UiT The Arctic University of Norway

The Faculty of Science and Technology
Department of Computer Science

Improving the text compression ratio for ASCII text

Using a combination of dictionary coding, ASCII compression, and Huffman coding

Sondre Haldar-Iversen

INF-3990 Master's thesis in Computer Science – November 2020

Improving the text compression ratio for ASCII text

Using a combination of dictionary coding, ASCII compression, and Huffman coding

Sondre Haldar-Iversen

Abstract

Data compression is a field that has been extensively researched. Many compression algorithms in use today have been around for several decades, like *Huffman Coding* and *dictionary coding*. These are general-purpose compression algorithms and can be used on anything from text data to images and video. There are, however, much fewer lossless algorithms that are customized for compressing certain types of data, like ASCII text. This project is about creating a text-compression solution using a combination of the three compression algorithms dictionary coding, ASCII compression, and Huffman coding. The solution is customized for compressing ASCII text, but it can be used on any form of text. The algorithms will be combined to create a prototype that will be evaluated against general-purpose compression programs. An evaluation will also be made against the ASCII compression program Shoco. The results from the evaluation show that combining dictionary coding, ASCII compression, and Huffman coding does not surpass the compression ratio achieved from general-purpose compression programs.

Keywords
Text compression, Dictionary coding, ASCII, Huffman coding

Table of Contents

1 Introduction

Data compression is the process of reducing the number of bits required to represent media [1]. In other words, reducing the size of data while still keeping its integrity. Data compression can save storage space or lower the bandwidth e.g. when streaming video [2]. A subset of data compression is text compression. Text compression is the compression of text files, which are files that contain characters and symbols. One important difference between compressing text and binary data, like images and video, [1] is that text compression has to be lossless. Lossless compression means that a compressed file can be decompressed to retrieve the original file without any loss of data [1]. Compression algorithms that are not able to decompress data use lossy compression. Lossy compression is mostly used on binary data where some of the data can be lost [3].

1.1 Background

Data compression is handled by compression algorithms. There are several common algorithms used by compression programs [4, 5, 6], and many of which have been around for several decades. A popular algorithm still in use today is Huffman coding [7], which was invented in 1952. Huffman coding compresses text by having frequently used characters use less space than rarely used characters. An English text might therefore achieve a good compression ratio because, e.g. the letter “e” is used more often than “z” [8]. Another algorithm is dictionary coding, which is the technique of replacing recurring substrings and words with references to a dictionary. The dictionary and references make up the compressed file, which can be decompressed to retrieve the original file. LZ77 and LZW are some of the most used dictionary coding algorithms [4, 5, 6, 9]. Both algorithms are based on the technique of replacing repeating substrings with references, but they use different approaches. The DEFLATE algorithm is a combination of dictionary coding (LZ77) [10] and Huffman coding [11]. DEFLATE is used in several compression program, like Gzip, Zip, and WinRar [4, 5, 6]. Even though the field of data compression has been extensively researched, new technologies are still emerging [12, 13, 14], like Shoco [15], which was released in 2014 [16]. Shoco is a text compression program that is particularly effective on ASCII text [15]. ASCII is a character encoding for simple text; that is, text that does not use a large variety of characters and symbols. The character encoding of a text tells the computer how the data must be read so that humans can understand it [17].

1.2 The problem

Transferring large amounts of data over the internet can be a time-consuming task. In situations where a user has excellent bandwidth, transferring megabytes or gigabytes of data can still take several minutes. This problem is often related to binary files, like videos, but it can also be the case for text-based data. One example can be cloning a Git repository, e.g. the Linux kernel GitHub repository, [18] which is close to 3 GB in size [19]. Text data can be compressed using algorithms like DEFLATE, which is used by several popular compression programs [4, 5, 6]. Compressing the data reduces its file size, which means

transferring it over the internet will be faster. The DEFLATE algorithm is designed to work on all kinds of texts and binary data [1]. However, this is a general-purpose compression algorithm that can be optimized for working with e.g. ASCII text. Most compression programs are general-purpose, and use lossless algorithms that will achieve compression on any kind of data, like text, image, and video files [20, 4, 5, 6]. There exists several lossy compression algorithms that are specialized for specific data types, but not as many lossless algorithms [20]. It is possible that using lossless algorithms customized for compressing specific data types, like ASCII text, will achieve a higher compression ratio than general-purpose algorithms.

The thesis answers the following research questions:

- "What combination of techniques can improve the compression ratio for ASCII text?"
- "Does using ASCII compression in combination with dictionary coding and Huffman coding surpass the compression ratio of DEFLATE?"
- "Will this text compression solution achieve a higher compression ratio than general-purpose compression programs for ASCII text?"
- "Will this text compression solution achieve a higher compression ratio than compression solutions that specialize on ASCII text?"

1.3 Purpose and goal

Most compression programs use the DEFLATE algorithm [4, 5, 6], which includes dictionary coding and Huffman coding. This project will also use these algorithms, as they are de facto standards and have proven to be effective [21, 22]. The following algorithms will be used in this project:

- Dictionary coding; Particularly effective on written text[1], like an encyclopedia [23].
- ASCII compression; Also particularly effective on written text, where certain characters are used more often than others [8].
- Huffman coding; A technique that has been used extensively in data compression [24, 25].

This project makes a solution that is optimal for compressing ASCII text, using these three methods, and attempts to have a higher compression ratio for ASCII text than a general-purpose compression algorithm.

[1] «Written text» is defined in this thesis as text that is written in some language, like a spoken language, or a programming language.

1.4 Motivation

The proposed text compression solution (which will hereby be referred to as TCS or "the solution") can be useful for anyone who happen to be dealing with large amounts of (mainly) ASCII text. The solution is designed to work on any form of text, but it is customized for ASCII text. The solution decreases the file size so that it does not take up too much of the storage space. This will also decrease the bandwidth when a compressed file is downloaded or transferred over the internet.

Dictionary coding [11, 26] and Huffman coding [11] have already been extensively used in compression algorithms [4, 5, 6] (both are used in DEFLATE), but few compression solutions make use of ASCII compression methods [4, 5, 6, 27]. This solution might therefore get a better compression ratio for ASCII text than existing general-purpose compression algorithms.

1.5 Methods

Quantitative research is statistical and deals with numerical data, whereas qualitative research is investigative and usually non-statistical [28]. The experiments for the system use numerical data when comparing the TCS to other methods, therefore, quantitative research gives better measurements for comparison, as opposed to qualitative research. Logical reasoning is needed to draw a conclusion from the experiments [29] [30]. Examples of reasonings can be deductive reasoning or inductive reasoning [29]. Deductive reasoning is the forming of a conclusion based on generally accepted statements or facts [30]. It follows the rationale of: If A = B and B = C then A = C. Inductive reasoning involves an element of probability, and might come to the conclusion that A is probably equal to B [30].

In this project, a number of compression techniques are tested on the same texts, and the compression ratio is measured to find which technique achieves the most compression. The experiments give definitive data on which solution achieves the best compression ratio, and therefore a conclusion is drawn with deductive reasoning.

1.6 Delimitations

This project is intended specifically for compressing text, and not binary data, such as image or video files. The experiments in this project are therefore only tested on text data. The solution is expected to be used for compressing large amounts of text because text data usually does not take up much storage space, compared to e.g. video data, and compressing a small text file is of little use. Therefore, the experimentation will only test text files of a significant size. The effectiveness of the solution (the latency) is not a factor in the evaluation, because the solution is only evaluated on its compression ratio, not its runtime length. Python is chosen as the programming language for the ASCII compression module (ACM) to make the source code simpler to understand. Implementing the program in C could have made the solution faster [104] [105], but compression capability is a higher priority than latency for the ACM.

The project will evaluate different implementations to be used in the TCS. The implementations will be evaluated based on their average result from the experiments. This means that every use case in the experiment is of equal importance. The project does not encounter any ethical dilemmas. The project is not a collaboration with a company, so there will be no problems with copyrights. The project only uses source code with an open license. The experiments on the Wikipedia files (see section 4.1.1) only uses a subset of the text for testing compression, as it is unfeasible to use such a large dataset for testing. The subset is sufficiently big enough to give clear results.

1.7 Outline

Chapter two of this thesis describes what text compression is and presents the background and related work for the three modules of the TCS; dictionary coding, ASCII compression, and Huffman coding. Chapter three presents the different methods that are used in the academic research for this project, and why those methods are chosen. Chapter four presents the design of the TCS and the requirements for the solution. Chapter five describes the evaluation process for the TCS and its modules, and presents the results from the evaluations. Chapter six covers the discussion of the results from the evaluations, and chapter seven provides the conclusion and future work of the thesis.

2 Data compression

Data compression is the process of reducing the number of bits required to represent media [1]. This means a compressed file will be smaller than the original file and can e.g. be transferred over the internet in a shorter amount of time. There are two types of data compression techniques: lossless and lossy. Lossless compression means that all the data can be recovered when it is decompressed [1]. This means that after compressing and then decompressing a file using lossless compression, it will be completely identical to its original state. Lossy compression, on the other hand, purposefully removes some of the data during the compression process [3]. The data that is lost cannot be recovered when the file is decompressed. Lossy compression is mostly used on binary data [3] in combination with lossless compression [1]. Binary data can e.g. be audio, image, and video files [1]. The advantage of lossy compression is that it usually achieves a much higher compression ratio than lossless compression, while users can still comprehend the information of the compressed data [3]. Often, the data lost in compression is not noticeable to the user [31].

Huffman coding and dictionary coding are examples of lossless compression algorithms. These algorithms are used in many compression programs, like Gzip, Zip, or Winrar [4, 5, 6]. What is common for these algorithms is that they get the highest compression ratio from files that repeat data or has an uneven distribution of characters/symbols. This means that random data or a "gibberish" text might get a small compression ratio. The compression ratio is the measurement of how much smaller the compressed file is compared to the original file [32]. For example, these algorithms are effective when compressing text written in some language, like English, German, or programming languages. This is because some words, phrases, and characters are more frequently used than others [32]. These algorithms are not only used on text. The PNG format uses these techniques to find matches and patterns in images [33].

2.1 Dictionary coding

Dictionary coding is a compression technique that replaces repeating strings of text with references. While Huffman coding looks at individual characters, and how frequently they are used in a text, dictionary coding looks at frequently used strings [23]. For example, if a text contains the word "compression" several times, the instances of the word can be replaced with a reference to the word in a dictionary. When the text is decompressed, the algorithm uses the reference to look up the word and replaces the reference with the original word. A dictionary is a list of strings of text that are frequently used [34]. Some dictionary coders, like LZ77 [10], does not use an explicit dictionary, but instead uses references to a prior occurrence in the text [35]. Dictionary coders can also be used on non-text data, like images. The PNG image format uses the LZ77 dictionary coder [33]. In the case of images, the algorithm would be searching for repeating sequences of pixels instead of text strings.

The advantage of using a dictionary coder is that it can achieve a very high compression ratio for certain files. Dictionary coders are particularly effective on repetitive data, like written text [23]. Text, written in a language, typically repeats some words or combinations of letters more often than others, and dictionary coders take advantage of this. The disadvantage of using some dictionary coders, like LZW, is that the compressed file may be bigger than the uncompressed file [36] [37]. This can happen if the data to be compressed does not repeat any information. If compressing a file does not decrease its size, then the purpose is defeated.

