

Comparing Steganography Algorithms

Rosanne English

A dissertation presented in part fulfilment
of the requirements of the Degree of MSc
in Information Technology at The
University of Glasgow

September 2009

Acknowledgements

I would like to take this opportunity to thank my supervisor, Dr Ron Poet, for his time and assistance.

Abstract

The objective of this project was to implement several different steganography algorithms with an aim to provide a basis for comparison in terms of effectiveness and hiding capacity. Effectiveness of the algorithms was measured by user evaluations determining at which point alteration to the images became apparent.

The algorithms implemented were the least significant bit, two least significant bits, four least significant bits and the Bit Plane Complexity Segmentation (BPCS) algorithms. The most noteworthy algorithm implemented was the BPCS algorithm as devised by Kawaguchi et al (1998). This dissertation also discusses the implementation of this algorithm, which was not covered in the papers written by the authors of the technique.

The outcome of the project showed that the two lower capacity algorithms (least significant bit and two least significant bits) were highly effective. The four least significant bits algorithm, when tested to the maximum capacity was ineffective. The claim made by Kawaguchi et al (1998) that up to 50% of the image could be used for hiding information in the BPCS algorithm was also proved.

Acknowledgements	ii
Abstract	ii
Chapter 1 – Introduction	1
Chapter 2 – Survey of other Programs.....	3
2.1 Steghide.....	3
2.2 Vecna	5
2.3 OpenStego.....	6
2.4 Digital Invisible Ink Toolkit	7
2.5 Qtech Hide and View	9
Chapter 3 – The Algorithms	12
3.1 Least Significant Bit Algorithm.....	12
3.2 Two Least Significant Bits Algorithm.....	12
3.2 Four Least Significant Bits Algorithm.....	12
3.3 BPCS Steganography.....	13
Chapter 4 – Requirements Analysis.....	16
4.1 Requirements Capture.....	16
4.2 Functional Requirements	16
4.3 Non-Functional Requirements	17
Chapter 5 – Design.....	18
5.1 The Classes	19
5.2 Graphical User Interface Design.....	22
Chapter 6 – Implementation of the Least Significant Bit Algorithm.....	24
6.1 Bit Manipulation	24
6.2 Header Information	26
6.3 FileIO Implementation.....	27
6.4 Hiding Implementation	28
6.5 Extraction Implementation.....	28
Chapter 7 – Implementation of the Two Least Significant Bits & Four Least Significant Bits Algorithms	30
7.1 Two Least Significant Bit Hiding	30
7.2 Two Least Significant Bits Extraction	31
7.3 Four Least Significant Bits Algorithm Implementation	32
Chapter 8 – Implementation of the BPCS Algorithm.....	33
8.1 Image Manipulation	33
8.2 Decomposition into Bit Planes.....	34
8.3 Complexity Calculation	34
8.4 Segmentation of Bit Planes	36
8.5 Splitting of the Payload Information.....	37
8.6 Replacement of Segments and Bit Planes.....	37
8.7 The Hiding Implementation	38
8.8 The Extraction Implementation	39
Chapter 9 – GUI Evaluation	41
9.1 GUI Interim Evaluation	41
9.2 GUI Second Evaluation	43
9.3 GUI Final Evaluations and Review	44
10. Testing and Evaluation	45
10.1 Testing.....	45
10.2 Hiding Capacity Analysis	45
10.3 Effectiveness Analysis	49

10.4 Image Analysis.....	50
Chapter 11 –Future Developments	53
Chapter 12- Conclusion	55
Appendix A – Requirements Capture Diagrams	56
UseCase.....	56
Use Case Description.....	56
Activity Diagram	58
Appendix B- Design Documents	59
Package Diagram	59
Class Diagram	60
Appendix C- Evaluation of Other Steganography Software	71
Appendix D – Bit Plane Decomposition.....	74
Appendix E – Testing Results.....	81
JUnit Test Cases Summary	81
Black Box Testing Results.....	84
Appendix F – Original Images.....	87
Appendix G- Image Perception Evaluation Results	89
Appendix H – Perception Evaluation Images	91
Appendix I – List of Figures	95
Appendix I – Image Histograms	96
Bibliography	97

Chapter 1 – Introduction

The project proposed was to produce software written in Java which implemented a minimum of three steganography algorithms in order to conduct an analysis and comparison of their effectiveness. The following provides an introduction to the concepts of steganography and its background in order to place the problem in context.

The process of hiding information in another document such that the information hidden is not perceivable to the uninformed user is called ‘steganography’. The technique can be used within the area of digital security by providing an alternative to cryptography or as an additional layer of security. Traditional steganography included the use of invisible ink and pin punctures, where small pin pricks were placed on selected letters, visible only when held against a light. (Stallings, 1998)

Digital watermarking is closely related to steganography. Watermarking is the process of embedding hidden information which is only visible after a process has been carried out. An example is the watermarking of bank notes to prove legitimacy, whereupon holding the note to the light produces a visible image, the watermark. The main differences between watermarking and steganography is that in watermarking, the information hidden relates to the object it is hidden in. Coming back to the bank note example, a dashed line along the bank note is completed by the watermark. Also, it is possible that a watermark does not attempt to hide the fact that information is hidden, but steganography invariably does (Johnson, 1998). Thus, the distinction between the two techniques can be made.

Modern-day steganography techniques differ considerably from traditional practices. With the development of the computer, steganography has been adapted and there are now a variety of contemporary computerised techniques. The basic process of modern steganography uses a ‘cover’ file (a seemingly innocuous file) to hide the secret information. In this project, digital image files will be used as the cover file. The main reason for this being that they are sufficiently large to hide substantial files within them. Other reasons include that they are innocuous by nature and information can be replaced in an image with no perceivable effects. In order to provide a complete context, the concept of a digital image should be explained.

A digital image is an array of numbers which represent light intensities at those points; these are called “pixels”. These digital images are commonly stored in 24-bit or 8 bit per pixel files. (Johnson et al, 1998) An 8 bit file is a greyscale image and thus has only black and white pixels, represented by 8 bits per pixel. The colour variations in an image are obtained by mixing the three primary colours. A standard 24 bit image uses 8 bits (one byte) for each of red, green and blue in order to represent the colour of an individual pixel. There are also image files such as PNG (Portable Network Graphics) which have 32 bits per pixel, the extra 8 bits representing ‘transparency’.

The least significant bit of a byte is defined as the 0th bit, e.g. given a byte 10001010, the least significant bit is the right-most bit 0 which has been highlighted by italics.

One of the more common steganography algorithms is least significant bit hiding (Johnson et al, 1998). In this process, the least significant bits of each colour component in each pixel of an image are discarded. Instead, the space is used to hide the bits of the secret file. This allows for 3 bits of information to be stored in one pixel (Johnson et al, 1998). This is possible without detection by the human eye because the replacement of the least significant bit will only affect the image by +1 or -1 if at all. That is to say, the bit from the file to be hidden can be the same as the least significant bit of the image byte, thus there is no change. Alternatively, the least significant bit can be a 1 and the bit to hide could be a 0 and vice versa, hence the difference of +1 or -1.

The main deficit of least significant bit steganography is that it only allows up to 12.5% (1/8) of the size of the vessel image to be hidden (Rebah, K., 2004). The first part of this project was to implement the least significant bit algorithm and determine its effectiveness. The effectiveness was to be assessed through user studies to establish at which point, if any, the user can spot alterations to the image which make them appear abnormal. This was then to be extended to determine if this capacity could be increased to 25% through the use of not only the first least significant bit, but also the second least significant bit.

In order to demonstrate the effect of hiding too much information within an image, the ‘four least significant bits’ algorithm was also implemented. This algorithm is an extension of the least significant bit algorithm; it uses the four least significant bits of each byte to hide the file information.

