Department of Electrical Engineering

Master Thesis

Lossless data compression –

methods for achieving better performance in a Wireless VPN

David Broman

LiTH-ISY-EX-3159

8 June 2001

Abstract

Recently, a new software segment called Wireless Virtual Private Networks (WVPN) has been developed to meet enterprises need of secure data access to corporate business critical data. To enable the practical use of interactive applications in the WVPN, the data communication has to be optimized for better performance, something that can be achieved through the use of data compression.

Data compression can reduce the amount of data sent over the network, which leads to faster transmissions. Unfortunately, data compression is not suitable in all situations. The time gained after reducing the data can be lost if the time of compression and decompression turns out to be longer. Many parameters are involved, such as network bandwidth, processing power on clients and servers, properties of algorithms and structure of data. The problem is to determine how these parameters affect performance and which compression algorithms that are most suitable in different situations.

The purpose of this study is to investigate, implement, test and analyze data compression algorithms in order to achieve higher performance in a Wireless VPN.

Data compression algorithms consist of three different elements: coding, modeling and transformation techniques. These techniques can be combined to form different compression algorithms, which in turn can be categorized into statistical, dictionary-based and block algorithms.

27 different types of algorithms were tested in an experiment, which measured the algorithm properties of compression ratio, compression bandwidth and decompression bandwidth.

The category of algorithms that was found to achieve the best compression ratios for all type of input data was statistical algorithms. Block algorithms could give nearly the same benefits as the statistical algorithms, but did not fit the requirements of a WVPN. The result showed that dictionary-based algorithms were less memory consuming and faster than the others, but gave less satisfying compression ratios.

Finally, it was found that the capability of the server in memory and processing power will in practice determine when data compression should or should not be used in a WVPN.

Acknowledgements

This thesis completes my studies in Master of Science of Industrial Engineering and Management at Linköping Institute of Technology.

Many working days and late nights have been spent to finish this study and report. But this could not have been done without help from several other persons.

First of all I would like to thank the staff at Columbitech and especially my supervisor Torbjörn Hovmark. Further, I would like to thank Joel Lindholm, developer at Columbitech, who has spent many hours in reading this thesis and giving many helpful comments.

My opponent Erik Bengtsson has given many comments, which have improved this report.

I would also like to thank Professor Robert Forchheimer, who has followed and approved my work.

Finally, I would like to thank the following persons for helping me by reading the report: Åsa Nilsson, Jonas Clausen, Fredrik Jansson, Olof Broman and Eva Broman.

Stockholm, 31 May 2001

David Broman

Contents

1 Introduction **1**
1.1 Background to the problem 1
1.1.1 Wireless virtual private network 2
1.1.2 Performance in WVPN 3
1.2 Problem discussion 4
1.2.1 Structure of data determines compression rate 4
1.2.2 Algorithms have different benefits 4
1.2.3 Questions for this study 5
1.3 Purpose of the study 5
1.4 Delimitations 6
1.5 Employer of the study 7
1.6 Reader's guide 8
1.6.1 The audience 8
1.6.2 Reference method 8
1.6.3 Structure of the report 8

2 Theoretical background **11**
2.1 Data compression concepts 11
2.1.1 Lossless and lossy compression 11
2.1.2 Streaming and block modes 12
2.1.3 Static, semi-adaptive and adaptive models 12
2.2 Data communication 13
2.2.1 Bandwidth, delay and product 13
2.2.2 Data compression in data communication 14
2.3 Data structure 18
2.3.1 Bits, bytes, characters, words and symbols 19
2.4 Information Theory 20
2.4.1 Entropy 20
2.5 Coding theory 23
2.5.1 Variable size codes 23
2.5.2 Prefix codes 24

2.6 Structure of compression algorithms . . . 25
2.6.1 Coding . . . 26
2.6.2 Modeling . . . 27
2.6.3 Transformation . . . 27
2.6.4 Compression algorithms . . . 28
2.7 Properties in a WVPN . . . 28
2.7.1 Layers in a WVPN . . . 28
2.7.2 Client properties . . . 29
2.7.3 Server properties . . . 30
2.7.4 Network carrier . . . 30

3 Method of this study 33
3.1 Introduction . . . 33
3.2 Collection of information . . . 34
3.3 Choice of algorithms . . . 34
3.4 Implementation and experiment . . . 35
3.4.1 The data compression test . . . 35
3.4.2 Bandwidth for different devices . . . 36
3.5 Analysis and conclusion . . . 37
3.6 Source of errors . . . 37

4 Coding techniques 39
4.1 Shannon-Fano coding . . . 39
4.1.1 Evaluation . . . 40
4.2 Huffman coding . . . 42
4.2.1 Static Huffman coding . . . 42
4.2.2 Adaptive Huffman coding . . . 45
4.2.3 Evaluation . . . 47
4.3 Arithmetic coding . . . 49
4.3.1 Encoding method . . . 49
4.3.2 Decoding method . . . 51
4.3.3 Practical concerns . . . 52
4.3.4 Evaluation . . . 53
4.4 Range Coding . . . 54
4.5 Run length encoding . . . 54
4.5.1 Encoding and decoding . . . 54
4.5.2 Evaluation . . . 55

5 Modeling techniques 57
5.1 Introduction . . . 57
5.2 Statistical modeling . . . 57

5.2.1 Finite context modeling 58
5.2.2 PPM 59
5.2.3 Evaluation 60
5.3 Dictionary-based modeling 61
5.3.1 LZ77 61
5.3.2 LZ78 63
5.3.3 Other lz algorithms 64
5.3.4 Evaluation 66

6 Transformation techniques 69
6.1 Move-to-front 69
6.1.1 Encoding 69
6.1.2 Decoding 71
6.1.3 Evaluation 71
6.2 Burrows-Wheeler transform 72
6.2.1 Forward transformation 72
6.2.2 Reverse transformation 73
6.2.3 Benefits with the transform 75
6.2.4 Evaluation 75
6.3 Differential coding transform 77
6.3.1 Evaluation 77

7 Compression algorithms 79
7.1 Combining transformation, modeling and coding 79
7.2 Statistical algorithms 80
7.3 Dictionary-based algorithms 81
7.4 Block algorithms 82

8 Experiment and result 85
8.1 Introduction 85
8.2 Test data 85
8.3 Devices processing capacity 86
8.4 Implementations in the experiment 87
8.5 Results 92

9 Analysis 93
9.1 Parameters affecting performance 93
9.1.1 Network bandwidth 94
9.1.2 Compression ratio 97
9.1.3 Compression and decompression bandwidths 101
9.1.4 Number of blocks 106

9.1.5 Memory 107
9.2 Strength-weakness analysis of algorithms 109
9.2.1 Statistical algorithms 109
9.2.2 Dictionary-based algorithms 109
9.2.3 Block algorithms 110
9.3 Improving the performance in a WVPN 110
9.3.1 Memory requirement 110
9.3.2 Streaming requirement 111
9.3.3 Bandwidth and compression ratio 112

10 Conclusions 115

11 Reflections and further research 117
11.1 Reflections 117
11.2 Further research 118

A Result from experiment 127

B Compression implementations 143

C Compression ratio diagrams 147

D Compression and decompression bandwidths 151

List of Figures

1.1 Outline of a WVPN 2

2.1 Network illustrated as a pipe 14
2.2 Compression procedure with model and encoder 26

4.1 Huffman tree with 3 nodes 43
4.2 Huffman tree with 5 nodes 44
4.3 Final Huffman tree 44

5.1 Sliding window in LZ77 example 62

9.1 Compression ratios for different input data 100
9.2 Compression and decompression bandwidth for different implementations 102
9.3 Relation between compression bandwidth and compression ratio 103
9.4 Relation between decompression bandwidth and compression ratio 104

C.1 Left: C++ source code, Right: Compressed files 148
C.2 Left: Executables, Right: Gif 148
C.3 Left: Html, Right: Jpg 148
C.4 Left: Mp3, Right: MS access 149
C.5 Left: MS Powerpoint, Right: MS Word document 149
C.6 Left: Pdf, Right: Postscript 149
C.7 Left: Text, Right: Wav 150
C.8 MS Excel 150

List of Tables

2.1 Data structure elements 20
2.2 Symbols, probabilities and codes 24
2.3 Correct codes 25

4.1 Codes generated by Shannon-Fano coding 40
4.2 Huffman: Probabilities of symbols 43
4.3 Huffman codes 45
4.4 Arithmetic: Probabilities of symbols 49
4.5 Symbols specified with ranges 50
4.6 Arithmetic encoding 51
4.7 Arithmetic decoding 52

5.1 LZ78 compressing example 64
5.2 LZ78 decompressing example 65

6.1 Result after MTF 70
6.2 Matrix in BWT forward transformation 73
6.3 First step of reconstructing matrix M 74
6.4 Compression ratios for different block sizes 76

7.1 Example of statistical algorithms 81
7.2 Example of dictionary-based algorithms 82
7.3 Example of Block algorithms 83

8.1 Data in experiment 86
8.2 Performance scale factors for different devices 87

C.1 Order of implementations 147

D.1 Algorithms sorted after compression bandwidth 152
D.2 Algorithms sorted after decompression bandwidth 153
D.3 Decompression bandwidths for different devices 154
D.4 Compression bandwidths for different simultaneous users . . . 155

Chapter 1

Introduction

In this introduction chapter the background to the problem is described and a brief introduction to the concept of data compression is given. The problems that form the basics for this thesis are stated, which results in the purpose of the study. The delimitations made are listed and a description of the employer of this study is given. Finally, guidelines for the reader are specified.

1.1 Background to the problem

The need for enterprises to have easy and reliable access to business critical data has always been an important issue. In the last decades, the computer revolution has radically changed the way companies manage information. The introduction of PCs connected to local networks have made it possible for employees to access important data easy and fast, but as the development goes further, new demands for improvement are discovered.

Nowadays, employees are not satisfied with reliable data access at the office only. In many business areas the demand of having access to critical enterprise data everywhere has dramatically increased. But reliability is just one important aspect of mobile enterprise data access. Since many data applications are business critical, the need for strong security is highly important. To be able to work efficiently with wireless applications, the response time and performance of the system must be very good. To meet the requirements of reliability, security and performance, a new software segment has been discovered: The wireless virtual private networks.

1.1.1 Wireless virtual private network

A Virtual Private Network (VPN) tunnels the data through encrypted logical channel. This allows it to use existing network infrastructure and still guarantee privacy and security. Thus, an enterprise can have a transparent private network by making use of public network solutions. The benefit and idea of a VPN is to give the company the same capabilities at a much lower cost compared to operate a private one. [*VPN - a whatis definition* 2000]

A wireless virtual private network (WVPN) basically has the same functionality as an ordinary VPN, but is designed for wireless applications. Due to the very high cost investing in a private wireless solution, buying this service from an operator is a natural choice for most companies.

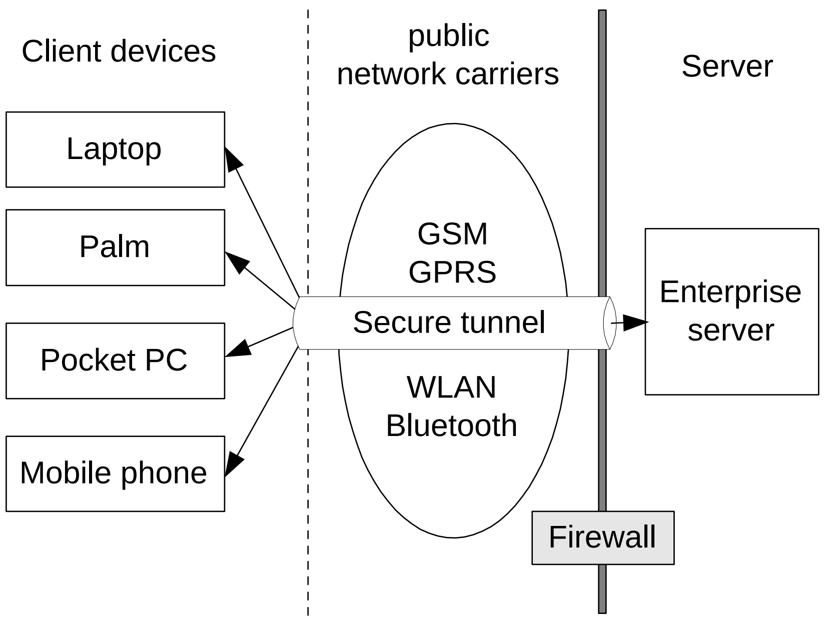


Figure 1.1: Outline of a WVPN

Figure 1.1 illustrates the basic structure of a WVPN. Different client devices can access the information stored in the enterprise server in a secure way. The data sent between the server and the client devices are encrypted end-to-end, which means that no one can read or modify the data without permission.

Due to the fact that a wireless network is quite different compared to a wired network, more properties than reliable data access needs to be taken into consideration when building a functioning product. Since the bandwidth for wireless networks generally are much lower than for wired networks, the design of the software must be optimized for performance.

1.1.2 Performance in WVPN

To achieve high performance when sending data between client and server, the data sent should of course arrive to the destination as fast as possible. The bandwidth and the delay between the systems are natural properties that affect the performance. While these properties belong to a lower level than the WVPN-software, other methods must be used to optimize the communication performance.

A convenient way of doing this is via data compression. The data to be sent is compressed by the sender and decompressed by the receiver, which results in less data transmitted and therefore a faster transaction. There are many different approaches to compress data and different algorithms result in different compression rates for the same data input. On the other hand, one compression algorithm can generate totally different results on different data inputs. How the data is structured has a major impact on the compression performance. Thus, different document types, for example html, word documents, http headers, binary executable files etc., should be treated in different ways.

Different compression algorithms require different amount of memory and processing capacity. A WVPN usually supports a number of clients, which all have different memory and processing capacity. To achieve optimal performance when transmitting data across the network, different methods have to be used depending on input data and client platform. The server can, depending on the number of simultaneous users, have different amount of load, which should also be taken into consideration when choosing compression method.

Many standard communication devices already use data compression to achieve better performance, for example modern modems. This is realized by streaming data compression at the hardware level, which means that data is compressed before it is transmitted by the modem. Such a device achieves relatively good result when data is suitable for compression. Example of input data that gives good compression result by the modem compression is normal text or html documents. The problems occur when a security level is added to the software. When this is done, the input data to the modem is close to random, which is difficult to compress with good results. The obvious solution to this problem is to compress the data before encrypting it. This cannot be done below the encryption level, which means that it must be done by the WVPN-software.

To achieve the best possible performance, consideration must be taken, among other things, of the type of wireless client, load on the server and structure of the data sent.

1.2 Problem discussion

Software developers face a number of challenges when trying to increase performance in a WVPN. In the following part, properties that must be taken into consideration, will be discussed.

1.2.1 Structure of data determines compression rate

Data sent through the network has various forms and patterns. The structure of an html-file is totally different from an executable binary file. Furthermore, executable files compiled for different CPUs may have various structures that could be treated in different ways in order to achieve best possible result when compressing them. More information about how the data is organized gives higher probabilities to achieve enhanced result. The problem is to know how data is structured and which compression method that is the most appropriate to use.

1.2.2 Algorithms have different benefits

The obvious goal when compressing and transmitting data over a network is to make the data sent as small as possible. The smaller the sent data is, the faster it can be transmitted over the network. But there are other factors that should be taken into consideration. Compressing data can be a process-demanding task that in worst case requires more time than transmitting the data uncompressed. The processing time is highly dependent on the implementation of the algorithm and the processing unit used when compressing. For example, the processing power on a Palm Pilot is significantly lower than on a modern laptop. Even if a server normally has high processing performance, from time to time the load can vary. It may therefore not be suitable to apply the same methods or algorithms under all circumstances.

Another aspect is the memory required during compression. Some compression algorithms call for much more memory to work properly than others do.

The limitations also differ from device to device. It is therefore not obvious what method should be used when compressing input data. Many factors are involved when deciding compression method. Bandwidth, processing capability, memory limits etc. play an important role in this decision. The problem is how to analyze these properties in a logical way in order to choose good compression methods.

1.2.3 Questions for this study

To summarize the problem discussion the fundamental research questions for this study are here stated:

- Which parameters affect when data compression achieves better performance in a WVPN?
- What sort of data compression algorithms are currently available and how are they designed?
- Approximately, how much memory do the different compression methods require?
- How fast can the different algorithms compress and decompress data?
- What are the restrictions for different wireless platform clients? Memory requirements? Processing capacity?
- How much can different compression algorithms reduce input data? How does the type of input data affect the result?
- Which categories of data compression is most suitable in a WVPN?

1.3 Purpose of the study

The purpose of this study is to investigate, implement, test and analyze data compression algorithms in order to achieve higher performance in wireless virtual private networks.

1.4 Delimitations

The above given purpose is limited with the below listed delimitations:

- Only lossless algorithms are investigated and tested, i.e. algorithms that return an identical copy of data after decompressing compared to the original data.

- Compressed data is assumed to be transmitted over the network without errors. The error correction handling is assumed to be situated at a lower level than the compression level.

- Communication network performance for GSM, GPRS, CDPD, HSCSD, Bluetooth and WLAN, is only tested on a theoretical level. Documented data about their performance is used to choose suitable compression methods.

- No economical aspects are investigated, as for example cost of bytes sent over the network, when deciding whether data compression is suitable.

- Performance in the WVPN implies in this study to the amount of data that is possible to send between the client and the server per time unit. The possible extra load on the client that could affect other processes running simultaneously is not taken into consideration. This, since most of the currently used applications in a WVPN are interactive, i.e. the user interact in real time with the enterprise server.

- Only data sent on the session level in the WVPN is analyzed.

- The theoretical number of possible compression algorithms is huge, since all different techniques can be implemented in different ways and all possible combinations of different techniques give many possibilities. Thus, it is not possible to include all different combinations in the experiment. Therefore, just a number of implementations are chosen and tested.

- Only data sent from the server to the client is analyzed, since this is the most probable case in a WVPN.

1.5 Employer of the study

The employer of this study is Columbitech, a Swedish company that develops software products for wireless data communication. The company was founded in April 2000 and has currently 25 employees. The founders and the external venture capital firms Ledstiernan, Servisen and Pelago Venture Partners own Columbitech. [*Columbitech homepage* 2001]

Columbitech develops and markets a wireless virtual private network (WVPN). The basic functionality of the product related to this study is:

Seamless network roaming and handover The wireless market is moving to a mosaic of wireless networks that consist of both public mobile networks and local wireless hotspots. Public mobile networks like GPRS and GSM are characterized by large coverage but low bandwidth, compared to local wireless hotspots like wireless LAN and Bluetooth, which have low coverage and cost but relatively high bandwidth. Columbitech's wireless VPN has support for roaming between networks, i.e. users can move between different types of wireless networks without abruption of the session. [*Columbitech wireless VPN* 2001]

Since different network carriers are used, which all have different characteristics in bandwidth, this must be taken into consideration when choosing data compression method.

Security framework Columbitech wireless VPN uses the wireless adaptation of the industrial standard Transport Layer Security Protocol, WTLS to ensure end-to-end security between client and server. [*Columbitech wireless VPN* 2001]

Since all data sent between the server and the client is encrypted, hardware data compression does not achieve satisfying results. Thus, data compression must be implemented at a higher level than the security level.

Independent of device and network The product will support leading platforms for wireless clients, such as Palm OS, Pocket PC and Windows [*Columbitech wireless VPN* 2001]. All these platforms have different characteristics such as memory requirements and processing capacity. These aspects must be evaluated when choosing compression method.

Optimised performance Columbitech's product is going to use various optimization techniques to achieve better performance. Both data reduction and data compression will be applied on the session and application levels in order to reduce transported data and therefore achieve better performance. [Hovmark 2001]

The commission for this report, from Columbitech's point of view, is to investigate and analyze lossless data compression on the session level. This should then form the basis of a strategy for the company how to develop the performance functionality in their product.

1.6 Reader's guide

1.6.1 The audience

The reader is assumed to have basic skills in programming, algorithm knowledge and mathematical expressions, though it does not assume deep knowledge in data compression or data communication. Most of the mathematical expressions and pseudo code will be discussed in detail and illustrated with examples.

The purpose of this study is not to give a full coverage of the field of data compression and exactly how different algorithms are designed. If this is the reader's interest, books like *Data compression, the complete reference* by David Salomon is more suitable [Salomon 1997].

1.6.2 Reference method

This report is written in LaTeX, which is a standard software tool for generating articles, reports and books. The program has a special referencing style, where all references are given inside brackets. The syntax of the referencing used follows the Harvard referencing style. The only exception is in the bibliography, where the reference brackets also are given.

1.6.3 Structure of the report

The report is divided into eleven chapters, which are here briefly described.

Introduction

In this chapter the background to the problem is described and the problems associated with the background discussed. The purpose of the study is stated and delimitations are listed. A description of the employer of this study is given and the guidelines for the reader are specified.

Theoretical background

The theoretical background gives an overview of data compression and data communications performance optimization. This overview is necessary to understand the area and to be able to analytically investigate the behavior of data compression algorithms.

Method of this study

This chapter describes how the study was made and how the collection of data was done. Further, a description is given about how the compression experiment was performed.

Coding techniques

This is the first of three chapters describing different parts of data compression. This chapter describes how the most common coding techniques are designed and gives a brief overview on how they can be implemented. Techniques described are Shannon-Fano coding, Huffman coding, Arithmetic coding and Range coding. Furthermore, a discussion about the techniques concerning memory usage, performance and compression ratio is given.

Modeling techniques

This chapter covers the second and probably the most significant part of data compression; the modeling technique. The main categories of modeling are in this study divided into statistical modeling and dictionary-based modeling. After the description of each technique, discussions about its properties are given.

Transformation techniques

This is the third and last part describing compression techniques. This chapter covers the principals of transformation techniques and a discussion about the properties of the techniques. Techniques handled in this chapter are Move-to-front, Burrows-Wheeler transform and Differential coding transform.

Compression algorithms

This chapter describes how the different techniques in chapter 4, 5 and 6 can be combined and categorized to form compression algorithms. Combinations stated in this chapter form the basics to implementations tested in the compression experiment.

Experiment and result

In this chapter the properties for the compression experiment are stated and the numerical result of it presented.

Analysis

In this chapter a discussion is made to sum up the conclusions about different compressing techniques discussed in chapter 4, 5 and 6. Furthermore, the results achieved from the compression experiment described in chapter 8 are analyzed and discussed. The results from the discussions of this chapter are stated in chapter 10; Conclusions.

Conclusions

In this chapter the conclusions achieved from the previous chapter are summarized.

Reflections and further research

In this final chapter, reflections of this study are stated and recommendations for further research are given.

Chapter 2

Theoretical background

This chapter consists of theoretical background necessary to understand the concept of data compression and how to achieve high performance in data communication.

2.1 Data compression concepts

Basically, the goal of data compression is to reduce the size of the original data such a way that it can later be restored. This can be achieved in many different ways and with different goals and purposes. This thesis covers data compression for data communication in wireless applications. To be able to understand different methods for data compression, some fundamental concepts will be outlined below.

2.1.1 Lossless and lossy compression

Data compression techniques can be divided into two major parts: lossless and lossy compression.

Lossless compression is a technique that guarantees the generation of an identical copy of the input stream after the compressing-decompressing cycle. These techniques are widely used in areas likes compressing and storing database records, spreadsheets and word processing files. In such files, loss of even one single bit risk making the data unusable. [Nelson & Gailly 1996]

Lossy data compression takes an input stream, transforms it via compression and then decompresses it. The term lossy means that the output data after decompression is not identical to the original data. The data that is lost or destroyed should affect the result as little as possible. This technique is mostly used on data that is not directly generated by computers, for example photos, sounds and videos. A human being cannot normally recognize all details in pictures and sounds, which the compression algorithm makes use of by removing the least important information. Most lossy compression techniques can be adjusted to different quality levels, gaining less accuracy in exchange for more effective data compression. In the past years, new and powerful lossy compression methods and standard file formats have been developed which enable new possibilities like streaming digital video over networks. [Nelson & Gailly 1996]

2.1.2 Streaming and block modes

Most data compression algorithms operate in so-called streaming mode, where the input data is processed continuously until the end of the stream is reached. This technique is suitable for applications where the length of the input data and the input data itself are unknown. An application where streaming is suitable is data transmission over networks with content like video or sound. Many archiving applications, i.e. software programs for compression and assembling files into archieves also use stream-based algorithms. The opposite of streaming is called block mode. Here the input stream is read and divided into blocks, where each block is encoded and treated separately. [Salomon 1997]

Streaming is often performed practically by dividing the input into small blocks. In this way, the blocks are compressed separately, but with help of information seen in earlier blocks.

2.1.3 Static, semi-adaptive and adaptive models

Compression methods could be either static, semi-adaptive or adaptive.

A *static model* maps the symbols in the message to fixed set of codewords, which are the same for the whole message. Normally, frequently occurring symbols are coded into shorter codes, and less frequently occurring symbols into longer codes. Since a static method must know the frequency before the transmission of code data begins, this information must be either stored in the compression algorithm or transmitted to the decoder before the code data is sent. The first alternative is, according to Bell, Cleary & Witten [1990], called static and the second alternative semi-adaptive.

The last alternative generates an overhead, which results in more data output from the compression unit. [Lelewer & Hirschberg 2001]

The opposite to a static model is a *dynamic* model. A dynamic model, or *adaptive model* as it also is called, changes the appearance of the code words over time. This means that a specific symbol is coded into different codes during compression. Most of the dynamic models use the data seen so far to decide how to code the next coming symbols. [Lelewer & Hirschberg 2001]

2.2 Data communication

In most computer systems high performance is an important area. This must be evaluated as a trade off between development cost and application benefit. The old programming adage "first make it right and then make it fast" is suitable in many cases, but usually not for network communication. The network system must be designed for performance. To be able to achieve high performance, it is therefore important to understand the various factors that have an impact on data communication over a network. [Peterson & Davie 2000]

2.2.1 Bandwidth, delay and product

The performance of a network can be measured in two fundamental ways: bandwidth and delay. Another name for bandwidth is throughput and another name for delay is latency. The bandwidth can be interpreted as the maximum amount of data that can be transmitted over the network per time unit. A common way of expressing bandwidth is bits or megabits per second. [Peterson & Davie 2000]

The second measurable unit, delay, is the time it takes for the first data unit in a message to be transmitted from the sender to the receiver. Latency or delay is strictly measured in time, often in milliseconds. It is relatively hard to measure the delay between the sender and the receiver, mainly because the clocks in the sender and receiver must be exactly synchronized. It is therefore common, and often more interesting, to measure the so-called round-trip-time. This is the time it takes to send a message from one end of the network to the other and back. [Peterson & Davie 2000]

There are mainly three factors that determine the delay. The first is the speed of the media. Second, there is the time it takes to transmit a data unit. This is a function dependent on the bandwidth of the network and the size of the data unit. Third, there are delays when handling, queuing and switching the data packages in the network. A channel between two processes can be illustrated as a pipe as shown in figure 2.1. [Peterson & Davie 2000]

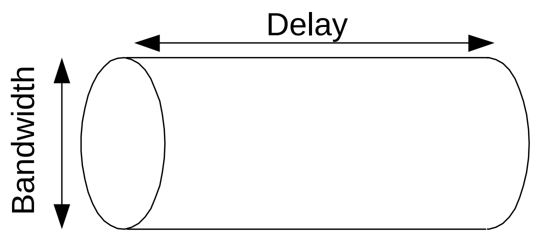


Figure 2.1: Network illustrated as a pipe

The product of the two metrics is often called the delay $*$ bandwidth product. The latency corresponds to the length of the pipe and the area indicates the bandwidth of the pipe. The product gives the volume of the pipe, which is identical to the number of bits the network can hold.
[Peterson & Davie 2000]

2.2.2 Data compression in data communication

To be able to discuss data compression in data communication, the expression compression ratio must be defined as follows:

$$r = \text{Compression ratio} = \frac{\text{size of the output stream}}{\text{size of the input stream}} \tag{2.1}$$

It might seem that compressing data before sending it would always be an obvious choice, presupposed that $r < 1$. Then the amount of data sent over the network will be less, which results in faster transaction. But this is unfortunately not always the case [Peterson & Davie 2000].

When should then data compression be used for data transmission? Since this is the main question for this study, the answer is not as trivial as it may seem.

According to Hovmark [2001], a WVPN is mostly used for interactive applications, such as web browsing. Thus, the goal of performance is to transmit a message over the network as fast as possible. This means that when a client requests a specific data message, it should receive the entire message as quickly as possible. There are of course many other aspects that could be taken into consideration. For example, the economical cost to transmit data over the network or that data compression can affect the performance of applications, which are running simultaneously on the server or the client. As stated in the delimitations in chapter 1, these aspects are not taken into consideration in this study.

Peterson & Davie [2000] defines when data compression is beneficial in data communication. They define the network bandwidth between the server and the client to B_n. B_{cd} is the average bandwidth at which data can be pushed through the compression and the decompression unit in series. r is the average compression ratio and x is the amount of uncompressed data to be transmitted. Peterson & Davie use another definition for compression ratio, and their formulations are therefore transformed to the definition for compression ratio used in this study.

They show that the time it takes for uncompressed data to be transmitted over the network is

$$\frac{x}{B_n} \tag{2.2}$$

The time it takes to compress the data, transmitted over the network and to decompress it at the receiver is

$$\frac{r * x}{B_n} + \frac{x}{B_{cd}} \tag{2.3}$$

According to Peterson & Davie [2000], compression is then beneficial if

$$\frac{r * x}{B_n} + \frac{x}{B_{cd}} < \frac{x}{B_n} \tag{2.4}$$

Since x is superfluous in the expression (2.4), it can be removed. Thus, the expression is

$$\frac{r}{B_n} + \frac{1}{B_{cd}} < \frac{1}{B_n} \tag{2.5}$$

The expression (2.5) does not take into consideration the difference in bandwidth for compression and decompression. Therefore, these definitions are redefined to:

- B_c is called compression bandwidth and is defined as the average amount of data that can be pushed through the compression unit per time unit. B_c is dependent of the current processing capacity of the device, the algorithms structure and the implementation of the algorithm.

