INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

Université d'Ottawa · University of Ottawa

Intelligent Document Format

A text encoding scheme

by
Charles Tran

Thesis
submitted to the School of Graduate Studies and Research
of the University of Ottawa
in partial fulfillment of the requirements for the
Master's Degree in Computer Science
Under the Auspices of
the Ottawa-Carleton Institute for Computer Science

Department of Computer Science
University of Ottawa
March 1996

Abstract

The issue of text representation is very important in text retrieval and natural language processing. The way the data is represented can significantly affect the efficiency of storage, retrieval, routing techniques, query formulation, and information extraction. This thesis describes a potential solution to encode documents efficiently and effectively so that information may be easily retrieved.

In this thesis, we present a technique for encoding textual data in a representation format called the *Intelligent Document Format (IDF)*. The IDF encodes English text so as to store textual data efficiently, permitting retrieval of text at sentence-, paragraph-, and document levels, and assisting term searching and retrieving, as well as providing linguistic processing such as morphological analysis and sense disambiguation. To illustrate the IDF encoding method, we describe IDFconvert, an IDF encoder and decoder program, and carry out encoding experiments on the electronic-text version of *Dracula* (Stoker 1897).

Acknowledgments

I wish to thank my supervisor, Dr. Douglas R. Skuce, for his guidance and for many stimulating discussions throughout this research. I am particularly grateful to him for giving me the freedom to pursue my own research. I would also like to thank Judy Kavanagh, Laura Proctor, Noreen Harrigan, Yzad Zayour, and Kristen MacKintosh for giving me countless valuable advice on my work, and for making the LAKE Lab a very enjoyable place to work.

I also want to take this opportunity to say thanks to Dr. Denys Duchier, he who had inspired me to investigate and learn the many treasures of Unix. I feel very fortunate to have had the opportunity to work with and learn from Denys.

To my friends Randy, Eric, Pankaj, Tuong, Thien, Huy and Rachel, I am grateful for your friendship and support.

Last, but not least, I wish to thank my parents, my sister Elizabeth, and my brothers Alex and John, for always being there.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Purpose

This thesis will present a technique for encoding textual data in a representation format called the *Intelligent Document Format (IDF)*. The IDF is a text encoding scheme to be used with natural language documents. The purpose of IDF encoding is to store the document efficiently and to facilitate the finding and understanding of words and sentences in the document. This representation format also provides support for certain linguistic processing such as morphological analysis and sense disambiguation.

In recent years the growth of the Internet has resulted in an overwhelming amount of on-line text. In response, there is a strong interest in finding ways to organize and access the information efficiently and effectively. Information retrieval (IR) is a discipline concerns with problems of managing and processing of information by use of computer. The problems of information retrieval, however, are not central and can be tackled from many angles. For example, the area of natural language processing (NLP) have contributed much to the success

of IR. Hence accordingly, IR can be viewed as a multidisciplinary field depending on the types of problems being addressed.

In order to gain better insight into the problems related to IR, we will give a very brief overview of the issues addressed in the IR and NLP research communities (sections 1.2, 1.3).

1.2 Information Retrieval

An information retrieval task concerns mainly with the locating and retrieving of text from a database in response to a user's query. IR researchers traditionally have been exploring a wide range of issues such as indexing techniques [1, 2, 3, 4], querying and ranking methods [5, 6, 7, 8], and IR conceptual models [3, 8, 9, 11]. In current large scale IR projects (e.g. TRECs, TIPSTER, and MUCs) researchers focus their efforts on the mentioned issues and as well as numerous other topics ranging from routing, ad-hoc retrieval techniques [9] to template filling and information extraction techniques [10].

An *index* is a mechanism that is used by an IR system to locate a given term in a text. An indexing structure plays a central role in an IR system because it affects the system's ability to resolve queries efficiently, and this in turn, influences how quickly text can be identified and retrieved. Although many approaches to indexing exist more commonly used techniques in IR systems are inverted files, clustered files, and signature files. Research in this area investigate topics such as implementation [12], compression [13], and generation [14] of inverted file.

Querying is a mechanism that is used by IR systems to make use of an index to locate documents in a textual database. Belkin and Croft [8] categorized retrieval (conceptual) models to be of two types: exact match and inexact match. And corresponding to these models, queries are either *Boolean* or *ranked* queries.

Researchers [5, 8, 15] discussed the advantages and disadvantages of each query type.

A Boolean query is constructed from a list of key words[1] that are connected together by connectives such as AND, OR, and NOT. An example of a Boolean query is *(Information **OR** Text) **AND** Retrieval*. The responses to such a query are documents that satisfied the Boolean condition. The results must have all searched key words occurring somewhere in the text but they need not to be adjacent nor appear in any specific order.

A ranked query, on the other hand, allows the user to input a sentence or a phrase (no Boolean connectors) and have the system retrieve a list of documents ranked in order of likely relevance. The documents retrieved are ranked based on factors such as the neighborhood of occurrence of the query words and *term weighting* model (e.g. statistical model and probabilistic model). In contrary to a Boolean query, a ranked query method is more lenient in its syntax and will produce results even if a query term is incorrect, e.g., it is not the same term as the one that is indexed from relevant documents. Standard works done in this field include experimentation with variations of the ranking models [5, 8, 15].

1.3 Natural Language Processing

Although retrieval systems offer very powerful on-line search capabilities they are deemed inadequate when it comes to complex information retrieval purposes (e.g. extraction of selected information from on-line sources) [16]. Conventional IR systems normally use complex Boolean queries with word-based ranking procedures to produce inadequate, even though state-of-the-art results [17]. In using a full-text retrieval system one can expect to get a 75 percent

[1] From the inverted file.

precision rate[2] and a 20 percent *recall* rate[3]. This means that the system requires the user to manually read irrelevant text as well as relevant ones in order to filter out what he seeks.

What are the gains in using NLP approaches in the retrieval context? Researchers [1, 18] suggested the benefits may come from the use of free language formulations by retrieval system users, both for the submission of initial query statements and for the various interactive processes in documents are adjusted based on information obtained from the users population. To be specific, [1, 18] reviewed that levels of NLP that are of interest to IR include morphological, lexical, syntactic, and semantic levels of language analysis for tasks like automatic and semi-automatic indexing to text, text retrieval, text abstracting and summarization, and conceptual information retrieval. Detailed overviews of NLP techniques can be found in [19, 20].

The morphological level of processing is concerned with the processing of individual word forms and of recognizable portions of words. Morphological analysis includes the removal of word suffixes and prefixes and the generation of word stems. These operations can be used to enhance search recall [1].

The *lexical* level of processing operates at the single word level, and is concerned with issues like common word deletion, dictionary processing of words, or replacement of words from thesaurus classes. This type of analysis also includes identifying linguistic features of words (noun, adjective, preposition, etc.) to be used in the syntactic analysis process. An application of lexical analysis to IR is the generation of an index based on some root or inflected form of words occurring in the text. This is an alternative method to the stemming and conflation techniques commonly used in the construction of index.

---

[2] The percentage of relevant documents compared with the total number of documents retrieved.
[3] The percentage of relevant documents actually retrieved in response to the query.

The *syntactic* level of processing is concerned with the "*grouping of words of a sentence into structural units such as prepositional phrases, and subject-verb-object groupings that collectively represent the grammatical structure of the sentence*" [1]. A syntactic analysis often uses a part-of-speech tagger [23, 24] to parse documents to set up the frame work for the recognition of structural characteristics of natural language text. This level of processing have been used to assist index text in terms of elements more complex than word units. For example, Sack-Davis *et al.* [21] experimented with syntactic analysis on texts and then indexed them by syntactic labels indicating whether a word is a head of a clause or a modifier, however, the experiment showed little improvement in retrieval effectiveness.

Finally, the *semantic* level of processing done on syntactically analyzed units of text to resolve contextual interpretation. Unlike lexical and syntactic analyses where indicators text content are key words, word senses, syntactic labels, determining of accurate semantic representations of information is difficult [18]. Nevertheless, semantic representation of text is commonly is implemented using *frames* [19], but there are disadvantages with frames because they require large, domain-specific knowledge base to support their construction. An example of a retrieval system that uses semantic level NLP is the RIME system [22].

1.4 The scope of the thesis

In this work, we focus on the IDF file structure and its values to both IR and the NLP aspect of IR. The IDF encoding scheme allows a retrieval system to identify texts (IDF encoded) that contain queried key words but without the use of an index. In addition, IDF documents store words in their root forms (similar to stems) and can have *part-of-speech-tags* embedded within it, a feature that is useful for sentence analysis methods associated with syntactic structures.

In summary, we will describe a format for storing documents that assist the following type of operations:

- storing of text compactly through encoding
- allowing a retrieval system to locate text without the use of an index
- searching and retrieval of terms
- storage of word syntax information (part of speech and inflection) to permit searching for words based on syntax.[4]

1.5 Organization of the thesis

Chapter 2 describes the background information relevant to this thesis, particularly on the subject of text encoding, the common file structures used to store information, and how these structures influence the way which information is stored and retrieved.

Chapter 3 gives the IDF specification. The chapter begins with a brief discussion of the goals of the Intelligent Document Format, followed by a detailed description of the IDF encoding specification.

Chapter 4 discusses the encoding and decoding algorithms as well as implementation issues concerning the prototype IDF encoder-decoder program.

Chapter 5 describes the encoding experiments carried out with IDFconvert, a prototype implementation of the IDF encoder and decoder.

Chapter 6 presents a summary of the thesis, concluding remarks and discussion of possible future work.

[4]In addition to storing word syntax information, the IDF can also store information that allows the discrimination between word senses.

1.6 References

[1] Salton, G. and M. J. McGill, *Introduction to Modern Information Retrieval.* NY, McGraw-Hill, 1983.

[2] Casey, R. G. Design of tree structures for efficient querying, *Communications of the ACM*, **16**, 548-566, 1973.

[3] Faloutsos, C., Access methods for text. *ACM Computing Surveys*, **17**, 50-74, 1985.

[4] Stanfel, L. E., Trees structures for optimal searching. *Journal of the ACM*, **17**, 508-517, 1970.

[5] Harman, D., Relevance Feedback and Other Query Modification Techniques, *Information retrieval: Data Structures and Algorithms*, W. B. Frakes and R. Baeza-Yates, eds., Prentice-Hall, 241-263, 1992.

[6] Harman, D., Ranking Algorithms, *Information Retrieval: Data Structures and Algorithms*, W. B. Frakes and R. Baeza-Yates, eds., Prentice-Hall, 363-392, 1992.

[7] Lucarella, D., A Document Retrieval System Based on Nearest Neighbor Searching. *Journal of Information Science*, **6**, 25-33, 1983.

[8] Belkin, J. J. and W. B. Croft, Retrieval Techniques, *Annual Review of Information Science and Technology*, ed. M. Williams, Elsevier Science Publishers, 109-145, 1987.

[9] Harman, D., Overview of the Fourth Text REtrieval Conference (TREC-4), *Proceedings of the Fourth Text Retrieval Conference*, 1994.

[10] Sundheim, B. M., Overview of the Third Message Understanding Evaluation and Conference, *Proceedings of the Third Message Understanding Conference (MUC-3)*, Morgan Kaufmann, 3-16, 1991.

[11] Strzalkowski, T. and J. Perez-Carballo, Natural Language Information Retrieval: TREC-4 Report, *Proceedings of the Fourth Text Retrieval Conference*, 1994.

[12] Buckley, C. and A. F. Lewit, Optimization of inverted vector searches. *Proceedings of ACM-SIGIR International Conference on Research and Development in Information Retrieval*, ACM Press, 97-110, 1985.

[13] Bookstein, A., S. T Klein, and T. Raita, Model based concordance compression. *Proceedings of the IEEE Data Compression Conference*, J. Storer and M. Cohn, eds., IEEE Computer Press Society, 82-91, 1992.

[14] Fox, E. A. and W. C. Lee, FAST-INV: A fast algorithm for building large inverted files. Technical Report 91-10. Virginia Polytechnic Institute and State University, 1991.

[15] Witten, I. H., A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images*, Van Nostrand Reinhold, 116-174, 1994.

[16] Jacobs, P. S. and L. F. Rau, SCISOR: Extracting Information from On-line News, *Communications of the ACM*, **33**, (11), 88-97, 1990

[17] Blair, D. C. and M. E. Maron, An evaluation of retrieval effectiveness for a full-text document retrieval system. *Communication of the ACM*, **28**, (3), 289-299, 1985.

[18] Smeaton, A. F., Progress in the Application of Natural Language Processing to Information Retrieval Tasks, *The Computer Journal*, **35**, 268-278, 1992.

[19] Gazdar, G. and C. Mellish, Natural Language Processing in LISP: *An Introduction to Computational Linguistics*, Addison-Wesley, 1989.

[20] Grishman, R., *Compuational Linguistics: An Introduction*, Cambridge University Press, 1986.

[21] Sacks-Davis, R. *et al.*, Using syntactic analysis in a document retrieval system that uses signature files. *Proceedings of the 13th International SIGIR Conference on Research and Development in Information and Retrieval*, J. L. Vidick, ed., 179-192, 1990.

[22] Berrut, C., Indexing medical reports: the RIME approach, *Information Processing and Management*, **26**, (1), 93-110, 1990.

[23] Brill, E., A simple rule-based part of speech tagger, *Proceedings of the Third Conference on Applied Natural Language Processing, ACL*, 1992.

