

**WORD - BASED COMPRESSION IN FULL - TEXT
RETRIEVAL SYSTEMS**

**A THESIS
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL
ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE**

**By
Ali Aydin Selçuk
May, 1995**

QA
76.9
.D33
S45
1995

WORD-BASED COMPRESSION IN FULL-TEXT
RETRIEVAL SYSTEMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF INDUSTRIAL
ENGINEERING
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Ali Aydın Selçuk
May, 1995

Ali Aydın Selçuk
tarafından beğlenmiştir.

QA

76.9

.A33

S45

1995

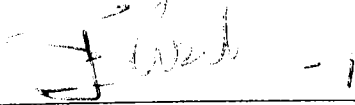
B030746

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



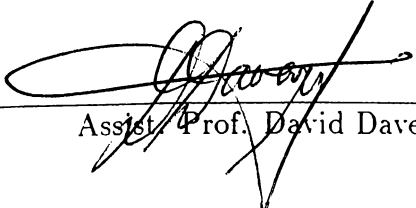
Prof. M. Akif Eyley (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



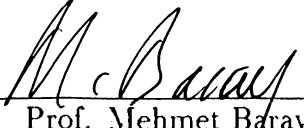
Assoc. Prof. Erdal Arıkan

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.



Assist. Prof. David Davenport

Approved for the Institute of Engineering and Sciences:



Prof. Mehmet Baray
Director of Institute of Engineering and Sciences

ABSTRACT

WORD-BASED COMPRESSION IN FULL-TEXT RETRIEVAL SYSTEMS

Ali Aydın Selçuk
M.S. in Industrial Engineering
Supervisor: Prof. M. Akif Eyller
May, 1995

Large space requirement of a full-text retrieval system can be reduced significantly by data compression. In this study, the problem of compressing the main text of a full-text retrieval system is addressed and performance of several coding techniques for compressing the text database is compared. Experiments show that statistical techniques, such as arithmetic coding and Huffman coding, give the best compression among the implemented; and using a semi-static word-based model, the space needed to store English text is less than one third of the original requirement.

Key words: Full-text retrieval, Data compression, Text compression, Word-based model

ÖZET

TAM METİN ERİŞİM SİSTEMLERİNDE KELİME TABANLI SIKIŞTIRMA

Ali Aydın Selçuk

Endüstri Mühendisliği Bölümü Yüksek Lisans

Tez Yöneticisi: Prof. Dr. M. Akif Eyler

Mayıs, 1995

Tam metin erişim sistemlerinin büyük yer ihtiyaçları veri sıkıştırma ile büyük ölçüde azaltılabilir. Bu çalışmada bir tam metin erişim sisteminin metin veritabanının sıkıştırılması problemi incelenmiş, ve ana metnin sıkıştırılması için değişik kodlama tekniklerinin performansları karşılaştırılmıştır. Yapılan deneyler uygulanan metodlar arasında en iyi sıkıştırmanın Huffman kodlaması ve aritmetik kodlama gibi istatistiksel teknikler tarafından sağlandığını göstermiştir.

Anahtar kelimeler: Tam metin erişimi, Veri sıkıştırma, Metin sıkıştırma, Kelime tabanlı modelleme

ACKNOWLEDGEMENT

I am very grateful to my supervisor, Professor M. Akif Eyer for his supervision, guidance, suggestions, and encouragement throughout the development of this thesis.

I am indebted to Associate Professor Erdal Arıkan and Assistant Professor David Davenport for their valuable comments.

I would also like to thank to Turgay Korkmaz, Hakan Koroğlu, M. Bayram Yıldırım and Kürşad U. Akpınar for their valuable comments on the presentation of the subject matter, to Ali Tamur for his enlightening discussions on arithmetic coding, to Mehmet Sürav and Hüseyin Simitçi for their help in coding and debugging the computer programs, and to Alper Şen, Selçuk Avcı, Yavuz Karapınar, Engin Topaloğlu and Abdullah Daşçı for their help in typing the thesis.

Contents

1	Introduction	1
1.1	Full-Text Retrieval Systems	1
1.1.1	Queries to Full-Text Databases	1
1.1.2	The Text Index	2
1.2	Problem Definition	5
1.3	Previous Work	7
1.3.1	Compression Techniques	7
1.3.2	Compressing the Main Text	8
1.3.3	Compressing the Text Index	9
2	Compression Techniques	12
2.1	Huffman Coding	12
2.2	Arithmetic Coding	17
2.2.1	Incremental Transmission and Reception	19
2.2.2	The Underflow Problem	21

<i>CONTENTS</i>	viii
2.2.3 Terminating the Message	23
2.3 Dictionary Encoders and Ziv-Lempel Coding	24
2.3.1 LZW	26
3 Implementation	30
3.1 Compression Schemes	30
3.2 Test Databases	32
3.3 Results	33
4 Conclusions	37
A Software Arithmetics for Arithmetic Coding	40

List of Figures

2.1	An example of Huffman coding	14
2.2	An example of canonical Huffman coding	16
2.3	Representation of the arithmetic coding process	18
2.4	Scaling the interval to prevent underflow	23
2.5	LZW coding of the string “aabababaaa” (phrases 0 and 1 are present before coding begins)	27
2.6	LZW decoding of the string “001352” (phrases 0 and 1 are present before decoding begins)	29

List of Tables

3.1	Information about test databases	32
3.2	Experiment results for <i>alice13a.txt</i> (av. frag. size = 1 KByte)	34
3.3	Experiment results for <i>alice13a.txt</i> (av. frag. size = 10 KByte)	34
3.4	Experiment results for <i>un.dalamp</i> (av. frag. size = 1 KByte)	35
3.5	Experiment results for <i>un.dalamp</i> (av. frag. size = 10 KByte)	35
3.6	Experiment results for <i>tb</i> (av. frag. size = 10 KByte)	36
3.7	Experiment results for <i>tb</i> (av. frag. size = 100 KByte)	36

Chapter 1

Introduction

1.1 Full-Text Retrieval Systems

A full-text retrieval(FTR) system is an information retrieval system enabling computer searching of text databases using an automatically-constructed index. FTR systems are used for storing and accessing document collections such as newspaper archives, on-line article collections, office automation systems, and on-line help facilities. The data in an FTR system is usually unstructured running text and the general topic and style of the documents are usually related. The text database of an FTR system usually includes a large number of *text fragments*, where each fragment is an individually retrievable portion of text. For example, a fragment might be a sentence, a paragraph, a page, or an entire document. The needs of full-text databases are not well served by traditional database systems, since, instead of key indexing, full-text requires facilities such as document indexing on text content.

1.1.1 Queries to Full-Text Databases

Queries in full-text databases can be either *Boolean* or *ranked*. Boolean queries involve searching for text fragments containing terms specified by a Boolean

expression, such as “information *and* (storage *or* retrieval)”. All fragments that contain the word “information” and either “storage” or “retrieval” or both would be answers to this query.

To obtain consistently good results by a Boolean query is usually not possible for ordinary end users. This is mainly due to two reasons. First, the user may not be able to formulate his query as a Boolean query. Second, at the end of the query, the user has a set of text fragments among which it is not possible to distinguish between the more and the less relevant fragments. Ranking is more oriented toward these end users. First, the user is allowed to input a simple informal query such as a sentence, a phrase or a text. Second, he ends up with a ranked solution set, from most to least relevant fragment. There is a wide variety of ranking techniques used to measure the similarity of a fragment to a query [41, 39, 16]. These techniques are usually based on statistical measures, whereas some of them use natural language processing methods. Cosine measure is a good example to the statistical techniques which not only performs well, but is cheap to compute [2].

The major drawback of ranking approach compared to the Boolean approach is that it does not allow using the Boolean logical operators such as *and*, *or* and *not*. A number of methods, known as the extended Boolean methods, have been proposed to combine the ranking and the Boolean approaches [40, 16].

1.1.2 The Text Index

Answering queries by scanning the entire text for the query terms is usually too slow --it takes about an hour to read all the data on a CD-ROM. [2]. Instead, an index must be provided with the text to enable queries to be answered within a reasonable delay. Indexes should enable query terms to be located in the index to get the information where the term appears in the main text.

In full-text retrieval, most (if not all) words in the text must be indexed.

Although some words are unlikely to be used in practical queries –for example common words such as “the”– there are relatively few such words, they can be stored so that they make only a small contribution to the size of indexes [2], and omitting them makes queries on these words much more expensive to evaluate.

The most common types of indexes are *bitmaps*, *inverted files*, and *signature files*. Inverted file and bitmap structures both require a *lexicon* or *vocabulary* –a list of all index terms– whereas the signature file method does not. The indexing method used in this study is the inverted file structure. However, we will discuss all these schemes briefly to provide an overview of the indexing concept.

Lexicon. A lexicon (also known as vocabulary) is a list of all index terms. It is one of the major components of the index in bitmap and inverted file index structures. In inverted file indexes pointers to inverted lists are stored together with the index terms in the lexicon. Frequency counts of the terms are also stored in the lexicon if the index is to support ranked queries depending on statistical methods. Usually words in the main text are stemmed and case-folded before being recorded to the lexicon, in order to improve the retrieval effectiveness. A conventional approach is not to include the stopwords (i.e. common words with low information content such as *the*, *of*, *at*, etc.) in the lexicon. Recently several authors have proposed to index every occurrence of every term including stopwords [2, 49]. The reason for this suggestion is the fact that bitmaps or inverted lists of the stopwords can be stored in a relatively small space by index compression techniques, and omitting them makes queries on these words much more expensive to evaluate.