- B_d is the decompression bandwidth. B_d has the same dependencies as B_c.
- B_n is the network bandwidth. This bandwidth is dependent on the network carrier, for example GSM, GPRS etc. It is also affected by physical properties as the signal intensity and position of the physical receiver.

The relationship between Peterson & Davie's definitions and the new ones are

$$\frac{1}{B_{cd}} = \frac{1}{B_c} + \frac{1}{B_d} \tag{2.6}$$

Furthermore, the expression (2.5) assumes that the whole data message is transmitted as one single block. In a WVPN, data compression must in some circumstances be able to split the entire message into smaller blocks and transmit them separately. This sort of streaming data compression gives other properties when data compression is advantageous. [Hovmark 2001]

y is therefore defined as the amount of data in one block. n is the number of blocks that one message contains. Thus, the expression $x = n * y$ is obvious. For a number of compression algorithms, the compression ratio r is highly dependent on the block size y. If the blocks are too small, the compression algorithm cannot achieve any good compression ratio. The block size must therefore be set to a carefully chosen number. On the session level of a WVPN, a block is normally an IP-packet, which has a size of approximately 1500 bytes. [Hovmark 2001]

The time it takes to transmit the whole message in separate blocks without compression is therefore

$$\frac{y * n}{B_n} + D_n \tag{2.7}$$

where D_n is the delay in the network. When the sender starts to send a message, it first compresses the first block y. This takes $\frac{y}{B_c}$ amount of time. Then this block will be transmitted over the network, which takes $\frac{r*y}{B_n} + D_n$ amount of time, followed by the time it takes to decompress it $\frac{y}{B_d}$. This is the total time to transfer one single block. The compression and decompression time is just calculated for the first block, since the following blocks can be compressed and decompressed meanwhile the data is transmitted over the network. At last, the network delay D_n is also added to the transfer time.

The time it takes to transmit the message with compression is therefore

$$\frac{r * y * n}{B_n} + \frac{y}{B_c} + \frac{y}{B_d} + D_n \tag{2.8}$$

Under these circumstances, data transmitted over the network will be faster with compression if

$$\frac{r*y*n}{B_n}+\frac{y}{B_c}+\frac{y}{B_d}+D_n<\frac{y*n}{B_n}+D_n \tag{2.9}$$

As can be seen in expression (2.9), the sizes of y and D_n do not directly affect the result and can therefore be eliminated as follows

$$\frac{r*n}{B_n}+\frac{1}{B_c}+\frac{1}{B_d}<\frac{n}{B_n} \tag{2.10}$$

If the number of blocks is equal to one, the expression (2.10) is equal to expression (2.5) by Peterson & Davie. The only difference is that they have not separated the compression and decompression bandwidths.

If the number of blocks is greater than one, the bottleneck must be forced to be the network bandwidth. Otherwise, the compression or decompression will not be able to compress all data sent through the network. Under these circumstances the following expression must be valid.

$$\frac{1}{B_c}<\frac{r}{B_n}\ and\ \frac{1}{B_d}<\frac{r}{B_n} \tag{2.11}$$

It is easy to see in expression (2.10) that the larger variable n is, the less does the compression and decompression bandwidth effect the total transmission time. This is the same as for a fixed message size, the smaller each block is, the less the compression and decompression bandwidth affects the result.

Encryptions influence

A WVPN uses encryption to handle security, i.e. all data sent over the network is encrypted. On the server side, this can be done by either software or hardware implementation, but on the client side currently only software is possible [Hovmark 2001].

Since encryption is a part of the network, the encryption or decryption routine can be the bottleneck in the system. This could be the case if a slow device is used and the network bandwidth is high. Encryption and decryption is done after data compression, which means that data compression can reduce the amount of data that is encrypted. If the encryption bandwidth is low and the compression bandwidth is high, data compression can actually increase the performance.

Delay

The compression bandwidth also affects the delay of a block of data. The delay time is increased by the time it takes to compress and decompress the first block sent. If y is the size of the first data block sent and D_n the network delay, the total delay time D_{tot} can be expressed as

$$D_{tot} = \frac{y}{B_c} + \frac{y}{B_d} + D_n \tag{2.12}$$

The total delay time becomes shorter, the smaller the data block sent is. Expression (2.12) plays an important role if each block sent is large, but when the requested data is divided into smaller blocks, its importance decreases.

Example

In an experiment performed by Wierlemann & Kassing [1998], a GSM-network with bandwidth 9.6 Kbit/s could deliver a bit more than 800 bytes/s. In this example there is a message containing 200000 bytes. The compression bandwidth is 2000 bytes/s, decompression bandwidth is 4000 bytes/s and the compression ratio 0.5. It would take $\frac{200000}{800} = 250$ seconds to transport this message without compression in one single block. If instead compression were used, the total time would be $\frac{200000*0.5}{800} + \frac{200000}{2000} + \frac{200000}{4000} = 275$ seconds. In this case, compression would increase the total time and is therefore not suitable.

The example is changed slightly and instead of sending the message in one block, it is divided into 100 blocks, each containing 2000 bytes. The time it takes to transport the data without compression is the same, but with data compression, the result is $\frac{200000*0,5}{800} + \frac{2000}{2000} + \frac{2000}{4000} = 126,5$ seconds. Thus, how the data is divided into blocks has a large impact on the total transmission time. It seems that the obvious choice would be to send blocks which are small, but this is unfortunately not so easy to do in practice. As we will see later, many compression algorithms are designed to have large input blocks to be able to render good compression ratios.

2.3 Data structure

Data compression is highly dependent on the structure of the input data. The more redundant the input data is, the more likely it is that a compression algorithm can render a satisfying result. Before explaining what redundant data is and how it can be used, some definitions are stated.

2.3.1 Bits, bytes, characters, words and symbols

There are a lot of definitions of bit, byte, word, character and symbol. In the following section the definitions of these terms that are used in this report, will be presented.

A *bit* is an atom unit of digital information with two possible states: 0 or 1. This is the smallest part of data. A *byte* has 2^N states, where N represents number of bits in the byte. Ratushnyak's [2001] defines a generalized N-bit byte, which can hold different amount of values depending on the number of bits used. Traditionally, a byte consists of 8 bits, which can hold $2^8 = 256$ different values. To avoid confusion between the traditional definition and Ratushnyaks definition, from now on, a byte consists of 8 bits. To be able to talk in general about the elements in data sources, the term *character* is used instead. A character is defined as a unit that can have n numbers of values. If a N-bit character is given, it can holds 2^N different values. If nothing else is specified, a general N-bit character is assumed.

Common terms used in compression theory are *code* and *codeword*. A codeword is a finite sequence of bits that represents a specific value. Thus, a byte can represent 256 different codewords and a N-bit character of 2^N codewords. It is important to point out that different codewords representing characters, do not need to have the same number of bits. For example, Huffman codes are constructed so that more frequent characters have shorter codes.

A finite sequence of characters is called a *string*. The number of characters that a specific string consists of is named *string length*. Thus, by combining the definitions of string and code, it is easy to see that a codeword is a string of generalized 1-bit characters.

A sequence of two bytes is, in the computer world, called a word. A grouping of four bytes is called a *double word* or a *dword*. It is important to point out that the concept of word and dword plays an important role considering CPU performance. A long time ago, the processors handled 8-bit elements most efficiently, but nowadays, the modern CPUs compute 32-bit elements or even 64-bit elements most efficiently. These concepts must be taken into consideration when designing and implementing data compression algorithms.

A *symbol* is defined as an element of an alphabet. In the ASCII (American Standard Code for Information Interchange) standard, each symbol is an 8-bit character. This standard has been used for a long period, but now when most of the world is computerized, a new character standard has been developed. The *UNICODE* standard is widely accepted and used, since it supports character sets for the most languages in the world. The UNICODE standard represents each symbol as a 16-bit character. [Ratushnyak 2001]

A normal 8-bit character holds therefore 256 different symbols. In this report, symbols are associated with an alphabet. If for example an alphabet contains 512 different symbols, these symbols can be stored in a 9-bit character. Thus, a symbol is the type and a character is its instance.

A summary of the data structure elements are listed in table 2.1

Element	**Description**
Bit	An atom of a data unit, which has two states: 0 or 1
Byte	A sequence of 8 bits that has 256 different states.
Codeword	A finite sequence of bits.
Alphabet	A sequence of symbols.
Symbol	An element of an alphabet.
Character	An instance of a symbol.
String	A finite sequence of characters.

Table 2.1: Data structure elements

2.4 Information Theory

Information theory is a branch of mathematics that was founded in the late 1940s mainly by Claude Shannon at Bell Labs. The theory concerns subjects about information and ways of storing and communicating messages. [Nelson & Gailly 1996]

Information theory concerns data compression in words of expressing redundancy of information in data. Data compression techniques are ways to reduce the size of the data by removing redundant information. The key of information theory is quantifying information. This makes it possible to measure information and redundancy. The fundamental concept to understand this theory is named *entropy*. [Salomon 1997]

2.4.1 Entropy

If a coin is tossed one time, the result could be either head or tail. If the result is interpreted as binary 0 or 1, it could be stored in one bit. However, if the coin is tossed six times in a row, the result could be stored in 6 bits, assumed the tosses are independent from each other. The number of possible toss-combinations is

$$2^6 = 64 \tag{2.13}$$

The English language consists of 26 symbols. How many bits are needed to code one character, presupposed that all symbols have the same probability to occur? Using the same idea as in (2.13)

$$2^4 = 16 \tag{2.14}$$

shows that four bits give 16 possibilities, which is less than 26 and not enough to encode a character. If five bits are used

$$2^5 = 32 \tag{2.15}$$

there are 32 possibilities, which is more than enough. Since the goal is to use as few bits as possible, something between 4 and 5 bits would be suitable. But the number of bits are always expressed in integers; the term to use in this case is *entropy*. Salomon [1997] shows that when each symbol has the same frequency and is independent of each other, entropy can be defined as

$$H = log_2 \, s \tag{2.16}$$

where H stands for entropy and s is the number of different symbols. The entropy for coding one English character would then be

$$H = log_2 \, 26 = 4,700439\ldots \tag{2.17}$$

But the frequencies of all symbols are not always the same, which is obvious when looking at the English language.

Entropy of a source

Phamdo [2000] states that the entropy of a source is a value depending on the statistical nature of the source data. A data source with n characters and s different symbols has a symbol alphabet

$$X = \{x_1, x_2, x_3, x_4, \ldots, x_s\} \tag{2.18}$$

where $x_1, \ldots, x_s$ are the symbols. The symbol x_i is assumed to occur in the data with probability P_i. The sum of probabilities for all symbols is

$$P_1 + P_2 + P_3 + \ldots + P_s = 1 \tag{2.19}$$

In the special case, where all symbols have the same probability,

$$P_i = P \tag{2.20}$$

and therefore

$$1 = \sum_{i=1}^{s} P_i = sP \tag{2.21}$$

which gives

$$P_i = \frac{1}{s} \tag{2.22}$$

If (2.16) gives the entropy for one symbol,

$$H = n\, log_2\, s \tag{2.23}$$

gives the entropy for the sequence of n bits if they are statistically independent. (2.23) together with (2.22) results in

$$H = n\, log_2\, s = n\, log_2(\frac{1}{P_i}) = -n\, log_2\, P_i \tag{2.24}$$

When a symbol x_i occurs with probability P_i, it occurs on average nP_i times in the data source. Therefore, a symbol x_i has the entropy

$$H = -n\, P_i\, log_2\, P_i \tag{2.25}$$

The total entropy for the whole data source is therefore

$$H = -n \sum_{i=1}^{s} (P_i\, log_2\, P_i) \tag{2.26}$$

or on average

$$H = -\sum_{i=1}^{s} (P_i\, log_2\, P_i) \tag{2.27}$$

bits per symbol. (2.27) is called the entropy of a data, where each symbol has different probability, but is independent of each other. [Salomon 1997]

2.5 Coding theory

Coding is a technique for transforming a source alphabet $S = \{s_1, \ldots, s_m\}$ into a code alphabet $A = \{a_1, \ldots, a_n\}$ and backwards. Coding has applications in various areas as error correction, cryptography and data compression. [Hankerson, Harris & Johnson 1998]

According to Bell, Cleary & Witten [1990], Shannon's *noiseless source coding theorem* shows that the average number of bits per source symbol can be made to approach the source entropy but not less. By this reason, the entropy indicates the limit of how good codes can be chosen to encode a message.

2.5.1 Variable size codes

Assume that a data source consists of four different source symbols

$$\{a_1,\ a_2,\ a_3,\ a_4\} \tag{2.28}$$

These symbols occur with identical frequency, thus the probabilities for all symbols are equal to 0.25. The entropy is

$$H = -\sum_{i=1}^{n}(P_i\, log_2\, P_i) = -4\,(0.25\, log_2\, 0.25) = 2 \tag{2.29}$$

bits per symbol. In this case, the symbols can be assigned four 2-bits codes

$$\{00, 01, 10, 11\} \tag{2.30}$$

Next, consider instead the following symbols $\{b_1,\ b_2,\ b_3,\ b_4\}$ with probabilities $\{0.3,\ 0.55,\ 0.05,\ 0.10\}$. The entropy in this example is

$$H = -\sum_{i=1}^{n}(P_i\, log_2\, P_i) = \tag{2.31}$$

$$-(0.3\, log_2\, 0.3\ +\ 0.55\, log_2\, 0.55\ + \tag{2.32}$$

$$0.05\, log_2\, 0.05\ +\ 0.10\, log_2\, 0.10) = 1.54\ldots \tag{2.33}$$

If these symbols were coded with four 2-bits codes, the average code size would be 2 bits per symbol. According to Salomon [1997] *redundancy* is defined as the difference between the current average coding size and the entropy. In this case, the redundancy would be

$$R = 2 - 1.54 = 0.46\,\text{bits / symbol} \tag{2.34}$$

Symbol	**Probability**	**Code**	**Code length**
b_1	0.30	01	2
b_2	0.55	1	1
b_3	0.05	011	3
b_4	0.10	000	3

Table 2.2: Symbols, probabilities and codes

But what happens if the different symbols are coded with different code lengths? Assume that the symbols are coded according to table 2.2

The average size is then

$$\text{Average size} = 2 * 0.3 + 1 * 0.55 + 3 * 0.05 + 3 * 0.10 = 1.6 \tag{2.35}$$

which results in the redundancy

$$R = 1.6 - 1.54 = 0.06 \tag{2.36}$$

It is thus clear that it is possible to achieve results quite close to the entropy by using variable-size codes. But it is important to point out that the redundancy is highly dependent on the spread of probabilities for the symbols. If the probability for one symbol is large, the entropy for this symbol will be small. Since variable-size codes are limited to be at least one bit, this can result in high redundancy. [Salomon 1997]

2.5.2 Prefix codes

When considering the codes in table 2.2 it is trivial to observe that these codes are hard or actually impossible to decode. If the first bit is 1, it must be symbol b_1, since this is the only code that starts with a bit 1. But if a sequence starts with a 0, it could be one of $\{b_1, b_2, b_3\}$. In this case, the following bit must be investigated in order to decide which symbol it is. If the second bit is 0; the only possible symbol is b_4. But if it is 1, the third bit has to be investigated. The problem is that a third bit does not exist for symbol b_1, which makes it impossible to map the sequence 01 to an unambiguous symbol. These codes are therefore unusable when trying to decode the data. But there are codes that have the same code length and are possible to decode correctly. An example is shown in table 2.3.

The property that makes it possible to map the codes in table 2.3 to the correct symbols is called *prefix property*. This property says that once a

Symbol	**Code**
b_1	01
b_2	1
b_3	000
b_4	001

Table 2.3: Correct codes

certain bit-pattern is assigned as a code of a symbol, no other codes are allowed to start with that pattern, i.e. the pattern cannot be the prefix of any other code. A *prefix code* is a variable-size code that satisfies the prefix property. The difficult task is to design smart variable-size codes that achieve the best possible compression ratio. The idea is to assign shorter prefix codes for more frequent symbols and longer prefix codes for symbols occurring less often. [Salomon 1997]

There are some different methods to do this, where the most known are Shannon-Fano and Huffman coding. These methods will be discussed in later chapters.

2.6 Structure of compression algorithms

Data compression generally takes an input stream and transforms the symbols or groups of symbols into codes, which are written to the output. The decision of which code that should be written is determined by a *model*. The model consists of a collection of rules, which together with the source data decides which code should be used. The coding part of a compression algorithm receives information from the model and writes a suitable code to the output stream. Modeling and coding are two distinctly different phenomena that are often mixed up. It is common that people are talking about compression algorithms, when they mean coding. [Nelson & Gailly 1996]

The compression procedure with modeling and coding is illustrated in figure 2.2. The output from the model depends on the type of modeling technique used. Different modeling and coding techniques will be discussed later in this section.

In addition using different modeling and coding techniques, there are methods to transform the input data. These methods do not actually compress the data, but reconstructs it, so that it can be better compressed by another method. Salomon [1997] describes these techniques as *other methods* of data compression. In this study, this type of method forms a separate part of data compression called *transformation techniques.*

The following part of this section gives an overview of coding, modeling and transformation techniques. More detailed descriptions and discussions can be found in chapter 4, 5 and 6.

2.6.1 Coding

As previously shown, Information Theory makes it possible to calculate the number of bits necessary to encode information. Unfortunately, the practical problem still remains. The normal character encoding, for example ASCII-coding, was not even close to the entropy, since all characters or symbols are encoded with the same bit length. [Nelson & Gailly 1996]

The first encoding method dicovered to produce good variable-size prefix codes is called Shannon-Fano coding. The basic idea was to do a forward pass over the symbols frequency, giving the most frequent symbols a shorter bit length. This method rendered good result but gave a slightly sub-optimal effect for integer-length codes. [Salomon 1997]

Huffman coding, named after its inventor D.A. Huffman, was the solution to Shannon-Fano coding sub-optimality. This technique achieves the minimum possible amount of redundancy for integer variable-length codes. Instead of using a forward pass like in Shannon-Fano coding, Huffman coding uses a

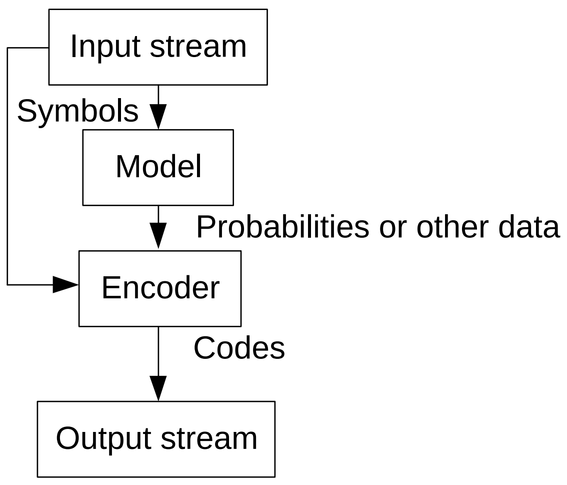


Figure 2.2: Compression procedure with model and encoder

backwards pass to calculate the codes. [Bloom 1996]

In time, a more sophisticated method for coding symbols with different probabilities was developed. *Arithmetic coding* codes symbols close to the size of the entropy, presupposed that the input data is large. Instead of producing a single code for each symbol, it generates one entire code for the whole data source. [Nelson & Gailly 1996]

Further descriptions and discussions of different coding techniques are given in chapter four.

2.6.2 Modeling

Modeling of lossless data compression can, according to Nelson & Gailly [1996], be divided into two different types: statistical and dictionary-based modeling.

The goal of statistical modeling is to estimate the probability of different symbols. The simplest form of statistical modeling uses a static table, which contains the symbols probability. This statistics can either be stored in the compression and decompression units, or be gathered before the actual coding is done. Another approach is to use an adaptive model, where the statistics are collected from the data already processed. When only the frequencies for each symbol is assembled, the model is called an order-0 model. [Nelson & Gailly 1996]

Dictionary-based models use a different approach. Instead of finding the probabilities for the symbols, it searches matching strings in a dictionary. If a match is found, the string can be substituted with a pointer to the dictionary. For generalized lossless data compression, i.e. compression that does not have to know the content of the data, adaptive methods for generating the dictionary are used. The two fundamental approaches of adaptive dictionary-based models were invented by Jacob Ziv and Abraham Lempel in 1977 and 1978. The models LZ77 and LZ78, have later been further developed, which has resulted in a large number of improved variants. [Nelson & Gailly 1996]

More detailed information about modeling techniques are given in chapter 5.

2.6.3 Transformation

The key idea of transformation methods is not to perform the compression itself. Normally, the output data from a transformation unit returns exactly the same amount of data, but in a totally different structure.

MTF, or *Move-to-front*, is a transformation method that codes symbols occurring closely to each other into small values. The output data can then give better compression with an ordinary compression method, than if the transformation was not performed. [Salomon 1997]

Block sorting, or *Burrows-Wheeler Transformation*, transforms the input characters, so that equal characters are placed closely to each other. In other words, the output data consists of the same symbols, but in a different order. After this transformation is done, other transformations or coding techniques can be applied to achieve higher compression. [Burrows & Wheeler 1994]

If characters next to each other in an input data have small differences in value, then a *differential coding* can transform the data into numbers expressing the differences between the values, instead of the absolute values. Thus giving better opportunities for later compression methods to achieve good results. [Bell, Cleary & Witten 1990]

A prerequisite for all the transformation methods described above is that they are reversible, i.e. a backward transformation can be done, so that the original data is restored.

Further details of transformation techniques are discussed in chapter six.

2.6.4 Compression algorithms

Combining the above described techniques gives different possible compression algorithms. In chapter seven an overview is given, showing different possible combinations that form usable algorithms.

2.7 Properties in a WVPN

A wireless VPN consists basically of software installed on server and clients. This software is situated at a lower level than the applications, with the goal that it should be fully transparent so that neither the applications on the server, nor the applications on the client know that it exists. [Hovmark 2001]

2.7.1 Layers in a WVPN

At this time, the concept of wireless VPN is totally new. Because of this, this chapter describes properties concerning Columbitech's WVPN. Most properties should be possible to generalize to other implementations in wireless data communication.

At the session layer, all data is sent as a long stream. The session layer is responsible for handling a user session and to encrypt the data for security matter. According to Hovmark [2001], encrypted data cannot be compressed. Therefore, data compression is done before encryption. The WVPN does not actually have any information about what sort of data that is transmitted. Simplified, it just takes one packet, encrypts it and sends it away. Each packet is normally around 1500 bytes large. [Hovmark 2001]

Thus, to be able to achieve good compression, some sort of streaming compression must be used.

2.7.2 Client properties

There is a wide range of different client devices that could be used in a WVPN. Since the wireless industry is growing fast, new devices are constantly developed and released on the market. In this study, the three most interesting platforms in Columbitech's point of view are investigated: Pocket PC, Palm variants and laptops with Windows 2000 as OS. [Hovmark 2001]

Pocket PC

Pocket PC is a handheld mobile device, which uses Windows CE 3.0 as the underlying operating system. Windows CE is an embedded and multitasking OS from Microsoft that supports 12 different processor architectures. The OS is a 32-bits operating system and is used as operating system for many different handheld devices. [*Windows CE 3.0 FAQ* 2001]

Pocket PC devices used in this study are:

- Cassiopeia E-115, 131 Mhz MIPS processor, 32 MB memory
- Compaq Aero 2130, 70 Mhz MIPS processor, 24 MB memory
- Compaq iPAQ 3630, 206 Mhz Intel StrongArm, 32 MB memory

Palm

The Palm OS operating system is the most widely used OS on handheld devices. According to Palm inc. the market share of Palm OS is at least 90 percent in the US retail channel. [*Palm OS - platform*]

Palm inc. markets a number of different Palm devices with different properties. There are also other companies that sell devices with Palm OS as operating system. The US company Handspring markets a number of different versions of their device *Visor*. [*Handspring homepage* 2001]

One bottleneck in Palm devices is the amount of memory that can be used by a software application. Even if the device itself supports around 8 MB memory, just a part of it can dynamically be used by an application. For Palm OS 3.0, the dynamically usable memory bound is 96 Kbytes, where 32Kbytes are reserved for the TCP/IP handling. [*Palm OS Memory Architecture*]

The Palm device used in this study is:

- Palm Vx 8 MB memory

Laptop with Windows 2000

There are an enormous amount of different types of laptops from a number of companies. As long as the laptop can have Windows 2000 as the underlying operating system, any device should be able to run in the WVPN.

Memory restrictions on a laptop using Windows 2000 is not set by the operating system, but of the physical hardware in the device. The processing capacity for different laptops varies from model to model. In this study the following device is used:

- Toshiba Satellite Pro 4300, Pentium III 650 Mhz 192 MB memory

2.7.3 Server properties

The server software of the WVPN could either be running stand-alone on a server, or together with other server software. The processing capacity for a server depends on the load on the server and the speed of the hardware. How much processing power a single user can get from the server, depends on the number of simulations users connected to the server. [Hovmark 2001]

2.7.4 Network carrier

Nowadays, there are a number of different network techniques used to transfer data wirelessly. The following network technologies are theoretically studied in this work.

GSM Nowadays, the most commonly used network technology is the Global System for Mobile Communication, *GSM*. This technology was invented and launched in Europe and has become the world leading wireless communication standard. It now serves over 200 million users on five continents. GSM is used both for voice and data access, with the maximal bandwidth of 9,6 Kbit/s.
[*GSM - digital mobile radio technology for beginners*]

GPRS A GSM system must first call and connect before data can be started to be delivered, which can take several seconds. To resolve this, the European Telecommunications Standards Institute, *ETSI*, has developed a General Packet Radio Service, *GPRS*. This packet switching data service requires network resources and bandwidth just when data is being transmitted. GPRS can use up to eight 14.4 Kbit/s time slots simultaneously. Thus, the theoretical maximal bandwidth is around 115 Kbit/s. [*GPRS - Data transmission for mobile telephony*],
[*GPRS - General Packet Radio System* 2000]

CDPD stands for Cellular Digital Packet Data, and is a standard used to get wireless data access. [*CDPD - a whatis definition*] The network technique is mainly used in the United States and can give a bandwidth up to 19.2 Kbit/s. The technology started to appear in middle of 1994 and is now available in most large cities in the states. [Geier 2000]

HSCSD stands for High-Speed Circuit-Switched data. This is an highspeed version of GSM, which should theoretically give a bandwidth of 38.4 Kbit/s. [*HSCSD - a whatis definition*]

Bluetooth is, in contrast to the above mentioned technologies, a wireless network technique that has higher bandwidth, but much shorter access range. The raw data rate that can be achieved from one access point is up to 1 Mbit/s. [*Motorola Bluetooth [FAQ]*]. But the bandwidth is highly dependent on the number of users and the distance between the client receiver and the access point. Thus, practical bandwidth is much lower than the theoretical. [Hovmark 2001]

WLAN The term WLAN, or Wireless local area network, is often used when talking about standard IEEE 802.11. This sort of wireless network is associated with high bandwidth and short access range. The first standard 802.11 can reach a bandwidth up to 1-2 Mbit/s. The newer version 802.11b, which is the standard commonly used today, can give a bandwidth of 5.5 Mbit/s or 11 Mbit/s. [*802.11a - a whatis definition*]

Chapter 3

Method of this study

This chapter describes how the study was made and how the collection of data was done. Further, a description is given about how the compression experiment was performed.

3.1 Introduction

When beginning with this study it was clear that the goal was to find out a way to achieve better performance in a wireless data communication. But it was not obvious which parameters that affected the performance. Data compression was one natural choice of reducing data sent over the network. Since the author of this study was inexperienced in the field of data compression, one of the largest parts of the study was to investigate and describe different methods and approaches.

The challenge in this study was to get a result about when and with which data compression methods better performance can be achieved. To be able to come to this conclusion, a deep knowledge about how compression methods are designed and can be implemented must be reached. Since the field of data compression is huge, and there has been extensive research made in the last fifty years, it is major challenge to reach a through understanding pf each algorithm and at the same time overview of all different methods.

3.2 Collection of information

In the beginning of the work, the goal was to get a broad overview of the concept of data compression. Relevant information was found in books, articles and on the Internet. The books gave an overview of the field and the articles and information found on the Internet gave more detailed descriptions of the area.

Most of the information found describes different algorithms for compressing data, especially compression of text. The descriptions treated mostly the design of the algorithms and less the actual implementations. Few sources were found that described the collaboration between data compression and data communication, especially in the field of wireless data communication.

The field of wireless virtual private networks is a new area, where written information about properties is hard to find. Information regarding this area was mainly received from the employees at Columbitech.

3.3 Choice of algorithms

Since the goal of the study was to create a strategy to achieve better performance in a WVPN, as many different compression approaches as possible had to be investigated. Algorithms and especially implementations of algorithms consist of combinations of different technologies. Therefore, the choice was made to relatively extensively describe the different parts of compression algorithms. The algorithms described and analysed in this study are chosen according to the following criteria:

- The algorithm occurs frequently in different literature sources.
- The algorithm is said to have good properties.
- The algorithm has a historical value.

In different literature, the dividing up of algorithms in different parts differs slightly. Most literature found divide it into the parts *coding* and *modeling.* It is more uncertain where dictionary methods and transformation techniques should be placed. In this study, the decision was made to divide it into three parts: *coding technique*, *modeling technique* and *transformation technique.*

3.4 Implementation and experiment

To enable better understanding of different algorithms and how they could be implemented, the author has implemented a number of compression methods. The implemented algorithms are:

LZDB3 This is a quite fast, streaming based, implementation of a variant of the algorithm lzss. No prefix coding technique is used for coding, just fixed size codes. Searching is done by using hash tables.

