UPTEC IT 09 018
Examensarbete 30 hp
December 2009

Evaluation of Image Compression Algorithms for Electronic Shelf Labels

Robin Kuivinen

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Evaluation of Image Compression Algorithms for Electronic Shelf Labels

Robin Kuivinen

An advantageous innovation for retail stores is the ESL system, which consists of many electronic units, called labels, showing product and price information to customers on small displays. Such a system offers, among other things, efficient price updates that implies lower costs by reducing man-hours and paper volumes. Data transfered to a label can be compressed in order to minimize the data size and thus lowering the updating time and the energy consumption of a label.

In this study, twelve lossless compression prototypes were implemented and evaluated, using a corpus set of bi-level images, with respect to compressibility, encoding and decoding time. Six of those prototypes were subject to case studies of real ESL data and further studies of memory consumption.

The results showed that the low-precision arithmetic coder LowPac, which uses a context-based probability model, was one of the best prototypes over all data sets in terms of compressibility and it was also the most memory-efficient prototype. Using the corpus set, LowPac obtained encoding and decoding times of 2 times longer than one of the fastest prototypes, but with 108% improvement of compressibility, on average.

Handledare: Nils Hulth
Ämnesgranskare: Cris Luengo
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT09 018
Tryckt av: Reprocentralen ITC

Sammanfattning

En butiksägare inom detaljhandeln står inför en tidsödande process när nya priser på varor ska märkas upp. I en normalstor butik kan ett sådant arbete innefatta flera medarbetare och pågå under en längre tid beroende på antal varor som ska prissättas. Därmed kommer mycket arbetstid gå åt till prissättning, men också mycket pengar i form avlönade medarbetare. Priset för själva pappersetiketterna är inte heller att förglömma. Papperskostnaden för att prissätta ett fåtal varor är självklart försvinnande liten, men antal varor i en butik är otroligt stor. Tänk då också på att vissa av dessa varor får nya priser varje dag! Den totala papperskostnaden under en längre period är därför markant.

I och med datorns utveckling, och speciellt mikroprocessorns, har ett nytt koncept börjat infinna sig i butiker över hela världen, nämligen *elektroniska hyllkantsetiketter* som går under benämningen ESL - *Electronic Shelf Labels*. Det nya konceptet ersätter traditionella pappersetiketter med elektroniska etiketter. En elektronisk etikett består förenklat av en mikroprocessor, en liten skärm och en mottagare för trådlös kommunikation. Mottagaren för trådlös kommunikation möjliggör att hyllor kan flyttas runt utan begränsningar. På skärmen visas, precis som med pappersetiketter, information om produkten, t.ex. namn, pris och ursprungsland.

Etiketterna är energisnåla och drivs av batterier vilket medför en lång, men begränsad livslängd. Allt arbete utfört att en etikett påverkar dess livslängd, därav är det motiverat att försöka minska mängden data som skickas till etiketterna över den trådlösa kommunikationskanalen. Minskad data mängd leder till längre livslängd, vilket skjuter upp processen med att byta batterier.

I det här examensarbetet har tolv prototyper av olika icke-förstörande komprimeringsmetoder utvärderats med avseende på komprimeringsgrad, kodningstid och avkodningstid. En av prototyperna representerar en komprimeringsmetod som används av företaget Pricer i deras ESL plattform. Den metoden används som en referenspunkt i det här arbetet för att se hur de andra metoderna presterar i jämförelse. Prototyperna utvärderades med hjälp av testbilder som föreställde produkter. Sex av de tolv prototyperna utvärderades också med avseende på minnesåtgång. Det gjordes även två fallstudier för att se om resultatet av testbilderna stämde överens med verkligt data. Fallstudierna baserades på data från en norsk butik.

Resultaten visade att en prototyp vid namn LowPac var en av de bästa pro-

totyperna med avseende på komprimeringsgrad. Förbättringarna jämfört mot Pricers metod var sammanfattningsvis $103-194\%$ beroende på vilken datakälla som användes, men LowPac var då $2-4$ gånger långsammare. LowPac hade också den lägsta minnesåtgången av de prototyper som analyserades för det. Resultaten av fallstudierna visade att testbilderna var representativa, vilket påvisades genom de lika resultaten med avseende på komprimeringsgrad, kodningstid och avkodningstid.

Contents

Chapter 1

Introduction

1.1 Background

A recent innovation in retail automation is the Electronic Shelf Label (ESL) system. An ESL system consists of many electronic price labels equipped with displays to provide up-to-date product information to customers, e.g. price, product name and country of origin. A typical ESL (or label) is attached to the edge of a shelf as an ordinary paper label. Information to customers, such as prices, are distributed to the labels via wireless communication.

A retail store using an ESL system benefits by getting reduced cost in terms of lower paper volumes, reduced labor when changing prices and autonomous behavior to detect and correct erroneous prices. The work of updating price information of products is minimized by utilizing an automated system. One feature provided by an ESL system is the possibility to use campaign pricing, which offers accurate campaign prices at requested hours. Another feature is price integrity, enabling correct prices in both cash registers and product labels.

This master thesis was carried out at Pricer AB in Sollentuna, Sweden. Pricer was founded in 1991 and has offices worldwide, but the product research and development is done in Sweden. The purpose of this master thesis was to evaluate prototype implementations of different compression schemes that could be used to improve Pricer's ESL platform.

1.2 Motivations and Objectives

A typical retail store using the Pricer ESL platform is equipped with 25,000 labels. A large retail store may have up to 100,000 deployed labels. For detailed information about Pricer's ESL platform see section 2.

Pricer's ESL platform uses a wireless IR-link to communicate between entities, enabling communication speeds up to 38 kbps. The update time of a label is therefore nearly instantaneous and the platform is capable of updating up to 20,000 labels per hour. However, updating 25,000 labels requires a huge amount of data to be transfered through the Pricer ESL platform. The procedure of updating all these labels is critical to the Pricer server, which coordinates the data communication with all labels, but an equally important aspect is the updating procedure of the actual labels. Thus, minimizing the data communication over the wireless IR-link is desirable for all components of the Pricer ESL platform.

How is the Pricer ESL platform affected by minimizing data communication? The first and foremost important motivation is to get increased speed and responsiveness of the Pricer ESL platform. Reducing the amount of data transfered to the labels by the Pricer server enables more computation time to handle other tasks, thus increasing responsiveness. As a side effect this enables the possibility to add more labels into the platform. The second important motivation involves the labels. A reduction of the updating time of a label would yield several benefits. One of the most important effects is preservation of batteries. Increased battery-life of a label implies longer deployment period, and postpones the work of changing batteries.

The focus of this study was applied to a popular Pricer ESL called DM3370. Due to popularity in retail stores a small reduction of data communication per DM3370 would yield a big reduction of data communication of the entire Pricer platform. The objective was to find suitable compression schemes to the Pricer ESL platform by evaluating different lossless compression schemes using a set of images equivalent to the idea of the Calgary Corpus[1].

A suitable compression scheme must compress data well, have low memory consumption and have reasonable encoding and decoding times compared to Pricer's compression scheme. The following questions are relevant and interesting in this study. How well does the Pricer compression scheme work compared to other schemes? Would it be useful to apply different compression schemes to

[1] The Calgary Corpus is a set of files used to benchmark lossless compression algorithms.

different types of images? In other words, does there exist any image patterns where some prototypes perform better than others? What image patterns yield good performance in terms compressibility? In terms of encoding and decoding times?

1.3 Coverage

All algorithms presented and implemented in this master thesis were limited to lossless compression for two reasons. The first reason involved the image size of DM3370; a lossy algorithm would probably destroy the content of an image and make it unreadable. The second reason was the vast number of compression algorithms available. Focusing on lossless compression algorithms narrowed the field of research.

All experiments in this master thesis were restricted to bi-level images since the DotMatrix family uses bi-level images exclusively. An implementation of a compression algorithm was called a prototype, which reflects the stand-alone basic functionality of the implementation. The criteria of a prototype were the abilities to encode, decode and produce valid results. A valid result specified that a compressed data stream could be reversed into the original input data stream. The experiments of the prototypes were limited to the measuring of compression factor, memory consumption and encoding and decoding time.

Three case studies, using real ESL data from a Norwegian retail store, were made to see if the results of the corpus set were representative of the performance in practice. The first case study, called *Partial Images*, evaluated the prototypes with respect to small images representing price updates. The second case study, called *DM3370 Images*, evaluated the prototypes with respect to DM3370 images. The third case study, called *DM110 Images*, evaluated the prototypes with respect to DM110 images.

Thus, this study covers:

- Descriptions of well-known lossless compression algorithms and schemes
- Compressibility studies of bi-level images
- Implementations of valid prototypes with encoding and decoding abilities
- Evaluation of prototypes with respect to compression factor, memory consumption, encoding and decoding times

- Evaluation of prototypes using real ESL data

1.4 Acknowledgements

I thank both my supervisors, Nils Hulth at Pricer and Cris Luengo at Uppsala University, for valuable discussions, suggestions and proofreading. It has been very interesting to make this master thesis at Pricer, not only because of the interesting subject and all the friendly people, but also for the interesting work of Pricer's R&D department related to my education. Additionally, I thank the following people at Pricer (in no particular order):

- Carl-Oskar Larsson for information about Pricer's compression scheme and DM3370 details
- Camilla Brodén for collecting real ESL data
- Jesper Tärnvik for information about the Pricer server
- Lars Andersson and Mikael Yttreus for information about the PPM protocols
- Agneta Grundström-Lindh for the DM3370 specification document

I thank my family for all the encouragement during my studies and for helping me when things have been difficult.

I thank my wife, Johanna, for being tolerant, supportive and helping me with the proofreading of this report. Finally, I thank my daughter, Noa, for all the happy smiles directed at me. Noa felt like doing her own contribution to this report by walking on the keyboard: "2eeerfgtrsraszzzzzzzzzzzZZZZZzzzz fqrwexAS ZZa qZzaq<qa<". I love you both :)

Chapter 2

The Pricer ESL Platform

This chapter gives the required information about Pricer's ESL platform and DM3370.

2.1 System Overview

The Pricer ESL platform is a complete solution for price labelling including infrastructure, software and different label models. An easy to use web-based user interface is used in the platform to help store personnel administrate and update labels. The infrastructure of the platform consists of a Pricer server, base stations and transceivers (TRX).

The Pricer ESL platform has support for several ESL models with different shapes and designs. One supported label family is the Continuum family, which is a segment-based family created for retail environments. Another supported label family is the DotMatrix family, which is a pixel-based family. The DotMatrix family consists of four labels shown in table 2.1. All labels in the Pricer ESL platform use the same underlying infrastructure. A system overview of the Pricer ESL platform is shown in figure 2.1.

The Pricer server runs on a general-purpose computer and the software is written in Java. The server enables an interface between the back-office system[1] in a retail store and the Pricer ESL platform. Price and product information updated by the back-office system are automatically interpreted by the server

[1] A back-office system is an underlying business system used in a retail store.

Model name	DM3370	DM90	DM110	DM200
Housing size (mm)	70 x 33	124 x 60	124 x 80	200 x 145
Display size (mm)	48 x 19.7	93 x 41	93 x 61	162 x 108
Size (pixels)	172 x 72	320 x 140	320 x 192	480 x 320
Resolution (dpi)	92	88	80/88	76

Table 2.1: Specifications of the DotMatrix family.

and appropriate actions are taken to distribute the information to the corresponding labels. The Pricer ESL platform is, in other words, transparent to store personnel.

Information to labels of the DotMatrix family are sent as bi-level images from the Pricer server via TRXs. The information of a label can be updated in the following ways: a complete update or a partial update. A complete update transfers an entire image to a desired label, while a partial update only transfers the region that has been changed in the image since last time.

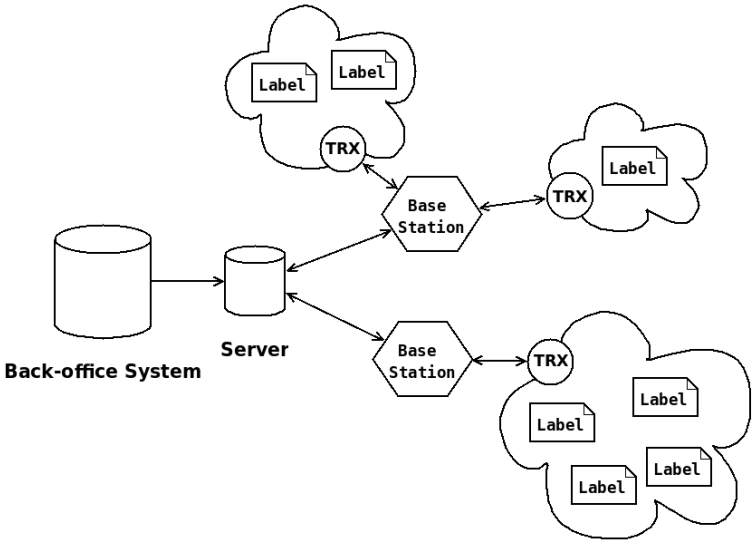


Figure 2.1: System overview of the Pricer ESL platform.

A TRX is a device equipped with several diods to accomplish two-way IR communication with labels. Usually, TRXs are mounted in the ceiling. A TRX device typically serves $500-700$ labels and covers an area of approximately 100 m^2. In order to minimize wireless traffic in the platform a compression scheme is used to reduce the amount of data sent between TRXs and labels. The Pricer

server encodes data and receiving labels decode data.

The Pricer ESL platform uses a diffuse high-speed IR-link to handle wireless communication between TRXs and labels. A diffuse IR-signal is reflected on all surfaces, making it usable without line of sight between two entities. The platform supports two pulse position modulation protocols enabling communication speeds up to 10 kbps and 38 kbps.

A base station is a device connected to the Pricer ESL server via Ethernet network or serial link. Wiring going from the backplane of a base station connects several TRXs, enabling a communication link between the Pricer ESL server and deployed labels. A base station device is able to serve up to 32 TRXs.

2.2 DM3370 Details

The DM3370 model is a member of the DotMatrix family, which is compatible with the Pricer ESL platform. The key components of the label are a microcontroller unit, a Liquid Powder Display (LPD), IR receiver, batteries and plastic housing. The components are sealed together by welded front and rear housing, yielding a total weight of 29 g. An optional snap-on frame, enabling a customizable look of different colors and prints, is attachable to the front housing. The standard design of a DM3370 can be seen in figure 2.2.

Figure 2.2: A standard DM3370 label.

The DM3370 uses a low-power 8-bit RISC microcontroller at 4 MHz with 16 kB ROM, 1 kB internal SRAM and 512 B EEPROM. The device uses an LPD to display information using bi-level images. An LPD continues to display information after the power supply has been turned off, which enables low power consumption. The display area is 47.3x19.8 mm and has 172x72 pixels at a resolution of 92 dpi.

