# Use Of Text Fragments In Compression And Searching Of Natural Language Databases

Venkat Kotamarti

A Major Technical Report

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montréal, Québec, Canada

September 1985

ISBN 0-315-30636-X

## ABSTRACT

Use Of Text Fragments For Compression And Searching Of Natural Language Databases

Venkat Kotamarti

The volume of data managed by information systems has increased rapidly over the last few years, making it necessary to find ways to compact the data to reduce storage and transmission costs, and to maintain a satisfactory access time. This report discusses the data compression techniques based on fragment dictionary to reduce the storage requirement for databases. The theoretical background for compression is presented. A survey of data compression methods which use fixed length code representations of variable length chracter strings as language elements is given. Program designs for generation of a symbol set, compression and decompression are given. Statistics are given with regard to compression ratios and processing times. Compressed databases are organized on the basis of a database partitioning scheme as proposed by [Goyal, 1983] to allow for string searching on compressed databases. The statistics for string search using such a scheme are presented.

## ACKNOWLEDGMENTS

I wish to thank Dr. P. Goyal for his patience and excellent support while supervising this work. I am grateful to Mr. J. P. Poirier for his encouragement, as my manager at Bell Northern Research, Montreal, and finally I thank the National Science and Engineering Research Council (NSERC) for the financial support for the duration of the research.

## TABLE OF CONTENTS

# 1.0 INTRODUCTION

## 1.1 OVERVIEW

The increased awareness and usage of information systems such as database management systems, information retrieval systems etc. has brought a rapid growth in the volume of data maintained by such systems. Advances in storage technology have not been able to keep pace with such growth. Increased online access to data accentuates this demand and thus requires data to be held more compactly than at present, if a satisfactory service at a reasonable cost is to be provided [Yannakoudakis,1982]. The volume of published material has also been growing rapidly, making it increasingly difficult for computerized storage. For example, if only titles and index terms of five years of Chemical Abstracts Condensates are to be represented in eight bit characer form, then atleast eight normal density 800 BPI, 2400 ft tapes would be required [Schuegraf,1973]. Even if the cost of storage is not to be considered, the inconvenience in handling multiple tapes is not bearable. Furthermore, data stored on tapes is not suitable for access in an online environment.

There are many instances when information in compressed form would bring in savings in storage, processing time and, more importantly, increased user satisfaction due to quick response time.

Compression of information has been very attractive in cases where information is not accessed regularly like archive files and where the main consideration is minimizing storage space. As we witness an increased growth in the use of local/wide area computer networks which result in the increased geographical diversity in the user groups and electronic mail, the volume of information transmitted through the networks increases. Often, the cost of transmission and the bit error rate are directly proportional to the volume of information being transmitted. This is especially true in the case of public data networks like Bell Canada's Datapac where cost of transmission is directly related to the number of packets transmitted. Savings could be achieved if we could find ways to compress the amount of information to be transmitted, yet keeping the exact information content.

As the amount of information being stored in the information systems increases, the task of maintaining a quick response time to user requests becomes difficult. Much effort has been put in the design of information systems so that access to the data is easier and faster. Various solutions to achieve reasonable response times have been suggested, ranging from the use of specialized hardware to the file structures tailored to a particular application [Schuegraf,1973]. To reduce the response time, the system must locate the required piece of information in a file very quickly. This means that the information must be organized in a manner that provides fast access to a particular item. Algorithms for speeding text searches strive to reduce the number of times a character of a text file has to be examined in searching for a string [Goyal,1983]. Some information systems like LEXIS, a legal database, use inversion at a word level. This approach would certainly improve the search process at the cost of large additional storage required for the inversion tables.

Most text compression techniques reduce the storage requirement by representing the original text in a coded representation at the price of increased processing time needed for compression and decompression. As the CPU time becomes cheaper relative to the cost of external storage devices, compression appears as an increasingly attractive option for dealing with large files. Since most compression methods make use of coded representations, the search for a string on a coded text would be much faster; decompression would not have to be performed until the target string is found. This would certainly bring an increased user satisfaction in most information retrival systems, interactive database management systems, etc.

This report discusses the possible methods of text compression and decompression, and procedures to perform a search on a compressed text file. Programs have been written to generate statistics such as compression factor, CPU time needed for compression and decompression, and to carry out a search for a string on a coded text file. The main objective of this project is not to arrive at an optimal method of coding, decoding and searching, but to generate comparative statistics in terms of storage and processing time needed to carry out a search for a string in non-compressed vs compressed environments.

All the statistics shown in this report are based on IBM 3084 computer system using CMS.

## 1.2 ORGANIZATION OF THE REPORT

Chapter II presents the theoretical background for compression including information theory concepts as applied to the information sciences. Chapter III discusses some of the approaches to data compression. In chapter IV, methods for compression, decompression are given. This chapter also discusses the possible data structures and the program design for a chosen method. chapter V explains a string search method on a compressed database. It also contains the details for data structures and program design for the string search method. Chapter VI concludes this report by analysing the statistics and places the complete project in a perspective.

## 2.0 INTERPRETATION OF INFORMATION THEORY FOR INFORMATION SCIENCES.

Traditionally, characters form the basic symbols set for a natural language. Some form of digital code is used to represent characters of a language for storage and transmission. A wide range of characters must be available if high 'quality' is to be obtained when computer typesetting is used to produce material ranging from non-technical English prose to mathematical, chemical, or linguistic articles [Lynch,1982]. This means a character set must include characters of both upper and lower cases, subscripts, superscripts, punctuation symbols, digits, graphical symbols etc. to support wider range of disciplines. Depending on the required 'quality' of the text, the number of bits used for a character representation can be anywhere between 4 to 12 bits. Shorter codes can encode only a reduced character set and hence, some form of 'shift' or 'escape' character has to be used to code an extended character set to have a reasonable quality in the text. On the other hand, a longer code provides a higher quality of ~~the text~~ because of its ability to support a larger character set at the price of an increased storage requirement for the resulting text.

If we observe natural language text, we notice that some characters are found relatively more often than others.

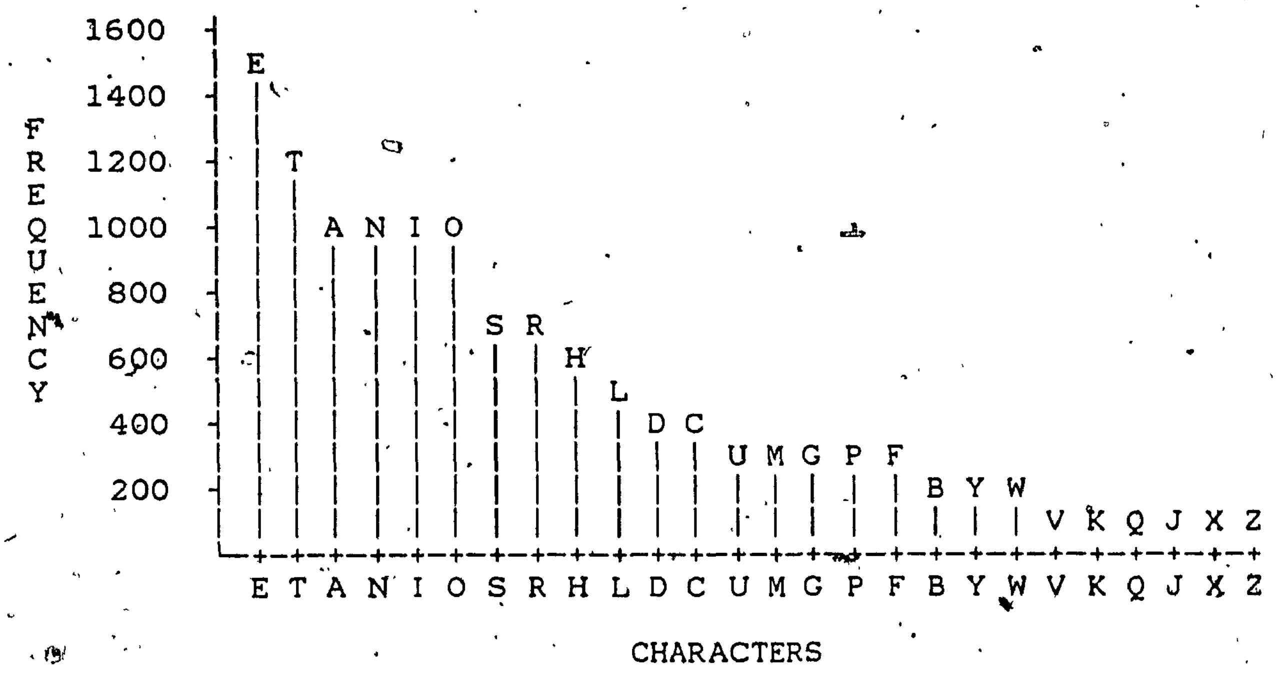
This implies that characters in a natural language do not occur with equal frequency. In English text, certain characters such as e, i, t etc, seem to occur much more often than j, k, q etc. The following table, created from a text of 15280 characters shows the frequency distribution of characters.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 'A'= | 988 | 'B'= | 207 | 'C'= | 386 | 'D'= | 442 | 'E'= | 1531 |
| 'F'= | 273 | 'G'= | 285 | 'H'= | 604 | 'I'= | 954 | 'J'= | 17 |
| 'K'= | 60 | 'L'= | 475 | 'M'= | 312 | 'N'= | 978 | 'O'= | 946 |
| 'P'= | 277 | 'Q'= | 9 | 'R'= | 727 | 'S'= | 743 | 'T'= | 1114 |
| 'U'= | 343 | 'V'= | 122 | 'W'= | 196 | 'X'= | 35 | 'Y'= | 204 |
| 'Z'= | 7 | | | | | | | | |
| '0'= | 22 | '1'= | 14 | '2'= | 4 | '3'= | 3 | '4'= | 8 |
| '5'= | 2 | '6'= | 3 | '7'= | 4 | '8'= | 6 | '9'= | 11 |
| ' '= | 2366 | '¢'= | [illegible] | '.'= | 93 | '<'= | 0 | '('= | 5 |
| '+'= | 0 | '\|'= | 0 | '&'= | 0 | '!'= | 1 | '$'= | 6 |
| '*'= | 0 | ')'= | 5 | ';'= | 6 | '¬'= | 0 | '—'= | 44 |
| '/'= | 0 | 'ù'= | 0 | ','= | 145 | '%'= | 0 | '_'= | 0 |
| '>'= | 0 | '?'= | 2 | ' '= | 0 | ':'= | 2 | '#'= | 0 |
| '@'= | 0 | '''= | 28 | '='= | 0 | '"'= | 36 | '°'= | 0 |
| '{'= | 0 | ']'= | 0 | '\'= | 0 | | | | |

```
text file size =    15280  characters
       letters =    12235  ==>80.07%
        digits =       77  ==>0.50%
        blanks =     2366  ==>15.48%
   punctuation =      373  ==>2.44%
   no of lines =      229  ==>1.50%
```