2.1.1 Common dictionary coding algorithms

This section describes the main characteristics and differences between popular dictionary coding algorithms. The characteristics of an algorithm can be its technical solution or what separates it from the others. An algorithm's characteristics can also mean it is more or less suitable for certain compression tasks.

LZ77

LZ77 is a popular dictionary coder used in, among other things, the PNG image format [33] and the DEFLATE algorithm [11]. The LZ77 algorithm uses a *sliding window* [10] when compressing data (e.g. an image file or a text file). The sliding window is a buffer that determines how much of the text (or data) the algorithm will analyze at any given point to find matching substrings. If a text file is being compressed, then a substring is a group of characters in a row. The sliding window consists of a *look-ahead* buffer and a *search* buffer. The look-ahead buffer analyzes the text that has not yet been compressed/encoded, e.g. the next 20 characters, while the search buffer is the recently encoded text and is usually much bigger than the look-ahead buffer. If a substring in the search buffer also appears in the look-ahead buffer, then the substring in the look-ahead buffer is replaced with a reference to a previous occurrence in the search buffer [10].

LZ78

While the LZ77 algorithm creates references to a previous point in the text for recurring substrings, LZ78 uses an explicit dictionary. The LZ78 algorithm creates a dictionary of substrings and replaces substrings in a text with references to the dictionary [38]. The LZ78 algorithm is a revision of the LZ77 algorithm, and is mostly similar, except for the explicit dictionary [39].

LZW

The LZW algorithm is a variant of LZ78 and also uses an explicit dictionary [35]. The basic premise of LZW is that it starts the encoding process with 256 entries/characters in the dictionary [37]. This is usually the ASCII character set, which uses one byte per character. The algorithm constantly appends substrings to the dictionary with a reference higher than 256. These entries may or may not be recurring in the text. If a substring is recurring, then the same reference to the substring is used [35].

Re-pair
The Re-pair algorithm uses an explicit dictionary and is similar to the LZ78 algorithm. How Re-pair differs from LZ78 is that Re-pair only stores two-character substrings in the dictionary, while LZ78 stores variable length substrings [38]. Re-pair finds all character pairs that occur more than once in a text and replaces them with a reference to the dictionary. This continues until there are no character pairs that occur more than once in the text [40].

2.2 ASCII compression

ASCII compression is defined in this thesis as compression algorithms that are particularly effective on text with ASCII encoding. ASCII has many variations for different languages, but this thesis will focus on US-ASCII, unless stated otherwise. US-ASCII is the preferred ASCII-encoding for internet communication, and "US-ASCII" is today synonymous with "ASCII" [44] [45]. ASCII encoding is a widely used text encoding [46] [45] for text that does not use a large variety of characters and symbols. This limitation in the number of symbols stems from how the text is stored on a computer. One symbol uses exactly one byte (8 bits) of storage, where 7 bits are the code for the symbol and 1 bit is the "most significant bit". The purpose of the most significant bit can be used as a parity check, to detect errors in data [47]. There is also a group of encodings known as "extended ASCII" which utilize all 8 bits for the symbol code, and thereby leave out the most significant bit. The encodings that use all 8 bits had 256 possible symbols, instead of standard ASCII's 128 symbols [48].

2.2.1 Shoco

Shoco is a text compression algorithm developed by Christian Schramm [16]. It is described as "A fast compressor for short strings" and is particularly effective on ASCII text. Shoco's compression solution takes advantage of the fact that in every language, some characters are used more often than others. As previously mentioned, the first bit in an ASCII character is redundant, unless it is used for purposes like error detection. Therefore, the algorithm uses the first bit in a byte to indicate whether the following bits refer to a common character or not. If the first bit is set to 0, it means the following 7 bits represent an uncommon character. The ASCII character will then be unaltered when compressed. If the first bit is set to 1, it means the following bits represent two common characters. The two characters will then be represented by 3 or 4 bits. All together, this will use 8 bits and can therefore be stored as a single byte.

Shoco is also able to compress non-ASCII characters, but it comes at a cost; "If your input string is not entirely (or mostly) ASCII, the output may grow" [15]. When the algorithm encounters a character that uses more than 7 bits (e.g. a UTF-8 character), it inserts a "marker" right before the character. The marker is a special character that takes up a byte. Its purpose is to signal that the next character is not ASCII and will therefore use more than 7 bits. The marker also tells how many bytes the next character uses (e.g. 1 or 2 bytes) [15].

A technique that Shoco uses is counting the frequency of bigrams in a text. Bigrams, in the context of Shoco, are unique pairs of two successive characters in a text. A common bigram in English is “qu”, as “q” is almost always followed by “u” in the English language. After Shoco has made a list of the most common characters in a text, it makes another list of which characters that are most likely to follow those common characters. If Shoco finds that e.g. “he” is a common bigram then all words containing this bigram can be compressed, like “the”, “she”, and “then” [15].

The advantage of using Shoco is that the compressed file will never be bigger than the uncompressed file, as long as the input is 100 % ASCII. Furthermore, Shoco can easily be used in combination with other compression algorithms, like dictionary coding algorithms, or Huffman coding to achieve an even greater compression ratio. Shoco is also a very fast algorithm. According to the website [15], Shoco is almost 7 times faster than Gzip when compressing a file of 4.9 megabytes. The disadvantage with Shoco is that using Shoco alone will give a much worse compression ratio than standard compression programs. In addition, the user also has to know of what type of text they are compressing. If the text contains a large amount of multi-byte characters, like UTF-8, then the compressed file may be bigger than the uncompressed file [15], which is highly undesirable when using a compression tool.

2.3 Huffman coding

Huffman coding is designed to represent the most used characters in a text with as few bits as possible. First, the algorithm analyses the input by traversing the text, character for character. The algorithm then builds a list of every character used in the text and orders the list by frequency [2]. After building the list, the algorithm then constructs a binary tree [49] based on the table. A binary tree is a data structure that consists of “nodes” and “branches”. Each node can have at most two “child nodes”. In figure 1, the nodes are the blue rounded rectangles. The “leaf nodes” are the last nodes in a branch. Think of a binary tree as a tree that is upside down. All the branches in a tree start from the same trunk, and the leaves are at the end of the branches. Each leaf node holds the value of one of the characters used in the text [2].

Once a Huffman tree is constructed, the compression of the text begins. The algorithm traverses the tree for every character in a text. To reach a leaf node the algorithm moves to the left or right child node until the specified character is found. This traversal is represented in binary with ones and zeros, where 1 means “go to the right” and 0 means “go to the left”. In order to decompress a text to retrieve the original data, the Huffman tree also has to be stored. This is why Huffman coding is more effective on larger texts; For example when there is a large text that only consists of the 26 letters in the English alphabet and a few punctuation marks, the tree would have just over 26 leaf nodes regardless of how large the text is [2].

Huffman coding is an entropy coder, meaning it compresses data based on symbol frequency. This is in contrast to dictionary coding which looks at the frequencies of substrings – or sequences of symbols [50]. Entropy coders like

Huffman coding are advantageous when the data uses certain symbols more often than others. Huffman coding can easily be used on both text data and binary data, and can be used in combination with other compression algorithms, like dictionary coding [51] [11]. Huffman coding is often compared to arithmetic coding, which is another type of entropy coder. Results from such comparisons show that Huffman coding is faster than arithmetic coding, but that arithmetic coding generally gets a better compression ratio [52] [53].

2.3.1 The compression process

This section will explain the compression process for Huffman coding. An example input string for the algorithm can be "this is an example of a huffman tree". Table 1 is a frequency table of every character in the example sentence.

Character	Frequency
[space]	7
a	4
e	4
f	3
h	2
i	2
m	2
n	2
s	2
t	2
l	1
o	1
p	1
r	1
u	1
x	1

Table 1. Character frequency example

From this table we can see that the space character is the most frequent, followed by "a" and "e" with four occurrences respectively. Figure 1 shows how the example sentence would look as a Huffman binary tree.

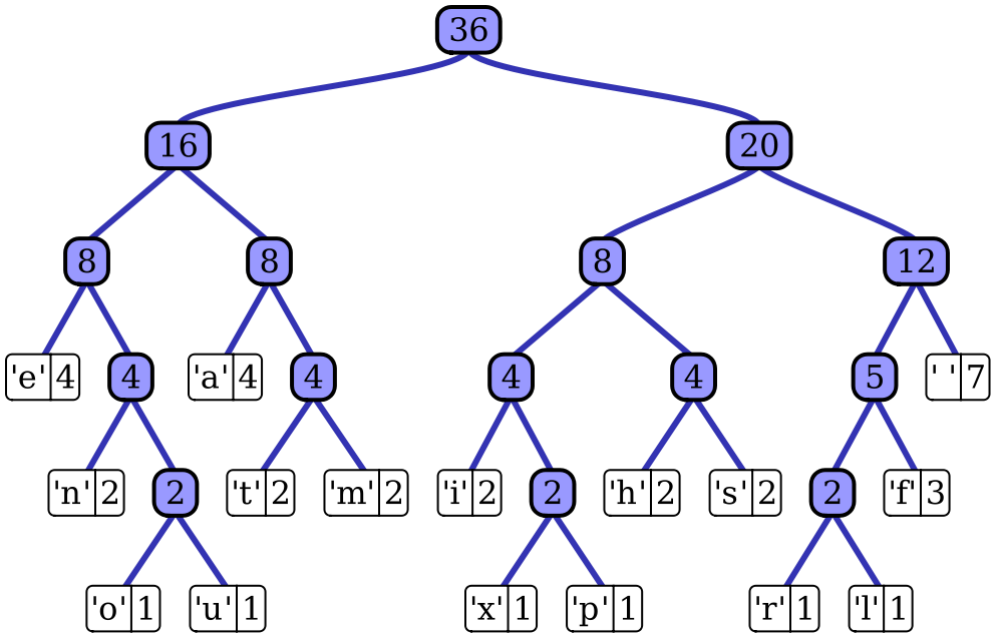

Figure 1. Huffman binary tree. Source: [54]

The nodes in this figure all have numbers that represent how many times a character in the child nodes have been used. This number is important when constructing Huffman trees because the most used characters should be as far up the tree (as close to the root node) as possible. Table 1 shows that space, “a”, and “e” are the most used characters, and therefore they are close to the root node. These three characters can be reached after three traversals from the root node.

The example sentence “this is an example of a huffman tree”, stored with ASCII character encoding, would use 36 bytes of storage. This is how large the file is before compression, where every character uses one byte (or 8 bits) of storage. When the compression starts, it begins with the first character, “t”. To reach the leaf node that holds the value “t” from the root node, the algorithm first has to go to the left child node, then twice to the right child node, and finally to the left. The compressed binary representation of “t” will therefore be “0110”, which is only 4 bits. The algorithm does this for every character in the text, and when it is finished it will end up with a binary stream of 135 bits. This stream would only use 17 bytes of storage on a computer, which is less than half of the uncompressed text. Granted, the Huffman tree would also have to be stored in order to decompress the text.

2.4 Related work

This section will discuss papers and solutions related to text compression or, more specifically, ASCII compression, as the presented solution in this thesis is a text compressor that is customized for compressing ASCII text.

2.4.1 Boosting Text Compression with Word-Based Statistical Encoding

This is a paper from the 2012 edition of *The Computer Journal* [55] written by Antonio Fariña, Gonzalo Navarro, and José R. Paramá [13]. The paper presents a possible improvement in the compression capabilities of text compression programs. The goal of the proposed solution is to increase the compression ratio and decrease the compression time for text compressors. The solution uses word-based and byte-oriented compression techniques as preprocessors to generic compression programs [13].

