Chapter 7
Text Operations

---

with Nivio Ziviani

7.1 Introduction

As discussed in Chapter 2, not all words are equally significant for representing the semantics of a document. In written language, some words carry more *meaning* than others. Usually, *noun* words (or groups of noun words) are the ones which are most representative of a document content. Therefore, it is usually considered worthwhile to preprocess the text of the documents in the collection to determine the terms to be used as *index terms*. During this preprocessing phase other useful text operations can be performed such as elimination of stopwords, stemming (reduction of a word to its grammatical root), the building of a thesaurus, and compression. Such text operations are discussed in this chapter.

We already know that representing documents by sets of index terms leads to a rather imprecise representation of the semantics of the documents in the collection. For instance, a term like '*the*' has no meaning whatsoever by itself and might lead to the retrieval of various documents which are unrelated to the present user query. We say that using the set of all words in a collection to index its documents generates too much *noise* for the retrieval task. One way to reduce this noise is to reduce the set of words which can be used to refer to (i.e., to index) documents. Thus, the preprocessing of the documents in the collection might be viewed simply as a process of controlling the size of the vocabulary (i.e., the number of distinct words used as an index terms). It is expected that the use of a controlled vocabulary leads to an improvement in retrieval performance.

While controlling the size of the vocabulary is a common technique with commercial systems, it does introduce an additional step in the indexing process which is frequently not easily perceived by the users. As a result, a common user might be surprised with some of the documents retrieved and with the absence of other documents which he expected to see. For instance, he might remember that a certain document contains the string '*the house of the lord*' and notice that such a document is not present among the top 20 documents retrieved in

response to his query request (because the controlled vocabulary contains neither 'the' nor 'of'). Thus, it should be clear that, despite a potential improvement in retrieval performance, text transformations done at preprocessing time might make it more difficult for the user to interpret the retrieval task. In recognition of this problem, some search engines in the Web are giving up text operations entirely and simply indexing all the words in the text. The idea is that, despite a more noisy index, the retrieval task is simpler (it can be interpreted as a full text search) and more intuitive to a common user.

Besides document preprocessing, other types of operations on documents can also be attempted with the aim of improving retrieval performance. Among these we distinguish the construction of a thesaurus representing conceptual term relationships and the clustering of related documents. Thesauri are also covered in this chapter. The discussion on document clustering is covered in Chapter 5 because it is an operation which might depend on the current user query.

Text normalization and the building of a thesaurus are strategies aimed at improving the precision of the documents retrieved. However, in the current world of very large digital libraries, improving the efficiency (in terms of time) of the retrieval process has also become quite critical. In fact, Web search engines are currently more concerned with reducing query response time than with improving precision and recall figures. The reason is that they depend on processing a high number of queries per unit of time for economic survival. To reduce query response time, one might consider the utilization of text compression as a promising alternative.

A good compression algorithm is able to reduce the text to 30–35% of its original size. Thus, compressed text requires less storage space and takes less time to be transmitted over a communication link. The main disadvantage is the time spent compressing and decompressing the text. Until recently, it was generally understood that compression does not provide substantial gains in processing time because the extra time spent compressing/decompressing text would offset any gains in operating with compressed data. Further, the use of compression makes the overall design and implementation of the information system more complex. However, modern compression techniques are slowly changing this understanding towards a more favorable view of the adoption of compression techniques. By modern compression techniques we mean good compression and decompression speeds, fast random access without the need to decode the compressed text from the beginning, direct searching on the compressed text without decompressing it, among others.

Besides compression, another operation on text which is becoming more and more important is *encryption*. In fact, due to the fast popularization of services in the Web (including all types of electronic commerce), key (and old) questions regarding security and privacy have surfaced again. More than ever before, impersonation and unauthorized access might result in great prejudice and financial damage to people and organizations. The solution to these problems is not simple but can benefit from the operation of encrypting text. Discussing encrypted text is beyond the scope of this book but an objective and brief introduction to the topic can be found in [498].

In this chapter, we first discuss five preprocessing text operations including thesauri. Following that, we very briefly summarize the problem of document clustering (which is discussed in detail in Chapter 5). Finally, a thorough discussion on the issue of text compression, its modern variations, and its main implications is provided.

7.2 Document Preprocessing

Document preprocessing is a procedure which can be divided mainly into five text operations (or transformations):

(1) Lexical analysis of the text with the objective of treating digits, hyphens, punctuation marks, and the case of letters.

(2) Elimination of stopwords with the objective of filtering out words with very low discrimination values for retrieval purposes.

(3) Stemming of the remaining words with the objective of removing affixes (i.e., prefixes and suffixes) and allowing the retrieval of documents containing syntactic variations of query terms (e.g., connect, connecting, connected, etc).

(4) Selection of index terms to determine which words/stems (or groups of words) will be used as an indexing elements. Usually, the decision on whether a particular word will be used as index term is related to the syntactic nature of the word. In fact, noun words frequently carry more semantics than adjectives, adverbs, and verbs.

(5) Construction of term categorization structures such as a thesaurus, or extraction of structure directly represented in the text, for allowing the expansion of the original query with related terms (a usually useful procedure).

In the following, each of these phases is discussed in detail. But, before proceeding, let us take a look at the logical view of the documents which results after each of the above phases is completed. Figure 1.2 is repeated here for convenience as Figure 7.1. As already discussed, by aggregating the preprocessing phases, we are able to move the logical view of the documents (adopted by the system) from that of a full text to that of a set of high level indexing terms.

7.2.1 Lexical Analysis of the Text

Lexical analysis is the process of converting a stream of characters (the text of the documents) into a stream of words (the candidate words to be adopted as index terms). Thus, one of the major objectives of the lexical analysis phase is the identification of the words in the text. At first glance, all that seems to be involved is the recognition of spaces as word separators (in which case, multiple

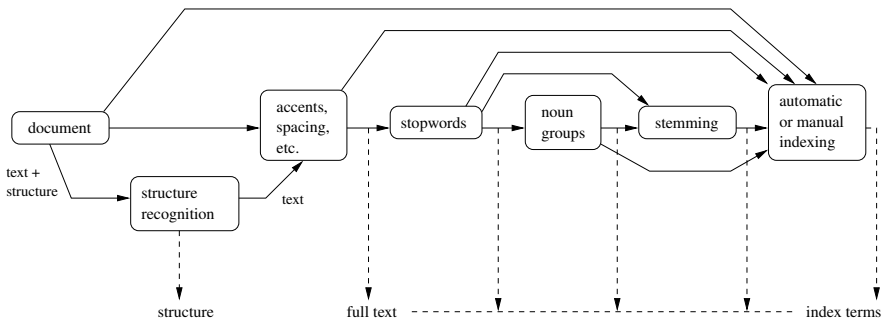


Figure 7.1 Logical view of a document throughout the various phases of text preprocessing.

spaces are reduced to one space). However, there is more to it than this. For instance, the following four particular cases have to be considered with care [261]: digits, hyphens, punctuation marks, and the case of the letters (lower and upper case).

Numbers are usually not good index terms because, without a surrounding context, they are inherently vague. For instance, consider that a user is interested in documents about the number of deaths due to car accidents between the years 1910 and 1989. Such a request could be specified as the set of index terms {deaths, car, accidents, years, 1910, 1989}. However, the presence of the numbers 1910 and 1989 in the query could lead to the retrieval, for instance, of a variety of documents which have a reference to either of these two years. The problem is that numbers by themselves are just too vague. Thus, in general it is wise to disregard numbers as index terms. However, we have also to consider that digits might appear mixed within a word. For instance, '510B.C.' is a clearly important index term. In this case, it is not clear what rule should be applied. Furthermore, a sequence of 16 digits identifying a credit card number might be highly relevant in a given context and, in this case, should be considered as an index term. A preliminary approach for treating digits in the text might be to remove all words containing sequences of digits unless specified otherwise (through regular expressions). Further, an advanced lexical analysis procedure might perform some date and number normalization to unify formats.