The DM3370 uses three CR2032 (3 V) lithium batteries and the lifetime is essentially dependent on the amount of communication time and number of

display updates made during operation. The device has an overall lifetime of 5 years minimum under standard conditions, which is based on 30 minutes of communication time, 4 price updates per day, 1 full image update per day and 1 information register display per day.

In order to maximize battery lifetime the IR receiver is turned off most of the time, but it is activated every second to see if a frame is available. If a valid frame is available during that time, the receiver will continue to read the transmission and handle the incoming frame appropriately. A transmitted frame contains a cyclic redundancy code, enabling the detection of all single and double errors.

Chapter 3

Data Compression

The first section gives an introduction to data compression and the second section gives an introduction to information theory. The other sections of this chapter describe a set of well-known compression schemes.

3.1 Introduction

Data compression is the application of one or several schemes to a data stream (or source message) with the intention to minimize the size of the data stream [44]. Data compression is desirable because data storage and data communication is expensive in terms of money and time. Data compression schemes are categorized in two classes, namely lossy and lossless methods.

In lossy compression schemes some of the original information of an input data stream is removed in an irreversible way. The decompressed version of the data stream is not same as the original input data. Lossy compression schemes achieve greater compression factors than lossless compression schemes in general. Lossy compression is popular in multimedia applications handling images, movies and music. In the domain of multimedia, unnoticeable data can often be removed without disturbing the experience.

A lossless compression scheme is reversible, which implies that the original data and the decompressed data are identical. Lossless compression is popular in archive formats, e.g. ZIP and RAR.

A compression scheme may use a probability model to encode and decode data.

The model can be created by reading and keeping count of symbols from an input stream. All symbol probabilities can be calculated in one pass of an input stream. Another pass is then needed to do the actual encoding, which is often too slow to be practical. However, a practical compression scheme uses either a predefined model or a dynamic model, where the latter adapts as the input stream is processed.

A compression scheme is called *static* (or non-adaptive) if it does not modify any parameters based on the input data. Static schemes perform well on specific types of data where parameters have been optimized regarding the symbols' probabilities. Other types of data usually yield poor performance. A compression scheme is called *adaptive* (or dynamic) if it modifies the probability model or any parameters during run-time.

In compression schemes, redundancies are removed from an input data stream, yielding reduction in size of the data stream. Many data streams have a structure that can be exploited by representing the data in a more efficient way. However, some types of data streams are not good candidates in terms of compressibility. A data stream of random data is not significantly compressible, because random data have no patterns or redundancies. The same reasoning can be applied to already compressed data. In such a data stream there are no redundancies left to compress, making it nearly incompressible.

Image compression schemes use *spatial redundancy* where a specific pixel is highly correlated with its nearest pixel neighbors. *Context-based* pixel prediction can be used to exploit the high correlation in an image. A pixel's probability is then estimated through a neighborhood context [33]. If a raster scan[1] is performed, the surrounding pixels can be used in an arbitary context shape. An example of such a shape is shown in figure 3.1.

3.2 Information Theory

Information theory is the study of transmission of information between two entities [29]. The entities are often denoted *sender* and *receiver*. All messages, e.g. data packets, pass through a *channel* that connects the two entities. The *alphabet* S of a message is defined by: $S = \{s_1, \ldots, s_q\}$, for $q > 1$, where s_q is a symbol. The information X is a message defined by: $X = X_1 \ldots X_n$, for $n > 1$,

[1] Raster scanning an image is reading the image from left to right and top down, starting at the top left corner.

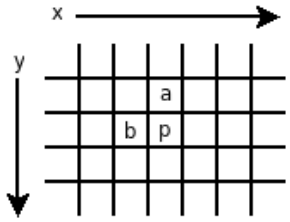


Figure 3.1: An example of a context shape of 3 pixels in an image where p is the current pixel, a is the past pixel row-wise and b is the past pixel column-wise.

where X_n is a symbol and $X_n \in S$. A symbol $X_n \in X$ occurs with a certain probability denoted: $P(X_n = s_i) = p_i$, for $i = 1, \ldots, q$. Like all probability distributions, the probabilities sum up to one $\sum_{i=1}^{q} p_i = 1$.

In 1948, Shannon introduced the concept of *entropy* in information theory [48]. The entropy $H(x)$ is the amount of uncertainty of an event with probability $P(x) = p$. If $P(x) = 1$ or $P(x) = 0$ there is no uncertainty since the outcome of the event is known a priori, giving $H(x) = 0$ in both cases. The entropy of an event, with probability p, is defined by $H(x) = -p \cdot \log_2 p$.

Two independent events have an entropy equal to the sum of the two entropies. The entropy of a sequence $X = X_1 \ldots X_n$ is therefore defined by $H(X) = -\sum_{i=1}^{n} p_i \cdot \log_2 p_i$.

A code C is a set of code words $w_i \in C$ for some $i > 1$. C is *uniquely decodable* if the mapping between the source message X and C is one-to-one [29]. Each $x \in X$ must map to a unique code word $w \in C$ and vice versa. The assumption is that no code word in C is identical. C is denoted prefix-free (or just prefix) code if no code word is a prefix of another code word. The set of prefix-free codes is a subset of the uniquely decodable codes.

The Kraft inequality [31] and McMillan inequality [35] are two inequalities regarding the existence of uniquely decodable codes. The former inequality states that there exists a prefix-free binary code with word lengths l_i, for $i = 1, \ldots, n$ if $\sum_{i=1}^{n} 2^{-l_i} \leq 1$ holds [29]. The latter inequality states that there exists a uniquely decodable binary code with word lengths l_i, for $i = 1, \ldots, n$ if the same condition holds. Thus a code C may satisfy the Kraft-McMillan inequality but still not be a prefix-free code [46]. If a code C does not satisfy the inequality it is guaranteed not to be a prefix-free code. Hence, the Kraft-McMillan inequality is a necessary, but not sufficent condition for the existence of a uniquely decodable

code C.

The average word length L of a uniquely decodable code C has a close relationship to the entropy of a message X [29]. The relationship states that C has a lowest average word length equal to the entropy of X: $L(C) \geq H(X)$. In other words, the entropy is the smallest number of bits needed per symbol to represent X on average and thus the lower bound of lossless compression.

However, a source message has several entropy measures, because the entropy of a message is directly related to the underlying model used by a compression scheme [32]. Thus, changing the underlying model of a compression scheme changes the entropy bound.

3.3 Universal Codes

A universal code encodes integers in some infinite number set to code words [47]. All universal codes in this master thesis are defined for at least $\mathbb{N}$. Universal codes are useful because a compression scheme may output integers as a result. The final stage of some compression schemes needs a way to translate these integers into binary representations that are uniquely decodable. A universal code manages this with a few arithmetic operations.

3.3.1 Elias Codes

In 1975, Elias presented three universal uniquely decodable codes denoted Gamma, Delta and Omega [18]. The idea of the Elias codes is to bound a positive integer n with a lower and upper bound by finding M, such that: $2^M \leq n < 2^{M+1}$ [45]. Hence, n can be defined as: $n = 2^M + L$. The difference between the Elias codes is how they encode M and L. Table 3.1 shows the code words for $0 < n \leq 10$ encoded by the Elias codes.

Elias Gamma Code

The Elias Gamma code encodes a positive integer n with a prefix and a suffix [45]. The prefix is the unary[2] representation of M. The suffix is the binary representation of L expressed in M bits. The resulting code word of n is the concatenation of the prefix and the suffix.

[2]The unary representation of an integer n is n zeroes followed by a one.

n	Gamma	Delta	Omega
1	1	1	0
2	010	0100	100
3	011	0101	110
4	00100	01100	101000
5	00101	01101	101010
6	00110	01110	101100
7	00111	01111	101110
8	0001000	00100000	1110000
9	0001001	00100001	1110010
10	0001010	00100010	1110100

Table 3.1: $0 < n \leq 10$ encoded with the Elias Gamma, Delta and Omega Codes.

Elias Delta Code

In the Elias Delta code, M is encoded in a different manner than the Elias Gamma code, giving shorter code words for large n [18]. A positive integer n is encoded with a prefix and a suffix [45]. The prefix is the Elias Gamma code of M plus one. The suffix is the binary representation of L expressed in M bits. The resulting code word of n is the concatenation of the prefix and the suffix.

Another way of constructing the Elias Delta code is by expressing n with three parts: a prefix, a infix and a suffix [45]. The suffix is the binary representation of n expressed in M bits. The infix is the length of the binary representation of n expressed in binary representation. The prefix is the length of the infix minus one zeroes. The resulting code word of n is the concatenation of the prefix, the infix and the suffix.

Elias Omega Code

The Elias Omega code builts code words recursively by prepending bit lengths of groups to a bit stream [45]. A group is represented by an integer in decimal representation. The first group is n, while the next group is the current group's length in bits. The recursive behavior stops when a group can be represented by two bits. Thus, a positive integer n is encoded by prepending the length of the groups to the binary representation of n. Each group starts with a one and a code word always ends with the binary representation of n followed by a zero.

The Elias Omega code can be seen in figure 3.2.

```
1 Append a zero to the output
2 Start
3 If ⌊log n⌋ = 0, stop
4 Prepend beta(n) to the output
5 n = ⌊log n⌋
6 Goto 2
```

Figure 3.2: The Elias Omega algorithm [18]. $beta(n)$ is the binary representation of n.

3.3.2 Fibonacci Codes

In 1987, Alberto and Fraenkel introduced a family of universal codes based on the well-known Fibonacci series [1]. A Fibonacci code encodes a positive integer as a binary vector using Fibonacci numbers as weights. A Fibonacci code of order m uses the m first Fibonacci numbers to form a Fibonacci series, e.g. the Fibonacci series of order 2 is recursively defined by $F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2}$. Table 3.2 shows the Fibonacci numbers for $1 \leq n \leq 10$.

n	1	2	3	4	5	6	7	8	9	10
F_n	1	1	2	3	5	8	13	21	34	55

Table 3.2: The Fibonacci representation of $1 \leq n \leq 10$

According to Sayood, Zeckendorf's theorem states that any integer can be expressed as the sum of Fibonacci numbers [47]. The Fibonacci codes use Zeckendorf's theorem to make code words uniquely decodable.

The C^1 code in the Fibonacci family is a code of order $m = 2$. Table 3.3 shows the code words for $1 \leq n \leq 10$ encoded by C^1. To encode a positive integer n a code word is formed using a prefix and a suffix. The prefix is the Zeckendorf representation of n, that is n expressed as a binary vector with numbers of the Fibonacci series (order 2) as weights. The suffix is always one. The resulting code word of n is the concatenation of the prefix and the suffix. The column *accumulated weights* in table 3.3 shows the Zeckendorf representation of $1 \leq n \leq 10$.

The C^2 code in the Fibonacci family is a code of order $m = 3$. To encode a

n	Zeckendorf sum	Accumulated weights	Code word
1	$1 \cdot 1$	1	11
2	$0 \cdot 1 + 1 \cdot 2$	01	011
3	$0 \cdot 1 + 0 \cdot 2 + 1 \cdot 3$	001	0011
4	$1 \cdot 1 + 0 \cdot 2 + 1 \cdot 3$	101	1011
5	$0 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 1 \cdot 5$	0001	00011
6	$1 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 1 \cdot 5$	1001	10011
7	$0 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 + 1 \cdot 5$	0101	01011
8	$0 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 8$	00001	000011
9	$1 \cdot 1 + 0 \cdot 2 + 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 8$	10001	100011
10	$0 \cdot 1 + 1 \cdot 2 + 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 8$	01001	010011

Table 3.3: $1 \leq n \leq 10$ encoded with C^1.

positive integer n a code word is formed using a prefix and a suffix. The prefix is the Zeckendorf representation of $n-1$, that is $n-1$ expressed as a binary vector with numbers of the Fibonacci series (order 3) as weights. The suffix is always 011. The resulting code word of n is the concatenation of the prefix and the suffix.

3.3.3 Goldbach Codes

In 2002, Fenwick introduced a universal code called the Goldbach G_0 code [21]. The code is based on the Goldbach conjecture, which states that every even integer is the sum of two primes. A positive integer in G_0 is expressed as the sum of two prime numbers and the code word is a binary vector where each bit is a weight to a prime number. To be able to use the Goldbach conjecture, G_0 maps an arbitrary positive integer N to $2 \cdot (N+3)$, which is encoded to a code word. Every code word of G_0 contains exactly two bits set to one. Thus, the decoding of a code word stops when two bits set to one are found.

3.3.4 Golomb and Rice Codes

In 1966, Golomb presented a family of codes that compress a series of binary events [23]. A binary event can either be a favorable event represented by a one or an unfavorable represented by a zero. A favorable event occurs with the probability p and thus the probability of an unfavorable event is $q = 1 - p$. The

idea is to encode sequences of favorable events as the run-lengths between two unfavorable events. The probability of a run-length n is the known geometric distribution $p^n \cdot q$. A code is determined by the parameter m, defined by $m = -\frac{\log 2}{\log p}$.

A positive integer n is encoded with a prefix and a suffix [44]. The prefix q is coded in unary, and defined by $q = \lfloor \frac{n}{m} \rfloor$. The suffix is the remainder $r = n - q \cdot m$, coded in truncated binary notation. The sequence of remainders is defined by $[0, 1, ..., c]$, where $c = \lceil \log m \rceil$. A truncated binary encoding of the sequence of remainders encodes the first $2^c - m$ remainders with $c - 1$ bits each and the rest of the remainders with c bits each, where the last remainder has a binary value of c ones. The resulting code word of n is the concatenation of the prefix and the suffix.

Table 3.4 shows two different Golomb codes for $0 < n \leq 10$.

n	Code word ($m = 14$)	Code word ($m = 16$)
1	0001	00001
2	00100	00010
3	00101	00011
4	00110	00100
5	00111	00101
6	01000	00110
7	01001	00111
8	01010	01000
9	01011	01001
10	01100	01010

Table 3.4: $0 < n \leq 10$ encoded with Golomb codes with parameters $m = 14$ and $m = 16$

In 1979, Rice introduced a code family with a parameter k [41]. The Rice codes were developed for deep-space telemetry applications where the data consist of numerical values within some small range [47]. A Rice code with parameter k is efficient for n close to 2^k and is close to the binary length (i.e. $\log_2 n$) of n.

A positive integer n is encoded with a prefix and a suffix. The prefix is $1 + \frac{n}{2^k}$ coded in unary. The suffix is $n \, mod \, 2^k$ coded in binary. The resulting code word of n is the concatenation of the prefix and the suffix.

Golomb codes can be seen as a generalization of Rice codes, because the latter

is a subset of the former [47]. A Rice code with parameter k is identical in terms of code words lengths to a Golomb code with parameter $m = 2^k$.