The final algorithm implemented is a more recent algorithm called ‘Bit Plane Complexity Segmentation (BPCS) steganography’. The method claims to be able to hide 50% (Kawaguchi, 2008) of the size of the vessel image. This algorithm was implemented and the claim of 50% carry capacity was investigated with the results of this investigation detailed in chapter 10. The main concept of BPCS is that of ‘bit planes’ and segments. In this approach a 24 bit image has 24 bit planes, one for each bit of each pixel. A segment is an 8x8 block of pixels from within a bit plane, these segments consist of 64 bits. The technique exploits attributes of human sight that rapidly changing bit patterns are not discernable (Kawaguchi, 1998). The result of this is that if any bit plane segment is complex enough, it can be swapped with complex information from the payload file.

Despite not being initially obvious to the casual observer that steganography has taken place, it is still susceptible to attack. The term for such an attack is ‘steganalysis’. The primary objective is to detect the presence of hidden data, with an occasional supplementary objective being to extract the hidden data. (Wang, 2004) However, investigation of this was outwith the scope of the project. It should also be noted, that this project was not intended to implement algorithms which would resist steganalysis.

Chapter 2 – Survey of other Programs

From the survey of other open source steganography products, it appeared that the most commonly implemented algorithm was the least significant bit algorithm. There was only one other program implementing BPCS steganography found, which was a by-product of the corresponding paper (Kawaguchi, Eiji, 1998).

In order to examine steganography programs which were already available, it was necessary to first decide the criterion against which they would be graded and ultimately against which the program resulting from this project would also be graded.

The criteria were decided to be as follows:

- Accessibility: The product must be easily obtainable and easy to install on different platforms
- Usability- The product must be easy to use
- Robustness- The product should not break easily
- Effectiveness – the result of the hiding should not be discernable to the naked eye
- Hiding Capacity – the program should be able to hide files equating to a significant proportion of the image size. For this purpose, a figure of 12.5% was determined as minimum (the carrying capacity of the least significant bit algorithm)

Several products were assessed against these criteria in order to better understand the process of steganography, and requirements of a computer program implementing such algorithms. The programs listed below are assessed individually under each measure.

2.1 Steghide

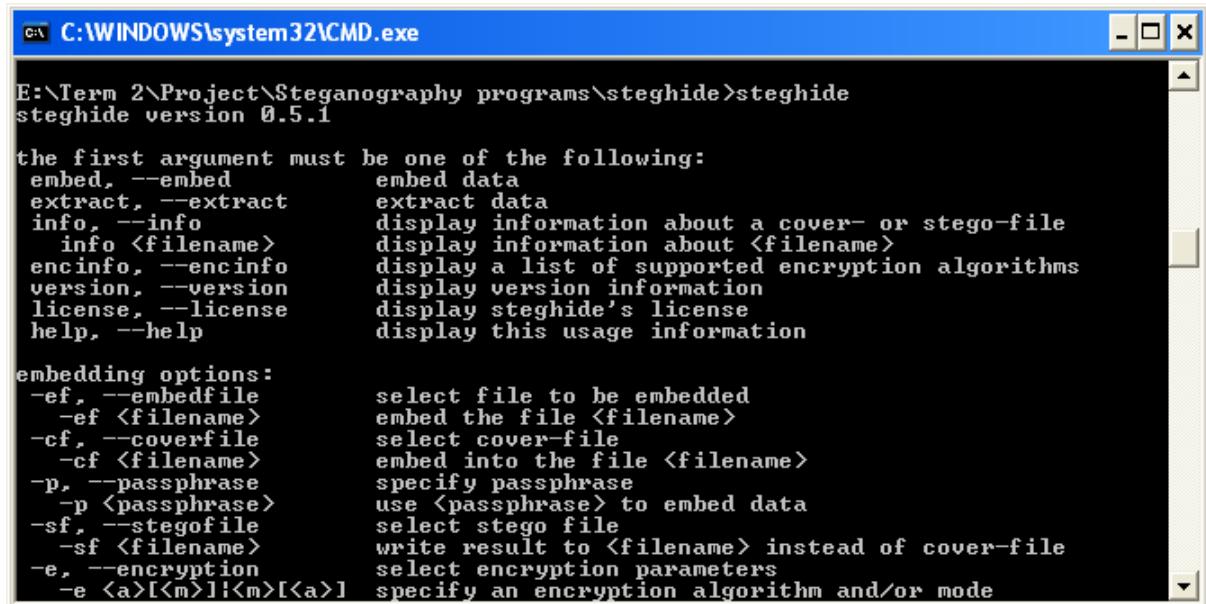
Steghide is an open source steganography program written in C++. It provides functionality of encoding the data hidden in the picture chosen and implements an algorithm based on graph theory. It can be run on both Windows and Linux operating systems.

2.1.1 Accessibility

The program was easily downloaded. However, installation was not straight forward as it had to be completed through command line prompts, thus, for a more casual user accessibility was somewhat diminished through this.

2.1.2 Usability:

As for the installation, the running of the program was completed through command line prompts. This adversely affected the usability of the program as tasks were performed through the entry of specified strings. The user interface can be seen in figure 2.1.1. The interactivity of the program was more difficult as commands were moved off screen if more operations were carried out.



The screenshot shows a Windows Command Prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The command entered is 'E:\Term 2\Project\Steganography programs\steghide>steghide'. The output displays the usage information for the 'steghide' command, which includes options for embedding, extracting, and displaying information about cover or stego files, as well as encryption algorithms and license information. The 'embedding options' section lists various flags such as -ef, -cf, -p, -sf, -e, and their descriptions.

```
E:\Term 2\Project\Steganography programs\steghide>steghide
steghide version 0.5.1

the first argument must be one of the following:
  embed, --embed      embed data
  extract, --extract  extract data
  info, --info        display information about a cover- or stego-file
  info <filename>    display information about <filename>
  encinfo, --encinfo   display a list of supported encryption algorithms
  version, --version   display version information
  license, --license   display steghide's license
  help, --help        display this usage information

embedding options:
  -ef, --embedfile    select file to be embedded
  -ef <filename>     embed the file <filename>
  -cf, --coverfile    select cover-file
  -cf <filename>     embed into the file <filename>
  -p, --passphrase    specify passphrase
  -p <passphrase>    use <passphrase> to embed data
  -sf, --stegofile    select stego file
  -sf <filename>     write result to <filename> instead of cover-file
  -e, --encryption    select encryption parameters
  -e <a>[<m>]<m>[<a>] specify an encryption algorithm and/or mode
```

Figure 2.1.1

2.1.3 Robustness:

No errors occurred during the use of this software, and although this was not considered to be extensive testing, it did bode well since other software had problems under the same level of scrutiny.

2.1.4 Effectiveness:

The program was deemed effective; this can be seen through the results provided in Appendix B.

2.1.5 Hiding Capacity:

5% of the image file was determined as the average capacity through tests detailed in Appendix B.

2.2 Vecna

Vecna is a program written in Java which implements the least significant bit algorithm. It allows for any type of file to be embedded into an image file.

2.2.1 Accessibility

The program, being written in Java was platform independent. It was easily downloaded and installed on a machine running Windows.

2.2.2 Usability

The program implemented a GUI which was simple and easily used. A screenshot of the GUI is provided in figure 2.2.1. There were no problems of either an ancillary gulf or a gulf of execution when aiming to hide and extract files.

However, when attempting to select a cover image, the program reported back a required number of pixels needed to cover the file. It would have been more helpful to receive this feedback as a file size required.

