Real-time loss-less data compression

by

Moegamat Zahir Toufie

(NHD Information Technology (Cape Technikon))

Thesis/dissertation submitted in fulfilment of the requirements for the M. Tech, Information Technology in the School of Business Informatics at the Cape Technikon.

2 February, 2000

Study Leader : Prof. P.J.S. Bruwer

Co-leader : Me E. Scott

Acknowledgements

The financial assistance of the Centre for Science Development, (HSRC, South Africa) towards this research is hereby acknowledged. Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the Centre for Science Development.

I also hereby state that the contents of the thesis/dissertation represent my own work and that the opinions contained herein are my own and not necessarily those of the Cape Technikon.

Summary

Data stored on disks generally contain significant redundancy. A mechanism or algorithm that recodes the data to lessen the data size could possibly double or triple the effective data that could be stored on the media. One mechanism of doing this is by data compression.

Many compression algorithms currently exist, but each one has its own advantages as well as disadvantages. The objective of this study is to formulate a new compression algorithm that could be implemented in a real-time mode in any file system. The new compression algorithm should also execute as fast as possible, so as not to cause a lag in the file systems performance.

This study focuses on binary data of any type, whereas previous articles such as (Huffman, 1952:1098), (Ziv & Lempel, 1977:337; 1978:530), (Storer & Szymanski, 1982:928) and (Welch, 1984:8) have placed particular emphasis on text compression in their discussions of compression algorithms for computer data.

The resulting compression algorithm that is formulated by this study is Lempel-Ziv-Toufie (LZT). LZT is basically an LZ77 (Ziv & Lempel, 1977:337) encoder with a buffer size equal in size to that of the data block of the file system in question. LZT does not make this distinction, *it discards the sliding buffer principle and* uses each data block of the entire input stream, as one big buffer on which compression can be performed. LZT also handles the encoding of a match slightly different to that of LZ77. An LZT match is encoded by two bit streams, the first specifying the position of the match and the other specifying the length of the match. This combination is commonly referred to as a <**position, length**> pair.

To encode the **position** portion of the <**position, length**> pair, we make use of a sliding scale method. The sliding scale method works as follows. Let the position in the input buffer, of the current character to be compressed be held by ***inpos***, where ***inpos*** is initially set to *3*. It is then only possible for a match to occur at position *1* or *2*. Hence the position of a match will never be greater than *2*, and therefore the **position** portion can be encoded using only 1 bit. As ***inpos*** is incremented as each character is encoded, the match position range increases and therefore more bits will be required to encode the match **position**.

The reason why a decimal *2* can be encoded using only 1 bit can be explained as follows. When decimal values are converted to binary values, we get $0_{10} = 0_2$, $1_{10} = 1_2$, $2_{10} = 10_2$, etc. As a position of 0 will never be used, it is possible to develop a coding scheme where a decimal value of 1 can be represented by a binary value of 0, and a decimal value of 2 can be represented by binary value of 1. Only 1 bit is therefore needed to encode match **position** 1 and match **position** 2. In general, any decimal value **n** can be represented by the binary equivalent for (**n** – **1**). The number of bits needed to encode (**n** – **1**), indicates the number of bits needed to encode the match **position**.

The **length** portion of the <**position, length**> pair is encoded using a variable length coding (vlc) approach. The vlc method performs its encoding by using binary blocks. The first binary block is 3 bits long, where binary values 000 through 110 represent decimal values 1 through 7. Where binary value 0 represents decimal value 1 as previously explained when coding a match **position**. This coding scheme is possible, since no match **length** of 0 will ever be encoded. The maximum binary value of a binary block is used to specify whether another binary block follows the current binary block, in which case it is called the block to follow flag (bff). In this case binary 111 specifies that there should be another binary block following this one. Next a 4 bit binary block is appended to the existing 3 bit binary block, resulting in a 7 bit binary block, where binary value 111 0000 represents decimal value 8 and where the maximum binary value of 111 1111 is meant to act as a bff. By continuing in this way the next binary block of bits are appended. Each consecutive binary block is 1 bit bigger in size than the previous binary block. The binary block size continues to grow until it reaches a size of 8 bits. At this point no further increase to the binary block size is made and all subsequent binary blocks will be 8 bits in size.

A distinction has to be made between a literal or compressed character in the output stream. This is needed in order to inform the decompressor of whether the next character that it reads has to be decompressed or not. Making use of a match flag fulfils this requirement. The match flags are encoded the same way as in LZ77 (Ziv & Lempel, 1977:337) where binary value 0 indicates that a literal character is encoded and binary value 1 indicates that a match pair is encoded. Additionally any literal character that is encoded is encoded using static Huffman (Section 2.3), where the frequency table is derived from the characters found in the current data block.

The LZT algorithm is tested against various current compression algorithms using the full test data set from the Calgary/Canterbury compression corpus. The Calgary/Canterbury compression corpus is a set of text and binary files, specifically selected for use by the Internet and academic community to test the efficiency of compression algorithms. The resulting average compression ratio on the test data is 3.249 bits per byte, with an average 180 kilobytes per second compression speed. The test machine is an Intel Pentium, running a 150 MHz processor with 96 MB RAM.

In conclusion, it can be derived from this study, that the LZT algorithm is more efficient for implementation in a real-time environment than currently available compression algorithms

(Section 4.2). Since it has a higher compression ratio than that of currently available real-time compression algorithms. It should also be noted that only 68 kilobytes of RAM is required by the LZT program in order to execute successfully on a computer.

Table of contents

Page

ACKNOWLEDGEMENTS I

SUMMARY II

CHAPTER 1 - INTRODUCTION AND PROBLEM STATEMENT 1
- 1.1 Introduction 1
- 1.2 Problem statement 1
- 1.3 Research objectives 1
- 1.4 Forward 2
- 1.5 Algorithmic conventions 3

CHAPTER 2 - INTRODUCTION TO LOSS-LESS COMPRESSION 5
- 2.1 Run-length encoding 5
- 2.2 Overview of statistical encoders 8
- 2.3 Huffman encoding 12
- 2.4 Shannon-Fano encoding 16
- 2.5 Arithmetic encoding 19
- 2.6 Overview of substitutional encoders 23
- 2.7 LZ77 encoding 24
- 2.8 LZ78/LZW encoding 29

CHAPTER 3 - THE LZT COMPRESSION ALGORITHM 32
- 3.1 Creating a new enhanced compression algorithm 32
- 3.2 Implementing LZT 39

CHAPTER 4 - BENCHMARKING LZT 42
- 4.1 The Calgary Corpus 42
- 4.2 Benchmarks 44

CHAPTER 5 - CONCLUSION 48

REFERENCES 50

GLOSSARY 53

APPENDIX A - LZT PARAMETER SETTINGS 54

APPENDIX B - LZT SOURCE LISTINGS 55

Chapter 1 - Introduction and problem statement

1.1 Introduction

Data stored on disks generally contain significant redundancy. A mechanism or algorithm that recodes the data to lessen the data size could possibly double or triple the effective data that could be stored on the media. One technique of doing this is by data compression.

1.2 Problem statement

In the past two decades, many advances have been made in the field of data compression and more so when implementing them to run in a real-time mode in file systems. Faster and more productive compression algorithms have been produced, but with every advantage that a new compression algorithm brings, there are also disadvantages. Be it in its speed of execution or in the ratio of the resulting compressed data. Another problem that becomes evident is that all these compression algorithms were implemented into file systems that were not originally designed to handle compression of its files. With this, a certain performance and compression loss had to occur in order to achieve successful implementation of a compression algorithm into existing file systems.

1.3 Research objectives

The objective of this study is to take what is currently known about data compression and file systems, and to formulate a new compression algorithm that could be implemented in a real-time mode in any file system. The new compression algorithm should also be as fast as possible in its speed of execution so as not to cause a lag in the file systems performance. The intention is not to design a new file system, but merely to design a compression algorithm that is universally adaptable into existing file systems. This study focuses on binary data of any type, whereas previous articles such as (Huffman, 1952:1098), (Ziv & Lempel, 1977:337; 1978:530), (Storer & Szymanski, 1982:928) and (Welch, 1984:8) have placed particular emphasis on text compression in their discussions of compression algorithms for computer data.

1.4 Forward

During the research many types of compres ion algorithms were found. Some of them being run-length encoding (Welch, 1984:8), statistical encoding (Huffman, 1952:1098), arithmetic encoding (Cormack & Horspool, 1987) and substitutional encoding (Ziv & Lempel, 1977:337; 1978:530) (Welch, 1984:8).

In the next chapter an introduction to data compression will be given. A detailed analysis of the different types of algorithms together with their respective advantages and disadvantages will be done. There are many more algorithms and derivatives in existence today. However those that are discussed, are the most commonly known algorithms and also the algorithms that will later form a basis for the new algorithm, namely Lempel-Ziv-Toufie (LZT).

In chapter 3, a detailed description of the LZT algorithm as well as the engineering behind the LZT algorithm will be given. Various implementations of the LZT algorithm will also be given. The main reason behind giving more than one LZT implementation is to show the advantages or disadvantages of certain mathematical coding techniques.

In chapter 4, a benchmarking data set namely The Calgary Corpus will be discussed. This benchmarking data set will be used to compare the performance of the various LZT algorithm implementations, against those existing data compression algorithms as discussed in chapter 2. It is necessary to note that only certain data compression algorithms that are covered in chapter 2 are used in the comparison. Most of the data compression algorithms given in the comparison are also current derivatives of the basic data compression algorithms covered in chapter 2.

In chapter 5, a conclusion of the findings of the study is given as well as a section on future trends and possibilities.

1.5 Algorithmic conventions

Algorithms are used in some sections to facilitate a better understanding of the examples given. With these algorithms, certain conventions had to be used. Figure 1.1 lists some of the conventions used, with the relevant syntax and the description of the syntax. Additionally whenever an input file is read character by character, the last character that is read at the end of the file will automatically set an end of file (EOF) flag to **true**. While the end of the file has not yet been reached, the EOF flag will be set to **false**.

Certain data structures were used in order to identify the type of a variable as used within the algorithms. These are **character string** variables that can contain one or more characters, **character** variables that can contain only one character and **numeric** variables that can contain any numerical value. Another data structures used is an **array**. This type identifies a table containing more than one entry based on the before-mentioned variable types, e.g. a **numeric array a** would be a table called **a** containing more than one entry of a numerical type.

An input stream refers to an input source of some kind. This can be a computer file or a portion of a computer file. Similarly, an output stream refers to an output source of some kind. This can also be a computer file or a portion of a computer file.

When using the syntax $\mathbf{a_i}$ with an array, reference is made to array **a** entry number **i**. When using the syntax $\mathbf{a_i}$ with reading an input stream, reference is made to reading the character at position **i** within the input stream and storing its value into variable **a**.

Syntax	Description
a ← b	Where **a** is assigned the value of **b**. e.g. If **b** = 3, then **a ← b** would result in **a** = 3. If **b** = "C", then **a ← b** would result in **a** = "C".
a ← b + c	Where **a** is assigned the value of the sum of **b** & **c**. e.g. If **b** = 3 and **c** = 5, then **a ← b + c** would result in **a** = 8. If **b** = "C" and **c** = "D", then **a ← b + c** would result in **a** = "CD".
a ← 0 $\mathbf{f_{[]}}$ **← 0**	Initialise numeric variable **a** to contain 0. Initialise each entry in the numeric array variable **f** to contain 0.
a ← "" $\mathbf{f_{[]}}$ **← ""**	Initialise character string variable or character variable **a** to contain the empty string. Initialise each entry in the character string array variable or character array variable **f** to contain the empty string.

Figure 1.1 - Algorithmic conventions.

Chapter 2 - Introduction to loss-less compression

2.1 Run-length encoding

Run-length encoding is achieved when sequences of identical characters are encoded as a count field plus an identifier of the character that is repeated. Figure 2.1 shows how a sample string of characters would be encoded using run-length encoding.

Sample text
```
aaabbbbaacccc
```
When compressed using run-length encoding
```
3a4b2a4c
```

Figure 2.1 - Run-length encoding example.

Sample text
```
aaa4444b22c11
```
When compressed using run-length encoding
```
3a44b22c21
```

Figure 2.2 - Run-length encoding example, using single character suppression.

Distinguishing the count fields from normal characters is a problem with run-length encoding for character sequences intermixed with other data. Figure 2.2 shows another run-length encoding example that uses single character suppression. This is when a single character occurrence is encoded without a count field. It is clear from the example that the compressor encodes correctly. The decompressor however will have a problem decompressing as there is no way of telling whether the *b* at position 5 is the count field for the *2* at position 6 or whether the *2* at position 6 is the count field for the *2* at position 7. One possible solution would be to use a special character to mark each run of characters.

Sample text
```
aaabbdccc
```
When compressed using run-length encoding
```
@3a@2b@d@3c
```

Figure 2.3 - Run-length encoding example, using character runs.

Variable descriptions:

o	: the current repeating character that is to be compressed.
c	: the total number of occurrences of **o**.
k	: the current character read from the input stream..
i	: the position of the character **k**, within the input stream.

Initial variable values:
```
c ← 0
i ← 1
o ← ""
```

Algorithm:
```
repeat
    read $k_i$ from the input stream

    if ($k_i$ = o) or (o = "") then
        c ← c + 1
        o ← $k_i$
    else
        write "@" into the output stream
        if c > 1 then
            write c into the output stream
        write o into the output stream

        o ← $k_i$
        c ← 1
    end if

    i ← i + 1
until EOF = true
```

Figure 2.4 - Run-length encoding algorithm, using character runs.

This procedure is sufficiently capable of handling normal text, but not arbitrary bit patterns as those found in binary data. Figure 2.3 shows an example of an implementation using character runs.

From Figure 2.3, it is clear that 2 or 3 characters are needed to mark each character run, hence this type of encoding would not be used for runs of less than 4 characters, in order to achieve compression. Figure 2.4 shows the actual algorithm for such a run-length encoding implementation.

Run-length encoding is a very primitive way of compressing data and only yields significant results in data files with long streams of recurring bit patterns. Suitable data files that might benefit from this compression algorithm would be graphics files that contain long streams of recurring bits, which are usually associated with a certain colour pattern. Unfortunately, normal data files such as word processor documents, database files or normal program binaries would not yield significant results (Welch, 1984:8), and as such, run-length encoding would not be suitable for use with a compressed file system.

2.2 Overview of statistical encoders

Statistical encoders use an algorithm which encodes or decodes a character with a number of bits proportional to ***-log₂(p)***, where ***p*** is the predicted probability. All statistical encoders have this characteristic. It must be noted that statistical encoders may round the encoded number of bits up or encode with extra bits. For example, let the input data consist of only three different characters, namely ***a***, ***b*** and ***c***. Let the statistical encoder then be told by virtue of a hard coded table that it is to expect that ***a*** occurs 50% of the time and that ***b*** as well as ***c*** occur 25% of the time. With this we would want to code as follows:

1 bit for character ***a*** as $-\log_2(0,5) = 1$

2 bits for characters, ***b*** and ***c*** as $-\log_2(0,25) = 2$

RAW character	Encoding code
a	0
b	10
c	11

RAW character	Encoding code
a	0
b	11
c	10

RAW character	Encoding code
a	1
b	01
c	00

RAW character	Encoding code
a	1
b	00
c	01

Figure 2.5 - The basic set of binary codes for the three character alphabet.

