Using semantic knowledge to improve compression on log files

Submitted in fulfilment
of the requirements of the degree
Master of Science
at Rhodes University

Frederick John Otten

June 2008

Abstract

With the move towards global and multi-national companies, information technology infrastructure requirements are increasing. As the size of these computer networks increases, it becomes more and more difficult to monitor, control, and secure them. Networks consist of a number of diverse devices, sensors, and gateways which are often spread over large geographical areas. Each of these devices produce log files which need to be analysed and monitored to provide network security and satisfy regulations. Data compression programs such as `gzip` and `bzip2` are commonly used to reduce the quantity of data for archival purposes after the log files have been rotated. However, there are many other compression programs which exist - each with their own advantages and disadvantages. These programs each use a different amount of memory and take different compression and decompression times to achieve different compression ratios. System log files also contain redundancy which is not necessarily exploited by standard compression programs. Log messages usually use a similar format with a defined syntax. In the log files, all the ASCII characters are not used and the messages contain certain "phrases" which often repeated. This thesis investigates the use of compression as a means of data reduction and how the use of semantic knowledge can improve data compression (also applying results to different scenarios that can occur in a distributed computing environment). It presents the results of a series of tests performed on different log files. It also examines the semantic knowledge which exists in maillog files and how it can be exploited to improve the compression results. The results from a series of text preprocessors which exploit this knowledge are presented and evaluated. These preprocessors include: one which replaces the timestamps and IP addresses with their binary equivalents and one which replaces words from a dictionary with unused ASCII characters. In this thesis, data compression is shown to be an effective method of data reduction producing up to 98 percent reduction in filesize on a corpus of log files. The use of preprocessors which exploit semantic knowledge results in up to 56 percent improvement in overall compression time and up to 32 percent reduction in compressed size.

Acknowledgements

Firstly I would like to thank God “For from him and through him and to him are ALL things” (Romans 11:36 [NIV], Emphasis added). My life is a testimony of his mercy and grace. I would also like to thank my family, friends and church (His People Grahamstown) for their support and encouragement as I have travelled along this long road.

I would like to specially thank my supervisors, Barry Irwin and Hannah Thinyane, for the countless hours they have spent helping me and for their wisdom and direction. I would also like to thank all the people in the Computer Science Department, particularly my peers Mosiuoa Tsietsi, J-P van Riel, Thamsanqa Moyo, Shaun Miles, Kevin Glass, Mamello Thinyane, Nyasha Chigwamba, Osedum Igumbor and John Richter, for their advice, ideas and encouragement during the course of this degree. They have been very valuable and helpful.

I would finally like to thank Telkom SA for their generous sponsorship over the course of my studies.

The Albany Schools Network is acknowledged for the provision of the log files. The sponsors of the Center of Excellence at Rhodes University (Telkom SA, Business Connexion, Comverse, Verso Technologies, THRIP, Stortech, Tellabs, Mars Technologies, Amatole Telecommunication Services, Bright Ideas 39, Open Voice and the National Research Foundation) are also acknowledged.

Contents

List of Figures

List of Tables

List of Algorithms

Glossary Of Terms

A.1: Word replacement preprocessor which uses a dictionary built from the file using word definition A and word score 1 (these are defined in Section 5.1.1 and Section 5.1.2 respectively).

A.2: Word replacement preprocessor which uses a dictionary built from the file using word definition A and word score 2 (these are defined in Section 5.1.1 and Section 5.1.2 respectively).

All: All is a preprocessor which uses a dictionary constructed using the whole corpus of maillogs.

ASCII: American Standard Code for Information Interchange (ASCII) is a standard used for character encoding in PCs. There are 256 different ASCII characters codes available which represent letters, new line characters, and punctuation marks.

B.1: Word replacement preprocessor which uses a dictionary built from the file using word definition B and word score 1 (these are defined in Section 5.1.1 and Section 5.1.2 respectively).

B.2: Word replacement preprocessor which uses a dictionary built from the file using word definition B and word score 2 (these are defined in Section 5.1.1 and Section 5.1.2 respectively).

BWCA: The Burrows and Wheeler Compression Algorithm (BWCA) is a block sorting compression algorithm. It is described in Section 2.3.3.

BWT: The Burrow-Wheeler Transform (BWT) is the central part of the BWCA. It computes a permutation of the input sequence which can easily be compressed by simple compression algorithms such as Move-To-Front coding.

C.1: Word replacement preprocessor which uses a dictionary built from the file using word definition C and word score 1 (these are defined in Section 5.1.1 and Section 5.1.2 respectively).

C.2: Word replacement preprocessor which uses a dictionary built from the file using word definition C and word score 2 (these are defined in Section 5.1.1 and Section 5.1.2 respectively).

CombNoNum: CombNoNum refers a preprocessor (or the dictionary) which uses a dictionary created by combining the dictionaries for C.2 but excluding all the 2 digit numbers.

CombNum: CombNum refers a preprocessor (or the dictionary) which uses a dictionary created by combining the dictionaries for C.2.

DEFLATE: DEFLATE is a compression algorithm which combines LZ77 and Huffman coding.

LZ77: LZ77 is a sequential data compression algorithm which replaces reoccuring text with reference to previous text. It is described in Section 2.3.1.

LZ78: LZ78 is the successor to LZ77. It scans forward in the input data. It is described in Section 2.3.1.

LZC: LZC is the implementation of LZW used by the compress program. It is free from the patents associated with LZW.

LZMA: Lempel-Ziv Markov Chain Algorithm (LZMA) is an LZ77-based compression algorithm which also uses Markov models and Range coding. It is used by 7zip for compression.

LZO: Lemple-Ziv-Oberhumer (LZO) is a family of fast compression algorithms based on LZ78.

LZW: LZW is a sequential compression program which builds a string table and replaces strings with references to the string table. Further information can be found in Section 2.3.1.

MTF: Move To Front (MTF) is a technique for rearranging a list so that recent characters occur at the begining of the list.

MyCust: MyCust is a preprocessor which uses MyCustD for word replacement.

MyCust2: MyCust2 is a preprocessor which uses MyCust2D for word replacement.

MyCust2D: MyCust2D is the dictionary based on MyCustD but includes changes based on an anylsis of the context in which words occur.

MyCustD: MyCustD is the dictionary built using the semantic knowlege derived from the analysis of maillog files.

MyPhrase: MyPhrase is a preprocessor which uses MyPhraseD for replacement.

MyPhraseD: MyPhraseD is a dictionary based on MyCust2D but includes phrases as well as words.

PM: PM is a preprocessor which performs word replacement using a dictionary created from the previous month's log file.

PMU: PMU is a preprocessor which performs word replacement using a dictionary created from the previous month's log file. The word for the month is, however updated to the current month.

PPM: Prediction by Partial Matching (PPM) is a technique for determining the probability of the next character based on previous occurances. This technique is described in Section 2.3.2.

PPMII: Prediction by Partial Matching with Information Inheritance (PPMII) is a PPM-based algorithm used by ppmd. It is described in Section 2.3.2.

RLE: Run length encoding (RLE) is a compression technique which replaces runs on characters with a count and a single character.

T: Preprocessor which converts the timestamps contained in the file to their binary representations.

T&IP: Preprocessor which converts the timestamps and IP addresses contained in the file to their binary representations.

Chapter 1

Introduction

Modern computer networks consist of a myriad of interconnected, heterogeneous devices in different physical locations (this is termed a distributed computing environment). Computer networks are used by many different sized corporations, educational institutions, small businesses and research organisations. In many of these networks, monitoring is required to determine the system state, improve security or gather information. Examples of such scenarios are: large global corporate networks, which consist of many different mail servers, web servers, file servers and other machines spanning different countries which need to be monitored to protect valuable information; wireless sensor networks set up to monitor the movement of a species of animals; and distributed mail servers which log to a central location. In these scenarios, large quantities of information need to be gathered for analytical or archival purposes. This requires the reduction of the quantity of data to minimise bandwidth utilisation, maximise throughput and reduce storage space requirements. This chapter introduces and motivates the need for data reduction. It also presents the problem statement, the approach taken and the structure of this document.

1.1 Motivation

This section motivates the need for data reduction and explains how data compression can be used to reduce the quantity of data. It also discusses the need for network monitoring, the approaches which can be used and the role of log files.

1.1.1 Network Monitoring

The growth of the Internet has resulted in a growth in the number and severity of threats to its users [1]. This creates a large problem for global and multi-national companies which have networks spread throughout the globe. These networks are under severe threat from sophisticated worms, viruses and targeted attacks - such as Distributed Denial of Service (DDoS) attacks - which can lead to downtime, loss of information and attackers obtaining important confidential information (such as credit card details, account numbers and passwords). Two examples of the severity of these attacks are: The theft of 1.6 million entries of personal details from the website monster.com [2]; and an undisclosed amount of data being stolen from the German government [3]. The vulnerability and economic value of data necessitates the need for monitoring facilities which can detect events before damage is done. The early diagnosis and response to a situation potentially saves the life of the system, and protects the data and information involved.

Providing effective network security requires that the confidentiality, integrity and availability (CIA) of the data, information and knowledge contained within the network is maintained [4]. An understanding and overview of the network, its users, and its operation is therefore key to being able to keep the network secure.

Situational awareness can be described broadly as a person's state of knowledge or mental model of the situation around him or her [5]. Cognitive Psychology defines three levels of situational awareness [5]:

1. Perceiving critical factors in the environment
2. Understanding what those factors mean, particularly when integrated together in relation to the decision maker's goals
3. Understanding what will happen with the system in the near future.

Essentially, network administrators require at least the first level of situational awareness in order to secure the network. For this to be possible, data from a myriad of heterogeneous devices in the network needs to be monitored. The lack of information from important devices in the network can lead to dangerous blind spots in the network [6] and hence ineffective network monitoring. "The phrase 'forewarned is forearmed' sums up the value situational awareness. Simply put, being aware is about being prepared to act and respond" [6]. The information which is required for effective network monitoring is found in the log files of the devices contained in the network [7].

Regulations such as the Health Insurance Portability and Accountability Act (HIPAA) [8] and the Sarbanes Oxley Act [9] in the United States of America, and the Electronic Communications and Transactions (ECT) Act [10] in South Africa, also require that the logs of widely-used services are kept and monitored for accountability and identity purposes [11]. This places an increased burden on the storage requirements for previous log files.

For many businesses, the most important requirement is to keep the network running without downtime, with little regard to the risk level [6]. Most of the time, security administrators spend their time deploying patches to fill security holes and performing investigations into previous incidents. Current best practices for Internet security rely on word-of-mouth reports of new intrusions and security holes through entities such as Computer Emergency Response Team (CERT) and DSHIELD [1]. The monitoring of logs is not a high priority and is often left to Perl scripts using regular expressions which are written hastily after CIA has already been violated [12]. The monitoring of logs is essential for effective network monitoring [7]. System logs, web server logs, traffic captures and call logs all contain valuable information about different systems.

While network security is a large reason for network monitoring, network monitoring needs to be performed for a number of purposes including: ensuring proper use and compliance to set policies [11]; satisfying regulations which require accountability, monitoring and storage of records; getting information about the users and use of a service for competitive gain [13]; and attaining usage information so that the users may be billed appropriately. This information is also contained in log files.

1.1.2 Distributed and Centralised monitoring

A good case can be made for the use of both centralised and distributed monitoring. The choice largely depends on the size, the number of administrators employed, the security needs of the business or organisation and criteria such as: the quantity of data involved; the available infrastructure; the need for situational awareness; and the regulations to which the business or organisation needs to adhere.

The previous section established that there is plenty of data and information distributed throughout the network which needs to be monitored. Providing network security and maintaining CIA involves an analysis of the large amounts of information generated by different devices dispersed throughout the network. The goal is to be able to use as much of this data and information to obtain better information and knowledge, which can then be used to provide better security and improve business processes.

For most networks and businesses, the most important requirement is to keep the network running without downtime, with little regard to the risk level, at a minimum cost [11]. The vast quantity of data and information means that compiling the resources necessary to review the data being received from these heterogeneous systems is an arduous task. There are millions of alerts and messages generated by each individual system (i.e. systems such as the intrusion detection systems, anti-virus systems, firewalls, operating systems and access control systems) [6] which need to monitored.

Employing a distributed approach results in a greater workload for more administrators. In large networks, this approach can lead to sections of the network being overlooked, creating holes in the perimeter for intruders to exploit. Most of the devices spread throughout the perimeter of the network display related activity and complement each other in analysis. In a distributed approach, administrators work in isolation and do not see the bigger picture of the network. Collaboration is possible through meetings, reports and email; however, this does not give them a real-time picture with information related to the event from the entire network at their disposal. Administrators are also kept busy securing their segment of the network and therefore have little time for meetings.

A centralised approach can reduce the redundancy by correlating multiple events into a single event. "Consolidating all of the reports from all of these devices and tying the information together into a coherent visual artifact closes the window of risk" [6]. A centralised approach, however, is not always practical and thus collaboration using email and other means generally suffices - particularly for small organisations with few administrators. For a centralised approach, new infrastructures need to be put into place and off-site redundancy needs to be provided to avoid the fatal results should the central facility go down. An example of such a situation would be when the World Trade Center in New York, which contained many important servers, was attacked by terrorists on 11 September 2001 [14]. Therefore, although the new infrastructure would cost the organisation millions of dollars, the long-term benefits and the value of the information being protected must be considered.

Current security methodologies place a large burden on security administrators. They rarely have time to monitor logs, and spend most of the time trying to repair and defend against attacks which have already occurred, rather than monitoring and researching to prevent attacks. The attack implications, particularly to large organisations, are drastic. Denial of Service (DoS) leads to a large loss of revenue and organisations could be sued if customer information is obtained through an attack. Centralisation results in fewer administrators who achieve more, providing a better utilisation of resources. It also makes automation easier, as management and control can

then be deployed from the central location [11].

Essentially then, the quality of network administrator's work is improved when using a centralised approach. This improvement is necessary since the frequency and complexity of attacks are increasing, as new technologies are developed and there is an expansion of networks and services. With the added pressure of regulatory compliance, and the pressures of security audits, network administrators are too busy to use inefficient systems and spending time doing tasks which, ideally, should be automated and centralised. It helps to have a central point where a few administrators can view the entire network and concentrate on maintaining the CIA within the network. Network administrators can then be assigned jobs that are more worthy of their talents, such as diagnosing anomalies and creating prevention schemes [15].

Centralised logs open the possibility of data mining to find new attacks and to detect anomalies. Statistics can also be generated regarding what constitutes "normal" activity on the network. The trends that are found during analysis may be utilised to further improve policies. Trends such as the cessation of events which normally occur with a given frequency, may be overlooked by a human being. These trends can be found using data mining and then further investigated [15]. The information retrieved from logs is also useful for researchers in their search to define "normal activity", which is still an open and difficult research problem [16]. The more data that can be gathered and correlated, the more accurate intelligence one then has to mitigate and resolve the event [6]. Forensic and historical data provide maps of past activity. An analysis of these maps gives a better picture of the attack, it's operation and the path along which it travelled. This may lead to the discovery of defense strategies [6]. These maps can also be used as evidence for legal proceedings against attackers and in strategic planning.

As presented above, there are many reasons for a centralised approach to network monitoring. In a distributed monitoring approach, a centralised console which can perform analysis on the activity across the network is still desired. Essentially, this centralised console is merely a node on the network which collects data and performs analysis or stores the results. The primary problem with the centralised approach is the large volume of data which needs to be sent to this point. The quantity, the value, the amount of detail and the number of systems involved are all increasing with business needs. This is leading to an explosive growth in the amount of data and information that needs to be monitored. In order to use a centralised approach, then, the quantity of data needs to be addressed. Even in distributed scenarios where a centralised point is not in use, it is necessary for log data to be stored at each point for security purposes, to use in forensics and to satisfy regulations.

1.1.3 Dealing with the quantity of information

The problem regarding the quantity of data is compounded by the explosive growth in the amount of data and information that is involved. The quantity, value, amount of detail and the number of systems involved are all increasing with business needs. This is leading to new systems which need to be monitored as well as increased activity which results in more data to be monitored. One solution for dealing with the quantity of data is to send samples or summaries to the central point. As mentioned in Section 1.1.1, archives need to be kept for log files associated with widely-used network services to satisfy regulations (whether using a centralised or distributed approach). Samples and summaries will not comply with regulations if the original data is not archived [13]. Samples and summaries are also not guaranteed to provide an accurate view of the network state and are not as useful for statistical analysis to find new trends since they discard certain "unimportant" data. These archives can take up a lot of space and so need to be reduced in size. Due to the costs of bandwidth and the quantity of traffic generated for network monitoring, steps need to be taken to reduce the quantity of data to be transferred. Data compression is designed to reduce the size of a given payload. This means that data compression can be used to deal with the large volume of data associated with network monitoring and can reduce the size of archives (for both a centralised and distributed approach). It is, therefore, a good approach when dealing with a large quantity of data.

1.2 Problem Statement

The previous section concluded that data compression was an effective approach for dealing with the quantity of data that is required for network monitoring. This section defines the problem statement and lists the research questions which need to be answered.

Much research has been performed on data compression, optimising data compression and text compression (specifically the compression of English text). This research will be explored in the next chapter. As mentioned, Log files contain information which is useful for network monitoring. These log files are usually text files which have a defined structure. Many programs use the syslog format (which is defined in RFC3164 [17]) for logging. The syslog messages contain a message (less the 1024 bytes) and information about the time, originating host and originating process of the message. The messages are sent by the originating process to the syslog daemon which performs the logging of this information. Using syslog, many different processes can

log to the same file. It is commonly used by applications on Unix-based operating systems for logging.

Data compression programs can be used to reduce the size of text files by between 80 and 90 percent. Data compression tests are traditionally performed on a number of standardised corpii including: The Calgary corpus [18, 19]; the Canterbury corpus [20]; and the Hector corpus [21]. These data sets contain a number of files, such as: fiction writing, non-fiction writing, bibliographies, geophysical data, executable code, program code and bitmaps. There are also a number of other corpii on which tests have been performed, including: the Gutenberg corpus [22] (which consists of e-books from Project Gutenberg); Reuters-21578 [23]; Reuters Corpus Volume 1 (RCV1) [24]; and Reuters Corpus Volume 2 (RCV2) [25] (which consist of data from the Reuters newswire service) .

There are also a number of web sites which contain regular updates on compression programs and their performance on a number of files including: audio files; text files; books; PDFs; executables; and video files. These web sites include: Maximum Compression [26], Black Fox benchmark [27], Matthew Mahoney's Large text benchmark [28], The Squeeze Chart [29], Jeff Gilchrist's Archive Comparison Test [30] and Squxe Archivers Chart [31].

Out of all the corpii, only the corpus used by the Maximum Compression website contains a log file. This means that tests need to be conducted to determine the performance of compression programs on log files. The performance of the different compression programs on log files needs to be measured and compared to determine the best compression program for a given scenario. When performing this comparison, a number of metrics need to be considered, such as: the resources used; the compression time; the decompression time; and the size of the resulting payload.

Standard compression programs are not created to compress specific types of files, but rather they use characteristics such as recently repeated sequences and the distribution of the characters to reduce the size of files. As mentioned, log files are rather verbose and contain messages which have a defined structure. This leads to a large amount of redundancy which might not be exploited by the compression program. A good question would be whether the use of a preprocessor and postprocessor (which exploits known semantic knowledge about the log files) would improve the compression ratio and what the performance impact would be. This thesis sets out to also improve the compression ratio achieved by creating text preprocessors which reduce the size of the file and lead to a lower compression ratio on the file and improve the performance of the compression programs. The process of exploiting the semantic information is referred to as

semantic compression.

This thesis aims to determine how effective data compression is as a means of data reduction and to ascertain how the use of semantic knowledge can improve data compression. This thesis also sets out to apply the results to different scenarios and determine the best combinations of standard compression programs and preprocessors in these scenarios.

1.2.1 Research Questions

To solve this problem, the following research questions need to be answered:

1. How effective is compression in reducing the size of the log file and how much resources (time and memory) are used by the compression programs?
2. What sort of semantic knowledge exists within the log files?
3. Can this semantic knowledge be exploited to improve data compression?
4. In different monitoring scenarios, which compression programs are the best choices to reduce the quantity of data?

1.3 Approach

The following approach was taken to answer these research questions:

1. A collection of scripts to determine the compression ratios, compression times and decompression times when using data compression was compiled
2. These scripts were used to run tests on a collection of log files and the obtained statistics recorded
3. Maillog files were investigated to determine their structure and evaluate what semantic knowledge can be exploited to improve the performance of standard compression programs
4. A number of dictionaries were constructed for each maillog file which can be used to perform word-replacement and improve the performance of standard compression programs

5. These dictionaries were evaluated and a single dictionary constructed which could be used by a preprocessor for all the log files and achieve a greater amount of improvement than the individual dictionaries

6. The results were analysed and the performance of the different techniques were evaluated using the information obtained from the analysis. Conclusions were drawn as to which combinations yielded the best results for different monitoring scenarios

This approach to answering these research questions is based on empirical evidence from the experiments (tests) performed. For the tests conducted in this thesis, the scope of compression programs is limited to compression programs which are available on many different platforms and which show acceptable compression and decompression times. Tests were conducted using `gzip` [32], `zip` [33], `lzop` [34], `ppmd` [35], `arj` [36], `7zip` [37] and `bzip2` [38]. These compression programs and the algorithms which they use will be described in the next chapter.

For different monitoring scenarios, different compression programs are likely to be more appropriate since they vary in terms of compression ratio, compression time, decompression time and memory utilisation. The scenarios which are examined in this thesis are:

1. Filtering logs through to the central point for analysis (where latency is not important, but there is a desire to use as little bandwidth as possible)

2. Real-time monitoring (where a minimum point-to-point time is desired, using as little resources (memory, time and bandwidth) as possible)

3. Quick access, storing it compressed and later decompressing it for analysis (where fast decompression is desired, but the compression time does not matter. In addition, there is a secondary desire to use as little bandwidth as possible)

4. Low system-time-usage for compression and decompression (where fast compression and decompression times are desired and the compression ratio does not matter)

1.4 Document Structure

The remainder of this document discusses the problem presented in this chapter and presents the results and analysis of tests performed to answer the research questions. The structure of the remainder of this document is as follows:

Chapter 2 describes the different compression algorithms which can be used for compression. It shows the main developments in this area and describes methods which have been used to improve the performance of these compression algorithms. This chapter also discusses the improvements gained by using word-based compressors and text preprocessors. It concludes by discussing previous work on compressing log files and showing the algorithms which are used by each of the compression programs tested and motivating their choice.

Chapter 3 investigates how effective data compression programs are at reducing the size of log files. Tests are performed using all of the compression levels of each of the compression programs. The memory usage, compression times, decompression times, total times, and time to transfer the compressed payload at a given rate are all discussed. For each scenario, a compression program is recommended based on the results presented.

Chapter 4 presents the initial semantic investigation. It investigates the improvement in compression ratio, compression time and decompression time when using a preprocessor which replaces the IP addresses and timestamps with their binary equivalents. It also investigates what semantic knowledge is present within the corpus of maillog files and evaluates how this knowledge can be effectively exploited to improve the performance of data compression programs on this corpus.

Chapter 5 describes a number of word-based preprocessors which perform word replacement using a dictionary based on the log file being compressed. It also discusses how the dictionary is constructed and investigates the improvement in compression ratio, compression time and decompression time when using these word-based preprocessors and the seven standard compression programs.

Chapter 6 first investigates the improvement by a preprocessor which uses a dictionary based on the previous month's log file. It then describes a method for combining the dictionaries to form a single dictionary and presents the results achieved by a preprocessor which uses this dictionary. This chapter also discusses the development of a single dictionary based on the semantics present in the maillog file and the knowledge of the network and presents the results achieved by a preprocessor which uses this dictionary. Finally, this chapter recommends a preprocessor and standard compression program for each scenario based on the results presented.

Chapter 7 concludes this thesis and explains how the research questions have been answered. It also details the research contributions of this thesis and discusses future work which can be performed in this project domain.

Chapter 2

Related Work

Data compression is a rather mature field of work. There are many different types of compression algorithms which are designed to compress different types of data. There are also lossless and lossy compressors which are tailor made for video and audio compression, as well as lossless compression algorithms which are designed to work on English text [39].

This chapter focuses on lossless text compression algorithms and the different methods which have been used in an attempt to improve the compression ratios and compression times of these algorithms. This chapter begins by defining some fundamental terms and explaining important concepts in the field of data compression. It then examines the different encoding techniques used by data compression and explains how important compression algorithms such as the Lempel-Ziv family of compression algorithms, Prediction by Partial Matching (PPM) and the Burrows and Wheeler Compression Algorithms (BWCA) perform compression. It also discusses other statistical compression algorithms as well as context-mixing algorithms. It then investigates the improvements provided by word-based compressors and text preprocessors and concludes by discussing previous work on log compression and the compression programs which are used for testing in this thesis.

2.1 Fundamental concepts

There are a number of fundamental terms and concepts which are used in the field of data compression. The entropy of a file using a model, redundancy present in a file, adaptive algorithms, dictionary-based compressors, statistical compressors and block based compressors are some of

the common terms and concepts which exist. This section elaborates on the important terms and concepts which are used in this thesis.

2.1.1 Redundancy

Redundancy is defined as "being redundant", "superfluity", "surplus" while redundant is defined as "Characterized by superfluity or excess in some respect; having some additional or superfluous part, element, or feature" [40]. These definitions hold true when referring to redundant data. Essentially, then redundancy in data refers to patterns in the data source, to data being repeated in a data source or to discernible differences in the data source.

Terry Welch [41] lists four different types of redundancy that exist in a data source without explicit knowledge of how the data is interpreted:

1. The distribution of characters - some characters are more probable than others, so shorter codes for these characters will reduce the size of the data.
2. The repetition of character such as spaces - using a count (such as run-length encoding) can reduce the size of the data.
3. The high-usage of certain patterns or sequences - this can be identified and given a certain code, which will reduce the data size
4. Positional redundancy - there may be certain data which always appears in a certain position, this can be noted and used to reduce the data size.

Redundancies within a source file are thus exploited by different compression techniques to create a compressed data source.

2.1.2 Compression Techniques

Compression programs exploit the redundancy in files to create smaller files which can be decompressed to produce the original file. There are many different types of compressors. They may be block-based, dividing the data into blocks and running an algorithm on each block; dictionary-based, where they replace words with references to a dictionary; or they may use complex statistical models to predict future characters (statistically-based compressors). Techniques might also

be performed to improve the possibility of compression. These may be transformations (such as Burrows-Wheeler transform) or normalizations (such as Branch Call Jump (BCJ), where jump targets in executable files are normalized before compression) of data.

Compression may also be performed using the distribution of characters in the data to create statistical codes (such as Arithmetic and Range coders), new fixed length codes or variable length codes (usually prefix free codes such as Huffman coding). Compressors often use adaptive algorithms where the statistics used are updated by the encoders and decoders as they encode the data source and decode the encoded data respectively.

2.1.3 Entropy

Entropy, occasionally referred to as Shannon's entropy, is a measure of how much randomness exists in a given data source using a particular model. Shannon defines entropy as a measure of the average information content associated with a random outcome. Quantitatively, this may be expressed as: $H(x) = -\sum_{i=1}^{n} p(i) log_2 p(i)$, where $p(i)$ is a measure of the probability of the i-th symbol in an alphabet of n symbols. $p(i) log_2 p(i)$ is also known as the information content or self information of i-th symbol.

The entropy rate of a data source is the average number of bits per symbol needed to encode it [42]. This is usually indicated in bits per character (bpc). This can be calculated by dividing the size of the output produced with a given source (which is the entropy of that model) by the size of the source and multiplying it by the amount of bits used for each character in the source.

The character-based entropy uses $p(i) = \frac{x_i}{\sum_{i=1}^{n} x_i}$, where x_i is the frequency of the i-th symbol (i.e. the probability of the i-th symbol occurring in the data source). The character-based entropy gives an indication of the redundancy present due to the distribution of the characters in the data source. This redundancy can be exploited by coding techniques such as Huffman encoding or Arithmetic encoding (also known as entropy-based coders).

2.1.4 Definitions

Compression Ratio

This is defined as the size of the compressed payload divided by the size of the original payload. A lower compression ratio therefore indicates a greater amount of compression. This definition

is also used by other authors (for example [43, 44, 45]). In some texts, the compression ratio is defined as the size of the original payload divided by the size of the compressed payload (For example [41, 46, 47]). Another common method of denoting the compression ratio is using the bits per character (bpc) (for example [38, 48, 49]). This is the same as multiplying the compression ratio using the first definition by 8.

Consider a 50KB which is compressed to 12.5KB. The compression ratio would be 0.25 (i.e. 25% or 2 bpc) or 4 using the respective definitions. Using the first definition it is easier to see that the filesize is reduced by 75% than in the second definition. It is for this reason that this definition is used.

Payload Ratio

When using preprocessors, there are three different compression ratios. There is the compression ratio achieved by the preprocessor (the ratio of the preprocessed payload to the original payload), the overall compression ratio (the ratio of the compressed payload to the original payload) and compression ratio achieved on the preprocessed file (the ratio of the compressed payload to the preprocessed payload). In this thesis, the ratio of the compressed payload to the preprocessed payload is referred to as the payload ratio.

Compression Time

This is defined as the time taken to compress a given payload and write this payload to a compressed file. All times in this thesis (unless specified) are denoted in seconds.

Decompression Time

This is defined as the time taken to decompress a given compressed payload and write this to disk, thus obtaining the original file which was compressed. All times in this thesis (unless specified) are denoted in seconds.

2.2 Coding Techniques

There are a variety of coding techniques. These techniques either use variable length codes or fixed length codes to represent the data. The variable length codes are usually prefix free codes

(i.e. no codeword is a prefix of another codeword) so that they can be decoded. This section elaborates on some of the coding techniques which are used in data compression.

2.2.1 Huffman coding

Assuming an alphabet and a cost for each letter in the alphabet (usually the total number of occurrences of that letter) is given, Huffman coding constructs a prefix-free binary character code with a minimum cost [50]. It uses a binary tree which is constructed using the following algorithm:

1. Make each symbol a leaf node with their weights being the cost for each letter (each their own tree) - i.e a forest of trees [46]
2. While you have more than one tree
   (a) Take the two trees with the lowest root node weights
   (b) Create a new tree whose root has the weight of the sum of the two trees roots and is parent of the two trees

In this tree, all the letters in the alphabet are leaf nodes. The connection to the left child is then assigned the 0 bit, while the connection to the right child is assigned the 1 bit. If the tree is traversed from the root node to the leaf node, the Huffman code for that particular leaf node is obtained.

For example consider the string "this is a test". This string consists of the following characters: "t", "h", "i", "s", "e", "a", " ". When encoding this string using the ASCII character codes, each of the characters occupies 8 bits. The string "this is a test" therefore occupies 112 bits using ASCII. Figure 2.1 shows the construction of the Huffman tree for the string "this is a test". If the tree is traversed from the root node to the leaf node containing "t", the code 10 is obtained. "t" is therefore encoded using 2 bits instead of being encoded using 8 bits (ASCII). The code for each letter is obtained in the same manner. Using Huffman coding, the string "this is a test" occupies 38 bits.

Huffman coding is used in the DEFLATE [51] compression algorithm and a modified version of Huffman coding is used in fax machines.

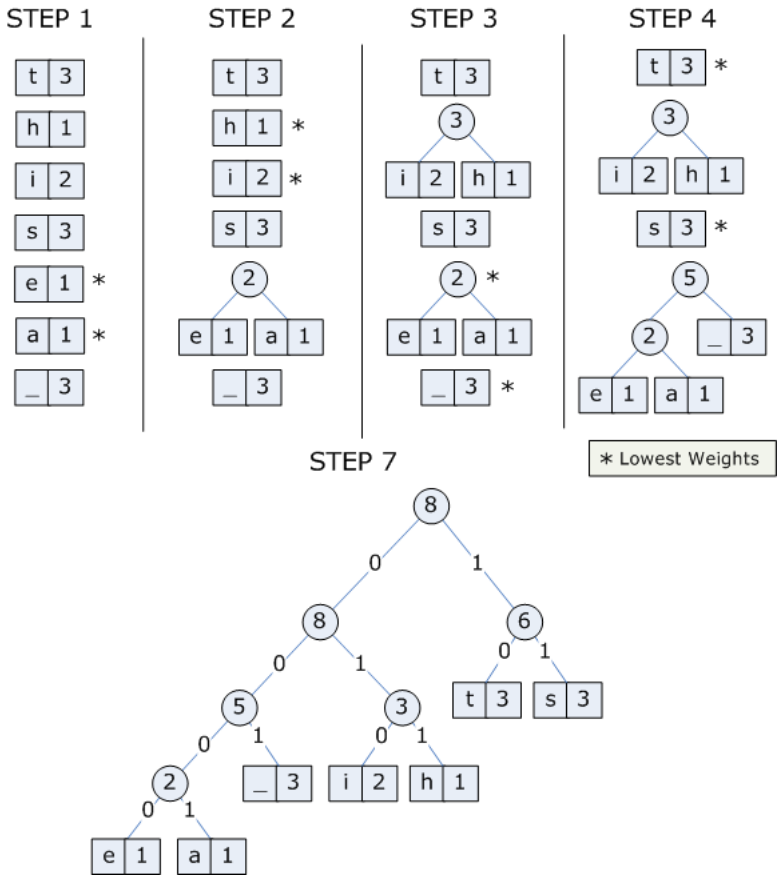


Figure 2.1: Huffman tree for the string "this is a test"

2.2.2 Unary coding

Unary coding is a simple form of entropy encoding developed for Turing machines which represents a natural number n as $n-1$ ones followed by a zero or $n-1$ zeros followed by a one [52]. For example, the newline character has a character code of 10 (in decimal) and would be represented as 1111111110 or 0000000001 using Unary coding.

2.2.3 Arithmetic encoding

Arithmetic encoding is a form of statistical coding defined in 1987 by Witten et al. [53]. Arithmetic encoding encodes the message into a single fraction. It is regarded as a generalisation of Huffman encoding where statistics do not have to have power-of-two relationships to each other to form optimal prefix free codes [54]. It uses a model of the data to divide the intervals based on the probability of each characters. This is performed recursively on the interval by splitting the interval obtained from the previous characters using the probability of the next character until a number which represents the message is obtained. Consider fixed probabilities of 0.5, 0.3, 0.2 for the characters "t", "e", and "s".

Start Interval	$[0, 1)$	$[0, 0.5)$	$[0.25, 0.4)$	$[0.37, 0.4)$
t	$[0, 0.5)$	$[0, 0.25)$	$[0.25, 0.325)$	$[0.37, 0.385)$
e	$[0.5, 0.8)$	$[0.25, 0.4)$	$[0.325, 0.37)$	$[0.385, 0.394)$
s	$[0.8, 1)$	$[0.4, 0.5)$	$[0.37, 0.4)$	$[0.394, 0.4)$

Table 2.1: Arithmetic encoding intervals for the string “test”

Table 2.1 shows how the intervals are divided starting from the interval $[0, 1)$. The first character is "t", so the interval becomes $[0, 0.5)$. The next character, "e", falls in the interval $[0.25, 0.4)$, so the interval becomes $[0.25, 0.4)$. Once the last character, "t", is reached, the interval is $[0.37, 0.4)$, so "t" falls in the interval $[0.37, 0.385)$. This means that the word "test" can be encoded using any value within this interval. This can be decoded using the intervals as above. Since any number within the interval $[0.37, 0.385)$ is also in the interval $[0, 0.5)$, the first character is a "t". This number is also in the interval $[0.25, 0.4)$, so the second character is a "e". Using this process, the word "test" can be obtained from any number within the interval $[0.37, 0.385)$.

A process called renormalization is used to prevent the finite precision from becoming a limit on the total number of symbols that can be encoded [46]. In this process, whenever the range of

the interval is reduced to the point where all values in the range share certain beginning digits, those digits are sent to the output. Existing digits can thus be shifted left and new digits can then be added on the right to expand the range as widely as possible [54]. This prevents the finite precision imposed by the machine becoming a limiting factor. Arithmetic encoding is a patented technique; however, it is used by many compression programs for entropy encoding.

2.2.4 Range Encoding

Range encoding is a form of arithmetic encoding that is free from the arithmetic encoding related patents. It uses integers instead of the numerators of fractions for its intervals (i.e. instead of starting with an interval of $[0, 1)$, it starts with an interval of $[1, 100000)$). Range encoding splits the intervals using probabilities in the same manner as arithmetic encoding (described in the previous section). Range encoders also perform renormalization; however, it is performed one byte at a time rather than one bit at a time [55].

2.2.5 Run length encoding

Run length encoding is a simple data compression technique. Runs of data (More than one consecutive occurrence of a single character) are stored as a single data value and a count. Run length encoding is used in fax machines, where there is a lot of white space [41].

2.2.6 Move-to-front coding

Move-to-front (MTF) coding is a good technique to improve the compression ratio of data where symbols occur in close locality. MTF coding is performed as follows:

1. A list (L) of the alphabet is constructed
2. The message which is to be encoded is the parsed from left to right. At each character, the position of that character in L is output to a stream S, and that character moved to the front of the L.
3. The stream S is the MTF coded message

The MTF causes letters which are used more often to be at the beginning of the list L and hence the lower code words will be used by an entropy encoder. The performance of an entropy encoder such as Huffman or arithmetic encoding is thus improved by the use of MTF. MTF coding is used by the Burrows and Wheeler Compression Algorithm [38, 56].

2.2.7 Other Coding Techniques

Besides the coding techniques mentioned, there are a number of other coding techniques which are important. This section investigates Golomb coding and Shannon-Fano coding.

Golomb coding

Golomb coding is an easy-to-implement entropy encoding scheme proposed by Solomon Golomb in 1966 [57]. It generally produces suboptimal codes; however, the codes are optimal for alphabets following a geometric distribution [57]. It operates by using an arbitrary parameter (M) as the divisor to find the quotient and remainder on the data source. The code word is then formed with a quotient code followed by a remainder code. The quotient code is the quotient encoded with unary coding. The remainder code is the remainder coded in binary using $b = \lceil \log_2 (M) \rceil$bits if is is greater than $2^b - M$ or $b - 1$ bits otherwise. Rice coding is a special case where the parameter is a power of 2. Rice encoding is less efficient, but easier to implement in hardware. It is used as the entropy encoding stage of a number of image and audio compressions methods [46].

Shannon-Fano coding

Shannon-Fano coding is an early form of entropy encoding. It generates sub-optimal prefix-free codes by arranging symbols from the most probable to least probable and dividing the symbols into two sets whose total probabilities are as close as possible to being equal. All symbols have the first digits of their code assigned, 0 for the first set and 1 for the second set. As long as any sets with more than one element exist, then the same process is applied to those sets. This is essentially a top-down approach to constructing prefix-free codes, as opposed to Huffman encoding which utilizes a bottom-up approach. [42]. Shannon-Fano coding is used in the Implode compression method for zip files [33].

2.3 Compression Algorithms

There are many different compression algorithms available. Some compression algorithms are optimised for different types of data. This section investigates the different compression algorithms available and previous work which has been done to improve their performance. It is divided into 5 parts. The first part describes the Lempel-Ziv family of algorithms, the second part describes the prediction by partial matching (PPM), the third part describes the Burrows and Wheeler compression algorithms, the fourth part describes Context mixing algorithms and the fifth section describes other statistical compressors such as Dynamic Markov Compression (DMC).

2.3.1 Lempel-Ziv Compression Algorithms

The are many different algorithms in the Lempel-Ziv (LZ) family of compression algorithms. These algorithms are based on the concepts presented by Ziv and Lempel in two papers written in 1977 [58] and 1978 [59]. These papers describe LZ77 and LZ78 respectively. They may be classified as sequential data compressors [46] since they encode variable-length sequences using a fixed-length code. These are also known as dictionary compressors since these variable-length sequences form a dictionary. This section gives details on the main algorithms in the LZ family: LZ77; LZ78; LZW and LZMA. It also gives details on the DEFLATE algorithm (which is used in compression programs such as `zip` and `gzip`).

LZ77

LZ77 is a compression algorithm defined by Jacob Ziv and Abraham Lempel in 1977 [58]. It operates using a buffer of previously encountered data from the string, looking for patterns called reproducible extensions and forming a code from their position in the buffer, the length of the reproducible extension and the character following it [58]. This is termed sliding window compression. The encoder and decoder keep track of a certain amount of recent data using a data structure (usually a buffer) called a sliding window. The compression is performed by replacing patterns in the file with references (length-distance pairs) to the data contained in the sliding window. “Go back distance characters in the buffer and copy length characters, starting from that point” is how the length-distance pairs are used.

LZSS is a derivative of LZ77 created in 1977 by James Storer and Thomas Szymanski. It is essentially a dictionary-encoding technique. It replaces a string of symbols with a reference to a dictionary location of the same string. The main difference between LZ77 and LZSS is that in LZ77 the dictionary reference can actually be longer than the string it is replacing. LZSS uses a 1 bit flag to indicate whether the data is a byte or a length-distance pair [60].

LZ78

LZ78 is a compression algorithm defined by Jacob Ziv and Abraham Lempel in 1978 [59] expanding on their LZ77 algorithm [58]. LZ78 works on future data, forward scanning the input data and matching it against a dictionary which it maintains. It scans into a buffer until it can not find a match in the dictionary. At this point it then adds this word to the dictionary and outputs the codeword corresponding to the last word found in the dictionary (zero if the dictionary is empty) as well as the character which, when added to the buffer, causes a match not to be found in the dictionary. Thereafter the buffer is emptied and the forward scanning continues.

LZW

LZW is a compression algorithm defined in 1984 by Terry Welch. It is organised around a translation table, referred to as a string table, which maps strings of input characters to fixed-length codes (12 bit codes). The string table has the property that for every string in the table, its prefix string is also in the table. LZW uses a "greedy" parsing algorithm, where the input string is examined in one pass, and the longest input string recognized in the string table is used for encoding. The parsed string, which is extended by the next input character, forms a new string which is added to the string table [41] (In a similar manner to LZ78). LZW is a patented method, and therefore has not been widely adopted. It is, however, used in GIF image format.

LZC is an implementation of LZW which uses variable size pointers. It also includes additional logic for restarting the algorithm when the source file changes its characteristics enough to worsen compression. It uses hashing to store the dictionary enabling quick lookup. There are few differences between LZC (which the unix `compress` program uses) and LZW. When the dictionary is full, LZW and LZC becomes non-adaptive, however when this occurs, LZC begins to monitor the compression performance (in terms of ratio). When the performance degrades too much, LZC clears the dictionary and restarts the algorithm building a new dictionary. This improves compression performance. Horspool [45] further improves the compression ratio

achieved by these programs by up to 8% using binary encoding of string numbers and allowing the dictionary to fill at a faster rate.

LZMA

Lempel-Ziv Markov Chain Algorithm (LZMA) is an LZ77-based technique which uses a large dictionary (up to 1GB) and reduces values with a range-encoder rather than Huffman encoding (used in DEFLATE). No official implementation details are available; however, the SDK available suggests the use of a range encoder, Patricia tries, hash chains and binary trees [61]. On the forum located on the source forge site, the author (Igor Pavlov) states that Markov models are used for class selecting (order 3 - literal, match and repeat match) and encoding the literals [62]. The match finders used by LZMA use Hash Chains or Binary Trees, depending on the compression method and compression mode used [37].

DEFLATE

DEFLATE is a compression algorithm which uses a combination of the LZ77 algorithm and Huffman coding. It is one of the most popular compression algorithms (certainly the most popular LZ77-based algorithm). It was defined by Phil Katz (creator of PKZIP) and is used for compression by `zip` and `gzip` [51]. `zlib` is a commonly-used library implementation of the DEFLATE algorithm [63].

2.3.2 Prediction by Partial Matching (PPM)

PPM is a technique of determining the probabilities of the next character based on previous occurrences. When combined with Arithmetic coding it creates an effective compression algorithm which achieves a low compression ratio. PPM was introduced in 1984 by John Clearly and Ian Witten with two variants - PPMA and PPMB. Since then, a number of variants and improvements have been suggested. This section takes a look at the different developments in the PPM technique, which include: PPMC; PPMP; PPMX; PPMD; PPM*; PPMZ; and PPMII. These PPM models have not only been applied for data compression, they have also been successfully applied in a number of different areas including: machine learning of human language; predicting the stock market; and natural language applications (such as: language identification; cryptography; and automatically correcting words).

PPMA and PPMB

PPM takes an adaptive approach to building a model, i.e the model is adapted dynamically as the stream is processed. As previously mentioned, PPM determines probabilities of the next character based on the previous characters. The previous characters are referred to as the context. An n^{th}-order context is defined as the previous n characters. In the string "This string is a great strin" (where the next character is "g"): the first-order context would be "n", the second-order "in", the third-order context would be "rin", etc. These contexts are used to predict the next character as "g". Two examples of previous third order-contexts which are followed by the letter "g" are: " a " and "rin". To predict the probabilities of a "g", the contexts and the characters which follow are stored with frequencies to calculate the probabilities. For the context "rin", the frequency for "g" being the next character is 1, yielding a probability of 0.5. In this manner a number of Markov models of order N are built (where N is the length of the context). A single higher-order model would not work since there is a conflict between the desire to use a high-order Markov model and the need to have them formed quickly as the initial part of the message is sent. This is resolved with partial string matching. A high-order model is formed but lower-order predictions are used in the case when high-order ones are not yet available (i.e. using a partial string of that context). The decoder and encoder use escape characters to back down to the previous order context. The last order is order-0, where the distribution of characters in the data source to that point is used to determine the probability. If a character has never occurred then a further escape character is used followed by the new character. Calculating the probability of an escape character (i.e. the probability of a character occurring for the first time - a novel event) is a large, open problem known as the zero frequency problem. The calculation of these probabilities is where many of the PPM models differ. Clearly and Witten [54] define two methods known as PPMA and PPMB.

PPMA considers the occurrence of any new character a novel event and counts it as one. This can be formally written as follows:

Let φ be the character which follow a given context, a the size of the coding alphabet, q the number of characters which has occurred and C the total number of times the given context has been seen. Using PPMA, $p(\varphi)$ is as follows:

$$p(\varphi) = \begin{cases} \frac{1}{1+C} \cdot \frac{1}{a-q} & c(\varphi) = 0 \\ \frac{c(\varphi)}{1+C} & c(\varphi) > 0 \end{cases}$$

PPMB is identical to PPMA except for the fact that it only considers the occurrence of a character a novel event when it has already occurred once. This can be written formally as follows:

Let φ be the character which follows a given context, a the size of the coding alphabet, q the number of characters which has occurred and C the total number of times the given context has been seen. Using PPMB, $p(\varphi)$ is as follows:

$$p(\varphi) = \begin{cases} \frac{q}{C} \cdot \frac{1}{a-q} & c(\varphi) \leq 1 \\ \frac{c(\varphi)-1}{C} & c(\varphi) > 1 \end{cases}$$

The maximum order for PPMA and PPMB is provided as a parameter. It has been found that the optimal maximum order grows with the length of the source being compressed [54]. PPMB achieves, on average, a 2.54 percent lower compression ratio than PPMA.

PPMC

PPMC was proposed by Moffat [64] as an improvement on PPMA and PPMB [54]. It is essentially a hybrid of its predecessors (PPMA and PPMB). Since it is wasteful to only start using a context for predictions when it has already occurred twice (method B) and it is desirable at the initial stages to allocate more than a count of 1 to an escape (method A), PPMC counts the escape as having occurred a number of times equal to the number of distinct symbols encountered in the context. The total count is hence also inflated by the same amount. This may be written formally as follows:

Let φ be the character which follows a given context, a the size of the coding alphabet, q the number of characters which has occurred and C the total number of times the given context has been seen. Using PPMC, $p(\varphi)$ is as follows:

$$p(\varphi) = \begin{cases} \frac{q}{C+q} \cdot \frac{1}{a-q} & c(\varphi) = 0 \\ \frac{c(\varphi)}{C+q} & c(\varphi) > 0 \end{cases}$$

PPMC also introduces a technique called update exclusion. In this technique, the symbol count is only increased in the context levels at or above the context level which the character is successfully predicted [64]. The modifications in PPMC result in a ten percent improvement lower compression ratio than PPMB.

PPMP and PPMX

PPMP and PPMX were proposed by Witten and Bell [65] as an improvement in estimating the escape probabilities. They use a Poisson process to improve the approach of estimating the escape probabilities (inspired by its original use to estimate the number of unseen biological species). Using this technique, a non-zero estimated probability for novel tokens can be obtained even though relative frequency is zero. The appearance of each token is formed as a separate Poisson process. This yields $p(esc) = \frac{t_1}{n} - \frac{t_2}{n^2} + \frac{t_3}{n^3} - \cdots$. This technique is known as PPMP. PPMX uses the first term of PPMP as a good approximation; i.e. $p(esc) = \frac{t_1}{n}$. Unfortunately, the low frequency contexts used by PPM cause the method to "break down" (i.e. yielding $p(esc) = 0$ or $p(esc) = 1$ for Method X and positive or negative probabilities for Method Y). This can be solved by altering the first term to be $\frac{t_1+\alpha}{n+\beta}$ (for small values of α and β) when break down occurs. $\alpha = 1$and $\beta = 2$ are found to be the best values in terms of compression performance [65]. Witten and Bell also define another method PPMXC which reverts to PPMC when break down occurs. The performance is investigated using two Books, "Far from the Maddening Crowd" and "Principle of Computer Speech" (book1 and book2 in the Calgary corpus). PPMP and PPMX were found to perform virtually identically, outperforming PPMA, PPMB and PPMC, while PPMXC achieves the lowest compression ratio. It must be noted that compression ratios achieved by PPMP are not significantly different to those achieved by PPMC.

PPMD

PPMC is considered one of the best methods for calculating escape probabilities. Howard [46] presented a new method called PPMD which builds on the work done by Moffat [64]. PPMD is very similar to PPMC, but achieves better results by treating new symbols more consistently. PPMD adds 0.5, as opposed to 1, to the symbol count and the escape count, which increases the total weight by 1 instead of 2. This may be written formally as follows:

Let φ be the character which follows a given context, a the size of the coding alphabet, q the number of characters which has occurred and C the total number of times the given context has been seen. Using PPMD, $p(\varphi)$ is as follows:

$$p(\varphi) = \begin{cases} \frac{\frac{q}{2}}{C} \cdot \frac{1}{a-q} & c(\varphi) = 0 \\ \frac{c(\varphi)-\frac{1}{2}}{C} & c(\varphi) > 0 \end{cases}$$

Tests show improvements of between 0.01 and 0.03 bpc on Calgary corpus [46]. PPMD achieves a one percent lower compression ratio than PPMC.

PPM* and PPMZ

In the previously mentioned PPM algorithms, the maximum context length is bounded by a fixed constant. PPM* and PPMZ are different to their predecessors because they exploit contexts of an unbounded length.

PPM*

PPM* uses considerably greater computational resources (both in terms of space and time) than PPMC. In order to practically use unbounded context lengths, the model needs to be stored in such as way that gives rapid access to predictions based on any context, eliminating the need for an arbitrary bound to be imposed. This requires a large amount of memory. Clearly and Witten [66] suggest using a trie structure to store the PPM model in conjunction with pointers back into the source and a linked list of pointers to the currently active contexts. This data structure is called a context trie. While this still uses a larger amount of memory than PPMC, it works well. More information about context tries can be found in Section 2.1 of [66]. Because the context length is unbounded, the selection of the model order becomes an issue. The use of the entropy to that point as a deciding metric is not effective because different model orders are optimal on different parts of the data source. PPM* chooses the shortest deterministic context (a context is deterministic if it only gives one prediction) if one exists, otherwise its chooses the longest context to estimate the probability of the next symbol. In PPM*, the use of escape characters is very important, since it will always be frequently used because of unbounded context lengths. In PPM*'s predecessors, escaping decays over time as new contexts occur since they have a bounded maximum context length and hence only a finite number of contexts are possible. This is not the case in PPM* because of the use of unbounded context lengths. PPM* results in a 5.6% improvement over PPMC and 3.7% over BWCA (to be discussed in the next section) on Calgary corpus [66].

PPMZ

PPMZ is an improvement of PPM*. With the use of the PPM* algorithm, there is a high dependence of typical data on local order. This is dealt with in PPMZ by only accepting very long

deterministic context lengths. PPMZ operates as follows:

1. First search for a deterministic context
2. If one is not found or an escape occurs, local order estimation (LOE) is performed to choose a finite-context length between 12 and 0. LOE is a scheme to decide which order to use for each character of a file. It is based on the idea that data at different parts of the file can be better compressed using a different optimal order (exploit the local redundancy of the data). The best confidence measure is found to be the most probable symbol's probability. Essentially, LOE is "a smart decision heuristic which throws away statistics (higher order contexts) it deems to be unreliable" [67].
3. Once this optimal order is found, ordinary PPM is performed and the higher order models which were skipped by the LOE are then updated [67].