Hyphens pose another difficult decision to the lexical analyzer. Breaking up hyphenated words might be useful due to inconsistency of usage. For instance, this allows treating 'state-of-the-art' and 'state of the art' identically. However, there are words which include hyphens as an integral part. For instance, gilt-edge, B-49, etc. Again, the most suitable procedure seems to adopt a general rule and specify the exceptions on a case by case basis.

Normally, punctuation marks are removed entirely in the process of lexical analysis. While some punctuation marks are an integral part of the word (for

instance, '510B.C.'), removing them does not seem to have an impact in retrieval performance because the risk of misinterpretation in this case is minimal. In fact, if the user specifies '510B.C' in his query, removal of the dot both in the query term and in the documents will not affect retrieval. However, very particular scenarios might again require the preparation of a list of exceptions. For instance, if a portion of a program code appears in the text, it might be wise to distinguish between the variables 'x.id' and 'xid.' In this case, the dot mark should not be removed.

The case of letters is usually not important for the identification of index terms. As a result, the lexical analyzer normally converts all the text to either lower or upper case. However, once more, very particular scenarios might require the distinction to be made. For instance, when looking for documents which describe details about the command language of a Unix-like operating system, the user might explicitly desire the non-conversion of upper cases because this is the convention in the operating system. Further, part of the semantics might be lost due to case conversion. For instance, the words *Bank* and *bank* have different meanings — a fact common to many other pairs of words.

As pointed out by Fox [261], all these text operations can be implemented without difficulty. However, careful thought should be given to each one of them because they might have a profound impact at document retrieval time. This is particularly worrisome in those situations in which the user finds it difficult to understand what the indexing strategy is doing. Unfortunately, there is no clear solution to this problem. As already mentioned, some Web search engines are opting for avoiding text operations altogether because this simplifies the interpretation the user has of the retrieval task. Whether this strategy will be the one of choice in the long term remains to be seen.

7.2.2 Elimination of Stopwords

As discussed in Chapter 2, words which are too frequent among the documents in the collection are not good discriminators. In fact, a word which occurs in 80% of the documents in the collection is useless for purposes of retrieval. Such words are frequently referred to as *stopwords* and are normally filtered out as potential index terms. Articles, prepositions, and conjunctions are natural candidates for a list of stopwords.

Elimination of stopwords has an additional important benefit. It reduces the size of the indexing structure considerably. In fact, it is typical to obtain a compression in the size of the indexing structure (for instance, in the size of an inverted list, see Chapter 8) of 40% or more solely with the elimination of stopwords.

Since stopword elimination also provides for compression of the indexing structure, the list of stopwords might be extended to include words other than articles, prepositions, and conjunctions. For instance, some verbs, adverbs, and adjectives could be treated as stopwords. In [273], a list of 425 stopwords is illustrated. Programs in C for lexical analysis are also provided.

Despite these benefits, elimination of stopwords might reduce recall. For instance, consider a user who is looking for documents containing the phrase '*to be or not to be*.' Elimination of stopwords might leave only the term *be* making it almost impossible to properly recognize the documents which contain the phrase specified. This is one additional reason for the adoption of a full text index (i.e., insert all words in the collection into the inverted file) by some Web search engines.

7.2.3 Stemming

Frequently, the user specifies a word in a query but only a variant of this word is present in a relevant document. Plurals, gerund forms, and past tense suffixes are examples of syntactical variations which prevent a perfect match between a query word and a respective document word. This problem can be partially overcome with the substitution of the words by their respective stems.

A *stem* is the portion of a word which is left after the removal of its affixes (i.e., prefixes and suffixes). A typical example of a stem is the word *connect* which is the stem for the variants *connected, connecting, connection,* and *connections*. Stems are thought to be useful for improving retrieval performance because they reduce variants of a same root word to a common concept. Furthermore, stemming has the secondary effect of reducing the size of the indexing structure because the number of distinct index terms is reduced.

While the argument supporting stemming seems sensible, there is controversy in the literature about the benefits of stemming for retrieval performance. In fact, different studies lead to rather conflicting conclusions. Frakes [273] compares eight distinct studies on the potential benefits of stemming. While he favors the usage of stemming, the results of the eight experimental studies he investigated do not allow us to reach a satisfactory conclusion. As a result of these doubts, many Web search engines do not adopt any stemming algorithm whatsoever.

Frakes distinguishes four types of stemming strategies: affix removal, table lookup, successor variety, and n-grams. Table lookup consists simply of looking for the stem of a word in a table. It is a simple procedure but one which is dependent on data on stems for the whole language. Since such data is not readily available and might require considerable storage space, this type of stemming algorithm might not be practical. Successor variety stemming is based on the determination of morpheme boundaries, uses knowledge from structural linguistics, and is more complex than affix removal stemming algorithms. N-grams stemming is based on the identification of digrams and trigrams and is more a term clustering procedure than a stemming one. Affix removal stemming is intuitive, simple, and can be implemented efficiently. Thus, in the remainder of this section we concentrate our discussion on algorithms for affix removal stemming only.

In affix removal, the most important part is suffix removal because most variants of a word are generated by the introduction of suffixes (instead of pre-

fixes). While there are three or four well known suffix removal algorithms, the most popular one is that by Porter because of its simplicity and elegance. Despite been simpler, the Porter algorithm yields results comparable to those of the more sophisticated algorithms.

The Porter algorithm uses a suffix list for suffix stripping. The idea is to apply a series of rules to the suffixes in the words in the text. For instance, the rule

$$s \longrightarrow \phi \tag{7.1}$$

is used to convert plural forms into their respective singular forms by substituting the letter s by nil. Notice that to identify the suffix we must examine the last letters in the word. Furthermore, we look for the longest sequence of letters which matches the left hand side in a set of rules. Thus, application of the two following rules

$$\begin{aligned} sses &\longrightarrow ss \\ s &\longrightarrow \phi \end{aligned} \tag{7.2}$$

to the word *stresses* yields the stem *stress* instead of the stem *stresse*. By separating such rules into five distinct phases, the Porter algorithm is able to provide effective stemming while running fast. A detailed description of the Porter algorithm can be found in Appendix B.

7.2.4 Index Terms Selection

If a full text representation of the text is adopted then all words in the text are used as index terms. The alternative is to adopt a more abstract view in which not all words are used as index terms. This implies that the set of terms used as indices must be selected. In the area of bibliographic sciences, such a selection of index terms is usually done by a specialist. An alternative approach is to select terms for index terms automatically.

Distinct automatic approaches for selecting index terms can be used. A good approach is the identification of noun groups (as done in the Inquery system [120]) which we now discuss.

A sentence in natural language text is usually composed of nouns, pronouns, articles, verbs, adjectives, adverbs, and connectives. While the words in each grammatical class are used with a particular purpose, it can be argued that most of the semantics is carried by the noun words. Thus, an intuitively promising strategy for selecting index terms automatically is to use the nouns in the text. This can be done through the systematic elimination of verbs, adjectives, adverbs, connectives, articles, and pronouns.

Since it is common to combine two or three nouns in a single component (e.g., *computer science*), it makes sense to cluster nouns which appear nearby in the text into a single indexing component (or concept). Thus, instead of simply

using nouns as index terms, we adopt noun groups. A *noun group* is a set of nouns whose syntactic distance in the text (measured in terms of number of words between two nouns) does not exceed a predefined threshold (for instance, 3).

When noun groups are adopted as indexing terms, we obtain a conceptual logical view of the documents in terms of sets of non-elementary index terms.

7.2.5 Thesauri

The word *thesaurus* has Greek and Latin origins and is used as a reference to a treasury of words [259]. In its simplest form, this treasury consists of (1) a precompiled list of important words in a given domain of knowledge and (2) for each word in this list, a set of related words. Related words are, in its most common variation, derived from a synonymity relationship.