The encoder could then use any one of the four sets of binary codes as shown in Figure 2.5 The most important factor is that the lengths of the binary codes conform to $-\log_2(p)$.

Data files composed of characters found in a predefined table, can be encoded by a statistical encoder by means of predicting each character with equal probability. An example of such a table that can be encoded using equal probabilities for each character is the ASCII table. The ASCII table contains 256 characters, each character of which uses one byte to uniquely represent itself. The number of bits used by a byte can be obtained by assigning an equal probability p, where $p = {}^{1}/_{256}$ for any of the 256 characters, hence

$$\textit{number of bits} = -\log_2({}^{1}/_{256}) = 8$$

RAW character	Encoding code
aa	1
ab	01
ba	001
bb	000

Figure 2.6 - A basic blocking set of binary codes.

Sample text
```
aaababbaaabb
```

When compressed using statistical encoding with blocking
```
101010011000
```

Figure 2.7 - Statistical encoding example, using blocking.

Another type of statistical encoder uses a fractional number of bits to represent a character. An example of such an encoder is an arithmetic encoder, detail of which can be found in section 2.5.

Another example is a blocking encoder, which works as follows. First, a table is built containing every two character combination that exists in the input stream. Using this table, a set of prefix-free codes (Huffman, 1952:1098) is created for each entry in the table, using the Huffman algorithm (Huffman, 1952:1098) as explained in section 2.3 or the Shannon-Fano

(Shannon & Weaver, 1949) algorithm as explained in section 2.4. For example, if an input stream consisted of ***aaababbaabb***, and our prefix-free codes are as shown in Figure 2.6, then the input stream after encoding would be as shown in Figure 2.7.

From Figure 2.6, it is stated that an ***aa*** combination is encoded with 1 bit. Thus each ***a*** in that combination is encoded with 0.5 bit. The ***ba*** and ***bb*** combinations are encoded in 3 bits. Thus, each character in those combinations is encoded with 1.5 bits. Hence, in this way a statistical encoder is able to encode a single character using a fractional number of bits.

A statistical encoder creates a set of binary codes by using an iterative or recursive algorithm as described by (Huffman, 1952:1098) or (Shannon & Weaver, 1949). The resulting binary codes are called prefix-codes. A prefix-code has the characteristic that no prefix-code is the prefix of another prefix-code (Huffman, 1952:1098). For example, let ***a, b*** and ***c*** be the only input characters. Figure 2.8 does not contain a prefix-free set of binary codes, since the decoder will have two possible outputs of "**aac**" or "**bc**" for the input stream of "**1101**". Figure 2.9 on the other hand does contain a set of prefix-free binary codes as it will output the correct sequence of "**aab**" for the input stream "**1101**".

RAW character	Encoding code
a	1
b	11
c	01

Figure 2.8 - A non prefix-free set of prefix-codes.

RAW character	Encoding code
a	1
b	01
c	00

Figure 2.9 - A prefix-free set of prefix-codes.

In the following two sections, Huffman encoding and Shannon-Fano encoding will be described with emphasis on how they achieve the creation of a set of prefix-codes using a binary tree.

2.3 Huffman encoding

Huffman first developed a classic binary tree used to generate a set of prefix-codes. The idea behind Huffman's algorithm is to use short bit strings to represent the most frequently used characters and to use longer bit strings to represent less frequently used characters. The algorithm as shown in Figure 2.11, can be explained as follows.

Create a table of frequencies containing each possible character found in the input stream as well as the corresponding number of times that each character occurs in the input stream. For example if the input stream is "**ababaca**" then Figure 2.10 shows what the resulting frequency table will look like.

RAW character	Frequency
a	4
b	2
c	1

Figure 2.10 - A basic frequency table.

The frequency table is a simple table listing each character in the input stream in non-increasing order of how many times they have occurred. The values obtained from the frequency table are then used to create a binary tree consisting of a root at the top and branches, which consists of more branches or leaf nodes.

The Huffman algorithm creates a binary tree from bottom up. The two entries in the frequency table with the smallest frequencies are combined into a binary tree as in Figure 2.12, resulting in a new combined frequency.

Variable descriptions:

s	: the sum of two frequencies from **f**.
f	: the array of frequencies (i.e. number of occurrences) of each character or sub-string.
k	: the current character read from the input stream.
i	: the position of the character **k**, within the input stream.
j	: the index of an element within a frequency table for a specific character.
v	: the vertex of a new binary tree.
x	: the index of an element within a frequency table for a character that has the smallest frequency.
y	: the index of an element within a frequency table for a character that has the second smallest frequency.

Initial variable values:

```
$f_{[]} \leftarrow 0$
$i \leftarrow 1$
```

Algorithm:

```
repeat
    read $k_i$ from the input stream
    $j \leftarrow$ index in f of element representing character $k_i$
    $f_j \leftarrow f_j + 1$
    $i \leftarrow i + 1$
until EOF = true

repeat
    $x \leftarrow$ index in f of element with the smallest value
    $y \leftarrow$ index in f of element with the second smallest value
    $s \leftarrow f_x + f_y$
    let the descendants of v be the terminal vertices representing $v_x + v_y$

    if $f_x$ = value of a vertex of an existing binary tree
        join the vertex represented by $f_x$ to the terminal vertice representing $v_x$
    else
        assign $f_x$ to the terminal vertice representing $v_x$
    end if
    if $f_y$ = value of a vertex of an existing binary tree
        join the vertex represented by $f_y$ to the terminal vertice representing $v_y$
    else
        assign $f_y$ to the terminal vertice representing $v_y$
    end if

    delete elements $f_x$ and $f_y$ from f
    add a new element s into f
until f contains only 1 elements
```

Figure 2.11 - Huffman algorithm for creating a prefix-free binary tree.

The two entries that were combined are deleted from the frequency table and inserted in their place in the frequency table is an entry for the new combined frequency. It is important to note that the table stays sorted in non-increasing order of frequencies. The above process is repeated until only one entry remains in the frequency table. The result is a Huffman prefix-free binary tree as in Figure 2.13.

The input stream is encoded by traversing the binary tree from top, down. By either moving left or right until the character that is being encoded is reached. Each time a left or right node is reached, we encode a "**0**" or "**1**" bit respectively.

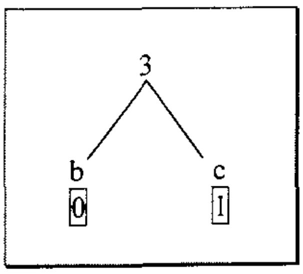


Figure 2.12 - The first binary tree section of a Huffman binary tree.

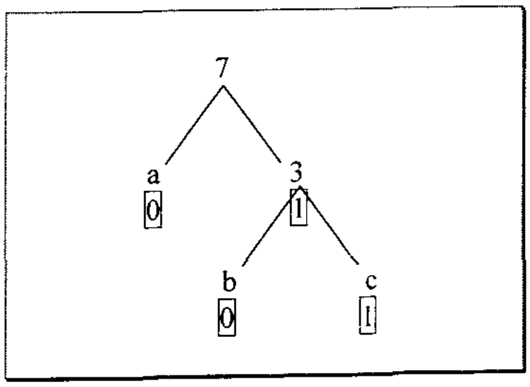


Figure 2.13 - Our completed Huffman binary tree.

Using the previous example, the resulting output bit stream would be "**0100100110**", where an "**a**" is encoded with a "**0**" bit, a "**b**" is encoded with the bit string "**10**" and a "**c**" is encoded with the bit string "**11**". This algorithm is referred to as a static Huffman algorithm. The static

Huffman algorithm yields a set of prefix-free binary codes, since no one prefix-code is the prefix of another prefix-code.

In conclusion, it is found that at least two passes over the input data, is required in order to encode it. The first would be to create the frequency table and the second would be to do the actual compression. The following two disadvantages become evident:

- the speed of execution will suffer due to the two passes required, and
- the decoder must receive the original freauency table in order to perform the decompression.

An adaptive approach can be used to overcome both disadvantages. The adaptive approach works as follows. Create an initial frequency table where all the characters found in the input stream have a frequency of one. From this frequency table, a Huffman binary tree is created while at the same time we preserve the initial frequency table. After inputting the first character, the frequency of the inputted character is incremented by one in the initial frequency table. The Huffman binary tree is then recreated from the updated initial frequency table. After inputting the next character, the initial frequency table is again updated and the Huffman table is again recreated. In this way, an adaptive Huffman binary tree is maintained.

The adaptive approach requires only one pass for compression as no detailed frequency table has to be build before compression starts. In addition, no frequency table has to be sent to the decoder, since the decoder builds its own frequency table as it goes along. Huffman's algorithm is optimal in length as no other prefix-free binary tree will produce a better compression result for the given input stream (Huffman, 1952:1098).

2.4 Shannon-Fano encoding

Shannon-Fano encoding (Shannon & Weaver, 1949) is similar to Huffman encoding in many ways but differs in the way in which the binary tree is created. The Shannon-Fano algorithm however does not guarantee to create optimal prefix-codes, but approaches optimal behaviour as the number of characters approaches infinity. The advantage of the Shannon-Fano algorithm is the simplicity of the creation of its prefix-free binary tree as shown in Figure 2.14.

Variable descriptions:

f	: the array of frequencies (i.e. number of occurrences) of each character or sub-string.
k	: the current character read from the input stream.
i	: the position of the character **k**, within the input stream.
j	: the index of an element within a frequency table for a specific character.
pc	: the prefix-code of a character or sub-string.

Initial variable values:

```
$f_{[]} \leftarrow 0$
$i \leftarrow 1$
```

Algorithm:

```
repeat
    read $k_i$ from the input stream
    $j \leftarrow$ index in f of element representing character $k_i$
    $f_j \leftarrow f_j + 1$
    $i \leftarrow i + 1$
until EOF = true

convert each frequency in f to a percentage based on the sum of all the
  frequencies within f

repeat
    divide the elements into two array's called $f_{x1..xn}$ and $f_{y1..yn}$, so that the
      sum of the percentages of the latter two arrays are approximately equal

    each element of the arrays $f_{x1..xn}$ and $f_{y1..yn}$ also contains an additional
      variable called pc

    append a 0 bit to each pc field of $f_{x1..xn}$
    append a 1 bit to each pc field of $f_{y1..yn}$
until all the array's $f_{x1..xn}$ and $f_{y1..yn}$ contain only 1 element
```

Figure 2.14 - Shannon-Fano algorithm.

Create a table of frequencies containing each possible character found in the input stream. A percentage based on the entire frequency table is calculated for each frequency in the frequency table. Each percentage in the frequency table serves as the probability of each character occurring in the input stream. For example if the input stream is "**ababaca**" then Figure 2.15 shows what the resulting probability table will look like. The probability table is a simple table listing each character in the input stream in non-increasing order of probability. The probabilities obtained from the probability table are then used to create a binary tree consisting of a root at the top and branches, which consists of more branches or leaf nodes.

RAW character	Probability
a	57%
b	29%
c	14%

Figure 2.15 - A basic probability table.

Prefix-free codes are created by dividing the probability table into two tables, in such a manner that the sum of the probabilities of the resulting tables is nearly equal. Each entry in the prefix-free code column of the first table receives a "**0**" bit and each entry in the second table receives a "**1**" bit as the first bit for their prefix-codes, as shown in Figure 2.16. Each of these tables is then divided again to the same criterion as before, again appending a "**0**" bit to each entry in the first table and a "**1**" bit to each entry in the second table. The result can be seen in Figure 2.17.

This process is repeated until no table contains more than one entry, resulting in the prefix-free codes as shown in Figure 2.18. The resulting prefix-free codes are similar to the ones obtained for the Huffman algorithm. The reason for this similarity is that the input stream is very short. With a bigger input stream the resulting binary trees tend do differ.

The encoding of the input stream is done exactly as in the Huffman algorithm. Whereby the binary tree is traversed from top, down, allocating a "**0**' or "**1**" bit as required. This static Shannon-Fano algorithm has the same disadvantages as with the static Huffman algorithm. Hence, an adaptive approach can also be used here as a possible solution.

RAW character	Probability	Prefix-free code
a	57%	0
b	29%	1
c	14%	1

Figure 2.16 - Shannon-Fano codes after the first division.

RAW character	Probability	Prefix-free code
b	29%	10
c	14%	11

Figure 2.17 – Shannon-Fano codes after the second division.

RAW character	Probability	Prefix-free code
a	57%	0
b	29%	10
c	14%	11

Figure 2.18 - Shannon-Fano prefix-free codes on completion of the algorithm.

The Shannon-Fano algorithm might not create an optimal prefix-free binary tree because the length of the prefix-code is equal to $-\log_2(p)$, where p is the predicted probability of a character. In the cases where it is possible to half a table into exactly two tables of equal probability, the prefix-codes will be optimal. In all other cases, it may not be possible to obtain optimal codes.

2.5 Arithmetic encoding

From section 2.3 and section 2.4, it could appear that Huffman or Shannon-Fano encoding is perfect for compressing a stream. However, this is not the case. Huffman and Shannon-Fano encoding are only optimal if and only if the character probabilities are integral powers of 1 or 2, which is almost never the case. Arithmetic encoding does not have this restriction. The original concept of arithmetic coding was suggested by Elias (1975:194).

Unlike Huffman and Shannon-Fano encoding, arithmetic encoding does not create prefix-codes in order to encode characters. Instead, an input stream is encoded as an interval represented by a real number ***r***, where $0 \geq r < 1$. After each character in the input stream is read, the value of the real number needed to represent the resulting encoding is decreased, and the number of bits needed to represent this real number is increased. Arithmetic encoding uses a probability table containing input characters whose probabilities are then used to successively narrow the interval used to represent the input stream. A high probability character narrows the interval less than a low probability entry would. Hence high probability characters contribute fewer bits to the encoding. The algorithm as shown in Figure 2.19 can be explained as follows.

