Hiding in Plain Sight

Using graph theory to hide data in JPEG images

Project Report

DAT2-A423

Aalborg University
Datalogi

This report has been typeset using LaTeX.

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
http://www.cs.aau.dk/

Title:
Hiding in Plain Sight - Using graph theory to hide data in JPEG images

Theme:
Programming and Problem Solving

Project Period:
Spring Semester 2016

Project Group:
DAT2-A423

Participants:
Henrik Herbst Sørensen
Jacob Askløf Svenningsen
Jakob Meldgaard Kjær
Leo Johannesen Mohr
Mathias Steen Jakobsen
Søren Madsen
Theresa Krogh-Walker

Supervisor:
Kurt Nørmark

Pages:
112

Date of Completion:
May 23, 2016

Abstract:

This report contains an analysis of the problems with steganography and how hidden messages can be detected through steganalysis.
It also gives insights into the uses of steganography, such as who benefits from it and why there is a need for concealed messages and thereby put it into context.
The report contains some steganographic experiments that have been conducted and also consists of an analysis of the image format known as JPEG. These investigations led to the creation of the final programme, which is developed to work on this very format during the writing of this report.
The programme is able to embed a message secretly within an image to make it as difficult as possible to detect by the human eye, using the branch of Mathematics known as graph theory.

Contents

Signatures

Aalborg University, May 23, 2016

Henrik Herbst Sørensen
hsaren14@student.aau.dk

Jacob Askløf Svenningsen
jsvenn15@student.aau.dk

Jakob Meldgaard Kjær
jkjar14@student.aau.dk

Leo Johannesen Mohr
lmohr15@student.aau.dk

Mathias Steen Jakobsen
msja15@student.aau.dk

Søren Madsen
smads15@student.aau.dk

Theresa Krogh-Walker
tkrogh15@student.aau.dk

Chapter 1

Introduction

1.1 Cluster Group

This section will focus on the cluster group, which was supposed to be a large part of the process. It will describe the various concerns that our group had before and during the project, as well as contain an analysis of the process that we went through.

Motive

Before the groups for this project were formed, everyone was encouraged to read a project catalogue containing all the project proposals available for the project. The steganography entry in the catalogue contained a paragraph that read (translated from Danish):

> *This project proposal is provided to first-year students from three different subjects: Mathematics, Computer Science/Software Engineering, and Internet Technologies and Computer Systems. The groups who choose the steganography entry are required to make their own project based on their respective subjects, but the end goal is to be able to coordinate the projects, so that each one of them are a part of an overall project. By doing it this way, this project will resemble the challenges that you will face after graduation.*

Working across studies posed an interesting challenge because it would prepare us for working with people with different abilities and knowledge.

The three groups in our cluster group were Mathematics (MAT) and Internet Technologies and Computer Systems (ITC), and us, Computer Science (DAT). At our first meeting we decided to make three independent projects, each able to stand on their own. This had the potential to give us a wider understanding of the topic of steganography as we would see it from different sides instead of only from a programmer's perspective.

1.2 Introduction to Steganography

The focus of this section is to make the reader familiar with what steganography is. It includes a few examples of uses throughout history, as well as a guess on who might be interested in using steganography.

Through the ages, people have been dependent on efficient forms of communication. In some cases, these messages would include information that should ideally be confidential and not read by anyone else other than the intended recipient. Doing this would require a way of concealing the details of the message so that an outsider would not be able to decipher what the actual message was.

Whereas cryptography is about concealing a message's content in a way that does not attempt to hide the existence of the message, steganography is about concealing the message's existence. This means that instead of the private message's content being obviously protected by a security-measure, people have had to find ways of circumventing interception and hide their messages in plain view to avoid any suspicion. This is one advantage steganography has over cryptography: steganography can go unnoticed, while cryptography typically cannot.

Steganography is the art of concealing a message in a cover object, without having other people being aware that the concealed message is being sent (Anderson, 1998). The cover object could be in any form such as images, audio or plain text. An example of this could be hiding information in something innocuous that is not likely to get any unwanted, extra attention. The concealment should be subtle, so that unless a person was aware there was something to be found, it is very unlikely to notice that a message is being passed in front of them without them noticing.

As not everyone wishes to have every detail of their lives scrutinised, people have had to implement forms of steganography to get their meaning across to their intended recipient. This goes back centuries, for example, people in ancient China would write messages on fine silk, which would then be pressed into a small sphere and covered in wax. The cover of this message was a person who would have to swallow the message to bring the information safely, without interception, to the intended receiver (Singh, 2001). Another example being medieval Europe, where they had a system using templates, which would then be placed over a text, highlighting what the actual message is (Anderson, 1998).

Since then, forms of steganography have become much more advanced, and naturally, digitalised. Steganography can be implemented by anyone, no matter their intention, but would most typically be people who feel like they have something to hide. With the rise of social media, sending messages via the internet - and specifically these social media networks - has become the norm. Therefore it is only natural that a modern form of steganography should be able to work on these platforms.

1.3 Establishing the Project

This project will focus on the topic of steganography and its uses in a digital context. Steganography was chosen as a topic, due to the fact that it is an interesting subject to study, as there are so many options as to how it can be carried out, but also has some ethical questions surrounding the entire process of sending hidden messages. The intentions behind sending these messages could potentially be malicious but could also be benign.

With the rise of social media, and its use in our everyday lives, a social network would be an ideal place to send messages, and therefore also steganographic messages.

Discussing the various problems surrounding steganography and what it is ultimately used for, has resulted in the following initiating problem:

> What obstacles arise when attempting to send hidden data across social media and which algorithms can be used to circumvent these?

Chapter 2

Problem Analysis

2.1 Ethical Questions and Steganography in Context

As previously described, steganography is a practice with several different uses. In this section we will study different uses of digital steganography.

The need to hide a message has historically been shown to be beneficial in times of war and revolution. Steganography has been seen used by people in war, by terrorists, activists, dissidents and suppressed citizens as a means of secretly sending a message ideally without the existence of the message being known to anyone but the sender and intended receiver. For example these messages could contain war targets, locations of allies, military orders or simply express political opinions without endangering the sender of the message (Singh, 2001).

With all these different uses of steganography, good and bad, the question "Is it really ethical to develop tools that makes this kind of secret communication possible?" arises. One could assume that a big percentage of the uses of steganography would involve criminal activity or terrorism. On the other hand, steganography has plenty of beneficial uses, some of which will be covered in the next section.

2.1.1 Uses of Steganography

To further study steganography in a real-life context and the relevance to the aforementioned question, it is important to examine the uses of this practice, and to do this a few examples of digital steganography will be described.

Film companies today make use of steganography to hide a message that is essentially a watermark. In every copy sent out to cinemas and other services that show the film, a few pixels, will be slightly altered to give every release-service its own unique ID. This means that the publisher will be able to determine what service their film was leaked from if that was to happen. The difference in the picture will be so subtle that it is hardly visible to the human eye.

Music companies make use of the same principle but instead alter specific audio frequencies slightly. This slight differences in the track will be inaudible to the human ear (Anderson, 1998).

A hypothetical example could be that a technology company is working on a new advanced product that they do not want anyone to know about before receiving a patent on all used production technologies or before a full product release. If they share a file containing related information protected by some sort of encryption, it is obvious that they are trying to send an important message that they wish to hide from press and competition. If they are able to send a message, concealed in a cover that seems innocent and perhaps unrelated to product development, the company will not draw attention to themselves in the same way.

These examples demonstrate that the uses of steganography may not be as black and white as they initially seem. It really does hold a lot of possibilities and the majority of these can be easily justified.

2.2 Least Significant Bit Method - An Experiment

During the beginning of the project, to learn more about and understand how steganography works, we investigated and implemented a relatively simple method of steganography with the ability to conceal an image within another image. The focus of this section is on explaining in detail how this method works, and how we implemented it.

2.2.1 Theory

A common way of digitally hiding information within other information is using the least significant bit (LSB) of the cover, which is to contain the information. The LSB is the bit of a binary integer, which gives it its unit value, since it is also the bit with the lowest bit weight in base-2: 2^0.

Using this bit is a very straightforward way of making it difficult for interceptors to see the information. This is because changing only this bit to a desired value can only change the integer stored in the bits by one, which is not a very noticeable difference if it is a large integer. If the integer, for example, represents an ASCII character, however, the change would be obvious.

In a true-colour bitmap image for example, each pixel contains three bytes. Each of these carry information about the amount of red, green and blue colours in the pixel in question. Since it is stored in a byte, there are 256 possible values for each colour. Adjusting this value by one has very little effect on the image as a whole. The difference is hardly noticeable to the human eye.

Because digitally stored data is saved as binary numbers it is an obvious choice to use the LSB for storing a secret message. This, however, also makes it easier for any interceptors to find the hidden information. This is a problem, because even though the human eye can not perceive the faint differences in nuance created by changing the LSB in an image, computers can. Comparing the frequency of certain colours in an image containing hidden information with an average value for an image of the same style, will bring out any discrepancies in the stego image. This will be described in greater detail in section 2.8.

It might still be difficult to actually extract the information, but it is certainly not impossible. A steganographic method, and therefore any communication channel that uses it, is to be considered compromised if there exists a way to predict if something contains a hidden message more reliably than random guessing (Böhme and Westfeld, 2004).

2.2.2 Implementation

The code for this programme can be seen in Appendix A, and the algorithm we have implemented can be seen in Procedure 1.

By storing the information we want concealed in the two least significant bits in the cover image's bytes, we can keep all of the message's original information, and changing each pixel of the cover image, by a maximum value of $11_2 = 3_{10}$, the concealed image will be almost invisible to the naked eye.

To do this, we had to make some restrictions on how big the cover image could be relative to the message image. To not cause corruption and to ensure all information gets saved, the cover image has to be four times as big as the hidden image, more specifically twice its height, and twice its width. Each pixel of the concealed image will be stored within four pixels of the cover image, each masked differently, hence the size requirements. We do this to get as much of the concealed image's information stored within the cover as possible.

To make both images easier to work with, we store each pixel's colour in their own array of type `Color`. To store each pixel within the cover, we first mask the cover's colour-bytes to remove the information stored within the least significant bits, using the bitwise operator &. We then split the concealed image's colour-bytes into 2-bit groups, then right-shift those bits so that they occupy the lowest order bits. Lastly we add this information to the cover's colour-bytes. The result will be a pixel with its value for the Red, Green and Blue channels slightly changed by a value between 0 and 3. The first pixel in the cover will contain the two most significant bits of the concealed image, the next will contain the third and fourth, and so on.

To recover the message image, we once again use the bitwise & operator to only keep the information within the two least-significant bits of every pixel, then left-shift them accordingly and save this to a `Color` array. Once every pixel has been read, the `Color` array can be transformed into an image which looks just like the original.

The entire process of encoding one pixel into four others using the two least significant bits is shown in figure 2.1. We have the cover image C, the message image M and the steganographic image S. Each pixel in the message image is split up into four parts M_1, M_2, M_3 and M_4 where each part contains two bits from the R, G and B channels of the original pixel.

For every pixel chosen in M, four pixels are chosen from C. For each of those four pixels, we store the six most significant bits from the R, G and B channels in C_1, C_2, C_3 and C_4.

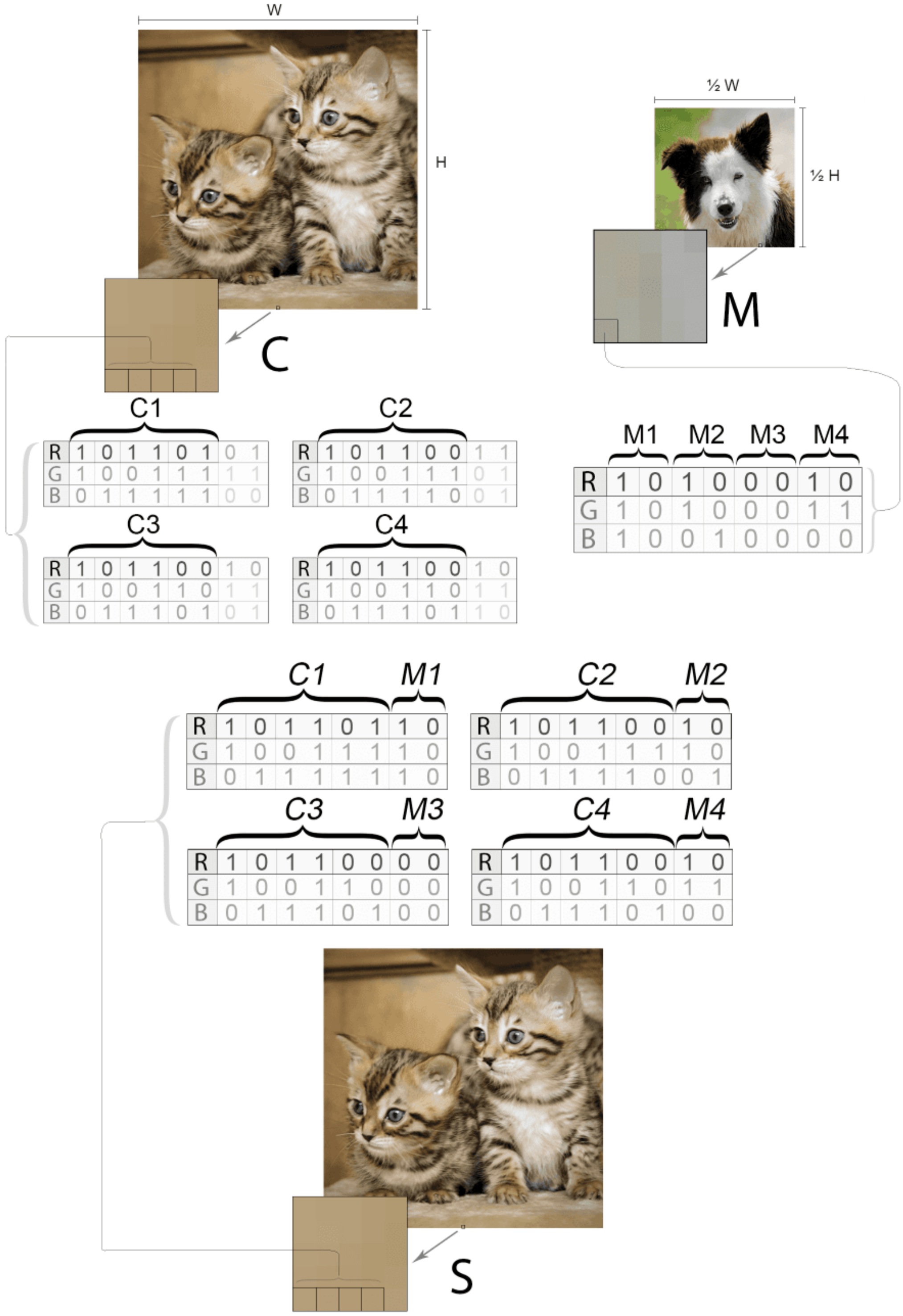
Lastly, four pixels in S are created by making pairs in the form of $C_n M_n$. This means that the four pixels in S will be the following combinations: C_1M_1, C_2M_2, C_3M_3 and C_4M_4.

Procedure 1 Using the Two Least Significant Bits for Storing Data in an Image

```
Input: True-color bitmap images $C = \{c_1, c_2 \dots c_{WH}\}$ with size $W \times H$ where $c_{xW+y}$
  is the pixel at location $(x, y)$ and $M = \{m_1, m_2 \dots m_{\frac{WH}{4}}\}$ with size $\frac{W}{2} \times \frac{H}{2}$ where
  $m_{xW/2+y}$ is the pixel at location $(x, y)$
Output: True-color bitmap image $S = \{s_1, s_2 \dots s_{WH}\}$ of size $W \times H$ and where
  $s_{xW+y}$ is the pixel at location $(x, y)$ and where $m$ is embedded in $c$
  $k := 1$
  for $i := 1$ to $i = \frac{WH}{4}$ do
    for $j := 1$ to $j = 4$ do
      Set the first 6 bits in the R, G and B bytes of $s_k$ to be the first 6 bits of the R,
      G and B bytes of the $c_k$
      Set the 7th bit in the R, G and B bytes in $s_k$ to be the $(j \cdot 2 - 1)$th bit in the R,
      G and B bytes of $m_i$
      Set the 8th bit in the R, G and B bytes in $s_k$ to be the $(j \cdot 2)$th bit in the R, G
      and B bytes of $m_i$
      $k := k + 1$
    end for
  end for
  return $S$
```

Conclusion

We now have an understanding of how basic steganography with the LSB-method works. This knowledge will help us understand other methods used in steganography, as well as help us develop a programme using a method that is not as easily compromised as this implementation is.



C1

R	1	0	1	1	0	1	0	1
G	1	0	0	1	1	1	1	1
B	0	1	1	1	1	1	0	0

C2

R	1	0	1	1	0	0	1	1
G	1	0	0	1	1	1	0	1
B	0	1	1	1	1	0	0	1

C3

R	1	0	1	1	0	0	1	0
G	1	0	0	1	1	0	1	1
B	0	1	1	1	0	1	0	1

C4

R	1	0	1	1	0	0	1	0
G	1	0	0	1	1	0	1	1
B	0	1	1	1	0	1	1	0

	M1		M2		M3		M4	
R	1	0	1	0	0	0	1	0
G	1	0	1	0	0	0	1	1
B	1	0	0	1	0	0	0	0

	C1						M1	
R	1	0	1	1	0	1	1	0
G	1	0	0	1	1	1	1	0
B	0	1	1	1	1	1	1	0

	C2						M2	
R	1	0	1	1	0	0	1	0
G	1	0	0	1	1	1	1	0
B	0	1	1	1	1	0	0	1

	C3						M3	
R	1	0	1	1	0	0	0	0
G	1	0	0	1	1	0	0	0
B	0	1	1	1	0	1	0	0

	C4						M4	
R	1	0	1	1	0	0	1	0
G	1	0	0	1	1	0	1	1
B	0	1	1	1	0	1	0	0

Figure 2.1: Illustration of how the message is saved within its cover image (Curtis, 2009; Haslam, 2007).

2.3 Social Media

This section will focus on social media, specifically whether it is possible to retrieve messages hidden in images shared on various social media websites. We will be looking at how some of the most popular social media sites process user-uploaded images, and if it hinders easy distribution of hidden messages. In the end, it will be concluded whether or not it is worth dedicating the project to social media based on the knowledge acquired in this section.

2.3.1 Using Social Media for Activism

Social media plays a big role in activism. It has been argued that the Egyptian Revolution in 2011 was made possible partly due to the ease of sharing information via social media (Eltantawy and Wiest, 2011). These networks could be used for readily sharing critical information such as staging protests and keeping the outside world updated on the status of the country. The large amount of users on social media makes them an obvious platform for activists, as they can easily reach their target audience.

The same reasoning can be applied to using social networks for sharing embedded information using steganography. By sharing information on social networks, the information can quickly be distributed to countless other activists. By embedding data in something inconspicuous, detection by the opponents of the cause can be avoided.

Most social media rely on video, images and text as the main components in information sharing. Because of this, the following section will examine how the social media process images uploaded to their sites, and if it is possible to send embedded data through them.

2.3.2 Image Sharing on Social Media

To test the level of compression on different social media, images encoded in different file formats were uploaded to multiple social media sites. After being uploaded, the compressed file was then downloaded and examined. To examine how much the compression algorithms distorted the images, the euclidean distance between the original image and the compressed version was calculated as described by Wang et. Al (Wang, Zhang, and Feng, 2005).

The results of using images saved with JPEG encoding can be seen in table 2.1. As can be seen in the results, some images were scaled in the encoding process, which makes computing the euclidean distance impossible without scaling the resulting image or the original image. Seen from the perspective of the sender, who sends hidden data embedded in an image, it does not make sense to scale the

Table 2.1: Social media compression on JPEG

	Before compression		After compression		
Social medium	Image resolution	Image size	Image resolution	Image size	Euclidean distance
Facebook	3648x2432	2.33MB	2048x1365	149KB	4884†
	1824x1216	1.13MB	1824x1216	126KB	2382
	912x608	369KB	912x608	44.5KB	1672
Twitter	3648x2432	2.33MB	1024x683	77.4KB	8731†
	1824x1216	1.13MB	1024x683	78.2KB	3619†
	912x608	369KB	912x608	65.4KB	1280
Imgur	3648x2432	2.33MB	3648x2432	2.32MB	0
	1824x1216	1.13MB	1824x1216	1.12MB	0
	912x608	369KB	912x608	355KB	0

†Image has been scaled to the original image size before calculating euclidean distance, which may have skewed the results.

original image, as the embedded data will be lost. Instead the resulting image is rescaled to the original width and height, before computing the euclidean distance.

From the results it can clearly be seen on JPEG images that Twitter compresses the file the most in terms of both file size and resolution. Combined, this results in a higher euclidean distance, as the image loses more information.

On the other hand, the image sharing site Imgur changed next to nothing in the uploaded JPEG image, and when downloaded again, all of the images reached a euclidean distance of 0, which means that every pixel in the images were equal.

After the initial tests of the compression systems, the same test was run with images saved with PNG encoding. This time, however, embedded in the images, using the LSB method described in section 2.2.2, was a secret image. The results from these tests are shown in table 2.2. From the results, it is clear that being able to preserve all pixels is a rare occurrence when sharing images on social media. In only two of the nine tested images the hidden image could be extracted. These images were the lowest in both resolution and file size, which shows that a threshold for when the social media compress the uploaded images exists. Below these thresholds both Twitter and Imgur could be used to send data without it being compressed. In the tests conducted, the highest amount transferred was $912 \text{ pixels} \cdot 608 \text{ pixels} \cdot 3 \text{ bytes/pixel} \cdot 2 \text{ bits/byte} = 3.327 \cdot 10^6$ bits which is enough to save $415,872$ characters using their ASCII value.

Table 2.2: Social media compression on PNG

	Before compression		After compression				
Social medium	Image resolution	Image size	Image resolution	Image size	Image format	Euclidean distance	Image extracted
Facebook	3648x2432	18.1MB	2048x1365	152KB	JPEG	5936†	No
	1824x1216	5.08MB	1824x1216	142KB	JPEG	2774	No
	912x608	1.35MB	912x608	49KB	JPEG	1775	No
Twitter	3648x2432	18.1MB	1024x683	79.4KB	PNG	9373†	No
	1824x1216	5.08MB	1024x683	80.9KB	PNG	4006†	No
	912x608	1.35MB	912x608	1.02MB	PNG	0	Yes
Imgur	3648x2432	18.1MB	3648x2432	419KB	JPEG	4227	No
	1824x1216	5.08MB	1824x1216	141KB	JPEG	2762	No
	912x608	1.35MB	912x608	1.35MB	PNG	0	Yes

†Image has been scaled to the original image size before calculating euclidean distance, which may have skewed the results.

From the tests conducted it is clear that when using social media to share information embedded in conventional image files, it is critical to understand how the social medium processes the file, before serving it to the users of the site. Otherwise you can very easily end up with your data being scrambled beyond repair, and the secret message lost.

After looking at the various social media, we have become aware a lot of the uploaded files are images (Meeker, 2014). It does not take that long to upload a picture, and social media relies heavily on images as a large component of information sharing. This is the reason for the focus on images in the following section on image compression in social media.

2.3.3 Why Use Image Compression?

Social media like Facebook, Twitter, and Imgur all use some form of image compression to reduce the size of user-uploaded images. This is partly shown in table 2.2. But why do these sites compress images? An obvious reason is to make room for more data on the sites' servers. This benefits the sites themselves. Another reason is that not every social media user has an unlimited amount of internet data, which means that there is a certain pressure on the websites to be lightweight for the benefit of their users. South Africa is a good example of this. According to a report from 2012, (Chetty et al., 2012), a vast majority of home-owners with internet access in South Africa had caps on their data plans. Another example is the result of a poll by CivicScience (Peoples, 2014) cited by the Billboard magazine. The poll suggested that 37% of respondents avoided streaming music while on their phones due to fear of overages for using more data than what their plan allowed.