[24] Church, K., A stochastic parts program and noun phrase parser for unrestricted text, *Proceedings of the Second Conference on Applied Natural Language Processing*, ACL, 136-143, 1988.

2. Encoding Concepts

In this chapter we give a brief discussion of text encoding concepts. In addition, we will review the *symbolwise* and *dictionary-based* text encoding approaches, and look at some encoding methods that are based on these models.

2.1 Text Encoding

Consider a set of *symbols* S = {a, b, c, d, e}. An *encoding* of S is an assignment of a *codeword* (binary sequence) to each symbol. For example, we can define one encoding of S as

a = 000, b = 001, c = 010, d = 011, e = 100,

and another encoding as

a = 00, b = 01, c = 10, d = 11, e = 100.

In general, the process of *text encoding* on a computer is concerned with the changing of the representation of a *text file* from one format to another. This process is to be done in such a way that the original file can be reconstructed exactly from the encoded representation. Normally, the encoding involves the

conversion of the text, which is a symbol (an ASCII character) or group of symbols depending on the model, into a shorter representation, in order to compress the file.

2.2 Approaches to encoding

Most *text encoding* methods can be categorized to be one of two approaches: symbolwise and dictionary [1, 2, 3, 4, 9]. The following two sections explain some text encoding models and give examples of these approaches.

2.2.1 Symbolwise approach

Text encoding methods based on symbolwise approach work by *estimating the probabilities* of symbols, encoding one symbol at a time, using shorter binary codewords for the more likely symbols and longer codewords for the rarer ones. Compression is achieved if the length of the codewords is always shorter than that of the symbols. An example of a method based on the symbolwise approach is the Morse code.

Since these methods encode symbols based on the estimated probabilities of the occurrence of the symbols, the more accurate the estimates, the greater the compression ratio achieved. Examples of compression methods based on the symbolwise approach are Huffman coding [5] and arithmetic coding [6].

In the sections below we examine two exemplary symbolwise encoding models: *Huffman* and *word-based* coding. We have paraphrased Witten *et al*'s [2] descriptions of the two approaches in sections 2.2.1.1 and 2.2.1.2.

2.2.1.1 Huffman method

For a given set of symbols the Huffman method uses a probability distribution for the symbols to generate the codewords. Table 2-1 shows an encoding for the set of symbols S = {a, b, c, d, e, f}.

Using this table, a string of symbols in S is encoded by replacing a symbol with the codeword given by the table. For example, a string like *eafc* is encoded as 100000110001.

Symbol	Codeword	Probability
a	0000	0.05
b	0001	0.05
c	001	0.2
d	01	0.25
e	10	0.25
f	110	0.2

Table 2-1. Codewords and probabilities for S.

An encoding is *ambiguous* if there exists a binary sequence which is the encoded form of more than one string of symbols. Thus, to be sure that an encoding is unambiguous, it is necessary to find a decoding algorithm (Huffman's) for it. Huffman's algorithm is based on constructing a binary *decoding tree.* The decoding tree is built from the bottom up, left branches are labeled 0, right branches are labeled 1, and each external node is labeled with the symbol. The codeword for an external node is the path containing the labels from the root to that node (Figure 2-1). A detailed description of Huffman encoding algorithm can be found in [5].

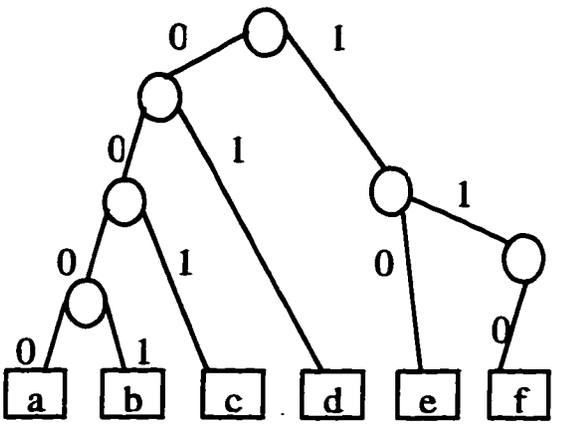

Figure 2-1. Decoding tree for set S.

The decoding operation is performed from left to right. The operation is stopped when a complete codeword is obtained. For example, to decode 100000110001, we follow the path 1-0 to the node *e*, then start the decoding again follow the path 0-0-0-0 to node *a*. The procedure is repeated for the remainder of the encoded string.

2.2.1.2 Word-based encoding methods

Word-based methods are another encoding technique based on the symbolwise approach. The method isolates (i.e. segments) "words[5]" and "nonwords" in a document. In the view of the symbolwise approach, words and nonwords constitute the two kinds of symbols to be encoded. Normally, the method uses two different models for the encoding, one for words and one for nonwords. It assumes that the text consists of strictly alternating words and nonwords (the word segmenter needs to ensure this), and so the two models can be used alternately. If the models are adaptive, then a means of encoding unrecognized words and nonwords is required. Usually, some escape symbol is

[5] Words are, typically, contiguous alphanumeric characters.

used as a delimiter marker and then the unknown string is stored in a character by character manner.

Witten *et al.* [2] provided an example of word-based encoding method where the symbols, i.e. units of text that get to be encoded, are the words that appear in the book. The frequency of each word and total number of words in the file are counted. Then the information is used to compute the probability of each word, and Huffman's algorithm is used to generate the codewords.

2.2.2 Dictionary Approach

Text encoding methods based on the dictionary methods work by replacing words and other fragments of text with an index to an entry in a *coding dictionary*. The Braille code is an example of the dictionary method, it illustrates how special codes are used to represent whole words. There are two main differences between the dictionary approach and the symbolwise approach:

1. dictionary-based methods usually employ some hash function to produce the codewords whereas symbolwise methods use estimated probabilities to generate the codewords.
2. codewords generated by dictionary-based methods are fixed in length, whereas codewords generated by symbolwise methods are varying in length.

In the sections below we examine two encoding models: *diatomic* and *LZ77* coding. These are examples of the dictionary-based encoding approach.

2.2.2.1 Diatomic method

One of the simplest dictionary based methods is the *diatomic encoding* method [2, 7]. In the set of ASCII characters, the 128 characters are represented by 7-bit codewords. Although all the codewords are 7-bits long, they are stored in 8-bit structures with the eighth bit (parity bit) not used. The diatomic method

improves the efficiency in data storage by 50 percent. It uses a coding dictionary to contain 8-bit codewords. The dictionary would contain all the 128 ASCII characters, and in addition, 128 common letter pairs. For example, some of the common letter pairs might be *an, er, in, is, of, to, ll, th* and *nn*, etc. Since the dictionary contains the complete ASCII character set, the scheme ensures that any input will be mapped to some codeword. In the best case scenario, every pair of characters is replaced by an 8-bit codeword, thus we have two 7-bit characters represented by one 8-bit codeword. In the worst case scenario, each input character is represented as one 8-bit codeword.

However, we must always be prepared to deal with unexpected input, that is, input that are non-ASCII bytes.[6] To do this, one codeword is reserved as an escape code to indicate that the next input byte is to be interpreted as a single 8-bit character rather than as a codeword for a pair of ASCII characters.

The diatomic coding scheme may be modified to have larger entries in the coding dictionary. Thus, instead of using 8-bit codewords to represent pairs of characters, they can be used to represent longer words, for example, common words in English like *and, the, this* and *that.* However, in using a dictionary with a predetermined set of words we narrow down the amount of suitable text for encoding. In order to have input independence, the dictionary entries should be short. Often when the dictionary is more suitable for a certain type of text, it tends to be less suitable for another type. For instance, if this thesis were to be encoded in this scheme, then it would do well if the coding dictionary contains entries like *codeword, text, encode*, and *decode.* On the other hand, it would be useless if the input text is on the subject of, say, biology.

[6] Non-ASCII bytes have values ranging from 128-255.

2.2.2.2 LZ77 method

Most well known dictionary based encoding methods are based on the compression technique known as the LZ77 coding, named after its inventors [8]. The LZ77 coding technique uses the principle of replacing strings of characters with a reference to a previous occurrence of the string. This approach is adaptive and effective because most characters can be coded as part of a string that has occurred earlier in the text.

The LZ77 method works by replacing a substring of text with a pointer to where it has occurred previously. Therefore, the coding dictionary contains all the text prior to the current "encoding" position, and the codewords are represented by pointers. The many variants of LZ77 methods differ primarily in how the pointers are represented and in the limitation they impose on what the pointers are able to refer to.

Since the algorithm of the LZ77 method is rather complicated to explain, the reader is referred to the reference [8] for detailed information concerning their algorithm.

2.3 References

[1] Gutmann, P. C. and T. C. Bell, A Hybrid approach to text compression, *Proceedings IEEE Data Compression Conference*, IEEE Computer Society Press, 225-33, 1994.

[2] Witten, I. H., M. Alistair, and T. Bell, *Managing Gigabytes: Compressing and indexing documents and images*, Van Nostrand Reinhold, 1994.

[3] Moffat, A., N. Sharman, and J. Zobel, Static compression for dynamic texts, *Proceedings IEEE Data Compression Conference*, IEEE Computer Society Press, 126-35, 1994.

[4] Langdon, G. G., A note on the Ziv-Lempel model for compressing individual sequences, *IEEE Transactions on Information Theory*, **29**, 284-287, 1983.

[5] Huffman, D. A., A method for the construction of minimum redundancy codes, *Proceedings of the IRE*, **40**, 1098-1101, 1952.

[6] Rissanen, J. and G.G. Langdon, Arithmetic Coding, *IBM Journal of Research and Development*, **23**, 149-162, 1979.

[7] Held, G. and T. R. Marshall, *Data Compression: Techniques and applications, Hardware and software Considerations*, Third ed., John Wiley & Sons, 100-109, 1991.

[8] Ziv, J. and A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, **23**, 337-3343, 1977.

[9] Store, J. A., *Data compression: Methods and Theory*, Computer Science Press, 1988.

3. Intelligent Document Format

3.1 Introduction

This chapter gives details on the Intelligent Document Format (IDF). The IDF is a technique for representing textual information efficiently. We use the term *representation* here to refer to the encoding of information for interchange. For example, the ASCII format is a representation of plain text, and Rich Text Format (RTF) is a representation of formatted text and graphics for interchange between software applications on different platforms. Therefore, from this point of view the IDF specification is a method of encoding English text to facilitate text retrieval and linguistic processing tasks. With IDF encoding, documents encoded under different operating systems with different software applications can be transferred between those operating systems and applications.

The Intelligent Document Format is designed to support several functions desired in text-based retrieval and text analyzer systems. These features include the compression of the document, term indexing, searching

and easy accessibility of data at the sentence-, paragraph-, and document-level. Other notable characteristics include the optional abilities to store word syntax (part of speech and inflection) and word sense information. The storing of word syntax information permits searching and/or replacing all forms of a word or retrieval based on the syntax. On the other hand, the storing of word senses helps to understand the context in which a word is used and to disambiguate words (e.g. for machine translation).

There are two aspects of the IDF encoding method. The first is the storing of text to facilitate text retrieval. The second, which is optional, is to facilitate certain linguistic processing that would be useful in syntactic analysis. The IDF encoding technique also can accommodate a text that contains embedded part of speech, so that some linguistic information may be easily extracted from the text.

A program that takes a text file and turns it into an IDF file will be called an IDF *encoder*. A program that translates an IDF file into a human readable format will be called an IDF *decoder*. In Chapter 4 we will discuss the algorithms and implementation for both IDF encoder and decoder respectively.

3.2 Design Consideration

Here are some of our design goals for the Intelligent Document Format. Each of these points is discussed below.

- reduction of file size by means of text encoding

- applying case folding[7] to **most** words to simplify word searching
- categorizing text strings to be either ***word*** or ***exception*** types. Word types are further classified into three subgroups: high-frequency, standard and special words. Exception types can be either *coded* or *uncoded* (to be explained below).
- efficiently encoding words to handle inflections
- storing part of speech tags.

3.2.1 IDF Encoding

The motivations for IDF encoding are two-fold: to assist text retrieval and to facilitate certain linguistic processing. With an IDF encoded file, the retrieval of text is usually done at the document level. However, units of text at subdocument level can also be retrieved. When text retrieval occurs at the document level the original text is reconstructed from the IDF encoded file. For example, if we want to view all *paragraphs* containing some words then only those paragraphs which were found to contain the searched words are decoded. The ability to retrieve only sections of a document that contain the sought after term(s) offers a very useful text retrieval operation.

3.2.2 Case Folding

To simplify word searching, a "case folding" transformation is applied to most words in a document when it is converted to IDF. Those words whose letters are either all capitalized or only the initial letter is capitalized will be case folded. Case folding involves converting the uppercase characters in a word to their lowercase equivalents. For instance, "OPERATION" and "Operation" would be case folded to the representative word "operation", with the difference

[7] converting to lower case.

encoded. Words which contains mixed cases are treated differently. They neither undergo case folding nor will they be encoded.

The case folding transformation is done so that a user who is looking for information need not be concerned with the exact case of words occurring in documents. In other words, case folding of terms makes it possible to have case-independent searches. Therefore, any input to a retrieval system, i.e. a search expression, will be case folded before it is searched for in the IDF file. Obviously, there are circumstances where the exact case of the terms in a query is important.