Create a table of probabilities containing each possible character found in the input stream as well as the probability of the number of times that each character occurs in the input stream, as is done with Shannon-Fano encoding. For example if the input stream is "**ababaca**". Figure 2.20 shows what the resulting probability table will look like. The entries in the probability table are then used to create a partitioned interval $\lfloor 0, 1 \rfloor$ as in Figure 2.21. Since each partition has a lower and an upper limit, one of the two values must be included in the current partition. For this study, the lower value will always be included in the current partition so that $^{0}/_{100}$ falls within the partition used for character "**c**", and so on for characters "**b**" and "**a**". Now since the first character is an "**a**" the interval is shrunk to $\lfloor ^{43}/_{100}, 1 \rfloor$.

```
Variable descriptions:
    f    : the array of frequencies (i.e. number of occurrences) of each character
           or sub-string.
    k    : the current character read from the input stream.
    i    : the position of the character k, within the input stream.
    j    : the index of an element within a frequency table for a specific
           character.
    r    : an interval within ⌊0, 1⌋
    r_h  : the maximum value of the interval r.
    r_l  : the minimum value of the interval r.

Initial variable values:
    $f_{[]} \leftarrow 0$
    $i \leftarrow 1$
    $r_h \leftarrow 1$
    $r_l \leftarrow 0$

Algorithm:
    repeat (read input stream for the 1st time)
        read $k_i$ from the input stream
        $j \leftarrow$ index in f of element representing character $k_i$
        $f_j \leftarrow f_j + 1$
        $i \leftarrow i + 1$
    until EOF = true

    convert each frequency in f to a percentage based on the sum of all the
      frequencies within f

    $i \leftarrow 1$
    repeat (read input stream for the 2nd time)
        read $k_i$ from the input stream

        $r \leftarrow$ interval within r representing the probability of $k_i$

        if one or more leftmost bits of $r_1$ = one or more leftmost bits of $r_2$ then
            write the corresponding leftmost bits of $r_2$ into the output stream

        $i \leftarrow i + 1$
    until EOF = true
```

Figure 2.19 - Arithmetic algorithm.

This process is repeated, by continually shrinking the resulting interval into a smaller interval until the entire input stream is encoded. In the above example, the arithmetic encoder is not completely efficient, which is due to the short size of the input stream. With longer input streams, the coding efficiency does indeed approach 100% as proven in Elias (1975:194).

RAW character	Probability
a	57%
b	29%
c	14%

Figure 2.20 - A basic probability table.

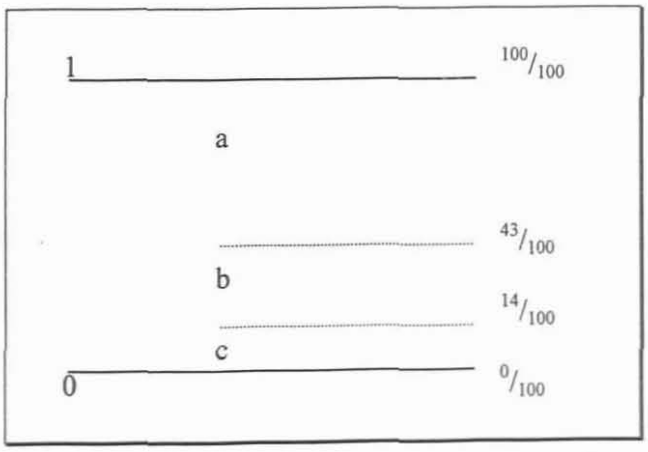


Figure 2.21 - Our initial partitioned interval $\lfloor 0, 1 \rfloor$.

An issue left unresolved by the concept of arithmetic encoding is that it appears that the encoding algorithm transmits nothing until the final interval is determined. However, this delay is not necessary. As the interval becomes narrower, the leading bits of the top and bottom points become the same. Any leading bits that are the same may be transmitted immediately, as they will not be affected by any further processing.

The above algorithm is also static and presents us with the same disadvantages as the static Huffman and Shannon-Fano algorithms. Hence, an adaptive approach as a possible solution can be used as well.

The static approach is converted into an adaptive one by adjusting the character probabilities after each new character is encoded, allowing the model to track the data being transmitted. This approach is however not a particularly good estimate of the true displacement of the various characters in the input stream.

Inter character probabilities should also be taken into account. Inter character probability is the probability of one character following another in the input stream. For example, if an ***a*** is inputted from the input stream, then the probability of a ***b*** being inputted next is the inter character probability between ***a*** and ***b***. One such arithmetic compression implementation that takes inter character probabilities into account is Dynamic Markov Coding or DMC, which is presented by Cormack and Horspool (1987).

One major disadvantage of arithmetic encoding is that the algorithm can consume rather large amounts of memory, especially in the case of DMC. The arithmetic encoding process itself involves a fair amount of number crunching, especially in the division of the interval. Hence arithmetic encoding is currently unsuitable for any real-time implementations due to its slow speed of execution and large memory requirements.

2.6 Overview of substitutional encoders

Substitutional encoders use an algorithm in which the encoder replaces an occurrence of a particular group of characters, with a reference to a previous occurrence of that group of characters. All substitutional encoders have this one characteristic.

There are two classes of substitutional encoders, both of which were first proposed by Ziv and Lempel (1977:337; 1978:530). They are commonly referred to as LZ77 and LZ78. A derivative of the LZ78 method and proposed by Welch (1984:8) commonly referred to as LZW, is used in real-time environments, such as disk drive controllers, modems and network routers.

One major advantage of a substitutional encoder is that it compresses an input stream very fast. Because of the speed advantage most current compression programmes like the UNIX compress command uses some derivative of LZ77 or LZ78. Another advantage is that it requires no prior knowledge of the input stream. This means that only a single pass through the input stream is required, since there is no need to build any probability or weight table. This type of encoder is usually used when a statistical test is either impossible or unreliable due to the length of the input stream.

The compression ratio achieved by substitutional encoders outperforms that of most statistically based encoders. The compression ratio can also be further increased when a substitutional encoder incorporates some type of statistical algorithm as well.

The following two sections describe the LZ77 and LZ78/LZW encoders in more detail.

2.7 LZ77 encoding

The LZ77 algorithm is one of the simplest compression algorithms. The algorithm was first proposed by Ziv and Lempel (1977:337). A slightly modified version was later proposed by Storer and Szymanski (1982:928) and is commonly called LZSS. The LZSS derivative of the LZ77 algorithm as shown in Figure 2.23 can be explained as follows:

The algorithm keeps track of the last **n** characters of data read. When a sub-string is encountered that has already been read, it outputs a pair of values corresponding to the **position** of the previous encounter as well as the **length** of the matched sub-string. The size of **n** is determined beforehand and can be set to virtually any value. In effect the encoder moves a fixed sized sliding buffer of size **n** over the input stream. The sliding buffer contains the current inputted character as well as any previous inputted characters. The current inputted character is contained in the first filled rightmost position within the sliding buffer noting that the sliding buffer is filled from left to right.

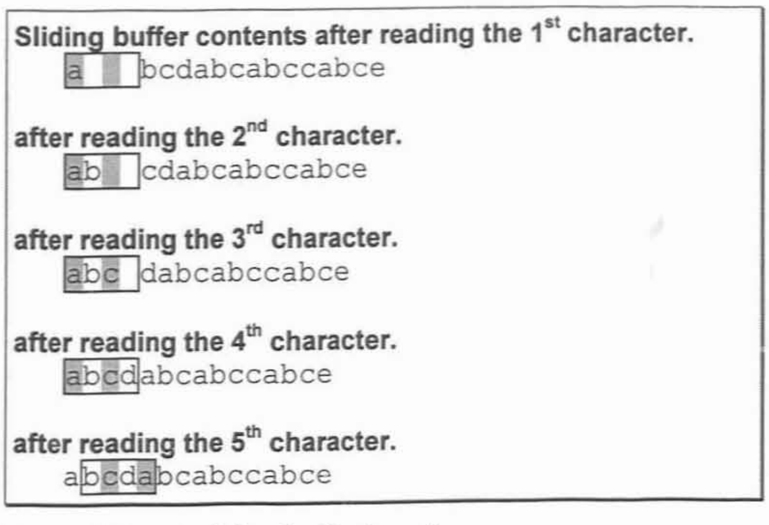

Sliding buffer contents after reading the 1st character.
a bcdabcabccabce
after reading the 2nd character.
ab cdabcabccabce
after reading the 3rd character.
abc dabcabccabce
after reading the 4th character.
abcdabcabccabce
after reading the 5th character.
abcdabcabccabce

Figure 2.22 – A sliding buffer in action.

For example, if an input stream was made up of the character string "**abcdabcabccabce**" and the sliding buffer size was equal to 4, then Figure 2.22 shows what the sliding buffer would contain after reading characters from the input stream from positions 1 through 5.

Variable descriptions:

b	: the sliding buffer, an array of characters from the input stream.
k	: the current character read from the input stream.
i	: the position of the character **k**, within the input stream.
w	: the sub-string to be compressed.
p	: the position of a matching sub-string.
l	: the length of a matching sub-string.
m	: the minimum allowable length of a matching sub-string.

Initial variable values:

```
m ← 3
i ← 2
```

Algorithm:

```
read k1 from the input stream and place into leftmost element in b
w ← k1

repeat
    read ki from the input stream
    w ← w + ki

    if b has no empty elements then
        shift the contents of b left by 1 element
    insert ki into first empty leftmost element in b

    if w exists previously in b then
        p ← position of the matching sub-string w within b
    else
        l ← (length of w) - 1
        if l >= m then
            write a value 1 bit to the output stream to specify a match
            write p to the output stream
            write l to the output stream
            w = ki
        else
            write a value 0 bit to the output stream to specify a literal character
            write leftmost character of w to the output stream
            w ← 2nd leftmost character from w
            i ← position of the character w, within the input stream
        end if
    end if

    i ← i + 1
until EOF = true
```

Figure 2.23 - LZ77 algorithm.

When encoding a match, the position portion of a **<position, length>** match pair refers to the position of a matching sub-string within the sliding buffer. A match is therefore only possible if the matching sub-string is within the sliding buffer. Any matching sub-strings that are no longer within the sliding buffer are ignored.

The modified LZSS algorithm works in more or less the same way except that it also makes use of a look ahead buffer, the contents in which it tries to find a match for in the sliding buffer. The LZSS derivative of the LZ77 algorithm will be explained, since the LZSS algorithm yields a better compression ratio than the original LZ77 algorithm.

To implement the LZSS algorithm, a fixed sized sliding buffer of a predefined size that is initially empty must be created. The size of the sliding buffer is very important as it has a direct relation to the size of the **<position, length>** match pair. For example, if the sliding buffer is of size 4096 characters, then the match **position** can be encoded in 12 bits since $2^{12} = 4096$. If the match **length** were to use 4 bits, then encoding matches of up to 2^4 or 16 characters would be possible. A total of 16 bits or 2 bytes would then be required to encode a sub-string match. Which would mean that we have to find a sub-string match of at least length ***m***, where ***m = length(position tag) + length(length tag) + 1***, before compression would be possible.

Initialise a sub-string **w** with the first character of the input stream. Once a suitable sliding buffer size is decided on, one character at a time is read from the input stream and placed in the first empty leftmost position of the sliding buffer.

After each character is read, a search is done from the left to the right of the buffer, for a match corresponding to **w**. If no match is found, the leftmost character in **w** is written to the output stream. The sub-string **w** is then set to contain only the second leftmost character of itself. The input stream is then read again starting from the character after the one now held by **w**. The next read character is then appended to sub-string **w**. For example if **w** = "**abcd**" the "**a**" would have been written to the output stream and **w** would be set to "**b**". The input stream would then be read starting from character "**c**". After which a search would be performed through the sliding buffer as before.

Once the sliding buffer is full, the contents of the sliding buffer are shifted one character to the left. In doing so the leftmost character within the sliding buffer is discarded and an empty space is created at the rightmost position of the sliding buffer.

If a match is found, then the match **position** is temporarily stored and no character is written to the output stream. Note that it is possible to find more than one match in which case a table will have to be kept to store all the possible match positions. The next character is then read and appended to **w**. If after $\boldsymbol{i}$ inputs a match of length $\boldsymbol{l}$ is found, where $\boldsymbol{l} >= \boldsymbol{m}$ and $\boldsymbol{m}$ is the minimum match length required, then a **<position, length>** match pair corresponding to the position and length of the matched sub-string is outputted. If however the $\boldsymbol{l} < \mathbf{m}$, then it is assumed that there is no match and the process is performed as above to handle a no match situation.

Note that an extra bit must also be written to the output st-eam in order to indicate to the decoder whether or not the next piece of data that it reads is a normal character or a **<position, length>** match pair. Decompression is achieved by reading the input stream character by character and whenever a **<position, length>** match pair is encountered, that match pair is replaced by a copy of the input stream found at the indicated **position** in the buffer with a size of **length** characters.

The sliding buffer implementation as described in the above example, automatically creates the least recently used (LRU) effect, which as will become evident later, has to be done explicitly in the LZ78 algorithms. Variants of the LZSS algorithm apply additional compression to the output stream of the algorithm by using simple variable length codes or by using some form of Huffman or Shannon-Fano encoding, all of which result in a certain degree of improvement over the basic compression algorithm.

2.8 LZ78/LZW encoding

The LZ78 algorithm is also a very simple compression algorithm. The algorithm was first proposed by Ziv and Lempel (1978:530). A slightly modified version was later proposed by Welch (1984:8) and is commonly called LZW. The LZ78 algorithm works as follows.

Sub-strings of data previously seen are entered into a dictionary of size ***n***. When a sub-string is encountered that is in this dictionary, the dictionary index corresponding to the position in the dictionary of the encountered sub-string is written to the output stream. The modified LZW algorithm is similar to the LZ78 algorithm except that the LZ78 algorithm starts with an empty dictionary. The LZW algorithm fills the first ***n*** positions with the full alphabet of current input stream. So that if an 8 bits per character alphabet is used, LZW would fill the first 256 positions in the dictionary with the actual 256 characters for that alphabet. The LZW algorithm yields a better compression ratio than LZ78 proof of which can be found in Welch (1984:8). Hence, the focus of the explanation of the LZ78 algorithm will be via the LZW derivative.

To implement the LZW algorithm, a dictionary of a predefined size ***n*** is kept, which is initially filled with each character found in the input streams. The size of the dictionary influences the size of the encoding bit stream. For example, if the dictionary is of size 4096, then the **indices** could be encoded in 12 bits since $2^{12} = 4096$. Unlike the LZ77 algorithm and its LZSS derivatives, a match of any size can be encoded and thus the algorithm is not limited to the minimum length constraint where a match of length ***m*** must conform to ***m >= length(position tag) + length(length tag) + 1***.

```
Variable descriptions:
    k   : the current character read from the input stream.
    i   : the position of the character k, within the input stream.
    w   : the sub-string to be compressed.
    t   : the dictionary, an array of sub-strings already inputted.
    j   : the index of an element within t of a matching sub-string.

Initial variable values:
    w ← ""
    i ← 1
    t[] ← ""

Algorithm:
    repeat
     read k_i from the input stream
     w ← w + k_i

     if w = an entry within t then
        j ← index in t of element representing the matching sub-string w
     else
        write j to the output stream
        add a new element w into t
        w ← k_i
        end if

     i ← i + 1
    until EOF = true

    write index in t of element representing w to the output stream
```

Figure 2.24 - The LZW compression algorithm.

The basic encoding algorithm can be outlined as in Figure 2.24. For example, if the input stream is "**abcdabcabccabce**" and the alphabet consisted out of 5 characters, then the inputs, outputs and dictionary entries will look similar to that of Figure 2.25.