In PPMZ, the unbounded length contexts are stored by using a series of linked lists and a hash of order-12 context indexes. The order-12 index contains a pointer into the actual data file so that the remaining (unbounded) characters can be matched. This context index also contains a minimum match length. The current context must match this length. If an escape is coded (i.e. no match has been found) then a new order-12 context index is created for the current unbounded context. The minimum match length is set to the current match length plus one and the failed context is also updated. This means that the contexts are never coded from again unless a subsequent context matches enough characters to distinguish between the two and also forces all unbounded-length contexts to be deterministic since the current context must match this length to be considered. As in PPM*, the coding of escapes is very important. Bloom [67] states that escape probabilities are overestimated by traditional PPM. Special care is taken with the coding of escapes in PPMZ in that a secondary model is used to estimate the frequency of escape characters. This technique is called Secondary Escape Estimation (SEE). It makes the escape estimation process adaptive. Ordinary PPMC escape counting is performed (number of novel characters in a context) and order 0, 1 and 2 escape contexts are constructed to send statistics which will be used for SEE. The escape probability is then calculated as a weighted sum from these statistics. The use of SEE causes an improvement in the compression ratio achieved.

PPMII

Despite the improvements in compression ratio, PPM-based algorithms have not largely been adopted. Applications tend to use programs which use Lempel-Ziv-based algorithms or algo-

rithms which use Burrows-Wheeler transform. This is largely due to the high computational complexity created by the use of large contexts, local order estimation, etc. PPMII is an improvement to the PPMD algorithm, which boasts a complexity which is comparable to the widespread compression schemes based on LZ77, LZ78 and BWT algorithms. PPM models are essentially based on the assumption that the longer the initial part of the context the more similarity there is between their conditional distribution. One of the main problems with PPM-based algorithms (besides the escape probabilities) is statistics insufficiency in the higher order contexts [48]. A number of different techniques such as LOE have been defined, however they require a lot of resources.

In PPM, lower-order contexts gather useful information which can be passed on to higher-order contexts which are essentially augmented lower-order contexts. The terms parent and child contexts are used to describe these two contexts. Given a string “This is a strin”. The order 1 context “n” is the parent context of the order 2 context “in" (and the order 2 context is the child of the order 1 context), which is the parent context of the order 3 context “rin” (with the order 3 context being the child of the order 2 context) and so forth. PPMII takes advantage of the similarity of distribution functions in the parent and child contexts by using "inherited frequencies". This is done by setting the initial generalised symbol frequency in the child context with regard to information about this symbol gathered in the parent context. Reference to the parent context only occurs at the addition of a new symbol to the child context which is rare and makes a fast solution possible. The rare use of the parent context again also enables the model to adapt quickly and makes it a fast alternative to LOE for local order estimation. This technique is known as Information Inheritance. It achieves a 8 percent lower compression ratio than PPMD. Information Inheritance also requires a modification to the update exclusion methodology. In the original method, parent contexts are used only for coding new symbols not seen in child contexts. For Information Inheritance to be effective, however, the frequency of the parent needs to be increased for the calculation of the initial generalised symbol frequency to be accurate should it have any more children. The solution is to increase the frequency of the parent but with weight of $\frac{1}{2}$. If a symbol is processed in the longest context there is no need to update the parent since the statistics can be considered stable with a small escape probability. By not updating these statistics, execution time is also increased by up to 10 percent. As in PPM*, PPMII uses a variation of SEE to improve escape estimations and hence improve the compression ratio achieved.

Summary

Prediction by Partial Matching is a finite-context statistical modeling technique that can be viewed as blending together several fixed-order context models to predict the next character in the input sequence [68]. Bounded and Unbounded contexts have been investigated and techniques such as LOE and SEE. PPMII achieves the lowest compression ratio using Information Inheritance and SEE. It is also the most efficient out of the PPM family; exhibiting complexity comparable to the Lempel-Ziv family of algorithms and the algorithms which use the Burrows-Wheeler transform (to be discussed in the next section).

2.3.3 Burrows and Wheeler Compression Algorithm

The Burrows and Wheeler Compression Algorithm (BWCA) was developed by Michael Burrows and David Wheeler during their time at the Digital Systems Research Center. Since the 1994 paper [38] in which the BWCA and the Burrows and Wheeler transform was introduced, much work has been done on this technique. This section introduces the BWCA, explains the main step in this algorithm (The Burrows-Wheeler Transform) and discusses the similarities in this approach to PPM and further improvements which have since been made to the BWCA.

The BWCA consists of 4 parts:

1. The Burrows-Wheeler Transform (BWT)
2. Recency Ranking Technique (Usually Move-to-front coding known as the General Structural Transformation [69])
3. Zero Run Length Encoding
4. Entropy coding

In some variations, Run length encoding (RLE) is included before the BWT to improve the speed of the BWCA. Unlike other compression programs (such as the Lempel-Ziv-based compression algorithms and the PPM compression algorithms) which process the input data sequentially, the Burrows and Wheeler compression algorithms process the input data in blocks. The main innovation of the BWCA is the BWT, which reorders the characters contained in the block into an order which is easily compressed with simple algorithms such as MTF. It is for this reason that the BWCA is often referred to as a block-sorting algorithm. The choice of MTF coding

is important. While contexts are grouped together, there is no per-context statistical information kept and so the encoder must rapidly adapt from the distribution of one context to the distribution of another. The MTF coder has this rapidly adapting quality [70] and is hence used.

The function of RLE to support the probability estimation by decreasing the amount of long runs of a single symbol which cause a high probability for that symbol. In the BWCA, RLE is used to take advantage of the long run of zeros formed by the MTF coding. The BWCA uses a variant of RLE introduced by David Wheeler, known as Zero Run length encoding (RLE0). In RLE0, the length of the run is represented using two characters (0 and 1). The value of the other characters are increased by 1, so that 0 and 1 can be used to encode a run of zeros. Fenwick [71] notes that RLE0 is the most effective run length scheme for the BWCA. More information can be found in Wheeler [72], Fenwick [71], Deorowicz [73] and the mathematical investigations into the properties of the BWCA conducted by Balkenhol et al [74, 75].

Once run-length encoding has been completed, the data is compressed using entropy encoders, such as Huffman encoding or Arithmetic encoding, which assign shorter codes to more frequently occurring symbols.

Burrows Wheeler Transform

The Burrows-Wheeler Transform is the central part of the BWCA. It is a reversible transform discovered by David Wheeler in 1983 while he was working for AT&T Bell Laboratories, but this remained unpublished until 1994. It essentially computes a permutation of the input sequence in which symbols with a similar context are grouped closely together so that the input sequence can then be easily compressed using simple compression methods such as MTF.

The transformation operates as follows:

1. On a block consisting of n characters, it performs n rotations (cyclic shifts) on the blocks
2. It then sorts the n rotations lexicographically and extracts the last character from each of the rotations, taking note of the number of the block which the contains the original string
3. The block formed from the last character of each of the rotations is then the transformed string. The number of the block containing the original string is used to identify the original string in the transformed matrix.

For example, consider performing the transform on the string "ALPHABETA". This is a block consisting of nine characters. The matrix of the n rotations is as follows:

$$\begin{matrix} A & L & P & H & A & B & E & T & A \\ L & P & H & A & B & E & T & A & A \\ P & H & A & B & E & T & A & A & L \\ H & A & B & E & T & A & A & L & P \\ A & B & E & T & A & A & L & P & H \\ B & E & T & A & A & L & P & H & A \\ E & T & A & A & L & P & H & A & B \\ T & A & A & L & P & H & A & B & E \\ A & A & L & P & H & A & B & E & T \end{matrix}$$

These cyclic rotations are sorted lexicographically. This produces the following matrix:

$$\begin{matrix} A & A & L & P & H & A & B & E & T \\ A & B & E & T & A & A & L & P & H \\ A & L & P & H & A & B & E & T & A \\ B & E & T & A & A & L & P & H & A \\ E & T & A & A & L & P & H & A & B \\ H & A & B & E & T & A & A & L & P \\ L & P & H & A & B & E & T & A & A \\ P & H & A & B & E & T & A & A & L \\ T & A & A & L & P & H & A & B & E \end{matrix}$$

The transformed string is “THAABPALE" and the third rotation contains the original string “ALPHABETA”.

This transformation is reversible by adding the input block as the first column of a matrix (originally empty) and performing a lexicographical sort until the matrix (then of size n by n) used to get the input string is reconstructed. Two methods may be used to identify the original block in this matrix. One is adding an end-of-block character to the block before performing the transformations, while the other is to retain the index of the original string in the matrix [38], as in the example shown.

The example string “ALPHABETA" is obtained from "THAABPALE" as follows:

$$
\begin{array}{cccccccccccccccc}
T & & A & & T & A & & A & A & & T & A & A & & A & A & L & \\
H & & A & & H & A & & A & B & & H & A & B & & A & B & E & \\
A & & A & & A & A & & A & L & & A & A & L & & A & L & P & \\
A & & B & & A & B & & B & E & & A & B & E & & B & E & T & \\
B & \rightarrow & E & \rightarrow & B & E & \rightarrow & E & T & \rightarrow & B & E & T & \rightarrow & E & T & A & \rightarrow \cdots \\
P & & H & & P & H & & H & A & & P & H & A & & H & A & B & \\
A & & L & & A & L & & L & P & & A & L & P & & L & P & H & \\
L & & P & & L & P & & P & H & & L & P & H & & P & H & A & \\
E & & T & & E & T & & T & A & & E & T & A & & T & A & A &
\end{array}
$$

$$
\begin{array}{cccccccccc}
T & A & A & L & P & H & A & B & E & \\
H & A & B & E & T & A & A & L & P & \\
A & A & L & P & H & A & B & E & T & \\
A & B & E & T & A & A & L & P & H & \\
\rightarrow B & E & T & A & A & L & P & H & A & \rightarrow \\
P & H & A & B & E & T & A & A & L & \\
A & L & P & H & A & B & E & T & A & \\
L & P & H & A & B & E & T & A & A & \\
E & T & A & A & L & P & H & A & B &
\end{array}
\begin{array}{ccccccccc}
A & A & L & P & H & A & B & E & T \\
A & B & E & T & A & A & L & P & H \\
A & L & P & H & A & B & E & T & A \\
B & E & T & A & A & L & P & H & A \\
E & T & A & A & L & P & H & A & B \\
H & A & B & E & T & A & A & L & P \\
L & P & H & A & B & E & T & A & A \\
P & H & A & B & E & T & A & A & L \\
T & A & A & L & P & H & A & B & E
\end{array}
$$

The original string ("ALPHABETA") can be found in the third row of the matrix.

The transformed block compresses well because the localized region of a string is likely to contain a large number of a few distinct characters. For instance, notice the proximity of the character “A” in “THAABPALE” in comparison to “ALPHABETA”. Consider any word, such as “the”. When the rotations are sorted, all the rotations beginning with “he” will be sorted together, which will cause the “t”s to be in close proximity to each other. The $P(c)$, where c is a character, at a given point in the string is very high if c occurs near that point in the string [38]. This is the exact property which is required for effective compression by a MTF coder as it encodes an instance of a character by a count of distinct characters seen since the previous occurrence.

Similarities to other programs

Although the BWT may seem to be completely different for other compression techniques, Clearly et al. [66] noted that the effect of this transformation is rather similar to PPM. The

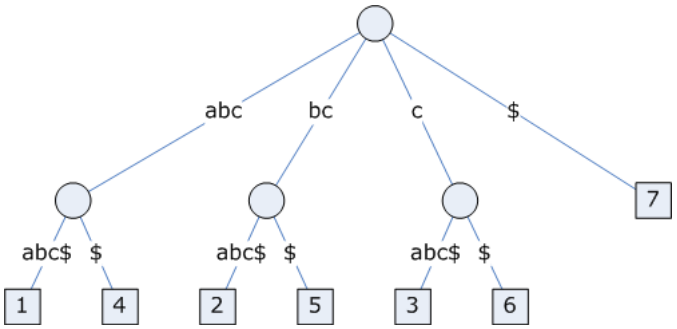


Figure 2.2: Suffix tree for the string "abcabc$"

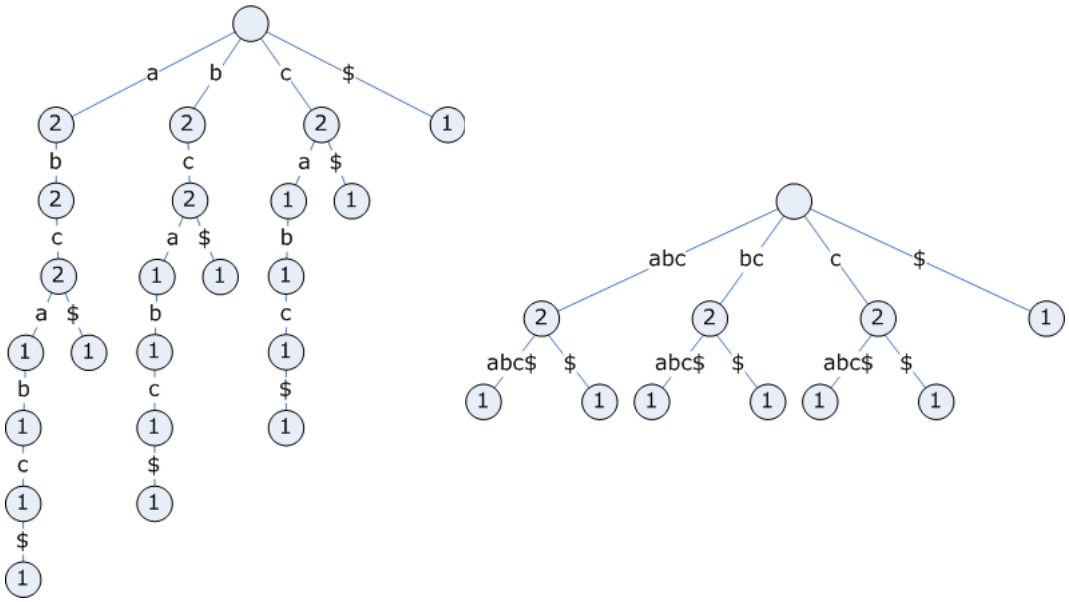
BWT transformation can be performed by sorting the suffixes of the input string through the creation a suffix tree and performing a walk in lexicographical order to recover the sorted suffixes [38]. The "$" character is included to denote the end of the string. For the string "abcabc", the transform is performed on "abcabc$" as follows:

$$
\begin{array}{ccccccc}
a & b & c & a & b & c & \$ \\
b & c & a & b & c & \$ & a \\
c & a & b & c & \$ & a & b \\
a & b & c & \$ & a & b & c \\
b & c & \$ & a & b & c & a \\
c & \$ & a & b & c & a & b \\
\$ & a & b & c & a & b & c
\end{array}
\rightarrow
\begin{array}{ccccccc}
\$ & a & b & c & a & b & c \\
a & b & c & \$ & a & b & c \\
a & b & c & a & b & c & \$ \\
b & c & \$ & a & b & c & a \\
b & c & a & b & c & \$ & a \\
c & \$ & a & b & c & a & b \\
c & a & b & c & \$ & a & b
\end{array}
$$

This gives a transformed string of "cc$aabb".

The suffix tree for the string "abcabc" is shown in Figure 2.2. A lexicographical walk visits the leaf nodes in the following order: 7, 4, 1, 5, 2, 6, 3. The characters preceding the characters at these positions are "c", "c", "$", "a", "a", "b", "b". These characters form the string "cc$aabb". When the inverse BWT is applied to this string, it results in "$abcabc". Since the "$" character is included to denote the end of the string, the string rotation with "$" at the end is the transformed string. This would be "abcabc$".

Figure 2.3 (a) shows the complete context tree, which is used by PPM to calculate statistics (Note that optimisations to PPM such as count-scaling and only adding nodes when certain criteria are met have been omitted in this tree). The counts of the characters are indicated in each node. Figure 2.3 (b) shows the context tree with compressed paths. Paths which consists of only single-child nodes are compressed into a single node by traversing the path and adding each single-child



(a) Complete Context Tree (b) Context Tree with compressed paths

Figure 2.3: Context Trees for string "abcabc$"

node to its parent node. This produces compressed paths. Note that this technique produces a structure identical to the suffix tree for the string "abcabc$". This result is also observed by Larsson [76], who suggests that the suffix tree incorporates exactly the structure of a context tree [76]. In PPM, however, it is the next character being predicted, not the character preceding the context (which is the case with the BWT). The BWT also omits the counts, which correlates to discarding a lot of structural and statistical information about the suffix tree before starting the move-to-front coding [77]. This is the essential difference between the BWT and PPM. The BWT is also not adaptive, but block-based [66]. Both PPM and BWCA predict symbols based on context (either provided by a suffix or a prefix) and both algorithms can be described in terms of trees [77].

Improvements

Since the first presentation of the BWCA by Burrows and Wheeler [38], there have been a number of developments and performance optimisations by numerous authors. Due to the large amount of developments, the scope of the improvements presented in this section is limited to the performance optimisations and improvements which are used in the `bzip2` compression program (for more details see [78]) or are of interest for preprocessing. However, other work will

be mentioned - albeit without detail - for reference and completeness.

"The most important factor in compression speed is the time taken to sort the rotations of the input block" [38]. Burrows and Wheeler [38] use a quick-sort to generate the sorted list of suffixes for the BWT. The sorting process is slow for files which contain long runs of identical symbols. Fenwick [79, 80] proposes the use of a word-orientated sort and the use of run-length encoding for runs longer than six to improve the speed of sorting. David Wheeler's implementation ("bred") [72] uses a different approach where the strings are split into groups according to the first letter, after which these groups are sorted with a quick-sort routine starting from the smallest to the largest group. This is also known as a forward radix sort [81]. Fenwick [71, 82] notes that this technique produces "impressive speeds". Further improvements in speed are made by Sadakane [83], who combines the techniques from Karp et al. [84], Bentley and Sedgewick [85], Anderson and Nillson [81] and Manber and Myers [86]. This technique requires more memory than Bentley and Sedgewick [85] (whose fast sorting algorithm blends a quick-sort and a radix sort); however, it is much faster. Based on these techniques, Seward developed the fast suffix sorting algorithms "copy" and "cache" which are used in `bzip2` (these are variants of Bentley and Sedgewick's algorithm [85]). `bzip2`, however, does fall back to Sadakane's technique if there are too many deep comparisons (for example, comparisons between rotations such as aaaaaaab and aaaaaaba or rotations of strings, such as abcabc, where one of the rotations is the original string) [87]. Balkenhol [74, 75], Kurtz and Balkenhol [88], Itoh and Tanaka [89], Kao [90] and Mazini and Ferragina [91] present other techniques which can be used to improve the speed of sorting. `bzip2` uses the "mergedTL" method for decompression which is defined by Seward [92]. It is 67 percent faster than the original presented by Burrows and Wheeler [38]. Seward also shows that less memory can be used in exchange for an increased decompression time. `bzip2` uses less memory in exchange for an increased decompression time when the `-s` flag is specified.

Schindler [93] presents an interesting alternative to the BWT which is between 2 and 30 times faster than the BWT, but produces higher compression ratios. This is done by sorting only limited contexts, rather than unlimited contexts, to increase the speed. A difference only arises between the two if no difference is found once the context limit is reached. When this occurs, BWT sorts on deeper contexts, while Schindler's transform sorts on the position in the original file.

Since the BWCA consists of sorting the rotations lexicographically, modifying the ordering provides an interesting way to improve the compression ratio achieved by BWCA. Chapin [70] performs experiments to determine which characters are "close" to each other in order to group characters such that changes in context are not so dramatic. Chapin shows that grouping vow-

els together and punctuation together results in a lower compression ratio than using the ASCII ordering. In contrast, grouping lower and uppercase letters yields a higher compression ratio than ASCII ordering [70]. This makes the possibility of a preprocessor for `bzip2` (For further discussion on this matter, refer to Section 2.5).

`bzip2` uses RLE before the BWT to increase the speed of sorting. There are mixed thoughts among the research community about using RLE before the BWT. Fenwick [71] uses RLE to eliminate long runs, decreasing the length of the sequence on which the BWT gets performed and hence improving the sorting time. This results in a reduced sorting time but an increase in compression ratio of approximately 0.1 percent. Balkenhol and Kurtz suggest only using RLE if the run is less than 70 percent of the string [74]. Deorowicz [73], meanwhile, recommends that RLE should not be used before the BWT since it causes a loss of some of the information of the context in which the symbols occur. Abel [69] also recommends not using RLE before the BWT since there are sorting algorithms which sort the runs of symbols practically in linear time. These methods result in higher compression times, but lower compression ratios than `bzip2`.

Further improvements have also been made by altering the MTF transform and using different forms of entropy encoding. Schindler [93] presents an altered form of the MTF coder where elements are not moved to the front of the list. Balkenhol and Shtarkov [75] presents MTF-1 and Balkenhol et al. [74] present MTF-2 which move elements to the second position of the list and only elements in second position of the front of the list. Other techniques such as Inversion Frequencies [94, 95], Weighted Frequency Count [73, 96] and Distance Coding (a technique by Edgar Binder originally presented in a news group, but explained by Deorowicz [96]) can be used in place of the MTF to exploit the recency that results from the use of the BWT. Good summaries of these improvements are provided by Abel [69] and Deorowicz [96].

Different forms of Entropy coding can be used to improve the compression performance. `bzip2` uses a modified form of Huffman coding. Arithmetic coding can also be used to give a lower compression ratio. Earlier versions of `bzip2` used Arithmetic coding; however, since it is a patented method, Huffman coding is now used. Shannon Coders and Structured Coding models have also been explored by Fenwick [71]. Hirchberg and LeLewer [97] also present a method of coding based on Huffman coding. The modified version of Huffman coding used in `bzip2` is similar to the one presented by Hirschberg and LeLewer [97].

`bzip2` shows a 3.06 percent improvement in compression ratio from the original by Burrow and Wheeler [38]. Other improvements mentioned in this section provide up to 5.7 percent improvements from the results achieved by `bzip2`. The text preprocessing methods presented in Section

2.5 reveal an improvement of up to 20 percent [77] using simple reversible transformations.

2.3.4 Statistical compressors

Since the advent of PPM, statistical compressors have been explored. These compressors produce low compression ratios, but are much slower than the Lempel-Ziv based programs and are often slower than PPMII as well. The main areas which have been explored in the following section are Dynamic Markov Compression (DMC) and Neural Networks.

Dynamic Markov Compression

DMC relies on a weak assumption that the stream of bits can be generated by a discrete-parameter Markov model. Ziv and Lempel [58, 59] and Cleary and Witten [54] also make a similar assumption in that they assume that the data source can be modeled using a Markov model.

DMC is rather similar to PPM. As in PPM, DMC uses a Markov model for predicting the probability and Arithmetic encoding for encoding. PPM predicts the next character (byte), while DMC predicts the next bit in the file. This makes DMC well-suited for files with a non-homogeneous nature. It also adapts far quicker than PPM since it is not biased towards byte-orientated data [43]. DMC, however, is more resource-intensive than PPM and produces higher compression ratios. It has therefore not been widely implemented. PAQ8 includes DMC as one of its models for prediction [98].

Neural Networks

Neural networks excel in complex pattern recognition, which makes them a good option for developing a model for text compression. Typical compression algorithms such as Lempel-Ziv, PPM and BWCA are based on simple n-gram models (i.e. they exploit the non-uniform distribution of text sequences found in most data). Neural networks are able to exploit other patterns in the text. Schmidhuber [99], Long [47] and Mahoney [100] have all conducted research on using neural networks to create models for compression.

Offline and Online variants of neural networks can be used. In an online variant, the neural network would learn from the text which is being compressed, while in an offline variant, the network would be trained and hard-coded into the compressor and decompressor. In an online

variant one would not need to transmit the modified weights if both the compressor and decompressor start with the same initial conditions and use the same learning algorithm [99]. The disadvantage of a offline method is that it cannot adapt to specific trends within the text file which it is compressing (making it language specific) while the disadvantage with the online method is that it is more computationally expensive. Schmidhuber [99] shows an offline method which uses a three-layer neural network trained using back-propagation to compress German text. This neural network is pre-trained using forty articles from a German newspaper (Muchner Merker). Test sets from the same newspaper and another German newspaper, the Frankenpost, are tested. As expected, the Frankenpost revealed higher compression ratio. This method does however prove to be very computationally expensive (being 1000 times slower than other compression methods). Schmidhuber [99] also tests an online method. This method is slower than the offline method, but shows a significant improvement in the compression ratio. Long [47] uses alphabet re-representation followed by a four-layer feed-forward back-propagation neural network with a sigmoidal activation function to predict the probability of a symbol given a context. This probability is used by an arithmetic coder as in PPM. This technique produces lower compression ratios than PPMC or the method presented by Schmidhuber [99] using a context size of 10. However, for lower context sizes, PPMC achieves a lower compression ratio. Mahoney [100] uses a two-layer neural network. This is much faster than the other neural network compressors and produces a lower compression ratio than them. This technique also uses similar amounts of memory to PPM based compressors and shows similar compression and decompression times, but achieves slightly higher compression ratios. Neural networks are used to combine the models in PAQ8 [101].

2.3.5 Context-Mixing Algorithms

In statistical compression algorithms, a model of the data is used to estimate the probability of the character after or before the given context. These techniques result in a number of different models which can be used to represent the data. Combining models is an effective procedure used in machine-learning to improve the modelling of a data source. Context-mixing data compression algorithms use this technique and combine the results from a number of data compression models to improve the compression ratio achieved. Since context-mixing requires that each model is performed and then combined into a single statistic, context-mixing algorithms are much slower than other statistical compression algorithms such as Neural Networks, PPM, DMC and BWCA. The following section serves to explain the PAQ context mixing algorithms.

PAQ

PAQ is a family of context mixing compression algorithms developed initially by Matthew Mahoney and enhanced by a number of authors detailed on the PAQ Website maintained by Matthew Mahoney [98]. In the PAQ family of algorithms, the next bit in the data stream is predicted independently by a large number of models. The models' predictions are outputted in the form of a probability and a confidence for that probability. These models are combined using a neural network or weighted average and the data source is then arithmetically encoded.

PAQ1, developed by Mahoney, uses a weighted average of five different model, which include:

1. a bland model with 0 or 1 being equally likely
2. a set of order-1 through 8 non-stationary n-gram models
3. a string matching model for n-grams longer than 8
4. a non-stationary uni-gram, bi-gram and word model for English text (with a word being up to 8 letters)
5. a positional context model for data with fixed records

PAQ2 adds a secondary symbol estimation technique which uses a table of values based on previous estimation errors to determine an improved probability based on the output from the context mixer. PAQ3 in turn provides small improvements to this technique. All the versions of PAQ up until PAQ4 use fixed weights. PAQ4 mixes models using adaptive weights rather than fixed weights. PAQ5 includes six new models for analog data in data sources containing audio and images. It also includes word models for text. PAQ6 adds non-stationary models to PAQ5. This results in an improved compression ratio. From this point in the development of the PAQ family, many different authors produced variations which included extra models and speed optimizations. One such variation was released by Alexander Ratushnyak (PAQAR). It includes a number of extra models and SSEs for each mixer at the expense of far greater memory utilisation (3.34 times greater than PAQ6). Another variation, PAsQDa, was released by Przemyslaw Skibinski, which combines the text preprocessor, WRT (described in Section 2.5) with the PAQ6 program [98, 102]. PAQ7 is a complete re-write of PAQ6 which includes several items from the PAQAR and PAsQDa. It also includes models for `bmp`, `tiff` and `jpeg` files. PAQ8, meanwhile, provides a few more improvements from PAQ7 (including the DMC model as one of the models).

PAQ8 uses a neural network to combine a large number of models [98, 101]. It also includes an x86 executable model as well as preprocessors for executables and `jpeg` files [101]. The latest development at the time of writing (PAQ808) achieves the lowest compression ratio on six of the ten of the compression tests conducted by the Maximum Compression Benchmarking Website. It is rated as the top compression program overall in terms of compression ratio [26]. Malcolm Taylor's compression algorithm PWCM (PAQ Weighted Context Mixing), which forms the basis of his commercial program WinRK [103], achieves the second lowest average compression ratio in tests performed by Maximum Compression Benchmarking Website [26]. PWCM is also based on the context-mixing techniques used by PAQ family of algorithms. A member of the PAQ family has held the Calgary Corpus Compression challenge [104] since January 2004 and the Hutter Prize [105] since its inception. These are both compression challenges to achieve the lowest compression ratio. They are performed on the Calgary Corpus and 100MB of Wikipedia data respectively.

2.4 Word-based Compression Algorithms

"If text compression used units larger than single characters they could take advantage of longer range correlations and perhaps achieve better compression" [106]. Using word-based models instead of character-based models (i.e. a single element is known as a word and not a character) has been shown to be a successful method of improving the compression achieved by compression algorithms. This section therefore investigates word-based compression and discusses the advantages and disadvantages of these methods.

Bentley et al. [56] presents a word-based compression scheme which uses a word-based MTF scheme and compares it to word-based Huffman encoding and byte-level Huffman (excluding the size of the Huffman tree). For the word-based compressors, the text stream is divided into two sets of words. The first set of words consists of the longest sequence of alphanumeric characters, while the second set of words consists of the longest sequence of non-alphanumeric characters. This means that the decoder and encoder both alternate between the two sets of words. The position in the list for the MTF method is encoded using a Huffman code, which requires two passes over that data (as do both byte level Huffman and word based Huffman). The word-based techniques achieve on average 78.6 percent lower compression ratios than byte-level Huffman. The 256 element MTF cache method achieves an average compression ratio which is 0.5 percent lower than the compression ratio achieved by the word-based Huffman method.

Alistair Moffat [107] describes a word-based compression scheme which uses Arithmetic encoding instead of the Huffman encoding. Moffat also uses Arithmetic encoding combined with a variable-ordered word-based Markov models instead of MTF to provide further improvement. ZeroWord is a non-MTF scheme which uses the word frequencies as the basis for Arithmetic coding. As expected, it achieves a lower compression ratio than the zero order character compressor using Arithmetic encoding (described by Witten et al. [53]). It also achieves a lower compression ratio than the 256 element MTF cache method described by Bentley et al. [56]. Moffat [107] also describes a word-level variation of the PPM scheme. FirstWord and SecondWord use first-order and second-order Markov models respectively as their basis for Arithmetic encoding. Escape characters (as in PPM) are used to move between contexts. These both achieve lower compression ratios than their character-based counterparts (On average seen percent lower). The word-based methods, however, do use considerably more memory space than the equivalent character-based methods.

In word-based schemes, it is necessary to use a two-pass scheme or an adaptive scheme that dynamically expands the dictionary as new words are encountered. Horspool [106] introduces a word-based LZW. The character-based LZW compression algorithm initialises its dictionary with each character in the alphabet. Horspool's word-based LZW, however, does not initialise with the alphabet but rather builds the dictionary adaptively by using escape characters when a new word is encountered. This word-based LZW achieves an average of 9.63 percent lower compression ratio than the `compress` program (which uses a character-based LZW). Horspool [106] also tests a word-based compressor using a first-order Markov model, a word-based compressor using adaptive Huffman coding and a word-based compressor using Arithmetic Coding with a State-Based Context Model based on five parts of speech: Article; Noun; Adjective; Verb; and Other. This technique requires that the vocabulary is specified together with their parts of speech and can only be used for natural language. This advanced technique achieves an average compression ratio which is six percent lower than the average compression ratio achieved by the word-based LZW.

Another word-based LZW compression algorithm, WLZW, is introduced by Dvorsky et al. [108]. Unlike the previous methods, WLZW uses a single data structure for the dictionary - not one for alphanumeric and another for non-alphanumeric - with a restriction on the lengths of words and non-words. WLZW, unlike the word-based LZW presented by Horspool [106], uses two passes through the data. In the first phase of WLZW, a lexical analysis is performed whereby the words and non-words are determined and an alphabet is formed. In the second phase, compression is performed using the standard LZW algorithm and the alphabet determined in the first

phase [109, 110].

Isal et al. [111, 112] and Moffat and Isal [113] have developed a word-based BWCA. In their implementations they use de Moura's spaceless-words approach [114]. In this method, a single dictionary is used containing both words and non-words. Any non-words which are a single space between two words are omitted. The main issue in creating a word based BWCA is the development of a MTF coder which can operate effectively on such a large set of symbols. This is solved by the use of splay trees. Isal uses a four step process: Parse into spaceless words, perform block sorting, perform the MTF transform and perform entropy encoding on the result. The generated spaceless words dictionary is compressed using `bzip2` and appended to the payload. This results in a 17.5 percent lower compression ratio than `bzip2` on the entire file. Further work on the MTF stage using a forest of trees leads to an improvement in compression ratio of 3.5 percent. The stored dictionary can also be further compressed by eliminating common prefixes before compressing with `bzip2`. While the compression ratio is lower for the word-based BWCA, it is more than 50 percent slower than `bzip2` for compression and 100 percent slower for decompression.

In a later paper, Isal et al. [112] makes use of five different methods which involve the use of a forest of trees to perform the MTF in the word based BWCA. Moffat and Isal [113] provides a summary of these methods and the results using different entropy encoders. These methods do result in lower compression ratios; however, they take at least double the amount of time taken by the original method presented by Isal et al. [111].

2.5 Using text preprocessors to improve text compression

There are essentially two approaches to text compression: design a "text aware" compressor or creating a text preprocessor (i.e. filter) which transforms the original input into a representation which is more redundant for general purpose compressors [115]. Skibiniski [115] states that universal compression nowadays is reaching a plateau and as a result there is an increasing interest towards creating specialized compression algorithms.

A number of text preprocessors have been developed to improve text compression by general purpose compressors. These text preprocessors generally replace items located in a dictionary with a set of characters or perform a reversible transformation that will improve the compression ratio achieved by general purpose compressors. This section discusses the following preprocessors: * Encoding, LPT, RLPT, SCLPT, LIPT, StarNT, WRT and TWRT. It also elaborates on

other text preprocessing methods which have been developed.

* Encoding

Robert Franceshini and Amar Mukherjee [44] developed star encoding as a method of improving compression on text files. It is a generic, reversible transformation that can be applied to a source text to improve an existing, or backend, algorithm's ability to compress [116]. A dictionary of words is created and mapped onto words in the source text for replacement. This dictionary (termed the cryptic dictionary) is then kept separate from the compressed file [117].

Franceschini [44] presents two approaches for generating the dictionary. The first approach finds the unique words for replacement by looking for unique subsequences within each word and replacing the other "unnecessary" characters with *s. The second approach uses the frequency of the words in the source text to determine the word used for replacement. The words are sorted into descending order according to frequency, then assigned letters followed by *s, i.e. entries 1 to 52 in the sorted dictionary of length i get assigned a single letter followed by $i - 1$ *s, entries 53 to 2757 of length i get assigned a two-letter combination followed by $i - 2$ *s, etc. With both methods, the length of the word is maintained and the amount of *s is maximised. This compression method combined with a standard compression algorithm typically achieves between 5 percent and 20 percent improvement in compression ratio on English text files [118] using an electronic English dictionary which contains around 60 000 words (1.1 MB). Gains for PPM and DMC are found to be insignificant [117] and an average gain of only 1.6 percent is achieved on `bzip2`. This method also destroys the natural contextual statistics of letters and bi-grams in the English language which are exploited by most compression algorithms [116].

LIPT, LPT, RLPT and SCLPT

The short comings of *-encoding lead to further developments. The run-length encoding in `bzip2` replaces the long sequences of * characters by shorter sequence, meaning that the * characters do not get assigned short codes as intended. The second stage of `bzip2` also relies on the observation that a character sequence in the input file can be used to help predict the character preceding it. This property holds for the English language, but not for * encoded files because *s don't provide enough context to make a reasonable prediction. To deal with these short comings, the Length Preserving Transform (LPT) was developed. In this transform, the first character is a * and the last three characters identify the words. For words of more than four characters, a

suffix of "...nopqrstuvw" is used instead of *s. This produces a higher gain on `bzip2`. As mentioned in Section 2.3.3, the context information used in BWT is actually the reverse of PPM. PPM predicts the character which follows a given context, while BWT predicts the character which occurs before the given context. The Reverse Length Preserving Transform (RLPT) was therefore developed to improve the compression ratio for PPM. The filler string used for LPT is reversed [116] i.e. the prefix from "yxwvutsrqpon..." is used to fill the gap for words of more than four characters.

Star Encoding, LPT and RLPT all preserve length information, which results in a file which is the same size before and after preprocessing. The length information can be discarded to improve the compression ratio as long as the unique mapping information is preserved. The Shortened-Context Length-Preserving Transform (SCLPT) reduces the filler characters to the shortest unique string. This leads to a smaller transformation dictionary and in addition a lower compression ratio than the other methods [116].

Length Index Preserving Transform (LIPT) is a further improvement on these techniques [119, 120]. The first letter after the * represents the length followed by the offset in the dictionary as before. Using the LIPT preprocessor results in a lower compression ratio using the BWCA than the improvements by Arnavut [95], Balkenhol et al. [74] and Chapin [121]. Using the LIPT preprocessor with PPM results in a lower compression ratio than improvements by Effros [122] and Sadakane et al. [123]. It also results in a 13.44 percent lower compression ratio than word-based Huffman.

StarNT

Star New Transform (StarNT) was developed using the same techniques as LIPT. Sun et al. [124] identifies that the length of 82% of English words is greater than three. This transform therefore sets out to recode English words with a representation of no more than three symbols while maintaining context which can then be exploited by the back-end compressor. The resulting transform, StarNT, is remarkably similar to LIPT, however it uses a star to denote a word that is not in the dictionary instead of using a star to denote the beginning of a codeword for a word that is in the dictionary. This reduces the size of the transformed file, which results in an improvement in compression ratio. To improve performance, ternary search trees are used to store the dictionary. Using StarNT results in an improvement in compression time and compression ratio in comparison to LIPT. The average transform encoding and decoding times for StarNT are 23.7 and 15.1 percent lower than LIPT and the improvement in compression ratios for StarNT

are 6.65 percent higher than LIPT on average [124].

Other Text Preprocessing Techniques

There are other specific properties of textual data which compression programs and the preprocessors presented so far do not exploit. These include: a large amount of zero-frequency characters; numerous new line characters; commonly used bi-grams (two-letter combinations e.g. ea) and trigrams (three-letter combinations e.g. ing); common phrases; and capitalisation, .

Teahan and Clearly [125, 126, 127] encode special bi-grams called di-grams which represent a single sound such as 'ng' with a single character to improve the entropy of English. Teahan [126] also looks at using function words (which are separated into parts of speech such as: articles, prepositions, verbs, etc. and irregular forms) and a scheme which uses a dictionary of frequently used words in the English language.

Grabowski [128] uses text preprocessing techniques to improve the compression ratio achieved on the BWCA, introducing three new techniques: Capital Conversion; space stuffing; and EOL coding.

Capitalised words lead to two different contexts which are essentially identical. In Capital conversion, an escape character is used to replace the capital letter of a word if only its first letter is capitalised. This technique increases context dependencies and similarities between words which can be exploited by compression programs.

Rapid context changes can be costly in the BWCA due to the high MTF ranks which would be used for coding. Line breaks create rapid context changes within a source file. Space stuffing and EOL coding are methods which can be used to counter this problem. In Space stuffing, a space symbol is placed at the beginning of each line in order to change the context which follows the end of line symbol to one space instead of a variety of symbols. EOL coding is another technique to deal with line breaks. This technique replaces the EOL symbol with a space and encodes a reference to its location using the number of spaces since the last EOL symbol (Grabowski attributes this idea to Taylor). Grabowski [128] also investigates alphabetical reordering. Tests performed on Calgary and Canterbury corpus show that these techniques result in an average improvement in compression ratio of 3 percent.

Isal and Moffat [129] and Kruse and Mukherjee [77, 117] also use preprocessing techniques to improve the compression ratio of the BWCA. Isal and Moffat [129] propose prepending a parsing transformation to BWT which replaces n-grams and words with unused characters. Kruse

and Mukherjee [124] inturn proposes the use of dictionary encoding and replacing the 96 most frequently-used bi-grams with unused characters.

Abel and Teahan [130] highlights a number of different methods to exploit textual properties, including: capital letter conversion; end of line (EOL) coding; word replacement; phrase replacement; and alphabetical reordering. These methods use reversible transformations which are performed before compression and after decompression. In phrase substitution an augmented alphabet is used to replace di-grams, trigrams and four-character phrases as in Teahan [126]. Abel and Teahan [130] also investigate using word substitution. Using these techniques results in an improvement of between two percent and nine percent improvement in the compression ratio by `bzip2`, `gzip` and `ppmd` on ten files from the Calgary Corpus and two files from the Canterbury Corpus [130].

WRT and TWRT

The previous sections have shown that a popular way to increase text compression is to replace words with references to a text dictionary.

The following techniques are Incorporated into WRT (Word Replacing Transformation): StarNT (using a new dictionary which contains 80 000 word from Aspell, longer codes, and a modified dictionary ordering), capital conversion, dictionary mapping (using extra characters), separate mapping, q-gram replacement (using capitals since extra codes already used) and EOL coding.

Lempel-Ziv based compressors are significantly different to compressors which use the BWCA and PPM for compression. With Lempel-Ziv based compressors, the input data is parsed into a stream of matching sequences and single characters, while in BWCA and PPM characters are predicted based on a given context. The text preprocessors can be adapted for LZ77 based compressors by reducing the number of single characters to encode, decreasing the offset of matching sequences and decreasing the length of the matching sequence (hence virtually increasing the size of the LZ77 sliding window). This can be done by using a spaceless-words model, but not using q-gram replacement. Using capitalisation flags after a period, quotation or exclamations mark if they are not capitalised can also be used to reduce the number of capital conversion flags. No spaces are put after capitalisation flags to improve the results of Lempel-Ziv based compression programs. Tests performed on the Calgary Corpus and Canterbury corpus using PAQ6, PPMonstr, UHBC (large BWT), `bzip2` and `gzip` result in improvements of between two percent and six percent over the results achieved when using the same compression programs in combination with StarNT [115].

Dictionary-based preprocessing results in the most significant improvement in compression performance [131]. Since some files may contain two different data types, such as programming commands and English comments, the use of two static dictionaries optimised for programming commands and the English language would be useful. Two-level Word Replacing Transformation (TWRT) uses two dictionaries which are dynamically selected before the preprocessing begins. To do this, TWRT uses a two-pass process: Dictionary detection is performed in the first pass and WRT using the selected dictionaries is performed in the second pass. The first level (small dictionaries) are specific to some kind of data (the programming language when compressing code), while the second level dictionaries (large dictionaries) are specific for natural languages (such as English, Russian or French. This dictionary would be used for references when compressing code).

The best dictionaries are detected by taking the first 250 words of the dictionaries and counting their occurrences in the file. The small dictionary is not used if the frequency of words from the large dictionary is more than five time greater than the frequency of words from the small dictionary. Dictionary detection can be performed faster by dividing the file into five parts using only the first few KB instead of the entire file. The dictionary detection phase can also be used to decide whether other steps such as using a binary data filter, surrounding words with spaces and EOL coding should be used. This step may also be used to detect record-based files that can use record preprocessing to improve the compression ratio. TWRT is 1.7 percent slower than WRT, but produces a 1.7 percent lower compression ratio on the Calgary corpus. The dictionary detection also makes it multilingual, unlike WRT which uses a single language-specific dictionary.

Summary

Text preprocessors provide a substantial improvement in the compression ratio achieved by compression programs. The steps taken by text preprocessors to improve the compression ratio are simple transformations, such as the replacement of text with a single character or the replacement of words with new words. With the exception of TWRT, where language and dictionaries to be used are determined, the text preprocessors are rather simple. However, they still provide an improvement in compression ratio which is comparable to that achieved by the word-based compressors which use substantially more resources to obtain their compression ratios. When the preprocessors reduce the size of the file to be compressed, the compression programs take less time and use less memory utilisation for compression. The main advantage of using preproces-

sors is that they can be customised for a particular file type and used with standard compression programs.

2.6 Improving Log Compression

Log files consist of a number of lines of data separated by end of line characters. Each of these lines consist of a number of tokens, separated by spaces, which contain information that has been recorded by a program about its activity and errors that have occurred. These lines often exhibit a common format and common elements such as timestamps. More than one line is often logged about a single event which leads to a relationship between the lines. There are several patterns which are very frequent in log files. These include: IP addresses; timestamps; and URLs. Log files also contain a number of unused characters which can be utilised for cheap substitution of frequent sequences. Hatonen [132] notes that the original data set of tens of thousands of rows can often be represented by merely a couple of identified patterns and the exceptions not matching these patterns.

The compression of logs is, at present, performed by programs such as: logrotate; rotatelogs; httplog; IIS Log Archiver; Web Log Miser and SafeLog. These programs use general purpose compression algorithms such as DEFLATE [133], which do not exploit the redundancy that is specific to log files. Not much work has been done until recently on log file specific compression.

The next three sections investigate the improvements in log compression made by LogPack [133], the improvement in web log compression by Grabowski [134] and the compression of logs on the IBM Blue Gene/L [135].

2.6.1 LogPack

LogPack is a log compressor presented by Skibinski and Swacha [133], which performs a number of transformations on the log file before compressing with `gzip`, LZMA or PPM. `gzip` is included as an online compression algorithm, while others (LZMA and PPM) are included for highly effective offline compression.

LogPack consist of five different transforms. The first three transforms exploit the resemblance between neighboring lines (addressing local redundancy), while the other two exploit the global repetitiveness of tokens and token formats.

The first transform (which is the core transform) finds matches in the previous line. The sequence of matching characters is replaced by a single value $(128 + l)$ denoting the length of the sequence (l). If this value is encountered, it is simply preceded by an escape flag (127) to avoid confusion. This transform is the simplest, fastest and least effective out of three. The second transform builds on this transform. Unlike the first transform, it uses a block of lines and encodes the index of the line used at the beginning of the line. The third transform uses a similar approach. It stores a block of recent lines which begin with different tokens. This is the most effective out of the three transforms. The fourth transform replaces words which repeat frequently throughout the entire log. After transform three is performed, the output is parsed into words and the frequency of each word is calculated. Words which occur more than 64 times are then included in a dictionary (limited to 2MB). The fifth transform converts numbers, dates, times and IP addresses to their binary equivalents. It is stored using a flag which denotes the type followed by encoded data. These transforms do not exploit all the redundancy of log files; however, when used with `gzip`, LZMA and PPM compression algorithms, they provide up to 36 percent improvement in the compression ratio achieved on the test corpus.

2.6.2 Web log compression

Grabowski and Deorowicz [134] observe that fields which are located adjacently are similar is consecutive lines of a log file (This is a property of web log files and not all log files). Since compressing each field separately is a known way of increasing the compression ratio in databases, they apply this technique to log files. Timestamps are encoded using the difference from the previous timestamp (This is encoded using one byte. A value of 255 serves as an escape for larger values, after which a four byte difference is used) and IP addresses are encoded using four bytes. In any other field, common prefixes and suffixes are identified and a dictionary is formed. This dictionary is then encoded using an order-1 model and arithmetic encoding. An MTF transform is also used to exploit the recency effect which is typical in many of the fields due to consecutive log entries from same events. Grabowzki and Deorowicz [134] also note that fields are often correlated and share identical values. In the web log format, the access request and size are correlated and the IP address and User agent are also correlated. By removing this redundancy, the compression ratio is further improved.

In tests performed using `gzip`, `7zip`, `bzip2` and `ppmd`, this transform resulted in improvements in compression ratio of up to 65 percent (This improvement is 75 percent greater than the improvement by LogPack).

2.6.3 IBM Blue Gene/L Log Files

IBM Blue Gene/L is a large scale cluster designed to meet the computational requirements and data handling demands of large scientific and industrial applications. This supercomputer is capable of running hundreds of parallel jobs and transferring many gigabytes of data per second. It also produces a large amount of systems logs. Balakrishnan and Sahoo [135] propose the use of data compression to aid the storage and transfer of this large amount of data.

A number of trends can be observed in the blue gene/l system logs. These include that fact that: most columns in adjacent records tend to be the same; record ids are ascending integers in sequence; and the time stamps tend to increase. In addition, the log files exhibit a temporal locality of adjacent records as well as a spacial locality of the devices creating them. Balakrishnan and Sahoo [135] use a simple preprocessing step which compares the record with the previous record and encodes this difference. This preprocessing step produces a small stand-alone compression ratio; however, in combination with generic compression utilities, such as `bzip2`, `gzip` and `7zip`, it shows a good improvement in compression ratio. It produces 28.3 percent lower compression ratios and 43.4 percent lower compression times. The data-set used was 103 days of log data collected from a 64 node IBM Blue Gene/L installation (This data-set is 195MB uncompressed).

2.7 Compression Program used in this Thesis

Program	Ratio	Time	Memory	Program	Ratio	Time	Memory
compress	0.40513	1.5	-	PAQ1	0.22813	68.1	48
pkzip	0.32859	1.5	-	PAQ2	0.22357	93.1	48
gzip	0.32392	2	-	PAQ3Na	0.21791	147.2	80
bzip2	0.26367	5	-	PAQ4b	0.21394	139	84
7zip	0.26167	20	-	PAQ607fb	0.20209	556.4	206
ppmd	0.24088	5.5	-	PAQ7	0.19470	740	525
WinRK	0.18657	1326	700	PAQ8L	0.18935	1872	837

Table 2.2: Results for Compression Tests on Calgary Corpus performed by Mahoney [98]

Table 2.2 shows the compression ratio and times for compression programs on the Calgary corpus obtained from tests performed by Mahoney [98]. This table shows very low compression ratios

achieved by programs such as WinRK and the PAQ compression programs. These programs, however, are very slow and have hence been excluded from the tests in this thesis. WinRK an PAQ are between 3.42 and 93.6 times slower than `7zip`; however, they do achieve a lower compression ratio. The compression ratios achieved, however, are only 25 percent lower than the compression ratio achieved by `ppmd`. It is for this reason that this thesis does not test the PAQ compression algorithms. The smallest file in the test corpus is 14.8 times larger than the Calgary Corpus. This would thus lead to incredibly slow compression times as well as significantly large amounts of memory usage.

This thesis limits the scope of compression programs to ones which are available on many different platforms, have a console based implementation (for testing purposes) and are widely used. Previous work presented on using preprocessors to improve log compression (see Section 2.6) used `ppmd`, `7zip`, `gzip` and `bzip2` for testing purposes.

The following compression programs have been used for tests in this thesis: `ppmd`, `7zip`, `gzip`, `lzop`, `arj`, `zip`, and `bzip2`. These programs are a representative sample of compression programs. `ppmd` uses the PPMII algorithm, `bzip2` uses the BWCA, `lzop` is a fast compressor, `7zip` achieves low compression ratios using the LZMA algorithm (which is an LZ77-based compressor using Markov models) and `gzip`, `zip` and `arj` are popular LZ77 based compressors. The following section will thus give an overview of these compression programs and shows the algorithms which they use to perform compression. Technically, `ppmd` is the only statistical compressor. `7zip` and `bzip2` both use hybrids of sequential and statistical methods. From this point onwards, however they will be referred to as statistical compressors because of their statistical properties, low compression ratio and slow speeds.

2.7.1 ppmd

`ppmd` is a compression program written by Dmitry Shkarin that uses the PPMII algorithm and arithmetic encoding (refer to Section 2.3.2 for more details on PPM). Shkarin was interested in the speed and performance improvements of the abstract PPM models when he wrote this program, therefore it is not tuned for a particular data type. It achieves low compression ratios on text files, but comparatively higher compression ratios for non-homogeneous files (executables) and noisy analog data (sounds, pictures etc.) [35].

2.7.2 gzip

`gzip` is a commonly-used compression program which uses the DEFLATE algorithm described in Section 2.3.1. The `gzip` file format is defined by Deutsch in Request for Comments (RFC) 1952 [32]. It contains a ten byte header which includes a `gzip` file identifier (0x1f8b), the compression method, a modification time and some flags to identify further header fields which follow after the initial ten byte header. This is followed by further header fields, the compressed payload and an eight byte footer containing a cyclic redundancy check (CRC) value using the CRC-32 algorithm and the original (uncompressed) payload size.

2.7.3 7zip

`7zip` is an archive format designed by Igor Pavlov. It has an open architecture, so it can support any new compression algorithms. Current algorithms which can be used with this format include: PPMD; Bzip; DEFLATE and Lempel-Ziv Markov Chain Algorithm (LZMA). LZMA is the default compression method of the 7z format. It has a high compression ratio, a variable dictionary size (up to 1GB) and requires little memory for decompression [136]. LZMA is thus essentially an improvement on the LZ77 algorithm backed by a range encoder. It supports several variants of hash chains, binary trees and Patricia tries (radix tree) and may be coupled with specific preprocessors designed for executable files such as Branch Call Jump (BCJ) and Branch Call Jump Version 2 (BCJ2) [37].