Suppose that the user is looking for a document containing "Intelligent Document Format" (i.e. this thesis), and does not want to get flooded with documents about "intelligent quotient test score", "paper documents", or "how to perform a low-level format of your hard drive". If a search query to the retrieval system contains the key words: *intelligent* and *document*, and *format*, then the retrieval system will likely return both the wanted and unwanted documents. Hence, this example illustrates a situation where the user may wish to use case sensitive word searching, and why it is necessary for the IDF representation to preserve the case attribute of the encoded word.

3.2.3 Document components

An ASCII document is a type of file that contains only one type of data — ASCII text, and no other data types such as sound or graphics. This section describes the relationship between an IDF encoded document and a normal ASCII document. It also provides an overview of the model of the ASCII document to IDF document transformation process.

The term '*ASCII text*' is an attribute of a text file that distinguishes this type of file from any other text file that may be encoded in a different

representation format. To be more specific, we characterize a file as ASCII text if it were found to contain natural language text with no formatting and encoded using only ASCII characters. The ASCII text attribute indicates only the style of which the information is encoded and not the language protocol for representing the information. For example, SGML and Postscript files are both ASCII files differing only in the formatting of data, whereas a file that contains only text strings but is encoded in some other proprietary format, say MS-Word format, is not an ASCII file. For the remainder of this chapter, we will use the term "text" to refer to ASCII text.

We begin with the definitions of the important terms used in the thesis.

Definitions

1. A ***token*** is a contiguous sequence of one or more alphanumeric[8] characters delimited by white space[9] on either side.

2. A ***word*** is a token composed of contiguous alphabetic characters; it may include hyphens and apostrophes but no other punctuation marks. It is possible to have a multi-token word.[10]

3. An ***exception token*** is a token which there is no entry for it in an English dictionary. Examples include a structure markup tag, a punctuation, numbers, ordinals, proper names, acronyms, and abbreviations.

A text document can be viewed as a sequence of word tokens connected by exception tokens such as punctuation marks, white spaces, markup tags, and

[8] letter or number.
[9] A space, tab, carriage-return, or line-feed character.
[10] An example of a multi-token word is "state-of-the-art".

control characters, e.g. horizontal and vertical tabs, form feed, new-line, and page-break. The word tokens are divided into three subgroups as follows:

- high-frequency words,
- standard words,
- special words.

The exception tokens are made up of the two groups:

- coded exception tokens,
- uncoded exception tokens.

Tokens are encoded in blocks of four types:

1. high-frequency word block
2. standard word block
3. special word block
4. exception block.

3.2.3.1 High Frequency Words

A list of *high-frequency words* is usually shared by all documents, however, the user may choose to use his/her own list of high-frequency words in the encoding of a particular document. This list contains some of the most frequently occurring English words. For example, *a*, *and*, *from*, and *the* are typical high-frequency words. The list is used to filter out words appearing in a search query because if such words are not removed then the search is likely to retrieve almost every document in the database regardless of its relevance [1]. In addition, these words make up a large fraction of the text of most documents. The ten most frequently occurring words in English account for 20 to 30 percent of the tokens in a document [2].

3.2.3.2 Standard words

In a file to be IDF encoded, the majority of words are *standard words*. Typically, standard words are words that can found in an English dictionary. Any word which has a meaning in English can be a standard word unless it has already been determined to be a high-frequency word or a special word (to be discussed below.) An example to illustrate the parsing of words in a sentence is shown below.

It was early in the morning, but our mother was already up cooking breakfast.

If that the sentence contains no special words, it can be categorized as follows:

Hi-frequency	Standard	Exception
it	early	" (quote)
was	morning	, (comma)
in	our	. (period)
the	mother	
but	already	
up	cooking	
	breakfast	

3.2.3.3 Special words

A document that contains information pertaining to some domain may have certain words or phrases (collocations) that bear special, distinguished meanings when they are used in the context of that domain. We refer to such words as *special words*. For example, suppose that we have a document on the subject of Object Oriented Programming in C++. An expert on this subject would likely agree that words such as *class*, *object*, *method*, *message*, and *argument* all have very specific meanings—they represent certain concepts when they are used in an OOP context. However, those same words may have a completely different meaning than in general language or in a different domain.

Thus, with respect to the document Object Oriented Programming in C++, "class", "object", "method", "message", and "argument" are considered to be special words. In Chapter 4, we will discuss about how words in a document become special words.

In order for the IDF encoder to encode special words differently from the other word tokens, it must be able to decide when a word is a special word. To do this it must have access to a dictionary containing all special words in the document. This dictionary which reflects some particular domain of knowledge is provided by the user.

3.2.3.4 Implementing special word dictionary

The special word dictionary needs to be implemented in a machine-readable form so that an IDF encoder can access it. One implementation is to make it a flat ASCII file that contains one word entry per line.

There are two ways for the user to create this file. The user either creates it manually by using a text editor to enter all terms into a file, or uses a program similar to an index builder to generate the file. There is, however, a small problem with the flat file implementation. Different morphological forms of a word must be stored as different entries. For example, "method" and "methods" will have to be stored as two separate entries in the dictionary. The reason for this is the lack of sophistication of a plain flat file and the way an IDF encoder work.

3.2.3.5 Exception tokens

All tokens in a document that belong to none of the three word groups are collectively referred to as *exception tokens*. These tokens include numbers, date, time, mis-spelled words, proper names, abbreviations, words that contain mixed case, and all the punctuation marks. In addition, document markup tags such as

those defined by the Standardized General Markup Language (SGML) to give a document structures are also classified as exception tokens. An example of a typical classification of word tokens and exception tokens in a text is shown below.

Text

It has been nearly six years since Tim Bajari and I agreed to launch a newsletter on multimedia computing, a newsletter that eventually became the journal you're now reading.

Hi-frequency word tokens

It	has	been	and	on	to
a	that	now	the	I	

Standard word tokens

nearly	six	years	since	agreed
launch	newsletter	multimedia	computing	eventually
became		journal	reading	

Special word tokens

Tim	Bajari	you' re

Exception tokens

, (comma)	. (period)

3.2.3.6 General behavior of an IDF encoder

The behavior of an IDF encoder has two distinct phases: a) dictionary initialization and b) document encoding.

Before the encoder processes a document, it must initialize its internal dictionaries (lookup tables.) The high-frequency word and special word dictionaries are initialized to contain entries found in the respective dictionary

files while the standard word dictionary is initialized to contain null entries. The latter is different from the others because standard words are determined dynamically. That is, the encoder uses an external dictionary to decide whether a token to be encoded is a standard word. Only when the external dictionary has determined that the token is a standard word an entry for it is created (if needed) in the encoder's standard word dictionary. Thus, it can be summarized that the differences between an encoder's standard word and special word dictionaries are twofold.

First, the special word dictionary is statically generated while the standard word dictionary is dynamically generated. That is, the size of the former is known as soon as its initialization is completed, while the size of the latter is known only after the encoder has finished encoding a document. Second, functionally, the special words dictionary is used to determine whether a given token is a special word which is contrary to that of the standard word dictionary whose function is to act as a repository for encoded codewords.

In the encoding phase, tokens in a document are encoded one at a time. The encoder checks a token against the lookup tables to determine its type. If an entry is found in some lookup table then the token type is ascertained, and consequently its encoding method is known. Therefore in order for a successful lookup the dictionary must contain an entry that is identical to the input token.

Figure 3-1 shows an overview of the model for the IDF encoding and decoding processes.

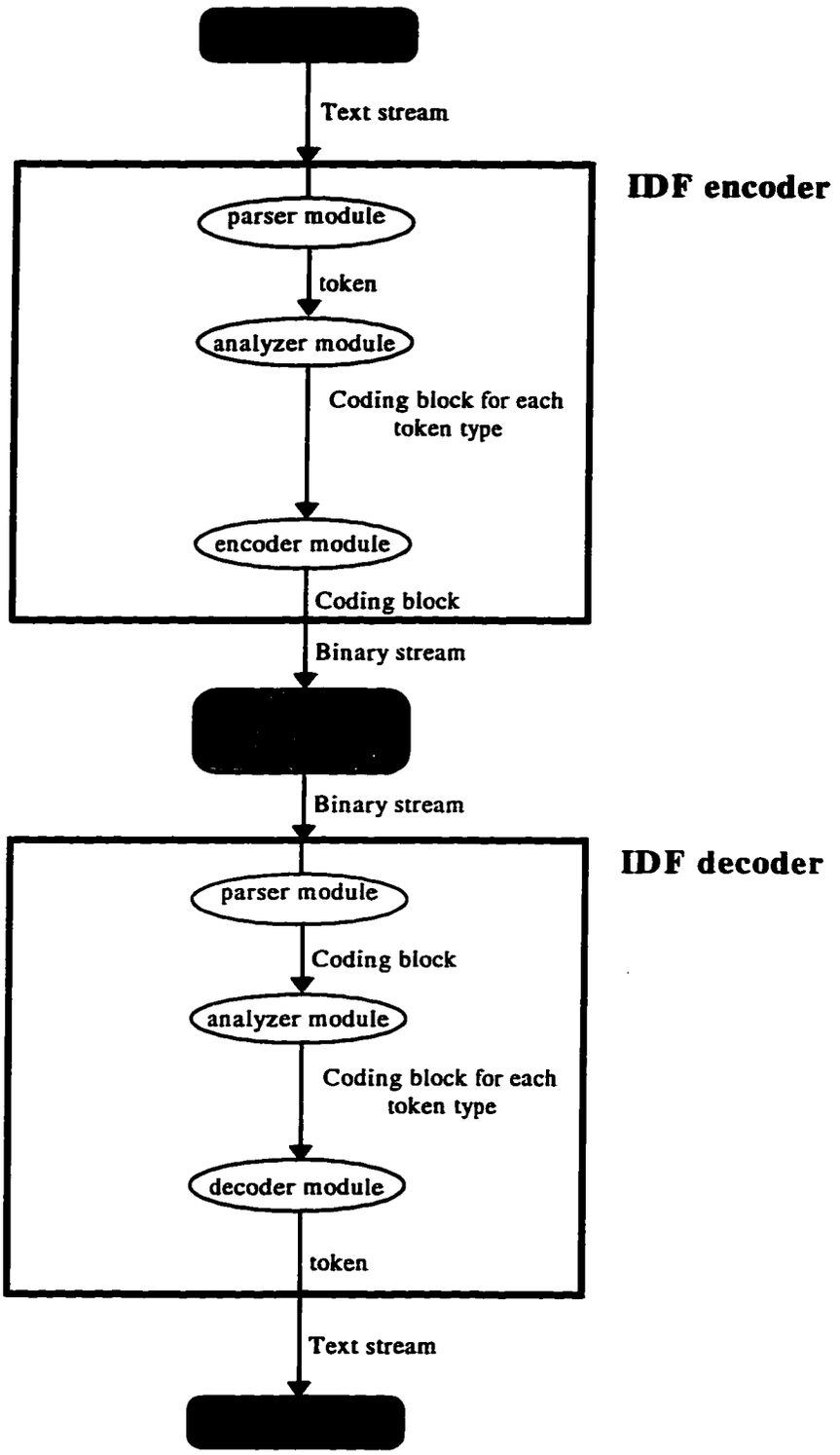

Figure 3-1. Overview of IDF encoding and decoding models.

3.2.4 Encoding standard words

The root form of a word is the base to which prefixes and suffixes are added. An inflected word is a word that has been changed from its root form to show a specific meaning or grammatical relationship to some other word or groups of words [3].

Verbs, nouns, and adjectives are the three major word types to undergo inflectional changes[11]. The way a word changes form determines its classification as regular or irregular. Examples of regular and irregular inflected words are given in Figure 3-2.

Regular Inflection	Word Type	Entry in dictionary
believe, believes, believed, believing	verb	believe
river, rivers	noun	river
happy, happier, happiest, unhappy	adjective	happy
sensitive, insensitive	adjective	sensitive

Irregular Inflection	Word Type	Entry in dictionary
write, writes, wrote, writing, written	verb	write, wrote, written
mother-in-law, mothers-in-law	noun	mother-in-law, mothers-in-law
child, children	noun	child, children
bad, worse, worst	adjective	bad, worse, worst

Figure 3-2. Examples of regular and irregular inflections.

In an IDF document the standard words are encoded in their *root forms* without any morphological modification. In general, those words that have regular inflected forms (e.g. by adding suffixes -s or -es to nouns, -s, -es, -ed and -ing to verbs, and -er and -est to adjectives and adverbs) will have the root forms encoded separately from the inflectional components. On the other hand, if the inflected form is irregular, each form will be encoded as a unique binary code.

[11] Inflections of adverbs do occur, however, we are not interested in them since they usually involve multiple words, e.g. helpful, more helpful, and most helpful.

Irregular inflected forms usually occur in the plurals of nouns, the principal parts of verbs, and the comparative and superlative forms of adjectives and adverbs.

Since a regular inflected word has the whole root or a fragment of it embedded in the word, we can use this behavior to our advantage in encoding regular inflections. Suppose that a document contains the word *believe* along with all of its inflected forms *believes*, *believed*, and *believing*. One approach to encoding these words is to treat them as if they were physically unrelated, and thus each is encoded as a unique binary codeword. This approach requires the creation of four entries in the coding dictionary. An alternative approach involves separate coding of the root and inflectional suffix of a word. Therefore, given a word with regular inflections, only one word and code value pair (the root and its corresponding code) is needed in the coding dictionary. In this approach, the encoding process contains two steps. First, the inflected word is mapped to the codeword associated with its root. Then the inflectional suffix and/or prefix of the word is extracted from the word and its corresponding binary code is determined. Next the codewords for the root and suffix are coded into the block.