Saving data for the users' benefit is therefore important for a lot of websites, perhaps even more so for social media sites, which often place focus on user-uploaded pictures. Mobile users can also benefit even more from well-compressed images, since their data plans may not only be limited, but also slow.

A paper by Facebook from 2010 (Beaver et al., 2010) states that - at the time of the paper's writing - there were approximately 260 billion images on Facebook's servers, translating to over $2 \cdot 10^{16}$ bytes (20 petabytes) of data. This means that in 2010, each image stored on Facebook's server was about 77 kilobytes in size after compression. Looking at the test results shown in table 2.2, we can see that the uncompressed version of the images would have been close to 2 MB in size. If Facebook stored the same images uncompressed, it would have required about $5.2 \cdot 10^{17}$ bytes (520 petabytes) instead. And this is from 2010.

A 2014 edition of an annual report by the venture capital firm KPCB, (Meeker, 2014), states that Facebook users alone uploaded about 350 million images per day to the website at the time of the report's writing. The report shows that users uploaded about 200 million images per day in 2010. Extrapolating from this data gives a net result of about 810 billion pictures on Facebook at the time of writing. If the average size of a compressed image on Facebook is unchanged since 2010, the space required is about $6.2 \cdot 10^{16}$ bytes (62 petabytes). Using the data from table 2.2, this would require about $1.6 \cdot 10^{18}$ bytes (1.6 exabytes) of space.

The calculations described in this section are just for Facebook and do not include backups. Instagram, Imgur, and Snapchat are all based on user-uploaded images, and they all need to store a vast amount of information. This is one of the reasons why image compression is so widely used on social media.

Conclusion

Now we are aware that social media generally compresses the images that get uploaded and thereby make it impossible to extract the original hidden image. This information has led us to move our focus away from social media and instead keep our focus more on the various image formats alone, instead of their relation to social media.

2.4 Image File formats

After researching social media it has become apparent that images are a widely shared medium over social media and they seem less suspicious than, for example, linking to an external file. This section will study different image formats and how they can be used as both cover and message files.

Image file formats are a standardised way for storing and organising digital images. Image files may store data in uncompressed, compressed or vector formats, though once rasterised the image becomes a grid of pixels each with a number of bits to designate its colour.

2.4.1 Bitmap

The BMP file format stores the colour components of each pixel separately, and therefore achieves a high file size. Though easy to work with, it is not a very common image file format to find on social media due to the file size.

The same goes for steganography, it is easy to hide information in the image, but the problems arise in that the file size of these images can become quite large in comparison to other more used image formats, and because of this, it is not used as widely as the the following image formats.

2.4.2 Joint Photographic Experts Group

JPEG is a widely used image format, which stems from the compression method used, as it is able to reduce the file size of an image substantially. This compression has a drawback though, in that it is a lossy compression method, meaning a compression will remove data from the original image and therefore degrade the image quality. Because the JPEG compression method is lossy, we cannot use simple methods like LSB for JPEG steganography because we do not have a good control over each individual pixel.

2.4.3 Portable Network Graphics

PNG offers a wide range of colour depths, from 24-bit (8 bits per channel) to 48-bit (16 bits per channel) and even up to 64-bit when an alpha channel is added. The compression method used for the PNG image format is lossless, so no data is lost under compression. Because the compression is lossless and PNG is widely used on the internet makes it ideal as a cover for use in steganography.

Conclusion

After looking at the various file formats and how image compression works we have decided to work with the JPEG format. As discovered in 2.3.3, we realised that most sites convert images to JPEG when uploaded. Because of this, JPEG appears to be the most commonly used file format when it comes to sharing images online. This is why the JPEG format will be the focus throughout the rest of the report.

2.5 A Study of the JPEG file format

This section is the result of researching the JPEG file format. The information in the following sections have been combined from multiple sources (Cuturicu, 1999; Miano, 1999; Union, 1992; Hamilton, 1992).

2.5.1 An Overview of JPEG

JPEG is an image format defined by the Joint Photographic Experts Group. This type of image requires much more computing power to process, both in terms of encoding and decoding, compared to a much more simple format like Windows BMP. This will become more clear in the following sections. A crucial thing to note about JPEG is that the encoding process is lossy, which means that when saving an image to JPEG, you lose information about the individual pixels. This is one of several things that allows the image to be compressed.

2.5.2 JPEG as a File Format

JPEG is a very comprehensive standard, which defines the inner workings of the JPEG compression. What this standard does not describe however, is an actual file format (Miano, 1999). The standard does not define an encoding of images that all JPEG encoders can decode. An example of this is that the standard does not define how colours are represented in the format, which means that one decoder could potentially use an RGB colour space, while another one would use an RBG space. Both systems are perfectly valid according to the JPEG standard.

Without a way to ensure that all decoders will read an encoded image the same way, an image file is not worth much. This is why the JPEG File Interchange Format (JFIF) specification was developed by Eric Hamilton (Hamilton, 1992). JFIF defines a standard which all JPEG files must abide. This standard includes a definition of the colour being encoded in one or three channels. One channel if it is a monochrome image, and three channels if it is a true-colour image. If encoded in three channels, the colour space used is YCbCr, if only one channel is used, only the luminance (Y) channel is used.

The JFIF specification is what allows JPEG images to be as widespread as they are today. JPEG is used by large services like Facebook to store and serve small images of reasonable quality.

2.5.3 The Process of Encoding a JFIF Image

When encoding a JFIF image, we start off with a Bitmap, that is, a two dimensional array of pixels each consisting of an R, G and B value. The first step is transforming those colour channels into the colour space used in JFIF images.

2.5.3.1 The Colour-space

When describing colours in terms of computer science, we often use the colour space RGB, where each component corresponds to the intensity of the red, blue and green pixels that make a screen. A colour in the YCbCr colour system is made of the luminance (Y), which describes light intensity together with the blue-difference and the red-difference chroma components. Together they describe the actual colour.

When converting from RGB to YCbCr matrix multiplication can be used to find the Y, Cb and Cr components as a matrix product:

$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ -0.1687 & -0.3313 & 0.5 \\ 0.5 & -0.4187 & -0.0813 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Likewise, we can calculate the R, G and B components from the YCbCr colour space:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} Y & + & 1.402 & \cdot & (Cr-128) & & & & \\ Y & - & 0.34414 & \cdot & (Cb-128) & - & 0.71414 & \cdot & (Cr-128) \\ Y & + & 1.772 & \cdot & (Cb-128) & & & & \end{bmatrix}$$

These values all range from 0 to 255, like the R, G and B values. However, when these values get encoded, it is more efficient to have them centred around 0, so that the values range from -128 to 127. To do this, we simply subtract 128 from each Y, Cb and Cr value.

2.5.3.2 Sampling

When looking at an image, the human eye is far more sensitive to small changes in the luminance channels than they are to changes in the chroma channels. We can use this fact to compress our image even further in terms of file size using a technique called sampling.

The JPEG standard allows each component to be sampled. Before sampling, the image must be split into smaller blocks, called the Minimum Coded Units (MCU). The default size of an MCU is 8x8 pixels, but it changes when sampling is used. The most commonly used sampling is 4:2:0 where the chroma channels were downscaled by 50% in both directions.

For a sampling of 4:2:0 we calculate luminance for each pixel in the image, while only calculating the chroma channels for each 2x2 square. This gives us a sampling of 2x2 for the luminance component, while the chroma components have a sampling of 1x1. This means that for every pixel in the x direction we calculate the chroma components, we calculate two luminance components, and the same with the y direction. In other words, we have fourfold the information about the luminance component than of the chroma components.

Many other samplings can be used, for example if a 1:1:1 sampling is used, all information about every pixel is saved.

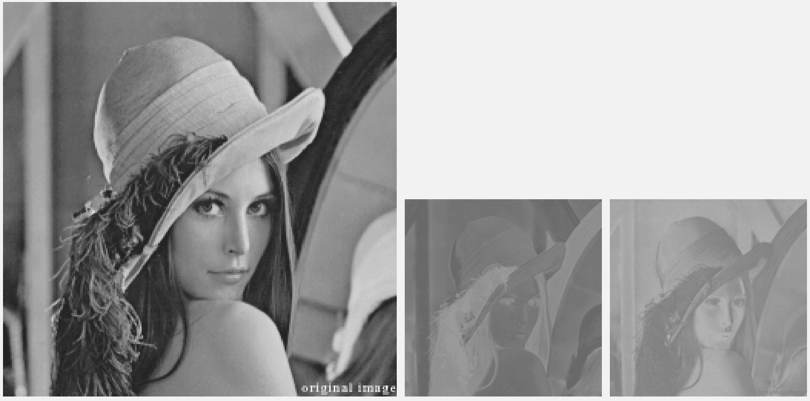


Figure 2.2: Y, Cb and Cr channels of an image(*Volume 3: Miscellaneous Lenna Image*)

2.5.3.3 Discrete Cosine Transform

For each 8x8 block of pixels in our MCUs we perform DCT. The two-dimensional DCT is given by:

$$G_{u,v} = c(u,v) \sum_{x=0}^{7} \sum_{y=0}^{7} g_{x,y} \cos\left(\frac{(2x+1)u\pi}{16}\right) \cos\left(\frac{(2y+1)v\pi}{16}\right)$$

where:

- u is the horizontal spatial frequency
- v is the vertical spatial frequency
- $c(u,v) = \begin{cases} \frac{1}{8} & \text{if } u = v = 0 \\ \frac{1}{4\cdot\sqrt{2}} & \text{if } u = 0 \text{ or } v = 0 \\ \frac{1}{4} & \text{otherwise} \end{cases}$
- $g_{x,y}$ is the value of pixel at coordinates (x,y)

2.5.3.4 Quantization

Quantization tables are what makes JPEG compression lossy. After the DCT process, the quantization progress is performed on the resulting 8x8 block of values. Quantization is the process of dividing each value in the 8x8 block with a corresponding value in a quantization table. The JPEG specification allows four quantization tables to be used, while JFIF limits it even further to two tables. This means that two of our three channels have to share quantization tables. As the human eye is much more sensitive to changes in the luminance channel, JFIF specifies that the luminance channel gets its own quantization table, while the chroma components must share a quantization table.

Although the JPEG standard does not force you to use default tables, they do provide some tables which have shown good results with their test images. The proposed quantization tables for the image channels are shown below.

Proposed Quantization Table for Luminance Channel							
16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
15	14	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	24	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Proposed Quantization Table for Chroma Channels							
17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

2.5.4 Zigzag-ordering

Because of the quantization process, there are now a lot of zeros in the lower-right corner of the 8x8 block of pixels. This feature can be utilised to compress the image data, but first the table must be ordered so that the zeros are placed next to each other.

To do this, the 8x8 block of pixels are now read in a zigzag manner displayed on figure 2.3. By reading the block this way, you have a 64 dimensional vector with a lot of consecutive zeros.

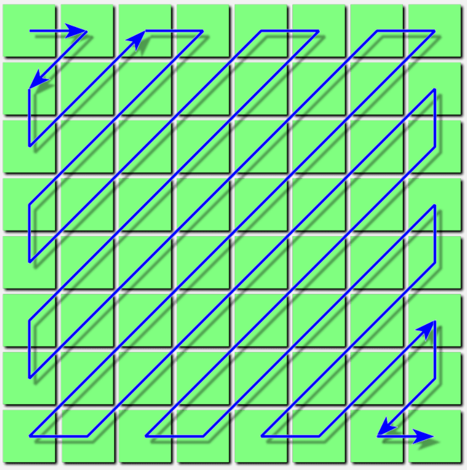


Figure 2.3: Zigzag-ordering of the 8x8 block of pixels

2.5.5 Encoding the Vector

The first entry in the quantization vector is called the DC coefficient, and will be encoded differently from the other 63. To encode the last 63 entries called the AC coefficients, run-length encoding (RLE) is used. Each value different from zero in the vector is represented as $(\text{Number of zeros before this entry}; \text{Entry})$. Because of the many consecutive zeros in the data, this allows a very compact encoding of the numbers. If at one point in the vector only zeros remain, we end the block with the EOB value (0,0).

The DC coefficient is often a large number, so it would be a waste of bits to write it in full to the file each time. Instead only the difference from the previous DC coefficient is stored, so that smaller numbers need to be encoded. So when reading a JPEG file, a decoder must keep track of the previous DC coefficient. For the first DC entry in the MCU, the decoder can assume that the last DC coefficient was 0.

There are some restrictions to this encoding, as only a nibble is used to represent the number of preceding zeros, the maximum value we can denote is $F_{16} = 15_{10}$. If there are more than 15 consecutive zeros, we can encode 16 zeros as (15,0), and by using multiple of these values, denote any number of consecutive zeros. The value (15;0) is known as ZRL.

2.5.6 Huffman Encoding

After the run-length encoding, the pairs must be written to the file, but before that can happen, the JPEG standard defines even more compression of the data. Instead of storing the actual values from the vector, a category of the number, along with a bit representation is stored.

The category of a number is the least amount of bits required to represent the number. The category of numbers in the range $[-32767, 32767]$ are shown in the table below (Cuturicu, 1999).

Values	Category	Bit representation
0	0	-
$-1, 1$	1	$0, 1$
$-3, -2, 2, 3$	2	$00, 01, 10, 11$
$-7, -6, -5, -4, 4, 5, 6, 7$	3	$000, 001, 010, 011, 100, 101, 110, 111$
$\vdots$	$\vdots$	$\vdots$
$-32767, -32766 \ldots, 32766, 32767$	16	$0000000000000000 \ldots 1111111111111111$

The vector entry of $(0, 6)$ would become $(0, (3, 110))$ when encoded as shown above. The last step is to Huffman encode these pairs of numbers. In a 8x8 block of pixels, there may be a lot of $(0, 6)$, so instead of having to spend two bytes every time this pair appears, we use Huffman Tables.

A Huffman Table is a table where you can look up pairs of numbers and get a bit representation which corresponds to the pair of numbers. To complete the example with the entry $(0, 6)$ which was encoded to $(0, 3, 110)$, $(0, 3)$ which we call the Run/Size is looked up in the Huffman Table (using the default Luminance AC table providided in the JPEG standard) and it can be seen that it has a category of 3, and a bit representation of 100.

So instead of having to write 0000000000000110, which is the byte representation of $(0, 6)$, it is enough to write the Huffman encoded 100110.

There are no universal Huffman tables, and JPEG allows an encoder to define its own. They do, however, provide some Huffman tables which have shown good results with a variety of images (Union, 1992, p. 153).

2.5.7 Dissection of a JPEG File

In this section a JFIF file which is compatible with the JPEG standard will be explained. The explanation is split into the same segments as the file itself is split into.

2.5.7.1 Segments in a JPEG File

The JPEG standard defines that a JPEG file must be ordered in segments. These segments are each indicated by a marker, and the most common segment markers are shown in table 2.3. Most of the markers are followed by two bytes of data describing the length of the following segment. Knowing the length makes it possible for a decoder to skip segments it does not know how to decode.

As shown in table 2.3, all markers start with the byte 0xFF. A special case exists, where 0xFF does not define a marker, and that case is when the following byte is 0x00. More on this later, when it is examined how image data is written to a file. The tables in the following sections describing each marker, are reproduced from the book *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*(Miano, 1999).

Table 2.3: Most commonly used markers in JPEG files

Marker	Identifier
0xFFD8	SOI
0xFFEn ($n = \{0 \ldots F\}$)	APPn
0xFFD8	DQT
0xFFC0	SOF
0xFFC4	DHT
0xFFDA	SOS
0xFFD9	EOI

2.5.7.2 The SOI and EOI markers

Start of Image Segment

Size in bytes	Description
2 bytes	SOI marker

End of Image Segment

Size in bytes	Description
2 bytes	EOI marker

The first marker a JPEG decoder meets in a JPEG file is the SOI marker. The marker does not specify a length in the following bytes, since the segment only consists of the marker itself. Likewise, the last marker a decoder will meet is the EOI marker, which specifies that all data has been read from the image.

2.5.7.3 The APP*n* marker

APP*n* Segment

Size in bytes	Description
2 bytes	APP*n* marker
2 bytes	Segment length
variable	Application specific data

The JPEG standard defines the APP segments to be application specific data, which can be placed anywhere in the image file. Applications can then use these segments to store additional information about the image file. JFIF specifies that the APP_0 is used for identifying the file as a valid JFIF file, and that the APP_0 segment must be placed immediately after the SOI marker.

By convention the APPn segments use a null terminated string for identifying themselves. The JFIF APP segment as found in almost all JPEG files can be seen in table 2.4.

JFIF APP_0 Segment

Size in bytes	Value
2 bytes	APP_0 marker
2 bytes	Segment length
5 bytes	Null terminated string “JFIF”
2 bytes	Major and minor version number
1 byte	Units for setting photo density
4 bytes	X-density and Y-density
2 bytes	Width and height of thumbnail image
variable	Thumbnail data saved in RGB colour space

Table 2.4: JFIF APP_0 Segment

2.5.7.4 The DQT marker

Define Quantization Table Segment

Size in bytes	Description
2 bytes	DQT marker
2 bytes	Segment length
4 bits	Quantization value size (0 = 1 byte, 1 = 2 bytes)
4 bits	Table identifier (0-1)
64 or 128 bytes	Unsigned quantization values

The Define Quantization Table segment encodes the 8x8 block of integers used in the quantization process. The table can use either one or two bytes for storing the integers in the quantization tables. Therefore a nibble is used for selecting how many bytes to use.

The JFIF standard allows a JFIF file to use two quantization tables, which means that some channels have to share quantization tables. Because of this, each quantization table gets an ID, so that each component can point to a quantization table.

While encoding the actual values of the table, they are encoded top to bottom, left to right.

2.5.7.5 The DHT marker

Define Huffman Table Segment

Size in bytes	Description
2 bytes	DHT marker
2 bytes	Segment length
4 bits	Table class (0 = DC table, 1 = AC table)
4 bits	Table identifier (0 or 1)
16 bytes	A byte for the count of Huffman codes for each length 1-16
Variable	The one byte codes encoded by the Huffman Table

The Define Huffman Table segment describes how values are encoded in the image data. The first part of the segment defines whether the table is used for DC or AC components. Next an ID is defined, so that each channel can be linked to a certain Huffman Table.

The next 16 bytes define how many codes of a certain length are encoded. The first byte describes how many codes of length one are encoded, the next how many codes of length two are encoded and so on, until all 16 bytes are used.

Now, the actual one byte codes which the Huffman Table has to convert, are supplied. There are as many bytes as the sum of the previous 16 bytes. The order in which they are supplied defines which codes in the Huffman Table they become encoded as.

2.5.7.6 The SOF marker

Start of Frame Segment

Size in bytes	Description
2 bytes	SOF marker
2 bytes	Segment length
1 byte	Sample precision in bits
2 bytes	Image height
2 bytes	Image width
1 byte	Number of components

The Start of Frame segment defines the components in the frame. First, the height and width of the image in pixels are encoded. After that, a byte is used for defining the number of following components. By using a byte for the number of components, the JPEG standard allows a total of $11111111_2 = 255_{10}$ components, but JFIF limits this to 1 for grey-scale images or 3 for true-colour images.

Next follows a Component Segment for each of the components defined in the SOF segment. Firstly the segment defines an ID for the component. Again, JFIF states that we must use the ID 1 for the luminance channel, 2 for the chroma-blue channel and 3 for the chroma-red channel.

Next, each component supplies the scaling of the particular component, both horizontal and vertical. Lastly, the ID of the quantization table to use for each component is supplied.

Component Segment

Size in bytes	Description
1 byte	Component identifier (1 = Y, 2 = Cb, 3 = Cr)
4 bits	Horizontal sampling
4 bits	Vertical sampling
1 byte	Quantization table identifier

2.5.7.7 The SOS marker

Start of Scan Segment

Size in bytes	Description
2 bytes	SOS marker
2 bytes	Segment length
1 byte	Number of components
2 · "Number of components" bytes	Scan component descriptor
1 byte	Spectral selection start
1 byte	Spectral selection end
1 byte	Successive approximation

The Scan Segment is where the actual image information is stored. First we have the Start of Scan segment which is a fixed length segment describing what image data will follow. Again the segment describes how many components to encode, then for each component supplies their Scan Component Descriptor which contains an ID for the component, together with the linked Huffman Tables for the DC and AC components.

Scan Component Descriptor Segment	
Size in bytes	Description
1 byte	Component identifier
4 bits	DC Huffman Table identifier
4 bits	AC Huffman Table identifier

2.5.7.8 The scan data

After the SOS segment, the scan data follows, which is the actual image information. The data is not preceded by any marker. This is where the Huffman encoded data is written, and is the data that the image processors use for calculating the actual pixel's colours.

In the beginning of this section it was explained that a marker with the name FF00 does not exist. The reason for this is that if the scan data must encode the byte $11111111_2 = FF_{16}$. It cannot do so without it being read as a marker. Therefore the JPEG specification reads that two consecutive bytes FF00 must be read as FF and not as a marker.

2.6 State of the Art

Different forms of steganography have been invented, either because there was an actual need to send a confidential message, or because it has been seen as a challenge by someone who has an interest in the field. In this section some of the tools already available for performing digital steganography will be mentioned and examined in the hopes of gaining a greater understanding of them. This could potentially give an idea of what these tools could be missing or otherwise do wrong in a real-life context.

EzStego is a Java-based steganography tool for embedding information in GIF-files. GIF images are limited to a palette of 256 colours per pixel, and EzStego takes advantage of this to embed information into the pixels. A GIF palette is usually sorted by luminance, but this is not ideal, since two colours with similar luminance could be very different. EzStego sorts a GIF palette in a way such that two adjacent colours are very similar, which means that if the colours are switched, it is unlikely that a human would notice. The embedding of data in EzStego works by switching colours if the bit to be embedded is '1', and leaving the colours if it is '0'. This will create an image that looks a lot like the original image, and it is done without changing the file size. It is, however, possible to extract the hidden data if the created stego-image is compared pixel-by-pixel to the original image (Westfeld and Pfitzmann, 2000).

JSteg-Jpeg has its focus on JPEG steganography, as the name suggests. The programme reads an image file and a secret message, which are then combined into a JPEG image. It utilises the fact that JPEG compression is split into two stages, the lossy stage using DCT and quantization to compress the image data, and the lossless stage, using Huffmann encoding to further compress the data. The secret message needs to be hidden in the image data between the two stages, since it would otherwise be lost in the lossy stage. The rounding process in the quantized DCT coefficients are also modulated. JSteg-Jpeg allows for adjustment of the quality factor of the JPEG compression as this together with the size of the embedded message determines the degradation of the quality of the final image ("Image Processing and Pattern Recognition: Fundamentals and Techniques").

OpenPuff is a freeware steganography and watermarking tool by Cosimo Oliboni and is a more modern approach to steganography, first released in 2004, last updated July 2012. It lets the user hide data by splitting it into several carrier files forming a so-called carrier chain. The carrier files supported include image files like JPEG and PNG, audio files like MP3 and WAV, and video files like 3GP and MP4. The programme implements up to five layers of security with the first four layers being encryption, scrambling, whitening (mixing scrambled data with noise) and encoding the data. Finally the data is injected in the carrier files (*OpenPuff Manual*). The principle behind deniable encryption, is that the existence of an encrypted file or message cannot be proven (Czeskis et al., 2008). This same principle is extended to steganography with OpenPuff, since the existence of a plaintext message cannot be proven. As so, OpenPuff not only hides data, but secures it with 256 bit cryptography.

This is just a small selection of the tools available. Most other programmes utilise simple methods similar to some of the examples and choose to deal with only simple file formats like BMP or GIF. It should be mentioned that several programmes do not share the specifics about the techniques used.