In general, however, a thesaurus also involves some normalization of the vocabulary and includes a structure much more complex than a simple list of words and their synonyms. For instance, the popular thesaurus published by Peter Roget [674] also includes *phrases* which means that concepts more complex than single words are taken into account. Roget's thesaurus is of a general nature (i.e., not specific to a certain domain of knowledge) and organizes words and phrases in categories and subcategories.

An example of an entry in Roget's thesaurus is as follows:

> **cowardly** *adjective*
> Ignobly lacking in courage: *cowardly turncoats.*
> **Syns:** chicken (slang), chicken-hearted, craven, dastardly, faint-hearted, gutless, lily-livered, pusillanimous, unmanly, yellow (slang), yellow-bellied (slang).

To the adjective *cowardly*, Roget's thesaurus associates several synonyms which compose a thesaurus class. While Roget's thesaurus is of a generic nature, a thesaurus can be specific to a certain domain of knowledge. For instance, the Thesaurus of Engineering and Scientific Terms covers concepts related to engineering and technical terminology.

According to Foskett [259], the main purposes of a thesaurus are basically: (a) to provide a standard vocabulary (or system of references) for indexing and searching; (b) to assist users with locating terms for proper query formulation; and (c) to provide classified hierarchies that allow the broadening and narrowing of the current query request according to the needs of the user. In this section, however, we do not discuss how to use a thesaurus for modifying the user query. This issue is covered on Chapter 5 which also discusses algorithms for automatic construction of thesauri.

Notice that the motivation for building a thesaurus is based on the fundamental idea of using a *controlled vocabulary* for the indexing and searching. A controlled vocabulary presents important advantages such as normalization

of indexing concepts, reduction of noise, identification of indexing terms with a clear semantic meaning, and retrieval based on concepts rather than on words. Such advantages are particularly important in specific domains, such as the medical domain for which there is already a large amount of knowledge compiled. For general domains, however, a well known body of knowledge which can be associated with the documents in the collection might not exist. The reasons might be that the document base is new, that it is too large, or that it changes very dynamically. This is exactly the case with the Web. Thus, it is not clear how useful a thesaurus is in the context of the Web. Despite that, the success of the search engine named 'Yahoo!' (see Chapter 13), which presents the user with a term classification hierarchy that can be used to reduce the space to be searched, suggests that thesaurus-based techniques might be quite useful even in the dynamic world of the Web.

It is still too early to reach a consensus on the advantages of a thesaurus for the Web. As a result, many search engines simply use *all* the words in all the documents as index terms (i.e., there is no notion of using the concepts of a controlled vocabulary for indexing and searching purposes). Whether thesaurus-based techniques will flourish in the context of the Web remains to be seen.

The main components of a thesaurus are its index terms, the relationships among the terms, and a layout design for these term relationships. Index terms and term relationships are covered below. The layout design for term relationships can be in the form of a list or in the form of a bi-dimensional display. Here, we consider only the more conventional layout structure based on a list and thus, do not further discuss the issue of layout of the terms in a thesaurus. A brief coverage of topics related to this problem can be found in Chapter 10. A more detailed discussion can be found in [259].

Theasurus Index Terms

The terms are the *indexing* components of the thesaurus. Usually, a term in a thesaurus is used to denote a *concept* which is the basic semantic unit for conveying ideas. Terms can be individual words, groups of words, or phrases, but most of them are single words. Further, terms are basically nouns because nouns are the most concrete part of speech. Terms can also be verbs in gerund form whenever they are used as nouns (for instance, *acting*, *teaching*, etc.).

Whenever a concept cannot be expressed by a single word, a group of words is used instead. For instance, many concepts are better expressed by a combination of an adjective with a noun. A typical example is *balistic missiles*. In this case, indexing the compound term directly will yield an entry under *balistic* and no entry under *missiles* which is clearly inadequate. To avoid this problem, the compound term is usually modified to have the noun as the first word. For instance, we can change the compound term to *missiles, balistic*.

We notice the use of the plural form *missiles* instead of the singular form *missile*. The reasoning is that a thesaurus represents classes of things and thus it is natural to prefer the plural form. However, the singular form is used for compound terms which appear normally in the singular such as *body temperature*.

Deciding between singular and plural is not always a simple matter.

Besides the term itself, frequently it is necessary to complement a thesaurus entry with a *definition* or an *explanation*. The reason is the need to specify the precise meanings of a term in the context of a particular thesaurus. For instance, the term *seal* has a meaning in the context of *marine animals* and a rather distinct meaning in the context of *documents*. In these cases, the definition might be preceded by a context explanation such as *seal (marine animals)* and *seal (documents)* [730].

Thesaurus Term Relationships

The set of terms related to a given thesaurus term is mostly composed of synonyms and near-synonyms. In addition to these, relationships can be induced by patterns of co-occurrence within documents. Such relationships are usually of a hierarchical nature and most often indicate broader (represented by BT) or narrower (represented by NT) related terms. However, the relationship might also be of a lateral or non-hierarchical nature. In this case, we simply say that the terms are related (represented by RT).

As discussed in Chapter 5, BT and NT relationships define a classification hierarchy where the broader term is associated with a class and its related narrower terms are associated with the instances of this class. Further, it might be that a narrower term is associated with two or more broader terms (which is not the most common case though). While BT and NT relationships can be identified in a fully automatic manner (i.e., without assistance from a human subject), dealing with RT relationships is much harder. One reason seems to be that RT relationships are dependent on the specific context and particular needs of the group of users and thus are difficult to identify without knowledge provided by specialists.

On the Use of Thesauri in IR

As described by Peter Roget [674, 259], a thesaurus is a classification scheme composed of words and phrases whose organization aims at facilitating the expression of ideas in written text. Thus, whenever a writer has a difficulty in finding the proper term to express an idea (a common occurrence in serious writing), he can use the thesaurus to obtain a better grasp on the fundamental semantics of terms related to his idea.

In the area of information retrieval, researchers have for many years conjectured and studied the usefulness of a thesaurus for helping with the query formation process. Whenever a user wants to retrieve a set of documents, he first builds up a conceptualization of what he is looking for. Such conceptualization is what we call his *information need*. Given the information need, the user still has to translate it to a query in the language of the IR system. This usually means that a set of index terms has to be selected. However, since the collection might be vast and the user inexperienced, the selection of such *initial* terms might be erroneous and improper (a very common situation with the largely

unknown and highly dynamic collection of documents and pages which compose the Web). In this case, reformulating the original query seems to be a promising course of action. Such a reformulation process usually implies expanding the original query with related terms. Thus, it seems natural to use a thesaurus for assisting the user with the search for related terms.

Unfortunately, this approach does not work well in general because the relationships captured in a thesaurus frequently are not valid in the local context of a given user query. One alternative is to determine thesaurus-like relationships at query time. Unfortunately, such an alternative is not attractive for Web search engines which cannot afford to spend a lot of time with the processing of individual queries. This and many other interesting issues related to the use of thesaurus-based techniques in IR are covered in Chapter 5.

7.3 Document Clustering

Document clustering is the operation of grouping together similar (or related) documents in classes. In this regard, document clustering is not really an operation on the text but an operation on the collection of documents.

The operation of clustering documents is usually of two types: global and local. In a global clustering strategy, the documents are grouped accordingly to their occurrence in the whole collection. In a local clustering strategy, the grouping of documents is affected by the context defined by the current query and its *local* set of retrieved documents.

Clustering methods are usually used in IR to transform the original query in an attempt to better represent the user information need. From this perspective, clustering is an operation which is more related to the transformation of the user query than to the transformation of the text of the documents. In this book, document clustering techniques are treated as query operations and thus, are covered in Chapter 5 (instead of here).

7.4 Text Compression

7.4.1 Motivation

Text compression is about finding ways to represent the text in fewer bits or bytes. The amount of space required to store text on computers can be reduced significantly using compression techniques. Compression methods create a reduced representation by identifying and using structures that exist in the text. From the compressed version, the original text can be reconstructed exactly.