Word-based compression is a form of dictionary coding where the algorithm searches for whole words [56]. Most dictionary coders, like the Lempel-Ziv algorithms, search for recurring substrings [10] [38]. Substrings can be of any length and can be a part of a word. Word-based compression will only add whole words to its dictionary. The paper states that using word-based compression, as opposed to standard dictionary coders, gives better compression and decompression times. In addition, it makes it possible to search for words and phrases on the compressed file without having to decompress it [13].

Byte-oriented compression is the second preprocessor that the proposed solution uses. Byte-oriented compression is a technique that only uses whole bytes, not bits, when writing compression codes [57]. Conversely, standard Huffman coding is a compression technique that uses sequences of bits as codes to navigate a Huffman tree [2]. Huffman coding therefore needs to be read as sequences of bits, rather than bytes. This can increase latency, as data is stored and read as bytes on computers. The proposed solution uses a byte-oriented algorithm called Tagged Huffman [58]. Tagged Huffman uses the same compression technique as standard Huffman coding, but every code to look up a Huffman tree uses a set number of bytes. According to the paper, using Tagged Huffman instead of standard Huffman coding decreases latency, but also decreases the compression ratio with around 0.6, which they accepted as a trade-off [13].

The results from the experiments show that a generic compression program is up to 5 times faster at compressing a preprocessed file, compared to the original file. This is, however, not including the time it takes for the preprocessor to compress. Comparisons when using the preprocessor before a generic compressor (among others: Gzip, Bzip2, 7-Zip, Re-pair) show that the final compressed files were between 0.5 – 10 % smaller in size compared to just using generic compressors [13].

Comparison with the TCS
The Fariña et al. solution (FS) and the TCS are both text compressors that achieves higher compression ratios for text written in some language. The word-based compression technique of the FS achieves compression because languages use certain words more often than others. The TCS also uses the characteristics of languages to increase the compression ratio; The ASCII compression module (ACM) uses a compression model (described in section 4.4.1) based on the frequency of letters in the English language, and the dictionary coding module achieves compression when a text has repeating combinations of characters.

The goal of the FS is to increase the compression ratio and decrease the latency for compression programs. Both the word-based and byte-oriented techniques are intended to decrease the compression time [13]. The TCS, however, only prioritizes compression ratio, not speed.

2.4.2 Online compression of ASCII files

This is a paper from the 2004 International Conference on Information Technology: Coding and Computing [59] written by John Istle, Pamela Mandelbaum, and Emma Regentova [60]. The core principle of the proposed solution is to give shorter codes to more frequent bigrams in an English text and consequently achieve compression. The paper claims that compression should occur because certain combinations of letters are more frequent than other in the English language. The solution will only work on ASCII-encoded texts and not any other encodings [60].

The algorithm works by using 28 static dictionaries. Static dictionaries mean they are not generated for a use case by analyzing a specific text, but instead they are a generic set of dictionaries that should achieve compression – especially for English texts. One of the dictionaries is used for numbers, another is used for punctuation marks and special characters, and the remaining 26 dictionaries are used for every letter in the alphabet. If a text to be compressed contains the letter “a” then the algorithm will look up the dictionary for that specific letter and find at what index the next letter is positioned. For example, the dictionary for the letter “a” will have the letter “t” as the first index, because “t” is the letter that most frequently follows “a”. If the text contained the letter “t” it would be replaced with the number 1 [60].

For every new word in a text, the algorithm starts by looking up where the first letter or symbol is positioned in the “default dictionary”. The default dictionary is used after punctuation marks or special characters. If the word to be compressed is “hat” then “h” would be index 13 in the default dictionary, “a” would be index 2 in the dictionary for the letter “h”, and “t” would be index 1 in the dictionary for the letter “a”. The numbers 13, 2, and 1 are then translated into bit sequences [60]. Shorter bit sequences will use less than one byte for each character, as opposed to uncompressed ASCII characters which use 1 byte for each character. Replacing the ASCII characters with bit sequences achieves compression.

The experiments conducted in the paper compared the results from the proposed solution with the DEFLATE-based WinZip compression program [61]. A file of "less than 100 KB" and "up to 1 MB" were used for the comparison [60]. The proposed solution achieved a compression ratio of 1.67 for both files, while WinZip achieved a ratio of 2.5 for the larger file and achieved a ratio of less than 1 for the smaller file. From this result, they concluded that "the multiple dictionary technique is consistent on compressing files of any size" [60].

Comparison with the ASCII compression module

There are several similarities between this multi-dictionary solution and the ACM of the TCS. Both solutions can only compress text files, both are intended for ASCII text, and both use static dictionaries that are customized for English texts. The multi-dictionary solution uses a technique on par with Huffman coding. Huffman coding gives shorter codes to frequent characters, while the multi-dictionary solution gives shorter codes to frequent bigrams. The ACM does not convert characters to codes, but instead uses a technique called merging, which will be described in section 4.4.1. Another difference is that the ACM can compress text of different encodings, while the multi-dictionary solution can not. In the paper they stated this as a future work [60].

3 Methods and methodologies

This chapter describes and compares different methods and methodologies and explains which methods were chosen for this project, and why. The chapter presents different academic methods that are used for research projects, as well as methods and models used specifically for software development.

3.1 Research methods

There are several research methods that can be applied to research projects. Research methods are procedures for accomplishing research tasks, and explains how the research is done. Common research methods include experimental, descriptive, and fundamental. The experimental approach looks at causes, effects, and variables. Experiments are performed and the results are analyzed. The descriptive research method describes characteristics for a situation, but not its causes. The descriptive approach often uses surveys or case studies. The fundamental research method is innovative, and generates new ideas, principles, and theories. Fundamental research is curiosity driven and is about observing a phenomenon [62]. This project will use the experimental research method, as experimentation is a vital part of the project. The experimentation will look at relationships between variables and how changing the variables will affect the results.

3.2 Research strategy

A research strategy is a more defined approach to how the research will be conducted, whereas the research method is the framework for the research. A research strategy is a guideline, or a methodology. A research strategy can be experimental, just like the experimental research method. The experimental approach aims at controlling all factors that may affect the result of an experiment. The strategy verifies or falsifies a hypothesis, based on the results from an experiment. Another common research strategy is using surveys or questionnaires. This strategy involves collecting information from people. The surveys can go in depth, and may only involve a few people, which is the qualitative method, or the surveys can be designed to analyze data from a larger amount of people, which is the quantitative method [62]. This project will use the experimental research strategy for the same reason it will use the experimental research method.

3.3 Data collection

Data collection methods are used to collect data for the research. This thesis uses quantitative research, and so, the most suitable data collection methods for quantitative research projects are experiments, questionnaires, case studies, and observations. Experiments collects a large data set that are used for variables. Questionnaires collects data through quantitative or qualitative questions. Case studies are in-depth analysis of one or more participants. Observations examines behavior with focus on situations and culture [62]. This project will use experiments for data collection, as it will use the experimental research strategy.

3.4 Data analysis

For quantitative research, the most common data analysis methods are statistics and computational mathematics. Statistics are inferential and includes calculating results for statistical samples, as well as evaluating the significance of the results. Computational mathematics involves calculating numerical methods, modeling, and use of algorithms [62]. Computational mathematics use computer code to analyze data, as opposed to statistics, which can be analyzed by people. The results from the experiments of this thesis will give numerical data. The analysis of this data involves comparing the data. Computational mathematics are not needed for the analysis of the data; therefore, statistics will be the data analysis method for this project.

3.5 Quality assurance

Quality assurance is the validation and verification of the research. The data has to be reliable, valid, replicable, and ethical in order to be used in the research. Reliability refers to the consistency and stability of the measurements. This means that different measurements should give reasonably expected results, when considering all the variables in the experiment. The reliability is assessed in every test conducted in this project.
Validity is the assurance that the instruments in an experiment are actually measuring what is expected to be measured. Validity is maintained in this project by evaluating the steps and instruments needed to conduct the experiments.
Replicability makes sure that a test will give the same results when it is repeated with the exact same variables. In this project, all the tests will be repeated to check if the results are identical.
The research also has to be ethical. Ethics covers the moral principles of planning, conducting, and reporting results from research, as well as maintenance of privacy and treating material with confidentiality [62]. These ethical aspects will be assessed in the research of this project.

3.6 Software development methods

A software development method is a plan (or a set of guidelines) for developing a system from conception to implementation. The advantages of using such a method include: it helps to understand the process life-cycle, it enforces a structured approach to development, and it enables planning of resources that are to be used in a project [63].
The waterfall model uses sequential, linear phases, where each phase is a continuation of the previous phase. The phases in the Waterfall model are: feasibility study (understanding the problem), requirements and specifications, designing the solution, coding and module testing, system testing, delivery, and maintenance [63].
The prototype method creates incomplete versions of a product, called prototypes, during development. The advantage of using prototypes is that developers can get feedback from users while the product is being developed, and possible changes that need to be done can more easily be made. Prototypes can also give developers a better overview on what parts of the product is

unfinished, which can give a better estimate on when the product might be finished [64].
The agile model is an adaptive and flexible approach that focuses on early and continuous delivery of software to the client. While the waterfall model is linear and consistent, the agile model is flexible and capable of change. The agile model has defined 12 principles that make up the “manifesto” for agile development [65].

All of these methods are typically organized for teams of developers where a product is implemented for a client. This thesis is a solo project and does not create a product for a client, therefore, it would be unnecessary and/or impossible to follow all the principles associated with these development methods. Instead, the general idea behind the methods will be considered.
The prototype method creates prototypes that are intended to be tested while the product is being developed. This project consists of three modules that are simplistic enough that creating prototypes would not be necessary. Instead the software is tested while it is being developed. This project will therefore not use the prototype method. The flexibility of the agile model is better suited for this project than the waterfall model because this project does not have any phases that are dependent on previous phases to be completed. This project will therefore use the general idea behind the agile model.

4 Design and requirements

This chapter describes the requirements for the system as a whole, as well as requirements for the data set and the implementations used in the TCS. The chapter also describes the overall design of the TCS, and the technical solution of the ASCII compression module (ACM).

4.1 System requirements

The system requirements are the functional and non-functional requirements that need to be fulfilled in order for the TCS to be usable and robust. The requirements are defined based on the intended use cases for the TCS. The functional requirements describe services the solution should provide, and how the solution should behave in particular situations. The non-functional requirements are constraints on the services that a solution provides. Non-functional requirements may focus on performance, reliability, and usability [66].

These are the functional requirements for the TCS:

- **No data will be lost during compression**

The TCS uses lossless compression, which means that the original data has to be retrievable after the compression process. If data is lost during compression, then the solution is of no use.

- **The TCS can compress text with common encodings**

Common encodings refer to UTF-8 and ASCII encoding. The TCS has to be able to handle the most common text encodings in order to be a useful solution. 4 files with UTF-8 encoding and 1 file with ASCII encoding will be used to test this requirement.

- **The TCS can compress text with uncommon encodings**

If the TCS is able to compress text with encodings that are not often used, in addition to common encodings, then the solution is sturdy and can handle a larger variety of use cases. A file with cp037 encoding will be used to test this requirement.