2.7 Hiding Data in JPEG images

Based on the information on JPEG in section 2.5 and the programmes discussed in section 2.6 it has been possible to explore various options in regards to concealing data in JPEG images. The following are simpler methods of this, where the data is simply ignored by the decoder, and therefore the image appears unchanged.

Comments — It is possible to store plain text in a JPEG-file in a comment marker (bytes 0xFFFE). Comments are usually used to store copyright information, but any information can be placed in the segment. Comments are ignored by decoders, but if the file is inspected, the marker can be clearly seen.

After EOI — The End of Image (EOI) marker (bytes 0xFFD9), and any data following that is ignored by the decoder. It is therefore possible to store information after the EOI marker, in much the same way as the comment marker.

Progressive bits — The Start of Scan (SOS) marker (bytes 0xFFDA) precedes 6 bytes of information, where only three of those are needed in sequential (and baseline) JPEG-compression. This means that it is possible to store a very small amount of data when encoding baseline JPEG. It may also be harder for a human to notice any hidden information in the SOS-section of a JPEG-image than in a comment or after an EOI-marker.

APP$_n$ markers — Application markers ($\text{APP}_0-\text{APP}_{15}$, bytes 0xFFE0-0xFFEF) are used for application-specific information. Apart from APP_0, which is used by the JFIF-format, any application can create one or more APP-markers to store information. These markers can be used to store secret information, though, as with comments, it is very easy for a human to see.

The next methods that will be described are based on our experiments with LSB in section 2.2.2 and from the study of JPEG in section 2.5.

LSB in Quantization tables — It is possible to hide information within the least significant bit of a quantization table without arousing suspicion. However if this is done, we will only get up to 128 bits that we can change, which is 16 bytes, or 16 characters, but still enough to list something like a certain time and place. As an example: "Basisbaren 12am'\0' ", is 16 characters.

Template from Huffman Tables — The Huffman tables are very important in producing a compressed JPEG file. Some exist that generally work well, but to get the optimal solution, it is necessary to analyse the image you want encoded, and then create Huffman tables based on that analysis. Thus, it is also possible to create Huffman tables where some of the values represent a character, and then provide the people you wish to see the encoded message with a template for the Huffman tables you are using, telling them in which order to read the pixels for the hidden image to show itself.

LSB on JPEG thumbnail — The JPEG standard also defines a thumbnail, which can also act as a cover image for a hidden image, or text, by doing something similar to what we previously did using the LSB method in section 2.2.2. Doing this will give us many more bytes to work on, compared to the two previously mentioned suggestions. Even for a 64x64 thumbnail, we will be able to encode 1536 characters if only the least significant bit is changed. 1536 characters is significantly more than the 16, which can be hidden within the quantization tables.

Lastly we have two more complicated methods based on two separate articles about steganography in image files.

A Reversible Data Hiding Scheme for JPEG Images — Qiming Li et al. proposes a method for embedding data into the DCT coefficients (Li, Wu, and Bao, 2010). The process is split into three parts. The selecting algorithm which selects a subset of the AC-DCT coefficients, for storing information. The embedding process which embeds the actual data into the image, and lastly the decoder which reverses the process, and retracts the hidden data from the image. Because of the way JPEG encoding works, as described in section 2.5, small changes in the DCT coefficients or the quantization, will not lead to visible distortion of the JPEG image.

A Graph-Theoretic Approach to Steganography — Stefan Hetzl and Petra Mutzel define a method where graph theory is used for data into a list of values (Hetzl and Mutzel, 2005). Used on a bitmap image, the algorithm finds which pairs of pixels need to be interchanged, to be able to store data hidden in the image. This process is in many ways superior to methods such as LSB, due to the fact that very few pixels are actually changed while using this approach, pixels are merely interchanged. This makes it much more difficult to use statistical analysis on the image, to figure out if an image contains embedded data. With JPEG images however, pixels are not set separately, and two pixels cannot be interchanged very easily. Instead, this method can be used on the DCT coefficients, and have the data embedded into them. This approach will be described in-depth in section 2.9.

2.8 Steganalysis

Steganalysis is the disipline of detecting information hidden with steganography and if possible, recover that information. Because of the nature of steganography, it is not always obvious if a file has hidden information or not. There are many different techniques used in detection of hidden data. Some of these techniques are described below.

2.8.1 Detection

Most steganography algorithms are able to hide data without any notable visual impact on images, an example being LSB where it is not easily visible on the images

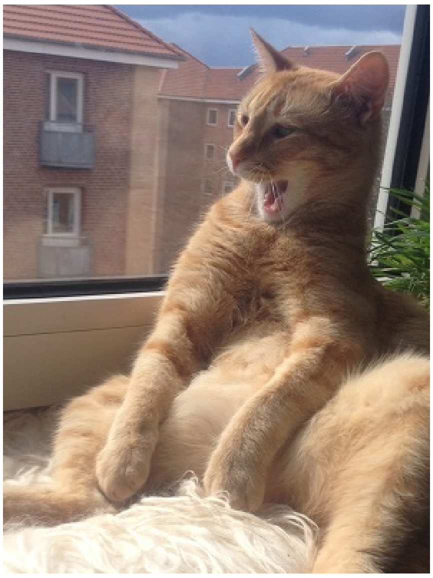


Figure 2.4: Cover image.

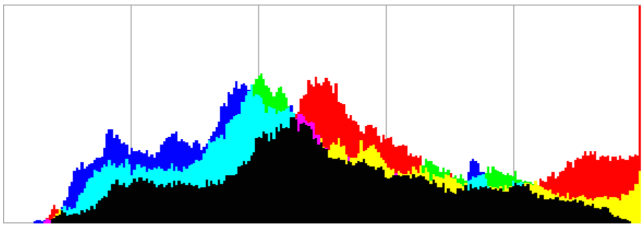


Figure 2.5: Colour histogram of image shown in figure 2.4 without hidden data.

where changes have been made to the pixels. It is therefore hard if not impossible to detect if something is hidden by simply looking at the image.

Detection of hidden data is therefore done with statistical tools like looking at histograms, modified DCT coefficients and signatures of different steganography algorithms.

Colour Histogram A colour histogram is a representation of the distribution of colours in an image. This can be used to detect unusual distributions of colour, which may occur when modifying the least significant bit or bits of each colour channel.

We have our cover image shown in figure 2.4 and its respective colour histogram in figure 2.5. We then hide another image in our cover image, using the implementation of LSB in section 2.2.2. This produces a new colour histogram shown in figure 2.6. The new stego image has quite an unusual histogram, though unfortunately we cannot be sure there is hidden data only by looking at these

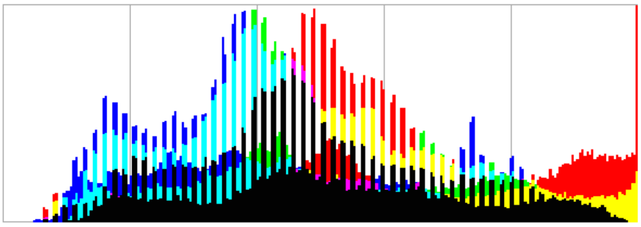


Figure 2.6: Colour histogram of image shown in figure 2.4 containing secret image hidden with LSB.

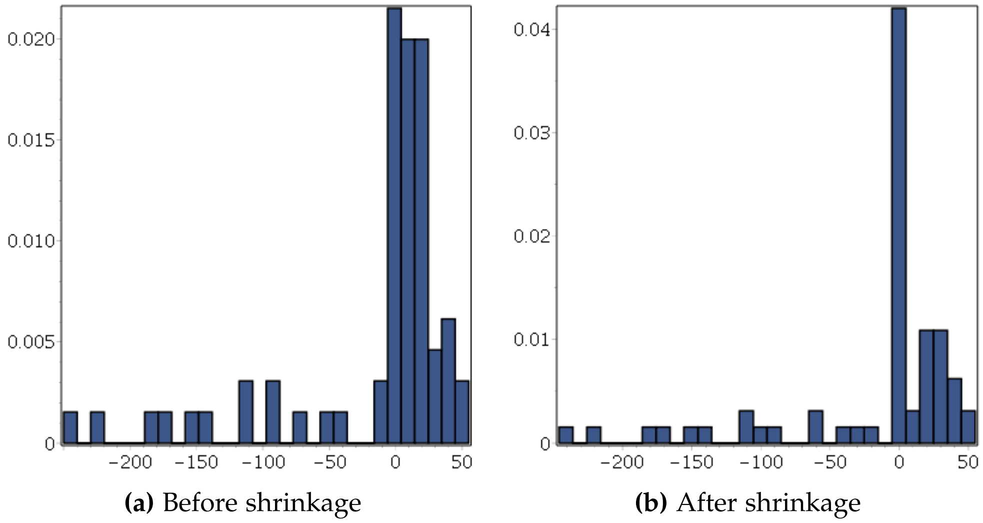
Figure 2.7: Cover image for algorithm employing shrinkage (Christensen, 2008)

histograms.

This unusual histogram might just be a product of a compression algorithm.

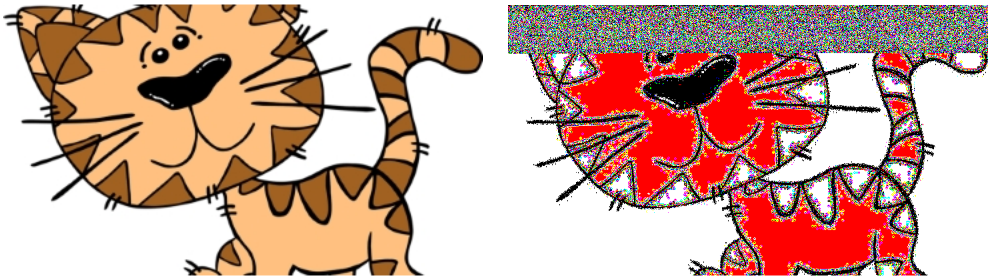
DCT coefficients histogram To detect if data is hidden using an algorithm which employs shrinkage, we can look at a histogram of the DCT coefficients. When hiding data in the least significant bit of the DCT coefficients instead of just changing the bits, we can analyse each of the bits. If the least significant bit matches our message we move on, if it does not we subtract 1 from the DCT value, if the value is negative we add 1. This is called shrinkage. This method results in a high amount of zeros in the DCT coefficients. This can be seen on the comparison between figure 2.8a and figure 2.8b, these are histograms of DCT coefficients of figure 2.7. We can clearly see that after data has been hidden using an algorithm which uses shrinkage, as there are almost only zeros remaining.

LSB enhancing LSB enhancing works by doing the opposite of what LSB usually does. It eliminates all seven high-level bits for each byte. The least significant bit will then remain, making all bytes either 0 or 1. These are then enhanced by setting each 0 byte to the minimum 0 and 1 to the maximum 255. This makes the LSB of the image very visible, allowing for a visual check by simply looking for patterns. Figure 2.9a shows a 24 bit BMP image with 4KB of random data hidden in it, and figure 2.9b is the same image LSB enhanced. We can see that there is something hidden in the top half of the image, while the bottom half remains untouched (Westfeld and Pfitzmann, 2000). The reason for the cat's body going from brown



(a) Before shrinkage **(b)** After shrinkage

Figure 2.8: Histogram of DCT coefficients



(a) 24 bit bmp image with a 4 KB image hidden within. (Gerald G., 2016) **(b)** 24 bit bmp image LSB enhanced.

Figure 2.9: Image of a cat before and after LSB enhancing

to red is that colour of the brown is (255,192,128) in RGB. The only component ending with a 1-bit, is the red component, which then gets enhanced to 255.

Steganography signatures Unusual patterns in an image are obvious and arise suspicion, for example unused bits in the file headers or even comment headers, these might give us some insight into which algorithm was used.

2.8.2 Recovery

Recovery of data hidden with steganography is difficult. To recover the data we must know what kind of algorithm was used to hide the data. Even if we know the algorithm used, we might still only get back meaningless data, because in addition to hiding the data, it might also be encrypted.

So the most common way of recovering hidden data is by using the same algorithm that was used to encode the image to decode the data.

2.9 Overview of "A Graph-Theoretic Approach to Steganography"

This section is based on the article by Hetzl and Mutzel (Hetzl and Mutzel, 2005) and Olav Geil's lecture on the article.

2.9.1 Goal of the Method

Methods such as LSB can be used to embed a lot of information in images. With the implementation described in section 2.2.2 we were able to embed six bits of information in each pixel in a true-colour bitmap. The downside of this method, as described in section 2.8, is that it is very easy to detect because it forcibly changes a lot of pixels in the image.

The goal of the following method is to change as little as possible in the image, while still being able to embed its message.

In the following examples, we start out with the image seen on figure 2.10. The message to be concealed in the image is the binary string 10100110011010.

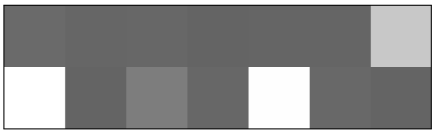


Figure 2.10: Grey-scale cover image

2.9.2 Terminology

First we define the function v as $v_m : \mathbb{Z} \Rightarrow \{0, 1, 2, \ldots, m-1\}$ such that $v_m(s) = s \mod m$. Furthermore we have the message, e a list of n elements, to encode in the image: $e = \{e_0, e_1, e_2, \ldots, e_n\}$, where $\forall i \, (e_i < m)$. Because all entries in the message list to be encoded must be less than m, m defines the number of actual bits that can be transferred in each pair of pixels. With $m = 2$ the only two possible values are 0 and 1, which means that only one bit can be embedded as one entry in the message vector. By choosing $m = 4$, the possible values are 0, 1, 2 and 3, or 00_2, 01_2, 10_2 and 11_2. In practice, this means that we can embed two bits in every entry in the message vector.

The $\oplus_m$ operator is used for computing sums modulo m. In other words, the operator describes the calculation $x \oplus_m y = v_m(x+y) = (x+y) \mod m$.

In the following example we will use $m = 4$, as this allows 2 bits to be used for each entry in the message vector. Our message vector is found by splitting the binary string into pairs of two bits, which then makes it $e = [10\,10\,01\,10\,01\,10\,10]$.

2.9.3 Method Described in Article

A grey-scale image can be viewed as a list of pixels $p = \{p_0, p_1, p_2, \ldots, p_n\}$, where $\forall i\,(0 \leq p_i \leq 255)$. These pixels are then ordered into consecutive pairs of two. Because we embed one entry from the message vector into each pair of pixels, an m value of 4, means that we embed two bit per two pixels, meaning the message vector can have the same dimension as the amount of pixels in the image. Were we to choose $m = 2$, only half of this message could fit in the same image. The $\oplus_m$ operator is then used on each individual pair, and compared to the message which is to be encoded. If the resulting remainder equals the wanted message, these pixels are a *perfect pair*. If not, they need to be changed by either switching two pixels around, or changing one of the values. In this method we want to avoid changing the values directly. Instead, we find all pairs of pixels that could be interchanged and result in perfect pairs. In table 2.5 this method is applied to the grey-scale image on figure 2.10 with 14 pixels, and the message vector with seven entries.

Table 2.5: Process of finding pixels in need of changing or switching

Pixel	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Value	106	102	103	100	101	101	200	255	100	125	103	254	104	100
$\oplus_4$	0		3		2		3		1		1		0	
Message	2		2		1		2		1		2		2	
	≠		≠		≠		≠		=		≠		≠	
Needed values	<104,100> <108,104>		<102,99> <106,103>		<100,100> <104,104>		<199,254> <203,002>				<100,251> <104,255>		<102,98> <106,102>	

From the table we can see that switching the following pixels would make more pairs correct $(4 \leftrightarrow 11)$, $(8 \leftrightarrow 12)$, $(2 \leftrightarrow 13)$ and $(1 \leftrightarrow 13)$.

As a graph, these switches can be represented as seen on figure 2.11. Each vertex represents a pair of pixels, where at least one of the pixels can be changed. The edges describe a switch between two pixels. This graph can then be used to determine which switches to make. Every time a switch is made, all other edges touching the two vertices where pixels are changed, are removed. This graph can be used to determine which switches are the best, by introducing edge weight, describing how much of a visual change a switch would result in.

Let us say that the switches $2 \leftrightarrow 13$ and $4 \leftrightarrow 11$ are chosen. The pixels after these switches are shown in table 2.6. We would still need to correct the pairs (5,6) and (7,8). There is no switch that allows us to correct these values, so instead the actual values are forcibly changed and by doing so, change the image ever so slightly.

The image obtained after these switches and changes is the one shown on figure 2.12b next to the original image shown on figure 2.12a.

By doing these switches, we minimise the need to alter the pixels in the image.

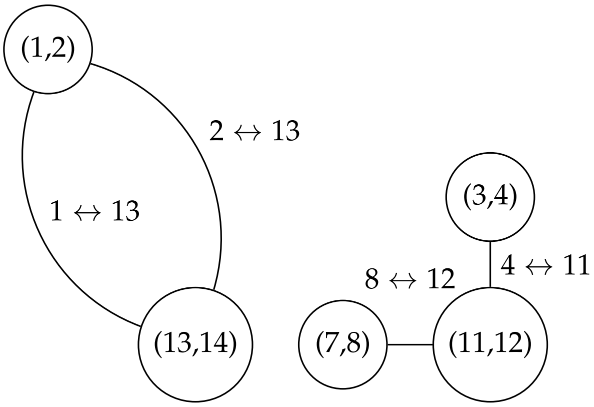
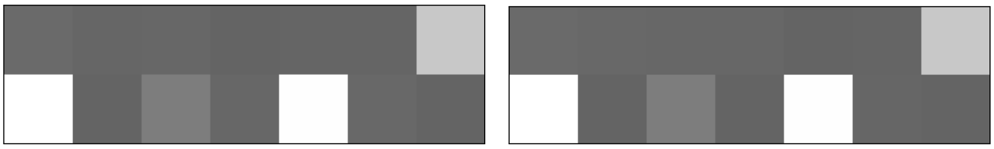


Figure 2.11: Graph containing the possible switches

Table 2.6: Pixels after switching

Pixels	p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}	p_{13}	p_{14}
Value	106	104	103	103	101	101	200	255	100	125	100	254	102	100
$\oplus_4$	2		2		2		3		1		2		2	
Message	2		2		1		2		1		2		2	
	=		=		$\neq$		$\neq$		=		=		=	
Needed values					<100,100> <104,104>		<199,254> <203,003>							



(a) Grey-scale cover image **(b)** Grey-scale stego image

Figure 2.12: Comparison of the image before (2.12a) and after (2.12b) the changes have been made.

With methods such as LSB it is rarely the case that pixels do not need to be altered. This method is much more resistant to statistical analysis, such as the colour histograms described in section 2.8.

When the message is to be extracted from the image, the same method of grouping pixels into pairs is applied. Now, instead of trying to find pairs to be switched, the value obtained after using the $\oplus_m$ operation are the values embedded in the image. These values can then be grouped into a vector, which is a copy of the original message vector.

2.10 The Confines of the Problem

Our initial problem statement specified that we would work with social media and how to use them to share data containing hidden messages. As previously mentioned our problem has been limited to only include images. We soon learnt that each social media we tested had a different compression system, as a means of saving server space, due to the vast amounts of images being shared every day.

We came to the conclusion that if we worked with social media there was a risk that trying to send images through them could result in the messages being destroyed by their compression systems. Even if we managed to find a way around it, the sites could change their algorithms with no warning, thereby rendering our work useless.

Despite not focusing on social media we are still going to use the knowledge acquired in section 2.3.3, about the compression of images. As discussed in section 2.3.1, JPEG is the most commonly used image format on the internet (W3Techs, 2016), and therefore should garner less attention than other formats. Using the JPEG image format also allows the ITC-group from our cluster group to save space in their system, if the files sent back and forth are compressed.

We have also decided to implement a version of the graph-theoretical approach described in section 2.9 that will work with the JPEG image format. This decision allows us to work closely with the mathematicians from our cluster group, who are developing an algorithm that is efficient at traversing a graph and selecting edges based on certain criteria. Working with the same method should ultimately help both groups understand it better and thereby form greater results.

Problem Statement

How do we modify the graph-theoretic approach for steganography described in section 2.9 to work with JPEG images where the data is hidden in the DCT coefficients without significant visual changes, and how do we implement this in an object-oriented programming language?

Throughout the entire project we will also be using object-oriented programming to understand algorithms by continuously developing small prototypes and experimental programmes. Some of these smaller programmes will be integrated in the final programme.

Choosing this project allows us to work with the two other groups in the cluster. The mathematicians will work on the theoretical definitions and inner workings of the graph-theoretic method, while the ITC group will implement a mobile application for sharing steganographic material between users. This means that we hope to receive algorithms from the mathematicians that we can implement in our programme, while we hope to deliver a class library with a working image encoder to the ITC group that they can implement in their application.

Chapter 3

Design

3.1 Using the Graph-theoretic Approach on JPEG images

In section 2.9, we took a closer look at the Graph-Theoretic Approach for Steganography. The goal of this section is to understand how the technique can be modified to work on JPEG images instead of grey-scale Bitmap images. The authors of the method define the following about using the technique on JPEG images (Hetzl and Mutzel, 2005):

- The samples (i.e. where the data will be embedded) are the coefficients of the DCT operation.
- The DC component must not be used for embedding data, as these are often large numbers and would result in visible distortion of the image, if those were to be altered.
- AC components with a value of 0 must not be used for embedding data. If they were to be used, the Run/Length encoding step would be useless, as there would be very few consecutive zeros in the scan data.

The process of encoding a JPEG image can be seen on figure 3.1. Drawn onto the figure is also the embedding step, which will take place after the quantization process.

There are two ways to go about encoding the message in the quantization coefficients. We could either create a graph for each 8x8 matrix of coefficients, or we combine all of the non-zero AC-coefficients into an array and use these values to embed the message. The easy way is obviously to create a graph for each matrix, but this would result in having graphs with a maximum of 31 vertices (2 AC values per vertex), and more realistically around 10 vertices after the quantization process which will make a lot of the values 0. Having few vertices in the graph, means that

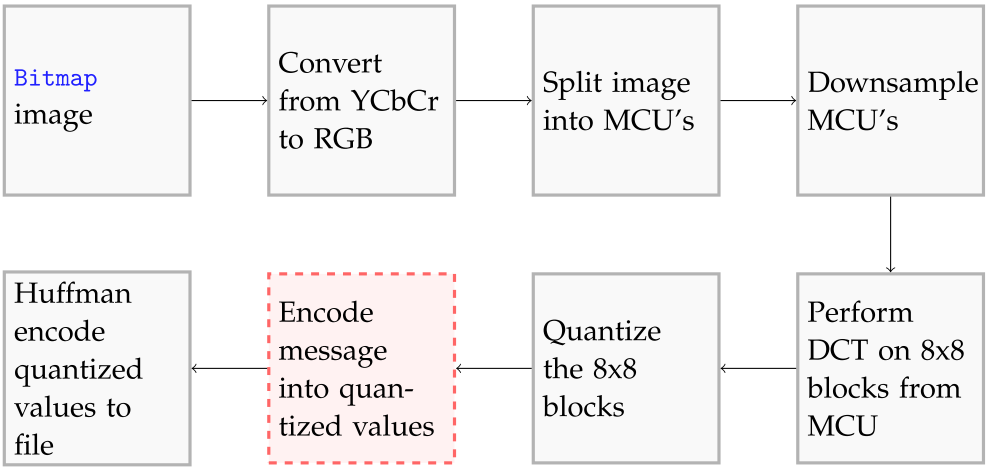

Bitmap image
Convert from YCbCr to RGB
Split image into MCU's
Downsample MCU's
Perform DCT on 8x8 blocks from MCU
Quantize the 8x8 blocks
Encode message into quantized values
Huffman encode quantized values to file

Figure 3.1: Process of encoding a JPEG image with embedding of data

there are not very many switches that would make the vertices *perfect pairs*, and we will have to forcibly alter more values to hide the message.