The process of reducing an inflection to its root form is similar to *stemming*. The goal of stemming is to bridge all morphologically related terms by removing their prefixes or suffixes. For example, morphologically similar words like *excite, excites, excited, exciting, excitable,* and *excitement* can be stemmed to the string ***excit***. The stemming process allows all inflected variations[12] of the same word to be searched in a single pass since the resulting stem is either a whole or partial match of the root word and its inflections. The scheme we use works in a similar fashion. By encoding the root and the suffix as separate components, we can locate different forms of a word by doing a search on its root form.

---

[12] Obviously, this does not apply to irregular inflected words.

The encoding of irregular inflected forms is handled in a straightforward manner. Each form is associated with a unique binary codeword as if it does not have any relationship with its root. In the case of *write*, *writes*, *wrote*, *writing*, and *written* it requires three entries in the coding dictionary to cover all possible instances (one to represent each of {write, writes, writing} which use the regular inflectional endings, {wrote}, and {written}).

In addition derivative words are treated as regular inflected words, i.e. they will be encoded by their root forms. A derivative word is one derived from a lexical word. Antworth [4] explained that "*derivation produces new words (in the sense of lexemes)*" while "*inflections produces forms of the same word*".

Ro	Inflection	Derivation
large	larger, largest	enlarge
write	writes, writing, written	writer
compute	computes, computed, computing	computer, computerize
home	homes	homeless, homelessness

Figure 3-3. Examples of Inflections and Derivations.

3.2.5 Storing part of speech tags

Storing part of speech tags is an optional feature of the IDF encoding method. The purpose of this feature is to facilitate a linguistic processing known as syntactic analysis. Methods for text analysis associated with syntactic structures and full sentence grammars are useful to activities involving extraction of information from documents [5, 6]. Syntactic analysis can also be used to analyse text in order to determine the boundaries of noun phrases which could then be used as internal representations. The IOTA system [7] is an example of indexing text on a noun phrase using NLP techniques. Other text analysis involve identifying content-carrying terms and/or inter-term dependencies. This can be

accomplished by using a part of speech tagger to parse and tag the document so that word-level analysis can be applied to study relationship between terms.

Consider the following paragraph before and after it was syntactically tagged.

Untagged text

It has been nearly six years since Tim Bajari and I agreed to launch a newsletter on multimedia computing, a newsletter that eventually became the journal you're now reading.

Tagged text (using Brill's tagger [5])

It/PRP has/VBZ been/VBN nearly/RB six/CD years/NNS since/IN Tim/NNP Bajari/NNP and/CC I/PRP agreed/VBD to/TO launch/VB a/DT newsletter/NN on/IN multimedia/NNS computing/VBG ,/, a/DT newsletter/NN that/IN eventually/RB became/VBD the/DT journal/NN you/PRP 're/VBP now/RB reading/NN ./.

(Appendix B shows a listing of the meaning of the tags used.)

If the words in the input file have been tagged by a part of speech tagger, the IDF method will encode a word and its associated part of speech together as a unit. This encoding scheme makes it easy to answer questions such as "*list all part of speech that the word 'computing' has been tagged, or show all inflective and derivative forms of 'compute' that has a verb tag*". To derive at an answer to the first question involves checking all standard word blocks that pertain to the word *compute*, and then further isolating those blocks whose inflectional tag has been set to 'ing' and then retrieving the associated part of speech tag.

3.2.5.1 Storing word senses

Another optional feature of the IDF encoding method is the storing of word sense information together with the word. A document may contain words

with multiple senses (polysemous words), and in general, no effort is made to distinguish the senses of the words. For example, the word *crane* can refer to one of the several senses,

a) a bird in the heron family (noun)

b) a machine that raises or lift heavy objects (noun)

c) to raise or to lift something (verb)

d) to stretch the neck toward an object of attention to get a better view (verb)

It can be seen from the example that *crane* has meanings in the noun senses: "a bird" or "a hoist machine"; and verb senses: "to raise" or "to stretch the neck". And even the noun sense, e.g. a bird or a machine, is ambiguous. Thus, by attaching word senses to polysemous words and storing both the sense and word together makes it feasible to locate a words with a particular sense. This particular feature of the IDF encoding method assists the understanding of the context of a word being used.

3.3 IDF Specification

Definitions

A ***codeword*** is an unsigned integer representing a bit pattern that is assigned to a text string. Encoding a string means replacing the string with the codeword.

A ***coding dictionary*** is a data structure which stores the codewords and their associated text strings.

Every IDF file has two parts:

a) A header section which contains the coding dictionaries of the codewords that appeared in the IDF file, and

b) a body section which contains encoded data.

The body of an IDF file is encoded as a sequence of *blocks*. A block is normally either one byte or three bytes long, but it can be four bytes long (to be explained.) Each token in the document is encoded as a bit pattern stored in a block. In essence, a three-byte block can be viewed as a sequence of bits whose function is to store the four codewords for

a) a word or an exception token,
b) a word's inflectional affix and uppercase, lowercase attributes,
c) a part of speech tag, and
d) a word sense tag.

The codewords for part of speech and word sense tags are stored in an optional fourth byte.

The number of bytes in a block depends on the type of token to be encoded. Standard words and special words are encoded in three byte blocks while stop words are encoded in one byte blocks. For exception tokens, coded ones are encoded in one byte blocks while the uncoded ones are stored "raw" (as a sequence of ASCII codes) delimited on either end by special markers.

The two highest order bits of a block are the *identifier bits* which encode the block's length and purpose (see Figure 3-4). The possible values for identifier bits are given in Table 3-1.

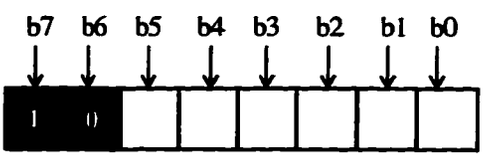

Figure 3-4. Block identifier bits (shaded) of a one byte block.

Purpose	# Bytes	Block Id
exception token		
uncoded token	N/A	N/A
coded token	1	11
hi-frequency word	1	00
standard word	3	01
special word	3	10

Table 3-1. Values of block id bits for word and exception tokens[13]

3.3.1 One-byte block encoding scheme

High-frequency word tokens and *coded* exception tokens are encoded in one byte blocks. Since the block identifier already takes up the two highest bits, the remaining lower six bits are available for storing encoding data and hence limiting the block's coding capacity to sixty four (2^6) entries. Similarly, the exception block can store sixty four distinct binary codes. The distribution of codes used in the exception block are as follows: thirty two codes are reserved for the encoding of all punctuation marks found on a standard US keyboard. From the other thirty two yet unused values, two *special* codes are reserved for marking the beginning and ending of a sequence of characters in the document that will not be encoded, and the remaining thirty may be used for the encoding of non-printable characters.

In the text file being encoded any exception token that does not fit the coding scheme for the exception block described above is referred to as an *uncoded* exception token. That is, it is not a high-frequency word, not a special word, not a standard word, and not a coded exception. Tokens of this type will be stored in the IDF file "as is", which means that the sequence of ASCII characters stored must be delimited by the special markers. Suppose that the

[13] Note that uncoded tokens have no block.

beginning and ending special markers are **100000** and **010000**, then the *uncoded* exception token "130km/hr" will be encoded as shown in Figure 3-5.

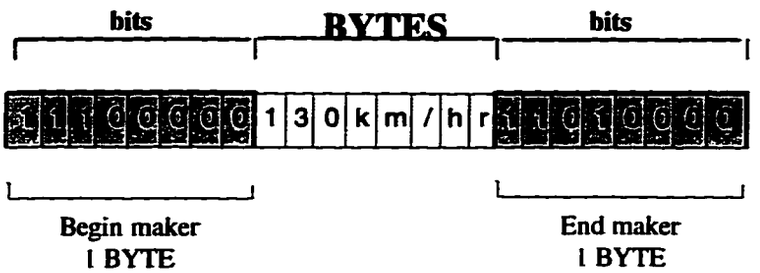

Figure 3-5. *Begin* and *End* codewords marking boundaries of unencoded string.

3.3.2 Three-byte block encoding scheme

If the text file to be encoded has not been tagged by syntactic tagger then the standard and special words in the file are encoded in three byte blocks, otherwise they are encoded in four byte blocks (Section 3.3.3). A three byte block is sub-divided into two logical fields: Info and Data. Info field is the first byte and Data field comprises the last two bytes of the block (see Figure 3-6).

In the Info field, the two highest-order bits are allocated for block id; the next four bits are used to encode the inflectional information, and the last two bits store the codeword for the word's case attribute.

In the IDF scheme we have decided to store prefixes and suffixes only for verbs, nouns, and adjectives. The list of encoded prefixes are those commonly found "dis", "en", "in", "re" and "un", and the suffixes encoded are "s", "es", "ies", "ed", "ing", "er", "est", "ion", "ment", and "ness". Considering that we have four bits to code the affixes, it is possible to encode a maximum of sixteen different inflections. Presently, we have used up all sixteen codewords since there are five prefixes and ten suffixes to be encoded, and an extra codeword is needed to represent the case where the word is already a root.

For any given word in the document, its case attribute belongs to exactly one of the following four categories: lowercase, uppercase, capitalized, or contains mixed cases. With only two bits to encode the word's case attribute, it is not possible to have a codeword which corresponds to the case where the word contains mixed cases. The reason for this is that the value of the two-bit encoding sequence only indicates that the word contained mixed cases but this information is not useful in the decoding process. Therefore a word that contain mixed cases such as InitializeDictionary (name of function in a program) has to be encoded as an exception token.

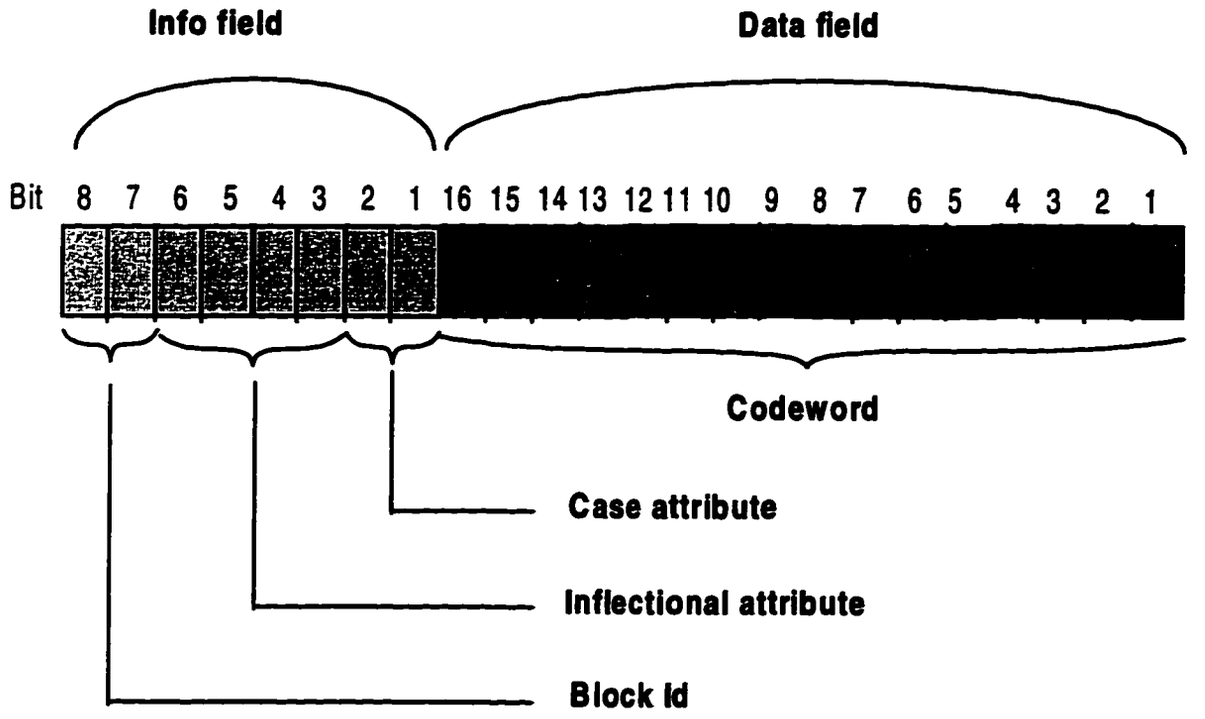

Figure 3-6. Three byte block encoding scheme.

The Data field (the last two bytes of the block) has all sixteen bits available for the storing of codewords for the tokens. Since there are two types

of three-byte blocks it means that a document can have at most 65,536 distinct standard words and 65,536 distinct special words. It is assumed that no single document would need more than 65,536 + 65,536 different entries in the lookup tables. Many of the standard words (verbs, nouns, and adjectives) do undergo morphologic modification, i.e. they can be mapped to the same codeword if they have the same root form. Therefore, it is possible to actually have more than 65,536 distinct standard words in a document. Similarly, there is not a one-to-one correspondence between the number of entries in a dictionary of special words and the actual number of words encoded. While most words in the special word dictionary are single words, there can also be frequent compounds that are made up of two or more words. These compound terms are to be encoded as a unit.