2.7.4 lzop

Lempel-Ziv-Oberhumer (LZO) is a data compression library written in C. It offers fast compression and very fast decompression with low memory usage. It was designed with speed in mind. The naming convention for the various algorithms is in the format `LZOxx-N`, where `N` is the compression level. 1-9 indicates fast, standard levels which use 64KB memory, level 99 offers better compression and is still reasonably fast, but uses more memory (256KB), while level 999 is supposed to achieve nearly optimal compression at the cost of slow compression times and greater memory usage. LZO is a block compression algorithm based on LZ78. It uses a sliding dictionary, compressing a block of data into matches and runs of non-matching literals [34].

2.7.5 arj

`arj` (Archiver Robert Jung) is a compression program, similar to PKZIP, which uses a proprietary compression format covered in part by US patent 5140321 [137]. Its attractive feature is being able to add, delete and/or modify files in a multi-volume archive. It specifies five methods: 0 (stored); and 1 (lowest ratio) to 4 (fastest speed). `arj` uses a LZ77 based algorithm with hashing functions [137]. More detailed description of the header fields may be found in the `arj` technical information [36].

2.7.6 zip

`zip` is a simple archive format in which each file is compressed separately. Each of the files in the archive may use a variety of compression methods. These include [33]:

- Shrinking (a variant of LZW)
- Reducing (compressing repeated byte sequences and applying probability-based encoding)
- Imploding (compressing repeated byte sequences with a sliding window, then compressing the result using Shannon-Fano coding)
- DEFLATE (LZ77 sliding windows of up to 32KB)
- Enhanced DEFLATE (LZ77 sliding windows of up to 64KB)
- PKWARE Data Compression Library Imploding
- BWCA

`zip` is one of the most popular and widely used archive format. There are many different programs available for creating `zip` archives. InfoZip is used in the tests conducted for this thesis since it is the standard `zip` program included with many different linux distributions.

2.7.7 bzip2

`bzip2` is an implementation of the BWCA developed by Julian Seward (based on the work performed by Fenwick [82] and Seward [87, 92]). The BWCA used by `bzip2` consists of run length encoding, the BWT and MTF coding, followed by RLE0 and entropy encoding using Huffman coding.

2.8 Summary

This chapter has presented a brief discussion on the state-of-the-art for text based data compression. It described the compression programs which are used in this thesis and presented their algorithms. It explained why the PAQ and WinRK compressors are not included among the testing programs and showed their results on the Calgary Corpus (which is the standard corpus for evaluating compression programs). This chapter has showed that preprocessors produce an improvement in compression ratio which is comparable to the improvement by enhancements to the compression programs. Section 2.6 revealed that the redundancy present in log files can be exploited to provide up to 65 percent improvement in compression ratio as well as detailing the recent works in log compression. The next chapter investigates the performance of the seven compression programs on a corpus of log files.

Chapter 3

Data Compression

Data compression is an effective method for reducing the quantity of data. Previous work, described in the previous chapter, showed that the use of data compression on English Text files lead to a reduction in filesize of between 80 and 90 percent. The maximum compression benchmarking site [49] shows the standard compression programs which this chapter is evaluating reduce the size of a 20MB log file by between 91.82 percent and 97.18 percent.

This chapter investigates using data compression to reduce the size of log files and seeks to evaluate the resources used and the performance of each compression algorithms. It first presents the methodology used for conducting tests. This is followed by a presentation and comparison of the results for the various compression programs. This chapter concludes by selecting the best compression programs for the four different scenarios described in Section 1.3.

3.1 Testing Methodology

As mentioned in Section 1.3 and described further in Section 2.7, the compression programs investigated in this thesis are `7zip`, `bzip2`, `lzop`, `gzip`, `zip`, `arj` and `ppmd`. In order to provide effective analysis of these compression programs, the compression time, decompression time, memory usage and the compression ratio for each level of the compression program need to be recorded.

This section presents the methodology used to obtain these results. It describes the corpus used, the test used to determine the compression times, decompression times and compression ratios and the test used to determine the memory utilisation. The testing process is split into two tests

so that the process of monitoring the memory will not effect the compression and decompression times which are recorded in the first test.

3.1.1 Test Corpus

The seven compression programs are tested on the a test corpus which consists of six text log files (a generic syslog, a squid access log, an apache access log, a postfix mail log, a kernel messages log and a ftp log) and a binary LibPCap data file.

Log Filename	Description	Filesize (bytes)
all.log	Generic syslog logs	62033795
access.log	Squid access logs	228045444
httpd-access.log	Apache web server access logs	88824626
maillog	Postfix mail server logs	48213167
messages	Kernel messages	46824482
xferlog	FTP logs	44236972
darknet-200508.cap	LibPCap Packet Capture	122867860

Table 3.1: Corpus of different log files used for testing data compression programs

Table 3.1 shows the filename, a description and the filesize of this files which are contained in the corpus used in this chapter for testing the data compression programs.

3.1.2 Compression and Decompression Tests

Since the compression and decompression times vary slightly between different runs, each file needs to be compressed and decompressed a number of times and an average obtained. The variation in time is due to factors which are out of the control of this project. The results in this chapter are obtained by calculating the average over five iterations. The testing procedure for each file in the corpus is as follows:

1. Compress the file five times using each compression program and record the time results for all five times
2. Decompress the obtained compressed files five times using the relevant decompressor and record the time results for all five iterations

3. Calculate the averages for each compression level of each program
4. Calculate the compression ratios for each compression level of each program
5. Record these results for further analysis

This process results in a number of files containing the original five samples for each compression level of each compression program and a file containing the average results for each compression level and the compression ratio achieved. The times are recorded using `time` which is available in most unix-based operating systems. This testing process is automated using Perl scripts which may be found in Appendix E.1.1.

3.1.3 Memory Utilisation tests

The amount of memory utilised is obtained by performing another iteration of compression and decompression on each file and monitoring how much memory is allocated to the process. The peak memory utilisation is the same for each of the five iterations of compression and decompression, so only a single iteration is necessary to obtain the results. The testing procedures is as follows:

1. Start the memory-testing script which records the memory-usage information for the process using `top` (a resource monitor which is available in most unix-based operating systems) and outputs it to a file
2. Compress and decompress each file with each compression program
3. Split the file containing the memory usage information and obtain the memory-usage statistics for each compression program (`top` records the command used to launch a process. This command contains the compression program and compression level used and can hence be used to split the file)

This process is automated using Perl scripts (which may be found in Appendix E.1.2). The memory usage statistics recorded are a combination of the resident memory and shared memory used by the compression program.

3.2 Results and Analysis

Test results are obtained for each file on the corpus. In order to analyse the large amount of results generated (65 compression level - compression program pairs for seven files) and make conclusions about the performance on the corpus, 2 metrics are used: The average (or sum); and the weighted average (or weighted sum).

The weighted average for a result on the corpus is calculated by giving larger files a lower weighting to minimise the effect of file size on the average (since larger files obtain a higher value). The weighting which is used is the average on the corpus divided by the file size. The weighted average is described formally as follows:

Consider a corpus C, containing n files. Let x_i be the result on the i-th file in the corpus ($i = 1, 2, 3, \cdots n$). Let s_i be the size of the i-th file in the corpus ($i = 1, 2, 3, \cdots n$). Then the weighted average for the result is calculated as follows:

$$\frac{\sum_{i=1}^{n} \left(\frac{\frac{\sum_{j=1}^{n} s_j}{n}}{s_i} \times x_i \right)}{n}$$

The next four sections will analyse the compression ratio, compression and decompression times, transfer times and memory utilisation on the test corpus. These results are obtained using the methodology presented in the previous section. Note that a summary of the statistical methods used can be found in Appendix A and the definitions of the terms such as compression ratio, compression time amd decompression time may be found in Section 2.1.4.

3.2.1 Compression Ratio

File	Entropy (bpc)
all.log	5.41494
access.log	5.35091
httpd-access.log	5.54863
maillog	5.44533
messages	4.86430
xferlog	5.32017
darknet-200508.cap	6.24628

Table 3.2: Shannon Entropy (in bpc) for each log file in the corpus

Table 3.2 lists the Shannon Entropy for each of the files in the corpus. It is calculated by using the probability of a single character in the file as its probability of that character occurring (i.e. the character-based entropy).

Algorithm	Correlation Coefficient
ppmd	0.89139
bzip2	0.88456
gzip	0.80520
zip	0.80520
arj	0.79545
7zip	0.77966
lzop	**0.64346**
Overall	0.81903

Table 3.3: Pearson correlation coefficient for Average Compression Ratio vs Shannon Entropy

Table 3.3 shows the Pearson correlation coefficient between the Shannon entropy listed in Table 3.2 and the average compression ratio for each compression program. These results show that character-based entropy is a good indication of compression performance. The overall correlation coefficient (0.81903) indicates a strong linear relationship between these two variables which means that as the entropy increases, so too does the average compression ratio. All the compression programs with the exception of `lzop` show a strong positive correlation. `lzop` is a fast compression program with nine compression levels. The first six compression levels are designed to be fast but achieve a high compression ratio, while the other three compression levels are designed to achieve a lower compression ratio with slower compression and decompression times. If `lzop` is split into 2 categories, `lzop(1)` for the first six compression levels and `lzop(2)` for the last three compression levels, then the resulting correlation coefficients are 0.58133 and 0.76197 for `lzop(1)` and `lzop(2)` respectively. `lzop(2)` therefore exhibits a strong positive correlation, while `lzop(1)` exhibits a weak positive correlation. This skews the results for `lzop`. Character-based entropy is therefore a good indication of the compression ratio for all levels of the compression programs except the first six levels of `lzop`.

Table 3.4 shows the rankings of the compression programs for the lowest average compression ratio on the corpus and the overall average compression ratio on the corpus. These results show the statistical compressors (`ppmd`, `7zip` and `bzip2`) achieving both the lowest compression ratios and the lowest average compression ratios. `ppmd` achieves the lowest compression ratio, while

	Lowest	Average
ppmd	1 (0.05824)	1 (0.07099)
7zip	2 (0.06418)	2 (0.07414)
bzip2	3 (0.07627)	3 (0.08150)
gzip	4 (0.10148)	4 (0.11185)
zip	5 (0.10148)	5 (0.11185)
arj	6 (0.10345)	6 (0.11791)
lzop	7 (0.12857)	7 (0.15417)

Table 3.4: Compression Ratio Ranking on entire corpus

`bzip2` achieves the highest compression ratio out of the three statistical compressors. `zip` and `gzip` achieve the lowest compression ratio out of the Lempel-Ziv-based compressors. `gzip` and `zip` both use the DEFLATE compression algorithm, which is a combination of LZ77 algorithm and Huffman encoding. `gzip` achieves a lower average compression ratio than `zip` (0.000002 lower) since it has slightly smaller headers. `gzip` and `zip` achieve a 0.6 percent lower average compression ratio than `arj`. `lzop` achieves the highest average compression ratio out of all the programs. The average compressed size when using `lzop` is more than double that of `ppmd` (which achieves the lowest compression ratio) and a third larger than `arj` (which achieves the second-largest compressed size out of all the programs). Detailed information of the performance of the different compression levels is contained in Appendix E.3.

The corpus contains a LibPCap data file (which is a binary data file) as well as six text log files. Many compression programs perform well on text (and hence on the log files), but achieve comparatively poor results on binary data.

	Lowest	Average
ppmd	1 (0.04401)	1 (0.05724)
7zip	2 (0.05674)	2 (0.06718)
bzip2	3 (0.06254)	3 (0.06826)
gzip	4 (0.09027)	4 (0.10145)
zip	5 (0.09027)	5 (0.10145)
arj	6 (0.09221)	6 (0.10911)
lzop	7 (0.11873)	7 (0.14676)

Table 3.5: Compression Ratio Ranking on corpus without LibPCap data file

Table 3.5 shows the rankings of the compression algorithms on the corpus without the LibPCap

data file. The results all show an improvement from the compression ratios shown earlier in Table 3.4. This is to be expected since the average compression ratio on LibPCap data file is 0.16573, while the average on the other text log files in the corpus is 0.09278. There is, however, no change in the rankings of the compression programs.

	Average		LibPCap
	Difference	StdDev	Mean Ratio
lzop(1)	0.04362	0.00059	0.20403
7zip	0.04744	0.00664	0.12425
lzop	0.05189	0.01277	0.19864
lzop(2)	0.06843	0.00105	0.18788
zip	0.07277	0.00692	0.17422
gzip	0.07277	0.00692	0.17422
arj	0.07557	0.00610	0.18268
bzip2	0.09266	0.00374	0.16092
ppmd	0.09625	0.01119	0.15349
ppmd*	0.09890	0.00391	0.14944

Table 3.6: Difference between the compression ratio achieved on the corpus and the compression ratio achieved on the LibPCap File

Table 3.6 shows the average difference in the compression ratio achieved by each program in the corpus and the compression ratio achieved by each program on the LibPCap data file. Since `lzop` and `ppmd` show a high standard deviation for their results, their results have been split into categories to reduce the standard deviation. The first six compression levels of `lzop` are designed to be very fast and yield a higher compression ratio than the last three levels (which are designed to be much slower and achieve a lower compression ratio). `lzop` has thus been split into two categories: `lzop(1)` for the first six compression levels; and `lzop(2)` for the last three compression levels. Both these categories show a lower standard deviation. The first compression level of `ppmd` (order 2) achieves an average compression ratio of 0.15958, which is 0.06162 higher than the average for the second compression level (order 3 - 0.09796) and 0.09491 higher than the average over compression levels 2-15 (0.06467). `ppmd`* excludes this outlier and shows both a lower standard deviation and a lower average than `ppmd`.

The first six levels of `lzop` (`lzop(1)`) return the lowest difference, but also the highest average compression ratio for LibPCap data files. This shows how the first six levels of `lzop` do not

exploit the redundancy contained in the files as effectively as the other programs, which results in it returning similar results for the binary LibPCap data file and the other text log files. `7zip` achieves the best results out of the statistical-based compressors. It achieves the second-lowest average difference and the lowest average compression ratio (3.4 percent lower than `ppmd`) on the LibPCap data file. `lzop(1)` also achieves the lowest average difference, however, the average compression ratio for `lzop(1)` is 0.20403, which is nearly twice the ratio achieved by `7zip` (0.11596).

`gzip` and `zip` achieve approximately average results for the difference, The average difference over all the compression programs is 0.07295. `gzip` and `zip` both achieve an average difference of 0.07277. The average compression ratio achieved by `zip` and `gzip`, however, is above the overall average of 0.16573.

`arj` achieves below average results for the difference. As with the results for the overall corpus, it achieves the highest compression ratio out of the LZ77-based compression algorithms. It does, however, achieve a lower difference between the LibPCap files and text log files than `ppmd` and `bzip2`. These two programs achieve poor results for the LibPCap file in comparison to the other programs relative to their results on the other files in the corpus. On text-based log files, `bzip2` achieves a compression ratio which is 3.3 percent lower than the ratio achieved by `zip` and `gzip`, whereas the compression ratio achieved on the LibPCap data file is only 1.3 percent lower than the ratio achieved by `zip` and `gzip`. `ppmd` achieves the highest difference between the compression ratio for the LibPCap data file and the text-based log files, emphasised by the fact that when the poor results for order 2 are removed (`ppmd`*), the difference increases. The reason for the marked difference in performance is that `ppmd` is not designed for any data types other than text. It was designed as a program to test the effectiveness of the PPMII algorithm and includes no optimisations for binary data [35]. Both `bzip2` and `ppmd`, however, do achieve low compression ratios on the LibPCap data file (even though their performance is not as impressive as on the text based log files).

Summary

In all the compression programs (with the exception of `lzop(1)`, where compression levels 2-5 are constant and compression level 1 achieves a lower ratio), there is, on average, a decrease in compression ratio as the compression level increases (as compression method increases for `7zip` and as compression level decreases for `arj`). In all the compression programs (with the exception of `ppmd`) the highest compression level (lowest for `arj`) achieves the lowest compression ratio.

The highest compression ratio is achieved by the lowest compression levels (highest compression level for `arj`) for all compression programs except `lzop(1)` where compression levels 2-5 achieve the highest compression ratio. `bzip2`, `lzop(2)`, `arj`, `gzip`, `zip` and `ppmd` (up to the first local minimum) all display a logarithmic decrease in compression ratio as the compression level increases (decreases for `arj`).

There is not much change in the compression program's rankings for compression ratio on the binary LibPCap file. `7zip` achieves the lowest ranking on the LibPCap file. The rest of the rankings remain the same. Closer inspection reveals that `bzip2` and `ppmd` have a higher difference between the compression ratio for the LibPCap file and the text based log files than the other compression programs; however, this has little effect on the compression rank.

Program	Level	Rank
ppmd	9	1
7zip	10	12
bzip2	9	20
gzip	9	34
zip	9	35
arj	1	40
lzop(2)	9	53
lzop(1)	1	60

(a) Lowest Rank

Program	Average Rank
ppmd	13
7zip	19.2
bzip2	25.89
gzip	43.78
zip	44.78
arj	47.25
lzop(2)	54.67
lzop(1)	62.5

(b) Average Rank

Table 3.7: Rankings of compression ratios achieved by each compression programs

The averages obtained for each compression level of each compression program (65 averages in total) are ranked according to the compression ratio achieved. Table 3.7 (a) show a summary of the lowest rank by each compression program as well as the compression level which achieves it. Table 3.7 (b), meanwhile, shows the average rank for each of the compression programs. These results show that the statistical compressors achieve the lowest compression ratios, `lzop` achieves the highest compression ratios and `arj` achieves the highest compression ratio out of the LZ77-based compression programs.

To achieve lower compression ratios, more time is required for compression. The next section investigates the times taken to achieve these compression ratios. It shows the average compression, decompression and total times for each of the compression programs on the test corpus.

3.2.2 Compression and Decompression Times

Larger files take longer to compress and decompress than smaller files. The test corpus contains seven files which range from 44MB to 228MB (with an average of 82MB). In order to not allow poor performance on larger files to skew the results, a weighted average is used where the times are scaled according to their size in relation to the average size. Interestingly, this weighted average exhibits a Pearson correlation coefficient of 0.99870 with the average and a Chi-squared goodness-of-fit test conducted produces a p-value of 0.96 which means that the null hypothesis fails to be rejected and, therefore, the weighted average and average exhibits the same distribution. It is, therefore, inconsequential as to which metric is used. This section uses the weighted average metric. Whenever a compression or decompression time is referred to in this section, it is referring to the weighted average calculated on the test corpus.

		lzop	zip	gzip	arj	7zip	ppmd	bzip2
	Comp	1 (0.76)	3 (1.74)	2 (1.67)	4 (2.81)	6 (17.96)	5 (11.98)	7 (30.7)
L	Decomp	2 (1.33)	1 (1.06)	3 (1.56)	4 (1.76)	5 (2.53)	7 (14.84)	6 (5.67)
O	Total	1 (2.24)	2 (2.89)	3 (3.24)	4 (5.08)	5 (20.79)	6 (27.3)	7 (36.37)
	Avg	1.33	2	2.67	4	5.33	6	6.67
	Comp	4 (20.27)	2 (7.67)	1 (7.02)	3 (8.18)	7 (111.9)	5 (20.46)	6 (72.24)
H	Decomp	3 (2.25)	1 (1.17)	2 (2.21)	4 (2.28)	5 (2.92)	7 (23.09)	6 (9.4)
I	Total	4 (21.61)	2 (8.73)	1 (8.58)	3 (10)	7 (114.57)	5 (43.55)	6 (81.64)
	Avg	3.67	1.67	1.33	3.33	6.33	5.67	6
	Comp	4 (6.13)	2 (4.02)	1 (3.65)	3 (5.08)	7 (58.23)	5 (16.38)	6 (53.03)
A	Decomp	2 (1.66)	1 (1.11)	3 (1.74)	4 (1.93)	5 (2.68)	7 (19.02)	6 (8.2)
V	Total	4 (7.79)	1 (5.12)	2 (5.39)	3 (7.01)	7 (71.19)	5 (35.39)	6 (61.23)
	Avg	3.33	1.33	2	3.33	6.33	5.67	6

Table 3.8: Ranking of compression time, decompression time and total time (weighed average in seconds) for all compression programs

Table 3.8 shows the rankings for the lowest, highest and average times for each compression program. The weighted average (in seconds) is contained in brackets after the ranking. In this table, figures for the compression times, decompression times and total times are shown. The average ranking is also shown for each compression program. As mentioned, `lzop` favours speed over compression ratio. The results show `lzop` producing impressive speeds. `lzop` achieves the

lowest compression time for all files in the corpus and the lowest decompression time for five out of the seven files in the corpus. The decompression time for the squid access log causes both the weighted average and the average to be higher for `lzop` than `zip`. This is because `lzop` takes double the time that `zip` takes to decompress the squid access log file. `lzop` achieves the lowest total time for all files in the corpus except the squid access log. The `lzop` program has two classes of compression levels. The first six compression levels (1-6) - `lzop(1)` - are fast and achieve a high compression ratio, while the last three compression levels (7-9) - `lzop(2)` - are slower but achieve a lower compression ratio. This causes `lzop` to achieve a higher average rank than `zip` and `gzip`.

	Lowest Time			Highest Time			Average Time		
	Comp	Decomp	Total	Comp	Decomp	Total	Comp	Decomp	Total
lzop(1)	0.76	1.41	2.24	1.02	2.25	3.27	0.85	1.81	2.65
lzop(2)	10.89	1.33	12.23	20.27	1.42	21.61	16.68	1.36	18.05

Table 3.9: Compression Time, Decompression Time and Total Time (all in seconds) for the two classes of lzop compression levels

Table 3.9 shows the lowest, highest and average times for the two different classes of `lzop` compression levels. It reveals how `lzop(1)` achieves lower times than `zip` and `gzip` and that `gzip` and `zip` achieve faster times than `lzop(2)`, despite it achieving lower compression ratios, `gzip` achieves a higher rank than `zip` for all compression times and total times, but achieves a lower rank for decompression times. The statistical compressors - `7zip`, `bzip2` and `ppmd` - achieve slow compression and decompression times. `ppmd` achieves the highest decompression times, but achieves lower compression times than `7zip` and `bzip2`. Detailed information on the performance of the different compression levels of the seven compression programs is contained in Appendix E.3.

Summary

The results presented in this section generally confirm that, in order to achieve a lower compression ratio, an increased amount of time is required. `zip`, `gzip` and `arj` show an exponential increase in compression times in order to obtain a logarithmic increase in compression ratio. For `lzop`, `zip`, `gzip`, `arj` and `bzip2`, the compression level which achieves the highest compression ratio achieves the fastest time and the compression level which achieves the lowest compression

ratio achieves the slowest compression time. `7zip` levels 8-10 have slower compression times than `bzip2`, causing `bzip2` to achieve a better average overall. However for all compression levels of `bzip2`, there exists at least two compression levels of `7zip` which achieve faster compression times and lower compression ratios than `bzip2` (This is detailed in Appendix E.3). Five out of the seven programs exhibit approximately constant decompression times. `ppmd` and `bzip2` are the only compression programs tested which do not exhibit a constant decompression time. Both show a linear increase and `ppmd` achieves slower decompression speeds than compression speeds.

Program	Level	Comp	Decomp	Total	Average
lzop(1)	4	1	12	1	4.67
zip	1	9	5	10	8
gzip	1	7	21	4	10.67
arj	4	14	29	17	20
lzop(2)	7	30	10	29	23
7zip	1	29	40	30	33
ppmd	1	29	51	34	38
bzip2	1	49	42	39	43.33

Table 3.10: Ranking of lowest compression time, decompression time and total time

For each compression level of each program (65 levels in total), the average compression, decompression, and total times over the corpus are calculated. These are ranked from highest to lowest. Table 3.10 shows the rank of the compression time, decompression time and total time for the compression level which achieves the lowest total time. This table shows that the first six levels of `lzop`, `lzop(1)`, are the fastest, followed by the three LZ77-based compressors. `gzip` achieves faster compression times and slower decompression times than `zip` and `arj` is the slowest out of the three LZ77-based compressors. The statistical compressors - `bzip2`, `7zip` and `ppmd` - achieve compression times than the LZ77-based compressors. Overall, `lzop(1)` achieves the fastest times and `bzip2` the slowest times.

3.2.3 Transfer Times

The transfer time for a particular compression program is the time taken to transfer the file at a given transfer speed from one point to another with the use of compression and decompression

at each end. This is calculated by taking the time to transfer the compressed payload at a given transfer speed and then adding it to the time taken to compress and decompress the original payload. Transfer times, therefore, combine the results of compression ratio, compression time and decompression time into a single practical metric which can help to choose a compression program for a given scenario. This section investigates the transfer times for transfer speeds of between 1 KBps (Kilobytes per second) and 30 000 KBps.

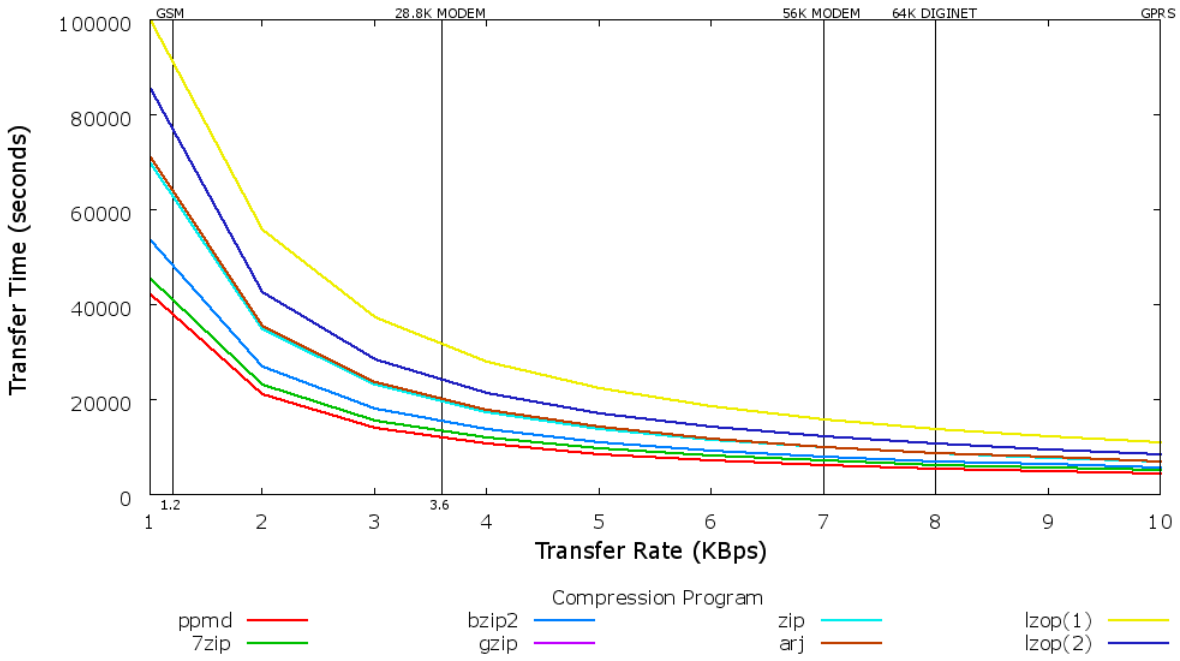


Figure 3.1: Lowest transfer times for all compression programs at transfer speed between 1 KBps and 10 KBps

Figure 3.1 illustrates the lowest transfer times for the different compression programs at transfer speeds between 1 KBps and 10 KBps. These are the speeds which are achieved by communication technologies such as GSM, Dial up Modems, DIGINET and GPRS. In this figure, the rank of the compression programs does not change. The statistical compressors (`ppmd`, `7zip` and `bzip2`) achieve the lowest transfer times. `ppmd` achieves the lowest transfer times and `lzop` the highest, which are more than double the transfer times achieved by `ppmd`. For these transfer speeds, `ppmd` provides up to 93.263 percent improvement over transfer times without compression.

Figure 3.2 shows the lowest transfer times for transfer speeds between 10 KBps and 1000 KBps. In this figure, the transfer speeds achieved by common communication technologies such as EDGE, 3G, ADSL and HSDPA are shown for reference. The rank achieved by the programs

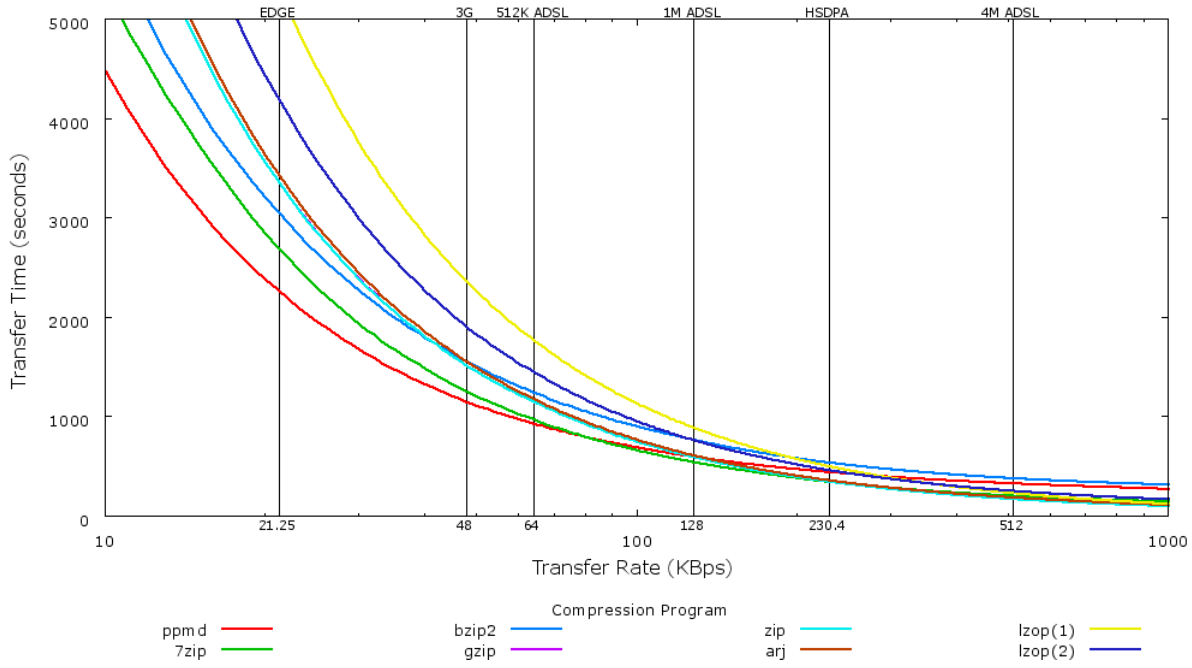


Figure 3.2: Lowest transfer times for all compression programs at transfer speed between 10 KBps and 1000 KBps

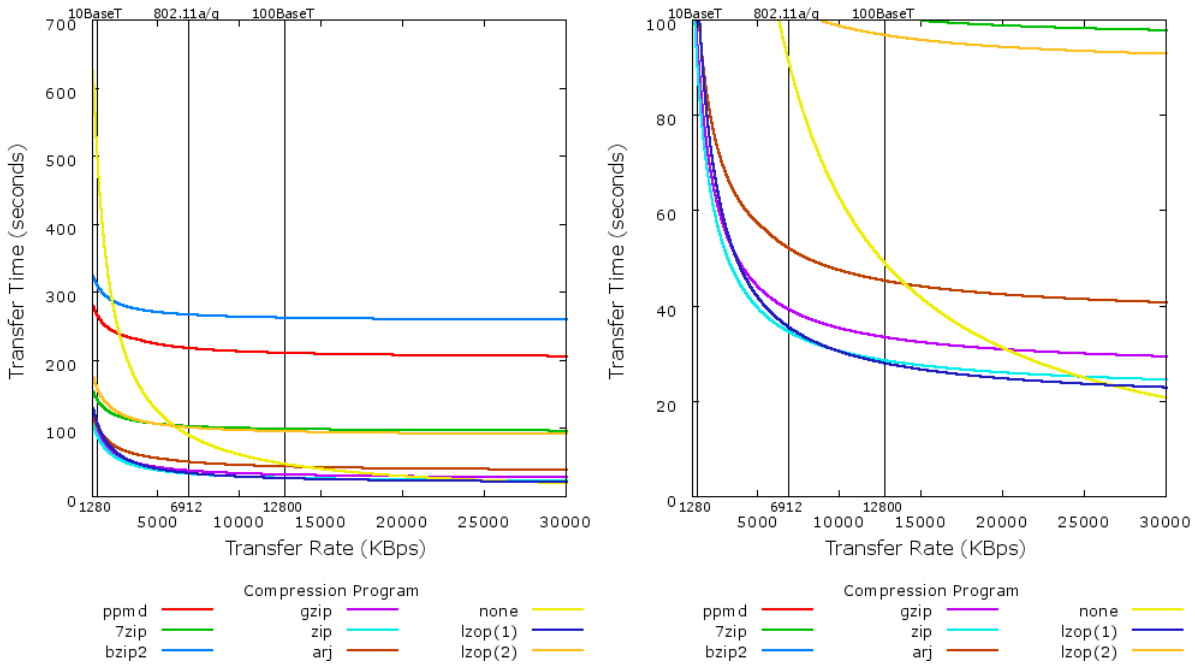
between 1 KBps and 10 KBps remains until transfer speeds of 19 KBps. `ppmd` achieves the lowest transfer time until speeds of 81 KBps, at which point, `7zip` becomes the fastest. It achieves the lowest transfer time until speeds of 235 KBps. From 236 KBps, `zip` achieves the fastest transfer times. From 306 KBps, the three LZ77-based compression programs achieve the first three positions. The slower statistical compressors - `ppmd`, `7zip` and `bzip2` - achieve lower compression ratios and higher total times but produce faster transfer times. This shows that when slow transfer speeds are being used, higher compression ratios are more valuable than faster speeds.

As previously mentioned, there are two classes of compression algorithms for `lzop`. `lzop(1)`, which consists of the first six compression levels, is designed to be fast and achieve a higher compression ratio while `lzop(2)`, which consists of the last three compression levels, achieves lower compression ratios and slower times. `lzop(2)` achieves lower transfer times than `lzop(1)` up until 354 KBps. `lzop(1)` achieves the highest transfer times up until transfer speeds of 194 KBps. Since it achieves high compression ratios, it does not produce fast transfer times for these transfer speeds.

This graph also shows that `gzip` achieves a lower transfer time than `zip` for transfer speeds below 18KBps. As previously noted, `zip` achieves a lower total time than `gzip` and a similar

compression ratio (0.000002 higher than `gzip`). This causes `zip` to achieve a lower transfer time than `gzip`. `arj` achieves the highest transfer speed out of the LZ77-based compression programs (this being on average, 6.21 percent higher than `gzip`). This is to be expected since it achieves a higher compression ratio and higher total times than both `gzip` and `zip`.

For all transfer speeds, `bzip2` is the worst performing statistical compressor. It achieves the third fastest transfer speed up until 42 KBps. Thereafter its high total time causes it to drop below `zip` and `gzip`. Between 42 KBps and 194 KBps, `arj` and `lzop` also become faster than `bzip2`. From this point, `bzip2` achieves the slowest transfer time. It achieves the highest overall average of 555.729 seconds, 15.99 percent higher than `lzop`, which achieves the second highest average.



(a) Lowest Transfer Times

(b) Zoomed in Image

Figure 3.3: Lowest transfer times for all compression programs at transfer speed between 1000 KBps and 30000 KBps

Figure 3.3 (a) illustrates the transfer speeds achieved by the compression programs for transfer speeds between 1001 KBps and 30 000 KBps. It also shows the transfer speed without the use of compression (Labeled none). Figure 3.3 (b) zooms in on the transfer times between 0 and 100. It clearly shows the improvement of `lzop(1)` and when the use of the LZ77-based compression and `lzop(1)` no longer results in an improvement from the transfer time using no compression. `lzop(1)` moves from last position (194 KBps) to producing the lowest transfer times from 9615

KBps. Up to this point `zip` produces the fastest transfer times. This illustrates how, at higher transfer speeds, the total compression time becomes more important than the compression ratio which it achieved. It is for this reason that the LZ77-based compression programs and `lzop(1)` achieve the fastest transfer times at higher transfer speeds. As the transfer speed increases, the use of compression provides a smaller improvement in transfer time.

Program	Speed (KBps)	Speed (Mbps)
lzop(1)	26567	207.55
zip	24690	192.89
gzip	20176	157.63
arj	13989	109.28
lzop(2)	5977	46.70
7zip	5902	46.11
ppmd	2612	20.41
bzip2	2163	16.90

Table 3.11: Maximum transfer speeds at which compression program shows improvement

Table 3.11 shows the maximum transfer speeds for each compression program which results in an improvement in transfer time. On 100BaseT networks which operate at a maximum of 100 Mbps (12800 KBps), the use of LZ77-based compression programs and `lzop(1)` still provides an improvement in transfer time. Using no compression, it takes 48.91 seconds to transfer the corpus, while with `lzop`, `zip`, `gzip` and `arj` it takes 28.11, 28.71, 33.57 and 45.42 seconds respectively to transfer the corpus. This means that, at this high speed, the compression ratio is low enough to compensate for the compression and decompression times and still provide a decrease in time. At this speed, the other compression programs take between 1.98 (`lzop(2)`) and 5.40 (`bzip2`) times longer than the use of no compression. This shows how the time taken for compression and decompression by the statistical compressors can affect the transfer times at high transfer speeds. `ppmd`, which achieved the lowest transfer time for 3G (11.30 times faster than no compression and 2.04 times faster than `lzop(1)`), is 4.33 times slower than no compression and 8.57 times slower than `lzop(1)` at 100 Mbps. This table shows that compression provides a reduction in transfer time for transfer speeds of up to 207.55 Mbps. The other advantage of using compression is that less bandwidth is used (due to the reduction of the payload size) and the "on the wire" time is shorter. Detailed results of the transfer times for each compression level of the compression programs can be found in Appendix E.3.

Summary

Generally, this section has shown that the compression ratio achieved is more important than the total time for lower transfer speeds. As the transfer speed increases, the importance of the total time increases. This means that the lower compression levels (higher in `arj`) of the compression programs produce faster transfer times at higher transfer speeds, while the higher compression levels (lower in `arj`) of the compression programs produce faster transfer times at lower transfer speeds.

	1.2	3.6	7	8	10	21.25	
	GSM	28.8K	56K	64K	GPRS	EDGE	Average
ppmd	1	1	1	1	1	1	1
7zip	2	2	2	2	2	2	2
bzip2	3	3	3	3	3	3	3
gzip	4	4	4	4	4	5	4.17
zip	5	5	5	5	5	4	4.83
arj	6	6	6	6	6	6	6
lzop(2)	7	7	7	7	7	7	7
lzop(1)	8	8	8	8	8	8	8
none	9	9	9	9	9	9	9

Table 3.12: Ranking of transfer time for compression programs at slower connection speeds

Table 3.12 shows the ranking of the compression programs for the maximum transfer speeds of GSM, 28.8 Kbps modem, 56 Kbps modem, 64 Kbps DIGINET, GPRS and EDGE. The last column contains the average ranking for these transfer speeds. The table shows the statistical compressors achieving the lowest transfer times for all these levels (and hence the lowest average ranks), with `ppmd` achieving the fastest transfer time for all these transfer speeds. At such transfer speeds, all the compression programs shows an improvement in transfer time.

Table 3.13 shows the ranking of the compression programs for the maximum transfer speeds of 3G, 512 Kbps ADSL, 1 Mbps ADSL, HSDPA and 4 Mbps ADSL. The last column contains the average ranking for these transfer speeds. This table shows the improvement by the LZ77-based compression programs for faster transfer speeds (`zip` has ranks of one and `gzip` has a rank of two for 4 Mbps ADSL). `7zip` becomes the fastest statistical-based compressor and the only one in the top three for the average. The fall of `ppmd` can also clearly be seen for speeds greater than

	48	64	128	230.4	512	
	3G	512K	1M	HSDPA	4M	Average
7zip	2	2	1	1	4	2
zip	3	3	2	2	1	2.2
gzip	4	4	3	3	2	3.2
ppmd	1	1	4	5	7	3.6
arj	5	5	5	4	3	4.4
lzop(2)	7	7	6	6	6	6.4
bzip2	6	6	7	8	8	7
lzop(1)	8	8	8	7	5	7.2
none	9	9	9	9	9	9

Table 3.13: Ranking of transfer time for compression programs at broadband connection speeds

1 Mbps. For 3G and 512 Kbps it has a rank of one, but for the rest achieve ranks of between four and seven. The improvement of `lzop(1)` can also be seen for 4 Mbps, where it achieves a rank of five. In summary, for all transfer speeds greater than 1Mbps, `zip` and `gzip` are better options than the statistical compressors.

	1280	6912	12800		>6912
	10BaseT	802.11a/g	100BaseT	Average	Average
zip	1	1	2	1.33	1.5
lzop(1)	4	2	1	2.33	1.5
gzip	2	3	3	2.67	3
arj	3	4	4	3.67	4
lzop(2)	6	6	6	6	6
7zip	5	7	7	6.33	7
none	9	5	5	6.33	5
ppmd	7	8	8	7.67	8
bzip2	8	9	9	8.67	9

Table 3.14: Ranking of transfer time for compression programs at LAN and Wireless connection speeds

Table 3.14 shows the ranking of the compression programs for the maximum transfer speeds of 10BaseT, 802.11a/g and 100BaseT. The second-last column contains the average ranking for these transfer speeds, while the last column contains the average for 802.11a/g and 100BaseT. `zip` and `lzop(1)` achieve the fastest times, with `lzop(1)` achieving a rank of one for 100BaseT.

For both 802.11a/g and 100BaseT, only the LZ77-based compressors and `lzop` provide an improvement in transfer time. The transfer time without the use of compression achieves a rank of five for these transfer speeds.

3.2.4 Memory Utilisation

When considering a compression program, it is important to consider the resources used as well as the performance of the compression program. The previous section investigated the times taken by the compression programs for compression and decompression. This is both a performance and a resource issue. The memory resources used by the compression program to perform compression is also important when picking a compression program for a given scenario. This section will therefore investigate the memory-usage of each compression program when compressing and decompressing the test corpus.

	Minimum	Maximum	Average
gzip	0.9	0.93	0.92
lzop(1)	1.18	1.22	1.19
zip	1.28	1.3	1.29
lzop	1.18	1.55	1.31
arj	1.4	1.42	1.41
lzop(2)	1.55	1.55	1.55
bzip2	1.84	7.3	4.57
ppmd	7.95	116.69	72.91
7zip	5.14	578.09	169.89

(a) Compression

	Minimum	Maximum	Average
gzip	0.8	0.8	0.8
lzop(1)	1.05	1.05	1.05
lzop	1.05	1.05	1.05
lzop(2)	1.05	1.05	1.05
zip	1.28	1.28	1.28
arj	1.29	1.3	1.3
bzip2	1.25	4.3	2.78
7zip	4.16	58.28	19.01
ppmd	7.94	116.83	72.94

(b) Decompression

Table 3.15: Memory utilised (in MB) by compression programs when compressing and decompressing the corpus

Table 3.15 (a) shows the minimum, maximum and average memory usage by each program for compression while Table 3.15 (b) shows the minimum, maximum and average memory-usage by each program for decompression. The memory-usage for each compression level of the compression programs is calculated by taking the average of the peak memory usages when compressing or decompressing each file in the corpus using that compression level. This table shows that the LZ77-based compression programs - `zip`, `gzip` and `arj` - have low memory

utilisation. Sections 3.2.1 and 3.2.2 noted that `arj` was the slowest and achieved the highest compression ratios out of the LZ77-based compression programs. This table shows that `arj` also uses the largest amount of memory out of the LZ77-based compression programs. `arj`, therefore, shows the poorest performance and uses the most resources out of the LZ77-based compression programs.

Out of all the compression programs tested, `gzip` uses the least memory for both compression and decompression. It shows a maximum memory utilisation of less than 1MB. This is less than the minimum memory utilisation for all the compression programs. `zip`, which achieves similar compression ratios and faster total times, uses on average 49.42 percent more memory than `gzip`.

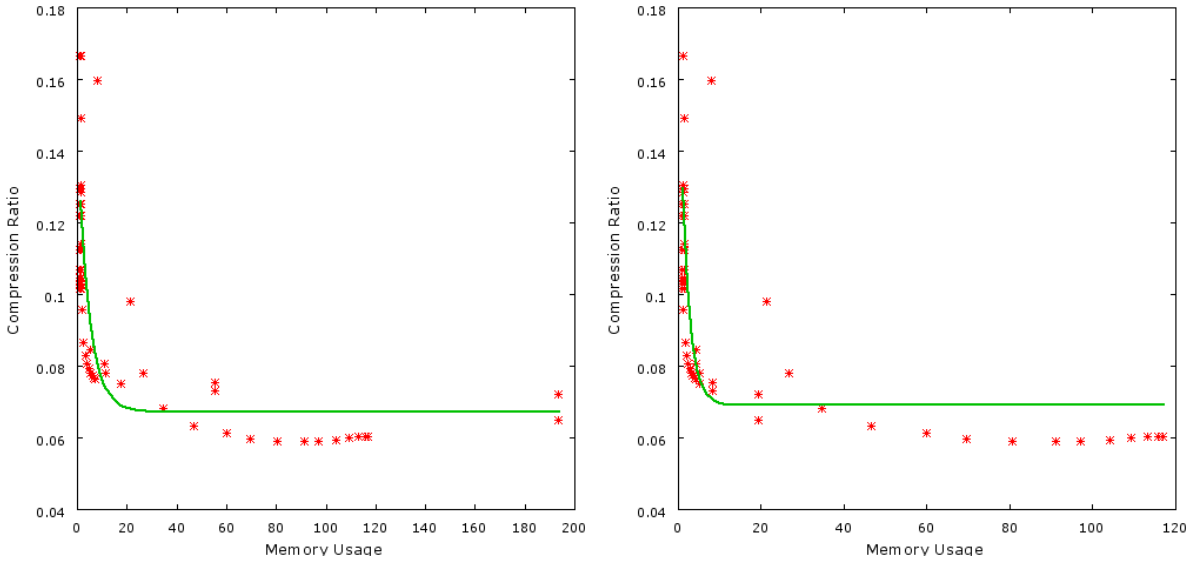
`lzop(1)` uses the second least amount of memory for both compression and decompression (this being 30 percent more than `gzip` on average). `lzop(1)` and `lzop(2)` use the same amount of memory for decompression, but `lzop(2)` uses an average of 1.55 MB, which is 30.25 percent more than the average for `lzop(1)`. `zip` and `arj` both use less memory than `lzop(2)` for compression as well as achieving lower compression ratios.

The statistical compressors - `7zip`, `ppmd` and `bzip2` - use between 2.13 and 184.66 times more memory, on average, than the rest of the compression programs. Out of the statistical compressors `bzip2` uses the least memory, while `ppmd` uses (on average) 15.95 times more memory than `bzip2` and `7zip` shows the highest average memory utilisation (2.33 times the average achieved by `ppmd`). All the compression programs show less memory utilisation for compression than decompression (except `ppmd`).

Summary

Generally, the compression levels which achieve the lowest compression ratios are expected to utilise the most memory.

Figure 3.4 (a) and Figure 3.4 (b) show scatter plots of the memory usage versus the compression ratio achieved for compression and decompression respectively. The curves present in both figures are fitted using the function $a\,e^{-\frac{x}{b}} + c$ and the Least Squares method to obtain the best fit. For compression a, b and c are 0.0713394, 0.21286 and 0.0674409, and for decompression a, b and c are 0.0911754, 1.96128 and 0.0693726. Figure 3.4 (a) and Figure 3.4 (b) both illustrate that an increase in memory is required to achieve a lower compression ratio. This amount of memory required to achieve a lower compression ratio grows exponentially as the compression ratios decreases. However, `zip` and `gzip` indicate an exception: as the compression level increases, both



(a) Memory usage for Compression (b) Memory Usage for Decompression

Figure 3.4: Compression Ratio versus Memory Utilisation for all compression levels of all compression programs

the compression ratio and the memory-usage decreases. For `bzip2`, `7zip` and `lzop`, the lowest compression levels - which achieve the highest compression ratios - use the least memory for compression and the highest compression levels - which achieve the lowest compression ratios - use the most memory for compression. Compression level nine achieves the lowest compression ratio for `ppmd`. It therefore does not make sense to use any compression level above nine for `ppmd`

Program	Comp	Decomp
gzip	5	5
lzop(1)	12.5	13.83
zip	20	24
arj	26.5	30.5
lzop(2)	30	14.33
bzip2	36.44	34.78
7zip	52.4	46.7
ppmd	53.13	57.07

(a) Average Rank

Program	Level	Rank
gzip	9	1
lzop(1)	2-6	10
zip	9	16
arj	4	25
lzop(2)	7-9	29
bzip2	1	32
7zip	1	37
ppmd	1	42

(b) Least Comp. Memory

Program	Level	Rank
gzip	1-9	1
lzop(1)	1-6	10
lzop(2)	7-9	10
bzip2	1	19
zip	1-9	20
arj	4	29
7zip	1	40
ppmd	1	45

(c) Least Decomp. Memory

Table 3.16: Average Rank for Memory Utilisation and Least Memory Utilisation by each compression program for compression and decompression

since they are slower and utilise more memory to achieve a higher compression ratio.

For each compression level of each program (65 levels in total), the memory-usage for compression and decompression are ranked from highest to lowest. Table 3.16 (a) shows the average rank of each program for compression and decompression memory usage. Tables 3.16 (b) and 3.16 (c) show the compression levels which use the least amount of memory for each program. They also show the rank of those compression levels. The tables show that the statistical compression programs - `7zip`, `ppmd` and `bzip2` - use more memory than the LZ77-based compressors. In addition, they reveal that `arj` uses the most memory for compression and decompression out of the LZ77-based compressors and that `gzip` uses the least memory out of all the programs.

3.2.5 Summary of Results

		Time				Memory			Overall
	Ratio	Comp	Decomp	Total	Average	Comp	Decomp	Average	Average
gzip	43.78	15.89	21.11	17.56	18.19	5	5	5	22.32
zip	44.78	17.67	5	15	12.56	20	24	22	26.44
lzop(1)	62.5	3.5	24.5	4.33	10.78	12.5	13.83	13.17	28.81
arj	47.25	20.75	19	21.75	20.5	26.5	30.5	28.5	32.08
lzop(2)	54.67	38.67	12.67	31.33	27.56	30	14.33	22.17	34.8
bzip2	25.89	55.67	46	54.33	52	36.44	34.78	35.61	37.83
ppmd	13	39.47	58	42.6	46.69	53.13	57.07	55.1	38.26
7zip	19.2	53	36.5	51.7	47.07	52.4	46.7	49.55	38.61

Table 3.17: Average ranks of the results for each compression program

Table 3.17 shows the average ranks of the results for each compression program. This table is a summary of results achieved by each compression program. Overall, `gzip` and `zip` are the most well rounded compression programs. They both achieve average compression ratios, low compression times and low memory utilisation. The average rank for time is calculated by taking the average of the average ranks achieved for compression time, decompression time and total time. The average rank for memory utilisation is calculated similarly by taking the average rank of the memory utilisation for compression and decompression. The overall average is calculated by averaging the average rank for time, memory utilisation and compression ratio. This table shows `lzop(1)` achieving a higher average rank than `lzop(2)`. `lzop(2)` achieves

higher compression times and greater memory utilisation, so this is expected. The statistical compressors - `bzip2`, `7zip` and `ppmd` - achieve the three highest overall average ranks. They achieve the three lowest average ranks for compression ratio; however, their slow times and high memory utilisation causes them to achieve a high overall average rank.

3.3 Scenarios

Section 3.2 showed that each compression program has their own set of strengths and weaknesses. Some compression programs were fast and achieved high compression ratios (such as `lzop`) while others had high compression times, slower decompression times and achieved low compression ratios (such as `7zip`). For each monitoring scenario a different compression program is thus likely to be more appropriate. Some of the possible scenarios include:

- Filtering logs through to the central point for analysis (where latency is not important, but there is a desire to use as little bandwidth as possible)
- Real-time monitoring (where a minimum point-to-point time is desired, using as little resources (memory, time and bandwidth) as possible)
- Quick access, storing it compressed and later decompressing it for analysis (where fast decompression is desired, but the compression time does not matter. In addition, there is a secondary desire to use as little bandwidth as possible)
- Low system-time-usage for compression and decompression (where fast compression and decompression times are desired and the compression ratio does not matter)

The following section investigates these scenarios and makes recommendations on which compression program is most applicable to the needs of that scenario.

3.3.1 Filtering logs through to the central point for analysis

In this case, the compression and decompression times are not very important but a low compression ratio is desired. This makes `ppmd`, `7zip` and `bzip2` the best options, since they achieve positions one, two and three respectively when ranking compression levels, which in turn shows the lowest compression ratios. Since the logs are going to be used for analysis at a central point,

it is also desirable to have a quick decompression time. The desired memory utilisation is not clear and will have an effect on the choice. `bzip2` is the best option if a low memory utilization is desired; however, it achieves the slowest times and highest compression ratio out of the three programs. `7zip` and `ppmd` are both very memory intensive. `7zip` has the fastest decompression time and it also achieves the best results for LibPCap data (making it significantly versatile). The best option is therefore `7zip` due to its high compression ratio, fast decompression times and versatility.