The more difficult way of encoding the data is to combine all valid (AC and non-zero) quantization coefficients into a list, and use this list for embedding the data. This means that we will not work on the individual 8x8 matrix, but instead on the values from all 8x8 matrices. This allows us to obtain a much larger graph, with more edges, and more possible switches of values in the vertices. A downside of using this method is that we cannot do the encoding of the message on-the-fly, as we will have to find all valid coefficients before we can start the encoding process.

Each method has its pros and cons, but ultimately, we have chosen to go with the latter, as the first method limits the graph in such a way that it becomes close to just using LSB, as there will be very few switches, and a lot of forced changes.

Now that we have chosen the dataset in which we will encode the data, we must find a way to get the data. Procedure 2 shows us how to obtain the values after the quantization process. The procedure itself is $\mathcal{O}(n)$, where n is the number of quantized matrices from the DCT step. With a sampling of 4:2:0 in the JPEG image we have 6 of these matrices in each 16x16 pixel MCU in the image. So the number of quantized matrices can be calculated as seen in equation 3.1 where W and H is the width and height of the image, respectively.

$$\left\lfloor \frac{W+15}{16} \right\rfloor \cdot \left\lfloor \frac{H+15}{16} \right\rfloor \cdot 6 \tag{3.1}$$

After finding the data in which our message is hidden, we can encode the actual message, as seen in procedure 3. After this procedure, we have a list of the valid coefficients with adjusted values, such that they contain the message. To save this to the file, we must first reverse the process of procedure 2 to replace the values in

Procedure 2 Finding valid entries from the quantized values

```
Input: Quantized 8x8 Matrices $Q = \{M_1, M_2, M_3 \dots M_n\}$ ,
Output: List $v$ with all coefficients available for hiding data in
  $v$ is an empty list
  for $i := 1$ to $i = n$ do
    for $j := 1$ to $j = 8$ do
      for $l := 1$ to $l = 8$ do
        if $\neg(j = 1 \wedge l = 1) \wedge ((M_i)_{jl} \neq 0)$ then
          Add $(M_i)_{jl}$ to $v$
        end if
      end for
    end for
  end for
  return $v$
```

the quantized values, so that when they are saved to a file, they contain the same values as we have calculated. This process would work for all data. However, there is still one problem when decoding the data. We do not know where to stop, and the decoder does not know the m value that was used while embedding the data. The m value was the value used for the $\oplus$ operation from section 2.9. Let us tackle the problem about length first. One often used technique is to have a certain marker to look for, and when that marker is encountered, the end of the data has been reached. This exact method is actually used in JPEG; when reading through the scan data, we keep on reading until we reach the EOF (End of File) marker. And when that marker is encountered, we have read all of the available scan data. But as we want our image to be able to contain arbitrary data, we do not want our implementation to depend on a sequence of bits or bytes that could be randomly found in the data we are embedding. So instead, we have to make sure that the decoder knows how much data to read before it starts to actually read it.

That is why we have decided to use the first 2 bytes of hidden data to contain information about the rest of the data. These bytes must also contain information about which m value is used in the data embedding. We have chosen to use 14 bits to indicate the length of the message (in bytes) and the last 2 bits to indicate what m value was used (00 = 2, 01 = 4, 10 = 16). This leads to an obvious problem: how do we know how to decode the first two bytes, if the information about the m value is hidden in these bytes? To overcome this problem, we have decided to always embed the first two bytes of information using $m = 4$. This means that a minor change to procedure 3 is needed, such that the first 16 bytes of information are embedded differently from the rest.

Procedure 3 Encoding data into the image data

```
Input: List $v = v_1, v_2, v_3, \ldots, v_k$ with all valid coefficients from the DCT process,
  message $e = \{e_1, e_2, e_3, \ldots, e_n\}$ as a bit-string and a $m$ value
Output: $v$ contains message $e$
  $G = (V, E)$ is an empty graph
  $i := 1$
  $verticesNeeded := \frac{n}{\log_2 m}$
  while $i < verticesNeeded$ do
    Make vertex from $v_i$ and $v_{i+1}$ and add to $G$
    i := i + 2
  end while
  for $i := 1$ to $i = verticesNeeded$ do
    for $j := 1$ to $j = verticesNeeded$ do
      if $i = j$ then
        continue
      end if
      if a switch between $V_i$ and $V_j$ is possible such that $(V_{i,1} + V_{i,2}) \mod m =$
      next $\log_2 m$ bits from e and $(V_{j,1} + V_{j,2}) \mod m =$ next $\log_2 m$ bits from e
      then
        Add edge between $V_i$ and $V_j$ to $G$.
      end if
    end for
  end for
  sort edges in $G$ by weight and remove all edges with large weight
  while G has any edges do
    Switch the values in the vertices connected by the first edge in the graph
    Remove all edges touching the two vertices from the graph
  end while
  for $i := 1$ to $i = verticesNeeded$ do
    if $(V_{i,1} + V_{i,2}) \mod m \neq$ next $\log_2 m$ bits from e then
      Change values in $V_i$ such that they match the bits
    end if
  end for
  merge $v$ with the values from the vertices in $G$
  return $v$
```

3.2 Design of the JPEG Image Encoder

In section 2.5 we examined how a JPEG image is encoded and saved to a file. In this section we will use this knowledge to design a JPEG encoder that can be implemented into our program. By making the encoder, we hope that we can create a valid JFIF file where we have also implemented the graph-theoretic approach as described in section 3.1. As we can recall from section 2.5, the conversion of a bitmap image to a JPEG image is the result of the following process, as shown in figure 3.1:

1. The image is split into three channels (Y, Cb and Cr).
2. The image is divided into MCU's where a down-sampling may take place.
3. DCT is performed on all 8x8 blocks in the MCU's.
4. Each new 8x8 block of DCT coefficients is quantized.
5. All 8x8 quantized values are then Huffman-coded and saved to a file using run-length encoding.

Some of these processes such as DCT are simply mathematical formulas performed on the data, and will not change from one image to another. However, other processes such as Huffman coding and quantization can yield very different results, depending on what Huffman- or quantization tables are used. If a quantization table with low values is used, the quality of the image will be higher, as the quantization process will not make as many of the entries in the 8x8 block zeros. A downside of quantization values being low is that because of the low amount of zeros in the 8x8 blocks, the run-length encoding will not be as efficient, which will result in a larger file-size.

Because the Huffman- and quantization tables have such a tremendous impact on the resulting JPEG image, it makes sense to have the user able to change these to their liking. Because of this, both quantization tables and the Huffman tables will be created as classes of their own, and because of this, changing the tables in the encoder will become as easy as changing a property.

Furthermore, a Huffman table is a collection of run-sizes and a bitstring of how to encode the run-size. These bitstrings can be saved in an unsigned data type, but because these bitstrings can start with the bit 0, there is no way to tell the length from the unsigned data type alone. Because of this, a class `HuffmanElement` was created. The class contains the unsigned data type along with the length of the bitstring.

What we have is a Huffman table where we want to look up a byte (the runsize) and get the corresponding bit encoding of the byte. To do this, the class must be implemented using a dictionary, where you can use the runsize as the key, and get the corresponding bitstring back. That way we can very efficiently translate from bytes to bit-encoding in the file.

To make the programme easier to expand at a later time, the JPEG image class will implement the interface `IImageEncoder`. Using an interface makes it much easier to switch the implementation later, or in our case, the encoding method. If someone wanted to encode a PNG image instead of a JPEG image, they would just switch the `JPEGImage` instance to for example a `PNGImage`. Implementing the interface means that the `JPEGImage` class will have the following public methods:

- `public void Save(string path)` which saves the image to a file at the given path.
- `public void Encode(byte[] message)` which embeds a message into the image.
- `public int GetCapacity()` which calculates how much information can be stored in the image.

Another reason for using the `IImageEncoder` interface is that we can keep working on the actual implementation of the JPEG-encoder, while the ITC-group can build their system around the interface. The interface is very sparse, but offers the basic functionality which should be present in all types of image-encoders.

3.3 Representing the Graph

There are multiple ways of representing a graph. To start with, we wanted to use an adjacency list and for each vertex, keep a list of its neighbours. Having the edges implied by the adjacency list instead of having actual edge objects would not only cause the programme to use less memory, it would also make the process of removing all neighbours of a vertex more efficient. This is a process we must often do in the graph-theoretic method described in section 2.9 and 3.1. As efficient as this would be, there are major problems with doing this. By implying the edges instead of storing them, we cannot save any additional information about what the edge actually describes. In our case, an edge means that there is a switch that would make both vertices contain the given message if the modulo operation is applied with the correct m-value. But having each vertex contain two values means that there are four possible switches between two vertices as shown in figure 3.2.

Another way to show the representation of a graph, is simply a list of vertices and a list of edges. Each edge then simply stores what vertices it connects. This makes it very easy to loop through all edges, but at the same time makes it computationally expensive to remove certain edges from the graph (i.e. to remove all edges between two vertices).

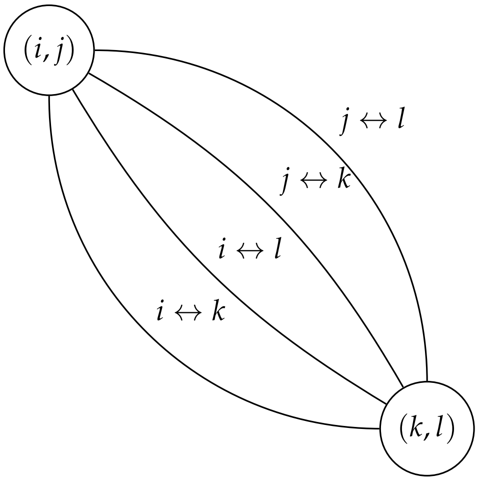


Figure 3.2: Two vertices in the graph can be connected 4 different edges

There are also other types of adjacency lists where each vertex keeps a list of the edges connecting them to other vertices. Here we get the advantages of having an actual `Edge` object, while still being able to quickly find all neighbours of a vertex. We have chosen to use the method of simply keeping a list of all vertices, instead of each vertex keeping a list of its neighbours. One of the main reasons for this, is that we need to be able to sort the edges, so that we can choose the edge with the lowest weight. If we did not have a list of the edges, we would have to search through every single edge in every single vertex to find the edge with the lowest weight and then after choosing that edge, we would have to search again.

3.4 Decoding

As an encoded message is of no use if it cannot be retrieved, this section will describe in detail how the decoder is created.

3.4.1 Find the Huffman Tables from a JPEG file

To find the encoded text using the graph-theoretic method, the first step is to read the Huffman tables from the JPEG-file, making it possible to look up the code written in the scan-data segment of the file. As mentioned in section 2.5, the Huffman tables are placed in the beginning of a JPEG-file, prior to the scan-data segment.

The first step is to look for the DHT marker 0xFFC4, since what follows is the information about the Huffman table. To read the information we need, and store it in a table, we came up with algorithm 4 based on an article (Nazar, 2013).

Procedure 4 Find Huffman table in a jpeg file

```
Input: bitstring, which is reading a JPEG file, and an output parameter so the
  caller knows the table's ID and Class
Output: A Huffman Table fully constructed from the bytes in the JPEG file
  $insideHuffmanTable := false$
  array of lengths $AoL$
  elements of current length $e$
  current code length $cL := 0$
  while $insideHuffmanTable = false$ do
    $marker$ = read byte from stream
    if $marker = 0xFF$ then
      $marker$ = read byte from stream
      if $marker = 0xC4$ then
        $insideHuffmanTable := true$
      end if
    end if
  end while
  $length :=$ read byte from stream $<< 8+$ read byte from stream $-19$
  $ClassAndID :=$ read byte from stream
  for $i := 1$ to $i = 16$ do
    $AoL_i :=$ read byte from stream
  end for
  $e := AoL_1$
  $code := 0$
  $cL := 1$
  for $i := 1$ to $i = length$ do
    while $e = 0$ and $cL < 16$ do
      $code * 2$
      $cL + 1$
      $e := AoL_{cL}$
    end while
    Create new Huffman element from the next byte in the stream, with the same
    code as $code$ and with the length $cL$
    Add Huffman element to table
    $e - 1$
    $code + 1$
  end for
  return $HuffmanTable$
```

3.4.2 Decoding the Huffman-encoded Values

As explained in section 3.1, the hidden message will be hidden in pairs of two quantized values using the $\oplus_m$ operator. To get the data out, we have to read the first 16 non-zero AC values, in order to find what m-value the message is hidden with, and the length of the message. We can then use this length and m-value to determine how many non-zero AC values to read, and decode the original message. Finding the length of the message, as well as the value of m, requires you to decode the scan data as follows:

1. Read 1 Huffman DC element for the Y channel
2. Read 63 Huffman AC elements for the Y channel, or until EOB
3. Repeat steps 1 and 2, three more times
4. Read 1 Huffman DC element for the Cb channel
5. Read 63 Huffman AC elements for the Cb channel, or until EOB
6. Read 1 Huffman DC element for the Cr channel
7. Read 63 Huffman AC elements for the Cr channel, or until EOB
8. Repeat steps 1 through 7 until 16 non-zero AC values are found

Reading the Huffman-encoded values in accordance to the table shown in section 2.5.6 is not completely trivial. The values with 1_2 as their most significant bit, are positive numbers, and can be interpreted as an unsigned data type.

To get the negative values, we have to do some calculations. First, we notice how with each category the minimum number is the same as adding one to the 2^n negative value, and the higher the bit-value becomes, the higher the actual value becomes as well. Based on this information, we have derived algorithm 5 to calculate the number from a category and a bit-string. These values, and the preceding zeros, will have to be read into an array in zig-zag ordering.

Procedure 5 Decode the huffman-encoded value

```
Input: A bitstring $s$ of 15 bits or less, and a category $n$
Output: A decoded huffman value
  $returnValue := 0$
  if most-significant bit of $s = 1$ then
    return $s$
  end if
  $returnValue := s + 1 - 2^n$
  return $returnValue$
```

3.4.3 Decoding the Message

We will need 16 non-zero values in order to find the length of the encoded message and the m-value, which we will need to use in order to decode the message itself. To find the length from the first 16 values, we combine the first 14 values into pairs of two. For each of those pairs, we then use the $\oplus_4$ operator, and combine the result from the 7 pairs into a bit-string with length 14 because each result can be represented by two bits. This bit-string can then be converted to a number which is the length of the message.

The m-value is hidden within the remaining values. We use the $\oplus_4$ operator again, and the result describes what m-value should be used for decoding the message itself. There are four different combinations of those two bits. Only three of them are valid: 00_2 means that the m-value is 2, 01_2 means the m-value is 4 and lastly 10_2 means that it is 16.

Once the length has been found, we can keep decoding the scan data segment until enough AC coefficients have been read to decode the full message, using the modulo value we just found.

When enough values have been read, the decoding process of the message is the same as listed in the beginning of this subsection, with the only difference being that the value of modulo may be different. Each byte will then have to be written to an array and returned to the user. This array will then contain the message that was encoded in the JPEG image.

3.5 User Interface

Even though we had an agreement, that the ITC group was to create an Android application implementing our programme, our project still had to be able stand on its own. We therefore decided to create a user interface displaying the programme's capabilities.

The user interface (UI) for Stegosaurus is created in Visual Studio using C# and WinForms. In one of our previous experiments, we had created a very basic UI. The only functionality of that UI was to show the cover image, the image to be hidden in the cover and the final output.

Something the experiment lacked was a choice of encoding. For example, whether they would like to use the least significant bit method or our graph theoretical method. Therefore, the final design should include several different options, so as to customise the encoding of their image to what suits their requirements.

Another thing lacking in previous designs has been a guide or explanation throughout the programme and description of its joint functionality. Previous versions have been less complex and perhaps did not require much explanation. However, now as the programme will have a larger set of settings, we believe that

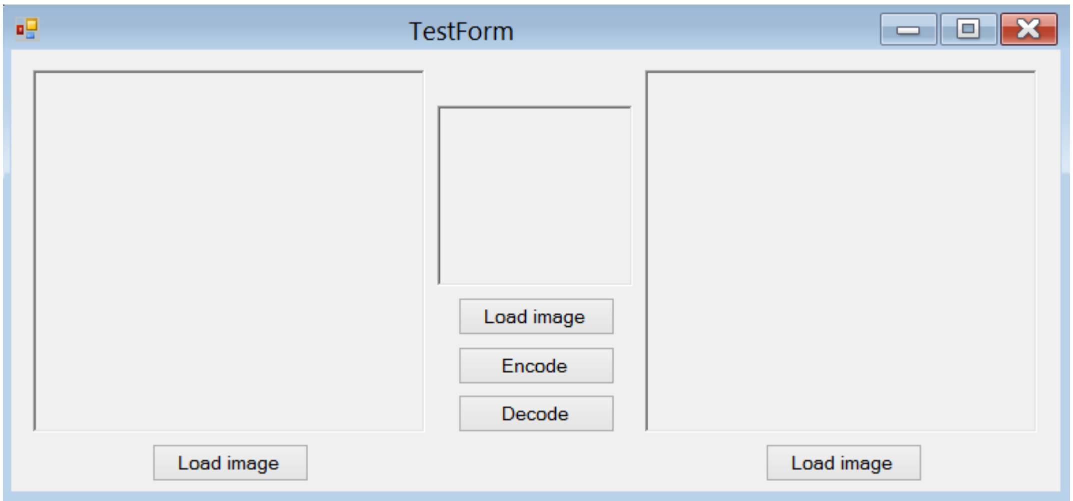


Figure 3.3: This is the form for our LSB experiment, which acted as inspiration to our final design.

it should be a requirement to have a short explanation of these, what impact they will have on the end result and what the programme will do in general.

All of these things should illustrate to the user how the programme works and what it is doing while running.

3.5.1 Design

The form as shown in figure 3.3 is what our UI looked like for our LSB experiment. We are taking the main ideas from this, where we show the user the images or files they have selected and the final product. This should ensure them, that the correct files have been selected and the encoding process has succeeded. We decided, however, that it was confusing to have three separate load buttons. As shown in figure 3.3 above, the cover image is uploaded in the left side, the message in the middle and the encoded image on the right. When you wanted to decode the image, the encoded image had to be uploaded using the load-button to the right. This led to a confusing user experience. Because of this we have decided to only have two buttons, one for loading an image that is to be either encoded or decoded and another for only getting the hidden message.

We have furthermore decided to include extra features to our overall design making use of the functionality of our programme's encoder and decoder. These features include giving the user options to use their own Huffman and quantization tables to control the encoding process with precision.

To do this, we decided that creating an options form would be ideal when it comes to guiding the user through this extended functionality. The options form would consist of a choice of encoding method, a quality-setting for use when encoding using our graph-theoretic method, customisation of Huffman and quantization tables. However, there is also an option to reset all settings to default, if the user in question has no specific preferences or made an error.

After their message has been encoded, there is, naturally, a need to be able to decode it after the encoding process. This means that the programme should be able to present these options to the user as openly as possible, as these are the key features of the programme.

As something more general, we have created a basic help guide to explain to the user how the programme works, and a short description of what each feature entails. Within this menu item, there will also be an *About* window that gives a short description of the programme, who it is by and under which circumstances it has been made.

Something that is absolutely fundamental to the programme, is showing the user the images they provide and the image they produce. This visual representation gives the user reassurance that they have chosen the correct files.

Likewise, another feature, that we deem to be a rather important visual representation is the quality of encoding. Therefore, we have decided to have a slider to show what level of quality the programme will encode with.

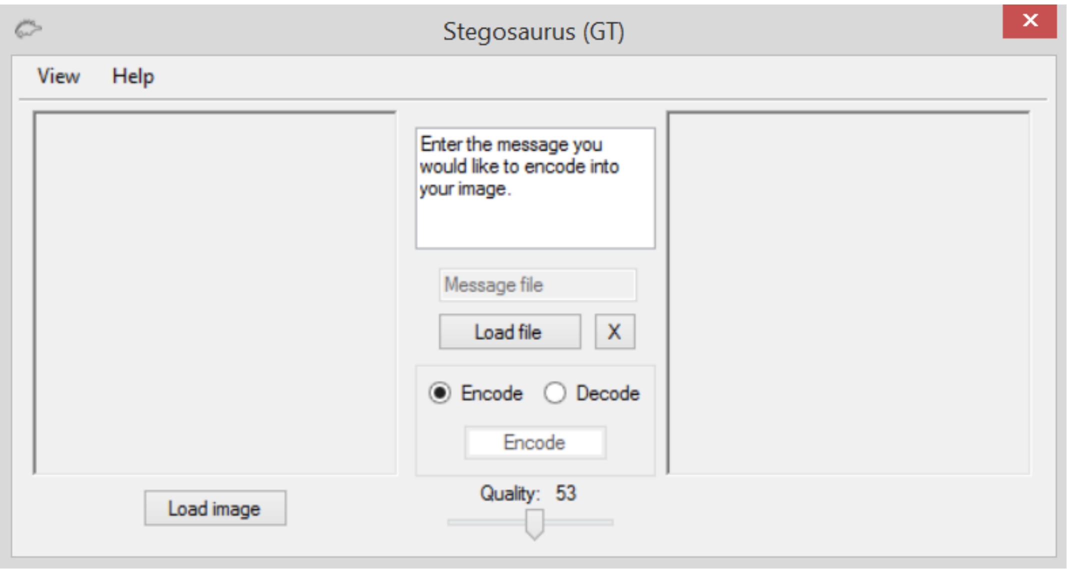
Saving of all these custom settings of course needed to be a function of the programme as they can be quite time-consuming to set and this is now done as soon as the programme is closed.

The following list is what the user interface will include:

Main form We wanted to make the main form as simple as possible. Therefore we have a single button for loading images. Additionally, there is also a single button for both encoding and decoding, this option can be selected by using the radio buttons. Also there is an option for encoding text, or simply encode a file.

LSB or Graph Theory Method The user will be able to switch between the two different methods in the options form. This is an improvement over our initial design that utilised two separate tabs on the main form, one for encoding or decoding using the LSB method and one for the GT method. This resulted in a simpler interface with almost identical functionality being less split across the programme. This improvement is made possible by the OOP design of the encoders and decoders, having the various ways of encoding/decoding available from the same interface.

Quality setting A slider shows the user what quality setting currently used when encoding JPEG images. This slider was initially planned to only be a part of the main form, but has since been moved to options as well as the main form for the graph theoretical method, as we decided it was as much a custom option as the Huffman and quantization tables.



Enter the message you would like to encode into your image.