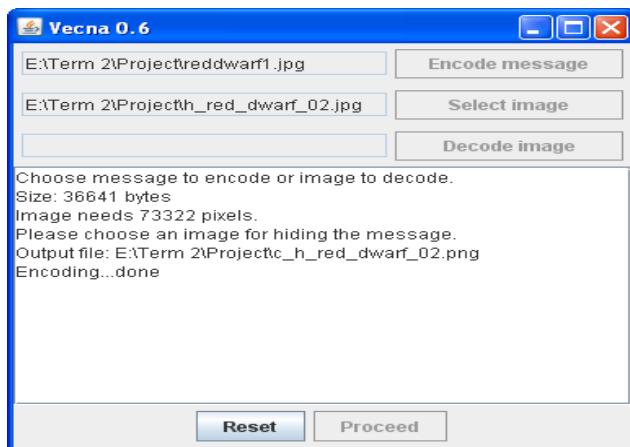


Figure 2.2.1

2.2.3 Robustness

This program proved to be one of the less robust of those examined. There are several known problems with the program (Strauss, 2008) and there were also problems encountered when trying to use the program.

2.2.4 Effectiveness

The program was deemed effective; this can be seen through the results provided in Appendix B.

2.2.5 Hiding Capacity

There were no reported hiding capacities for this program since the selection of the file to be hidden was completed before the selection of the cover image. It is then reported back to the user in terms of number of pixels, what size of image was required to cover the selected file. However, since the algorithm used was the least significant bit algorithm, it is known that a maximum hiding capacity of 12.5% was achieved. The reason this was a maximum is because header information is possibly stored within the image.

2.3 OpenStego

OpenStego is a Java steganography program. The algorithms it implements are least significant bit and random least significant bit insertion (where the embedded data is stored in random places, not sequentially). A particular strength of this program is that it allows for the selection of which algorithm to use.

2.3.1 Accessibility

The program was easily run as it provided a .bat file. A .bat file is a command line batch file which takes the onus off the user to use the command line to run the program. Being written in Java also meant it was platform independent and could be run in different environments.

2.3.2 Usability

A GUI was implemented which made carrying out the tasks required much more convenient than e.g. command line operations. The interface was easy to use, consistent and provided informative feedback. A screenshot of the GUI is provided in figure 2.3.1

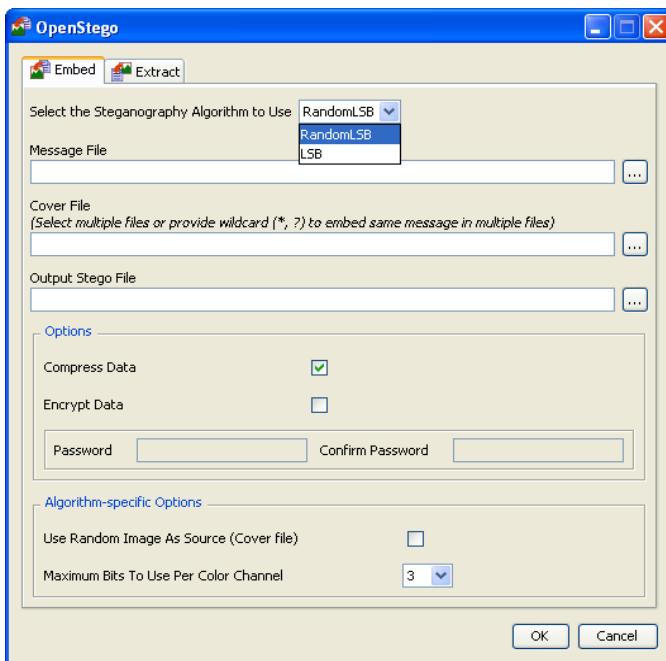


Figure 2.3.1

2.3.3 Robustness

The major weakness of the program was that only image files could be hidden. When attempting to hide any other kind of file, an error message was thrown indicating that there was an ‘Out of Memory’ error. This is demonstrated in figure 2.3.2



Figure 2.3.2

2.3.4 Effectiveness

The program was deemed effective; this can be seen through the results provided in Appendix B.

2.3.5 Hiding Capacity

Since the program did not supply information on the GUI regarding the size of the file required to cover a secret file, one can only deduce (due to the use of the least significant bit algorithm) a capacity of roughly 12.5% is achieved (1/8 of the file size).

2.4 Digital Invisible Ink Toolkit

This program is written in Java and allows the use of image files as cover files for embedding and extracting. It provided the user with 6 possible algorithm choices for embedding the data. These were; BattleSteg, BlindHide, DynamicBattleSteg, DynamicFilterFirst, FilterFirst and HideSeek. A brief description of each is provided below:

BattleSteg:

This algorithm first analyses the image to acquire a list of the most appropriate places to hide the secret information. The algorithm then randomly selects places to hide until it finds one of the places in the list. It then embeds information there and in the surrounding pixels before starting again.

DynamicBattleSteg:

Similar to “BattleSteg” but uses dynamic programming to reduce the amount of memory used in the process.

BlindHide:

This algorithm starts embedding the secret data at pixel (0,0) and scans along the pixels embedding as it goes.

FilterFirst:

This algorithm uses “edge detecting filters” to acquire an ordered list of the optimal places to embed. It then starts to embed in the order detailed.

DynamicFilterFirst:

Similar to FilterFirst, but implements “dynamic programming” to reduce the size of memory used.

HideSeek:

This method randomly selects a bit to hide data in and continues until all the secret data is hidden.

2.4.1 Accessibility

A JAR file was used to run the program, which made it easily accessible. In addition, being written in Java meant it was platform independent.

2.4.2 Usability

The GUI was very easily navigated and used. Everything was labelled and displayed well. A screenshot of the GUI is provided in figure 2.4.1