- **The modules of the TCS will be loosely coupled**

The three modules of the TCS will be independent programs that *can* be used in combination with each other. The modules need to be loosely coupled in order to be evaluated individually.

These are the non-functional requirements for the TCS:

- **The TCS can compress files that are over 20 MB**

A compression solution has to be able to compress larger files, because compression is usually performed on larger files. The reduction of the file size has more of an impact on larger files than on smaller files. An XML file of 21.6 MB will be used in the evaluation to test this requirement.

- **The TCS can compress files to at least half of its original size**

Several tests will be conducted on each of the texts, and if at least one of the tests gets a compression ratio of at least 2 then the requirement is met.

- **All third-party code used in the TCS has to be open source**

Third-party code used in the TCS has to have an open license that allows for modifications. Modifications to the code might be necessary in order to integrate the modules into one solution.

4.2 Data set characteristics

The data set is defined in this thesis as a collection of different texts that the TCS is using when evaluating its compression ability. These texts use different encodings, e.g. UTF-8 and ASCII, which are typical encodings that a user might employ. UTF-8 is a very common character encoding that include a large variety of symbols [67]. ASCII is used for text that does not use a large variety of characters and symbols and is usually written in English [68]. The different encodings are used to evaluate the difference in compression, and what type of encoding is optimal for the solution. The texts will use the encodings ASCII, cp037, and UTF-8.

The data set also has to be representative for the type of text that the solution can be used for. A typical compression use case can be a book or an encyclopedia. This type of text would typically be encoded with UTF-8 or some other encoding that includes a variety of symbols because ASCII (or other 7/8-bit character encodings) is very limited. Another use case can be programming code, or text data like XML [69] or JSON [70]. This type of text can be encoded in ASCII [71].

4.2.1 Defining the data set

The data set used in the evaluation are examples of larger text files that a user might apply the TCS on. The data set consists of 6 texts with different characteristics;

- A Wikipedia XML file. UTF-8 encoding; 21.6 MB [72]
- A Wikipedia XML file. cp037 encoding; 18.1 MB [72]
- A code file written in C. ASCII encoding; 64 KB [73]
- A book written in English. UTF-8 encoding; 680 KB [74]
- A book written in Italian. UTF-8 encoding; 626 KB [75]
- A book written in Chinese. UTF-8 encoding; 285 KB [76]

The Wikipedia XML file is a local, text-only version of a selection of articles (individual pages) from Wikipedia. The XML file mostly contains ASCII characters, meaning most of the characters in the text can be encoded in ASCII. The data set also includes the same XML file encoded with cp037. cp037 is the encoding of a file after it has been compressed with the ASCII compression module. This text is included to test if the other two modules of the solution (Huffman coding and dictionary coding) are compatible with the ACM. The C code file is taken from the Linux operating system GitHub repository, and

contains only ASCII characters. The three books are included to test if there is a difference between structured code language and natural language. The English book contains mostly ASCII characters, the Italian book also contains mostly ASCII characters, but uses several characters that are not included in the ASCII encoding scheme, and the Chinese book does not include any ASCII characters. The different texts have very diverse sizes, with some being a couple kilobytes large and other being a few megabytes. Regardless, the texts should have sufficient sizes to properly evaluate the TCS.

4.3 Implementation requirements

The TCS consists of three modules; ACM, dictionary coding, and Huffman coding. The ACM is created specifically for this project, but dictionary coding and Huffman coding are implementations made by other people that are integrated into the solution. Implementations are programs that others have made that implement algorithms. The implementations have to fulfill a set of requirements, but they also have to be able to be integrated with the other modules of the TCS so that the solution as a whole can fulfill the system requirements. Several implementations are found for the dictionary coding and Huffman coding algorithms. The implementations have to meet the following requirements:

- The implementations have to have an open license [77]
- The implementations have to be functioning correctly, even when the input data can be many megabytes
- The implementations have to be compatible with text of different encodings

The implementations have to have an open license in case changes has to be made to the code. Specifically, the implementations need a license that allows modifications. Modifications to an implementation might be necessary to integrate the modules into one functioning solution.
The implementations have to be able to compress files of considerable sizes. The largest file that will be compressed is 21.6 MB. Fulfilling this requirement is also necessary for the TCS to fulfill the system requirement; “The TCS can compress files that are over 20 MB”.
The implementations also have to be able to work with text of different encodings. The data set used for the evaluation uses 7-bit ASCII, 8-bit cp037, and variable length UTF-8. This is required so that the solution can work with all types of text.
The implementations will preferably be written in the Python programming language as the ACM is written in Python. This is, however, not a requirement. There are no requirements for which version of Python the implementations are using. Implementations with a different version than the ACM can be converted so that the modules are better integrated. If less than three Python implementations are found for evaluation then C implementations will be used, as C can easily be integrated with Python programs.
The implementations that meet the requirements are tested against each other, and the implementations that get the best result (the highest compression ratio) are chosen to be the dictionary coding and Huffman coding module for the TCS.

4.4 Design of the TCS

The TCS is a compression solution that consists of three modules: the ASCII compression module (ACM), the dictionary coding module, and the Huffman coding module. These modules are loosely coupled, meaning they are independent programs that can be integrated into one solution, or they can be used as standalone programs [78]. Having loosely coupled modules gives agility in the search for the highest compression ratio, and it is necessary in order to evaluate the modules individually. The modules are written in C or Python, and they can be executed sequentially using a Python program or a Unix shell script.

The TCS is intended to first use the ACM, then the dictionary coder, and finally the Huffman coding module. The ACM is used first because it is only able to compress ASCII characters. The output from the dictionary coder or Huffman coding modules are binary data that do not contain ASCII characters, so these modules should not be used before the ACM. The output from the ACM is a text file in cp037 encoding. This output is sent to the dictionary coder which searches for repeating substrings in the text. The binary output from the dictionary coder is then sent to the Huffman coding module which searches for byte values that are used more frequently than others. The output file from the Huffman coding module is the final compressed file from the TCS.

4.4.1 Design of the ACM

The ACM is a program written in Python 3.6.9 [79] consisting of two script files: ascii-compression.py and charprocessing.py. ascii-compression.py is responsible for reading and writing files, while charprocessing.py handles the actual text compression.

The compression algorithm that the ACM uses is similar to Shoco's algorithm (see section 2.2.1) and works by *merging* characters together. Merging characters means that two characters from a text file, e.g. "a" and "b", can be stored together in the space of a single character. The algorithm is only able to merge ASCII characters. As mentioned in section 2.2, ASCII characters only use the last 7 bits for the symbol code, while the first bit can be used to detect errors in data. The output from the ACM (the compressed file) is a text file with cp037 encoding. Unlike ASCII, the cp037 encoding uses all 8 bits for the symbol code. In the case of ACM, the first bit of a cp037 character is used to signal whether the character is merged or not, and the last 7 bits are the symbol code for either one or two characters. The cp037 encoding is chosen as the output format because it uses all 8 bits, instead of ASCII's 7 bits. cp037 is also a standard encoding in Python, and it is intended for English language, just like US-ASCII [80].

ascii-compression.py

The ascii-compression.py script takes a text file with ASCII, UTF-8, or cp037 encoding as input. It will not accept binary files or other text encodings. It then sequentially sends every line from the text file to the charprocessing.py script which handles the compression. After the output file is written, it performs a simple test to check if the output file is bigger than the original file. This can happen if the input file contains a large number of characters that can not be

encoded in ASCII. If the output file is bigger than the original file then the compressed data is discarded, as the point of a compression program is to make files smaller. Instead, the output file is rewritten with the original data, making the compressed file just as big as the original file. Finally, the compressed file gets appended a new line that contains either a 0 or a 1. This number signifies whether the compressed file has been altered (0) or if it is a copy of the original file (1). This information is important for the decompressor, which will either use the decompression algorithm or copy the file over as is. When a compressed file is decompressed, the number is removed, and the text file is restored to its original state.

charprocessing.py

The charprocessing.py script is responsible for the compression and decompression. The compression algorithm iterates through every character in a text and stores them in a compressed file with cp037 encoding. How each character is stored depends on two factors: can the character be encoded in ASCII, and does the following bigram only consist of common characters.

If the character is not included in the ASCII encoding scheme it means the character may take up more than one byte of storage. An example is the UTF-8 character "é" which uses two bytes. When this character is compressed, the algorithm has to signal that it uses more than one byte. Otherwise, the decompression algorithm will think that there are two characters that each use one byte. Before the character is written, two backslashes (\\) gets prepended to the character, signifying the next character is not ASCII. The special character then gets converted to the number that represents the character. This is necessary because the special character can not be encoded in cp037 if it is longer than one byte. Finally, one backslash is appended after the character, signifying the special character is over. Having a signal before and after the character is necessary because the character can be from 2 to 4 bytes long. The compressed version of the character "é" would be represented as `\\233\`.

If the original file contains backslash characters, then they need to be signaled so the decompression algorithm does not confuse then for a non-ASCII character. E.g. if two backslashes in a row are not signaled by the compressor then the decompressor will expect the next characters to be a special character. The way the compressor signals a backslash is by adding two more backslashes before it. This means that every backslash in the original file will be replaced with three backslashes. When the decompressor encounters a backslash in the compressed file it knows the next character will also be a backslash, but the one after that can be the beginning of a special character, or a third backslash. If it is a special character, it is properly decompressed, and if it is a third backslash then two backslashes are removed.

If an ASCII character and its succeeding character are both included in the list of common character, then they will be merged together. Merging will only work if two sequential characters are common. If none, or only one, of the sequential characters are in the list of common characters then they will not be merged. Instead, they will be stored in the compressed file just as they were in the original file. For example, the character "a" has a bit value of `01100001`. This is the binary representation of the number 97 and this is how the ACM

reads the characters. The first bit is 0, which tells the algorithm that the following 7 bits represent a character that has not been merged.

```
2
3  firstCommonChars = ["e","t","a","o","n","i","h"," "]
4  secondCommonChars = ["e","t","a","o","n","i","h"," ","s","r","l","d","u","c","m","w"]
5
```

Figure 2. List of common characters.

If a character is included in the `firstCommonChars` list, as shown in Figure 2, and the next character is included in `secondCommonChars` then they can be merged together. This is a list of the most frequently used letters (plus the space symbol) in the English language [8] and is the compression model for the ACM. A compression model can be a list of frequently used characters, symbols, or bigrams. The ACM uses the same model for all kinds of texts, which means that English texts should get a better compression ratio than texts in other languages.

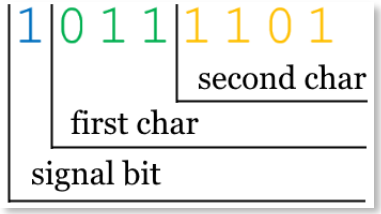

Figure 3. Binary representation of merged character.

A merged character consists of 3 parts: the signal bit, the first character, and the second character. For a merged character, the first bit is set to 1. The three next bits represent the position of the first character in the `firstCommonChars` list. In the example in Figure 3, the first character bits are `011`, which corresponds to the number 3. This tells the decompressor that the first character is the character with index 3 in `firstCommonChars`, which is "o". The second character uses 4 bits, which means it can represent twice as many characters as the first one. In the example, the second character bits are `1101`, which corresponds to the number 13. The character with index 13 in `secondCommonChars` is "c". The results from the decompression shows that the merged character in Figure 3 represent the bigram "oc".