3.4 Run-Length Encoding

3.4.1 Static Run-Length Encoding

Run-Length Encoding (RLE) is a simple compression scheme mentioned in many books about data compression [46, 44, 47]. An RLE scheme counts repetitive symbols, e.g. bytes or bits, of the same value until another value is discovered. The count of a symbol is often refered to as the *run-length* of that symbol. The run-length together with the symbol is appended to the output as a two-tuple. The encoding of the two-tuples varies depending on implementation. Table 3.5 shows compression of different source messages using static RLE.

Golomb codes efficiently encode run-lengths that occur with a geometric probability distribution. See 3.3.4 for the description of the Golomb codes.

Message	Output
12, 12, 12, 12, 35, 76, 112, 5, 5	[(4, 12), (1, 35), (1, 76), (1, 112), (2, 5)]
1, 2, 3, 4, 5, 6, 7	[(1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7)]
1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0	[(8, 1), (6, 0)]

Table 3.5: An example of RLE applied to different source messages.

3.4.2 Adaptive Run-Length Encoding

One adaptive RLE scheme uses simple run-length encoding of zeroes together with an adaptive Golomb code [45]. Every output of a run-length is prepended by a one. The algorithm is adaptive because the Golomb-parameter m changes depending on the input data. A run-length r is encoded using the current parameter m. A new value of m is estimated by accumulating the lengths of the runs processed up to that point.

A pseudo algorithm of an adaptive RLE is shown in figure 3.3.

```
1  L = 0;
2  N = 0;
3  m = 1;
4
5  for each run of r zeroes do
6    construct the Golomb code for r using current m
7    write it on compressed stream
8    L = L + r;
9    N = N + 1;
10   p = L / (L + N);
11   m = ⌊ −1 / log2 p + 0.5⌋;
12 end
```

Figure 3.3: An example of an RLE algoritm using an adaptive Golomb code [45]. L is the total number of zero run-lengths, N is total number of ones, p is the probability of a zero and m is the current Golomb parameter.

3.4.3 PackBits

One variant of RLE is the PackBits scheme developed by Apple [28]. PackBits is used in the TIFF image format [2].

The PackBits scheme uses a header byte to tell how the next data byte should be interpreted. The header byte can state that the next data byte should be read as usual, but it can also state that the next byte was compressed as a two-tuple of a run and a symbol. Table 3.6 shows how the header byte is interpreted.

Header n	Meaning of the following bytes
[0, 127]	The next $n + 1$ bytes are read as usual
[128, 255]	The next byte is repeated $257 - n$ times

Table 3.6: Explanation of the PackBits header format.

3.5 Huffman Coding

3.5.1 Introduction

In 1952, Huffman presented an algorithm to construct optimal variable-length codes for an arbitary probability distribution of a source message [26]. Huffman codes are optimal prefix-free codes, where probable symbols are assigned short code words and less likely symbols are assigned longer code words. When the symbols in a source message occur with probabilities of negative powers of two, the average size of the code words is equal to the entropy of the message [46].

A Huffman code has two important properties [26]. The first property is that all code words are prefix-free. This means that no code word is a prefix of another code word, which yields easy decoding. The second property states that no redundent information is used in terms of delimiters in the code words. These properties are enforced by using a binary tree to represent a Huffman code. A node is either a leaf node or a parent to two nodes. Each leaf node is composed of a unique source symbol and the corresponding probability or frequency of that symbol.

A code word of a specific symbol is a traversal in the tree from the root node to the leaf node containing the requested symbol. A code word is constructed by appending ones and zeroes to a binary string. A zero is appended if the left path is taken and a one is appended if the right path is taken.

3.5.2 Static Huffman Coding

The static Huffman algorithm is a two-pass job [46]. The first pass is to accumulate the frequency of each symbol in the source message. The second pass is to construct an actual Huffman code and apply it to the source message.

To construct a Huffman code the probabilites of the source symbols should be sorted in descending order in a queue. Each source symbol is added to the queue as a root node with no children. In each iteration, the last two root nodes, those with lowest probability, in the queue are added as children to a new node. The new node's probability is the sum of the probabilities of its children. The number of root nodes in the queue is now decreased by one since two nodes were moved. At the end of each iteration the root nodes are sorted in descending order again. The algorithm is greedy and stops when only one root

node exists in the queue. An example of the construction of a Huffman code can be seen in appendix A.

A Huffman algorithm using a min-heap can be seen in figure 3.4.

```
1 n = |C|
2 Q = C
3 for i = 1 to n − 1
4   allocate new node z
5   left[z] = x = EXTRACT-MIN(Q)
6   right[z] = y = EXTRACT-MIN(Q)
7   f[z] = f[x] + f[y]
8   INSERT(Q, z)
9 return EXTRACT-MIN(Q)
```

Figure 3.4: A Huffman algorithm using a min-heap [12]. C is the set of symbols, where $c \in C$ has a frequency $f[c]$, a left child $left[c]$ and a right child $right[c]$.

3.5.3 Adaptive Huffman Coding

There are two significant issues with a static Huffman implementation [46, 49]. The first issue is that it requires two passes over the input data. The second issue is that the Huffman tree must be sent to the decoder or be known in advance to both the encoder and decoder. An adaptive Huffman algorithm builds the Huffman tree on-the-fly in one pass. Not recently seen symbols are added as new nodes and previously seen symbols have their frequency counts updated. To ensure unique decodability in form of prefix-free code words in the Huffman code, structural changes may be required to the Huffman tree.

The FGK Algorithm

Faller [20] and Gallager [22] independently developed an adaptive Huffman algorithm [47, 46]. This algorithm was later improved by Knuth [30]. Hence, the algorithm is called *FGK* after the first letter in the names of its inventors. The algorithm states that a binary code tree must satisfy the *sibling property* to be a legal Huffman code [22]. Two nodes that have the same parent node are called siblings. A binary tree statisfies the sibling property if each node, except the root node, has a sibling and if the frequencies of all nodes can be

ordered non-increasingly in a list where siblings are next to each other. In order to satsify the sibling property the Huffman tree may be rearranged whenever a new symbol appears or if a symbol's frequency count is increased.

As in the case of static Huffman coding, FGK traverses the tree when a new symbol is read from the source message [47]. If the symbol is not in the tree an uncompressed variant of the symbol is written to the output, which means that the decoding phase is able to continuously build the Huffman tree also. The symbol is added to the tree and the frequency count updated. The tree may be rearranged at this stage. The next time the symbol appears a node exists in the tree with a specific code word. The code word is obtained in the same way as in static Huffman coding, that is by traversing the path from the root to the node of the symbol.

FGK produces at worst an output of $2 \cdot S + t$ bits, where S is the output length of the static Huffman algorithm and t is the symbol count of the source message.

Vitter's Algorithm Λ

In 1987, Vitter presented an algorithm called Λ that is based on FGK [49]. The algorithm uses a node ordering scheme called *implicit numbering*, reflecting the visual appearance of a tree, which grows downward from the root. The nodes of a tree are ordered in level by level in increasing order from the bottom up, meaning that nodes on one level have lower numbers than the nodes on the next level. Nodes on the same level are ordered in increasing order from left to right. The algorithm uses an invariant to minimize the sum of all code word lengths and to minimize the maximum length of each code word. The invariant states that for each weight w, all leaves of weight w precede all the internal nodes of the same weight. A data structure called *floating tree* is used to represent the Huffman tree in Vitter's algorithm. This data structure can enforce the invariant in real time.

Vitter's algorithm Λ produces at worst an output of $S + t$ bits, where S is the output length of the static Huffman algorithm and t is the symbol count in the source message.

3.6 Facsimile Compression

In the late 1970s the Consultative Committee for International Telephone and Telegraph (CCITT) developed several standards for compression of facsimile documents [34]. The compression technique used today in fax machines is the one proposed by the CCITT Group 3 called T.4. A facsimile document is a digital version of a scanned document represented by a bi-level image with a width of 1,728 pixels [46]. The T.4 standard includes two compression schemes, i.e. modified Huffman and modified READ, and a document is scanned from left to right and row by row.

Modified Huffman (MH) is a one-dimensional coding scheme that compresses run-lengths with a simple RLE scheme combined with a set of Huffman code words. Each line in a document is encoded separately to increase the ability to handle errors. An encoded run-length represents either a complete white or black run. Run-lengths alternate between white and black and a white run-length is assumed at the beginning of each line. The maximum run-length is 1,728. Every line is followed by an End Of Line (EOL) code word. The use of an EOL code word enables resynchronization if errors occur.

The code word of a run-length is composed of a *terminating* code word and a *make up* code word. Small run-lenghts in the range $[0, 63]$ are encoded with terminating code words only. Large run-lengths in the range $[64, 1728]$ are encoded with a make up code word followed by a terminating code word. The run-length r can be defined by $r = 64 \cdot m + t$, where m is the make up run-length and t is the terminating run-length [47].

The other scheme proposed in T.4 is the Modified READ (MR) [34], which is a modification of the Relative Element Address Designate (READ) scheme [53]. MR is a two-dimensional coding scheme for facsimile documents. The scheme was developed to accommodate the complicated characters of Japanese writings. Such writings require high-density scanning and the two-dimensional scheme exploits the vertical correlations in these characters better than MH.

MR uses different coding states depending on the relationship between five positions (a_0, a_1, a_2, b_1 and b_2) located on the current coding line and a reference line [27]. Initially the reference line is a line with only white pixels. The reference line is updated successively to the last line processed.

The position a_0 is the reference pixel or the starting element of the current coding line. The position a_1 is the start of the run-length next to a_0. The

position a_2 is the start of the run-length next to a_1. Hence, the positions a_x are located on the current coding line. The position b_1 is the start of the run-length, colored as a_1, next to a_0, but on the reference line. The position b_2 is the start of the run-length next to b_1 on the reference line. Hence, the positions b_x are located on the reference line.

The three coding states used in MR are called *pass mode*, *vertical mode* and *horizontal mode*. Pass mode is applied if b_2 is less than a_1. The output is the pass mode code word appended by the code word of the length b_1b_2. Vertical mode is applied if the horizontal length between a_1 and b_1 is less than three units. The output is an individual code word depending on the horizontal length. Horizontal mode is applied if vertical mode fails to encode the position of a_1. The output is the horizontal mode code word appended by the code words of the lengths a_0a_1 and a_1a_2.

3.7 Arithmetic Coding

Elias presented the idea of arithmetic coding in the early 1960s [4, 52, 43, 32, 39, 46]. Practical methods of arithmetic coding were introduced by Pasco [39] and Rissanen [42] in 1976.

As mentioned in 3.5.1, Huffman coding can produce code words with average length equal to a source message's entropy, but only if the probabilities of the symbols are negative powers of two [4]. Arithmetic coding proposes another way to compress a source message close to the entropy. Instead of assigning fixed variable-length codes to every symbol, as in Huffman coding, a real number is assigned to the whole source message. Where Huffman assigns code words of integral bits, arithmetic coding assigns fractions of bits to symbols yielding more efficient compression in some cases.

An arithmetic coder has a probability model that is static or adaptive. The latter updates the probability model continuously when new symbols are read from the source message [4]. One advantage of an arithmetic coder is the ability to separate the underlying model and the coding algorithm [47]. Hence, various models can be tested without the need to change the coding algorithm. In the static Huffman algorithm the complete code tree needs to be reconstructed if the model is changed.

A source message compressed using arithmetic coding is a real number in the

interval [0, 1) [52]. The distribution of the symbols' probabilities are calculated using a specific model. By iteratively partitioning the initial interval, i.e. [0, 1), relative to the symbol probabilities a narrower interval is achieved. Symbols that appear with higher probability reduce the interval less than symbols with lower probability and therefore fewer bits are added to the output when a highly probable symbol is processed. At the end of the last iteration any number within the current interval will represent the source message.

Table 3.8 shows an example of how the source message *hello* can be encoded using a fixed probability model, which is specified in table 3.7. The *CACM* implementation is regarded as the de facto standard implementation of arithmetic coding [36]. This particular implementation refers to state variables [52] L_n and H_n to hold the interval of the n^{th} symbol processed. L_n is defined by: $L_n = L_{n-1} + (H_{n-1} - L_{n-1}) \cdot L(S)$, and H_n is defined by: $H_n = L_{n-1} + (H_{n-1} - L_{n-1}) \cdot H(S)$, where $L_0 = 0$, $H_0 = 1$, $H(S)$ is the upper interval of a symbol S and $L(S)$ is the lower interval of a symbol S .

Symbol S	Frequency	Probability	Interval	$L(S)$	$H(S)$
e	1	0.2	[0.8, 1.0)	0.8	1.0
h	1	0.2	[0.6, 0.8)	0.6	0.8
l	2	0.4	[0.2, 0.6)	0.2	0.6
o	1	0.2	[0.0, 0.2)	0.0	0.2

Table 3.7: The fixed probability model for the example in table 3.8.

3.8 Dynamic Markov Compression

The Dynamic Markov Compression (DMC) scheme introduced in 1995 by Cormack and Horspool [11] uses a probability model based on Markov chains and an arithmetic coder to achieve compression.

A Markov chain is a stochastic process where the outcome of a specific experiment influences the outcome of the next experiment [24]. The process is defined by a set of states: $S = s_1, s_2, ..., s_r$ where one of the states is the starting state. The process moves from a state s_i to a state s_j with a probability of p_{ij}. The decision to move from s_i to s_j is only based on the current state, thus no previous states are involved. A Markov chain can be fully described by a finite state machine [9].

Message	Current interval	L and H
h	[0.0, 1.0)	$L = 0.0 + (1.0 - 0.0) \cdot 0.6 = 0.6$
		$H = 0.0 + (1.0 - 0.0) \cdot 0.8 = 0.8$
he	[0.6, 0.8)	$L = 0.6 + (0.8 - 0.6) \cdot 0.8 = 0.76$
		$H = 0.6 + (0.8 - 0.6) \cdot 1.0 = 0.8$
hel	[0.76, 0.8)	$L = 0.76 + (0.8 - 0.76) \cdot 0.2 = 0.768$
		$H = 0.76 + (0.8 - 0.76) \cdot 0.6 = 0.784$
hell	[0.768, 0.784)	$L = 0.768 + (0.784 - 0.768) \cdot 0.2 = 0.7712$
		$H = 0.768 + (0.784 - 0.768) \cdot 0.6 = 0.7776$
hello	[0.7712, 0.7776)	$L = 0.7712 + (0.7776 - 0.7712) \cdot 0.0 = 0.7712$
		$H = 0.7712 + (0.7776 - 0.7712) \cdot 0.2 = 0.77248$
hello	[0.7712, 0.77248)	

Table 3.8: An arithmetic coding example showing how the source message *hello* is read character by character (see column *Message*) and how the current interval (see column *Current interval*) is narrowed according to the symbols' probabilities. The output is a real number in the interval [0.7712, 0.77248).

The assumption of DMC is that the input data stream can be generated by a discrete Markov chain model [11]. A generated Markov chain model of the first symbols of a source message can predict future symbols of the message. The Markov chain model provides the probability of every possible symbol, which is encoded by an arithmetic coder.