3.3.2 Real-time monitoring

For real time monitoring, a low point-to-point time is desired. `zip`, `gzip` and `ppmd` are therefore the three best options, depending on the speed of the point-to-point links. `ppmd` uses more memory for compression and decompression than `zip` and `gzip` and in addition it does not achieve low compression and decompression times like `zip` and `gzip`. `ppmd` does, however, achieve a much lower compression ratio and superior point-to-point times for the slower links. `zip` produces the lowest transfer times for higher transfer speeds and is not very resource intensive. `7zip` is also an option since it achieves the fastest or second-fastest transfer speeds for lower transfer speeds. Essentially, then, it depends on the resources available and the transfer speed of the point-to-point link. If the transfer speed is low, then `7zip` or `ppmd` is the best option, otherwise `zip` would be the preferred choice.

3.3.3 Quick access to stored compressed information

If the compression time does not matter but quick decompression is desired, then `7zip` is a very good option. It achieves a low compression ratio, provides quick decompression, but results in high compression times. `lzop`, `gzip`, `arj`, and `zip` all have faster decompression times but achieve higher compression ratios than `7zip`. Out of these, `gzip` achieves the lowest compression ratio and `lzop` the highest (`lzop` being nearly 25 percent larger (24.548) than `gzip`). `gzip`'s compressed payload is also a third larger than `7zip`, but exhibits faster compression and decompression times. The best choice would be `zip` due to the low compression ratio and fast decompression time.

3.3.4 Low system time usage for compression and decompression

For quick compression and decompression without taking any regard to size, `lzop` would undoubtedly be the best choice since it exhibits both the fastest compression and decompression times. `zip` and `gzip` also perform really well in this respect and achieve lower compression ratios than `lzop`. Since fast compression and decompression is desired (i.e. low system time), `lzop` would be the best choice. Should the compression ratio be taken into account, `zip` would be the best option due to a lower total time than `gzip`.

3.4 Summary

This chapter shows data compression to be a good method for reducing the quantity of data. It provides an improvement in point-to-point times for transfer speeds of up to 207 Mbps and results in up to 98 percent reduction in the amount of data. This substantial decrease in amount of data to be stored can aid with the compliance to regulations such as HIPAA, Sarbanes-Oxley and the ECT Act, which require the storage of system log files for accountability and legal purposes. The use of compression also decreases the "on the wire" time which can save expensive bandwidth at the expense of the memory and processing time required for compression and decompression. The next chapter investigates how semantic knowledge can be used to improve the performance of standard compression programs. It presents the results of preprocessors which replace timestamps and IP addresses with their binary equivalents and also presents an analysis of the semantic knowledge present in maillog files.

Chapter 4

Initial Semantic Investigation

Chapter 3 showed that the use of standard compression programs results in a large reduction (up to 98 percent) in the size of the log files. It also showed that the time to transfer a log file between two points was improved for transfers speed of up to 207 Mbps.

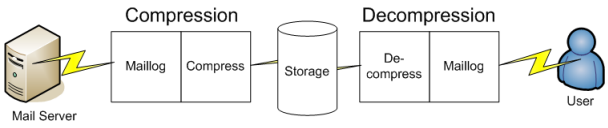


Figure 4.1: Mail server storing log file and user accessing it without preprocessing

Figure 4.1 shows a diagram illustrating a typical scenario of a mail server producing a maillog and then compressing it. Once logs have been created they are usually rotated at a set period and then archived. When the log files are rotated, they are compressed (usually using `bzip2` or `gzip`). The logs are rotated using programs such as: logrotate; rotatelogs; syslog; IIS Log Archiver; Web Log Miser; and Safelog [7]. These compressed logs are usually viewed using tools such as `bzcat` which outputs the contents of a file compressed with `bzip2` to the screen.

Log files contain a large amount of semantic information which is exploited by compression programs. These compression programs use trends such as common sequences and the distribution of characters to compress the log files. Log files are text files and hence are rather verbose and do not use all the available ASCII characters. Skibinski [131] states that universal compression nowadays has reached a plateau and hence there is an increase in creating specialised compression algorithms. Text preprocessing can be used to exploit specific qualities which are present in a text file. Section 2.5 shows a number of text preprocessors which have been developed to

improve the text compression by general purpose compressors by exploiting known semantic knowledge.

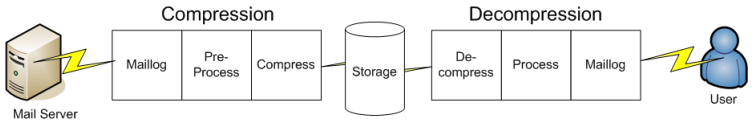


Figure 4.2: Mail server storing log file and user accessing it with preprocessing

Figure 4.2 shows a diagram of the same scenario with the use of a preprocessor. The preprocessor gets called before the compression program is run. When the compression program is run, it will perform compression on the preprocessor's output. In order to decompress the file or view the file, a process needs to be run to reverse the transformation done by the preprocessor after decompression.

An issue that may arise from this is the added time for compression due to the preprocessing of data and the processing of data to reverse the transformation performed by the preprocessor. The time taken for this process could negate the improvements that the preprocessor makes to the compression ratios and compression times achieved. This problem can be solved by integrating the preprocessor into the logging process. In this manner it becomes transparent, and hence there is no time overhead at the time of rotation since the preprocessing is performed as the log is generated. `syslogd` can be configured to output the log data to a named pipe. A process can then be run which performs the preprocessing on the data retrieved from the named pipe and outputs the transformed data to the log file. This process would result in a log file that is not human readable in its stored form. However, this can easily be solved by creating a tool which performs the reverse transformation and outputs the data to the screen in a similar manner to `bzcat`.

This chapter investigates the use of two simple preprocessors which transform the timestamps and IP addresses to their binary equivalents. It then investigates the semantic knowledge which is present in maillog files. Definitions for the metrics (compression time, decompression time, compression ratio and payload ratio) used in this chapter may be found in Section 2.1.4.

4.1 Timestamps and IP addresses

Two elements which are common in many system log files are timestamps and IP addresses. These are used to denote times of entries and hosts which are involved in the activity being

reported. They are stored in text format and are hence more verbose than their binary equivalents. Considering the number of repetitions, converting these to binary results in a large reduction. The log files contained in the corpus presented in the previous chapter use three different types of text log formats: syslogs; apache access logs; and squid access logs. These all contain timestamps in a fixed position and a number of IP addresses in varying positions within a line of the log file. These different types of log formats also use different types of timestamps. Their time are formatted as follows:

syslog: `MMM DD HH:MM:SS`
e.g. `Aug 10 00:02:06`

apache access log: `DD/MMM/YYYY:HH:MM:SS +TZTZ`
e.g. `12/Nov/2005:16:20:55 +0000`

squid access log: `TTTTTTTTTT.TTT SSSSSS` (Unix timestamp with millisecond resolution and size)
e.g. `1140897487.640 139452`

In the squid access log, the timestamp is directly followed by a size which can also be included when transforming the timestamp. The timestamps for the syslog, apache and access logs take up 15, 26 and 21 bytes respectively within the log file. Regardless of the timestamp format, by converting them to binary their size can be reduced. Their binary equivalents use 4, 8 and 8 bytes respectively. This transformation results in a saving of 11, 16 and 13 bytes respectively for each occurrence of a timestamp (which is large considering that timestamps occur on every line).

As mentioned, IP addresses are contained within the log files to denote host information and connection details. They are stored in a textual form consisting of 4 numbers (between 1 and 255) separated by dots e.g. 192.168.0.1. These numbers can be converted to their binary equivalents which each occupy one byte. This leads to a reduction of between four bytes (in the case of an IP address which consist of four single digit numbers, such as 1.1.1.1) and 12 bytes (in the case of an IP address which consists of four three digit numbers, such as 255.255.255.255).

Unlike the timestamps, which occur in a fixed position within a line on every line, IP addresses are located in different positions within the line and not every line contains an IP address. These positions need to be identified, especially in the case where other preprocessing techniques such as word replacement are being used. Skibinski et al. [133] suggest the use of an escape character to identify IP addresses which are converted to their binary equivalents. This means that the IP addresses are encoded using five bytes rather than four bytes (the extra byte is used for the

escape character). This results in a reduction of between 3 bytes and 11 bytes when storing an IP address in its binary form.

The remainder of this section describes the methodology which is used to replace IP addresses and timestamps with their binary equivalents and the improvements in compression ratio and compression time achieved when using this methodology.

4.1.1 Methodology

Since the timestamp is always found at a fixed point in the line, no escape characters need to be used to identify the timestamp. IP addresses, however can occur in a number of different positions in the line. For this reason, an arbitrary escape character (in this case ASCII character 5 is used - which is not found in the file) is included before the IP address. After this escape character, the four bytes which follow represent the IP address. In the case where there is no character which doesn't occur in the file, the ASCII character 5 can be escaped when it occurs (in the same manner as \ is escaped as \\ in text files) so that the IP address can be identified. The aim of the initial semantic investigation is to determine the effects that a simple preprocessor will have on the compression ratio and compression times of the standard compression programs.

Preprocessing

The preprocessing is performed by calculating a number which represents the entity. This number is stored in binary in the file in place of the entity which it represents. The numbers are calculated as follows:

Let the date be divided up into year, month and day. `MMM` is used to represent the month from 0 to 11, `DD` is used to represent the day from 0 to 30 and `YYYY` is used to represent the year. Let the time be divided up into hours, minutes, seconds and timezone. `HH` is used to present the hour from 0 to 23, `MM` is used to represent the minutes from 0 to 59, `SS` is used to represent the amount of seconds from 0 to 59, `MS` is used to represent the amount of milliseconds from 0 to 999, and `TZ` is used to represent the timezone (the amount of hours ahead or behind GMT). For the squid access log, let `T` represent the unix time, `MS` the millisecond resolution and `S` the size.

Algorithm 1 Algorithm to calculate binary date for syslog file

The binary representation of the number returned by:

```
return (((MMM*31)+DD-1)*5184000)+(HH*3600)+(MM*60)+SS
```

Algorithm 1 shows how the binary representation of the timestamp for syslog files is calculated. The number returned in this algorithm is in the range $[0, 1928447999]$, which can be stored in less than four bytes.

Algorithm 2 Algorithm to calculate binary date for squid access log

The binary representation of `T` followed by the binary representation of the number returned by:

```
while (S>4294966)
{
     remove the last character from S and insert it at the beginning
     of the line
}
return MS+(S*1000)
```

Algorithm 2 shows how the binary representation of the timestamp and size for squid access log files is calculated. `T` is in the range $[0, 1140897491]$, while `MS` is in the range. The number returned by the algorithm (MS+(S*1000)) is in the range $[0, 4294966999]$. This means that the binary number can be stored in less than eight bytes.

Algorithm 3 Algorithm to calculate binary date for apache web log

The binary representation of the number returned by:

```
Date = (YYYY*12*31)+(MMM*31)+DD
Time = (HH*3600)+(MM*60)+SS
return (((Date*86400)+Time)*12)+|TZ|*(TZ/|TZ|)
```

Algorithm 3 shows how the binary representation of the timestamp for apache web log files is calculated. This algorithm returns a signed integer. The range for `Date` is not clear since it is dependent on the range of the year. The number returned by the algorithm has a range of $[-3856895999999, +3856895999999]$ for `YYYY<10000`, which requires 41.81 bits (5.22 bytes) to

store, and a range of $[-836075519999, +836075519999]$ for `YYYY<2100`, which requires 39.60 bits (4.95 bytes) to store. This means that the binary number can be stored in less than eight bytes.

Calculating the number for the IP address is simpler. It is calculated by representing each of the four numbers in the IP address as a byte. The preprocessing described in this section is performed by Python scripts which may be found in Appendix E.1.3.

Conducting the tests

The tests are divided into two categories: The preprocessor which only transforms the timestamp (T); and the preprocessor which transforms both the timestamp and the IP address (T&IP). These preprocessors transform the file with their respective transformations and produce smaller binary files. These smaller files are then compressed and decompressed five times and the results recorded. This is the same procedure that was used to obtain the results found in chapter 3.

Test corpus

Four out of the seven files of the corpus introduced in the previous chapter are used as the test corpus for the initial semantic investigation.

Log Format	Log files (in order of size)
apache	httpd-access.log
squid	access.log.1
LibPCap	darknet-200508.cap
syslog	all, mail.log, message, xfer.log

Table 4.1: Log formats used by the seven log files in the corpus

The seven files in the corpus use four different formats for logging. The log formats used by the seven files are shown in Table 4.1. Since these tests are focused on text preprocessing, the darknet capture file is not tested. Four out of the seven files in the corpus use the syslog format. Only the two largest files the generic syslog file (all) and the mail log file (mail.log) are included in the test corpus for this section. The corpus for this section therefore contains four text log files: the squid access log (access.log.1); the apache access log (httpd-access.log); the generic syslog (all); and the mail log (mail.log).

4.1.2 Results and Analysis

File	Original Size	T	T&IP
access.log.1	228045444	205782323 (0.9)	189764939 (0.83)
all	62033795	56442347 (0.91)	54807466 (0.88)
httpd-access.log	88824626	80700506 (0.91)	76829332 (0.86)
mail.log	48213167	44013239 (0.91)	43146796 (0.89)

Table 4.2: Size of files in bytes (ratio to Original Size) after preprocessing with T and T&IP

Table 4.2 shows the size of the files following the transformations. The figures contained within the brackets are the ratio of that file to the original file before transformation. The file sizes of the original file are also included in the table. As expected, this table shows T&IP producing a larger reduction in filesize than T. This is since T does not perform any reductions on the IP addresses. This table also shows that these simple preprocessors produce reductions in the filesize of between 9 and 17 percent.

File Name	File Type	Avg Line Length	Reduction	Percentage
all	syslog	132.133	11	8.32
access.log.1	squid	142.406	16	11.24
httpd-access.log	apache	217.669	13	5.97
mail.log	syslog	136.754	11	8.04

Table 4.3: Percentage reduction in line length (line length and reduction in bytes) when transforming timestamp to binary

Table 4.3 shows the percentage reduction in the average line length by transforming the timestamp into binary. This explains the results for the ratios achieved by T. access.log.1 shows the highest reduction, while the other files show a similar percentage reduction. Table 4.2 shows a difference between the ratios achieved by T and the ratios achieved by T&IP. This is due to the conversion of IP addresses to binary.

Table 4.4 shows the number of IP addresses which occur in the file and the difference between the compression ratios achieved by T and T&IP. The more IP addresses which occur, the larger the reduction and hence the larger difference between T and T&IP.

	T&IP - T	# IP addresses
access.log.1	0.07024	258525
all	0.02635	195127
httpd-access.log	0.04358	208027
mail.log	0.01797	192007

Table 4.4: Number of IP Addresses and difference in compression ratio for T and T&IP

File	None	T	T&IP
access	5.35091	5.76037	5.96430
all	5.41494	5.63265	5.77689
httpd-access	5.54863	5.74911	5.85994
mail	5.44533	5.64922	5.77780

Table 4.5: Character-based entropy for files before and after preprocessing with T and T&IP

The question is how these processed files will be compressed using other compression programs such as the ones presented in the previous chapter. The transformed files are reduced in size, however they now contain binary characters introduced by the transformation of the IP addresses and timestamps to binary. This means that some of the textual redundancy has been removed and replaced with a different form of redundancy. There is also a change in the distribution of characters and the context in which characters occur. Section 3.2.1 noted a higher compression ratio on the binary LibPCap data files than the text files. At this point it is unclear as to the extent to which these new binary characters will affect the payload ratio achieved by the compression programs. A higher payload ratio is expected due to the reduced redundancy. The reduced redundancy, however, is complemented by a smaller file size which needs to be compressed. This leads to the possibility that a higher payload ratio combined with a smaller file may result in an overall lower compression ratio than a larger file on which a lower payload ratio is achieved. The next three sections will analyse the entropy, compression ratio and compression and decompression times achieved on preprocessed files.

Entropy

As noted in Section 3.2.1, the character-based entropy is a good predictor of the compression ratio since it reflects the redundancy due to the distribution of characters.

Table 4.5 shows the character-based entropy for the files and the transformed file using T and T&IP. T&IP achieves a higher character-based entropy than T. This indicates that T contains a

higher amount of redundancy due to the distribution of characters than T&IP, and hence a lower payload ratio is expected on T than T&IP. The reduction in file size, however is higher for T&IP than T and might lead to an overall improvement in compression ratio.

File	T			T&IP			Entropy Rate
	Avg	Min	Max	Avg	Min	Max	Character 5
access	12.8027	6.36285	13.4781	12.1449	5.53729	13.2372	6.22261
all	11.799	3.99743	14.199	11.3348	3.95485	13.9492	7.68407
httpd-access	11.5704	3.90046	14.7	10.9647	3.82895	14.113	7.41089
mail	11.855	4.11936	14.3348	11.3676	4.09055	14.0578	8.11845

Table 4.6: Number of bits required to encode binary characters contained in the files preprocessed with T and T&IP

Table 4.6 shows the statistics for the number of bits required to encode to the binary characters (i.e the entropy rate for those characters introduced in Section 2.1.3) which are introduced by the binary encoding of the timestamps and IP addresses with the preprocessors. T&IP contains a higher ratio of binary characters than T since the IP addresses are in textual form in T, and binary form in T&IP. The higher frequency causes a lower average entropy rate for these characters (using a character based model), however the average entropy rate for the other characters is increased leading to a higher overall entropy rate for the data source. Not surprisingly the entropy rates for ASCII character 5 when using T&IP is lower than the average entropy rate for the binary characters. The entropy rate for ASCII character 5, in this case, is related to the number of IP addresses which occur in the file (`access` has a lower entropy rate for character 5 than `mail` since more IP addresses occur in `access` than in `mail`).

If only considering character-based redundancy, a higher average entropy for characters means that less bytes are saved by the transformation when the files are compressed. Compression programs, however, also use other techniques such as sliding windows and context statistics when performing compression.

Compression Ratio

Overall, the compression ratio achieved by the preprocessors is higher for T than T&IP, but the entropy is lower which means a lower payload ratio is expected. It is therefore unclear what the improvement in compression ratios for T and T&IP in combination with other compression

programs will be. This section investigates the results achieved for the compression ratio on the corpus. It investigates the improvements in compression ratio when using these preprocessors and also the difference between the results for the compression ratios of the two preprocessors.

	T		T&IP	
	Average Ratio	Average Improve	Average Ratio	Average Improve
lzop(1)	0.17284	0.01188 (6.87)	0.16937	0.01535 (9.06)
lzop(2)	0.12708	0.00683 (5.37)	0.12958	0.00433 (3.34)
7zip	0.07871	0.00279 (3.54)	0.07981	0.00169 (2.12)
arj	0.12329	0.00283 (2.3)	0.12603	0.00009 (0.07)
gzip	0.11816	0.00244 (2.06)	0.12058	0.00003 (0.02)
zip	0.11817	0.00244 (2.06)	0.12058	0.00003 (0.02)
bzip2	0.08872	-0.00081 (-0.91)	0.09201	-0.00433 (-4.71)
ppmd	0.07246	-0.00732 (-10.1)	0.07502	-0.00988 (-13.17)

Table 4.7: Average compression ratio and average improvement in compression ratio with the use of T and T&IP (ordered by average improvement from largest to smallest)

Table 4.7 shows the average improvement in compression ratio by each compression program with the use of T and T&IP as preprocessors. The rankings of the compression ratios when using both preprocessors are the same as the rankings in the results presented in Section 3.2.1. As seen in the results presented in Section 3.2.1, without the use of the preprocessors `gzip` and `zip` produce similar compression ratios. When using both preprocessors, `gzip` and `zip` display similar compression ratios and hence similar improvements in compression ratios. `arj` shows a greater amount of improvement than `gzip` and `zip`, however the compression ratio achieved is still higher than `gzip` and `zip`.

Out of the statistical based compressors, `7zip` is the only compression program which shows an improvement in compression ratio when using the preprocessors. `bzip2` and `ppmd`'s compression ratios are not improved using the preprocessors. Despite the increase in compression ratio for `ppmd` with the use of preprocessors, the compression ratios with the use of the preprocessors remains lower than the compression ratios achieved by `7zip` with the use of the preprocessors.

A closer investigation into the results reveals that for all the compression programs except `bzip2`, there exists a compression level which produces an improvement in compression ratio when using a preprocessor. The first three levels of `zip` and `gzip`, all levels of `lzop`, level 7 of `7zip`, level 4 of `arj` and level 1 of `ppmd` achieve higher compression ratios when using the T&IP

preprocessor. All the compression levels of `bzip2` and all the compression levels of `ppmd` from level two do not show an improvement in compression ratio when using the T preprocessor. Compression level one of `ppmd` does, however show an improvement as well as the rest of the compression levels of the compression programs.

Table 4.7 shows that `zip` and `gzip` show an improvement on average, however it is not a dramatic improvement. `zip` and `gzip` use the DEFLATE algorithm. LZ77 replaces sequences of characters with back references. Once the LZ77 is completed, it encodes the result using Huffman encoding. LZ77 would therefore use a sequence e.g. "`Aug 10 00:`" and make a reference of 2 bytes for each subsequent appearance. This result would then be encoded using Huffman encoding. By changing the timestamp to binary, this redundancy is removed and is replaced with a bit-level redundancy. This removes the improvement achieved by LZ77.

In Table 4.7, the results for the compression ratio achieved by T&IP tend to be higher than the results achieved by T (with the exception of `lzop(1)`).

	Min	Max	Average
bzip2	0.00334	0.00369	0.00352
arj	0.00120	0.00340	0.00274
ppmd	-0.00784	0.00418	0.00256
lzop(2)	0.00159	0.00296	0.00250
gzip	0.00045	0.00356	0.00241
zip	0.00045	0.00356	0.00241
7zip	-0.00007	0.00296	0.00110
lzop(1)	-0.00415	-0.00333	-0.00347

Table 4.8: Difference between improvement of T and T&IP

Table 4.8 shows the difference between the compression ratios achieved by compression programs with the use of T as a preprocessor and with the use of T&IP as a preprocessor. A negative value for the difference indicates a greater compression ratio when using T than when using T&IP and a positive value indicates a greater compression ratio when using T&IP than when using T. Before compression, the preprocessor T&IP resulted in a greater reduction in file-size than T (i.e a lower compression ratio). After compression, however, the larger file produced by T shows a lower overall compression ratio due to a lower payload ratio on the preprocessed file. This is suprising, but expected since the file produced by T does have a lower entropy than

the file produced by T&IP. T achieves a lower compression ratio than T&IP for all compression levels of `bzip2`, `arj`, `lzop(2)`, `gzip` and `zip`. For all compression levels of `lzop(1)`, the first two compression levels of `ppmd` and compression level 4 of `7zip`, T&IP achieves a lower compression ratio than T.

Compression and Decompression Times

Section 3.2.2 noted that generally an improved compression ratio requires an increase in the amount of time to compress and decompress the log file. When using preprocessors, an improvement in the compression and decompression times may result, however this might not necessarily be as high as the time taken by the preprocessor to perform transformations.

	Transform		Reverse		Total	
	T	T&IP	T	T&IP	T	T&IP
access.log.1	51.97	129.24	51.79	110.1	103.76	239.34
all	17.88	32.81	11.99	22.64	29.87	55.45
httpd-access.log	20.64	43.41	25.19	38.27	45.83	81.68
mail.log	14.22	23.94	8.79	16.12	23.01	40.06
Average	26.18	57.35	24.44	46.78	50.62	104.13

Table 4.9: Time (in seconds) to perform transformations on log files in the corpus for T and T&IP

Table 4.9 shows the times taken by the preprocessors to perform the transformations on the file. These preprocessors are unoptimised scripts which replace IP addresses and timestamps with their binary equivalents in the manner discussed in Section 4.1.1. As expected, T&IP takes longer to perform the transformation than T. Transforming IP addresses and timestamps can be performed as it is logged to disk, which removes this time.

Table 4.10 shows the average compression time, decompression time and total time by T (shown in Table 4.10 (a)) and T&IP (shown in Table 4.10 (b)). The percentage improvement is shown in brackets. If the preprocessor is not integrated into the logging system, the average time for compression is increased since the total transform time is greater than the highest improvement in total time for both T and T&IP.

The improvements cause a change in the ranks observed in Section 3.2.2. `lzop(2)` becomes faster than `ppmd` for both T and T&IP. `7zip` also becomes faster than `bzip2` for T due to its high

	Comp	Decomp	Total
7zip	59.12 (28.18)	2.89 (19.73)	62.01 (27.83)
lzop(1)	0.96 (14.17)	1.94 (33.18)	2.9 (27.9)
lzop(2)	16.12 (11.09)	2.2 (9.83)	18.31 (10.94)
bzip2	58.92 (10.42)	9.42 (5.7)	68.33 (9.8)
arj	5.4 (8.74)	2.16 (8.85)	7.56 (8.77)
zip	4.38 (4.72)	1.23 (15.2)	5.61 (7.23)
gzip	3.93 (3.72)	2.39 (10.25)	6.32 (6.3)
ppmd	20.33 (-22.39)	22.97 (-17.22)	43.3 (-19.59)

(a) T

	Comp	Decomp	Total
7zip	63.61 (22.73)	2.96 (17.81)	66.57 (22.52)
lzop(1)	0.94 (16.03)	2.43 (16.43)	3.37 (16.32)
lzop(2)	17.39 (4.07)	2.29 (5.81)	19.68 (4.28)
bzip2	63.67 (3.2)	9.82 (1.66)	73.49 (3)
arj	5.81 (1.75)	2.27 (4.15)	8.08 (2.44)
zip	4.64 (-0.91)	1.24 (14.31)	5.88 (2.73)
gzip	4.11 (-0.66)	2.48 (6.88)	6.59 (2.32)
ppmd	19.72 (-18.75)	22.53 (-14.97)	42.26 (-16.7)

(b) T&IP

Table 4.10: Average times in seconds (percentage improvement) when using preprocessors T and T&IP with standard compression programs

percentage improvement. There is no change in rank for decompression time. For the total time, `7zip` becomes faster than `bzip2` when using the preprocessors. Section 5.3.2 notes that `gzip` is faster than `zip` for compression, however its total time and decompression time is slower than `zip`. Table 4.10 (a) and Table 4.10 (b) show that in terms of compression time, decompression time and total time, `zip` shows a higher percentage of improvement than `gzip`. This improvement does not affect the rank for either of the preprocessors. With the use of both T and T&IP, `gzip` remains faster than `zip` for compression, but achieves a slower decompression and total time than `zip`.

The increase in compression times by `ppmd`, which is seen in the tables, is unexpected, but easy to explain. The increase is due to the increased amount of symbols which leads to a larger context tree and slower times. The rest of the programs are faster when compressing the transformed text due to the decrease in file size. `7zip` and `bzip2` show a large amount of reduction in times. Since `bzip2` and `7zip` produce the slowest times, the resulting time improvements when using the preprocessors is welcome.

	Comp	Decomp	Total
ppmd	0.61 (3.64)	0.44 (2.25)	1.05 (2.89)
gzip	-0.18 (-4.39)	-0.09 (-3.37)	-0.27 (-3.98)
zip	-0.26 (-5.63)	-0.01 (-0.89)	-0.27 (-4.5)
7zip	-4.49 (-5.45)	-0.07 (-1.92)	-4.56 (-5.3)
arj	-0.41 (-6.98)	-0.11 (-4.7)	-0.52 (-6.33)
lzop(2)	-1.27 (-7.02)	-0.1 (-4.01)	-1.37 (-6.66)
bzip2	-4.75 (-7.22)	-0.4 (-4.04)	-5.15 (-6.8)
lzop(1)	0.02 (1.86)	-0.49 (-16.75)	-0.47 (-11.58)

Table 4.11: Average difference between times (percentage difference) for T and T&IP (ordered by percentage difference)

Table 4.11 shows the difference between the average compression times, average decompression times and average total times for the two preprocessors T and T&IP. The percentage difference between the average improvements is indicated in brackets. A positive value for these differences indicates a slower time for T&IP than T, while a negative time indicates a slower time for T than T&IP. The values are expected to be negative since the size of the transformed file being compressed is greater for T than T&IP (which results in the expectation of a slower time when using the T preprocessors than the T&IP preprocessor). All the times for `ppmd` display results contrary to this expectation. These results occur because when the files are transformed with T, only the timestamps are converted to binary. This means that the context tree will only include these values after a newline character (since the timestamp is at the beginning of the line) and other binary values. When the IP addresses are included, the locality of these binary values is no longer just the beginning of the line, but these values are contained within the line after characters other than binary characters or newline characters. This leads to a larger context tree and greater time for prediction and hence greater time for ppmd when using T&IP than T.

4.1.3 Summary

This section has described the improvement in compression ratio, compression time and decompression time with the use of T and T&IP. The preprocessors reduce the file size by between 9 and 17 percent (with T&IP achieving a greater amount of improvement than T). The entropy of the files, however, show a higher entropy for T&IP than T. This indicates that the redundancy due to the distribution of characters is greater for T than T&IP. Section 4.1.2 showed that a higher payload ratio is achieved on T&IP than T. This results in a greater improvement in compression ratio for T than T&IP. The average improvement in compression ratio is 2.88 percent

for T, while the average improvement in compression ratio is 1.64 percent for T&IP. With the exception of `bzip2`, all compression programs have a compression level for that program which shows an improvement in compression ratio. T&IP does, however, show a greater improvement in time than T. There is a difference of up to 11.58 percent in terms of the amount of improvement between the two preprocessors. All programs with the exception of `ppmd` show a faster total time for T&IP than T. T&IP shows an improvement in total time of up to 27.9 percent, while T shows an improvement in total time of up to 22.52 percent.

This initial semantic investigation has shown that by exploiting known semantic knowledge, the compression ratio and compression times can be improved using simple transformations. While this section does show an improvement, much further improvement is possible. This initial semantic investigation only compresses the timestamps and IP addresses present in the file. This disregards aspects such as the frequency of that IP address and the gain of compressing it. When using T&IP, all IP addresses gets compressed to occupy five bytes. The results presented show that the improvement in compression ratio by compressing the IP addresses is negligible; in fact it often leads to a higher compression ratio. Commonly used IP addresses such as `127.0.0.1` (the IP address for localhost) can be assigned a single character in a dictionary rather than five bytes. The IP address and timestamps are also not the only semantically based redundancy present in the log files. There are also many other fields which occur often (such as `TCP_HIT` within the squid access log file). Chapter 2 illustrated that word based models are very effective at improving the compression ratio and that preprocessors which use a dictionary are effective at improving the compression ratio of English text. The next section investigates the semantics which are present in maillog files which can be used to create a dictionary to improve the compression ratio achieved on these log files.

4.2 Analysis of Semantics contained in maillog files

Apart from the timestamp and IP addresses, log files contain many other frequent patterns. From this point, the focus shifts to extracting semantic information from maillog files that can be used by preprocessors in a similar manner to the IP addresses and the timestamps. Maillog files are used since they use the syslog format there are many different processes which send messages to these log files which leads a diverse range of messages and a hence more complex semantics. The aim of this analysis is to determine the structure of the log files and the common elements which are present. It also aims to determine the value of using this semantic information to

improve compression. This section presents a tool which can be used to analyse the semantics of a log file and also investigates the structure of maillog files. It also shows how the log analysis tool can be used to determine the structure of log files.

4.2.1 Methodology

To determine the semantics present in a log file, the structure of the file needs to be analysed and the values which commonly occur need to be determined. Some log files can simply be tokenised using a single character such as a space since each field contains a small subset of values. In the case of syslog files, such as Postfix maillogs, the log files cannot simply be tokenised to determine the values since there are a wide range of messages which are outputted to the log files with different formats. The following line is a sample from the 3MB maillog file which is used for initial testing and development:

```
Aug 10 00:00:00 titania newsyslog[46545]: logfile turned over
```

The syslog format contains a timestamp, followed by a hostname, information about the originating process and a message [17]. The originating process information is of the form `Process[PID]` e.g. `newsyslog[46545]` where the originating process is `newsyslog` and its PID is `46545`. The timestamp also has a defined form of a three letter month identifier, two characters for the day of the month (e.g. `Aug _9` or `Aug 10` where _ indicates a space) and 8 characters for the time, which is in the format `HH:MM:SS`. Since these fields are separated by a space, they can simply be tokenised. The format of the message, however, is defined by the process which is generating the log line. These messages differ in format and length and hence cannot all be tokenised in the same manner to determine values of each successive token. The different messages for each process need to be determined and these can be tokenised to determine the static and variable parts of those messages.

The methodology for analysing the maillog file consists of two steps. The first step consists of performing a "handmade" analysis using tools such as `grep` and lookup tables within a spreadsheet to determine the semantic knowledge which a maillog file contains. Due to the size, this can only realistically be performed on a small corpus. The second step consists of developing a tool which can be used to determine the semantics contained in a log file based on a set of rules which are formed iteratively. The rules which are formed can be used to perform analysis on a larger corpus. The analysis was initially conducted using a 3MB maillog file which consisted of

one day of log traffic and then it was applied to a 1.6 GB corpus of mailogs which consists of 15 months of log traffic. The sections which follow discuss the methodology which was used.

"Handmade" analysis

The "handmade" analysis was performed to determine the type of semantics which are present in the maillog file. The purpose was also to determine how an automated tool can be developed to analyse the semantic knowledge which is present in a given syslog file. The "handmade" analysis can also be used to verify that the tool can accurately determine the semantics present in the file. As mentioned, the maillog can not be simply tokenised since it uses the syslog format. The messages for each process need to be determined in order to investigate the semantics. The handmade analysis was conducted using a 3MB Postfix maillog file which contains one day of log traffic (24892 messages). Each line of the log file was inserted into a spreadsheet and tokenised using a space. Lookup tables were then used to isolate the messages for each process and determine their semantics. The GNU regular expression parser (`grep`) was also used to isolate certain phrases which are used for a given process. It was used in the following manner:

```
grep postfix/smtp\[ maillog | grep connect\ to | less
```

This command was used to get all the log lines for the `postfix/smtp` which contained the phrase "`connect to`". This process was performed for each of the processes in the log file to make sure that the messages for that process can be described using the results. In this manner, the semantics were determined.

Tools to analyse files

When analysing a larger corpus, lookup tables and `grep` cannot be used due to the quantity of data which needs to be analysed. A tool is therefore desired to analysis the corpus and determine the semantics. During the "handmade" process, patterns are identified within the messages. These patterns consist of variable parts which exhibit certain formats such as a date and time or a number. These variable parts are complemented by static parts which are the same for messages of that type. For example, consider the following message:

```
connect to [hostname] Connection Refused (port 25)
```

In this example, the variable token is the hostname described in the log line. During the analysis of the file, the variable token can be replaced with a placeholder in the same manner as placeholders were used in the "handmade" analysis. The token which is replaced by a placeholder can be determined by a rule. In this example, the rule would replace all hostnames with the placeholder `[hostname]`. An advantage of using a rule-based tool is that the frequency of the various types of messages can be easily determined.

The tool developed begins by determining all of the processes which occur in the specified log file and their frequency. It then performs operations on the messages for that log line to strip away information and replace them with placeholders. Elements such as IP addresses and email addresses are replaced with values (placeholders) such as IP and email. This is done using a defined set of rules which are general or process specific. These rules are formed iteratively by identifying semantics from the output of the program. Since each process outputs a number of different messages, the lines are separated into the processes.

There are essentially two types of replacements which can occur: replacement on the whole string; and replacement on part of the string. This tool also uses two different types of rules called inner rules and outer rules. The inner rules run on the string as each tokenised element of the message is added (where the tokens are spaces). The outer rules run on the entire message after the inner rules have been performed. There are these two types of rules since the patterns can exist as patterns within a line or as separate tokens (such as numbers) which need to be replaced as the string is built up otherwise. The program designed to determine the semantics can be summarised by the following steps:

1. Determine the different processes involved and store a list of them
2. Print out the processes followed by the frequency of messages for that process
3. Read the rules from the file
4. Tokenise string using a space and run inner rules as each token is added to form the entire string
5. Run the outer rules on the resulting string
6. Update the frequency of the resulting string in the relevant process's dictionary
7. Print out the dictionary for each process

The rules which the program uses can perform a number of operations on the string. These rules take in two arguments: the first is the search criteria which differs according to the operation; and the second is the replacement string. The operations which can be performed in the tool developed by the researcher are: replregex, rfind, search. These are defined as follows:

replregex replaces a given regular expression (defined in the search criteria) with a given string (defined in the replacement string).

rfind searches for the first occurrence of a given string and concatenates the string before that point with the replacement string.

search is similar to replregex, however if it finds an occurrence of that regular expression occurs, the the entire string is replaced by the replacement string

The rules are stored in a file beginning with the inner rules for all processes and followed by the inner rules which are process specific, which are in turn followed by the outer rules which run on all processes and the outer rules which are process specific. The rules are run on the string in the order in which they occur in the rules file. Rules files developed for the 3MB maillog file and the corpus of maillogs can be found in the Appendix E.2. These are described in the next two sections. This program is implemented using Python due to its good support for regular expressions and the dictionary data structure. The code for this program can be found in Appendix E.1.4.

Building a rule set

The rule set is built iteratively using observations from the output and adding appropriate rules to replace variable elements such as hostnames, IP addresses and message identifiers with placeholders.

The rules are written in the following format:

```
Pr: process (for process specific rules)
O:  operation
P:  parameter
R:  replacement string
```

The following rule would therefore be used to replace all the hostnames with the word hostname:

O: replregex
P: localhost|(([A-Za-z0-9\-_]+\.)+[A-Za-z]{2,3})
R: hostname

Appendix B details the rest of the rules which can be applied to a maillog file.

Adding rules for the larger corpus

The larger corpus contains 15 months of log data. Within these 15 months, 21049570 log messages were generated by 40 different processes. The 3MB maillog used in the previous section, however, only contained 24892 log messages generated by 17 different processes. The results on the larger corpus using the rules presented in the previous section can still be improved. The larger corpus shows a number of new messages and new processes. `postfix/smtpd`, `postfix/qmgr`, `postfix/smtp`, `postfix/cleanup`, `amavis`, `sqlgrey`, `postfix/anvil`, `postfix/error`, `postfix/virtual`, `postfix/master` and `postfix/scache` produce 99.5 percent of the log messages. Each the other processes produce less than 0.16 percent of the log lines. Hence, the focus is only on creating rules which improve these ten processes. Rules are only formed for patterns which occur in more than 0.1 percent of the messages for a process. Appendix B details the rules which are added.

4.2.2 Results and Analysis

This section presents an analysis of the results of the semantics contained in the mail log files using the "handmade" analysis on and the results from when using the tool with the rules described in Appendix B. The detailed results are contained in Appendix C.

As mentioned, the 3MB maillog file consists of 24 892 messages which are created by 17 different processes.

Table 4.12 shows the distribution of the processes. It shows the number of lines for each process and the resulting percentage of the total lines. This table shows that `postfix/smtp` and `postfix/smtpd` account for 59.95 percent of the log entries, while `postfix/smtp`, `postfix/smtpd`, `postfix/qmgr`, `postfix/cleanup` and `amavis` account for 97.60 percent of the log entries. This section details the results for these five processes.

Table 4.13 shows the percentage of log lines which occur in the corpus which contains 15 months of log data for each of the top five processes. This corpus contains 15 files which each contain

Process	Number	%	Process	Number	%
postfix/smtpd	9534	33.83	postfix/bounce	42	0.15
postfix/smtp	7363	26.12	postfix/local	24	0.09
postfix/qmgr	6242	22.15	postfix/discard	7	0.02
postfix/cleanup	2893	10.26	postfix/tlsmgr	7	0.02
amavis	1476	5.24	postfix/postfix-script:	2	0.01
postfix/anvil	258	0.92	postfix/master	2	0.01
postfix/scache	181	0.64	imapd:	2	0.01
postfix/virtual	98	0.35	newsyslog	1	0
postfix/pickup	52	0.18			

Table 4.12: Frequency of processes in the 3MB maillog file

	Min	Maximum	Average	Concat
postfix/smtpd	32.571	47.396	42.820	42.808
postfix/smtp	10.889	25.340	14.391	12.899
postfix/qmgr	18.765	29.193	22.856	22.358
postfix/cleanup	8.661	11.622	10.127	9.978
amavis	3.814	6.286	5.026	4.959

Table 4.13: Distribution of processes in 15 month maillog corpus (Top 5 processes shown)

one month of log data. The minimum percentage, maximum percentage and average percentage for these 15 files is shown in the table. The percentage of log lines for the concatenated corpus is also shown in the table. This table shows that these five processes generate between 87.867 percent and 98.215 percent (average of 95.220 percent) of the log entries in the 15 files. This section shows the results for the analysis of the log messages for these five processes. The results for the five processes are contained in Appendix C.

By inspection, the results for both the 3MB maillog file and the maillog corpus show a number of trends. The following words (or phrases, phrases are contained in "") commonly occur in the lines for each of the five processes:

postfix/qmgr: relay, to, from, size, nrcpt, delay, delays, status, expired, dsn, deferred, "queue active", "returned to sender", "delivery temporarily suspended", "Connection refused", "connect to hostname"

postfix/smtpd: connect, disconnect, client, RCPT, connections, unavailable, warning, verification, "NOQUEUE: reject: RCPT from", "Recipient address rejected: Domain not found", "connect from", "disconnect from", "Connection concurrency limit exceeded", "setting up

TLS connection from"

postfix/cleanup: message-id, warning

postfix/smtp: host, said, to, relay, delay, dsn, status, sent, queued, connect, certificate, verifications, "Queued mail for delivery", "connections refused", "refused to talk to me:", "Operation timed out", "No route to host", "The users mailbox is full. Please try re-sending your e-mail later. (in reply to RCPT TO command)", "certificate verification failed for", "Server certificate could not be verified"

amavis: Passed, CLEAN, Message-ID, mail_id, Hits, queued_as, "Passed CLEAN"

By replacing these words or phrases with tokens from the zero frequency characters, the filesize can be reduced before compression. There are many different word or phrases which can be replaced, so a score need to be calculated to determine which words take precedence for inclusion in the dictionary. This is discussed in the next chapter.

4.2.3 Summary

Within the maillog files, there are a number of different elements that can influence the semantics contained in the log file, including: the versions of software being used; the processes which are set to log to that file; and the verbosity level at which the logging is taking place.

The results of the analysis of the log files shows that there are a number of words and elements (such as hostnames, IP addresses and numbers) which occur frequently within the log file. For both the results for the 3MB log file and the corpus of 15 months of maillogs, phrases such as "connect from" and words such as "to", "from", "relay", "delay" occur often. These semantics can be exploited by a preprocessor. Section 4.1 showed that using preprocessors to reduce the timestamps by converting them to binary causes an improvement in the compression ratios achieved by compression programs. Within the log files, there are a number of properties which can be exploited by a preprocessor. These include:

1. Text log files contain many unused characters (zero-frequency characters)
2. Timestamps and IP addresses take up alot more space than they should
3. Certain hostnames and IP addresses occur many times within a log file

4. There are a number of other words and phrases which occur often to describe values which vary

The common hostnames and IP addresses can be replaced using the unused characters within the log file. Section 2.5 showed that preprocessors using word-based replacement provide an improvement in compression ratio for English text by replacing frequent words with tokens. This section has shown that there are a number of different words which occur frequently within the log file. These words can be used to perform word-replacement using the zero-frequency characters. In order for this method to work, dictionaries need to be built which can be used for replacement. The rest of this thesis focuses on building dictionaries which can be used by preprocessors to exploit the semantic information which has been shown in this section.

4.3 Summary

In the previous chapter, it was shown that the use of standard compression programs resulted in a large reduction (up to 98 percent) in the size of log files (refer to Section 3.2.1 for more details). This chapter has shown that simple preprocessors which replaced the timestamps and IP address with their binary equivalents resulted in an improvement in the compression ratio, compression time and decompression time achieved by the standard compression programs. Section 4.2 also showed that within a mailog file there are many different commonly occurring words and phrases which can be replaced with a single character. The next chapter investigates the improvements in compression ratio, compression time, decompression time and memory utilisation which result when performing word replacement. It also presents a methodology for constructing a dictionary based on a word definition and word score.

Chapter 5

Word-based replacement

There are many repeated elements within log files apart from the timestamps and IP addresses. These include elements such as: hostnames; frequently used mail addresses; and field tags such as "to" and "from". The initial semantic investigation presented in Section 4.1 showed an overall improvement in the compression ratio, compression time and decompression time achieved with the use of preprocessors which reduced the size of timestamps and IP addresses within the log file. However, these preprocessors do not exploit the redundancy due to the other commonly repeated elements within the log file.

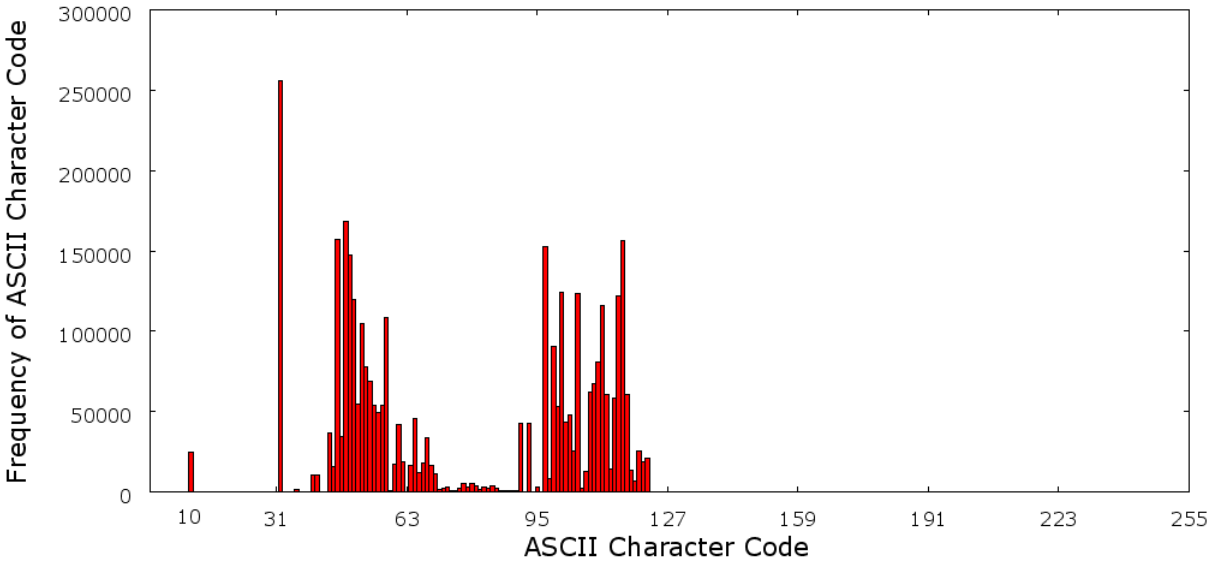


Figure 5.1: Character Distribution of the 3MB maillog file

Figure 5.1 shows a histogram of the character distribution of a three megabyte Postfix maillog file which contains one day of log messages. The X-Axis shows the ASCII value of the characters, while the Y-axis shows the frequency of the characters. This figure shows that there are many zero-frequency characters contained in a maillog (A table of the zero-frequency characters can be found in Appendix D). Replacing the repeated elements with zero-frequency characters can therefore be used to decrease the file size before compression.

The replacement of words does, however, remove character-based redundancy and sequential redundancy which are exploited by compressors. Since the words are replaced with a unique identifier, the context in which the word occurs still remains. This introduces a new form of redundancy and means that some sequences which are used by compressors remain (however they are now shorter and include the unique identifier instead of the word). Abel [130] shows that replacing commonly occurring words with zero frequency characters leads to an improvement in the compression ratio for various English based texts from the Calgary Corpus and the Canterbury Corpus.

This chapter discusses a method for constructing a dictionary for a file based on a given word definition and word score. It then presents and discusses the results for preprocessors which perform word replacement using these dictionary and the improvements which results when using these preprocessors with standard compression programs. Definitions for the metrics (compression time, decompression time, compression ratio and payload ratio) used in this chapter may be found in Section 2.1.4.

5.1 Constructing a dictionary

The first step in the process is to construct a dictionary to be used for word replacement. The words in the dictionary are replaced with zero-frequency characters in a similar manner to the process outline by Abel [130]. This requires criterion for building the dictionary to be defined. This dictionary can be built and "tweaked" manually until it is optimal, however an algorithm for automatically generating a near optimal dictionary for a file is desired. The first step in this process is to define what constitutes a word or phrase.

5.1.1 Definition of a word

Essentially a word contains one or more letters from `a` to `z` in lower case or uppercase. A single letter can be considered a word in any given language. However these cases are not considered to be words in this definition because there is no value in reducing a single character. The regular expression `[A-Za-z]{2,}` can therefore be used to describe a word. Within a maillog, there are many common words which contain dashes and underscores. In order to take these words into account, the regular expression can be extended to be `[A-Za-z\-_]{2,}`. Further observation reveals that certain hostnames appear very often within the maillog. In order for a word definition to include these hostnames, a dot needs to be part of a word. The updated regular expression is therefore `[A-Za-z\-_\.]{2,}`. The maillog also contains many instances of "words" which contain numbers. The updated regular expression, `[A-Za-z0-9\-_\.]{2,}`, includes these words in the word definition. The maillog file also contains many instances of email addresses. The `@` character could also be included as part of the word definition, however, this would give poor results since all instances of hostnames in email addresses would become part of different words which leads to a lower frequency for `hostname.co.za` and a low frequency of words such as `person@hostname.co.za`. The lower frequency means that neither of these words are likely to be included in the dictionary. The `@` character is, therefore, not include in the definition.

The test results presented in Section 5.3 will therefore show results for the following regular expressions (referred to as word definitions):

A: `[A-Za-z\-_]{2,}`

B: `[A-Za-z\-_\.]{2,}`

C: `[A-Za-z0-9\-_\.]{2,}`

Once the words that are contained in a file are determined (using the word definitions), the next step is to define how the words are selected for the dictionary.

5.1.2 Function for a score

A feasible method for determining whether a word should be included is to define a score which can determine the value of replacing the word. This may be accomplished by using a function based on the frequency of that word. A basic function would be $g_1(f_i) = f_i$ where f_i denotes the

frequency of a word i. (i.e. In the text "This is a test, Yes it is", $f_i = 3$ where $i =$"is"). This basic function is not ideal, since if there are 5000 instances of the word "abc" and 1000 instances of a 50 letter word j, then j might not make it into the dictionary since $g_1(1000) = 1000$ and $g_1(5000) = 5000$ which means that "abc" would be chosen before j for the dictionary using this function.

An effective function would ideally represent the gain of replacing a particular word in the file. This can be done by taking into account the length of the word. $g_2(f_i, l_i) = f_i \times (l_i - 1)$ where $l_i$is the length of the word i (i.e $l_i = 2$ where $i =$"is"). $(l_i - 1)$ is used rather than l_i since the word i of length l_i is being replaced with a single character, i.e. The text is being reduced by $(l_i - 1)$ characters every time the word i occurs (which is f_i times). Using the earlier example of "abc" and the 50 letter word (j); $g_2(5000, 3) = 10000$ and $g_2(1000, 50) = 49000$. This means that j would be chosen before "abc" for the dictionary using this function. More complex formulas may be developed, however this is left for future work.

The test results presented in Section 5.3 use the following scores to select their dictionaries from a set of words:

1. f_i
2. $f_i \times (l_i - 1)$ where $l_i$is the length of the word i

5.1.3 Putting this together

Now that a word definition and a criteria to select a dictionary has been formed, these definitions need to be applied to build a dictionary from a file. This process is defined formally as follows:

1. Determine the set of zero-frequency characters, A
2. Determine the frequency, f_i, of each word i
3. Calculate a score s_i based on the frequency f_i (i.e. $s_i = g(f_i)$)
4. Construct the set $W = \{j : s_i < s_j \ for\ all\ i \notin W\}$ such that $\mid W \mid = \mid A \mid$, and create a one-to-one mapping, $h(w)$, of W onto A. Thus $h(w)$ where $w \in W$ yields the ASCII value associated with the word w and the inverse function (which exists since h is a bijection), $h^{-1}(a)$ where $a \in A$ yields the word associated with the ASCII value a.

Essentially the set W contains all the words with the highest scores (i.e. the words which need to be included in the dictionary).

Algorithm 4 Construction of Dictionary Mapping (the function $h^{-1}(a)$)

```
counter=256
for line in file
    for each character
        if character not used before
            mark as used
            counter=counter-1
    for each word in line
        update word count for that word
        update list w which contains sorted list (by score) of counter word
i=0
for each unused character (a)
    associate a with w[i]
    i=i+1
```

Algorithm 4 shows an algorithm which performs this procedure. To improve speed of construction and make the algorithm adaptive (i.e. it updates as it goes along), it begins with all the characters available and keeps a sorted list (w) of the words with the highest scores. The algorithm processes each line of the file, updating the available characters, the words in the list w and keeping the list w smaller than the number of available characters. Once all the lines have been processed, the dictionary is formed by going through the list of unused characters and associating each character, a, with a word in the list w. In this algorithm, the first character will be associated with the word that has the largest score and the second character with the word with the second highest score and so on.