Bitmaps. A bitmap is perhaps the most obvious indexing structure. For every term in the vocabulary (also known as the lexicon) a bitvector is stored, each bit corresponding to a text fragment. A bit is set to one if the term appears anywhere in that fragment, and zero otherwise. Bitmaps are particularly efficient for answering Boolean queries –the bitvectors for the terms are simply combined using the appropriate Boolean operations, which are often available

in fast dedicated hardware [2]. Bitmaps are fast, easy to use, but extravagant in storage. For a text of N fragments and n distinct index terms, a bitmap occupies Nn bits.

Inverted Files. An inverted file contains, for each index term, an *inverted file entry* or an *inverted list* that stores a list of pointers to occurrences of that term in the main text, where each pointer is the number of a text fragment (usually a document) in which that term appears. This approach is quite natural and corresponds closely to the index of a book.

The *granularity* of an index is the accuracy to which it identifies the location of a term. A coarse-grained index might identify only a block of text, where each block stores several documents; while a fine-grained one will return a sentence or a word number. Coarse indexes require less storage, but are less efficient in retrieval performance. At the other extreme, word-level indexing enables queries involving proximity. However, adding such precise location information significantly increases the size of the index. More generally, an inverted file may provide a multi-level index structure with a hierarchical set of addresses—for example, a word number within a sentence number within a paragraph number within a section number within a chapter number. In this case each pointer in the list will be a k -tuple in a k -level index.

An inverted file index can be augmented to store the within document frequency of index terms together with the pointers in the index. This kind of information is extremely important to support ranked queries that use similarity measures based on statistical techniques.

A major drawback of an inverted file index is the space it requires. An uncompressed inverted file may occupy 50 percent to 100 percent of the space of the text itself.

Description of the implementation of inverted file retrieval systems have been given by numerous authors [8, 27, 21, 28].

Signature Files. A signature file is a probabilistic method for indexing

text. Each text fragment has an associated *signature*, in which every index term is used to generate several hash values, and the bits of the signature corresponding to those hash values are set to one. To test whether a query term occurs in a given fragment, the values of the hash functions for that term are determined. If all corresponding bits in the signature are set, the term probably occurs in the fragment. The fragment should then be read to check that the term really does occur. The probability of a false match can be kept arbitrarily low by setting several bits for each term and making the signature sufficiently large.

Signature files become more effective as the queries become more specific, since the queries involving the conjunction of several terms can check more bits in the signature file. Only one bit needs to be zero to cause the match to fail, and this leads to a low probability for false matches. Signature files cannot be used directly to implement Boolean negation, because even if a signature indicates that a term might occur, that fragment still needs to be obtained from the main text to check that the word actually does appear and that fragment cannot be an answer. Thus any negations must be ignored while checking signatures, and instead have to be checked after the text has been read.

Faloutsos surveys signature file techniques [14]. Various structures based on signature files are described by Sacks-Davis *et al.* [38, 23]. The tradeoff between storage space and the probability of false matches in signature files is examined by Faloutsos and Christodoulakis [12, 13].

1.2 Problem Definition

FTR systems are traditionally large [16]. Therefore, their space requirement has been a problem and reducing the space required has been studied by several people [9, 24, 53, 32]. The advances in the CD-ROM (compact disk-read only memory) technology made compression of FTR systems more attractive, because a very large text database can be distributed very easily if it can be

compressed to fit on a single CD; and data compression in FTR systems has become an active area of research in the last years.

Two major components of an FTR system are the main text and the index. The main text is usually a large amount of running text in natural language. The information content of several natural languages has been studied and it is shown that a text in natural language usually contains a lot of redundancies [1, 24]. For example, in English, the letter *q* is almost always followed by *u*, and in French *yi* is almost always followed either by *ons* or by *ez* [24].

The index of the text may also occupy as much space as the text does, or even more. For a text of N fragments and n distinct index terms, a bitmap occupies Nn bits. More than 90 % of these bits are usually zeros. Signature files and inverted files can be considered as special forms of bitmaps and also occupy significant amount of space with removable redundancies [2].

Our study has concentrated on the problem of compressing the main text of an FTR system by using the information stored at the index as a word-based semi-static model. Word-based approach is to take each word as a token instead of individual characters. "Semi-static" indicates that the model is static throughout a collection, but is different for different collections.

The objective of the study was to compare several coding techniques for compression of the text database, and to find the most appropriate one(s).

For this purpose we have implemented and compared a variety of compression techniques on several full-text databases indexed with an inverted file that stores the overall frequencies and the within fragment frequencies of the index terms.

Criteria used in measuring the performance are *compression ratio*, *encoding speed* and *decoding speed*. Compression ratio is defined as the proportion of the size of the compressed text to the size of the original text. Encoding speed is important if the compressed text is not likely to be used again, such as backups and archives, but can be overlooked in compression of an FTR system,

especially when the system is static. Decoding speed is the most important speed consideration of a coding scheme for our purpose.

1.3 Previous Work

1.3.1 Compression Techniques

The relationship between probabilities and codes was established in Shannon's source coding theorem [43], which shows that a symbol that is expected to occur with probability p can be represented in no less than $-\log p$ bits¹, averaged over all symbols emitted from a stochastic source. Later, Shannon and Fano independently discovered an asymptotically optimal coding algorithm [1]. Shortly after Shannon's work, Huffman discovered a way of constructing optimal codes for any given discrete memoryless source. [22]. The code produced by Huffman's algorithm was optimal given that each message must be coded with an integral number of bits. Later, Gallager showed that the redundancy of Huffman codes, defined as the average code length less the entropy, is bounded above by $p_{max} + 0.086$, where p_{max} is the probability of the most likely message [17].

The honor of first realizing the idea of arithmetic coding is usually attributed to Elias [1, 25]. The discovery that the calculation could be approximated in finite-precision arithmetic was made independently in the mid 1970s by Pasco and Rissanen [33, 34]. Shortly after that, first practical implementations appeared [36, 20, 35]. Witten *et al.* [50] presented a full description and evaluation of arithmetic coding.

In 1967 White made the first remark that better compression could be obtained by "replacing a repeated string by a reference to an earlier occurrence" [46]. His idea was not pursued until 1977, when Ziv and Lempel described an adaptive dictionary encoder [51]. Since that time, together with a different

¹Throughout this thesis the base of logarithms is 2

adaptive dictionary coding technique that came one year later [52], their work has been the basis for almost all practical adaptive dictionary encoders. This family of adaptive dictionary encoders is known as Ziv-Lempel coding, abbreviated as LZ coding. Welch introduced a very practical variant of Ziv-Lempel coding, that is known as the LZW algorithm [45].

Rissanen and Langdon first expressed that data compression process can be split into two parts: an encoder that actually produces the compressed bit-stream and a modeler that feeds information to it. These two separate tasks are called *coding* and *modeling*. Modeling assigns probabilities to symbols, and coding translates these probabilities to a sequence of bits [3].

Word-based text compression was studied by Ryabko, Bentley *et al.*, and Moffat [37, 4, 29]. Ryabko and Bentley *et al.* proposed a move-to-front (MTF) coding scheme, a technique that assigns shorter codes to more recently appeared words, and have given results that their scheme can represent English text in 3 to 4 bits per character. Moffat made a word-based implementation of adaptive arithmetic coding and attained compressed representation of English text requiring as little as 2.2 bits per character.

1.3.2 Compressing the Main Text

Many implementations have been made investigating the compression of the main text of an FTR system using the information stored at the index. It has been shown that good compression can be achieved by coding words based on their frequency [47, 48, 30, 53]. Witten *et al.* have investigated the use of arithmetic coding with the semi-static zero-order word model [47, 48]. Moffat and Zobel investigated the performance of Huffman coding and compared their results with the performance of several other compression schemes, including the Unix utility *Compress*; *ZeroWord*, a zero-order word-based adaptive arithmetic encoder, and *PPMC*, a variable context character-based model [30, 53].

All these experiments showed that the approach of using the lexicon as a semi-static word-based model results in good compression performance, in terms of both time and space.

Bookstein *et al.* proposed an algorithm based on Markov-modeled Huffman coding on an extended alphabet and obtained good compression with relatively slower encoding and decoding speed [7].

1.3.3 Compressing the Text Index

Several authors have proposed storing the differences between consecutive entries rather than the document numbers in the lists of an inverted file [42, 15, 6, 47, 48, 30]. In fact this is the same as the run-length encoding of zeros in the corresponding bitvectors [31, 54]. Then the problem of compressing the inverted lists is reduced to forming a good model for these interword gaps—the run lengths in the bitmap. Several methods have been proposed for modeling the interword gaps.

A simple technique to represent the run lengths is to use the universal codes discovered by Elias [11]. Moffat and Zobel implemented these techniques and compared them with several others [31, 54].

The simplest model to estimate the run length probabilities is to assume that a particular term's occurrence probability is constant for each document and independent among the documents throughout the collection. Then the probability distribution function for the run lengths is the geometric distribution with the probability of an interword gap of size k being $(1 - p)^{k-1}p$, where p is the number of documents including the term divided by the total number of documents.

Witten *et al.* and Bookstein *et al.* independently investigated coding the inverted file with arithmetic coding with respect to the geometric distribution model [47, 48, 6]. Their experiments showed that the concordance can be

stored in less than half of its uncompressed size.

The geometric distribution also yields a surprisingly effective infinite Huffman code. Golomb [19], and Gallager and Van Voorhis [18] describe a b -block code, in which a positive integer x is coded as $(x - 1) \text{ div } b$ bits set to one, followed by a zero bit, followed by $(x - 1) \text{ mod } b$ coded in binary. They proved that if b is chosen to satisfy the inequality

$$(1 - p)^b + (1 - p)^{b+1} \leq 1 < (1 - p)^{b-1} + (1 - p)^b,$$

this generates the infinite Huffman code for the geometric distribution. Moffat and Zobel [31, 54] applied this coding scheme to inverted files. They report compression ratios similar to those obtained by arithmetic coding with much higher coding and decoding speeds.