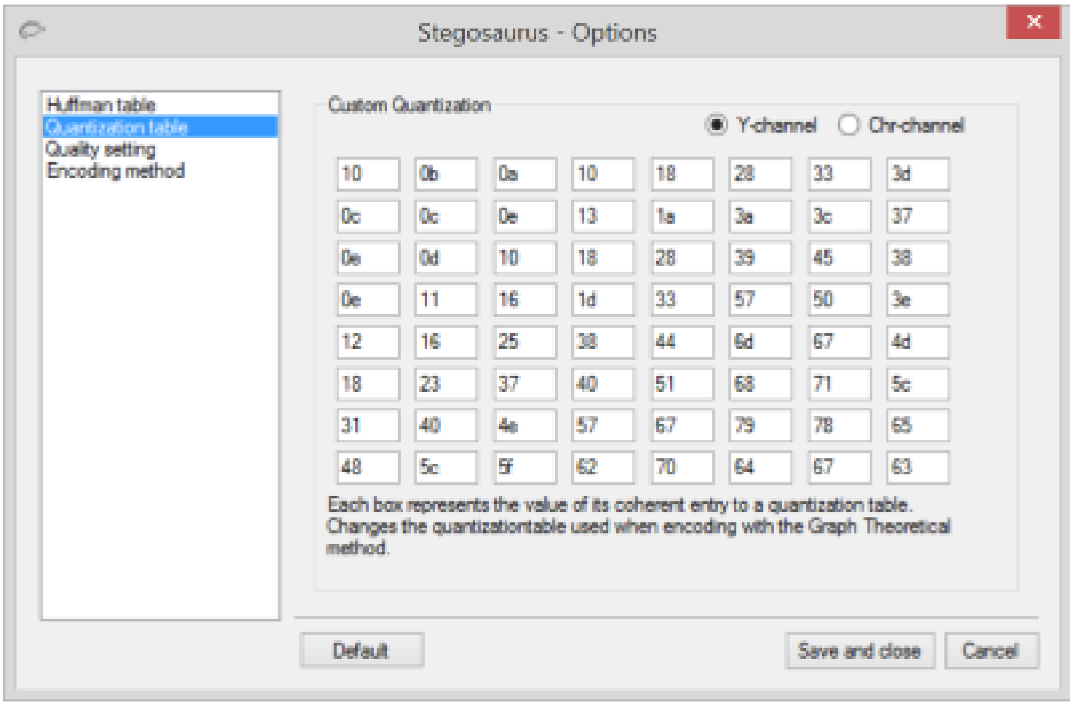
● Encode ○ Decode

Quality: 53

Figure 3.4: This is how the user will meet our graph theoretical method and LSB, where the message to be encoded no longer has to be an image, but can be a text message written directly in the programme or any sort of file freely selected

Quantization Tables A difficulty that arose when deciding on these features was how we were going to make it possible for a user to input their custom quantization and Huffman codes.

An initial idea included a CSV-file (Comma Separated Values file), but we decided that for quantization tables, it was ideal and no trouble to have the user customise their tables by typing them in text boxes, as the size of each quantization table is always 64 for both the luminance and chromium channels. This can be seen in figure 3.5.

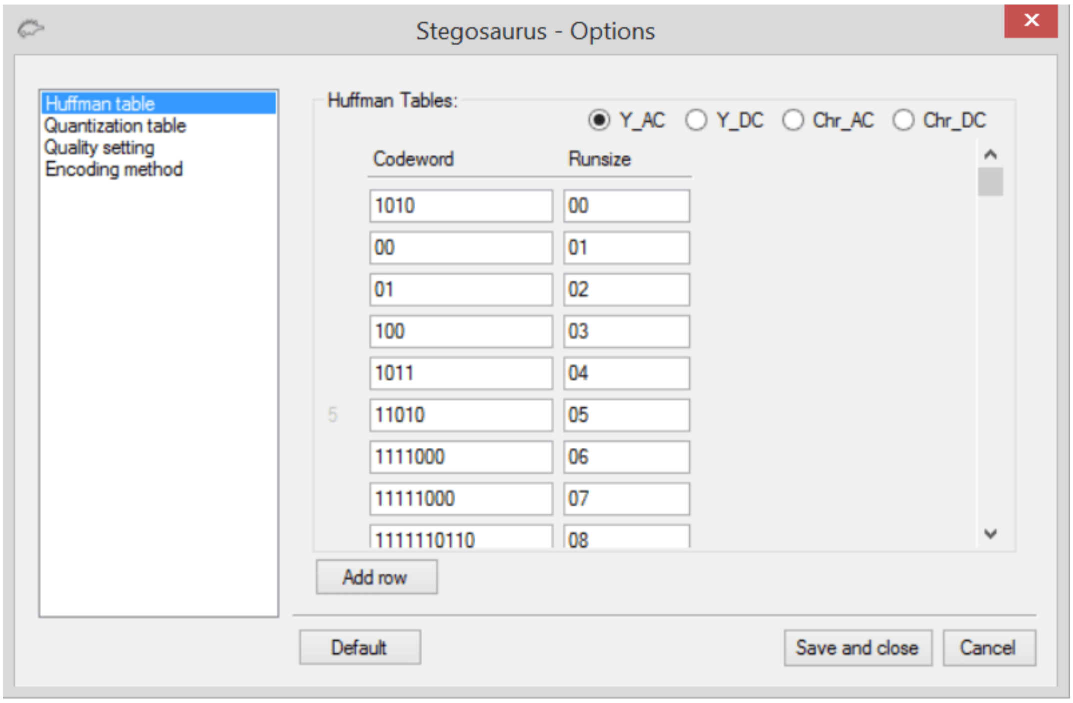


10	0b	0a	10	18	28	33	3d
0c	0c	0e	13	1a	3a	3c	37
0e	0d	10	18	28	39	45	38
0e	11	16	1d	33	57	50	3e
12	16	25	38	44	6d	67	4d
18	23	37	40	51	68	71	5c
31	40	4e	57	67	79	78	65
48	5c	5f	62	70	64	67	63

Each box represents the value of its coherent entry to a quantization table. Changes the quantizationtable used when encoding with the Graph Theoretical method.

Figure 3.5: This is how the custom quantization table is shown to the user in the options setting form.

Huffman Tables Huffman tables requires a similar design to quantization tables. There are four radio buttons to switch between the four different channels. Text boxes have been set up to be able to take in a string from the user. However, as the sizes of the Huffman tables vary, it is necessary to have a button that allows the user to add rows as needed. Huffman table customisation can be seen in figure 3.6.



Stegosaurus - Options

- Huffman table
- Quantization table
- Quality setting
- Encoding method

Huffman Tables: ● Y_AC ○ Y_DC ○ Chr_AC ○ Chr_DC

Codeword	Runsize
1010	00
00	01
01	02
100	03
1011	04
11010	05
1111000	06
11111000	07
1111110110	08

Figure 3.6: The picture above shows how the user will see the Huffman table options setting.

Help form A very general description of what the programme does and descriptions of the different settings in the options form. We believe this to be important to the overall understanding of how the programme is to be used; something that our previous experiments have not had.

There were some things with the UI that has been seen as problematic. Although the programme did tell the user that an error occurred when they tried to save an illegal set of settings, it did not save any of the settings but reverted to the previous set. That meant that the user had to reconfigure everything from even just one single mistake.

This cumbersome way of handling this was caused by a poor, overall structure, where the main form was handling the actual saving of the settings, not the options-form. This far from ideal structure meant that it was required to send variable values back and form between the two forms. This has been improved on and the options-form now handles everything that has to do with settings, including saving.

This new interface has advantages over previous interfaces in our experiments throughout our process, but there are still some features that could be included, not to improve the programme, but the overall user experience.

Chapter 4

Implementation

4.1 Implementation of a JPEG Image

Looking back to section 3.2 we can start implementing this into our steganography programme *Stegosaurus*. We can also refer to figure 3.1 for the steps the encoder must take.

There are multiple constructors for the `JPEGImage` class, as a user can choose to specify custom quantization or Huffman tables, or simply use the default tables which both the `QuantizationTable` and `HuffmanTable` classes provide. This is done by not providing any tables in the constructor. The default tables in the respective classes are both found in the JPEG specification (Union, 1992, Annex k).

The `Encode(byte[])` is where the actual JPEG encoding and message embedding takes place. The method can be seen in listing 4.1. The most significant part of the JPEG file is the scan data, but before it can be written to the file, we have to encode most of the headers first. We do that by calling `_writeHeaders()`. All information needed in these headers is already known, as both the bitmap image and tables provided in the constructor.

```
public void Encode(byte[] message) {
  _jw = new JpegWriter();

  const int fourteenBitMax = 16884;
  int capacity = GetCapacity();

  if (message.Length > fourteenBitMax) {
    throw new ImageCannotContainDataException(message.Length,
       ↪ fourteenBitMax);
  } else if (message.Length > capacity) {
    throw new ImageCannotContainDataException(message.Length,
       ↪ capacity);
  }

  _breakDownMessage(message);

  _writeHeaders();
  _writeScanData();
  _writeEndOfImage();
}
```

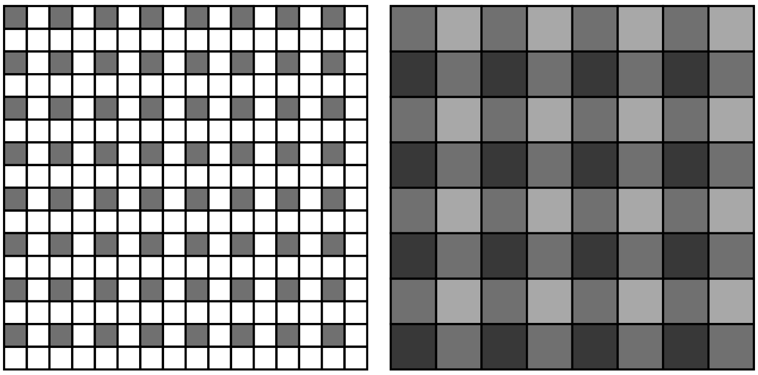
Listing 4.1: JPEGImage.Encode method **File:** JPEGImage.cs

From here on, we need to perform all of the regular steps from JPEG encoding up until the Huffman encoding. We start by converting the Bitmap's RGB channels into Y, Cb and Cr channels. Then in the method `_encodeAndQuantizeValues(sbyte[][,], int, int)` we have the loop which can be seen in listing 4.2. While these nested loops may seem very computationally heavy, what they actually do is that they loop through the three channels of every pixel in the image. So while we have five nested for-loops, we run through every pixel 3 times, and split the image into 16x16 MCUs. After every 16x16 MCU has been filled with values, the MCU must be encoded. As the name Minimum Encoded Unit implies, we can encode one MCU without depending on the other. That means that after we fill one MCU, we can encode that directly. We do that by invoking the method `_encodeBlocks(float[][,])`.

```
for (int MCUY = 0; MCUY < imageHeight; MCUY += 16) {
  for (int MCUX = 0; MCUX < imageWidth; MCUX += 16) {
    for (int i = 0; i < 3; i++) {
      for (int x = 0; x < 16; x++) {
        for (int y = 0; y < 16; y++) {
          channels[i][x, y] = YCbCrChannels[i][MCUX + x, MCUY
             ↪ + y];
        }
      }
    }
    _encodeBlocks(channels);
  }
}
```

Listing 4.2: Shows how the image is divided into MCUs **File:** JPEGImage.cs

In `_encodeBlocks(float[][,])`, we split the MCU into 8x8 blocks. As we have a fixed sampling of 4:2:0, we split the Y channel into four 8x8 blocks and effectively save all information about the luminance channel. The chroma channels are downsampled from 16x16 blocks to 8x8 blocks. As we implemented this, we had two ways of doing so. One way is taking every fourth value in the block as seen on figure 4.1a, and the other is taking an average of each 2x2 block in the 16x16 MCU



(a) Downsampling by choosing only every fourth value

(b) Downsampling by taking an average of every four values

Figure 4.1: Different methods for downsampling of 16x16 MCU to an 8x8 block

as seen on figure 4.1b.

In our implementation we chose to use the average. While it requires more calculations, we do get better image quality. The problem with using just one value for every 2x2 block, is that if the image contains sharp edges, those lines will become distorted. If using the average instead, the lines will, at most, become a bit blurry.

After the downsampling process, only the DCT and quantization processes are left before we can start embedding the message. On each 8x8 block we perform DCT as seen in listing 4.3. The DCT formula was described in section 2.5, but one thing we realised about the formula was, that we could calculate the cosines only once, and use these instead of repeatedly calculating the trigonometric function. Therefore we calculate these results before even beginning the DCT process, and then just reuse the results in the DCT method.

```
private static float[,] _discreteCosineTransform(float[,] block8) {
  float[,] cosineValues = new float[8, 8];

  for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 8; j++) {
      float tempSum = 0.0f;
      float cCoefficient = _c(i, j);
      for (int x = 0; x < 8; x++) {
        for (int y = 0; y < 8; y++) {
          tempSum += cCoefficient * block8[x, y] * CosinesCoefficients[x, i] * CosinesCoefficients[y, j];
        }
        }
        cosineValues[i, j] = tempSum;
      }
    }
    return cosineValues;
}
```

Listing 4.3: Multidimensional DCT on 8x8 block **File:** JPEGImage.cs

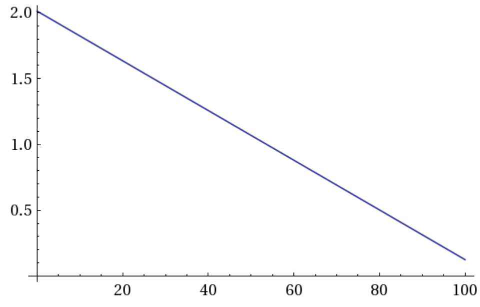
After calculating the DCT coefficients, it is simply a matter of dividing each entry in the 8x8 block with the corresponding value in the quantization table.

We wanted our JPEG encoder to be able to have a quality setting, so that the user of the class can choose whether they want to focus on image size or image quality. To lower the filesize, and also the quality, we must divide by larger values in the quantisation step. The way we do this, is by scaling the quantization table based on an integer between 0 and 100 (the quality setting).

This scaling progress was made a part of the `QuantizationTable` class, as this is basic functionality of the quantization, and makes it able to be scaled very easily in the code.

We use the function $f(q) = \frac{100-q}{53} + 0.125$ to convert from a quality to a factor we can multiply the entries in the quantization table with. There is no standard function for choosing multiplication factors, so we just use a function that will scale nicely between the numbers 0 and 100. On figure 4.2 a plot of the function can be seen.

After the scaling process, we divide each entry in the DC coefficients by the corresponding entry in the quantization table. The results of this process are saved to the private field `_quantizedValues`.



plot (100 − x) / 53 + 0.125 from 0 to 100 | Computed by Wolfram|Alpha

Figure 4.2: Plot of $f(q) = \frac{100-q}{53} + 0.125$ that shows how the quality of the JPEG image impacts the quantization tables

4.1.1 Implementation of the Graph-theoretic Method

Now we are ready to perform the actual encoding of the data, before saving the image data to a file. We use procedure 2 to find the valid entries from our newly filled _quantizedValues. The first 16 values from our valid entries are added as vertices to the graph, with an m-value of 4. This is because the length and m-value is encoded in the first 8 vertices with a fixed m-value of 4. Then we keep on adding vertices, until we are certain that there are enough vertices to embed all the data in the byte[]. We use the Graph class to handle the vertices and edges. We simply add vertices for each pair of numbers in the valid entries, until we have enough vertices to contain our message byte[].

After adding the vertices, we use the private method _addEdge(bool, bool, ↪ Vertex, Vertex, int, Graph) in the JPEGImage class to determine whether an switch between the two specified vertices would result in a perfect pair. If the switch would result in a perfect pair, an edge is added to the graph.

We run this method four times for every distinct pair of vertices in the graph, for each of the possible switches shown earlier in figure 3.2. In procedure 3, for each vertex, we would loop through all other vertices. If we had the vertices $\{a, b, c, d\}$ we would get the pairs $\{(a, b), (a, c), (a, d)(b, a), (b, c), (b, d), (c, a), (c, b), (c, d), (d, a), (d, b), (d, c)\}$. To find all possible switches, there is no need to check both (a, b) and (b, a), as those two vertices would have the same switches. In fact, we only need to check the switches $\{(a, b), (a, c), (a, d), (b, c), (b, d), (c, d)\}$, which is half of the original pairs. We implemented this as seen in listing 4.4. The inner loop starts at the vertex following the current vertex in the outer loop. This way, we do not find the reversed counterparts of already found pairs. This particular piece is still $\mathcal{O}(n^2)$, but instead of checking $n(n-1) = n^2 - n$ pairs, we only need to check $(n \cdot \frac{n}{2}) = \frac{n^2}{2}$ where n is the number of vertices.

```
const int threshold = 5;

List<Vertex> tbc = graph.Vertices.Where(x => (x.SampleValue1 +
    ↪ x.SampleValue2).Mod(x.Modulo) != x.Message).ToList();
int length = tbc.Count;

Parallel.For(0, length, i => {
  for (int j = i + 1; j < length; j++) {
    _addEdge(true, true, tbc[i], tbc[j], threshold, graph);
    _addEdge(true, false, tbc[i], tbc[j], threshold, graph);
    _addEdge(false, true, tbc[i], tbc[j], threshold, graph);
    _addEdge(false, false, tbc[i], tbc[j], threshold, graph);
  }
});
```

Listing 4.4: Checks all edges between distincive pairs of vertices **File:** JPEGImage.cs

After adding the edges, we need to determine what edges to keep. We do that by sorting our edges by weight and then choose the edge with the lowest weight. We then remove all other edges connected to the two vertices connected by the chosen edge, and then repeat the process. All of this is implemented in the `Graph` class as seen in listing 4.5.

```
public List<Edge> GetSwitches() {
  Edges.Sort();
  List<Edge> chosenEdges = new List<Edge>();

  while (Edges.Any()) {
    chosenEdges.Add(Edges[0]);
    _removeEdge(Edges, Edges[0]);
  }
  return chosenEdges;
}

private static void _removeEdge(List<Edge> list, Edge e) {
  list.RemoveAll(x => x.VStart == e.VStart || x.VStart ==
      ↪ e.VEnd || x.VEnd == e.VStart || x.VEnd == e.VEnd);
}
```

Listing 4.5: Implementation of the greedy algorithm for choosing switches **File:** Graph.cs

After running the `GetSwitches()` method on the graph, it is simply a matter of switching the values in the vertices appointed by the chosen edges, and then running through all vertices in the graph, and for every vertex not containing the right message, forcibly change the sample values so that the right message is contained.

4.1.2 Encoding the Data to a File

The last thing that remains before saving the image to the file, is the Huffman encoding. We start out by encoding the DC as seen in listing 4.6. This is done by finding the difference from the last DC entry in the current MCU, then finding the Huffman code from the DC Huffman table, before saving the bits to the image.

```
short diff = (short)(block8[0, 0] - _lastDc[DCIndex]);
_lastDc[DCIndex] += diff;

if (diff != 0) {
  byte category = _bitCost(diff);
  HuffmanElement huffmanCode =
      ↪ huffmanDC.GetElementFromRunSize(0, category);
  _ushortToBits(bits, huffmanCode.CodeWord,
      ↪ huffmanCode.Length);

  _ushortToBits(bits, _numberEncoder(diff), category);
} else {
  HuffmanElement EOB = huffmanDC.GetElementFromRunSize(0x00,
      ↪ 0x00);
  _ushortToBits(bits, EOB.CodeWord, EOB.Length);
}
```

Listing 4.6: Encoding DC coefficient into the image **File:** JPEGImage.cs

The next part of the Huffman encoding is to run through the 63 AC values in zigzag. If we find a zero, we count that, but otherwise keep on going as that zero must be encoded as EOB, ZRL or as part of the next non-zero value. If we have more than 16 zeroes before the next non-zero value we encode the ZRL, and if the rest of the block consists of only zeroes, we encode the EOB. This can all be seen in listing 4.7

```
int zeroesCounter = 0;
for (int i = 1; i < 64; i++) {
  int x = QuantizationTable.RoadPoints[i, 0], y =
      ↪ QuantizationTable.RoadPoints[i, 1];
  if (block8[x, y] == 0) {
    zeroesCounter++;
    continue;
  }
  while (zeroesCounter >= 16) {
    HuffmanElement ZRL = huffmanAC.GetElementFromRunSize(0x0F,
        ↪ 0x00);
    _ushortToBits(bits, ZRL.CodeWord, ZRL.Length);
    zeroesCounter -= 16;
  }

  byte cost = _bitCost(Math.Abs(block8[x, y]));
  HuffmanElement codeElement =
      ↪ huffmanAC.GetElementFromRunSize((byte)zeroesCounter,
      ↪ cost);
  zeroesCounter = 0;
  _ushortToBits(bits, codeElement.CodeWord,
      ↪ codeElement.Length);

  _ushortToBits(bits, _numberEncoder(block8[x, y]), cost);
}

if (zeroesCounter != 0) { //EOB
  HuffmanElement EOB = huffmanAC.GetElementFromRunSize(0x00,
      ↪ 0x00);
  _ushortToBits(bits, EOB.CodeWord, EOB.Length);
}
```

Listing 4.7: Encoding AC coefficients into the image **File:** JPEGImage.cs

This process is repeated for each MCU in the image.

After the Huffman encoding, it is simply a matter of adding the EOI marker to the end of the image, and saving the bytes to a file.

4.1.3 Working with Bits

While working with JPEG images we found that bit-patterns and bitwise operations often occur. Due to hardware limitations, a single bit in the memory cannot be changed individually, as we can only address the individual bytes. There are two ways that we can go about this: Either we use bytes to represent bits, and effectively waste seven bits of memory for each bit, or we use the class `BitArray` from the `System.Collections` library. What this class does, is that it handles the packing of the bits into bytes, and makes it possible to address the individual bits.

The `BitArray` seemed like the way to go, and we therefore implemented a class `BitList` which offered `List`-like features such as `Add` and `Insert` while using the `BitArray` to store the data. The implementation can be seen in appendix B.

The `BitArray` class offers the functionality of modifying the length of the array, so that more values can be added. This means that every time the `BitList` runs out of space in the underlying `BitArray`, we simply multiply the length of the array by 2, so that more values can be added. This makes appending to the array relatively fast, since we have enough room to add more values most of the time. Inserting values in the middle of the array, however, proved to be much more difficult.

When inserting a value into the array, all values after the value to be inserted have to be shifted, so that room is made from the new value. In an example 512x512 image, we have over 750,000 bits to save, and if we were to insert a bit in the beginning of the array, we would have to move 750,000 elements in the array. All in all a very computationally expensive operation.

So while the `BitArray` seemed promising in theory, saving us a lot of memory, the need for insertion of bits into the middle of the array makes using the `BitArray` not feasible. What we would need to solve this problem is to eliminate the need to shift a large part of the array. One way of solving this problem is to make a data type which combines the functionality of the `BitArray` with a linked list. By splitting up the large `BitArray` into smaller linked arrays we could shift the values in the smaller arrays. We would have a slightly longer access time, but have a much faster insertion time. Another way to solve the problem, and what we ended up doing, was to eliminate the need for insertion of the values. We realised that the only time we need to insert values into the `BitList` was when there were eight consecutive ones in the scan data. So what we did was to create the method

CheckedAdd which we would use while writing scan data to the list. This method would then keep track of the last inserted values, and insert the zeros on-the-fly instead of after the whole process. The method CheckedAdd can also be seen in appendix B.

4.2 Decoding

The first thing we do in the decoding process is to read the Huffman tables from the JPEG file.

To do that, we implemented procedure 4, which gave us all the information we needed about the Huffman table, including what kind it was, so that we could store it in the appropriate property.

The Huffman tables are read from a stream from a given filepath. This stream gets assigned to a BinaryReader, because we are reading bytes. All of this can be seen in the constructor in listing 4.8.

```
public JPEGDecoder(string path) {
  StreamReader sr = new StreamReader(path);
  file = new BinaryReader(sr.BaseStream);
  for (int i = 0; i < 4; i++) {
    byte ClassAndID = 0;
    HuffmanTable temp = getHuffmanTable(out ClassAndID);
    if ((byte)(ClassAndID & 0xf0) == 0) {
      if ((byte)(ClassAndID & 0x0f) == 0) {
        YDCHuffman = temp;
      } else {
        ChrDCHuffman = temp;
      }
    } else if ((byte)(ClassAndID & 0x0f) == 0) {
      YACHuffman = temp;
    } else {
      ChrACHuffman = temp;
    }
  }
}
```

Listing 4.8: The Decoder constructor **File:** JPEGDecoder.cs

The listing also shows how we find the HuffmanTable and store it in the correct property, based on its ID and class.

Once we have read all the lengths of the Huffman encoded values, we read the runsize of the Huffman codes. To get the correct code, we bitshift the current value one to the left. Once there are no more Huffman codes of the current length to read, we increment the current length by one, as described in procedure 4.

If the public method called `Decode()` is called, the method will decode the message hidden in the JPEG image using the graph-theoretic method. `findScanData()` returns a bitlist, which will be used by `decodeScanData()`, and then returns the message.