The decompression of LZW works exactly like the compression. A new sub-string is added to the dictionary when the next character read from the input stream. All it needs to do in addition is to translate each incoming index into a string and write it to the output stream. Using the above example, the decompression would look something like that of Figure 2.26. There is one exception to the LZW algorithm on the compression side that cause some trouble on the decompression side. If the decompressor inputs an index before it is actually in the dictionary, it won't know what to do. Fortunately this is the only exception and can be resolved by hard coding a handler to deal with this exception in an appropriate manner.

Character input (k)	w	Output index	New index	New sub-string
a	a			
b	b	a	256	ab
c	c	b	257	bc
d	d	c	258	cd
a	a	d	259	da
b	ab			
c	c	256	260	abc
a	a	c	261	ca
b	ab			
c	abc			
c	c	260	262	abcc
a	ca			
b	b	261	263	cab
c	bc			
e	e	257	264	bce
		e		

Figure 2.25 - LZW compression.

Character input (k)	w	Output index	New index	New sub-string
a	a	a		
b	b	b	256	ab
c	c	c	257	bc
d	d	d	258	cd
256	a	ab	259	da
c	c	c	260	abc
260	a	abc	261	ca
261		ca	262	abcc
257		bc	263	cab
e		e		

Figure 2.26 - LZW decompression.

Unlike LZ77 and its derivatives, the dictionary in LZW does not perform any LRU maintenance on the dictionary indexes and sub-string. An implementation of LZW has to explicitly clear the dictionary once full or implement some type of LRU method, so as to ensure that only the most recently or most frequently used sub-strings remain in the dictionary.

Chapter 3 - The LZT compression algorithm

3.1 Creating a new enhanced compression algorithm

In a DOS based file system the drive is divided into a root sector which holds system specific data and the boot loader, 2 file allocation tables or FAT and a root directory. The rest of the drive is split into equal clusters, each cluster is made up of 1 or more sectors. These clusters are the file systems data blocks, usually multiples of 2 kilobytes (k), and ranging from 2k to 32k, depending on the size of the drive in question. Other file systems data blocks might be of different sizes. The Lempel Ziv Toufie (LZT) compression algorithm, which is derived from this study, is a derivative of the LZ77 encoder with a fixed sized buffer equal in size to that of the file systems data block.

An explanation of the basic LZT algorithm, of which Figure 3.4 shows the basic processing, follows below. In section 2.7, it is shown that LZ77 uses a sliding buffer, which moves, over the input stream. LZT does not make this distinction, it discards the sliding buffer principle and uses each data block of the entire input stream, as one big buffer on which compression can be performed. Therefore, if the input stream is ***n*** characters in length, and each data block is ***m*** characters in length, in the case of x data blocks, $\boldsymbol{n} = \boldsymbol{m} * x$. Then the buffer size on which compression will be done is ***m*** characters large. The fact that the sliding buffer principle is discarded invariably makes the algorithm much easier to implement, less resource hungry and much faster to execute.

The compression algorithm LZ77 keeps track of the last **n** characters of data read, when a sub-string is encountered that has already been seen, the algorithm outputs a pair of values corresponding to the **position** of the previous encounter as well as the **length** of the matched sub-string. LZT handles the **<position, length>** match pair exactly as LZ77 does. LZT however uses some prediction, as explained below, on the actual size of the **position** portion of the **<position, length>** match pair and it also uses a variable length coding (vlc) method to encode the **length** portion of the **<position, length>** match pair.

For example, if an LZ77 encoder used a fixed sized sliding buffer of size 16384 characters, the match **position** could be encoded in 14 bits since $2^{14} = 16384$. With LZT this changes, let the position in the input buffer, of the current character to be compressed be held by ***inpos***, where ***inpos*** is initially set to 3. It is then only possible for a match to occur at position ***1*** or ***2***. Hence, the position of a match for ***inpos*** will never be greater than ***2***, and therefore the match **position** portion can be encoded using only 1 bit. As ***inpos*** is incremented as each character is encoded, the match position range increases and therefore more bits will be required to encode the match **position**. For this study, LZT uses a file system data block of 16k in size unless stated otherwise. Figure 3.1 shows all the possible bit lengths required to encode the match **position** if a maximum 16k file system data block size is used.

When decimal values are converted to binary values, we get $0_0 = 0_2$, $1_{10} = 1_2$, $2_{10} = 10_2$, etc. As a position of 0 will never be used, it is possible to develop a coding scheme where a decimal value of 1 can be represented by a binary value of 0, and a decimal value of 2 can be represented by binary value of 1. Only 1 bit is therefore needed to encode match **position** 1 and match **position** 2. In general, any decimal value **n** can be represented by the binary equivalent for **(n – 1)**. The number of bits needed to encode **(n – 1)**, indicates the number of bits needed to encode the match **position**. This sliding scale **position** method achieves about a 0.1 bits per byte (bpb) increase in compression ratio, over another method that uses a static Huffman encoding scheme. Where static Huffman is meant to refer to the two-pass method originally explained in section 2.3.

Position From	Position To	Bits Required
1	2	1
3	4	2
5	8	3
9	16	4
17	32	5
33	64	6
65	128	7
129	256	8
257	512	9
513	1024	10
1025	2048	11
2049	4096	12
4097	8192	13
8193	16384	14

Figure 3.1 - Bits required for a sliding scale position pointer.

The LZT match **length** portion is handled differently from that of the original LZ77 compression algorithm. In LZ77, a pre-determined amount of bits has to be reserved for the match **length**, exactly as with the match **position**. This however is also found to be wasteful. Two possible solutions were found by which the match **length** could be encoded more efficiently. One is to use a static Huffman encoding scheme and the other is to use a method called variable length coding or vlc, as explained below.

The static Huffman encoding scheme is the two-pass method originally explained in section 2.3. Where the frequency table is based on all the possible match **length** values that will be needed to encode the input stream. Hence, when a match **length** needs to be encoded, a prefix-free Huffman code is used, instead of fixed bit encoding.

The vlc method performs its encoding by using binary blocks. The first binary block is 3 bits long, where binary values 000 through 110 represent decimal values 1 through 7. Where binary value 0 represents decimal value 1 as previously explained when coding a match **position**. This coding scheme is possible, since no match **length** of 0 will ever be encoded. The maximum binary value of a block is used to specify whether another binary block follows the current binary block, in which case it is called the block to follow flag (bff). In this case, binary 111 specifies that there should be another binary block following this one.

Next a 4 bit binary block is appended to the existing 3 bit binary block, resulting in a 7 bit binary block, where binary 111 0000 represents decimal value 8 and where the maximum binary value of 111 1111 is meant to act as a bff. Continuing in this way the next binary block of bits are appended. Each consecutive binary block is 1 bit bigger in size than the previous binary block. The binary block size continues to grow until it reaches a size of 8 bits. At this point no further increases to the binary block size is made and all subsequent binary blocks will be 8 bits in size.

Figure 3.2 lists some example matching sub-string lengths and what they would be encoded as using vlc. The vlc method has been compared to that of the static Huffman encoding scheme and achieves nearly identical performance. The static Huffman encoding scheme however still outperforms the vlc method on files with a uniform stream of characters as those found in graphics files, which usually contain long streams of recurring bit patterns. The disadvantage of the vlc method is that it reserves too long bit streams for the longer match **length** codes, which decreases coding efficiency.

Length	VLC
1	000
2	001
7	110
8	111 0000
16	111 1000
22	111 1110
23	111 1111 00000
32	111 1111 01001
64	111 1111 11110 001010
128	111 1111 11111 111110 0001011
256	111 1111 11111 111111 1111110 00001100
512	111 1111 11111 111111 1111111 11111111 00001101

Figure 3.2 - Examples of typical variable length codes.

As with the LZ77 algorithm, the LZT encoder needs to write one or more bits to the output stream. This acts as an indication to the decoder whether the next bit stream that it reads is a literal character or a **<position, length>** match pair. These extra bits are commonly referred to as the match flags.

Code	Description
0	Both input sub-strings are literal characters
10	First a literal character then a match pair
11	Only a <position, length> match pair

Figure 3.3 – Match flag codes, based on static Huffman prefix-free codes.

The LZT algorithm was tested with match flags based on a predefined static Huffman encoding scheme as shown in Figure 3.3. This method is said to improve compression (Bloom 1995). However, the tests performed by this study indicated that this method yields no increase in compression ratio. A simpler method, where binary value 0 indicates that a literal character encoded and binary value 1 indicates that a match pair is encoded, was also tested. It was found that both methods are equivalent in compression performance. LZT therefore uses the simpler method, since it is found that this method yields a better performance in terms of speed of execution over the predefined static Huffman encoding scheme.

Variable descriptions:

hc	: the array of static Huffman prefix-free codes created as per Figure 2.11, based on all the literal characters in the data block
hl	: the array of static Huffman prefix-free codes created as per Figure 2.11, based on all the possible match lengths in the data block
b	: the sliding buffer, an array of characters from the input stream.
k	: the current character read from the input stream.
i	: the position of the character **k**, within the input stream.
w	: the sub-string to be compressed.
c	: the leftmost character of the sub-string in **w**.
p	: the position of a matching sub-string.
l	: the length of a matching sub-string.
m	: the minimum allowable length of a matching sub-string.
sc	: the no of bits used for the matching sub-string position obtained from Figure 3.1 based on **i**.

Initial variable values:

```
m ← 3
i ← 2
```

Algorithm:

```
read $k_1$ from the input stream and place into leftmost element in b
w ← $k_1$

repeat
    read $k_i$ from the input stream
    w ← w + $k_i$
    if b has no empty elements then
        shift the contents of b left by 1 element
    insert $k_i$ into first empty leftmost element in b

    if w exists previously in b then
        p ← position of the matching sub-string w within b
    else
        l ← (length of w) – 1
        if l >= m then
            write a value 1 bit to the output stream to specify a match
            write (p – 1) using sc bits to the output stream
            write $hl_l$ to the output stream
            w ← $k_i$
        else
            write a value 0 bit to the output stream to specify a literal character
            c ← leftmost character from w
            write $hc_c$ to the output stream
            w ← $2^{nd}$ leftmost character from w
            i ← position of the character w, within the input stream
        end if
    end if
    i ← i + 1
until EOF = true
```

Figure 3.4 – The LZT algorithm.

The literal characters are also encoded using a static Huffman encoding scheme. Where the frequency table is derived from the characters found in the current data block. See section 2.3 for a complete explanation of creating the static Huffman binary tree. By using the static Huffman encoding scheme, a compression performance increase of 0.289 bits is achieved over another encoding method that did not perform compression on its literal characters.

3.2 Implementing LZT

The LZT algorithm is derived by implementing different encoding methods into one algorithm. Appendix A and Figure B.10 lists the detail of each encoding method. More than one LZT routine is used to provide some means of cross-referencing the results that is obtained in the following chapter. Listed in no particular order, the LZT routines can explained as follows:

3.2.1 The base LZT routine

This is an LZT routine with no additional methods of encoding and can be explained as a LZ77 encoding process with hashing. This LZT routine executes faster than any of the other LZT routines, as far as speed of execution is concerned. It however also has the lowest compression ratio. For purposes of this study, this method of execution will be referred to as the base LZT method of execution.

3.2.2 The base LZT routine with literal character encoding

This is the base LZT routine with an additional static Huffman encoding scheme used to encode the literal characters of the output stream. Comparative to the base LZT routine, this routine yields an average improvement in the compression ratio of 0.289 bpb over the test data set, as per Chapter 4. For this reason alone, Huffman has become a big part of many compression algorithms including LZT, in that it can significantly increase the compression ratio.

3.2.3 The base LZT routine with match flag encoding

This is the base LZT routine with an additional match flag blocking as per Figure 3.3. While this study's findings did indicate that this added encoding method yields an average improvement in the compression ratio of 0.003 bpb. It is regarded as negligible given the extra execution time required to process the match flag blockings.

3.2.4 The base LZT routine with match position encoding

These are two base LZT routines each with additional encoding methods to compress the match **position** portion of the <**position**, **length**> match pair. One of these two LZT routines implements a sliding scale position pointer method to encode its match **position**. This method yields an average improvement in compression ratio of 0.109 bpb.

The other LZT routine implements a static Huffman encoding scheme based on all the possible match positions of the output stream. This method however, yields an average loss in compression ratio of 0.351 bpb compared to the sliding scale position pointer method. The final LZT routine therefore uses the sliding scale position pointer encoding method, since this method provides both faster speed of execution and compression ratio than the other routine based on the Huffman encoding scheme.

3.2.5 The base LZT routine with match length encoding

These are two base LZT routines each with additional encoding methods to compress the match **length** portion of the <**position**, **length**> match pair. One of these two LZT routines implements the vlc method to encode its match **length**. This method yields an average improvement in compression ratio of 1.157 bpb.

The other LZT routine implements a static Huffman encoding scheme based on all the possible match lengths of the output stream. This method yields an average improvement in compression ratio of 1.275 bpb. The final LZT routine therefore uses the static Huffman encoding scheme, since this method provides a better compression ratio than the other routine based on the vlc encoding method.

It must be noted that the static Huffman encoding scheme which is used to encode the match **position** and **length** portions of the <**position**, **length**> match pair, is inherently slow to execute since excessive use of linked lists is made. With these, excessive linked lists, excessive programming loops is used in order to process them and therefore such poor execution times for both methods are recorded.

3.2.6 The final LZT routine

The final LZT routine as explained in section 3.1, is based on the base LZT routine with an additional static Huffman encoding scheme used to encode the literal characters, an additional sliding scale position pointer method to encode its match **position** and an additional static Huffman encoding scheme used to encode its match **length**. The result of which produces a high performance compression algorithm with superior results in both speed of execution and compression ratio.

Chapter 4 - Benchmarking LZT

4.1 The Calgary Corpus

The Calgary/Canterbury compression corpus is used to evaluate the compression performance of various compression schemes. Several other researchers are now using the corpus to evaluate their compression schemes. For this reason, the corpus is chosen as the data set for the benchmarking of this study.

The corpus comprises of these text files, namely:

book1, book2, paper1, paper2, paper3, paper4, paper5, paper6, bib, news, progc, progl, progp, trans

The corpus also includes these binary files, namely:

obj1, obj2, geo, pic,

To confirm that the performance of schemes is consistent for any given type, many of the types have more than one representative. Normal English text, both fiction and non-fiction, are represented by two books and several papers, called book1, book2, paper1, paper2, paper3, paper4, paper5 and paper6. More unusual styles of English text can be found in a bibliography, called bib and a batch of unedited news articles, called news.

Three computer programs representing programming languages called progc, progl and progp, are also included. A transcript of a terminal session, called trans is included to indicate the increase in speed that could be achieved by applying compression to a slow line to a computer terminal.

Two files of executable code, called obj1 and obj2, some geophysical data called geo, and a black and white bitmapped picture, called pic can be found. The file geo is particularly difficult to compress because it contains a wide range of data values, while the file pic is highly compressible because of large amounts of white space in the picture, represented by long runs of zeros.