3.3.3 Four-byte encoding scheme

This scheme is used when the file to be encoded contains part of speech tags and/or word sense tags. A four byte block is created by inserting a byte between the Info and Data fields of the three byte block. The addition of the new byte results in a new field called Tag field. In Tag field (the second byte) the distribution of the bits are as follows: the higher two bits store a codeword for word sense and the lower six bits store the word's part of speech tag. Therefore, the above distribution assumes the following: a word in any document has no more than four homonyms, and the linguistic tagger used to tag the document does not use more than sixty four different tags. If the encoding capacity of any subfield is reached then the overflow will be encoded as raw. Although the number of tags used by tagger systems varies from one system to another thirty two seems to be sufficient. For example, the Penn Treebank Corpus [8] uses thirty six part of speech tags for terminals (words and tokens)

and twenty for non-terminals (clauses), whereas the Old English Corpus uses only nineteen part of speech tags [9].

3.4 Summary

In this chapter we have described the specification for the Intelligent Document Format. The IDF encoding method uses the four dictionaries: high-frequency, standard and special words, and exception to encode word tokens. Any text tokens excluded from the four dictionaries are stored as raw ASCII sequence. In the normal mode of IDF encoding, standard and special words have their codewords stored in three byte blocks while high-frequency and certain exception tokens are encoded using one byte blocks. If the input text file has been parsed with a syntactic parser, contains part of speech tags, and/or has been marked word sense tags then the coding scheme for standard and special words changes to four byte blocks.

3.5 References

[1] Fox, C., Lexical Analysis and Stoplists, *Information Retrieval: Data Structures & Algorithms*, W. B. Frakes and R. Baeza-Yates eds., 113-117, 1992.

[2] Francis, W. N. and H. Kucera. *Frequency Analysis of English Usage, Lexicon and Grammar*. Houghton Mifflin Co., Boston, 1982.

[3] Hodges, J. C., M. E. Whitten, J. Brown, and J. Flick, *Harbrace College Handbook for Canadian Writers*, Third ed., Harcourt Brace Jovanovich, 183-187, 1990.

[4] Antworth, E. L., *PC-KIMMO: a two-level processor for morphological analysis*. Summer Institute of Linguistics. 1990.

[5] Lehnert, W. G. and B. Sundheim, A Performance Evaluation of Text Analysis Technologies, *AI Magazine*, 81-94, 1991.

[6] Cardie, C., A Case-Based Approach to Knowledge Acquisition for Domain-Specific Sentence Analysis, *Proceeding of the Eleventh National Conference on Artificial Intelligence*, MIT Press, 798-803, 1993.

[7] Evans, D. A., Concept managment in text via natural language processing: the CLARIT approach, *Working Notes for the AAAI Spring Symposium on Text-Based Intelligent Systems*, Stanford, 1990.

[8] Marcus, M., B. Santorini, and M. Marcinkiewicz, Building a large annotated corpus of English: the Penn Treebank, *Computational Linguistics*, 1993.

[9] Brill, E. *A Corpus-based Approach to Language Learning.* Ph.D. Dissertation, Department of Computer and Information science, University of Pennsylvania, 1993.

4. Implementation

IDFconvert is an implementation of an IDF encoder and decoder. This chapter describes its functional specification, algorithms and structure. In addition, two utility programs that accompany IDFconvert will be discussed. These utility programs provide certain auxiliary support such as preparing the file that is the input to IDFconvert and assisting the user in generating a flat file containing the *special words* with respect to the document encoded.

IDFconvert and its ancillary utilities were developed on a UNIX system (Sun OS 4.1.3.2). Written in C, IDFconvert is made up of the six modules: token parser, codeword dictionary, PC-KIMMO (an interface to a morphological database), encoder, decoder, and bit manipulation. These modules together provide the essential services needed by the encoding and decoding operations. On the other hand, the utility programs were written in Perl [1], a scripting language that offers easy text string manipulations.

4.1 Functional Specification

Program Name:	IDFconvert
Functions:	Encodes an ASCII text file according to the IDF encoding specification.
	Decodes an IDF encoded file to ASCII text.
Resources:	Use dictionary files: high-frequency.dict and special.dict.
Input:	The input is an ASCII file containing English text. The file must have been preprocessed by the Tokenizer program (to be described in section 4.4.) In each line of text, word and punctuation tokens are separated from each other by at least a white space. A line cannot be longer than 1024 bytes (characters). Any line whose size is greater than 1024 bytes will be not be processed. The file is assumed to contain a maximum of 65535 distinct standard words and 65535 special words. If a dictionary of a given type is full (64K) then any extraneous word of this type will be stored as an ASCII string.
Output:	The output is a binary file containing IDF encoded data. The file begins with the dictionary header section. The dictionary header section consists of the four dictionaries in the following order: hi-frequency word, exception token, special and standard dictionaries. A dictionary is written to the IDF file in the following manner. The first two bytes is an unsigned integer which corresponds to the number of entries in the dictionary. Immediately following it are pairs of *number* and *word*, i.e. the dictionary entries. The number is a two byte unsigned integer and the word is a series of ASCII characters that terminates with a new-line character. To mark the boundary of a dictionary two one byte markers (opening and closing special markers) are placed at the end of each dictionary. Following the pair of opening and closing markers of the last dictionary is the beginning of the IDF encoded data.

4.2 Algorithms

The inner workings of IDFconvert are fairly straightforward and can be described by a *top down* analysis. The top down approach will first analyze the sequence of steps that take place in the encoding or decoding process, and then identify the required services. As an example, when a line of text is input from a file, there must be a function to deal with each of the series of events: segment the input line into tokens, obtain a token and classify its status, determine the units of information to be encoded, and carry out the actual encoding actions.

4.2.1 The encoding algorithm

The algorithm for the encoding of a text file can be divided into two successive phases. The first phase is the initialization of the data structures of the codeword dictionaries (lookup tables.) Once all the dictionaries have been initialized the second phase begins. It is a cycle which contains five steps related to the encoding process (see list below). The procedure is repeated with a new line of input, beginning a new cycle, until the end of the input file is reached. .

1) a line of text is read from the input file.
2) a token retriever extracts tokens from the line one at a time.
3) A token is converted to lowercase and passed on to the dictionary module where the token's type is first determined,
4) a codeword is generated to create a key-value pair for the token and its codeword (if necessary) in the lookup table corresponding to the token's type.
5) the codeword is passed to the encoder module where it is stored in an array.

4.2.1.1 Encoding exception token in special word block

Back in section 3.3 (IDF specification), we stated that the token type dictates how a token is encoded. Particularly, we said that exception tokens, (if they are not found in either the high-frequency word, special word, or standard

word dictionary) are to be stored as "raw" ASCII strings. This assertion assumes that the special word dictionary supplied by the user is a comprehensive one. However, in the development of IDFconvert and experimenting with encoding test files (Chapter 5) we have found that it is sometimes not practical to generate a comprehensive dictionary because the task requires just too much of the user's time and effort. This experience shaped our final implementation of IDFconvert. In the final version of IDFconvert, the user has the option to tell the program (via a command line argument) to encode any exception token as if it were a special word. That is, once the encoder has determined that there is no corresponding entry for a token in any of the three word dictionaries, i.e. the token has been classified as an exception, it can encode the token as a special word. The default behavior of IDFconvert, however, is to encode exception tokens in the IDF file as ASCII characters. Figure 4-1 shows the pseudocode for the encoding algorithm.

4.2.2 Time complexity of encoding algorithm

We now discuss the worst and average time complexities of the encoding algorithm of the program IDFconvert. A condensed summary of the encoding algorithm is given below,

```
1.    while (lineCount > 0) {
2.      getLine(Line);
3.      getFirstToken(Line,token);
4.      while (token!=NULL) {
5.        setCaseFlag(token,caseFlag);
6.        codeword = inLookupTable(token);
7.        block = encodeToken(token, codeword);
8.        getNextToken(Line, token);
9.        storeBlock(block);
10.     };
11.     lineCount--;
12.   }
```

```
/* ************************************** */
/* pseudocode of IDF encoding implementation */
/* ************************************** */
for ( ; ; ) {                                  /* loop through all lines */
  if (readFromFile(Line)!= NULL) {             /* read a line until EOF  */
    token = getFirstToken(Line);               /* parser extracts token  */
    while (token != NULL) {                    /* process until no more  */
                                               /* token in input line    */
      caseFlag = caseFoldedString(token);      /* set caseFlag of token  */

      /* ********************************************************** */
      /* Determine token is in which lookup table & map to codeword */
      /* ********************************************************** */
      if ((codeword = inLookupTable(mscWLT, token)) >= 0) {
        block = encodeToken(codeword, 3, NONE, caseFlag);
      }
      else if ((codeword=inLookupTable(hifreqWLT, token))>= 0) {
        block = encodeToken(codeword, 0, NONE, caseFlag);
      }
      else if ((codeword = inLookupTable(spcWLT, token)) >= 0) {
        block = encodeToken(codeword, 1, NONE, caseFlag);
      }
      else if ((posInfoStr=accessMorphologicalDB(token)!=NULL)) {
        geRootForm(posInfoStr, token, rootForm);
        getInflection(posInfoStr, token, inflection);
        codeword = insertLookupTable(stdWLT, rootForm);
        block = encodeToken(codeword, 2, inflection, caseFlag);
      }
      else {
        if (encodeNonwordAsSpecialWord) {/*exception token in spc blk*/
          codeword = insertLookupTable(spcWLT, token);
        } else {
          encodeAsIsFlag = 1;                  /* set flag - token is to */
        }                                      /* be stored as ASCII seq */
      }

      if (!encodeAsIsFlag) {                   /* if !encodeAsIsFlag     */
        storeBlockInSentenceArray(block);      /* stores block in array  */
      } else {                                 /* otherwise,             */
        storeTokenAsIsInSentenceArray(token); /* stores token as a      */
      }                                        /* sequence of ASCII codes*/
      token = getNextToken();
    }
  } else {                                     /* EOF character was read */
    break;                                     /* exit loop              */
  }
}                                              /* finishes encoding      */
```

Figure 4-1. Pseudocode of IDF encoding.

We choose the statement

```
block = encodeToken(token, codeword)        [ line 7]
```

to be the *characteristic operation* since the algorithm perfoms this statement repeatedly for all words that get encoded. We define a function f(n) to be the number of times the function encodeToken is invoked in processing an input file of size n tokens. Also, suppose that *m* is the actually number of tokens that will be encoded, and let p(m,n) be the probability that *m* tokens out of *n* tokens are encoded. Then, f(n) = p(m,n) * n.

In the worst-case situation, all of the tokens in the input file are encoded, that is p(n,n) = 1, then f(n) = n and thus f(n) has the average time compexity of O(n).

In the average-case situation, most[14] of the tokens in the input file are encoded then we can treat p(m,n) like a small constant[15]. and so f(n) also has the average time complexity of O(n).

4.2.3 The decoding algorithm

The series of steps in the decoding algorithm is very similar to those in the encoding counterpart, except that they work in the reverse direction. There are two phases to this algorithm. In the first phase the codeword lookup tables are rebuilt from the dictionary header section of the IDF file. Once the lookup tables have been initialized with values, the second phase, which concerns with the decoding of data, begins. The second phase has five major steps.

First, a sequence of bytes is read from the IDF file into a buffer, and while there are undecoded data remaining in the buffer, the decoding steps will occur.

---

[14] We can make this assumption because if a lot of the tokens are not encoded then there is no point to convert a text file to the IDF file since there is little gains in doing so.
[15] We assume that the value p(m,n) do not vary much for texts of the same genre.

In the second step, the first byte of an undecoded block in the buffer indicated by the *nextBlock* index is examined for the block id value.

The next step involves examining the block's id, determining the block's type and length, and subsequently extracting its content (codeword). At this point there are five possible types of codeword. The codeword might represent

1. a high-frequency word,
2. a punctuation mark,
3. an *opening* special marker to denote that the sequence of bytes following it are ASCII codes (the sequence ends whenever a *closing* special marker is encountered),
4. a standard word, or
5. a special word.

In step four, the codeword is checked in the appropriate lookup table to obtain the decoded string. If the codeword represents an opening special marker, then the bytes ahead are to be extracted on a character by character basis until the closing special marker is encountered. After this "raw" ASCII sequence has been extracted from the buffer, the *nextBlock* index is updated to an appropriate value. Figure 4-2 gives the algorithm for extracting an ASCII string.

The last step of the second phase pertains to the "cosmetic" mending to the decoded string if it is a standard word or special word, which additional encoded items such the word's case attribute and inflectional information may need to be retrieved and applied to the decoded word.