Character counts in text of 15280 characters.

The following graph shows the hyperbolic distribution of character occurrences shown in the above table.



Graphical representation of the character distribution of the above table. Blanks and punctuation marks are ignored.

This distribution has been known for a very long time and is, among others, the basis of MORSE code for message transmission. Because of this wide disparity in the frequency distribution of characters, the traditional assignment in computer systems of fixed equal length code to all characters needs to be examined.

We could adopt the mathematical theory of communication as given by C. E. Shannon to find ways to convert the hyperbolic distribution of character occurrences to a more rectangular distribution so that all characters will have equal frequency. Shannon, a communications engineer at Bell

Labs, provided some generalizations with regard to the efficiency in transmission of information in communication channels and information carrying capacity of these channels whether free from or subject to noise. [Lynch,1977] has interpreted Shannon's theory to information science. He concluded that in order for fixed length coding to be used, symbols must have equal frequency. In such a case, the most efficency in storage and transmition of information is achieved.

Shannon provided some quantitative measures of this efficiency expressed as the entropy of the message or the information content of the message.

He noted that messages contain less information than they are capable of containing implying the presence of redundancy. This redundancy appears mainly in the form of dependencies between adjacent symbols of the message.

Shannon provided the following quantitative measures to calculate the information content in a message which in fact provides the minimum length of the message to represent the same information content. Consider ($S_1$, $S_2$, $S_3$,,,$S_N$) are N different symbols in a language and their respective probabilities of appearing in a given message are ($P_1$, $P_2$, $P_3$,,,$P_N$). Then, the average information content or the entropy of a message is given by the following:

$$H = -\sum_{i=1}^{N} [P_i * \text{LOG}_2 P_i] \qquad (1)$$

H being measured in bits. Also, the average length, A, of a symbol in a message may be calculated by:

$$A = \sum_{i=1}^{N} [P_i * B_i] \quad \text{bits} \qquad (2)$$

where $B_i$ represents the length of code, in bits, for symbol i and $P_i$ indicates the probability of symbol i to appear in a message.

If there are T symbols in a message, the total length of the message is given by

$$L = T * A \qquad (3)$$

Hence, it is desirable to obtain the lowest possible i.e. minimum value for A. Shannon also indicated that for encoding without error H must be less than or equal to A. This means that the average number of bits needed to represent a symbol must be greater than or equal to the entropy. The desired equality can only be obtained when the following condition is satisfied for all the symbols in the language:

$$b_i \geq - \mathrm{LOG}_2 \, P_i \qquad (4)$$

Since, the number of bits needed to represent a symbol must be an integral number, this condition has little practical value. Instead, the following condition ensures the most efficient transmission and storage of information:

$$A = - \sum_{I=1}^{N} [P_i * \lceil ( \mathrm{LOG}_2 \, P_i ) \rceil ] \qquad (5)$$

This is the theoretical minimum. By calculating this value for a natural language i.e. its theoretical minimum value, one could derive the measure of performance.

We could apply Shannon's findings to information sciences. We could measure the performance quantitatively of any coding scheme, by comparing the length of a text file using the coding scheme with its theoretical minimum.

Hence, the performance of any coding technique that tries to represent the information content of a text file or a database in a coded representation can be measured as follows:

$$\text{Efficiency} = \frac{\text{Theoretical minimum length of the coded text file}}{\text{Length of the coded text file}}$$

Efficiency value of 1 is optimal. However, a more practical measure of efficiency in a computing environment can be given as follows:

$$\text{Coding efficiency} = \frac{\text{Length of the coded text file}}{\text{Length of the original text file}}$$

where the length of the original text file is given in the terms of the character coding scheme employed on the specific computer or in the standard coding schemes ASCII and EBCDIC, where 8 bits are used to represent a character.

This efficiency is actually the compression factor and a direct indication of savings in storage achieved by using a specific coding scheme.

Thus, information theory provides some insights into the characteristics of the natural language and some quantitative measures of performance for coding schemes.

## 3.0 APPROACHES TO TEXT COMPRESSION

Text compression is achieved by reducing the number of bits needed to represent the text for storage and transmission. There are different approaches to achieve this goal. There are methods that achieve higher compression factors by tailoring the technique to the environment, and there are 'universal' methods with lower but significant compression factors which can operate on any kind of text.

Some methods achieve compression by reformating the original text to remove the series of blanks, redundant and unnecessary information. A significant compression factor can be achieved by such a rearrangement, particularly in the case of file organizations with fixed length records. Some methods compress the text by assigning codes to the words with distinct words being assigned the same code. Even though these methods achieve compression, they are not considered in this report. Only reversible compression techniques are dealt with. Reversible means that a chosen compression technique must allow the original text or database to be regenerated from the coded representation with out any loss of information.

Text compression is possible because of the nature of frequency distribution of the characters, which approximates to hyperbolic or zipfian distribution, and the presence of redundancy in the form of dependencies among the characters. Compression methods differ mainly in the symbol sets used (which are totally different from the traditional character sets). In this process, the text is made up of 'symbols', the language elements. The symbol set may include the basic character set and additional symbols. The additional symbols, which are the combination of basic chracters, are included to handle the disparate frequencies. The symbols chosen may be of fixed or variable length. The code assignment to these symbols can vary from method to method. The code chosen may be of fixed or variable length. And hence, there exist four possible compression schemes. They are:

1. Fixed length symbols - Variable length codes
2. Fixed length symbols - Fixed length code
3. Variable length symbols - Variable length codes
4. Variable length symbols - Fixed length codes

All these schemes perform text compression. They differ mainly in the ease of implementation and their dependence on the text. All these schemes are briefly described in the following sections.

## 3.1 FIXED LENGTH SYMBOLS - VARIABLE LENGTH CODES

The very first significant compression of a natural language is the familiar Morse code, in which Samuel Morse coded the text by replacing the characters with combinations of dots, dashes and spaces. Characters that have higher probabilities of occurring were assigned shorter codes, while those with low probabilities were given longer codes. By assigning shorter codes to most frequent characters, the total length of the message is reduced. This way of coding was later improved by Shannon and Huffman, who made use of a binary alphabet in assigning the codes. Huffman provided a method where the total number of bits in the text is reduced, while decoding of the text coded using his scheme can be done instantaneously. His method is essentially to create a decode tree, in which the external nodes represent characters, and where the external path length is minimal. That means, for a character k with a probability of p and distance d from the root to the node for character k, the method tries to minimize the sum of all the products of p and d for all characters.

Huffman's coding scheme provides compressed databases or text files of minimum length. However, the most significant disadvantage is that it would require a highly complicated logic to manipulate variable length codes on machines with fixed length words. Besides the time spent ~~on~~ such manipulations of these codes, a considerable amount of CPU time is needed to be spent at the decompression phase as it involves the search of a tree (the codes do not have unique code beginnings or endings). Hence, this scheme is highly suitable for situations where minimization of the physical storage space is important and processing time is not of concern, implying that this technique can not be used for on-line databases and only archival files are better candidates.