4.2 Benchmarks

Various compression algorithms and LZT encoding methods were used to draw up this benchmark. The compression ratios in terms of bpb are reported on. Speed in kilobytes per second (kbps) is also reported on, and is measured, rounded to the nearest kilobyte. The speed is also measured with all disk accesses. The hardware used for the benchmarking is an Intel Pentium, running a 150 MHz processor with 96Mb RAM. No disk or memory caches were used, in order to determine a uniform result. All tests were performed using executable code running under Microsoft DOS 6.22.

Figure 4.1 and Figure 4.2 list the compression results for the various algorithms. The list is in order of total average compression ratio from left to right, in decreasing order. Figure 4.3 shows the average compression ratio and speed per compression algorithm.

It was found by this study that when the throughput of data to and from a file system drops to below 150 kbps, the file system suffers from lag, which will mean a loss in speed in the file system processing. Such a loss will not outweigh the benefits of an actual saving in disk space. Many users would rather have less disk space than less speed. It is therefore concluded that a compression algorithm will have to average at least a 150 kbps throughput on the benchmarking hardware, in order to have a negligible effect on the file system in question. From Figure 4.2 one can clearly see that PKZip, developed by PKWare USA, still has the highest compression ratio, but unfortunately, it has a low 115 kbps throughput.

The LZS algorithm, produced by STAC Electronics is currently the most well known file system, real-time compression algorithm. This algorithm is used in the famous Stacker Double-Disk software and in the new Microsoft Double-Space software. As one can see in Figure 4.2, it performs poorly with regard to compression ratio in comparison with some of the LZT encoding methods. The LZRW1 algorithm (Williams, 1987) improves over LZS, but it too does not achieve quite the compression ratio of the LZT algorithm. LZP (Bloom, 1995), which is an as yet unpublished algorithm, is also included in the benchmarking.

File	LZT4	LZ77	LZT2	LZT3	LZT1	LZT5	LZT6
bib	5.273	5.004	5.020	4.898	4.671	3.758	3.712
book1	6.679	6.384	6.442	6.250	5.925	4.566	4.390
book2	5.778	5.482	5.525	5.363	5.104	4.015	3.905
geo	6.300	6.288	6.117	6.176	6.204	6.162	4.790
news	5.868	5.536	5.525	5.440	5.080	4.180	4.175
obj1	5.309	5.103	4.948	5.024	5.110	4.209	4.294
obj2	4.543	4.274	4.202	4.195	4.204	3.441	3.357
paper1	5.672	5.380	5.410	5.252	5.013	3.965	3.912
paper2	5.982	5.707	5.763	5.583	5.327	4.107	3.995
paper3	6.224	5.906	5.955	5.773	5.488	4.277	4.157
paper4	6.101	5.806	5.843	5.650	5.391	4.243	4.154
paper5	6.051	5.684	5.694	5.527	5.247	4.296	4.229
paper6	5.527	5.264	5.291	5.136	4.907	3.913	3.849
pic	1.347	1.319	1.299	1.291	1.269	1.066	1.084
progc	5.274	5.017	5.031	4.895	4.686	3.753	3.741
progl	3.759	3.579	3.607	3.486	3.349	2.665	2.667
progp	3.643	3.443	3.459	3.369	3.244	2.586	2.619
trans	3.832	3.628	3.621	3.545	3.388	2.791	2.828
Ratio (bpb)	**5.176**	**4.934**	**4.931**	**4.825**	**4.645**	**3.777**	**3.659**
Speed (kbps)	**180**	**345**	**200**	**185**	**300**	**180**	**10**

Figure 4.1 - Various Calgary Corpus compression results.

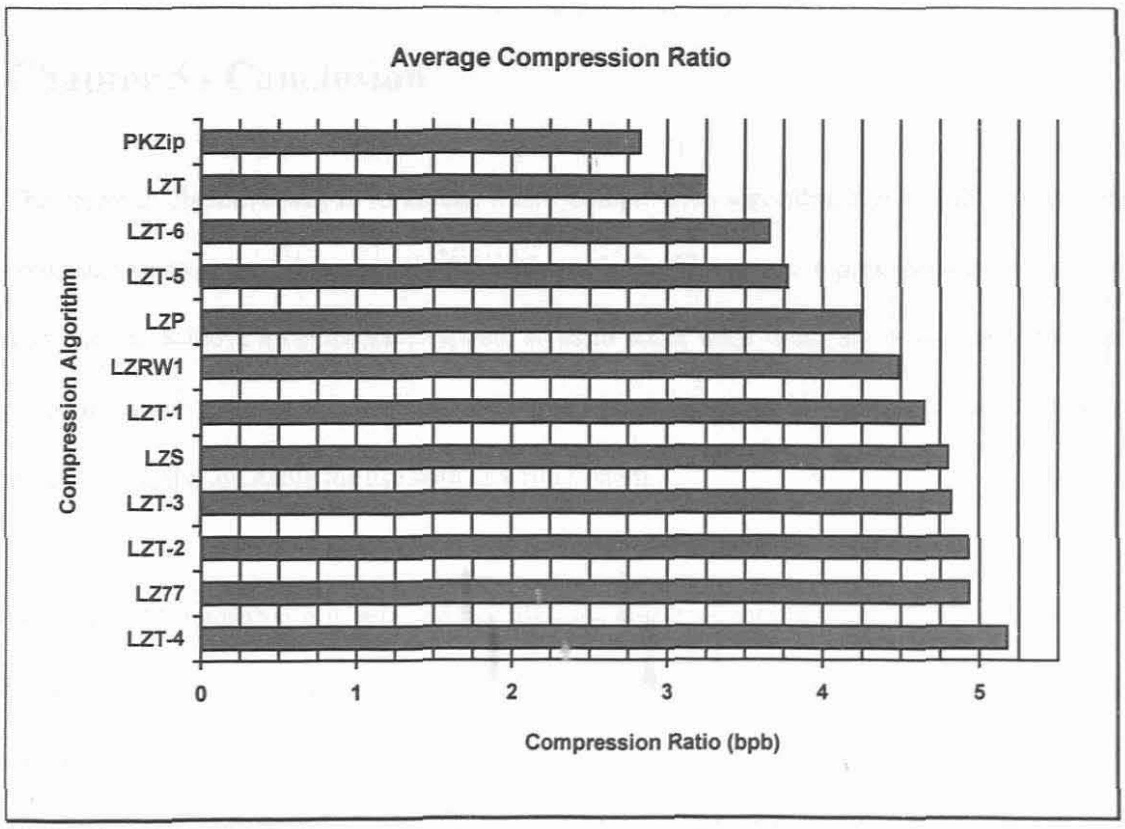
Various encoding methods implemented in LZT were included in order to demonstrate the efficiency or lack thereof, of the various encoding methods. For example, where encoding method 4 is the fastest but least efficient in term of compression ratio whereas encoding method 6 is the slowest but has the most efficient compression ratio. From the results, it can be deduced that by combining encoding methods 1, 3 and 6 into one algorithm, a high performance compression algorithm with superior results in both speed of execution and compression ratio would be produced. See Appendix A for a full lists the exact nature of each encoding method implemented in LZT.

The LZT algorithm was developed using the Pascal, programming language, which is not optimal in performance as opposed to the assembler, programming language. If one were to redevelop it in a low-level language such as assembler, one would get an even better performance in terms of kbps.

File	LZS	LZRW1	LZP	LZT	PKZip
bib	5.247	4.753	4.146	3.264	3.096
book1	5.849	5.434	5.718	3.781	3.683
book2	5.117	4.717	4.563	3.395	3.150
geo	6.810	6.754	6.799	4.573	5.475
news	5.170	4.907	4.777	3.609	3.426
obj1	4.556	4.931	4.861	4.214	3.864
obj2	3.982	4.100	3.765	3.196	2.889
paper1	5.117	4.627	4.526	3.406	3.114
paper2	5.391	4.880	4.890	3.473	3.246
paper3	5.817	4.756	4.658	3.595	3.369
paper4	5.155	4.525	4.723	3.562	3.317
paper5	5.874	4.575	4.238	3.628	3.321
paper6	5.175	4.763	4.713	3.353	3.043
pic	1.585	2.045	1.397	0.999	0.856
progc	4.533	4.368	4.147	3.274	2.931
progl	3.618	3.496	2.956	2.335	2.004
progp	3.473	3.426	2.859	2.326	1.989
trans	3.972	3.687	2.639	2.497	2.231
Ratio (bpb)	**4.802**	**4.486**	**4.243**	**3.249**	**3.056**
Speed (kbps)	**280**	**550**	**630**	**180**	**115**

Figure 4.2 - Various Calgary Corpus compression results.

A test was also done using a disk cache with the LZT algorithm and it is found that a 100% to 110% increase in performance could be accomplished on the test hardware. Running a disk cache with PKZip yielded more or less the same results. This would mean that the PKZip algorithm would be more suitable to the research objectives. It is however concluded that the LZT algorithm is a better choice, since not all file systems would be using a disk cache.



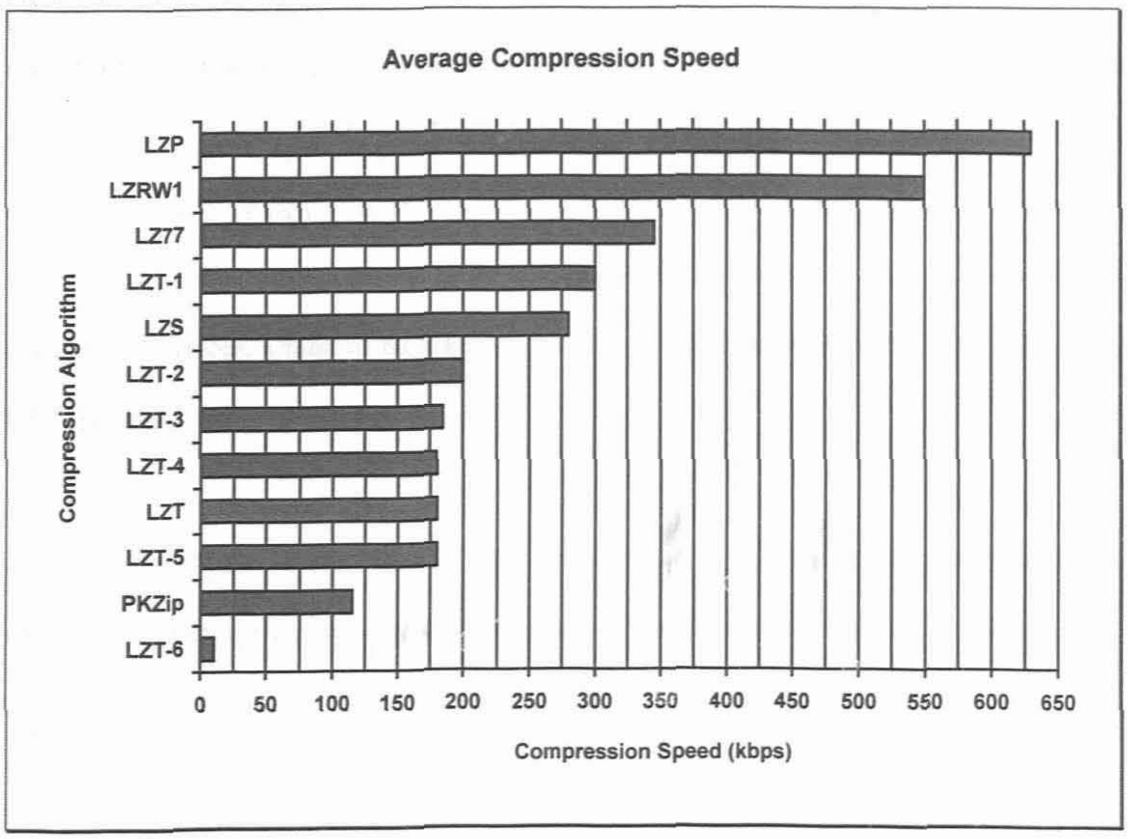


Figure 4.3 - Average Compression Ratio and Speed per Compression Algorithm.

Chapter 5 - Conclusion

The research objective was to formulate a new compression algorithm that would yield a better compression ratio than the currently available methods. The new compression algorithm would also need to achieve a compression speed, so as to make itself transparent to a user. The third problem faced was that not all file systems are created equally. Therefore, the new algorithm should be easily adaptable for use with any file system.

With the LZT algorithm it is believed that all three objectives of the study has been accomplish. It has been proven in section 4.2 that the LZT algorithm outperforms most currently available compression algorithms in terms of compression ratio. Not even the world famous LZS algorithm yields results comparable to that of LZT algorithm. It has also been proven that LZT executes faster than the minimum required speed of 150 kbps. LZT in its Pascal form produces an executable program, which is less than 60 kilobytes and can therefore by easily implemented as a device driver or operating system extension without any extreme memory overhead. The static Huffman used to encode the literal characters of LZT users less than 8 kilobytes at any one time. Therefore, a total of 68 kilobytes should be sufficient to implement and execute the LZT algorithm.

Thus, it can be concluded that the LZT compression algorithm satisfies all the research objectives in that it provides adequate solutions to all the problems originally stated.

During this study, it has been noted that arithmetic compression holds lots of potential in that it is currently the cutting-edge as far as data compression goes. The disadvantage however of this algorithm is that it uses excessive amounts of CPU power and system resources in order to perform the actual compression. However with the increase in speed of current day microprocessors, it might soon be possible to produce an arithmetic compression algorithm for use in a real-time environment. The resulting algorithm most probably would have a performance better than that of the LZT algorithm.

References

[1] Arimura, M. Yamamoto, H. 1998. **The asymptotic optimality of the block sorting data compression algorithm.** IEICE Transactions on Fundamentals, E81-A(10), 2117-2122.

[2] Bloom, C. 1995. **LZP, A new data compression algorithm**. Unpublished paper.

[3] Balakrishnan, V. K. 1991. **Introduction to Discrete mathematics**. Prentice-Hall International Editions. ISBN 0-13-478678-5.

[4] Cormack, G. V. Horspool, R. N. 1987. **Data Compression using Dynamic Markov Modelling**. Computer Journal. (12).

[5] Deutsch, P, 1996. **DEFLATE Compressed Data Format Specifications version 1.3**. Network Working Group, RFC 1951, 1-16.

[6] Elias, P. 1975. **Universal Codeword Sets and Representations of the Integers.** IEEE Transactions on Information Theory. 21(2), 194-203.

[7] Elias, P. 1987. **Interval and Recency Rank Source Coding.** IEEE Transactions on Information Theory. 33(1), 3-10.

[8] Fano, R. M. 1949. **Transmission of Information.** Cambridge, M.I.T. Press.

[9] Hankerson, D. Harris, G. A. Johnson Jnr, P. D. 1997. **Introduction to Information Theory and Data Compression**. ISBN 0-8493-3985-5.

[10] Huffman, D. A. 1952. **A Method for the Construction of Minimum-Redundancy Codes.** Procedures of the IRE. 40(9), 1098-1101.