In practice the assumption that the one bits are uniformly and independently distributed within a bitvector is quite unrealistic. The natural ordering of the documents means that most of the terms will be relatively frequent over some sections of the collection, and relatively infrequent in the remainder. Teuhola [44] described an encoding similar to the Golomb codes but which also exploits the skewness in the run lengths. Moffat and Zobel [31, 54] showed that this scheme gives significantly better results than the Golomb code.

Another model that assumes the skewness of the bitvectors is the hyperbolic distribution model proposed by Schuegraf [42]. Bell *et al.* reported this model gives better compression than the geometric distribution model, but is more complex to implement [2].

Bookstein and Klein [5] has developed models which exploit possible correlations between rows and between columns of a bitmap. They tested their models with Shannon-Fano, Huffman and arithmetic coding. They reported improvements over previous methods.

Another alternative is to use an exact model that gives the exact number of occurrences of all run length values. Huffman coding is preferred to arithmetic

coding for coding the run lengths with respect to an exact model [54, 2]. This approach is implemented by Fraenkel and Klein [15] and by Moffat and Zobel [31, 54]. Experiments showed that exact modeling gives better compression than the other techniques. The major drawback of this approach is that it requires two passes, one for modeling and one for coding, and it is not suitable if the updates are frequent.

A completely different approach to compress sparse bitmaps is proposed by Choueka *et al.* [10]. They propose a tree representation that enables fast random access to a compressed bitmap. The bits of the map become leaves whose parent nodes are the disjunction of their values. This continues recursively up to the root. A zero at any node indicates that all its descendants are also zero, obviating the need to inspect lower levels when searching for a term. Nodes containing zeros can then be deleted, and nodes that contain few ones can be replaced with a short list of their positions. In this manner the bitmap is compressed. However the reported compression is not as good as the ones discussed above.

Another redundancy in the inverted files occurs in multi-level indexes. These indexes provide positional information about several levels of the text—for example, a word number within a sentence number within a paragraph number within a section number within a chapter number. In this case each pointer in the list will be a k -tuple in a k -level index. In such an index the higher level coordinates of consecutive entries would usually be the same. An obvious method to remove this redundancy is to replace the common fragment numbers occurring in consecutive entries with a flag of a few bits that tells how many coordinates are the same as the coordinates of the previous entry. This technique is known as the prefix omission technique (POM) and different variants have been studied by several authors [9, 26].

Chapter 2

Compression Techniques

2.1 Huffman Coding

Let a source S output independently chosen messages from the set $M = \{m_1, m_2, \dots, m_n\}$, with respective probabilities p_1, p_2, \dots, p_n . In his seminal paper, Shannon showed that the expected number of bits used to represent the messages m_i 's, cannot be less than $-\sum_{i=1}^n (p_i \cdot \log p_i)$ [43]. The quantity $-\sum_{i=1}^n (p_i \cdot \log p_i)$ is known as the entropy of the source S and shown as $H(S)$ or $H(p_1, p_2, \dots, p_n)$ [1].

A set of binary strings $C = \{c_1, c_2, \dots, c_n\}$ is called a code for the source S , if each message m_i is to be coded into c_i . A code C is called prefix code or instantaneous code if no codeword is a prefix of another codeword.

Huffman [22] gave an algorithm to produce prefix codes with minimal expected codeword lengths. The algorithm is easy to implement and the code it generates is optimal given that each message must be coded with an integral number of bits. Later, Gallager showed that the redundancy of Huffman codes, defined as the average code length less the entropy, is bounded above by $p_{max} + 0.086$, where p_{max} is the probability of the most likely message [17]. The average length of Huffman codes is equal to the entropy if occurrence probability

of each message is a negative power of 2.

Huffman's algorithm begins with the following construction:

```

construct_Huffman_tree()
 $T \leftarrow \{\{m\} : m \in M\}$ 
repeat n-1 times,
  set  $s_1$  and  $s_2 \leftarrow$  the two sets of least probability in  $T$ 
   $T \leftarrow T \cup \{\{s_1, s_2\}\} - \{s_1\} - \{s_2\}$ 
   $p(\{s_1, s_2\}) \leftarrow p(s_1) + p(s_2)$ 

```

This procedure produces a recursively structured set of sets, each of which contains exactly two members. It can therefore be represented as a binary tree with the original messages at the leaves. Then codes are assigned to messages by the following algorithm:

```

assign_codes()
construct_Huffman_tree()
for each message do
  Traverse the tree from the root to the message, recording 0 for a left
  branch and 1 for a right branch.

```

Figure 2.1 illustrates the process for an example message set.

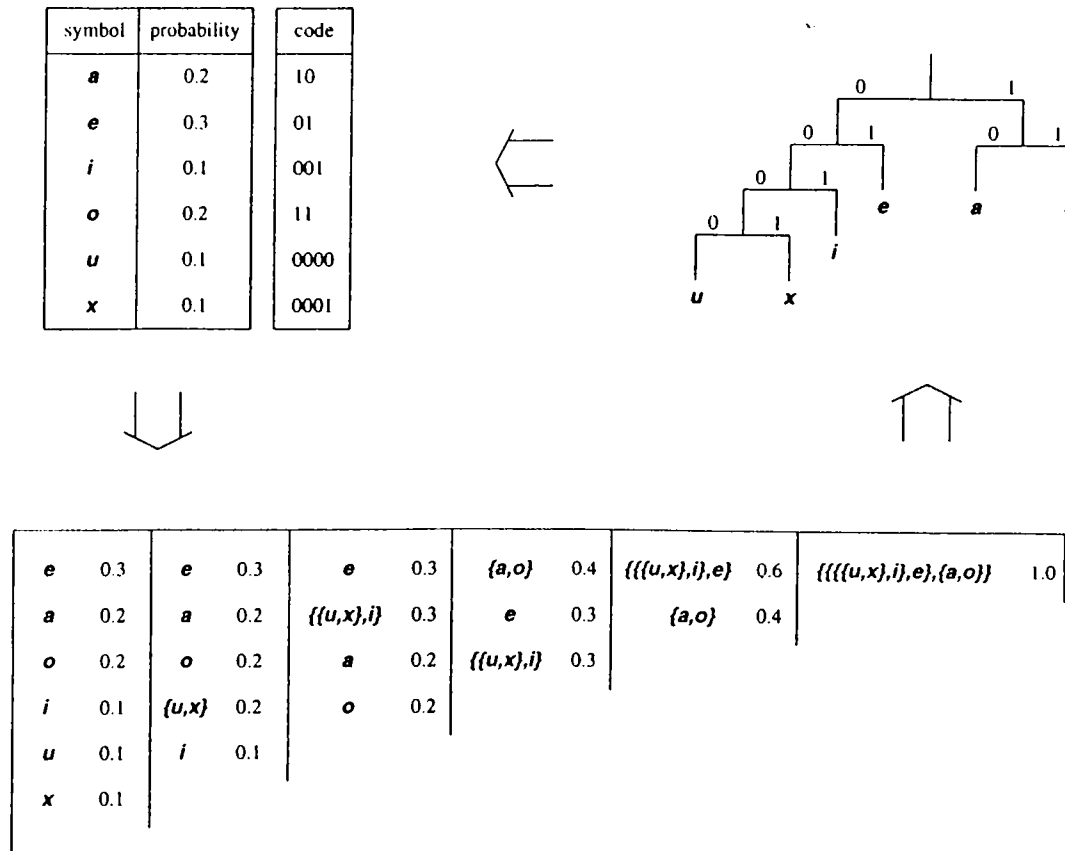


Figure 2.1: An example of Huffman coding

Decoding is done similarly. It can be done by the following algorithm if the Huffman tree is available at decoding time:

```

decode_message()
node ← root
while node is not a leaf node do
    bit ← next_input_bit()
    if bit=0 then
        node ← left[node]
    else
        node ← right[node]
return(message[node])
    
```

The main problem of Huffman codes is the decoding procedure. Keeping the code tree may be an easy solution if the total number of messages is small (e.g. 128 ASCII characters). But this solution is quite wasteful when the total number of messages is large. There is a slightly different representation of Huffman codes that decodes very efficiently despite the extremely large models that can occur in FTR systems. This representation is known as the canonical Huffman code. It uses the same codeword lengths as a Huffman code, but imposes a particular choice on the codeword bits. The canonical Huffman algorithm is as follows:

```

assign_codes()
construct_Huffman_tree()
Use the Huffman tree to find the code length for each message and
  keep the total number of messages of each code length in
  the array numl[min_length, max_length]
for  $\ell = \text{max\_length}$  downto  $\text{min\_length}$  do
  if  $\ell = \text{max\_length}$  then
     $\text{first\_code}[\ell] \leftarrow 0$ 
  else
     $\text{first\_code}[\ell] \leftarrow (\text{first\_code}[\ell + 1] + \text{numl}[\ell + 1])/2$ 
     $\text{next\_code}[\ell] \leftarrow \text{first\_code}[\ell] + 1$ 
  for each message  $m$  with the code length  $\ell$  do
    Assign the code  $\text{next\_code}[\ell]$ , represented in  $\ell$  bits, to  $m$ 
     $\text{next\_code}[\ell] \leftarrow \text{next\_code}[\ell] + 1$ 
for each length  $\ell$  of which no codeword exists do
   $\text{first\_code}[\ell] \leftarrow 2^\ell$ 

```

The algorithm above uses the Huffman tree to find the length of the codewords. After that, it processes the messages in descending codeword length, and assigns the smallest available codeword to each message. An available codeword is one which is not the prefix of another. Figure 2.2 illustrates the process for our example message set.