Text compression is becoming an important issue in an information retrieval environment. The widespread use of digital libraries, office automation systems, document databases, and the Web has led to an explosion of textual information available online. In this scenario, text compression appears as

an attractive option for reducing costs associated with space requirements, input/output(I/O) overhead, and communication delays. The gain obtained from compressing text is that it requires less storage space, it takes less time to be transmitted over a communication link, and it takes less time to search directly the compressed text. The price paid is the time necessary to code and decode the text.

A major obstacle for storing text in compressed form is the need for IR systems to access text randomly. To access a given word in a compressed text, it is usually necessary to decode the entire text from the beginning until the desired word is reached. It could be argued that a large text could be divided into blocks that are compressed independently, thus allowing fast random access to each block. However, efficient compression methods need to process some text before making compression effective (usually more than 10 kilobytes). The smaller the blocks, the less effective compression is expected to be.

Our discussion here focuses on text compression methods which are suitable for use in an IR environment. For instance, a successful idea aimed at merging the requirements of compression algorithms and the needs of IR systems is to consider that the symbols to be compressed are words and not characters (character-based compression is the more conventional approach). Words are the atoms on which most IR systems are built. Moreover, it is now known that much better compression is achieved by taking words as symbols (instead of characters). Further, new word-based compression methods allow random access to words within the compressed text which is a critical issue for an IR system.

Besides the economy of space obtained by a compression method, there are other important characteristics to be considered such as compression and decompression speed. In some situations, decompression speed is more important than compression speed. For instance, this is the case with textual databases in which it is common to compress the text once and to read it many times from disk.

Another important characteristic of a compression method is the possibility of performing compressed pattern matching, defined as the task of performing pattern matching in a compressed text without decompressing it. In this case, sequential searching can be speeded up by compressing the search key rather than decoding the compressed text being searched. As a consequence, it is possible to search faster on compressed text because much less text has to be scanned. Chapter 8 presents efficient methods to deal with searching the compressed text directly.

When the text collection is large, efficient text retrieval requires specialized index techniques. A simple and popular indexing structure for text collections are the inverted files. Inverted files (see Chapter 8 for details) are especially adequate when the pattern to be searched for is formed by simple words. Since this is a common type of query (for instance, when searching the Web), inverted files are widely used for indexing large text collections.

An inverted file is typically composed of (a) a vector containing all the distinct words in the text collection (which is called the *vocabulary*) and (b) for each word in the vocabulary, a list of all documents (identified by document

numbers) in which that word occurs. Because each list of document numbers (within the inverted file) is organized in ascending order, specific compression methods have been proposed for them, leading to very efficient index compression schemes. This is important because query processing time is highly related to index access time. Thus, in this section, we also discuss some of the most important index compression techniques.

We first introduce basic concepts related to text compression. Following that, we present some of the most important statistical compression methods. At the end, we briefly review compression methods based on a dictionary.

7.4.2 Basic Concepts

There are two general approaches to text compression: *statistical* and *dictionary* based. *Statistical methods* rely on generating good probability estimates for each symbol. The more accurate the estimates are, the better the compression obtained. A *symbol* here is usually a character, a text word, or a fixed number of characters. The set of all possible symbols in the text is called the *alphabet*. The task of estimating the probability on each next symbol is called *modeling*. A *model* is essentially a collection of probability distributions, one for each context in which a symbol can be coded. Once these probabilities are available the symbols are converted into binary digits, a process called *coding*. In practice, both the encoder and decoder use the same model. The decoder interprets the output of the encoder (with reference to the same model) to find out the original symbol.

There are two well known coding strategies: Huffman coding and arithmetic coding. The idea of Huffman coding is to assign a fixed-length bit encoding to each different symbol of the text. Compression is achieved by assigning a smaller number of bits to symbols with higher probabilities. Huffman coding was first published in the early 1950s and was the most important compression method until the late 1970s, when arithmetic coding made higher compression rates possible.

Arithmetic coding computes the code incrementally, one symbol at a time, as opposed to the Huffman coding scheme in which each different symbol is pre-encoded using a fixed-length number of bits. The incremental nature of arithmetic coding does not allow decoding a string which starts in the middle of a compressed file. To decode a symbol in the middle of a file compressed with arithmetic coding, it is necessary to decode the whole text from the very beginning until the desired word is reached. This characteristic of arithmetic coding makes it inadequate for use in an IR environment.

Dictionary methods substitute a sequence of symbols by a pointer for a previous occurrence of that sequence. They usually use pointer representations to code references to entries in a dictionary. The dictionary is a list of symbols (often called phrases) that are expected to occur frequently. Pointers to the dictionary entries are chosen so that they need less space than the phrase they replace, thus obtaining compression. The distinction between modeling and coding does not

exist in dictionary methods and there are no explicit probabilities associated to phrases. The most well known dictionary methods are represented by a family of methods, known as the Ziv-Lempel family.

The relationship between probabilities and codes was established by Claude Shannon in his source code theorem [713]. He showed that, in an optimal encoding scheme, a symbol that is expected to occur with probability p should be assigned a code of length $\log_2 \frac{1}{p}$ bits. The number of bits in which a symbol is best coded represents the *information content* of the symbol. The average amount of information per symbol over the whole alphabet is called the *entropy* of the probability distribution, and is given by:

$$E = \sum p_i \log_2 \frac{1}{p_i}$$

E is a *lower bound* on compression which applies to any coding method based on the probability distribution p_i. It is measured in bits per symbol and represents a lower bound that can never be beaten for compression based on the probability distribution p_i. It is important to note that E is calculated from the probabilities and so is a property of the model. See Chapter 6 for more details on this topic.

The traditional implementations of the Huffman method are character-based, that is, they adopt the characters as the symbols in the alphabet. Character-based Huffman methods are typically able to compress English texts to approximately five bits per character (usually, each uncompressed character takes seven to eight bits to be represented). More recently, a word-based Huffman method has been proposed as a better alternative for natural language texts. This method is able to reduce English texts to just over two bits per character. As we will see later on, word-based Huffman coding achieves compression rates close to the entropy and allows random access to intermediate points in the compressed text. Ziv-Lempel methods are able to reduce English texts to fewer than four bits per character. Methods based on arithmetic coding can also compress English texts to just over two bits per character. However, the price paid is slower compression and decompression, and the impossibility of randomly accessing intermediate points in the compressed text.

Before proceeding, let us present an important definition which will be useful from now on.

Definition Compression ratio *is the size of the compressed file as a percentage of the uncompressed file.*

7.4.3 Statistical Methods

In a statistical method, a probability is estimated for each symbol (the modeling task) and, based on this probability, a code is assigned to each symbol at a time (the coding task). Shorter codes are assigned to the most likely symbols.

Modeling

The *function of a model* is to provide a probability assignment for the next symbol to be coded. High compression can be obtained by forming good models of the text that is to be coded. The probability assignment is explained in the following section.

Compression models can be *adaptive*, *static*, or *semi-static*. *Adaptive models* start with no information about the text and progressively learn about its statistical distribution as the compression process goes on. Thus, adaptive models need only one pass over the text and store no additional information apart from the compressed text. For long enough texts, such models converge to the true statistical distribution of the text. One major disadvantage, however, is that decompression of a file has to start from its beginning, since information on the distribution of the data is stored incrementally inside the file. Adaptive modeling is a good option for general purpose compression programs, but an inadequate alternative for full-text retrieval where random access to compressed patterns is a must.

Static models assume an average distribution for all input texts. The modeling phase is done only once for all texts to be coded in the future (i.e., somehow a probability distribution is estimated and then used for all texts to be compressed in the future). These models tend to achieve poor compression ratios when the data deviates from initial statistical assumptions. For example, a model adequate for English literary texts will probably perform poorly for financial texts containing a lot of different numbers, as each number is relatively rare and so receives long codes.