5 Evaluation and results

This chapter describes the evaluation process for the individual modules of the TCS, and the evaluation of the system as a whole. The evaluation of the dictionary coding and Huffman coding modules consists of choosing the implementations with the highest compression ratio to be used in the TCS. The evaluation of the ACM includes measuring its compression ratio, measuring the compression ratio of the compression program Shoco, and comparing the results. The evaluation of the whole system (the TCS) includes measuring its compression ratio and comparing its performance to other general-purpose compression programs.

5.1 Evaluating the dictionary coding and Huffman coding modules

A collection of potential implementations for the dictionary coding and Huffman coding module are compared against each other. The data set (see section 4.2.1), which includes 6 texts of different characteristics, is tested on each of the implementations. The data set is stored locally on a computer. The compression ratio is measured for each of these texts and for each of the implementations. The compression ratio is the measurement of how much smaller the compressed file is compared to the uncompressed file, and is calculated by having the uncompressed size divided by the compressed size of a file. For example, if an uncompressed file is 10 MB and the compressed file is 8 MB, then the implementation has a compression ratio of 1.25 for that specific file. If the compression ratio was 1 then the compressed file would have the same size as the uncompressed file, meaning the algorithm was unable to compress the file. The implementations that have the highest compression ratio on average for every text in the data set are chosen to be the dictionary coding and Huffman coding modules for the TCS. Average compression ratio was chosen as the criteria because the solution is expected to be used for different kinds of texts, and the implementations that have the highest compression ratio for every text are therefore the most fitting implementations.

5.2 Choosing the dictionary coding implementation

Several dictionary coding implementations are considered for the TCS. These implementations represent some dictionary coding algorithm. The dictionary coding algorithms considered are LZW [81], LZ77 [10], LZ78 [38], and Re-Pair [40]. These algorithms are considered because they are used by popular compression programs [4, 5, 6] and are therefore proven to be effective. There exist more dictionary coding algorithms, but they are not used by popular compression programs.

A lot of the implementations found on the version control website GitHub does not specify what license their solution has and can therefore not be considered. An open license implementation of the LZ78 algorithm, called MeZip [82] appears to work with ASCII text, but not with cp037 encoding. One implementation of the LZW algorithm [83] does not work for UTF-8 encoding and can not be in consideration. Another implementation of the LZW algorithm [84] appears to work for all encodings, but decompressing the compressed file

causes an error. The non-profit website Rosettacode [85] has implementations for the LZW algorithm in several programming languages, including Python [86], but the solution does not work for all character encodings.

There are standard modules in Python for popular compression programs, such as Gzip [87], Bz2 [88], or Zipfile [89], but these modules all use multiple algorithms in combination with each other. For example, all of these modules use Huffman coding and some use dictionary coding, but it is not possible to only use the dictionary coding algorithm and not Huffman coding. The implementations that are used can not be packages of multiple algorithms.

5.2.1 Comparing the implementations

One Python implementation and two C implementations meet the requirements; they have an open license, they function correctly even for large files, and they are compatible with text of different encodings. The implementations are then compared against each other and the compression ratio is measured for every text in the data set. The Python implementation is executed with Python version 2.7.17, and the C implementations are compiled with GNU Compiler Collection version 7.5.0.

One implementation is of the LZ77 algorithm and is written in Python [90]. The implementation was downloaded June 15th, 2020. This implementation gives the user the ability to change the algorithm's window size (sliding window) [10]. The window size determines how much of the text the algorithm will analyze at any given point to find matching substrings. A bigger window size gives a higher compression ratio, but also makes the algorithm slower. A window size of 400 characters was used for all experiments, as this was the maximum possible window size.

Another implementation is a C implementation of the LZW algorithm [91]. The implementation was downloaded June 9th, 2020. The LZW algorithm does not use a sliding window like LZ77 uses, but instead appends substrings to a dictionary that may or may not reoccur in the text. If a substring is repeated then it is replaced with a reference to an entry in the dictionary [35].

The last implementation is a C implementation of the LZ77 algorithm [92]. The implementation was downloaded June 15th, 2020. Just like the Python implementation, it is possible to change the window size. The maximum possible window size for this implementation is 1,048,576 characters, and this is therefore used for all experiments.

Data set	LZ77 (Python)	LZW (C)	Lz77 (C)
XML file (21.6 MB)	1.47	1.92	1.85
cp037 encoded XML file (18.1 MB)	1.45	1.61	1.75
C code file (64 KB)	1.75	2.13	2.5
English book (680 KB)	1.3	1.96	1.61
Italian book (626 KB)	1.33	2.13	1.56
Chinese book (285 KB)	1.16	1.59	1.34
Average	1.41	1.89	1.77

Table 2. Results from the dictionary coding implementations comparison

Table 2 shows the compression ratio of the three dictionary coding implementations for each text in the data set. As mentioned in section 5.1, the compression ratio is the measurement of how much smaller the compressed file is compared to the uncompressed file. We see that the Python implementation of the LZ77 algorithm has the worst compression ratio. This is most likely due to the restriction of the implementation's window size. The C implementation of the LZ77 algorithm has a much better compression ratio, even though it is also an implementation of the LZ77 algorithm. This is because it had a much bigger window size. We see that the LZW implementation has a higher compression ratio than the LZ77 C implementation for some of the texts, but a lower ratio for other texts. As mentioned in section 5.1, the implementation that has the best average results from the experiment is chosen as the dictionary coding module for the TCS. The C implementation of the LZW algorithm will therefore be used as the dictionary coding module.

5.3 Choosing the Huffman coding implementation

A number of open source implementations of the Huffman coding algorithm are found and evaluated. Most implementations that are found do not specify what license their solution has and can therefore not be considered, just like when the dictionary coding implementation was found. As mentioned in 5.2, there are standard Python modules for popular compression programs that use the Huffman algorithm, but this is in combination with other algorithms. There are no standard Python modules for just the Huffman algorithm. Two Python implementations are found that are able to compress ASCII and UTF-8 text, but neither of them are able to compress cp037 encoded text [93] [94]. An implementation in C is found, but it is unable to build the source code so it can not be evaluated [95].

5.3.1 Comparing the implementations

Three implementations meet the requirements and are compared against each other; one Python implementation, and two C implementations. The Python implementation is executed with Python version 3.6.9, and the C implementations are compiled with GNU Compiler Collection version 7.5.0.

The Python implementation is called Dahuffman and is made by GitHub user Stefaan Lippens [96]. The implementation was downloaded July 26th, 2020. The implementation has the ability to build a code table (the Huffman tree) based on a list of symbol frequencies. The symbol frequencies are how many times individual characters are used in a text. The symbol frequency can be provided by the user or the implementation can calculate it from an input text. If the symbol frequency is calculated from an input text then the implementation first creates a code table, and then compresses the text using said code table.

The two C implementations, made by GitHub users Gagarine Yaikhom and Doug Richardson [97] [98], build the code tables automatically during compression. Yaikhom's implementation was downloaded July 26th, 2020, and Richardson's implementation was downloaded July 28th, 2020.

Data set	**Dahuffman**	**Yaikhom C impl.**	**Richardson C impl.**
XML file (21.6 MB)	1.51	1.5	1.5
cp037 encoded XML file (18.1 MB)	1.2	1.2	1.2
C code file (64 KB)	1.48	1.47	1.47
English book (680 KB)	1.8	1.7	1.7
Italian book (626 KB)	1.95	1.73	1.73
Chinese book (285 KB)	2.32	1.31	1.31
Average	1.71	1.49	1.49

Table 3. Results from the Huffman coding implementations comparison

Table 3 shows the compression ratio of the three Huffman coding implementations for each text in the data set. As table 3 shows, for most of the texts there is little variation between the implementations; in fact, the two C implementations got the exact same results, even though they are two different implementations made by different people [97] [98]. The Python implementation, Dahuffman, gets a slightly better compression ratio than the two C implementations for the English and Italian book, but a significantly better ratio for the Chinese book. For the three other texts, Dahuffman gets more or less the same results as the other implementations. This could be because the books contain more UTF-8 characters than the other texts, and Dahuffman is better equipped at compressing non ASCII characters. The Chinese book consists entirely of UTF-8 characters, so this seems like the most plausible theory. Dahuffman, has the highest average compression ratio of the implementations and will therefore be used as the Huffman coding module for the TCS.

5.4 Evaluating the ACM

The compression program most similar to the TCS is Shoco. Both TCS and Shoco are lossless text compression programs that specialize in compressing ASCII text [15]. One key difference is that Shoco is one algorithm while the TCS is a compression solution consisting of the 3 algorithms LZW, ACM, and Huffman coding. The ACM has a very similar functionality and solution to Shoco. In order to have a fair comparisson, only the ACM will be compared with Shoco in this section.

The paper discussed in 2.4.2 is another ASCII compression program. This solution is, however, not included in the comparison with the ACM because of its limitations. The proposed solution is only able to compress ASCII text and no other encodings. From the texts used in the data set, only the C code file is compatible with the solution, and there could therefore not be a fair comparison with the ACM.

The evaluation of the ACM consists of measuring its compression ratio for all the texts of the data set, and comparing the results with Shoco. The data set consists of 6 texts with different characteristics. Some of the texts are entirely or mostly ASCII characters, and other texts are mostly or entirely UTF-8 characters. The UTF-8 texts are included to test how well the ACM and Shoco can compress files that are not of intended use. Every test is repeated once to verify that the data is correct.

The ACM uses a default compression model that has not been trained for the data it is about to compress. A compression model (as mentioned in section 4.4.1) is a list of characters, symbols, or bigrams that are frequently used in a text. Shoco has the functionality to train a compression model for a specific text file, meaning the symbols and bigrams in the model will be frequently used in a text. The ACM does not have this functionality and uses a generic compression model that is optimized for English words. In the comparison, Shoco will be tested both with its default model and with a trained model.

5.4.1 Results from the ACM and Shoco comparison

Data set	**ACM**	**Shoco default**	**Shoco trained**
XML file (21.6 MB)	1.19	1.23	1.4
cp037 encoded XML file (18.1 MB)	1	0.59	0.75
C code file (64 KB)	1.21	1.17	1.35
English book (680 KB)	1.46	1.32	1.56
Italian book (626 KB)	1.29	1.15	1.5
Chinese book (285 KB)	1	0.53	0.75
Average	1.19	1	1.22

Table 4. Results from the ACM and Shoco comparison.

As table 4 shows, the compression ratio for the different texts are varied. This is expected, as the ACM and Shoco should get a higher compression ratio for ASCII-heavy texts as they can only compress ASCII characters. The cp037-encoded XML file and the Chinese book are the two texts that do not get a compression ratio over 1 for either algorithm, meaning the compressed file does not get smaller in size. These texts consist entirely of cp037 and UTF-8 characters, respectively. Both the ACM and Shoco are only able to compress ASCII characters, and other encodings can therefore make the output bigger. For Shoco, the compression ratio is less than 1, both with a default and trained model, meaning the compressed file is bigger than the original file. The ACM gets a ratio of 1, meaning the compressed file is the same size as the original file. The reason for this is that the ACM performs a test to check if the compressed file is bigger than the original. If it is then the compression is discarded, and the compressed file remains unaltered. Shoco does not have this test, and therefore gets a bigger compressed file.