## 3.2 FIXED LENGTH SYMBOLS - FIXED LENGTH CODES

The symbol set, in this scheme, contains symbols that are of fixed length. They could be single characters or a combination of characters. As all possible combination of characters of the required length have to be encoded, this scheme is suitable for static environments i.e., where the exact combinations are precisely defined. The scheme may also be employed in situations where the number of symbols is limited and could employ a code of length shorter than that on the host machine. For example, in situations where the case of the characters is immaterial and some punctuation symbols never occur, the number of symbols in the character set could be restricted to, say 64, and thus the characters can be uniquely encoded using a 6 bit rather than the usual 8 bit code. This results in a saving of 25%.

## 3.3 VARIABLE LENGTH SYMBOLS - VARIABLE LENGTH CODES

This scheme attracted a lot of attention from researchers. Many forms of compression techniques are possible using this scheme, because a varible length language element can be chosen in many ways. A chosen language element may be a word, which is a natural unit of the language, or a fragment of a word, which is called a word fragment, or even a textual fragment which can include blanks and punctuation marks.

Thiel and Heaps proposed a compression scheme in which words were used as language elements [Thiel,1972]. Words were given the variable length frequency dependent codes. The resulting database was stored in compressed form on a magnetic tape and the dictionary establishing the relationship between the words and the codes, was stored on a disk. The dictionary had to be referred for coding and decoding operations. The complete storage analysis is given by [Heaps,1972]. Even though this method offers a good compression ratio, it suffers from many disadvantages. The size of the dictionary is dependent on the database and hence, needs to be updated whenever the database is modified, which means there is no 'universal' dictionary. Zipf's law indicates that 50 per cent of the words in a database or a text file appear only once, 16.67 per cent of the words appear only twice, and for small values of n, a fraction of (1/(n(n+1))) of the words occur n times. This means, that a majority of the words appear only once, and most of the remaining appear infrequently. Hence, savings achieved by replacing the words by codes disappear by having to store them in the dictionary, instead. Thus, the use of words as language elements does not provide any significant savings.

There have been a number of proposals to form language elements that are shorter than words, called word fragments which are textual fragments that are completely contained within the words. Symbol sets, which contain symbols that may cross word boundaries and contain blanks and punctuation marks, called text fragments, have also been proposed. The procedures to make a selection of a symbol set containing word fragments have been documented by [Colombo, 1969] and those of text fragments by [Lynch,1973], [Schuegraf,1973] and [Yannakoudakis,1982].

After the selection of a proper symbol set, variable length codes must be assigned. Huffman's codes provide compressed databases of minimum length. This scheme suffers from all the disadvantages of Huffman's scheme like the probabilities of the symbols must be found before the assignment of codes and the complicated processing and manipulation of variable length codes. Decompression is slow due to the size of the dictionaries for larger files or databases.

## 3.4 VARIABLE LENGTH SYMBOLS - FIXED LENGTH CODES

Many compression techniques use this scheme. This scheme is very popular because efficient and fast decompression is possible. Once the proper symbol set is chosen, which is similar to that of the previous method, codes of fixed length are assigned. A dictionary is maintained to establish the relation between the symbols and the codes. Decompression is very efficient because this process is simply a table look up with the code as the index in to the table. A proper size for the codes can be chosen to meet the machine characteristics. For example, a code size of eight would allow a symbol set of 256 and would nicely fit in any byte oriented machine. However, compression is some what ineffcient because of variable length language elements.

Fixed length coding implies that all the language elments chosen as the symbols are expected to have equal probability, and have equal number of bits in their codes. This means that value of A in equation (2) and that of H in equation (1) can be made to have the same value. This produces the best possible compression with the given set of symbols. That means, that the average number of bits needed to represent the information content of a symbol is equal to the number of bits used to represent such a symbol. The problem of getting the probability value such that $b_i$ in equation (4) is an integer requires the $p_i$ in the equation (5) to be of the form:

$$P = 2^{-N}$$

where N represents the size of the dictionary or the total number of symbols [Schuegraf,1976]. A code of N bits is sufficient to represent a symbol. Thus, after selecting a suitable size for N, the task is to select $2^N$ symbols so that they will have constant probabilities of $2^{-N}$.

Selection procedures for the symbol—set satisfying the above condition have been well documented by [Lynch,1973] and [Schuegraf,1973]. The symbol set may include word or text fragments. Compression techniques using this scheme have been documented by [Schuegraf,1974]. Compression using this scheme is very attractive because we can choose the size of the dictionary to be of any convenient value and decoding is simply a table look up procedure; for small symbol sets the dictionary can also be kept in the memory. Because of all these features, compression and decompression using this scheme can be implemented by microprogramming or hardware and often, in such cases the speed of these operations is increased by a factor of 1000 [Lea,1976].

The above sections briefly discussed the schemes used for text compression. The next chapter discusses the exact methods to generate the symbol set, compression and decompression, and method to perform a search for a string in the compressed database.

# 4.0 ENCODING AND DECODING USING FIXED LENGTH REPRESENTATIONS OF VARIABLE LENGTH CHARACTER STRINGS

A compression scheme based on the variable length character strings with fixed length code representation is discussed in more detail in this chapter.

This scheme received a lot of attention because of it's fast and straight forward decoding procedure. Many approaches have been suggested to create a symbol set with variable length strings like words, word-pairs, phrases, word segments, text fragments etc. In a compression scheme, proposed by [Thiel, 1972], words in a database were replaced by codes. Since the frequency distribution of words in a text or a database follow Zipf's law, and because of the large number of words, efficient coding techniques require the use of variable length frequency dependent codes for words. Word coding schemes can not have the convenience of having a 'universal' symbol set or dictionary, because of its dependency on the size and the contents of the database. Often, the dictionary in a word coding scheme is quite large.

Thus, even though a word forms a natural language unit, efficient compression can not be achieved with a symbol set made solely of words or phrases. Since, only variable length codes can possibly map such variable length strings efficiently, processing of such codes is complicated in machines with fixed word size.

[Schuegraf,1973] proposed the use of variable length character strings called text fragments, which form the language elements for compression coding. Unlike word fragments which are character strings that are contained entirely in words, text fragments are strings that are part of a more general textual record and may contain blanks or punctuation symbols. [Clare,1973] suggested that these text fragments should occur in the database with equal frequency, so that fixed length codes can be used. Selection procedures for such variable length, equifrequent character strings are documented by [Schuegraf,1973], [Lynch,1973], [Clare,1973]. Because of the use of a fixed length coding scheme, it is possible to have a fixed sized dictionary. This simplifies the operation of decoding, since decoding becomes a process of table look up operation with the code being the index. Also, unlike word coding schemes where a variable length string is mapped on to a variable length code, in fragment coding schemes variable length strings are mapped on to fixed length codes. Dictionaries in fragment coding schemes also tend to be 'universal', in the sense that they can be applied on wide variety of texts or databases in similar category [Lynch,1982]. Sections that follow describe the procedures for the creation of text fragments and the selection process to create the symbol table.

## 4.1 GENERATION OF TEXT FRAGMENTS

A text fragment of restricted length is called an n-gram, where n refers to the maximum length of the text fragment. For example, a set of n-grams may contain a text fragment, one to n characters long, and that occurs frequently enough in a text to justify it being considered a symbol in its own right in addition to the conventional symbols which comprise the text [Yannakoudakis,1982].

Creation of n-gram set involves the selection of all strings of different lengths. This analysis can be performed on the database itself or if the database is too large, on a sufficiently large sample. In the latter case, care should be taken so that the chosen sample is representative of the entire database.

Before proceeding to identify all the possible n-grams, one should fix the maximum length of an n-gram i.e., the value of n. Larger n would take a lot of processing time, while too short a code could ignore longer, but frequent character strings in the text. A value of eight would likely cover most of the text fragments. An 8-gram set can include all character strings of length one(unigram), two(bigram), three(trigram), four, five, six, seven, and eight.

A PASCAL program was developed to generate n-grams, with value of n equal to eight. The complete text was scanned in the windows of length eight. First window consists of the first eight characters. Second window consists of eight characters starting from the second character, and third window starts at third character and contains the eight characters, so on. The following are the different windows identified by the program for a string 'DATA_COMPRESSION".

```
DATA_COM
ATA_COMP
TA_COMPR
A_COMPRE
_COMPRES
COMPRESS
OMPRESSI
MPRESSIO
PRESSION
```

The number of possible n-grams is directly proportional to the size of the text. Efficient storage and access to n-grams is required during the creation process as their occurrence frequencies are required for symbol set generation. If the text file or the database is very large, the only possible approach would be to store them on a disk. This would involve searching for an n-gram on the disk. The disk should be organized so that a minimum number of comparisons are done. For this project, a different approach was used. A trie data structure was used to store all the possible n-grams. This was kept in main memory. The trie data structure was very efficient in identifying an n-gram and it lends itself to simple recursive programming. This approach was suggested by [Yannakoudakis,1982]. The schematic layout of this structure is presented in the appendix A.