AHUFFDB1 This is an implementation of adaptive Huffman coding, using a simple order-0 model.

SARITHDB1 This is an implementation of arithmetic coding, using a static order-0 model.

These implementations were also included in the compression experiment. Further information about them are given in chapter 8, *Experiment and result.*

Since the goal was to find the best possible solution to achieve better performance, other person's implementations were also included in the experiment. Both the compression ratio and the speed of the compression and decompression units depend not only on the algorithm design, but also on the implementation. The third part implementations were chosen, so that they covered as many combinations of different compression approaches as possible. For some of these implementations, the source code was not available. For this reason, the actual algorithm implementation can be difficult to analyse. The reason to include these implementations in the test was to show how good data compression algorithms could be implemented if done correctly.

3.4.1 The data compression test

After the different compression implementations were chosen, test data was collected. Document types were chosen, which were most probable to be transferred in a WVPN. This decision was made after discussions with Columbitech's CTO, Torbjörn Hovmark. The data was collected from randomly chosen sites on the Internet. All files for a certain document type were then stored into one single file.

After the data collection was done, a test script program was written. This program executed all compression implementations for each document type. Statistics about the uncompressed file size, compressed file size and execution time for compression and decompressing were written to a log file. Each compression algorithm was run three times to ensure that the calculation time did not vary much for each try. All three tries were saved in the log file. Since the tested files were quite large, they had to be stored on a hard disc, which gives an extra time for the compression implementations to read and write the data to disc. To eliminate this extra time, a program was written that just read and wrote the file to disc. These read and write times were then subtracted from the compression and decompression time.

The log file was inserted into an Excel document, where compression ratio, compression bandwidth and decompression bandwidth were calculated. The deviation of compression and decompression time between the trials was also calculated.

These statistics were then used to analyze the result of the experiment.

3.4.2 Bandwidth for different devices

The compression and decompression bandwidth that were achieved in the compression test was the result when the compression implementations were running on a stationary Windows 2000 computer with Pentium III 800 Mhz processor. To be able to get the compression and decompression bandwidths for client devices such as Palm, Pocket PC and laptop, the compression program could have been run on these platforms. This was unfortunately not possible, since many algorithm implementations required more memory than the devices had. Furthermore, the memory required to save the data on the device was not even nearly enough. Thus, another solution had to be used to estimate the compression and decompression bandwidth for these devices.

This was done by creating a reference compression program, which was run on all different devices and the reference computer. The program consisted of the lzdb3 implementation and a 30Kbytes html document. The compression algorithm was then executed a number of times, and the execution times were measured. Then a scale factor for each device was calculated so that the bandwidth compared to the reference computer could be estimated.

3.5 Analysis and conclusion

From the discussion about different compression algorithms benefits and disadvantages and the explicit statistics achieved from the compression experiment, conclusions are made. Since the number of parameters involved and the uncertainty surrounding them, the conclusions have a outline characteristic.

3.6 Source of errors

In the following section, source of errors that might influence the result of this study, are listed.

- Probably not all types of lossless data compression algorithms that exists have been managed in this report. This might result in that there are other methods that are better suitable than the described in this study.
- The comparison of compression and decompression bandwidth for the different devices could give a misleading result, since the compression experiment was only performed on one platform. The scale factor estimated for the different platforms was calculated by using just one compression method. Since different CPUs have different characteristics in the sense of cost to allocate memory and to handle different instructions, the result could be a bit misleading.
- Since the memory usage for algorithms is hard to measure, the approximately estimated values are not exact.
- In the experiment program, data that is compressed is read from disc, which can influence the compression and decompression bandwidth. Parameters that can affect this are the disc cache in Windows 2000 and the IO-implantations in the test program.
- To be able to analyse the results from the experiment, network bandwidth for different network carriers has to be estimated. The practical bandwidth can under various circumstances be different, which results in that the estimated bandwidths can be incorrect.

- The collected data for the experiment is as randomly selected as possible. Since the compression ratio is highly dependent of the data structure, the compression ratios achieved from the experiment can and most probably will be different for other documents than the tested ones.

Chapter 4

Coding techniques

This is the first of three chapters describing different parts of data compression. This chapter describes how the most common coding techniques are designed and gives a brief overview on how they can be implemented. Techniques described are Shannon-Fano coding, Huffman coding, Arithmetic coding and Range coding. Furthermore, a discussion about the techniques concerning memory usage, performance and compression ratio is given.

4.1 Shannon-Fano coding

Shannon developed the first method to code messages according to their probabilities. At roughly the same time, R. M. Fano at M.I.T. developed a similar approach, therefore the name Shannon-Fano coding. [Bell, Cleary & Witten 1990]

According to Bell, Cleary and Witten [1990], the method can simply be described as follows:

1. List the probabilities for all symbols in decreasing order.
2. Divide the list into two parts, so that the sums of the probabilities in each part are as equal as possible.
3. The code for the symbols in the first part should start with 0, and those in the second part with 1.
4. Continue recursively until each subpart contains just one symbol.

4.1.1 Evaluation

Shannon-Fano coding is one of the oldest coding techniques in the era of modern data compression. In the following section different properties, for this coding technique, is discussed.

Compression ratio

To be able to understand how good compression ratio the Shannon-Fano coding can achieve, its optimality property is first discussed.

Shannon showed that the average code length for Shannon-Fano codes lie in the range $[H, H+1]$ where H is the entropy. [Bell, Cleary & Witten 1990] But does this coding technique give the optimal prefix codes? The answer is no, even if the result is quite close to optimal fixed length codes. To prove that it is not optimal an example is given.

For a given set of symbols $S = \{s_1, s_2, s_3, s_4, s_5\}$ the probabilities are $\{0.35, 0.17, 0.17, 0.16, 0.15\}$. The symbols are recursively divided into groups as described in the algorithm above, which results in the codes shown in table 4.1.

Symbol	**Code**
s_1	00
s_2	01
s_3	10
s_4	110
s_5	111

Table 4.1: Codes generated by Shannon-Fano coding

The average code length for these codes is

$$L = 0.35*2+0.17*2+0.17*2+0.16*3+0.15*3 = 2.31\,bits/symbol. \quad (4.1)$$

This result can be compared to another set of prefix codes $\{1, 011, 010, 001, 000\}$ which have the average code length

$$L = 0.35*1+0.17*3+0.17*3+0.16*3+0.15*3 = 2.30\,bits/symbol. \quad (4.2)$$

It is therefore proven that Shannon-Fano coding does not result in optimal codes, but with codes that are close to optimum. The advantage with this technique is its simplicity, even if it does not construct optimal codes.

The problem with this coding technique, and all other prefix code coding techniques, is that the lowest entropy a symbol can be encoded to is 1, since one bit is the smallest representation. This does however only produce high redundancy when the probability for some symbol is very high.

It is actually not possible to answer the question how good compression ratio a Shannon-Fano coding technique can achieve, since this depends on the model used to predict the frequencies. If the model is adaptive, with higher orders, the coding algorithm has better possibilities to render a good compression ratio. There is also no need to transmit the probability data. But if the algorithm is static, both the sender and the receiver must know the frequencies of the symbols. One way of doing this is to transfer this information before the actual coding stream, but this results in an overhead. For a large data block, this does not significantly impact on the compression ratio, but for smaller blocks, the overhead can be larger than the original data. Thus, sending the frequency data does not seem to be a good alternative when dividing and sending data in small blocks.

Performance

The performance of the method is of course dependent of its implementation. Fetching the codes can be done for each input character read, but this is not a suitable solution for static coding. For this, it is better to read out all the codes from the beginning, and then saving it into a tree or an index table.

Memory requirements

The amount of memory required for encoding and decoding Shannon-Fano codes, is directly proportional to the number of symbols used. Exactly how much memory that is required depends on the implementation, but generally this coding technique does not require so much memory. Mainly, the memory required is to store the probabilities and to do the sorting of probabilities.

Streaming possibilities

The literature found concerning Shannon-Fano coding describes only Shannon-Fano coding as a method for static or semi-adaptive coding. For this purpose, the coding technique can give a relatively good result on large blocks. As mentioned earlier, the overhead for transmitting statistics can result in devastating result for small source blocks. Thus, semi-adaptive coding could not be recommended for streaming purposes.

4.2 Huffman coding

A short time after the introduction of Shannon-Fano coding, D. A. Huffman of M.I.T. discovered a new method that was proven to give optimal prefix codes with given probabilities. The coding technique D. A. Huffman developed is called static Huffman coding and is basically implemented via building a so called Huffman tree. This method is static, since it needs all probabilities to be able to calculate the mapping codes. Later, a new adaptive version of Huffman coding was developed, which made it possible to relatively fast update the tree and getting new Huffman codes. [Salomon 1997]

4.2.1 Static Huffman coding

The algorithm for constructing prefix Huffman codes first builds up a Huffman tree, and then reads out the codes from the tree. According to Bell, Cleaty and Witten [1990], the algorithm to build the Huffman tree can be described as follows:

1. Line up all symbols in a list with falling probabilities.
2. Locate the two symbols with smallest probability.
3. Link these two symbols together and create a new symbol, which have the probability equal to the sum of the two located symbols.
4. Replace the located symbols in step 2 with the new symbol created in step 3.
5. Repeat from step 2 until the list contains only one symbol. This symbol should now have the probability 1.

The tree created above contains all necessary information to read out the Huffman codes. This is done by traversing the tree from the root, down to the leaf where the original symbols are situated. If it is a left branch a 0 is added to the code, and if it is a right branch a 1 is added. [*Huffman Coding* 1997-2000]

Example

A given alphabet with four symbols

$$S = \{s_1, s_2, s_3, s_4\} \tag{4.3}$$

can be straight forward coded as a 2-bits codes. A message containing 10 character

$$X = \{s_1, s_2, s_1, s_4, s_3, s_1, s_2, s_1, s_4, s_1\} \tag{4.4}$$

has the symbol probabilities

Symbol	**Probability**
s_1	0.5
s_2	0.2
s_3	0.1
s_4	0.2

Table 4.2: Huffman: Probabilities of symbols

When applying the algorithm on this data, symbol s_3 and s_4 are grouped into a new node called n_1. This new node has the probability 0.3. The tree build so far is illustrated in figure 4.1.

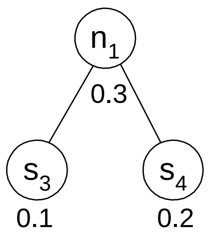


Figure 4.1: Huffman tree with 3 nodes

Now there is three possible nodes / symbols left $\{s_1, s_2, n_1\}$ with probabilities {0.5, 0.2, 0.3}. The two nodes with lowest probability is $\{s_2, n_1\}$, which together results in a new node n_2 with probability 0.5. The tree is illustrated in figure 4.2.

There are two nodes / symbols left: $\{s_1, n_2\}$ with probabilities {0.5, 0.5}. These are grouped together, which results in the final Huffman tree, given in figure 4.3.

The codes are read out from the Huffman tree from top to down. For example, to get the code for s_4, start at the top node n_3 an go $\{left, left, right\}$ which gives the code 001. The complete list of codes are shown in table 4.3

The average code lengths for this example is

$$l = 0.5 * 1 + 0.2 * 2 + 0.1 * 3 + 0.2 * 3 = 1.8 \tag{4.5}$$

and the entropy for the given probabilities

$$H = 0.5 * log_2 5 + 0.2 * log_2 2 + 0.1 * log_2 1 + 0.2 * log_2 2 = 1.56 \tag{4.6}$$

Thus, the Huffman codes achieve better compression ratio than the intuitive byte coding, but not as good as the entropy. It is shown that Shannon-Fano coding does not give optimal fixed codes, but it is proven that Huffman codes do. For a proof of this optimality, the reader is referred to the book *Introduction to Information Theory and Data Compression* by Hankerson, Harris and Johnson [Hankerson, Harris & Johnson 1998].

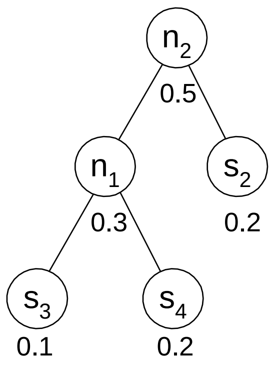


Figure 4.2: Huffman tree with 5 nodes

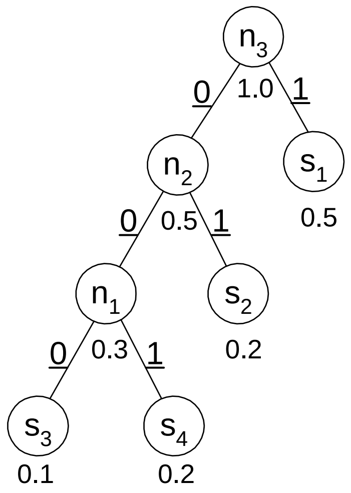


Figure 4.3: Final Huffman tree

Symbol	Huffman code
s_1	1
s_2	01
s_3	001
s_4	000

Table 4.3: Huffman codes

The static Huffman encoding and decoding algorithm assumes that both the sender and the receiver have the symbols probabilities available. For some applications, for example message data containing plain English text, it would be possible to use the same statistics for all data. But in most cases, the sender has to transmit this information to the receiver. Since the size of the Huffman tree is proportional to the number of symbols used, this data size does not change according to the message length. Therefore, this overhead is not significantly large on larger messages, but for shorter messages, the overhead can be greater than the original message. [Storer 1988]

Canonical Huffman

Canonical Huffman is a method to minimize the size of the overhead achieved when transmitting the statistics. The canonical representation consists of a number of rules, which makes it possible to just send the code lengths, instead of both the code lengths and the Huffman code. Details on this method is given in a article by Arturo S.E. Campos [Campos 1999]

4.2.2 Adaptive Huffman coding

Adaptive coding, as described in chapter 2, means that the codes change during the encoding phase. Compared to the static method, the adaptive method reads the data only once and does not need to save any extra information about the probabilities. The information regarding the probabilities is collected from the previously data seen. [Bell, Cleary & Witten 1990]

An intuitive but naive solution to perform adaptive Huffman coding would be to collect the probabilities when reading the data, and then for each symbol read reconstruct the Huffman tree. If the decoder uses the same procedure, the result would be correct. Unfortunately it would be rather inefficient. Especially if the number of symbols is large. [Bell, Cleary & Witten 1990]

There is a much more efficient method, which makes use of the so called *Sibling property.* According to Lelewer & Hirschberg, the sibling property is valid if

1. Each node, except the root node, has a sibling.
2. All nodes in the tree can be listed by weight in a decreasing order so that each node is adjacent in the list to its sibling.

Then the tree is a Huffman tree if and only if it has the sibling property. According to Storer [1988] an efficient adaptive Huffman encoding can be done as follows:

1. In addition to the tree, a list is maintained with the nodes in the tree listed in decreasing order of weight.
2. Read a new character from the input stream, until the end of the stream is reached.
3. Set the leaf node, which corresponds to the new character, as current.
4. The weight of the current node is increased by one.
5. If the current node is the root node, go to step 2.
6. If the new weight of the node is less or equal to the left node in the list, go to step 8.
7. The current node is exchanged with the left most node in the list that has a smaller weight than the current node.
8. Set the parent node as current.
9. If the last current node was a left child, add a 0-bit to the output code, else add a 1-bit.
10. Go to step 4.

Since advancing one level up in the tree generates an output bit, this algorithm works in linear time, presupposed that the search for a switch node in step 7 can be done in constant time. [Storer 1988]

One detail that is important to notice is what will happen if the weight of the nodes becomes too large, and does not fit into an dword. A simple approach is to scale down all weights when an dword reaches its maximum. [Storer 1988]

There are also different ways of handling the initialization of the probabilities for symbols. One way of solving this problem is to initially give all symbols the same probability. The probabilities will then successively change while compressing the stream. Another solution is to insert an extra escape symbol, with a zero probability. Then when this symbol is seen, a new symbol is followed, which will be inserted into the Huffman tree. [Storer 1988]

4.2.3 Evaluation

Compression ratio

Huffman coding achieves, compared to Shannon-Fano coding, optimal prefix codes. Thus, it should be able to give a bit better compression ratio than Shannon-Fano coding. The coding itself does not give any compression, but together with a well chosen model, it has the ability to render a good result.

Performance

Building the Huffman tree could be done in near linear time and achieve good performance. The performance is however highly implementation dependent. Implementing a tree structure can be done in several different ways, with different benefits and disadvantages. The traditional way, where the memory is allocated dynamically for each node, can result in poor performance, especially on platforms where memory allocations are expensive. On the other hand, dynamically allocated trees do not allocate more memory than needed. Another way of implementation strategy is to statically allocate tables, where the pointers are indices in these tables. This is in most cases the fastest method, but with the drawback that memory must be allocated initially. Furthermore, using indices instead of pointers requires less memory, since an index needs not to be larger than the number of elements in the collection. For example, if there are 50000 nodes, it is enough to use two bytes for each index. In the case of dynamic memory allocation, generally four bytes are needed on a 32-bit processor. Thus, when the number of nodes is small, statically allocated tables can result in less memory required compared to dynamical allocation.

Implementing the adaptive version of Huffman coding is much more complex and therefore harder to make efficient. The difference in performance between adaptive and static Huffman coding is quite large. For a static Huffman coding, most calculations are made initially, when the Huffman tree is generated. After that, each symbol can be read from the tree fast. But in the case of adaptive coding, processing capacity must be used when updating the tree for each character.

Memory requirements

Concerning memory, both the static Huffman method and the adaptive method require memory proportional to the number of nodes in the tree. It is easy to see that the maximal number of nodes is $2 * N - 1$ where N is the number of symbols used. It is also important to point out that if the nodes are less than 256, then just one byte is needed when referencing this node, but if there are up to 65536 nodes, two bytes are needed. The differences in amount of memory are therefore highly dependent on the number of symbols used.

According to Bell, Cleary & Witten [1990], adaptive Huffman coding is invariably used with an order-0 model, since a higher model would consume a lot of memory. The reason for this is that each combination of symbols must have an own Huffman tree. Higher order models also tries to give higher probabilities for symbols. If the probability becomes very high, the Huffman codes cannot code these probabilities close to the entropy. Thus, the effort in memory consumption does not pay back in compression ratio.

Streaming possibilities

Static Huffman coding has the same disadvantages as Shannon-Fano coding, that both the coder and the decoder must know the statistics. An alternative to transmit this information before the data sent is that both the coder and the decoder use predefined statistics. This can render good compression ratios if the documents almost always have the same probabilities. But if the probabilities for a document differ a lot, it could result in no compression or even expansion.

Both static and adaptive Huffman can be in block modes, but if the blocks are too small, the overhead of static Huffman can result in poor compression ratio. On the other hand, adaptive Huffman coding is designed to change behavior during the process. It is said to be relatively fast and can therefore be suitable for streaming.

4.3 Arithmetic coding

This rather new method of coding was not discovered until the late 1970s and became popular in the 1980s. Arithmetic coding does not map symbols into integral number of bits, as in the case for Huffman and Shannon-Fano codes. Instead it takes a stream of input characters and encodes it into a single floating-point output number. The longer the input stream, the more bits are required for the output number. It was not until the 1980s a practical method was discovered, that could perform this method with fixed size integer registers. [Nelson 1991]

An arithmetic coded message is given by an interval of real numbers between 0 and 1. The longer the message is, the smaller the interval representing the message becomes, which results in that more bits are needed to specify this interval. The higher the probability is for a symbol, the larger is the interval. Reverse, the lower the probability is, the smaller is the interval. This results in a less precise decimal number for higher probability, and therefore smaller output code. [Nelson 1991]

The following two sections gives examples of how the arithmetic coding technique encodes and decodes a message. Information regarding the technique is found in Nelson [1991].

4.3.1 Encoding method

For a given alphabet with four symbols

$$S = \{s_1, s_2, s_3, s_4\} \tag{4.7}$$

there is an input message containing 10 characters

$$X = \{s_1, s_2, s_2, s_4, s_3, s_2, s_2, s_1, s_2, s_2\} \tag{4.8}$$

which gives the following probabilities

Symbol	**Probability**
s_1	0.2
s_2	0.6
s_3	0.1
s_4	0.1

Table 4.4: Arithmetic: Probabilities of symbols

Once all the probabilities for the symbols are known, the symbols must be assigned a range between two predefined values. In this example, the whole range is between 0 and 1, but for efficient implementations, it could be other values. It does not matter which symbol that is assigned to which segment. The important thing is the size of each segment range. Table 4.5 shows how the symbols could be assigned different ranges.

Symbol	**Probability**	Range
s_1	0.2	[0.0,0.2)
s_2	0.6	[0.2,0.8)
s_3	0.1	[0.8,0.9)
s_4	0.1	[0.9,1.0)

Table 4.5: Symbols specified with ranges

Note that a symbol owns everything up to, but not including the high boundary. For example, the symbol s_2 has it range between 0.2 and 0.799999... A lower boundary *low* is defined initially to 0.0 and a higher boundary *high* to 1.0. When encoding the message X, the first character seen is s_1. This symbol has the range $[0.0, 0.2)$. Therefore *low* is set to 0.0 and *high* to 0.2.

The next character is symbol s_2, which has a range $[range_{low}, range_{high}) = [0.2, 0.8)$. This range is now scaled to fit into the *coderange*, which is

$$coderange = high - low = 0.2 - 0.0 = 0.2 \tag{4.9}$$

The new boundaries are

$$low = low + coderange * range_{low} = 0.0 + 0.2 * 0.2 = 0.04 \tag{4.10}$$

$$high = low + coderange * range_{high} = 0.0 + 0.2 * 0.8 = 0.16 \tag{4.11}$$

A pseudo code to accomplish this for the whole message is listed below

```
Range_low[x] := Lower boundary for symbol x
Range_high[x] := Higher boundary for symbol x
Set Low to 0.0
Set High to 1.0
While there are more input characters to read
      S := Read in next character
      Coderange := High - Low
      High := Low + CodeRange * Range_high[S]
      Low := Low + CodeRange * Range_low[S]
End of while

Output Low
```

Table 4.6 shows the result when the above algorithm is applied on message X.

Symbol	**Low**	**High**
	0.0	1.0
s_1	0.0	0.2
s_2	0.04	0.16
s_2	0.064	0.136
s_4	0.1288	0.136
s_3	0.13456	0.13528
s_2	0.134704	0.135136
s_2	0.1347904	0.1350496
s_1	0.1347904	0.13484224
s_2	0.134800768	0.134831872
s_2	0.1348069888	0.1348256512

Table 4.6: Arithmetic encoding

The output value is 0.1348069888, which is enough to reconstruct the original message. This will be shown in the following section. [Nelson 1991]

4.3.2 Decoding method

The decoding procedure is the reverse of the encoding. First thing to do is to find which symbol that has the corresponding interval. It is easy to see that the first symbol must be s_1, since 0.1348069888 falls into the interval $[0.0, 0.2)$. Next thing to do is to remove this symbol from the input data. This is done by reversing the encoding procedure. That is, subtract the input code with the *low* value of the symbol and divide with the range of the symbol. This gives the new code value 0.674. This new code value falls into the interval $[0.2, 0.8)$, which means that the next symbol is s_2.

A simple algorithm for decoding the input value is listed below:

```
N = Encoded number
Range_low[x] := Lower boundary for symbol x
```

```
Range_high[x] := Higher boundary for symbol x
Do
      S := The symbol that has it range around N
      Output S
      SymbolRange = Range_high[S] - Range_low[S]
      N := (N - Range_low[S]) / SymbolRange
While no more symbols
```

Decoding message X will generate the output shown in table 4.7

Encoded number	Output symbol	Low	High	Range
0.1348069888	s_1	0.0	0.2	0.2
0.674034944	s_2	0.2	0.8	0.6
0.79005824	s_2	0.2	0.8	0.6
0.9834304	s_4	0.9	1.0	0.1
0.834304	s_3	0.8	0.9	0.1
0.34304	s_2	0.2	0.8	0.6
0.2384	s_2	0.2	0.8	0.6
0.064	s_1	0.0	0.2	0.2
0.32	s_2	0.2	0.8	0.6
0.2	s_2	0.2	0.8	0.6
0				

Table 4.7: Arithmetic decoding

Note that there is no method applied to decide whether there are more symbols to decode. This could be done either with a value stored at the beginning, telling how many symbols the message contains or by encoding a special end of buffer code. [Nelson 1991]

4.3.3 Practical concerns

The procedure of the encoding and decoding arithmetic methods is not too complicated to understand, but at first glance it seems almost impossible to implement in practice. Normal 32-bits processors with floating point calculations will lose precision fairly fast. It turns out that the best way of implementing arithmetic coding is by using normal 16-bits or 32-bits integer math. Instead of generating one single output code, an incremental transmission schema is used, where the fixed integer receives new bits at the low end and shifts out the bits in the high end. In the examples above, the range was set to $[0.0, 1.0)$, which is not possible to achieve with a integer register. Instead the interval is between 0 and 0xFFFF for a 16-bits register. The ranges of the symbols are then divided into this interval instead. [Nelson & Gailly 1996]

The strategy for outputting bits, is that when the highest bits of the *low* boundary and the *high* boundary are equal, they will never change and can therefore be shifted out to the output stream. There are other phenomena that must be taken into consideration when implementing arithmetic coding, such as overflow when the *high* and *low* boundaries are to close to each other. For the interested reader, details on implementing arithmetic coding can be found in the *Data compression book* by Nelson and Gailly [1996].

4.3.4 Evaluation

Compression ratio

It is shown that the Huffman method is more efficient than the Shannon-Fano method, but both have the problem that they produce codes that are integer number of bits. Arithmetic coding uses one long integer number instead to represent the whole coding stream, which results in a coding close to the entropy. While this eliminates the problem that a code cannot be smaller than one bit, it can produce very good compression ratios when the probabilities for symbols are high. Arithmetic coding combined with a strong modeling has the possibility to produce really good compression ratios.

Performance

The performance of arithmetic coding is surprisingly good. When integer values are used, the processing demands are lower and the algorithm can achieve good performance. When the encoder receives the probability range from the model, few calculations need to be done. The time consuming part is not the coder or the decoder, but the way of the model to handle the frequency table.

Memory requirements

The algorithm does not actually require any memory for doing the encoding and decoding. The memory required is part of the model.

Streaming possibilities

Arithmetic coding is excellent for adaptive methods, since it only needs the probability range to be able to encode a symbol. This range can be directly given by the probabilities. Arithmetic coding can also be implemented statically, where the probabilities are sent before the encoded message. This gives the same overhead as described for Huffman and Shannon-Fano coding.

4.4 Range Coding

Range encoding is a variant of arithmetic coding. This technique was first presented on the Video & Data Recording Conference held in Southampton 1979 by G.N.N. Martin. [Schindler 1999]

According to Campos [1999] the difference between range coding and arithmetic coding is that range coding renormalize in bytes instead of bits, and can in this way run twice as fast as arithmetic coding. The coding technique cannot achieve compression equal to the entropy, but Schindler [1999] claims that it just results in 0.01% worse compression compared to arithmetic coding.

More information about the design of algorithm can be found in the article Range coder by Campos [1999] and the original report by Martin [1999].

4.5 Run length encoding

Run length encoding is a simple and frequently used coding technique. The idea is that if a symbol s occurs in the input stream n consecutive times, it can be substituted with a pair nd. n consecutive occurrences of a symbol are called a *run length* of n. Therefore the name *run length encoding* or *RLE.* When encoding a stream with RLE, the encoder must notify the decoder when there is a run length, and when there is a normal symbol. An ordinary way of doing this is inserting an escape symbol when repeating symbols are encoded. [Campos 1999]

4.5.1 Encoding and decoding

A simple RLE encoding algorithm can be described as follows:

```
Choose an escape symbol ESC
While more symbols to read
      S := Read in next symbol
      If S is equal to ESC then
            Output ESC code twice
Continue to beginning of while loop
      End if
      N := Nr of symbols that match S sequentially.
      If N > 2 then
            Output ESC
            Output S
            Output N+1
      Else
            Output S
      End if
End of While
```

Let say an alphabet has 256 symbols, which are stored in the input source as 8-bit characters. The input message is

$$X = 3, 55, 23, 55, 55, 55, 55, 55, 23, 72, 72, 72, 72, 23 \tag{4.12}$$

The escape code could for example be set to 0, since this code does not occur in the input message. The output message, after encoding would then be

$$Y = 3, 55, 23, 0, 55, 5, 23, 0, 72, 4, 23 \tag{4.13}$$

However, if the input message would use all different symbols, an escape code must be chosen that is the least used in the input message. If the escape code is frequently used, and the number of sequentially strings is few, this method can result in expansion.

4.5.2 Evaluation

Compression ratio

RLE compression is of course suitable in applications where repetition of symbols is common. In this case, good compression ratios can be achieved. It is often suitable for binary data, for example monochrome pictures [Kieffer 1999]. But for normal texts, repeating symbols are not that common. This method is therefore often combined with other suitable compression methods.

Performance

This method can be implemented to be very fast, because of its simplicity.

Memory requirements

RLE based methods do not require very much memory, since no tables have to be build.

Streaming possibilities

The problem with using RLE for streaming is that it has to look forward for repeating characters. If repeating characters are divided into several blocks, the ability to achieve compression is decreased.

Chapter 5

Modeling techniques

This chapter covers the second and probably the most significant part of data compression; the modeling technique. The main categories of modeling are in this study divided into statistical modeling and dictionary-based modeling. After the description of each technique, discussions about its properties are given.

5.1 Introduction

In the last chapter, different coding techniques were discussed. These techniques give better compression; the better the upcoming symbols can be predicted.

The first modeling approach described in this chapter, *statistical modeling*, describes different techniques to estimate the symbols probabilities.