[11] Johnsonbaugh, R. **Discrete Mathematics (3rd edition).** MacMillan Publishing Company. ISBN 0-02-360721-1.

[12] Khalid, S. 1996. **Introduction to Data Compression**. Morgan Kaufmann Publishers. ISBN 1-55860-346-8.

[13] Larmore, L. L. Hirshberg, D. S. 1990. **A fast Algorithm for Optimal Length-Limited Huffman Codes**. Journal of the ACM. 37(3), 464-473.

[14] Lelewer, D. A. Hirshberg, D. S. 1990. **Efficient Decoding of prefix Codes**. Communications of the ACM. 33(4), 449-458.

[15] Gailly, J.L. Nelson, M. R. 1995. **The Data Compression Book**. M&T Books. ISBN 1-55851-434-1.

[16] Matias, Y. Rajppot, N. Sahinalp, S. C. 1999. **The effect of Flexible parsing for dynamic dictionary based data compression**. IEEE Data Compression Conference.

[17] Nelson, M. R. 1996. **Priority Queues and the STL**. Dr Dobb's Journal. 243(1), 18-26, 96.

[18] Salomon, D. 1997. **Data Compression: The Complete Reference**. ISBN 0-387-98280-9.

[19] Shannon, C. E. Weaver, W. 1949. **The Mathematical Theory of Communication.** Urbana, Il1, University Of Illinois Press.

[20] Schindlers, M. 1997. **A Fast Block Sorting Algorithm for Lossless Data Compression**. IEEE Data Compression Conference.

[21] Storer, J. A. Szymanski, T. G. 1982. **Data Compression via textual Substitution**. Journal of the ACM. 29(4), 928-951.

[22] Schwartz, E. S. Kallick, B. 1964. **Generating a Canonical Prefix Encoding**. Communications of the ACM. 7(3), 166-169.

[23] Welch, T. A. 1984. **A Technique for High-Performance Data Compression.** IEEE Computer. 17(6), 8-19.

[24] Yokoo, H. 1992. **Improved Variations Relating the Ziv-Lempel and Welch-Type Algorithms for Sequential Data-Compression.** IEEE Transactions on Information Theory. 38(1), 73-81.

[25] Yu, T. L. 1996. **Dynamic Markov Compression**. Dr Dobb's Journal. 243(1), 30-32, 96-100.

[26] Ziv, J. Lempel, A. 1977. **A Universal Algorithm for Sequential Data Compression.** IEEE Transactions on Information Theory. 23(3), 337-343.

[27] Ziv, J. Lempel, A. 1978. **Compression of Individual Sequences via Variable-Rate Coding.** IEEE Transactions on Information Theory. 24(5), 530-536.

[28] http://www.faqs.org/faqs/compression-faq/index.html Internet Data Compression FAQ.

[29] http://www.internz.com/compression-pointers.html A list of links to data compression sites and documents.

Glossary

Algorithm: Pseudo code used to represent the workings of an information processing process.

bpb: Bits per byte. A byte usually contains 8 bits.

File system: A logical entity used to store computer files in which is usually implemented on a computers disk drive.

k: Kilobyte, where 1024 bytes makeup a kilobyte

kbps: Kilobytes per second. Equal to 1024 bytes per second.

Loss-less: No information loss occurs. We merely represent the data in less space.

Real-time: A process whereby events take place without any delay.

Appendix A - LZT parameter settings

Method Flag	1	2	3	4	5	6	LZT
Huffman Literals	✓						✓
Match Flag Blocking		✓					
Growing Coding Position			✓				✓
Huffman Position				✓			
Variable Coding Length					✓		
Huffman Length						✓	✓

Figure A.1 - The different settings for the LZT compression algorithms.

Appendix B - LZT source listings

The source code used for the various LZT implementations is given below. The source code is compiled with a Turbo Pascal 7.0 compiler under Windows NT 4.0 with service pack 4 installed. The hardware used is an Intel Pentium, running a 150 MHz processor with 96Mb RAM. Note that there is not much documentation explaining the source code. However care has been taken in order to make it easily understandable. The Basic Unit as shown in Figure B.1, contains all those miscellaneous variables and routines used by the LZT program. Figure B.2 lists the Clock Unit, which is used to measure compression performance of the LZT program.

```
Unit Basic;

Interface

Uses Dos;

Const
 HCMethod : Boolean = False;
 MFBMethod : Boolean = False;
 GCPMethod : Boolean = False;
 HPMethod : Boolean = False;
 VCLMethod : Boolean = False;
 HLMethod : Boolean = False;

 T8BitDic  = 255;
 T14BitDic = 16383;

 InBufferLen  = 16384;
 OutBufferLen = (InBufferLen Div 4) + InBufferLen;

 Pow : Array [0..14] Of Word =
        ( 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384
);

Var
 SingleFreq : Boolean;
 RunType : Char;

 InBuffer : Array [1..InBufferLen] of Byte;
 OutBuffer : Array [1..OutBufferLen] of Byte;

 InPos, OutPos, OutSize : LongInt;
 OutBitsUsed, Pass : Byte;

 MatchPos, MatchPosBits : Word;
 MatchFound : Boolean;
 MatchLen, MinMatchLen, MatchLenBits, CompLen : Word;
 LiteralChar : Word;
 LiteralBits : Byte;

Function Power2X(X : Integer) : LongInt;
```

```
Implementation

Function Power2X(X : Integer) : LongInt;
Begin
Power2X := Pow[X];
End;

End.
```

Figure B.1 - The Basic Unit.

```
Unit Clock;

Interface

Uses Dos;

Type
 TTime = Record
          H, M, S, I : Word;
          End;

Var
 STime, CTime : TTime;
 SClock, CClock : Real;
 L : LongInt;
 EClock : Real;

Procedure StartClock;
Procedure WriteClock;

Implementation

{ Start the timers clock }
Procedure StartClock;
Begin
GetTime(STime.H, STime.M, STime.S, STime.I);
L := STime.H;
SClock := (L * 3600) + (STime.M * 60) + STime.S + (STime.I / 100);
End;

{ Writes the elapsed time }
Procedure WriteClock;
Begin
GetTime(CTime.H, CTime.M, CTime.S, CTime.I);
L := CTime.H;
CClock := (L * 3600) + (CTime.M * 60) + CTime.S + (CTime.I / 100);
EClock := CClock - SClock;
If EClock = 0 Then
 EClock := 0.001;
{WriteLn('Elapsed time      : ', EClock:7:3, ' seconds');}
End;

End.
```

Figure B.2 - The Clock Unit.

The Compress Unit as shown in Figure B.3 is used to manage the whole compression execution as selected via the command-line flags from the LZT program.

```
Unit Compress;

Interface

Uses Basic, Clock, IO, Hash, Huffman, VLC, Flag;

Var
 LiteralsHuf : TLitRecArrayPtr;
 LiteralsFreq : TFreqRecArrayPtr;

 PosHuf : THufRecArrayPtr;
 PosFreq : TFreqRecArrayPtr;

 LenHuf : THufRecArrayPtr;
 LenFreq : TFreqRecArrayPtr;

 MaxFreq : Word;

Procedure CompressBlock;

Implementation

Procedure ResetVars;
Begin
MatchFound := False;
MatchPos := 0;
MatchLen := 0;

If HCMethod Then
 Begin
 LiteralChar := LiteralsHuf^[Ord(InBuffer[InPos])].Code;
 LiteralBits := LiteralsHuf^[Ord(InBuffer[InPos])].Len;
 End
Else
 Begin
 LiteralChar := Ord(InBuffer[InPos]);
 LiteralBits := 8;
 End;

If GCPMethod Then
 Begin
 If InPos > Power2X(MatchPosBits) Then
  Inc(MatchPosBits);
 End
Else
 MatchPosBits := 14;

If Not(VCLMethod) Then
 Begin
 MatchLenBits := 14;
 MinMatchLen := MatchPosBits - MatchLenBits - 2;
 End;
End;

Procedure SearchBuffer;

Var
 S : HashStr;
 P, P1, P2, W : Word;
 L : LongInt;

Begin
```

```
ResetVars;
{ Ignore the last 2 input bytes as it is no use compressing them, so just }
{ write them out as normal literals                                        }
If InPos >= InFileSize - 1 Then
 Exit;

For P := 1 To HashLen Do
 S[P] := InBuffer[InPos + P - 1];

{ Don't do any searches if we at input position 1 as no hashes has been }
{ setup yet                                                             }
If InPos = 1 Then
 Begin
 If Pass = 1 Then
  AddHash(S, InPos);
 Exit;
 End;

P := FindFirstHash(S);
{ If we at the 3rd last byte then ouput immediately as the match length }
{ will not get bigger than 3 for obvious reasons                        }
If InPos = InFileSize - 2 Then
 Begin
 If P <> 0 Then
  Begin
  MatchFound := True;
  MatchPos := P;
  MatchLen := HashLen;
  End;
 End
Else
 While (P <> 0) And (P < Inpos) Do
  Begin
  L := HashLen;
  P1 := P;
  P2 := InPos;

  While (InBuffer[P1 + L] = InBuffer[P2 + L]) And
        ((P2 + L) < InFileSize) Do
   Inc(L);

  If InBuffer[P1 + L] = InBuffer[P2 + L] Then
   Inc(L);

  If VCLMethod Then
   Begin
   MatchLenBits := GetVLCLen(L);
   MinMatchLen := MatchPosBits + MatchlenBits + 2;
   End;

  If (L > MatchLen) And ((L * 8) >= MinMatchLen) Then
   Begin
   MatchFound := True;
   MatchPos := P;
   MatchLen := L;
   End;

  P := FindNextHash;
  End;

If Pass = 1 Then
 AddHash(S, InPos);
End;

Procedure Encode;
Begin
Inc(Pass);
SearchIdx := 1;

If GCPMethod Then
```

```
 MatchPosBits := 1;
Repeat
 SearchBuffer;
 CompLen := MatchLen;

 If MatchFound Then
  Begin
  If HPMethod Then
   Begin
   MatchPosBits := PosHuf^[MatchPos - 1].Len;
   MatchPos := PosHuf^[MatchPos - 1].Code;
   End;

  If HLMethod Then
   Begin
   MatchLenBits := LenHuf^[MatchLen - 1].Len;
   MatchLen := LenHuf^[MatchLen - 1].Code;
   End;
  End;

 OutputFlagResult
 Until InPos > InFileSize;
End;

Procedure CreatePosHuf;

Var
 P : Word;
 MaxPosWeight : LongInt;

Begin
Inc(Pass);
New(PosFreq);

MaxPosWeight := 0;

For P := 0 To 16383 Do
 PosFreq^[P].Weight := 0;

Repeat
 SearchBuffer;

 If MatchFound Then
  Begin
  Inc(PosFreq^[MatchPos - 1].Weight);

  If PosFreq^[MatchPos - 1].Weight > MaxPosWeight Then
   MaxPosWeight := PosFreq^[MatchPos - 1].Weight;

  InPos := InPos + MatchLen;
  End
 Else
  Inc(InPos);
 Until InPos > InFileSize;

{ Scale frequency counts to limit the huffman codes to 16 bits }
MaxPosWeight := (MaxPosWeight Div 255) + 1;

For P := 0 To 16383 Do
 Begin
 If ((PosFreq^[P].Weight Div MaxPosWeight) = 0) And
     (PosFreq^[P].Weight <> 0) Then
  PosFreq^[P].Weight := 1
 Else
  PosFreq^[P].Weight := PosFreq^[P].Weight Div MaxPosWeight;

{ Initialize weight table }
 If PosFreq^[P].Weight = 0 Then
  PosFreq^[P].Weight := $FFFF
 Else
```

```
  MaxFreq := P;
 End;
PosHuf := CreateHuffArray(PosFreq, MaxFreq);
Dispose(PosFreq);
OutputHufDic(PosHuf, MaxFreq);

InPos := 1;
End;

Procedure CreateLenHuf;

Var
 P : Word;
 MaxLenWeight : LongInt;

Begin
Inc(Pass);
New(LenFreq);

MaxLenWeight := 0;

For P := 0 To 16383 Do
 LenFreq^[P].Weight := 0;

Repeat
 SearchBuffer;

 If MatchFound Then
  Begin
  Inc(LenFreq^[MatchLen - 1].Weight);

  If LenFreq^[MatchLen - 1].Weight > MaxLenWeight Then
   MaxLenWeight := LenFreq^[MatchLen - 1].Weight;

  InPos := InPos + MatchLen;
  End
 Else
  Inc(InPos);
 Until InPos > InFileSize;


{ Scale frequency counts to limit the huffman codes to 16 bits }
MaxLenWeight := (MaxLenWeight Div 255) + 1;

For P := 0 To 16383 Do
 Begin
 If ((LenFreq^[P].Weight Div MaxLenWeight) = 0) And
     (LenFreq^[P].Weight <> 0) Then
  LenFreq^[P].Weight := 1
 Else
  LenFreq^[P].Weight := LenFreq^[P].Weight Div MaxLenWeight;

{ Initialize weight table }
 If LenFreq^[P].Weight = 0 Then
  LenFreq^[P].Weight := $FFFF
 Else
  MaxFreq := P;
 End;
LenHuf := CreateHuffArray(LenFreq, MaxFreq);
Dispose(LenFreq);
OutputHufDic(LenHuf, MaxFreq);

InPos := 1;
End;

Procedure CompressBlock;
Begin
InitializeIO;
InitHash;
Pass := 0;
```

```
If HCMethod Then
 Begin
 New(LiteralsFreq);
 CreateInBufFreqArray(LiteralsFreq, MaxFreq);
 LiteralsHuf := CreateHuffArray(LiteralsFreq, MaxFreq);
 Dispose(LiteralsFreq);
 OutputHufDic(LiteralsHuf, MaxFreq);
 End;

If HPMethod Then
 CreatePosHuf;

If HLMethod Then
 CreateLenHuf;

If Not(SingleFreq) Then
 Encode;

If HCMethod Then
 DisposeHuffArray(LiteralsHuf, T8BitDic);

If HPMethod Then
 DisposeHuffArray(PosHuf, T14BitDic);

If HLMethod Then
 DisposeHuffArray(LenHuf, T14BitDic);

DisposeHash;
End;

End.
```

Figure B.3 - The Compress Unit.