The method will read through the input until meeting the SOS marker, then skip the next 12 bytes which contain information about the scan data segment.

Once this is done, the method will add every single byte in the scan data segment to a `List<byte>` until a marker is found. The implemented method can be seen in Listing 4.9.

```
private BitList findScanData() {
  byte a;
  findMarker(0xda);
  for (int i = 0; i < 12; i++) {
    file.ReadByte();
  }

  List<byte> scanData = new List<byte>();
  int length = (int)file.BaseStream.Length;

  while (file.BaseStream.Position < length) {
    a = file.ReadByte();

    if (a == 0xff) {
      byte b = file.ReadByte();
      if (b != 0) {
        break;
      }
    }
    scanData.Add(a);
  }

  BitList bits = new BitList();
  foreach (byte current in scanData) {
    byte mask = 1;
    for (int i = 0; i < 8; i++) {
      bits.Add((current & (mask << (7 - i))) >> (7 - i));
    }
  }
  return bits;
}
```

Listing 4.9: Read in the ScanData section into a `BitList` **File:** JPEGDecoder.cs

Since the decoding process works on a bit-level, we will need to convert this list of bytes into an array of bits instead. To do this, we have to get the `BitList` class, which suits our needs. This is shown in lines 132-138.

This list will then contain all the bits in the scan data segment of the file.

Because we use the `BitList` class we can create bitstrings of variable lengths, by reading the bits one at a time. After finding a bitstring representing the codeword from the corresponding Huffman table, we do a reverse lookup of the codeword to get the original runsize. The runsize tells us how many zeros precedes the current value and the category, which tells us how many bits we need to read to find the value. After determining the bitstring we do the calculation from procedure 5 to find the actual value.

By repeating this process we can reassemble the original quantization coefficients.

Once we have quantization coefficients

We can keep decoding MCUs until we have enough MCUs, such that they contain 16 nonzero AC-values. These are the same 16 values in which we embedded the message length and the m-value used. We decode these 16 values by using an m-value of 4.

To get the length, we look at the first 14 values in pairs of two, and use the $\oplus_4$ operator on them.

This value will be added to a `ushort` value, which will be bitshifted right twice before adding the value. Doing this seven times, we will end up with the length of the message.

To get the m-value, we use the $\oplus_4$ operator on the 15th and 16th values. From the result we can determine which m-value had been used during embedding.

Once we have the length and the m-value, we can keep reading MCUs until the values containing the message have been found. The reason we do not just read all of scan data, is to save time, since there is no reason to decode all of it, when the message might only be hidden within the first few values.

When all the required values have been read and saved in the `List<int>` variable, the message is found according to Listing 4.10.

```
158 byte logOp = (byte)(Math.Log(modulo, 2));
159 byte steps = (byte)(8 / logOp);
160 int elementsToRead = length * steps * 2;
161
162 [...]
163
164 for (int i = 0; i + 1 < elementsToRead; i += 2) {
165   messageParts.Add((byte)(validNumbers[i] + validNumbers[i +
        ↪ 1]).Mod(modulo));
166 }
167
168 List<byte> message = new List<byte>();
169 length = messageParts.Count;
170 for (int i = 0; i < length; i += steps) {
  byte toAdd = 0;
  for (int j = 0; j < steps; j++) {
    toAdd <<= logOp;
    toAdd += messageParts[i + j];
  }
  message.Add(toAdd);
}
```

Listing 4.10: Decodes the message encoded into the JPEG image using the graph-theoretic method
File: JPEGDecoder.cs

The message found by `decodeScanData()` will be returned as `byte[]` to the caller of `Decode()`.

4.3 Implementation of Custom Tables in the GUI

In section 3.5 the need for custom options was described. One of these options was the possibility to choose custom Huffman and quantization tables. This is implemented into our programme as a custom component for the Forms designer.

We want the user to be able to define their own Huffman tables. This is done using textboxes, which will be converted into entries of a `HuffmanTable`. We want these textboxes to function as a joint set so that we can easily move the textboxes around and have them use them as one component.

To do this, a class `HuffmanTableComponent` inheriting from the `System.Windows.Forms.Panel` class was created.

This class, taking a `HuffmanTable` in its constructor, calls the `AddRow()` method to create a `Panel` and then for each `HuffmanElement` in the given `HuffmanTable` adds a `TextBox` to the `List<TextBox> runSizeBoxes` and sets its size, position, max length and font and then a `TextBox` to the `List<TextBox> codeWordsBoxes` and sets the same properties. This can be seen in listing 4.11. The corresponding data from the `HuffmanTable` the component is made from is then assigned as the boxes `.Text` property.

```
public HuffmanTableComponent(HuffmanTable huffmanTable) {
  HuffmanTable table = huffmanTable;
  var elementList = Table.Elements.ToList();
  Size = new Size(410, 244);

  _addTopDescription();

  for (int i = 0; i < Table.Elements.Count; i++) {
    AddRow();

    string codeWord =
       ↪ Convert.ToString(elementList[i].Value.CodeWord, 2);

    if (codeWord.Length != elementList[i].Value.Length) {
      codeWord = codeWord.PadLeft(elementList[i].Value.Length,
         ↪ '0');
    }

    codeWordsBoxes[i].Text = codeWord;

    runSizeBoxes[i].Text =
       ↪ Convert.ToString(elementList[i].Value.RunSize,
       ↪ 0x10).PadLeft(2,'0');
  }

  InitializeComponent();
}
```

Listing 4.11: HuffmanTableComponent constructor **File:** HuffmanTableComponent.cs

As mentioned, the class implements the AddRow() which adds one more row as seen in listing 4.12.

```
public void AddRow() {
  int j = codeWordsBoxes.Count();

  codeWordsBoxes[0].Select();
  VerticalScroll.Value = 0;

  addCodeWordsBox(j);
  addRunSizeBox(j);

  _addNumberIndicator(j);

  codeWordsBoxes[runSizeBoxes.Count() - 1].Select();
}
```

Listing 4.12: AddRow() method **File:** HuffmanTableComponent.cs

Lastly a SaveTable() method is implemented as seen in listing 4.13. This method begins with the initialisation of a new HuffmanTable and then uses a for-loop to run through the number of rows in the component. This for-loop checks if each pair of text boxes are empty. If they are, we skip the text boxes. If they are not, the entries in the text boxes are added as the runSize and the codeword of a HuffmanElement.

```
public HuffmanTable SaveTable() {
    HuffmanTable h = new HuffmanTable();

    for (int i = 0; i < codeWordsBoxes.Count; i++) {
      if (string.IsNullOrWhiteSpace(runSizeBoxes[i].Text) ||
         ↪ string.IsNullOrWhiteSpace(codeWordsBoxes[i].Text))
         ↪ {
        continue;
      }

      byte runSize = Convert.ToByte(runSizeBoxes[i].Text, 16);
      ushort codeword =
         ↪ Convert.ToUInt16(codeWordsBoxes[i].Text, 2);
      h.Elements.Add(runSize, new HuffmanElement(runSize,
         ↪ codeword, (byte)codeWordsBoxes[i].Text.Length));
    }
    return h;
  }
}
```

Listing 4.13: Save method of the `HuffmanTable` method **File:** HuffmanTableComponent.cs

A custom component `QuantizationTableComponent` is implemented in an almost identical way, only simpler, as a `QuantizationTable` is always the same size, 64. The final version of the graphical user interface can be found in appendix C.

Chapter 5

Tests and Optimisations

5.1 Software Testing

During the planning phase of the project we decided to continually test the code that we wrote. Our strategy for testing revolved around one member of the group creating tests, for each new component, method and logical statement. The aim of this continual testing alongside the software development, was to catch errors and bugs as soon as possible, so there would not be problems further down the line. This would then in turn minimise time and effort otherwise used on debugging, and therefore also lead to a more stable programme. The aforementioned tests would additionally improve the maintainability of the software, allowing future changes and upgrades.

5.1.1 Testing Private Methods

The goal of unit tests is to make sure that the code works as intended. If the unit tests are thorough, they become a way to prove that the class does everything in the specifications, and shows that other people can count on the class to perform as expected. Now if someone wants to use a class, they should only have access to the public methods and properties. This means that as long as the unit tests show that the public methods and properties of a class work, they do not have to explicitly show that the private methods work as intended.

If the private methods do not work, there is a risk that the public methods will not work either. But as developers we cannot guarantee that the private methods have any effect, as it is only the public methods that we have chosen to let other people see. So to test the private methods, one would have to test the public methods thoroughly enough so that every part of the private methods would be tested as well.

So while data-hiding makes it much easier to implement other people's work, and makes it clearer what a certain class offers of opportunities, we lose some of the flexibility when testing our code. This is because unit testing is basically an implementation of the class under test, and seeing if that class gives the expected result, given certain criteria.

Our programme consists of multiple classes, but a lot of the work is done in private methods in the class JPEGImage, and the only way we would be able to test the output is to check the file, which the public method Save(string) can provide.

Testing every logical statement in the programme with the aforementioned method, would certainly break one of the principles of unit testing, namely the fact that they should be quick to run.

An optimal scenario would be that the press of a button, would quickly tell us if something has been broken due to a change somewhere in the programme. Having to wait multiple minutes on images being created in full and tested byte-for-byte, to know if we broke something in one private method, would result in the unit tests not being run as often.

Of course people before us have run into this problem, and therefore solutions are readily available. Microsoft offers a library called Microsoft.VisualStudio.TestTools.UnitTesting which contain the class PrivateObject and PrivateType. With PrivateObject we can pry open an object, and access private methods on an instance via reflection. PrivateType similarly allows us to access private static members.

The syntax becomes somewhat awkward as we have to rely on strings containing method names to access the private members, but it does offer possibilities to test the private methods much easier than through the public methods as described earlier.

The actual usage can be seen in listing 5.1, where a PrivateObject is used to invoke the private method _breakDownMessage. As it can be seen, the syntax is quite different from how one would normally invoke a method, but it is still clear what is going on. We first create a new PrivateObject which contains an object that we want to test. From there on we can use reflection to invoke the method _breakDownMessage, and lastly use an assert as we would normally do.

```
[Test()]
public void BreakDownMessage_Test() {
  PrivateObject po = new PrivateObject(
                       new JpegImage(
                         new Bitmap(200, 100), 100, 4));
  byte[] message = new byte[] {1,1,1};

  po.Invoke("_breakDownMessage", message);
  List<byte> messageList = new List<byte>();
  messageList = (List<byte>)po.GetField("_message");

  List<byte> expectedList = new List<byte> {0, 0, 0, 0, 0, 0,
    ↪ 3, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1};
  NUnit.Framework.Assert.AreEqual(expectedList, messageList);
}
```

Listing 5.1: Example usage of the PrivateObject class **File:** JPEGImageTests.cs

5.1.2 Unit Testing Stegosaurus

The construction of tests was done using the open source unit testing framework NUnit. Using this framework allows the creation of tests to be streamlined and automated, allowing a group member to frequently test the new components as they are created.

Bitlist BitList is a custom implementation based on the BitArray from the .NET framework, It is often used in our programme, which means that it has to be tested thoroughly. First of all, it has to act like a List, which includes being able to have values inserted at specific indices. This is tested in the InsertTest_IndexInMiddle_InsertionOnIndex() method. Another property of the BitList is that 0 should be evaluated to false, and 1 should be true.

Vertex Vertex is comprised of two sample values, a message, and an m-value. It has a single method, ToString(), which is simply a string containing each sample value. All in all, a simple class to test. The test, Vertex_ToStringTest(), is an assertion that, when creating a Vertex object from two sample values, a message values, and an m-value, the string returned from ToString() is equal to the two sample values concatenated. This test passes, making us able to test other classes that make use of Vertex.

Edge This class Edge has five properties, two of which are the vertices that define the edge, one weight value, and two bools defining the properties of vertex value switching. Other than that, the class has no methods that define functionality, it just has to be correctly compared to other edges and be printed. This means that testing the class is fairly easy. The first test, Edge_ToStringTest(), creates an Edge object from two defined vertices, and then asserts that it prints the correct vertex values.

The goal of second test, ComparisonEdgesWithDifferentWeight_LowestWeightFirst(), is to test difference between two edges.

We create two vertices with different sample values, and assert that one edge is higher positioned in the sort order than the other. The final test, Equal_EdgesWithSameVerticesAndValues_True, asserts that two edges with the same vertices, message values, and modulo values, are equal.

Graph The Graph class has two properties: List<Vertex> and List<Edge>. As with Vertex and Edge, Graph overrides ToString(), which means that it will have to be tested.

Graph's ToString() returns a header as well as a list of vertex sample values. To test this, we create a number of different vertices and add them to the graph. The ToString() method is then compared to a string with an expected List<Vertex>.

Another method to be tested is GetSwitches(). This is done by creating a graph with a number of different edges, each with a different set of vertices. Then we assert that the List<Edge> returned by the method is the same as an expected List<Edge> kept after the method is called.

HuffmanElement The HuffmanElement class has three properties: runsize, length, and codeword. Objects of this class must be comparable to other objects of the same class, which means that CompareTo() is implemented and has to be tested.

Two Huffman elements are created with different runsizes and codewords and added to a list which is then sorted. It is then asserted that the position of the element with the smaller runsize is located in the list before the other element.

Another method that is tested is Equals, since that is overridden. Two Huffman elements are created with different runsizes, codewords, and lengths, and they are compared using Equals(). It is asserted that the first set of elements are not equal, since they have different properties. In the same test, two elements with the same properties are created, and it is asserted that they are actually equal.

The last test is for ToString(). A Huffman element is created, and it is asserted that a formatted string with expected variables is the same as the call to ToString().

HuffmanTable This class is fairly comprehensive, as it has a lot of methods that need to be tested. It is comprised of a Dictionary<byte, HuffmanElement>. First, we need to test that a table has access to an element's codeword. A number of Huffman elements - the first of which has a codeword 2 - are created and added to a table. It is then asserted that the number 2 and the codeword of the table's first element are equal.

Next, we need to test Combinations(), which returns a byte[] of the number of codes of a specific length. Two HuffmanElement objects are made with different runsizes and codewords, but same length: eight. It is asserted that there are two elements of length eight.

HasCode() checks if a certain code is found in the table. To test this, a table with a number of HuffmanElement objects are created. It is then asserted that the runsize of one of the elements is equal to an expected value, 0. It is also tested that, if no element has a specified code, it returns null.

ToString() is overridden, and it is asserted that it returns an expected string.

Lastly, we need to test GetElementFromRunSize(), which, as the name implies, returns a HuffmanElement from a specific runsize. A table with a number of HuffmanElement objects is created, one of which is tested against an expected input to GetElementFromRunSize().

QuantizationTable QuantizationTable is yet another important class in our programme, and thus it needs to be tested. Since the table entries need to be arranged in a zigzag order, it is tested in QuantizationTableTest_Zig_Zag_Entries(). A default quantization table is created and arranged in a zigzag order in QuantizationTable's constructor. A manually calculated order is compared to the order in the constructor, and it is asserted that they are equal.

The Scale() method scales quantization values using a certain quality as input. An expected table is manually calculated, and it is asserted that this is the same as the one returned by Scale. Scale() should throw exceptions when the quality parameter is above 100 or below zero.

It is asserted that the method throws OutOfRangeException() when called with a quality of -4 and 101. Since we know that QuantizationTable behaves as expected, we can comfortably use it in JpegImage.

JpegImage The JpegImage class is a comprehensive class, making the need for tests large. The first test is simple, testing if the *m*-value of an instance of the class gets set. Since it is a private variable, we need to use PrivateObject as previously described. The next two tests test whether exceptions are thrown when an instance of JpegImage is created without a Bitmap cover image and when Save() is called without a JpegWriter.

BreakDownMessage_Test() tests whether the _breakDownMessage() method behaves as expected. PrivateObject is once again required, as we need to look at the value of a private variable. A message is broken down into a byte[] containing length, an *m*-value and the message. It is then asserted that a manually calculated encoding of the same message and the value that the method returns is the same.

The next test tests whether _padCoverImage() works as expected when the dimensions of an image is divisible by 16. It is asserted that _padCoverImage() does not do anything to the image. However, when the dimensions of an image is not divisible by 16, it is expected that _padCoverImage() returns a padded image. It is then asserted that the returned image and an expected image with dimensions divisible by 16 are the same. Much like the previous test, colour should also be the same in an original image and a padded version. This is tested by picking a pixel from a specific location from the original image, and an equivalent pixel from the padded image, and asserting that the colour is the same.

The _copyBitmap() method is tested as well. It is asserted that a specific pixel from the original image and a copied image are the same.

Another test required is for `_splitToChannels()`, which tests that a pixel with specific RGB-values is correctly split into YCbCr-channels.

The method `_downSample()` is tested by downsampling a `float[,]`, and asserting that the returned array is the same as a manually calculated array.

The `_discreteCosineTransform()` method must be tested as well, since that is an essential part of our programme. This is done by creating an instance of `JpegImage` and then asserting that a table of 64 manually calculated cosine values is the same as the table returned from the method under test. Since it returns a `float[,]`, we set a tolerance for when a number is equal to another.

The `_c` method is tested four times with different inputs.

`_quantization()` is under test as well, and this is done by filling an array with 64 manually calculated quantized values and asserting that it is the same as the array returned by `_quantization()`.

Tests are also conducted for `_addVertices()` and `_addEdge()`. `_addVertices` add vertices to a graph, where `sampleValue1` and `sampleValue2` are incremented by two for each vertex, starting from zero and one, respectively. We create a number of vertices with the same properties and manually add these to another graph, and assert that this graph contains the same vertices as the one in the method.

`_addEdge()` is tested in much the same way. We can now decide, based on the test results of the `JpegImage` class, that it works as expected, and can be safely used in our programme.

5.1.3 Conclusion on Testing

Testing the programme has definitely helped the programming. By testing we have caught arithmetic errors, such as the method `GetCapacity()` in the `JPEGImage` class calculating the amount of available bytes wrongly. This caused the programme to return a much lower capacity than what was actually correct.

Another problem we caught while testing, was with the method `Scale_MultiplierInRange_ScalesTable()` while testing the `QuantizationTable`. The programme calculates a multiplier for the quantization table, based on the quality setting. The original line of code for doing this, was: `double scale = ((100 ↪ - quality)/ 53 + 0.125);`. From testing we learned that this returned wrong results, due to the fact that integer division was used when doing the division. Changing this to `double scale = ((double)(100 - quality)/ 53 + 0.125);` made the test succeed.

More serious mistakes were caught as well. A mistake in the `Graph` class caused the edges returned by the `GetSwitches()` method to not make the vertices perfect pairs. And they would have to be forced after the swaps anyway. This essentially nullified the benefits from the graph theoretic method, as the graph was not being used. The tests made us find this mistake, and the performance of the programme improved greatly.

Looking back to the unit testing of this programme, it definitely helped us find mistakes, but there are things that could have made this process much easier. Instead of cramming a lot of the fundamental functionality of the programme into the `JPEGImage` class, we could have split the process into smaller parts. If we had made classes along the line of `GraphEncoder` which handled all of the graph theory, i.e. adding vertices and edges, the `JPEGImage` would have been easier to test, and most likely have made it so that we could test purely by using the public methods.

5.2 Optimisation

We have managed to implement several complex algorithms in our programme. This was immediately visible if the JPEG-encoder was run with larger images (such as the 4K (3840x2160) resolution images that the camera in some smartphones output) or if long messages of several hundred bytes were embedded in the image. Doing so took upwards of several minutes on slower computers, so some optimisation of the code was clearly a requirement. This section will focus on how these optimisations were implemented.

To see what kinds of optimisations were useful and where they were needed, it was necessary to objectively measure their worth. The easiest way to do this, was by timing how long different parts of the code took to complete. To show this, we created a `System.Diagnostics.Stopwatch` object in `JPEGImage`, which allowed us to keep track of how many milliseconds something took. In order to ensure that we could compare the timings from different runs, it was important to establish some base rules to follow. We used the same image and message for every run. The image is a 4K resolution JPEG image of a landscape with a hot air balloon in it. To ensure that we ran the method `_padCoverImage`, we removed eight pixels from the right and bottom sides of the image, bringing the resolution to 3832x2152. The message consisted of 640 bytes of the ASCII characters *A*-*Z* and was encoded with an m-value of 4. This was equivalent to the length of four regular SMS messages end-to-end, and therefore constituted what we believed to be a message capable of containing enough information for most regular uses.

All runs were done on a *Release* build, as there seemed to be significant overhead on a *Debug* build. Some basic optimisations such as moving the calculation of the upper-bound outside the declaration of for-loops had already been done at this point. While being a very simple thing to do, this did have some visible results in a few areas.

We used a profiler to get a better understanding of which parts of the code took a substantial amount of time to run. When run with the aforementioned parameters, the following methods and procedures constituted around 90% of the total time spent by our mostly unoptimised programme: `_padCoverImage`, `_splitToChannels`, `_encodeAndQuantizeValues`, Adding edges to the graph, GetSwitches and _huffmanEncoding. All numbers are averages based on ten consecutive runs, following a single "warm-up run", whose results were not saved. This was done to get more stable timings, seeing as the first run always had slightly different times than the remaining. It is quite possible that this was caused by the garbage collector, since it changes its behaviour depending on the patterns of code being run. Therefore, running the programme once without saving the data allowed it to learn the memory allocation patterns of the code and react in the same way the next ten times it was run. The warm-up run also allowed us to run the decoder to make sure that the message was being encoded properly in the image and that our optimisations had not changed the programme's output. We completed a total of three rounds of optimisations, each further improving the performance of the programme. The four different versions of the programme can be found in appendix D. The first runs and the control for our further tests are shown in table 5.1.

Table 5.1: Unoptimised encoding for three different processors.

Code segment	Processor		
	Intel® i5-4670K	Intel® i7-4700HQ	AMD® A6-3420M
Image padding	7127 ms	8060 ms	39189 ms
Channel split	3275 ms	3670 ms	20775 ms
Encoding	4294 ms	5304 ms	28603 ms
Adding edges	524 ms	626 ms	4017 ms
Get switches	1439 ms	1764 ms	10393 ms
Huffman	641 ms	761 ms	3576 ms
Total run time	**17516 ms**	**21393 ms**	**108139 ms**
Time covered	*98.8%*	*94.4%*	*99.0%*

5.2.1 First Round of Optimisations

It became clear that _padCoverImage, _splitToChannels and _encodeAndQuantizeValues were the most time-consuming. The first thing we focused on was _padCoverImage, simply because it took almost twice as long as the second most time-consuming method. JPEG-encoding requires images with both a width and a height that is divisible by 16. If an image does not fulfil this, the last pixel is copied up to 15 times in the direction needed. Since the System.Drawing.Bitmap class does not allow for resizing of images, we needed to copy the entire image before being able to pad its bottom and right side. We did this by looping through every pixel and copying it to the new Bitmap. This was done in two nested for-loops that ran from zero the width and zero to the height respectively. On the 3832x2152 image used, this amounted to 8,246,464 iterations. Every iteration used the `Bitmap` class' methods `GetPixel` and `SetPixel`. It is very likely that there is more to the property than simply returning a value and therefore quite a bit of overhead. They might very well have to check several things before being able to return the pixel.

Using the `System.Drawing.Graphics` class as shown in listing 5.2 we were able to copy portions of `Bitmap`s much more efficiently, (*HOWTO: Copy Bitmap Regions*) as can be seen in by the roughly 8000% decrease in time in table 5.2.

```
private static Bitmap _copyBitmap(Bitmap bitmapIn, int width,
   ↪ int height) {
  Bitmap bitmapOut = new Bitmap(width, height);
  Graphics g = Graphics.FromImage(bitmapOut);
  Rectangle rect = new Rectangle(0, 0, bitmapIn.Width,
     ↪ bitmapIn.Height);
  g.DrawImage(bitmapIn, rect, rect, GraphicsUnit.Pixel);
  g.Dispose();

  return bitmapOut;
}
```

Listing 5.2: Copying a `Bitmap` using the `Graphics` class. **File:** first_round/CS/JPEGImage.cs.