5.1.4 Applying the dictionary

The dictionary is applied by using a search and replace function. Each occurrence of the word in each line of the file is found and replaced with the associated character set determined by the dictionary mapping. This character set is made up of combinations of the zero-frequency characters. Different Methods of coding can be used. Prefix-free codes could be created from the zero frequency characters, two letter combinations could also be used. These methods would make more character combinations available and hence a larger dictionary can be used. The character combination which replaces a given word is determined by the dictionary mapping. In this thesis, single zero-frequency characters are used for replacement. Other methods of coding are discussed in Section 5.1.6

5.1.5 Implementation

There are two parts to the implementation of this process: determining the dictionary mapping and performing search and replace. The implementation is done on a small data set (3MB maillog file which consists of one day of log traffic) and then applied to a larger data set which consists of 15 months of log data. This section describes how these two parts are implemented.

Determining the Dictionary Mapping

Determining the dictionary mapping can be divided into three steps: determining the words; determining the zero frequency characters; and determining the dictionary mapping. To determine the words, all consecutive character sets which match a given word definition and their respective frequencies are determined. Using these frequencies, the word score is calculated and the required number of words for the dictionary mapping are determined. This research describes three different implementations for the method of determining the words. All three implementations are developed in Python as it is a simple language which has good support for regular expressions and incorporates a dictionary structure (which is required to keep track of the count of words).

The first implementation stores the frequencies for each word in a Python dictionary structure and then performs a selection sort on the dictionary so that the words which have the highest score are outputted first. The second implementation is equivalent to the algorithm presented in Algorithm 4. This is an alteration of the first method to remove the need for the selection sort (which proves slow on large dictionaries). In this method, an array is kept which contains the words which have the highest scores. This array is updated every time a word is found, and is used to output the dictionary. The third implementation also uses a Python dictionary to keep track of all the words and their counts. At the completion of a day's log lines, the dictionary structure is stored in an SQLite database and emptied. The SQLite database contains fields for the Date, the Word and the Frequency of the word. Once the file has been processed, the database contains the words and their respective counts for each day which occurs in the log file. A complex SQL SELECT statement is then used to get the resulting dictionary for a particular date range. The following statement would be used to get the dictionary (160 words) for the 3MB log file using word definition C, word score two:

```
select word, sum(freq), sum(freq)*(length(word)-1) from maillog
```

```
group by word order by 3 desc limit 160
```

In this database, the table `maillog` contains the words for the 3MB maillog file using word definition C and their respective frequencies.

Implementation	(A,1)	(A,2)	(B,1)	(B,2)	(C,1)	(C,2)
1	33.844	66.91	34.91	70.728	262.974	556.138
2	42.458	51.63	37.37	43.392	60.808	75.53
3(a)	2.148	2.148	1.976	1.976	3.156	3.156
3(b)	0.144	0.144	0.152	0.154	0.452	0.462

Table 5.1: Times (in seconds) to build dictionary for each of the six word-score combinations using the three implementations

Table 5.1 shows the times to build a dictionary for each of the six word score combinations (WSCs). These times are calculated by taking the average of five iterations on the 3MB maillog file. Implementation three has been divided into two parts: 3(a) consists of processing the file and creating the database; while 3(b) consists of using a SELECT SQL statement to obtain the dictionary from the database. The first implementation is slow since it uses a selection sort to sort the entire dictionary in order of frequency. The second implementation is faster than the first implementation, on average, since it uses a list to keep track of the dictionary instead of performing a selection sort. The third implementation is between 93.23 and 99.35 percent faster than the other two implementations since it writes straight to the database.

In the selection sort, the complexity is $O(n^2)$ where n is the number of unique words. The number of unique words identified is 6590, 6734 and 18929 for word definitions A, B and C respectively. This causes the first implementation to achieve faster times for word definition A than word definition B. The second implementation, on the other hand, updates a list every time a word occurs rather than performing a selection sort. Each time a word occurs, it goes through the entire list to check if that word is in the list. If the word is in the list, then its position is updated. If the word is not in the list, then it is added to the list if its word score is greater than the word score of any word which occurs in the list. This algorithm has a complexity of $O(ab)$ where a is the length of the list and b is the number of words which occur in the file. The number of words which get matched by the regular expression are 335 055, 273 931 and 460 424 for word definitions A, B and C respectively. With the length of the list is set to 255, the approximate number of iterations for the second implementation are therefore 8.54×10^7,

6.98×10^7 and 1.17×10^7 for word definitions A, B and C respectively. This causes the second implementation to achieve faster time for word definition A than word definition B. The first part of the third implementation (3(a)) is only dependent on the number of words which get matched by the regular expression. It therefore also shows faster times for word definition A than word definition B.

Interestingly, the first implementation is faster than the second implementation for WSC (A,1) and WSC (B,1). Approximately 4.34×10^7, 4.53×10^7 and 3.58×10^8 iterations are required to perform the selection sort for word definitions A, B and C respectively (n^2). The second implementation does not use a selection sort, but rather keeps a list. The approximate number of iterations required to maintain the list are 8.54×10^7, 6.98×10^7 and 1.17×10^8 for word definitions A, B and C respectively. The number of iterations are hence greater for the second implementation when using word definition A and word definition B with word score one. The times when using word score two are always slower than the times when using word score one due to the calculation of the score by calculating the length of the word and multiplying it by the frequency. When using word score two, the amount of calculations of the word score ($(l - 1) \times f$)) in the first implementation is n^2 where n is the number of unique words, while in the second implementation it is ab, where a is the length of the list and b is the number of words which occur in the file. These numbers are greater for the first implementation than the second implementation. This results in a greater difference between word score one and word score two for the first implementation than the second implementation. This greater difference leads to the preprocessor based on WSC (A,2) achieving faster times than the preprocessor based on WSC (B,2) using the second implementation. The results presented in Section 5.3 use the third implementation to generate the dictionary.

Performing search and replace

Two different methods are evaluated for performing search and replace using the dictionary. The first implementation uses a Python script which goes through each line of the file and searches for the dictionary words in the line and replaces them with the code specified in the dictionary mapping. The second implementation uses a Unix program called `sed`. This program is designed to perform search and replace on a file given a number of regular expressions to search for and text to replace occurrences of these regular expressions. To use this program, the dictionary map is outputted as a `sed` script. These script contains statements of the form `s/Word/Code/g` for compression and `s/Code/Word/g` for decompression. In these statements, `Word` represents the

word which is being replaced and `Code` represents the zero frequency character which is replacing the word. On the 3MB corpus, The `sed` scripts take 7.91 seconds, while the Python scripts takes 22 seconds to perform search and replace. The results presented in Section 5.3 therefore use `sed` scripts to perform search and replace.

5.1.6 The length of a code

As the number of words which are replaced increases, so too does the reduction by the preprocessor and the amount of time taken to to perform the replacement. A number of different coding techniques could be used with the zero-frequency characters available. Prefix-free codes can be constructed using the available characters, however this is not necessary since the words are always separated by a non-words (they would be parsed as a single word and not two distinct words in the dictionary generation process if they were not separated by a non-word) which means that the code does not need to be prefix-free to be identified. This section investigates using a larger dictionary for replacement combined with two character codes and its effect on the reduction by the preprocessor and the amount of time taken to perform the replacement.

The codes used to replace words are extended by including two letter codes. They are constructed as follows: Let the zero frequency characters be represented as a_i where $i = 1, 2, 3, \ldots, n$ and n is the number of zero frequency characters. The first set of codes are a_i, the second set of codes are $a_1 a_i$, the third set of codes are $a_2 a_i$, etc. The words with the highest n word scores are mapped to the first set of codes, the next n words with the highest word scores are mapped to the second set of codes, etc. The tests are performed using the 3MB maillog file which contains one day of log traffic.

Figure 5.2 shows the amount of reduction by the preprocessor and the time taken to achieve the reduction. The X-Axis shows the number of characters which are used as a first letter, i.e the number of sets of codes which are used. In this figure, the dictionary creation times do not change since this process counts all of the words regardless of the codes given. Only the amount of words required by the preprocessor changes. If this process did not count all the words then the locality of words would effect the outcome, and words which occurred infrequently at certain times would be drowned out by words which occur frequently at a point but then do not occur frequently at other points in the file. Thus the frequency of all words are counted. There is a linear increase in time taken to perform the search and replace procedure for a logarithmic decrease in the compression ratio achieved. The transform time on the 3MB log file increases from 7.91 seconds to 288.89 seconds, while the compression ratio decreases from 0.54625 to

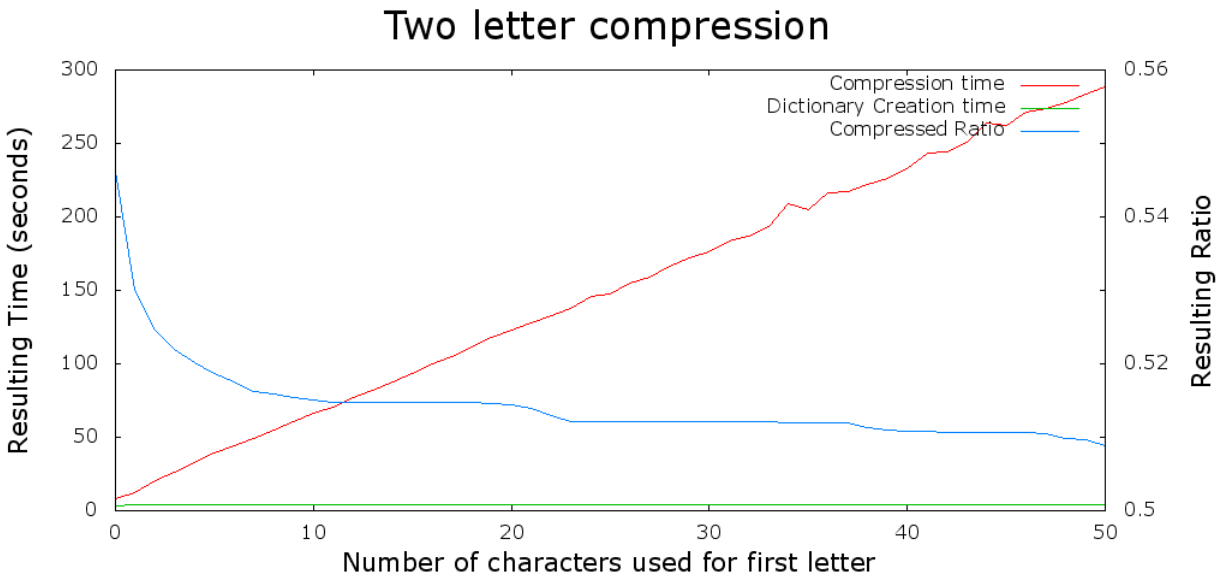


Figure 5.2: Compressed size and compression time for 3MB log file when using longer codes

0.50888. This is an 6.84 percent reduction in compression ratio with a 3552.21 percent increase in time. For each additional set of code words, the number of words which can be used in the dictionary increases by the number of zero frequency characters (typically 160 in maillog files). As the dictionary grows, the improvement in compression ratio decreases but the increase in the compression time remains constant. The introduction of longer codes also alters the entropy of the text and also changes the dynamics of when a character will be predicted. The second binary character will not be well predicted since any of the other binary characters are possible as well as a space. Context information is removed by replacing words with codes and the ratio of binary characters in the file is increased. An increased payload ratio is therefore expected. In light of these results, this thesis only considers using the zero-frequency characters for replacement since the use of longer codes results in a high increase in time for a low increase in compression ratio.

5.2 Testing Methodology

The tests were performed using a corpus which contains 15 months of Postfix mail log files. This corpus is larger than the previous corpus and is also more generalised since is only contains maillog files. The maillog files use the syslog format and contain a larger variety of messages. The larger corpus also means that the results are more statistically valid. The process used for testing is as follows:

1. Generate the dictionaries for each file using the Python script implementation, described in Section 5.1.4.

2. Apply these dictionaries to the files and record the statistics. SHA1 and MD5 hashes are performed to verify that the files are identical after decompression to the ones before compression with the preprocessor. If they do not match then `diff` is used to determine where the differences between the two files lie.

3. Run compression tests using the methodology presented in previous chapter on each of the resulting files for each of the 6 WSCs - (A,1), (A,2), (B,1), (B,2), (C,1) and (C,2) - and the original files.

This process generates seven (six for the WSCs and one for the original file) sets of statistics for each of the compression programs. From these results, the improvements in compression ratio, compression time, decompression time, total time, memory-usage and the improvements in the transfer times at a given transfer rate for each of the six WSCs can be calculated. The next section analyses these results and comments on the improvements made. It also shows which of the WSCs show the greatest amount of improvement and what the cost is. The dictionaries are generated and used for compression on the file, so hence these results show an upper bound for the results using the given word definition. In the sections which follow, the preprocessors based on a given word definition and word score will be referred to by the word definition followed by a full stop and the word score i.e. the preprocessor based on word definition C and word score 1 (WSC (C,1)) will be referred to as C.1.

5.3 Results and Analysis

WSC	(A,1)	(A,2)	(B,1)	(B,2)	(C,1)	(C,2)
Average	0.68664	0.66598	0.65616	0.64291	0.56470	0.54310

Table 5.2: Average Compression Ratios achieved by the the six preprocessors

Table 5.2 shows the average compression ratio obtained by the preprocessors based on each of the six WSCs on the test corpus of maillogs. In this section, when the average compression ratio achieved by a word definition or the average compression ratio achieved by a word score

is mentioned, this refers to the average compression ratio of the preprocessors which are based on that word score or word definition. This table shows across all WSCs, the compression ratios achieved by word score 2 are lower than the compression ratios achieved by word score 1. The average compression ratio achieved by word score 1 is 0.63583, while the average compression ratio achieved by word score 2 is 0.61733 (2.91 percent lower).

Word definition C achieves the lowest average compression ratio out of the three word definitions. It achieves an average compression ratio of 0.55390, which is 14.72 percent lower than the compression ratio achieved by word definition B (0.64954) and 18.10 percent lower than the compression ratio achieved by word definition A (0.67631).

The dictionaries generated for word definition A contain the least amount of words out of the three word definitions. This is expected, since any word which satisfies word definition A, also satisfies word definition B and word definition C. It must be remembered, however, that the set of word generated by A is not necessarily a subset of those generated by B. A word which gets matched by word definition A may be followed by a full stop and a number. This means that the the word plus the full stop would be matched as the word by word definition B and the word plus the full stop and the number would be matched by word definition C. For example, consider the text "maillog.1". Word definition A would match "maillog" as a word, while word definition B would match "maillog." as a word and word definition C would match "maillog.1" as a word. As mentioned in Section 5.1.5, this does not imply that the number of words matched is larger for a word expression which produces a larger dictionary. Section 5.1.5 showed that the number of matches for word definition A is larger than word definition B, however word definition B produces a larger dictionary than word definition A. Word definition B produces a lower compression ratio since it contains fewer longer words than word definition A.

The smaller file created by a preprocessor exhibits a different distribution of characters to the original file due to the replacement of words with zero-frequency characters. This file contains the previously unused ASCII values (zero-frequency characters) and no longer contains the particular instances of the characters which occur in the replaced words. The combination of the usage of more characters and the removal of other characters leads to a loss of redundancy which is used by different compression programs. For example, the word "hostname" contains 8 characters used elsewhere in the text, would be removed and replaced with a new ASCII values. This results in a reduction in the size of the file by 7 bytes for each occurrence of the word "hostname". The word "host", however, could be used as reference for the LZ77, thus removing redundancy which the LZ77-based compressors exploit. The removal of these characters also affects the probability of the character "s" following the context "t" in an order-1 Markov Model

(or "t" following "s" in the BWCA). The removal of these characters thus effects the performance of the BWCA and PPM compression algorithms. The use of the preprocessors does, however, introduce a new redundancy similar to that which is used in word-based compressors. A space will predict the character which is used to replace "hostname" with a high probability in the case where a space is often followed by the word "hostname" in the original file. The LZ77 window is also virtually enlarged since "hostname" now occupies a single character. Previous work (presented in Section 2.5) and the work presented in the next section show that these changes in redundancy, however, result in a increase in compression ratio by the program on the transformed file. The overall compression ratio (the ratio to the original file), however, is reduced due to the decrease in filesize by the preprocessor.

The sections which follow investigate the improvements in compression ratio, compression and decompression times, transfer time (which combines the times and ratios achieved) and memory utilisation due to the use of the preprocessor. This is followed by a summary of the results.

5.3.1 Compression Ratio

The compression ratio in this section, unless specified, refers to the ratio of the filesize after preprocessing and compression to the original filesize (as defined in Section 2.1.4). A lower ratio, therefore, indicates a smaller filesize and hence greater compression.

Rank	Lowest	Average
1	C.2 (0.12448)	C.2 (0.13309)
2	C.1 (0.12497)	C.1 (0.13312)
3	B.2 (0.12768)	B.2 (0.13779)
4	A.2 (0.12862)	A.2 (0.13866)
5	B.1 (0.12876)	B.1 (0.13916)
6	A.1 (0.12999)	A.1 (0.13986)
7	None (0.14028)	None (0.15284)

(a) Compression Ratio

Rank	Lowest	Average
1	C.2 (197)	C.2 (195)
2	C.1 (208)	C.1 (200)
3	B.2 (213)	B.2 (215)
4	A.2 (230)	A.2 (225)
5	B.1 (234)	B.1 (237)
6	A.1 (250)	A.1 (242)
7	None (264)	None (286)

(b) Summed Rank

Table 5.3: Compression Ratio statistics for each Word Score

Tables 5.3 (a) and 5.3 (b) both illustrate statistics for the average of the lowest compression ratios for each program using the six preprocessors and the average over all compression programs and

levels using the six preprocessors. Table 5.3 (a) shows the average compression ratio achieved, while Table 5.3 (b) shows the summed rank. The value of the statistic for a given ranking is found in brackets next door to the preprocessor which achieves the rank. The summed rank statistic for the preprocessors is calculated by taking the compression ratio statistics for each program using each of the six preprocessors (A.1, A.2, B.1, B.2, C.1, C.2) and the original results without preprocessing with the eight compression programs (`7zip`, `arj`, `bzip2`, `gzip`, `lzop(1)`, `lzop(2)`, `ppmd` and `zip`) and ranking them from lowest to highest. The ranks achieved from 1 to 56 by each compression program with a preprocessor are summed to give the summed rank for that preprocessor. These results show that word score two generally achieves a lower compression ratio than word score one. They also show that word definition C achieves the lowest average compression ratio out of the three word definitions, while word definition B achieves a lower average compression ratio than word expression A (0.13848). These results show that the use of preprocessors improves the compression ratio achieved for all six preprocessors.

The compression programs perform compression on the file which has been transformed by the preprocessor. As mentioned, a higher payload ratio is expected on the transformed file than on the original file due to the change in redundancy.

Rank	Lowest	Average
1	A.1 (0.04048)	A.1 (0.04193)
2	A.2 (0.04352)	A.2 (0.04548)
3	B.1 (0.04645)	B.1 (0.04888)
4	B.2 (0.04840)	B.2 (0.05095)
5	C.1 (0.06807)	C.1 (0.06974)
6	C.2 (0.07520)	C.2 (0.07798)

Table 5.4: Difference between Payload Ratio (for all six preprocessors) and Compression Ratio (without preprocessing)

Table 5.4 illustrates the difference between the payload ratio and the compression ratio on the original files (without preprocessing). The results are the inverse of the overall compression ratios achieved by the preprocessors. Word definition C shows a higher payload ratio than word definition B and word definition A. The preprocessed files for word definition C are smaller than the files for word definition B, which are in turn smaller the the files for word definition A. The reductions by the preprocessor compensate for the higher payload ratios. The higher payload ratios are caused by the loss of context information and the higher ratio of binary characters.

WSC	(A,1)	(A,2)	(B,1)	(B,2)	(C,1)	(C,2)
Ratio	0.13156	0.12863	0.11268	0.10520	0.25901	0.22157

Table 5.5: Ratio of characters preprocessed files which are binary characters

WSC	(A,1)	(A,2)	(B,1)	(B,2)	(C,1)	(C,2)
Average length	4.78	6.35	6.19	9.73	5.36	7.76

Table 5.6: Average length of words in dictionary for each word-score combination

WSC	Avg # Replacements	WSC	Avg # Replacements	WSC	Avg # Replacements
(A,1)	17094721.27	(B,1)	13815354.13	(C,1)	20360521.67
(A,2)	16766379.27	(B,2)	13390792.8	(C,2)	1924265.47

Table 5.7: Average number of word-replacements by each preprocessor

Table 5.5 shows the average ratio of binary characters contained in the preprocessed files, Table 5.6 shows the average length of the words which are contained in the dictionaries for each WSC and Table 5.7 shows the average number of replacements which are made for each WSC on the test corpus. Table 5.6 shows that the average length of the words for word score two is larger than the average length of words for word score one for all word definitions. Word score two therefore selects longer words and thus replaces longer sequences with a single character. Table 5.7 shows that there are more replacements for word score one than word score two for all word definitions. There is therefore a lower frequency of longer words being replaced for word score two and a larger frequency of shorter more frequent words being replaced for word score two. Less replacements leads to a smaller ratio of binary characters (seen in Table 5.5). The smaller amount of replacements, however, results in a larger removal of context information, which leads to a higher payload ratio for word score two than word score one (seen in Table 5.4). This table also shows a larger amount of replacements for word definition B than word definition A. Table 5.6 shows that the average match length for word definition B is larger than word definition A for all word scores. The combinations of these two results results in a lower ratio of binary characters (seen in Table 5.5) and a greater improvement in compression ratio by word definition B than word definition A.

Rank	Lowest	Average
1	ppmd (0.06041)	ppmd (0.07472)
2	7zip (0.07373)	7zip (0.09068)
3	bzip2 (0.08658)	bzip2 (0.09291)
4	gzip (0.11889)	gzip (0.12801)
5	zip (0.11889)	zip (0.12801)
6	arj (0.12131)	arj (0.13409)
7	lzop(2) (0.14175)	lzop(2) (0.14212)
8	lzop(1) (0.18322)	lzop(1) (0.18399)

(a) Compression Ratio

Rank	Lowest	Average
1	ppmd (28)	ppmd (28)
2	7zip (77)	7zip (89)
3	bzip2 (126)	bzip2 (114)
4	gzip (206)	gzip (204)
5	zip (213)	zip (211)
6	arj (253)	arj (268)
7	lzop(2) (322)	lzop(2) (311)
8	lzop(1) (371)	lzop(1) (371)

(b) Summed Rank

Table 5.8: Compression Ratio for each Compression Program when using preprocessors

Table 5.8 (a) and Table 5.8 (b) both illustrate statistics for the lowest compression ratios of each program on the test corpus and the average compression ratios of each program on the test corpus when using the six preprocessors. Table 5.8 (a) shows the average compression ratio over the six preprocessors, while Table 5.8 (b) shows the summed rank. The ranks for each compression program - preprocessor combination is calculated using the technique previously described. The ranks achieved from 1 to 56 by each compression program with the use of the seven preprocessors are summed to give the summed rank for each compression program. The rankings of the compression programs shown in this table are the same as those presented in Section 3.2.1, which presents the compression ratios achieved by standard compression programs on a corpus of log files without preprocessing.

Table 5.9 shows the average improvement in compression ratio for standard compression programs when using preprocessors. As before, two compression ratio statistics are considered for each program (the lowest compression ratio achieved and the average compression ratio achieved) and `lzop` is split into two categories. `ppmd` is also split into 2 categories since there is a large difference between the results achieved by the first seven compression levels and compression levels greater than seven. The average improvement in compression ratio and the average percentage improvement in compression ratio are shown for all compression programs. The first seven levels of `ppmd` (`ppmd`(1-7)) achieve the highest amount of improvement and a higher average improvement than the other two statistical processors (`bzip2` and `7zip`), while `ppmd`(8-15) achieves a negative improvement in compression ratio. `lzop`, which achieves the highest compression ratio on average, achieves the greatest amount of improvement on average. Section 3.2.1

	Lowest			Average		
Rank	Program	Improve	% Improve	Program	Improve	% Improve
1	ppmd(1-7)	0.05308	26.17561	lzop(1)	0.03125	14.81530
2	arj	0.03041	16.36523	lzop(2)	0.01859	11.74164
3	lzop(1)	0.03134	14.90451	arj	0.01631	10.50141
4	gzip	0.02154	13.06344	ppmd(1-7)	0.01368	10.44285
5	zip	0.02154	13.06335	gzip	0.01457	10.19786
6	lzop(2)	0.01928	12.08503	zip	0.01457	10.19778
7	bzip2	0.01263	10.53796	bzip2	0.00632	6.23975
8	7zip	0.01045	10.05494	7zip	0.00572	5.83692
9	ppmd(8-15)	-0.00020	-0.28635	ppmd(8-15)	-0.00429	-6.91730

Table 5.9: Average Improvement in compression ratio for Compression Programs when using preprocessors

notes that `arj` achieves the highest compression ratio out of the LZ77-based compressors. This table shows `arj` achieving greater improvement in compression ratio than `gzip` and `zip`. This improvement, however does not lead to `arj` achieving a lower compression ratio than `gzip` or `zip` with the use of any of the preprocessors. As in Section 3.2.1, `gzip` and `zip`, which both use DEFLATE algorithm, achieve very similar compression ratios when using preprocessors. They show the same amount of improvement for all the preprocessors. Each of the compression levels of the compression programs show a varying amount of improvement. The results and analysis for each compression level can be found in Appendix E.3.

Summary

For all the compression programs with the exception of `7zip` and `ppmd`, all the compression levels show an improvement in compression ratio with the use of preprocessors. The higher compression levels of `ppmd` show a negative improvement, while compression levels one, four and eight of `7zip` show a negative improvement with the use of at least one preprocessor.

Ranking	Program	A.1	A.2	B.1	B.2	C.1	C.2
1 (5)	ppmd*	6	4	5	3	1	2
2 (6)	7zip	4	3	6	5	1	2
3 (7)	bzip2	6	4	5	3	2	1
4 (3)	gzip	6	4	5	3	2	1
5 (4)	zip	6	4	5	3	2	1
6 (2)	arj	6	4	5	3	2	1
7 (1)	lzop	6	5	4	3	1	2
	Average	5.71	4.57	5	3.29	1.57	1.43

Table 5.10: Compression program ranking summary

Table 5.10 shows a summary of ranking of the compression ratio achieved by the compression programs with the use of preprocessors. The ranking is shown in the first column and the ranking of the amount of improvement in compression ratio by that program is shown in brackets. The rankings of the improvement in compression ratio by the different preprocessors is also found in the table. `ppmd`* represents the first seven levels of `ppmd`. These compression levels show a positive improvement in compression ratio with the use of preprocessors. The ranking of the compression programs with the use of preprocessors is the same as the rankings of the compression programs without the use of preprocessors (it is also the same as the ranking of the programs presented in Section 3.2.1). For all compression programs, word definition C shows the highest average improvement in compression ratio and the compression levels which achieve higher compression ratios show a higher amount of improvement in compression ratio. The LZ77-based compressors and `lzop` show a higher amount of improvement than the statistical compressors (`7zip`, `bzip2` and `ppmd`). Out of the LZ77-based compressors, `arj` shows a higher amount of improvement than `zip` and `gzip`, but still achieves higher compression ratios than `zip` and `gzip`.

Table 5.11 shows the compression levels of the programs which achieve lower compression ratios than the lowest compression ratio for that program without preprocessing when using each of the six preprocessors. This table shows how the use of preprocessors improves the compression ratio achieved. For `zip` and `gzip`, all compression levels greater than or equal to four produce lower compression ratios for all preprocessors than than compression level nine achieves without the use of preprocessing.

	A.1	A.2	B.1	B.2	C.1	C.2
ppmd	-	-	-	-	-	-
7zip	8,10	10	10	10	10	10
bzip2	6,7,8,9	6,7,8,9	6,7,8,9	5,6,7,8,9	5,6,7,8,9	5,6,7,8,9
gzip	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9
zip	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9	4,5,6,7,8,9
arj	1,2	1,2	1,2	1,2,3	1,2,3	1,2,3
lzop(1)	-	-	-	-	-	-
lzop(2)	7,8,9	7,8,9	7,8,9	7,8,9	7,8,9	7,8,9

Table 5.11: Compression levels which show improvement on lowest compression ratio without preprocessing

5.3.2 Compression and Decompression Times

	MakeDict	GetDictionary	Compress	Decompress	Total	Reduction
A.1	124.594	17.9649	284.679	335.139	762.38	32.34%
A.2	124.594	18.6177	347.713	344.449	835.37	33.44 %
B.1	110.013	19.2553	270.393	304.503	704.16	34.38 %
B.2	110.013	19.864	316.915	310.595	757.39	35.71 %
C.1	172.061	67.3121	373.915	431.708	1045	43.53 %
C.2	172.061	68.4077	397.041	403.328	1040.84	45.69 %

Table 5.12: Times (in seconds) taken by preprocessors to perform the transformations

Table 5.12 shows the time taken by the preprocessors to perform reductions on the maillog corpus using the process presented in Section 5.1.5. This table exhibits the same trends which are seen for the 3MB maillog file (consisting of 1 day of log traffic). Word definition B achieves faster times than Word defintion A but achieves a greater reduction and the figures for getting the dictionary increase as the reduction increases. This is due to the number of matches and the number of unique words which satisfy the word score.

The compression and decompression times for the preprocessors are affected by the amount of times a word is replaced with a zero-frequency character and the number of unique words. The average amount of unique words contained in the corpus is 345 733 for word definition A, 357 172 for word definition B and 950 555 for word definition C, while the average amount of words which can be found in the file are 19 913 988.33, 16 316 646.33 and 26 243 978.13 for word definitions A, B and C respectively.

Program	None	A.1	A.2	B.1	B.2	C.1	C.2	Avg % Improve
lzop(1)	1.84	1.75	1.75	1.73	1.71	1.59	1.58	8.31
gzip	7.79	7.19	7.23	7.22	7.27	7.24	7.14	7.37
zip	8.63	8.15	8.27	8.12	8.24	8.04	7.97	5.73
arj	11.11	9.83	9.83	9.67	9.69	9.46	9.2	13.49
ppmd	33.34	31.21	31.23	30.16	30.49	30.5	29.6	8.43
lzop(2)	33.41	31.95	32.47	32.36	32.25	32.21	31.88	3.67
bzip2	100.05	61.08	58.26	56.76	54.51	48.15	44.29	46.19
7zip	174.64	134.84	132.3	131.45	129.81	81.06	82.92	33.92

Table 5.13: Compression Times (in seconds) for standard compression programs when using different preprocessors

For the six preprocessors, the dictionary is determined by the file which is being compressed. This means that a two pass process is involved: determining the dictionary and zero-frequency characters; and performing search and replacement using the dictionary mapping formed in the first pass. This means that these six preprocessors can not be integrated into the logging process since the entire log file is required. If the dictionary is predefined (perhaps using trends or dictionaries formed from the log file for the previous month), then the preprocessing step can be integrated into the logging process since it only requires in a single pass. This does, however, mean that the log files which are generated will not be human readable. A program which performs the search and replace and outputs log data can, however, be used to transform the output (in the same manner as `bzcat` can be used to look at files compressed using `bzip2`). There are also a number of advantages to using preprocessors on the log files. The dictionary can be used as an index and searches performed within the file can be performed faster due to smaller files. The dictionary generated on the file will also contain common hostnames and IP addresses which appear in the log file being compressed. Investigating the dictionary can also help to detect anomalies within the log file. An important part of security log management which involves creating reports which contain the top hostnames and IP addresses which appear in the log files [7]. These hostname and IP addresses would appear in the dictionary.

This section investigates the improvement in compression and decompression time without the inclusion of the time taken by the preprocessor to perform the transformation so that the effect of the use of the preprocessors can be seen. The times listed in Table 5.12 can be added to these results to see how much the compression and decompression times are increased when using the preprocessors.

Table 5.13 shows the average compression times achieved with the use of preprocessors and the

Program	None	A.1	A.2	B.1	B.2	C.1	C.2	Avg % Improve
zip	2.12	2.28	2.16	2.16	2.07	1.83	1.81	3.39 %
lzop(2)	2.7	3.26	3.07	3.04	3.09	2.42	2.13	-4.82 %
lzop(1)	3.05	3.43	3.37	3.35	3.20	2.53	2.36	0.30 %
gzip	3.36	3.69	3.47	3.50	3.45	2.77	2.64	3.00 %
arj	4.09	3.64	3.57	3.50	3.65	3.24	3.24	15.04 %
7zip	6.22	5.72	5.59	5.56	5.57	5.29	5.20	11.76 %
bzip2	19.09	14.37	13.97	13.85	13.72	12.45	11.85	29.98 %
ppmd	36.73	35.79	35.19	35.22	34.74	34.41	32.76	5.58 %

Table 5.14: Decompression Times (in seconds) for standard compression programs when using different preprocessors

average compression times for each compression program without the use of preprocessors. This table shows that the use of the preprocessors improves the compression times achieved; however, this improvement is not as great as the time taken by the preprocessor to perform the transform and reverse transform. An improvement in compression time is expected since the transformed file is smaller than the original file. In the initial semantic investigation, which is presented in Section 4.1, all the compression programs with the exception of `ppmd` show an improvement in compression time. Table 5.13, however, shows that all the compression programs (including `ppmd`) show an improvement on the average compression time achieved. This table shows an improvement in compression time of between 2.82 percent and 55.73 percent. `bzip2` and `7zip`, which are statistical compressors, both show large amounts of improvement. On average, `bzip2` shows an average improvement of 46.19 percent (which is the highest average out of all the compression programs), while `7zip` shows an average improvement of 33.92 percent (the second-highest average). Out of the LZ77-based compression programs, `arj` shows the highest amount of improvement on average. It achieves an average improvement of 13.49 percent, however for all six preprocessors it remained slower than both `zip` and `gzip`. `gzip`, which achieves faster compression times than `zip` without the use of a preprocessor, shows a higher average amount of improvement than `zip` (`gzip` achieves an average of 7.37 percent improvement, while `zip` shows 5.73 percent improvement). This table also shows that there are no changes in rank due to the improvements by the six preprocessors. The use of preprocessors, however does lead to `lzop(2)` and `7zip` achieving lower compression times than times achieved by `ppmd` and `bzip2` respectively without the use of preprocessing.

Table 5.14 shows the average decompression times achieved with the use of preprocessors. It also shows the average decompression times for each compression program without the use of

Program	None	A.1	A.2	B.1	B.2	C.1	C.2	Avg % Improve
lzop(1)	4.89	5.17	5.12	5.08	4.91	4.12	3.94	3.31 %
zip	10.75	10.43	10.43	10.28	10.31	9.87	9.78	5.27 %
gzip	11.15	10.89	10.7	10.72	10.72	10.01	9.79	6.05 %
arj	15.2	13.47	13.4	13.16	13.35	12.7	12.44	13.91 %
lzop(2)	36.12	35.21	35.54	35.4	35.34	34.62	34.01	3.03 %
ppmd	70.07	67	66.42	65.38	65.23	64.91	62.36	6.93 %
bzip2	119.15	75.45	72.23	70.62	68.22	60.6	56.14	43.59 %
7zip	180.86	140.56	137.89	137.01	135.38	86.35	88.12	33.16 %
Average	56.03	44.99	44.2	43.71	43.15	35.6	34.76	26.71 %

Table 5.15: Total Times (in seconds) for standard compression programs when using different preprocessors

preprocessors. This table shows an increase in the decompression times when using word definition A or word definition B with `zip`, `lzop(2)`, `lzop(1)` and `gzip`. Word definition C, however shows an improvement in the average decompression time for all compression programs. Out of all the programs, `bzip2` shows the highest amount of improvement in decompression time. It achieves an average improvement of 29.98 percent. This improvement, however does not lead to it achieving faster decompression times than `7zip` (which shows an average improvement of 11.76 percent). `ppmd`, which achieves the slowest compression times, shows the least amount of improvement out of the statistical compressors (5.58 percent improvement). Out of the LZ77-based compression programs, `arj` achieves the highest amount of improvement in decompression time (15.04 percent on average). This improvement leads to `arj` achieving faster decompression times than `gzip` for A.1 and the same decompression time as `gzip` for B.1. `zip` shows a smaller increase in decompression time than `gzip` for word definition A and word definition B, however `gzip` shows a greater percentage of improvement for word definition C (19.33 percent on average) than `zip` (14.38 percent on average). This greater improvement, however does not lead to `gzip` achieving faster decompression times than `zip` for word definition C. Its times for word definition C are also faster than `gzip`'s times without the use of preprocessing. Out of all compression programs, `lzop` shows the least improvement in decompression times. It does, however, show a greater percentage of improvement than `ppmd` for word definition C. The improvement in decompression time with the use of preprocessors leads to C.1 and C.2 of `arj` achieving faster decompression times than `gzip` without the use of a preprocessor. It also leads to C.1 and C.2 of `gzip` achieving faster decompression times than `lzop(2)` without the use of preprocessing, C.2 of `gzip` achieving faster times than `lzop(1)` and C.1 and C.2 of `lzop(1)` faster than `lzop(2)` without the use of preprocessing.

Table 5.15 shows the average total times achieved with the use of preprocessors and the average total times without the use of a preprocessors. The total time is calculated by the addition of the compression time and decompression time. This is calculated for all levels of a compression program and then the average is obtained.

Despite showing the largest amount of improvement in total time out of the LZ77-based compression programs (13.91 percent on average), `arj` remains the slower than `zip` and `gzip`. `gzip` also shows a higher percentage of improvement than `zip`, however `zip` still achieves faster times than `gzip` for all six preprocessors. `bzip2` shows the highest percentage of improvement in total (43.59 percent on average). The improvement is due to the smaller files created by the preprocessors. Most of the time for `bzip2` is spent performing the BWT, which now only has to transform smaller files. This results in much faster times for `bzip2`. Due to the high percentage of improvement, `bzip2` achieves faster times than `ppmd` for C.1 and C.2. `ppmd` shows the lowest average percentage of improvement out of the statistical compressors (6.93 percent). `7zip`, which achieves the highest average times, shows a 33.16 percent improvement in total time. This leads to it achieving faster total times for C.1 and C.2 than `bzip2` without preprocessing. `lzop` shows the smallest amount of improvement out of the compression programs with `lzop(1)` showing greater improvement than `lzop(2)`. `lzop(1)` still, however, achieves the fastest total time with and without the use of preprocessors.

The average total time over all compression programs is improved by between 19.71 percent and 37.96 percent with the use of preprocessors. This time difference, however, is not high enough to provide an improvement when the times for the preprocessors are included. In this case, it results in average times which are between 14 times (for A.1) and 19 (for C.2) times greater than the average times without the use of a preprocessor.

For all the compression programs, the different compression levels shows different amounts of improvement which make up the averages presented in the section. Appendix E.3 investigates the improvements in the compression and decompression times for each compression level of the various programs.

Relationship between the Improvement in times and the reduction by the preprocessors

As shown in the previous sections, each of the six preprocessors results in a reduction in the compression and decompression times. The reduction in compression and decompression time is due to the reduction in payload size which is to be compressed. The number of different characters in the files, however is increased. This section shows that the improvement in compression time

and the improvement in decompression time are both highly correlated with the reduction in the payload size by the preprocessor.

	Comp	Decomp
7zip	-0.97321	-0.99161
arj	-0.95968	-0.93166
bzip2	-0.99232	-0.99598
lzop	-0.73988	-0.99418
ppmd	-0.74455	-0.91146
lzop(1)	-0.98742	-0.99251
gzip	-0.99335	-0.99320
zip	-0.97504	-0.98931

Table 5.16: Correlation (using Pearson correlation coefficient) between the improvement in time and the reduction by the preprocessor

Table 5.16 shows the correlation between the improvement in compression time and the reduction by the preprocessor and the correlation between the improvement in decompression time with the reduction by the preprocessor using the Pearson correlation coefficient. This shows a strong correlation between the improvement in the times for all compression programs except `lzop` and `ppmd` for compression. `lzop(1)`, however shows a strong correlation.

The average of all the improvements in compression times shows a negative correlation of 0.98858 with the reduction by the preprocessor. This indicates a strong linear relationship between the compression ratio achieved by the preprocessor and the improvement in compression time. The negative coefficient means that as the ratio increases, the improvement in compression time decreases. The average of all improvements in decompression times also shows a negative correlation coefficient of 0.98122, indicating a strong linear relationship between the improvement in decompression time and the compression ratio achieved by the preprocessor.

These results indicate that inferences about the improvement in compression and decompression time can be made using the reduction by the preprocessor to determine the magnitude of the improvement in compression and decompression time.

Summary

The use of preprocessors leads to an improvement in compression time of up to 61.71 percent, an improvement in decompression time of up to 39.73 percent and hence an improvement in total time of up to 61.71 percent. The improvement in these times is due to the reduction in the filesize by the preprocessor. The greatest improvement by the preprocessors, however, is not greater than the time taken by the preprocessor to perform the reductions on the maillog corpus. All the compression programs, except `lzop(2)`, achieve an improvement in compression time, decompression time and total time when using the preprocessors. `lzop(2)` achieves a negative improvement in the average decompression time with the use of preprocessors but does achieve a positive improvement for compression time and total time. This improvement, however is the lowest out of all the compression programs.

Program	Comp	Decomp	Total
lzop(1)	1 (7)	3 (7)	1 (7)
zip	3 (5)	1 (5)	2 (5)
gzip	2 (6)	4 (6)	3 (6)
arj	4 (3)	5 (2)	4 (3)
lzop(2)	6 (8)	2 (8)	5 (8)
ppmd	5 (4)	8 (4)	6 (4)
bzip2	7 (1)	7 (1)	7 (1)
7zip	8 (2)	6 (3)	8 (2)

Table 5.17: Rankings of times (improvement) for standard compression programs when using preprocessors

Table 5.17 shows a summary of rankings of the times achieved by the compression programs with the use of preprocessors. The ranking of the program's average times are shown, while the ranking of the average amount of improvement is shown in brackets. This table shows the rankings for the compression time, decompression time and total time. For compression, decompression and total time, `bzip2` shows the highest improvement, while `lzop(2)` shows the lowest improvement. `lzop(2)` achieves a negative amount of improvement in decompression time. `arj` achieves a higher amount of improvement than `7zip` for decompression time, otherwise the ranking are the same as the ones for compression and decompression. `zip` achieves a higher ranking than `gzip` for the average improvement in compression time, decompression time

and total time. `gzip`, however still achieves a faster compression time than `zip` despite a lower amount of improvement. `arj` achieves the highest improvement out of the LZ77-based compression programs. It achieves a greater improvement in compression time, decompression time and total time than `zip` and `gzip`. This improvement, however, does not lead to `arj` achieving faster times than `gzip` or `zip` on average. `bzip2` and `7zip` shows the highest average total times, but achieve the highest and second-highest average improvement in total time respectively. The statistical compressors achieve the highest amount of improvement, followed by the LZ77-based compressors. `arj`, however, does achieve a higher amount of improvement than `ppmd`.

	A.1			A.2			B.1			B.2			C.1			C.2		
Program	T	C	D	T	C	D	T	C	D	T	C	D	T	C	D	T	C	D
lzop(1)	6	5	6	5	6	5	4	4	4	3	3	3	2	2	2	1	1	1
zip	6	5	6	5	6	5	4	3	4	3	4	3	2	1	2	1	2	1
gzip	6	6	6	4	5	4	5	4	5	3	3	3	2	2	2	1	1	1
arj	6	5	5	5	6	4	3	3	3	4	4	6	2	2	2	1	1	1
lzop(2)	3	2	6	6	6	4	5	5	3	4	4	5	2	3	2	1	1	1
ppmd	6	6	6	5	5	4	4	2	5	2	3	3	3	4	2	1	1	1
bzip2	6	6	6	5	5	5	4	4	4	3	3	3	2	2	2	1	1	1
7zip	6	6	6	5	5	5	4	4	3	3	3	4	1	1	2	2	2	1
Average	6	5	6	5	6	5	4	4	4	3	3	3	2	2	2	1	1	1

T = Total Time, C = Compression Time, D = Decompression Time

Table 5.18: Preprocessors rankings for improvement in time

Table 5.18 shows a breakdown of the rankings of the improvements achieved by individual preprocessors, detailing the improvement in compression time (C), decompression time (D) and total time (T). The average ranking is calculated by ranking the average rankings achieved across all programs. C.2 achieves the lowest average ranking for the improvement in total time and compression time, while A.1 achieves the highest average ranking for improvement in total time and compression time. C.2 shows the lowest average ranking for improvement in decompression time, however A.2 achieves a lower ranking than A.1, which achieves the highest ranking for improvement in decompression time. In order of rank, the average improvement achieved by the preprocessors is: C.2, C.1, B.2, B.1, A.1, A.2 (from most improved to least improved). These rankings are the same as the ranking of the reduction by the preprocessors. This relationship can also be seen by the high correlation between the improvement in the times for compression

and decompression and the reduction by the preprocessors. The reduction by the preprocessor is therefore a good predictor of the improvement in time caused by the use of a preprocessor.

5.3.3 Transfer Times

As mentioned in Section 3.2.3, the transfer time combines the results for the compression ratio and total times into a practical metric which can be used to evaluate the performance in a given scenario. Since the use of preprocessors generally causes the compression programs to produce lower compression ratios and faster total times, the transfer times are improved on average. This section investigates the improvement in transfer times on the corpus with the use of preprocessors.

The maillog corpus exhibits a larger average file size than the corpus of log files presented in Chapter 3. This causes the compression payload to be larger than in Chapter 3, which in turn leads to an increase in the transfers times. Section 3.2.3 shows that at lower transfer speeds, the compression levels which achieve lower compression ratios (resulting in a smaller payload to transfer) and higher total times achieve the lower transfer times, whereas at higher transfer speeds compression levels which achieve higher compression ratios and faster times achieve lower transfer times. Effectively the transfer times is a weighted average of the payload size and the compression time where the weighting of the total times is always one and the weighting of the payload size is inversely proportional to the transfer speed (its has a weighting of $\frac{1}{t}$ where t is the transfer speed).

Figure 5.3 shows the difference between the lowest transfer times for each program and the lowest overall transfer time at a given transfer speed when transferring the maillog corpus without the use of preprocessors. In this figure, `ppmd` shows the lowest transfer time until 261 KBps at which point `zip` achieves the lowest transfer time. `zip` achieves the lowest transfer time until 7527 KBps, at which point `lzop(1)` becomes the fastest. `lzop(1)` initially shows the slowest transfer times and becomes faster than `lzop(2)` at 527 KBps. `bzip2` shows a rapid drop from being third fastest at 10KBps to being the slowest at 251 KBps. On the corpus of log files in Chapter 3, `ppmd` initially achieved the fastest time until 81 KBps at which point `7zip` became the fastest. `7zip` remained the fastest until 236 KBps when `zip` became the fastest. `zip` remained fastest until 9615 KBps. `bzip2` showed a rapid drop achieving last position at 194 KBps. The trends shown in the figure are similar results to those presented in Section 3.2.3 with the exception of `7zip` achieving the fastest time. The transitions, however occur later than in Section 3.2.3. This is because of the larger corpus (and hence larger time to transfer the payload).

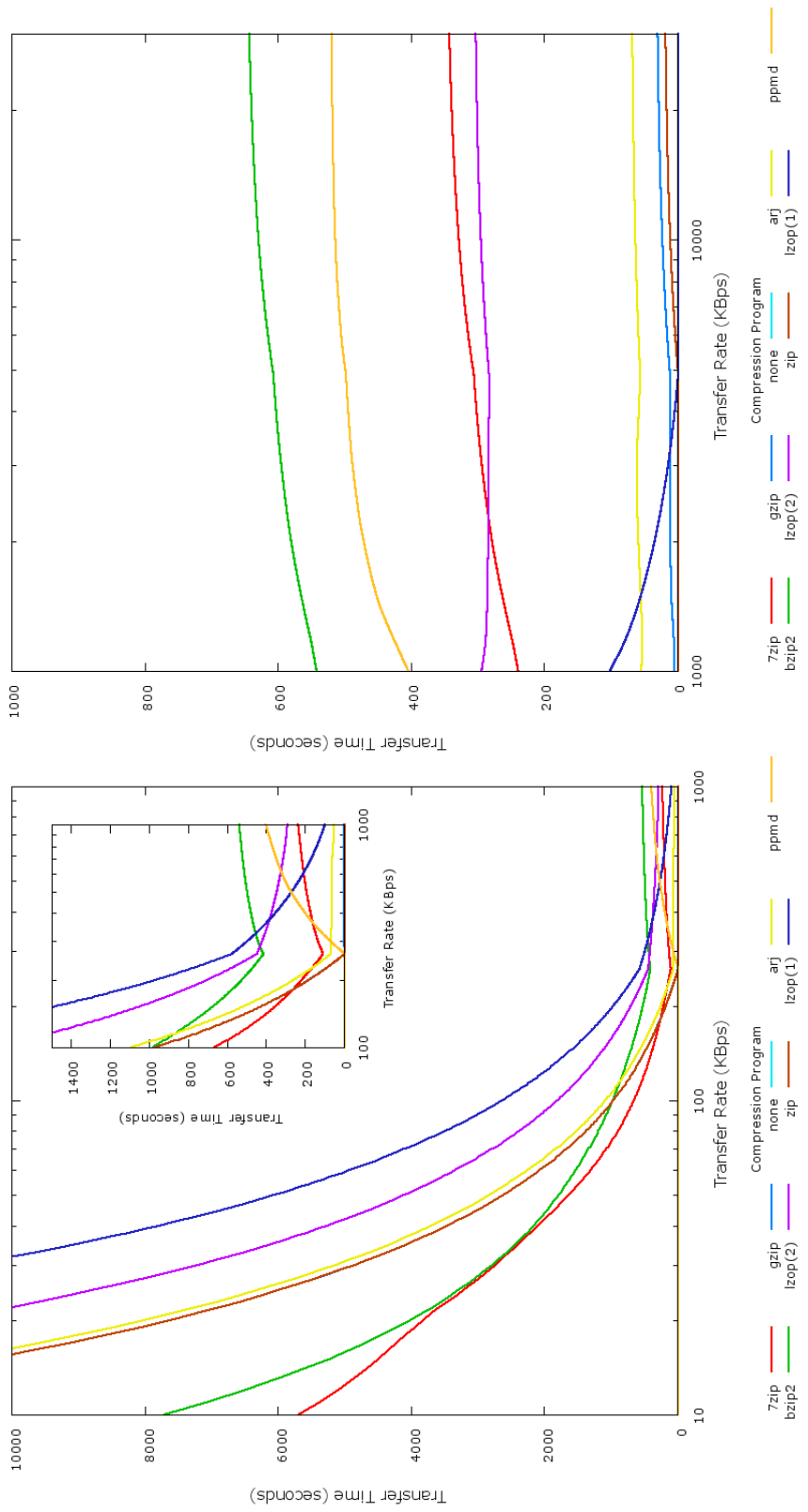


Figure 5.3: Difference between lowest transfer times for each program and lowest overall transfer time for the maillog corpus

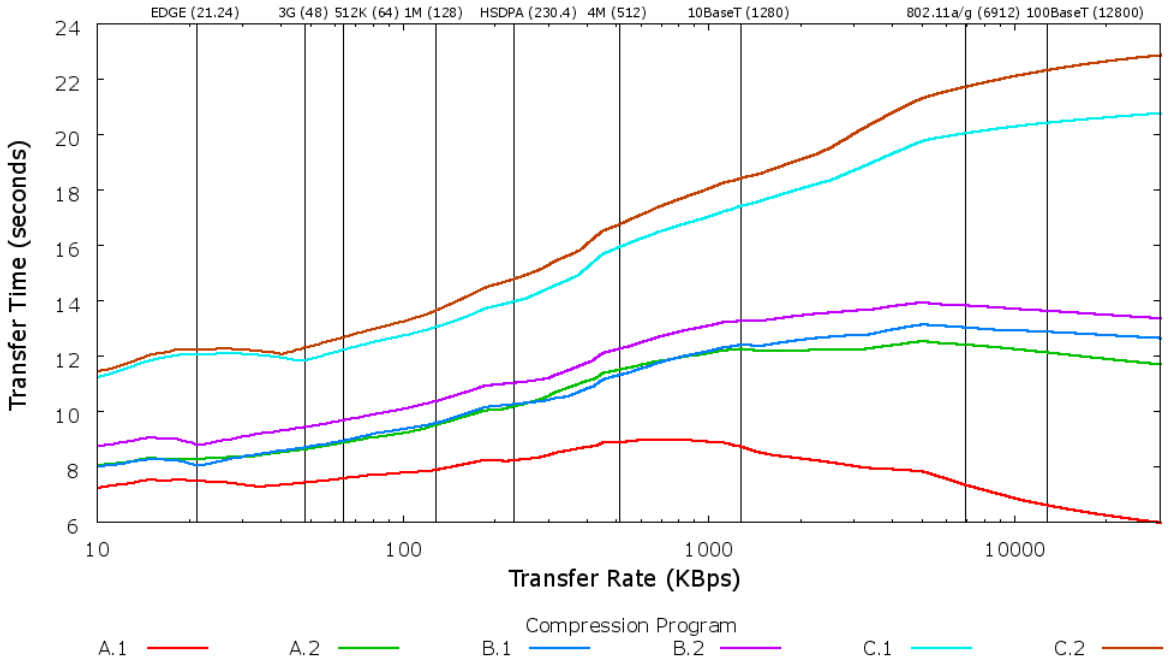


Figure 5.4: Average Percentage Improvement in Transfer Time for the six preprocessors

Figure 5.4 shows the average percentage improvement in transfer times with the use of preprocessors. This figure shows C.2 achieving the highest percentage of improvement for all transfer speeds. By inspection, it can be seen that B.1 and A.2 perform comparably until 768 KBps, while B.2 achieves a greater improvement than A.2. At its peak, the use of preprocessors shows reductions in transfer time of up to 22.89 percent.