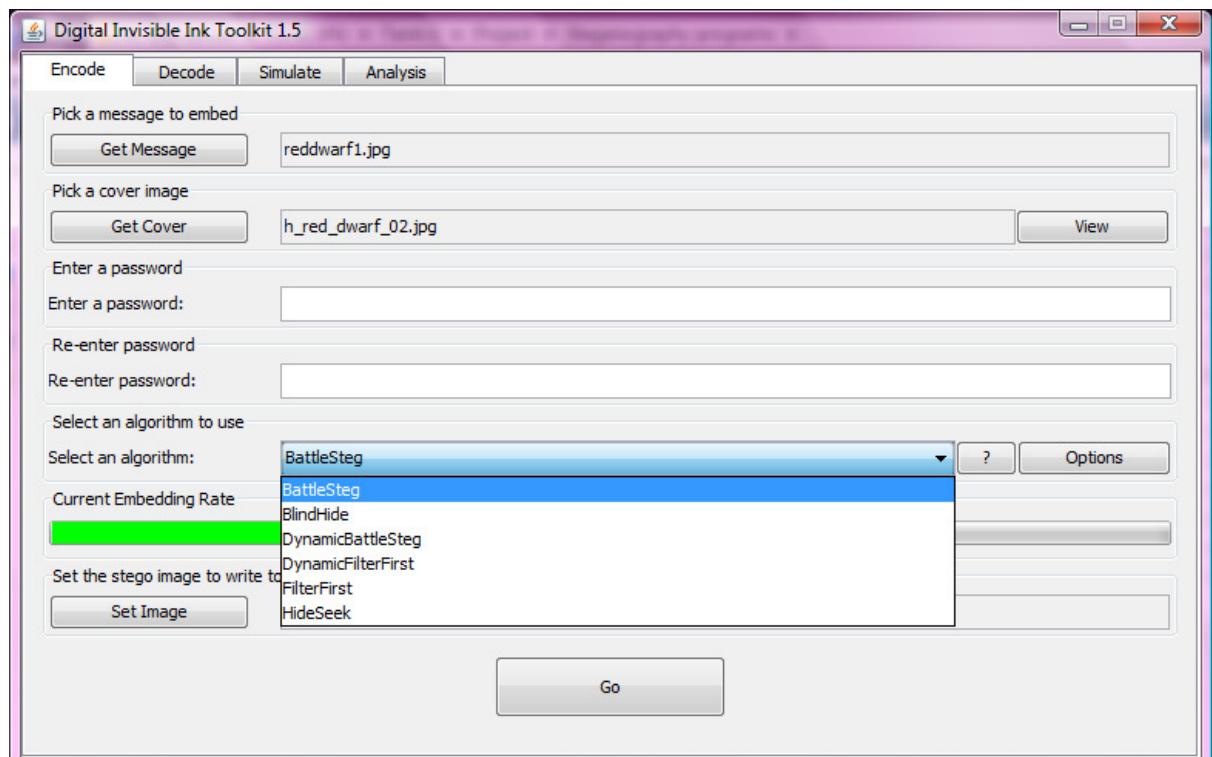


Figure 2.4.1

2.4.3 Robustness

Quite often, if selecting a larger file, an ‘Out of Memory’ error was thrown. This is shown in figure 2.4.2



Figure 2.4.2

This caused problems as larger images could not be used and hence larger files could not be hidden.

2.4.4 Effectiveness

Due to error messages, as shown in figure 2.4.3, it was not possible to test the effectiveness in this instance.



Figure 2.4.3

2.4.5 Hiding Capacity

The program displayed an embedding rate bar which showed the user in a graphical manner how much of the storage capacity of an image would be used by embedding the chosen file. However, it was not clear the capacity (in terms of a percentage) which the different algorithms achieve. One would estimate a maximum of 12.5% due to the basis of the algorithms being variations on the least significant bit algorithm.

2.5 Qtech Hide and View

Qtech is the only program available which implements the BPCS steganography algorithm. It was written in conjunction with the paper on BPCS steganography (Kawajuki et al 1998). The platform for QTech is Windows (written in Visual C++). However, use of the full program is limited to 90 days, and the source code is not available. The program is available for download from ‘datahide.com/BPCSe/QtechHV-program-e’ (Kawaguchi,2009, QTech).

2.5.1 Accessibility

The program was installed with relative ease. However, some characters on the screen prompts were unintelligible, presumably due to the program being developed in Japan. These were as shown in figure 2.5.1

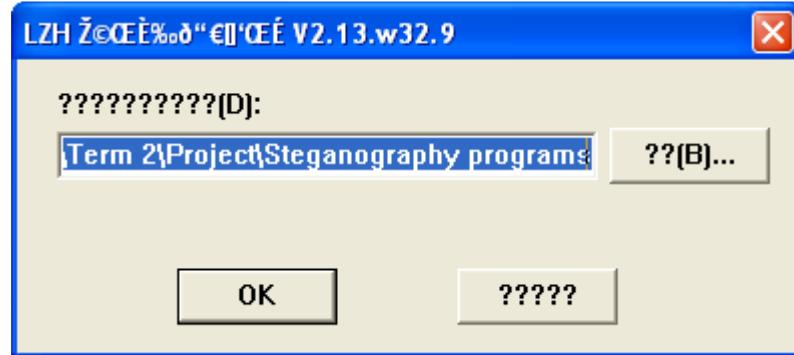


Figure 2.5.1

2.5.2 Usability

Once the program was installed, a simple effective GUI was provided, which made using the program straight forward. It provided the function to select the format of the output (JPEG, BMP or PNG) and also provided a function to embed an individual file or a folder. A screenshot of the GUI is shown in figure 2.5.2

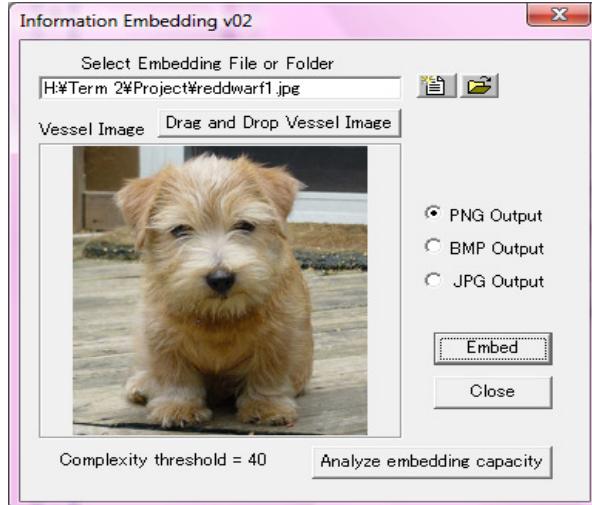


Figure 2.5.2

2.5.3 Robustness

Upon using the program, no error messages were thrown. Thus, although testing was not full and extensive, the program appeared to have a high level of robustness.

2.5.4 Effectiveness

The program was deemed effective. The results of the evaluation are provided in Appendix B.

2.5.5 Hiding Capacity

As reported in the corresponding paper (Kawajuki et al, 1998), the hiding capacity was up to 50% of the cover image.

Chapter 3 – The Algorithms

3.1 Least Significant Bit Algorithm

The least significant bit approach was based on a simple concept, the right most bit (termed the least significant bit) was replaced with a bit from the file data to be hidden (the payload data). This meant that the least significant bits of the cover file were changed to match the bits of the payload.

With regards to using images in particular as the cover file, the choice of image and the format of the image were important (Kipper, 2004). The images used were inconspicuous, thus not drawing attention to the fact that they could be concealing information. It was also attempted as far as possible to reduce large blocks of one colour. If an image with such blocks were used, the change of one bit could have resulted in obvious distortion of the image. In addition to this, the format of the image was important. The use of a ‘lossy’ file format such as JPEG could have meant that on reconstruction of the embedded file, bits could be lost due to the compression (hence the term “lossy”). On the other hand, the use of ‘lossless’ images (specifically bmp files) allowed for the hidden file to be reconstructed completely. There was no loss of information, hence the term “lossless” (Johnson et al, 1998). However, the use of such an algorithm was limited as it allowed for up to only 12.5% (1/8) of the size of the image to be hidden.

3.2 Two Least Significant Bits Algorithm

The use of the least significant bit algorithm was extended to increase hiding capacity by using not only the first least significant bit of each byte, but the second in order to hide information. E.g. given a byte *1 0 1 1 0 1 1 0* the two least significant bits are the right most bits of the byte as indicated by the italics. The result of this was that twice the capacity of the least significant bit algorithm could be obtained (a maximum of 25% of the cover image).

The possible problem with the two least significant bits algorithm was that since the second least significant bit was also changed to match bits of the secret file, there was a chance that the result would be noticeable to the human eye. In order to investigate this, user evaluations were carried out. The results of these evaluations were as detailed in Chapter 10 and also in Appendix E.

3.2 Four Least Significant Bits Algorithm

This algorithm was an extension of the least significant bit algorithm (in the same manner as the two least significant bits algorithm). It used the four least significant bits of each byte to hide information, thus it had a maximum capacity of 50%. The purpose of implementing this

algorithm was to demonstrate the results when too much information was hidden. The resulting images then provided a benchmark against those in which information had been successfully hidden.

3.3 BPCS Steganography

The first step in the BPCS (Bit Plane Complexity Segmentation) steganography was splitting the image into ‘bit planes’. Each bit plane was a binary image which contained the i^{th} bit of each pixel where i was the plane number. In the example of a 24 bit image, the 0^{th} bit plane (the first bit plane, by convention) was the least significant, and the 23^{rd} bit plane was the most significant.