As each window is identified, it was entered in the trie with the necessary frequency counters incremented. For example, for windows 'DATA_COM' and 'DATABASE', the following n-grams are possible:

| | |
|---|---|
| D | D |
| DA | DA |
| DAT | DAT |
| DATA | DATA |
| DATA_ | DATAB |
| DATA_C | DATABA |
| DATA_CO | DATABAS |
| DATA_COM | DATABASE |

With only the above windows, n-grams with the corresponding frequencies are as follows:

| | |
|---|---|
| D | 2 |
| DA | 2 |
| DAT | 2 |
| DATA | 2 |
| DATA_ | 1 |
| DATAB | 1 |
| DATABA | 1 |
| DATABAS | 1 |
| DATABASE | 1 |
| DATA_C | 1 |
| DATA_CO | 1 |
| DATA_COM | 1 |

This process is simple with the use of trie data structure. Once the complete representative text is scanned, the trie must contain all the possible n-grams with the corresponding frequency counters. Each node in the trie is associated with a fragment and its frequency. It is obvious that the frequency of a node also include the frequency of its child. A fragment of any node is obtained by concatenating it's character to the fragment of it's parent node. The following section describes the selection procedures for the creation of the symbol set.

### 4.1.1 CREATION OF A SYMBOL SET

Once the trie is constructed, it contains all the possible n-grams and the possible candidates for the symbol set. The elements in the symbol set form the language elements which will be used for encoding and decoding processes. The task is to identify equifrequent n-grams, so that we can assign fixed length codes to them. The procedures for the selection of symbols from these n-grams are presented in [Lynch,1973], [Yannakoudakis,1982], [Schuegraf,1973].

[Schuegraf,1973] suggested that the following guidelines should be borne in mind during the formulation of any selection of symbols as the basic language elements:

- The set of dictionary fragments must be complete in the sense that the database or a text file should be representable by concatenation of dictionary elements.
- The selected fragments should occur with equal frequency.
- The set of fragments shoud be chosen to maximize the average fragment length.
- There should be very few words of the uncoded database that do not contain at least one bigram or larger fragment.
- The set of fragments should not be over redundant. That is, it should not be possible to choose a smaller set that also satisfies the above requirements.

Even though the above criteria for fragment selection are meant originally for word fragment selection, they can equally be applied to the selection of text fragments.

The fragment selection algorithm starts with the basic symbol set and then keeps adding the most frequent bigrams whose frequencies are above a predetermined threshold. This process continues until the frequency of the most frequent trigram is equal to or greater than the frequency of the bigram under consideration. Then we start considering the trigrams. This process continues until either we reach a predefined dictionary size or there are no more n-grams to consider.

The following is the fragment selection algorithm as suggested by [Yannakoudakis,1982].

1. Specify a threshold t.
2. Select a basic symbol set which includes all the basic characters.
3. get next n-gram, if its frequency exceeds the threshold then perform step 4 else perform step 5 for this n-gram.
4. Add the n-gram to the symbol set only if the difference between frequencies of (n-1)-gram and n-gram exceeds the threshold. If an n-gram is added to the symbol set then the frequency of (n-1) gram is reduced by that of n-gram. Go to step 3.
5. If the size of the resultant symbol set is not the required size, then increase the threshold if the size is greater else decrease and go to step 2.

The selection algorithm is a function of threshold t, the minimum frequency at which a fragment must be considered as a possible candidate for inclusion in the dictionary. The size of the resulting dictionary is also a function of this threshold.

For this project, the symbol set generated by [Lynch,1981] was used. This symbol set was modified to remove some of the special characters; the special characters were coded using a shift character. This allows for some extra n-grams to be included in the symbol set at the expense of a longer code for the special characters. Appendix B shows the symbol set used in this project.

## 4.2 CODING ALGORITHMS USING VARIABLE LENGTH STRINGS-FIXED LENGTH CODES

This section is concerned with the general problem of database compression once the selection of the set of equifrequent symbols is completed. The dictionary is expected to have a certain number of symbols of a maximum length n. In this project, the size of the dictionary was set to 256 and the maximum length of a symbol was eight. The 256 symbols can be coded using an eight bit code. The dictionary included all the basic characters to ensure that a complete representation of the database is possible with the dictionary elements, though it is expected that it is not often that we have to use single charcters in the coded database. With an eight bit code, even the use of single characters does not require more space than the original database in machines with eight bit character representatins. The problems encountered in basic compression and possible compression methods are described in this section.

The selection procedures for symbol set presume to have a knowledge of the database or a representative sample. In contrast, the coding procedures can not be expected to have knowledge of the complete text or the complete database. Instead they only look ahead a short sequence of text characters. This restriction is important to control the processing time spent during compression. The symbol set selection procedures must have complete knowledge, because a symbol set is designed to be static over a period of time and has to be reasonably 'universal' so that the symbol set can be used on several texts.

A knowledge of the degree of optimality of the stored dictionary fragments is useful since it provides a criterion with which to compare the result of subsequently coding the database by use of a particular coding algorithm. The problem is to choose the coding algorithm so that in the resulting fragmented database the fragment distribution is sufficiently close to having the same measure of equifrequency that was obtained by the optimal selection process used to create the fragments stored in the dictionary [Schuegraf,1973].

The selection criteria for this comparison of different coding alogorithms are mainly the resulting compression factor and processing time needed. The methods covered are:

1. Minimum Space Method (MS)
2. Longest Fragment First Method (LFF)
3. Longest Match method (LM)

## 4.2.1 MINIMUM SPACE METHOD

As the name implies, this method produces the compressed database of minimum length. A database of minimum length will take up minimum storage.

For a given string, this method requires the identification of all text fragments that are sub-strings of the given string i.e., the codes whose strings are entirely contained with in the given string. Then, generate all the combinations of codes that form the given string. Now the task is to choose the combination of codes which has the least number of codes.

This method has the advantage of producing the best compression factor at the price of increased processing time. This is essentially an exhaustive search of all the possible sequences of codes that could fit within a string. This requires that the dictionary be organized so that the identification of fragments is faster. Another major disadvantage of the method is that substring searching can only be carried out on decompressed strings.

## 4.2.2 LONGEST FRAGMENT FIRST METHOD

Though the Minimum Space method guarantees the best compression of the database fragments in the resulting database will not be equifrequent.

Longest Fragment method is similar to that of Minimum Space method in that all the possible codes whose text fragments are entirely contained within the given string be found and stored before the selection of the combination is found. Once all the possible codes are found, this method finds recursively the largest fragment and removes all the other codes whose strings are either completely contained with in the largest fragment or overlap this chosen fragment. Once this process is completed, the codes that still remain will form the combination of codes that form the given string. In general, the resulting compression factor due to Longest Fragment method is lower than that of Minimum Space method.

It has the same disadvantage of Minimum Space method in that all the codes of text fragments whose string are contained in the given string be found and stored in advance. Hence, this method also requires an efficient organization of the dictionary. In addition, substring searches have to be carried out on decompressed databases.

## 4.2.3 LONGEST MATCH METHOD

Both Minimum Space method and the Longest Fragment First suffer from the same disadvantage of having to find all subfragments of the given string and, hence would require additional storage and processing time.

We select the longest fragment that matches the characters in the string starting at the first character , and substitute its code for the string. Starting from the next character not included in the first string, we repeat the process until the end of the text is reached.

Since there is no need to look for all fragments, the process allows considerable saving of coding time. There is no need for any additional table to store intermediate codes.

[Schuegraf,1973] has surveyed different compression algorithms and presented the experimental results.

For the project, the Longest Match method was chosen. The next section will describe the program design to implement compression and decompression procedures.

## 4.3 PROGRAM DESIGN FOR COMPRESSION AND DECOMPRESSION

As a part of the project, programs were developed to generate the symbol set, and perform compression and decompression.

As was mentioned earlier, variable length text fragments of restricted length were used. The programs were written to be independent of the size and symbols of the dictionary. For the purpose of the testing, the size of the dictionary was set to be 256 so that a code of length eight bits or a byte can be assigned to each symbol. The method used to generate symbols was described in section 4.1. It used a trie data structure as shown in the appendix A to store all the possible n-grams along with the frequency counts. Then the most equiprobable symbols were identified according to the method described in section 4.1.1.

The dictionary contained all the basic characters, so that representation of a database is always possible, though it is expected that the use of symbols of length of one character are not preferred in the compression algorithm. Only upper case letters were included in the dictionary, yet programs were developed to handle multiple case. For testing purposes, it was decided to use the dictionay proposed by [Lynch,1973]. The dictionary was modified to include all the possible characters with all the special punctuation marks deleted to include more text fragments in the dictionary. A shift character '@' is used to precede any special character. This assumes that the database will mostly contain alphabetical data. All the punctuation symbols and the digits are stored in a separate linear structure and codes assigned according to their position. The modified dictionary is shown in Appendix B.