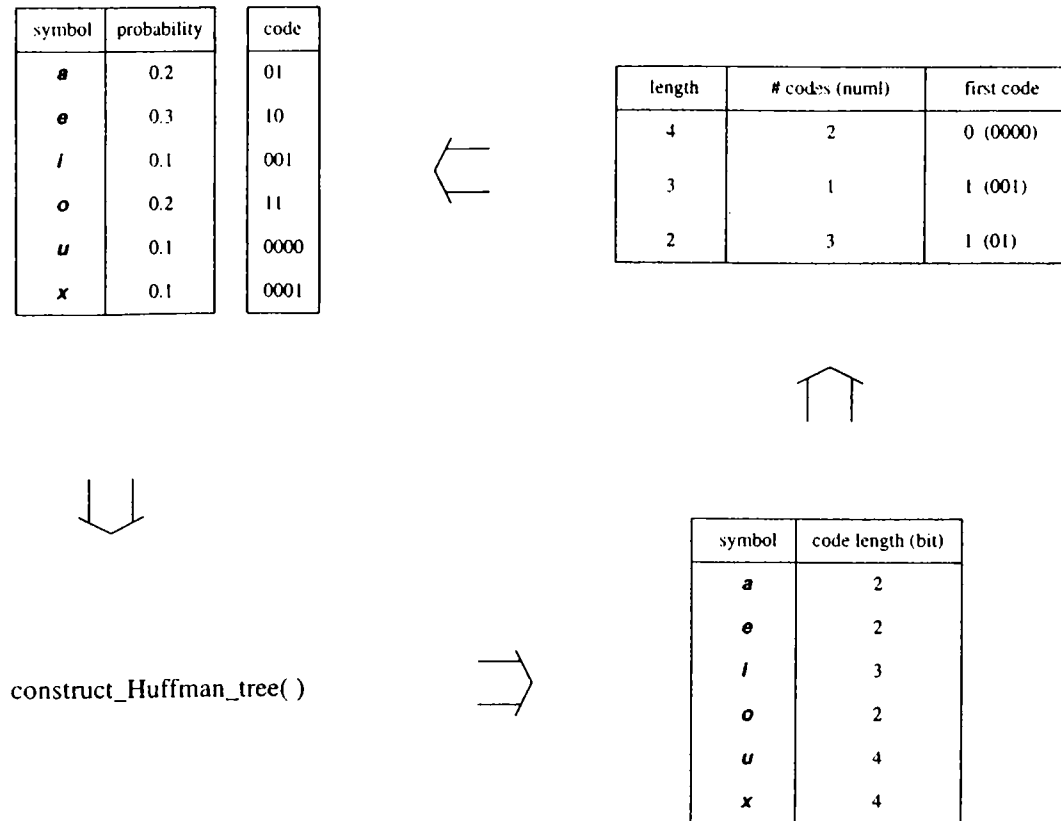


Figure 2.2: An example of canonical Huffman coding

The code generated by the algorithm above can be decoded very fast without the code tree if the information in the array *first_code* is stored together with the code table. Decoding a message can be achieved by the following algorithm:

```

decode_message()
code ← next_input_bit()
length = 1
while code < first_code[length] do
    code ← 2 · code + next_input_bit()
    length ← length + 1
return the message whose codeword is code

```

Compression and decompression algorithms for the Huffman codes are quite straight forward.

```
compress_huff()  
assign_codes()  
while not end of stream  
     $m \leftarrow$  read_next_message()  
    output codeword[ $m$ ]  
encode a special end of stream symbol
```

```
decompress_huff()  
repeat  
     $m \leftarrow$  decode_message()  
    if  $m$  is the end of stream symbol then  
        break  
    else  
        output  $m$ 
```

2.2 Arithmetic Coding

Arithmetic coding dispenses with the restriction that messages translate into an integral number of bits. It actually achieves the theoretical entropy bound for any source, with a small termination overhead of maximum two bits.

In arithmetic coding a stream is represented by an interval of real numbers between 0 and 1. As the stream becomes longer, the interval needed to represent it becomes smaller, and the number of bits needed to specify that interval grows. Successive messages of the stream reduce the size of the interval in accordance with the message probabilities generated by the model. The more likely messages reduce the range by less than unlikely messages and hence add fewer bits to the stream.

Before anything is transmitted, the range for the stream is the entire half open interval from zero to one, $[0,1)$. As each message is processed, the range is narrowed to the portion of it allocated to the message. In Figure 2.3 arithmetic coding process of a string beginning with *aaba* is illustrated where individual symbol probabilities of *a* and *b* are 0.6 and 0.4 respectively.

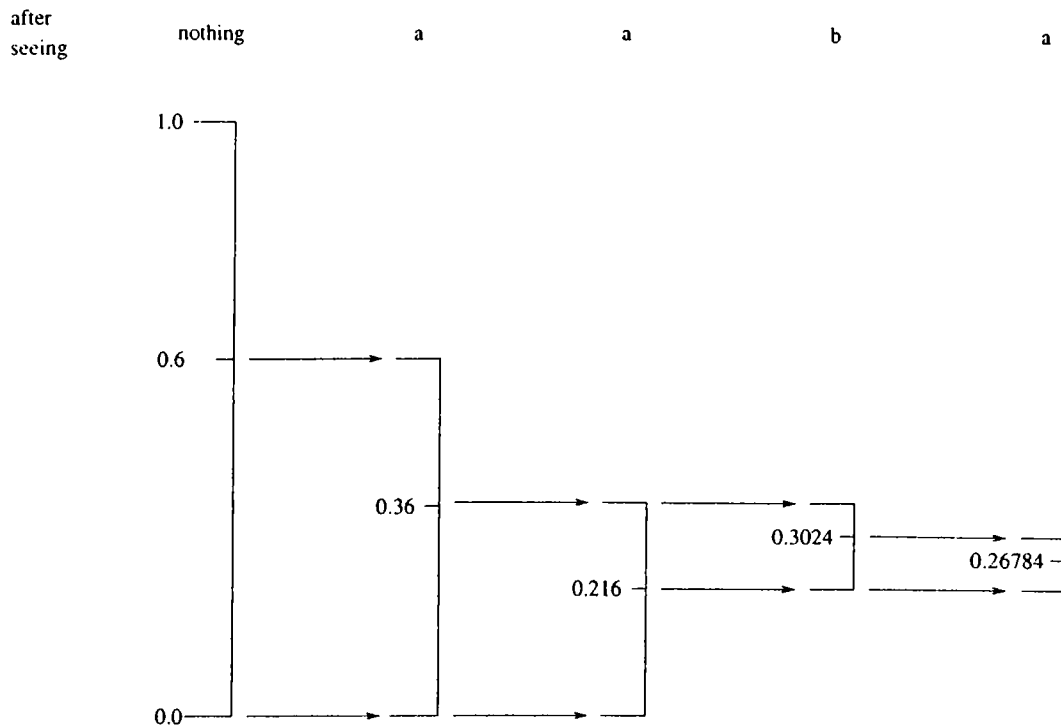


Figure 2.3: Representation of the arithmetic coding process

It is not really necessary for the decoder to know both ends of the range produced at the end of encoding. Instead, a single number within the range will suffice. To resolve the ambiguity, a special termination symbol must be encoded at the end of the stream.

The compression and decompression algorithms for the arithmetic coding are as follows:

compress_arth()

Set the working interval $work_int \leftarrow [0,1)$
while not end of stream
 $m \leftarrow \text{read_next_message}()$
 set $work_int \leftarrow$ the range in $work_int$ that corresponds to the message m
 Transmit any number in $work_int$ as the output

decompress_arth()

read the number $value$ in $[1,0)$ representing the message ensemble
 set $work_int \leftarrow [0,1)$
repeat:
 Find the message m for which the corresponding interval in
 $work_int$ includes $value$
 if m is the termination symbol **then**
 break
 else
 output the message m
 set $work_int \leftarrow$ the range in $work_int$ that corresponds
 to the message m

2.2.1 Incremental Transmission and Reception

The algorithm above is overly simplistic. An important question arising is how to represent the shrinking interval in $[0,1)$ as the process advances. Any finite-precision representation will be inadequate after a certain point. An approach to solve this problem is the incremental transmission and reception of the code. That is to encode and decode each bit as soon as it is determined. For example, consider the encoding situation where $work_int$ is completely included in the interval $[0, 0.5)$. Since the final code must be within this range, we can be certain that it will begin with “0.0...” in binary representation. Similarly if $work_int$ is completely included in the interval $[0.75, 1)$, then we can guarantee

that the final code begins with “0.11...”. The decoder can also interpret the codes incrementally in a similar fashion. It receives the final code bit by bit and decodes a message as soon as it is determined. If coding is incremental, it can be performed using finite-precision arithmetic, because once a digit has been transmitted it will have no further influence on the calculations. For example, if the “0.11” of the interval $[0.11001, 0.11100)$ has been sent, future output would not be affected if the interval were changed to $[0.00001, 0.00100)$ or even $[0.001, 0.100)$, thus decreasing the precision of the arithmetic required.

To adapt the given algorithms with respect to incremental transmission and reception let *low* and *high* represent the low and high end points of *work_int* respectively. The following step must be appended into the while loop in the encoding algorithm.

```

while high < 0.5 or low > 0.5 do
  if high < 0.5 then
    outputbit(0)
    low ← 2 · low
    high ← 2 · high
  if low > 0.5 then
    outputbit(1)
    low ← 2 · (low - 0.5)
    high ← 2 · (high - 0.5)

```

And the last step must be replaced by the termination step which will be discussed later.