Seeing this overhead on the `Bitmap` class, made us suspect the same of the related `System.Drawing.Color` class, which holds the data of a single pixel. During `_splitToChannels` we also looped through every pixel of the image and saved it in a temporary variable. From this we used the bytes `R`, `G` and `B` three times each. In this round of optimisations we saved these values to temporary variables instead of getting them from the saved pixel each time they were needed.

This reduced the calls to the `Color`'s `R`, `G` and `B` getters, which had a positive impact on the overall performance of the method. We also managed to get a slight increase in performance on the Huffman-encoding, by changing a loop in the `BitList` to only run every eighth time a bit was added to the list. This change can be seen in listings 5.3 and 5.4.

```
public void CheckedAdd(int val) {
  if (_addCounter % 8 == 0) {
    _latestEntries.SetAll(false);
  }
  _latestEntries[_addCounter % 8] = (val == 1);
  Add(val == 1);
  bool allOne = true;

  for (int i = 0; i < 8; i++) {
    if (!_latestEntries[i]) {
```

```
 95        allOne = false;
 96        break;
 97      }
 98    }
 99
100    if (allOne) {
101      for (int i = 0; i < 8; i++) {
102        Add(false);
103      }
104    }
105
106    _addCounter++;
107  }
```

Listing 5.3: Original `CheckedAdd` in `BitList`. Note the for-loop on line 93. It always runs. **File:** unoptimized/CS/BitList.cs.

```
public void CheckedAdd(int val) {
  if (_addCounter % 8 == 0) {
    _latestEntries.SetAll(false);
  }
  _latestEntries[_addCounter % 8] = (val == 1);
  Add(val == 1);
  bool allOne = false;

  if (_addCounter % 8 == 7) {
    allOne = true;
    for (int i = 0; i < 8; i++) {
      if (!_latestEntries[i]) {
        allOne = false;
        break;
      }
    }
  }

  if (allOne) {
    for (int i = 0; i < 8; i++) {
      Add(false);
    }
  }

  _addCounter++;
}
```

Listing 5.4: Improved `CheckedAdd` in `BitList`. The for-loop only runs an eighth of the time. **File:** first_round/CS/BitList.cs.

Table 5.2: First round of optimisations. Improved Bitmap copying, fewer calls to the properties of Pixel and an improved BitList.

Code segment	Processor					
	Intel® i5-4670K	Decrease in time†	Intel® i7-4700HQ	Decrease in time†	AMD® A6-3420M	Decrease in time†
Image padding	92 ms	7646%	101 ms	7861%	533 ms	7246%
Channel split	3148 ms	4%	3786 ms	-3%	19431 ms	7%
Encoding	4306 ms	0%	5145 ms	3%	29396 ms	-3%
Adding edges	536 ms	-2%	627 ms	0%	4054 ms	-1%
Get switches	1464 ms	-2%	1765 ms	0%	10990 ms	0%
Huffman	520 ms	23%	651 ms	17%	3111 ms	15%
Total run time	**10287 ms**	**70%**	**12328 ms**	**74%**	**68588 ms**	**58%**
Time covered	*97.9%*		*98.0%*		*98.4%*	

†As compared to the respective timings of table 5.1.

5.2.2 Second Round of Optimisations

While there was a slight speed-up by not accessing the properties of the Color as often, it did not do as much. It was apparent that any use of GetPixel() was going to cause problems on larger images. Using LockBits we were able to use an IntPtr to point at the data in memory (*BitmapData Class*). The reason for doing this instead of using an actual pointer, is the fact that pointers in C# requires the code they are used in to be wrapped in the unsafe keyword. We could not be entirely sure of the implications of this, since it can require different privileges on different platforms such as in the phone application that the ITC was working on. Using this method to get the R, G and B components of the source Bitmap was beneficial to the programme's execution time. In fact, it was almost eight times faster than the previously used method, as seen in table 5.3.

Several changes to the two nested loops, which added edges to the graph, resulted in an improvement to the *Adding edges* step as well as GetSwitches(). The original algorithm (as shown in listing 5.5) had complexity $\mathcal{O}(n^2)$, so longer message lengths could result in much longer running times.

```
int threshold = 5;
foreach (Vertex currV in graph.Vertices) {
  foreach (Vertex otherV in graph.Vertices.Where(otherV =>
      ↪ currV != otherV)) {
    _addEdge(true, true, currV, otherV, threshold, graph);
    _addEdge(true, false, currV, otherV, threshold, graph);
    _addEdge(false, true, currV, otherV, threshold, graph);
    _addEdge(false, false, currV, otherV, threshold, graph);
  }
}
```

Listing 5.5: Original algorithm for adding edges to the graph. **File:** first_round/CS/JPEGImage.cs.

We made sure to only look at the vertices whose vertices did not already fit with the message. With an m-value of four this meant that we would not have to check if we could add edges to around 25% of the vertices. We further reduced the overall amount of times the loops ran, by making sure that the inner loop only looked ahead in the list of vertices, which meant we only had to add each edge once. Before this, we were adding every edge twice, with the start and end vertex flipped. This meant that our list of edges was halved, which effectively reduced the running time of `GetSwitches` to a fourth. These changes can be seen in listing 5.6.

```
List<Vertex> tbc = graph.Vertices.Where(x => (x.SampleValue1 + x.SampleValue2).Mod(x.Modulo) != x.Message).ToList();
int length = tbc.Count;
int threshold = 5;
for (int i = 0; i < length; i++) {
  for (int j = i + 1; j < length; j++) {
    _addEdge(true, true, tbc[i], tbc[j], threshold, graph);
    _addEdge(true, false, tbc[i], tbc[j], threshold, graph);
    _addEdge(false, true, tbc[i], tbc[j], threshold, graph);
    _addEdge(false, false, tbc[i], tbc[j], threshold, graph);
  }
}
```

Listing 5.6: Improved algorithm for adding edges to the graph. **File:** second_round/CS/JPEGImage.cs.

On another note, we also changed the order of the three checks that took place in `_addEdge`. Two of these checks were more computationally demanding than the third. By moving the least demanding check so it was the first one evaluated, it meant that the more demanding checks were not performed quite as often. This allowed us to save quite a few operations on each iteration. Furthermore, we added the calculated edge weights to the edges instead of having a property that calculated them every time. This saved a few operations at a later time, when the edges were ordered by weight. The two different versions of `_addEdge` can be seen in listings 5.7 and 5.8.

```
private static void _addEdge(bool firstFirst, bool secondFirst, Vertex first, Vertex second, int threshold, Graph g) {
  if (((firstFirst ? first.SampleValue2 : first.SampleValue1) + (secondFirst ? second.SampleValue1 :
     ↪ second.SampleValue2)).Mod(first.Modulo) ==
     ↪ first.Message) {
    if (((firstFirst ? first.SampleValue1 :
       ↪ first.SampleValue2) + (secondFirst ?
       ↪ second.SampleValue2 :
       ↪ second.SampleValue1)).Mod(second.Modulo) ==
       ↪ second.Message) {
      Edge e = new Edge(first, second, firstFirst,
         ↪ secondFirst);
      if (e.Weight < threshold) {
        g.Edges.Add(e);
      }
    }
  }
}
```

Listing 5.7: The original _addEdge method. **File:** first_round/CS/JPEGImage.cs.

```
private static void _addEdge(bool firstFirst, bool
   ↪ secondFirst, Vertex first, Vertex second, int threshold,
   ↪ Graph g) {
  int weight = Math.Abs((firstFirst ? first.SampleValue1 :
     ↪ first.SampleValue2) - (secondFirst ?
     ↪ second.SampleValue1 : second.SampleValue2));
  if (weight < threshold) {
    if (((firstFirst ? first.SampleValue2 :
       ↪ first.SampleValue1) + (secondFirst ?
       ↪ second.SampleValue1 :
       ↪ second.SampleValue2)).Mod(first.Modulo) ==
       ↪ first.Message) {
      if (((firstFirst ? first.SampleValue1 :
         ↪ first.SampleValue2) + (secondFirst ?
         ↪ second.SampleValue2 :
         ↪ second.SampleValue1)).Mod(second.Modulo) ==
         ↪ second.Message) {
        Edge e = new Edge(first, second, weight, firstFirst,
           ↪ secondFirst);
        g.Edges.Add(e);
      }
    }
  }
}
```

Listing 5.8: The improved _addEdge method. **File:** second_round/CS/JPEGImage.cs.

Table 5.3: Second round of optimisations. Using a pointer to recieve Bitmap data directly and improved edge adding logic.

Code segment	Processor					
	Intel® i5-4670K	Decrease in time†	Intel® i7-4700HQ	Decrease in time†	AMD® A6-3420M	Decrease in time†
Image padding	92 ms	7666%	97 ms	8249%	503 ms	7687%
Channel split	367 ms	791%	386 ms	850%	5096 ms	308%
Encoding	4303 ms	0%	4796 ms	11%	29812 ms	-4%
Adding edges	102 ms	415%	116 ms	442%	629 ms	539%
Get switches	256 ms	462%	289 ms	511%	2244	388%
Huffman	540 ms	19%	610 ms	25%	3125 ms	14%
Total run time	**5883 ms**	**198%**	**6534 ms**	**227%**	**42444 ms**	**155%**
Time covered	*96.2%*		*96.3%*		*97.6%*	

†As compared to the respective timings of table 5.1.

5.2.3 Third Round of Optimisations

The final optimisations we did, had the potential to make a large difference on most systems. Taking advantage of the multi-core design of most modern processors, we made some of the intensive parts of our code run in parallel on multiple threads, which can utilise multi-core systems. The effects of this can vary widely depending on the processor architecture and not all smart phones would be able to take advantage of it. The ones that can, should be able to see a dramatic increase in performance of some parts of the code.

Running loops in parallel as we did, required seeing the code and its data dependencies in a different way than we usually do. With the exception of loops, normally, code is run from the top to the bottom. The data dependencies follow the same pattern. When it comes to multi-threaded code though, this pattern changes, since the same lines of code can potentially be accessed by different threads at the same time. This can lead to race-conditions, where the result of code can change depending on which thread gets to make changes first. Data can potentially be overwritten, since the same parts of memory can be written to at the same time. In a high-level language such as C#, this would most likely cause a run-time exception instead. An example of a parallelised for-loop can be seen in listing 5.9.

```
List<Vertex> tbc = graph.Vertices.Where(x => (x.SampleValue1 +
    ↪ x.SampleValue2).Mod(x.Modulo) != x.Message).ToList();
int length = tbc.Count;
int threshold = 5;
Parallel.For(0, length, i => {
  for (int j = i + 1; j < length; j++) {
    _addEdge(true, true, tbc[i], tbc[j], threshold, graph);
    _addEdge(true, false, tbc[i], tbc[j], threshold, graph);
    _addEdge(false, true, tbc[i], tbc[j], threshold, graph);
    _addEdge(false, false, tbc[i], tbc[j], threshold, graph);
  }
});
```

Listing 5.9: Parallelisation of the algorithm for adding edges to the graph. **File:** third_round/CS/JPEGImage.cs.

Because of these dangers, it is important that one manages any and all of these synchronisation issues properly. It is not always possible to use the same data-structure as single-threaded code does, and since we implemented multi-threading after making the code single-threaded we were not able to convert all the code. Specifically Huffman-encoding proved too cumbersome to efficiently multi-thread. However, DCT-calculation, quantization, adding edges and splitting the image to YCbCr-channels were successfully rewritten to allow for the code to run in parallel on several cores with the same output.

By using the smallest data types capable of holding our data and properly unloading large unused variables, we effectively reduced peak memory usage from almost 400MB to around 100MB. This had the potential to improve the spacial locality of the data, which would lead to better utilisation of the different levels of caches, all in all leading to a better performing programme.

Since we knew that there was a certain overhead on method invocations, we tried to force the compiler to in-line some of the most invoked methods we had. This did not seem to have an effect, so it is quite likely that the compiler already in-lined the methods we tried it on.

Table 5.4: Third and last round of optimisations. Memory optimisations and multi-threading of several methods.

Code segment	Processor					
	Intel® i5-4670K	Decrease in time†	Intel® i7-4700HQ	Decrease in time†	AMD® A6-3420M	Decrease in time†
Image padding	94 ms	7511%	98 ms	8107%	511 ms	7576%
Channel split	113 ms	2789%	122 ms	2914%	769 ms	2600%
Encoding	1879 ms	129%	2447 ms	117%	6948 ms	312%
Adding edges	51 ms	993%	53 ms	1074%	197 ms	1943%
Get switches	295 ms	388%	333 ms	430%	2286	378%
Huffman	559 ms	15%	624 ms	22%	3025 ms	18%
Total run time	**3210 ms**	**446%**	**3916 ms**	**446%**	**14764 ms**	**632%**
Time covered	*93.2%*		*93.9%*		*93.0%*	

†As compared to the respective timings of table 5.1.

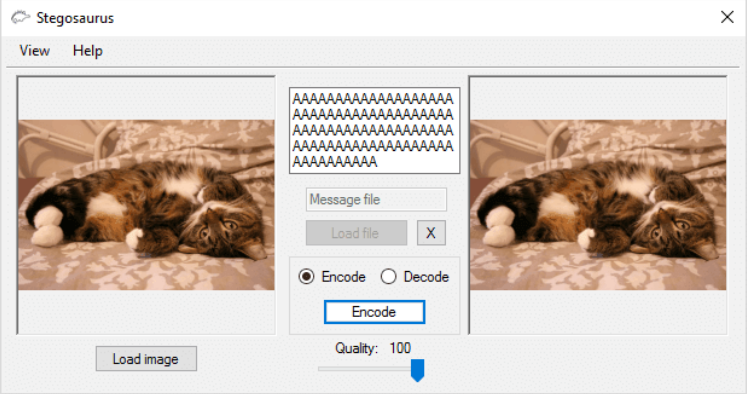
5.2.4 Final product

After having done these steps to optimise our programme, we ended up with a product that performed well enough for us to try testing it with larger messages to see what we were able to encode in files. The final implementation can be seen in appendix C. We wanted to know if the capacity of the cover image or the time it took to encode the data was going to be the bottleneck.

The application can be seen on figure 5.1. To test the image quality, we embedded a 4KB message into a 1024x683 image. With a quality level of 53% 4KB was the upper limit of how much data we could encode, while with a quality of 100%, the image could contain 19KB of data.

On an Intel® i5-3317U processor it took around 20 minutes to encode a 4KB message, while a 1KB message took 20 seconds to be encoded in the same image. This tells us that the encoding time plays a much larger role than the capacity of the image.

On figure 5.2 the results of the encoding are shown. There are no obvious visible changes from figure 5.2b to figure 5.2a. This tells us that it is difficult for a person to tell the difference from the original image to the image with the encoded message.



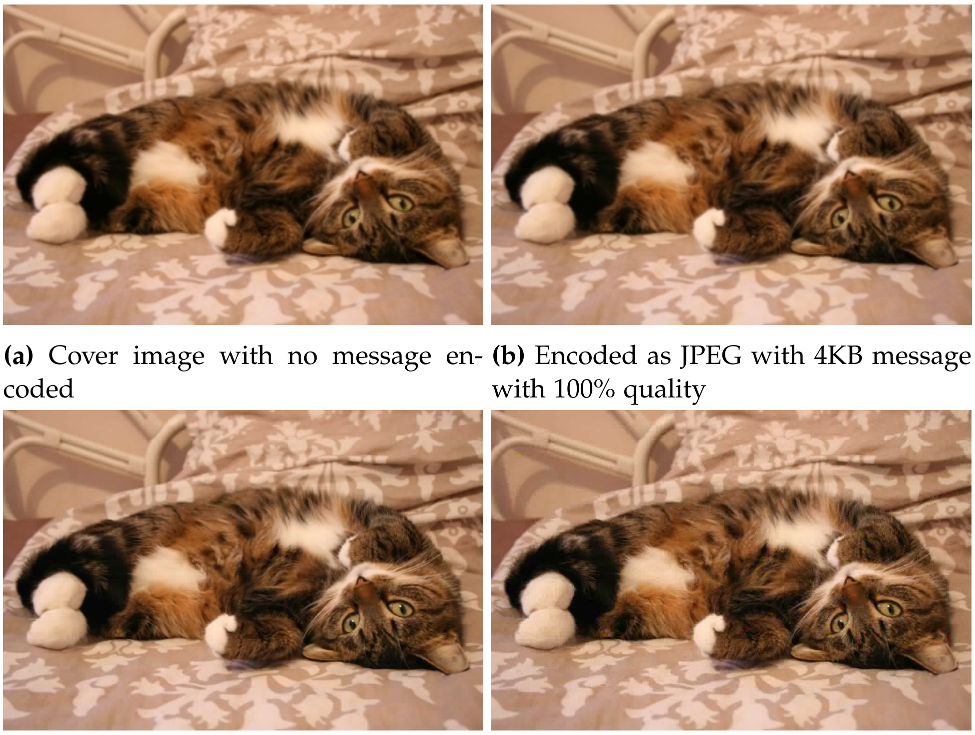
AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAA

Message file

● Encode ○ Decode

Quality: 100

Figure 5.1: Form



(a) Cover image with no message encoded

(b) Encoded as JPEG with 4KB message with 100% quality

(c) Encoded as JPEG with 53% quality with no message

(d) Encoded as JPEG with 53% quality with 4KB message

Figure 5.2: Encoding of the images (tiina93, 2009)

There are very slight differences between figures 5.2c and 5.2d. They are only visible if greatly enlarged, and only noticeable because we have both the cover and stego image to compare. The reason for the differences is that the quantization tables used when saving at lower qualities have much higher values, which means any changes are multiplied by a greater number when decoding the image in an image viewer.

The only noticeable differences from figure 5.2d to the original cover image, is the overall lower quality from the JPEG compression, and not from the embedding of the message.

Not being able to see the message with the naked eye is of course not enough, but it does tell us that the images are not changed greatly by embedding the message. In the next chapter, we will test the stego images by better means than simply looking at them.

Chapter 6

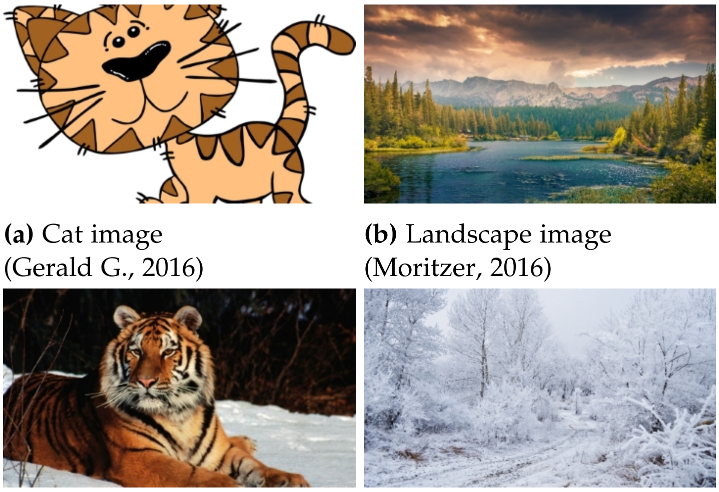
Discussion

In the problem analysis we used a simple LSB-method for hiding data in an image. Specifically, we hid an image inside of another PNG image. While the differences in the image with and without the message-image were hardly noticeable to the human eye, we did find that comparing colour histograms of the two images revealed that there had in fact been made changes to it. In an effort to diminish this problem, the graph-theoretical approach we implemented attempted to move pixels around in the image, rather than change their values. This meant that colour histograms of the image before and after the message had been encoded should look very similar, almost identical. Any changes would come from the unmatched pairs: the quantized DCT values used in the encoding that we did not find any interchangeable values for. When this happened we were forced to change the individual values, which led to a change in the colour composition of the image. Using this method, we expected the changes to be small enough so that it would not immediately draw attention to the image, if it was being subjected to a colour histogram.

6.1 Changes to the Image

To find out roughly how many of these forces there were in a given image, we used our programme to save different messages in different images. The five different message lengths were: 70, 140, 280, 560 and 1120 bytes. Each message consisted of the ASCII characters A-Z repeated to fill out the message. Each of these messages were encoded in four images with varying motives, but the same resolution (1920x1080). The images can be seen in figure 6.1.

We used a cartoon-like drawing of a cat to see how our software would work on an image that was not ideal for the JPEG format. The landscape image contained many different colours and very complex patterns, which made it ideal for JPEG compression. The same could be said for the tiger, but it contained larger areas of similar colours than the landscape did. The image of the snowy forest road contained very few colour nuances, but was almost entirely made up of the luminance channel. This led to a total of 20 tests as seen in table 6.1.



(a) Cat image (Gerald G., 2016)

(b) Landscape image (Moritzer, 2016)

(c) Tiger image (Vegetaveg, 2015)

(d) Snow image (Koshkina, 2016)

Figure 6.1: The four images used for the tests in table 6.1

Table 6.1: The percentages of values which the program swapped or forced. Since this was run with *m*-value of 4, about a fourth of the values already fit

Image	Message length	Swaps	Forces	Already fit
Cat	70	56%	19%	26%
	140	58%	18%	24%
	280	65%	12%	23%
	560	65%	11%	24%
	1120	66%	8%	25%
Landscape	70	55%	15%	30%
	140	63%	11%	27%
	280	62%	11%	27%
	560	62%	13%	25%
	1120	63%	12%	25%
Tiger	70	63%	8%	28%
	140	64%	8%	27%
	280	68%	6%	26%
	560	70%	5%	25%
	1120	72%	4%	24%
Snow	70	53%	21%	26%
	140	58%	13%	29%
	280	66%	8%	26%
	560	67%	8%	25%
	1120	69%	7%	24%

As the table showed, some of the images required much fewer forces than others. The main reason for this was that the changes in colour within the area in which the message was encoded was different from image to image. For example, the tiger performed much better than the other images, because it contained mostly the same colour in the area that contained the message. When constructing the graph, we only added edges with a weight under a certain threshold. When the colours of an area were closer together, there would also be a higher concentration of edges with a smaller weight, which would lead to a larger graph.

6.2 Size of the Graph

It became apparent that a longer message required fewer forces. This made sense, seeing as a longer message meant a larger graph with more edges and therefore possible ways of swapping values. During development of the programme we played around with the idea of using more vertices than what was required for the message, as we predicted it would mean fewer forces. These predictions would seem to be correct, but we never implemented the idea for two reasons.

One, we were not entirely certain how to do it: should we just use twice as much as was actually needed? This would lead to using much more than what was necessary with a longer message, since the improvement quickly diminishes with longer messages. Instead it would make more sense to always use a certain minimum of values, so shorter messages could be encoded properly, but longer ones did not take too long to encode. What were to happen if the image simply did not contain enough values then? Should the programme inform the user that they needed to use a larger image or attempt to encode the message with the vertices it could make? What would this mean for the `GetCapacity` method? Should it take into consideration these extra values or just ignore it entirely? An entirely different approach would be to let the user decide themselves, how many values they wished the algorithm to use. Presenting this in a user-friendly way constituted a challenge in itself though.