The symbols were read from a file and stored in memory in a trie data structure similar to that shown in the appendix A. This allowed the faster and simpler identification of the longest fragment [Yannakoudakis,1982].

### 4.3.1 COMPRESSION PHASE

The longest match method as described in section 4.2.3 was used to perform the compression of the database. Though it does not provide the compressed database of minimum length, it is simpler and faster in terms of processing time.

If the number of consecutive blanks is more than two, then only the combination of a shift character, position of blank in the table for special characters, and the count is kept. This allows the original data base to be created from the coded representations exactly.

The organization of the compressed database is designed to allow for faster string search procedure on the compressed database, and of course, for faster decompression. But this design in no way can be claimed to be optimal. As was mentioned in chapter I, the main objective of this project was to find ways to improve string searching on larger text files or databases.

The compressed database is organized in to pages. Each page contains two parts: A position matrix and an overflow table. The position matrix is organized essentially in the form of a table, with 256 rows and each row containing a fixed number of position slots. Each row indicates the positions of the corresponding code in the compressed database. The overflow area is to store the positions of codes for which their respective rows have been filled up. The overflow area is also a table where each entry contains the code, and position. Appendix C shows the organization of a page.

As each code is identified by the compression procedure, a separate procedure stores the position of the code in the page. If the row of the code is filled up, then the overflow area is used. In case the overflow area is also filled up, a new page will be started. The page is organized in a matrix form because of the fact for a good symbol set and for a good compression process, the resulting coded representations must occur with equal frequency. Because of this observation, it was assumed that a matrix form is alright. But experience has shown that a typical page is very sparse and often only a few codes occur more often than others and they fill up the overflow area. Though this organization allows for string search, there is a room for improvement.

## 4.3.2 DECOMPRESSION PHASE

In schemes where variable length language elements are coded using fixed length code representa[illegible]e decompression procedure is essentially a table look up procedure with code being the index.

That is essentially true in this current implementation, except for the fact that codes are not stored in a linear structure. Since, they are stored in pages, as described earlier, preprocessing is needed for each page. This preprocessing essentially involves the mapping each code position in a linear structure to obtain a single stream of codes for that page. After this process, each code is indexed into the dictionary to obtain the corresponding text fragment. Concatenation of all the text fragments of all consecutive codes provides the original file. Some logic is needed to process the shift codes and repeat counts.

The detailed statistics regarding the compression ratio, CPU timings, frequency distributions of codes in a page for a test case are presented in the following section.

## 4.4 STATISTICS

The following sections describe the results of the compression technique used. The symbol set shown in appendix A is used. All the results shown are based on IBM 3084 computer system (running VM/SP CMS operating system).

To illustrate the technique, An extract from Newsweek magazie is used as a text file to be compressed. All the following results are based on this text. The article is as follows:

FOR WEEKS IT HAD SEEMED CLEAR
THAT SOMEONE WAS ACCUMULATING
STOCK IN PAN AM CORP. THE
SHARES OF THE AIRLINE COMPANY
WERE REGULARLY ON THE MOST
ACTIVE LIST AND THE PRICE HAD
BEEN NUDGED UP. LAST WEEK THE
NAME OF AT LEAST ONCE OF THE
BUYERS WAS ANNOUNCED. RESORTS
INTERNATIONAL, WHICH RUNS
HOTELS AND CASINOS, SAID IT
BOUGHT WHAT AMOUNTS TO ABOUT
7 PERCENT OF THE AIRLINE'S STOCK.

An extract from 'An Old Bidder's New Interest In Pan Am', taken from Newsweek magazine (august 26, 1985; pp.46,-)

The following is the frequency distribution of characters in the text file used:

```
'A'=  30  'B'=   4  'C'=  14  'D'=   9  'E'=  38
'F'=   5  'G'=   4  'H'=  16  'I'=  16  'J'=   0
'K'=   4  'L'=  11  'M'=   8  'N'=  24  'O'=  23
'P'=   6  'Q'=   0  'R'=  16  'S'=  23  'T'=  30
'U'=  11  'V'=   1  'W'=   7  'X'=   0  'Y'=   3
'Z'=   0

'0'=   0  '1'=   0  '2'=   0  '3'=   0  '4'=   0
'5'=   0  '6'=   0  '7'=   1  '8'=   0  '9'=   0

' '= 131  '¢'=   0  '.'=   4  '<'=   0  '('=   0
'+'=   0  '|'=   0  '&'=   0  '!'=   0  '$'=   0
'*'=   0  ')'=   0  ';'=   0  '¬'=   0  '—'=   0
'/'=   0  'ù'=   0  ','=   2  '%'=   0  '-'=   0
'>'=   0  '?'=   0  ' '=   0  ':'=   0  '#'=   0
'@'=   0  '''=   1  '='=   0  '"'=   0  '°'=   0
'['=   0  ']'=   0  '\'=   0


text file size =      455  characters
       letters =      303  ==>65.93%
        digits =        1  ==>0.22%
        blanks =      131  ==>29.45%
   punctuation =        7  ==>1.54%
   no of lines =       13  ==>2.86%
```

Input File Statistics
Character Distribution

The size of the compressed text file, the number of pages in the partitioned compressed text file, compression factor, and the time taken to perform the compression are given below:

```
Coded file is      210 bytes long.
Number of pages in coded file is  1.
Coded file is    46.15 percent of original file.
Time taken for compression is    14022 micro seconds.
```

The following is the decompressed text and the time taken for the operation.

```
        Decompression is in progress.
        The following is the decompressed text for the first page.


  ( FO)(R )(WE)(E)(K)(S I)(T )(HA)(D )(SE)(EM)(ED )(C)
  (LE)(AR)(TH)(AT)( S)(OM)(E)(ON)(E )(W)(AS)( A)(C)(C)(U)
  (M)(U)(LA)(TI)(NG)(ST)(OC)(K )(IN )(PA)(N A)(M )(CO)(R)
  (P)(.)( )( TH)(E)(SH)(AR)(ES )(OF)( TH)(E A)(IR)(LI)
  (NE)( CO)(MP)(AN)(Y)(WE)(RE)( RE)(G)(U)(LA)(R)(LY )
  ( L)(IST)( AND)( TH)(E P)(RI)(CE)( H)(AD)(BE)(EN)( N)
  (U)(D)(GE)(D )(U)(P)(.)( )( L)(AS)(T )(WE)(E)(K )(THE)
  (NA)(ME)( OF )(AT)( L)(EA)(ST)( O)(NC)(E O)(F T)(HE)
  (B)(U)(Y)(ERS)( W)(AS)( AN)(NO)(UN)(CE)(D)(.)( RE)(SO)
  (RT)(S)(IN)(TER)(NA)(TIO)(NAL)(,)( W)(HI)(CH)( R)(UN)
  (S)(HO)(TE)(L)(S A)(ND T)(CA)(SI)(NO)(S)(,)( S)(AT)
  (D )(IT)(BO)(U)(G)(H)(T )(W)(HA)(T )(AM)(OU)(NT)(S )
  (TO )(A)(BO)(UT)(7)( P)(ER)(CE)(NT )(OF)( TH)(E A)(IR)
  (LI)(NE)(')(S )(ST)(OC)(K)(.)


       Time taken for decoding is    21696 micro seconds.
```

The programs were run on different kinds of text files. All the programs were written in PASCAL on IBM 3084 computer system using SP/CMS. All the text files are on this machine. The compression factors ranged from 30.57 percent when this report was used as a text file to 67.87 percent on a computer program written in PASCAL language. The characteristics of the text files were also obtained. The following gives the description of the text files used.

1. <u>This Report.</u> This report in its original form with the typeset commands is used as one of the test data. This report is formatted using the Document Composition Facility (DCF).

2. A computer program. Programs written for this project to perform compression, decompression and string search are used. These programs are developed using PASCAL.

3. A magazine Article. This article entitled "Hail, Halley's !" is taken from NEWSWEEK magazine dated september 9, 1985.

4. A computer manual. This is a reference manual for a database management system based on Generalized Entity Relationship Model (GERM). This system was developed by Bell-Northern Research, Ottawa.

The following shows the results obtained for each of these text files.