Other undecoded blocks are decoded in the same way, and the procedure is repeated until all blocks in the buffer array have been decoded. Once all blocks in the buffer have been processed, the next collection of data is read into the buffer and the process repeats once more. Figure 4-3 shows the pseudocode for the decoding algorithm.

```
if (codeword == openningSpecialMarker) {
   done = 0;
   i  = 0;
   repeat {
      blockIndex++;
      ASCIIchar = buffer[ blockIndex] ;
      if (ASCIIchar == ClosingSpecialMarker)
         done++;
      else
         ASCIIstring[ i++]  = ASCIIchar;
   } until done;
}
```

Figure 4-2. Extracting "raw" ASCII sequence.

```
/* ************************************** */
/* pseudocode of IDF decoding implementation */
/* ************************************** */
for ( ; ; ) {                                   /* loop through IDF file */
  u_char shortBlock = byte = 0;                 /* initialize some temp  */
  u_int longBlock = 0;                          /* variables             */

  if (byteCount=(readIntoArray(fp,bufer,BUFSIZE))!= 0) {
    blockIndex = 0;                             /* set index of 1st avail*/
                                                /* encoded block */
    while (byteCount > 0) {
      shortByte = buffer[ blockIndex] ;         /* get 1st byte of block */
      blockID = valBlockId(shortByte);          /* and examine block id  */
                                                /* which gives blk length*/
      switch(blockID) {
        case hifreqByteID:
        case miscByteID:
          shortBlock = byte;
          break;
        case spcByteID:
        case stdByteID:
          getBlock(longBlock,blockID,buffer);/* get block content    */
          break;
      }
      codeword = extractLTValue(shortBlock, longBlock, blockID);
      if (blockID == hifreqByteID) {
        token = lookupTable(dictList[ hifreqByteID] , codeword);
      } else if (blockID == miscByteID) {
        if (codeword == openningSpecialMarker) {
          char *str;
          getASCIIString(str, blockIndex, buffer);
          break;                                /* jump to next iteration*/
        } else {
          token = lookupTable(dictList[ miscByteID] , codeword);
        }
      } else {
          if (blockID == spcByteID) {
            token = lookupTable(dictList[ spcByteID] , codeword);
          } else if (blockID == stdByteID) {
            token = lookupTable(dictList[ stdByteID] , codeword);
          }
          valInflectionId(blockValue, blockID, inflectFlag);
          valWordCaseId(blockValue, blockID, caseFlag);
          mapRoot2Word(token, inflectFlag, caseFlag);
      }
      updateCounter(blockID, blockIndex, byteCount);
      strcat(Line, token);
    }
  } else {                                      /* no more data to decode */
    break;
  }
}                                               /* finishes decoding */
```

Figure 4-3. **Pseudocode of IDF decoding algorithm.**

4.3 PC-KIMMO: a morphological database

PC-KIMMO[16] is a two-level morphological database developed by Antworth [2]. According to Karp and co-workers [3], PC-KIMMO can be characterized as follows: "*Using the morphological rules for English inflections ... and our lexicons, PC-KIMMO outputs all possible analyses of each input word, giving its root form and its inflectional attributes.*" The *two-level* attribute of the parser refers to the model of morphology for word form recognition and production [4, 6]. The model asserts that a word can be viewed from two levels; the *morphotactics*, which enumerates the inventory of morphemes[17] and specifies in what order they can occur, and the *morphophonemics*, which deals with the way morphemes are "spelled" according to the phonological context in which they occur [2]. For example, the two inflected words **chased** and **played** are morphotactically parsed as the stems **chase** and **play**, each is followed by the suffix -ed. However, it should be noted that in the case of **chase**, the grafting of -ed causes the loss of the final **e**. Hence, from the morphophonemic point of view "chase" and "chas" are alternate forms of the same morpheme, while "play" is the only morpheme that happens to be identical to its root.

From a developer's vantage point, the PC-KIMMO parser provides just the kind of services required by the IDF encoder. To encode a word the encoder needs to know two things; whether it is a standard word, and what is the corresponding root form if it turns out to be a standard word. These two items can be retrieved from the morphological database (or lexicon) via the services provided by PC-KIMMO. Thus PC-KIMMO defines the standard words. PC-KIMMO's lexicon, derived from the 1979 edition of the *Collins Dictionary of the*

[16] The UNIX version of PC-KIMMO is implemented as a package of C library functions interfacing an English morphological database.

English Language, contains over 90 000 root forms, excluding proper nouns, and this results in more than 317 000 inflected forms [3]. Table 4-1 shows a summary of the size of the lexicon.

Category	# Root Forms	# Inflected Forms
Pronoun	92	93
Preposition	148	150
Determiner	100	100
Conjunction	64	64
Adverb	6992	7176
Noun	50370	199303
Adjective	20550	65146
Verb	11880	45445
Total	90196	317000

Table 4-1. Summary of size of PC-KIMMO lexicon.

When a word is input to PC-KIMMO, its parser looks up the word's entry in the lexicon. If no such entry is found the parser returns an empty string which allows one to conclude that the word is not a standard word. If the lexicon contains an entry for the input word, then the parser retrieves the information about the entry. The string returned by PC-KIMMO is a sequence of tagged morphemes which always contains the root form of the input word along with the root's part of speech tag. This is the mechanism employed to obtain the root of a regular inflected standard word. Table 4-2 shows sample output strings by the parser for various input words.

---

[17] Webster dictionary defines *morpheme* as a meaningful linguistic unit whether a free form (as *walk*) or a bound form (as the -s of *walks*) that contains no other meaningful parts.

Input	Output String
talk	talk N sg#talk V INF
talks	talk V 3SG PRES
person	person N GEN sg
writing	writing N sg#write V PROG
walkin	walking N sg#walking A#walk V PROG
funky	funky A
funkier	funky COMP A
funkie	funky SUPER A
disapp	dis+PREFIX appear V#disappear V
unhap	un+PREFIX happy A#unhappy A

Table 4-2. Sample outputs by PC-KIMMO parser.

4.3.1 Forward mapping: inflection to root form

The searching for the root from the PC-KIMMO's output that corresponds to the input word involves analyzing the output string. The output string is scanned for the word that is most likely be the root. This involves the examination of every pair of word and associated part of speech tag. Since an irregular inflected word is encoded as a separate entry from its root, the candidate word in the output string must be compared against the input word to check for the irregular case. It is true that in English there is a great similarity between an inflected form and its root form. In many cases the root itself or a large fragment of it is contained within the inflected form. Thus, by comparing the candidate word against the input, it can be determined whether the input word is regularly or irregularly inflected. Based upon empirical observations, we have come up with some guidelines for discovering a potential root word from the PC-KIMMO's output string. The series of rules listed below, in order of descending precedence[18], can be used to determine the root word.

[18] These rules are empirically derived as a result of observing texts that have been syntactically tagged by Brill's tagger.

If there is a word W_i whose tag T_i = **V INF**
  then W_i is a potential root
else if there is a word W_i whose tag T_i = **V 3SG PRES**
  then W_i is a potential root
else if there is a word W_i whose tag T_i begins with **V**
  then W_i is a potential root
else if there is a word W_i whose tag T_i = **N sg**
  then W_i is a potential root
else if there is a word W_i whose tag T_i = **A**
  then W_i is a potential root
else if there is a word W_i whose tag T_i begins with **N**
  then W_i is a potential root
else
  W_o is an irregular inflection

According to the rules above, the way to single out the potential root word from the other candidates is to scan the output string for the presence of a part-of-speech tag specified by the highest rule. If the search is successful then the word Wi associated with this tag is the potential root. When this happens W_i is stemmed, and the resulting token is matched against the input word W_o to see if it is a substring of W_o. If the token is a substring of the input word, then the root has been found and it is returned to the encoder; otherwise, it's likely that the input word has an irregular inflection, the search for the root is stopped, and the input word is returned to the encoder.

On the other hand, if the tag search fails to satisfy the current executing rule, then the next rule in the series is taken into consideration and the procedure mentioned above is applied. In the event that all rules have been applied and still no root word was confirmed, it may be concluded that the input word may be an irregular inflection.

4.3.2 Reverse mapping: root to inflection form

Among the codewords embedded in a three-byte encoded block only the two bytes corresponding to the word and inflectional ending are needed in the reverse transformation. It is where the root is changed back to the inflected word originally appearing in the ASCII text. The approach for reconstructing the inflected word is accomplished by using a set of rules put forth by Hodges and others [7] for adding suffixes[19] to roots. These rules specify instructions in a similar context to that given by the set of morphological rules for English described by Karttunen and Wittenburg [5]. PC-KIMMO adopted Karttunen and Wittenburg's morphological rules to deal with the following phenomena: epenthesis[20], y to i correspondences, i to y correspondences, s-deletion, elision[21], gemination[22], and hyphenation [3].

RULE FOR ADDING PREFIXES

1. **Attach the prefix to the root.**

When the prefix is added to the root neither component undergoes any changes.

agree	disagree
happy	unhappy
write	rewrite
relevant	irrelevant
fortune	misfortune

[19] The reason that there are no rules for adding prefixes onto roots because there is no change to be made when this is done, e.g. *unnecessary*, *unhappy*, *disagree*, *repolish*.
[20] Webster dictionary defines *epenthesis* as the insertion or development of a sound or letter in the body of a word.
[21] Elision refers to the omission or deletion of a pattern in a word.
[22] Gemination is defined as to become double or paired.

RULES FOR ADDING SUFFIXES

1. **Remove the *-e* before a suffix beginning with a vowel.**

   engage engaged

   engage engaging

2. **Keep the *-e* before a suffix beginning with a consonant.**

   manage management

   care careful

   - Examples of some exceptions: argument, agreeable, courageous.

3. **Double a final consonant before a suffix beginning with a vowel if both a) the consonant ends a stressed syllable or a one-syllable word , and b) the consonant is preceded by a single vowel.**

   jog jogged

   plan planning

   admit admitted

   conceal concealling

4. **Change the *-y* to *i* before suffixes, except *-ing*.**

   apply applies, applied

   happy happier, happiest, happily

   - Examples of some exceptions: applying, studying

5. **Verbs ending in *y* preceded by a vowel do not change the *y* before *-s* or *-ed*.**

   stay stays, stayed

   dismay dismays, dismayed

6. **Keep the last *l* before a suffix *-ly*.**

casual casually

formal formally

7. **Add *-s* or *-es* to the singular of nouns.**

table tables

book books

- Examples of some exceptions: woman, women, analysis, analyses

8. **Add *-es* to singular nouns ending in *s, ch, sh,* or *x*.**

dress dresses

speech speeches

wash washes

box boxes

4.3.3 Handling exceptional cases

As shown in the above examples there are exceptions to a given rule. The derivation of a set of procedures to handle all exceptional cases is not an easy task. This task requires the expertise of a linguist and is very well beyond the scope of this thesis. However, there is a simple alternate solution that circumvents this problem, and it works as follows. In the encoding phase, after consulting with PC-KIMMO, an inflected word is decomposed into two parts, the root and inflectional ending. The root then is checked against those patterns (e.g. root ends in *-y*) that are known to cause exceptions. If such a match was

found then the parts are put together to test if the original word can still be reconstructed exactly from the parts. If the reconstruction is successful then the encoder proceeds with the encoding of the root and suffix components into a standard word block. Otherwise, the word is treated as the case of an irregular inflection, that is, a new standard dictionary entry is created to store this word.

4.4 Utility Programs

Tokenizer and BuildTermList are the two Perl utility programs. These programs operate on the input ASCII text of the IDFconvert program. The functionality of these utilities is discussed below.

4.4.1 Tokenizer

The program Tokenizer processes the source file one paragraph[23] at a time and it does two things 1) dehyphenates any line terminating with a hyphenated word and 2) inserts a space character between word and punctuation marks to separate them.

The reason for all this preprocessing to the input file is that it is difficult to implement a token segmenter routine in IDFconvert that can extract "completely free"[24] words. Thus, in order for IDFconvert to process a line of text, the elements or tokens of the line must already be isolated. An alternate solution to developing an internal token segmenter is to implement 1) a preprocessor such as Tokenizer to produce tokenized text, and 2) a build-in token retriever routine in IDFconvert. In having the input text tokenized the token retriever's job has been simplified a great deal. As a line of text is read

[23] A paragraph is a section of text which begins and ends with an empty line.
[24] A completely free word has no punctuation attached to it and is one that has not been "hyphenated" due to the effect of text wrapping at the end of a line.

tokens are retrieved one at a time by copying a sequence of characters until a white space is encountered.

4.4.1.1 Dehyphenation

The first step of the program Tokenizer deals with the dehyphenation of the input file. For each paragraph processed the program searches for any line that terminates with a hyphenated word, and removes the two characters (hyphen and new-line) occurring at the end of the line. The outcomes of this action are twofold. One, the word gets dehyphenated, and two, the adjacent lines are joined together to form a single line. Thus the paragraph is shortened by one line. Then a new-line character is then inserted at the end of the recently dehyphenated word unless it already ends with a new-line. This splits the line into two again and the number of lines in the paragraph remains the same. However, if the dehyphenated word already terminates with new-line then the insertion step will not happen, and the paragraph will be one line shorter.

It should be stressed that dehyphenation applies to only hyphenated words occurring at the end of a line. Other hyphenated words appearing elsewhere in a line are left unmodified. For example, in the previous paragraph, all occurrences of the word "new-line" do not get dehyphenated since they never occurred at the end of a line. It should also be noted that there are situations in which the removal of the hyphen in a word produces a *damaged* word. A damaged word is usually resulted from the dehyphenation of a *hyphenated compound.* Antworth [2] and Quirk *et al.* [8] viewed that compounds in English can be summarized as follows:

- solid compounds, e.g. bedroom
- hyphenated compounds, e.g. state-of-the-art
- open compounds, e.g. information retrieval

Whenever a hyphenated compound is broken off at the end of a line (due to text wrapping) the break always comes immediately after the hyphen. Therefore, if the procedure described above is applied it will produce an erroneous word. Consider the following example. In a given line it is observed that the hyphenated compound "state-of-the-art" is broken into two parts, with "state-" terminates one line and "of-the-art" begins the next. When dehyphenation is applied to these two lines, the word produced would be "stateof-the-art". Clearly, this behavior is neither desirable nor acceptable. Unfortunately, due to the nature of automatic dehyphenation this type of error can not be avoided. There is only one way to handle the problem and that is having the user manually correct the errors. In order to assist the user to detect damaged words introduced into the document by Tokenizer, the program outputs a list of all the words it dehyphenated. The user needs only to inspect this list for the erroneous words and proceeds to make the necessary corrections.