Each pixel was constructed from 3 bytes (3 lots of 8 bits) due to the use of 24 bit bitmap images. Each byte corresponded to one each of the colours red, green and blue. The 1^{st} bit plane contained the least significant bit of each pixel (the blue least significant bit for each pixel). The second bit plane contained all the second least significant bits and so on until the final bit plane was reached, which was the most significant bit plane, containing all the most significant bits (the most significant bit of the red byte for each pixel). A bit plane was a complete one-bit per pixel image. The decomposition of an image into bit planes resulted in lower planes which were ‘noisy’ and had no apparent significant information in terms of human perception. On the other hand, the higher planes had a simple structure, making them key to human sensitivity of the image (Kamata et al, 1995). Figure 3.3.1 provides a visual representation of how a 24 bit image can be decomposed into its composite planes.

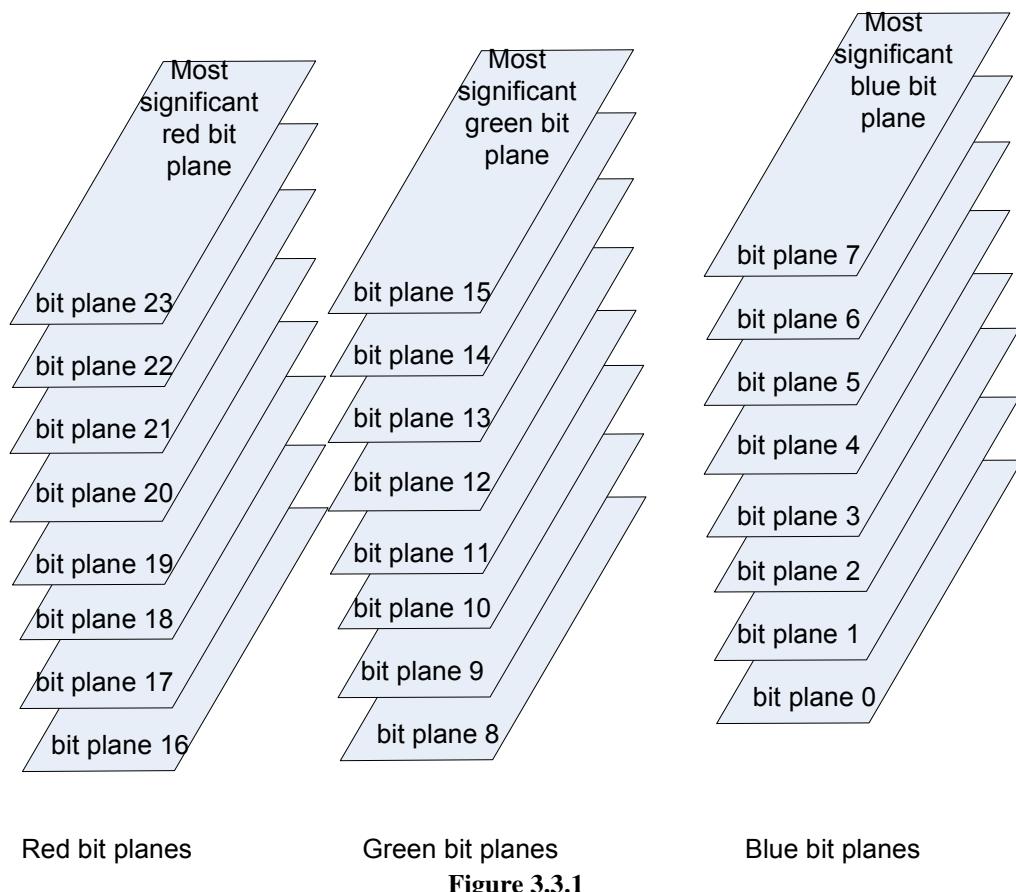


Figure 3.3.1

A demonstration of the result of splitting an image into bit planes can be seen in Appendix D

The BPCS technique relied on an attribute of human vision, which was that the human eye is unable to discern a difference between two rapidly changing binary patterns (Nozaki et al, 1998). The result of this was that one could change specific segments (8x8 pixel regions) of a digital image and store data equating to 50% of the size of the image file without detection by the human eye. (Kawaguchi, 1998)

Thus, the first step in implementing a BPCS steganography program was automating the deconstruction of an image into its bit planes. Once this was completed, analysis of the bit planes was carried out. The purpose of this analysis was to determine which 8x8 pixel regions (segments) of each bit plane were ‘complex’ and thus could have ‘noisy’ information hidden in them since the human eye is unable to discern a difference between two rapidly changing binary patterns (Nozaki et al, 1998).

The complexity level of each segment of the plane was determined by the use of a ‘black and white border’ measure (Kawaguchi et al, 1998). The border complexity was as described below:

Given a binary image, the border length is the sum of the number of colour changes along the rows and columns in an image. In the example shown in figure 3.3.1 the border length is 4.

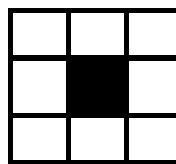


Figure 3.3.2

The changes are: in the middle column from white to black then from black to white and in the middle row there is a change from white to black, then black to white. This totals 4 changes, thus the border length of 4.

The complexity measure is then defined for an image with a $2^m \times 2^m$ frame as the border complexity of the image, divided by the maximum border complexity of an image of size $2^m \times 2^m$. The resulting complexity ranges between 0 and 1. (Nimi et al, 1997)

Each bit-plane was split into 8x8 regions (segments) in order to carry out a calculation of the complexity of these areas. (using the example above, a $2^3 \times 2^3$ frame) These local segments of a bit-plane were grouped into 3 categories; natural informative sections, artificial informative portions and noise-like portions.

The original data was either naturally informative (with complexity less than the chosen threshold value) or noise-like (with a complexity greater than the threshold value). This threshold was used to define a ‘noisy’ region and hence also simple regions in all parts of the algorithm. A threshold of 0.3 was determined as optimum by Kawaguchi et al (1998) and this was implemented in the project. Noisy regions were replaced with noisy information from the file to be hidden (payload information). If the complexity of the payload data did not reach the chosen threshold, it was categorised as ‘artificially informative’. As a result, a conjugation

operation was performed on the payload data making it noisy in order to make extraction possible. The conjugation method was defined by Kawaguchi et al (1998) as follows:

Given a $2^n \times 2^n$ size binary image with black as the foreground area and white as the background area, then there are corresponding all white and all black patterns of the same size. In addition, there are two checkerboard patterns. One has a white pixel in the upper-left location, and the other has the upper-left pixel as black.

The given binary image is understood by the following. Pixels in the foreground area have the all black pattern, while pixels in the background area have the all white pattern. Then the conjugate image of the given image is defined by the following attributes:

- The foreground area shape is the same as the shape of the original image
- The foreground area has the checkerboard pattern with black in the top left-most position.
- The background area has the checkerboard pattern with white in the top left-most position.
- The complexity of the conjugate image is then 1- complexity of P.

(Kawaguchi et al, 1998)

It was now possible to embed data as it was suitably complex such that the insertion of this data would not result in noticeable alteration to the image.

Each block of the secret data to be hidden was embedded in the noise-like regions of the bit-planes. If the block had been conjugated to ensure complexity, it was required to store the fact that it had been conjugated in a ‘conjugation map’ which was then stored in the bit plane. The conjugation map detailed the “location of the simple pattern which has had the conjugate operation applied”. (Niimi et al, 1999) Despite the description of the purpose of a conjugation map, the actual specifics of how to implement such a concept was not discussed in the paper by Kawaguchi et al (1998). It was therefore necessary to devise how best to do this and consider exactly what the ‘conjugation map’ was.