3.9 Lempel-Ziv-Welch Compression

A *dictionary-based* compression algorithm builds strings of symbols and maps them to code words using a dictionary [47]. The dictionary is either static or dynamic, where the latter grows during run-time. A static dictionary must be sent to the decoder or be known in advance to both the encoder and decoder, while a dynamic dictionary is built by both the encoder and decoder in a synchronized way. Dictionary-based compression algorithms are popular and used in well-known file formats: PNG [25], GIF [6] and ZIP [40].

In 1984, Welch developed a dictionary-based compression algorithm named *LZW* based on the concept of *LZ77* [54] and *LZ78* [55]. The concept of the latter algorithms were introduced by Lempel and Ziv in 1977 and 1978 respectively. LZW and its predecessors compress characters and exploits redundancy

of character frequencies, character reptitions and frequent character patterns [51].

LZ77 uses a sliding window technique of previously seen characters as a dynamic dictionary [46]. The term sliding window is used because a finite buffer slides over the input data. One part of the window is a search buffer that forms the current dictionary. Another part of the window is a look-ahead buffer of yet not compressed characters. In LZ77 the first character in the look-ahead buffer is searched for in the whole search buffer in a reverse manner. Every time a character match is found a string match between the two buffers is performed in a forward manner. The longest match is used as output by the encoder. The output from the algorithm is 3-tuples, where each 3-tuple is the character offset, the match length and the next character in the look-ahead buffer.

LZ78 does not use a sliding window, but only a dynamic dictionary where previously found strings are stored [46]. The limit of the dictionary is the amount of memory available. Whenever a new symbol is read from the input character stream the dictionary is searched with an accumulated string. If a match is found a new character is read and appended to the accumulated string. If no match is found a new dictionary entry is added with a 2-tuple consisting of the index of the current accumulated string and the last character read. An entry in the dictionary represents the concatenation of the string at the given index and the charater.

In LZW, and its predecessors (LZ77 and LZ88), variable-length strings are mapped into fixed-length code words [51]. LZW manages this by having pointers into a dictionary table initialized with all strings of length one. Hence, there is no need to have the 2-tuple used in LZ78, but only the index to the dictionary is enough. Fixed-length code words of 12 bits, and thus maximum dictionary size of $2^{12} = 4{,}096$ entries, are common in practical implementations of LZW.

A string in the dictionary has the form ωK, where ω is some string and K is a single character. LZW has a prefix property, which states that for each string ωK in the dictionary its prefix ω is also in the dictionary. The algorithm is initialized by populating the dictionary with all occurrences of strings with length one, implying that ω is the empty string. A parsed character from the source message is denoted K.

The dictionary is searched for the longest match of the string ωK. If a match is found a new character is read from the source message, yielding $\omega K \rightarrow \omega$. If no match is found the code word of ω is added to the output stream, the

accumulated string ωK is added to the dictionary and K becomes the first character of the next string, yielding $K \rightarrow \omega$. These steps are repeated until the whole source message is parsed.

By adding a character at the time, the algorithm adapts slowly to the source message and thus makes it possible to achieve good compression for large source messages [46]. A pseudo algorithm of LZW can be seen in figure 3.5.

```
Initialize dictionary to contain all single-character strings
Read first input character → ω

Step:
    Read next input character →K

    If input is exhausted
        code(ω) → output
        exit

    If ωK exists in dictionary
        ωK → ω
        goto Step
    Else ωK not in dictionary
        code(ω) → output
        ωK → dictionary
        K → ω
        goto Step
```

Figure 3.5: A pseudo version of the LZW algorithm [51].

3.10 The Burrows-Wheeler Transform

In 1994, Burrows and Wheeler presented a block-sorting algorithm, now called the Burrows-Wheeler Transform (BWT), originally discovered in 1983 [5].

The algorithm processes a source message in substrings of some predefined length, called blocks, rather than one symbol at the time. The BWT algorithm does not compress the source message, but instead it permutes the bytes in each block to increase the probability of finding a byte close to another byte with

the same value, making it easy to compress with simple compression schemes. According to the authors of BWT a few kilobytes of block size is preferable to achieve good compression.

The BWT algorithm using a block size of N bytes creates N cyclic rotations of an original block S that are rows in a matrix M. The rows in M are sorted in lexicographical order and the last byte of each row is concatenated to a string L. The output of the transformation is L and I, which is the index of S in M. Table 3.9 shows an example of BWT applied to the source message *swiss_miss*. The output is *swm_siisss* and index 8.

Index	Rotations	Sorted rotations
0	swiss_miss	_missswiss
1	wiss_misss	iss_misssw
2	iss_misssw	isssswiss_m
3	ss_missswi	missswiss_
4	s_missswis	s_missswis
5	_missswiss	ss_missswi
6	missswiss_	sssswiss_mi
7	issswiss_m	sswiss_mis
8	sssswiss_mi	swiss_miss
9	sswiss_mis	wiss_misss

Table 3.9: BWT applied to the source message *swiss_miss* [46].

Why do strings transformed by BWT compress well? Burrows and Wheeler give the following explanation. Imagine a large block of English text where a common word is *the*. When BWT is applied to the block, many sorted rotations will start with the word *he*. If the word *the* is frequent many of the rotations will also end with *t*. Thus, one region of the string L will contain a large number of the character *t*.

The reverse transformation of BWT is based on that any column of M is a permutation of the original block S. The procedure of reversing a block is started by creating the first column of M by sorting L lexicographically. A matrix M' is defined by rotating the blocks in M one step to the right. A vector T holds the mapping between a row j in M' into M: $M'[j] = M[T[j]]$. For each character $i = 0, ..., N-1$, $F[i]$ and $L[i]$ hold the first and last byte of $M[i]$. Using the vector T, F and L has the mapping: $F[T[j]] = L[j]$. Hence, $F[I]$ and $L[I]$ hold the first and last byte of the original block S respectively. The predecessors

each byte is given by: $S[N-1-i] = L[T^i[I]]$, for each $i = 0, ..., N-1$ and where $T^0[x] = x, T^{i+1}[x] = T[T^i[x]]$. Thus, the original block S is restored in a reverse manner.

3.11 Move-to-Front Compression

The idea of the Move-to-Front (MTF) algorithm was introduced by Bentley, Sleator, Tarjan and Wei [3] and independently discovered by Elias [19].

The MTF algorithm is denoted as locally adaptive since it has the ability to adapt to local occurrences of frequent source symbols [3]. This behavior is achieved by moving symbols to the front of an alphabet list, allowing frequently occuring symbols in some local domain to have a position near the front of the list.

The basic variant of MTF holds a queue of alphabet symbols [44]. Whenever a symbol is read from the source message the current symbol is searched for in the queue. The current symbol is encoded by its position in the queue and the position may be encoded using a variable-length code. The alphabet queue is altered by moving the current symbol to the front of the queue. These steps are repeated until there are no symbols left in the source message.

Table 3.10 shows an example of MTF applied to the source message *abaacabad* using the alphabet queue [*a, b, c, d*]. The output is [*0, 1, 1, 0, 2, 1, 2, 1, 3*].

Message	Alphabet queue	Position queue
a	[a, b, c, d]	[0]
ab	[b, a, c, d]	[0, 1]
aba	[a, b, c, d]	[0, 1, 1]
abaa	[a, b, c, d]	[0, 1, 1, 0]
abaac	[c, a, b, d]	[0, 1, 1, 0, 2]
abaaca	[a, c, b, d]	[0, 1, 1, 0, 2, 1]
abaacab	[b, a, c, d]	[0, 1, 1, 0, 2, 1, 2]
abaacaba	[a, b, c, d]	[0, 1, 1, 0, 2, 1, 2, 1]
abaacabad	[d, a, b, c]	[0, 1, 1, 0, 2, 1, 2, 1, 3]

Table 3.10: An example of MTF applied to the source message *abaacabad*. The column *Position queue* shows the current queue of symbol positions.

Chapter 4

Materials and Methods

4.1 Data

The material used in the evaluation stages were promotion images created for DM3370. In total, twelve bi-level images with a size of 172x72 pixels were used to form a corpus set.

All images in the corpus set can be seen in figure 4.1. The corpus set covers a wide variety of images that possibly could be used in the Pricer ESL platform. The majority of the images show information of interest to customers, e.g. *dove* and *samsung*. Three images, i.e. *chateau_page2*, *chateau_page3* and *unlinked*, show information of interest to store personnel.

Three case studies were conducted. All images in the case studies were real data used in the Pricer ESL platform of a Norwegian retail store, specifically in their DM3370 and DM110 labels.

In case study *Partial Images*, 3,757 images with a size of 88x36 pixels were used. The images were product prices cropped[1] from DM3370 data. In case study *DM3370 Images*, 7,498 images with a size of 172x72 pixels were used. In case study *DM110 Images*, 83 images with a size of 320x192 pixels were used.

[1] Cropping is the process of removing unwanted information from an image.

(a) absinth (b) chateau_page1 (c) chateau_page2

(d) chateau_page3 (e) dove (f) hand

(g) kellogs (h) peanuts (i) refill

(j) samsung (k) unlinked (l) vante

Figure 4.1: The images of the corpus set.

4.2 Evaluation Details

4.2.1 Overview

The purpose of the evaluation stages were to quantify the performance of the prototypes in different ways. The prototypes were written in Java and evaluated in two stages using a Dell XPS M1210 laptop (Intel Core2, 2.0 GHz CPU, 1.0 GB RAM).

Evaluation stage number one contained 12 prototypes, and they were evaluated based on compression factor, encoding time and decoding time. The purpose of the first stage was to narrow down the number of prototypes to the ones that did well in terms of compressibility.

Evaluation stage number two contained the top six best prototypes of the first evaluation stage in terms of compressibility. The prototypes in the second stage were subject to studies of memory consumption.

4.2.2 Measured Parameters

Compression Factor

The compression factor is a measure of how much a compression scheme is able to reduce the size of a data stream. A compression factor greater than 1.0 means that compression was achieved and a compression factor less than 1.0 indicates negative compression, i.e. the output data was bigger in size than the input data. A compression factor equal to 1.0 implies that no reduction nor expansion of the data was obtained.

The compression factor F is defined by $F = \frac{|x_{in}|}{|x_{out}|}$, where $|x_{in}|$ is the length of the input binary data and $|x_{out}|$ is the length of the output binary data of a prototype.

Encoding and Decoding Time Measurements

The time measurements of the encoding and decoding phases of a prototype were conducted by repeating the same experiment 100 times per image. An experiment consisted of starting the timer, starting the prototype and stopping the timer, where the time elapsed was the result of a timing experiment. The result of a prototype experiment using a specific image was the average value of the timing experiments.

Figure 4.2 shows a pseudo code of the program used for the timing experiment of the encoding phase of prototype i using 12 images and 100 repetitions. The timing experiment of the decoding phase was done analogously.

```
1    t_1 = ... = t_12 = 0
2
3    for x_k ∈ images
4        sub_time = 0
5
6        do 100 times
7            timer = Start_timer()
8            Encode(x_k)
9            sub_time = sub_time + (End_timer() - timer)
10       end
11
12       t_k = sub_time / 100
14   end
```

Figure 4.2: Timing experiment of the encoding phase of prototype i. The current image is x_k and t_k is the encoding time of that image.

Memory Consumption

The memory consumption was measured by analyzing the prototypes' decoding functions in terms of statically allocated arrays. If applicable, a prototype's decoding function written in C was analyzed.

In a microcontroller, constant arrays end up in ROM and static arrays end up in SRAM [38]. Due to the tight memory resources of DM3370 (section 2.2) and due to the aforementioned memory organization, it was natural to split this evaluation parameter into *ROM* and *SRAM* parts.

If a prototype used statically allocated arrays the evaluation was done by multiplying the array size by the array's element size, for each allocated array. The sizes of the primitive data types followed the ANSI C standard, i.e. a *char* was 1 byte, a *short* was 2 bytes and an *int* was 4 bytes.

Some of the prototypes implemented in this master thesis used Java's *StringBuffer* and *Arrays.fill()* for design and performance issues. However, no Java data structures, except the knowledge of the number of nodes of a data structure, were part of the memory consumption evaluation.

4.2.3 Motivation of Parameters

There is no doubt that one of the most important aspects of a prototype is the ability to compress data well. However, a prototype with high compressibility may not be suitable in an ESL system if it takes too long to decompress the data. The explanation is that a label's awake time is related to its deployment period. The more awake time a prototype has, the less battery-life time it will have, yielding a shorter deployment period.

If a label spends more time doing compression computations than it takes to send raw data nothing is gained by having a compression scheme. Therefore, a balance between compressibility and computation time is needed. Due to this, several evaluation parameters were needed. However, a reduction of data is probably preferable to the platform as a whole.

The compression factor parameter was chosen because of its simplicity and easily interpreted results. The encoding and decoding time parameters were chosen to see the relative timings between the prototypes and the current compression scheme in the Pricer ESL platform. The memory consumption parameter was chosen to see if a prototype met the memory limitations. In summary, the chosen parameters were necessary to reflect the limited resources of DM3370.

4.3 Implementation Details

4.3.1 Overview

An implementation of a compression scheme is called a *prototype*, reflecting the basic functionality of an implementation. A prototype fulfilles two criteria: the first criterion requires the prototype to be fully runnable with respect to the encoding and decoding phases, the second criterion requires the prototype to produce valid results. An explanation of what a valid result is and how the prototypes were validated can be read in section 4.4.

Algorithm	Specification	Source code
Elias Gamma	[46]	-
Golomb RLE	[23]	[16]
PackBits	-	[17][2]
LZW	-	[15, 8]
LowPac	[32]	[37][2]
Vitter	-	[50][1]
MTF	[3]	-
BWT	[5]	[14]
DMC	-	[10][2]
Facsimile	[44]	[7][2]

[1] Originally in Pascal, but was reimplemented in Java.

[2] Originally in C, but was reimplemented in Java.

Table 4.1: References to the algorithms used by the prototypes.

The input data used by the prototypes were fed as a continuous bit stream in a raster scan format, where each bit represented a bi-level pixel value. Prototypes based on character compression used alphabet extension [32] to read input data. In this case, alphabet extension was based on reading a binary input stream in substrings to create a suitable context. Character-based prototypes read eight bits at a time in blocks of 2x4 pixels and formed a character of those bits.

Table 4.1 shows all algorithms that the prototypes were based on. A reference in the specification column describes the algorithm in detail and/or provides a pseudo implementation of the algorithm. A reference in the source code column gives a full implementation of the given algorithm in a programming language. A prototype with a specification reference only was implemented from scratch based on the reference given in the specification column. A prototype with a source code reference only was reimplemented using the supplied source code. A prototype with references in both columns was based on the specification reference together with pieces of source code from the source code reference.

4.3.2 Prototype Details

A total of twelve prototypes were implemented for this master thesis. This section covers implementation aspects of the prototypes. The theory of each prototype is covered in section 3.