For incremental reception, the following step must be inserted into the decoding algorithm just after the second clause. The `include_next_bit()` procedure reads the next input bit to the least significant bit of *value*, the number representing the message ensemble.


```

while high < 0.5 or low > 0.5 do
  if high < 0.5 then
    value ← 2 · value
    include_next_bit()
    low ← 2 · low
    high ← 2 · high
  if low > 0.5 then
    value ← 2 · (value - 0.5)
    include_next_bit()
    low ← 2 · (low - 0.5)
    high ← 2 · (high - 0.5)

```

2.2.2 The Underflow Problem

The encoder must guarantee that *work_int* is always large enough to maintain the adequacy of the finite-precision arithmetic. Incremental transmission and reception guarantees that *work_int* will be expanded as long as it completely falls into one of the upper and lower halves. So we know that *low* and *high* can only become close together when they straddle 0.5. Suppose that, in fact, they become as close as

$$0.25 \leq low < 0.5 \leq high < 0.75$$

Then the next two bits sent will have opposite polarity, either 01 or 10. For example, if the next bit turns out to be 0 (i.e. *high* descends below 0.5 and $[0, 0.5)$ is expanded to $[0, 1)$), the bit after that will be 1, since the range has to be above the midpoint of the expanded interval. Conversely, if the next bit happens to be 1, the one after that will be 0. Therefore, the interval can safely be expanded right now, if only we remember that whatever bit actually comes next, its opposite must be transmitted afterward as well. In this situation we simply expand $[0.25, 0.75)$ to $[0, 1)$ remembering in a variable – we will call it *BitsToFollow* – that the bit that is output next must be followed by an opposite bit.

```

if  $0.25 \leq low$  and  $high \leq 0.75$  then
     $BitsToFollow \leftarrow 1$ 
     $low \leftarrow 2 \cdot (low - 0.25)$ 
     $high \leftarrow 2 \cdot (high - 0.25)$ 

```

But what if, after this operation, it is still true that

$$0.25 \leq low < 0.5 \leq high < 0.75 ?$$

Figure 2.4 illustrates this situation, where the current *work_int* has been expanded a total of three times. Suppose that the next bit will turn out to be 0, as indicated by the arrow in Figure 2.4.a being below 0.5. Then the next three bits will be 1's, since not only is the arrow in the top half of the bottom half of $[0,1)$, it is in the top quarter, and moreover in the top eighth, of that half—that is why the expansion can occur three times. Similarly, as Figure 2.4.b shows, if the next bit turns out to be 1, it will be followed by three 0's. Consequently, we need only count the number of expansions and follow the next bit by that number of opposites, replacing the code fragment above by

```

while  $0.25 \leq low$  and  $high \leq 0.75$  then
     $BitsToFollow \leftarrow BitsToFollow + 1$ 
     $low \leftarrow 2 \cdot (low - 0.25)$ 
     $high \leftarrow 2 \cdot (high - 0.25)$ 

```

Using this technique, the encoder guarantees that after the shifting operations,

$$\text{either } low < 0.25 < 0.5 \leq high, \quad (2.1)$$

$$\text{or } low < 0.5 < 0.75 \leq high. \quad (2.2)$$

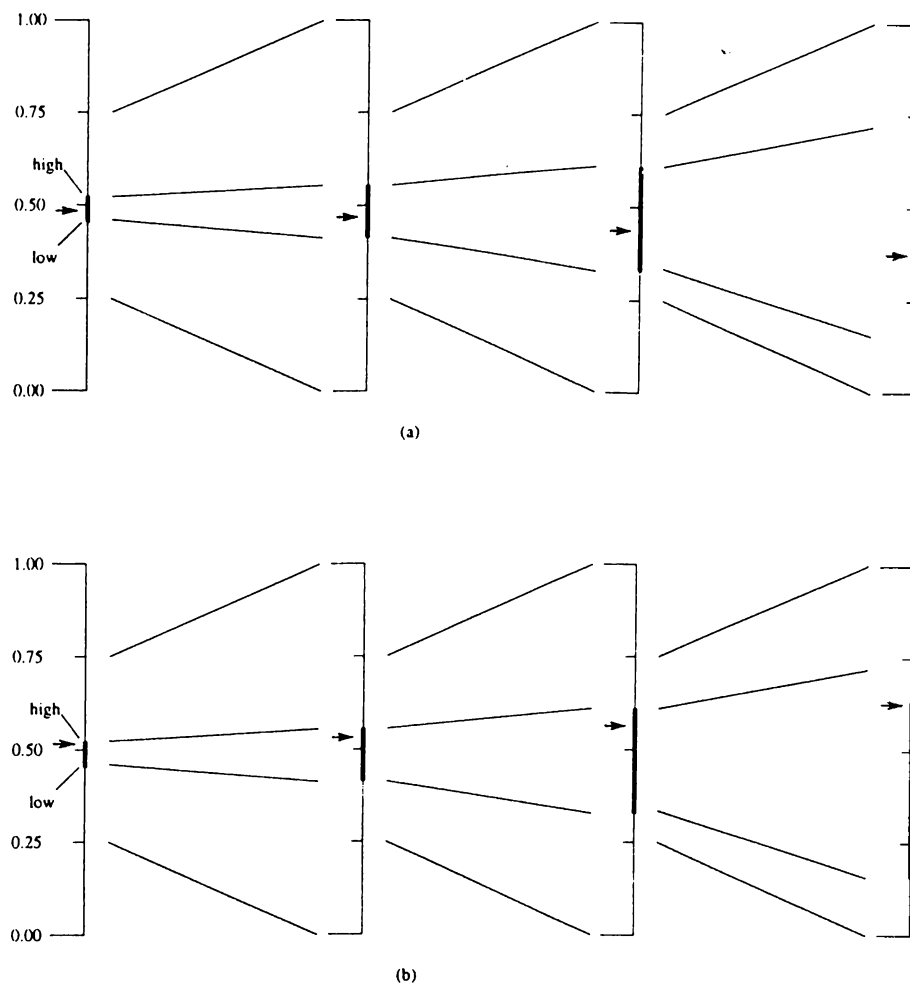


Figure 2.4: Scaling the interval to prevent underflow

2.2.3 Terminating the Message

To finish a transmission, it is necessary to send a unique terminating symbol and then follow it by enough bits to ensure that the encoded string falls within the final range. After the terminating symbol has been encoded, *low* and *high* are constrained by either (2.1) or (2.2) above. Consequently it is only necessary to transmit 01 in the first case and 10 in the second to remove the remaining ambiguity.

The decoder's `include_next_bit()` procedure will actually read a few more bits than were sent by the encoder's `outputbit()`. It does not matter what value these bits have, because the termination symbol is uniquely determined by the

last two bits actually transmitted.

Witten *et al.* present a full description of arithmetic coding and discusses further details for implementation [50].

2.3 Dictionary Encoders and Ziv-Lempel Coding

Dictionary-based compression methods use the principle of replacing substrings in a text with a codeword that identifies that substring in a “dictionary”, or “codebook”. The dictionary contains a list of substrings, and a codeword for each substring. This type of substitution is used naturally in everyday life, for example, in the substitution of the number 12 for the word *December*. Unlike statistical coding techniques such as arithmetic coding or Huffman coding, dictionary methods often use fixed-length codewords.

The simplest dictionary compression methods use small codebooks. For example, in *digram coding*, selected pairs of letters are replaced with codewords. A codebook for the ASCII character set might contain the 128 ASCII characters, as well as 128 common letter pairs. The output codewords are 8 bit each, and the presence of the full ASCII character set in the codebook ensures that any input can be represented. At best, every pair of characters is replaced with a codeword, reducing the input from 7 bits per character to 4 bits per character. At worst, each 7-bit characters will be expanded to 8 bits. Furthermore, a straightforward extension caters to files that might contain some non-ASCII bytes –one codeword should be reserved as an escape, to indicate that the next byte should be interpreted as a single 8-bit character rather than as a codeword for a pair of ASCII characters. Of course, a file consisting of mainly binary data will be expanded significantly by this approach; this is the inevitable price that must be paid for a static model.

Another natural extension of this system is to put even larger entries in the codebook –perhaps common words like *and* and *the*, or common components of words such as *pre* and *tion*. Strings like these that appear in the dictionary are sometimes called *phrases*. A phrase may sometimes be as short as one or two characters, or it may include several words. Unfortunately, having a dictionary with a predetermined set of phrases does not give very good compression, because the entries must usually be quite short if input-independence is to be achieved. In fact, the more suitable the dictionary is for one sort of text, the less suitable it is for others. For example, if this thesis were to be compressed, then one would do well if the codebook contained phrases related to compression, but such a codebook may be unsuitable for a text on linear programming.

One way to avoid the problem of the dictionary being unsuitable for the text at hand is to use a semi-static dictionary scheme, constructing a new codebook for each text that is to be compressed. However, the overhead of transmitting or storing the dictionary is significant, and deciding which phrases should be put in the codebook to maximize compression is a difficult problem.

An elegant solution to this problem is to use an *adaptive* dictionary scheme. Practically all adaptive dictionary compression methods are based on one of just two related methods developed by Ziv and Lempel in the 1970s. These methods are usually labeled as LZ77 and LZ78, depending on the years in which they were published. These methods are the basis for many schemes that are widely used in utilities for compression and archiving, although they have undergone much fine-tuning since their invention.

Both methods use a simple principal to achieve adaptivity: a substring of text is replaced with a pointer where it has occurred previously. Thus, the codebook is essentially all the text prior to the current position, and the codewords are represented by pointers. The prior text makes a very good dictionary, since it is usually in the same style and language as upcoming text; furthermore, the dictionary is transmitted implicitly at no cost, because the decoder has access to all previously encoded text. The many variants of Ziv-Lempel coding differ primarily in how pointers are represented, and in the limitations they impose

on what the pointers are able to refer to. Bell *et al.* give detailed discussions of these techniques [1, 3].