The embedding itself was initially carried out on the lower level bit-planes progressing to the higher level bit planes; this was because the lower planes were less significant (Niimi et al, 1999). The selection of which planes should be embedded in first was chosen as the 0th plane through to the 23rd bit plane. This was decided at the start of the project since there was no discussion of which planes should be used, and in which order within the defining paper (Kawaguchi et al, 1998). It was later discovered that a more optimal solution was to use the least significant plane of each colour (red, green and blue) before progressing to the more significant planes of each colour. This is discussed further in section 8.7.

The extraction of the information which had been embedded was the reverse of the hiding method.

Chapter 4 – Requirements Analysis

4.1 Requirements Capture

Due to the precise specifications of the problem from the outset, requirements capture was limited. The aim of the project was to produce a program which implemented the hiding and extracting of files within images using 3 algorithms; least significant bit, second least significant bit and BPCS steganography. A fourth was implemented in addition in order to demonstrate the effect of hiding too much information.

There was no client in the traditional sense; hence requirements capture techniques such as interviews and observation were not feasible or appropriate. However, the requirements were derived from the statement of the problem. Requirements were gathered through a series of initial meetings with supervisor Ron Poet who assisted in the initial refinement of the problem during the proposal stage of the project. The following functional and non functional requirements were captured as a result of these discussions along with background reading and review of other similar software. In some cases these were refined due to responses as a result of implementation or design.

It should be noted that UML was used to represent the requirements as well as parts of the design and analysis phase. The resulting diagrams (specifically the use case diagram and description along with the activity diagram) are provided in Appendix A.

4.2 Functional Requirements

On a very high level, the requirements of the program were that the program allowed the user to hide and extract a file within an image.

4.2.1 Hiding

The decision of how large a file can be hidden in any given image should be automated, that is to say it is not down to the user to determine at which point the image reaches maximum capacity.

The restriction of selection of a cover image to specific image formats (e.g. .bmp, .png) should be implemented by the program and thus the hiding operation should not commence if an inappropriate cover file is chosen (e.g. a non-image file or a lossy format image such as jpg)

4.2.2 Extracting

The onus should not be upon the user to be aware of the type of file hidden in the image, thus the details of the type of file hidden should be catered for within the program.

The onus should not be on the program to determine which algorithm was used to hide the file within the image upon extraction. The user should be able to inform the program of which algorithm was used, and it is acceptable for the process to fail if the wrong algorithm is selected.

It is acceptable that, if user selects a cover file which has nothing hidden in it the extraction process will fail.

4.3 Non-Functional Requirements

In addition to the functional requirements detailed in section 4.2, there was a small selection of non-functional requirements, these are listed below:

The system should be

- Usable
- Reliable
- Developed in Java
- Accessible

Chapter 5 – Design

The following chapter aims to justify the choice of classes and their interaction with each other by detailing the purpose of each class and the reasoning behind the selection of each. In order to determine the classes required, the problem was analysed in depth. It became apparent that the main areas of the problem were to be able to hide and extract using each algorithm, the reading and writing of images and files and user interaction with the program. The classes selected reflect this and a detailed class diagram is provided in Appendix B. A class structure diagram is shown in figure 5.1. In order to improve design and maintainability of code, the classes were split into several packages. This can be seen through the package diagram in Appendix B. The packages followed a two tier system consisting of the Interaction Subsystem (the GUI) and the Application Subsystem (comprising of the file manipulation, image manipulation and steganography logic classes). It should be noted that packages within this context differ from the package structure in Java. However, this structure should also be noted in the design as it was intended that the packages could be slotted into another program in the same manner as the Java packages such as IO and Swing are commonly used.

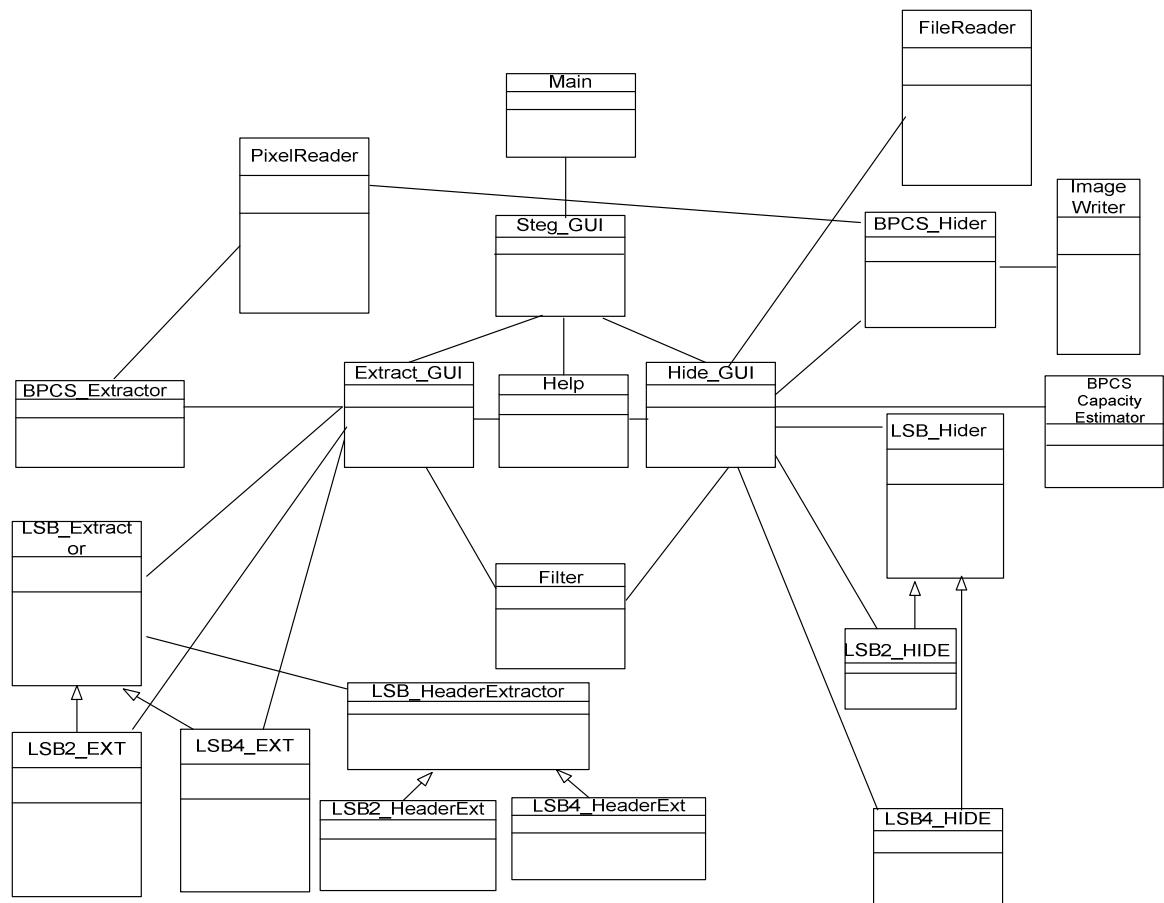


Figure 5.1

5.1 The Classes

5.1.1 File Reader

The purpose of the file reader class was to abstract the reading in of the file to be hidden from the ‘Hiding’ classes. This class read in the selected file and split it into bytes and subsequently bits which were accessed from outwith the class from each of the four classes which implemented hiding algorithms. This allowed the file access to be separate from the algorithm using it, improving code reuse and separation of concern. After further investigation, and as a result of attempting to implement the first hiding algorithm, it became apparent that in order to be able to extract successfully a certain amount of added information would need to be hidden in the file before the file bits themselves. As a result, the file reader class had a method called `getNextBit()`. This method was called from each of the extraction classes to access the next bit of information to be hidden, this included the header information populated by the class.