The second approach, *dictionary-based modeling*, uses a bit different approach. Instead of giving a probability for a symbol, dictionary modeling replaces a string of characters with a new code. The frequency of these new codes can then be calculated and coded with a suitable coding technique. [Nelson & Gailly 1996]

5.2 Statistical modeling

The fundamental part of a statistical model is to predict which character that will come up next. To be able to do this sort of prediction, the data seen so far can be used as information. When both the compression and decompression routine use the same model to predict the probability for the next character, this information can be used by a coding technique to achieve compression. [Salomon 1997]

The simplest form of statistical modeling is to use a static table indicating probabilities for the symbols. This table must be available for both the model of compression and decompression. If the data corresponds to the static model, good compression ratio can be achieved, but if input data differ from the static model, the output can actually expand instead of being compressed. [Nelson & Gailly 1996]

Another approach, which is called semi-adaptive, makes a pass over the input data and collects information about the probabilities. These probabilities are then uses to encode the data. The drawback with this approach is that the statistical data must be sent along with the data stream, which gives an overhead to the output size. [Nelson & Gailly 1996]

To avoid the overhead of transmitting data with the compressed stream, a statistical model can be designed to be adaptive. An adaptive statistical model adaptively changes the probabilities based on the data already compressed. In this way, the statistical data does not have to be sent along with the output data. [Nelson 1991]

5.2.1 Finite context modeling

Modeling described so far, covers just how often different symbols occur in the input. When this is the case, the model is called an order-0 model. [Nelson & Gailly 1996]

If instead the probabilities for each incoming character is calculated based on the context where the character appears, the term *Finite context modeling* is used instead. The order of the context model refers to the number of previous characters that makes up the context. [Nelson 1991]

For an order-0 model, just one table is needed to keep track of the frequencies for the symbols. But an order-1 model must take into consideration the character that precedes the next coming character. Therefore, a table of symbol frequencies must be saved for each symbol that can occur before the predicted symbol. For a symbol alphabet with 256 symbols, an order-1 model would require 256 tables. Likewise, an order-2 model must be able to handle 65536 different tables. [Nelson 1991]

There are several techniques to estimate the probabilities when using finite context modeling. The finite context models mostly described and used are based on the so-called PPM model.

5.2.2 PPM

J.Cleary and I.Witten developed a method called PPM, which stands for *prediction by partial match.* [Bell, Cleary & Witten 1990]

An overview of the modeling technique is easiest described by an example. Information about the techniques design is collected from Salomon [1997] and Bell, Cleary & Witten [1990].

Example

PPM starts with an order-N context, where N is a context number known both for the model in the compression and decompression unit. Let say that the current order is three and the context is *exp.* This string has been seen 40 times in the past, followed by the symbol e 8 times, symbol l 12 times and symbol o 20 times. The encoder can then assign these symbols the probabilities $\frac{1}{5}$, $\frac{3}{10}$ and $\frac{1}{2}$. If the next character actually was symbol e, then the probability $\frac{1}{5}$ is sent to the encoder. But what happens if the next symbol is an r, which has not occurred yet? This problem, handling zero-frequency symbols, is the main problem with adaptive finite-context modeling, according to Bell, Cleary & Witten [1990]. The way PPM handles this is by inserting a *escape symbol.* When a character with zero probability occurs, the model sends an escape symbol, and jump to the lower order model, in this case to order-2. The procedure is the same for all lower orders. If the zero-probability occurs even for order-0, the so called order-(-1) is used. Here all the symbols have the same frequency, which makes it always possible to encode a symbol.

Variants of PPM

There has been extensive research made in the area of finite-context-models, and a number of variants of PPM have seen its daylight. One of the models known to achieve very good compression ratios is PPMZ, developed by Charles Bloom. [Bloom 1998]

5.2.3 Evaluation

Compression ratio

A statistical model does not achieve any compression ratio, unless it is followed by a coder. But with a suitable coding technique, a statistical model can achieve good compression ratios. According to both Nelson[1991] and Bell, Cleary & Witten [1990], statistical modeling can outperform all other known compression methods. Generally, higher order context gives better compression ratios, which is shown in tests made by Nelson [1991]. But according to Bloom [1998], the major problem with PPM is that different input data have different highest optimal order. This is typically between order-4 and order-8 he claims. Bloom [1999] has developed the model PPMZ, which is an infinite length deterministic context controller, i.e. the maximal order number is not bounded to any particular number. This renders, according to him, the best compression ratios today known.

Performance

The performance of statistical models depends how much historical data that is saved in the model. More data can give more complex searches through the model, and higher order number can result in more escapes to lower models. Furthermore, more data and memory usage risks that modern CPUs makes cache misses, which results in poor performance. According to most authors, adaptive statistical modeling techniques main strength is not the compression and decompression performance, but the compression ratio.

Another aspect that is interesting to point out is that adaptive statistical models must update the model in the same way in both the compression and decompression routine. Thus, the processing requirements are high on both sides.

Memory requirements

According to Bell, Cleary & Witten [1990], the memory usage is almost the same in both the compression and decompression routine. This since the adaptively updated model must have exactly the same appearance on both sides. The memory usage increase with the order number. How much memory different orders or models need is impossible to generally estimate, but the memory increases dramatically with the order number.

Streaming possibilities

The streaming possibilities for adaptive statistical modeling tend to be very good, since the probabilities are estimated character per character. Since the model can be stored between blocks of data, the compression ratio should not become much worse when the size of the blocks decreases. For semi-adaptive modeling, the problem with the overhead of statistical data makes it not so suitable for smaller blocks.

5.3 Dictionary-based modeling

A dictionary-based model reads in input data and looks for groups of characters that match a dictionary. If a string in the dictionary matches, the index to this string could be output instead of the string itself.
[Nelson & Gailly 1996]

When dictionary-based modeling is used, the focus is placed on the modeling technique prior the coding technique. Often just simple coding techniques are used for output coding. If the dictionary is static, the compressing and the decompressing unit must have access to the dictionary. The problems with static dictionary are the same problems that a static model faces. Since both the sender and the receiver must know the dictionary information, either the dictionary must be sent before the code data, or the dictionary have to be saved in the compression and decompression unit[Nelson & Gailly 1996]. Sending the dictionary before code data, gives a large overhead, which can result in poor compression ratio. Having permanent dictionaries can give good result if the input data streams contain similar phrases, but if the different input data is dissimilar, the result can be quite poor compression ratio. [Bell, Cleary & Witten 1990]

But there is another way of handling the dictionaries. In the year of 1977 and 1978, Jacob Ziv and Abraham Lempel described two compression methods using an adaptive dictionary. The algorithms, LZ77 and LZ78, have been widely used since then, because of their good performance and relatively good compression ratio. [Nelson & Gailly 1996]

5.3.1 LZ77

LZ77 was the first adaptive method developed by Lempel and Ziv. The method is also called the *sliding window method*, since the dictionary used is a sliding buffer of characters already encoded. [Bell, Cleary & Witten 1990]

Encoding

The sliding window consists of N characters. The $N - F$ first characters represent the already encoded data and the last F characters is the data to be encoded. These last F characters together is called the *lookahead buffer*. The algorithm then searches in the window after the longest match to the lookahead buffer. The longest match is then output as a triple $< i, j, char >$ where i is the offset in the window to the match, j is the length of the match and *char* is the first character that did not match. This triple is then output and window is shifted $j + 1$ characters forward. The attached explicit character to each pointer ensures that it is possible to code characters even if no match was found. [Bell, Cleary & Witten 1990]

To illustrate the encoding algorithm, a short example is given.

Example of encoding

An input message X contains 17 characters

$$X = \{k, b, a, d, a, k, b, a, d, a, k, b, d, a, a, b\} \tag{5.1}$$

The next symbol to encode is located on position 9 in the X. The current size of the window N is equal to 12 characters and the lookahead buffers size F is 5. The situation is illustrated in figure 5.1

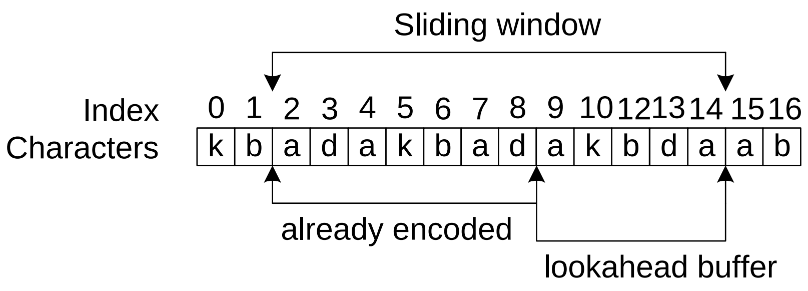


Figure 5.1: Sliding window in LZ77 example

The algorithm then searches for the longest match in the window, and finds it at position 4 in the message, thus position 2 in the window. The match length is three characters. The character after the three matched characters has symbol d. Thus, the triple output is

$$< i, j, char > = < 2, 3, d > \tag{5.2}$$

The window is moved four characters to the left, so that the next character to decode starts at position 14.

Decoding

The decoding process is easy to see and straight forward. Since the output just consists of triples, it is just to read in the triple, find the position in the sliding window and copy as many characters that is given in j to the look ahead buffer. Then copy the explicit character *char* to the end. The sliding window is then moved $j + 1$ characters forward, and the next triple is read from the input stream. [Bell, Cleary & Witten 1990]

5.3.2 LZ78

LZ78 was the second approach developed by Lempel and Ziv 1978. Instead of using a search buffer, lookahead buffer and a sliding window, this method builds up a dictionary of previously strings seen. The output from this method is a two-field token, where the first field indicates a pointer to a string in the dictionary and the second field the next upcoming input character. The tokens do not contain the length of the string encoded, since each pointer points to a fixed size string. [Salomon 1997]

Compress

First, the dictionary is empty and contains only one element 0 with an empty string. Then the first character is read in from the input buffer. Since this character does not exist in the dictionary, a token is output $(0, x_0)$, where the first element is a pointer to the nullstring, and x_0 the first character. The character x_0 is now added to the dictionary in position 1. Then the next character is read in from the input. If this character is equal to x_0, then nothing is output and the next character is read in. Now it searches in the dictionary, to see if the string x_1, x_2 is found. Since this string is not found, the token $(1, x_2)$ is sent to the output, and the string x_1, x_2 is added to the dictionary in position 1. [Salomon 1997]

To illustrate the encoding procedure, an example is given. Assume an input message containing the following character

$$X = \{a, b, c, a, b, d, b, c, a, b, c, a, b, c, d\} \tag{5.3}$$

In table 5.1, the building of the dictionary and the tokens sent to output are shown.

Dictionary item nr	**Dictionary content**	**Output token**
0	null	-
1	a	(0,a)
2	b	(0,b)
3	c	(0,c)
4	ab	(1,b)
5	d	(0,d)
6	bc	(2,c)
7	abc	(4,c)
8	abcd	(7,d)

Table 5.1: LZ78 compressing example

As can be seen in the example above, the strings are continuously added to the dictionary, and longer and longer strings are represented.

Decompress

The goal of an adaptive model is that the dictionary does not have to be sent along with the compressed stream. Thus, the decompressing unit has to build the dictionary in the same way as the compressor. In table 5.2, the decompressing procedure of the above given example is shown.

5.3.3 Other lz algorithms

Almost all practical adaptive dictionary methods are derived from Ziv and Lempel's work. In the following section, some variants are described briefly. It should be pointed out that there exist a lot of other variants of lz techniques, which are not described here.

Input token	**Dic item nr**	**Dic content**	**Total output**
	0	null	
(0,a)	1	a	a
(0,b)	2	b	ab
(0,c)	3	c	abc
(1,b)	4	ab	abcab
(0,d)	5	d	abcabd
(2,c)	6	bc	abcabdbc
(4,c)	7	abc	abcabdbcabc
(7,d)	8	abcd	abcabdbcabcabcd

Table 5.2: LZ78 decompressing example

LZSS

LZ77 uses a series of triples to output pointers, length and end character. Instead of outputting this extra character, the LZSS method searches for the longest string in the windows and checks the length of the longest match. If the length is shorter then the bits required for expressing the pointer and the length, the first character in the lookahead buffer is written to the output. To distinguish between a pointer and an explicit character, an extra bit is output to tell the difference between them. [Storer 1988]

LZW

LZW was developed by T. Welch in 1984 and is a popular variant of LZ78. The main difference of LZW compared to LZ78 is that it eliminates the second field in the output token, i.e. the character. To be able to just output pointers to the dictionary, LZW initialize the dictionary with all symbols. Thus, for a symbol alphabet with 256 symbols, the first 256 entries in the dictionary are filled. [Salomon 1997]

The LZW algorithm can be described as follows. First, the dictionary is initialized with all symbols. The routine starts with an empty match string I. The encoding routine then reads in a character x from the input data. After each read character, the dictionary is searched to see if any entry matches I. If a match is found, the character x is concatenated to I and the next character is read from the input. If no match is found, the encoder writes the index pointer to the entry that matched I. Then the string Ix (x concatenated to I) is added to the next available entry in the dictionary.

Match string I is now initialized to character x and the algorithm restarts from the beginning. [Salomon 1997]

5.3.4 Evaluation

Compression ratio

Algorithms bases on LZ-techniques should give good compression ratios only if the input data contains strings that occur frequently. If, for example, an input block contains two characters strings, which are not dependent of each other, LZ compression routine would not give such a good compression ratio. Therefore, this type of compression would be suitable for files containing text, html etc., which consist of many identical strings.

Since variants of the LZ78 technique handle fixed size strings, compression of long strings of repeated characters are not efficiently encoded. However LZ77 variants get this feature for free, since a match string can be located in the lookahead buffer.

One of the main differences between variants of LZ algorithms is how they handle the codes generated when finding match strings. If the code for a pointer is coded with fixed size, the number of bits used determines the length of the sliding window in LZ77 techniques. For a LZ78 method, the code size determines how many entries that a dictionary contains. But is it so simple that a larger window or dictionary automatically gives better compression ratios? Unfortunately not, since this gives larger code sizes to output. There are many variants of implementations using different window sizes and dictionary entries. One variant, described by Bell, Cleary & Witten [1990], called LZB, codes the size of the pointer and the match length differently, depending on how often the values occur. Other techniques, sometimes referred to LZH, use Huffman coding techniques to code the pointers.

The algorithms described above use so called *greedy parsing*, which means that the algorithm compares strings from the beginning to the end, and codes a string if a long match is found [Bell, Cleary & Witten 1990]. It does not know if it would have been better to skip one character to find a longer match. The best solution would have been to use *optimal parsing*, which finds the best places to parse the strings. This is unfortunately a quite process-demanding task, since the whole input data must be parsed and evaluated. The approach that lies in between these methods is called *lazy coding* [*FLZ Data Compression*]. If a match was found too short, according to some criteria, the program checks if it would achieve better compression ratio if it skips the first character in the lookahead buffer. According to Gailly [1996], this method is used in many standard compressing utilities, such as GZIP.

Other factors that affect the compression ratio for LZ-techniques, are the *locality*, i.e. how close characters and matching strings occur to each other. If the structure of the input data changes often, it is important that the model quickly adapts to these new environments. LZ77 techniques automatically uses locality, since the sliding window just remember a certain amount of data. How quickly it changes behavior can depend on the window size or priority of encoding pointers. In the case of LZ78 methods, it is a bit different. Here it depends on how many entries the dictionary can contain and when the strings are removed from the dictionary. According to Nelson [1989] there are two main principals for handling entities in the dictionary.

1. The compression ratio is measured continuously. If the dictionary is full, and the compression ratio falls to a certain level, the whole dictionary is removed, and rebuild from scratch.

2. Keep track of how frequently different strings are used and remove strings that are rarely used.

The last alternative would probably adapt better to changes in input data, since each string is validated and removed after a certain time. But this process would require more memory and probably more processing time.

Performance

For LZ77 techniques, the compression part requires much more processing capacity than the decompression part, due to the searching of matching strings. The searching can be implemented with linear search, but this would result in poor performance. Therefore, a better alternative would be to use hash tables or binary search trees to find the strings.

According to Salmon [1997], LZ78 algorithms can be efficiently implemented with so called *trie* structures. This is a multiway tree where each path is an inserted string. This allows according to Bell, Cleary & Witten [1990] that strings and substrings can rapidly be located. Since an LZ78 implementation must build the similar dictionary on both the compression and decompression part, this modeling technique does not have such a large difference in processing requirements between compression and decompression.

Memory requirements

The memory requirements for LZ77 based models are highly dependent of the size of the sliding window. This effects other parts of the implementation, such as hashtable size or size of binary search tree. On the decompression side, implementations can be designed to require extremely small amount of memory. This since the only direct amount of memory that must be allocated is the sliding window. This counts only if simple fixed codes are used. If other coding techniques are used, such as Huffman, these implementations require more memory.

The amount of memory required for LZ78 implementations is dependent on the size of the dictionary. If a trie structure is used, the memory required for entity in the dictionary is just the size require for the memory in each node. The amount of memory needed for one node depends on the implementation, for example the size of each pointer used in the trie. Since both the compression and decompression side must build the same dictionary, the amount of memory should be almost identical for both sides.

Streaming possibilities

Since the dictionary methods LZ77 and LZ78 and all their variants are adaptive, these techniques are suitable for streaming. One thing that could limit the size of the block is the lookahead buffer used. Therefore, the block size should not be less than the size of the lookahead buffer.

Chapter 6

Transformation techniques

This is the third and last part describing compression techniques. This chapter covers the principals of transformation techniques and a discussion about the properties of the techniques. Techniques handled in this chapter are Move-to-front, Burrows-Wheeler transform and Differential coding transform.

6.1 Move-to-front

Move-to-front, or MTF, could either be called a coding method or a transformation method, since the output data has a different structure but the same length as the input. The method itself does not therefore achieve compression, but can under special circumstances facilitate other methods to render better compression ratio. [Campos 1999]

The basic idea of this method is to maintain all symbols in a list where the frequently occurring symbols are placed in the beginning of the list. This method is *locally adaptive*, i.e. the method adapts itself to frequent symbols near the input stream. [Salomon 1997]

6.1.1 Encoding

From the description given by Salomon [1997], the encoding method can be explained as follows:

1. Assign a list L containing all symbols.

2. Read in the first input character from the input stream.

3. Find the symbol of the character in the list L and print the index number to the matching position in the list.
4. Move the symbol to the front in the list L
5. Go to step 2 until the input stream is empty.

For an alphabet with 4 different symbols there is an input stream

$$X = \{0, 1, 0, 2, 0, 0, 2, 2, 1, 1, 3, 3, 1, 2, 3, 2, 2, 3, 1, 1\} \tag{6.1}$$

The input symbols have probability $\{0.2, 0.3, 0.3, 0.2\}$, which gives an entropy of 1.985. Table 6.1 shows the result after applying the MTF algorithm on the input stream.

Input symbol	**Output code**	**Symbol list**
		0,1,2,3
0	0	0,1,2,3
0	0	0,1,2,3
1	1	1,0,2,3
1	0	1,0,2,3
1	0	1,0,2,3
0	1	0,1,2,3
0	0	0,1,2,3
1	1	1,0,2,3
0	1	0,1,2,3
2	2	2,0,1,3
2	0	2,0,1,3
3	3	3,2,0,1
3	0	3,2,0,1
3	0	3,2,0,1
2	1	2,3,0,1
2	0	2,3,0,1
2	0	2,3,0,1
3	1	3,2,0,1
3	0	3,2,0,1
2	1	2,3,0,1

Table 6.1: Result after MTF

The entropy for the output stream is 1.462 bits per character, which is 0.52 bits per character better then without using MTF. It should be pointed out that it is not always easier to achieve better compression ratio using MTF. If the input data has not got concentrations of identical symbols, a transformation with MTF can give worse result.

6.1.2 Decoding

The decoding algorithm is obvious to see, when understanding the encoding algorithm. Encoding can be described as

1. Assign a list L containing all symbols.
2. Read in the first input code from the input stream.
3. Read the symbol from the list by using the code as index.
4. Output a character of the read symbol.
5. Move the symbol to the front in the list L
6. Go to step 2 until the input stream is empty.

6.1.3 Evaluation

Compression ratio

The compression ratio that can be achieved with MTF depends on which methods that are performed before and after MTF. It should be kept in mind that MTF achieves better results only when the data has locality, i.e. symbols occur close to each other. The next transformation technique described, Burrows-Wheeler transform, has been shown to achieve good results when combined with MTF.

Performance

MTF could be implemented as a linked list, where the latest symbol is moved to the front. This operation could be done fast, but looking up the symbols in the linked list would then demand a linear search. If the number of symbols were few, as often is the case when handling bytes, this search would not take a significant amount of time.

Memory requirements

The memory used for this transformation is just the amount of memory needed to allocate the list with symbols. If the number of symbols is fewer, the memory requirement decreases.

Streaming possibilities

Since each character is transformed one by one in a flow, streaming is easy to achieve.

6.2 Burrows-Wheeler transform

Most compression methods work in *streaming mode*, i.e. the compressor does not need to see in to the future, to be able to compress. This fairly new method, which was presented by Michael Burrows and David Wheeler in 1994, works in *block mode*. This method, which is called Burrows-Wheeler Transform (BWT), does not actually compress the input block. Instead it transforms the data, so that it more easily can be compressed later. [Nelson 1996]

The main feature of this transform is that it is reversible, i.e. the original data string can be reconstructed.

6.2.1 Forward transformation

According to Burrows & Wheeler [1994], the algorithm takes an input message $X = \{x_1, x_2, \ldots, x_n\}$ with n characters consisting of the alphabet $S = \{s_1, s_2, \ldots, s_m\}$ with m different symbols.

To illustrate the algorithm, an alphabet

$$S = \{a, b, d\} \tag{6.2}$$

is used and an example message

$$X = \{d, a, b, d, a\} \tag{6.3}$$

The first thing done is constructing a $n * n$ matrix M whose elements are characters. Each row is a rotation of message X, cyclically shifted and sorted in lexicographical order. Thus, at least one of the rows in matrix M contains the original message X. Let I be the index number of the first row that is identical to the original message. [Burrows & Wheeler 1994]

The example of matrix M is shown in table 6.2

	0	**1**	**2**	**3**	**4**
0	a	b	d	a	d
1	a	d	a	b	d
2	b	d	a	d	a
3	d	a	b	d	a
4	d	a	d	a	b

Table 6.2: Matrix in BWT forward transformation

It is important to point out that this matrix is not actually created when implementing the algorithm. Instead a list of index pointers is sorted, which have pointers to the original message block. [Nelson 1996]

In the example, the original message block is located at row 4, therefore $I = 3$. The last column in matrix M is called L and the first column F. Characters in column L do not at the first glance appear to be in any particular order, but the column consists in fact of the *prefix character* to the strings in the same row. [Burrows & Wheeler 1994]

The output from the BWT is the characters in column L and the primary index, I. For the example above the output is

$$Y = \{d, d, a, a, b\}\,,\ I = 3 \tag{6.4}$$

6.2.2 Reverse transformation

At the first glance, it looks almost impossible to retransform the data, to get the original message. The key to do this is to recreate the matrix M. Data needed are column L, column F and the index I. Since column L consists of exactly the same characters as column F, but in a dissimilar order, the only thing that must be done to get F, is to sort L. [Burrows & Wheeler 1994]

Table 6.3 shows the reconstruction of matrix M so far.

Each character in column F corresponds unambiguously to a character in column L. But what happens if there is more than one character with the same symbol in column L? It is proven that the first appearance of a symbol in F corresponds to the first appearance of the same symbol in L. The second

	0	**1**	**2**	**3**	**4**
0	a	?	?	?	d
1	a	?	?	?	d
2	b	?	?	?	a
3	d	?	?	?	a
4	d	?	?	?	b

Table 6.3: First step of reconstructing matrix M

appearance of the symbol in F corresponds to the second appearance in L, and so forth. The proof of the statement above can be found in the report of Burrows & Wheeler [1994].

Thus, it is possible to create a transformation vector, T, which maps each character in L to its corresponding character in F. If $L[j]$ is the kth instance of a specific symbol, then $T[j] = i$ where $F[i]$ is the kth instance of the same symbol in F. [Burrows & Wheeler 1994]

Therefore the transformation vector T for the example is equal to

$$T = \{3, 4, 0, 1, 2\} \tag{6.5}$$

For this transformation vector, it is easy to generate the original message. Since the matrix M consists of rotated strings, the character $L[i]$ is the preceding character to $F[i]$. The construction of vector T gives that

$$F[T[j]] = L[j] \tag{6.6}$$

which directly indicates that $L[T[j]]$ cyclically precedes $L[j]$. The index I is defined in the forward transformation, as the row in matrix M that consists of the original string. Therefore, $L[I]$ is the last character in the original message. [Burrows & Wheeler 1994]

Following Burrows & Wheeler's [1994] description, the following algorithm shows how to reconstruct the original message string:

1. Set i equal to I.
2. Output $L[i]$.
3. i := $T[i]$
4. If not the last character in block, go to step 2.

5. Reverse the output string.

For the example, the output message Y is

$$Y = \{d, a, b, d, a\} \tag{6.7}$$

6.2.3 Benefits with the transform

What does this transformation achieve anyway? The symbols probability is the same, why is this new data easier to compress?

To illustrate this, a message containing English text is explored. Let say that this text contains a lot of words *the*. Then when the list of rotated inputs is sorted, many strings starting with *he* will be grouped together. Then the column L will probably have many symbols t in a row. The same thing will happen for other words that occur frequently in the input, thus will the output column L consist of characters occurring with great locality. This output can then be further transformed with for example MTF, which is proposed by Burrow & Wheeler. After this step any probability encoder can be used, for example Huffman coding or arithmetic coding. This approach has been shown to give great compression ratio with a relatively high compression bandwidth.

It should also be pointed out that there has been new research done to further improve the Burrow-Wheeler method. The main focus is done to improve the backend methods, i.e. having other methods than MTF, Huffman and arithmetic coding after the transformation. [Fenwick 1996]

6.2.4 Evaluation

Compression ratio

This fairly new method achieves, according to the inventors Burrows & Wheeler [1994] compression ratios comparable with the best known statistical modeling techniques. The compression ratios become better, the larger the block size is. This since the output from the transform probably gives more identical symbols in a row.

Performance

According to Burrows & Wheeler [1994], the most critical part of BWT is the sorting algorithm. The area of sorting algorithms is large, an extensive research are made. It is not possible to go into details about sorting algorithms, but there are methods such as quick-sort that can sort in $O(N\,log\,N)$ time [Weiss 1996]. Burrows & Wheeler [1994] have developed a more complex sorting algorithm using quick-sort which avoids the worst case scenario, where a string consists of repeating characters. This can be very time consuming to sort, since each string comparison in the sort algorithm will go through many characters.

Memory requirements

The memory requirements for BWT depend mainly on the block size and the implementation of the sorting algorithm. Burrows & Wheelers sorting algorithm requires 6 bytes per character in the block. Thus, the memory requirements for their implementation require at least 7 times more memory than the size of the block.

Streaming possibilities

BWT is a block-sorting algorithm, which achieves best results if the size of the block is huge. Table 6.4 shows the compression ratios achieved by Burrows & Wheelers test when using different block sizes. The test was performed on the file *book1* from the Calgary Compression Corpus.
[Burrows & Wheeler 1994]

Block size	**Compression ratio**
1 kByte	54%
4 kByte	48%
16 kByte	43%
64 kByte	38%
256 kByte	34%
750 kByte	31%

Table 6.4: Compression ratios for different block sizes

The table shows that the compression ratio dramatically becomes better the larger the block size is. Thus, Burrow-Wheeler performs great compression ratios for large blocks, for smaller blocks the compression ratio is not that impressive. By this reason, this method is not that suitable for streaming purposes.

6.3 Differential coding transform

Some input data source can consist of symbols with almost the same probability, but with small difference between the character values. A typical example is a sound sample, where the samples follow a curve. Instead of trying to compress the data directly, a differential coding can be applied. The idea is simply to output the difference between two characters, instead of the absolute value. Thus, the output values become smaller. The method itself does not perform any compression, but makes for some input data, later compression more efficient. [Bell, Cleary & Witten 1990]

6.3.1 Evaluation

Compression ratio

This transform does not achieve any compression itself, but it can for some data input really facilitate compression.

Performance

This transform is extremely fast, since the only thing that has to be done is to calculate the difference between two values.

Memory requirements

The transform does not require any particular memory.

Streaming possibilities

Since the characters are read in one by one, streaming is easy to achieve.

Chapter 7

Compression algorithms

This chapter describes how the different techniques in chapter 4, 5 and 6 can be combined and categorized to form compression algorithms. Combinations stated in this chapter form the basics to implementations tested in the compression experiment.

7.1 Combining transformation, modeling and coding

In previous chapters different techniques for transformation, modeling and coding were discussed. It is obvious that each of these techniques does not achieve any compression, but if they are combined together, suitable compression ratios can be achieved.

The number of combinations is huge, both since there exist several approaches for each technique part, and that each technique can be implemented and configured to behave in lots of different ways.

The fundamental and most important part is, according to Nelson & Gailly [1996], the model of the algorithm. Since this part decides either the probabilities or the string matching ability, this has a large impact on the possible compression ratios achieved. Therefore, the modeling technique forms the foundation of the categorisation of algorithms in this study.

In this study, compression algorithms are divided into three main categories:

- Statistical algorithms
- Dictionary-based algorithms

- Block algorithms

Statistical algorithms are algorithms that use statistical modeling to estimate probabilities and then use a coding technique to compress the data.

Dictionary-based algorithms use variants of dictionary-based modeling techniques to find strings in the already seen data, and then use a coding technique to encode the output from the dictionary-based model.

Block algorithms do not compress the data from the beginning to the end, as the above described algorithms do. Instead, they manipulate the whole block of data, and then writes the whole compressed block to the output.