As table 4 shows, the English book gets the highest compression ratio for both algorithms. The default models that the two algorithms use are optimized for English words and therefore get a high ratio for the English book. The ratio is even higher for Shoco when using a trained model as the model is customized for that specific text. The reason why the ACM and the default version of Shoco get different results for the same texts is because their default compression models are different [99]. The ACM model is relatively simple. It is a list of the 15 most used characters in the English language and the space symbol. Shoco uses a similar list, but also has a much bigger list of which characters are most likely to follow certain characters [99].

5.5 Evaluating the TCS

The evaluation of the TCS consists of measuring its compression ratio for all the texts in the data set and then comparing the results with other general-purpose compression programs. When the TCS is evaluated, all three modules are used. This means the output from one module is sent to the next module. The compression ratio for the output of the previous module is measured. This gives insight into how much of the compression each module is responsible for. Finally, the total compression ratio is measured. This is the ratio of the original file size compared to the final compressed file size. As mentioned in section 4.4, the ACM is used first, then the dictionary coding (LZW), and finally the Huffman coding.

The compression programs used for the comparison use lossless compression algorithms that can be used on all types of data, both text files and binary files like image, audio, or video. These programs are used for the comparison because they are generally used for the same tasks as the TCS is intended for. For example, Wikipedia has a service for downloading all the articles in their database in a text-only format. These text files have been compressed with Bzip2, Zip, or 7-Zip [100]. The TCS will be compared with the aforementioned programs. Gzip will also be used in the comparison as it is a popular compression program in use today [101] [102].

5.5.1 Results from the TCS evaluation

Data set	ACM	LZW	Huffman coding	Total ratio
XML file (21.6 MB)	1.19	1.6	1.01	1.93
cp037 encoded XML file (18.1 MB)	1	1.6	1.01	1.62
C code file (64 KB)	1.21	1.83	1.01	2.23
English book (680 KB)	1.46	1.36	1	1.99
Italian book (626 KB)	1.29	1.57	1	2.02
Chinese book (285 KB)	1	1.59	1	1.59
Average	1.19	1.59	1.01	1.9

Table 5. Results from the TCS evaluation.

As mentioned, the compression ratio measured for each module is based on the output from the previous module. The «total ratio» is the compression ratio achieved from the TCS (all three modules) for the texts in the data set. The total ratio is the result that will be used in the evaluation with other compression programs in the next section.

As table 5 shows, some of the files benefit more from the ACM and some benefit more from the LZW. E.g. the C code file gets a smaller ratio than the English book from the ACM, while it gets a higher ratio than the English book from the LZW. This is because the English book uses more characters that are frequent in the English language, and therefore get a better ratio from the ACM, while the C code file has more recurring words and substrings, and therefore get a better ratio from the LZW.

Table 5 also shows that the Huffman coding achieves virtually no compression ratio for any of the files. This could be because the Huffman coding implementation was not designed to compress binary data. It is also possible that the binary output from the LZW is too random and irregular for the Huffman coding to achieve any compression. The output from the LZW has already been compressed by the ACM, and it is possible that the combination of these two modules removes the need for Huffman coding. The results from this evaluation requires a new evaluation where only the LZW and Huffman coding is used. This will give answer to whether the ACM disrupts the capabilities of the Huffman coding module, or if the module is unable to compress binary data.

Data set	LZW	Huffman coding	Total ratio
XML file (21.6 MB)	1.92	1	1.92
C code file (64 KB)	2.13	1.01	2.14
English book (680 KB)	1.96	1	1.96
Italian book (626 KB)	2.13	1	2.13
Average	2.04	1	2.04

Table 6. Results from the LZW and Huffman coding module evaluation.

The cp037 encoded XML file and the Chinese book are not included in this evaluation as the ACM will not have any effect on these files, and the purpose of this evaluation is to compare with the results where the ACM does have an effect. Table 6 shows that even when the ACM is not used, the Huffman coding module gives virtually no compression ratio. This means that the module is unable to compress binary data.

5.5.2 Results from the TCS comparison

Data set	TCS	Bzip2	Zip	7-Zip	Gzip
XML file (21.6 MB)	1.93	3.79	2.96	4.24	2.96
cp037 encoded XML file (18.1 MB)	1.62	3.12	2.45	3.48	2.45
C code file (64 KB)	2.23	4.22	3.88	4.25	3.91
English book (680 KB)	1.99	3.52	2.65	3.18	2.65
Italian book (626 KB)	2.02	3.54	2.66	3.14	2.66
Chinese book (285 KB)	1.59	2.59	2.01	2.35	2.01
Average	1.9	3.46	2.77	3.44	2.77

Table 7. Results from the TCS comparison.

Table 7 shows that the TCS gets a lower compression ratio than the other programs for every file in the data set. Bzip2 and 7-Zip get the highest average compression ratios because they use different algorithms than Zip and Gzip, which are based on the DEFLATE algorithm [27] [103] [5] [4]. The TCS uses a solution similar to DEFLATE, but with the ACM as a preprocessor. The DEFLATE algorithm is a combination of the LZ77 dictionary coder and Huffman coding [11], while the TCS uses the LZW dictionary coder and Huffman coding. Table 6 shows that the Huffman coding module used for the TCS is not compatible with the LZW module, and therefore the TCS does not have a working alternative to the DEFLATE algorithm. To see if the TCS will get better results if the LZW and Huffman coding module used on the TCS were replaced with a pure DEFLATE algorithm, a new evaluation must be conducted. In this evaluation, the Zip and Gzip programs will be used with the ACM as a preprocessor. This evaluation will give answer to whether using the ACM as a preprocessor will increase the compression ratio of the DEFLATE algorithm.

Data set	**Zip**	**ACM + Zip**	**Gzip**	**ACM + Gzip**
XML file (21.6 MB)	2.96	2.92	2.96	2.92
C code file (64 KB)	3.88	3.77	3.91	3.82
English book (680 KB)	2.65	2.66	2.65	2.66
Italian book (626 KB)	2.66	2.58	2.66	2.58
Average	3.04	2.98	3.05	3

Table 8. Results from the ACM + DEFLATE comparison.

As with Table 6, the cp037 encoded XML file and the Chinese book are not included in this evaluation as the ACM will not have any effect on these files. As table 8 shows, when the ACM is used as a preprocessor for either Zip or Gzip, the compression ratio is slightly lower. Only for the English book does the ACM give a marginally higher compression ratio, but a 0.01 increase in compression ratio is insignificant. Table 8 also shows that using the ACM plus the DEFLATE algorithm that Zip and Gzip uses gives a much higher average compression ratio than the TCS.

6 Discussion

This chapter covers the discussion of the results from the evaluations. The chapter also discusses the shortcomings of the solution, what the contributions of this thesis are, and in what situations they can be used.

6.1 Efficacy of the TCS

The TCS was intended to be a contender to the DEFLATE algorithm, but with a focus on compressing ASCII text. Conceptually, the TCS uses the same techniques as DEFLATE (dictionary coding and Huffman coding) but introduces the ACM as a new step in the compression pipeline. The difference between DEFLATE and the TCS is that DEFLATE uses the LZ77 dictionary coder [11] while the TCS uses the LZW dictionary coder. The evaluation of the TCS shows that the DEFLATE-alternative that the TCS uses is imperfect. Therefore, using the TCS (in its current state) as an alternative to the DEFLATE algorithm is not a viable option. The comparison of compression programs in table 7 also showed that the programs that used alternative solutions to the DEFLATE algorithm got a higher compression ratio. Therefore, even if the TCS had a working DEFLATE algorithm, it could still not compete with the compression ratios of Bzip2 and 7-Zip. Instead, the contribution of this thesis should be the ACM, and not the TCS.

6.2 ACM and Shoco comparison

Shoco is a close contender to the ACM and is the related work that is discussed the most in this thesis. Both algorithms have the same premise and similar technical solutions, but they are not identical. One difference is the complexity of the compression models that the two algorithms use. Another difference is the efficiency of the algorithms. The Shoco algorithm is described as a “Fast compressor for short strings”, and the main selling point for Shoco seems to be its speed [15]. The ACM has not been measured for latency, but the experimentation has shown that Shoco (with the default compression model) is faster than the ACM. The reason that the ACM was not measured for latency is because the focus of the ACM lies on its compression capabilities and not on its efficiency. The ACM was intended to be a proof of concept that the ASCII compression algorithm was theoretically possible. This was before the existence of Shoco was known to the author, which proved that ASCII compression was possible.

6.2.1 Trained models

In the comparison between the ACM and Shoco (table 4), Shoco was used both with its default compression model and with a trained compression model. The ACM does not have the functionality to make a trained model and therefore used its default model. The simplest way of making a trained model for Shoco is done in three steps. First, a Python script takes a text file as input and outputs the compression model as a C header file. Then, the Shoco program, which is written in C, has to be recompiled with the new model. Finally, the compiled program is executed with the intended text file as input. It is also possible to

customize several aspects about how the model is made, e.g. whether punctuation marks will be included, but this should require knowledge about the file that is about to be compressed.

Creating trained models for Shoco may increase the compression ratio (see table 4), but it is a tedious and time-consuming task. As mentioned, the main selling point for Shoco is its speed, and creating a trained model every time the program is used will make the compression process slower. Trained models can be a benefit for a one-time compression task, e.g. if a single, large file needs to be compressed. However, as a general-purpose text compression program, the default model should be used.

6.2.2 Discussing the results from the comparison

The results from table 4 shows that the ACM (using its default model) has a higher compression ratio than the default version of Shoco for every text except one. Even so, the results are relatively similar for the texts that mainly contain ASCII characters. The two texts that do not contain any ASCII characters (the cp037 encoded XML file and Chinese book) get a compression ratio of less than 1 on Shoco, which means the compressed file grows in size. As mentioned, the ACM performs a test to check if the compressed file is bigger than the original file and discards the compression if it is. Shoco does not perform this test and can therefore end up with a bigger compressed file. This test is advantageous for a general-purpose text-compression program because it does not require any prior knowledge about the encoding of the text file. Shoco should therefore only be used if the user knows that a text contains mostly ASCII characters, while the ACM can be used on any text without knowledge about its encoding.

6.2.3 Shortcomings of the ACM

The compression technique that the ACM uses for non-ASCII characters is not optimal. Certain UTF-8 characters, like emojis, can use up to 4 bytes in an uncompressed file. These characters will use 9 bytes when compressed. The nonoptimal compression is caused by converting the character to the number that represents it and writing the number digit for digit. This could have been optimized, but the time constraint lead to it being an issue that has yet to be fixed.

A file that is compressed with the ACM will have an added line that is either a 1 or a 0. This is to signal whether the file is altered or not, and it is removed when the file is decompressed. A problem that could occur is if a file happens to have a 1 or 0 as its last line, and it is unknown whether the file is compressed or not. If a user mistakes this uncompressed file for a compressed file then it will be altered when it is decompressed. A compressed file is saved as a text file (.txt) and therefore probably has the same file type as an uncompressed file, so it is impossible to say if the file is compressed or not.

The ACM is written in the Python programming language, which is noticeably slower during runtime than e.g. C [104] [105]. The C programming language is generally faster than Python, which is why Python programs often integrate C modules for efficiency reasons [106]. The ACM could potentially have been faster if it was written in C.