Semi-static models do not assume any distribution on the data, but learn it in a first pass and send it to the decoder before transmitting the encoded symbols. In a second pass, they compress the data by using a fixed code derived from the distribution learned from the first pass. The disadvantages are that they must make two passes over the text and the encoding data must be stored to be used by the decoder to decompress. In situations where interactive data communications are involved it may be impractical to make two passes over the text. However, semi-static models have a crucial advantage in IR contexts: since the same codes are used at every point in the compressed file, direct access is possible.

Word-based models take words instead of characters as symbols. Usually, a word is a contiguous string of characters in the set {A..Z, a..z} separated by other characters not in the set {A..Z, a..z}. There are many good reasons to use word-based models in an IR context. First, much better compression rates are achieved by taking words as symbols rather than taking characters because words carry a lot of meaning in natural languages and, as a result, their distribution has much more to do with the semantic structure of the text than the individual letters. Second, words are the atoms on which most information retrieval systems are built. Words are already stored for indexing purposes and so might be used as part of the model for compression. Third, the word frequencies are also useful in answering queries involving combinations of words because the best strategy is to start with the least frequent words first.

Since the text is not only composed of words but also of separators, a model must also be chosen for them. There are many different ways to deal with separators. As words and separators always follow one another, two different alphabets are usually used: one for words and one for separators. Consider the following example: `each rose, a rose is a rose`. In the word-based model, the set of symbols of the alphabet is {`a`, `each`, `is`, `rose`}, whose frequencies are 2, 1, 1, and 3, respectively, and the set of separators is {'`,`␣', ␣}, whose frequencies are 1 and 5, respectively (where ␣ represents a space). Once it is known that the text starts with a word or a separator, there is no confusion on which alphabet to use.

In natural language texts, a word is followed by a single space in most cases. In the texts of the TREC-3 collection [340] (see Chapter 3), 70–80% of the separators are single spaces. Another good alternative is to consider the single space that follows a word as part of the same word. That is, if a word is followed by a space, we can encode just the word. If not, we can encode the word and then the following separator. At decoding time, we decode a word and assume that a space follows unless the next symbol corresponds to a separator. Notice that now a single alphabet for words and separators (single space excluded) is used. For instance, in the example above, the single alphabet is {'`,`␣', `a`, `each`, `is`, `rose`} and there is no longer an alphabet for separators. As the alphabet excludes the single space then the words are called *spaceless words*.

In some situations word-based models for full-text databases have a potential to generate a great quantity of different codes and care must be exercised to deal with this fact. For instance, as discussed in the section on lexical analysis (at the beginning of this chapter), one has to consider whether a sequence of digits is to be considered as a word. If it is, then a collection which contains one million documents and includes document numbers as identifiers will generate one million words composed solely of digits, each one occurring once in the collection. This can be very inefficient for any kind of compression method available. One possible good solution is to divide long numbers into shorter ones by using a null (or implicit) punctuation marker in between. This diminishes the alphabet size resulting in considerable improvements in the compression ratio and in the decoding time.

Another important consideration is the size of the alphabet in word-based schemes. How large is the number of different words in a full-text database? It is empirically known that the vocabulary V of natural language texts with n words grows sublinearly. Heaps [350] shows that $V = O(n^{\beta})$, where β is a constant dependent on the particular text. For the 2 gigabyte TREC-3 collection [340], β is between 0.4 and 0.6 which means that the alphabet size grows roughly proportional to the square root of n. Even for this growth of the alphabet, the generalized Zipf law shows that the probability distribution is skewed so that the entropy remains constant. This implies that the compression ratio does not degrade as the text and hence the number of different symbols grow. Heaps and Zipf laws are explained in Chapter 6.

Finally, it is important to mention that word-based Huffman methods need large texts to be effective (i.e., they are not adequate to compress and transmit a

single Web page over a network). The need to store the vocabulary represents an important space overhead when the text is small (say, less than 10 megabytes). However, this is not a concern in IR in general as the texts are large and the vocabulary is needed anyway for other purposes such as indexing and querying.

Coding

Coding corresponds to the task of obtaining the representation (code) of a symbol based on a probability distribution given by a model. The main goal of a coder is to assign short codes to likely symbols and long codes to unlikely ones. As we have seen in the previous section, the entropy of a probability distribution is a lower bound on how short the average length of a code can be, and the quality of a coder is measured in terms of how close to the entropy it is able to get. Another important consideration is the speed of both the coder and the decoder. Sometimes it is necessary to sacrifice the compression ratio to reduce the time to encode and decode the text.

A semi-static Huffman compression method works in two passes over the text. In a first pass, the modeler determines the probability distribution of the symbols and builds a coding tree according to this distribution. In a second pass, each next symbol is encoded according to the coding tree. Adaptive Huffman compression methods, instead, work in one single pass over the text updating the coding tree incrementally. The encoding of the symbols in the input text is also done during this single pass over the text. The main problem of adaptive Huffman methods is the cost to update the coding tree as new symbols are read.

As with Huffman-based methods, arithmetic coding methods can also be based on static, semi-static or adaptive algorithms. The main strength of arithmetic coding methods is that they can generate codes which are arbitrarily close to the entropy for any kind of probability distribution. Another strength of arithmetic coding methods is that they do not need to store a coding tree explicitly. For adaptive algorithms, this implies that arithmetic coding uses less memory than Huffman-based coding. For static or semi-static algorithms, the use of canonical Huffman codes overcomes this memory problem (canonical Huffman trees are explained later on).

In arithmetic coding, the input text is represented by an interval of real numbers between 0 and 1. As the size of the input becomes larger, the interval becomes smaller and the number of bits needed to specify this interval increases. Compression is achieved because input symbols with higher probabilities reduce the range less than symbols with smaller probabilities and hence add fewer bits to the output code.

Arithmetic coding presents many disadvantages over Huffman coding in an IR environment. First, arithmetic coding is much slower than Huffman coding, especially with static and semi-static algorithms. Second, with arithmetic coding decompression cannot start in the middle of a compressed file. This contrasts with Huffman coding, in which it is possible to index and to decode from any position in the compressed text if static or semi-static algorithms are used.

Third, word-based Huffman coding methods yield compression ratios as good as arithmetic coding ones.

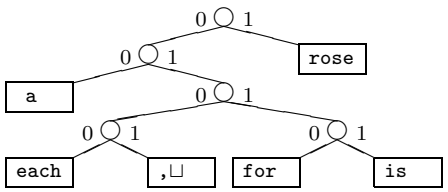
Consequently, Huffman coding is the method of choice in full-text retrieval, where both speed and random access are important. Thus, we will focus the remaining of our discussion on semi-static word-based Huffman coding.

Huffman Coding.
Huffman coding is one of the best known compression methods [384]. The idea is to assign a variable-length encoding in bits to each symbol and encode each symbol in turn. Compression is achieved by assigning shorter codes to more frequent symbols. Decompression uniqueness is guaranteed because no code is a prefix of another. A word-based semi-static model and Huffman coding form a good compression method for text.

Figure 7.2 presents an example of compression using Huffman coding on words. In this example the set of symbols of the alphabet is {'`,`␣', `a`, `each`, `for`, `is`, `rose`}, whose frequencies are 1, 2, 1, 1, 1, and 3, respectively. In this case the alphabet is unique for words and separators. Notice that the separator '␣' is not part of the alphabet because the single space that follows a word is considered as part of the same word. These words are called *spaceless words* (see more about spaceless words in Section 7.4.3). The Huffman tree shown in Figure 7.2 is an example of a binary trie built on binary codes. Tries are explained in Chapter 8.

Decompression is accomplished as follows. The stream of bits in the compressed file is traversed from left to right. The sequence of bits read is used to also traverse the Huffman compression tree, starting at the root. Whenever a leaf node is reached, the corresponding word (which constitutes the decompressed symbol) is printed out and the tree traversal is restarted. Thus, according to the tree in Figure 7.2, the presence of the code `0110` in the compressed file leads to the decompressed symbol `for`.