Figure 5.5 shows the difference between the lowest average transfer time at a given speed and the lowest average transfer time achieved by each of the compression programs with the use of preprocessors. This figure shows how the improvement in total time and compression ratio for `bzip2` improves its transfer times. The gap between `bzip2` and the other statistical preprocessors is improved in comparison to Figure 5.3. `arj` shows the greatest improvement out of the LZ77-based compressors for both compression ratio and total time. This improvement causes the gap between `arj` and the other LZ77-based compressors (`zip` and `gzip`) to be reduced.

Table 5.19 shows the transfer speeds (in KBps) at which transitions occur between the compression programs when using the six different preprocessors. It also shows the points at which the transitions occur when the average time for the preprocessors is taken. The values which are larger than the original values (without the use of preprocessors) are shown in bold. For all the preprocessors, the transfer speed from which `bzip2` achieves the slowest transfer time out of all

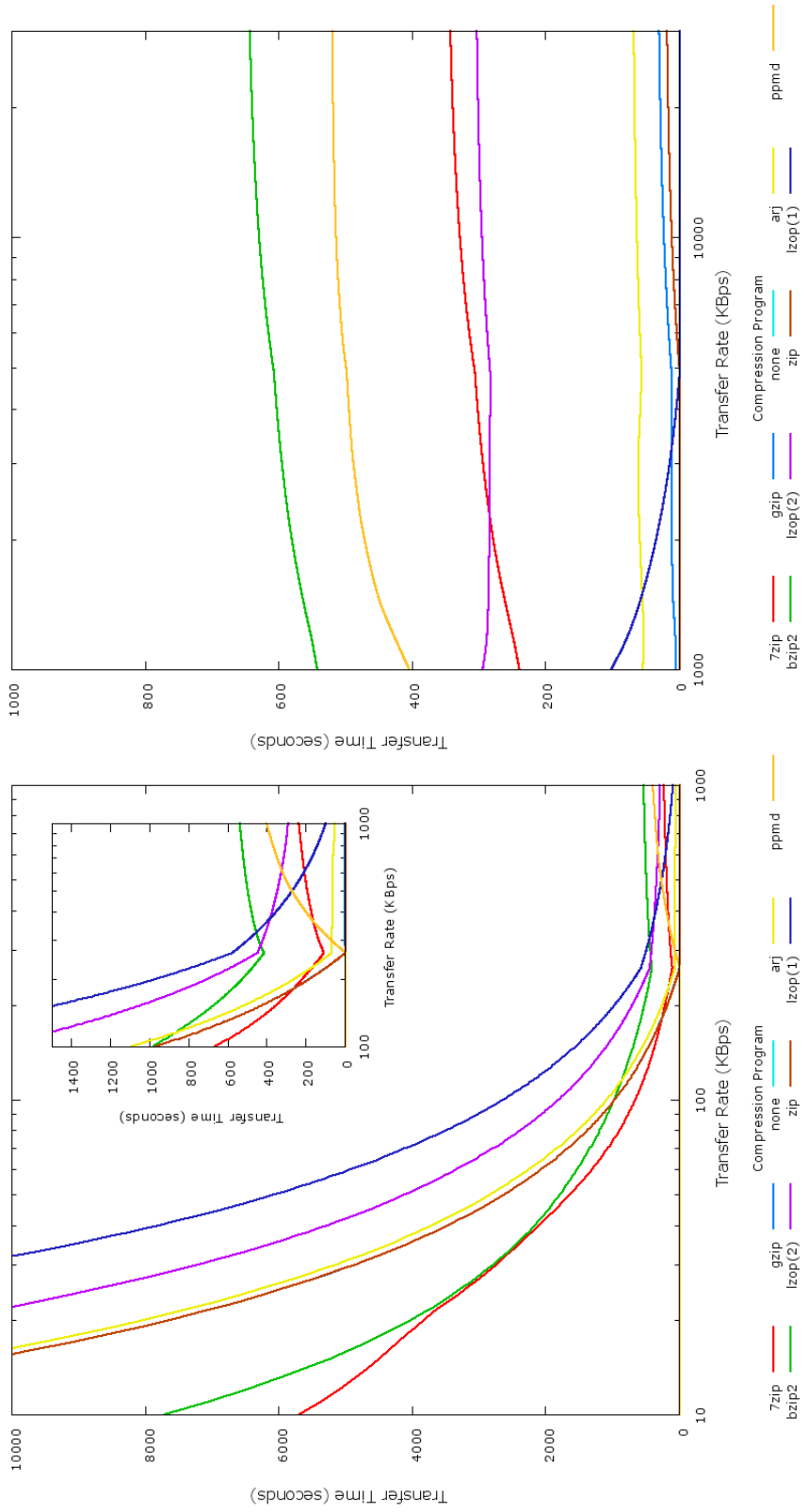


Figure 5.5: Differences between lowest average transfer times over all word scores for each algorithm

	None	A.1	A.2	B.1	B.2	C.1	C.2	Average
ppmd -> zip	261	254	257	258	261	**270**	**282**	**263**
ppmd -> gzip	267	255	258	260	262	**270**	**282**	264
zip -> lzop(1)	7527	5311	6522	5441	5712	3561	3588	4874
lzop(2) -> lzop(1)	557	291	422	**1384**	411	372	379	375
bzip2 last	251	**347**	**325**	**325**	**338**	**308**	**341**	**325**
7zip -> zip	186	165	**188**	184	**189**	**193**	174	182
7zip -> gzip	194	167	187	186	190	192	174	182
gzip -> zip	-	**28**	**188**	**28**	**26**	**261**	**848**	**106**
gzip -> lzop(1)	3673	3511	3501	3137	3253	2876	1308	3213

Table 5.19: Transfer speeds (in KBps) for different preprocessors at which transitions occurs between the compression programs

the compression programs is improved. This is due to the great improvement in total time with the use of preprocessors. `ppmd` achieves a larger amount of improvement in total time than `zip` and `gzip` when using C.1 and C.2. This causes the transfer speed at which the transitions occur to increase. For lower compression levels, `gzip` achieves a greater amount of time improvement than `zip`. This causes the transition from `gzip` to `zip` to be increased. This is since `gzip` achieves greater amount of improvement in total time than `zip` for higher compression levels (which produces the fastest transfer times at these transfer speeds) and also achieves lower compression ratios than `zip`. The transfer speed at which the transition from `7zip` to `zip` occurs is increased due to the same reason. Generally, however, the transfer speed at which a transition occurs with the use of preprocessors is reduced due to the reduction in filesize by the preprocessors.

Figure 5.6 shows the average percentage of improvement in the fastest transfer time by each compression program with the use of preprocessors. This figure shows how the improvement in compression ratio and total time for `bzip2` leads to it achieving the greatest improvement in transfer time. As the transfer speed increases, the weighting on the total time increases. Since `bzip2` achieves the greatest amount of improvement in total time, it follows that as the transfer speed increases, so too does the improvement in transfer time. This figure also shows how the improvement in `ppmd` is not high for lower transfer speeds. This is because `ppmd` shows an increase in compression ratio and compression times for the higher compression levels. `ppmd` achieves the lowest compression ratio, which leads to it achieving fastest transfer times due to the lower compression ratio.

Table 5.20 shows the transfer speed at which each compression program no longer achieves a faster transfer speed than the transfer speed without the use of compression or preprocessors.

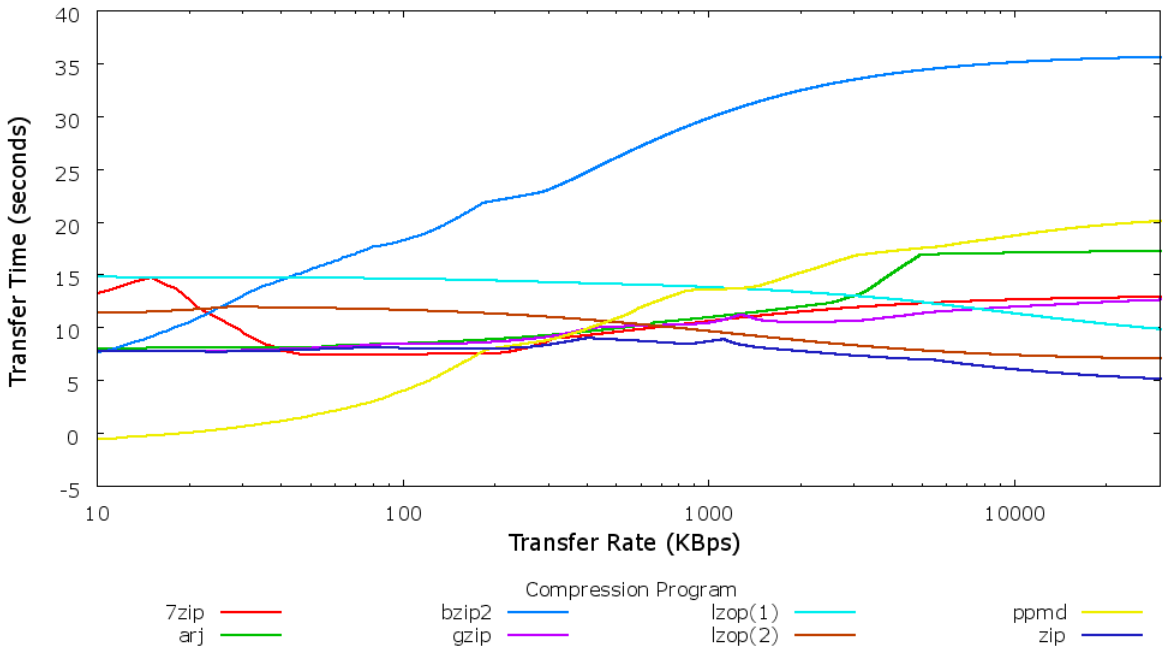


Figure 5.6: Average percentage improvement in the fastest transfer time by compression programs when using preprocessors

The compressed size is reduced by the preprocessor, which leads a smaller payload being compressed. This table shows that the transfer time at which the compression programs are no longer beneficial has increased with the use of preprocessors. The preprocessed file, however can be transferred. This file is reduced in size already and hence shows a faster transfer time than the transfer time without the use of preprocessors.

Table 5.21 shows the transfer time at which the transfer speed from which the compression program with the use of preprocessors is slower than the use of no compression, transferring

	A.1	A.2	B.1	B.2	C.1	C.2	None
zip	26592	27900	27932	27999	29977	30878	26722
lzop(1)	33229	33013	34513	34822	42967	44002	32208
lzop(2)	4971	7076	7159	7075	7694	7786	6192
gzip	23114	23917	23943	23910	28607	28444	21446
arj	17346	17485	17919	17630	19140	19734	14468
7zip	6164	6291	6241	6311	6564	6488	5458
bzip2	3336	3485	3551	3662	3978	4225	2325
ppmd	4113	4247	4187	4376	4336	4868	3502

Table 5.20: Transfer speed (in KBps) at which compression programs are no longer beneficial

	A.1	A.2	B.1	B.2	C.1	C.2	None
zip	16682	17078	16737	16376	14862	14516	26722
lzop(1)	20488	19602	20043	19693	20475	19772	32208
lzop(2)	3161	4338	4299	4146	3827	3675	6192
gzip	14680	14639	14347	13985	14141	13372	21446
arj	10899	10597	10627	10202	9339	9136	14468
7zip	4005	3961	3835	3798	3379	3176	5458
bzip2	2166	2189	2185	2204	2042	2070	2325
ppmd	2680	2675	2587	2641	2224	2305	3502

Table 5.21: Transfer speed (in KBps) at which compression programs are no longer beneficial (preprocessed)

the preprocessed file. This table shows a reduction in the figures for all compression programs. As expected, there is a reduction in the maximum beneficial times. This is since the use of preprocessors reduces the filesize which results in a lower transfer time for the preprocessed file then the original file.

Each compression level of the different compression program shows a different amount of improvement in transfer time. Some compression levels (such as compression level two of `lzop(1)`) achieve fast transfer time at high transfer speeds, while achieving slow transfer times at lower transfer speeds. Other compression levels (such as compression level nine of `bzip2`) achieve faster transfer times at low transfer speeds but slow transfer times at higher transfer speeds. Appendix E.3 investigates the improvement in transfer time for each compression level of `bzip2`, `gzip` and `zip`, `arj`, `lzop`, `7zip` and `ppmd`.

Summary

The improvement in compression ratio and total time by the preprocessors leads to an improvement in transfer time. The transfer time is essentially a linear combination (or weighted average) of the compression ratio and the total time. Section 3.2.3 showed that at lower transfer speeds, the compression ratio achieved is more significant than the total time in determining the transfer time. As the transfer speed increases, the significance of the total time in determining the transfer time increases. It therefore follows that a higher improvement in compression ratio has greater bearing on the improvement in transfer time at lower transfer speeds than the improvement in total time, while at higher transfer speeds, the improvement in total time has a greater bearing on the improvement in transfer time.

	1.2	3.6	7	8	10	21.25	
	GSM	28.8K	56K	64K	GPRS	EDGE	Average
ppmd	1 (8)	1 (8)	1 (8)	1 (8)	1 (8)	1 (8)	1 (8)
7zip	2 (2)	2 (2)	2 (2)	2 (2)	2 (2)	2 (2)	2 (2)
bzip2	3 (7)	3 (7)	3 (7)	3 (7)	3 (7)	3 (4)	3 (6.5)
gzip	4 (5)	4 (5)	4 (5)	4 (5)	4 (5)	4 (6)	4 (5.17)
zip	5 (6)	5 (6)	5 (6)	5 (6)	5 (6)	5 (7)	5 (6.17)
arj	6 (4)	6 (4)	6 (4)	6 (4)	6 (4)	6 (5)	6 (4.17)
lzop(2)	7 (3)	7 (3)	7 (3)	7 (3)	7 (3)	7 (3)	7 (3)
lzop(1)	8 (1)	8 (1)	8 (1)	8 (1)	8 (1)	8 (1)	8 (1)
none	9	9	9	9	9	9	9

Table 5.22: Ranking of transfer time (improvement in transfer time) for compression programs at slower compression speeds

Table 5.22 shows the rankings of the average transfer times achieved by the various compression programs with the use of preprocessors. It shows the ranking of the average transfer times for the maximum transfer speeds which can be achieved by GSM, a 28.8 Kbps modem, a 56.6 Kbps modem, a 64 Kbps DIGINET line, GPRS and EDGE. These are slower transmission technologies which are used today. The rank of the improvement in transfer time is shown in brackets. `lzop(1)`, which achieves the slowest transfer times for these transfer speeds, achieves the highest amount of improvement. This is since it achieves the highest improvement in compression ratio. At these transfer speeds, the higher compression levels of `ppmd` achieve the lowest transfer speeds. There are no compression levels for `ppmd` which achieve lower compression ratios than the optimal compression level of `ppmd`. Therefore, at these transfer speeds, `ppmd` shows the least improvement. `arj` achieves a higher amount of improvement than `zip` and `gzip` for both compression ratio and total time, so it, therefore, achieves a higher improvement in transfer time.

Table 5.23 shows the rank of the average transfer times achieved by the various compression programs with the use of preprocessors. It shows the ranking of the average transfer times for the maximum transfer speeds which can be achieved by 3G, 512 Kbps ADSL, 1 Mbps ADSL, HSDPA and 4 Mbps ADSL. These are faster transmission technologies which are classified as broadband. The rank of the improvement in transfer time is shown in brackets. For these transfer speeds, `bzip2` achieves the highest improvement in transfer time. This is since it achieves the highest improvement in total time. `7zip` shows the second least amount of improvement. This is since at these transfer speeds, compression levels, the fast and fastest methods of `7zip` achieve the fastest times. These compression levels do not show great improvement in total time or compression ratio.

	48	64	128	230.4	512	
	3G	512K	1M	HSDPA	4M	Average
ppmd	1 (8)	1 (8)	1 (8)	1 (6)	6 (3)	2 (6.6)
zip	5 (6)	5 (6)	3 (6)	2 (7)	1 (8)	3.2 (6.6)
gzip	4 (5)	4 (5)	4 (5)	3 (5)	2 (5)	3.4 (5)
arj	6 (4)	6 (4)	5 (4)	4 (4)	3 (6)	4.8 (4.4)
7zip	2 (7)	2 (7)	2 (7)	5 (8)	4 (7)	3 (7.2)
bzip2	3 (1)	3 (1)	6 (1)	6 (1)	8 (1)	5.2 (1)
lzop(2)	7 (3)	7 (3)	7 (3)	7 (3)	7 (4)	7 (3.2)
lzop(1)	8 (2)	8 (2)	8 (2)	8 (2)	5 (2)	7.4 (2)
none	9	9	9	9	9	9 (9)

Table 5.23: Ranking of transfer time (improvement in transfer time) for compression programs at broadband connection speeds

	1280	6912	12800		>6912
	10BaseT	802.11a/g	100BaseT	Average	Average
zip	1 (8)	2 (8)	2 (8)	1.67 (8)	2 (8)
lzop(1)	4 (3)	1 (5)	1 (6)	2 (4.67)	1 (5.5)
gzip	2 (5)	3 (6)	3 (5)	2.67 (5.33)	3 (5.5)
arj	3 (4)	4 (3)	4 (3)	3.67 (3.33)	4 (3)
lzop(2)	6 (7)	5 (7)	5 (7)	5.33 (7)	5 (7)
7zip	5 (6)	6 (4)	6 (4)	5.67 (4.67)	6 (4)
ppmd	7 (2)	7 (2)	7 (2)	7 (2)	7 (2)
bzip2	8 (1)	8 (1)	8 (1)	8 (1)	8 (1)
none	9 (9)	9 (9)	9 (9)	9 (9)	9 (9)

Table 5.24: Ranking of transfer time (improvement in transfer time) for compression programs at LAN and Wireless connection speeds

Table 5.24 shows the rank of the average transfer times achieved by the various compression programs with the use of preprocessors. It shows the ranking of the average transfer times for the maximum transfer speeds which can achieved by 10BaseT, 802.11a/g and 100BaseT networks. These are the technologies used in LAN environments. The rank of the improvement in transfer time is shown in brackets. `ppmd` shows the second-highest amount of improvement. This is since at these transfer speeds, compression level one achieves the fastest transfer times for `ppmd`. Compression level one of `ppmd` achieves a high amount of improvement in both compression ratio and total time.

	A.1	A.2	B.1	B.2	C.1	C.2
zip	6	5	4	3	2	1
lzop(1)	6	5	4	3	2	1
gzip	6	4	5	3	2	1
arj	6	5	3	4	2	1
lzop(2)	6	5	3	4	2	1
7zip	6	4	5	3	1	2
ppmd	6	4	5	2	3	1
bzip2	6	5	4	3	2	1
none	6	5	4	3	2	1
Average	6	4.67	4.11	3.11	2	1.11

Table 5.25: Rank of average improvement in transfer time for standard compression programs when using preprocessors

Table 5.25 shows the rank of the average improvement in transfer time by the preprocessors for each compression program. For all transfer speeds, A.1 achieves the slowest average transfer times, while C.2 achieves the fastest average transfer times. This table shows that A.1 achieves the slowest transfer time for all compression programs, while C.2 achieves the fastest transfer time for all compression programs except `7zip`. For `7zip`, C.1 achieves faster transfer times than C.2. This is since C.1 achieves a higher improvement in total time and compression ratio than C.2 for `7zip`. The average ranking for the preprocessors is the same as the ranking as observed for the improvement in total time (C.2, C.1, B.2, B.1, A.2, A.1). At lower transfer speeds, however, the average ranking observed is C.2, C.1, B.2, A.2, B.1, A.1. This is the ranking achieved for the improvement in compression ratio.

5.3.4 Memory Utilisation

The previous sections have shown that the use of preprocessors generally causes an improvement in the compression ratio, compression times and decompression times which are achieved. The memory utilisation by the compression program is also an important factor in the selection of compression programs for a particular scenario. When embedded devices are being used (such as in wireless sensor networks [138, 139]), there is a limited amount of memory available and the selection of a compression program is subject to the constraints for the scenario. The

preprocessors transform the text files into files which contain binary characters in the place of certain words. This leads to a greater amount of characters and hence larger symbol tables. The files which are being compressed, however are smaller than the original files, which can lead to an improvement in memory utilisation. This section investigates the improvement in memory utilisation with the use of preprocessors.

	Average	
	Log Files	Maillogs
gzip	0.92	0.92
lzop(1)	1.19	1.19
zip	1.29	1.29
lzop	1.31	1.31
arj	1.41	1.41
lzop(2)	1.55	1.55
bzip2	4.57	4.57
ppmd	72.91	100.36
7zip	169.89	190.56

(a) Compression

	Average	
	Log Files	Maillogs
gzip	0.8	0.8
lzop(1)	1.05	1.05
lzop	1.05	1.05
lzop(2)	1.05	1.05
zip	1.28	1.28
arj	1.3	1.3
bzip2	2.78	2.78
7zip	19.01	20.85
ppmd	72.94	100.39

(b) Decompression

Table 5.26: Average memory utilisation (in MB) by compression programs on the two corpii

Table 5.26 shows the average amount of memory usage for all the compression levels of the respective compression programs on the corpus of log files (presented in Section 3.2.4) and the corpus of maillog files. The average file size of maillog corpus, as mentioned, is larger than the corpus of log files presented in Chapter 3. This table shows differences for `ppmd` and `7zip`. They both show an increase in the average amount of memory usage due to the larger corpus.

	A.1	A.2	B.1	B.2	C.1	C.2
7zip	0.76	0.97	1.02	1.29	3.56	4.24
lzop	-0.01	-0.02	-0.02	-0.02	-0.02	-0.03
ppmd	-0.95	-0.41	-0.95	-0.29	-5.85	-4.17
lzop(1)	-0.02	-0.02	-0.02	-0.02	-0.02	-0.03
lzop(2)	-0.01	-0.01	-0.02	-0.02	-0.02	-0.02

Table 5.27: Improvement in Memory usage for compression when using preprocessors

Table 5.27 shows the average amount of improvement in the memory utilisation for compression with the use of six preprocessors. The compression programs which are not shown do not show a change in the amount of memory utilisation for compression with the use of preprocessors. These improvements cause no change in the rank of the compression programs in terms of their memory utilisation. Out of the programs shown in this table, only `7zip` indicates an improvement in the average amount of memory utilisation for compression. The rest of the programs show an increased memory utilisation for compression. The use of word score one results in a smaller amount of improvement than the use of word score two for all the compression programs. This is due to a higher ratio of binary characters for word score one and the smaller filesizes for word score two.

	A.1	A.2	B.1	B.2	C.1	C.2
7zip	0	0.01	0.05	0.05	0.29	0.37
ppmd	-0.95	-0.4	-0.96	-0.32	-5.83	-4.21

Table 5.28: Improvement in Memory usage for decompression when using preprocessors

Table 5.28 shows the improvement in the amount of memory usage for decompression with the use of preprocessors. `ppmd` achieves a similar amount of improvement in memory usage for decompression and compression. As is the case for compression, `7zip` shows an improvement in memory usage, while `ppmd` shows an increase in memory usage. Word score one also shows a smaller amount of improvement than word score two for both `ppmd` and `7zip`.

The difference in memory utilisation is caused by how the compression programs use memory. For example if the compression program uses a dictionary then it depends on how much of the dictionary gets used and if it uses a symbol table then it depends on the number of symbols which are used. If it builds a tree of contexts, then more symbols will lead to a faster growth of the tree and hence the maximum memory utilisation will be reached earlier.

Summary

Out of all the programs, `7zip` is the only compression program which shows an improvement in the amount of memory utilisation. This is achieved due to an improvement in the memory utilisation by the ultra compression method since some of the files in the corpus are smaller than the dictionary size. The files which are larger than the dictionary size show a constant amount of memory utilisation. All the compression programs except `7zip` and `ppmd` show the same amount

of memory utilisation, on average, for the maillog corpus as they showed for the log files corpus presented in Section 3.2.4. `lzop` shows an increase in memory utilisation for compression, while `ppmd` shows an increase in memory utilisation for both compression and decompression. `bzip2`, `gzip` and `zip` show no improvement in memory utilisation with the use of preprocessors.

5.3.5 Summary of Results

The previous sections have shown that the use of preprocessors provides an improvement in the compression ratio and compression times. They also show that the memory usage with the use of preprocessors, however, is increased or remains constant for all programs except `7zip`. The six preprocessors based on the six word - score combinations achieve a greater amount of improvement than the preprocessors presented in the initial semantic investigation (which replaced the timestamps and IP addresses with their binary equivalents).

		Time				Memory			Overall
	Ratio	C	D	T	Avg	C	D	Avg	Average
gzip	4 (4)	2 (6)	4 (6)	3 (5)	3 (5.67)	1 (2)	1 (2)	1 (2)	2.67 (3.89)
zip	5 (5)	3 (7)	1 (5)	2 (6)	2 (6)	3 (2)	4 (2)	3.5 (2)	3.5 (4.33)
lzop(1)	8 (1)	1 (5)	3 (7)	1 (7)	1.67 (6.33)	2 (4)	2 (2)	2 (3)	3.89 (3.44)
arj	6 (3)	4 (3)	5 (2)	4 (3)	4.33 (2.67)	4 (2)	5 (2)	4.5 (2)	4.94 (2.56)
lzop(2)	7 (2)	6 (8)	2 (8)	5 (8)	4.33 (8)	5 (3)	3 (2)	4 (2.5)	5.11 (4.17)
bzip2	3 (6)	7 (1)	7 (1)	7 (1)	7 (1)	6 (2)	6 (2)	6 (2)	5.33 (3)
ppmd	1 (8)	5 (4)	8 (4)	6 (4)	6.33 (4)	7 (5)	7 (3)	7 (4)	4.78 (5.33)
7zip	2 (7)	8 (2)	6 (3)	8 (2)	7.33 (2.33)	8 (1)	8 (1)	8 (1)	5.78 (3.44)

C = Compression, D = Decompression, T = Total, Avg = Average

Table 5.29: Rankings of the results (Ranking of improvement) achieved by compression programs with the use of preprocessors

Table 5.29 presents a summary of the results presented in the previous four sections. It shows the rankings of the average results achieved with the use of preprocessors for each compression program. The table shows the average results for compression ratio, compression time, decompression time, total time and the improvement in memory usage for compression and decompression. The table also shows the rankings for average improvement achieved with the use of preprocessors for each compression program (This is shown in brackets). Overall, `arj` shows

the greatest improvement with the use of preprocessors, followed by `bzip2`, while `ppmd` shows the least improvement. `lzop(1)` achieve a greater amount of improvement than `lzop(2)`. The improvements, however do not lead to a change in the rankings of the compression programs which are observed in Chapter 3.

		Time				Overall
	Ratio	Comp	Decomp	Total	Average	Average
A.1	6	6	5	6	5.67	5.84
A.2	4	5	6	5	5.33	4.67
B.1	5	4	4	4	4	4.5
B.2	3	3	3	3	3	3
C.1	2	2	2	2	2	2
C.2	1	1	1	1	1	1

Table 5.30: Rankings of improvements by the preprocessors

Table 5.30 presents the ranks of the improvements by the preprocessors for compression ratio, compression time, decompression time and total time. C.2 shows the highest average amount of improvement in compression ratio, compression time, decompression time and total time. This leads to it achieving the highest overall average rank for improvement. A.1 achieves the least amount of improvement in compression ratio, compression time and total time. A.2 achieves a lower amount of improvement than A.1 for decompression time. Overall, however, A.1 achieves a lower average rank for the time improvement, which leads to it achieving the lowest overall rank for improvement. The order of the average improvement (C.2, C.1, B.2, B.1, A.2, A.1) by the preprocessors is the same as the order of the reduction by the preprocessors.

The dictionaries used for replacement in this section are generated from the files which are being compressed. This section has shown that the time taken by the preprocessors to perform the word replacement is longer than the improvement in total time achieved with the use of preprocessors. This section has shown that the use of preprocessors improves the performance of standard compression programs.

5.4 Summary

Section 5.1 discussed a method for constructing a dictionary for a file based on a given word definition and word score. Six word - score combinations were then used to create dictionaries which were used for word-replacement using the zero-frequency characters. The results for the improvement in compression ratio, compression time, decompression time, total time, transfer time and memory utilisation achieved by the different compression programs were presented in Section 5.3. These preprocessors reduced the size of the maillog corpus by between 31.34 percent and 45.69 percent and resulted in an average improvement in compression ratio of up to 14.82 and an average improvement in total time of up to 43.59 for standard compression programs. A summary of these results was presented in the previous section. Through these results, this chapter has shown that preprocessors which perform word-based replacement can be used to exploit the semantic knowledge present in the log file. The next chapter investigates dictionaries which be can used to compress the files that are generated from previous data and their performance is compared to C.2 which achieves the highest average improvement of out the preprocessors presented so far.

Chapter 6

Further Semantic Compression

The previous chapter showed that by creating a dictionary based on the frequency and length of the words contained in a file and using it to perform word-based replacement, a significant improvement in the performance of the compression program can be obtained. The problem with this methodology is that the entire file needs to be parsed in order to create the dictionary. Section 5.3.2 showed that the time taken by the preprocessor to transform the file is significantly longer than the improvement in time achieved by the compression programs with the use of the preprocessors. Section 5.3.3 also showed that the improvement in transfer time is also not greater than the time taken by the preprocessor.

In order to integrate the preprocessors into the logging system, the dictionary needs to exist before the file is logged. This means that the dictionaries presented in the previous chapter cannot be integrated into the logging system. Dictionaries generated from past log data or dictionaries based on the semantic knowledge present in log files can, however, be integrated into the logging system. The previous chapter showed that C.2 achieved the greatest amount of improvement out of the six preprocessors presented. The dictionaries investigated in this chapter are therefore based on WSC (word-score combination) (C,2) - the WSC used by C.2.

This chapter begins by investigating the performance of the preprocessors when using the previous month's dictionary. It then presents an analysis of the dictionaries for the corpus and presents a methodology for the construction of a custom dictionary based on the semantic knowledge contained in the log files and the knowledge known about the network. Definitions for the metrics (compression time, decompression time, compression ratio and payload ratio) used in this chapter may be found in Section 2.1.4.

6.1 Using the previous month's dictionary

In order for the preprocessor to be used, it needs to use a dictionary which is created before the log file is compressed. This means that in most scenarios, the preprocessor needs to use a previously generated dictionary or a dictionary which is specific to the file. This section investigates using the previous month's dictionary to perform preprocessing on the log file.

6.1.1 Methodology

When using word-based replacement, part of the timestamp is replaced by zero-frequency characters. The dictionary contains a word which refers to the month e.g. The dictionary for March 2006 contains the word "`Mar`". To make the previous month's dictionary more effective this month can be updated to form a new dictionary i.e. the word "`Mar`" is replaced by the word "`Apr`" (since it is going to be applied to the log file for April 2006) to form a new dictionary. These two dictionaries (before and after the alteration) are referred to as PastMonth and PastMonthUpdated. These dictionaries are mapped to the zero-frequency characters for the log file to which it is going to be applied. The preprocessing is then performed in the same manner as described in Section 5.1.4. The resulting files are then compressed and decompressed using each of compression levels for the seven compression programs. A MD5 and SHA1 hash is performed before and after to validate that the files before and after the transformations are the same. PM is used to refer to the preprocessor which uses the dictionary PastMonth, while PMU is used to refer to the preprocessors which uses the dictionary PastMonthUpdated.

6.1.2 Results and Analysis

Table 6.1 shows the compression ratios for PM and PMU. As expected, the compression ratios for PMU are lower than the compression ratios for PM. The compression ratios for both PM and PMU are higher than the compression ratios achieved by C.2. These compression ratios are on average 5.29 percent lower than PM and 2.88 percent lower than PMU. This is expected, since C.2 uses the optimal dictionary based on WSC (C,2) for the log file, whereas PM and PMU uses dictionaries based on the optimal dictionary for the previous month.

Month	C.2	PM	PMU	Month	C.2	PM	PMU
mar2006	0.54553	-	-	nov2006	0.54720	0.56968	0.55550
apr2006	0.54934	0.57913	0.56400	dec2006	0.55621	0.57995	0.56624
may2006	0.54868	0.58743	0.57256	jan2007	0.55434	0.56936	0.55513
jun2006	0.53226	0.56199	0.54708	feb2007	0.55380	0.57084	0.55680
jul2006	0.52342	0.55027	0.53541	mar2007	0.53982	0.57725	0.56298
aug2006	0.54116	0.56890	0.55499	apr2007	0.52794	0.56084	0.54652
sep2006	0.54924	0.57105	0.55708	may2007	0.52722	0.60896	0.59680
oct2006	0.55035	0.57307	0.55903	Average	0.54310	0.57348	0.55893

Table 6.1: Performance of own and previous month dictionaries

Program	C.2	PM	PMU
7zip	0.0107	0.0120	0.0127
arj	0.0361	0.0341	0.0350
bzip2	0.0152	0.0141	0.0146
gzip	0.0264	0.0247	0.0254
lzop	0.0396	0.0376	0.0383
ppmd	0.0706	0.0688	0.0691
zip	0.0264	0.0247	0.0254
lzop(1)	0.0396	0.0376	0.0383
lzop(2)	0.0240	0.0227	0.0234
ppmd(1-7)	0.0706	0.0688	0.0691
ppmd(8-15)	-0.0005	0.0018	-0.0005

(a) Maximum Improvement

Program	C.2	PM	PMU
7zip	0.0062	0.0078	0.0082
arj	0.0197	0.0183	0.0190
bzip2	0.0077	0.0072	0.0074
gzip	0.0178	0.0166	0.0172
lzop	0.0341	0.0323	0.0330
ppmd	0.0062	0.0070	0.0061
zip	0.0178	0.0166	0.0172
lzop(1)	0.0395	0.0374	0.0381
lzop(2)	0.0234	0.0221	0.0228
ppmd(1-7)	0.0189	0.0192	0.0186
ppmd(8-15)	-0.0045	-0.0034	-0.0044

(b) Average Improvement

Table 6.2: Improvement in compression ratio using PM and PMU

Table 6.2 shows the improvement in compression ratio for standard compression programs when using C.2, PM and PMU. Table 6.2 (a) shows the maximum improvement for a compression program, while Table 6.2 (b) shows the average amount of improvement for a compression program. These tables show that the same rank in improvement for compression programs as seen for C.2. All the compression programs except `ppmd` and `7zip` show a greater improvement in compression ratio when using C.2 than PM and PMU. PMU achieves a greater amount of improvement than PM for all compression programs except for `ppmd`.

6.2 Combining The Results of Other Dictionary

Using the log data from previous months to create a dictionary for the preprocessor means that a dictionary needs to be kept for each month so that the data can be compressed. The dictionaries are rather small, however if they are lost, important information such as hostnames and IP addresses, which are replaced with zero-frequency characters will be hard to recover. A single dictionary which can be used on all the files and achieve an acceptable improvement in compression ratio is desired. In this section, a dictionary is formed by combining the dictionaries which were generated for each of the files in the maillog corpus in attempt to create a single dictionary which can be used on all the files and achieve an acceptable compression ratio. This section describes the methodology used for combining the dictionaries and presents the compression ratio achieved and the amount of improvement achieved for each compression program when using a preprocessor based on this dictionary.

6.2.1 Methodology

To begin with, the top 256 words using WSC (C,2) for each of the log files are determined. Once these words are determined, the words which occur in less than seven of the fifteen log files are discarded.

Num of Dictionaries	1	2	3	4	5	6	7
Num of Words	682	389	315	275	261	245	234
Excl 2 Digit Numbers	622	329	255	215	201	185	174

Num of Dictionaries	8	9	10	11	12	13	14	15
Num of Words	220	213	201	186	178	160	146	135
Excl 2 Digit Numbers	160	153	141	126	118	100	86	75

Table 6.3: Number of Words which can be found in given amount of dictionaries

Table 6.3 shows the number of words and the number of words excluding the words which are two digit numbers which appear in a given number of dictionaries. Seven is chosen for two reasons: firstly it is approximately half of the dictionaries ($\lfloor \frac{15}{2} \rfloor = 7$); and secondly for any larger number of dictionaries, the number of words which satisfy the condition is less than 161 (which is the highest number of zero-frequency characters for all files in the corpus of maillogs). The resulting set of words are ranked according to their average position within the the

dictionaries. This ranking is used as the word score when selecting the words to be replaced by zero frequency characters. Interestingly the word set formed contains a number of two digit integers. These twp digit integers are included due to their frequency of occurrence within timestamp data. Using this word set, two dictionaries are formed: CombNum and CombNoNum. The first dictionary (CombNum) includes these integers in the dictionary, while the second dictionary (CombNoNum) does not include these integers in the dictionary. These dictionaries also include the 12 month prefixes from the timestamp (i.e. Jan, Feb, Mar, etc.). These prefixes achieve the second-highest word score out of all the words for their respective dictionaries, however fail the requirement of occurring in seven files. This dictionary is meant to be able to be used as a single dictionary on all the log files and hence they all need to be included. From this point, the name of the dictionary is used to refer to the preprocessor which uses the respective dictionary.

6.2.2 Results and Analysis

File	C.2	PMU	All	CombNoNum	CombNum
mar2006	0.54553	-	0.57901	0.61787	0.56992
apr2006	0.54934	0.57913	0.57924	0.61541	0.57041
may2006	0.54868	0.58743	0.56736	0.60166	0.55646
jun2006	0.53226	0.56199	0.55704	0.59280	0.54786
jul2006	0.52342	0.55027	0.55474	0.58999	0.54650
aug2006	0.54116	0.56890	0.56591	0.60524	0.55922
sep2006	0.54924	0.57105	0.56804	0.60897	0.56081
oct2006	0.55035	0.57307	0.56714	0.61234	0.56023
nov2006	0.54720	0.56968	0.56743	0.61214	0.55851
dec2006	0.55621	0.57995	0.57225	0.61011	0.56641
jan2007	0.55434	0.56936	0.56529	0.61160	0.56238
feb2007	0.55380	0.57084	0.56491	0.61198	0.55714
mar2007	0.53982	0.57725	0.56511	0.61453	0.55986
apr2007	0.52794	0.56084	0.56124	0.61664	0.56079
may2007	0.52722	0.60896	0.55794	0.63663	0.60536
Average	0.54310	0.57348	0.56618	0.61053	0.56279

Table 6.4: Results for CombNum and CombNoNum dictionaries

Table 6.4 shows the results for the compression ratio achieved by COMBNUM and COMBNONUM. ALL is a preprocessor which uses a dictionary constructed on the entire corpus using WSC (C,2). The results for C.2, PMU and ALL are included in the table for comparison. As expected, COMBNONUM achieves a higher compression ratio than COMBNUM for log files. On average, COMBNONUM achieves the highest average compression ratio out of the preprocessors show in the table. COMBNUM, however achieves lower compression ratios than PMU for all fifteen log files and lower compression ratios than ALL for all the log files except `may2007`. This leads to COMBNUM achieving a lower average compression ratio then PMU and ALL. A hypothesis test to determine if the paired difference between PMU and COMBNUM is significant (i.e Using a null hypothesis of $\mu_d = 0$) confirms that the paired difference is significant (p-value 0.00008). The resulting files from the preprocessors are compressed using standard compression programs.

	C.2	PMU	All	CombNoNum	CombNum
7zip	0.0062	0.0082	0.0081	0.0110	0.0083
arj	0.0197	0.0190	0.0184	0.0174	0.0188
bzip2	0.0077	0.0074	0.0070	0.0069	0.0073
gzip	0.0178	0.0172	0.0168	0.0160	0.0171
lzop	0.0341	0.0330	0.0318	0.0260	0.0326
ppmd	0.0062	0.0061	0.0056	0.0042	0.0059
zip	0.0178	0.0172	0.0168	0.0160	0.0171
lzop(1)	0.0395	0.0381	0.0367	0.0297	0.0376
lzop(2)	0.0234	0.0228	0.0221	0.0186	0.0226
ppmd(1-7)	0.0189	0.0186	0.0177	0.0134	0.0184
ppmd(8-15)	-0.0045	-0.0044	-0.0046	-0.0037	-0.0045

Table 6.5: Average improvement in compression ratio when using COMBNONUM and COMBNUM

Table 6.5 shows the average improvement in compression ratio for the compression programs by COMBNONUM and COMBNUM. This table also shows the improvements in compression ratio for the compression programs by C.2, PMU and ALL. COMBNUM shows a higher amount of improvement than COMBNONUM for all compression programs except `7zip`. COMBNUM also achieves a higher amount of improvement than ALL for all compression programs. PMU, however, achieves a higher amount of improvement than COMBNUM for all compression programs

except `7zip`. A hypothesis test to determine if the paired difference between PMU and COMBNUM is significant (i.e Using a null hypothesis of $\mu_d = 0$) confirms that the paired difference is significant (p-value 0.00368). On average, the improvement by PMU is 0.74 percent greater than the improvement by COMBNUM.

6.2.3 Summary

This section has shown that COMBNUM achieves a higher compression ratio than PMU, but achieves a lower amount of improvement in the compression ratio for the compression programs than PMU. For all compression programs except `7zip`, however, neither of these preprocessors achieve greater improvement than C.2. PMU and COMBNUM, however, can be integrated into the logging process since they do not depend on the log file which is being compressed. COMBNUM uses a single dictionary which means that many different dictionaries do not need to be kept. Forming a dictionary such as COMBNUM also requires much analysis and does not draw on the semantic knowledge and knowledge of the network, but rather on the results for the dictionaries. A more specialised dictionary can be formed by using other known semantic information from the analysis of the log files and concatenating common phrases which always occur after each other. The next section investigates the contents of the dictionaries and their relationship to the semantics present in the log file. It also discusses a methodology for forming dictionaries based on this investigation and presents the results of using those dictionaries with preprocessors.

6.3 Constructing a Custom Dictionary

The previous section showed that a single dictionary can be produced which achieves a compression ratio which is only 3.63 percent higher than the compression ratio achieved by C.2 and 0.6 percent lower than the compression ratio achieved by the dictionary formed on the entire corpus. This section investigates the semantics present in a log and analyses the dictionaries presented so far to form a new dictionary which achieves the lowest compression ratio on all files in the corpus and is based on known semantics and the knowledge of the network structure involved.

6.3.1 Methodology

The initial aim is to develop a dictionary based on the known semantics and the knowledge of the network structure involved. The second aim is to improve this dictionary so that it achieves a

lower compression ratio than C.2 on average while still using words (i.e. not including phrases). The final aim is to include phrases into the dictionary which improve the compression ratio achieved when using the previous dictionary.

In developing these dictionaries, the spiral model for software development, which was defined by Boehm [140], is used. This is an iterative development model where for each successive prototype, objectives are identified for the next prototype, the results are analysed and changes are made to meet objectives resulting in the next prototype.

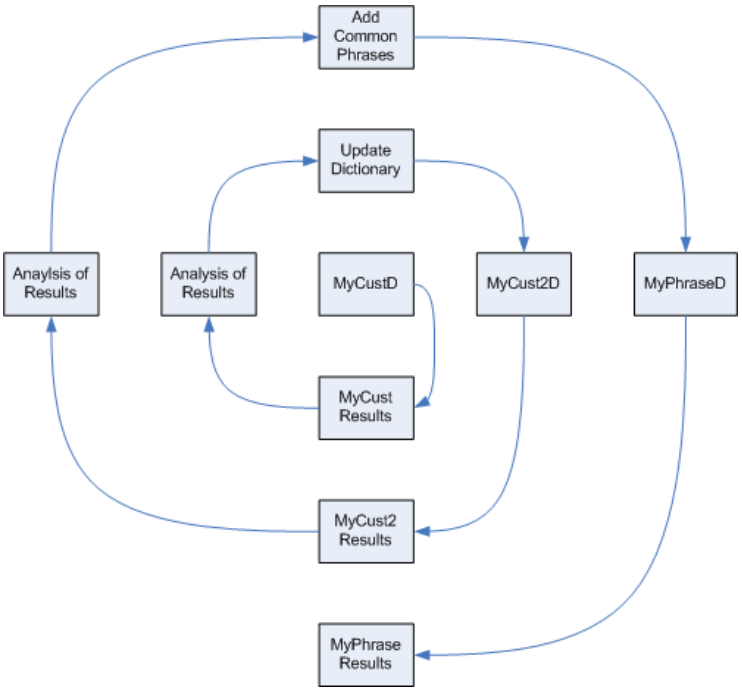


Figure 6.1: Spiral model for developing final custom dictionary

Figure 6.1 shows the spiral model which is used for the development of the final custom dictionary. In the spiral model, the first prototype is constructed from a preliminary design and is usually a scaled-down version which represents an approximation of the characteristics of the final product. In this case, the prototype is a dictionary, referred to as MYCUSTD, which is formed using the knowledge discovered by analysing the words which occur in the dictionaries for C.2, COMBNUM and ALL, the network structure and the semantics from the analysis of the maillog files (presented in Section 4.2). The preprocessor which uses this dictionary is referred to as MYCUST. The objective for the second prototype is for the preprocessor based on that dictionary to achieve a lower compression ratio than C.2. The second prototype, a dictionary referred to

	Num	Used	Freq		Num	Used	Freq
apr2006	160	94	228	may2006	160	96	8
apr2007	160	94	1	may2007	160	96	6
dec2006	158	94, 232, 96	3, 2, 1	nov2006	160	96	4
feb2007	160	96	12	oct2006	160	96	2
mar2007	159	252, 96	4, 8	sep2006	160	96	12

Table 6.6: Characters from the list of zero-frequency characters which are used within files of the maillog corpus

as MYCUST2D, is formed by performing analysis of the context in which the words occur and adjusting the dictionary accordingly. The preprocessor which uses MYCUST2D is referred to as MYCUST2. The objective of the final prototype, which is a dictionary called MYPHRASED, is to include phrases which improve the compression ratio achieved by a preprocessor. The final prototype is formed by adding phrases to the dictionary which are beneficial. This includes joining words which occur together and adding phrases which are observed in the semantic investigation whose gain is higher than words which occur in the dictionary. The gain is calculated by multiplying the length less one and the frequency of the phrase. The preprocessor based on MYPHRASED is referred to as MYPHRASE.

This section discusses the development of each of these three dictionaries. The first part presents an analysis of the contents of the previous dictionaries and discusses the semantics involved. The second part discusses how to build a dictionary based on the semantics present in the log file and the knowledge of the network structure. The resulting dictionary is MYCUSTD. The third part investigates the contexts in which these words occur, and discusses how this dictionary can be improved. The resulting dictionary is MYCUST2D. The final part explores the common phrases contained within a maillog file which can be added to the dictionary to improve the compression ratio achieved by the preprocessor. The resulting dictionary is MYPHRASED.

Analysis of dictionaries

To determine the required size of the dictionary, the number of zero-frequency characters within the log files need to be determined. The files in the maillog corpus contain between 158 and 161 zero-frequency characters (average of 160.13). This means that the dictionary needs to contain at least 161 words in order to take advantage of all the zero-frequency characters present in the log file. These characters are all contained within the character numbers 0-9, 11-31, 94, 96 and 127-254.

Table 6.6 shows the characters from the list of zero-frequency characters which are used within certain files of the maillog. It also shows the frequency of occurrence of these characters within that maillog. The maillog files for which all 161 zero-frequency characters are unused are omitted from this table. This table shows that the frequency of these characters is rather low. Since two "words" cannot be followed by each other - "words" are always separated by "non-words" - all 161 characters can be used in the dictionary mapping. When these characters do occur in the file, they can just be escaped by doubling them. This does not result in a larger file size since the frequency of these characters is much lower than the improvement due to word replacement using that character. This means that the dictionary mapping can be determined prior to knowledge of the zero-frequency characters.

The dictionary for COMBNUM (shown in the previous section) is essentially a summary of the common words contained within the dictionaries for C.2. The contents of this dictionary show that a dictionary can be constructed through the knowledge of the network and the knowledge of the semantics which are present in log files. It contains elements such as: hostnames; IP addresses; field names (such as `Message-ID`, `to` and `from`); process names (such as `amavis` and `postfix`); months and numbers from the timestamp data.

The dictionary for ALL is generated using WSC (C,2) on the entire corpus of maillogs. COMBNUM achieves a lower compression ratio than ALL for all files except may2007. The dictionary of the entire corpus (dictionary for ALL) displays a number of differences to the dictionary for COMBNUM. Inspection shows that the followings words are not included in the dictionary COMBNUM but are included in the dictionary for ALL:

`256, 3886, 34868, 196.23.167.10, 2.5.0, 203.188.197.9, a.mx.ru.ac.za, ADH-AES256-SHA, altair.sac.ecape.school.za, BOUNCE, cipher, conn_use, error, established, facebookmail.com, hs.facebook.com, leopard.gc.ecape.school.za, mx1.mail.tw.yahoo.com, mx2.mail.tw.yahoo.com, setting, TLS, TLSv1, www.facebook.com, yahoo.com.tw`

The inclusion of these words causes an improvement in the compression ratio achieved for `may2007`. The analysis of the maillogs, presented in Section 4.2, shows that the following lines can be found which include the words: `256`, `TLS`, `TLSv1`, `cipher`, `established`, `ADH-AES256-SHA` and `setting`.

- `TLS connection established from` *`hostname[IP]`*`: TLSv1 with cipher ADH-AES256-SHA (256/256 bits)`
- `setting up TLS connection from` *`hostname[IP]`*

These log lines are caused by using TLS between mailservers for secure exchange of messages. These words only start appearing in the log files from March 2007 and hence are not inluded in the dictionary for COMBNUM.

The words which are included in the dictionary for ALL but not in the dictionary for COMBNUM include a number of references to yahoo. `mx1.mail.tw.yahoo.com`, `mx2.mail.tw.yahoo.com` and `yahoo.com.tw` are all related to yahoo, while the IP address `203.188.197.9` resolves to a yahoo mailserver. These are only included in the `may2007` dictionary and are hence not included in the dictionary for COMBNUM.

Some of the words which are included in the dictionary for ALL, but not in the dictionary for COMBNUM include references to facebook. These include: words such as `hs.facebook.com`, `www.facebook.com` and `facebookmail.com`. The growth in the popularity of facebook, causes these words to start appearing in the dictionaries from April 2007.

The inclusion of the words `3886` and `34868` in the dictionary for ALL is interesting. At first glance, these numbers appear to not have any significance. They are not ports which are used for anything and it is not clear from the semantic investigation either. These numbers are, however, included in the dictionaries for `dec2006` and `jan2007`, and `sep2006` and `oct2006` respectively. Closer investigation reveals that the PID for `postfix/qmgr` did not change from 3886 between 18 December 2006 and 31 January 2007. 3886 is therefore in the dictionary for dec2006 and jan2007. The PID for `postfix/qmgr` was 34848 between 18 September 2006 and 16 October 2006, therefore 34868 was included in the dictionaries for `sep2006` and `oct2006`. This leads these numbers to be included in the dictionary for ALL.

The words `altair.sac.ecape.school.za`, `leopard.gc.ecape.school.za` and `196.23.167.10` (which resolves to lair.moria.org) are all due to the network structure and various changes during the course of the 15 months. `altair.sac.ecape.school.za` is included in the dictionaries for the first three months, while `mercury.sac.ecape.school.za` is not included in the dictionaries for the first two months. `196.23.167.10` is also only included in dictionaries for the first three months. `leopard.gc.ecape.school.za` is a server on the network, however the load of messages which contain this hostname produced causes it to only appear in the dictionaries for three of the maillog files.