7 Conclusion and future work

This thesis presents a solution, the TCS, for compressing text, with a special focus on compressing ASCII text. The TCS tackles the problems with storing and time-consuming transferring of large amounts of text data. The TCS decreases file sizes so they require less storage space. Moreover, the TCS decreases the bandwidth when a compressed file is downloaded or transferred over the internet. The TCS is customized for ASCII text but is designed to work on any form of text. The ambition of the TCS is to compress files for users that are dealing with large amounts of ASCII or ASCII-heavy texts.

The TCS consists of the three modules ACM, dictionary coding module, and Huffman coding module. The TCS is based on the DEFLATE algorithm, which also combines dictionary coding and Huffman coding. The TCS uses the ACM as a preprocessor to the DEFLATE alternative to boost the compression for ASCII test. The ACM was developed by the author specifically for this project, while the dictionary and Huffman coding modules are implementations made by other people [91] [96]. The individual modules have been evaluated, and the TCS has been evaluated and compared with general-purpose compression programs.

The results from the evaluation of the TCS shows that, in its current state, it is not a contender to other general-purpose compression programs. A comparison with the ASCII compression algorithm Shoco shows that the ACM gets a better average compression ratio when generic compression models are used. The ACM can therefore be a contender for the more specialized field of ASCII compression.

The evaluation of the ACM (see table 4) shows that it can be used on any form of text without increasing the compressed size, but a significant compression ratio will only be achieved on ASCII-heavy texts. Nonetheless, the compression ratio achieved by the ACM is very small, compared to general purpose compression programs (see table 7). Combining the ACM with DEFLATE-based compression programs (see table 8) gives an average compression ratio just under DEFLATE, with one example of a minor advantage to the ACM. It is possible that improvements to the compression algorithm of the ACM could increase its compression ratio, and maybe even exceed that of DEFLATE (if the ACM is used as a preprocessor to DEFLATE).

7.1 Answering the research questions

- “What combination of techniques can improve the compression ratio for ASCII text?”

It is apparent that the ACM achieves compression for ASCII files, although some text files get higher compression ratios than others (see table 4). Despite the merit of the ACM, using it in combination with the dictionary coding and Huffman coding module (as well as DEFLATE-based compression programs) has not improved the compression ratio for ASCII text. The reason for this is that the output from the ACM is less compressible than plain text. The ACM has

only been tested as a preprocessor to DEFLATE-based compression programs, as the TCS was intended to be a DEFLATE alternative. It is possible that combining other algorithms (such as the ones used in Bzip2 and 7-Zip) with the ACM as a preprocessor could increase the compression ratio for ASCII text. This is future work for the TCS.

- "Does using ASCII compression in combination with dictionary coding and Huffman coding surpass the compression ratio of DEFLATE?"

In its current state, the TCS (ACM, dictionary coding module, and Huffman coding module) does not surpass the compression ratio of DEFLATE. As mentioned, the Huffman coding module of the TCS is not compatible with the dictionary coding module, and the TCS therefore does not have a working DEFLATE-alternative. Instead, the ACM has been tested in combination with the dictionary coding and Huffman coding of DEFLATE-based compression programs in table 8. Based on the average compression ratio from 4 texts of the data set, using the ACM in combination with dictionary coding and Huffman coding does not surpass the compression ratio of DEFLATE.

- "Will this text compression solution achieve a higher compression ratio than general-purpose compression programs for ASCII text?"

In the comparison between the TCS and 4 general-purpose compression programs in table 7, 4 of the texts contain mostly ASCII characters. As mentioned, neither the TCS nor the ACM + DEFLATE combination achieves a higher compression ratio than DEFLATE-based compression programs for ASCII texts. Furthermore, Bzip2 and 7-Zip outperform both the DEFLATE-based compression programs and the TCS for every text in the use case. In conclusion, the TCS does not achieve a higher compression ratio than general-purpose compression programs for ASCII text.

- "Will this text compression solution achieve a higher compression ratio than compression solutions that specialize on ASCII text?

In this thesis, the ACM has only been compared with the ASCII compression algorithm Shoco. As far as the author is aware, Shoco is the only other ASCII compression algorithm that is customized for compressing ASCII text, and is also capable of compressing other encodings. ASCII compression solutions that are incapable of compressing any other encodings, like the paper discussed in 2.4.2, have not been considered for comparisons because very few text files use only ASCII characters today [60]. In the ACM and Shoco comparison, ACM has a higher compression ratio than Shoco – using its default compression model, but ACM has a slightly lower ratio than Shoco – when using a trained model. As mentioned in section 6.2.1, there are pros and cons to Shoco's default and trained models. Both models are viable and can be used depending on the situation. Therefore, this research question can not be answered with a definitive yes or no.

7.2 Future work

Future work on the TCS include finding a Huffman coding module that can compress the binary output from the dictionary coding module. Achieving this could make the compression capabilities of the TCS on par with the DEFLATE algorithm. Experimenting with algorithms not used by DEFLATE to improve the compression ratio is also future work.

Improvements to the ACM can be made to increase its compression ratio. The ACM can improve its compression of non-ASCII characters, as the current technique is not optimal. Adding the ability to create trained models might also notably increase its compression ratio. With these improvements, it is possible that the ACM could increase the compression ratio of general-purpose compression programs if used as a preprocessor.

Improvements can also be made to increase the latency of the ACM. The code can be rewritten in C and optimizations can be made to the compression and decompression process. If the ACM was more focused on latency than compression ratio then it can be beneficial to use the ACM in situations where speed is a higher priority than compression ratio. For example, it could be used in combination with the LZ77-based compressor Snappy [107] [108] in a server back end context. Snappy compromises compression ratio for speed and will achieve better ratios for text files than binary files [107]. The results from table 5 shows that a dictionary coder (LZW) can achieve decent compression ratios with the ACM as a preprocessor so using the ACM as a preprocessor to Snappy could increase the compression ratio.

Bibliography

[1] K. Sayood, Introduction to data compression, 3rd ed. Amsterdam ; Boston: Elsevier, 2006.

[2] D. Salomon, Data compression the complete reference. New York: Springer, 2004.

[3] K. K. Shukla and M. V. Prasad, Lossy image compression: domain decomposition-based algorithms. London ; New York: Springer, 2011.

[4] «GNU Gzip». https://www.gnu.org/software/gzip/manual/gzip.html#Overview (opened Sep. 13th, 2020).

[5] «Untitled». PKWARE Inc., Opened: Sep. 13th, 2020. [Online]. Available at: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT.

[6] «WinRAR - WinRAR Documentation». https://documentation.help/WinRAR/ (opened Sep. 14th, 2020).

[7] D. Huffman, «A Method for the Construction of Minimum-Redundancy Codes», Proc. IRE, vol. 40, nr. 9, s. 1098–1101, sep. 1952, doi: 10.1109/JRPROC.1952.273898.

[8] S. Trost, «Alphabet and Character Frequency: English». https://www.sttmedia.com/characterfrequency-english (opened Sep. 15th, 2020).

[9] «Graphics Interchange Format (GIF) Specification», Opened: Sep. 14th, 2020. [Online]. Available at: https://www.w3.org/Graphics/GIF/spec-gif87.txt.

[10] J. Ziv and A. Lempel, «A universal algorithm for sequential data compression», IEEE Trans. Inform. Theory, vol. 23, nr. 3, pp. 337–343, May 1977, doi: 10.1109/TIT.1977.1055714.

[11] «RFC 1951 DEFLATE Compressed Data Format Specification ver 1.3». https://www.w3.org/Graphics/PNG/RFC-1951 (opened Sep. 15th, 2020).

[12] «Data Compression Conference - Home». https://www.cs.brandeis.edu/~dcc/index.html (opened Sep. 13th, 2020).

[13] A. Farina, G. Navarro, and J. R. Parama, “Boosting Text Compression with Word-Based Statistical Encoding”, The Computer Journal, vol. 55, no. 1, pp. 111–131, Jan. 2012, doi: 10.1093/comjnl/bxr096.

[14] S. Wiedemann et al., «DeepCABAC: A Universal Compression Algorithm for Deep Neural Networks», IEEE J. Sel. Top. Signal Process., vol. 14, nr. 4, s. 700–714, May 2020, doi: 10.1109/JSTSP.2020.2969554.

[15] «Shoco - a fast compressor for short strings». https://ed-von-schleck.github.io/shoco/ (opened Sep. 14th, 2020).

[16] «Ed-von-Schleck/shoco», GitHub. https://github.com/Ed-von-Schleck/shoco/commits/master (opened Sep. 14th, 2020).

[17] N. Indurkhya and F. J. Damerau, Red., Handbook of natural language processing. Boca Raton, FL: Chapman & Hall/CRC, 2010.

[18] «torvalds/linux: Linux kernel source tree». https://github.com/torvalds/linux (opened Sep. 14th, 2020).

[19] «Untitled». https://api.github.com/repos/torvalds/linux (opened Sep. 14th, 2020).

[20] C. McAnlis and A. Haecky, Understanding compression: data compression for modern developers, First edition. Sebastopol, CA: O'Reilly, 2016.

[21] Institute of Electrical and Electronics Engineers, and Amirkabir University of Technology, Red., 2011 19th Iranian Conference on Electrical Engineering (ICEE 2011): Tehran, Iran, 17 - 19 May 2011. Piscataway, NJ: IEEE, 2011.

[22] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, «A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs», in 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines, Vancouver, BC, Canada, May 2015, pp. 52–59, doi: 10.1109/FCCM.2015.46.

[23] Y. Q. Shi and H. Sun, *Image and Video Compression for Multimedia Engineering,* Boca Raton, FL: CRC Press, 1999, pp. 140–141.

[24] M. Aggarwal and A. Narayan, «Efficient Huffman decoding», in Proceedings 2000 International Conference on Image Processing (Cat. No.00CH37101), Vancouver, BC, Canada, 2000, vol. 1, pp. 936–939, doi: 10.1109/ICIP.2000.901114.

[25] S. T. Klein, «Parallel Huffman Decoding with Applications to JPEG Files», The Computer Journal, vol. 46, nr. 5, pp. 487–497, May 2003, doi: 10.1093/comjnl/46.5.487.

[26] A. Langiu, «On parsing optimality for dictionary-based text compression—the Zip case», Journal of Discrete Algorithms, vol. 20, pp. 65–70, May 2013, doi: 10.1016/j.jda.2013.04.001.

[27] «bzip2 and libbzip2, version 1.0.8». https://www.sourceware.org/bzip2/manual/manual.html (opened Sep. 14th, 2020).

[28] D. Pickell, «Qualitative vs Quantitative Data – What's the Difference?» https://learn.g2.com/qualitative-vs-quantitative-data (opened Sep. 15th, 2020).

[29] C. Read, Deductive and Inductive. Outlook Verlag, 2018.

[30] Merriam-Webster, «'Deduction' vs. 'Induction' vs. 'Abduction'». https://www.merriam-webster.com/words-at-play/deduction-vs-induction-vs-abduction (opened Sep. 15th, 2020).

[31] D. Kleiman, Red., The official CHFI exam 312-49 study guide: for computer hacking forensics investigators. Burlington, MA: Syngress Pub, 2007.

[32] R. Togneri and C. J. S. DeSilva, Fundamentals of information theory and coding design. Boca Raton: Chapman & Hall/CRC, 2002.

[33] «PNG Documentation». http://www.libpng.org/pub/png/pngdocs.html (opened Sep. 15th, 2020).

[34] R. Hoffman, Data Compression in Digital Systems. Boston, MA: Springer US, 1997.