To build a Huffman tree, it is first necessary to obtain the symbols that constitute the alphabet and their probability distribution in the text to be compressed. The algorithm for building the tree then operates bottom up and starts by creating for each symbol of the alphabet a node containing the symbol and



Original text: `for each rose, a rose is a rose`

Compressed text: `0110 0100 1 0101 00 1 0111 00 1`

Figure 7.2 Huffman coding tree for spaceless words.

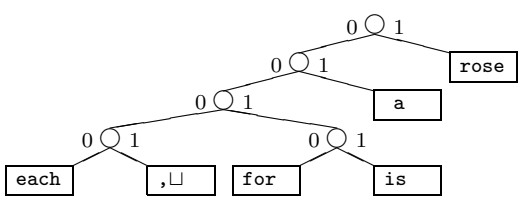
its probability (or frequency). At this point there is a forest of one-node trees. Next, the two nodes with the smallest probabilities become children of a newly created parent node. With this parent node is associated a probability equal to the sum of the probabilities of the two chosen children. The operation is repeated ignoring nodes that are already children, until there is only one node, which becomes the root of the decoding tree. By delaying the pairing of nodes with high probabilities, the algorithm necessarily places them closer to the root node, making their code smaller. The two branches from every internal node are consistently labeled 0 and 1 (or 1 and 0). Given s symbols and their frequencies in the text, the algorithm builds the Huffman tree in $O(s \log s)$ time.

The number of Huffman trees which can be built for a given probability distribution is quite large. This happens because interchanging left and right subtrees of any internal node results in a different tree whenever the two subtrees are different in structure, but the weighted average code length is not affected. Instead of using any kind of tree, the preferred choice for most applications is to adopt a *canonical tree* which imposes a particular order to the coding bits.

A Huffman tree is canonical when the height of the left subtree of any node is never smaller than that of the right subtree, and all leaves are in increasing order of probabilities from left to right. Figure 7.3 shows the canonical tree for the example of Figure 7.2. The deepest leaf at the leftmost position of the Huffman canonical tree, corresponding to one element with smallest probability, will contain only zeros, and the following codes will be in increasing order inside each level. At each change of level we shift left one bit in the counting. The table in Figure 7.3 shows the canonical codes for the example of Figure 7.2.

A canonical code can be represented by an ordered sequence S of pairs (x_i, y_i), $1 \leq i \leq \ell$, where x_i represents the number of elements at level i and y_i represents the numerical value of the first code at level i. For our example in Figure 7.3, the ordered sequence is $S = \langle (1,1), (1,1), (0,\infty), (4,0) \rangle$. For instance, the fourth pair $(4,0)$ in S corresponds to the fourth level and indicates that there are four nodes at this level and that to the node most to the left is assigned a code, at this level, with value 0. Since this is the fourth level, a value 0 corresponds to the codeword 0000.

One of the properties of canonical codes is that the set of codes having the



Symbol	Prob.	Old code	Can. code
each	1/9	0100	0000
,␣	1/9	0101	0001
for	1/9	0110	0010
is	1/9	0111	0011
a	2/9	00	01
rose	3/9	1	1

Figure 7.3 Canonical code.

same length are the binary representations of consecutive integers. Interpreted as integers, the 4-bit codes of the table in Figure 7.3 are 0, 1, 2, and 3, the 2-bit code is 1 and the 1-bit code is also 1. In our example, if the first character read from the input stream is 1, a codeword has been identified and the corresponding symbol can be output. If this value is 0, a second bit is appended and the two bits are again interpreted as an integer and used to index the table and identify the corresponding symbol. Once we read '`00`' we know that the code has four bits and therefore we can read two more bits and use them as an index into the table. This fact can be exploited to enable efficient encoding and decoding with small overhead. Moreover, much less memory is required, which is especially important for large vocabularies.

Byte-Oriented Huffman Code.
The original method proposed by Huffman [384] leads naturally to binary coding trees. In [573], however, it is proposed to build the code assigned to each symbol as a sequence of whole bytes. As a result, the Huffman tree has degree 256 instead of 2. Typically, the code assigned to each symbol contains between 1 and 5 bytes. For example, a possible code for the word `rose` could be the 3-byte code '`47 131 8`.'

The construction of byte Huffman trees involves some details which must be dealt with. Care must be exercised to ensure that the first levels of the tree have no empty nodes when the code is not binary. Figure 7.4(a) illustrates a case where a naive extension of the binary Huffman tree construction algorithm might generate a non-optimal byte tree. In this example the alphabet has 512 symbols, all with the same probability. The root node has 254 empty spaces that could be occupied by symbols from the second level of the tree, changing their code lengths from 2 bytes to 1 byte.

A way to ensure that the empty nodes always go to the lowest level of the tree follows. We calculate beforehand the number of empty nodes that will arise. We then compose these empty nodes with symbols of smallest probabilities (for

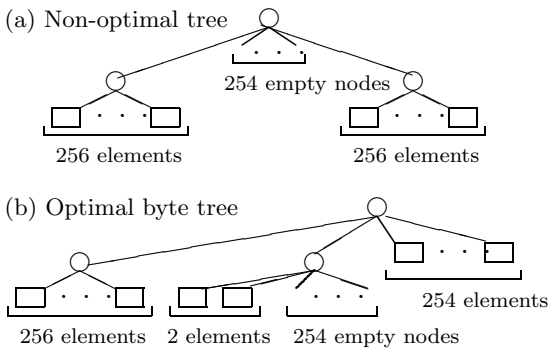


Figure 7.4 Example of byte Huffman tree.

moving the empty nodes to the deepest level of the final tree). To accomplish this, we need only to select a number of symbols equal to $1 + ((v - 256) \bmod 255)$, where v is the total number of symbols (i.e., the size of the vocabulary), for composing with the empty nodes. For instance, in the example in Figure 7.4(a), we have that 2 elements must be coupled with 254 empty nodes in the first step (because, $1 + ((512 - 256) \bmod 255) = 2$). The remaining steps are similar to the binary Huffman tree construction algorithm.

All techniques for efficient encoding and decoding mentioned previously can easily be extended to handle word-based byte Huffman coding. Moreover, no significant decrease of the compression ratio is experienced by using bytes instead of bits when the symbols are words. Further, decompression of byte Huffman code is faster than decompression of binary Huffman code. In fact, compression and decompression are very fast and compression ratios achieved are better than those of the Ziv-Lempel family [843, 844]. In practice byte processing is much faster than bit processing because bit shifts and masking operations are not necessary at decoding time or at searching time.

One important consequence of using byte Huffman coding is the possibility of performing direct searching on compressed text. The searching algorithm is explained in Chapter 8. The exact search can be done on the compressed text directly, using any known sequential pattern matching algorithm. Moreover, it allows a large number of variations of the exact and approximate compressed pattern matching problem, such as phrases, ranges, complements, wild cards, and arbitrary regular expressions. The algorithm is based on a word-oriented shift-or algorithm and on a fast Boyer-Moore-type filter. For approximate searching on the compressed text it is eight times faster than an equivalent approximate searching on the uncompressed text, thanks to the use of the vocabulary by the algorithm [573, 572]. This technique is not only useful in speeding up sequential search. It can also be used to improve indexed schemes that combine inverted files and sequential search, like Glimpse [537].

7.4.4 Inverted File Compression

As already discussed, an inverted file is typically composed of (a) a vector containing all the distinct words in the text collection (which is called the *vocabulary*) and (b) for each word in the vocabulary, a list of all documents in which that word occurs. Inverted files are widely used to index large text files. The size of an inverted file can be reduced by compressing the inverted lists. Because the list of document numbers within the inverted list is in ascending order, it can also be considered as a sequence of *gaps* between document numbers. Since processing is usually done sequentially starting from the beginning of the list, the original document numbers can always be recomputed through sums of the gaps.