The Flag Unit as shown in Figure B.4 is used to implement the match flag binary coding routine as described with Figure 3.3. Figure B.5 lists the Hash Unit. This unit does the fast dictionary searches. It achieves overwhelming performance by implementing a three dimensional dictionary. In the last dimension, it implements a two character dictionary.

```
Unit Flag;

Interface

Uses Dos, Basic, IO, VLC;

Type
 TMatchRec = Record
               MatchPos, MatchPosBits,
               MatchLen, MatchLenBits : Word;
               LiteralChar : Word;
               LiteralBits : Byte;
               End;

Var
 SearchRes : Array [1..2] Of TMatchRec;
 SearchIdx : Byte;
```

```
Procedure OutputFlagResult;

Implementation

Procedure OutputFlagResult;
Begin
If MatchFound Then
 Begin
 If MFBMethod Then
  Begin
  SearchRes[SearchIdx].MatchPos := MatchPos;
  SearchRes[SearchIdx].MatchPosBits := MatchPosBits;
  SearchRes[SearchIdx].MatchLen := MatchLen;
  SearchRes[SearchIdx].MatchLenBits := MatchLenBits;
  SearchRes[SearchIdx].LiteralChar := 0;
  SearchRes[SearchIdx].LiteralBits := 0;
  End
 Else
  Begin
  OutputSingleBit(1);
  OutputBits(MatchPos, MatchPosBits);
  OutputBits(MatchLen, MatchLenBits);
  End;

 InPos := InPos + CompLen;
 End
Else
 Begin
 If MFBMethod Then
  Begin
  SearchRes[SearchIdx].MatchPos := 0;
  SearchRes[SearchIdx].MatchPosBits := 0;
  SearchRes[SearchIdx].MatchLen := 0;
  SearchRes[SearchIdx].MatchLenBits := 0;
  SearchRes[SearchIdx].LiteralChar := LiteralChar;
  SearchRes[SearchIdx].LiteralBits := LiteralBits;
  End
 Else
  Begin
  OutputSingleBit(0);
  OutputBits(LiteralChar, LiteralBits);
  End;

 Inc(InPos);
 End;

If (MFBMethod) Then
 Begin
 Inc(SearchIdx);

 If (SearchIdx = 3) Then
  If (SearchRes[1].LiteralChar <> 0) And
     (SearchRes[2].LiteralChar <> 0) Then
   Begin
   OutputSingleBit(0);
   OutputBits(SearchRes[1].LiteralChar, SearchRes[1].LiteralBits);
   OutputBits(SearchRes[2].LiteralChar, SearchRes[2].LiteralBits);
   SearchIdx := 1;
   End
  Else
   If (SearchRes[1].LiteralChar <> 0) And
      (SearchRes[2].LiteralChar = 0) Then
    Begin
    OutputBits(2, 2);
    OutputBits(SearchRes[1].LiteralChar, SearchRes[1].LiteralBits);
    OutputBits(SearchRes[2].MatchPos, SearchRes[2].MatchPosBits);

    If SearchRes[2].MatchLenBits <> 0 Then
     OutputBits(SearchRes[2].MatchLen, SearchRes[2].MatchLenBits)
    Else
```

```
     OutputVLC(SearchRes[2].MatchLen);

    SearchIdx := 1;
    End
   Else
    Begin
    OutputBits(3, 2);
    OutputBits(SearchRes[1].MatchPos, SearchRes[1].MatchPosBits);

    If SearchRes[1].MatchLenBits <> 0 Then
     OutputBits(SearchRes[1].MatchLen, SearchRes[1].MatchLenBits)
    Else
     OutputVLC(SearchRes[1].MatchLen);

    SearchRes[1].MatchPos      := SearchRes[2].MatchPos;
    SearchRes[1].MatchPosBits  := SearchRes[2].MatchPosBits;
    SearchRes[1].MatchLen      := SearchRes[2].MatchLen;
    SearchRes[1].MatchLenBits  := SearchRes[2].MatchLenBits;
    SearchRes[1].LiteralChar   := SearchRes[2].LiteralChar;
    SearchRes[1].LiteralBits   := SearchRes[2].LiteralBits;
    SearchIdx := 2;
    End;
 End;
End;

End.
```

Figure B.4 - The Flag Unit.

```
Unit Hash;

Interface

Uses Basic;

Const
 HashLen = 3;

Type
 THashSearchRecPtr = ^THashSearchRec;
 THashSearchRec = Record
                   Ch : Byte;
                   Position : Word;
                   Next : THashSearchRecPtr;
                   End;

 THashLevelRecPtr = ^THashLevelRec;
 THashLevelRec = Record
                  NextLevel : Array [0..T8BitDic] Of Pointer;
                  End;

 HashStr = Array[1..HashLen] Of Byte;

Procedure InitHash;
Procedure AddHash(NewHash : HashStr; HashPos : Word);
Procedure DisposeHash;

Function FindFirstHash(FindHash : HashStr) : Word;
Function FindNextHash : Word;

Implementation

Var
 RootHash : Array [0..T8BitDic] Of Pointer;
 SecondLevelHash, NewLevelHash : THashLevelRecPtr;
 CurrentHash : HashStr;

 CCol, CRow : Byte;
 SHashPtr, CHashPtr, NHashPtr, CRowHash : THashSearchRecPtr;

{ Initialize the hashing system }
Procedure InitHash;
Begin
New(NewLevelHash);
For CCol := 0 To T8BitDic Do
 Begin
 RootHash[CCol] := Nil;
 NewLevelHash^.NextLevel[CCol] := Nil;
 End;
End;

{ Add a new hash to the current list }
Procedure AddHash(NewHash : HashStr; HashPos : Word);
Begin
CCol := NewHash[1];
If RootHash[CCol] = Nil Then
 Begin
 New(SecondLevelHash);
 RootHash[CCol] := SecondLevelHash;
 SecondLevelHash^.NextLevel := NewLevelHash^.NextLevel;
 End
Else
 SecondLevelHash := RootHash[CCol];

New(NHashPtr);
NHashPtr^.Ch := NewHash[3];
NHashPtr^.Position := HashPos;
NHashPtr^.Next := Nil;

CRow := NewHash[2];
If SecondLevelHash^.NextLevel[CRow] = Nil Then
 SecondLevelHash^.NextLevel[CRow] := NHashPtr
Else
 Begin
 SHashPtr := SecondLevelHash^.NextLevel[CRow];
 CHashPtr := SHashPtr;

 While CHashPtr^.Next <> Nil Do
  CHashPtr := CHashPtr^.Next;

 CHashPtr^.Next := NHashPtr;
 End;
End;

Procedure DisposeHash;
Begin
For CRow := 0 To T8BitDic Do
 If RootHash[CRow] <> Nil Then
  Begin
  SecondLevelHash := RootHash[CRow];

  For CCol := 0 To T8BitDic Do
   Begin
   SHashPtr := SecondLevelHash^.NextLevel[CCol];

   While SHashPtr <> Nil Do
    Begin
    CHashPtr := SHashPtr;
    SHashPtr := SHashPtr^.Next;
    Dispose(CHashPtr);
    End;
   End;

  Dispose(SecondLevelHash);
  End;

Dispose(NewLevelHash);
End;

{ Find the first position using the current hash }
Function FindFirstHash(FindHash : HashStr) : Word;
Begin
CurrentHash := FindHash;

If RootHash[CurrentHash[1]] = Nil Then
 Begin
 FindFirstHash := 0;
 Exit;
 End;

SecondLevelHash := RootHash[CurrentHash[1]];
CCol := CurrentHash[2];
CRowHash := SecondLevelHash^.NextLevel[CCol];

If CRowHash = Nil Then
 FindFirstHash := 0
Else
 FindFirstHash := CRowHash^.Position;
End;

{ Find the next position using the current hash }
Function FindNextHash : Word;
Begin
CRowHash := CRowHash^.Next;

If CRowHash = Nil Then
 FindNextHash := 0 { We are at the end of a row }
Else
 FindNextHash := CRowHash^.Position;
End;

End.
```

Figure B.5 - The Hash Unit.

The Huffman Unit as shown in Figure B.6 is used to implement the various routines associated with the creation of the Huffman binary trees.

```
Unit Huffman;

Interface

Uses Dos, Basic, IO;

Const
 Dic0Pad4 = 15; { binary 00 1111 }
 Dic0Pad8 = 31; { binary 01 1111 }
 DicXPad4 = 47; { binary 10 1111 }
 DicXPad8 = 63; { binary 11 1111 }

 MaxFreqRecs  = 16383;
 MaxCanonRecs = 16;

```

```
Type
 THufRec = Record
            Code : Word;

            Len : Byte;
            End;

 TLitRecArrayPtr = ^THufRecArray;
 TLitRecArray = Array [0..255] Of THufRec;

 THufRecArrayPtr = ^THufRecArray;
 THufRecArray = Array [0..MaxFreqRecs] Of THufRec;

 TPtrRec = Record
            Next : Word;
            End;

 TPtrRecArrayPtr = ^TPtrRecArray;
 TPtrRecArray = Array [0..MaxFreqRecs] Of TPtrRec;

 TFreqRec = Record
             Weight : Word;
             End;

 TFreqRecArrayPtr = ^TFreqRecArray;
 TFreqRecArray = Array [0..MaxFreqRecs] Of TFreqRec;

 TCanonRec = Record
              LenCount, NextCode : Word;
              End;

 TCanonRecArrayPtr = ^TCanonRecArray;
 TCanonRecArray = Array [1..MaxCanonRecs] Of TCanonRec;

Procedure  CreateInBufFreqArray(Var  Freq  :  TFreqRecArrayPtr;  Var  MaxFreq  :
Word);
Function CreateHuffArray(Freq : TFreqRecArrayPtr; Size : Word) : Pointer;
Procedure OutputHufDic(P : Pointer; Size : Word);
Procedure DisposeHuffArray(P : Pointer; HufSize : Word);

Implementation

{ Create frequency table of the literals in the input buffer }
Procedure  CreateInBufFreqArray(Var  Freq  :  TFreqRecArrayPtr;  Var  MaxFreq  :
Word);

Var
 P : Word;
 MaxWeight, Freqs : LongInt;

Begin
{ Initialize Frequency pointers }
For P := 0 To T8BitDic Do
 Freq^[P].Weight := 0;

{ Count frequencies of an 8 bit alphabet }
MaxWeight := 0;

For P := 1 To InFileSize Do
 Begin
 Inc(Freq^[InBuffer[P]].Weight);

 If Freq^[InBuffer[P]].Weight > MaxWeight Then
  MaxWeight := Freq^[InBuffer[P]].Weight;
 End;

{ Scale frequency counts to limit the huffman codes to 16 bits }
MaxWeight := (MaxWeight Div 255) + 1;
Freqs := 0;
```

```
For P := 0 To T8BitDic Do
 Begin
 If ((Freq^[P].Weight Div MaxWeight) = 0) And (Freq^[P].Weight <> 0) Then
  Freq^[P].Weight := 1

 Else
  Freq^[P].Weight := Freq^[P].Weight Div MaxWeight;

{ Initialize weight table }
 If Freq^[P].Weight = 0 Then
  Freq^[P].Weight := $FFFF
 Else
  Begin
  MaxFreq := P;
  Inc(Freqs);
  End;
 End;

If Freqs = 1 Then
 SingleFreq := True
Else
 SingleFreq := False;
End;

{ Create canonical huffman codes for an alphabet }
Function CreateHuffArray(Freq : TFreqRecArrayPtr; Size : Word) : Pointer;

Var
 HufBig : THufRecArrayPtr;
 HufSmall : TLitRecArrayPtr Absolute HufBig;
 Huf : THufRecArrayPtr Absolute HufBig;
 Canon : TCanonRecArrayPtr;
 P, PrevLen : Word;
 B1, B2, Low1, Low2, PrevCanon : LongInt;
 FreqPtr : TPtrRecArrayPtr;

Begin
If Size > T8BitDic Then
 New(HufBig)
Else
 New(HufSmall);

{ In cases where only one frequency was recorded, we need to do this }
{ ie. The same characater was repeated for the whole block           }
If SingleFreq Then
 Begin
 Inc(Huf^[Size].Len);
 Huf^[Size].Code := 0;
 CreateHuffArray := Huf;
 Exit;
 End;

{ Create and Initialize huffman pointers }
New(Canon);
For P := 1 To MaxCanonRecs Do
 Begin
 Canon^[P].LenCount := 0;
 Canon^[P].NextCode := 0;
 End;

New(FreqPtr);
For P := 0 To Size Do
 Begin
 FreqPtr^[P].Next := $FFFF;
 Huf^[P].Code := 0;
 Huf^[P].Len := 0;
 End;

Repeat
 Low1 := -1;
 Low2 := -1;

 For P := 0 To Size Do
  If Freq^[P].Weight <> $FFFF Then
   If Low1 = -1 Then

    Low1 := P
   Else
    If Low2 = -1 Then
     Low2 := P
    Else
     If Freq^[P].Weight < Freq^[Low1].Weight Then
      Low1 := P
     Else
      If Freq^[P].Weight < Freq^[Low2].Weight Then
       Low2 := P;

 If Low2 <> -1 Then
  Begin
  B1 := Low1;
  Repeat
   If Huf^[B1].Len <> 0 Then
    Dec(Canon^[Huf^[B1].Len].LenCount);

   Inc(Huf^[B1].Len);
   Inc(Canon^[Huf^[B1].Len].LenCount);
   B2 := B1;
   B1 := FreqPtr^[B1].Next;
  Until B1 = $FFFF;
  FreqPtr^[B2].Next := Low2;

  B1 := Low2;
  Repeat
   If Huf^[B1].Len <> 0 Then
    Dec(Canon^[Huf^[B1].Len].LenCount);

   Inc(Huf^[B1].Len);
   Inc(Canon^[Huf^[B1].Len].LenCount);
   B1 := FreqPtr^[B1].Next;
  Until B1 = $FFFF;

  Freq^[Low1].Weight := Freq^[Low1].Weight + Freq^[Low2].Weight;
  Freq^[Low2].Weight := $FFFF;
 End;
Until Low2 = -1;

PrevLen := 0;
PrevCanon := -1;
For P := 1 To MaxCanonRecs Do
 If Canon^[P].LenCount <> 0 Then
  Begin
  Canon^[P].NextCode := (PrevCanon + 1) shl (P - PrevLen);
  PrevCanon := PrevCanon + Canon^[P].LenCount;
  PrevLen := P;
  End;

{ Create and Initialize huffman pointers }
For P := 0 To Size Do
 If Huf^[P].Len <> 0 Then
  Begin
  Huf^[P].Code := Canon^[Huf^[P].Len].NextCode;
  Inc(Canon^[Huf^[P].Len].NextCode);
  End;

Dispose(FreqPtr);
Dispose(Canon);

CreateHuffArray := Huf;
End;
```

```
{ Outputs the huffman code lengths for the alphabet }
Procedure OutputHufDic(P : Pointer; Size : Word);

Var
 HufPtr : THufRecArrayPtr Absolute P;
 W, CodeLen, Same : Word;


Procedure OutputCleanChar;
Begin
If CodeLen = 14 Then
 OutputBits(CodeLen, 5)
Else
 OutputBits(CodeLen, 4);
Inc(W);
End;

Begin
{ In cases where only one frequency was recorded, we need to do this }
{ ie. The same characater was repeated for the whole block           }
If SingleFreq Then
 Begin
 OutputSingleBit(1);
 OutputBits(Size, 14);
 Exit;
 End
Else
 Begin
 OutputSingleBit(0);
 OutputBits(Size, 14);
 End;

W := 0;
Repeat
 CodeLen := HufPtr^[W].Len;;

 If (W = 0) Or (W > (Size - 2)) Then
  OutputCleanChar
 Else
  Begin
  Same := 1;
  While (Same < 256) And (W + Same < Size + 1) And
        (HufPtr^[W + Same].Len = CodeLen) Do
   Inc(Same);

  If Same < 3 Then
   OutputCleanChar
  Else
   If CodeLen = 0 Then
    If Same < 19 Then
     Begin
     OutputBits(Dic0Pad4, 6);
     OutputBits(Same, 4);
     Inc(W, Same);
     End
    Else
     Begin
     OutputBits(Dic0Pad8, 6);
     OutputBits(Same, 8);
     Inc(W, Same);
     End
   Else
    If Same < 19 Then
     Begin
     OutputBits(DicXPad4, 6);
     OutputBits(Same, 4);
     OutputCleanChar;
     Dec(W);
     Inc(W, Same);
     End
    Else
     Begin
     OutputBits(DicXPad8, 6);
     OutputBits(Same, 8);
     OutputCleanChar;
     Dec(W);
     Inc(W, Same);

     End;
  End;
 Until W > Size;
End;

{ Disposes of a huffman array }
Procedure DisposeHuffArray(P : Pointer; HufSize : Word);

Var
 LitPtr : TLitRecArrayPtr Absolute P;
 HufPtr : THufRecArrayPtr Absolute P;

Begin
Case HufSize Of
 T8BitDic  : Dispose(LitPtr);
 T14BitDic : Dispose(HufPtr);
 End;
End;

End.
```

Figure B.6 - The Huffman Unit.

