardware algorithms for Huffman coding and arithmetic coding have also been proposed by several authors [33, 224, 252, 253]. Hardware architectures for BWT-based compression have been proposed in [139, 274].

Several hardware architectures have been reported for image or video coding using JPEG2000 [2], and JPEG-LS [192]. Tseng et al. [348] provide a comprehensive survey on architectures for image and video coding. Similarly, several hardware implementations for general pattern matching (for uncompressed text) have been proposed [105, 199]. However, not much has been done with respect to hardware implementation of compressed pattern matching algorithms. Only a few papers have investigated this topic [63, 108, 252]. Future work on hardware algorithms for data compression could be directed toward implementing some of the better performing lossless algorithms, such as PPM and DMC, and context-based predictive lossless coding methods for images. Developing hardware platforms for compressed domain pattern matching both for lossless and lossy compression algorithms will be an exciting challenge for future work.

7.7 New Applications

Traditionally pattern matching (especially exact pattern matching) has been used mainly in text-based environments. Key applications, such
as image retrieval, have motivated the need for approximate pattern matching for images. But the simple problem of exact matching is yet to be solved for images, due to the difficult question of image registration. Notwithstanding this, we still envisage that emerging applications, such as web-based information retrieval [195], multimedia information systems and concept retrieval [100, 319], and searching on resource-constrained mobile devices such as smartphones [65] will drive the need for compressed pattern matching in new application environments involving different data types.

This will particularly be the case for applications that require access to multimedia content, where the large data sizes typically involved still make efficiency considerations a major issue, for example, in image browsing on PDAs or smartphones, and online map applications. Another instance here is the potential impact of solutions to the problems of compressed super-pattern matching or compressed dictionary matching on new applications, such as data mining. More generally, in the long term the development of compressed pattern matching algorithms will encourage the drive for new application areas, which have so far been thought to be too time consuming. An example is object-level retrieval from large compressed image repositories. For text, we are already witnessing this trend, for instance, with new results on computing convolutions between grammar-compressed strings [334], or on searching straight-line programs for longest common patterns between strings [237], squares [40], and palindromes [237].
In this review we have surveyed recent and past efforts in searching compressed text and images. We have outlined the basic concepts and assumptions used in data compression and in pattern matching, and the benefit of having searching occur in the compressed domain, where there is less data to process and (in some cases) structures available that can aid searching.

The review also identified the special relationship between data compression and pattern matching (searching). It was observed that searching is an important part of the compression process, while on the other hand, compression can be used to improve later searching. Algorithms that search directly on lossless compressed text have focused mainly on Huffman codes, the BWT, and the LZ family of coding algorithms. Searching on compressed images has generally been in the form of transform domain analysis, with emphasis on lossy compression schemes. This has primarily been in the context of image retrieval, and generally on an approximate basis, with emphasis on image compression schemes based on the DCT, wavelets or vector quantization. Little has been done on the important problem of
searching directly on lossless compressed images, based on the popular context-based predictive image coding schemes.

We considered several measures of performance for compressed pattern-matching: complexity and speed, extra space, optimality, precision and recall, ranking, and comparison with decompress-and-search algorithms. Trends indicate that, in the future, compressed pattern matching research will likely focus on key issues such as new compressed pattern matching algorithms for existing compression systems, new search-aware compression algorithms, the development of new applications, performance measures and benchmarks, adaptation in compressed pattern matching, parallel algorithms, and hardware implementations.
Acknowledgments

We thank Syed Rahman for his help with some of the figures and tables.
References


References


[26] A. Apostolico, P. L. Erdos, and A. Juttner, “Parameterized searching with
mismatches for run-length encoded strings,” Theoretical Computer Science
(in press), 2012.
strategies revisited,” SIAM Journal on Computing, vol. 15, no. 1, pp. 98–105,
February 1986.
compression for retrieval system, November 2000.
content-based copy detection,” in Proceedings, International Symposium on
Communications and Information Technologies (ISCIT ’07), pp. 1476–1481,
October 2007.
[31] F. Arnia, I. Iizuka, M. Fujiyoshi, and H. Kiya, “DCT sign-based similarity
measure for JPEG image retrieval,” IEICE Transactions, vol. 90-A, no. 9,
and identification of JPEG coded images nased on DCT sign,” in Proceedings,
purpose VLSI CAD chip for adaptive data compression of bilevel images,” IBM
Journal of Research and Development, vol. 32, no. 6, pp. 775–795, November
[34] M. Atallah, Y. Génin, and W. Szpankowski, “Pattern matching image com-
pression: Algorithmic and experimental results,” IEEE Transactions on Pat-
Press/Addison-Wesley, 1999.
retrieval in DCT domain,” Intelligent Interactive Multimedia, vol. SIST 11,
[38] B. S. Baker, “Parameterized pattern matching: Algorithms and applications,”
arrays of more than one dimension,” SIAM Journal on Computing, vol. 7,
no. 4, pp. 533–541, November 1978.
[40] H. Bannai, T. Gagie, I. Tomehiro, S. Inenaga, G. M. Landau, and M. Lewen-
stein, “An efficient algorithm to test square-freeness of strings compressed
by straight-line programs,” Information Processing Letters, vol. 112, no. 19,
[41] P. Barcaccia, A. Cresti, and S. D. Agostino, “Pattern matching in text com-
pressed with the ID heuristic,” in Proceedings, IEEE Data Compression Con-
References


References

References


References


References


References


References

References


References


References


References