Details of PackBits, Vitter and DMC are not covered in this section, because those prototypes were direct reimplementations of the source codes specified in section 4.3.1 (table 4.1). The reader is refered to the references for implementation details.

Golomb RLE

In the Golomb RLE, or G.RLE, prototype sequences of zeroes were encoded using a Golomb-Rice code with parameter $m = 2$. Occurrences of ones were not encoded, but a one was always appended after each sequence of zeroes. The parameter m was chosen by estimating the probabilitiy of a zero pixel in the corpus set.

LZW

In the LZW prototype indices of the dictionary were appended to the output stream using a variable-length code. Indices of the dictionary were written using binary notation in as many bits as specified by a code word length parameter.

Initially the length parameter was set to 9 bits, but it was increased whenever the last index exceeded the maximum bits implied by the length parameter. The maximum code word length was 10 bits, implying that indicies greater than 2^{10} would not be uniquely decodable, because several code words would then have the same least significant bits. Hence, dictionary reinitialization was not done in the event of reaching a full dictionary. The prototype used a hash table as dictionary, where each entry was a pair of a string and an integer, holding the cumulative string and its index.

LowPac

The LowPac prototype was a low precision implementation of an arithmetic coder suitable for systems with simple hardware properties, e.g. systems without parallel multiply/divide hardware. The original version of LowPac, written by Neal, was based on the CACM implementation [52].

LowPac used a context-based probability model adapted for bi-level images together with an arithmetic coder based on integers and shift operations rather than floating point numbers and multiplication operations. A context was defined as an integer created from a number of pixels within the defined neighbor-

hood, which was based on trial-and-error and ideas by Langdon and Rissanen [32]. A context was created from 3 pixels, yielding a total of 2^3 possible combinations.

If the current pixel processed had the position P(x, y) in a matrix interpreted input bit stream, the context was represented by the pixels with positions P(x + 1, y - 1), P(x, y - 1) and P(x - 1, y). The probability of a pixel was estimated on the frequency count of the specific context for that pixel. A precision of 6 bits to represent frequencies was used.

MTF

The MTF prototype used an alphabet array consisting of all eight bit characters initially sorted in lexicographical order. Each character from the input data stream was searched in a sequential order from the array. The position of a matched character, denoted *pos*, was encoded using the Elias Gamma code. All elements from position 0 to *pos* were shifted one position to the right in the array, i.e. array[x] was moved to array[x+1]. The final step was to set first element in the array to the matched character.

BWT

The BWT prototype was a combination of BWT followed by MTF as final coder. Input data was first transformed with BWT and then compressed with MTF. Characters were formed from the input binary stream by reading 8 bi-level pixels at a time. The block size was configurable and was always set to the maximum character length of the input data. Hence, a DM3370 image with 172x72 pixels had a block size of 1,548 characters and a DM110 image with 320x192 had a block size of 7,680 characters.

Rotated blocks were represented by an array where each element held the starting point of a rotated block into the original block. The end point of a rotated block was represented by modulo arithmetic, thus wrapping around at the array boundary and creating a circular array. The rotated blocks were sorted by applying radix sort using the first two characters in each rotated block as key. Each radix sort bucket was also sorted using mergesort, which has a complexity of $\Theta(n \cdot \log n)$.

MH

The MH prototype was an implementation of the one-dimensional method introduced by the T.4 standard. Run-lengths were extracted from an input bitstream and encoded using the modified Huffman code supplied by the Facsimile standard.

MH used two encoding lookup tables for the modified Huffman code, consisting of black run-length code words and white run-length code words. No maximum run-length existed other than when end of file was reached.

MH used two lookup tables to be able to decode a code word into a black or white run-length. Code words were always decoded as a white run-length followed by a black run-length. The decoding phase read the maximum code word length of each type and decoded the run-length using the lookup table for that type. The maximum black and white code word lengths were 13 and 8 respectively. Hence, the black and white lookup table were of sizes 2^{13} and 2^8 respectively.

MR

The MR prototype was an implementation of the two-dimensional method introduced by the T.4 standard. Run-lengths were encoded using the MH method. Additionally, MR used another set of Huffman codes to express the different coding states. The new set of code words was a subset of the modified Huffman code words, yielding unique decodability of all code words. The input bit stream was interpreted as a matrix with a maximum run-length equal to the width of the input image.

Whenever MR encoded a state the corresponding code word was appended to the output stream. The pass and vertical coding states were complete by their respective code words, but the horizontal coding state was followed by two run-length code words. A lookup table was used to decode a specific coding state. The maximum code word length of the state code words was 7, yielding a lookup table of size 2^7.

FEG

FEG was a combination of the well-known Facsimile standard and the Elias Gamma code, which is a combination not seen in any research papers, but

rather an idea from the author of this master thesis. The prototype was called FEG because of first letters in the combined schemes: Facsimile Elias Gamma.

The FEG prototype was based on the MR method combined with the Elias Gamma code. The encoding and decoding phases from MR were used, but run-lengths were encoded using the Elias Gamma code instead of the modified Huffman code. As in MR, a lookup table was used to decode a specific coding state. Due to the Elias Gamma code no other lookup tables were needed.

4.4 Validation Details

All prototypes were subject to a validation step to guarantee correct results. If a prototype passed the validation step it was assumed to produce correct results in all cases.

The prototypes were validated using the fundamental property of lossless compression, which states that encoded data is reversible and no information is lost during the encoding and decoding phases. Let $E(x_{in})$ denote the encoding function and $D(x_{out})$ denote the decoding function of some input data x_{in} and some output data x_{out}. A prototype that passed the validation step satisfied: $x_{in} = D(E(x_{in}))$.

Chapter 5

Results and Discussion

5.1 Evaluation Stage One

In the first evaluation stage, 12 prototypes were evaluated, using the corpus set, with respect to compressibility, encoding time and decoding time.

Compressibility

As seen in figure 5.1, Pricer and G.RLE were the only prototypes that produced negative compression for at least one image of the corpus set. The maximum compression factor 3.53 was achieved by LowPac and the minimum compression factor 0.63 was obtained by G.RLE.

Prototypes optimized for bi-level images, i.e. LowPac, MR and FEG, performed better than the other prototypes in almost all cases of the corpus set. Especially good results were achieved by LowPac, yielding best compression factor in all cases of the corpus set. LowPac achieved $52 - 220\%$ increased compressibility compared to Pricer, resulting in a 108% increased average compression factor. LowPac also achieved an average compression factor of 2.91, which is a 25% improvement over the next best prototype, namely MR.

LZW, Vitter, MTF and BWT were designed to compress general data, but they actually performed better than the Pricer prototype in all cases of the corpus set. This suggests that alphabet extension can be used to achieve reasonable compressibility results on small bi-level images. However, other general-purpose

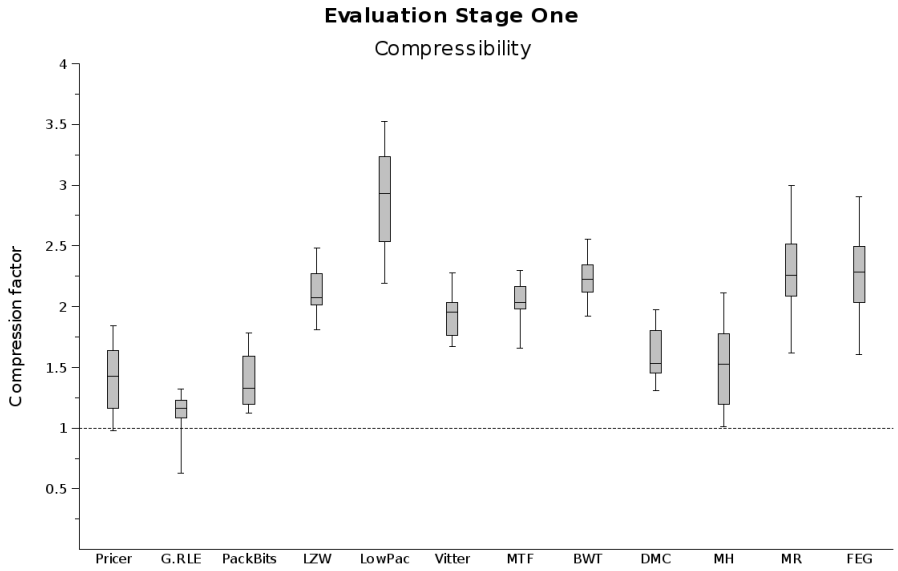


Figure 5.1: Compressibility result of evaluation stage one (12 images of size 172x72 pixels) shown with boxplots, representing the 25th, 50th and 75th percentiles, and the T-bars indicate minimum and maximum sample values. A reference line ($y = 1.0$) shows the upper bound for negative compressibility.

compression schemes, e.g. PackBits and DMC, did not achieve any remarkable improvements over the Pricer prototype.

An interesting observation in figure 5.1 is the similar results of MR and FEG. Both prototypes use the Facsimile MR method, but with different run-length coders; MR uses Huffman coding and FEG uses the Elias Gamma code. A facsimile document, which typically has an image width of 1,728 pixels, is more likely to generate longer run-lengths than DM3370 and DM110 images, which have image widths of 172 and 320 pixels respectively. The similarity of the results are probably due to the use of small images instead of real facsimile documents. According to the results, it seems that applying a Huffman code to short run-lengths does not pay off compared to using a universal code like the Elias Gamma code.

Langdon and Rissanen proposed a context-based arithmetic coding scheme with a context of 10 pixels [32]. Their compression scheme achieved a $20 - 30\%$ improvement of compressibility compared to the Facsimile MR method when applied to the CCITT test documents. That result is similar to the result

of LowPac and MR when applied to the corpus set in this study. The LowPac prototype achieved a $13-57\%$ improvement, yielding an increased average compression factor of 25% compared to MR.

As seen in table 5.1, a count of 6 and 5 prototypes achieved maximum compression factors when applied to the images *vante* and *hand* respectively. Comparing *vante* and *hand* to the rest of the images in the corpus set show that these two images have large white areas and low information density. The amount of text is low and the font size is big compared to other images, e.g. *absinth* and *chateau_page2*. These observations may explain why *vante* and *hand* achieved maximum compression factors.

Prototype	Maximum	Minimum
Pricer	hand	chateau_page2
G.RLE	kellogs	samsung
PackBits	hand	chateau_page2
LZW	hand	chateau_page1
LowPac	vante	chateau_page2
Vitter	vante	refill
MTF	vante	chateau_page2
BWT	vante	chateau_page2
DMC	hand	chateau_page2
MH	hand	chateau_page2
MR	vante	chateau_page2
FEG	vante	chateau_page2

Table 5.1: Images causing maximum and minimum compression factors for the prototypes.

A count of 9 prototypes obtained minimum compression factors when applied to *chateau_page2*. The high information density of *chateau_page2* and the small font sizes are probably the reasons why this image obtained minimum compression factors. The low compressibility result of G.RLE when applied to *samsung* is expected because of the prototype's poor ability to handle black run-lengths. G.RLE is described in section 4.3.2.

It seems that image-based prototypes, e.g. LowPac and MR, have large variations and character-based prototypes, e.g. LZW and Vitter, have small variations. One explanation could be that image-based compression prototypes are more sensitive to images with high information density, causing low com-

pressibility results, while character-based prototypes are not that sensitive to different image types.

Encoding Time

Figure 5.2 shows the encoding time measurements of the prototypes. The minimum encoding time 0.45 ms was achieved by Pricer and the maximum encoding time 4.47 ms was obtained by G.RLE. All prototypes had positively skewed distributions, except Vitter that had a negative skewness. The wide positive skewness of G.RLE was caused by the images *peanuts*, *samsung* and *refill*, which are images with long black run-lengths, generating many computations and thus long encoding times. The skewness of BWT, DMC, MH, MR and FEG was caused by outliers from *peanuts*, *unlinked* and *chateau_page2*. These image are information-dense and this is probably the reason why long encoding times were needed.

A discussion about the encoding and decoding time variations is covered in the next section.

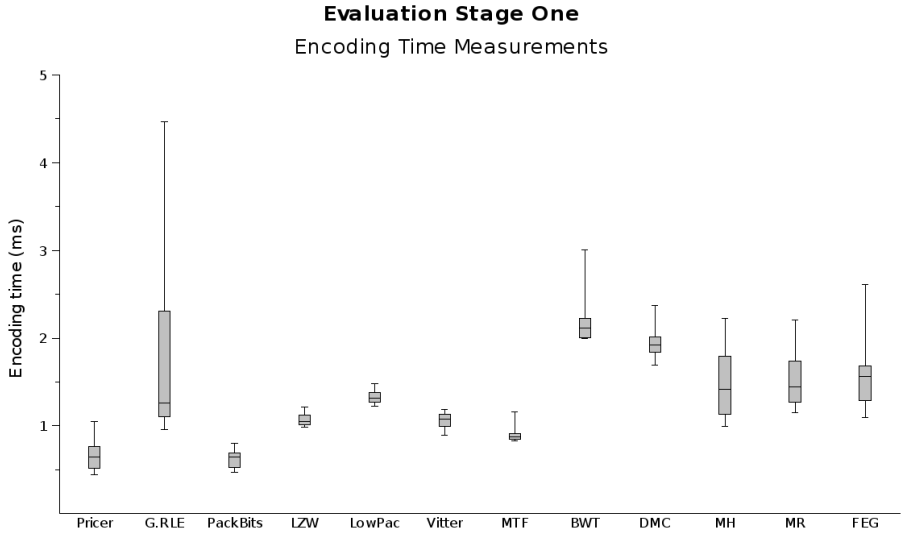


Figure 5.2: Encoding times of evaluation stage one (12 images of size 172x72 pixels).

As seen in table 5.2, a count of 5 and 3 prototypes achieved minimum encoding times when applied to *hand* and *vante* respectively. A count of 5 and 4 prototypes obtained maximum encoding times when applied to *peanuts* and

chateau_page2 respectively. The image *chateau_page2* dominated with low compressibility as seen in table 5.1. A possible explanation to the correlation between low compressibility and long encoding times is that an information dense image produces more unique statistical events than an ordinary image. A high-density image would likely cause a compression scheme to perform more computations, because of the more evenly distributed statistical events generated by such an image. A low-density image, on the other hand, would not likely generate evenly-distributed statistical events, meaning that some events would occur with a higher probability, which would acquire fewer computations in some compression schemes. An analogous discussion can be made for the correlation between high compressibility and short encoding times.

An interesting observation, violating the aforementioned discussion, is the maximum encoding times obtained by *peanuts*. The compressibility results of *peanuts* lie within the range $1 - 18\%$ degradation compared to *hand*, which was one of the most compressible images. The similarity between *peanuts* and *hand* is the large white areas and the usage of one price unit in a large font size, but *peanuts* also has a region with black background and more text segments using small font sizes. The former properties could explain the good compressibility results and the latter properties could explain the long encoding times of *peanuts*.