1. This Report.

```
text file size =    255360  characters
       letters =     54955  ==>21.52%
        digits =      1856  ==>0.73%
        blanks =    190928  ==>74.77%
   punctuation =      5701  ==>2.23%
   no of lines =      1920  ==>0.75%

Coded file is    75026 bytes long.
Number of pages in coded file is 22.
Coded file is    29.40 percent of original file.
Time taken for compression is  3643425 micro seconds.

Time taken for decoding is  1473445 micro seconds.
```

2. A computer program.

```
text file size =    60761   characters
       letters =    18999   ==>31.27%
        digits =      727   ==>1.20%
        blanks =    31401   ==>51.68%
   punctuation =     7310   ==>12.03%
   no of lines =     2324   ==>3.82%


Coded file is    40661 bytes long.
Number of pages in coded file is 21.
Coded file is    66.92 percent of original file.
Time taken for compression is  1625702 micro seconds.

Time taken for decoding is  927830 micro seconds.
```

3. A magazine article.

```
text file size =    11855   characters
       letters =     8276   ==>69.81%
        digits =      295   ==>2.49%
        blanks =     2761   ==>23.29%
   punctuation =      309   ==>2.61%
   no of lines =      214   ==>1.81%


Coded file is     4115 bytes long.
Number of pages in coded file is  3.
Coded file is    34.71 percent of original file.
Time taken for compression is   263234 micro seconds.

Time taken for decoding is   286017 micro seconds.
```

4. A computer manual.

```
text file size =   370144  characters
       letters =   210125  ==>56.77%
        digits =    10262  ==>2.77%
        blanks =   117075  ==>31.63%
   punctuation =    21600  ==>5.84%
   no of lines =    11082  ==>2.99%


Coded file is   173561 bytes long.
Number of pages in coded file is 99.
Coded file is    46.89 percent of original file.
Time taken for compression is  9889963 micro seconds.

Time taken for decoding is  5838217 micro seconds.
```

The compression factor averages around 50 percent. For some text files like computer programs, the compression factor is not as high as, say English prose texts. This is partly because of the presence of a significant number of punctuation marks in computer programs. With the symbol set used, special characters are coded using a shift character.

The following is the tabulation of the results.

| Text file | Number of characters in the original file | Number of codes in the compressed file | Compression factor |
|---|---|---|---|
| This report | 256424 | 74337 | 29.40% |
| Computer program | 60315 | 40938 | 66.92% |
| Magazine article | 11855 | 4115 | 34.71% |
| Computer manual | 370144 | 173561 | 46.89% |

The next section deals with the string search on a compressed text file.

## 5.0 STRING SEARCH ON COMPRESSED DATABASE

Compression schemes reduce the storage requirement by replacing the original text by codes. The incresed access to on-line databases and the information retrieval systems resulted in the great growth in the volume of data managed by these systems. As larger files are becoming common, it is almost essential to provide ways to represent the same information in a compact form. This is achieved through compression techniques described in the previous sections. An additional problem due to the 'information explosion', is that the traditional techniques for searching on files are no longer providing a satisfactory response time. Mostly, these techniques employ character by character comparison on the text file. Some information retrieval systems use inversion at word level to speed up the search process.

Compression techniques which are becoming practical and useful can also be applied so that string search is speeded up. Since the compressed database is smaller compared to the original database, even a linear search on a compressed data base must yield better results. Time taken for decompression is often negligible in compression schemes where variable length language elements are mapped on to fixed length codes. [Goyal,1983] suggested that by a proper file organization for the compressed database, we can indeed, speed up the search.

As was mentioned earlier, the main objective of the project was to find the feasibility of string search on compressed databases. This process is seen to be very significant, because even if the advances in the hardware can keep pace with the storage devices, the traditional string search methods would take significant amounts of time to locate a string in extremely large files, unless special file organizations are adopted.

Unlike some compression schemes where decompression needs to be performed before string search, [Goyal,1983] suggested a method where string search can be carried out directly on the compressed database. This requires a different organization for the compressed database.

Essentially in this method, the coded file is divided in to pages. Each page is organized as specified in the previous section. Thus, in this organization, we assume that the frequency distribution of codes in the compressed database is rectangular i.e., they are equifrequent. It requires the inversion at the symbol level.

Searching for a string on compressed database requires the encoding of the search string using the same symbol table that was used to compress the database. It is quite possible that the string may not have the same combination of codes in the database. This is because of the use of the text fragments; this is not the case for word coding schemes. In the case of fragment coding, the beginnings of the search string and the tail ends may be combined with the preceeding and the subsequent text. Because of this it is necessary to identify all possible code patterns that have fragment endings that correspond to the beginning of the search string. Similarly, we need to identify all code patterns that begin with the tail portion of the search string. Hence, there could be many code patterns that need to be searched for in the compressed database. The following algorithm tries to locate the string in the compressed database with a minimum number of searches.

The algorithm for a string search involves the following steps:

1. Encode the string to be searched.
2. Read a page. If no more pages in the comresse‌d database then return with a message that string is not found.
3. Search for the existance of the code pattern in the compressed database.
4. If found return else record the position where mis-match occurred.
5. If mis-match occurred in the head portion of the code pattern then find the next code that has fragment ending that matches the start search string. Go to step 3.
6. If mis-match occurred in the last code then find the next possible code that has a fragment begining

that matches the ending of the the search string. Go to step 3.

7. Go to step 1.

The above algorithm finds if the string to be searched exists in the compressed database. If it is found, it returns the position of the first code in the code pattern. The following algorithm displays the context in which the string appears for the specified page.

1. For each code(i,j) where code(i,j) indicates the jth position of ith code, set the list(j) to i, where list is an linear array containing all the code representations of the original database. This is essentially a process of converting the page which uses the inversion at the word level to a structure where inversion at the position level is used.

2. Using the position as returned by the previous algorithm, back track this linear structure by a specified number of places.

3. Using code as an index in to the dictionary, display the text by concatenating all the text fragments for the codes.

The detailed statistics regarding the CPU timings and the different code patterns generated for a test strings are given in the next section.

## 5.1 STATISTICS

This section illustrates the search procedure. For this purpose, the same text file as used in section 4.4 is used. Using this file a compressed file is created and partitioned into pages.

At first the search string is encoded using the same symbol set as used for encoding the original text file. Then all the possible 'front' codes and 'end' codes are generated. Then each such coding sequence is searched for in the compressed text file. As soon as any two consecutive codes are not matched, the search operation for that particular code sequence is aborted, and search operation for the subsequent code pattern is initiated. The number of comparisons needed to make before the target string is found is equal to number of times pairs of codes are compared.

The following shows the different comparisons performed before the target string is found.

```
        Interested in searching ? (y/n)
        y
        Enter string:
        Pan Am
        string=PAN AM, length= 6


    ====( P) (AN ) (AM)
             ( P):  191,
            (AN ):  —— NOT FOUND.

    ====(E P) (AN ) (AM)
            (E P):   85,
            (AN ):  —— NOT FOUND.

    ====(MP) (AN ) (AM)
             (MP):   61,
            (AN ):  —— NOT FOUND.

    ====(OP) (AN ) (AM)
             (OP):
            (AN ):  —— NOT FOUND.

    ====(P) (AN ) (AM)
              (P):   44,  99,
            (AN ):  —— NOT FOUND.

    ====(PA) (N A) (M)
             (PA):   39,
            (N A):   40, —— FOUND.
              (M):   29, —— NOT FOUND.

    ====(PA) (N A) (M )
            (N A):   40,
             (M ):   41, —— FOUND.

        Line containing the string:

    STOCK IN PAN AM CORP.  THE

        Time taken for search is    17458 micro seconds.
        The number of comparisons is 8.

         NEXT STRING ? (Y/N)
         N
```

The following shows the number of comparisons done for different strings:

```
•    string=AIRLINE, length= 7


          Line containing the string:

     SHARES OF THE AIRLINE COMPANY

          Time taken for search is    18484 micro seconds.
          The number of comparisons is   8.



•    string=HOTELS, length= 6


          LINE CONTAINING THE STRING:

     HOTELS AND TCASINOS, SAID IT

          Time taken for search is    20226 micro seconds.
          The number of comparisons is  10.



•    string=RESORTS, length= 7


          LINE CONTAINING THE STRING:

     BUYERS WAS ANNOUNCED. RESORTS

          Time taken for search is    18694 micro seconds.
          The number of comparisons is   4.


          string=ANNOUNCED, length= 9

          Line containing the string:

     BUYERS WAS ANNOUNCED. RESORTS

          Time taken for search is    19145 micro seconds.
          The number of comparisons is   5.
```

The following shows the results of the search procedure when applied to different text files for a set of strings. The text files used are the same ones as described in the section 4.4. The following strings are used:

| String | String index |
|---|---|
| 'Purpose of the testing' | 1 |
| 'Purpose of the test' | 2 |
| 'pose of the testing' | 3 |
| 'pose of the test' | 4 |
| 'dictionary' | 5 |
| 'diction' | 6 |
| 'tionary' | 7 |
| 'ictio' | 8 |

The following is the tabulation of the results obtained.