In the following sections, these three categories of compression algorithms will be discussed. Further categorization will be made, and a name convention for combinations will be defined. To be able to get an overview of the whole field of lossless data compression, the categories do not show individual variants of different techniques.

7.2 Statistical algorithms

Statistical methods performance of speed and compression ratio depend highly on the model used. There are many ways of modeling the probabilities for symbols. The two main properties of a statistical model discovered in this study is the order of the model and whether it is static, semi-adaptive or adaptive.

In all resources found about statistical algorithms, the model follows directly by an encoder. The four principals of coding techniques discussed in this study are Shannon-Fano, Huffman, Arithmetic and Range coding. The implementations of these coding techniques often differ a bit depending on the type of model used. For example, static Huffman coding and adaptive Huffman coding use different algorithms to calculate the codes.

Besides the above described coding techniques, fixed size coding can be used. This means that the codes always have the same length. Since the goal with statistical algorithms is to give different code lengths for different symbols, this is obviously not a suitable solution for this sort of algorithm.

To easily be able to describe the different types of statistical algorithms, a notation is defined. The fist word describes that it is a statistical algorithm, and is written with the keyword *Statis*. The second word indicates the model order and the third word tells whether it is a static, a semi-adaptive or an adaptive model. The last mentioned word is written with the keywords *Static*, *Semi* or *Adap*. The fourth and last word explains which sort of encoding technique that is used. The keywords are *Shan* for Shannon-Fano coding, *Huff* for Huffman coding, *Arith* for arithmetic coding and *Range* for range coding.

Some possible combinations that form statistical algorithms are shown in table 7.1.

Algorithm categories
Statis-0-Static-Huff
Statis-0-Static-Arith
Statis-0-Semi-Shan
Statis-0-Semi-Huff
Statis-0-Semi-Arith
Statis-0-Semi-Range
Statis-0-Adap-Arith
Statis-0-Adap-Range
Statis-1-Adap-Arith
Statis-1-Adap-Range
Statis-4-Adap-Arith
Statis-4-Adap-Range
Statis-8-Adap-Arith
Statis-8-Adap-Range

Table 7.1: Example of statistical algorithms

7.3 Dictionary-based algorithms

Dictionary-based algorithms use an dictionary-based model to compress the data. Since dictionary-based models actually generate codes, these codes can be directly written to the output. Here, this is called that the output uses fixed codes, i.e. the codes have fixed sizes, even if their probabilities differ.

There are extremely many variants of dictionary-based models, where each technique can be configured in different ways. For example, techniques based on LZ77 can have different window sizes, methods for finding matching strings etc. Both the performance and compression ratio of these techniques are highly dependent on the implementation.

To be able to get an overview of this area, it is chosen to divide dictionary-based algorithms into two fields: Techniques derived from LZ77 and techniques derived from LZ78. Later in the experiment, no difference is made concerning the modeling variants of these two types.

Since both these areas are adaptive dictionary techniques, no tests are made of static or semi-adaptive dictionary models. Since the data used in a WVPN can vary a lot, these methods are assumed not to be so suitable. Furthermore, Static dictionary-based models are rarely described in literature as suitable techniques for lossless data compression.

To describe different categories of dictionary-based compression algorithms, the following notation is used. The first keyword that describes that it is a dictionary-based algorithms is *Dic*. The second keyword explains if the model is based on LZ77 or LZ78 and is therefore written as *LZ77* or *LZ78*. The third keyword explains whether the codes generated by the model is coded with adaptive methods, semi-adaptive or static, which is written *Adap*, *Semi* or *Static*. If fixed coding is used, i.e. the codes generated by the model have always the same size, the keyword *Fixed* is used instead. The fourth and last keyword states the coding technique used to encode the codes generated by the model. The possible alternatives are *Shan* for Shannon-Fano coding, *Huff* for Huffman coding, *Arith* for arithmetic coding and *Range* for range coding.

To show how these category notations can look like, example are shown in table 7.2.

Algorithm categories
Dic-Lz77-Fixed
Dic-Lz77-Semi-Huff
Dic-Lz77-Adap-Arith
Dic-Lz78-Fixed
Dic-Lz78-Semi-Huff

Table 7.2: Example of dictionary-based algorithms

7.4 Block algorithms

The last main category investigated in this study is called block algorithms. The only suitable technique of block algorithm known by the author is the Burrows-Wheeler transform invented by Burrows & Wheeler [1994]. The large difference of this technique compared to statistical algorithms and dictionary-based algorithms is that it does not compress the data for the beginning to the end. Instead it transforms the whole data block, compresses it and then writes it to the output.

The first keyword to explain that it is a block algorithm is written *Block*. The second keyword explains which sort of main transform that is used. Since the only known in this study is Burrows-Wheeler transform, it is simply written *BWT*. Both Fenwick [1996] and Burrows & Wheeler [1994] propose that the BWT should be followed by MTF, since the output from BWT gives strong locality. By this reason, MTF is not written as an own keyword. An aspect that affects both the compression ratio and the compression speed is the block size used by the transform. Since this parameter is important for the result, it is part of this categorisation. Therefore, the third keyword indicates the block size in Kbytes. The fourth and last parameter shows which coding technique that is used to compress the MTF codes. The keyword is written *Arith* or *Huff*.

Examples of categories of block algorithms are shown in table 7.3.

Algorithm categories
Block-BWT-100-Huff
Block-BWT-500-Arith
Block-BWT-100-Huff
Block-BWT-500-Arith

Table 7.3: Example of Block algorithms

Chapter 8

Experiment and result

In this chapter the properties for the compression experiment are stated and the numerical result of it presented.

8.1 Introduction

To be able to evaluate different compression methods an experiment of compressing was performed. To see how the different compression implementations behaved on different document types, each compression implementation was performed on all different data document types. The actual testing was done by a script program, which took over 30 hours to execute. The result from the experiment should give compression ratio, compress bandwidth and decompress bandwidth for all implementations on the different input data.

8.2 Test data

The test was performed on totally 150 MB of data, dividing into 15 different types. Files for each document type was added together into one single file, in which the compression test was made. Information about the test data is given in table 8.1.

Filetype	**Number of files**	**Size in Mbyte**
C++ source code	435	03.30
Compressed files, Winzip	3	03.22
Executables for Intel x86	86	28.20
Gif	87	02.32
Html	192	05.54
Jpg	80	02.08
Mp3	9	29.30
MS access	2	01.19
MS Powerpoint	9	12.60
MS Word document	136	05.22
Pdf	33	20.50
Postscript	150	07.42
Text	510	23.10
Wav (sound)	136	03.90
MS Excel document	35	02.41

Table 8.1: Data in experiment

8.3 Devices processing capacity

The compression test was performed on a Dell Precision Workstation 220 with a Pentium III 866 Mhz and 192 MB memory. Therefore, the compression and decompression bandwidth of the result is calculated for this reference computer, when all processing power was used for the testing.

To be able to estimate overall performance in the WVPN, the compression and decompression bandwidth must be estimated for different devices. Since many of the tested implementations require much memory together with the memory the test data requires, it is not possible to perform this experiment on all devices.

To be able to estimate the compression and decompression rate for the devices anyway, a performance test program was designed. This program contains a compression algorithm, which run several times on each device. The execution time was measured, which gave after calculations a scale factor of performance, compared to the reference computer. The measured scale factors are given in table 8.2. The scale factor shows how many times longer it takes to execute the test program compared to the reference computer.

Device	**Scale factor**
Toshiba Satellite Pro 4300	1.5
Compaq iPAQ 3630	11
Cassiopeia E-115	24
Compaq Aero 2130	45
Palm Vx	1000

Table 8.2: Performance scale factors for different devices

8.4 Implementations in the experiment

In the following section, the algorithm implementations included in the experiment are described briefly.

AHUFFDB1

This algorithm was implemented by the author of this study. It is a version of adaptive Huffman, designed to take variable block size as input. The adaptively collected probabilities of the symbols are stored in an object, which makes it possible for streaming purposes. This means that when each block is sent to the compressing unit, the statistics about symbols probabilities are kept from earlier compressed blocks.

The model used is an adaptive order-0 model, which counts the frequencies for each symbol. Each input character is handled as a byte, thus the model keeps track of 256 different symbols.

The Huffman tree is implemented as a statically allocated table, which uses indexes as pointers. Thus, the memory usage in this implementation does not change over time. The total amount of memory this implementation requires, excluded memory needed for storing data, is 9Kbyte.

ARJ

The compressing program ARJ was implemented by Robert Jung. According to Jung [2001], the ARJ program uses a LZSS implementation where semi-adaptive Huffman coding was used to encode the explicit characters, length of matches and pointers to matches. The algorithm has a 26624 bytes sliding window using hash linked lists to access matching sub strings. The Huffman tables are stored in the beginning of each block, which is 16 Kbytes large. Jung [2001] did not say exactly how much memory ARJ requires, but Jiang [1996] claims that ARJ requires about 290 Kbytes memory.

BZIP

The BZIP compressing program was implementation by Julian Seward. This implantation uses a Burrows-Wheeler transform first on the input data, which followed by a semi-adaptive Huffman implementation.
[*Bzip2 and libbzip2 - Introduction*]

The implementation can be configured to use different block sizes when compressing, which will produce different result in compression ratio and compression performance. [Seward 2000]

In this test, three different block sizes are tested. *BZIP1* uses 100 Kbytes blocks, *BZIP5* uses 500 Kbytes blocks and *BZIP9* uses 900 Kbytes blocks. According to Seward [2000], these three implementations would require about 1200 Kbytes, 4400 Kbytes and 7600 Kbytes memory during compression. For decompression, it would require 500 Kbytes, 2100 Kbytes, 3700 Kbytes.

GZIP

The GZIP program was developed by Jean-loup Gailly. This implementation uses a variant of LZ77 as model. The search window is 32 Kbytes and the longest possible match is limited to 258 bytes. The search of duplicated strings is done via a hash table and a hash chain. If the algorithm does not find any matching string, a single character is written to the output. [Gailly 1996]

For coding, semi-adaptive Huffman coding is used. Characters not found in the search buffer and the match lengths are coded with one Huffman tree, and the match distance is coded in another. The tree information is stored in the beginning of each block of codes, which can have variable length.
[Gailly 1996]

In this test, three different settings are used to configure the implementation: *GZIP1*, *GZIP5* and *GZIP9*. The difference between them is how many matching tests that are performed in the hash chain. GZIP1 uses least matching tests and GZIP9 the most. GZIP5 and GZIP9 also use lazy match evaluation, which GZIP1 does not do. [Gailly 1996]

HA

The HA implementation is written by Harri Hirvola. After analyzing the source code of this implementation, the following information was found.

The first algorithm in HA (here called HA1) uses a LZ77 variant with a window size of 16384 bytes. The searching is done with hash tables. Coding of window position, match length and explicit characters are done with adaptive arithmetic coding. Memory requirements are hard to estimate for this algorithm, but it should not be more than about 200 kByte.
[*Source code of HA*]

The second algorithm in HA (here called HA2) uses an order-4 context modeling. [*comp.compression Frequently Asked Questions (Part 1/3)*].
The data is then encoded with an adaptive arithmetic encoding implementation. [*Source code of HA*]

JAM

This implementation was written by W. Jiang. The basic algorithm is a variant of LZ78. The implementation uses a dictionary of 8192 entries and requires less than 50 Kbyte of memory. [Jiang 1996]

LHA

LHA was written by Haruyasu Yoshizaki. Accoring to Jung [2001], LHA uses the same algorithms as ARJ, but with a different implementation. Thus, it uses a LZSS variant followed of semi-adaptive Huffman encoding.

LZDB3

The LZDB3 algorithm was implemented by the author of this study. This is a LZSS variant that uses a 4096 Kbytes sliding window. To find strings in the given window, a hash table is used. This hash table points to a circular buffer used as a linked list. By using a fixed buffer as a linked list, memory does not have to be allocated during compression. This also gives the result that old pointers to data do not have to be removed, they are automatically written over when the circular buffer is full.

Pointers to the window are coded into 12 bits and the match length into 4 bits. The maximal match length for this implementation is 18 characters. Since both the match length and the window pointer consist of 16 bits together, this pair is coded into two bytes. If no match is found, the next coming character is directly sent to the output. To separate characters from coding pairs, a byte is sent to the output every 8 time a character or a coding pair is sent to the output. This byte consists of 8 bits, indicating if the next 8 codes are characters or coding pairs.

LZOP

This implementation was written by Marcus Oberhumer. He claims that it uses an algorithm called LZO, which according to the documentation seems to be a variant of LZSS with fixed size codes. According to Oberhumer [1999], this implementation is designed to be fast for decompression and uses almost no memory for decompression.

LZW

A simple LZW implementation written by David Bourgin. The maximal number of strings in the dictionary is limited to 4096. [Bourgin 1995]

MDCD

MDCD is a simple LZW implementation written by Mike Devenport. The implementation uses 13-bits codes to encode each pointer to the dictionary. According to the Devenport [1988], the implementation requires about 88k of memory.

PPMD

This compressing program was implemented by Dmitry Shkarin. It uses a finite-context model, which can be set to different orders. As entropy coder the implementation uses a range coder. [Shkarin 2001]

According to Shkarin [2001], this implementation consumes a huge amount of memory, but achieves good compression ratios.

In this experiment, the implementation of PPMD is tested with four different configurations.

PPM1_4 Order-4 model. Uses 1 Mbyte memory.

PPM10_4 Order-4 model. Uses 10 Mbyte memory.

PPM10_8 Order-8 model. Uses 10 Mbyte memory.

PPM2_4 Order-4 model. Uses 2 Mbyte memory.

RAR

According to Roshal [2001], the RAR implementation uses LZSS with a 64 Kbyte sliding window. The characters, match lengths and string positions are coded with semi-adaptive Huffman. The minimum match length is 2 characters.

RK

This implemention is written by Malcom Taylor and is according to the archiving test made by Jeff Gilchrist the compressor that gives best compression ratio for most file types. [*Archive Comparison Test*]

Information about its implementation is quite poor, but it seems to use a PPMZ model followed by an arithmetic coder. [*RK Software*]

The purpose to include this implementation in this compression test is to show how good compression ratios that can be achieved. In the test, two different configurations of the program are used. *RKMX1* and *RKMX2*, where the latest should achieve the best compression ratio.

SARITHDB1

The implementation SARITHDB1 stands for semi-adaptive arithmetic coding implemented by the author. This implementation uses an order-0 model to gather the probabilities for symbols. The algorithm makes two passes over the input data, first to collect the probabilities and secondly to encode the data. The probabilities are stored in a header before the actual coded data.

ZZIP

Zzip is implemented by Damien Debin, and is a compression program which is build on the Burrows-Wheeler transformation. The routine uses RLE, BWT and MTF followed by arithmetic coding. [*Zzip's webpage*]

The implementation is tested in three different modes

ZZIP1 Uses 100 Kbytes block.

ZZIP5 Uses 600 Kbytes block.

ZZIP9 Uses 900 Kbytes block.

Summary of implementations

A summary of all implementations used in this compression experiment is listed in Appendix B. All the implementations are categorized according to chapter seven.

8.5 Results

The results from the test program show compression ratio, compression bandwidth and decompression bandwidth for all the implementations and document types. The numeric results are listed in Appendix A. The compression ratio r is defined as shown in 2.2.2 on page 14. Both compression bandwidth B_c and decompression bandwidth B_d show the compression ratio on the reference computer. The values for B_c and B_d are given in the unit Kbytes/s.

These numeric results together with the algorithm discussion in earlier chapters will be analysed in the following chapter.

Chapter 9

Analysis

In this chapter a discussion is made to sum up the conclusions about different compressing techniques discussed in chapter 4, 5 and 6. Furthermore, the results achieved from the compression experiment described in chapter 8 are analyzed and discussed. The results from the discussions of this chapter are stated in chapter 10; Conclusions.

9.1 Parameters affecting performance

In section 2.2.2, a theoretical discussion concerning the relationships between parameters affecting compression performance was made. The conclusion was that data compression was beneficial when

$$\frac{r * n}{B_n} + \frac{1}{B_c} + \frac{1}{B_d} < \frac{n}{B_n} \tag{9.1}$$

This theoretical expression forms the foundation of this analyze chapter. There are many parameters involved, which all are hard to estimate. Because of this, assumptions must be made, to be able to analyze which methods that are suitable in different situations.

In expression (9.1), there are seven parameters affecting the need of data compression. These parameters are dependent on several other circumstances that can change from time to time.

This section is divided into six subsections, describing the following parameters:

- B_n, Network bandwidth

- r, Compression ratio
- B_c & B_d, Compression and decompression bandwidths
- n, Number of blocks
- Memory

The last section, *memory*, is not actually a part of expression (9.1). Nevertheless, memory requirements are highly important when choosing a suitable compression algorithm.

9.1.1 Network bandwidth

The network carriers discussed in this chapter are GSM, GPRS, CDPD, HSCSD, Bluetooth and WLAN. In chapter 2.7.4, an overview of the different networks technologies was given. Another factor that could be seen as part of the network bandwidth is the encryption and decryption bandwidths. Since the actual network bandwidth in a WVPN is the amount of data that could be sent per time unit, the bottleneck of the system decides this bandwidth. If the encryption bandwidth is lower than the network bandwidth, this throughput is the bottleneck.

To be able to qualitatively evaluate if data compression is beneficial, certain values for network bandwidth must be used. In the overview of different network technologies, the theoretical maximal bandwidth was given. These values could of course be used when analyzing the compression results, but Hovmark [2001] claims that the real bandwidth for some network carriers is generally much lower.

GSM

According to Englund [2001], GSM is a network carrier that has fairly stable bandwidth when the receiving device changes location. Wierlemann & Kassing [1998] also noticed that the bandwidth for GSM did not decrease significantly much when the signal coverage decreased. Therefore, the bandwidth can be estimated to be around 9.6 Kbit/s, which is equal to 1.2 Kbyte/s.

GPRS

GPRS is designed to be able to retrieve data from 8 different time slots where each slot has a capability of 14.4 Kbit/s.
[*GPRS - Data transmission for mobile telephony*]

This gives a theoretic maximal bandwidth of 115 Kbit/s. This sounds like an impressive bandwidth, but according to Hovmark [2001], this is not a trustworthy value. Currently, GPRS networks do not prioritize data transmission. The priority is given to voice communications, which results in that less data can be transferred per second. If more than one user are in the range of a sender station, then they are sharing the same time slots. Hovmark [2001] therefore claims that not more than one or two time slots are actually given to a user. According to Englund's [2001] experience from testing GPRS networks by British telecom and Sonera, the real bandwidth was not much better than for normal GSM. Hovmark claims further that the problem with GPRS is that it does not give a continuous flow of data. This means that the bandwidth for a period of time may appear to be very good, but then for a period be close to zero. Thus, the worst-case scenario is zero bandwidth, and the best case reaches the theoretical limit. It is therefore hard to estimate a trustworthy value for the bandwidth of GPRS, since so many parameters are involved. Testing done by Schäfer [2000] shows that the bandwidth just reached 28.8 Kbit/s sometimes, and that it quite often becomes as low as 9.6 Kbit/s. Even though the tests above show that the bandwidth does not achieve much more than the speed of GSM, better developed networks can probability give better results in the future. The bandwidth for GPRS is therefore estimated to be somewhere between 9.6 Kbit/s and 28.8 Kbit/s, which is the same as 1.2 Kbyte/s and 3.6 Kbyte/s.

CDPD

CDPD can theoretically handle 19.2 Kbit/s, but according to Englund [2000]this performance cannot be achieved under normal circumstances. He performed tests in California this year, where the bandwidth did not become better than 4-5 Kbit/s. Since the bandwidth of CDPD is quite uncertain, it is estimated to be approximately equal to GSM.

HSCSD

High-Speed Circuit-Switched data, *HSCSD*, for mobile data communication, is built on the GSM system [*HSCSD - a whatis definition*]. The theoretical bandwidth of HSCSD is 38.4 Kbit/s, which according to tests done by Schäfer [2000] also is the practical bandwidth. The bandwidth for HSCSD is therefore estimated to be around 4.8 Kbyte/s or below.

Bluetooth

Bluetooth has a theoretical bandwidth of 1 Mbit/s according to Motorola's FAQ [*Motorola Bluetooth [FAQ]*]. The bandwidth for a user is however highly dependent on how many users are accessing the same access point. Furthermore, the distance between the user device and the access point has a large impact on the bandwidth [Hovmark 2001]. Therefore, the actual bandwidth must be estimated by experience from testing. According to Englund's [2000] testing results an estimation of 150 Kbit/s would be realistic. The estimation of Bluetooth bandwidth is therefore set to around 19 Kbyte/s.

WLAN - 802.11

A WLAN implemented using the 802.11 standard apply to wireless Ethernet LANs [*802.11a - a whatis definition*]. Since all users connected to a WLAN hotspot share the same bandwidth, the actual bandwidth per user can vary. If the wireless hotspot is connected directly to the WVPN server, the user has the bandwidth of the WLAN directly to the server. But, if there is another media between the WLAN and the server, for example an Internet connection, this can be the bottleneck of the whole connection. A 802.11b network can have up to 11 Mbit/s bandwidth, which is the most common version today according to Englund [2001]. Therefore, the network bandwidth can vary from time to time from just some Mbit/s up to 11 Mbit/s. This would be the same as a range from approximately 125 Kbyte/s to 1375 Kbyte/s.

Conclusions about network bandwidth

- The network bandwidth for different network carriers differs quite much depending on different properties. The most stable network carriers are GSM and HSCSD.
- The lowest bandwidth of encryption, decryption or network carrier will be the bottleneck of the system and determine the actual network bandwidth.
- GSM and CDPD are estimated to have a bandwidth of approximately 1.2 Kbyte/s.

- GPRS should be able to have a bandwidth between 1.2 and 3.6 Kbyte/s.
- HSCSD should have a fairly stable bandwidth of 4.8 Kbyte/s.
- The bandwidth of bluetooth can vary a lot, but it should be approximately 19 Kbyte/s.
- WLAN can have very different bandwidth depending on how many users that are using the system simultaneously. Everything between 125 Kbyte/s to 1375 Kbyte/s are possible values.

9.1.2 Compression ratio

The compression ratio achieved by data compression is probably the most important parameter deciding benefits of data compressions in data communication. The compression ratio is of course dependent of the design of the compression algorithm and its implementation. Further, the structure of the input data plays an important role for how good compression ratio can be.

Algorithms and structure of input data

A number of interesting questions arise regarding compression ratios. How well can different types of compression algorithms compress different sorts of input data? Is it always the same algorithm types that achieve the best results? Are there document types that are almost impossible to compress? These questions will be discussed and answered in the following section.

In Appendix A, the results of the compression experiment are listed. Each table represents a document type, which are individually sorted by compression ratio. Since it is difficult to analyze the results in forms of tables, diagrams of the compression ratios are given in Appendix C. Each diagram shows a specific document type and how good it can be compressed by different algorithm implementations. It is important to notice that all compression algorithm implementations are listed in the same order in all diagrams.

The diagrams give an interesting observation; almost all diagrams shows that the rightmost algorithms achieve the best compression result, and the leftmost the poorest. The most significant exceptions is the *Dic-Lz77-Fixed* implementation *LZOP* in the Postscript diagram. The compression ratio is here worse than the other implementations, relatively its result on other document types. The *LZDB3* implementation also achieves a relative poor result on the same document type. This probably depends on the fact that

Postscript files contains text, but with without long matching strings. Since *Dic-Lz77-Fixed* implementations does not have any coding at the end, such as Huffman coding, this file type is difficult to compressed. There are other small exceptions, which can of course depend on the selected documents. If the test data was collected in another way, maybe the result would have been a bit different. The important observation of this analyze is that the algorithms that achieve the best compression ratio tend to do so for all document types. Further, the algorithms that achieve the worst compression ratio do also so for all document types.

When analyzing the diagrams, some more interesting observations can be made. First of all, it seems like the 13 implementations that achieve the best compression ratios are of the category *block algorithms* and *statistical algorithms*. Malcom Taylor's implementation of PPMZ achieves the best compression ratio in all tests. For the other statistical and block implementations, compression performance varies, but the difference between them is no more than five to six percent. A general observation of the block algorithms is that the larger the block size, the better the compression ratio. With the same block sizes, it also seems that the block algorithm using arithmetic coding as end coder achieves slightly better result than the algorithms using Huffman. This sounds reasonable, since arithmetic encoding gives, as discussed in earlier chapters, better coding results. But it is important to point out that this can also depend on how these algorithms are implemented.

The algorithms listed with number 8 to 14 in Appendix C are all lz77-based algorithms. These algorithms achieve a bit worse result compared to statistical and block algorithms, even though it is often not much more than ten percent. For some document types, such as Executable files, the difference is not much more than three to four percent. All algorithms mentioned in this range are of the category *Dic-Lz77-Semi-Huff* except for the *HA1* which is of the type *Dic-Lz77-Adap-Arith*. The difference in compression ratio between them is small and probably mostly depends on the window-size, if they are using lazy-coding and how good they are at locating long strings in the sliding window.

The seven algorithm implementations that perform worst compression ratio have totally different characteristics. The algorithms with order number three to six are of type *Dic-Lz78-Fixed* or *Dic-Lz77-Fixed*. They yield different result on different documents, which means that none of them seems to give generally much better result than the other do. A comment should be made regarding the compression ratios that implementation *LZW* achieves on the file type *compressed*. Here, this implementation results in a compression ratio of 136%, i.e. the output data are larger than the input data.

This occasion is avoided by the other implementations by writing the original data, if no compression could be achieved. This should of course be done in an implementation used in a WVPN.

The two implementations that gives worst result are of the category *Statis-0-Adap-Huff* and *Satis-0-Semi-Arith.* It is not a surprise that these algorithms does not perform as well as the other statistical algorithms. This since these algorithms are implemented just with an order-0 model. This is of course sub optimal for input files where the relation between characters are high.

The characteristics of the diagrams of document types C++ code, Html, Ms access, MS Word, Postscript, Text and MS Excel are quite similar. At least 50 % of the original size is reduced by the majority of the implementations. In the case of Executeable files and Wave music files, compression ratio is not that good, but most algorithms can achieve at least a compression ratio of 80%. It should be pointed out that there are probably other possibilities to produce better compression ratios for sound files, since these files have a pattern of a signal. Applying differential coding before other compression techniques would therefore probably result in better compression. The other file types Compressed files, Gif, Jpg, Mp3, MS Powerpoint and Pdf give all poor compression ratios. This is not surprising, since all these file formats consist of compressed data. The reason that some algorithm implementations can achieve some compression anyway is that the files are compressed with a worse compression method.

Figure 9.1 shows four document types and how good they can be compressed by different algorithm implementation. As can be seen in the figure, Html, Word and Text files are compressed to nearly the same size. On the other hand, Executable files are not reduced to more than about 60 percent of the original size. The difference between the algorithms is greater for the last three document types compared to Executable files.

Block sizes

Another aspect that is interesting to observe is how the block size of the input affects the compression ratio. In the case of statistical algorithms, the block size should not affect the compression ratio at all, presupposed that the model data is saved between the compressed blocks. The reason for this is that characters are encoded one by one without the need of a lookahead buffer.

In the case of dictionary-based coding, a lookahead buffer is needed to search for matching strings. If the block size is too small, the lookahead buffer might

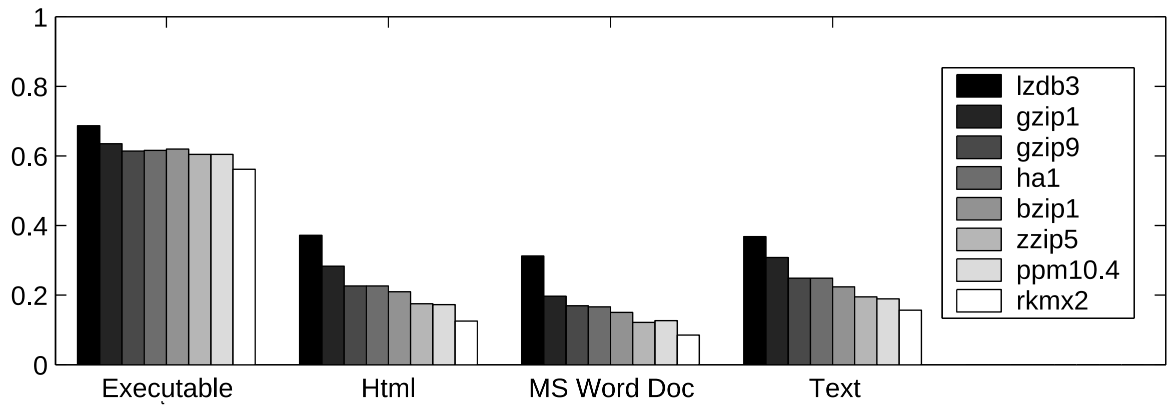


Figure 9.1: Compression ratios for different input data

be cut of more often than if the block was larger. Therefore, the compression ratio might become slightly worse when the block size is small. Since matching strings of normal text or html code are not more than a few characters long, the compression ratio would probably not become significantly worse if the block size is around 1.5 Kbyte. But, there might be some difference when using other coding techniques than fixed size coding. The problem occurs when using semi-adaptive methods for coding. Since these methods gather the statistics in one pass over the data and then writes this statistics in the beginning of each block, the overhead produced can make the compression ratio worse for small blocks.

When it comes to block algorithms, the situation is a bit different. Since the Burrows-Wheeler transform sort one block at the time, information from earlier blocks cannot be used to improve the compression ratio. In this study, no experiments were done using block sizes of 1.5 Kbyte. The reason for this was that the implementations tested were not able to handle such small blocks. In the implementation made by Burrows & Wheeler [1994] it turned out that a 1 Kbyte block gave 1.7 times worse compression ratio compared to when using a 750 Kbyte block. The input data used in this experiment was a text file, which could be compared to the text data used in the experiment of this study. If this relation would be valid, the *ZZIP9* implementation would with 1 Kbyte block give a compression ratio of about 32%, which is about 10% worse than the *Dic-Lz77-Semi-Huff* implementations.