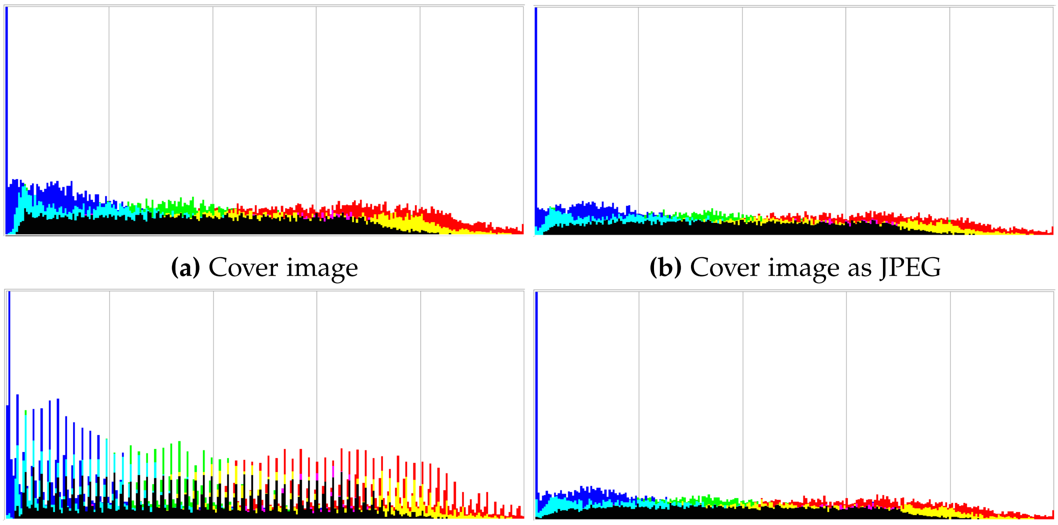
The second reason we did not implement it was due to performance concerns. Since our programme performed rather poorly during most of the development we considered it a very bad idea to construct a larger graph than we already were constructing. Seeing as we managed to optimise the programme quite significantly, we could have probably implemented a solution regardless.

6.3 Colour Histograms

Actually comparing colour histograms from the two methods was not as trivial as one could have hoped. Using the LSB-method only made immediately visible changes to the colour histograms if about a quarter of the image had data encoded in it. When using an image of substantial size, a quarter of an image can hold a large amount of data. A 512x512 image for example can hold 196,604 bytes, when using the two least significant bits, this means that to see the noticeable difference, we would have to encode 50,000 bytes.

Encoding 50,000 bytes using our graph-theoretic (GT) programme was completely out of the question due to the complexity of the algorithm. The longest message we tried encoding was just under 8960 bytes and took around half an hour on a decent desktop computer. Running several test runs with different message lengths made it clear that the complexity of the programme made it infeasible to ever complete an encoding of this much data.

This made a direct comparison using realistic real-world images and messages impossible. The only real comparison we could make was between images containing different messages that used up every single available byte in the image. Since the GT method became very time consuming with larger messages, we would need to use an image that was small enough that we could embed all bytes possible, without it taking too long to complete. We ended up using an image with a



(a) Cover image

(b) Cover image as JPEG

(c) Stego image with message (LSB)

(d) Stego image with message (GT)

Figure 6.2: Histogram of a 120x120 px image

resolution of 120x120 pixels.

In figure 6.2 four histograms are shown. Figures 6.2a and 6.2b show the histogram of the original image and for the image encoded as JPEG without a message. Figure 6.2c and 6.2d show the histograms of the images filled with a message using the LSB and GT methods respectively. From these results it was clear that there was not much difference between the colour histograms of the image saved as JPEG with or without the message. The image saved with LSB's histogram, however, clearly showed signs that the image had been tampered with.

The reason the histogram looked like that for the LSB image was because the bit-patterns of the encoded data became very clear in the histograms. The bytes encoded in the images were the characters A-Z in ASCII repeated the capacity of the image had been reached. A-Z in ASCII ranges from $65 - 90$, or in base two: $01000001_2 - 01011010_2$. As the range was quite short, a lot of the higher order bits would rarely change, and those patterns became obvious in the histogram, as pixels ending in these patterns became more common.

The histogram in figure 6.2c also showed how the LSB method changed images where a colour is very common. The leftmost bar describes pixels whose B-component is 0. In the original cover image, this bar was much higher than the others, and therefore much more common in the image, but the LSB has made it so that those components now get spread out, which again becomes very obvious in the histogram.

Images	**All channels**	**R**	**G**	**B**
Original and LSB	34671	17098	16800	16824
Original and GT	130389	71112	56972	71886
Original and JPEG	102460	56758	40707	57569
JPEG and GT	79163	42154	38833	43217

Table 6.2: Calculated euclidean distances for several combinations of images.

6.4 Euclidean Distance

A different way of looking for changes in an image is by looking at their euclidean distance to the original image. This can be done by calculating the distance between the colour values of each pixel in two equally sized images. The sum of all these distances can be used as a measure of the change from one image to the other. We used this same method for determining what changes were imposed on images of different sizes and formats, when uploaded to various social media and image-sharing websites. The images used for the method were the same as those previously used in the colour histogram tests.

Comparing the results of GT directly with the ones obtained from the LSB-method would be folly. While the LSB-method might incur a smaller difference in the euclidean distance, it did not necessarily make it any better at hiding the information. Comparing those two directly meant that we were looking at an image saved in the lossless PNG format and a file that not only had a message in it, but had also been compressed in a lossy way. Therefore a comparison between the original and an image saved with our JPEG encoder, but containing no message, should yield a difference in euclidean distance as well. The results of these tests can be seen in table 6.2.

It was clear that compressing the image with JPEG increased the euclidean distance between the two images. But the distance between the JPEG image with a message and the one without it, was still larger than that of the LSB method and the original. This was due to the fact that using the LSB of every pixel there was only the potential to change the value of each pixel by four. With the GT method and an m-value of four, means that each individual quantized value could also be changed by four. So why the big difference? We applied the change after the quantization step of JPEG encoding to avoid losing the data again, since the quantization step is what makes JPEG encoding lossy. This in turn meant that the small change of up to four was multiplied by the quantization table, when the image was shown and we calculated the distance.

But what did this actually mean for using the GT method to hide data? Using euclidean distance to see if images had been tampered with, required that the intercepting party be in possession of both the original and the stego image. This would indeed be possible if those sharing secret messages in images, were using cover images found on the internet and tampered with them. It would not be all that difficult to reverse image search the internet to find what was probably the original cover image and measure the euclidean distance between the two images. However, this would mean that any risk of being compromised by an outside party measuring euclidean distance, could be completely negated by simply using cover images that were not available to any intercepting parties.

Such a system could, however, check the histogram of the image, and discover abnormalities such as those the LSB method produces, as shown in figure 6.2c. Doing this would lead to very few useful results on images with messages hidden in them, using the GT programme.

Chapter 7

Conclusion

Through the ages people have sent concealed messages to each other, and now with the help of digital steganography it has become more accessible.

Our problem analysis allowed us to establish boundaries of the topic as well as goals to complete. We came to the conclusion that it was not as important to try and get messages across social media, but instead more important to get messages across in the most discrete and undetectable way possible. We did learn from social media, that the most common image format online is JPEG.

We discovered through steganalysis that by using a simple method such as LSB, it was not possible to discretely send messages, as even a simple colour histogram could reveal the existence the message. Instead we needed a method more resilient to statistical analysis. The graph-theoretic approach to steganography described in section 2.9 was the method we needed.

What we wanted to achieve was modifying the method to work with JPEG images. We wanted the algorithm to be able to encode the data without incurring visual distortion to the image. The product also had to be implemented in an object-oriented programming language.

To hide the data in JPEG images, we first had to design and implement a JPEG encoder. After having a functional JPEG encoder, we designed and implemented the graph-theoretic method along with a decoder to be able to retrieve the embedded message. The programme was tested using the .NET testing framework NUnit.

This resulted in the programme *Stegosaurus*, that can encode messages discretely as well as decode the messages, and this is something that we desired when we chose this method.

Our tests of the programme showed that it was impossible to discover the method by simply looking at colour histograms. Looking back to figure 6.2 we can see that there were barely any differences between the cover image encoded as JPEG in figure 6.2b and the stego image saved as JPEG in figure 6.2d. Even though there are some differences, these are negligible. The euclidean distance in the colour channels are generally greater using the graph-theoretic method instead of LSB, though looking back on section 5.2 the changes are not visible to the human eye when encoding the data.

This means, that it is possible for us to share images with concealed messages without raising suspicion, which was our goal. We have succeeded in creating a programme which fulfils the problem statement and we have shown that the graph-theoretic method works with JPEG images. The programme has been tested and we have found the testability of it to be low. This is partly due to the amount of private methods instead of splitting the functionality up into several more, smaller classes. Each class would then handle a subset of the problem, making it easier to test the parts individually. Additionally, the modularity of the programme would have been higher with the introduction of more classes. An example of such a class could be `GraphEncoder` class that would accept a list of numbers, and use the graph-theoretic method on them. That would move a lot of the functionality out of the `JPEGImage` class and make our implementation of the method much more versatile. In conclusion, we accomplished what we intended to do with our problem statement, but there was room for improvement in relation to our teamwork with the cluster group.

7.1 Conclusion on the cluster work

With our programme we hoped to contribute to the cluster group's collective project. Using an object-oriented programming language allowed us to define interfaces. This made it easier and less time-consuming for us to create a graphical user interface and to help the ITC group. These interfaces were made far in advance, so both groups could them in their development without being dependant on each other.

As described in section 2.10, the grounds for working with the method implemented, was to help improve our understanding and help both groups with creating better projects. This did not happen quite as planned, due to insufficient communication. Thankfully this was not catastrophic as there had also been made an agreement that the groups should be able to work together individually and this was how the course of the project went.

It was hard for three groups with completely different curricula, ideas, and requirements to come together and work on a shared project. Given enough time and more in-depth talks with each other, it could have worked, but none of the groups was prepared or experienced enough for the challenge. It seems like this kind of cluster project is well-suited for a corporate setting, where the goal is to develop a complete solution across all teams. In a university setting, however, each group was required to learn and possibly develop something based on their respective curriculum, and it makes sense that they first and foremost focus on their own project, and not on cluster work, which can only be considered a “bonus objective” that can be skipped.

7.2 Future Work

This section contains a list of possible improvements for the programme. These improvements would either enhance functionality or make hidden information harder to detect. It should be noted that the programme fulfils its purpose as it is right now.

Compatibility

For future use, it would have been optimal for our programme to be able to run on other platforms. For example, our cooperation with the Internet Technology and Computer Systems (ITC) group would have gone a lot better, if we had made our programme more compatible with other systems, such as being able to work on Android based phones.

Encoding in Different Images

Something that could make the encoding of messages more difficult to detect, would be to be able to encode the message in the middle of an image. As the programme currently is, the message is always encoded at the start of an image from left to right. In a lot of the pictures we tested, the top of the image typically was very bare, and the focus of the image, where there were more colours and shapes was in the centre. Because of this, it is a little easier to see that there is some form of distortion there. This change would make any encoded message a little more difficult to detect. An example of encoding images with different colours and forms can be seen in section 6.1 and with each image, we received different results. However as the four images in the aforementioned section, this comes down to what image is used, so a change in this, is not always necessarily a good idea.

Cryptography

Another improvement to the programme would be to add a layer of cryptography to the message before it gets embedded into the cover image. The strength of steganography is that, unlike cryptography, a hidden message does not garner the same level of attention. However, if a cover image is known to contain hidden information, then the whole concept of steganography is meaningless. Running a steganalysis and a subsequent reversal of the steganographic algorithm will reveal a message in plaintext. If there was a layer of encryption, the suspecting party would not be able to know if they uncovered an encrypted message, or if they simply used the wrong steganographic algorithm. This would provide an immense layer of security on top of steganography, and since each sharing party has already communicated with each other beforehand, this would not be a huge obstacle in terms of sharing keys.

Usability Testing

A process often used in software development is usability testing, where people from outside the group use the actual programme and describe their experiences. This is usually a good idea, since it gives direct knowledge about how real users will use the programme, instead of a team member who often assumes that a user is going to behave in a certain way. We decided not to do any usability testing, since the focus of our project was to implement an algorithm that would allow ITC to develop a user interface on top of it. In essence, our own user interface was a means of testing whether our algorithm behaved as expected, and not to allow real users to use the programme directly. But since our work with the cluster group did not entirely go as planned, and if we were to continue developing the programme, usability testing would definitely be required for different kinds of user interfaces.

Bibliography

Anderson, R. J. (1998). "On the Limits of Steganography". English. In: 16, pp. 474–481. ISSN: 0733-8716. DOI: 10.1109/49.668971.

Beaver, Doug et al. (2010). "Finding a Needle in Haystack: Facebook's Photo Storage". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI'10. Vancouver, BC, Canada: USENIX Association, pp. 1–8. URL: http://dl.acm.org/citation.cfm?id=1924943.1924947.

Böhme, Rainer and Andreas Westfeld (2004). "Computer Security – ESORICS 2004: 9th European Symposium on Research in Computer Security, Sophia Antipolis, France, September 13 - 15, 2004. Proceedings". In: ed. by Pierangela Samarati et al. Berlin, Heidelberg: Springer Berlin Heidelberg. Chap. Breaking Cauchy Model-Based JPEG Steganography with First Order Statistics, pp. 125–140. ISBN: 978-3-540-30108-0. DOI: 10.1007/978-3-540-30108-0_8.

Chetty, Marshini et al. (2012). "You're Capped". In: *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*. Association for Computing Machinery (ACM). DOI: 10.1145/2207676.2208714. URL: http://dx.doi.org/10.1145/2207676.2208714.

Christensen, Erik (2008). *PuffinCC*. URL: https://commons.wikimedia.org/wiki/File:Puffin.Mykines.jpg.

Curtis, Alison (2009). https://www.flickr.com/photos/alisonlongrigg/3784643116/in/photolist-6Lrhxw-hRzHTs-wjVDkr-hRzy73-hRzBdy-8wGn6R-skBR4Z-eaotnF-hRzLb1-97RRsS-8mxWjU-duzkc8-9zzu7C-qHiJuG-9YJw2D-dcZzYf-dcZy8i-9YJxQi-e4ws2L-6pufPi-BKda92-bWrwJN-djAP8Z-bgR3fc-6pypws-912AAD-5FaT6z-doaBKv-e4wrDy-9YMuvm-a9RJd4-bWrwqY-csqZfh-iZzMMJ-4smYLr-6pyptw-4SWsnK-9YJvm2-rszuKj-afLGEo-4YcrZE-bWrwjA-bWroGC-YxaRg-9YJxza-dm8oPk-78Qgho-5vyTMy-jaa8ej-g4mxHr.

Cuturicu, Cristi (1999). *CRYX's note about the JPEG decoding algorithm*. https://www.opennet.ru/docs/formats/jpeg.txt. Accessed May 23, 2016.

Czeskis, Alexei et al. (2008). "Defeating Encrypted and Deniable File Systems: TrueCrypt V5.1a and the Case of the Tattling OS and Applications". In: *Proceedings of the 3rd Conference on Hot Topics in Security*. HOTSEC'08. San Jose, CA: USENIX

Association, 7:1–7:7. URL: http://dl.acm.org/citation.cfm?id=1496671.1496678.

Eltantawy, Nahed and Julie Wiest (2011). "The Arab Spring| Social Media in the Egyptian Revolution: Reconsidering Resource Mobilization Theory". In: *International Journal of Communication* 5.0. ISSN: 1932-8036. URL: http://ijoc.org/index.php/ijoc/article/view/1242.

Gerald G. (2016). *Cat*. URL: http://www.freestockphotos.biz/stockphoto/10688.

Hamilton, Eric (1992). *JPEG File Interchange Format*. https://www.w3.org/Graphics/JPEG/jfif3.pdf.

Haslam, John (2007). https://www.flickr.com/photos/foxypar4/565452906/in/photolist-RY6s3-5juL9o-iVXQu-fh23oJ-9NbMRt-p3AsAj-63Rak6-nAbZv6-4KWc7y-dNfsNz-amQtab-bUK9cd-eLmNbc-4RQS2e-ogvEVw-7Ppa8-qifjGr-pP2bA5-djJec6-27HFJC-9ytcxJ-czE2VS-5K6811-63RaoZ-opfxrN-9BGZwJ-evueqj-5pMuWm-55EmK-599TDx-A71RSw-cq6EfA-bLSj6x-dfzUT4-5GBfQs-hE5uNT-swCKpG-aZE3HV-fUbcXN-5Qdp5x-4oN3ua-dZ3mae-dCJu7H-psa24Q-bLcG1z-qnisb2-khvgq8-bftwyK-4KWbUm-cjXQNo.

Hetzl, Stefan and Petra Mutzel (2005). "A Graph–Theoretic Approach to Steganography". In: *Communications and Multimedia Security*. Springer Science+Business Media, pp. 119–128. DOI: 10.1007/11552055_12. URL: http://dx.doi.org/10.1007/11552055_12.

Koshkina, Larisa (2016). *Snow*. URL: http://www.publicdomainpictures.net/view-image.php?large=1\&image=17839.

Li, Qiming, Yongdong Wu, and Feng Bao (2010). "A Reversible Data Hiding Scheme For JPEG". In:

Meeker, Mary (2014). "INTERNET TRENDS 2014-CODE CONFERENCE". In: *Glokalde* 1.3. URL: http://www.kpcb.com/file/kpcb-internet-trends-2014.

Miano, John (1999). *Compressed Image File Formats: JPEG, PNG, GIF, XBM, BMP*. ACM Press.

Moritzer, Wolfgang (2016). *Landscape*. URL: http://tumblr.unsplash.com/post/71804706827/download-by-wolfgang-moritzer.

MSDN. *BitmapData Class*. URL: https://msdn.microsoft.com/en-us/library/system.drawing.imaging.bitmapdata.aspx.

— *HOWTO: Copy Bitmap Regions*. URL: https://msdn.microsoft.com/en-us/library/aa457087.aspx.

Nazar, Imran (2013). *Let's build a JPEG decoder*. http://imrannazar.com/Let's-Build-a-JPEG-Decoder:-Huffman-Tables. Accessed: 04-20-2016.

Oliboni, Cosimo. *OpenPuff Manual*. http://embededsw.net/doc/OpenPuff_Help_EN.pdf.

Peoples, Glenn (2014). *T-Mobile to Make Customers' Data Free When Streaming Music*. Ed. by Billboard. Accessed: 2016-02-24. URL: http://www.billboard.com/

biz/articles/news/digital-and-mobile/6121645/t-mobile-to-make-customers-data-free-when-streaming.

Sawchuk, Alexander. *Volume 3: Miscellaneous Lenna Image*. http://sipi.usc.edu/database/database.php?volume=misc&image=12.

Shih, Frank Y. "Image Processing and Pattern Recognition: Fundamentals and Techniques". English. In:

Singh, Simon (2001). *Kodebogen : Videnskaben om Hemmelige Budskaber - fra oldtidens Ægypten til kvantekryptering*, pp. 19–20.

tiina93 (2009). https://www.flickr.com/photos/tiina93/3274624318/in/photolist-5ZniHY-5Zni6S-8qTG4k-6cBTLH-8wjaRr-77RXTc-5Zi3Wt-7T8hsP-9yZN48-h7n8mb-4p76dD-rQL7RN-8vXAa5-QJ4NT-96aT2f-qnB4i6-qHEGeE-reYMzZ-9z3L7m-oJ9ZmW-9z3ZZ3-4tHLYi-dvbKrw-7RshHy-fyTNaN-9mvpcC-9yZWfx-Etdqov-qkv56h-g7g4bu-ebUyKE-b1CqCF-o8ZzRk-eAh1r-3XZwCn-f3s5GH-8VUsmx-pr2qUB-dnBdhf-9z3WuY-hePhKB-6MAgRK-8gSzjF-bnVk8W-rnnX9-wRU3N-7onQF-9F5Ae8-8zvhUw-4UT7Qv.

Union, International Telecommunication (1992). *INFORMATION TECHNOLOGY — DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES — REQUIREMENTS AND GUIDELINES, Recommendation T.81*.

Vegetaveg (2015). *Tiger*. URL: https://commons.wikimedia.org/wiki/File:Winter-Tiger-Wild-Cat-Images.jpg.

W3Techs (2016). URL: http://w3techs.com/technologies/details/im-jpeg/all/all.

Wang, Liwei, Yan Zhang, and Jufu Feng (2005). "On the Euclidean distance of images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27.8, pp. 1334–1339. DOI: 10.1109/tpami.2005.165. URL: http://dx.doi.org/10.1109/tpami.2005.165.

Westfeld, Andreas and Andreas Pfitzmann (2000). "Attacks on Steganographic Systems". In: *Information Hiding: 5th International Workshop, IH'99, Dresden, Germany, September 29 - October 1, 1999 Proceedings*. Springer Science + Business Media, pp. 61–76. DOI: 10.1007/10719724_5.

Appendix A

LSB Experiment

This is a digital appendix containing source code for a C# programme. It is too large to fit into the report itself. Instead the relevant Visual Studio Solution can be found at the following location.
`Programmes\LSB_Experiment\LSB_Experiment.sln`

The programme in its executable form can be found at
`Programmes\LSB_Experiment\LSB_Experiment.exe`

Appendix B

Stegosaurus - BitList implementation

```
public class BitList : IEnumerable<bool> {
  private BitArray _bits;

  public int Count { get; private set; }

  public BitList(int length) {
    _bits = new BitArray(length);
    Count = length;
  }

  public BitList() {
    _bits = new BitArray(1);
    Count = 0;
  }

  public bool this[int i] {
    get {
      return _bits[i];
    }
    set {
      _bits[i] = value;
    }
  }

  public IEnumerator<bool> GetEnumerator() {
    for (int i = 0; i < Count; i++) {
      yield return _bits[i];
    }
  }
```

```

public void InsertAt(int index, bool value) {
  BitArray newArr = new BitArray(Count + 1);
  for (int i = 0; i < index; i++) {
    newArr[i] = this[i];
  }
  newArr[index] = value;
  for (int i = index; i < Count; i++) {
    newArr[i + 1] = this[i];
  }

  Count++;
  _bits = newArr;
}

public void Add(int val) {
  switch (val) {
    case 1:
      Add(true);
      break;
    case 0:
      Add(false);
      break;
    default:
      throw new ArgumentOutOfRangeException();
  }
}

public void Add(bool val) {
  if (Count == _bits.Length) {
    _bits.Length *= 2;
  }
  _bits[Count] = val;
  Count++;
}

private readonly BitArray _latestEntries = new BitArray(8);
private int _addCounter;

public void CheckedAdd(int val) {
  if (_addCounter % 8 == 0) {
    _latestEntries.SetAll(false);
  }
  _latestEntries[_addCounter % 8] = (val == 1);
  Add(val == 1);
  bool allOne = false;

```

```
    if (_addCounter % 8 == 7) {
      allOne = true;
      for (int i = 0; i < 8; i++) {
        if (!_latestEntries[i]) {
          allOne = false;
          break;
        }
      }
    }

    if (allOne) {
      for (int i = 0; i < 8; i++) {
        Add(false);
      }
    }

    _addCounter++;
  }

  IEnumerator IEnumerable.GetEnumerator() {
    return GetEnumerator();
  }
}
```

Appendix C

Stegosaurus

This is a digital appendix containing the source code of all the C# programmes contained in our Stegosaurus solution. Since this is far too large to fit into the report itself. Instead the relevant Visual Studio solution containing several projects can be found at the following location.
`Programmes\Stegosaurus\Stegosaurus.sln`

The solution contains the following projects

Stegosaurus is the class library in which we have implemented the graph theoretical and least significant method used by our GUI.

SteGUI is the graphical user interface we developed to showcase our use of interfaces.

StegosaurusTests contains all the unit tests we have used in the development of Stegosaurus.

Appendix D

Stegosaurus - Four different levels of optimisation

Each folder contains the same OptimisationTester.exe and a compiled version of Stegosaurus in dll format. Furthermore, the files which have been changed in each generation can be found within the *CS* folders.

```
Programmer\Optimeringer\Stegosaurus\unoptimized
Programmer\Optimeringer\Stegosaurus\first_round
Programmer\Optimeringer\Stegosaurus\second_round
Programmer\Optimeringer\Stegosaurus\third_round
```

Index