Words such as those caused by the addition of TLS and those relating to facebook can be included in the dictionary to improve the compression ratio on the newer log files and future log files. There are a number of words which appear in all of the dictionaries for C.2 (and hence in COMBNUM). These words include:

titania, postfix, 127.0.0.1, gauntlet.mithral.co.za, 209.67.212.202, smtpd, mail.kc.ecape.school.za, from, localhost, disconnect, kc.ecape.school.za, connect, message-id, qmgr, status, cleanup, active, relay, removed, sac.ecape.school.za, delay, client, dsg.ecape.school.za, 2.6.0, queue, nrcpt, sacschool.com, smtp, Message-ID, kwc-ntfs002.kc.ecape.school.za, size, sent, 250, mail_id, queued, amavis, dsgschool.com, to, NOQUEUE, Passed, delivery, connection, rejected, 10025, reject, CLEAN, Recipient, services.async.org.za, address, Hits, RCPT, proto, vghs.ecape.school.za, deferred, Ok, statistics, helo, MTA, ESMTP, for, mail, Operation, id, command, ecape.school.za, saprep.ecape.school.za, saprepschool.com, async.org.za, as, not, ms, ru.ac.za, gc.ecape.school.za, timed, virtual

These words contain hostnames and IP addresses which are used in the network and words which have been identified during the investigation into the semantics of maillogs (presented in Section 4.2)

The dictionaries for C.2 also include a number of 2 digit numbers from 00 to 60. These numbers are included in the dictionary since the timestamps, which occur on every line, use numbers from 00 to 60. For all the log files, all the numbers from 00 to 60 are included in the top 255 words based on WSC (C,2). Some of these numbers, however are not included in the dictionaries for C.2 (i.e. the top x words, where x is the number of zero-frequency characters for that log file. In the corpus, $158 \leq x \leq 161$)

	Num	Not Incl
mar2006	32	-
apr2006	31	31
may2006	32	-
jun2006	38	31
jul2006	38	-
aug2006	31	05
sep2006	29	24,30,31
oct2006	32	-

	Num	Not Incl
nov2006	32	31
dec2007	30	30,31
jan2007	32	-
feb2007	30	29,30,31
mar2007	32	-
apr2007	29	31
may2007	26	24,26-31

Table 6.7: Frequency of two digit numbers between 00 and 60 included in the dictionary and numbers between 00 and 31 which are not included

Table 6.7 shows the frequency of two digit numbers between 00 and 60 which are included in the relevant dictionaries. It also shows the numbers between 00 and 31 which are not included in

the relevant dictionaries. When the average rank of the numbers is taken, the numbers from 00 to 31 are included in the top 160 words. This is expected, since the number between 00 and 23 are hours and minutes and the numbers between 10 and 31 are dates. It is for this reason that in the dictionaries for February, the numbers between 29 and 31 are not included the dictionary. 31 is also not included in dictionaries for April, June, September and November which are 30 day months.

In summary, the dictionaries essentially contain a number of different elements:

1. Numbers from Timestamps and Months
2. Process names for frequent processes
3. Words which occur often within log messages
4. Hostnames and IP addresses from the network (including hostname which is performing the logging) and common domains involved in mail exchange

The elements which are included can vary due to the network structure, the email domains which are used often and the version of software which are used. Applications changes therefore warrant a review of the dictionary being used. TLS needs to be included into the dictionary. This section has shown that there are, however, a number of elements which occur in all the log files presented. Using the knowledge of the network and the knowledge of the semantics a dictionary which contains the four elements discussed above can be developed.

Building a dictionary

When building a dictionary, the four elements mentioned in the previous section need to be included.

The first element which needs to be included is the numbers from the timestamps and the twelve months. In the dictionaries shown in the previous sections, the months were included as well as a number of two digit numbers. The previous section showed that the top 30 numbers should be included into the dictionary. The twelve months (Jan, Feb, Mar, ...) and the numbers from 00 to 30 are therefore included into the dictionary.

The second element which needs to be included is the process names for frequent processes. In the syslog format, the processes which are logging are included in every log line. It is hence important to include the process names within the dictionary. Within the entire corpus, there are 40

processes which occur. `postfix/smtpd`, `postfix/qmgr`, `postfix/smtp` and `postfix/cleanup` generate 90.04 percent of the log lines. The question is how to decide which processes need to be included in the dictionary.

Word	Num	Avg Pos	All	CombNum
postfix	15	2	2	2
smtpd	15	7.87	7	18
qmgr	15	19.67	18	27
smtp	15	33.53	30	44
cleanup	15	23.87	21	31
amavis	15	52.13	47	55
sqlgrey	13	106.38	111	111
anvil	14	177.43	210	179
error	2	136.5	187	-
virtual	15	168.8	158	172
master	3	108	-	-
scache	5	241.8	-	-
verify	3	127.67	-	-
dccproc	1	169	-	-
clamd	1	239	-	-
local	5	151.8	-	-

Table 6.8: Ranking of words relating to process names in previous dictionaries

Table 6.8 shows the ranking of the words which relate to process information in the dictionaries for the preprocessors presented so far. The average position (Avg Pos) is calculated by taking the average of the ranking in the the top 255 entries for words in the log files based on WSC (C,2). The column Num shows the number of log files for which the word appears in the top 255 entries. In the dictionary generation process, the "/" splits the words since it is not included in the word definition i.e. `postfix/qmgr` becomes two words - `postfix` and `qmgr`. They are hence included as separate words in the dictionary.

`postfix/smtpd`, `postfix/qmgr`, `postfix/smtp`, `postfix/cleanup`, `amavis` and `sqlgrey`: all occur in most of the dictionaries presented. These six processes generate 96.58 percent of the messages and individually produce at least 1.58 percent of the log messages. These processes

are therefore included in the dictionary as words. `postfix` is also included as a word since there are 27 different processes which begin with `postfix`.

The third element which needs to be included into the dictionary is words which occur often within log messages, These are items which relate to the semantics present in the file. The initial semantic investigation, presented in Section 4.2, showed a number of words which need are observe in the maillog corpus. These words are observed in the dictionaries for the various log files. This section showed the words which can be observed for the five main processes: `postfix/smtpd`, `postfix/smtp`, `postfix/qmgr`, `postfix/cleanup` and `amavis`.

The following words are observed for these processes from this section (if a word is found that has already been listed for another process, it is omitted from the list for that process):

postfix/smtpd connect, from, disconnect, client, NOQUEUE:, reject:, RCPT, Recipient, address, rejected, to, proto, helo, TLSv1, TLS, connection, cipher, established, ADH-AES256-SHA, (256/256, not, found

postfix/smtp relay, delay, delays, dsn, status, certificate, verification, for, refused, Operation, conn_use, verify

postfix/qmgr queue, active, nrcpt, size, delivery, temporarily, removed, with

amavis CLEAN, Passed, Message-ID,mail_id, Hits, queued_as

postfix/cleanup message-id

These words are included in the dictionary. Word such as "`statistics`" which occurs often in the `postfix/anvil` process are also included in the dictionary.

There are a number of other words which occur in messages which are replaced during the semantic investigation. Word such as "`sent`" and "`error`", occur in a number the messages, such as "`sent more than once`" and "`delivery error`", while other words such as "`hostname`" and "`deferred`" also occur in a number of messages such as "`fully-qualified hostname`", "`status=deferred`" and "`Message temporarily deferred`". The word "`Ok`" occurs as part of a status message e.g. "`250 2.5.0 Ok`", while "`queued`" occurs in at the end of messages in the form "`queued as`". Words such as "`completed`" also occur in messages such as "`Requested mail action okay, completed.`" The word "`ESMTP`" occurs for 624 178 out of the 846 239 values for the field "`proto`" (i.e. "`proto="ESMTP"`"occurs in the message). In both the `amavis`

process and the `postfix/smtp` process, words such as "`MTA`" can also be found in messages. These words are also included in the dictionary.

The messages which are output include codes such as `450`, `250`, etc. These were originally specified in the RFC 821 [141] (this RFC was made obsolete by RFC2821 [142] which also specifies these codes). The two most common codes are `250` and `450` which represent "requested mail action okay" and "requested mail action not taken" respectively. These codes are included in the dictionary. RFC 1893 [143] (this RFC was made obsolete by RFC3463 [144] which also defines extended error codes) and RFC 2034 [145] define extended error codes of the form X.X.X, where the first X is either 2, 4 or 5 which represent "successfully sent", "temporary problem" and "permanent/fatal error" respectively. More information can be found in the RFCs. The most common delivery status notification (dsn) codes observed in the maillog corpus are `2.0.0`, `2.5.0` and `2.5.0`. These are therefore included in the dictionary.

The ports 10024 and 10025 are also replaced in the `postfix/smtp` process. Messages such as "`Connection refused (port 10024)`" are generated by the `postfix/smtp` process, while messages such as "`from MTA([127.0.0.1]:10025)`". Due to the frequency of these ports, these numbers are included as words in the dictionary.

The fourth element which needs to be included into the dictionary is the hostnames and IP addresses which are involved in the network. The first hostname which needs to be included is the hostname which is performing the logging (in this case `titania`). In some cases there may be more than one. If this is the case, then an analysis needs to be performed on previous log data to determine the frequency which each host logs to the file and therefore determine which hostnames need to be included into the dictionary. The hostnames are often observed in close proximity to their IP addresses within the log files.

IP	Hostnames
127.0.0.1	localhost, localhost.domain
146.231.128.21	elephant.ru.ac.za, a.mx.ru.ac.za
196.23.167.21	mercury.sac.ecape.school.za
196.23.167.26	mail.kc.ecape.school.za
196.23.167.44	vicky.vghs.ecape.school.za
209.67.212.202	gauntlet.mithral.co.za

Table 6.9: IP addresses and hostnames involved in the network

Table 6.9 shows the various IP addresses and their hostnames which are involved in the network.

All of these hostnames also occur in the dictionary. 10 903 188 out of 21 049 570 lines in the corpus (51.80 percent) contain these IP addresses.

hostname[IP]	Number	IP occurring	%
elephant.ru.ac.za[146.231.128.21]	1426	42029	3.39
a.mx.ru.ac.za[146.231.128.21]	38675	42029	92.02
mercury.sac.ecape.school.za[196.23.167.21]	1127902	1262917	89.31
mail.kc.ecape.school.za[196.23.167.26]	1307283	1551976	84.23
vicky.vghs.ecape.school.za[196.23.167.44]	132135	144813	91.25
gauntlet.mithral.co.za[209.67.212.202]	2452746	313160	76.81

Table 6.10: Lines in the corpus containing the format hostname [IP]

Table 6.10 shows the number of lines which contain the format *hostname[IP]*. This table shows that a large percentage of the times the IP address occurs, it occurs in the form *hostname[IP]*. Therefore these are included into the dictionary. The hostnames and IP addresses are also included into the dictionary since replacing them still achieves a large gain due to the length of the strings. The parent domains for each hostname are included up to the third level if they occur in the corpus i.e. for `mercury.sac.ecape.school.za`: `sac.ecape.school.za` and `ecape.school.za` are included in the dictionary. When developing a dictionary, a sample of previous log data can be inspected to determine which hostnames, email domains occur. Servers may be on the network which do not generate a significant amount of traffic and hence are not beneficial to the dictionary.

In this network, there are a number of email domains which are used. These include: `dsg.ecape.school.za`, `gc.ecape.school.za`, `kc.ecape.school.za`, `saprep.ecape.school.za`, `sac.ecape.school.za`, `vghs.ecape.school.za`, `sacschool.com`, `dsgschool.com` and `saprepschool.com`. These need to be included in the dictionary since they often occur. From observation, unknown also occurs as a hostname in many of the previous dictionaries and is hence included in the dictionary.

There are also other mail domains which are commonly involved in email exchanges. `gmail.com`, `yahoo.com` and `hotmail.com` are commonly used and hence need to be included in the dictionary. The high utilisation of social networking sites such as facebook.com means that the associated mail domains need to be included. `hs.facebook.com` and `facebook.com` are therefore included into the dictionary. The hostname `spl-xbl.spamhaus.org` occurs in the `amavis`, `postfix/smtp` and `postfix/smtpd` processes. This hostname is a source of information on

SPAM which needs to be blocked. This hostname occurs often and hence also needs to be included into the dictionary.

The dictionary containing these four elements is referred as MyCustD, while the preprocessor which uses this dictionary is referred to as MyCust.

Improving the dictionary

MyCustD can be improved by looking at the context of the words which occur and adding characters based on these contexts. The processes, for example, are all followed by the PID, which is contained within square brackets. The opening square bracket can, therefore, be appended to the process name. e.g. "`postfix/smtpd`" becomes "`postfix/smtpd[`"

The word "`status`" appears often in messages for the `postfix/smtp` process. This often appears as a field for which a value can be specified.

	Number of Lines	%
status	44796	1.37
status=bounced	24799	0.76
status=deferred	578426	17.63
status=deliverable	667	0.02
status=expired	1588	0.05
status=sent	2629913	80.16
status=undeliverable	583	0.02

Table 6.11: Number of lines containing a given value for status

Table 6.11 shows the different values for assigned to word "`status`". This table shows that 97.79 percent of the occurances are either `status=sent` or `status=deferred`. The words sent and deferred also occur in MyCustD. "`status=sent`" accounts for 98.63 percent of the occurrences of the word "`sent`". The reduction of the dictionary on the maillog corpus can be improved by 10 705 019 bytes by including both "`status=sent`" and "`status=deferred`" instead of "`status`" and "`sent`". This improvement is greater than the improvement caused by any of the words with a rank greater than 27.

Within the maillog corpus, there are 1 078 445 occurrences of the word "`MTA`".

	Number	%
MTA([127.0.0.1]:10025),	191	0.018
MTA([127.0.0.1]:10025):	1078232	99.98
MTA	22	0.002

Table 6.12: Occurrences of the word "MTA"

Table 6.12 shows the breakdown of the number occurrences of the word "`MTA`" within the mailog corpus. There are 7 380 845 occurrences of `127.0.0.1` and 1 078 590 occurrences of `10025` within the maillog corpus. By replacing "`MTA`" in the dictionary with "`MTA([127.0.0.1]:10025):`", the reduction by the dictionary on the corpus is improved by 7 545 168 bytes.

There are 1 158 090 occurrences of the word "`10024`" within the maillog corpus. 1 155 349 (99.76 percent) of these occurrences occur in the string "`relay=127.0.0.1[127.0.0.1]:10024,`". The string "`relay=127.0.0.1[127.0.0.1]`" occurs 1 477 237 times in the maillog corpus. The word "`10024`" can be replaced by "`:10024`", while the word "`relay=127.0.0.1[127.0.0.1]`" can be added to the dictionary. There are still 3 991 915 occurrences of the word "`127.0.0.1`" and 3 235 679 occurrences of the word "`relay`" in the maillog corpus. This means that it will still beneficial for these two words to be included in the dictionary.

The results for the initial semantic investigation (contained in Appendix C) showed that the words "`mail_id`", "`queued_as`", "`Message-ID`", "`Hits`" and "`statistics`" are followed by a "`:`", while the words "`delays`" and "`conn_use`" are followed by an "`=`". These characters are therefore appended to these words resulting in an extra character being removed each time these words are replaced.

An improved dictionary is the result of these changes to MyCustD. This improved dictionary is referred to as MyCust2D, while the preprocessor which uses this dictionary is referred to as MyCust2.

Adding common phrases

The previous dictionaries have just contained words from the maillog file. The initial semantic investigation in Section 4.2 showed that within the maillog files there are also a number of commonly occurring phrases. By replacing an entire phrase with a single character, a larger reduction is achieved than replacing a single word. Section 4.2 also showed that there are a number of cases where two words often occur consecutively. By replacing a phrase containing these two

words with a single character, rather than replacing each word with a single characters, a larger reduction is achieved and an extra character is available to be used for replacement. This section investigates expanding MyCust2D to include commonly occurring phrases.

A number of commonly occurring phrases were observed in Section 4.2. These included: "Recipient address rejected:"; "Connection refused"; "Operation timed out"; "delivery temporarily suspended: connect to:"; "queue active"; and "certificate verification failed for".

Within the syslog file, the process names occur in the file after the hostname which is performing the logging. In these log files, "titania" is the only hostname. titania can therefore be included before the process names i.e. the word "postfix/qgmr[" can be replaced by the phrase "titania postfix/qmgr[" in the dictionary.

Section 4.2 shows that a number of log lines are generated based on the TLS connections which are established. In MyCust2D, there are eight different words included for these TLS connections. gauntlet.mithral.co.za uses a TLS connection using the cipher ASD-AES256-SHA (256/256 bit).

This results in the following two phrases:

- setting up TLS connection from gauntlet.mithral.co.za[209.67.212.202]
- TLS connection established from gauntlet.mithral.co.za[209.67.212.202]: TLSv1 with cipher ADH-AES256-SHA (256/256 bits)

MyCust2D contained the following words: "(256/256", "TLS", "TLSv1", "connection", "established", "cipher", "setting" and "ADS-AES256-SHA". These two phrases can be included into the dictionary instead of these eight words. This results in six extra characters which can be used for other words or phrases.

The word "status=sent" is often followed by the message "(250 Requested mail action okay, completed.)". The following phrase can therefore be added to the dictionary in place of the word "status=sent":

- status=sent (250 Requested mail action okay, completed.)

The words "Passed" and "CLEAN" are included in MyCust2D since they occur in messages for the amavis process. These words occur successively and therefore can be concatenated into a single phrase "Passed CLEAN" and included in the dictionary.

The words "disconnect" and "connect" are included in the dictionary since the occur for postfix/smtpd. For all these occurrences, the string occurs after the PID and is followed by the word "from". These words can therefore be replaced with the phrases "]: disconnect from" and "]: connect from:" respectively. The word "connect" occurs in a number of other places, so it is beneficial for it to remain in the dictionary.

The word "statistics:" is included in MYCUST2D since it is included in every message for the postfix/anvil process. 66.7 percent of these messages contain the phrase "statistics: max connection". This phrase can therefore replace the word "statistics:" in the dictionary.

By making these modifications to MYCUST2D and adding the phrases discussed, a new dictionary which is referred to as MYPHRASED is formed. The preprocessor which uses this dictionary is referred to as MYPHRASE.

6.3.2 Results and Analysis

Table 6.13 shows the compression ratios achieved by MYCUST, MYCUST2 and MYPHRASE on the maillog corpus. It also shows the compression ratios achieved by C.2, COMBNUM and PMU. As mentioned, COMBNUM achieves a lower compression ratio than PMU, however its compression ratio is higher than the compression ratio achieved by C.2. MYCUST achieves a lower compression ratio than COMBNUM for all files in the corpus. It also achieves a lower compression ratio than C.2 for may2006, jan2007, feb2007 and mar2007. On average, however it achieves a higher compression ratio than C.2 (0.77 percent higher). As expected, MYCUST2 achieves a lower compression ratio than MYCUST for all files in the corpus. The average compression ratio achieved by MYCUST2 is also lower than the average compression ratio achieved by C.2 (2.03 percent lower). For two files (mar2006 and may2007) in the corpus, however, MYCUST2 achieves a higher compression ratio than C.2. As expected, MYPHRASE achieves a lower compression ratio than MYCUST2 for all files in the corpus. The use of phrases results in a 5.07 percent lower compression ratio. It also achieves lower compression ratios than C.2 for all files in the corpus. On average, it shows a 7 percent lower compression ratio than C.2. MYPHRASE uses a single dictionary which can be used on all the files in the corpus and is built based on the semantics contained in the log files and the knowledge of the network structure. This dictionary can be integrated into the logging system. The compression ratio achieved by MYPHRASE is 11.93 percent lower than PMU and 10.25 percent lower than COMBNUM.

File	C.2	PMU	CombNum	MyCust	MyCust2	MyPhrase
mar2006	0.54553	-	0.56992	0.55935	0.55066	0.52477
apr2006	0.54934	0.57913	0.57041	0.56112	0.54765	0.51850
may2006	0.54868	0.58743	0.55646	0.54464	0.53071	0.50204
jun2006	0.53226	0.56199	0.54786	0.53292	0.51758	0.48899
jul2006	0.52342	0.55027	0.54650	0.53046	0.51506	0.48672
aug2006	0.54116	0.56890	0.55922	0.54998	0.53456	0.50153
sep2006	0.54924	0.57105	0.56081	0.55049	0.53655	0.50809
oct2006	0.55035	0.57307	0.56023	0.55150	0.53716	0.50907
nov2006	0.54720	0.56968	0.55851	0.54780	0.53206	0.50504
dec2006	0.55621	0.57995	0.56641	0.56297	0.54644	0.51468
jan2007	0.55434	0.56936	0.56238	0.55341	0.53763	0.50970
feb2007	0.55380	0.57084	0.55714	0.54531	0.52929	0.50340
mar2007	0.53982	0.57725	0.55986	0.53383	0.51687	0.49839
apr2007	0.52794	0.56084	0.56079	0.52859	0.51289	0.49592
may2007	0.52722	0.60896	0.60536	0.55715	0.53587	0.50936
Average	0.54310	0.57348	0.56279	0.54730	0.53207	0.50508

Table 6.13: Compression Ratios achieved by MyCust, MyCust2 and MyPhrase on the maillog corpus

MyPhrase achieves the lowest compression ratio out of all the preprocessors tested so far. The inclusion of these phrases, however does increase the ratio of binary characters and also reduces the redundancy, therefore increasing the payload ratio achieved. The average payload ratio achieved on the corpus is increased by 0.00928 (4.45 percent). The overall improvement is, however, greater for MyPhrase than MyCust2.

Table 6.14 shows the average improvement in compression ratio achieved for the various programs when using MyCust, MyCust2 and MyPhrase. PMU and CombNum show higher improvements for `7zip` than C.2, but lower improvements than C.2 for all the other programs. MyCust2 and MyPhrase achieve higher amounts of improvement than C.2 for all compression programs except `ppmd`. MyPhrase achieves the greatest amount of improvement for all compression programs except `ppmd`. The average improvement in compression ratio for the seven compression programs is 0.01590 for MyCust, 0.01678 for MyCust2 and 0.01765 for MyPhrase. MyPhrase results in an improvement of 5.18 percent over MyCust2, while

	C.2	PMU	CombNum	MyCust	MyCust2	MyPhrase
7zip	0.00615	0.00819	0.00834	0.00883	0.00947	0.00977
arj	0.01969	0.01897	0.01880	0.01960	0.02045	0.02166
bzip2	0.00767	0.00739	0.00731	0.00808	0.00829	0.00854
gzip	0.01784	0.01719	0.01707	0.01779	0.01874	0.02001
lzop	0.03412	0.03303	0.03260	0.03381	0.03663	0.03928
ppmd	0.00618	0.00608	0.00594	0.00542	0.00512	0.00425
zip	0.01784	0.01719	0.01707	0.01779	0.01874	0.02001
lzop(1)	0.03949	0.03814	0.03763	0.03878	0.04253	0.04593
lzop(2)	0.02337	0.02282	0.02255	0.02386	0.02482	0.02598
ppmd(1-7)	0.01890	0.01862	0.01841	0.01770	0.01696	0.01565
ppmd(8-15)	-0.00445	-0.00441	-0.00447	-0.00490	-0.00500	-0.00545

Table 6.14: Average improvement in compression ratio achieved for compression programs when using MYCUST, MYCUST2 and MYPHRASE

MYCUST2 results in an improvement of 5.5 percent over MYCUST. `ppmd` is the only program for which MYCUST2 achieves a higher amount of improvement in compression ratio than MYPHRASE2. MYCUST achieves the highest amount of improvement for `ppmd` out of the three preprocessors. The improvement in compression ratio for MYPHRASE is 16.99 percent higher than improvement for MYCUST2, while the improvement for MYCUST2 is 5.99 percent higher than the improvement by MYCUST.

6.3.3 Summary

This section used the semantic knowledge present in a log file and the knowledge of the network to develop a dictionary which achieves the lowest compression ratio on all the files in the maillog corpus. The analysis of the contents of the previous dictionaries revealed that all the dictionaries essentially contained four different elements.

1. Numbers from Timestamps and Months
2. Process names for frequent processes
3. Words which occur often within log messages
4. Hostnames and IP addresses from the network (including hostname which is performing the logging) and common domains involved in mail exchange

The aim of this section was to develop a dictionary based on the knowledge of the semantics and the network structure which achieves a lower compression ratio than C.2 for all files in the maillog corpus. This dictionary was developed using an iterative process based on the spiral model for software development defined by presented by Boehm [140]. This process consisted of three different prototypes. The first prototype was formed by using the results from analysis of the maillog files presented in Section 4.2 and the knowledge of a network structure to form a dictionary. The second prototype was formed by updating the words in the dictionary based on an analysis of the contexts in which these word appear. The final prototype was formed by adding phrases which improve the compression ratio achieved. The objective for the second prototype was for a preprocessor based on that dictionary to achieve a lower compression ratio than C.2, while the objective of the final prototype was to include phrases which improve the compression ratio achieved by a preprocessor.

This section showed that these objectives were achieved. The preprocessor based on the initial dictionary, MYCUST achieved a lower compression ratio than COMBNUM and also achieved a lower compression ratio than C.2 for four maillog files. The preprocessor which used the dictionary formed as the second prototype, MYCUST2, achieved a lower compression ratio than MYCUST and achieves a lower compression ratio on thirteen out of the fifteen files in the maillog corpus resulting in an average compression ratio which is 2.03 percent lower than C.2. The preprocessor which used the dictionary from the final prototype, MYPHRASE, achieved the lowest compression ratio out of all the preprocessors presented. It achieved a 5.07 percent improvement over the compression ratio achieved by MYCUST2.

When used with compression programs, MYCUST2 achieved a higher average improvement in compression ratio than C.2 for all compression programs except `ppmd`, while MYPHRASE achieved a higher average improvement in compression ratio than MYCUST2 for all compression programs except `ppmd`. Out of the three prototypes, MYCUST achieved the highest average improvement in compression ratio for `ppmd`. The improvement achieved by MYCUST, however was still lower than the improvement achieved by C.2. These preprocessors did not cause an improvement for C.2 since the compression ratios were increased due to the modified distribution of characters caused by the word-replacement. On average, MYCUST2 achieved a 5.5 percent higher improvement than MYCUST, while MYPHRASE achieved a 5.18 percent higher improvement than MYCUST2 when used with compression programs.

MYPHRASE can be easily integrated into the logging process since its dictionary does not depend on the log file which it is being, but rather on the known semantics and the network structure. It achieved the lowest compression ratio on all the files in the corpus and is based on known se-

mantics and the knowledge of the network structure involved. It is also a single dictionary which means that multiple dictionaries do not need to be kept in order to not lose important information. The next section discusses the different scenarios and which combination of preprocessors and compression programs would be best suited to the requirements.

6.4 Scenarios

The use of preprocessors generally improves the compression ratio, compression time, decompression time and transfer time achieved by the compression programs. Chapter 5 showed that C.2 achieved the greatest improvement on average, however this chapter has showed that MyPhrase achieves the greatest improvement for all compression programs except `ppmd`. C.2 cannot be integrated into a logging system and requires the log file to be present. C.2 also requires the log files to be parsed for the dictionary to be created. The time taken by the preprocessor negates the improvement in compression time. MyPhrase is therefore a better choice for all scenarios where the best compression program is not `ppmd`.

Section 3.3 showed that for each monitoring scenario a different compression program is more appropriate. These scenarios included:

- Filtering logs through to the central point for analysis (where latency is not important, but there is a desire to use as little bandwidth as possible)
- Real-time monitoring (where a minimum point-to-point time is desired, using as little resources as possible)
- Quick access, storing it compressed and later decompressing it for analysis (where fast decompression is desired, but the compression time does not matter. In addition, there is a desire to use as little bandwidth as possible)
- Low system-time-usage for compression and decompression (where fast compression and decompression times are desired and the compression ratio does not matter)

This section investigates these scenarios and discusses which compression program - preprocessor combinations would be more effective in these scenarios.

6.4.1 Filtering logs through to the central point for analysis

In this scenario, the compression and decompression times are not important, however a low compression ratio is desired. Section 3.3 shows that `ppmd`, `7zip` and `bzip2` are the best options for this scenario and concludes that `7zip` is the best option due to its low compression ratio, fast decompression times and versatility. `ppmd` achieves the lowest compression ratio out of the compression programs, however the use of preprocessors does not lead to an improvement in the compression ratio achieved. Section 5.3.1 shows that the use of preprocessors does not improve the lowest compression ratio achieved by `ppmd` (This is achieved by compression level nine). Since the logs are going to be used for analysis at a central point, a fast decompression time is desired. `ppmd` achieves the slowest decompression time out of all the compression program which means that it is not the best choice for this scenario.

`bzip2` achieves the highest compression ratio out of these three programs, however it also achieves the highest amount of improvement in compression and decompression time. Even with this improvement, however, `bzip2` achieves higher decompression times and higher compression ratios than `7zip`. This means that `7zip` is a better choice than `bzip2` for this scenario.

MyPhrase achieves the highest amount of improvement in compression ratio for `7zip`. It also achieves the lowest compression ratio out of the preprocessors which means that it achieves a higher amount of improvement in compression and decompression time. With the use of preprocessors, `7zip` is also the only compression program which achieves a decrease in the amount of memory utilisation. `7zip` and MyPhrase are therefore the best choices in this scenario.

6.4.2 Real-time monitoring

For real time monitoring, the lowest possible transfer time between the two points is desired. The use of preprocessors provides an improvement in the transfer time. Section 3.3 shows that best compression program depends on the transfer speed between the two points. For high transfer speeds, `lzop(1)` achieves the fastest transfer time, while for low transfer speeds, `ppmd` achieves the fastest transfer time. For low transfer speeds, the transfer time for `ppmd` is not improved with the use of preprocessors. Section 5.3.3 shows that using C.2 does improve the transfer time achieved by `ppmd`, however it cannot be used in a real time situation since it requires the entire log file to be present before compression. Section 6.3 shows that C.2 achieves a greater amount of improvement in compression ratio than MyPhrase. Section 5.3.3 shows that at the low transfer speeds in which `ppmd` achieves that fastest transfer time, the improvement in compression ratio

has a larger weighting that the improvement in total time, which means that C.2 achieves a higher improvement in transfer time than MyPhrase. Therefore the best choice is to use `ppmd` with no preprocessor for transfer speeds below 263 KBps, `zip` and MyPhrase for transfer speeds between 263 KBps and 4874 KBps, and `lzop(1)` with MyPhrase for higher transfer speeds. For speeds above 150 Mbps, no compression and MyPhrase should be used.

6.4.3 Quick access to stored compressed information

In this scenario, a fast decompression time is desired. Since the data is stored, a low compression ratio is also desired, however this is not the primary criterion in this scenario. Section 3.3 concludes that since the compression time does not matter, `7zip` is the best option due to its low compression ratio. `lzop`, `gzip`, `arj`, and `zip` all achieve faster decompression times than `7zip`, however they also achieve higher compression ratios than `7zip`. The use of a preprocessor means that in order for the data to be accessed, it needs to be transformed. An advantage of using the preprocessor, however, is that the decompressed file effectively has an index (the dictionary) which can be used to improve the time to find information within the log file. If the entire file has to be accessed, then the advantage is negated due to the time required to perform the reverse transformation. In this case a preprocessor should not be used. `gzip` and `zip` are both better options in this scenario than `arj` since `arj` achieves slower compression times, decompression times and higher compression ratios than both `zip` and `gzip`. `lzop` achieves faster total times than `gzip` and `zip`, however it also achieves much higher compression ratios than `gzip` and `zip`. `zip` achieves the fastest decompression time and a slightly higher compression ratio than `gzip`, which means it is the better choice out of the two for this scenario. The decompression time for `zip` is one third of the time for `7zip`, and it requires less resources for compression and decompression than `7zip`. `zip` and `7zip` using MyPhrase would be the two best choices for this scenario. Since `zip` achieves lower decompression times, `zip` and MyPhrase is the best choice for this scenario. In the case where the entire file needs to be accessed, then MyPhrase should not be used.

6.4.4 Low system time usage for compression and decompression

In this scenario, low compression and decompression times are desired. Section 3.3 concludes that `lzop` would be the best choice. With the use of preprocessors, `lzop` still achieves the lowest total time. For quick compression and decompression without taking any regard to size, `lzop`

would therefore undoubtedly be the best choice. Both `zip` and `gzip` also achieve fast compression and decompression times. The also achieve lower compression ratios. Since this scenario is primarily concerned with the system time, `lzop` and MYPHRASE would be the best choices for this scenario. MYPHRASE can be integrated into the logging system and hence be transparent. If it is not integrated into the logging system then the preprocessor should not be used. In this case, `lzop` and no preprocessor would be the best choices for this scenario.

6.5 Summary

The previous chapter showed that by creating a dictionary based on the frequency and length of the words contained in the file and using it to perform word-based replacement, a significant improvement in the performance of the compression program can be obtained. The problem with this methodology is that the entire file needs to be parsed in order to create the dictionary. The time to generate the dictionary and transform the file is significantly greater than the improvement in time with the use of the preprocessor. This time can be made transparent by integrating the preprocessor into the logging system. In order to integrate the preprocessors into the logging system, however the dictionary needs to exist before the file is logged. This is not the case with the dictionaries presented in the previous chapter.

In this chapter, the use of the previous month's dictionary and a single dictionary was explored. Section 6.1 showed that using the previous month's dictionary does not result in a greater improvement than C.2. Using the previous month's dictionary, however, still resulted in a separate dictionary for each month. Section 6.2 combined the dictionaries for the files in the corpus into a single dictionary which achieved a lower compression ratio than the past months when used by a preprocessor. A single dictionary which can be formed using the knowledge of the network and the semantics involved in the log file is desired. Section 6.3 explained how such a dictionary can be constructed. When used by a preprocessor, the resulting dictionary achieved a lower compression ratio than C.2 on all the files in the corpus.

This chapter has therefore shown that the semantics of the log file and knowledge of the network can be used to create a dictionary which achieves a lower compression ratio than C.2. It has also discussed which compression programs and preprocessors are most effective in the four different monitoring scenarios.

Chapter 7

Conclusion

This thesis set out to determine how effective data compression is as a means of data reduction and evaluate how the use of semantic knowledge can improve data compression by creating text preprocessors which transform the file into a representation which results in lower overall compression ratio on the file. This chapter presents a summary of the findings of this thesis. The first section presents a summary of the work, the second section shows how the research questions identified in Section 1.2.1 have been answered, the third section presents a reflection on the work done and the final section presents future work which can be done based on the work presented in this thesis.

7.1 Summary of work

Chapter 2 introduced the compression algorithms used by the different compression programs. It also showed that word-based techniques can be used to improve the performance of standard compression programs. At the time of writing, a detailed analysis of the performance of compression programs on log file had not been performed.

Chapter 3 presented an analysis of the compression ratios, compression times, decompression time and memory utilisation for standard compression programs on a corpus of log files. This chapter showed that compression provides an improvement in point-to-point times for transfer speeds of up to 207 Mbps and results in a reduction of up to 98 percent in filesize. This chapter also selected compression programs for each four different scenarios based on these results. Statistical compressors achieved lower compression ratios, while sequential compressors achieved faster times and used less memory.

Chapter 4 went on to investigate the improvement in compression ratio and time achieved by standard compression programs when combined with preprocessors which replace the timestamps and IP addresses with their binary equivalents. It showed that this technique resulted in an improvement in the compression ratio and times achieved by the compression programs. This chapter also presented an investigation into the semantic knowledge contained in a corpus of 15 months of maillog files. This analysis showed that there are a number of words which occur in the file which could be replaced with tokens.

Chapter 5 investigated the improvements which result when using word-replacement with a dictionary built based on a given word definition and word score. This resulted in a reduction in the size of the corpus of up to 45.69 percent and an improvement in compression ratio and total times for standard compression programs of up to 14.82 percent and 43.59 percent respectively.

The problem with the methodology used in Chapter 5 was that the entire file needed to be parsed in order to create the dictionary. The time to generate the dictionary and transform the file was also significantly greater than the improvement in time when using the preprocessors. Chapter 6 investigated the results when building a dictionary based on the previous month's data. Chapter 6 also discussed the development of a single dictionary based on the known semantics present in the log file. When used by a preprocessor, the resulting dictionary achieved the lowest compression ratio out of all preprocessors presented.

7.2 Answering Research Questions

Section 1.2.1 identified four research questions which this thesis set out to answer. This section shows how and where these questions have been answered within this thesis and details the conclusions that were drawn.

7.2.1 How effective is compression in reducing the size of the log file and how much resources (time and memory) are used by the compression programs?

Data compression programs reduce the size of text files by between 80 and 90 percent. The maximum compression benchmarking site shows between 91.82 percent and 97.18 reduction on a 20MB web traffic log file using the compression programs evaluated in this thesis [49].

Chapter 3 showed that compression is effective in reducing the size of the log files. It presented tests conducted on a corpus of log files consisting of six text log files (a generic syslog, a squid access log, an apache access log, a postfix mail log, a kernel messages log and a ftp log) and a binary LibPCap data file. The results showed average reductions in the size of the maillog corpus of up to 94.28 percent with the statistical compressors (`ppmd`, `7zip` and `bzip2`) achieving lower compression ratios than the LZ77-based compressors (`zip`, `gzip` and `arj`) and `lzop`. These results were detailed in Section 3.2.1. The lowest time for compressing the corpus was between 0.76 seconds and 72.24 seconds across the compression programs, while the time for decompressing the corpus was between 1.06 seconds and 23.09 seconds across the compression programs (These results are detailed in Section 3.2.2). These programs showed a maximum memory utilisation of between 0.9 MB and 578.09 MB of memory for compression and between 0.8 MB and 116.83 MB of memory for decompression (These results are detailed in Section 3.2.4). The statistical compressors required a greater amount of time and memory than the LZ77-based compressors and `lzop`.

The transfer time for a particular compression program is the time taken to transfer the file at a given transfer speed from one point to another with the use of compression and decompression at each end. This is calculated by taking the time to transfer the compressed payload at a given transfer speed and then adding it to the time taken to compress and decompress the original payload. Section 3.2.3 showed that for all transfer speeds up to 207.55 Mbps, there exists a compression program which produces a faster transfer time than the time taken to transfer the file without compression. All the compression programs showed an improvement for transfer speeds less than 16.90 Mbps.

7.2.2 What sort of semantic knowledge exists within the log files?

Analysis of log files revealed that they contain verbose timestamps and IP addresses which can be represented in fewer characters by using binary characters. Apart from the timestamp and IP addresses, however, the log files also contained many other frequent patterns. Section 4.2 presented an analysis of Postfix maillogs which use the syslog format. This section revealed that there are a number of common words and phrases contained within a maillog file. Based on this result, preprocessors which use word replacement were developed. For these preprocessors, dictionaries were constructed for each file in the maillog corpus based on a given word definition and word score. This process was described in Section 5.1. Section 6.3.1 performed an analysis of these dictionaries and a dictionary produced for the entire corpus. The analysis revealed that

the following elements make up a dictionary:

1. Numbers from Timestamps and Months
2. Process names for frequent processes
3. Words which occur often within log messages
4. Hostnames and IP addresses from the network (including hostname which is performing the logging) and common domains involved in mail exchange

These elements essentially summarise the semantic knowledge which can be exploited. Each process showed a set of messages which it outputs with a few variable elements (such as hostnames and IP addresses). Among the variable elements, common values were also observed. Section 6.3.1 showed that these common values can be determined using the knowledge of the network structure. It also showed that some of words which occur within dictionaries are the words which were seen in the analysis presented in Section 4.2.

7.2.3 Can this semantic knowledge be exploited to improve data compression?

Standard compression programs were not created to compress specific types of files, but rather to use characteristics such as recently repeated sequences and the distribution of the characters to compress files. Log files are rather verbose and have a set structure of outputting messages.

Section 4.2.3 showed that within the log files, there are a number of properties which can be exploited by a preprocessor. These included:

1. Text log files contain many unused characters (zero-frequency characters)
2. Timestamps and IP addresses take up alot more space than they should
3. Certain hostnames and IP addresses occur many times within a log file
4. There are a number of other words and phrases which occur often to describe values which vary

This thesis has presented a number of different preprocessors which have exploited these properties.

Section 4.1 showed that a simple preprocessor which replaces IP address and timestamps with binary equivalents resulted in an improvement in compression ratio for all programs except `bzip2`. These preprocessors reduced the file size by between 9 and 17 percent. The payload ratio on the processed files was higher than the compression ratio on the original file, however the reduction in the size of the file by the preprocessor caused an overall improvement in compression ratio. This was the case for all the preprocessors presented. The average improvement in compression ratio was 2.88 percent for T (which replaced the timestamps with their binary equivalents), while the average improvement in compression ratio was 1.64 percent for T&IP (which replaced the timestamps and IP addresses with their binary equivalents). The use of preprocessing also resulted in an improvement in the total time. T&IP showed an improvement in total time of up to 27.9 percent, while T showed an improvement in total time of up to 22.52 percent.

Section 4.2 identified that there was a number of common words and phrases present within the log files. Section 5.1 presented a methodology for developing six preprocessors which perform word replacement using dictionaries based different word definitions and word scores (These were described in Section 5.1.1 and Section 5.1.2 respectively). These preprocessors reduced the size of a corpus of maillogs by up to 45.69 percent. Out of these six preprocessors, C.2 caused the greatest improvement in compression ratio, compression time and decompression time on average. Section 5.3.1 showed that the overall compression ratio is improved by up to 14.82 percent. These results also showed that the use of preprocessors is beneficial for all compression programs except `ppmd`.

These preprocessors also generally resulted in an improvement in the compression and decompression times achieved by the programs (due to the reduction in filesize). Section 5.3.2 showed that the compression times were improved up to 46.91 percent, while the decompression times were improved by up to 29.98 percent which resulted in an average improvement in total times of up to 43.59 percent. It also showed that the improvement in compression and decompression time is related to the reduction by the preprocessor.

The problem with these preprocessors was that they needed the entire file to exist before compression. Section 6.1 discussed the use of the past month's data and Section 6.2 discussed the use of a combined dictionary with a preprocessor which performs word replacement. These preprocessors, PMU and COMBNUM respectively, also achieved an improvement in the compression ratio for all programs except `ppmd`. This improvement, however, was not greater than

the improvement by C.2. Section 6.3 discussed the creation of a single dictionary based on the semantic knowledge present in the log file. The preprocessor which used this dictionary, MYPHRASE, achieved the highest amount of improvement out of all the preprocessors tested. It achieved an average reduction of 49.49 percent in the size of the corpus (7 percent lower than the reduction by C.2) and achieved an average improvement in compression ratio of 0.01765 (12.81 percent greater than the improvement achieved by C.2).

From these results, it can be concluded that the semantic knowledge can be exploited to improve the compression ratio and the compression and decompression times achieved by all the compression programs except `ppmd`. The use of preprocessors, however, does improve the compression and decompression times for `ppmd`.

7.2.4 In different monitoring scenarios, which compression programs are the best choice to reduce the quantity of data?

Section 3.3 examined which compression programs were the best choices for the four different scenarios based on the analysis of the results presented in that chapter. As mentioned in the previous section, the use of the preprocessors improved the compression ratio and times achieved by the compression programs. Section 6.4 examined which preprocessors were the best choices for the different scenarios and how the improvements effected the compression programs and hence the best choices for each scenario.

	1		2	
Scenario	Program	Preprocessor	Program	Preprocessor
1	7zip	MyPhrase	bzip2	MyPhrase
2	zip	MyPhrase	gzip	MyPhrase
3	zip / gzip	MyPhrase	7zip	MyPhrase
4	lzop	MyPhrase	zip / gzip	MyPhrase

Table 7.1: Top two compression programs and preprocessors for each of the four scenarios

Table 7.1 shows a summary of which compression programs and preprocessors were found to be the best for the four different scenarios. It showed the top two compression programs and preprocessors for each of the four scenarios. The scenarios are as follows:

1. Filtering logs through to the central point for analysis (where latency is not important, but there is a desire to use as little bandwidth as possible)

2. Real-time monitoring (where a minimum point-to-point time is desired, using as little resources as possible)

3. Quick access, storing it compressed and later decompressing it for analysis (where fast decompression is desired, but the compression time does not matter. In addition, there is a desire to use as little bandwidth as possible)

4. Low system-time-usage for compression and decompression (where fast compression and decompression times are desired and the compression ratio does not matter)

In the first scenario, a slower decompression time was desired. This means that while `ppmd` achieved the lowest compression ratio, it was not considered. For the second scenario, the table shows the generalised choices. The best choice was to use `ppmd` with no preprocessor for transfer speeds below 263 KBps, `zip` and MYPHRASE for transfer speeds between 263 KBps and 4874 KBps, and `lzop(1)` with MYPHRASE for higher transfer speeds. For speeds above 150 Mbps, no compression and MYPHRASE should be used. In the case where the entire original file needs to be restored in the third scenario, the no preprocessor should be used. For the fourth scenario, if the preprocessing is not integrated into the logging system then no preprocessor should be used.

7.3 Reflection

This thesis has shown that the semantic knowledge can be exploited to improve the lowest compression achieved by all of the compression programs except `ppmd`. It has also shown that for all compression programs, there exists at least one compression level whose compression ratio is improved with the use of the preprocessor. This thesis has also shown that the improvement in compression time and decompression time is related to the reduction in filesize by the preprocessor. It has also shown that the semantic information obtained from an analysis of the files can be used to build a dictionary which achieves a lower compression ratio when used for word-replacement. A methodology was presented which can be used to construct a dictionary to compress maillogs on a server based on the knowledge of the network.

Within network monitoring, there are essentially two categories where compression can be used. These are for archival purposes and operational purposes. For archival purposes, the main issue is

the compression ratio and accessibility. The use of preprocessors provides an improvement in the compression ratio achieved by all the compression programs except `ppmd`. Since the time taken to perform processing and determine the dictionaries is not an issue, any of the preprocessors could be used. More sophisticated techniques could also be used to determine the dictionaries. For operational use, compression can be used to improve the transfer time and the search speed. The use of preprocessors which can be integrated into the logging system (such as MyPhrase) can improve the compression ratio and also provide an index to improve the search time. This thesis has shown that the semantic knowledge can be exploited to improve the performance of compression programs on log files.

7.4 Future work

There are a number of ways in which this work can be extended. This section expands on three possible improvements to the work: Improving the preprocessors; analysing other types of log data and implementing a plugin for a logging program such as `rsyslog`.

7.4.1 Improving the preprocessors

The speed of the preprocessors is a large issue, however these preprocessors are not optimised and hence can be further improved. One possibility is by off-loading the string processing onto a Nodal Core or a Graphic Processing Unit (GPU). Other possibilities include altering the way the search and replace operates and writing the preprocessors in languages such as C/C++. Preprocessors such as C.2 can also be integrated into the compression programs. A block based and adaptive approach can also be investigated. Integrating word-based replacement with the replacement of timestamps and IP addresses with their binary equivalents can also be investigated.

7.4.2 Analysing other types of log files

There are many different types of log files which exist. This thesis has shown how the semantic of maillogs can be exploited. Some log files, such as UNICODE log files, use different types of encoding. UNICODE log files also contain a larger number of free characters and hence larger dictionaries can be investigated. There are also binary files such as LibPCap files which contain a set structure. Preprocessors can also be developed for these binary files using common values for the fields.

7.4.3 Implementing a plugin for rsyslog

`rsyslog` is a syslog replacement which "supports on-demand disk buffering, TCP, writing to databases, configuring output formats, high-precision timestamps, filtering on any syslog message part, on-the-wire message compression and the ability to convert text files to syslog" [146]. `rsyslog` can be extended by creating third party plugins. A `rsyslog` plugin can be implemented which maintains a dictionary data structure and a plugin can be implemented which uses a custom dictionary such as MYPHRASE for word replacement.

References

[1] V. Yegneswaran, P. Barford, and J. Ullrich. Internet intrusions: global characteristics and prevalence. In *SIGMETRICS '03: Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 138–147, New York, USA, 2003. ACM Press.

[2] News Limited. Phishers hit monster jobs site. Available Online: http://www.australianit.news.com.au/story/0,24897,22293082-15306,00.html (Last Accessed 29 August 2007), August 2007.

[3] J. Blau. German government pcs hacked. Available Online: http://www.pcworld.com/article/id,136421-c,hackers/article.html (Last Accessed: 29 August 2007), August 2007.

[4] T. Bass. Intrusion detection systems and multisensor data fusion. *Communications of the ACM*, 43(4):99–105, April 2000.

[5] M.R. Endsley and D. J. Garland, editors. *Situational Awareness Analysis and Measurement*. Lawrence Erlbaum Associates, Mahwah, New Jersey, USA, 2000.

[6] S. Read-Miller and R. A. Rosenthal. Best Practices for building a Security Operations Center. Computer Associates White Paper, Available Online: http://www.ca.com/za/whitepapers (Last Accessed: 25 February 2006), April 2005.

[7] J. Babbin, D. Kleiman, E. Carter Jr.and J. Faircloth, and M. Burnett. *Security Log Management*. Syngress, Rockland, Knox County, Maine, USA, 2006.

[8] US Government. Health Insurance Portability and Accountability Act (HIPAA) of 1996. Available Online: http://www.cms.hhs.gov/HIPAAGenInfo/Downloads/HIPAALaw.pdf (Last Accessed: 3 June 2008), 1996.

[9] US Government. Public Company Accounting Reform and Investor Protection Act (Sarbanes-Oxley Act) of 2002. Available Online: http://frwebgate.access.gpo.gov/cgi-bin/getdoc.cgi?dbname=107_cong_bills&docid=f:h3763enr.tst.pdf (Last Accessed: 3 June 2008), July 2002.

[10] Republic of South Africa. Electronic Communications Act, Number 36 of 2005. Government Gazette, Available Online: http://www.polity.org.za (Last Accessed: 25 February 2008), April 2006.

[11] S. Read-Miller. Security Management: A New Model to Align Security With Business Needs. Computer Associates White Paper, Available Online: http://www.ca.com/za/whitepapers (Last Accessed: 25 February 2006), April 2005.

[12] F. Otten, B. Irwin, and H. Slay. The need for centralised, cross platform information aggregation. In *Proceedings of Information Security South Africa (ISSA) Conference*, Sandton, Johannesburg, South Africa, July 2006.

[13] A. Sah. A new architecture for managing enterprise log data. In *LISA '02: Proceedings of the 16th USENIX conference on System administration*, pages 121–132, Berkeley, CA, USA, November 2002. USENIX Association.

[14] S. Hanning. Recovering from disaster: Implementing disaster recovery plans following terrorism. Available Online: http://www.sans.org/reading_room/whitepapers/recovery (Last Accessed: 25 February 2008), September 2001.

[15] B. Glass. Log monitors in BSD UNIX. In *Proceedings of the BSDCon Conference*, San Francisco, Calafornia, USA, February 2002. USENIX Association.

[16] C. Siaterlis, B. Maglaris, and P. Roris. A novel approach for a distributed denial of service detection engine. In *Proceedings of 12th Annual HP OpenView University Association (HPOVUA) Workshop*, Geneva, Switzerland, July 2003.

[17] C. Lonvick. The BSD Syslog Protocol. RFC 3164 (Informational), August 2001.

[18] T. C. Bell, J. G. Clearly, and I. H. Witten. *Text compression*. Prentice Hall, Upper Saddle River, New Jersey, USA, 1990.

[19] T. C. Bell, J. G. Cleary, and I. H. Witten. Modeling for text compression. *ACM Computer Surveys (CSUR)*, 21(4):557–591, December 1989.

[20] R. Arnold and T. C. Bell. A corpus for the evaluation of lossless compression algorithms. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 201–210, Snowbird, Utah, USA, March 1997. IEEE Computer Society.

[21] L. Glassman, D. Grinberg, C. Hibbard, L. G. Reid, and M-C van Leunen. Hector: Connecting words with definitions. Research Report SRC92A, Systems Research Center, Digital Equipment Corporation, 1992.

[22] Project gutenberg ebooks. Available Online: http://www.gutenberg.org/dirs/etext00 (Last Accessed 28 Februrary 2008), November 1999.

[23] D. Lewis. The reuters-21578 text categorization test collection, distribution 1.0. Available Online: http://www.daviddlewis.com/resources/testcollections/reuters21578 (Last Accessed: 28 February 2008), May 2004.

[24] D. Lewis, Y. Tang, T. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning*, 5:361–397, April 2004.

[25] National Institute of Standards and Technology. Reuters corpora. Available Online:http://trec.nist.gov/data/reuters/reuters.html (Last Accessed 28 February 2008), September 2006.

[26] Maximum Compression Benchmarking Site. Summary of all single file-type lossless data compression tests. Available Online: http://www.maximumcompression.com/data/summary_sf.php (Last Accessed: 25 February 2008), 2007.

[27] R. Liska. Black_foxs benchmark. Available Online: http://blackfox.wz.cz/benchmark (Last Accessed: 28 February 2008), February 2008.

[28] M. Mahoney. Large text compression benchmark. Available Online: http://cs.fit.edu/mmahoney/compression (Last Accessed: 28 Februrary 2008), February 2008.

[29] S. Busch. Squeeze chart. Available Online: http://www.freehost.ag/squeezechart (Last Accessed: 28 February 2008), February 2008.

[30] J. Gilchrist. Jeff gilchrists archive comparision test. Available Online: (Last Accessed: 28 February 2008), February 2008.

[31] Squxe archivers chart. Available Online: http://maxcompress.narod.ru (Last Accessed: 28 Februrary 2008), November 2007.

[32] P. Deutsch. GZIP file format specification version 4.3. RFC 1952 (Informational), May 1996.

[33] PKWARE. ZIP File Format Specification. Available Online: http://www.pkware.com/business_and_developers/developer/popups/appnote.txt (Last Accessed: 25 February 2008), September 2006.

[34] M. Oberhumer. LZO - a real-time data compression algorithm. Available Online: http://www.oberhumer.com/opensource/lzo (Last Accessed: 25 February 2008), October 2005.