Conclusions about compression ratio

- Different compression algorithms seem to achieve the same compression ratio for different input data relatively to other algorithms. I.e. the compression implementations that achieve best compression ratio do so for most document types, and the algorithms that achieves the worst compression ratio do so also for most document types.
- Statistical algorithms with high order achieves best compression ratio followed by block algorithms. After that comes dictionary-based algorithms and mainly lz77 variants followed by Huffman or arithmetic coding. Worst result produces statistical algorithms using order-0 models.
- Document types, such as Compressed files, Mp3, Gif, Jpg, MS Powerpoint and Pdf, which are already compressed with some algorithms are difficult to compress further with other lossless data compression methods.
- In the case of statistical and dictionary-based algorithms, the compression ratio is not significantly affected when changing the size of a block. But for block algorithms, the compression ratio becomes a lot worse, when the block becomes smaller.

9.1.3 Compression and decompression bandwidths

The compression and decompression bandwidths are dependent on several factors. These factors are analyzed and discussed in the following section.

Algorithms design and implementation

One of the most important factors that determine the compression and decompression bandwidth is the design and implementation of the algorithm. In chapter seven, three main categories for compression algorithms were given: statistical, dictionary-based and block algorithms. Each of these categories can be combined and result in many different algorithms. Since many algorithms can be implemented differently, the implementation also determines the compression and decompression bandwidth.

In Appendix D, two tables show the average compression and decompression bandwidths of algorithm implementations. The algorithms are sorted after the average bandwidth for the different document types. The standard deviation shows that the bandwidth for different document types varies quite much, which probably depends on the fact that different algorithms can compress data structures with different speed. The important information is not the exact values for certain implementations, but the relationship between different categories of algorithms. Figure 9.2 shows the average compression and decompression bandwidths for different implementations. The order of the algorithms are the same as the one given in Appendix C, which means that the rightmost achieves the best compression ratios.

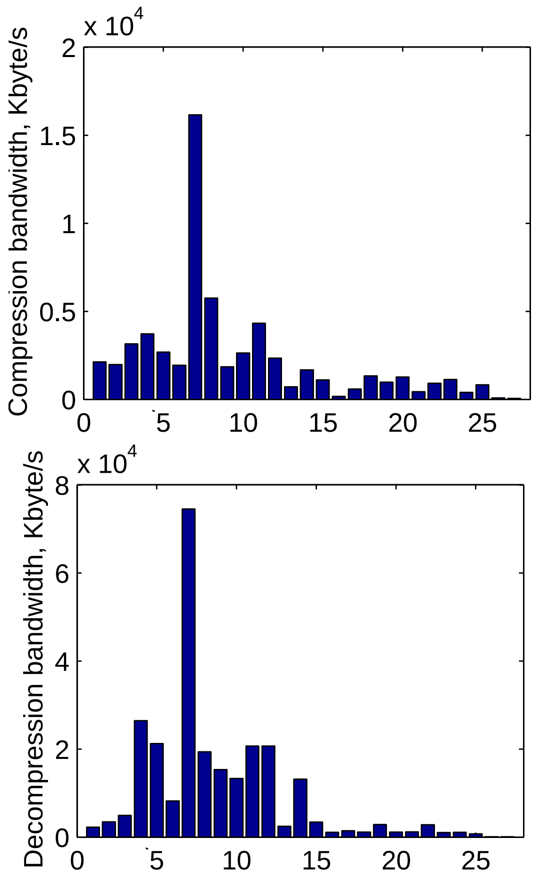


Figure 9.2: Compression and decompression bandwidth for different implementations

The diagrams in figure 9.2 clearly show that the algorithms that give the best compression ratios also achieves the worst compression and decompression bandwidths. The algorithms that perform worst compression ratio also produces quite low compression and decompression bandwidths. This has probably not so much to do with the algorithms design, it probably depends more on the implementation. The algorithms that produce both best compression and decompression bandwidths are the dictionary-based algorithms. Especially interesting is the *LZOP* implementation, which produces extremely higher compression and decompression ratios. This algorithm is of the category *Dic-Lz77-Fixed*, which means that it does not have to perform any complex coding technique to generate the codes. As can be seen in the tables in Appendix D, the *Dic-Lz77-Semi-Huff* algorithms have a compression bandwidth of approximately 1500 - 5000 Kbyte/s, but a decompression bandwidth of 13000 - 20000 Kbyte/s. The large difference is obvious, because the compression routine must perform time consuming searches after matching strings. For statistical and block algorithms, the difference between compression and decompression bandwidths is not as large. This since the block algorithms must perform the sorting for both compression and decompression and the statistical algorithms must build the same model on both sides.

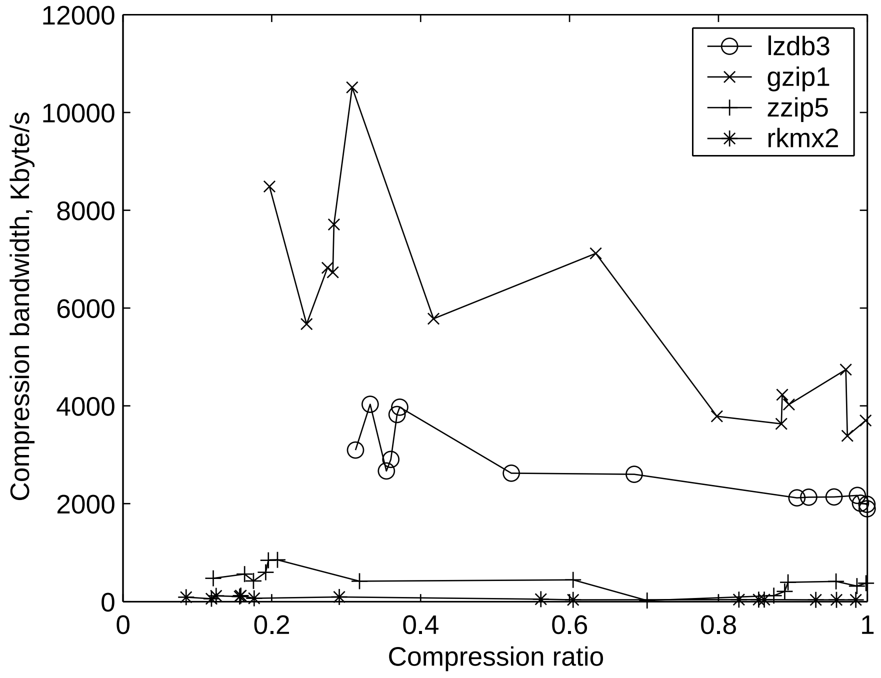


Figure 9.3: Relation between compression bandwidth and compression ratio

Dependence of the input

An interesting question is if the compression bandwidths of different algorithms are dependent of the input data. Figure 9.3 and figure 9.4 shows the relation between compression and decompression bandwidths for different algorithm implementations and compression ratios achieved on different input data. The algorithms chosen are showing the different categories of algorithms.

Figure 9.3 shows that the compression bandwidth tends to increase for the dictionary-based algorithms *lzdb3* and *gzip1* when the compression ratio becomes better. Since greedy parsing is used by these implementations, this observation seems reasonable. This because better compression ratios implies more long string matches. The longer the string matches are, the less searches must be done when compressing.

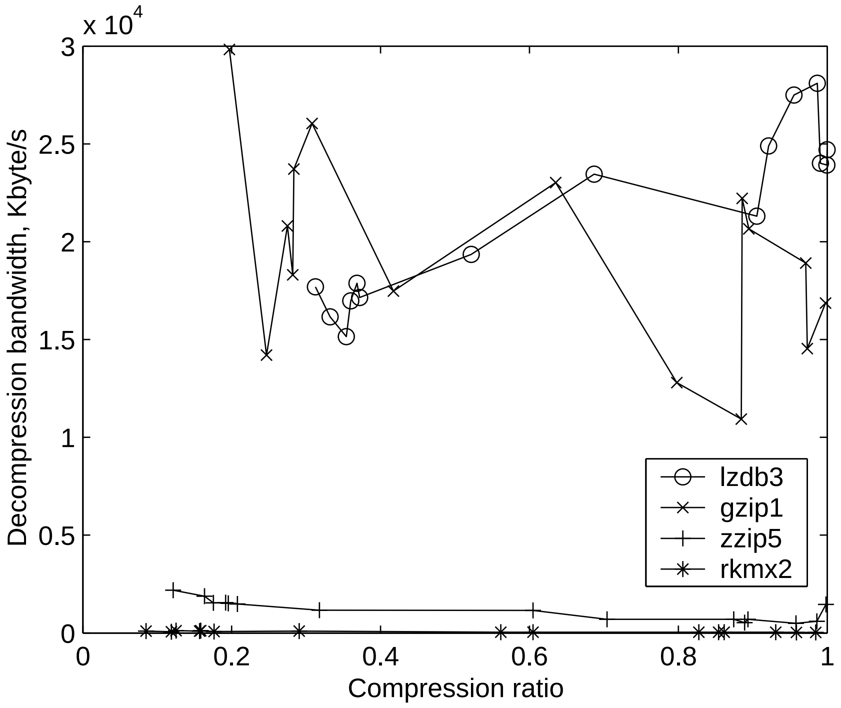


Figure 9.4: Relation between decompression bandwidth and compression ratio

In the case of the block and the statistical algorithms *zzip5* and *rkmx2*, it is quite difficult to see if there are any real trends. When observing the data shown in Appendix A, it shows that the compression bandwidth does not change significantly for the different document types.

Figure 9.4 shows the decompression bandwidths in relation to compression ratio. Here, the *lzdb3* implementation achieves better bandwidth when the compression ratio is poor. This might seem strange, but the probable reason is that the algorithm store the data uncompressed when no compression is achieved. Therefore, decompression is just a simple memory copy. In the case of *gzip1*, which is a *Dic-Lz77-Semi-Huff* implementation, the bandwidth vary very much. It is therefore hard to see any particular trend in this illustration. The block algorithm *zzip5*, seems to have worse decompression ratio when the data is poor compressed. Except for the compression of the already compressed file. The reason for this is probably the same as for the one of *lzdb3*. When observing the data of the statistical algorithm *rkmx2*, it seems like the decompression bandwidth is fairly the same independent of the input data type.

Dependences of the device

The most significant part determining the compression and decompression bandwidth is of course the processing capacity of the current device. As was shown in chapter 8.3, the difference in processing capacity differs much between different devices. For example, a normal laptop is more than 600 times faster than a Palm Vx. Approximately, this means that the same algorithm applied on the same data has a 600 times better compression or decompression bandwidth on a laptop compared to a Palm.

Conclusions about compression and decompress bandwidths

- Algorithms that produce the best compression ratios achieves the worst compression and decompression bandwidths.
- The algorithms that achieve best bandwidths, both for compression and decompression, seem to be dictionary-based algorithms. It seems like the dictionary-based algorithms that uses fixed end coding produce the best results.
- It seems to be a trend that worse compression ratios results in worse compression bandwidths. In the case of decompression bandwidth, no direct trends could be found.

9.1.4 Number of blocks

From expression (9.1), it easy to see that the number of blocks affects the expression significantly. Unfortunately, it is hard to estimate a value of n, which results in that it is difficult to say how much this parameter affects the result. In the case of the WVPN, the block size should be around 1.5 Kbyte. If the size of the document was known, it would be possible to estimate a value of N. But since the size of a requested data message varies this is not possible to calculate. To get an idea of what the size of N could be, a search in the search engine Altavista was made. This resulted in a document of the size 24 Kbyte, which would in this case mean a value $N = 16$ for a block size of 1.5 Kbyte. If the network carrier is GSM, the network bandwidth would be around 1.2 Kbyte/s. This means that the transfer time without compression would be about $24/1.2 = 20$ seconds. If instead a simple data compression algorithm was used, such as *Lzdb3* which is of type *Dic-Lz77-Fixed*, the transfer time over the network would decrease dramatically.

In the experiment, this algorithm achieved a compression ratio of approximately 37%, which would give the network transfer time $24 * 0.37/1.2 = 7.4$ seconds. The compression bandwidth for this algorithm is approximately 4000 Kbyte/s and decompression bandwidth 17000 Kbyte/s. If the delay produced by compression and decompression is calculated, this would be $1.5/4000 + 1.5/17000 = 0.4$ milliseconds. This value was calculated in the case that the devices were as fast as the reference computer used in the experiment. In the reality this is probably not the case, so the values are adjusted as if a Palm was used instead as the client. The server have a processing power per user 100 times worse than the reference computer. In this case, the extra delay time would be $1.5 * 100/4000 + 1.5 * 1000/17000 = 0.13$ seconds. This example shows that extra delays that the compression and decompression produce are small compared to the time saved by compression.

This is of course no evidence that the compression and decompression time is not important, but it shows that the importance decreases when the compression ratio is good. If almost no compression was achieved, it is obvious that compression should not be used, since it just adds a delay to the transmission. The problem in the WVPN is that the compressing unit does not know if it will achieve any compression before the compression is applied. More important to notice is that if the network bandwidth is high, the compression or decompression part can be the bottleneck in the system. Therefore, it must be evaluated for which network bandwidth that data compression should be turned off.

Conclusions about number of blocks

- When the value n is large, the delay produced by compression and decompression decreases.
- It is hard to almost impossible to estimate a certain value of n.

9.1.5 Memory

How much memory an algorithm requires is important because different devices have different amount of memory available. More memory usage can also affect the performance of thealgorithms, since memory cache misses are expensive. It is impossible to say exactly how much memory a certain algorithm category requires. Therefore, this section gives a brief outline of memory usage in different algorithm categories.

Statistical algorithms

All statistical algorithms used in this test consume more than one Mbyte of memory, except for the order-0 model implementations. It is probably possible to implement a statistical model that only uses a couple of hundred Kbyte. But if this is done, the compression ratio will decrease. The implementation *PPM1_4* uses 1 Mbyte of memory with a order-4 model. But it achieves a compression ratio that is just a few percent better than the best dictionary-based implementation. Therefore, a statistical implementation that uses less than 500 Kbyte memory probably just produces a bit better compression ratio than a dictionary-based algorithm.

Dictionary-based algorithms

Dictionary-based algorithms with an Lz77 implementation require memory to save the sliding window. If fixed end coding is used, no more memory is needed to actually decompress the data. To encode the data, the extra memory needed is the hash tables used to search for matching strings. But if an end coding such as Huffman coding is used, memory must be allocated to handle the Huffman trees. The amount of memory required for these trees depends on how many symbols that are used. If the end coding used is arithmetic, the memory requirements would also be proportional to the number of symbols, since an order-0 model must hold the frequency of the symbols.

Lz78 based algorithms need memory to store the dictionaries. The amount of memory required is of course proportional to the number of entries in the dictionary.

Dictionary-based algorithms could be implemented to use very little memory if fixed codes are used. If a small window or dictionary is used, no more than approximately 30 Kbyte is needed for compression. For decompression, even less is required, since no hash tables have to be allocated.

If larger window sizes are used and an end coding such as Huffman are used, more memory is required. But these algorithms should be possible to implement in a couple of 100 Kbyte. For example, Jiang [1996] claims that the implementation *ARJ* requires about 290 Kbyte of memory.

Block algorithms

Block algorithms that use Burrows-Wheeler transform require different amount of memory depending on how large the block size is. According to Burrows & Wheeler [1994], their implementation requires about 7 times more memory than the size of the block. Since the sorting algorithm can be optimized both for memory and speed, the algorithm can probably be implemented to require less memory. The end coding algorithms tested in this experiment use arithmetic or Huffman coding. Since these algorithms are executed after the sorting, they should not require any extra memory.

Conclusions about memory requirements

- Statistical algorithms require quite large amount of memory when the order of the model is high. If the order of the model is low, the amount of memory required is significantly lower with the drawback that the compression ratio becomes much worse.

- Dictionary-based algorithms can be implemented to use small amounts of memory, especially when decompressing. If an end coder is used, such as Huffman, memory requirement becomes a bit higher.

- Block algorithms require memory proportional to the block size used.

9.2 Strength-weakness analysis of algorithms

From the discussion given earlier in this chapter, strengths and weaknesses of the different algorithm categories are here summarized:

9.2.1 Statistical algorithms

Strengths

- With a good model, statistical algorithms can produce the state of the art compression ratios.
- They are highly suitable for streaming purposes, since character are coded one by one. No lookahead buffer is needed.

Weaknesses

- The models that produce good compression ratios consume a lot of memory.
- Large models result in both low compression and decompression bandwidths.
- Large amount of memory must be allocated both on the compression and the decompression side.

9.2.2 Dictionary-based algorithms

Strengths

- Can be implemented to require a small amount of memory.
- The memory requirements are lower when decompressing compared to when compressing.
- Lz77 based algorithms that uses an end coder such as Huffman, can produce a compression ratio only a few percent worse than the statistical algorithms.
- Algorithms with fixed coding size can be implemented to be extremely fast for compression and decompression.

Weaknesses

- Dictionary-based algorithms can not produce as good compression ratios as statistical or block algorithms.

9.2.3 Block algorithms

Strengths

- Compression ratios are almost as good as the best statistical algorithms, if the block size is large.

Weaknesses

- When the block size is small, the compression ratio becomes much worse.

9.3 Improving the performance in a WVPN

In this section a discussion is made to estimate which category of algorithms that is most suitable in a WVNP.

9.3.1 Memory requirement

The less memory the data compression algorithm requires, the better it is. But this is of course a trade-off to compression bandwidth and compression ratio. On the client side, both Pocket PC devices and laptops have fairly much memory. On these devices, algorithms could use up to some Mbyte of memory, but the less, the better. The bottleneck of the devices is the Palm. Here not more than about 64 Kbyte is available and this memory should be shared with the actual application used.

On the server side, it is not that easy to say how much memory that is available for data compression. A modern server can be configured to have enormous amount of memory, but since this is an economical aspect, it is outside this study. Another factor that determines the amount of memory available for data compression is the number of simultaneous users. This is a requirement of the WVPN product. In the case of Columbitech's WVPN, Hovmark [2001] estimates that a reasonable memory size would be a maximal of 500 Kbyte.

The question is then, which algorithms can be used to fit into these requirements? When a Palm is used as the client, it is obvious that statistical algorithms are not that suitable. If a statistical model were used with these memory requirements, a very low model order must be used. In this case, dictionary-based algorithms would probably achieve much better compression ratios. In the case of Block algorithms, very small blocks must be used, which gives worse compression. Even dictionary-based algorithms followed by Huffman coding probably require too much memory. It seems like the most proper solution available for a Palm is dictionary-based algorithm using fixed size codes.

If the other devices are assumed to be able to handle quite large memory requirements, the bottleneck of memory is probably the server. The dictionary-based algorithms are obviously possible to use, but is for example the block algorithms usable? The answer is likely to be yes, presupposed that the block size is not too large. Since the block size in the WVPN is about 1.5 Kbyte, block algorithms have no problem to fit into the memory requirements of the server. When it comes to statistical algorithms, the answer is a bit more complicated. In this experiment, only statistical algorithms that uses more than 1 Mbyte of memory was tested. These algorithm implementations are probably too memory consuming to be used. Especially if the estimated memory requirements stated by Hovmark is the limit. But there might be implementations using less memory. In Bell, Cleary & Witten's [1990] compression experiment, they used a variant of PPM called PPMC that needed less than 500 Kbyte memory. In their tests, this implementation resulted in better compression ratios than dictionary-based algorithms.

9.3.2 Streaming requirement

As discussed earlier, it seems like statistical and dictionary-based algorithms do not have any particular problems to divide the data into small blocks. The only exception is probably dictionary-based algorithms using semi-adaptive coding at the end. In the case of dictionary-based algorithms, it would therefore be more suitable with adaptive end coding. But for block algorithms, it seems like the compression ratio will be much worse when the blocks are small. By this reason, it can be assumed that block algorithms are not that suitable in the WVPN.

9.3.3 Bandwidth and compression ratio

The compression and decompression bandwidths are important factors when choosing a suitable compression algorithm. But since the factor n is unknown, it is hard to estimate how large the delay is compared to the time saved by compression. Since the WVPN must stream the data in small blocks, it is important that the bottleneck is not the data compression. If the compression or decompression is the bottleneck, data compression ought to be turned off. The question is; when does the data compression become the bottleneck? This depends on several factors, where the speed of the device, the server and the algorithm used are the main properties. The third table in Appendix D shows the decompression for different devices and implementations.

As can be seen in the table, decompressing on the Palm is quite time consuming, because of the slow CPU. According to earlier discussions, the only usable type of implementation on a Palm would be a *Dic-Lzss-Fix* implementation, mainly because of the memory requirements. It turns out that these are probably the most suitable solutions even when it comes to decompression bandwidth. For these sorts of algorithms, it seems like they would only be the bottleneck when a really fast network is used, such as WLAN. But, since the CPU is that slow, the probable bottleneck is the decryption routine. If the decryption is much slower than the decompression, data compression could in fact be beneficial independent of the bandwidth of the network carrier.

In the case of Pocket PC, it is a lot more complicated. There are several different versions, with different processing capacity. The table shows decompression for a fast and a slow Pocket PC, which is the same as the Compaq iPAQ and Compaq Aero explained in chapter 8. As discussed earlier, probably any statistical and dictionary-based algorithm can be used on a pocket PC. The decompression bandwidth varies quite much depending on which pocket PC used. When a slow network is used, such as GSM, GPRS, CDPD or HSCSD, the most suitable solution is probably to use a compression algorithm that produces best compression ratio. It was shown earlier that the statistical algorithms outperform other methods in compression ratio. But for faster network such as Bluetooth and WLAN, these methods might be too slow. In these cases, a dictionary-based algorithm is probably more suitable.

Laptops nowadays are very fast. In the table D.3, it shows that even the statistical algorithm could be possible to use in a WLAN. The bottleneck when using a laptop is probably not the decompression speed.

When it comes to the server, it is much harder to estimate the processing capability. Not just the fact that there exist many different types of servers, the number of simultaneous users varies also. Since the devices described above can be configured to use suitable algorithms, it is probably the capability of the server that states the limit of when data compression can be used. In table D.4 the compression bandwidth is shown for different number of users. As can been seen, the compression bandwidth varies extremely much depending on the number of simultaneous users. This table shows the compression time on the reference computer, but there exist of course lots of more powerful servers. Since the compression is done by the server, it can decide in run time when compression should be turned off. This is probably a quite complicated task to perform in practice, but could give a good result if implemented correctly. An easier course of action would be to just turn of compression when using fast connections such as WLAN and then test and restrict the number of simultaneous users on the server.

Chapter 10

Conclusions

In this chapter the conclusions achieved from the previous chapter are summarized.

It has been shown in this study that there were several parameters affecting the behaviour of data compression in a WVPN. It was possible to give a theoretical model of when data compression was beneficial, but since all parameters varies a lot it was not trivial to use this expression in practice. It has been shown that adding data compression can both increase and decrease the performance in a WVPN, depending on several factors. The main factor that decreased the performance when using data compression was the extra delay time added when compressing and decompressing the data. Since a WVPN had to split the entire data message into small blocks, the delay produced is only for the first block sent. This was true only if the bottleneck in the network was not the compression or decompression routine. If the bottleneck would be compression or decompression, the actual network bandwidth would not be used optimally. The main and obvious factor that could increase the performance was when the data compression reduces the amount of data sent on the network. But, since the WVPN could never know how good compression ratio achieved before compression was applied, it was not possible to decide if compression should be used or not with help of this information.

In the experiment it was noticed that the compression algorithms that achieved the best compression ratios seemed to do so for all document types and the algorithms that achieved worst compression ratio did so also for most document types. By this reason, it would be proper to use the compression algorithms that could perform the best compression ratios, presupposed that they could handle the memory requirements, streaming requirements and requirements of not being the bottleneck in the network.

There were three main categories of compression algorithms found in this study: statistical, dictionary-based and block algorithms. The experiment showed that they had all different benefits and disadvantages, which were important to consider when to use them in a WVPN.

The algorithms found giving best overall compression ratios were statistical algorithms. The larger the order of the model was, the better compression ratio was achieved. Block algorithms gave nearly the same compression ratios as the best statistical algorithms, presupposed that the block size was large. When the block size was the same size as in the WVPN, it seemed to give even worse result than for a dictionary-based algorithm. Dictionary-based algorithms did not achieve that good compression ratios as the above described algorithm categories, but for most document types acceptable ratios. The largest advantage of dictionary-based algorithms was that they had low memory requirements and showed to be able to achieve really good compression bandwidth and even better decompression bandwidth.

To achieve better performance in a WVPN, it was suggested that the most suitable solution for a Palm would be a dictionary-based algorithm that consumes very small amount of memory. The probable solution would be to use a LZ77 based algorithm followed by fixed coding.

In the case of Pocket PC, the suggestion was to use different approaches for different network bandwidths. For slow network carriers, such as GSM, CDPD, HSCSD and GPRS, a statistical algorithm was suggested. But for faster networks, such as Bluetooth and GPRS, the faster category dictionary-based algorithms were suggested.

When the device is a laptop, the decompression could be done so fast that the probable best solution was to use a statistical algorithm all the time.

In the case of the server of the WVPN, it was shown to be quite much more complicated. The processing power of the server depends on many factors such as the type of server and the number of simultaneous users. The conclusion was that it is the capabilities of the server in memory and processing power that will probably set the limit when data compression should or should not be used.

Chapter 11

Reflections and further research

In this final chapter, reflections of this study are stated and recommendations for further research are given.

11.1 Reflections

When the work with this thesis started, the author of this study was relatively inexperienced in the field of data compression. In the beginning, a lot of time was spent on implementing different algorithms and investigating different approaches to achieve good data compression. From the beginning, the idea was to implement all different algorithms and then compare them, but when it was clear how much the implementation affected the result, it was decided to use another approach. Instead a number of different implementations were investigated in order to see how good different algorithms could behave in reality. The drawback with this approach was that the variants of different algorithms decreased and the possibility of investigating the implemented algorithms more deeply became harder.

As the knowledge of the areas of data compression and data communication became more extensive, the clearer it was that the field was really huge. For this reason it was difficult getting an overview and thus knowing how to focus on the problem. To be able to handle the problem, the study had to have a more outlined character. Even if this approach was not the optimal solution, it was probably the most realistic course of action.

Data compression is a very exciting field, which has been important for many years and will most certainly be at least as important in the future. Even if extensive research has been made, many areas remain unexplored.

11.2 Further research

In the following section, a number of areas interesting to further investigate are listed.

Encryptions influence As have been discussed in the study, encryption can be the bottleneck in bandwidth, especially for slow devices. But what is this limitation in practice for different devices? How much can the performance improve by fast data compression?

Dictionary-based algorithms This study has shown that there exist extremely many variants of dictionary-based algorithms. To be able to analyze the properties of different variants, further experiments and implementations must be done. Is it possible to implement end coding with static Huffman coding that achieves nearly the same compression ratios as semi-adaptive versions? Is it possible to optimize adaptive Huffman as end coder? Which are the most suitable sizes of windows for LZ77 based algorithms? Can variants of LZ77 and LZ78 be combined to achieve even better results?

Statistical algorithms Statistical algorithms is a huge area, where extensive research has been made during the last decade. Further investigation of research made and comparisons with requirements of the properties in a WVPN can be very interesting. How can the model be designed to quickly adapt to changes in the input data? Is it possible to create models that need less memory, but keep the compression ratio?

Experiments in the WVPN Since Columbitech's WVPN is not yet finished, real tests in the system were not possible to perform in this study. Many interesting experiments can be done, to see if a certain device actually can handle a certain compression algorithm. Does the performance in fact increase? Can a user notice the difference? Many possibilities exist to verify how much compression actually increases the performance.

Patent rights The field of data compression is mined with patents of different compression techniques. To analyze and know if a certain compression algorithm conflicts with patents is an extremely important issue for companies using compression in their products. Which algorithms do patents strictly protect? In which countries do these patents apply? Does it exist algorithms that are not patented?

Bibliography

Books

[Bell, Cleary & Witten 1990] Bell, T.C., Cleary, J.G., Witten, I.H. 1990, *Text Compression*, Prentice Hall

[Hankerson, Harris & Johnson 1998] Hankerson, D., Harris, G.A., Johnson, P.D. 1998, *Introduction to Information Theory and Data Compression*, CRC Press LLC, Florida

[Nelson & Gailly 1996] Nelson, M. & Gailly J-L. 1996, *Data Compression Book*, 2nd edn, M&T Books, New York

[Peterson & Davie 2000] Peterson, L. L. & Davie B.S. 2000, *Computer Networks - A systems approach*, 2nd edn, Academic Press, United States of America

[Salomon 1997] Salomon, D. 1997, *Data compression - the complete reference*, Springer-Verlag New York, Inc

[Storer 1988] Storer, J. A. 1988, *Data compression - methods and theory*, Computer Science Press, Inc., USA

[Weiss 1996] Weiss, M. A. 1996, *Algorithms, Data structures, and problem solving with C++*, Addison Wesley Longman, Inc.