5.1.2 Filter

This class was created in order to restrict choice of files to selected image types when using a JFileChooser. The reason for this was to reduce validation of the selection of an image. If it had not been designed as such, there would have been a need for a check to ensure the file selected was indeed an image, and that it was of a valid type. The types were restricted to .bmp (for hiding or extracting using any least significant bit algorithm, or hiding using bpcs) and .png (for extraction using the bpcs algorithm).

5.1.3 Image Writer

The purpose of this class was to facilitate the writing out of the image from an array of pixels. It was determined that this would be needed when implementing the BPCS algorithm since the algorithm involved manipulation of the pixels through bit plane replacement. Thus, the image writer class was used in the ‘BPCS hider’ class.

5.1.4 Pixel Reader

This class was the complement of the image writer class, its purpose was to read in an image and store it as an array of pixels in order to aid manipulation in the BPCS algorithm. This class played a main role in the ‘BPCS hider’ as it was designed to hold the two crucial methods relating to image manipulation which allowed the program to access and replace a specified bit plane.

5.1.5 Main

The ‘Main’ class was necessary to initiate the GUI and thus provided the launch pad for the program functionality.

5.1.6 BPCS Capacity Estimator

The purpose of this class was to provide an estimate for the capacity of a specified image file. An alternative would have been to iterate through the image to determine the exact complexity and hence capacity, however this was not achieved within the time frame. The option to do this is discussed further in ‘Future Developments’. The class allowed a check to be performed before operations were carried out and deemed unsuccessful, warning the user of insufficient covering space for the file selected before carrying out a potentially intensive operation of hiding which would be unsuccessful.

5.1.7 BPCS Extractor

This class was designed to implement the reverse of the hiding algorithm for the BPCS Steganography approach to allow for extraction. It was called from the extraction GUI when the file had been hidden using the BPCS algorithm.

5.1.8 BPCS Hider

This class was designed to implement the BPCS hiding algorithm and was called from the hiding GUI when the user selected to hide their chosen file with the BPCS algorithm. This class made use of the PixelReader and ImageWriter classes in order to obtain, manipulate and subsequently write out the image once more according to the BPCS algorithm.

5.1.9 LSB Extractor

This class was designed to implement the least significant bit extraction algorithm. It was called from the extraction GUI when the file had been hidden using the LSB algorithm. The design of this class was such that a superclass was deemed appropriate in order to allow easy addition of the second least significant bit and four least significant bit algorithms.

5.1.10 LSB Header Extractor

In order to reduce the workload on the extractor class, another superclass for the least significant bit extraction process was designed in the form of the ‘LSB Header Extractor’ class. This class allowed a ‘header extractor’ object to be created in the main extractor class and accessor methods allowed access to the header information before the bits relating to the file itself are extracted.

5.1.11 LSB Hider

The class ‘LSB Hider’ was designed to implement the least significant bit algorithm. It was called from the hiding GUI when the user selected to hide using this algorithm.

5.1.12 LSB2 Ext

This class was subclass of the LSB Extractor class and implemented the extraction algorithm for the second least significant bit method. This essentially did the same as the least significant bit method, but extracts an extra bit.

5.1.13 LSB2 Header Extractor

This was a subclass of the LSB Header Extractor class and allowed the extraction of the header information when two least significant bits had been used to hide the information. It had the same methods as the header extractor, but with an extra method which allowed it to extract the second least significant bit as well as the first.

5.1.14 LSB2 Hide

This class was a subclass of the LSB Hiding class. It implemented the majority of the methods of the superclass, and added additional steps to allow the hiding of two bits of information for each byte.

5.1.15 LSB4 Ext

In a similar vein to the LSB2 Extraction class this class was a subclass of the LSB Extraction class. The additional method and overridden sections enhanced the LSB by extracting the four least significant bits instead of just one.

5.1.16 LSB4 Header Ext

This class was another subclass of the LSB Header Extractor which allowed header information to be extracted when it had been hidden using four least significant bits.

5.1.17 LSB4 Hide

This was a subclass of the least significant bit hiding class which extended the class by allowing four least significant bits to be used instead of one for the hiding of the header information and the bits of the file to be hidden.

5.1.18 Plane

The purpose of the plane class was to represent the concept of a plane in a way which would allow it to be altered as required by the BPCS algorithm. Since an image was constructed from bit planes, the replace bit plane and get bit plane methods were within the PixelReader class. The get bit plane method returned an object of type ‘Plane’ and the replace bit plane had a parameter of type ‘Plane’. Since the BPCS method’s pivotal points were concerning the segmentation and complexity, the plane class was designed to implement the calculation of

complexity and also to implement the segmentation through methods such as analysePlane() and getNextSegment().

5.2 Graphical User Interface Design

When selecting a user interface design, it was decided that a WIMP style of interaction be used. This was in order to facilitate the ease of use since the people using this may not have extensive knowledge of computers or steganography. It was also to assist in learning to use the program through association with similar interfaces. The use of a graphical interface (GUI) was most appropriate for this purpose. The initial mock up of a possible GUI was as in figure 5.2.1

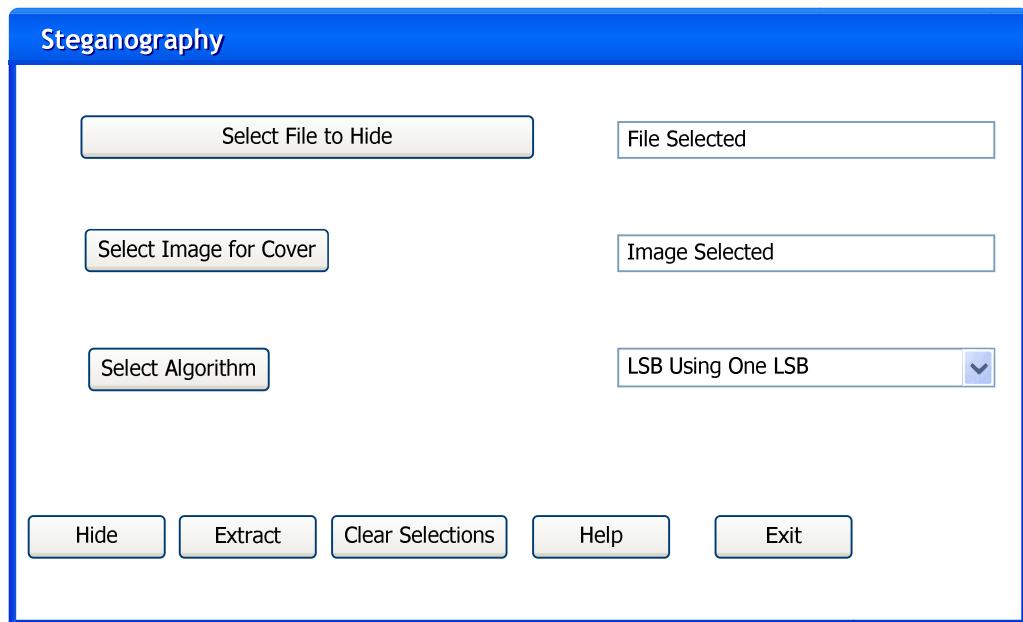


Figure 5.2.1

However, an initial prototype was created using this design and user evaluation resulted in a redesign. The result of the review was that there was confusion over which tasks were required to carry out hiding, and which for extraction. From here, a redesign meant that separate screens were introduced for each of the tasks, with a third being the initial screen, allowing the user to select which task they wish to carry out, hiding or extraction. In addition to this, it was decided that a 'Help' section should be introduced in order to allow the user further information on how to carry out tasks as required. The design for this included selection of an area on which more detail is required. The areas most important to increasing comprehension of the program and its functionality were an introductory area, an area detailing how to hide a file, a corresponding area for extraction and one on the different algorithms which could be used within the program.