2.3.1 LZW

LZW is one of the most widely known variants of Ziv-Lempel coding. It has been used as the basis of several popular programs, including the Unix *compress* program and some personal computer archiving systems.

The main difference between LZW and LZ78 is that LZW encodes only the phrase numbers, and does not have explicit characters in the output. This is made possible by initializing the list of phrases to include all characters in the input alphabet. LZW uses the *greedy* parsing algorithm, where the input string is examined character-serially in one pass, and the longest recognized input string is parsed off each time. A recognized string is one that exists in the string table. The strings added to the string table are determined by this parsing: Each parsed input string extended by its next input character forms a new string added to the string table. Each such added string is assigned a unique identifier, namely its code value. In precise terms, this is the algorithm:

compress_LZW()

```

Initialize the table to contain single-character strings
read first input character c
set the prefix string  $\omega \leftarrow c$ 
while not end of stream
    read next input character c
    if  $\omega c$  exists in the string table then
         $\omega \leftarrow \omega c$ 
    else
        output code( $\omega$ )
        append  $\omega c$  to the string table
         $\omega \leftarrow c$ 
output code( $\omega$ )

```

At each iteration of the while loop an acceptable input string ω has been parsed off. The next character c is read and the extended string ωc is tested to see if it exists in the string table. If it exists, then the extended string becomes the parsed string ω and the step is repeated. If ωc is not in the string table, then it is entered, the code for the successfully parsed string ω is put out as the compressed data, the character c becomes the beginning of the next string, and the step is repeated. An example of this procedure is shown in Figure 2.5. For simplicity a two-character alphabet is used.

INPUT SYMBOLS	a	a	b	a	b	a	b	a	a	a			
OUTPUT CODES	0	0	1	3	5		2						
NEW STRINGS ADDED TO TABLE	3		5			2					4	6	

Figure 2.5: LZW coding of the string “aabababaaa” (phrases 0 and 1 are present before coding begins)

This algorithm makes no real attempt to optimally select strings for the string table or optimally parse the input data. It produces compression results that, while less than optimum, are effective.

A source is said to be ergodic if any sequence it produces becomes entirely representative of the source as its length tends to infinity. An important theoretical property of LZW (in fact of LZ78) is that when the input text is generated by a stationary, ergodic source, compression is asymptotically optimal as the size of the input increases. That is, LZW will code an indefinitely

long string in the minimum size dictated by the entropy of the source. In fact very few coding methods enjoy this property [1].

Decompression: The LZW decompressor logically uses the same string table as the compressor and similarly constructs as the message is translated. Each received code value is translated via the string table.

An update to the string table is made for each code received (except the first one). When a code has been translated, its initial character is used as the extension character, combined with the prior string, to add a new string to the string table. This new string is assigned a unique code value, which is the same code that the compressor assigned to that string. In this way, the decompression incrementally reconstructs the same string table that the compressor used.

The basic algorithm can be stated as follows:

```
decompress_LZW()
last_code ← code ← read first input code
output string_table[code]
while not end of stream
    code ← read next input code
    output string_table[code]
    append "string_table[last_code],c" to string_table where
        c is the first character of string_table[code]
    last_code ← code
```

Unfortunately, this simple algorithm has a complicating problem. The problem occurs when a new phrase is used by the encoder immediately after it is constructed. In this case the decoder reads a code for which no entry corresponds in the string table yet. To tackle this difficulty the *while* clause of the decoding algorithm must be extended as follows:


```

while not end of stream
  code ← read next input code
  if code not defined (special case) then
    output string_table[last_code]
    output the first character of string_table[last_code]
    append “string_table[last_code],c” to string_table where
      c is the first character of string_table[last_code]
    last_code ← code
  else
    output string_table[code]
    append “string_table[last_code],c” to string_table where
      c is the first character of string_table[code]
    last_code ← code

```

A decompression example is shown in Figure 2.6. Decoding of the phrase “aba” illustrates the tricky case mentioned above.

INPUT						
CODES	0	0	1	3	5	2
OUTPUT						
STRINGS	a	a	b	ab	aba	aa
NEW STRINGS			3		5	
ADDED TO						
TABLE	2		4		6	

Figure 2.6: LZW decoding of the string “001352” (phrases 0 and 1 are present before decoding begins)

Chapter 3

Implementation

3.1 Compression Schemes

The Index. The test databases are indexed using an inverted file index structure. Every string of alphabetic characters is indexed without case folding and stemming. Stemming is the automated conflation of related words, usually by reducing the words to a common root form [16]. Case folding is converting all characters to either upper or lower case. Each index term is stored at the vocabulary together with its frequency count. Within fragment frequencies of the index terms are also stored in the inverted lists together with the pointers. These frequency counts may be used to support ranked queries [41, 16].

Compression. The information of index terms and their overall frequencies, which is provided by the index, is used as a word-based semi-static model. Zero-order word-based applications of Huffman coding, canonical Huffman coding, arithmetic coding and LZW coding have been investigated. LZW has also been implemented as character-based. Zero-order word-based approach is to take each word as a token instead of individual characters. Non-alphabetic characters are also taken as tokens since they are not included in the index. The Unix utilities *Compress* and *Ucbcompress* are also included in performance

comparisons.

Several versions of arithmetic coding have been proposed in the literature [1]. The one applied in this study depends on the algorithm proposed by Witten *et al.*[50, 1]. This version brings the restriction that the number of bits used to represent the cumulative frequencies of tokens can be at most one less than half of the number of bits used to represent the maximum integer [50, 1]. This implies that software arithmetics must be used at machines providing hardware arithmetic operations on 32-bit integers maximum, to handle large collections with cumulative frequency of all tokens larger than 2^{15} . The first two implemented versions use software arithmetics that is developed in this study (see Appendix A). The third one uses hardware arithmetics and it can be considered only for small collections if the machine used provides hardware arithmetic operations on 32-bit integers maximum. This version is not applied to the *tb* database which totally includes 453867 words and 782157 non-alphabetic characters. The difference between the first and the second versions is that the first keeps a table of cumulative frequencies, where the second calculates it from the token frequencies of the vocabulary at the beginning of decompression. The third version also calculates cumulative frequencies at the beginning of decompression, like the second version.

LZW is applied both as character-based and word-based. In both applications 16-bit codewords are used. Although the character-based implementation completely disregards every information stored at the index, it is included in this study. The reason for this is that if the database is not static and a document insertion with new words occurs then the whole collection must be recompressed if it was previously compressed with respect to the word-based model provided by the index. Therefore, a word-based semi-static model is not convenient if document insertions with new words are likely to occur. So, this technique is hardly of any use for a static database, but may be the most convenient one if document insertions are frequent. This technique deserves attention for one more thing. That is the character-based LZW is a one-pass approach and makes the compression and indexing simultaneously. So it can be considered as a fast alternative that gives relatively poor compression.

Several data structures have been proposed for Ziv-Lempel coding [1]. One of the most convenient is a *trie* structure. Also there are several ways to represent trie nodes [1]. We have implemented LZW using trie structures. Nodes of tries are implemented as linked lists for LZW_word(1) and LZW_char(1) and as binary search trees for LZW_word(2) and LZW_char(2).

The Unix utilities *Compress* and *Ucbcompress* operate on files and cannot compress and decompress fragments within a file separately. To test their performance on the test databases, each fragment is copied to an artificial file and then compressed using the Unix utilities. Therefore, their time figures do not provide a good basis for comparison.

3.2 Test Databases

Three different text files are used to test the programs; *alice13a.txt*, *un.dalamp* and *tb*. The first one is Lewis Carroll's *Alice in Wonderland*, the second is a text about finding e-mail addresses of users at universities from all around the world, and the third is the collection of TidBITS, an electronic weekly newsletter. Concise information about the test databases are given in Table 3.1.

	Size (in KByte)	# words		# non-alphabetic characters	
		Distinct	Total	Distinct	Total
<i>alice13a.txt</i>	150	2976	26863	27	44197
<i>un.dalamp</i>	190	4598	28651	43	53597
<i>tb</i>	2900	21931	453867	44	782157

Table 3.1: Information about test databases

Each text is fragmentized twice; *alice13a.txt* and *un.dalamp* with average fragment size of 1 KByte and 10 KByte, and *tb* with average fragment size of 10 KByte and 100 KByte.

3.3 Results

Programs are implemented on Sun SPARCstations with CPU architecture SUNW 4/25 in a single user environment. The results are summarized in the following tables. Compression ratio is defined as the proportion of the size of the compressed text to the size of the original text. The time figures given are for the whole collections. In the tables, the columns *user* and *system* show the time figures given by the Unix *time* command. User time is the CPU time devoted to the user's process. System time is the CPU time consumed by the kernel on behalf of the user's process. Each of these figures are in seconds.