Articles and reports

[Bloom 1998] Bloom, C. 1998, *Solving the Problems of Context Modeling*, [Online], Available: http://www.cbloom.com/papers/ppmz.zip
[2001, May 5]

[Burrows & Wheeler 1994] Burrows, M., Wheeler, D.J. 1994, *A Block-sorting Lossless Data Compression Algorithm*, SRC Research Report, Digital Systems Research Center, [Online], Available: http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html [2001, April 10]

[Campos 1999] Campos, A.S.E. 1999, *Canonical Huffman*, [Online], Available: http://www.arturocampos.com/ac_canonical_huffman.html [2001, April 25]

[Campos 1999] Campos, A.S.E. 1999, *Range coder*, [Online], Available: http://www.arturocampos.com/ac_range.html [2001, May 8]

[Campos 1999] Campos, A.S.E. 1999, *Run Length Encoding*, [Online], Available: http://www.arturocampos.com/ac_rle.html [2001, April 26]

[Campos 1999] Campos, A.S.E. 1999, *Move to front*, [Online], Available: http://www.arturocampos.com/ac_mtf.html [2001, April 27]

[Fenwick 1996] Fenwick, P. 1996, *Block Sorting Text Compression*, [Online], Available: ftp://ftp.cs.auckland.ac.nz/out/peter-f/ACSC96paper.ps [2001, April 5]

[Lelewer & Hirschberg 2001] Lelewer, D.A., Hirschberg, D.S. 2001, *Data Compression*, [Online], Available: http://www.ics.uci.edu/~dan/pubs/DataCompression.html [2001, April 24]

[Martin 1979] Martin, G. N. N. 1979, *Range encoding: an algorithm for removing redundancy from a digitised message*, IBM UK Scientific Center, [Online], Available: http://www.compressconsult.com/rangecoder/rngcod.pdf.gz [2001, April 7]

[Nelson 1991] Nelson, M. 1991 *Arithmetic Coding + Statistical Modeling = Data Compression*, [Online], Available: http://www.dogma.net/markn/articles/arith/part1.htm [2001, April 16]

[Nelson 1996] Nelson, M. 1996 *Data compression with the Burrows-Wheeler Transform*, Dr. Dobb's Journal, [Online], Avaliable: http://www.dogma.net/markn/articles/bwt/bwt.htm [2001, April 10]

[Nelson 1998] Nelson, M. 1998 *LZW Data Compression*, Dr.Dobb's Journal, October 1989, [Online], Available: http://www.dogma.net/markn/articles/lzw/lzw.htm [2001, May 3]

Internet

[*802.11a - a whatis definition*] *802.11a - a whatis definition*, [Online], Available: http://whatis.techtarget.com/definition/ 0,289893,sid9_gci341007,00.html [2001, May 10]

[*Archive Comparison Test*] *Archive Comparison Test*, [Online], Available: http://web.act.by.net/~act/act-summary.html [2001, May 15]

[Bloom 1996] Bloom, C. 1996, *Compression: Algorithms: Statistical Coders*, [Online], Available: http://www.cbloom.com/algs/statisti.html [2001, April 11]

[Bourgin 1995] Bourgin, D. 1995, *Introduction to the losslessy compression schemes, Description of the source files and the methods*, File: compress.doc in archive codecs.zip, [Online], Available: http://www.cs.pdx.edu/~idr/compression/source/codecs.zip [2001, May 15]

[*Bzip2 and libbzip2 - Introduction*] *Bzip2 and libbzip2 - Introduction*, File: manual_1.html in archive bzip2-1.0.1.tar.gz [Online], Available: ftp://sourceware.cygnus.com/pub/bzip2/v100/bzip2-1.0.1.tar.gz [2001, May 15]

[*CDPD - a whatis definition*] *CDPD - a whatis definition*, [Online], Available: http://whatis.techtarget.com/definition/ 0,289893,sid9_gci213843,00.html [2001, May 10]

[*Columbitech homepage* 2001] *Columbitech homepage*, [Online], Available: http://www.columbitech.com/ [2001, April 2]

[*Columbitech wireless VPN* 2001] *Columbitech wireless VPN*, [Online], Available: http://www.columbitech.com/documents/ ColumbitechWVPNProductSheet.pdf [2001, April 3]

[*comp.compression Frequently Asked Questions (Part 1/3)*] *comp.compression Frequently Asked Questions (Part 1/3)*, [Online], Available: http://www.math.ist.utl.pt/stat/help/Compression.html [2001, May 13]

[Devenport 1988] Devenport, M. 1988 *MDCD Version 1.0* file: mdcd.doc in archive mdcd10.zip [Online], Available: ftp://ftp.cdrom.com/pub/sac/pack/mdcd10.zip [2001, May 2]

[*FLZ Data Compression*] *FLZ Data Compression*, [Online], Available: http://www.cs.pdx.edu/∼idr/unbzip2.cgi?compression/flz.html.bz2 [2001, May 23]

[Gailly 1996] *Algorithm*, File: algorithm.doc in source package gzip in cygwin installation. [Online], Available: http://sources.redhat.com/cygwin/ [2001, April 5]

[Geier 2000] Geier, J. 2000, emphCDPD Concepts, [Online], Available: http://www.wireless-nets.com/whitepaper_cdpd.htm [2001, May 10]

[*GPRS - Data transmission for mobile telephony*] emphGPRS - Data transmission for mobile telephony, Siemens, [Online], Available: http://www.siemens.ie/mobile-business/gprs.htm [2001, May 10]

[*GPRS - General Packet Radio System* 2000] *GPRS - General Packet Radio System* 2000, Siemens Whitepaper, [Online], Available: http://www.siemens.ie/mobile-business/gprs.pdf [2001, May 10]

[*GSM - digital mobile radio technology for beginners*] [*GSM - digital mobile radio technology for beginners*, Siemens, [Online], Available: http://www.siemens.ie/mobile-business/gsm.htm [2001, May 4]

[*Handspring homepage* 2001] *Handspring homepage* 2001, [Online], Available: http://www.handspring.com/ [2001, May 6]

[*HSCSD - a whatis definition*] *HSCSD - a whatis definition*, [Online], Available: http://whatis.techtarget.com/definition/0,289893,sid9_gci213692,00.html [2001, May 11]

[*Huffman Coding* 1997-2000] *Huffman Coding* 1997-2000, DataCompression Reference Center, [Online], Available: http://www.rasip.fer.hr/research/compress/algorithms/fund/huffman/ [2001, April 25]

[Jiang 1996] Jiang, W. 1996 *JAM/UNJAM manual*, File: jam.doc in archive jam.zip, [Online], Available: ftp://ftp.cdrom.com/pub/sac/pack/jam.zip [2001, May 15]

[Kieffer 1999] Kieffer, J.C. 1999, *Class notes*, [Online], Available: http://www.ece.umn.edu/users/kieffer/ece5585.html [2001, April 20]

[*Motorola Bluetooth [FAQ]*] *Motorola Bluetooth [FAQ]*, [Online], Available: http://www.motorola.com/bluetooth/faq/faq.html [2001, May 3]

[Oberhumer 1999] Oberhumer, M.F.X.J 1999, *LZO – a real-time data compression library*, [Online], Available: http://wildsau.idv.uni-linz.ac.at/mfx/lzodoc.html [2001, May 16]

[*Palm OS Memory Architecture*] *Palm OS Memory Architecture*, [Online], Available: http://oasis.palm.com/dev/kb/papers/1145.cfm [2001, May 9]

[*Palm OS - platform*] *Palm OS - platfrom*, [Online], Available: http://www.palmos.com/platform/ [2001, May 8]

[Phamdo 2000] Phamdo, N. 2000, *Theory of Data Compression*, [Online], Available: http://www.data-compression.com/theory.html [2001, April 9]

[Ratushnyak 2001] Ratushnyak, A. 2001, *The Green Tree Of Compression Methods - A Practical Introduction To Data Compression*, [Online], Available: http://geocities.com/eri32/int.htm [2001, Mars 18]

[*RK Software*] *RK Software*, [Online], Available: http://rksoft.virtualave.net/rk.html [2001, May 16]

[Schindler 1999] Schindler, M. 1999, *Range encoder Homepage*, [Online], Available: http://www.compressconsult.com/rangecoder/ [2001, May 8]

[Schäfer 2000] Schäfer, V. 2000, *HSCSD und GPRS im Teltarif-Test*, [Online], Available: http://www.teltarif.de/arch/2000/kw37/s3058.html [2001, May 11]

[Seward 2000] Seward, J. 2000, *Bzip2 description*, File: bzip2.txt in archive bzip2-1.0.1.tar.gz [Online], Available: ftp://sourceware.cygnus.com/pub/bzip2/v100/bzip2-1.0.1.tar.gz [2001, May 10]

[Shkarin 2001] Shkarin, D. 2001, *Readme file.* File: read_me.txt in archive ppmde.rar, [Online], Available: ftp://ftp.cdrom.com/pub/sac/pack/ppmde.rar [2001, May 10]

[*Source code of HA*] *Source code of HA*, ftp://ftp.cdrom.com/pub/sac/pack/ha0999.zip [2001, May 11]

[*VPN - a whatis definition* 2000] *VPN - a whatis definition* 2000, [Online], Available: http://whatis.techtarget.com/definition/0,289893,sid9_gci213324,00.html [2001, April 23]

[Wierlemann & Kassing 1998] Wierlemann, T. & Kassing, T. 1998, *Performance of TCP/IP and its Application Protocols over Narrowband Bearers with high Latency*, [Online], Available: http://www.w3.org/1998/11/05/WC-workshop/Papers/wierlemann.html [2001, May 7]

[*Windows CE 3.0 FAQ* 2001] *Windows CE 3.0 FAQ* 2001, [Online], Available: http://www.microsoft.com/windows/embedded/ce/guide/features/ce30faq.asp [2001, May 6]

[*Zzip's webpage*] *Zzip's webpage* [Online], Available: http://www.zzip.f2s.com/ [2001, May 16]

Contacted persons

[Englund 2001] Englund, Tobias, Technical business developer, Columbitech AB, Stockholm, Oral interview, 2001-05-10

[Hovmark 2001] Hovmark, Torbjörn, Chief technology officer, Columbitech AB, Stockholm, Oral interview 2001-04-12, 2001-05-07

[Jung 2001] Jung, R., Creator of the archiving program ARJ, e-mail interview, 2001-05-15

[Roshal 2001] Roshal, E. Creator of the archiving program RAR, e-mail interview, 2001-05-16

Index

802.11, 31, 96

Adaptive Huffman, 45
Adaptive model, 13
AHUFFDB1, 35
Algorithms, 28
Arithmetic
  Decoding, 51
  Encoding, 49
Arithmetic coding, 27, 49
ASCII, 19

Bandwidth, 13, 15
Bit, 19
Block mode, 12
Block sorting, 28, 72
Bluetooth, 31, 96
Burrows-Wheeler Transform, 28, 72
BWT, 28, 72
Byte, 19

Canonical Huffman, 45
CDPD, 31, 95
Character, 19
Client device, 29
  Laptop, 30
  Palm, 29
  Pocket PC, 29
Code, 19
Codes
  Prefix, 24
  Variable size, 23
Coding, 26
  Arithmetic, 27
  Huffman, 26, 42
  Shannon-Fano, 26, 39
Columbitech, 7
Communication, 14
Compression
  Bandwidth, 15
  Ratio, 14

Data communication, 14
Data Structure
  Bit, 19
  Byte, 19
  Character, 19
  Code, 19
  DWORD, 19
  Word, 19
Data structure, 19
  Symbol, 19
Delay, 13
Delimitations, 6
Dictionary-based modeling, 61
DWORD, 19
Dynamic Huffman, 45
Dynamic model, 13

Entropy, 20

Finite context modeling, 58

GPRS, 31, 95
Greedy parsing, 66
GSM, 30, 94

HSCSD, 31, 95
Huffman, 26, 42

Huffman tree, 42

Information Theory
  Entropy, 20
Information theory, 20

Latency, 13
Lazy coding, 66
Lookahead buffer, 61
Lossless, 11
Lossy, 11
LZ77, 61
LZDB3, 35

Modeling, 27
  Dictionary-based, 27
  Statistical, 27
Modes
  Block, 12
  Streaming, 12
Move-to-front, 27, 69
MTF, 27, 69, 75

Network carrier, 30
  802.11, 96
  Bluetooth, 31, 96
  CDPD, 31, 95
  GPRS, 31, 95
  GSM, 30, 94
  HSCSD, 31, 95
  WLAN, 31, 96

Palm, 29
Performance, 3, 13
Pocket PC, 29
Prefix codes, 24
Prefix property, 24
Product, 13

Redundancy, 20
RLE, 54
Roaming, 7
Run length encoding, 54

SARITHDB1, 35
Security, 7
Server properties, 30
Session layer, 29
Shannon, 20
Shannon-Fano, 26, 39
Sibling property, 46
Static model, 12
Statistical modeling, 57
Streaming mode, 12
Symbol, 19

Throughput, 13
Transformation, 27

Visor, 30
VPN, 2

WLAN, 31, 96
Word, 19
WVPN, 2

Appendix A

Result from experiment

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	C++ source code	15,85%	118,2	123,5
RKMX1	C++ source code	16,11%	141,6	142,5
PPM10_8	C++ source code	17,35%	1675,5	1541,8
ZZIP9	C++ source code	18,82%	519,8	1532,5
ZZIP5	C++ source code	19,17%	596,2	1554,7
BZIP9	C++ source code	19,30%	1093,9	3659,8
BZIP5	C++ source code	19,67%	1174,3	3843,2
PPM2_4	C++ source code	19,95%	2678,7	2319,5
PPM10_4	C++ source code	20,04%	2500,2	2224,7
PPM1_4	C++ source code	20,15%	2782,1	2360,7
ZZIP1	C++ source code	20,79%	687,6	1684,9
BZIP1	C++ source code	21,31%	1320,0	4533,2
RAR	C++ source code	21,44%	2131,2	16354,9
HA1	C++ source code	21,85%	830,0	3220,6
GZIP9	C++ source code	22,16%	2025,6	23865,2
HA2	C++ source code	22,67%	387,5	406,5
GZIP5	C++ source code	22,74%	4966,4	20788,1
ARJ1	C++ source code	22,84%	2298,9	16174,0
LHA	C++ source code	24,44%	1558,2	17713,1
GZIP1	C++ source code	27,48%	6823,2	20809,3
LZDB3	C++ source code	33,22%	4033,9	16161,3
LZOP	C++ source code	35,26%	22575,9	72869,4
JAM	C++ source code	37,65%	2729,8	3523,1
MDCD	C++ source code	41,41%	4648,4	6011,8
LZW	C++ source code	43,80%	3503,8	6067,1
SARITHDB1	C++ source code	65,03%	2044,4	951,0
AHUFFDB1	C++ source code	71,07%	2173,7	2349,3

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX1	Compressed files	98,16%	43,9	42,2
RKMX2	Compressed files	98,47%	33,9	33,6
ZZIP9	Compressed files	99,35%	313,0	837,8
BZIP9	Compressed files	99,58%	589,5	1621,1
GZIP9	Compressed files	99,76%	3449,5	14581,9
GZIP5	Compressed files	99,76%	3690,3	16721,3
GZIP1	Compressed files	99,79%	3701,3	16863,4
ZZIP5	Compressed files	99,83%	376,8	1462,6
ZZIP1	Compressed files	99,86%	409,4	3585,5
LZOP	Compressed files	99,90%	9711,8	66548,4
RAR	Compressed files	99,94%	1107,5	8131,0
LZDB3	Compressed files	99,96%	1986,5	23922,1
SARITHDB1	Compressed files	99,97%	1623,1	27018,3
JAM	Compressed files	100,00%	1012,1	16498,7
LHA	Compressed files	100,00%	1977,4	35287,2
HA1	Compressed files	100,00%	711,1	6852,6
HA2	Compressed files	100,00%	34,0	6975,5
ARJ1	Compressed files	100,00%	6862,1	14412,4
MDCD	Compressed files	100,00%	2238,8	52743,1
PPM10_8	Compressed files	100,02%	166,7	157,3
AHUFFDB1	Compressed files	100,07%	1650,1	1680,9
BZIP5	Compressed files	100,21%	607,6	1674,1
PPM10_4	Compressed files	100,49%	165,6	156,5
BZIP1	Compressed files	100,57%	646,3	1992,5
PPM1_4	Compressed files	101,73%	293,6	262,4
PPM2_4	Compressed files	102,08%	263,6	238,5
LZW	Compressed files	136,08%	2199,3	3448,9

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	Executables	56,15%	46,2	44,3
RKMX1	Executables	56,24%	59,4	56,6
PPM10_8	Executables	59,54%	318,0	293,6
ZZIP9	Executables	60,34%	409,0	969,3
RAR	Executables	60,43%	1493,1	12605,0
PPM10_4	Executables	60,43%	320,9	296,3
ZZIP5	Executables	60,47%	445,0	1149,4
PPM2_4	Executables	60,52%	496,2	434,4
PPM1_4	Executables	60,63%	552,6	474,7
BZIP9	Executables	60,92%	826,8	2372,4
ZZIP1	Executables	61,22%	562,1	1679,7
BZIP5	Executables	61,29%	906,8	2420,0
GZIP9	Executables	61,37%	1946,7	24819,1
ARJ1	Executables	61,56%	2322,4	13203,8
HA1	Executables	61,56%	717,8	1538,6
GZIP5	Executables	61,61%	5306,2	24431,7
BZIP1	Executables	61,96%	994,1	2845,5
HA2	Executables	62,35%	59,9	62,7
LHA	Executables	62,36%	1920,8	13425,2
GZIP1	Executables	63,52%	7116,9	23017,0
LZOP	Executables	68,33%	12204,7	67433,7
LZDB3	Executables	68,68%	2600,6	23455,2
JAM	Executables	73,79%	1520,3	2683,4
SARITHDB1	Executables	80,84%	1895,9	1654,0
LZW	Executables	86,90%	3077,9	4765,2
MDCD	Executables	89,21%	3607,1	5012,2
AHUFFDB1	Executables	90,46%	1993,2	2002,5

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	Gif	93,10%	35,1	34,8
RKMX1	Gif	96,77%	43,6	41,7
RAR	Gif	97,06%	1090,9	7205,7
GZIP9	Gif	97,21%	3291,2	9163,0
GZIP5	Gif	97,22%	3202,0	14105,9
GZIP1	Gif	97,33%	3389,4	14536,0
ARJ1	Gif	97,37%	1981,6	7109,0
HA1	Gif	97,59%	804,5	997,1
LZOP	Gif	98,06%	8488,7	53975,1
ZZIP1	Gif	98,47%	331,3	808,6
PPM10_4	Gif	98,60%	180,0	170,2
ZZIP5	Gif	98,60%	314,8	600,3
PPM10_8	Gif	98,63%	180,9	170,3
ZZIP9	Gif	98,63%	322,6	471,6
BZIP5	Gif	98,81%	630,1	1701,4
BZIP9	Gif	98,81%	612,8	1664,2
PPM2_4	Gif	98,84%	270,5	245,5
BZIP1	Gif	98,91%	670,1	1981,4
LHA	Gif	99,01%	1985,8	7698,3
PPM1_4	Gif	99,34%	302,1	271,0
SARITHDB1	Gif	99,62%	1603,3	1513,1
AHUFFDB1	Gif	99,75%	1689,3	1727,7
LZDB3	Gif	99,99%	1899,3	24703,6
JAM	Gif	100,00%	970,8	15024,6
HA2	Gif	100,00%	37,0	6791,7
MDCD	Gif	100,00%	2153,8	65015,5
LZW	Gif	132,99%	2025,4	3481,0

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	Html	12,53%	116,1	118,9
RKMX1	Html	13,88%	136,9	137,0
PPM10_8	Html	15,15%	1540,4	1437,9
ZZIP9	Html	17,05%	367,1	1543,6
PPM10_4	Html	17,26%	2702,9	2456,3
BZIP9	Html	17,53%	1048,1	3495,7
ZZIP5	Html	17,53%	419,2	1542,3
PPM2_4	Html	17,65%	2774,5	2469,8
BZIP5	Html	17,93%	1114,3	3587,7
PPM1_4	Html	18,20%	2799,5	2453,3
ZZIP1	Html	20,64%	581,4	1688,0
HA2	Html	20,78%	403,9	423,1
BZIP1	Html	20,97%	1350,4	4343,7
RAR	Html	21,38%	2144,4	19483,2
HA1	Html	22,62%	1078,6	3303,9
GZIP9	Html	22,65%	3305,1	27954,1
GZIP5	Html	23,25%	5108,9	27592,0
ARJ1	Html	23,26%	2947,5	19472,0
LHA	Html	27,15%	1569,1	19483,2
GZIP1	Html	28,35%	7707,8	23713,4
LZOP	Html	35,28%	33572,5	103574,7
JAM	Html	37,09%	3008,4	3640,6
LZDB3	Html	37,20%	3972,5	17149,5
MDCD	Html	42,97%	5256,2	6266,3
LZW	Html	47,36%	4060,5	6105,7
SARITHDB1	Html	65,56%	2061,2	974,1
AHUFFDB1	Html	68,30%	2293,5	2407,2

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX1	Jpg	85,87%	45,6	43,5
RKMX2	Jpg	86,16%	36,6	36,3
RAR	Jpg	87,19%	1154,3	7884,6
PPM10_8	Jpg	87,71%	199,1	187,3
GZIP9	Jpg	87,71%	3073,7	13257,8
GZIP5	Jpg	87,76%	3363,1	13424,4
PPM10_4	Jpg	87,77%	192,5	181,7
PPM2_4	Jpg	88,30%	302,1	273,2
GZIP1	Jpg	88,45%	3630,8	10938,8
HA1	Jpg	88,52%	848,4	1091,2
ARJ1	Jpg	88,88%	2041,1	7653,9
ZZIP5	Jpg	88,89%	209,0	524,7
ZZIP9	Jpg	89,23%	219,1	518,0
BZIP5	Jpg	89,39%	658,5	1753,8
PPM1_4	Jpg	89,65%	338,1	302,0
BZIP9	Jpg	89,70%	640,0	1644,9
LZOP	Jpg	90,05%	8384,8	45462,0
BZIP1	Jpg	90,13%	698,8	2040,8
ZZIP1	Jpg	90,47%	254,8	549,2
HA2	Jpg	94,07%	39,3	41,5
LHA	Jpg	94,09%	1963,0	8207,6
SARITHDB1	Jpg	99,10%	1634,6	1516,8
LZDB3	Jpg	99,10%	2010,7	24008,0
AHUFFDB1	Jpg	99,53%	1701,9	1719,9
JAM	Jpg	100,00%	1009,2	14372,5
MDCD	Jpg	100,01%	2286,5	43166,0
LZW	Jpg	127,96%	2235,8	3492,3

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX1	Mp3	95,58%	44,6	41,5
ZZIP9	Mp3	95,61%	408,0	489,6
ZZIP5	Mp3	95,80%	411,1	492,7
RKMX2	Mp3	95,87%	33,7	31,6
BZIP9	Mp3	95,99%	656,6	1776,2
BZIP5	Mp3	96,14%	675,6	1808,8
ZZIP1	Mp3	96,39%	443,5	516,2
PPM10_4	Mp3	96,65%	173,0	161,8
PPM10_8	Mp3	96,67%	173,3	161,5
GZIP9	Mp3	96,78%	4117,3	19024,5
BZIP1	Mp3	96,84%	720,1	2109,8
GZIP5	Mp3	96,86%	4274,1	19217,0
RAR	Mp3	96,90%	1186,4	9490,3
ARJ1	Mp3	96,95%	2275,6	9577,9
GZIP1	Mp3	97,12%	4741,0	18903,0
LHA	Mp3	97,12%	2363,1	9672,4
HA2	Mp3	97,31%	35,8	37,7
PPM1_4	Mp3	97,42%	316,3	275,0
PPM2_4	Mp3	97,43%	278,2	246,1
LZOP	Mp3	98,27%	8637,6	70791,6
HA1	Mp3	98,32%	824,7	1035,5
SARITHDB1	Mp3	98,67%	1764,0	6853,8
LZDB3	Mp3	98,67%	2168,8	28102,7
AHUFFDB1	Mp3	99,42%	1892,3	1813,8
JAM	Mp3	100,00%	1022,3	21466,5
MDCD	Mp3	100,00%	2521,6	61106,6
LZW	Mp3	131,14%	2692,7	3740,2

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	MS access	11,91%	58,1	67,1
RKMX1	MS access	12,52%	72,6	74,7
PPM10_8	MS access	13,66%	1604,8	1482,7
BZIP9	MS access	15,71%	1556,8	4051,3
ZZIP9	MS access	16,02%	527,1	1829,4
PPM10_4	MS access	16,13%	2116,7	1903,4
ZZIP5	MS access	16,34%	562,9	1880,5
BZIP5	MS access	16,63%	1649,1	4159,1
PPM2_4	MS access	16,64%	2155,2	1895,5
PPM1_4	MS access	18,00%	2090,8	1843,6
RAR	MS access	19,68%	2070,2	10676,0
ZZIP1	MS access	20,50%	856,5	1967,6
BZIP1	MS access	20,70%	1708,9	4705,4
HA1	MS access	21,35%	545,3	2934,5
GZIP9	MS access	22,08%	1101,7	13832,5
HA2	MS access	22,41%	257,7	276,4
GZIP5	MS access	22,43%	4556,5	14207,1
ARJ1	MS access	22,75%	2450,0	10691,5
GZIP1	MS access	24,69%	5671,9	14207,1
LHA	MS access	24,95%	1836,7	13046,3
LZOP	MS access	29,76%	15175,7	36542,6
JAM	MS access	32,33%	2670,9	3760,9
LZDB3	MS access	35,40%	2670,9	15144,4
MDCD	MS access	36,48%	4565,0	5392,8
LZW	MS access	37,46%	3704,0	5477,3
SARITHDB1	MS access	50,39%	2384,8	945,6
AHUFFDB1	MS access	50,73%	2625,1	2973,7

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	MS Powerpoint	82,75%	38,0	36,3
RKMX1	MS Powerpoint	87,17%	47,5	44,7
RAR	MS Powerpoint	88,76%	1222,3	9856,4
GZIP9	MS Powerpoint	88,92%	2448,4	21276,0
GZIP5	MS Powerpoint	89,05%	3738,5	21281,8
ARJ1	MS Powerpoint	89,07%	2166,2	10040,8
ZZIP1	MS Powerpoint	89,21%	448,3	981,4
ZZIP5	MS Powerpoint	89,34%	393,5	698,8
LHA	MS Powerpoint	89,43%	2072,2	10124,4
PPM10_8	MS Powerpoint	89,46%	190,7	179,5
GZIP1	MS Powerpoint	89,48%	4025,8	20660,2
ZZIP9	MS Powerpoint	89,48%	398,2	606,3
BZIP1	MS Powerpoint	89,80%	716,0	2220,1
PPM1_4	MS Powerpoint	89,88%	341,5	303,8
HA1	MS Powerpoint	89,88%	734,1	1109,6
BZIP9	MS Powerpoint	89,91%	651,8	1846,4
BZIP5	MS Powerpoint	89,92%	674,0	1893,5
PPM10_4	MS Powerpoint	89,96%	188,5	177,9
PPM2_4	MS Powerpoint	90,18%	302,3	272,3
HA2	MS Powerpoint	90,54%	38,9	40,7
LZOP	MS Powerpoint	90,70%	7671,7	81557,8
LZDB3	MS Powerpoint	92,13%	2134,5	24898,0
SARITHDB1	MS Powerpoint	94,42%	1704,3	3202,7
AHUFFDB1	MS Powerpoint	97,75%	1802,2	1832,1
JAM	MS Powerpoint	99,05%	1104,7	2231,1
MDCD	MS Powerpoint	100,00%	2482,4	53401,6
LZW	MS Powerpoint	121,73%	2434,0	3911,8

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	MS Word document	8,51%	89,3	92,2
RKMX1	MS Word document	8,71%	102,6	101,9
PPM10_8	MS Word document	10,44%	2110,6	1887,7
ZZIP9	MS Word document	11,59%	444,6	2139,0
BZIP9	MS Word document	11,61%	831,8	5509,5
BZIP5	MS Word document	12,14%	1170,9	5595,0
ZZIP5	MS Word document	12,14%	476,6	2187,0
PPM10_4	MS Word document	12,70%	2374,7	2091,7
RAR	MS Word document	13,56%	3350,1	25981,6
PPM2_4	MS Word document	13,68%	2563,3	2201,1
PPM1_4	MS Word document	13,90%	2697,8	2299,2
BZIP1	MS Word document	15,03%	1561,7	6561,8
ZZIP1	MS Word document	15,80%	592,0	2292,1
HA1	MS Word document	16,66%	573,3	3681,4
GZIP9	MS Word document	16,98%	1046,1	33148,0
GZIP5	MS Word document	17,48%	5897,3	32575,9
ARJ1	MS Word document	18,22%	3020,3	22098,7
HA2	MS Word document	19,60%	305,0	326,3
GZIP1	MS Word document	19,69%	8486,4	29820,9
LHA	MS Word document	20,26%	2017,4	22821,7
LZOP	MS Word document	22,34%	32609,0	187644,7
JAM	MS Word document	27,70%	3515,6	4544,9
LZDB3	MS Word document	31,26%	3097,5	17688,7
MDCD	MS Word document	31,53%	5487,8	7055,2
LZW	MS Word document	31,85%	4604,9	6825,6
SARITHDB1	MS Word document	45,21%	2673,7	976,4
AHUFFDB1	MS Word document	49,78%	3059,7	3312,4