| Text file | String index | Number of possible code patterns | Number of comparisons performed | time taken in microsecs | found (y/n) |
|---|---|---|---|---|---|
| This report | 1 | 6 | 2170 | 334609 | y |
| | 2 | 6 | 2162 | 327941 | y |
| | 3 | 8 | 7342 | 475981 | y |
| | 4 | 8 | 7334 | 468992 | y |
| | 5 | 11 | 3969 | 62064 | y |
| | 6 | 198 | 523 | 145067 | y |
| | 7 | 27 | 701 | 59136 | y |
| | 8 | 17 | 265 | 48729 | y |
| Computer program | 1 | 6 | 221 | 596691 | n |
| | 2 | 6 | 221 | 593068 | n |
| | 3 | 8 | 7259 | 888585 | n |
| | 4 | 8 | 7259 | 883419 | n |
| | 5 | 11 | 1166 | 1077402 | n |
| | 6 | 198 | 1166 | 1116286 | n |
| | 7 | 27 | 199 | 2076640 | n |
| | 8 | 17 | 50 | 65190 | y |
| Magazine article | 1 | 6 | 998 | 65517 | n |
| | 2 | 6 | 998 | 63634 | n |
| | 3 | 8 | 1584 | 91015 | n |
| | 4 | 8 | 1584 | 90401 | n |
| | 5 | 11 | 466 | 97388 | n |
| | 6 | 198 | 466 | 98861 | n |
| | 7 | 27 | 2212 | 205223 | n |
| | 8 | 17 | 778 | 74185 | y |
| Computer manual | 1 | 6 | 14148 | 3086005 | n |
| | 2 | 6 | 14148 | 3045581 | n |
| | 3 | 8 | 47191 | 4317806 | n |
| | 4 | 8 | 47191 | 4247738 | n |
| | 5 | 11 | 143 | 250748 | y |
| | 6 | 198 | 177 | 322689 | y |
| | 7 | 27 | 428 | 460770 | y |
| | 8 | 17 | 52 | 62133 | y |

Thus, string search on a compressed databases is possible. However, the search procedure can be improved. A different approach can be given as follows:

1. Encode the string.

2. Search for this pattern. If string is found then return('found').

3. Get the stem of the string by trimming the first and last characters. Encode the stem.

4. Search for this pattern. If found then record the position of the first code in the pattern, say p1, and last code of the pattern, say p2, else return('not found').

5. For every possible n-gram that has an ending which could be the beginning for the search string, find if that n-gram is present at (p1-1). if present proceed to next step else check next n-gram.

6. If front code is present, now for every possible n-gram that has the beginning which could be the ending for the search string i.e., first character of n-gram is also the last character of the search string, find if that n-gram is present at position (p2+1). If present return('found') else check next n-gram.

The above alogorithm tries to minimize the number of comparisons done. It is expected that it will take minimum amount of processing time if string is not found.

## 6.0 CONCLUSIONS

Data compression techniques reduce the storage and transmission costs. In this report, we outlined the need for data compression and surveyed the different techniques which use fragment dictionary; and different methods of fragment compression which use fixed length codes to represent variable length character strings are discussed. The main advantage of fragment compression is that it allows fixed length codes and an in—core dictionary. Because a fragment dictionary can be stored in memory, the process of decoding becomes simple and fast.

As the use of larger files is becoming common, it is necessary to have more efficient algorithms for string search. In this project, the possiblility of performing search operations on the comrpessed database using partitioned database scheme is explored.

As the results of the experiment show that on machines with eight bit character representation, a compression factor of about 50 per cent can be achieved. It is also shown that a string search on a compressed database is possible.

It can be noted that the concept of having a partitioned database is indeed helpful for search operations. But this organization tends to lower effective compression factor. Determination of a set of equifrequent symbol set is almost a prerequisite for such on organization to be more helpful.

Even though most compression techniques aim at reducing the storage requirements, these techniques have not become very popular. This is because of the rapid developments in the hardware that have taken place in the last decade, making it rather unnecessary to push the existing technology to its limits. Indeed, this process is expected to continue. But as more and more users become aware of on-line databases due to the development of computer based information systems, it is expected that technological advances can not keep the same pace. Even if devices with great storage capacities do appear in the near future, the traditional search methods have to be changed. Compression techniques based on partitioned database indeed attempt to solve those problems [Cooper,1982].

## 6.1 FUTURE WORK

Most of the compression schemes require a dictionary with equifrequent symbols. Determination of such a dictionary is a necessity for an effective compression operation and would be highly favourable for search operations on compressed databases.

To be useful, a dictionary must be stable and representative over longer periods. [Lynch,1973] examined the frequencies of variable character strings on INSPEC database for a period of three years and found that there was no radical change in the frequencies of character strings. Indeed, such studies are important. We still need to find if it is possible to have a true 'universal' dictionary, which can be used on a wide variety of texts.

Much work has to be done in the determination of a good dictionary with equifrequent dictionaries. Also, work needs to be done to have good compression methods which can produce a compressed database where the coded representations also exhibit the equifrequency property.

Until a reasonable dictionary is found, it is possible to store the compressed dictionary in a linear structure in secondary storage and can be organized in memory with inversion at symbol level at the search times. This would indeed improve the compression factor at the price of an incresed processing time for search operations.

## 7.0 BIBLIOGRAPHY

1. [Barton, I.J. et al.,1973] 'Variable-length character string analysis of three data-bases, and their application for file compression,' Proceedings of a conference held by the Aslib co-ordinate indexing group on April 11-13, 1973 at Durham University.

2. [Barton, I.J. et al.,1974] 'An information theoretic approach to text searching in direct access systems,' CACM, Vol.17, No.6, pp.345-350.

3. [Chen T.C.,1975] 'Storage-efficient representation of decimal data,' CACM, Vol.18, No.1, pp.49-52.

4. [Clare A.C. et al.,1972] 'The identification of variable-length, equifrequent character strings in a natural language database,' The computer journal, Vol.15, No.3, pp.259-262.

5. [Cooper D. et al.,1980] 'Compression of continuous prose texts using variety generation,' Journal of American Society for Information Science.

6. [Cooper D., et al.,1982] 'Text compression using variable to fixed length encodings,' Journal of American society for information science,January, pp.18-31.

7. [Goyal P.,1983] 'Coding methods for text string search on compressed databases,' Inform. Systems, Vol.8, No.3, pp.231-233.

8. [Hahn B.,1974] 'A new technique for compression and storage of data,' CACM, Vol.17, No.8, pp.434-436.

9. [Heaps H.S.,1972] 'Storage analysis of a compression coding for document databases,' Infor, Vol.10, No.1, pp.47-61.

10. [Lynch M.F.,1973] 'Compression of bibliographic files using an adaptation of run-length coding,' Inform. Stor. Retr., Vol.9, pp.207-214.

11. [Lynch M.F. et al,1973] 'Analysis of the microstructure of titles in the INSPEC database,' Inform. Stor. Retr. Vol.9, pp.331-337.

12. [Lynch M.F.,1975] 'Creation of bibliographic search codes for hash addressing using the variety-generator method,' Program, Vol.9, No.2 pp. 46-55.

13. [Lynch M.F.,1977] 'Variety generation—A reinterpretation of Shannon's mathematical theory of communication, and its implications for information science,' Journal of the American Society for Information Science, January, pp.19-25.

14. [Lynch M.F.,1980] 'Sorting of textual databases: A variety generation approach to distribution sorting,' Inform. Processing & Management, Vol.16, pp.49-56.

15. [Rubin F.,1976] 'Experiments in text file compression,' CACM, Vol.19, No.11, pp.617-623.

16. [Schuegraf E.J. et al.,1973] 'Selection of equifrequent word fragments for information retrival,' Inform. Stor. Retr., Vol.9, pp.697-711.

17. [Schuegraf E.J. et al.,1974] 'A comparison of algorithms for database compression by use of fragments as language elements,' Inform. Stor. Retr., Vol.10, pp.309-319.

18. [Schuegraf E.J.,1976] 'A survey of data compression methods for non-numeric records,' Canadian journal of information science, Vol.2, No.1, pp.93-105.

19. [Schuegraf E.J.,1976] 'Compression of large inverted files with hyperbolic term distribution,' Information processing and management, Vol.12, pp.337-384.

20. [Severance D.G.,1983] 'A practitioner's guide to database compression,' Inform. Systems, Vol.8, No.1, pp.51-62.

21. [Smit G.DE V.,1982] 'A comparison of three string matching algorithms,' Software-practice and experience, Vol.12, pp.57-66.

22. [Storer J.A.,1982] 'Data compression via textual substitution,' JACM, Vol.29, No.4, pp.928-951.

23. [Suen C.Y.,1979] 'n-Gram statistics for natural language understanding and text processing,' IEEE Transactions on pattern analysis and machine intelligence, Vol.PAMI-1, No.2, pp.164-172.