The measured encoding times are likely negligible compared to other work car-

Prototype	Maximum	Minimum
Pricer	peanuts	hand
G.RLE	samsung	kellogs
PackBits	samsung	hand
LZW	peanuts	samsung
LowPac	peanuts	vante
Vitter	chateau_page2	vante
MTF	peanuts	vante
BWT	hand	chateau_page1
DMC	peanuts	kellogs
MH	chateau_page2	hand
MR	chateau_page2	hand
FEG	chateau_page2	hand

Table 5.2: Images causing maximum and minimum encoding times for the prototypes.

ried out by the Pricer server. The best prototypes in terms of compressibility all had their upper quartile, yielding 75% of the encoding time samples, below 2 ms at all times. LowPac obtained an average encoding time of 1.34 ms, which is 2 times longer than Pricer's average encoding time.

Other tasks, such as I/O, are probably more significant when talking about time consuming tasks in the server software. Nevertheless, too long encoding times caused by a compression scheme may cause bottleneck problems due to the huge number of labels, typically 25,000, handled by the server. But such encoding times would probably be at least a factor 10 longer than the encoding times measured in this study according to Pricer's software developers.

Decoding Time

Figure 5.3 shows the decoding times of the prototypes. The minimum decoding time 0.06 ms was achieved by PackBits and the maximum decoding time 1.45 ms was obtained by MR. All prototypes had distributions that were positively skewed, except Vitter that had a negative skewness. G.RLE had a wide positive skewness with a high maximum value.

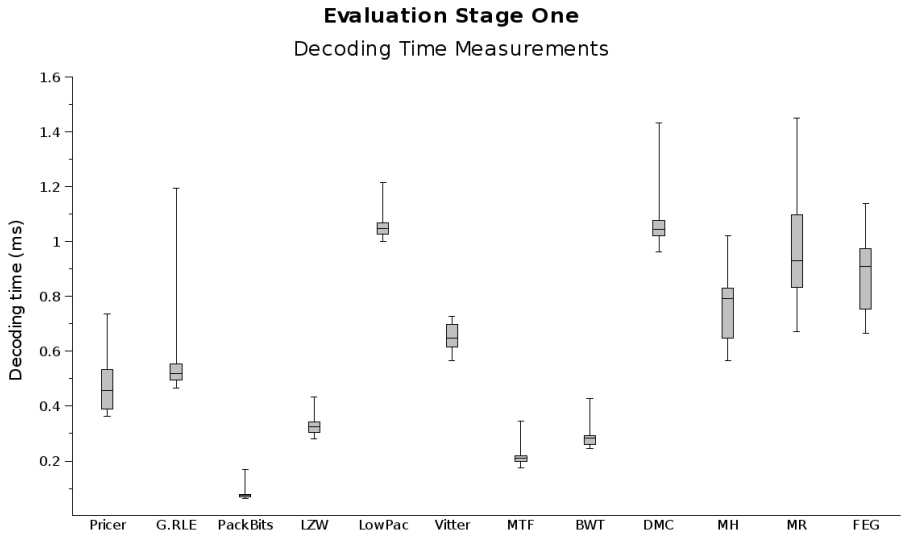


Figure 5.3: Decoding times of evaluation stage one (12 images of size 172x72 pixels).

It can be seen that some prototypes, e.g. LowPac and LZW, have small decoding time variations compared to other prototypes. One explanation to the small

variations of LowPac and LZW could be the execution of the same computations regardless of the generated statistical events of an image. Prototypes that use run-lengths, i.e. Pricer, G.RLE, MH, MR and FEG, have greater decoding time variations, probably due to different run-lengths requiring different number of computations. Comparing figure 5.2 and 5.3 suggest that such a reasoning can be applied to both encoding and decoding time variations.

Burrows and Wheeler measured encoding and decoding times of BWT when applied to the Calgary corpus [5]. Their results showed that the decoding phase of BWT was $75-87\%$ faster than the encoding phase. A similar result, $83-90\%$ faster decoding times, was obtained in this study when BWT was applied to the corpus set.

As seen in table 5.3 a count of 5 and 5 prototypes achieved minimum decoding times when applied to *vante* and *hand* respectively. A count of 8 and 3 prototypes obtained maximum decoding times when applied to *peanuts* and *chateau_page2* respectively. 6 prototypes obtained minimum and maximum decoding times with the same images with which they obtained minimum and maximum encoding times.

Prototype	Maximum	Minimum
Pricer	peanuts	hand
G.RLE	peanuts	dove
PackBits	peanuts	hand
LZW	peanuts	vante
LowPac	peanuts	vante
Vitter	chateau_page2	vante
MTF	peanuts	vante
BWT	peanuts	vante
DMC	dove	kellogs
MH	chateau_page2	hand
MR	peanuts	hand
FEG	chateau_page2	hand

Table 5.3: Images causing maximum and minimum decoding times for the prototypes.

The purpose of measuring decoding times was to see the relative effects between the Pricer prototype and other prototypes. A decoding time increase of a factor 10 compared to the Pricer prototype would probably indicate an unsuitable

compression scheme. The results of the corpus set showed that none of the prototypes took off in terms of decoding time. LowPac obtained an average decoding time of 1.06 ms, which is 2.1 times longer than Pricer's average decoding time.

However, the interpretation of decoding times imposes an issue of different computer architectures. The prototypes were evaluated using a modern laptop with an Intel Core2 Duo processor. The Intel architecture uses a complex instruction set (CISC) and a pipeline with 14 stages to achieve fast instruction cycles [13]. The microcontroller in DM3370 on the other hand uses a reduced instruction set (RISC) and a one-stage pipeline where an entire instruction needs to be finished before another one is fetched. Comparing relative decoding times would be more accurate if the architectures of the laptop and the microcontroller had been more similar.

5.2 Evaluation Stage Two

In the second evaluation stage 6 prototypes, being the top prototypes from evaluation stage one with respect to compressibility, were evaluated, using the image size of DM3370, with respect to memory consumption. Memory consumptions were measured in terms of statically allocated arrays.

The result of the memory consumption evaluation is shown in table 5.4. Figure 5.4 shows an example of how the memory consumption of LowPac, MTF and FEG would be affected by varying the image width.

Prototype	ROM	SRAM	Total
LZW	0	5,120	5,120
LowPac	0	46	46
MTF	0	256	256
BWT	0	6,960	6,960
MR	49,980	344	50,324
FEG	135	344	479

Table 5.4: Memory consumption result (in bytes) of the second evaluation stage.

The LZW prototype used Java's *HashMap* to implement a dictionary, which made it difficult to determine the prototype's memory consumption due to the unknown implementation details of the *HashMap* class. However, it was noted

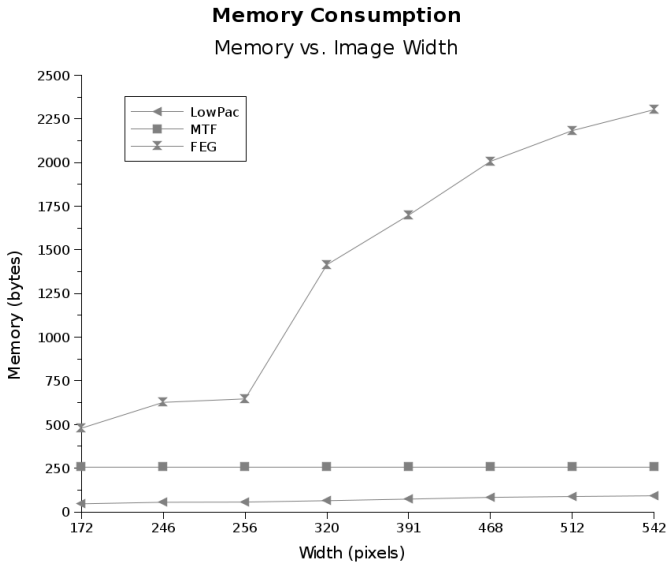


Figure 5.4: Memory consumption vs. image width.

that the LZW prototype needed at least a dictionary size of $2^{10} = 1{,}024$ elements when applied to the corpus set. An example calculation using Dipperstein's dictionary implementation [15], which required $1 + 4$ bytes per dictionary element, gave 5,120 bytes in total memory consumption of the LZW prototype.

The LowPac prototype used four static arrays. Three arrays, holding context frequencies, had sizes of 8 elements with an element size of 1 byte. The fourth static array, holding a byte-packed reference row, had a size of $\lceil \frac{image\ width}{8} \rceil$, yielding $\lceil \frac{172}{8} \rceil = 22$ elements with an element size of 1 byte. In total, the LowPac prototype used: $8 \cdot 3 \cdot 1 + \lceil \frac{172}{8} \rceil \cdot 1 = 46$ bytes.

The MTF prototype used one static array, holding the characters of the alphabet. The alphabet array had a size of 256 elements with an element size of 1 byte. In total, the MTF prototype used: $256 \cdot 1 = 256$ bytes.

The BWT prototype used five static arrays. Two arrays, holding rotated and unrotated strings in the current block, had sizes of 1,548 elements with an element size of 1 byte. Two arrays, holding character counts in the current block, had sizes of 256 and 1,548 elements respectively with an element size of 2 bytes. The fifth static array, holding the MTF alphabet, had a size of 256 elements with an element size of 1 byte. In total, the BWT prototype used: $1{,}548 \cdot 2 \cdot 1 + 1{,}548 \cdot 2 + 256 \cdot 2 + 256 \cdot 1 = 6{,}960$ bytes.

The MR prototype used seven arrays; six constant lookup arrays and one static array for reference rows. Three constant lookup arrays, used for encoding states, black and white run-lengths, had sizes of 104, 104 and 7 elements, with element sizes of 2, 2 and 4, respectively. Three constant lookup arrays, used for decoding states, black and white run-lengths, had sizes of 4,096, 8,192 and 128 elements, with element sizes of 4, 4 and 3, respectively. The seventh static array, holding two reference rows, had a size of $2 \cdot \textit{image width}$, yielding $2 \cdot 172 = 344$ elements with an element size of 1 byte. In total, the MR prototype used: $4{,}096 \cdot 4 + 8{,}192 \cdot 4 + 128 \cdot 3 + 104 \cdot 2 + 104 \cdot 2 + 7 \cdot 4 + 2 \cdot 172 = 50{,}324$ bytes.

The FEG prototype used three arrays; two arrays for constant lookup values and one static array for reference rows. The two lookup arrays, used for decoding states, had sizes of 128 and 7 elements with an element size of 1 byte. The third array, holding two reference rows, had a size of $2 \cdot \textit{image width}$, yielding $2 \cdot 172 = 344$ elements with an element size of 1 byte. In total, the FEG prototype used: $128 \cdot 1 + 7 \cdot 1 + 2 \cdot 172 = 479$ bytes.

5.3 Case Studies

The corpus set contained images created for promotional purposes. The purpose of the case studies was to see if the results on the corpus set are representative of the performance in practice.

5.3.1 Partial Images

A possible scenario using partially updated images is when a product changes price. In such a scenario, only a small region of a label's image needs to be updated. The label overlays a partial image on the current image, thus rendering new information, e.g. a new price. This case study was made to see the effect of the 6 prototypes from evaluation stage two using 3,757 partial images of size 88x36.

Figure 5.5 shows the compressibility result of some prototypes when applied to images representing product prices. The maximum compression factor 7.52 was achieved by MR and the minimum compression factor 1.89 was obtained by Pricer. A result of 127%, 133% and 122% increased average compression factor was achieved by LowPac, MR and FEG compared to Pricer respectively. Despite the small images in this case study, the prototypes' average compres-

sion factors were increased compared to the corpus set. Pricer achieved a 50% improvement, while LZW, MTF and BWT achieved approximately the same average compression factors compared to the corpus set. Thus, only image-based prototypes achieved essential improvements over the corpus set.

The variations in compressibility are probably caused by the numbers representing prices in the images. A symmetric number, e.g. 0 and 8, likely makes it easier for an image compression scheme to predict the data since the number's shape is changing slowly on a row basis. An asymmetric number, e.g. 5 and 7, on the other hand changes rapidly. This especially applies to LowPac, MR and FEG due to their ability to adapt to a previous image row.

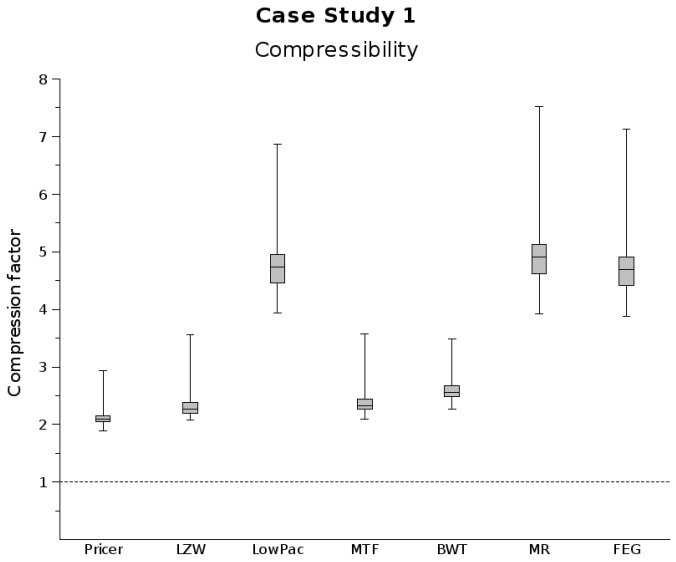


Figure 5.5: Compressibility result of case study 1 (3,757 images of size 88x36 pixels). A reference line ($y = 1.0$) shows the upper bound for negative compressibility.

As expected, the encoding times in figure 5.6 and decoding times in figure 5.7 are significantly shorter than the encoding and decoding times of the corpus set. This is mostly due to the smaller images used. The minimum encoding time 0.06 ms was obtained by Pricer and the maximum encoding time 4.32 ms was obtained by BWT. The minimum decoding time 0.04 ms was obtained by MTF and the maximum decoding time 1.14 ms was obtained by LowPac. LowPac obtained average encoding and decoding times of 0.39 and 0.36 ms,

which are 3 and 3.2 times longer than Pricer's average encoding and decoding times, respectively.

The wide spread of BWT's sample distribution in figure 5.6 is probably caused by the mergesort procedure used in BWT. A worst-case scenario, yielding long encoding time, would be if many rotated blocks ended up with the same first two characters, which would imply an execution of mergesort with many elements.