	Comp. Time (sec.)		Decomp. Time (sec.)		Comp. Ratio	Code Table Overhead
	User	System	User	System		
Arth(1)	14	1	53	80	32%	8%
Arth(2)	14	1	54	78	32%	-
Arth(3)	8	1	15	44	32%	-
Can. Huff.	20	1	6	58	32%	8%
Huffman	23	1	6	65	32%	16%
LZW_word(1)	59	1	19	12	65%	-
LZW_word(2)	4	1	19	12	65%	-
LZW_char(1)	12	1	3	1	110%	-
LZW_char(2)	4	0	3	1	110%	-
compress	6	21	4	20	57%	-
ucbcompress	5	19	4	21	66%	-

Table 3.2: Experiment results for *alice13a.txt* (av. frag. size = 1 KByte)

	Comp. Time (sec.)		Decomp. Time (sec.)		Comp. Ratio	Code Table Overhead
	User	System	User	System		
Arth(1)	16	1	52	80	32%	8%
Arth(2)	15	1	52	78	32%	-
Arth(3)	8	1	15	44	32%	-
Can. Huff.	21	1	5	57	32%	8%
Huffman	23	1	5	64	32%	16%
LZW_word(1)	50	1	3	2	48%	-
LZW_word(2)	3	1	3	2	48%	-
LZW_char(1)	9	1	2	1	72%	-
LZW_char(2)	3	0	2	1	72%	-
compress	2	2	1	2	43%	-
ucbcompress	1	2	1	3	51%	-

Table 3.3: Experiment results for *alice13a.txt* (av. frag. size = 10 KByte)

	Comp. Time (sec.)		Decomp. Time (sec.)		Comp. Ratio	Code Table Overhead
	User	System	User	System		
Arth(1)	19	1	65	116	32%	10%
Arth(2)	18	1	66	114	32%	-
Arth(3)	10	1	23	65	32%	-
Can. Huff.	43	1	7	70	33%	10%
Huffman	46	1	7	73	33%	20%
LZW_word(1)	131	1	44	29	60%	-
LZW_word(2)	6	1	44	29	60%	-
LZW_char(1)	16	1	5	1	114%	-
LZW_char(2)	5	0	5	1	114%	-
compress	8	29	5	25	56%	-
ucbcompress	6	26	5	27	69%	-

Table 3.4: Experiment results for *un.dalamp* (av. frag. size = 1 KByte)

	Comp. Time (sec.)		Decomp. Time (sec.)		Comp. Ratio	Code Table Overhead
	User	System	User	System		
Arth(1)	18	1	63	114	32%	10%
Arth(2)	18	1	64	112	32%	-
Arth(3)	11	1	21	62	32%	-
Can. Huff.	43	1	65	64	32%	10%
Huffman	47	1	7	70	32%	20%
LZW_word(1)	118	1	6	3	48%	-
LZW_word(2)	4	1	6	3	48%	-
LZW_char(1)	12	1	3	1	77%	-
LZW_char(2)	4	0	3	1	77%	-
compress	2	3	6	3	43%	-
ucbcompress	1	3	1	3	51%	-

Table 3.5: Experiment results for *un.dalamp* (av. frag. size = 10 KByte)

	Comp. Time (sec.)		Decomp. Time (sec.)		Comp. Ratio	Code Table Overhead
	User	System	User	System		
Arth(1)	468	3	1303	2242	33%	3%
Arth(2)	467	2	1266	2076	33%	-
Can. Huff.	527	30	112	1190	33%	3%
Huffman	442	28	108	1213	33%	6%
LZW_word(1)	10626	5	339	208	49%	-
LZW_word(2)	77	2	339	208	49%	-
LZW_char(1)	135	11	50	7	81%	-
LZW_char(2)	43	6	50	7	81%	-
compress	38	42	13	40	46%	-
ucbcompress	22	42	16	43	56%	-

Table 3.6: Experiment results for *tb* (av. frag. size = 10 KByte)

	Comp. Time (sec.)		Decomp. Time (sec.)		Comp. Ratio	Code Table Overhead
	User	System	User	System		
Arth(1)	467	2	1297	2219	33%	3%
Arth(2)	466	1	1251	2075	33%	-
Can. Huff.	509	31	102	1158	33%	3%
Huffman	435	27	100	1185	33%	6%
LZW_word(1)	9547	410	58	25	39%	-
LZW_word(2)	64	2	58	25	39%	-
LZW_char(1)	141	8	35	5	56%	-
LZW_char(2)	35	5	35	5	56%	-
compress	14	8	9	7	38%	-
ucbcompress	47	6	6	5	47%	-

Table 3.7: Experiment results for *tb* (av. frag. size = 100 KByte)

Chapter 4

Conclusions

The purpose of this study was to compare various coding techniques to compress the text database of an FTR system. For this purpose the index has been used as a semi-static word-based modeler and several variations of arithmetic coding, Huffman coding and LZW coding have been implemented.

The results reveal that both arithmetic coding (the third variant) and Huffman coding give good compression with similar decoding speed. They compress the English text being represented in 2.5 bits per character and this quantity is not very sensitive to the fragment size. Arithmetic coding is superior since it does not require keeping a code table and provides faster encoding. The main disadvantage of arithmetic coding is that, at machines providing hardware arithmetic operations on 32-bit integers maximum, the cumulative frequency of all tokens cannot exceed 2^{15} . Decoding time increases dramatically if software arithmetics is used to handle larger collections (the first and the second variants of arithmetic coding). Comparison of the performance of the first and the second variants reveal that keeping a table of cumulative frequencies, which may be as large as 10% of the text, does not significantly improve the decompression speed, and must not be preferred.

Performance comparison between Huffman coding and canonical Huffman coding shows that they achieve the same compression with similar encoding and

decoding speed. Canonical Huffman coding is superior because the codetable overhead is less, and is the most convenient method when the size of the text requires software arithmetics for arithmetic coding.

Performance of LZW is not good in compression ratio but gets better as the fragment size increases. This is due to the fact that LZW is an adaptive compression technique and normally gives better compression as the stream to be compressed gets larger. LZW's compression and decompression speed is relatively good but decompression speed of the word-based version significantly gets worse as the average fragment size decreases. This is due to the dictionary initialization that must be done at the beginning of decompression for each fragment. The system's standard compression programs behave similar to LZW. Their compression performance gets better as the fragment size increases. This is natural since they are variants of the Ziv-Lempel coding, like LZW. Especially *ucbcompress* uses the LZC variant which is a slightly improved version of LZW. The reason of the significant difference between the performances of the word-based LZW and *ucbcompress* is the word-based application of LZW. This increases the compression performance of LZW since it uses words instead of characters; but decreases both compression and decompression speed due to the initialization of the large dictionary. One more thing to note is the system's compression utilities operate on files and cannot compress and decompress fragments within a file separately. This is the main reason for the seeming slowness of the system's utilities, especially when the average fragment size is small. Therefore these time figures do not provide a good basis for comparison.

Character-based implementation of LZW shows that the information of existing words contributes greatly to the performance of LZW. Therefore, word-based LZW may be a convenient technique if term frequency information is not available in the index.

The character-based implementation of LZW can even increase the size of the text when the average fragment size is small. Therefore, using shorter codes instead of 16-bit codes may be more appropriate for compressing collections

of short documents. Still the speed figures are very meaningful and shows the improvement in compression time when a one-pass approach is used. But compression time is hardly of any concern for static FTR systems and may matter only if the database is dynamic and insertion of new documents with new words is likely to occur frequently.

Another noteworthy point is the great improvement in compression time obtained by the binary search tree implementation of the trie nodes for LZW. Word-based compression time of the *tb* collection has dropped from three hours to approximately one minute. This improvement is less significant for small vocabularies and character-based implementations, but is extremely significant for texts including a high number of distinct words.

Suggestions for Future Research

Performance of sophisticated adaptive models such as Dynamic Markov Compression (DMC) and Prediction by Partial Match (PPM) for compression of the text database of an FTR system may be analyzed [1]. These techniques give very good models on a long text, but compression is relatively poor at the initial steps. Therefore, an appropriate way to use these models may be to split the text into large blocks, each consisting of several documents; and compress each block independently. To access any part of the block, it must be decoded from the beginning. Then determination of the block size emerges as an essential tradeoff question between retrieval time and storage space.

Coding techniques used in this study may be used in conjunction with different types of inverted files which does not give the exact information of term frequencies. A challenging and common situation is index terms may be stemmed or case-folded. In this case some sophistication is needed to use the index as a modeler.

Appendix A

Software Arithmetics for Arithmetic Coding

Consider the problem of finding the exact result of the operation $(a \cdot b)/c$ ¹, where all a , b and c are integers known to be less than half of max_int , the maximum integer representable in the registers, but $a \cdot b$ is larger than max_int . One more thing that is known is either a or b or both are less than c , and c is greater than zero, so that the result of $(a \cdot b)/c$ is also less than max_int . Such a problem arises several times in the decoding procedure of arithmetic coding, the version proposed by Witten *et al.* [50, 1] and applied in this study.

First calculating the decimal value of a/c and then multiplying this with b usually does not return the exact result because of the finite-precision floating point operations involved. To find the exact value we have developed an algorithm that uses some basic facts from the elementary number theory. For convenience, assume that $a \leq b$. Let b be equal to $d_1 \cdot c + r_1$; where d_1 and r_1 are the quotient and the remainder resulting from the division of b by c . Then the problem of finding $(a \cdot b)/c$ becomes the problem of finding $a \cdot d_1 + (a \cdot r_1)/c$. If $a \cdot r_1$ is less than max_int , then the rest is straight forward. Otherwise, some more effort is needed.

¹All divisions in this discussion are integer divisions

At this point, one may think that the value of $(a \cdot r_1)/c$ can be calculated by using the same simple partitioning recursively. But there is no guarantee that this recursion will terminate after a finite number of iterations. In fact, it will never stop if b is less than c . Therefore, a more delicate solution is needed.

Let n be the maximum integer such that $n \cdot a$ is less than max_int , and let $multi_a$ denote $n \cdot a$. Let r_1 be equal to $d_2 \cdot n + r_2$: where d_2 and r_2 are the quotient and the remainder resulting from the division of r_1 by n . Then the problem of calculating $(a \cdot r_1)/c$ becomes the problem of calculating $(multi_a \cdot d_2 + a \cdot r_2)/c$.