By observing that these gaps are small for frequent words and large for infrequent words, compression can be obtained by encoding small values with shorter codes. One possible coding scheme for this case is the *unary code*, in

Gap x	Unary	Elias-γ	Elias-δ	Golomb $b = 3$
1	0	0	0	00
2	10	100	1000	010
3	110	101	1001	011
4	1110	11000	10100	100
5	11110	11001	10101	1010
6	111110	11010	10110	1011
7	1111110	11011	10111	1100
8	11111110	1110000	11000000	11010
9	111111110	1110001	11000001	11011
10	1111111110	1110010	11000010	11100

Table 7.1 Example codes for integers.

which an integer x is coded as $(x-1)$ one bits followed by a zero bit, so the code for the integer 3 is 110. The second column of Table 7.1 shows unary codes for integers between 1 and 10.

Elias [233] presented two other variable-length coding schemes for integers. One is Elias-γ code, which represents the number x by a concatenation of two parts: (1) a unary code for $1+\lfloor \log x \rfloor$ and (2) a code of $\lfloor \log x \rfloor$ bits that represents the value of $x - 2^{\lfloor \log x \rfloor}$ in binary. For $x = 5$, we have that $1 + \lfloor \log x \rfloor = 3$ and that $x - 2^{\lfloor \log x \rfloor} = 1$. Thus, the Elias-$\gamma$ code for $x = 5$ is generated by combining the unary code for 3 (code 110) with the 2-bits binary number for 1 (code 01) which yields the codeword 11001. Other examples of Elias-γ codes are shown in Table 7.1.

The other coding scheme introduced by Elias is the Elias-δ code, which represents the prefix indicating the number of binary bits by the Elias-γ code rather than the unary code. For $x = 5$, the first part is then 101 instead of 110. Thus, the Elias-δ codeword for $x = 5$ is 10101. In general, the Elias-δ code for an arbitrary integer x requires $1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$ bits. Table 7.1 shows other examples of Elias-δ codes. In general, for small values of x the Elias-γ codes are shorter than the Elias-δ codes. However, in the limit, as x becomes large, the situation is reversed.

Golomb [305] presented another run-length coding method for positive integers. The Golomb code is very effective when the probability distribution is geometric. With inverted files, the likelihood of a gap being of size x can be computed as the probability of having $x - 1$ non-occurrences (within consecutively numbered documents) of that particular word followed by one occurrence. If a word occurs within a document with a probability p, the probability of a gap of size x is then

$$Pr[x] = (1 - p)^{x-1} p$$

which is the *geometric distribution*. In this case, the model is parameterized and makes use of the actual density of pointers in the inverted file. Let N be the number of documents in the system and V be the size of the vocabulary. Then, the probability p that any randomly selected document contains any randomly chosen term can be estimated as

$$p = \frac{number\ of\ pointers}{N \times V}$$

where the number of pointers represent the 'size' of the index.

The Golomb method works as follows. For some parameter b, a gap $x > 0$ is coded as $q+1$ in unary, where $q = \lfloor (x-1)/b \rfloor$, followed by $r = (x-1) - q \times b$ coded in binary, requiring either $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$ bits. That is, if $r < 2^{\lfloor \log b \rfloor} - 1$ then the number coded in binary requires $\lfloor \log b \rfloor$ bits, otherwise it requires $\lceil \log b \rceil$ bits where the first bit is 1 and the remaining bits assume the value $r - 2^{\lfloor \log b \rfloor} - 1$ coded in $\lfloor \log b \rfloor$ binary digits. For example, with $b = 3$ there are three possible remainders, and those are coded as 0, 10, and 11, for $r = 0$, $r = 1$, and $r = 2$, respectively. Similarly, for $b = 5$ there are five possible remainders r, 0 through 4, and these are assigned codes 00, 01, 100, 101, and 110. Then, if the value $x = 9$ is to be coded relative to $b = 3$, calculation yields $q = 2$ and $r = 2$, because $9-1 = 2\times3+2$. Thus, the encoding is 110 followed by 11. Relative to $b = 5$, the values calculated are $q = 1$ and $r = 1$, resulting in a code of 10 followed by 101.

To operate with the Golomb compression method, it is first necessary to establish the parameter b for each term. For gap compression, an appropriate value is $b \approx 0.69(N/f_t)$, where N is the total number of documents and f_t is the number of documents that contain term t. Witten, Moffat and Bell [820] present a detailed study of different text collections. For all of their practical work on compression of inverted lists, they use Golomb code for the list of gaps. In this case Golomb code gives better compression than either Elias-γ or Elias-δ. However, it has the disadvantage of requiring two passes to be generated, since it requires knowledge of f_t, the number of documents containing term t.

Moffat and Bell [568] show that the index for the 2 gigabytes TREC-3 collection, which contains 162,187,989 pointers and 894,406 distinct terms, when coded with Golomb code, occupies 132 megabytes. Considering the average number of bits per pointer, they obtained 5.73, 6.19, and 6.43 using Golomb, Elias-δ, and Elias-γ, respectively.

7.4.5 Dictionary Methods

Dictionary methods achieve compression by replacing groups of consecutive symbols (or phrases) with a pointer to an entry in a dictionary. Thus, the central decision in the design of a dictionary method is the selection of entries in the dictionary. The choice of phrases can be made by static, semi-adaptive, or adaptive algorithms. The simplest dictionary schemes use static dictionaries containing short phrases. Static dictionary encoders are fast as they demand little effort for

achieving a small amount of compression. One example that has been proposed several times in different forms is the digram coding, where selected pairs of letters are replaced with codewords. At each step the next two characters are inspected and verified if they correspond to a digram in the dictionary. If so, they are coded together and the coding position is shifted by two characters; otherwise, the single character is represented by its normal code and the coding position is shifted by one character.

The main problem with static dictionary encoders is that the dictionary might be suitable for one text and unsuitable for another. One way to avoid this problem is to use a semi-static dictionary scheme, constructing a new dictionary for each text to be compressed. However, the problem of deciding which phrases should be put in the dictionary is not an easy task at all. One elegant solution to this problem is to use an adaptive dictionary scheme, such as the one proposed in the 1970s by Ziv and Lempel.

The Ziv-Lempel family of adaptive dictionary scheme uses the idea of replacing strings of characters with a reference to a previous occurrence of the string. This approach is effective because most characters can be coded as part of a string that has occurred earlier in the text. If the pointer to an earlier occurrence of a string is stored in fewer bits than the string it replaces then compression is achieved.

Adaptive dictionary methods present some disadvantages over the statistical word-based Huffman method. First, they do not allow decoding to start in the middle of a compressed file. As a consequence direct access to a position in the compressed text is not possible, unless the entire text is decoded from the beginning until the desired position is reached. Second, dictionary schemes are still popular for their speed and economy of memory, but the new results in statistical methods make them the method of choice in an IR environment. Moreover, the improvement of computing technology will soon make statistical methods feasible for general use, and the interest in dictionary methods will eventually decrease.

7.5 Comparing Text Compression Techniques

Table 7.2 presents a comparison between arithmetic coding, character-based Huffman coding, word-based Huffman coding, and Ziv-Lempel coding, considering the aspects of compression ratio, compression speed, decompression speed, memory space overhead, compressed pattern matching capability, and random access capability.

One important objective of any compression method is to be able to obtain good compression ratios. It seems that two bits per character (or 25% compression ratio) is a very good result for natural language texts. Thus, 'very good' in the context of Table 7.2 means a compression ratio under 30%, 'good' means a compression ratio between 30% and 45%, and 'poor' means a compression ratio over 45%.

	Arithmetic	Character Huffman	Word Huffman	Ziv-Lempel
Compression ratio	very good	poor	very good	good
Compression speed	slow	fast	fast	very fast
Decompression speed	slow	fast	very fast	very fast
Memory space	low	low	high	moderate
Compressed pat. matching	no	yes	yes	yes
Random access	no	yes	yes	no

Table 7.2 Comparison of the main techniques.