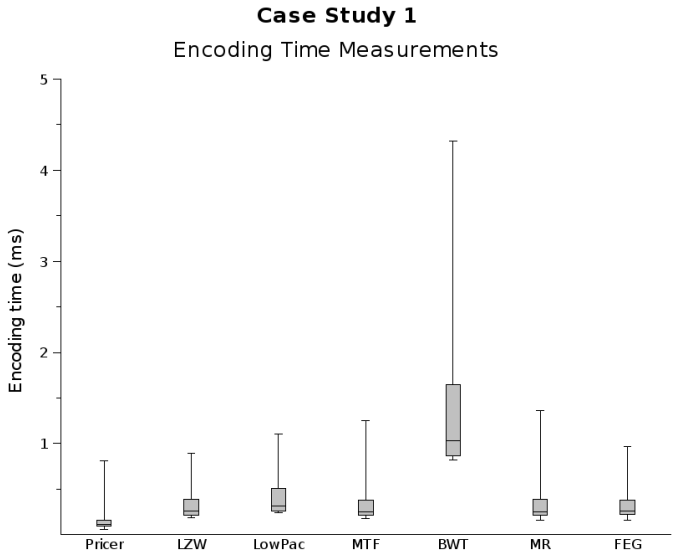


Figure 5.6: Encoding times of case study 1 (3,757 images of size 88x36 pixels).

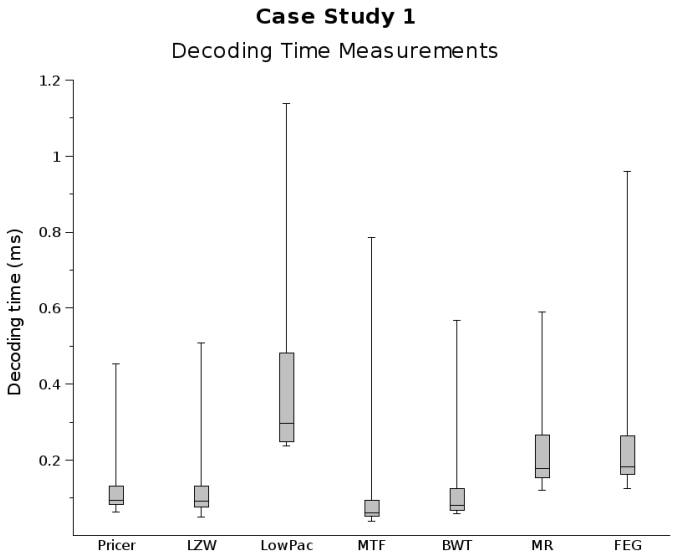


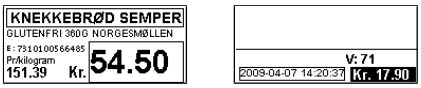
Figure 5.7: Decoding times of case study 1 (3,757 images of size 88x36 pixels).

5.3.2 DM3370 Images

This case study was based on 7,498 DM3370 images (172x72 pixels) from a Norwegian retail store, using the 6 prototypes from evaluation stage two.

Figure 5.9 show the compressibility result of the DM3370 images. A result of $34 - 186\%$ increased compressibility was achieved by LowPac compared to Pricer when applied to the DM3370 images, yielding 103% increased average compression factor.

Comparing figure 5.1 and 5.9 show that the corpus set is representative to real DM3370 data in terms of compressibility, but the latter result is somewhat better. The spread of the sample distributions is wider in case study 2, but that can be explained by the layouts of the images. The DM3370 set of images consisted of two image layouts, which can be seen in figure 5.8. Images using layout *(a)*, which has a high information density, had low compressibility, while images using layout *(b)*, which has a low information density, had high compressibility.



(a) Customer view (b) Personnel view

Figure 5.8: Two image layouts of the DM3370 images (172x72 pixels) from case study 2. The black outer borders are not part of the images.

As in the case of compressibility, figure 5.2 and 5.10 show that the corpus set is representative of real DM3370 in terms of encoding times. The same reasoning applies to the decoding times of real DM3370 data, which can be seen by comparing figure 5.3 and 5.11. The variations of encoding and decoding times of case study 2 are probably due to the image layouts of the DM3370 image set. For DM3370 data LowPac obtained average encoding and decoding times of 1.13 and 1.12 ms, which are 2 and 2.5 times longer than Pricer's average encoding and decoding times, respectively.

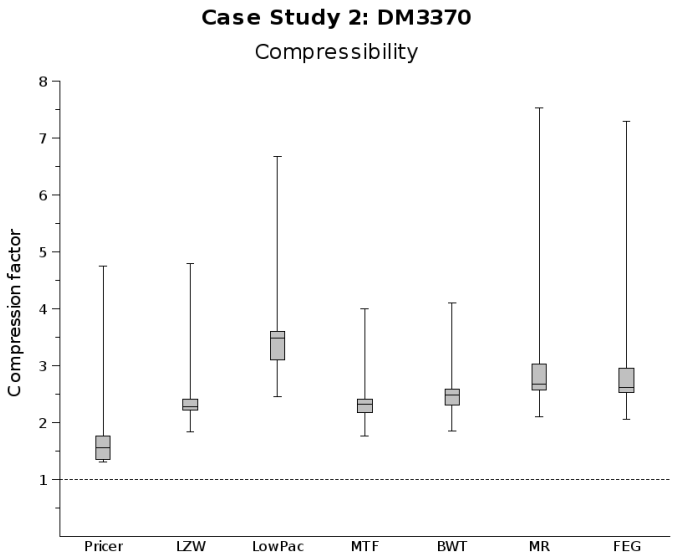


Figure 5.9: Compressibility result of case study 2 (7,498 images of size 172x72 pixels). A reference line ($y = 1.0$) shows the upper bound for negative compressibility.

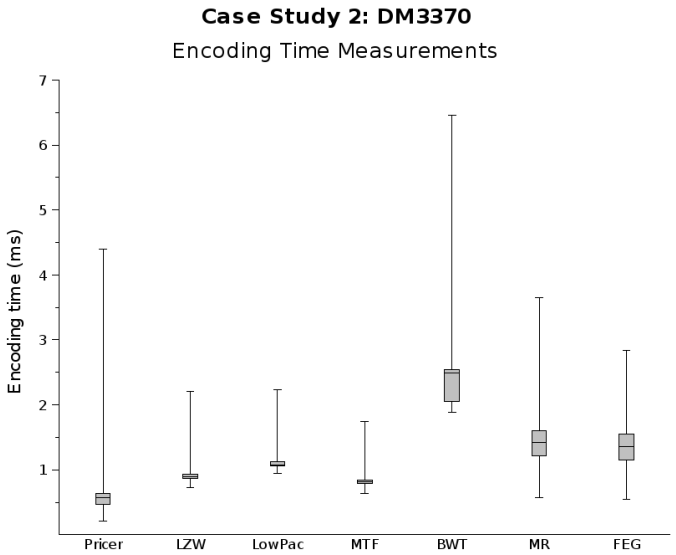


Figure 5.10: Encoding times of case study 2 (7,498 images of size 172x72 pixels).

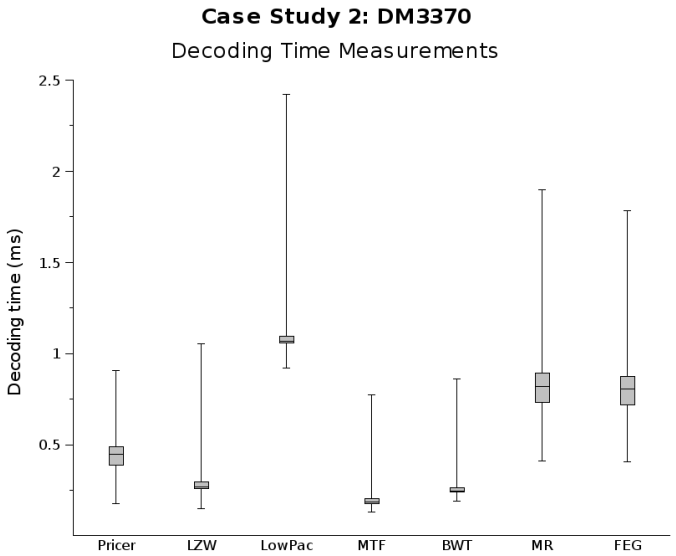


Figure 5.11: Decoding times of case study 2 (7,498 images of size 172x72 pixels).

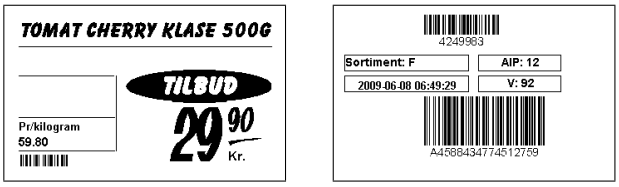
5.3.3 DM110 Images

This case study was based on 83 DM110 images (320x192 pixels) from a Norwegian retail store, using the 6 prototypes from evaluation stage two.

Figure 5.13 show the compressibility result of the DM110 images. An improvement of $109 - 365\%$ was achieved by LowPac compared to Pricer when applied to the DM110 images, yielding 194% increased average compression factor.

Figure 5.9 and 5.13 show that a higher compressibility was achieved when DM110 images were used. The maximum compression factor 7.52 was achieved by MR, while the minimum compression factor 1.31 was obtained by Pricer for the DM3370 images. The maximum compression factor 20.89 was achieved by MR, while the minimum compression factor 2.01 was obtained by Pricer for the DM110 images. A DM110 image is 5 times bigger than a DM3370 image, enabling the possibility to use larger font sizes. This is often the case of the DM110 images; they use large font sizes and have a lot of white areas, which likely explain the higher compressibility results.

The variations of compressibility of the DM110 images can also be explained, as in the case of the DM3370 images, by the image layouts. The DM110 set of images consisted of mainly two image layouts, which can be seen in figure 5.12. Images using layout *(a)*, which uses large bold fonts, achieved high compressibility, while images using layout *(b)*, which uses barcodes and small font sizes, obtained low compressibility.



(a) Customer view (b) Personnel view

Figure 5.12: Two image layouts of the DM110 images (320x192 pixels) from case study 2. The black outer borders are not part of the images.

In general, the average encoding time per prototype was $3 - 5$ times longer for the DM110 image set than for the DM3370 image set. The average decoding time per prototype was $2 - 5$ times longer for the DM110 image set than for the DM3370 image set. For DM110 data LowPac's encoding and decoding times

were 3.2 and 3.7 longer than Pricer's encoding and decoding times, respectively.

Longer encoding and decoding times for DM110 data are expected, because of the larger size of such an image. The long encoding times of BWT when applied to the DM110 data set are likely due to the increased block size. As mentioned in section 4.3.2, DM3370 and DM110 images had block sizes of 1,545 and 7,680 bytes respectively. A larger block size implies longer execution times of BWT's mergesort operation, which probably is the cause of the longer encoding times.

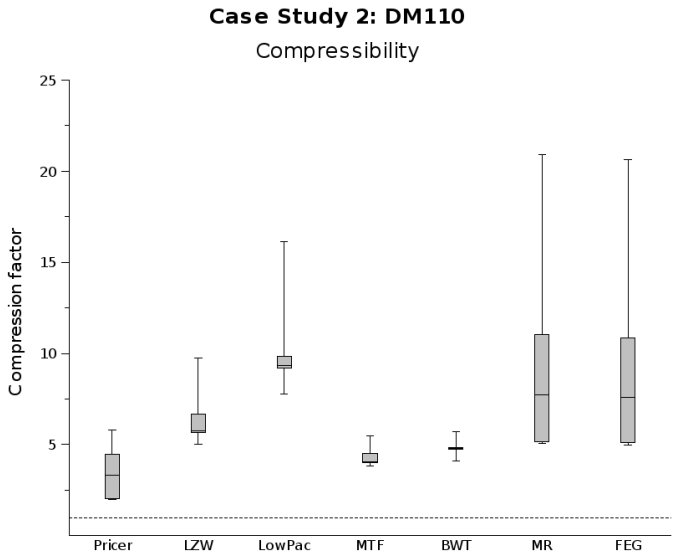


Figure 5.13: Compressibility result of case study 2 (83 images of size 320x192 pixels). A reference line ($y = 1.0$) shows the upper bound for negative compressibility.

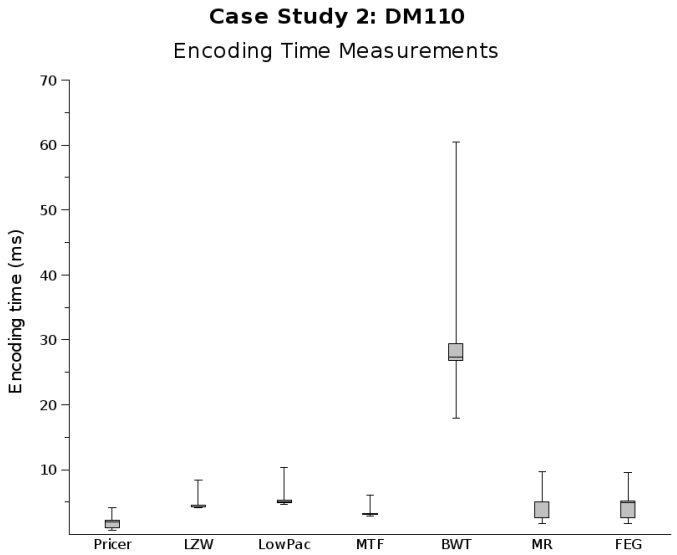


Figure 5.14: Encoding times of case study 2 (83 images of size 320x192 pixels).

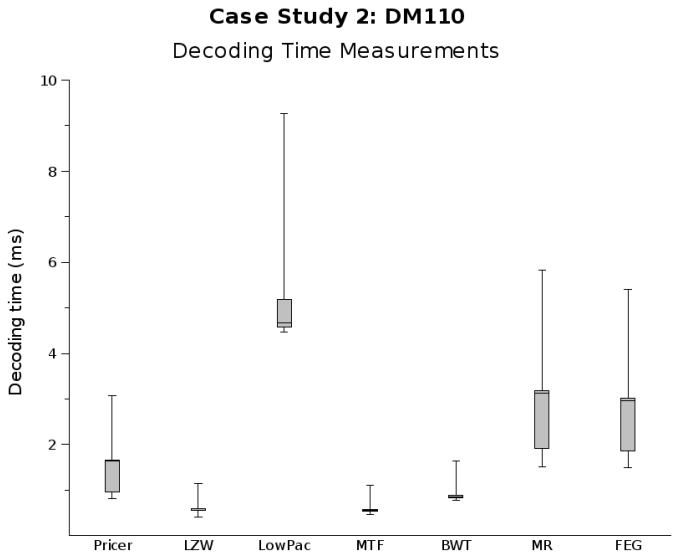


Figure 5.15: Decoding times of case study 2 (83 images of size 320x192 pixels).

Chapter 6

Conclusions

6.1 Summary of Results

Bi-level images used in Pricer's ESL platform can be compressed a lot more using another compression scheme compared to the existing scheme. The low-precision arithmetic coder LowPac achieved good compressibility results over the corpus set, resulting in 108% improvement on average compared to Pricer's compression scheme. The improvement on real ESL data was 103%, 194% and 127%, on average, when applied to DM3370, DM110 and partial images respectively. LowPac was also the most memory-efficient prototype.

The encoding time measurements showed that most of the prototypes are suitable compression schemes for the Pricer server. The decoding time measurements are difficult to interpret due to the different computer architectures used on the evaluation computer and the microcontrollers. However, the decoding times did not differ much between the different prototypes, but LowPac obtained encoding and decoding times of 2 times longer than Pricer using the corpus set.