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX1	Pdf	85,21%	48,5	45,8
RKMX2	Pdf	85,42%	37,3	35,7
RAR	Pdf	87,09%	1236,8	10377,4
ZZIP9	Pdf	87,26%	70,9	612,5
PPM10_8	Pdf	87,32%	195,7	184,2
ZZIP5	Pdf	87,45%	120,6	693,2
PPM10_4	Pdf	87,57%	194,3	183,3
GZIP9	Pdf	87,84%	3697,1	22707,9
BZIP9	Pdf	87,86%	626,5	1831,3
BZIP5	Pdf	87,95%	663,4	1875,0
GZIP5	Pdf	87,95%	3947,9	22401,9
ARJ1	Pdf	88,16%	2279,5	10486,8
GZIP1	Pdf	88,57%	4228,1	22212,8
PPM2_4	Pdf	89,02%	309,5	277,5
HA1	Pdf	89,06%	856,8	1119,5
ZZIP1	Pdf	89,10%	294,0	867,5
LZOP	Pdf	89,37%	9347,0	69600,1
PPM1_4	Pdf	89,53%	342,5	304,0
BZIP1	Pdf	89,57%	712,8	2178,2
LHA	Pdf	91,58%	2109,8	10142,4
HA2	Pdf	92,32%	38,0	39,6
LZDB3	Pdf	95,54%	2138,9	27497,6
SARITHDB1	Pdf	97,40%	1670,1	2715,9
AHUFFDB1	Pdf	99,66%	1789,5	1809,9
JAM	Pdf	100,00%	1033,2	21284,6
MDCD	Pdf	100,00%	2408,4	67916,2
LZW	Pdf	126,38%	2368,9	3829,9

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	Postscript	29,06%	91,8	93,1
RKMX1	Postscript	29,61%	115,8	116,9
PPM10_8	Postscript	30,52%	971,5	911,2
ZZIP9	Postscript	31,69%	442,8	1148,5
ZZIP5	Postscript	31,80%	414,3	1164,5
PPM2_4	Postscript	32,01%	1548,8	1400,7
PPM10_4	Postscript	32,20%	1124,6	1068,7
BZIP9	Postscript	32,29%	1110,1	2984,6
BZIP5	Postscript	32,35%	1186,3	3063,1
PPM1_4	Postscript	32,41%	1706,2	1498,0
ZZIP1	Postscript	33,23%	683,1	1268,3
BZIP1	Postscript	33,72%	1408,4	3719,5
HA2	Postscript	33,93%	292,5	304,6
RAR	Postscript	35,86%	1462,6	13515,8
HA1	Postscript	36,85%	561,0	2304,2
GZIP9	Postscript	37,31%	1358,5	20687,5
ARJ1	Postscript	37,63%	2161,1	13448,1
GZIP5	Postscript	37,66%	3695,7	20400,7
LHA	Postscript	38,71%	1468,6	14752,9
GZIP1	Postscript	41,74%	5780,8	17470,7
JAM	Postscript	45,56%	2541,1	3714,7
MDCD	Postscript	48,62%	4914,4	6251,3
LZW	Postscript	50,89%	3776,8	6029,9
LZDB3	Postscript	52,17%	2623,0	19345,2
SARITHDB1	Postscript	56,64%	2247,4	969,6
LZOP	Postscript	57,55%	16876,0	69536,4
AHUFFDB1	Postscript	67,04%	2377,3	2499,5

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	Text	15,69%	102,6	101,8
RKMX1	Text	16,04%	129,8	128,7
PPM10_8	Text	17,63%	1421,1	1321,0
ZZIP9	Text	18,90%	727,6	1486,2
PPM10_4	Text	18,91%	2861,4	2557,9
PPM2_4	Text	19,39%	2910,5	2526,0
ZZIP5	Text	19,54%	844,2	1513,8
BZIP9	Text	19,55%	1254,1	3664,0
PPM1_4	Text	19,80%	2957,9	2514,8
BZIP5	Text	20,11%	1323,9	3775,3
ZZIP1	Text	21,91%	1040,3	1660,8
HA2	Text	22,39%	426,7	448,0
BZIP1	Text	22,41%	1590,5	4588,0
RAR	Text	23,95%	2113,7	21699,9
HA1	Text	24,88%	668,6	3219,2
GZIP9	Text	24,90%	1215,1	32335,6
ARJ1	Text	25,46%	2655,4	22914,8
GZIP5	Text	25,49%	5464,9	31436,4
LHA	Text	27,91%	1565,8	22015,2
GZIP1	Text	30,79%	10512,8	26044,6
JAM	Text	35,53%	3299,4	4043,0
LZDB3	Text	36,82%	3822,6	17876,3
LZOP	Text	39,10%	23551,3	75404,0
MDCD	Text	40,50%	5958,0	6843,4
LZW	Text	43,97%	4306,4	6493,1
SARITHDB1	Text	56,60%	2247,6	992,7
AHUFFDB1	Text	57,56%	2653,2	2825,0

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	Wav	60,49%	35,1	34,8
RKMX1	Wav	61,43%	46,0	44,2
PPM10_4	Wav	69,78%	365,3	339,5
ZZIP5	Wav	70,43%	22,4	693,2
PPM2_4	Wav	71,15%	477,9	429,4
PPM10_8	Wav	71,20%	363,6	336,6
ZZIP9	Wav	71,37%	22,0	679,4
BZIP5	Wav	71,70%	826,2	2098,5
BZIP9	Wav	72,27%	796,9	2015,9
ZZIP1	Wav	72,98%	495,8	728,7
PPM1_4	Wav	73,06%	520,1	456,6
BZIP1	Wav	73,38%	997,5	2462,2
HA2	Wav	74,50%	61,5	65,9
HA1	Wav	77,31%	695,1	1194,8
RAR	Wav	77,60%	1173,8	8078,7
ARJ1	Wav	78,20%	1984,7	8469,8
LHA	Wav	78,82%	1621,7	9229,7
GZIP9	Wav	79,06%	2339,4	13927,7
GZIP5	Wav	79,16%	3202,8	13681,3
GZIP1	Wav	79,81%	3786,7	12790,6
SARITHDB1	Wav	85,00%	1788,1	999,4
JAM	Wav	85,05%	1137,1	2535,7
LZDB3	Wav	90,55%	2121,8	21306,3
LZOP	Wav	90,94%	10230,3	61147,0
AHUFFDB1	Wav	92,57%	1853,2	1891,7
MDCD	Wav	98,94%	2645,7	4487,0
LZW	Wav	99,59%	2226,6	4203,7

Algorithm	**Data type**	r	B_c	B_d
			Kbyte/s	Kbyte/s
RKMX2	Xls	17,63%	66,4	70,9
RKMX1	Xls	17,65%	79,1	79,1
ZZIP9	Xls	20,77%	821,0	1441,7
ZZIP5	Xls	20,77%	852,7	1479,6
PPM10_8	Xls	20,91%	1219,6	1109,9
ZZIP1	Xls	20,95%	1043,9	1672,0
RAR	Xls	21,53%	2082,0	16080,5
BZIP1	Xls	21,91%	1463,9	4565,5
BZIP5	Xls	22,26%	1435,2	3878,3
PPM1_4	Xls	22,35%	1896,1	1669,4
BZIP9	Xls	22,41%	1357,6	3698,6
PPM2_4	Xls	22,66%	1775,6	1575,5
HA1	Xls	23,62%	333,9	2843,9
PPM10_4	Xls	23,72%	1461,9	1317,6
GZIP9	Xls	24,68%	737,0	19816,2
ARJ1	Xls	24,73%	1975,6	13755,6
GZIP5	Xls	25,07%	4461,2	17968,9
LHA	Xls	26,03%	1772,6	16400,4
HA2	Xls	26,70%	224,6	249,3
GZIP1	Xls	28,20%	6731,2	18301,3
JAM	Xls	34,29%	2553,7	4112,6
LZOP	Xls	35,44%	23227,4	55381,8
LZDB3	Xls	36,00%	2905,1	16982,1
MDCD	Xls	36,62%	4608,0	5939,3
LZW	Xls	36,95%	4012,5	6166,3
SARITHDB1	Xls	54,70%	2305,8	925,8
AHUFFDB1	Xls	58,85%	2410,6	2753,2

Appendix B

Compression implementations

Name	Category	Author	Description
AHUFFDB1	Statis-0-Adap-Huff	David Broman	Order-0 model with adaptive Huffman
ARJ1	Dic-Lz77-Semi-Huff	Robert Jung	LZSS followed by semi-adaptive Huffman
BZIP1	Block-BWT-100-Huff	Julian Seward	BWT together with semi-adaptive Huffman, 100Kbyte block
BZIP5	Block-BWT-100-Huff	Julian Seward	BWT together with semi-adaptive Huffman, 500Kbyte block
BZIP9	Block-BWT-100-Huff	Julian Seward	BWT together with semi-adaptive Huffman, 900Kbyte block
GZIP1	Dic-Lz77-Semi-Huff	Jean-loup Gailly	LZ77 variant with semi-adaptive huffman. Some searching in hash chains. No lazy match evaluation.
GZIP5	Dic-Lz77-Semi-Huff	Jean-loup Gailly	LZ77 variant with semi-adaptive huffman. More searching in hash chains. Lazy match evaluation.
GZIP9	Dic-Lz77-Semi-Huff	Jean-loup Gailly	LZ77 variant with semi-adaptive huffman. The most searching in hash chains. Lazy match evaluation.

Name	Category	Author	Description
HA1	Dic-Lz77-Adap-Arith	Harri Hirvola	LZ77 variant with adaptive arithmetic coding
HA2	Statis-4-Adap-Arith	Harri Hirvola	Order-4 context-model followed by adaptive arithmetic coding
JAM	Dic-Lz78-Fixed	W. Jiang	Variant of LZ78
LHA	Dic-Lz77-Semi-Huff	Haruyasu Yoshizaki	LZSS followed by semi-adaptive Huffman encoding.
LZDB3	Dic-Lz77-Fixed	David Broman	Variant of LZSS using a 4096 bytes long sliding window.
LZOP	Dic-Lz77-Fixed	Markus Oberhumer	LZO, which is probably a variant of LZSS.
LZW	Dic-Lz78-Fixed	David Bourgin	LZW implementation with 4096 entries in dictionary
MDCD	Dic-Lz78-Fixed	Mike Devenport	LZW implementation with 8192 entries in dictionary
PPM1_4	Statis-4-Adap-Range	Dmitry Shkarin	Order-4 finite-context model followed by range coder. Uses 1 Mbyte memory.
PPM10_4	Statis-4-Adap-Range	Dmitry Shkarin	Order-4 finite-context model followed by range coder. Uses 10 Mbyte memory.
PPM10_8	Statis-8-Adap-Range	Dmitry Shkarin	Order-8 finite-context model followed by range coder. Uses 10 Mbyte memory.
PPM2_4	Statis-4-Adap-Range	Dmitry Shkarin	Order-4 finite-context model followed by range coder. Uses 2 Mbyte memory.

Name	**Category**	**Author**	**Description**
RAR	Dic-Lz77-Semi-Huff	Eugene Roshal	LZSS variant followed by semi-adaptive Huffman coding
RKMX1	Statis-?-Adap-Arith	Malcom Taylor	PPMZ with arithmetic encoding.
RKMX2	Statis-?-Adap-Arith	Malcom Taylor	PPMZ with arithmetic encoding. Best compression
SARITHDB1	Statis-0-Semi-Arith	David Broman	Order-0 model with semi-adaptive arithmetic coding
ZZIP1	Block-BWT-100-Arith	Damien Debin	RLE, BWT, MTF and artihmetic coding. 100 Kbytes block.
ZZIP5	Block-BWT-100-Arith	Damien Debin	RLE, BWT, MTF and artihmetic coding. 500 Kbytes block.
ZZIP9	Block-BWT-100-Arith	Damien Debin	RLE, BWT, MTF and artihmetic coding. 900 Kbytes block.

Appendix C

Compression ratio diagrams

The diagrams illustrated in this section show the compression ratio on the y-axle and the different implementations on the x-axle. All implementations are shown in the same order in the diagrams. The order of the implementations is given in table C.1.

Nr	**Name**	**Nr**	**Name**	**Nr**	**Name**
1	AHUFFDB1	10	ARJ1	19	BZIP5
2	SARITHDB1	11	GZIP5	20	PPM2_4
3	LZW	12	GZIP9	21	ZZIP5
4	MDCD	13	HA1	22	BZIP9
5	LZDB3	14	RAR	23	PPM10_4
6	JAM	15	BZIP1	24	ZZIP9
7	LZOP	16	HA2	25	PPM10_8
8	GZIP1	17	ZZIP1	26	RKMX1
9	LHA	18	PPM1_4	27	RKMX2

Table C.1: Order of implementations

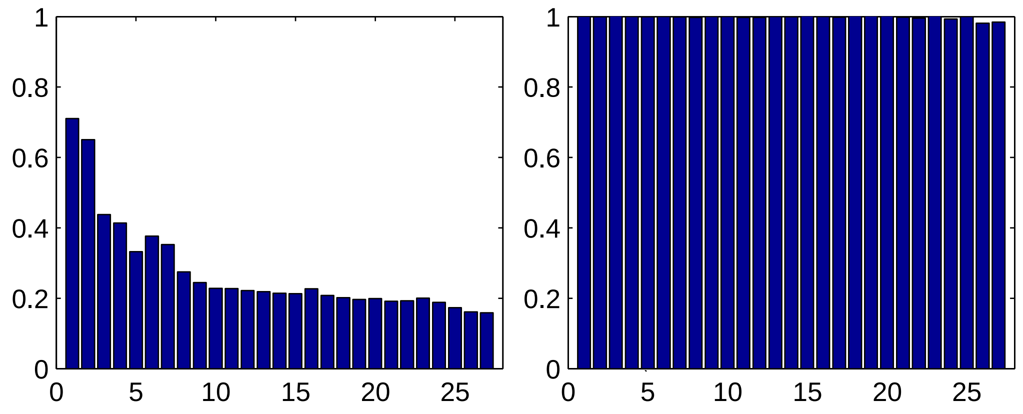


Figure C.1: Left: C++ source code, Right: Compressed files

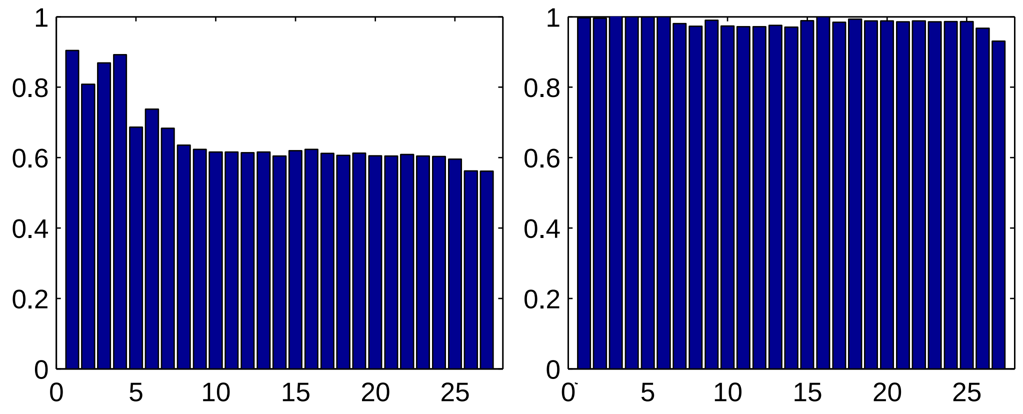


Figure C.2: Left: Executables, Right: Gif

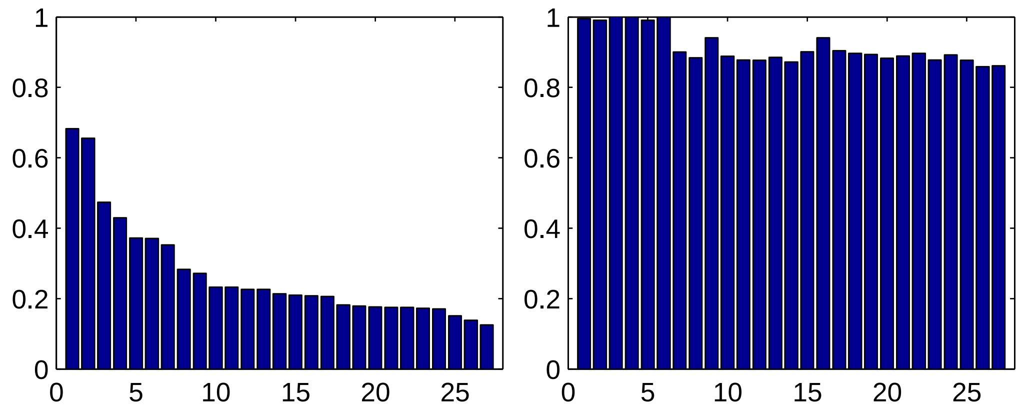


Figure C.3: Left: Html, Right: Jpg

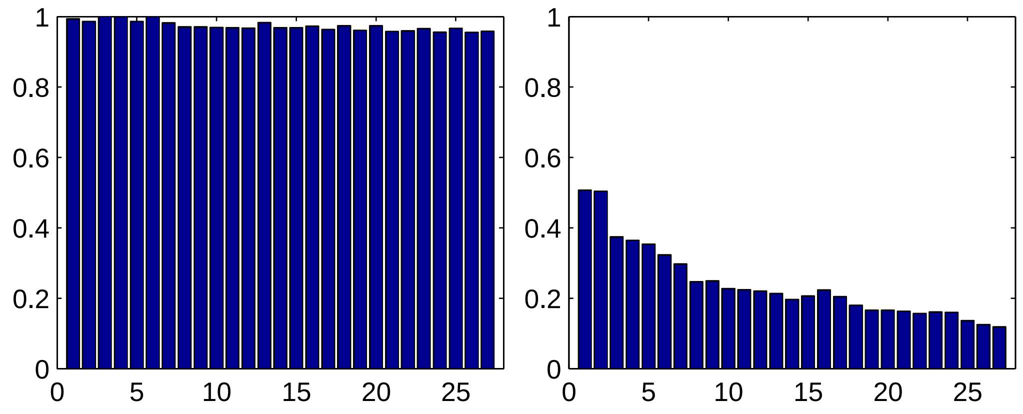


Figure C.4: Left: Mp3, Right: MS access

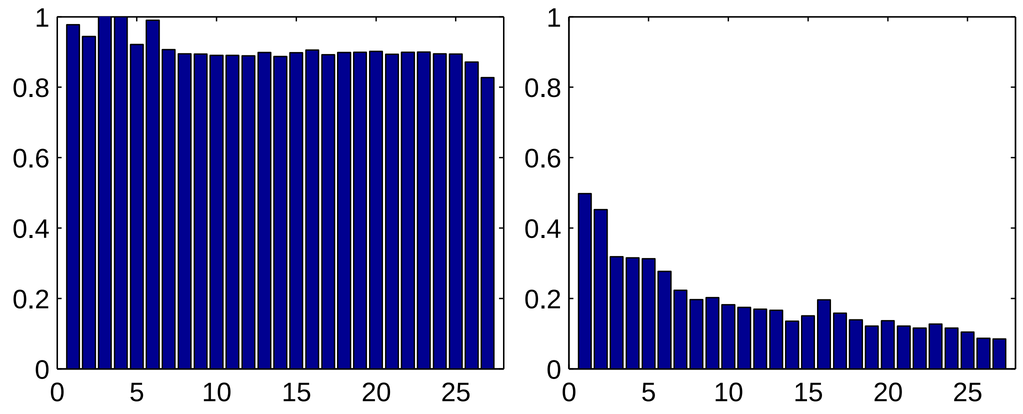


Figure C.5: Left: MS Powerpoint, Right: MS Word document

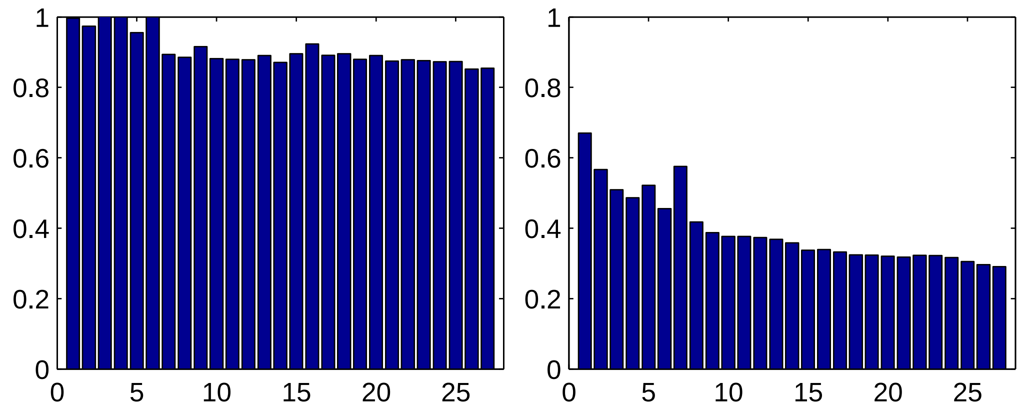
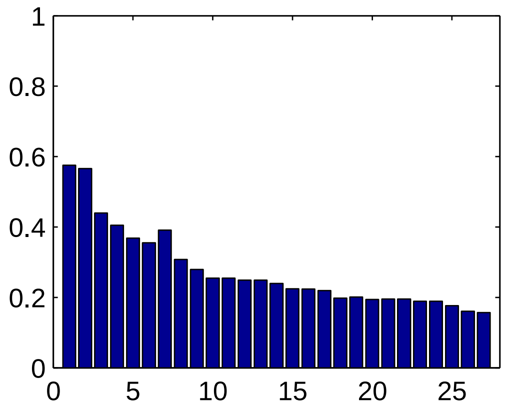


Figure C.6: Left: Pdf, Right: Postscript



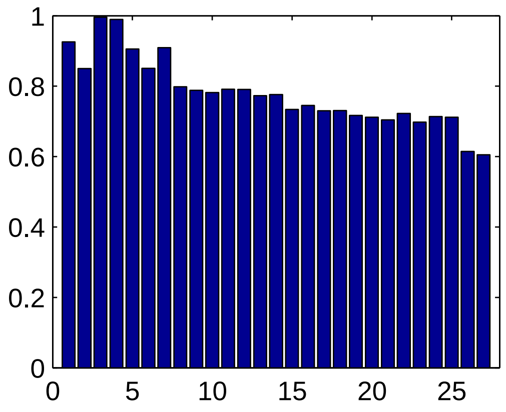


Figure C.7: Left: Text, Right: Wav

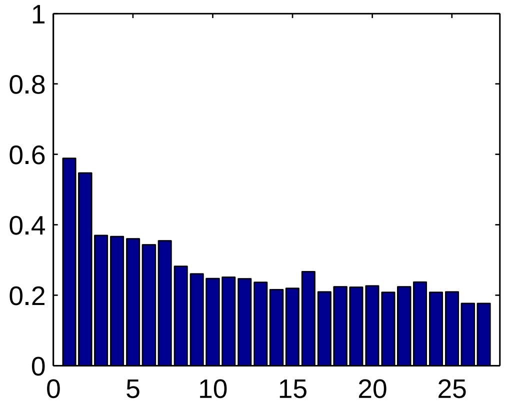


Figure C.8: MS Excel

Appendix D

Compression and decompression bandwidths

On the first two pages in this chapter, the compression and decompression bandwidths are given. The algorithms are listed in sorted order after the average bandwidth for different documents. To show the difference of bandwidth for different document types, the maximal and minimal bandwidth are given, and the standard deviation of the documents.

The third table shows the average decompression bandwidth on different client devices. The forth and last table shows the average compression bandwidths on the server. Each column points out the number of simulations users. The computer used is the reference computer in this experiment.

Alg	**Category**	**Average**	**Max**	**Min**	**Deviation**
		Kbyte/s	Kbyte/s	Kbyte/s	Kbyte/s
LZOP	Dic-Lz77-Fixed	16151	33572	7672	8420
GZIP1	Dic-Lz77-Semi-Huff	5756	10513	3389	2020
GZIP5	Dic-Lz77-Semi-Huff	4325	5897	3202	822
MDCD	Dic-Lz78-Fixed	3719	5958	2154	1314
LZW	Dic-Lz78-Fixed	3149	4605	2025	853
LZDB3	Dic-Lz77-Fixed	2679	4034	1899	621
ARJ1	Dic-Lz77-Semi-Huff	2628	6862	1976	1171
GZIP9	Dic-Lz77-Semi-Huff	2343	4117	737	1057
AHUFFDB1	Statis-0-Adap-Huff	2131	3060	1650	411
SARITHDB1	Statis-0-Semi-Arith	1977	2674	1603	322
JAM	Dic-Lz78-Fixed	1942	3516	971	915
LHA	Dic-Lz77-Semi-Huff	1853	2363	1469	233
RAR	Dic-Lz77-Semi-Huff	1668	3350	1091	601
PPM1_4	Statis-4-Adap-Range	1329	2958	294	997
PPM2_4	Statis-4-Adap-Range	1274	2911	264	988
PPM10_4	Statis-4-Adap-Range	1128	2861	166	986
BZIP1	Block-BWT-100-Huff	1104	1709	646	377
BZIP5	Block-BWT-500-Huff	980	1649	608	321
BZIP9	Block-BWT-900-Huff	910	1557	590	294
PPM10_8	Statis-8-Adap-Range	822	2111	167	642
HA1	Dic-Lz77-Semi-Arith	719	1079	334	165
ZZIP1	Block-BWT-100-Arith	582	1044	255	236
ZZIP5	Block-BWT-500-Arith	431	853	22	216
ZZIP9	Block-BWT-900-Arith	401	821	22	200
HA2	Statis-4-Adap-Arith	176	427	34	140
RKMX1	Statis-?-Adap-Arith	77	142	44	32
RKMX2	Statis-?-Adap-Arith	63	118	34	27

Table D.1: Algorithms sorted after compression bandwidth

Alg	**Category**	**Average**	**Max**	**Min**	**Deviation**
		Kbyte/s	Kbyte/s	Kbyte/s	Kbyte/s
LZOP	Dic-Lz77-Fixed	74498	187645	36543	33786
MDCD	Dic-Lz78-Fixed	26441	67916	4487	25110
LZDB3	Dic-Lz77-Fixed	21216	28103	15144	3910
GZIP9	Dic-Lz77-Semi-Huff	20693	33148	9163	6763
GZIP5	Dic-Lz77-Semi-Huff	20682	32576	13424	5969
GZIP1	Dic-Lz77-Semi-Huff	19353	29821	10939	4974
LHA	Dic-Lz77-Semi-Huff	15335	35287	7698	7111
ARJ1	Dic-Lz77-Semi-Huff	13301	22915	7109	4772
RAR	Dic-Lz77-Semi-Huff	13161	25982	7206	5405
JAM	Dic-Lz78-Fixed	8229	21467	2231	6856
LZW	Dic-Lz78-Fixed	4936	6826	3449	1182
SARITHDB1	Statis-0-Semi-Arith	3481	27018	926	6433
BZIP1	BWT-Huff	3390	6562	1981	1340
BZIP5	BWT-Huff	2875	5595	1674	1135
BZIP9	BWT-Huff	2789	5509	1621	1114
HA1	Dic-Lz77-Semi-Arith	2430	6853	997	1516
AHUFFDB1	Statis-0-Adap-Huff	2240	3312	1681	513
ZZIP1	BWT-Arith	1463	3586	516	772
ZZIP5	BWT-Arith	1176	2187	493	506
PPM1_4	Statis-4-Adap-Range	1153	2515	262	859
PPM2_4	Statis-4-Adap-Range	1120	2526	239	862
HA2	Statis-4-Adap-Arith	1099	6976	38	2266
ZZIP9	BWT-Arith	1087	2139	472	510
PPM10_4	Statis-4-Adap-Range	1019	2558	157	881
PPM10_8	Statis-8-Adap-Range	757	1888	157	585
RKMX1	Statis-?-Adap-Arith	76	142	42	33
RKMX2	Statis-?-Adap-Arith	64	124	32	29

Table D.2: Algorithms sorted after decompression bandwidth

Implementation	**Palm**	**Slow Pocket PC**	**Fast Pocket PC**	**Laptop**
	Kbyte/s	Kbyte/s	Kbyte/s	Kbyte/s
LZOP	74	1656	6773	49665
MDCD	26	588	2404	17627
LZDB3	21	471	1929	14144
GZIP9	21	460	1881	13795
GZIP5	21	460	1880	13788
GZIP1	19	430	1759	12902
LHA	15	341	1394	10223
ARJ1	13	296	1209	8867
RAR	13	292	1196	8774
JAM	8	183	748	5486
LZW	5	110	449	3291
SARITHDB1	3	77	316	2320
BZIP1	3	75	308	2260
BZIP5	3	64	261	1917
BZIP9	3	62	254	1859
HA1	2	54	221	1620
AHUFFDB1	2	50	204	1493
ZZIP1	1	33	133	976
ZZIP5	1	26	107	784
PPM1_4	1	26	105	768
PPM2_4	1	25	102	747
HA2	1	24	100	733
ZZIP9	1	24	99	725
PPM10_4	1	23	93	679
PPM10_8	1	17	69	505
RKMX1	0	2	7	51
RKMX2	0	1	6	42

Table D.3: Decompression bandwidths for different devices

Implementation	**50 user**	**10 users**	**100 users**	**1000 users**
	Kbyte/s	Kbyte/s	Kbyte/s	Kbyte/s
LZOP	16151	323	162	16
GZIP1	5756	115	58	6
GZIP5	4325	87	43	4
MDCD	3719	74	37	4
LZW	3149	63	31	3
LZDB3	2679	54	27	3
ARJ1	2628	53	26	3
GZIP9	2343	47	23	2
AHUFFDB1	2131	43	21	2
SARITHDB1	1977	40	20	2
JAM	1942	39	19	2
LHA	1853	37	19	2
RAR	1668	33	17	2
PPM1_4	1329	27	13	1
PPM2_4	1274	25	13	1
PPM10_4	1128	23	11	1
BZIP1	1104	22	11	1
BZIP5	980	20	10	1
BZIP9	910	18	9	1
PPM10_8	822	16	8	1
HA1	719	14	7	1
ZZIP1	582	12	6	1
ZZIP5	431	9	4	0
ZZIP9	401	8	4	0
HA2	176	4	2	0
RKMX1	77	2	1	0
RKMX2	63	1	1	0

Table D.4: Compression bandwidths for different simultaneous users