4.4.1.2 Inserting a space character

The second step of the program involves the tokenization of words. A blank space is inserted before and after every punctuation mark found in the paragraph text, and thereby tokenize the words (i.e., establish a clear separation between the words and the punctuation marks.) All this is done while special care is taken to leave alone exceptions that involves punctuation marks embedded in a string of numbers. For example, text such as "$560.50", "7:30", and "9/23/1996" will remain intact instead of becoming "$ 560 . 50", "7 : 30", and "9 / 23 / 1996". Figures 4-4 and 4-5 show an excerpt from *Dracula* [9] before and after it was processed by Tokenizer.

4.4.2 BuildTermList

The program BuildTermList takes the input file to IDFconvert as an argument, and produces a list of special words (or terms) which the user deemed to be important. The output of the program is an ASCII file containing one term entry per line. A term entry can be a single word or multiple words.

The program first generates a list containing all the words in the input file. It then uses a list of 425 high-frequency words to filter out the most frequently occurring words in general text (see Appendix A for a complete listing). For each word in the remaining list, the program request the user to confirm whether the word is a single-word term, a part of a multi-word term, or not a term. If the user indicates that the word is not a term then the next word in the list is processed. If the user indicates that the word is a single-word term then it is immediately written to the output file. If the user thinks that the word is a part of a multi-word term, it is put on a multi-word word list for subsequent processing. Once the status of all the entries in the current list has been verified, the list containing candidate multi-word terms is processed.

To make it easier to identify multi-word terms in the document, BuildTermList begins to search for bi-grams (two word terms). The program takes an entry in the multi-word term list and begins to scan the file to locate all paragraphs containing the word. In each located paragraph the neighborhood of ±1-word around the candidate word are used to form phrases. The user is then prompted to specify which of the scenarios constitute a compound term.

If the user indicates that none of the listed scenarios is a term then the next located paragraph is processed. BuildTermList can be used to find terms ranging from single to three word terms. Figure 4-6 shows a list of words that BuildTermList extracted from a NASA article.

4.5 Summary

In this chapter we described *IDFconvert*, our prototype of the IDF encoder and decoder. *IDFconvert* was developed to illustrate an exemplary software that reads and writes data according to the IDF specification. First, we described the functional specification for the program, then we provided the outlines of the encoding and decoding algorithms used. Next, we presented the issue of employing PC-KIMMO as a standard word dictionary and discussed the problems of transforming an inflection form to its root form and vice versa. Finally, we described two Perl programs: Tokenizer and BuildTermList that accompany IDFconvert. Tokenizer and BuildTermList operate on the input file to produce a tokenized file and special words file respectively.

BEFORE

This could not be true,because up to then he had understood it perfectly; at least, he answered my questions exactly as if he did.

He and his wife, the old lady who had received me,looked at each other in a frightened sort of way. He mumbled out that the money had been sent in a letter,and that was all he knew. When I asked him if he knew Count Dracula, and could tell me anything of his castle, both he and his wife crossed themselves, and, saying that they knew nothing at all,simply refused to speak further. It was so near the time of starting that I had no time to ask anyone else, for it was all very mysterious and not by any means comforting.

Just before I was leaving, the old lady came up to my room and said in a hysterical way: "Must you go? Oh! Young Herr, must you go?" She was in such an excited state that she seemed to have lost her grip of what German she knew, and mixed it all up with some other language which I did not know at all. I was just able to follow her by asking many questions. When I told her that I must go at once, and that I was engaged on important business, she asked again:

Figure 4-4. Sample text before it was processed by Tokenizer.

AFTER

This could not be true , because up to then he had understood it perfectly ; at least , he answered my questions exactly as if he did .

He and his wife , the old lady who had received me , looked at each other in a frightened sort of way . He mumbled out that the money had been sent in a letter , and that was all he knew . When I asked him if he knew Count Dracula , and could tell me anything of his castle , both he and his wife crossed themselves , and , saying that they knew nothing at all , simply refused to speak further . It was so near the time of starting that I had no time to ask anyone else , for it was all very mysterious and not by any means comforting .

Just before I was leaving , the old lady came up to my room and said in a hysterical way : " Must you go ? Oh ! Young Herr , must you go ? " She was in such an excited state that she seemed to have lost her grip of what German she knew , and mixed it all up with some other language which I did not know at all . I was just able to follow her by asking many questions . When I told her that I must go at once , and that I was engaged on important business , she asked again :

Figure 4-5. Effects of dehyphenation and tokenization on the sample text.

AP
Aerospace Daily
Aerospace Industries Association
Launch Control Center
Launch Processing System
Long March
Los Angeles Times
Magellan
Martin Marietta
Mercury
Mir
NASA Select
National Science Foundation
New Mexico
New York Times
Nicogosian
Orbital Science
Orbital Sciences
Pacific
President
Rocketdyne
Russian
Russians
SYNCOM
Skynet
Soviet
Soviet Union
Soviets
Space Services
Space Shuttle
Space Technology
Swedish Space
Test Range
UPI
United Press
United Press International
United States
University
Vice President
Washington Post
Western
White House
White Sands
Zenit

Figure 4-6. Special words generated by *BuildTermList*.

4.6 References

[1] Wall, L. and R. L. Schwartz, *Programming in Perl*, O'Reilly & Associates, Inc., 1990.

[2] Antworth, E. L. *PC-KIMMO: a two-level processor for morphological analysis*. Summer Institute of Linguistics. 1990.

[3] Karp D., Y. Schabes, M. Zaidel, and D. Egedi, A Freely Available Wide Coverage Morphological Analyzer for English, *Proceedings of the fifteenth International Conference on Computational Linguistics COLING-92*, 3, 950-954, 1992.

[4] Koskenniemi, K., Two-level morphology: a general computational model for word-form recognition and production, Technical report, University of Finland, 1983.

[5] Karttunen, L. and K. Wittenburg, A two-level morphological analysis of English, *Texas Linguistic Forum*, **22**, 217-228, 1983.

[6] Karttunen, L., KIMMO: A two-level morphological analyzer, *Texas Linguistic Forum*, **22**, 165-186, 1983.

[7] Hodges, J. C., M. E. Whitten, J. Brown, and J. Flick, *Harbrace College Handbook for Canadian Writers*, Third ed., Harcourt Brace Jovanovich, 183-187, 1990.

[8] Quirk, R., S. Greenbaum, G. Leech, and J. Svartvik, *A Comprehensive Grammar of the English Language*, Longman, 1972.

[9] Stoker, B., *Dracula*, 1897.

5. Experiment

This chapter describes the results of IDF encoding texts using the program IDFconvert. Testing of the IDF encoding method showed to be satisfactory according to the IDFconvert's specification. The experimental data used for the test was the electronic text version of Dracula [1].

5.1 Sample Data

Dracula has been chosen as test data because it provides a large sample of general language text for testing purpose. The file comes in both HTML and plain text formats, and is one of the many public domain etexts. These etexts are available on many of the Internet's gopher, ftp, and World Wide Web sites.

The plain text format of the file Dracula (kindly supplied by T. and F. Daniels) is used as a source of testing data for program *IDFconvert.* The size of the text can be described from two points of view: physical and logical. From a physical point of view, the file size can be summarized according to the number of lines of text, number of tokens, and number of bytes. From a logical point of view, the size can be described in terms of the number of chapters, paragraphs,

and words. We run the file Dracula through the UNIX utility program *wc* to find that it contains 15174 lines of text, 160419 tokens[25], 862746 bytes. We then use our Perl program *cntSection* to count the number of logical sections in the file. *cntSection* reports that Dracula has 27 chapters, 2023 paragraphs, and 160419 words. A chapter is defined as a section of text which begins with a line containing the chapter title followed first with two empty lines and then the series of body paragraphs. Paragraphs in a chapter are separated from each other by a single empty line, and adjacent chapters are separated from each other by two empty lines.

In order to get an accurate evaluation of IDFconvert's performance, we run the program on three data files of different sizes. We use the file Dracula to create two smaller test files, one small sized, Dracula_S.txt, and one medium sized, Dracula_M.txt. Dracula_S.txt consists of the first chapter of Dracula.txt and is roughly 32K in size, whereas Dracula_M.txt consists of the first ten chapters and is about 300K in size. The source file Dracula was renamed to Dracula_L.txt as to make the name consistent with the others.

File name	#bytes	#lines	#tokens	#chapters	#paragraphs	#words
Dracula_S.txt	32205	566	5709	1	80	5709
Dracula_M.txt	300303	5292	55593	10	676	5292
Dracula_L.txt	862746	15174	160419	27	2023	160419

Table 5-1. Summary of the sizes of the data files.

[25] This type of token includes word that have a punctuation mark attached at the end of the word. e.g. "hello!"

5.2 Experimental Procedure

5.2.1 Preprocessing

Before the data files can be encoded by *IDFconvert*, they must be prepared by the Tokenizer preprocessor. This is the first step of the experimental procedure. Each data file is run through the Tokenizer, which outputs a tokenized file. From the input files Dracula_S.txt, Dracula_M.txt, and Dracula_L.txt, three tokenized files Dracula_Stok.txt, Dracula_Mtok.txt, and Dracula_Ltok.txt are generated. After the Tokenizer finishes processing each file, it reports the following facts on the output file: the number of lines of text, number of tokens, number of bytes, and the number of different word tokens there are among the accounted tokens. Table 5-2 shows the above information on the three tokenized files.

File	# of lines	# of tokens	# of bytes	# of distinct word tokens	% increase in file size
Dracula_Stok.txt	521	6554	33793	1549	4.9%
Dracula_Mtok.txt	4849	64610	317121	6249	5.6%
Dracula_Ltok.txt	14057	187424	913343	10423	5.8%

Table 5-2. Summary of the size of the tokenized data files.

5.2.2 Creating dictionary files

Before IDFconvert begin to encode a tokenized file it looks for the files *hi-frequency.dict* and *special.dict* to initialize its coding dictionaries. These files must be provided by the user.

We use a text editor to create hi-frequency.dict. The file contains the 64 top ranked frequently used word in English as described in [2]. See Appendix A for this list of the high-frequency words.

To generate the special.dict file we use the Perl program BuildTermList with Dracula_L.txt as the input file. But generating a comprehensive list of terms requires a lot of human effort and is time consuming. The number of word tokens that need to be checked and confirmed is just so great (over 9000 tokens) even if we are doing it semi-automatically. So after 154 terms have been identified we decided to stop the program. By taking this step we know for certain then that there are many words in the file will not be encoded since they are not found in either the high-frequency dictionary or the PC-KIMMO lexicon. The candidate special words tend to be proper nouns, foreign words, hyphenated compounds, or words that have a different meaning in the genre than in regular language. In chapter 3, we described that when the encoder encodes a file, words that are not found in the four coding dictionaries will be stored on a character-by-character basis. However, if we were to store a word as a "raw" ASCII string in the IDF file, such form of storage does not facilitate search and retrieval of text, nor does it help compress the file any better. Instead, we tell IDFconvert to encode all unknown text tokens as special words (see 4.2.1.1). Appendix C shows a listing of the 154 special words entries in file *special.dict.*

5.2.3 IDF Encoding

Having done all the necessary preparation, it is time to IDF encode the files. We run IDFconvert with three arguments, (1) an option switch that instructs IDFconvert to encode exceptional tokens in special word blocks, (2) the name of the tokenized input file, and (3) the name of the output IDF file. A summary of the outputs is shown in Figure 5-1.

5.2.4 Discussion

Due to the (side) effect of encoding, IDF encoded files are compressed. The compression ratios observed for encoding the files Dracula_Stok.txt (33 KBytes), Dracula_Mtok.txt (317 KBytes), and Dracula_Ltok.txt (917 KBytes) were 17.7%, 41.2%, and 42.7% respectively. An examination of IDFconvert's output reveals that for a small text file, the size of the coding dictionaries is the same as the size of the encoded data. On the other hand, for larger text files, the size of the coding dictionaries can be as small as one third of the size of the encoded data. Thus, it appears that there is a minimum size of text such that if the text size is smaller than this threshold value then no compression will be achieved in the IDF file. Of course, we assume that the number of words that are stored raw in the IDF file is small and their contribution to the size of the IDF file is negligible.

We also observed that the times taken to encode the three files were about 1, 14, and 40 minutes respectively. Since all of these files are less than 1 MB in size, the execution times observed are below a satisfactory level. A reasonable expectation of execution time for the large data file is at most five minutes. There are two factors that have been determined to be the primary causes for the encoder's slowness: 1) the number of accesses to PC-KIMMO, and 2) the algorithm it uses to determine a standard word.