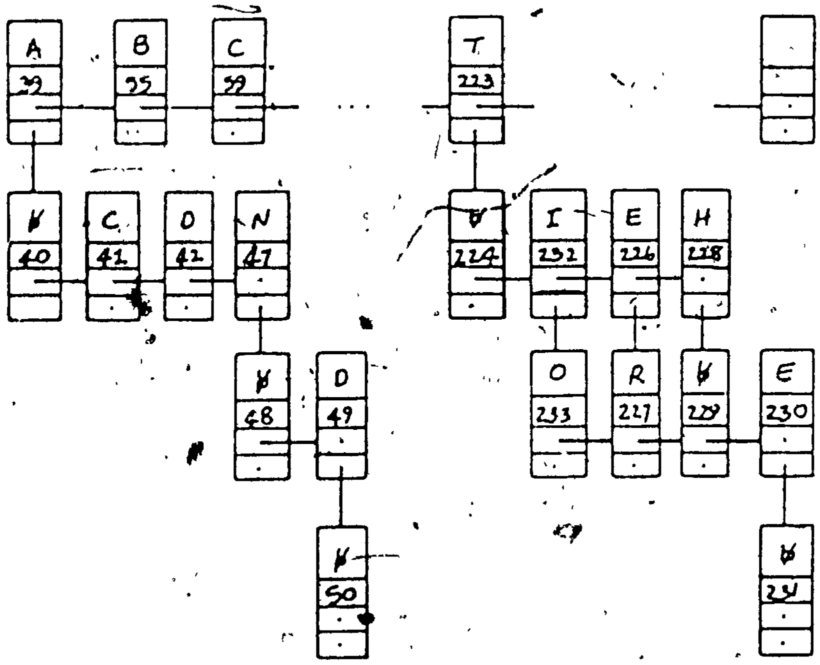
24. [Thiel L.H. et al.,1972] 'Program design for retrospective searches on large databases,' Information Storage and Retrieval, Vol.8, pp.1-20.

25. [Wagner R.A.,1973] 'Common phrases and minimum-space text storage,' CACM, Vol.16, No.3, pp.148-152.

26. [Yannakoudakis, E. J. et al.,1982] 'The generation and use of text fragments for data compression,' Information processing and management, Vol.18, No.1, pp.15-21.

## 8.0 APPENDIX A

### TRIE DATA STRUCTURE



N-grams shown in this diagram are:

A, A_, AC, AD, AN, AN_, AND, AND_,
T, [illegible], TI, TIO, TE, TER, TH, TH_, THE, THE_

where '_' stands for a blank.

## 9.0 APPENDIX B

### FRAGMENT DICTIONARY

Current Dictionary Size is 256.

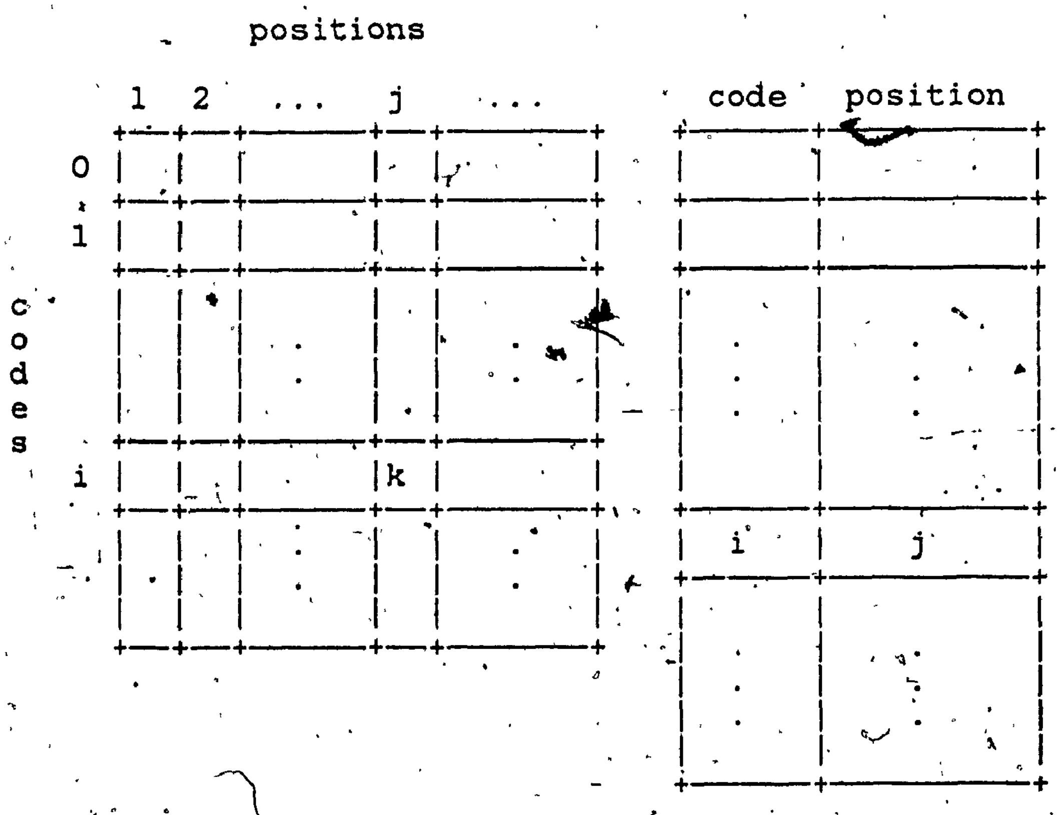
All the special characters like punctuation symbols and digits are stored in a separate dictionary. A shift character '@' is preceeded for all the characters in this separate dictionary.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | ( ) | ( A) | ( AN) | ( AND) | ( B) | ( C) | ( CH) | ( CO) | ( D) | ( DE) |
| 1 | ( E) | ( F) | ( FO) | ( G) | ( H) | ( I) | ( IN) | ( IN ) | ( L) | ( M) |
| 2 | ( MA) | ( N) | ( O) | ( OF) | ( OF ) | ( P) | ( PO) | ( PR) | ( R) | ( RE) |
| 3 | ( S) | ( SE) | ( ST) | ( T) | ( TH) | ( TO) | ( W) | (@) | (°) | (A) |
| 4 | (A ) | (AC) | (AD) | (AG) | (AI) | (AL) | (AM) | (AN) | (AN ) | (AND) |
| 5 | (AND ) | (AR) | (AS) | (AT) | (ATI) | (B) | (BE) | (BO) | (BR) | (C) |
| 6 | (C ) | (CA) | (CE) | (CH) | (CI) | (CO) | (COM) | (CON) | (CT) | (D) |
| 7 | (D ) | (D T) | (DE) | (DI) | (DU) | (E) | (E ) | (E A) | (E C) | (E O) |
| 8 | (E P) | (E S) | (E T) | (EA) | (EC) | (ED) | (ED ) | (EE) | (EL) | (EM) |
| 9 | (EN) | (ENG) | (ENT) | (ENT ) | (ER) | (ER | ERN) | (ERS) | (ES) | (ES ) |
| 10 | (ET) | (F) | (F ) | (F T) | (FI) | (FO) | (G) | (G ) | (GE) | (GR) |
| 11 | (H) | (H ) | (HA) | (HE) | (HE ) | (HI) | (HO) | (I) | (IA) | (IC) |
| 12 | (IC) | (ICA) | (ID) | (IE) | (IL) | (IN) | (IN ) | (ING) | (IO) | (IR) |
| 13 | (IS) | (IST) | (IT) | (J) | (K) | (K ) | (L) | (L ) | (LA) | (LAN) |
| 14 | (LE) | (LI) | (LL) | (LO) | (LY ) | (M) | (M ) | (MA) | (ME) | (MEN) |
| 15 | (MI) | (MO) | (MP) | (N) | (N ) | (N A | (N T) | (NA) | (NAL) | (NC) |
| 16 | (ND) | (ND T) | (NE) | (NG) | (NG ) | (NI) | (NO) | (NS) | (NS) | (NT) |
| 17 | (NT ) | (O) | (O ) | (OC) | (OD) | (OF) | (OG) | (OL) | (OM) | (ON) |
| 18 | (ON ) | (OO) | (OP) | (OR) | (OR ) | (ORT | (OT) | (OU) | (P) | (PA) |
| 19 | (PE) | (PH) | (PL) | (PO) | (PR) | (PRO | (Q) | (R) | (R ) | (RA) |
| 20 | (RD) | (RE) | (RI) | (RN) | (RO) | (RS) | (RT) | (RU) | (RY) | (S) |
| 21 | (S ) | (S A) | (S I) | (S O) | (SC) | (SE) | (SH) | (SI) | (SO) | (SP) |
| 22 | (SS) | (ST) | (SU) | (T) | (T ) | (TA) | (TE) | (TER) | (TH) | (TH ) |
| 23 | (THE) | (THE ) | (TI) | (TIO) | (TO) | (TO ) | (TR) | (TRA) | (TS) | (TU) |
| 24 | (U) | (UC) | (UN) | (UR) | (US) | (UT) | (V) | (VE) | (VI) | (W) |
| 25 | (WE) | (X) | (Y) | (Y ) | (Y O) | (Z) | | | | |

Time taken for trie construction is 96970 micro seconds.

## 10.0 APPENDIX C

### ORGANIZATION OF THE COMPRESSED DATABASE



PAGE OVERFLOW

The above shows the organization of a single page in the compressed database. In the above page structure, k indicates the position of j th occurrence of code i. In the overflow structure, j indicates the position of code i. In an optimal environment, overflow area is not used until all the positions are filled up in the page.