[35] D. R. Hankerson, G. A. Harris, and P. D. Johnson, Introduction to information theory and data compression, 2nd ed. Boca Raton, Fla: Chapman & Hall/CRC Press, 2003.

[36] S. Senthil and L. Robert, « Text Compression Algorithms - A Comparative Study», IJCT, vol. 02, nr. 04, pp. 444–451, Dec. 2011, doi: 10.21917/ijct.2011.0062.

[37] S. A. A. Taleb et al., «Improving LZW Image Compression», EJSR, vol. 44, pp. 502–509, Aug. 2010.

[38] J. Ziv and A. Lempel, «Compression of individual sequences via variable-rate coding», IEEE Trans. Inform. Theory, vol. 24, nr. 5, pp. 530–536, Sep. 1978, doi: 10.1109/TIT.1978.1055934.

[39] M. Atwal and L. Bansal, «Fast Lempel-ZIV (LZ'78) Algorithm Using Codebook Hashing», International Journal of Engineering and Technical Research (IJETR), pp. 220–223, Mar 2015.

[40] N. J. Larsson and A. Moffat, «Offline Dictionary-Based Compression», Opened: Sep. 14th, 2020. [Online]. Available at: http://www.larsson.dogma.net/dcc99.pdf.

[41] R. Bose, Information theory, coding and cryptography. New Delhi: Tata McGraw-Hill, 2008.

[42] M. Burrows and D. J. Wheeler, «A block-sorting lossless data compression algorithm», 1994.

[43] D. C. Wyld, M. Wozniak, ACITY, and Academy & Industry Research Collaboration Center, Red., Advances in computing and information technology: first international conference, ACITY 2011, Chennai, India, July 15 - 17, 2011 ; [the First International Conference on Advances in Computing and Information Technology] ; proceedings. Berlin: Springer, 2011.

[44] M. Arregoces and M. Portolani, Data center fundamentals. Indianapolis, Ind: Cisco, 2004.

[45] A. Carlsson and A. B. Miller, «Future Potentials for ASCII art», PostDigital Art - Proceedings of the 3rd Computer Art Congress, pp. 13–24, Nov 2012.

[46] D. Oluwade, «A Comparative Analysis and Application of the Compression Properties of Two 7-Bit Subsets of Unicode», 2012, doi: 10.1.1.645.8168.

[47] J. K. Korpela, Unicode explained, 1st ed. Sebastopol, CA: O'Reilly, 2006.

[48] B. Ediger, Advanced rails, 1st ed. Sebastopol, CA: O'Reilly, 2008.

[49] G. A. V. Pai, Data structures and algorithms: concepts, techniques and applications. New Delhi: Tata McGraw-Hill, 2008.

[50] International Conference on Intelligent Computing, D.-S. Huang, X.-P. Zhang, and G.-B. Huang, Eds., Advances in intelligent computing: International Conference on Intelligent Computing, ICIC 2005, Hefei, China, August 23-26, 2005 : proceedings. Berlin; New York: Springer, 2005.

[51] A. Desoky and M. Gregory, "Compression of text and binary files using adaptive Huffman coding techniques," in Conference Proceedings '88., IEEE Southeastcon, Knoxville, TN, USA, 1988, pp. 660–663, doi: 10.1109/SECON.1988.194940.

[52] A. Bookstein and S. T. Klein, "Is Huffman coding dead?" Computing, vol. 50, no. 4, pp. 279–296, Dec. 1993, doi: 10.1007/BF02243872.

[53] J. Duda, "Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding," arXiv:1311.2540 [cs, math], Jan. 2014, Accessed: Oct. 03, 2020. [Online]. Available: http://arxiv.org/abs/1311.2540.

[54] Meteficha, https://commons.wikimedia.org/wiki/File:Huffman_tree_2.svg. 2007.

[55] «The Computer Journal». https://academic.oup.com/comjnl (opened Oct. 29th, 2020).

[56] A. Moffat, "Word-based text compression," Softw: Pract. Exper., vol. 19, no. 2, pp. 185–198, Feb. 1989, doi: 10.1002/spe.4380190207.

[57] E. S. De Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, “Direct pattern matching on compressed text,” in Proceedings. String Processing and Information Retrieval: A South American Symposium (Cat. No.98EX207), Santa Cruz de La Sierra, Bolivia, 1998, pp. 90–95, doi: 10.1109/SPIRE.1998.712987.

[58] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates, “Fast and flexible word searching on compressed text,” ACM Trans. Inf. Syst., vol. 18, no. 2, pp. 113–139, Apr. 2000, doi: 10.1145/348751.348754.

[59] «Information Technology Coding and Computing – ITCC». http://www.itcc.info/ (opened Nov. 3rd, 2020).

[60] J. Istle, P. Mandelbaum, and E. Regentova, “Online compression of ASCII files”, ITCC, vol. 2, Apr. 2004, doi: 10.1109/ITCC.2004.1286559

[61] «What is the difference between "Legacy" and "Best method" compression?». https://support.winzip.com/hc/en-us/articles/115011643067-What-is-the-difference-between-Legacy-and-Best-method-compression- (opened Nov. 5th, 2020).

[62] A. Håkansson, «Portal of Research Methods and Methodologies for Research Projects and Degree Projects», 2013, [Online]. Available at: https://www.semanticscholar.org/paper/Portal-of-Research-Methods-and-Methodologies-for-H%C3%A5kansson/e591fa1db4a633cd956cf06e204f82cffdc02e3e.

[63] B. B. Agarwal, S. P. Tayal, and M. Gupta, Software engineering & testing: an introduction. Sudbury, Mass: Jones and Bartlett, 2010.

[64] G. Guida, G. Lamperti, and M. Zanella, «Software Prototyping in Data and Knowledge Engineering». Heidelberg, Germany: Springer Netherlands, 1999.

[65] J. A. Highsmith, Agile software development ecosystems. Boston: Addison-Wesley, 2002.

[66] I. Sommerville, Software engineering, 8th ed. Harlow, England; New York: Addison-Wesley, 2007.

[67] «UTF-8 and Unicode Standards». http://www.utf-8.com/ (opened Sep. 16th, 2020).

[68] A. M. McEnery and R. Z. Xiao, «Character encoding in corpus construction.», in Developing Linguistic Corpora : A Guide to Good Practice, M. Wynne, Red. Oxford, UK: AHDS, 2005.

[69] «Extensible Markup Language (XML)». https://www.w3.org/XML/ (opened Sep. 16th, 2020).

[70] «JSON Syntax». https://www.w3schools.com/js/js_json_syntax.asp (opened Sep. 16th, 2020).

[71] «HTML Charset». https://www.w3schools.com/html/html_charset.asp (opened Sep. 16th, 2020).

[72] «Wikimedia Downloads». https://dumps.wikimedia.org/backup-index.html (opened Sep. 24th, 2020).

[73] «torvalds/linux». https://github.com/torvalds/linux/blob/master/kernel/sys.c (opened Sep. 24th, 2020).

[74] F. Nietzsche and T. Common, Thus spake Zarathustra. Ware: Wordsworth Editions, 1997.

[75] Dante Alighieri, B. Garavelli, and L. Magugliani, La divina commedia. Milano: BUR Rizzoli, 2012.

[76] Ainajushi, 豆棚閒話. Salt Lake City: Project Gutenberg, 2008.

[77] «Open license - Creative Commons». https://wiki.creativecommons.org/wiki/Open_license (opened Sep. 16th, 2020).

[78] D. Kaye, Loosely coupled: the missing pieces of Web services. Marin County, Calif: RDS Press, 2003.

[79] «Python 3.6.9». https://www.python.org/downloads/release/python-369/ (opened Oct. 20th, 2020).

[80] «Standard encodings». https://docs.python.org/3/library/codecs.html#standard-encodings (opened Oct. 22nd, 2020).

[81] Welch, «A Technique for High-Performance Data Compression», Computer, vol. 17, nr. 6, pp. 8–19, Jun. 1984, doi: 10.1109/MC.1984.1659158.

[82] «meZip». https://github.com/aaronscode/meZip (opened Sep. 16th, 2020).

[83] «lzw-compression». https://github.com/biroeniko/lzw-compression (opened Sep. 16th, 2020).

[84] «python-lzw». https://github.com/joeatwork/python-lzw (opened Sep. 16th, 2020).

[85] «Rosetta Code». http://rosettacode.org/wiki/Rosetta_Code (opened Sep. 16th, 2020).

[86] «LZW compression». https://rosettacode.org/wiki/LZW_compression#Python (opened Sep. 16th, 2020).

[87] «gzip — Support for gzip files — Python 3.8.6rc1 documentation». https://docs.python.org/3/library/gzip.html (opened Sep. 16th, 2020).

[88] «bz2 — Support for bzip2 compression — Python 3.8.6rc1 documentation». https://docs.python.org/3/library/bz2.html (opened Sep. 16th, 2020).

[89] «zipfile — Work with ZIP archives — Python 3.8.6rc1 documentation». https://docs.python.org/3/library/zipfile.html (opened Sep. 16th, 2020).

[90] «LZ77-Compressor». https://github.com/manassra/LZ77-Compressor (opened Sep. 16th, 2020).

[91] «lzw-ab». https://github.com/dbry/lzw-ab (opened Sep. 16th, 2020).

[92] «ulz77». https://github.com/zooxyt/ulz77 (opened Sep. 16th, 2020).

[93] «huffman». https://github.com/emersonmde/huffman (opened Sep. 16th, 2020).

[94] «HuffmanEncoding». https://github.com/nrutkowski1/HuffmanEncoding (opened Sep. 16th, 2020).

[95] «hcomp». https://github.com/ronchauhan/hcomp (opened Sep. 16th, 2020).

[96] «dahuffman». https://github.com/soxofaan/dahuffman (opened Sep. 16th, 2020).

[97] «huffman». https://github.com/gyaikhom/huffman (opened Sep. 16th, 2020).

[98] «huffman». https://github.com/drichardson/huffman (opened Sep. 16th, 2020).

[99] «shoco». https://github.com/Ed-von-Schleck/shoco/blob/master/shoco_model.h (opened Oct. 9th, 2020).

[100] «Wikipedia:Database download». https://en.wikipedia.org/wiki/Wikipedia:Database_download#Dealing_with_compressed_files (opened Oct. 15th, 2020).

[101] T. Summers, "Hardware based GZIP compression, benefits and applications," CORPUS, vol. 3, pp. 2–68, 2008.

[102] D. Belanger, K. Church, and A. Hume, "Virtual Data Warehousing, Data Publishing and Call Detail," in Databases in Telecommunications, vol. 1819,

W. Jonker, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 106–117.

[103] «7z Format». https://www.7-zip.org/7z.html (opened Oct. 18th, 2020).

[104] Li Jun and Li Ling, "Comparative research on Python speed optimization strategies" in 2010 International Conference on Intelligent Computing and Integrated Systems, Oct. 2010, pp. 57–59, doi: 10.1109/ICISS.2010.5655011.

[105] M. Salib, "Faster than C: Static type inference with Starkiller" in PyCon Proceedings, Mar. 2004, pp. 2–26.

[106] D. M. Beazley, Python essential reference, 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2009.

[107] «snappy». https://github.com/google/snappy (opened Nov. 9th, 2020).

[108] «format_description.txt». https://github.com/google/snappy/blob/master/format_description.txt (opened Nov. 9th, 2020).