Two other important characteristics of a compression method are compression and decompression speeds. Measuring the speed of various compression methods is difficult because it depends on the implementation details of each method, the compiler used, the computer architecture of the machine used to run the program, and so on. Considering compression speed, the LZ78 methods (Unix *compress* is an example) are among the fastest. Considering decompression speed, the LZ77 methods (*gzip* is an example) from the Ziv-Lempel are among the fastest.

For statistical methods (e.g., arithmetic and semi-static Huffman) the compression time includes the cost of the first pass during which the probability distribution of the symbols are obtained. With two passes over the text to compress, the Huffman-based methods are slower than some Ziv-Lempel methods, but not very far behind. On the other hand, arithmetic methods are slower than Huffman methods because of the complexity of arithmetic coding compared with canonical Huffman coding. Considering decompression speed, word-based Huffman methods are as fast as Ziv-Lempel methods, while character-based Huffman methods are slower than word-based Huffman methods. Again, the complexity of arithmetic coding make them slower than Huffman coding during decompression.

All Ziv-Lempel compression methods require a moderate amount of memory during encoding and decoding to store tables containing previously occurring strings. In general, more detailed tables that require more memory for storage yield better compression. Statistical methods store the probability distribution of the symbols of the text during the modeling phase, and the model during both compression and decompression phases. Consequently, the amount of memory depends on the size of the vocabulary of the text in each case, which is high for word-based models and low for character-based models.

In an IR environment, two important considerations are whether the compression method allows efficient random access and direct searching on compressed text (or compressed pattern matching). Huffman methods allow random access and decompression can start anywhere in the middle of a compressed file, while arithmetic coding and Ziv-Lempel methods cannot. More recently, practical, efficient, and flexible direct searching methods on compressed texts have been discovered for word-based Huffman compression. Direct searching has also

been proposed for Ziv-Lempel methods, but only on a theoretical basis, with no implementation of the algorithms [248, 19].

More recently, Navarro and Raffinot [587] presented some preliminary implementations of algorithms to search directly Ziv-Lempel compressed text. Their algorithms are twice as fast as decompressing and searching, but slower than searching the decompressed text. They are also able to extract data from the middle of the compressed text without necessarily decompressing everything, and although some previous text has to be decompressed (i.e., it is not really 'direct access'), the amount of work is proportional to the size of the text to be uncompressed (and not to its position in the compressed text).

7.6 Trends and Research Issues

In this chapter we covered various text transformation techniques which we call simply text operations. We first discussed five distinct text operations for preprocessing a document text and generating a set of index terms for searching and querying purposes. These five text operations were here called lexical analysis, elimination of stopwords, stemming, selection of index terms, and thesauri. The first four are directly related to the generation of a good set of index terms. The fifth, construction of a thesaurus, is more related to the building of categorization hierarchies which are used for capturing term relationships. These relationships can then be used for expanding the user query (manually or automatically) towards a formulation which better suits the user information need.

Nowadays, there is controversy regarding the potential improvements to retrieval performance generated by stopwords elimination, stemming, and index terms selection. In fact, there is no conclusive evidence that such text operations yield consistent improvements in retrieval performance. As a result, modern retrieval systems might not use these text operations at all. A good example of this trend is the fact that some Web search engines index all the words in the text regardless of their syntactic nature or their role in the text.

Furthermore, it is also not clear that automatic query expansion using thesaurus-based techniques can yield improved retrieval performance. The same cannot be said of the use of a thesaurus to directly assist the user with the query formation process. In fact, the success of the 'Yahoo!' Web search engine, which uses a term categorization hierarchy to show term relationships to the user, is an indication that thesaurus-based techniques might be quite useful with the highly interactive interfaces being developed for modern digital library systems.

We also briefly discussed the operation of clustering. Since clustering is more an operation of grouping documents than an operation of text transformation, we did not cover it thoroughly here. For a more complete coverage of clustering the reader is referred to Chapter 5.

One text operation rather distinct from the previous ones is compression. While the previous text operations aim, in one form or another, at improving the quality of the answer set, the operation of compressing text aims at reducing space, I/O, communication costs, and searching faster in the compressed text

(exactly or approximately). In fact, the gain obtained from compressing text is that it requires less storage space, takes less time to be transmitted, and permits efficient direct and sequential access to compressed text.

We covered in greater detail text compression methods that are suitable for we in an IR environment. We first reviewed basic concepts related to compression in general, the relationship between probabilities and codes, and the role of entropy as a lower bound on the amount of compression that can be obtained. We then considered statistical methods with emphasis in word-based Huffman methods, followed by inverted file compression. We also briefly discussed compression schemes based on dictionary.

For effective operation in an IR environment, a compression method should satisfy the following requirements: good compression ratio, fast coding, fast decoding, fast random access without the need to decode from the beginning, and direct searching without the need to decompress the compressed text. A good compression ratio saves space in secondary storage and reduces communication costs. Fast coding reduces processing overhead due to the introduction of compression into the system. Sometimes, fast decoding is more important than fast coding, as in documentation systems in which a document is compressed once and decompressed many times from disk. Fast random access allows efficient processing of multiple queries submitted by the users of the information system. We compared various compression schemes using these requirements as parameters. We have seen that it is much faster to search sequentially a text compressed by a word-based byte Huffman encoding scheme than to search the uncompressed version of the text. Our discussion suggests that word-based byte Huffman compression (which has been introduced only very recently) shows great promise as an effective compression scheme for modern information retrieval systems.

We also discussed the application of compression to index structures such as inverted files. Inverted files are composed of several inverted lists which are themselves formed by document numbers organized in ascending order. By coding the difference between these document numbers, efficient compression can be attained.

The main trends in text compression today are the use of semi-static word-based modeling and Huffman coding. The new results in statistical methods, such as byte-Huffman coding, suggest that they are preferable methods for use in an IR environment. Further, with the possibility now of directly searching the compressed text, and the recent work [785] of Vo and Moffat on efficient manipulation of compressed indices, the trend is towards maintaining both the index and the text compressed at all times, unless the user wants to visualize the uncompressed text.

7.7 Bibliographic Discussion

Our discussion on lexical analysis and elimination of stopwords is based on the work of Fox [261]. For stemming, we based our discussion on the work of Frakes [272]. The Porter stemming algorithm detailed in the appendix is

from [643], while our coverage of thesauri is based on the work of Foskett [259]. Here, however, we did not cover automatic generation of thesauri. Such discussion can be found in Chapter 5 and in [734, 730]. Additional discussion on the usefulness of thesauri is presented in [416, 730].

Regarding text compression, several books are available. Most of the topics discussed here are covered in more detail by Witten, Moffat and Bell [820]. They also present implementations of text compression methods, such as Huffman and arithmetic coding, as part of a fully operational retrieval system written in ANSI C. Bell, Cleary and Witten [76] cover statistical and dictionary methods, laying particular stress on adaptive methods as well as theoretical aspects of compression, with estimates on the entropy of several natural languages. Storer [742] covers the main compression techniques, with emphasis on dictionary methods.

Huffman coding was originally presented in [384]. Adaptive versions of Huffman coding appeared in [289, 443, 784]. Word-based compression is considered in [79, 567, 375, 75]. Bounds on the inefficiency of Huffman coding have been presented by [289]. Canonical codes were first presented in [708]. Many properties of the canonical codes are mentioned in [372]. Byte Huffman coding was proposed in [573]. Sequential searching on byte Huffman compressed text is described in [573, 572].

Sequential searching on Ziv-Lempel compressed data is presented in [248, 19]. More recently, implementations of sequential searching on Ziv-Lempel compressed text are presented in [588]. One of the first papers on arithmetic coding is in [670]. Other references are [818, 76].

A variety of compression methods for inverted lists are studied in [569]. The most effective compression methods for inverted lists are based on the sequence of gaps between document numbers, as considered in [75] and in [568]. Their results are based on run-length encodings proposed by Elias [233] and Golomb [305]. A comprehensive study of inverted file compression can be found in [820]. More recently Vo and Moffat [785] presented algorithms to process the index with no need to fully decode the compressed index.