Figure B.7 lists the IO Unit, which implements all the disk IO routines. It also does the statistics for measuring the compression performance of the LZT program.

```
Unit IO;

Interface

Uses Dos, Basic, Clock;

Var
 InFile : File;
 InFileSize : Word;

 OutFile : File;

Procedure InitializeIO;
Procedure OutputBits(LStream : LongInt; Len : Word);
Procedure OutputSingleBit(Bit : Byte);

Implementation

{ Initializes the IO system variables }
Procedure InitializeIO;
Begin
InPos := 1;

{ Make sure OutBuffer starts off clean }
FillChar(OutBuffer, SizeOf(OutBuffer), #0);
```

```
{ Reserve the next 6 bits of the output buffer so we know how/which     }
{ methods was used to do the compression with. Matches the flag entered }
{ on the command line                                                   }
OutBitsUsed := 6;
OutPos := 1;
End;

{ Outputs bit stream to the output buffer }
Procedure OutputBits(LStream : LongInt; Len : Word);

Var
 I : Word;
 LPtr : ^LongInt;

Begin
LStream := LStream shl OutBitsUsed;

LPtr := @OutBuffer[OutPos];
LPtr^ := (LPtr^ Or LStream);

{ Recalc the new output position in the output buffer }
I := Len + OutBitsUsed;
OutPos := OutPos + (I Div 8);

{ Recalc how many bit's are left at the current output position }
OutBitsUsed := I Mod 8;
End;

{ Outputs a single bit to the output buffer }
Procedure OutputSingleBit(Bit : Byte);
Begin
Bit := Bit shl OutBitsUsed;

OutBuffer[OutPos] := (OutBuffer[OutPos] Or Bit);

Inc(OutBitsUsed);
If OutBitsUsed = 8 Then
 Begin
 OutBitsUsed := 0;
 Inc(OutPos);
 End;
End;

End.
```

Figure B.7 - The IO Unit.

The VLC Unit as shown in Figure B.8 is used to implement the variable length coding routine as described with Figure 3.2.

```
Unit VLC;

Interface

Uses IO;

Function GetVLCLen(Len : Word) : Word;
Procedure OutputVLC(Len : Word);
```

```
Implementation

Var
 BTF, VLCBits : Byte;
 VLCValue, VLCLen : Word;

Function GetVLCLen(Len : Word) : Word;
Begin

BTF := 7;
VLCValue := BTF;
VLCLen := 3;
VLCBits := 3;

While Len > VLCValue Do
 Begin
 If BTF <> 255 Then
  Begin
  BTF := BTF shl 1;
  Inc(BTF);
  Inc(VLCBits);
  End;

 Inc(VLCValue, BTF);
 Inc(VLCLen, VLCBits);
 End;

GetVLCLen := VLCLen;
End;

Procedure OutputVLC(Len : Word);
Begin
BTF := 7;
VLCValue := BTF;
VLCBits := 3;

While Len > VLCValue Do
 Begin
 OutputBits(BTF, VLCBits);

 If BTF <> 255 Then
  Begin
  BTF := BTF shl 1;
  Inc(BTF);
  Inc(VLCBits);
  End;

 Inc(VLCValue, BTF);
 End;

OutPutBits(0, VLCBits);
End;

End.
```

Figure B.8 - The VLC Unit.

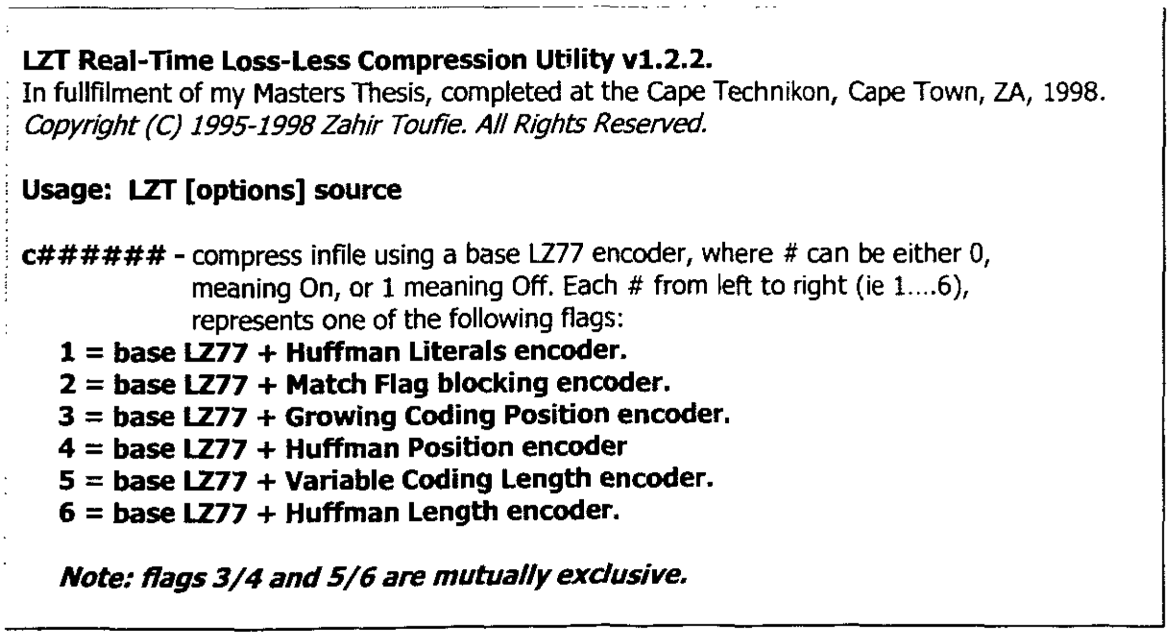
Figure B.9 lists the LZT front end executable. This executable takes as one of its inputs, a six bit flag where each bit represents the required state of an algorithm found within the LZT program. Figure B.10 shows the help screen of the LZT executable, which lists the required input parameters and there descriptions. The LZT executable performs some pre-processing before the actual compression is done. One of these pre-processing events is to break the input file into smaller sector size chunks. This is done because the LZT algorithm is designed to work with data blocks that are less than or equal to a disk sector.

Note that the Decode routine is not implemented and does nothing. It is included merely to show were such a routine would be placed. The internal workings of the Decode routine were not required for the study and as such, it is not implemented.

```
{$A-,B-,D+,E-,F-,G+,I+,L+,N+,O-,P+,Q+,R+,S+,T-,V-,X+,Y+}
{$M 4096, 0, 524288}

Program Lempel_Ziv_Toufie;

Uses Crt, Dos, Basic, Clock, IO, Compress;

Const
 FileCount : LongInt = 0;
 EmptyStr = '            ';

Var
 B : Byte;
 Totalbpb, Averagebpb : Real;
 Totalkbps, Averagekbps : Real;
 TotalBlocks, CurrentBlock : LongInt;
 Name : String[12];

 CompFile : SearchRec;
 LZTFile : String[12];
 InFile, OutFile : File;

Procedure Copyright;
Begin
WriteLn('LZT Real-Time Loss-Less Compression Utility v1.2.2. In fulfilment
of');
WriteLn('my Masters Thesis, completed at the Cape Technikon, Cape Town, ZA,
1998.');
WriteLn('Copyright (C) 1995-1998 Zahir Toufie. All Rights Reserved.');
WriteLn;
End;

Procedure Usage;
Begin
WriteLn('Usage:  LZT [options] source');
WriteLn;
WriteLn('c###### - compress infile using a base LZ77 encoder, where # can
be');
WriteLn('           either 0, meaning On, or 1 meaning Off. Each # from left
to');
WriteLn('               right (ie 1....6), represents one of the following
flags:');
WriteLn('     1 = base LZ77 + Huffman Literals encoder.');
WriteLn('     2 = base LZ77 + Match Flag blocking encoder.');
WriteLn('     3 = base LZ77 + Growing Coding Position encoder.');
WriteLn('     4 = base LZ77 + Huffman Position encoder');
WriteLn('     5 = base LZ77 + Variable Coding Length encoder.');
WriteLn('     6 = base LZ77 + Huffman Length encoder.');
WriteLn;
WriteLn('     Note: flags 3/4 and 5/6 are mutually exclusive.');
WriteLn;
WriteLn('e  - decompress infile.');

Halt(0);
End;

Procedure Error (E : String);
Begin
WriteLn;
WriteLn(E);
Halt(0);
End;

Procedure Initialize;

Var
 S : String;
 I, B, Status : Integer;

Begin
Copyright;

If (ParamCount <> 2) Or (Length(ParamStr(1)) <> 7) Then
 Usage;

FindFirst(ParamStr(2), Archive, CompFile);
If DosError <> 0 Then
 Error('Fatal error: Source file(s) not found, nothing to compress!');

S := ParamStr(1);
S := UpCase(S[1]);
If Not(S[1] In ['C', 'E']) Then
 Usage;
RunType := S[1];

For I := 2 To 7 Do
 Begin
 Val(S[I], B, Status);

 If (B < 0) Or (B > 1) Or (Status <> 0) Then
  Usage;

 Case I Of
  2 : HCMethod  := boolean(B);
  3 : MFBMethod := boolean(B);
  4 : GCPMethod := boolean(B);
  5 : HPMethod  := boolean(B);
  6 : VCLMethod := boolean(B);
  7 : HLMethod  := boolean(B);
  End;
 End;

{ Flags 3/4 or mutually exclusive }
If (GCPMethod) And (HPMethod) Then
 Usage;

{ Flags 5/6 or mutually exclusive }
If (VCLMethod) And (HLMethod) Then
 Usage;

WriteLn('Compressing files(s) using: base LZ77 encoding');
If HCMethod Then
 WriteLn('                    + Huffman Literals encoding');
If MFBMethod Then
 WriteLn('                    + Match Flag blocking encoding');
If GCPMethod Then
```

```
 WriteLn('                           + Growing Coding Position encoding');
If HPMethod  Then
 WriteLn('                           + Huffman Position encoding');
If VCLMethod Then
 WriteLn('                           + Variable Coding Length encoding');
If HLMethod  Then
 WriteLn('                           + Huffman Length encoding');

WriteLn;
End;

Procedure WriteIODetails;
Begin
Averagekbps := (CompFile.Size / EClock) / 1024;
Averagebpb := (OutSize * 8) / CompFile.Size;

GotoXY(15, WhereY);
Write(TotalBlocks:2, ' blocks, Speed: ', Averagekbps:7:3, ' kbps, ');
WriteLn('Ratio: ', Averagebpb:5:3, ' bpb');
End;

Begin
Initialize;

Repeat
 Inc(FileCount);
 TotalBlocks := CompFile.Size Div InBufferLen;
 If (CompFile.Size Mod InBufferLen) > 0 Then
  Inc(TotalBlocks);
 CurrentBlock := 1;
 OutSize := 0;

 Name := CompFile.Name + Copy(EmptyStr, 1, (12 - Length(CompFile.Name)));
 Write(Name, ', block 1 of ', TotalBlocks);
 Assign(InFile, CompFile.Name);
 Reset(InFile, 1);
 If (DosError <> 0) or (IOResult <> 0) Then
  Error('Fatal error: Cannot open file ' + CompFile.Name);

 B := Pos('.', CompFile.Name);
 If B = 0 Then
  B := Length(CompFile.Name);
 LZTFile := Copy(CompFile.Name, 1, B) + '.lzt';
 Assign(OutFile, LZTFile);
 Rewrite(OutFile, 1);
 If DosError <> 0 Then
  Error('Fatal error: Cannot create file ' + LZTFile);

 BlockRead(InFile, InBuffer, SizeOf(InBuffer), InFileSize);
 StartClock;

 While Not(EOF(InFile)) Do
  Begin
  CompressBlock;
  OutSize := OutSize + OutPos;
  BlockRead(InFile, InBuffer, SizeOf(InBuffer), InFileSize);

  GotoXY(21, WhereY);
  Inc(CurrentBlock);
  Write(CurrentBlock, ' of ', TotalBlocks);
  End;
 CompressBlock;
 OutSize := OutSize + OutPos;

 WriteClock;
 WriteIODetails;
 Totalbpb := Totalbpb + Averagebpb;
 Totalkbps := Totalkbps + Averagekbps;

 Close(InFile);
 Close(OutFile);

 FindNext(CompFile);
 Until DosError <> 0;

Averagebpb := Totalbpb / FileCount;
Averagekbps := Totalkbps / FileCount;
WriteLn;

WriteLn(' 1 block = 16k (16384 bytes)');
Write(FileCount:3, ' files processed, Avg Speed: ', Averagekbps:7:3, ' kbps,
');
WriteLn('Ratio: ', Averagebpb:5:3, ' bpb');
End.
```

Figure B.9 - The LZT executable.



```
LZT Real-Time Loss-Less Compression Utility v1.2.2.
In fullfilment of my Masters Thesis, completed at the Cape Technikon, Cape Town, ZA, 1998.
Copyright (C) 1995-1998 Zahir Toufie. All Rights Reserved.

Usage:  LZT [options] source

c###### - compress infile using a base LZ77 encoder, where # can be either 0,
          meaning On, or 1 meaning Off. Each # from left to right (ie 1....6),
          represents one of the following flags:
  1 = base LZ77 + Huffman Literals encoder.
  2 = base LZ77 + Match Flag blocking encoder.
  3 = base LZ77 + Growing Coding Position encoder.
  4 = base LZ77 + Huffman Position encoder
  5 = base LZ77 + Variable Coding Length encoder.
  6 = base LZ77 + Huffman Length encoder.

  Note: flags 3/4 and 5/6 are mutually exclusive.
```

Figure B.10 - The LZT executable help screen.