In short, our algorithm uses the fact that

$$(a \cdot b)/c = a \cdot d_1 + (multi_a \cdot d_2 + a \cdot r_2)/c.$$

$(multi_a \cdot d_2)/c$ and $(a \cdot r_2)/c$ are calculated by calling the algorithm recursively. Remainders are accumulated in an external variable and the result is corrected whenever the accumulated value exceeds c . The question of convergence of the algorithm arises because of the recursive calls. This question will be addressed after the presentation of the pseudo-code.

The following algorithm calculates $(a \cdot b)/c$ in the way discussed above if $a \cdot b$ exceeds max_int . It also considers the possibility of the trivial case that $a \cdot b$ is less than max_int . The variable *residue* is a static external variable and accumulates the remainder values resulting from different divisions. The variable *quotient* is a local variable and is used to store the value of the division being calculated. All variables are integers and the operations *div* and *mod* stand for the integer division and modulation operations respectively.

The algorithm is as follows:

```

multiply_divide(a, b, c)
if a=0 or b=0 then
    return(0)
if b ≤ (max_int div a) then
    quotient ← (a · b) div c
    residue ← residue + ((a · b) mod c)
    if residue ≥ c then
        quotient ← quotient + 1
        residue ← residue - c
else
    quotient ← a · (b div c)
    r1 ← b mod c
    n ← max_int div b
    if r1 ≤ n then
        quotient ← quotient + ((r1 · a) div c)
        residue ← residue + ((r1 · a) mod c)
        if residue ≥ c then
            quotient ← quotient + 1
            residue ← residue - c
    else
        quotient ← quotient + multiply_divide(r1 div n, a · n, c)
        quotient ← quotient + multiply_divide(r1 mod n, a, c)
return(quotient)

```

Convergence of the Algorithm:

Since all variables are integers, we can assure that the algorithm will terminate after a finite number of iterations if we can show the multiplication factors decrease in recursive calls. Recall that the basis of the algorithm is partitioning the question to calculating $a \cdot d_1$, $(multi_a \cdot d_2)/c$ and $(a \cdot r_2)/c$, and calculation of $(multi_a \cdot d_2)/c$ and $(a \cdot r_2)/c$ is done by calling the algorithm recursively. It is obvious that numerators of these recursive calls will be less than $a \cdot b$ if d_1 is greater than zero. Then suppose d_1 is equal to zero. The

algorithm will enter the last *else* clause in which the recursive calls occur, only if both a and d_2 are different than zero. Then the only possibility that can lead to an infinite loop is the possibility of r_2 being equal to zero. Even in that case we know, by definition of *multi_a*, that *multi_a* is greater than half of *max_int*, so greater than c . Hence, we can assure that d_1 will be greater than zero at *multiply_divide*($d_2, \text{multi_}a, c$), or at *multiply_divide*($r_1 \text{ div } n, a \cdot n, c$), as it appears in the algorithm. Therefore, the numerator of recursive calls will decrease in at most two consecutive steps, and the algorithm will terminate after a finite number of iterations.

Bibliography

- [1] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Englewood Cliffs, N.J.: Prentice-Hall, 1990.
- [2] T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44:508–531, 1993.
- [3] T. C. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Comput. Surveys*, 21:557–592, 1989.
- [4] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Comm. ACM*, 29:320–330, 1986.
- [5] A. Bookstein and S. T. Klein. Generative models for bitmap sets with compression applications. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 63–71. 1991.
- [6] A. Bookstein, S. T. Klein, and T. Raita. Model based concordance compression. In J. A. Storer and J. H. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 82–91. IEEE Computer Society Press, 1992.
- [7] A. Bookstein, S. T. Klein, and D. A. Ziff. A systematic approach to compressing a full-text retrieval system. *Information Processing and Management*, 28.
- [8] C. Buckley and A. F. Lewit. Optimization of inverted vector search. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 97–110. 1985.

- [9] Y. Choueka, A. S. Fraenkel, and S. T. Klein. Compression of concordances in full-text retrieval systems. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 597–612. 1988.
- [10] Y. Choueka, A. S. Fraenkel, S. T. Klein, and E. Segal. Improved hierarchical bit-vector compression in document retrieval systems. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 88–97. 1986.
- [11] P. Elias. Universal codeword sets and representations of the integers. *IEEE Trans. Information Theory*, IT-21:194–203, 1975.
- [12] C. Faloutsos. Signature files: Design and performance comparison of some signature extraction methods. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 63–82. 1985.
- [13] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Trans. Office Information Systems*, 2:276–288, 1984.
- [14] C. Faloutsos and H. V. Jagadish. On B-tree indices for skewed distributions. In L. Y. Yuan, editor, *Proc. International Conference on Very Large Databases*, pages 363–374. 1992.
- [15] A. S. Fraenkel and S. T. Klein. Novel compression of sparse bit-strings. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series*, pages 169–183. Springer Verlag, 1985.
- [16] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Englewood Cliffs, N.J.: Prentice-Hall, 1992.
- [17] R. G. Gallager. Variations on a theme by Huffman. *IEEE Trans. Information Theory*, IT-24:668–674, 1978.

- [18] R. G. Gallager and D. C. Van Voorhis. Optimal source codes for geometrically distributed alphabets. *IEEE Trans. Information Theory*, IT-21:228–230, 1978.
- [19] S. W. Golomb. Run-length encodings. *IEEE Trans. Information Theory*, IT-12:399–401, 1966.
- [20] M. Guazzo. A general minimum-redundancy source-coding algorithm. *IEEE Trans. Information Theory*, IT-26:15–25, 1980.
- [21] D. K. Harman and G. Candela. Retrieving records from a gigabyte of text on a minicomputer using statistical ranking. *Journal of the American Society for Information Science*, 41:581–589, 1990.
- [22] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE*, volume 40, pages 1098–1101. 1952.
- [23] A. J. Kent, R. Sacks-Davis, and K. Ramamohanarao. A signature file scheme based on multiple organizations for indexing very large text databases. *Journal of the American Society for Information Science*, 41:508–534, 1990.
- [24] S. T. Klein, A. Bookstein, and S. Deerwester. Storing text retrieval systems on CD-ROM: compression and encryption considerations. *ACM Trans. Informartion Systems*, 7:230–245, 1989.
- [25] D. Lelewer and D. Hirschberg. Data compression. *ACM Comput. Surveys*, 19:261–296, 1987.
- [26] G. Linoff and C. Stanfill. Compression of indexes with full positional information in very large text databases. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 88–97. 1993.
- [27] D. Lucarella. A document retrieval system based upon nearest neighbor searching. *Journal of Information Science*, 14:25–33, 1988.
- [28] K. J. McDonell. An inverted index implementation. *Computer Journal*, 20:116–123, 1977.

- [29] A. Moffat. Word-based text compression. *Software-Practice and Experience*, 20:209–224, 1989.
- [30] A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In J. A. Storer and J. H. Cohn, editors, *Proc. IEEE Data Compression Conference*, pages 72–81. IEEE Computer Society Press, 1992.
- [31] A. Moffat and J. Zobel. Parameterized compression for sparse bitmaps. In P. Ingwersen N. Belkin and A. M. Pejtersen, editors, *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 274–285. 1992.
- [32] A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. In *Proc. Australian Database Conference*, pages 79–91. 1994.
- [33] R. Pasco. *Source coding algorithms for fast data compression*. PhD thesis, Department of Electrical Engineering, Stanford University, 1976.
- [34] J. J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM J. Research and Development*, 20:198–203, 1976.
- [35] J. J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM J. Research and Development*, 23:149–162, 1979.
- [36] F. Rubin. Arithmetic stream coding using fixed precision registers. *IEEE Trans. Information Theory*, IT-25:672–675, 1979.
- [37] B. Y. Ryabko. Data compression by means of a “book stack”. *Problemy Peredachi Informatsii*, 16, 1980.
- [38] R. Sacks-Davis, A. J. Kent, and K. Ramamohanarao. Multi-key access methods based on superimposed coding techniques. *ACM Trans. Database Systems*, 12:655–696, 1987.
- [39] G. Salton. *Automatic Text Processing: The transformation, Analysis, and Retrieval of Information by Computer*. Addison-Wesley, 1989.
- [40] G. Salton, E. A. Fox, and H. Wu. Extended boolean information retrieval. *Comm. ACM*, 26:1022–1036, 1983.

- [41] G. Salton and M. J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.
- [42] E. J. Schuegraf. Compression of large inverted files with hyperbolic term distribution. *Information Processing and Management*, 12:377–384, 1976.
- [43] C. E. Shannon. A mathematical theory of communication. *Bell Systems Technical Journal*, 27:329–423, 1948.
- [44] J. Teuhola. A compression method for clustered bit-vectors. *Information Processing Letters*, 7:308–311, 1978.
- [45] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17:8–19, 1984.
- [46] H. E. White. Printed English compression by dictionary encoding. *Proc. IEEE*, 55:390–396, 1967.
- [47] I. H. Witten, T. C. Bell, and C. Nevill. Models for compression in full-text retrieval systems. In J. A. Storer and J. H. Reif, editors, *Proc. IEEE Data Compression Conference*, pages 23–32. IEEE Computer Society Press, 1991.
- [48] I. H. Witten, T. C. Bell, and C. Nevill. Indexing and compressing full-text databases for CD-ROM. *Journal of Information Sciences*, 17:265–271, 1992.
- [49] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Van Nostrand Reinhold, 1994.
- [50] I. H. Witten, R. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Comm. ACM*, 30:520–540, 1987.
- [51] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Information Theory*, IT-23:337–343, 1977.
- [52] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Information Theory*, IT-24:530–536, 1978.

- [53] J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. In *Proc. 15th Australian Computer Science Conference*, pages 1077–1089. 1992.
- [54] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proc. 18th Australian Conference on Very Large Databases*, pages 352–362. 1992.