The biggest bottleneck in IDFconvert can be attributed to the accessing of the PC-KIMMO lexicon. In the current implementation of IDFconvert, a word lookup in PC-KIMMO's lexicon (database) requires the program to open a connection to the database. Then IDFconvert submits the query word and waits for PC-KIMMO to return the information string. Next, IDFconvert closes its connection to the database. Therefore, accessing PC-KIMMO is the major

reason for IDFconvert's tardiness which takes place for every standard word lookup.

Moreover, using an inefficient algorithm to determine standard word type also impedes the program's execution speed. As already mentioned, the encoder consults the PC-KIMMO's lexicon every time it encounters a word for which it cannot find an entry in the high-frequency word and special word dictionaries. This consultation step is carried out regardless of the fact that the program has already encoded the same word before. For example, suppose that the word *walking* occurs in the input file 1000 times. With the current implementation, whenever IDFconvert encounters the token *walking* it must consult the PC-KIMMO lexicon. This means that it performs 999 unnecessary accesses since every time it arrives at the same conclusion, i.e. encode the word as *walk* with inflection *ing*. Thus, the simple approach taken by IDFconvert makes the algorithm for determining standard words very inefficient. A great deal of computation time can be saved if the encoder maintains an encoding history of every entry in the standard word coding dictionary.

5.3 References

[1] Stoker, B., *Dracula*, 1897.

[2] Francis, W. N. and H. Kucera. *Frequency Analysis of English Usage, Lexicon and Grammar*. Houghton Mifflin Co., Boston, 1982.

Input file:	Dracula_Stok.txt	33793 bytes	
Output file:	Dracula_S.idf	27797 bytes	Compression Ratio[26] 17.7%
CODING DICTIONARIES:		14337 bytes	
IDF DATA:		13460 bytes	

hi-frequency word dictionary:	64 entries
exception token dictionary:	32 entries
standard word dictionary:	1131 entries
special word dictionary:	116 entries

Input file:	Dracula_Mtok.txt	317121 bytes	
Output file:	Dracula_M.idf	186331 bytes	Compression Ratio 41.2%
CODING DICTIONARIES:		53194 bytes	
IDF DATA:		133137 bytes	

hi-frequency word dictionary:	64 entries
exception token dictionary:	32 entries
standard word dictionary:	3932 entries
special word dictionary:	624 entries

Input file:	Dracula_Ltok.txt	913343 bytes	
Output file:	Dracula_L.idf	522687 bytes	Compression Ratio 42.7%
CODING DICTIONARIES:		136706 bytes	
IDF DATA:		385981 bytes	

hi-frequency word dictionary:	64 entries
exception token dictionary:	32 entries
standard word dictionary:	9486 entries
special word dictionary:	811 entries

Figure 5-1. IDFconvert on files Dracula_Stok.txt, Dracula_Mtok.txt, and Dracula_Ltok.txt.

[26] The calculated compression ration takes into account the size of the dictionary and IDF data.

6. Conclusion

6.1 Summary

The main objective of this thesis was to present the Intelligent Document Format encoding technique The proposed IDF encoding technique is based on the concepts of the *dictionary-based approach* to text compression [1, 2, 3, 4]. In particular, our work shares many similarities with the work of Akman [4].

The intention of IDF encoding is an attempt to make it easier for retrieval systems to manage information as an IDF document supports three useful operations:

1) allows a search program to coarsely locate documents of interest without the use of an index. A program can accomplish this by performing a search scan in the dictionary section of the IDF document looking for the queried words. The dictionary section of an IDF document is often less than 1/3 the size of the whole document[27].

[27] This is true only if the IDF encoded document is very large.

2) allows a search program to perform searches for words based on their syntactic labels. This feature is possible only if a text has been syntactically parsed before it is IDF encoded.

3) allows a search program to perform searches on different forms of a word, namely, base form and inflected forms. This feature is similar to conventional retrieval program first stems the queried key words before it search the index.

The method classifies words into four types and uses codeword dictionaries for storing them. The codeword dictionaries encode words in blocks of varying length, depending on the type of word being encoded. In general, words are encoded in either short or long blocks. Short blocks encode high-frequency words, and punctuation marks. Long blocks are used to encode standard and special words. The following information are encoded in long blocks: root, inflection, and case attribute, and optionally part of speech, and word sense data. In addition, two short blocks are used as special markers to delimit unencoded words. A word will not be encoded if either 1) the high-frequency word, special word, and standard word dictionaries do not contain an entry for it, or 2) an overflow occurs in the special word and/or standard word coding dictionary.

6.2 Concluding Remarks

We have reported some disappointing news based on the experiment performed (chapter 5). The performance of the prototype encoder, IDFconvert, was rather unsatisfactory. It took the program 40 minutes to encode a 1 MB text file, a fact that renders the program impractical to use by any one. Nevertheless, IDFconvert's performance can be attributed to the author's inefficient coding fashion as well as the inherent complexity O(n) of the encoding algorithm. In addition, although the goal of our work is not text compression, the IDF encoding

method does achieve that as a side effect. The compression ratios observed from the experiments ranged from 18% to 43% (about 50% to 60% if we don't include coding dictionaries). As far as compression ratio goes, the values produced by IDFconvert are not impressive compared to that of general purpose compressors such as zip, gzip, and compress. However, we need to bear in mind that these values are not true indicators of the lower and upper bounds of the compression ratios of the IDF method.

6.3 Future work

We now suggest some of the possible future directions in which this work can be continued.

- We have observed from the encoding experiments in Chapter 5 that the major bulk of an IDF encoded file is attributed to the storing of the coding dictionaries. With IDFconvert, a dictionary is written out as a sequence of *entry* and *value* pairs, with the entry is stored as a two byte integer and the value is stored as a string of characters. Thus, there is no efficient storage of the dictionaries. The encoder can be modified so that it uses a different encoding model when writing out the coding dictionaries in order to obtain a better compression ratio of the IDF encoded file. A suitable encoding model for this task could the Huffman method (Chapter 2).
- Another possible area of future study is the development of a *two-pass* encoder. Currently, IDFconvert is implemented as a *single-pass* encoder, that is, the program encodes the input in its first pass through the text. The weakness of this implementation is that it takes too long to encode a large text file. A *two-pass* encoder would significantly improve the encoding time.

- Presently the output produced by IDFconvert is a single file that contains two components, the IDF encoded text and the coding dictionaries. However, an independence relationship between these components ought to be established so that they can be saved as separate files. The proposed scheme will have the storage cost remain unchanged, while the transmission cost on a limited bandwidth communication line, say a modem line, will be significantly reduced (assuming that the receiver already has the dictionary component).
- A file format conversion program can be developed so that documents stored in one format can be easily converted to another.
- Developing a more sophisticated token extractor for IDFconvert.

6.4 References

[1] Gutmann, P. C. and T. C. Bell, A Hybrid approach to text compression, *Proceedings IEEE Data Compression Conference*, IEEE Computer Society Press, 225-33, 1994.

[2] Witten, I. H., M. Alistair, and T. Bell, *Managing Gigabytes: Compressing and indexing documents and images*, Van Nostrand Reinhold, 1994.

[3] Moffat, A., N. Sharman, and J. Zobel, Static compression for dynamic texts, *Proceedings IEEE Data Compression Conference*, IEEE Computer Society Press, 126-35, 1994.

[4] Akman, I. K., A new text compression technique based on language structure, *Journal of Information Science*, **21**, (2), 87-94, 1995.

Appendix A

A listing of general language high-frequency words as was summarized by C. Fox 1992. The list is derived from the Brown Corpus (Francis and Kurcera 1982).

a
about
above
across
after
again
against
all
almost
alone
along
already
also
although
always
among
an and
another
any
anybody
anyone
anything
anywhere
are
area
areas
around
as
ask
asked
asking
asks
at
away
b
back
backed
backing
backs
be
because
become
becomes
been
before
began
behind
being
beings
best
better
between
big
both
but
by
c
came
can
cannot
case
cases
certain
certainly
clear
clearly
come
could
d
did
differ
different
differently
do
does
done
down
downed
downing
downs
during
e
each
early
either
end
ended
ending
ends
enough
even
evenly
ever
every
everybody
everyone
everything
everywhere
f
face
faces
fact
facts
far
felt
few
find
finds
first
for
four
from

full
fully
further
furthered
furthering
furthers
g
gave
general
generally
get
gets
give
given
gives
go
going
good
goods
got
great
greater
greatest
group
grouped
grouping
groups
h
had
has
have
having
he her
herself
here
high
higher
highest
him
himself
his
how
however
I
if
important
in
interest
interested
interesting
interests
into
is
it
its
itself
j
just
k
keep
keeps
kind
knew
know
known
later
latest
least
less
let
lets
like
likely
long
longer
longest
m
made
make
making
man
many
may
me
member
members
men
might
more
most
mostly
mr
mrs
much
must
my
myself
n
necessary
need
needed
needing
needs
never
new
newer
newest
next
no
non
not
nobody
none
nothing
now
nowhere
number
numbered
numbering
numbers
o
of
off
often
old
older
oldest
on
once
one
only
open
opened
opening
opens
or
order
ordered
ordering
orders
other
others
our
out
over
p
part
parted
parting
parts
per
perhaps
place
places
point
pointed
pointing
points
possible

present
presented
presenting
presents
problem
problems
put
puts
q
quite
r
rather
really
right
room
rooms
s
said
same
saw
say
says
second
seconds
see
seem
seemed
seeming
seems
sees
several
showing
shows
side
sides
since
small
smaller
smallest
so
some
somebody
someone
something
somewhere
state
states
still
such
sure
t
take
taken
than
that
the
their
them
then
there
therefore
these
they
thing
things
think
thinks
thins
those
though
thought
thoughts
three
through
thus
to
today
together
too
took
toward
turn
turned
turning
turns
two
u
under
until
up
upon
us
use
uses
used
using
v
very
w
want
wanted
wanting
wants
was
way
ways
we
well
wells
went
were
what
when
where
whether
which
while
who
whole
whose
why
will
with
within
without
work
worked
working
works
would
x
y
year
years
yet
you
young
younger
youngest
your
yours
z

Appendix B

A listing of the Penn Treebank Part of Speech Tags as was provided by Brill (1993).

1.	CC	Coordinating conjunction
2.	CD	Cardinal number
3.	DT	determiner
4.	EX	Existential "there"
5.	FW	foreign word
6.	IN	Preposition or subordinating conjunction
7.	JJ	Adjective
8.	JJR	Adjective, comparative
9.	JJS	Adjective, superlative
10.	LS	List item marker
11.	MD	Modal
12.	NN	Noun, singular or mass
13.	NNS	Noun, plural
14.	NP	Proper noun, singular
15.	NPS	Proper noun, plural
16.	PDT	Predeterminer

17.	POS	Possessive ending
18.	PP	Personal noun
19.	PP$	Possessive pronoun
20.	RB	Adverb
21.	RBR	Adverb, comparative
22.	RBS	Adverb superlative
23.	RP	Particle
24.	SYM	Symbol
25.	TO	"to"
26.	UH	Interjection
27.	VB	Verb, base form
28.	VBD	Verb, past tense
29.	VBG	Verb, gerund or present participle
30.	VGN	Verb, past participle
31.	VBP	Verb, non-3rd person singular present
32.	VBZ	Verb, 3rd person singular present
33.	WDT	Wh-determiner
34.	WP	Wh-pronoun
35.	WP$	Possessive wh-pronoun
36.	WRB	Wh-adverb

Appendix C

A listing of the 154 special words entries in file *special.dict* generated by BuildTermList.

Abbey
Acherontia
Adelphi
Admiralty
Aerated
Albemarle
America
American
Americanism
Amramoff
Blyme
Bosphorus
Bucharest
Buda
Buda-Pesth
Bukovina
By-and-by
Carpathian
Carpathians
Casabianca
Cassova
Chicksand
Conquistadores
Crucifix
Dailygraph
Deil
Demeter
Disraeli
Enoch
Enoch's
Esk
Euthanasia
Fenchurch
Festina
Ghoorka
Gibraltar
Godalming
Godalming's
Hermanstadt
Herr
Herr's
Herren
Hildesheim
Honfoglalas
Hospadars
Ignotum
Inured
Ittin
Korkrans
Krone
Kukri
Lenore
Lethe
Leutner
Malvolio
Marmion
Marquand
Marquesas
Matapan
Mediasch
Mein
Nordau
Norfolk
Norway

Nuremberg
Odessus
Olgaren
Omme
Omnia
Omnipotent
Ophelia
Ordog
Pampas
Piccadilly
Pokol
Pouf
Prund
Pruth
Ristics
Roman
Romanoffs
Romany
Rome
Roumanian
Roumanians
Rufus
Runswick
Sackville
Sanguine
Saxon
Saxons
Schnell
Scholomance
Scythia
Sereth
Servian
Soho
Spencelagh
Sphinges
Strasba
Szgany
Titicaca
Tobolsk
Todten

Transcendentalism
Transylvania
Transylvanian
Tyke
Ugric
Vampire
Vampire's
Wafer
Wallach
Wallachian
Wallachs
Wodin
Yabblins
Ye
Yus
Zoophagous
acant
acrewk
acrid
adamantine
addle
adduce
adduced
agglomeration
antherums
asinine
baccabox
badinage
barometrical
basilisk
bauble
bestrewed
bierbier-bank
blood-stained
body-snatcher
by-and-by
cicatrised