The best prototypes in terms of compressibility achieved best results in all data sets, i.e. they had the same relative order when plotting the sample distributions of compression factor. Images with low information density had high compressibility results and short encoding and decoding times. The opposite reasoning goes for images with high density, i.e. having low compressibility results and long encoding and decoding times.

6.2 Recommendations

Due to the outstanding compressibility results over all data sets and low memory consumptions of LowPac and FEG, it is recommended to do further studies of those prototypes. Further studies involve implementing the prototypes' decoding functions on real hardware and measuring decoding times. Without real decoding time measurements it is difficult to say whether or not a single label would benefit from one of the two compression schemes. However, the Pricer platform would probably still benefit from such compression schemes in terms of increased responsiveness.

6.3 Future Work

LowPac and FEG can be improved using simple optimization steps, such as reducing function-call overheads of important functions with inlining or macro definitions and speeding up array handling with pointer arithmetic. Witten, Neal and Cleary managed to reduce the execution time of their CACM implementation with a factor two by introducing macros, using pointer arithmetic and replacing multiplications with additions [52]. Another reduction of a factor two was achieved by reimplementing parts of the algorithm in an assembly language.

It may be interesting to experiment with LowPac's context parameters to create a balance between compressibility and decoding time. Using fewer bits to create a neighborhood context will result in a reduced number of arithmetic operations, implying shorter decoding time and slightly lower compressibility performance. But the question is whether or not a context bit reduction is significant in terms of decoding time? Applying profiling techniques to LowPac's encoder and decoder will reveal critical sections in terms of encoding and decoding times. Profiling the LowPac decoder is the most important task of the two.

Both LowPac and FEG exploit the spatial redundancy between two rows using a reference row. Future work may include the process of finding a better way to represent the reference row in FEG. If a new efficient representation could be found the memory consumption of FEG would not increase so dramatically when increasing the image width as shown in figure 5.4. Of course, a lower memory consumption with small images, e.g. DM3370 images, would also be beneficial. One way to lower the memory consumption, without to modify the

prototypes, would be store a reference column instead of a row, which can be achieved by raster scanning an image column-wise instead of row-wise.

A future improvement of ESL systems may be the use of indexed images, e.g. grayscale. But prototypes optimized for bi-level images do not work out of the box for such images and it is probably difficult to modify such a prototype and at the same time maintain a low memory consumption. However, it would be interesting to see how character-based prototypes would perform on indexed images with use of alphabet extension. Especially MTF is an interesting prototype, because of its low memory consumption.

References

[1] Alberto Apostolico and Aviezri S. Fraenkel. Robust Transmission of Unbounded Strings Using Fibonacci Representations. *IEEE Transactions on Information Theory, Vol. 33, No. 2*, pages 238–245, 1987.

[2] Adobe Developers Association. TIFF Specification (Revision 6.0), 1992.

[3] Jon L. Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A Locally Adaptive Data Compression Scheme. *Communications of the ACM, Vol. 29 No. 4*, pages 320–330, 1986.

[4] Eric Bodden, Malte Clasen, and Joachim Kneis. Arithmetic Coding Revealed. *Sable Technical Report, No. 5*, 2007.

[5] Michael Burrows and David J. Wheeler. A Block-sorting Lossless Data Compression Algorithm. *Digital SRC Research Report*, 1994.

[6] World Wide Web Consortium. Graphics Interchange Format Specification (Version 89a), 1990. `http://www.w3.org/Graphics/GIF/spec-gif89a.txt`, 2009-11-26.

[7] ImageMagick Contributors. ImageMagick trunk libtiff, 2009. `http://trac.imagemagick.org/browser/tiff/trunk/libtiff/`, 2009-11-26.

[8] Rosetta Code Contributors. LZW compression. `http://rosettacode.org/wiki/LZW_compression#Java`, 2009-11-26.

[9] Wikipedia Contributors. Markov chain. `http://en.wikipedia.org/wiki/Markov_chain`, 2009-11-26.

[10] Gordon V. Cormack. Dynamic Markov Compression (DMC). `ftp://ftp.sac.sk/pub/sac/pack/dmcsrc.zip`, 2009-11-26.

[11] Gordon V. Cormack and R. Nigel Horspool. Data Compression Using Dynamic Markov Modelling. *The Computer Journal, Vol. 30, No. 6*, pages 541–550, 1987.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2001.

[13] Intel Corporation. Product brief: Intel core2 duo processor. `http://download.intel.com/products/processor/core2duo/mobile_prod_brief.pdf`, 2009-11-26.

[14] Michael Dipperstein. Burrows-Wheeler Transform Discussion and Implementation. `http://michael.dipperstein.com/bwt/index.html`, 2009-11-26.

[15] Michael Dipperstein. Lempel-Ziv-Welch (LZW) Encoding Discussion and Implementation. `http://michael.dipperstein.com/lzw/index.html`, 2009-11-26.

[16] Michael Dipperstein. Rice (Golomb) Coding Encoding Discussion and Implementation. `http://michael.dipperstein.com/rice/index.html`, 2009-11-26.

[17] Michael Dipperstein. Run Length Encoding (RLE) Discussion and Implementation. `http://michael.dipperstein.com/rle/index.html`, 2009-11-26.

[18] Peter Elias. Universal Codeword Sets and Representations of the Integers. *IEEE Transactions on Information Theory, Vol. 21, No. 2*, pages 194–203, 1975.

[19] Peter Elias. Interval and Recency Rank Source Coding: Two On-Line Adaptive Variable-Length. *IEEE Transactions on Information Theory, Vol. 33, No. 1*, pages 3–10, 1987.

[20] Newton Faller. An Adaptive System for Data Compression. *Asilomar Conference on Circuits, Systems and Computers, 7th*, pages 593–597, 1973.

[21] Peter Fenwick. Variable-Length Integer Codes Based on the Goldbach Conjecture, and Other Additive Codes. *IEEE Transactions on Information Theory, Vol. 48, No. 8*, pages 2412–2417, 2002.

[22] Robert G. Gallager. Variations on a Theme by Huffman. *IEEE Transactions on Information Theory, Vol. 24, No. 6*, pages 668–674, 1978.

[23] Solomon W. Golomb. Run-length Encodings. *IEEE Transactions on Information Theory, Vol. 12, No. 3*, pages 399–401, 1966.

[24] Charles M. Grinstead and J. Laurie Snell. *Introduction to Probability*. American Mathematical Society, 1997.

[25] PNG Development Group. PNG Specification (Version 1.2), 1999.

[26] David A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers, Vol. 40, No. 9*, pages 1098–1101, 1952.

[27] Roy Hunter and A. Harry Robinson. International Digital Facsimile Coding Standards. *Proceedings of the IEEE, Vol. 68, No. 7*, pages 854–867, 1980.

[28] Apple Inc. Apple Technical Note TN1023. `http://17.254.2.129/technotes/tn/tn1023.html`, 2009-11-26.

[29] Gareth A. Jones and J. Mary Jones. *Information and Coding Theory*. Springer, 2000.

[30] Donald E. Knuth. Dynamic Huffman Coding. *Journal of Algorithms, Vol. 6, No. 2*, pages 163–180, 1985.

[31] Leon G. Kraft. A Device for Quantizing, Grouping and Coding Amplitude Modulated Pulses. *MS Thesis, Electrical Engineering Department, Massachusetts Institute of Technology*, 1949.

[32] Glen G. Langdon and Jorma Rissanen. Compression of Black-White Images with Arithmetic Coding. *IEEE Transactions on Communications, Vol. 29, No. 6*, pages 858–867, 1981.

[33] Henrique S. Malvar. Fast Adaptive Encoder for Bi-level Images. *Proceedings of the Data Compression Conference*, pages 253–262, 2001.

[34] T. L. McCullough. CCITT Standardization for Digital Facsimile. *AFIPS, National Computer Conference*, pages 409–413, 1980.

[35] Brockway McMillan. Two Inequalities Implied by Unique Decipherability. *IEEE Transactions on Information Theory, Vol. 2, No. 4*, pages 115–116, 1956.

[36] Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic Coding Revisited. *ACM Transactions on Information Systems, Vol. 16, No. 3*, pages 256–294, 1998.

[37] Radford M. Neal. Low-precision Arithmetic Coding Implementation, 1991. `ftp://ftp.cpsc.ucalgary.ca/pub/projects/arithmetic.coding/low.precision.version/`, 2009-11-26.

[38] Department of Computer Science Oliver Kasten, ETH Zürich. Atmel AVR microcontrollers. `http://www.inf.ethz.ch/personal/kasten/research/bathtub/avr/`, 2009-11-26.

[39] Richard Clark Pasco. Source Coding Algorithms for Fast Data Compression. *Ph.D. thesis. Department of Electrical Engineering, Standford University*, 1976.

[40] PKWARE. .ZIP File Format Specification (6.3.2), 2007. `http://www.pkware.com/documents/casestudies/APPNOTE.TXT`, 2009-11-26.

[41] Robert F. Rice. Some Practical Universal Noiseless Coding Techniques. *Proceedings of the Society of Photo-Optical Instrumentation Engineers, Vol. 207*, pages 247–267, 1979.

[42] Jorma J. Rissanen. Generalized Kraft Inequality and Arithmetic Coding. *IBM Journal of Research and Development, Vol. 20, No. 3*, pages 198–203, 1976.

[43] Frank Rubin. Arithmetic Stream Coding Using Fixed Precision Registers. *IEEE Transactions on Information Theory, Vol. 25, No. 6*, pages 672–675, 1979.

[44] David Salomon. *Data Compression: The Complete Reference*. Springer, 2007.

[45] David Salomon. *Variable-length Codes for Data Compression*. Springer, 2007.

[46] David Salomon. *A Concise Introduction to Data Compression*. Springer, 2008.

[47] Khalid Sayood. *Lossless Compression Handbook*. Academic Press, 2003.

[48] Claude E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal, Vol. 27*, pages 379–423 and 623–656, 1948.

[49] Jeffrey S. Vitter. Design and Analysis of Dynamic Huffman Codes. *Journal of the Association for Computing Machinery, Vol. 34, No. 4*, pages 825–845, 1987.

[50] Jeffrey S. Vitter. Algorithm 673: Dynamic Huffman Coding. *ACM Transactions on Mathematical Software, Vol. 15, No. 2*, pages 158–167, 1989.

[51] Terry A. Welch. A Technique for High-Performance Data Compression. *IEEE Computer, Vol. 17, No. 6*, pages 8–19, 1984.

[52] Ian H. Witten, Radford M. Neal, and John G. Cleary. Arithmetic Coding for Data Compression. *Communications of the ACM, Vol. 30, No. 6*, pages 520–540, 1987.

[53] Yasuhiko Yasuda. Overview of Digital Facsimile Coding Techniques in Japan. *Proceedings of the IEEE, Vol. 68, No. 7*, pages 830–845, 1980.

[54] Jacob Ziv and Abraham Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory, Vol. 23, No. 3*, pages 337–343, 1977.

[55] Jacob Ziv and Abraham Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory, Vol. 24, No. 5*, pages 530–536, 1978.

Appendix A

Example: Construction of a Huffman code

Figure A.1 shows the construction of a Huffman code. Step (a) shows the original parent nodes and their frequencies recovered from the first pass of some source message. Step (b) to (g) shows the greedy approach of the Huffman algorithm. Each iteration ends with the list of parent nodes being sorted in descending order based on the frequencies. The final step (g) shows the final Huffman tree of this particular code.

The code words are constructed by traversing the tree in an in-order fashion. An in-order walk, where a left branch outputs a zero and a right branch outputs a one, yields the code words: 0, 11, 101, 1000, 10010 and 10011. The average code word length is: $55/100 \cdot 1 + 17/100 \cdot 2 + 13/100 \cdot 3 + 9/100 \cdot 4 + 5/100 \cdot 5 + 1/100 \cdot 5 = 1.94$. The theoretical minimum average code word length is achieved by the entropy: $-(log_2(55/100) \cdot (55/100) + log_2(17/100) \cdot (17/100) + log_2(13/100) \cdot (13/100) + log_2(9/100) \cdot (9/100) + log_2(5/100) \cdot (5/100) + log_2(1/100) \cdot (1/100)) = 1.89$

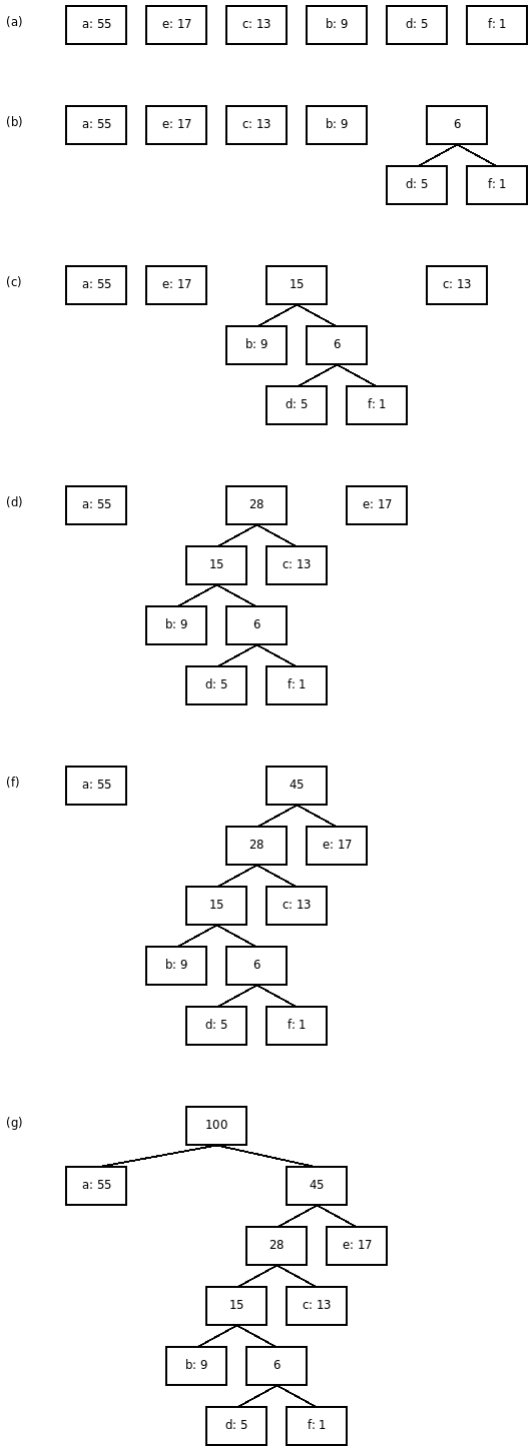


Figure A.1: The construction of a Huffman code where the frequencies of the initial symbols are shown in step (a). Step (b) to (g) show the greedy approach of the Huffman algorithm.