[35] D. Shkarin. *PPMd 9.1-12 System Manual Page*.

[36] ARJ Technical Information. Available Online: http://datacompression.info/ArchiveFormats/arj.txt (Last Accessed: 25 February 2008), April 1993.

[37] 7zip documentation. Documentation in p7zip package - DOCS/MANUAL/switches/method.htm. Package Available Online: http://downloads.sourceforge.net/p7zip/p7zip_4.43_src_all.tar.bz2 (Last Accessed: 25 February 2008), November 2005.

[38] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Research Report SRC124, Digital Systems Research Center, May 1994.

[39] A. Moffat. Special issue editorial: Lossless compression. *The Computer Journal*, 40(2):65–66, March 1997.

[40] Oxford english dictionary online, oxford university press. Available Online: http://dictionary.oed.com/ (Last Accessed: 3 June 2008), June 2008.

[41] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.

[42] C. E. Shannon. A mathematical theory of communication. *Bell System Technology Journal*, 27:379–423 and 623–656, July and October 1948.

[43] G. V. Cormack and R. N. S. Horspool. Data Compression Using Dynamic Markov Modelling. *The Computer Journal*, 30(6):541–550, December 1987.

[44] R. Franceshini and A. Mukherjee. Data compression using encrypted text. In *Proceedings of the Third Forum on Research and Technology Advances in Digital Libraries (ADL)*, pages 130–138, Washington DC, USA, May 1996.

[45] R. N. Horspool. Improving LZW. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 332–341, Snowbird, Utah, USA, April 1991.

[46] P. G. Howard. *The Design and Analysis of Efficient Lossless Data Compression Systems*. PhD thesis, Department of Computer Science, Brown University, Providence, Rhode Island, USA, June 1993.

[47] P. M. Long, A. I. Natsev, and J. S. Vitter. Text compression via alphabet re-representation. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 161–170, Snowbird, Utah, USA, March 1997.

[48] D. Shkarin. PPM: One step to practicality. In *Proceedings of 12th IEEE Data Compression Conference (DCC)*, pages 202–211, Snowbird, Utah, USA, April 2002.

[49] Maximum Compression Benchmarking Site. Logfile compression test. Available Online: http://www.maximumcompression.com/data/log.php (Last Accessed: 25 February 2008), 2007.

[50] D. A. Huffman. A method for construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[51] P. Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational), May 1996.

[52] P. Elias. Universal codeword sets and representation of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, March 1975.

[53] I. H. Witten, R. M. Neal, and J. G. Clearly. Arithmetic coding for data compression. *Communications of the ACM*, 30(30):520–540, June 1987.

[54] J. G. Cleary and I. H. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.

[55] G. N. N. Martin. Range encoding: an algorithm for removing redundancy from a digitised message. In *Video and Data Recording Conference*, Southhampton, England, July 1979.

[56] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.

[57] S. W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3):399–401, July 1966.

[58] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions of Information Theory*, 23(3):337–343, May 1977.

[59] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions of Information Theory*, 24(5):530–536, September 1978.

[60] J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM (JACM)*, 29(4):928–951, October 1982.

[61] I. Pavlov. LZMA 4.43 SDK. Available Online: http://www.7zip.org/sdk.html (Last Accessed: 25 February 2008), 2005.

[62] I. Pavlov. Open Discussion - LZMA Algorithm. Available Online: https://sourceforge.net/forum/forum.php (Last Accessed: 25 February 2008), February 2005.

[63] P. Deutsch and J-L. Gailly. ZLIB Compressed Data Format Specification version 3.3. RFC 1950 (Informational), May 1996.

[64] A. Moffat. Implementing the PPM Data Compression Scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, November 1990.

[65] I. H. Witten and T. C. Bell. The zero frequency problem: Estimating the probabilities of novel events in adaptive text compression. *IEEE Transactions on Information Theory*, 37(4):1085–1094, July 1991.

[66] J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. In J.A Storer and M. Cohn, editors, *Proceedings of the IEEE Data Compression Conference*, pages 52–61, Snowbird, Utah, March 1995. IEEE Computer Society Press, Los Alamitos, California.

[67] C. Bloom. Solving the problems of context modeling. Technical report, California Institute of Technology, March 1998.

[68] W. J. Teahan. Probability estimation for PPM. In S. Reeves and S. Cranefield, editors, *Proceedings of The New Zealand Computer Science Research Students Conference*, Hamilton, New Zealand, 1995. University of Waikato.

[69] J. Abel. Improvements to the Burrows-Wheeler Compression Algorithm: After BWT Stages. *Preprint submitted for publication in ACM Transactions*, 2003.

[70] B. Chapin and S. R. Tate. Higher compression from the Burrows-Wheeler transform by modified sorting. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 532–, Snowbird, Utah, USA, March 1998.

[71] P. Fenwick. Block sorting compression - final report. Technical Report 130, University of Auckland, April 1996.

[72] D. Wheeler. An implentation of block coding. Unpublished, Available Online: ftp://ftp.cl.cam.ac.uk/users/djw3/bred.ps (Last Accessed: 25 February 2008), October 2005.

[73] S. Deorowicz. Improvements to Burrows-Wheeler Compression Algorithm. *Software: Practice and Experience*, 30(13):1465–1483, June 2000.

[74] B. Balkenhol, S. Kurtx, and Y. M. Shtarkov. Modifications of the burrows wheeler data compression algorithm. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 188–197, Snowbird, Utah, USA, March 1999.

[75] B. Balkenhol and Y. Shtarkov. One attempt of a compression algorithm using the bwt. In *SFB343: Sonderforshungsbereich 343 - Diskrete Strukturen in der Mathematik (Discrete Structures in Mathematics)*. Department of Mathematics, University of Bielefeld, 1999.

[76] N. J. Larsson. The context trees of block sorting compression. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 189–198, Snowbird, Utah, USA, March 1998.

[77] H. Kruse and A. Mukherjee. Improve text compression ratios with the burrows-wheeler transform. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 536–, Snowbird, Utah, USA, March 1999.

[78] J. Seward. bzip2 and libbzip2, version 1.0.3: A program and library for data compression. Available Online: http://www.bzip.org/1.0.3/bzip2-manual-1.0.3.html (Last Accessed: 25 February 2008), February 2005.

[79] P. Fenwick. Experiments with a block sorting text compression algorithm. Technical Report 111, University of Auckland, May 1995.

[80] P. Fenwick. Improvements to the block sorting text compression algorithm. Technical Report 120, University of Auckland, August 1995.

[81] A. Anderson and S. Nilsson. A new efficient radix sort. In *Proceedings of 35th Symposium on Foundations of Computer Science*, pages 714–721, 1994.

[82] P. Fenwick. Block sorting text compression. In *Proceedings of the 19th Australasian Computer Science Conference*, Melbourne, Australia, February 1996.

[83] K. Sadakane. A fast algorithm for making suffix arrays and for burrows-wheeler transformation. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 129–138, Snowbird, Utah, USA, March 1998.

[84] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid idetification of repeated patterns in strings, arrays and trees. In *Proceedings of the fourth annual ACM Symposium on Theory of Computing*, pages 125–136, Denver, Colarado, USA, May 1972. ACM Press, New York, USA.

[85] J. Bentley and R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, New Orleans, Louisiana, USA, January 1997.

[86] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of 1st ACM-SIAM Sympoium on Discrete Algorithms*, pages 319–327, San Francisco, California, USA, January 1990.

[87] J. Seward. On the performance of bwt sorting algorithms. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 173–182, Snowbird, Utah, USA, March 2000.

[88] S. Kurtz and B. Balkenhol. Space effecient linear time computation of the burrows and wheeler-transformation. Technical report, Technische Fakultat, University Bielefeld, Germany, 1999.

[89] H. Itoh and H. Tanaka. An efficient method for in memory construction of suffix arrays. In *String Processing and Information Retrieval Symposium*, pages 81–88, Cancun, Mexico, September 1999.

[90] T-H. Kao. Improving suffix-array construction algorithms with applications. Master's thesis, Department of Computer Science, Gunma University, Maebashi City, Gunma, Japan, 2001.

[91] G. Manzini and P. Ferragina. Engineering a Lightweight Suffix Array Construction Algorithm (Extended Abstract). *Lecture Notes in Computer Science, Algorithms*, 2461:698–710, January 2002.

[92] J. Seward. Space-time tradeoffs in the inverse b-w transform. In *Proceedings of the IEEE Data Compression Conference (DCC)*, pages 439–448, Snowbird, Utah, USA, March 2001.

[93] M. Schindler. A fast block-sorting algorithm for lossless data compression. In *Proceedings of IEEE Data Compression Conference*, pages 469–, Snowbird, Utah, USA, March 1997.

[94] Z. Arnavut and S. S. Maglieras. Block sorting and compression. In *Proceedings of IEEE Data Compression Conference (DCC)*, page 181, Snowbird, Utah, USA, 1997. IEEE Computer Society.

[95] Z. Arnavut. Move-to-front and inversion coding. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 193–202, Snowbird, Utah, USA, 2000.

[96] S. Deorowicz. Second step algorithms in the Burrows-Wheeler compression algorithm. *Software: Practice and Experience*, 32(2):99–111, February 2002.

[97] D. S. Hirschberg and D.A. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 23(4):449–458, April 1990.

[98] M. Mahoney. The paq data compression programs. Available Online: http://cs.fit.edu/ mmahoney/compression/paq.html (Last Accessed: 25 February 2008), 2007.

[99] J. Schmidhuber and S. Heil. Sequential neural text compression. *IEEE Transcations on Neural Networks*, 7(1):142–146, January 1996.

[100] M. V. Mahoney. Fast text compression with neural networks. In J. Etheridge and B. Manaris, editors, *Proceedings of the Thirteenth International Florida Artificial Intelligence Research Society Conference*, pages 230–234, Orlando, Florida, USA, May 2000.

[101] M. Mahoney. Paq8 source code - paq8f file compressor/archiver. Available Online: http://cs.fit.edu/ mmahoney/compression/paq8f.cpp (Last Accessed: 25 February 2008), 2006.

[102] M. V. Mahoney. Adaptive weighting of context models for lossless data compression. Technical Report CS-2005-16, Florida Institute of Technology, USA, 2005.

[103] M Software. State of the Art Data Compression Technology - About WinRK. Available Online: http://www.msoftware.co.nz/WinRK_about.php (Last Accessed: 25 February 2008), 2007.

[104] The Calgary Corpus Compression Challenge. Available Online: http://mailcom.com/challenge (Last Accessed: 25 February 2008), 2007.

[105] Compress the 100MB file enwik8 (The Hutter Prize). Available Online: http://prize.hutter1.net/ (Last Accessed: 25 February 2008), 2007.

[106] R. N. Horspool and G. V. Cormack. Constructing word-based text compression algorithms. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 62–71, Snowbird, Utah, USA, March 1992.

[107] A. Moffat. Word-based text compression. *Software: Practices and Experience*, 19(2):185–198, February 1989.

[108] J. Dvorsky, J. Pokorny, and V. Snasel. Word-based compression methods and indexing for text retrieval systems. *Lecture Notes in Computer Science*, 1691:75–84, September 1999.

[109] J. Dvorsky, J. Pororny, and V. Snasel. Word compression methods for large documents. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 523–, Snowbird, Utah, USA, March 1999.

[110] J. Dvorsky, J. Pororny, and V. Snasel. Word compression methods with empty words and nonwords for text retrieval systems. In *Proceedings of DATSEM*, 1998.

[111] R. Y. K. Isal and A. Moffat. Word-based block-sorting text compression. In *Proceedings of 24th Australasian conference on Computer Science*, volume 11 of *ACM International*

Conference Proceedings Series, pages 92–99, Gold Coast, Queensland, Australia, January 2001. ACM Press.

[112] R. Y. K. Isal, A. Moffat, and A. C. H. Ngai. Enhanced word-based block sorting text compression. *Australian Computer Science Communications*, 24(1):129–137, January 2002.

[113] A. Moffat and R. Y. K. Isal. Word-based text compression using the Burrows-Wheeler transform. *Information Processing and Management*, 41(5):1175–1192, September 2005.

[114] E. S. de Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *Proceedings of the Fourth South American Workshop on String Processing*, volume 8 of *International Informatics Series*, pages 95–111. Carleton University Press, 1997.

[115] P. Skibinski, S. Grabowski, and S. Deorowicz. Revisiting dictionary-based compression. *Software: Practice and Experience*, 35(15):1455–1476, December 2005.

[116] R. Franceschini, H. Kruse, N. Zhang, R. Iqbal, and A. Mukherjee. Lossless, reversible transformation that improve text compression ratios. Technical report, University of Florida, 2000.

[117] H. Kruse and A. Mukherjee. Preprocessing text to improve compression ratios. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 556–, Snowbird, Utah, USA, March 1998.

[118] H. Kruse and A. Mukherjee. Data compression using text encryption. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 447–, Snowbird, Utah, USA, March 1997.

[119] F. Awan and A. Mukherjee. LIPT: A lossless text transform to improve compression. In *Proceedings of International Conference on Information and Theory : Coding and Computing*, pages 452–460, Las Vegas, Nevada, USA, April 2001. IEEE Computer Society.

[120] F. Awan, N. Zhang, N. Motgi, R. Iqbal, and A. Mukherjee. A new text preprocessing algorithm for bzip2 and PPM. In *Proceedings of Data Compression Conference*, page 481, Snowbird, Utah, USA, 2001.

[121] B. Chapin. Switching between two on-line update algortihms for higher compression of Burrows-Wheeler transformed data. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 183–192, Snowbird, Utah, USA, March 2000.

[122] M. Effros. PPM performance with BWT complexity. *Proceedings of the IEEE*, 88(11):1703–1712, November 2000.

[123] K. Sadakane, T. Okazaki, and H. Imai. Implementing the context tree weighting method for text compression. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 123–132, Snowbird, Utah, USA, March 2000.

[124] W. Sun, A. Mukherjee, and N. Zhang. A dictionary-based multi-corpora text compression system. In *Proceedings of IEEE Data Compression Conference (DCC)*, page 448, Snowbird, Utah, USA, March 2003.

[125] W. J. Teahan and J. G. Cleary. Models of english text. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 12–21, Snowbird, Utah, USA, March 1997.

[126] W. J. Teahan. *Modelling English text*. PhD thesis, University of Waikato, Waikato, New Zealand, 1998.

[127] W. J. Teahan and J. G. Cleary. The entropy of english using PPM-based models. In *Proceeedings of IEEE Data Compression Conference (DCC)*, pages 53–62, Snowbird, Utah, USA, March 1996.

[128] S. Grabowski. Text preprocessing for burrows-wheeler compression algorithm. In S. Klimczak, editor, *Proceedings of VII Konferencja Sieci i Systemy Informatyczne - Teoria, Projekty, Wdrozenia*, pages 229–239, 1999.

[129] R. Y. K. Isal and A. Moffat. Parsing strategies for BWT compression. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 429–438, Snowbird, Utah, USA, March 2001.

[130] J. Abel and W. Teahan. Universal text preprocessing for data compression. *ACM Transactions on Computers*, 54(5):497–507, May 2005.

[131] P. Skibinski. Two-level dictionary based compression. In *Proceedings of IEEE Data Compression Conference (DCC)*, pages 481–, Snowbird, Utah, USA, March 2005.

[132] K. Hatonen, J. F. Boulicaut, M. Klemettinen, M. Miettinen, and C. Masson. Comprehensive log compression with frequent patterns. *Lecture Notes in Computer Science*, 2737:360–370, September 2003.

[133] P. Skibinski and J. Swacha. Fast and efficient log file compression. In *Proceedings of 11th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, Varna, Bulgaria, September 2007.

[134] S. Grabowski and S. Deorowicz. Web log compression. *AGH Automatyka*, 2007. To appear. Preprint Available Online: http://sun.aei.polsl.pl/sdeor/pub/gd07a.pdf (Last Accessed: 25 February 2008).

[135] R. Balakrishnan and R. K. Sahoo. Lossless compression for large scale cluster logs. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 7–, Rhodes Island, Greece, April 2006.

[136] I. Pavlov. 7z format. Available Online: http://www.7-zip.org/7z.html (Last Accessed: 28 February 2008), 2008.

[137] R. K. Jung. Data compression/decompression method and apparatus. US Patent 5140321, August 1992.

[138] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: a survey. *Computer Networks*, 38:393–422, 2002.

[139] Christopher M. Sadler and Margaret Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys) 2006*, 2006.

[140] B. W. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, May 1988.

[141] J. Postel. Simple Mail Transfer Protocol. RFC 821 (Standard), August 1982. Obsoleted by RFC 2821.

[142] J. Klensin. Simple Mail Transfer Protocol. RFC 2821 (Proposed Standard), April 2001.

[143] G. Vaudreuil. Enhanced Mail System Status Codes. RFC 1893 (Proposed Standard), January 1996. Obsoleted by RFC 3463.

[144] G. Vaudreuil. Enhanced Mail System Status Codes. RFC 3463 (Draft Standard), January 2003. Updated by RFCs 3886, 4468, 4865, 4954.

[145] N. Freed. SMTP Service Extension for Returning Enhanced Error Codes. RFC 2034 (Proposed Standard), October 1996.

[146] R. Gerhards. Rsyslog, the enhanced syslogd for Linux and Unix. Available Online: http://www.rsyslog.com/ (Last Accessed: 3 June 2008), June 2008.

Appendix A

Statistical Methods

Within this thesis, there are a number of places where statistical methods have been used apart from averages and standard deviations. This appendix aims to explain how these statistics are calculated and why they were used.

A.1 Hypothesis Tests

In a hypothesis test, a hypothesis is defined and then this hypothesis is tested using an appropriate statistical test. A value known as the p-value is obtained from this statistical test. This is the observed significance level of the result from the statistical test (i.e. the probability of obtaining a test statistic at least as extreme as the test statistic calculated from the sample). A desired level of significance is also defined (i.e. the probability of rejecting the hypothesis when it is in fact true). This is often denoted using α, $\alpha = 0.05$ is a 95 percent confidence level. If the p-value is smaller than α then the hypothesis is rejected, otherwise the hypothesis fails to be rejected.

A.2 Paired Difference

In cases where it is not clear whether there is a difference between the average differences then a paired difference can be used. This test is used with Paired data (i.e. data which can be grouped in pairs such as 2 compression ratios on the same file using different preprocessors). The t-distribution (which approximates the normal distribution based on a parameter called the degree of freedom) is used to verify the paired difference. To verify if there is a difference between

the average difference between samples, the difference between each of the paired samples is calculated and then a standard t-test is performed using this data.

A.3 Pearson's Correlation Coefficient

The Pearson's correlation coefficient (Also called the Pearson Product Moment Correlation) is calculated using the values for the two variables which are being analysed. The closer the absolute value of the coefficient is to 1 (perfect linear relationship), the stronger the correlation between the two variables. The coefficient is calculated as follows:

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{[\sum_{i=1}^{n}(x_i - \bar{x})][\sum_{i=1}^{n}(y_i - \bar{y})]}}$$

x_i and y_i represents each of the n observations for the two variables being analysed. $\bar{x}$ and $\bar{y}$ are the averages for the values of the x and y.

A.4 Chi-Squared Goodness-of-Fit Test

This test is used to determine is a given dataset follows a specified distribution. The hypothesis is that the data follows a specified distribution. The Chi-Squared statistic is calculated using observed and expected values as follows:

$$\chi^2 = \sum_{i=1}^{n} \frac{(o_i - e_i)^2}{e_i},\ df = n - 1$$

$o_i$and e_i represent the observed and expected values for n values and df is the degree of freedom for the chi-squared distribution. Using Paired data as observed and expected values, it can be verified whether the paired variables follow the same distribution.

Appendix B

Construction of a ruleset

When analysing the semantics of the maillog files, a ruleset is constructed for the analysis tool. This appendix explains how this ruleset was constructed and why the rules were included.

B.1 Using a single maillog file

The first obvious rules are to replace email addresses, IP addresses and hostnames. This done using either an inner rule or an outer rule. An inner rule, however is better since there are no spaces within an email addresses, an IP address or a hostname which means that email addresses, IP addresses and hostnames are contained within single token. The inner rules are also run before the outer rules. The email addresses are generally contained within <>. The following rules can be used to replace email addresses, IP addresses and hostnames with their relevant placeholders:

```
O: replregex
P: <[^@]+@[^>]+>
R: <@>
```

```
O: replregex
P: (25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\.
   (25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\.
   (25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\.
   (25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])
R: IP
```

```
O: replregex
P: localhost|(([A-Za-z0-9\-_]+\.)+[A-Za-z]{2,3})
R: hostname
```

There are 11 hex digits at the beginning of some of the messages for postfix/cleanup, postfix/bounce, postfix/discard, postfix/local, postfix/pickup, postfix/qmgr, postfix/smtp, postfix/smtpd, postfix/tlsmgr, postfix/virtual. These are message identifiers. The following inner rule can be used to replace the message identifiers with a placeholder:

```
O: replregex
P: [0-9ABCDEF]{11}
R: #Hex#
```

postfix/discard, postfix/local, postfix/qmgr, postfix/smtp, postfix/smtpd and postfix/virtual produce messages which contain "to=*misc",* "from=*misc*", etc. where *misc* is variable text such as email addresses. The variable text can be replaced with a placeholder by using the rfind function. The following inner rule can be used to replace the variable text with a placeholder:

```
O: rfind
P: =
R: []
```

postfix/anvil and postfix/scache contain dates within some of their log lines. The following outer rule can be used to replace the date with the placeholder DATE:

```
O: replregex
P: (Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec)\
   ((\ [1-9])|[0-9]{1,2})\ [0-9]{1,2}:[0-9]{1,2}:
   [0-9]{1,2}
R: DATE
```

amavis is a mail virus scanner. The log messages which are outputted by this process are quite different. Many of the messages for the process, however begin with two numbers in brackets separated by a dash contained in brackets. The following line is an example:

```
Aug 10 00:28:59 titania amavis[45211]: (45211-10) Passed CLEAN...
```

Two numbers separated by a dash and contained within brackets can can also be found in the messages from the postfix/cleanup, postfix/qmgr and postfix/smtp processes. The following inner rule can be used to replace the two numbers which are separated by a dash and contained within brackets with a placeholder:

O:	`search`
P:	`([0-9]{5}-[0-9]{2})`
R:	`(#####-##)`

Further reductions can be made by using process specific rules. postfix/smtp contains a number of lines which contain descriptions of the status which is reported. The following line is an example of this:

```
#Hex#: to[] relay[] delay[] delays[] dsn[] status[] (250 2.0.0 k7ACI06C0
18260 Message accepted for delivery)
```

These messages can be replaced by using the following process specific outer rule:

Pr:	`postfix/smtp`
O:	`replregex`
P:	`status\[\]\ (.*)`
R:	`status[] (message)`

postfix/anvil contains a number of instances of numbers. The following process specific outer rule can be used to replace these numbers with a placeholder:

Pr:	`postfix/anvil`
O:	`replregex`
P:	`[0-9]+`
R:	`#`

The amavis process also requires a number of process specific rules since these rules do not lead to any groupings of semantic (i.e all processed messages are still unique). The following text is an example of one the messages for amavis after the rules are applied.

```
(#####-##) Passed CLEAN, [IP] [IP] <@> -> <@>, Message-ID: <@>, mail_id:
EvirT0Jc-Wu4, Hits: -2.186, queued_as: #Hex#, 7486 ms
```

Within this line, the values for the mail_id, the Hits and the amount of milliseconds cause the messages to be unique. These need to be replaced with placeholders. The following process specific inner rules can be used to replace these elements with placeholders:

Pr:	`amavis`
O:	`rfind`
P:	:
R:	`{}`

Pr:	`amavis`
O:	`replregex`
P:	`[0-9]+\ ms`
R:	`# ms`

There are a number of cases of multiple email addresses e.g. <@> -> <@>, <@>. This can be reduced by adding an outer rule which replaces these with <@> -> <@>+. This outer rule is as follows:

O:	`replregex`
P:	`<@>\ ->\ (<@>,)+`
R:	`<@> -> <@>+`

B.2 Extra rules for larger corpus

A number of email addresses occur within the results using the rules from the previous section. The following rule inner rule can be added to replace email addresses with a placeholder:

```
O: replregex
P: [a-zA-Z0-9\-_\.]+@(localhost|(([A-Za-z0-9\-_]+\.)+
   [a-zA-Z]{2,3}))
R: email
```

This rule needs to be added before the rule which replaces hostnames. Since it includes the regular expression used to replace the hostnames with a token. If it is added after the rule which replaces hostname then the search criteria needs to be modified to be [a-zA-Z0-9\-_\.]+@hostname.

amavis, postfix/cleanup, postfix/qmgr, postfix/smtp and sqlgrey all include instances of three numbers separated by dashes contained within brackets. The following rule can be used to replace these instances with a placeholder:

```
O: search
P: ([0-9]{5}-[0-9]{2}-[0-9]{1,3})
R: (#####-##-##)
```

Within postfix/smtpd, there are a number of instances of messages such as:

```
warning: Connection rate limit exceeded: 244 from hostname[IP] for
service smtp
```

The number (in this case 244) needs to be replaced with a placeholder. The following process specific rule can be used to replace the number with a placeholder:

```
Pr: postfix/smtpd
O: replregex
P: exceeded:\ [0-9]+
R: exceeded:  #
```

The amavis process produces a number of messages which contain INFECTED(*virusname*) where *virusname* is the name of a virus. This following process specific rule can be used to replace the virus name with a placeholder:

```
Pr: amavis
O: replregex
P: INFECTED\ \(.*\),
R: INFECTED (virus),
```

Many of the messages for the postfix/master process contain information about processes which have exited (50.16 percent of the messages). These messages contain the PID of the process which has exited. The following process specific outer rule can be used to replace the PID with a #:

```
Pr: postfix/master
O: replregex
P: pid\ [0-9]+
R: pid #
```

The sqlgrey process did not occur in the 3MB syslog file. There are a number of rules which are necessary to determine the semantics of the sqlgrey process. The following is an example of a message for the sqlgrey process:

```
2006/03/29-01:49:10 CONNECT TCP Peer: "127.0.0.1:55313" Local:
"127.0.0.1:2501"
```

When the rules from the previous section are run, 12.0.0.1:55313 becomes IP:55313. The port needs to be replaced with a placeholder. The port and the date can be replaced with placeholders using the following rules:

```
Pr: sqlgrey:
O: replregex
P: IP:[0-9]+
R: IP:port
```

```
Pr: sqlgrey:
O: replregex
P: (19|20)[0-9]{2}\/(0|1)[0-9]\/[0-3][0-9]\-[0-2][0-9]
   :[0-5][0-9]:[0-5][0-9]
R: YYYY/MM/DD-HH:MM:SS
```

In some of the messages produced by sqlgrey, there is an occurrence of the time contained in brackets. This can be replaced with a placeholder using the following rule:

```
Pr: sqlgrey:
O: replregex
P: \([0-2][0-9]:[0-5][0-9]:[0-5][0-9]\)
R: (HH:MM:SS)
```

A partial IP address which refers to a class C network occurs within some the messages for sqlgrey. This can be replaced with a placeholder by using the following rule:

```
Pr: sqlgrey:
O: replregex
P: (25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\.
   (25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])\.
   (25[0-5]|2[0-4][0-9]|1[0-9]{2}|[1-9][0-9]|[0-9])
R: C-IP
```

Appendix C

Analysis of Semantics

The analysis of the semantics of the log file showed a number of trends which were highlighted in Section 4.2.2. This appendix presents the results for the top 5 processes - postfix/smtp, postfix/smtpd, amavis, postfix/cleanup and postfix/qmgr. More detailed results can be found in Appendix E.2.

C.1 "Handmade" analysis

The "handmade" analysis of the 3MB maillog file produced the following results:

In these results, #hex# represents a hex number, # represents a number, <> represents an email address, *hostname* represents a hostname, *code* represents a code, *explanation* represents an explanation of that code, *description* represents a description, *IP* represents an IP address, *date* represents a date, *mailaddress* represents a mail address, and *misc* represents miscellaneous text.

- postfix/smtp
  - host *hostname* said: *code explanation*
  - host *hostname* refused to talk to me: *code explanation*
  - #hex# to=<>, relay=, delay=#, dsn=#, status=sent (*message*) where *message* is:
    - 250 dsn *mailaddress* Queued mail for delivery
    - 250 dsn Ok, id=#####-# from *hostname*([*IP*]:#): BOUNCE

        * 250 2.0.0 Ok: queued as #hex#
        * 254 Date discarded, id=#####-# - VIRUS: *description*
        * 250 2.0.0 OK # #
        * 250 2.0.0 Ok: queued as #hex#
    - warning: peer certificate has no subject CN
    - connect to *hostname*[*IP*]: *message* where *message* is:
        * connection refused
        * operation timed out
        * No route to host
    - certificate verification failed

- postfix/smtpd
    - connect from *hostname*[*IP*]
    - disconnect from *hostname*[*IP*]
    - #hex# client=*hostname*[*IP*]
    - NOQUEUE: reject: RCPT from *hostname*[*IP*]: *code date mailaddress* Recipient address rejected: Domain not found from=<> proto=ESMTP helo=<>

- postfix/cleanup
    - #hex# message-id=*mailaddress*

- postfix/qmgr
    - #hex# from=<>, (status=expired, returned to sender) size=#, nrcpt=#, (queue active)
    - #hex# to=<>, relay=none, delay=#, delays=#/#/#/#, dsn=4.4.1, status=defered (*explanation*)
    - #hex# removed

- amavis
    - (#####-##) Passed CLEAN, [IP] [IP] <> -> <> Message-ID: *misc* Resent-Message-ID: *misc* mail_id: *misc* Hits: #.### queued_as: *misc* # ms

  - (#####-##) Passed BAD-HEADER, [IP] [IP] <> -> <> quarantine: *misc* Message-ID: *misc* mail_id: *misc* Hits: #.### queued_as: *misc* # ms
  - (#####-##) Blocked INFECTED (), [IP] [IP] <> -> <> quarantine: *misc* Message-ID: *misc* mail_id: *misc* Hits: #.### # ms
  - (#####-##) Blocked SPAM, [IP] [IP] <> -> <> quarantine: *misc* Message-ID: *misc* mail_id: *misc* Hits: #.### # ms
  - (#####-##) Passed SPAMMY, [IP] [IP] <> -> <> quarantine: *misc* Message-ID: *misc* mail_id: *misc* Hits: #.### queued_as: *misc* # ms

amavis also contains a number of other messages reporting information such as Perl version and messages such as Found Decoder for .... The frequency of these messages is small in comparison to the messages shown above for amavis.

These results reveals that there are defined trends which can be observed. The tool described in the previous section was designed to automate the process used for analysing the log file on hand so that it can be applied.

C.2 Analysis with tool

The output from the tool is first the frequency of the different processes and then the output for each process. Each line is followed by a count for lines containing that pattern. The results for the analysis using the tool with the rule set which is built iteratively are as follows:

- postfix/qmgr
  - #Hex#: from[] status[] returned to sender 1
  - #Hex#: to[] relay[] delay[] delays[] dsn[] status[] (delivery temporarily suspended: connect to hostname[IP]: Connection refused) 2
  - #Hex#: removed 2859
  - #Hex#: from[] size[] nrcpt[] (queue active) 3380
- postfix/smtpd

  - NOQUEUE: reject: RCPT from hostname[IP]: 450 4.1.2 <@>: Recipient address rejected: Domain not found; from[] to[] proto[] helo[] 522
  - #Hex#: client[] 2766
  - disconnect from hostname[IP] 3122
  - connect from hostname[IP] 3124

- postfix/cleanup
  - #Hex#: resent-message-id[] 4
  - #Hex#: message-id[] 2889

- postfix/smtp
  - #Hex#: host hostname[IP] said: 450 <@>: Recipient address rejected: Policy Rejection: Greylisting in effect. Please try again later. (in reply to RCPT TO command) 1
  - #Hex#: host hostname[IP] said: 451 Temporary local problem - please try later (in reply to end of DATA command) 1
  - #Hex#: lost connection with hostname[IP] while receiving the initial server greeting 2
  - #Hex#: host hostname[IP] refused to talk to me: 421 Server busy. Try later ... 2
  - #Hex#: host hostname[IP] said: 451 hostname Resources temporarily unavailable. Please try again later [#4.16.5]. (in reply to end of DATA command) 3
  - certificate verification failed for hostname: num[] signed certificate in certificate chain 3
  - #Hex#: host hostname[IP] said: 451 Could not complete sender verify callout (in reply to RCPT TO command) 3
  - warning: peer certificate has no subject CN 5
  - certificate verification failed for hostname: num[] has expired 6
  - #Hex#: to[] orig_to[] relay[] delay[] delays[] dsn[] status[] (message) 7
  - #Hex#: to[] relay[] conn_use[] delay[] delays[] dsn[] status[] (message) 7
  - certificate verification failed for hostname: num[] not trusted 10

  - certificate verification failed for hostname: num[] to get local issuer certificate 10
  - certificate verification failed for hostname: num[] to verify the first certificate 10
  - connect to hostname[IP]: No route to host (port 25) 11
  - certificate verification failed for hostname:certificate has expired 12
  - certificate verification failed for hostname: num[] signed certificate 19
  - #Hex#: host hostname[IP] said: 451 The users mailbox is full. Please try re-sending your e-mail later. (in reply to RCPT TO command) 28
  - connect to hostname[IP]: Operation timed out (port 25) 96
  - connect to hostname[IP]: Connection refused (port 25) 395
  - #Hex#: to[] relay[] delay[] delays[] dsn[] status[] (message) 3440

- amavis

  - (#####-##) Passed CLEAN, [IP] [IP] <@> -> <@>+ Message-ID{} Resent-Message-ID{} mail_id{} Hits{} queued_as{} # ms 2
  - (#####-##) Passed BAD-HEADER, [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} queued_as{} # ms 6
  - (#####-##) Blocked INFECTED (hostnametion-157), [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} # ms 6
  - (#####-##) Blocked INFECTED (Win32{} [Wrm]), [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} # ms 10
  - (#####-##) Passed SPAMMY, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} queued_as{} # ms 14
  - (#####-##) Blocked SPAM, [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} # ms 35
  - (#####-##) Passed CLEAN, [IP] [IP] <@>, Message-ID{} mail_id{} Hits{} queued_as{} # ms 40
  - (#####-##) Passed CLEAN, <@> -> <@>+ Message-ID{} mail_id{} Hits{} queued_as{} # ms 52
  - (#####-##) Passed CLEAN, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} queued_as{} # ms 1224

These results are similar to the ones shown in the "handmade" analysis. This shows that the hand made analysis can be replicated with the use of the tool and a rule set which is built up iteratively.

C.3 Analysing the larger corpus

The results on the larger corpus using the augmented rule set are as follows:

These results only show patterns which have more than 1000 occurrences.

- postfix/qmgr
  - #Hex#: from[] status[] returned to sender 1588
  - #Hex#: to[] relay[] delay[] status[] (delivery temporarily suspended: connect to hostname[IP]: Operation timed out) 2661
  - #Hex#: to[] relay[] delay[] status[] (delivery temporarily suspended: lost connection with hostname[IP] while sending message body) 4343
  - #Hex#: to[] relay[] delay[] status[] (delivery temporarily suspended: connect to hostname[IP]: Connection refused) 5400
  - #Hex#: to[] relay[] delay[] status[] (delivery temporarily suspended: connect to hostname[IP]: server refused to talk to me: 421 Insufficient System Storage.(IMail 7.07)) 6098
  - #Hex#: to[] relay[] delay[] delays[] dsn[] status[] (delivery temporarily suspended: connect to hostname[IP]: Connection refused) 7886
  - #Hex#: to[] relay[] delay[] delays[] dsn[] status[] (delivery temporarily suspended: connect to hostname[IP]: No route to host) 16792
  - #Hex#: to[] relay[] delay[] delays[] dsn[] status[] (delivery temporarily suspended: connect to IP[IP]: Connection refused) 55848
  - #Hex#: to[] relay[] delay[] delays[] dsn[] status[] (delivery temporarily suspended: connect to hostname[IP]: Operation timed out) 65344
  - #Hex#: removed 2077468
  - #Hex#: from[] size[] nrcpt[] (queue active) 2457350
- postfix/smtpd

- timeout after CONNECT from hostname[IP] 1017
- NOQUEUE: reject: RCPT from hostname[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; Dynamic/Residential IP range listed by NJABL dynablock - http://hostname/hostnamel; from[] to[] proto[] helo[] 1041
- NOQUEUE: reject: RCPT from hostname[IP]: 450 4.7.1 <@>: Recipient address rejected: Throttling too many connections from new source - Try again later.; from[] to[] proto[] helo[] 1083
- too many errors after RCPT from unknown[IP] 1088
- NOQUEUE: reject: RCPT from hostname[IP]: 450 4.1.2 <@>: Recipient address rejected: Malformed DNS server reply; from[] to[] proto[] helo[] 1181
- warning: malformed domain name in resource data of MX record for hostname: 1206
- warning: valid_hostname: empty hostname 1220
- NOQUEUE: reject: RCPT from unknown[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; Dynamic/Residential IP range listed by NJABL dynablock - http://hostname/hostnamel; from[] to[] proto[] helo[] 1223
- NOQUEUE: reject: RCPT from hostname[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; Escalated Listing (Spam or Spam Support) See: http://hostname/hostnameml?IP; from[] to[] proto[] helo[] 1246
- lost connection after MAIL from hostname[IP] 1497
- warning: hostname: RBL lookup error: Host or domain name not found. Name service error for name[] type[] Host not found, try again 1665
- NOQUEUE: reject: RCPT from unknown[IP]: 554 Service unavailable; Client host [IP] blocked using hostname; http://hostname/listing?IP; from[] to[] proto[] helo[] 1751
- NOQUEUE: reject: RCPT from hostname[IP]: 554 Service unavailable; Client host [IP] blocked using hostname; Dynamic IP Addresses See: http://hostname/hostnameml?IP; from[] to[] proto[] helo[] 1792
- NOQUEUE: reject: RCPT from unknown[IP]: 450 <@>: Recipient address rejected: Greylisted for 5 minutes; from[] to[] proto[] helo[] 1836
- TLS connection established from hostname[IP]: TLSv1 with cipher DHE-RSA-AES256-SHA (256/256 bits) 1931

- too many errors after RCPT from hostname[IP] 2065
- lost connection after EHLO from hostname[IP] 2359
- NOQUEUE: reject: RCPT from unknown[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; Blocked - see http://hostname/hostnameml?IP; from[] to[] proto[] helo[] 2411
- NOQUEUE: reject: RCPT from hostname[IP]: 554 Service unavailable; Client host [IP] blocked using hostname; http://hostname/listing?IP; from[] to[] proto[] helo[] 2910
- NOQUEUE: reject: RCPT from unknown[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; Dynamic IP Addresses See: http://hostname/hostnameml?IP; from[] to[] proto[] helo[] 2995
- TLS connection established from unknown[IP]: TLSv1 with cipher DHE-RSA-AES256-SHA (256/256 bits) 3103
- setting up TLS connection from unknown[IP] 3128
- NOQUEUE: reject: RCPT from unknown[IP]: 450 <@>: Sender address rejected: Domain not found; from[] to[] proto[] helo[] 3156
- lost connection after RSET from hostname[IP] 3558
- NOQUEUE: reject: RCPT from unknown[IP]: 551 <@>: Recipient address rejected: This mailbox is no longer valid. Goodbye!; from[] to[] proto[] helo[] 3859
- NOQUEUE: reject: RCPT from unknown[IP]: 554 Service unavailable; Client host [IP] blocked using hostname; http://hostname/query/bl?ip[] from[] to[] proto[] helo[] 4496
- NOQUEUE: reject: RCPT from unknown[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; http://hostname/listing?IP; from[] to[] proto[] helo[] 4611
- NOQUEUE: reject: RCPT from unknown[IP]: 450 <@>: Recipient address rejected: Domain not found; from[] to[] proto[] helo[] 5239
- warning: IP: address not listed for hostname hostname 5378
- NOQUEUE: reject: RCPT from hostname[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; http://hostname/listing?IP; from[] to[] proto[] helo[] 5390

- NOQUEUE: reject: RCPT from hostname[IP]: 450 <@>: Recipient address rejected: Greylisted for 5 minutes; from[] to[] proto[] helo[] 5611
- warning: Connection rate limit exceeded: # from hostname[IP] for service smtp 5775
- NOQUEUE: reject: RCPT from hostname[IP]: 554 Service unavailable; Client host [IP] blocked using hostname; http://hostname/query/bl?ip[] from[] to[] proto[] helo[] 6414
- timeout after DATA from hostname[IP] 6628
- NOQUEUE: reject: RCPT from hostname[IP]: 551 <@>: Recipient address rejected: This mailbox is no longer valid. Goodbye!; from[] to[] proto[] helo[] 6975
- #Hex#: reject: RCPT from hostname[IP]: 450 4.1.2 <@>: Recipient address rejected: Domain not found; from[] to[] proto[] helo[] 7028
- NOQUEUE: reject: RCPT from hostname[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; Blocked - see http://hostname/hostnameml?IP;from[] to[] proto[] helo[] 8361
- NOQUEUE: reject: RCPT from hostname[IP]: 450 <@>: Sender address rejected: Domain not found; from[] to[] proto[] helo[] 8420
- NOQUEUE: reject: RCPT from unknown[IP]: 450 4.7.1 <@>: Recipient address rejected: Greylisted for 5 minutes; from[] to[] proto[] helo[] 8962
- lost connection after DATA from unknown[IP] 11455
- NOQUEUE: reject: RCPT from unknown[IP]: 450 4.1.8 <@>: Sender address rejected: Domain not found; from[] to[] proto[] helo[] 12342
- lost connection after CONNECT from unknown[IP] 12416
- NOQUEUE: reject: RCPT from hostname[IP]: 504 <@> to[] proto[] helo[] 13922
- lost connection after CONNECT from hostname[IP] 14587
- NOQUEUE: reject: RCPT from unknown[IP]: 504 <@> to[] proto[] helo[] 15466
- NOQUEUE: reject: RCPT from hostname[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; Dynamic IP Addresses See: http://hostname/hostnameml?IP; from[] to[] proto[] helo[] 15978
- lost connection after DATA from hostname[IP] 17201

- NOQUEUE: reject: RCPT from unknown[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; http://hostname/query/bl?ip[] from[] to[] proto[] helo[] 19497
- NOQUEUE: reject: RCPT from unknown[IP]: 551 5.7.1 <@>: Recipient address rejected: This mailbox is no longer valid. Goodbye!; from[] to[] proto[] helo[] 22551
- NOQUEUE: reject: RCPT from hostname[IP]: 450 4.1.8 <@>: Sender address rejected: Domain not found; from[] to[] proto[] helo[] 22900
- warning: Connection concurrency limit exceeded: # from hostname[IP] for service smtp 23437
- NOQUEUE: reject: RCPT from hostname[IP]: 450 4.7.1 <@>: Recipient address rejected: Greylisted for 5 minutes; from[] to[] proto[] helo[] 29384
- NOQUEUE: reject: RCPT from hostname[IP]: 554 5.7.1 Service unavailable; Client host [IP] blocked using hostname; http://hostname/query/bl?ip[] from[] to[] proto[] helo[] 33330
- NOQUEUE: reject: RCPT from hostname[IP]: 551 5.7.1 <@>: Recipient address rejected: This mailbox is no longer valid. Goodbye!; from[] to[] proto[] helo[] 36728
- NOQUEUE: reject: RCPT from unknown[IP]: 504 5.5.2 <@> to[] proto[] helo[] 44377
- warning: IP: hostname hostname verification failed: hostname nor servname provided, or not known 45577
- NOQUEUE: reject: RCPT from hostname[IP]: 504 5.5.2 <@> to[] proto[] helo[] 47996
- lost connection after RCPT from unknown[IP] 58639
- lost connection after RCPT from hostname[IP] 79175
- NOQUEUE: reject: RCPT from hostname[IP]: 450 <@>: Recipient address rejected: Domain not found; from[] to[] proto[] helo[] 129813
- disconnect from unknown[IP] 155933
- connect from unknown[IP] 156003
- TLS connection established from hostname[IP]: TLSv1 with cipher ADH-AES256-SHA (256/256 bits) 171060
- setting up TLS connection from hostname[IP] 173965

  - NOQUEUE: reject: RCPT from hostname[IP]: 450 4.1.2 <@>: Recipient address rejected: Domain not found; from[] to[] proto[] helo[] 281910
  - #Hex#: client[] 2068726
  - disconnect from hostname[IP] 2556625
  - connect from hostname[IP] 2557395

- postfix/cleanup
  - #Hex#: resent-message-id[] 1055
  - warning: #Hex#: virtual_alias_maps map lookup problem for email 2270
  - #Hex#: message-id[] 2093881

- postfix/smtp
  - connect to hostname[IP]: Permission denied (port 25) 1136
  - #Hex#: to[] orig_to[] relay[] delay[] delays[] dsn[] status[] (message) 1313
  - #Hex#: lost connection with hostname[IP] while sending message body 1390
  - #Hex#: to[] orig_to[] relay[] delay[] status[] (message) 1434
  - #Hex#: host hostname[IP] said: 451 hostname Resources temporarily unavailable. Please try again later [#4.16.5]. (in reply to end of DATA command) 1539
  - connect to hostname[IP]: server dropped connection without sending the initial SMTP greeting (port 25) 1945
  - #Hex#: host hostname[IP] said: 451 Message temporarily deferred - [170] (in reply to end of DATA command) 2128
  - connect to IP[IP]: Connection refused (port 10024) 2305
  - #Hex#: lost connection with hostname[IP] while receiving the initial server greeting 2366
  - connect to hostname[IP]: server refused to talk to me: 554- (RTR:BG) http://hostname/errors/hostna (port 25) 2392
  - certificate verification failed for hostname: num[] signed certificate in certificate chain 2593

  - #Hex#: host hostname[IP] refused to talk to me: 554 5.7.1 IP: Connection refused. Your IP address is blocked(anti-spam). 3202
  - connect to hostname[IP]: server refused to talk to me: 421 Insufficient System Storage.(IMail 7.07) (port 25) 3226
  - certificate peer name verification failed for hostname: CommonName mis-match: hostname 7032
  - connect to hostname[IP]: No route to host (port 25) 7939
  - #Hex#: host hostname[IP] said: 451 The users mailbox is full. Please try re-sending your e-mail later. (in reply to RCPT TO command) 10852
  - certificate verification failed for hostname: num[] not trusted 12723
  - certificate verification failed for hostname: num[] to verify the first certificate 13137
  - certificate verification failed for hostname: num[] to get local issuer certificate 13349
  - certificate verification failed for hostname: num[] signed certificate 17234
  - Server certificate could not be verified 17490
  - certificate verification failed for hostname: num[] has expired 18321
  - certificate verification failed for hostname:certificate has expired 36627
  - connect to hostname[IP]: Connection refused (port 25) 60401
  - connect to hostname[IP]: Operation timed out (port 25) 87784
  - #Hex#: to[] relay[] conn_use[] delay[] delays[] dsn[] status[] (message) 255658
  - #Hex#: to[] relay[] delay[] status[] (message) 694954
  - #Hex#: to[] relay[] delay[] delays[] dsn[] status[] (message) 1831814

- amavis

  - (#####-##) Passed SPAM, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} # ms 1024
  - (#####-##) (!!)ClamAV-clamd av-scanner FAILED{} Too many retries to talk to /var/run/clamav/clamd (Can't connect to UNIX socket /var/run/clamav/clamd{} refused) at (eval 68) line 293. at (eval 68) line 491. 1472

- (#####-##) (!)run_av (ClamAV-clamd, built-in i/f){} many retries to talk to /var/run/clamav/clamd (Can't connect to UNIX socket /var/run/clamav/clamd{} refused) at (eval 68) line 293. 1472
- (#####-##) (!)ClamAV-clamd{} connect to UNIX socket /var/run/clamav/clamd{} refused, retrying (2) 1487
- (#####-##) Passed CLEAN, <@> -> <@>+ Message-ID{} mail_id{} Hits{} queued_as{} # ms 2593
- (#####-##) Passed CLEAN, [IP] [IP] <@>, Message-ID{} mail_id{} Hits{} # ms 2665
- (#####-##) Blocked INFECTED (virus) [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} # ms 2844
- (#####-##-##) Passed CLEAN, [IP] [IP] <@>, Message-ID{} mail_id{} Hits{} queued_as{} # ms 3647
- (#####-##) (!!)avast! Antivirus - Client/Server Version av-scanner FAILED{} unexpected exit 35, output[] failed to scan{} connect to avast! server" at (eval 68) line 491. 6471
- (#####-##) (!!)run_av (avast! Antivirus - Client/Server Version) FAILED - unexpected exit 35, output[] failed to scan{} connect to avast! server" 6471
- (#####-##-##) Passed CLEAN, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} # ms 7014
- (#####-##) Passed BAD-HEADER, [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} queued_as{} # ms 7965
- (#####-##-##) Blocked SPAM, [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} # ms 15819
- (#####-##) Passed SPAMMY, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} queued_as{} # ms 16903
- (#####-##-##) Passed CLEAN, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} queued_as{} # ms 35688
- (#####-##) Passed CLEAN, [IP] [IP] <@>, Message-ID{} mail_id{} Hits{} queued_as{} # ms 37620
- (#####-##) Blocked SPAM, [IP] [IP] <@> -> <@>+ quarantine{} Message-ID{} mail_id{} Hits{} # ms 52013

- (#####-##) Passed CLEAN, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} # ms 99783
- (#####-##) Passed CLEAN, [IP] [IP] <@> -> <@>+ Message-ID{} mail_id{} Hits{} queued_as{} # ms 718719

Appendix D

Zero-Frequency characters

The ASCII values can be divided into three parts, the non printable characters (system codes) which are between 0 and 31, the lower ASCII values which are between 32 and 127 and the higher ASCII values which are between 128 and 255. The higher ASCII values are also known as known as Extended ASCII charcaters and are all zero-frequency characters. In the table below, the zero-frequency characters are marked in bold and ^x (where x is a character) refers to the control key sequence for a character. The only system code which is used is ASCII character 10 which is the newline character (\n).

0	^@	**13**	^M	**26**	^Z	39	’	52	4	65	A	78	N	91	[	104	h	117	u
1	^A	**14**	^N	**27**	^[	40	(	53	5	66	B	79	O	92	\	105	i	118	v
2	^B	**15**	^O	**28**	^\	41	)	54	6	67	C	80	P	93	]	106	j	119	w
3	^C	**16**	^P	**29**	^]	42	*	55	7	68	D	81	Q	**94**	^	107	k	120	x
4	^D	**17**	^Q	**30**	^^	43	+	56	8	69	E	82	R	95	_	108	l	121	y
5	^E	**18**	^R	**31**	^_	44	,	57	9	70	F	83	S	**96**	‘	109	m	122	z
6	^F	**19**	^S	32	‘	45	-	58	:	71	G	84	T	97	a	110	n	123	{
7	^G	**20**	^T	33	!	46	.	59	;	72	H	85	U	98	b	111	o	124	|
8	^H	**21**	^U	34	“	47	/	60	<	73	I	86	V	99	c	112	p	125	}
9	^I	**22**	^V	35	#	48	0	61	=	74	J	87	W	100	d	113	q	126	~
10	^J	**23**	^W	36	$	49	1	62	>	75	K	88	X	101	e	114	r	**127**	^?
11	^K	**24**	^X	37	%	50	2	63	?	76	L	89	Y	102	f	115	s		
12	^L	**25**	^Y	38	&	51	3	64	@	77	M	90	Z	103	g	116	t		

Appendix E

Electronic Appendix

The attached DVD contains the following:

E.1 Scripts

There are a number of scripts which were used in this thesis. The **Scripts** folder contains all the different scripts which are used.

E.1.1 Compression tests

The scripts for compression tests are contained in the **Compression and Decompression Time** folder.

E.1.2 Memory Utilisation tests

The scripts for the memory utilisation tests are contained in the **Memory Utilisation** folder.

E.1.3 Date and IP preprocessors

The scripts for the preprocessors which transform the date and IP are located in the **Date and IP** folder.

E.1.4 Analysis of syslog message semantic

The scripts for the analysing the syslog message sematics are located in the **Analyse syslog** folder.

E.1.5 Dictionary generation

Three different scripts were used for dictionary generation. These scripts are contained in the **Word Replacement** folder.

E.2 Detailed Results

Detailed Results are contained in the **Results** folder. Within this folder, each chapter has a separate folder which contain the full results from the compression tests and memory tests.

E.3 Analysis of Results for different compression levels

Each compression level of the seven compression programs achieves different results and the preprocessors cause a different amounts of improvement in compression ratio, compression time, decompression time, transfer time and memory utilisation. **CompressionLevels.pdf** contains graphs and detailed analysis of the results for the different compression levels of the seven compression programs.

E.4 Electronic References and Web References

Electronic copies of the references and web reference used can be found in the **References** folder. A web interface (index.html) which lists the references and contains links to the files can also be found in this folder.