Once all the functionality had been implemented, the GUI design was tweaked to reflect this and maximise usability. The final design is shown in figures 5.2.2, 5.2.3 and 5.2.4

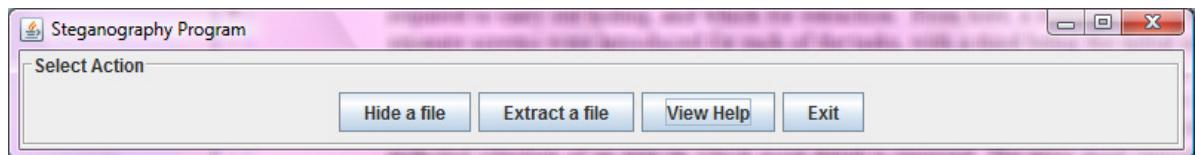


Figure 5.2.2

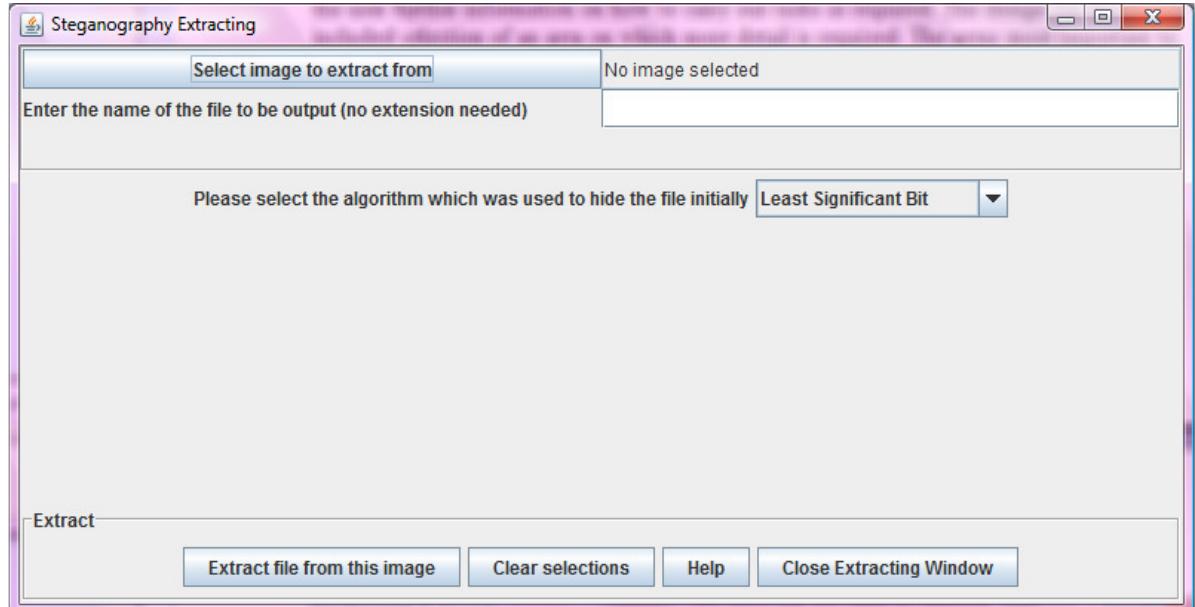


Figure 5.2.3

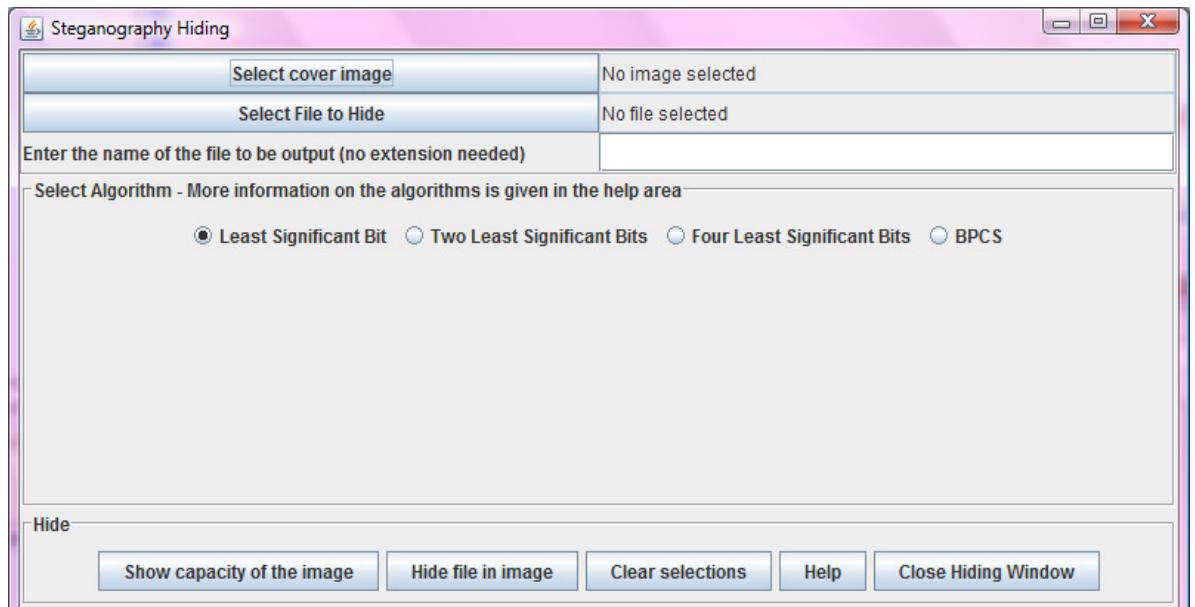


Figure 5.2.4

Chapter 6 – Implementation of the Least Significant Bit Algorithm

6.1 Bit Manipulation

The key concept to be grasped when approaching this project was that of bit manipulation. This was quintessential since the alteration of bits from the image file allowed bits of the payload (the file to be hidden) to be hidden and subsequently extracted. In order manipulate bits in Java, the following bit operations were used: bitwise and (`&`), or (`|`), shift of the bits to the right (`>>`), shift of bits to the left (`<<`), the complement operation (`~`) and finally the exclusive or operation (`^`). The following paragraphs aim to provide a brief description of the results of these operations and for convenience, provide logic tables. To facilitate understanding, it should be noted that a bitwise operation is the result of performing an operation on the first pair of bits of two bytes, and each corresponding pair of bits of the byte till the last pair has been reached.

6.1.1 Bitwise AND

The result of the bitwise and is the result of `&` each pair of bits. 1 if both inputs are 1, and otherwise the result is a 0.

input x	input y	output
1	0	0
0	1	0
0	0	0
1	1	1

So, e.g. $10110111 \& 11111110 = 10110110$

6.1.2 Bitwise OR

For each corresponding pair of bits, if either one input or the other is 1 then the result of the OR operation is 1, otherwise it is 0.

input x	input y	output
1	0	1
0	1	1
0	0	0
1	1	1

e.g. the result of $10110100 \mid 01000100 = 11110100$

6.1.3 Complement (\sim)

When applied to a set of bits, the complement operation swaps any 0's to 1's and vice versa.
E.g given a byte 10110101, the result of the complement operation is 01001010.

input x	output
1	0
0	1

6.1.4 Shift Operations (>> and <<)

The result of the left shift operation on a row of bits is that every bit is shifted to the left by the value of the operand on the right hand side of the shift, and a 0 is inserted in place of the right most bit, e.g. given $01011101 \ll 1$ gives 10111010

Similarly, the right shift operation shifts each bit to the right by the value of the right hand side operand, and places 0s in the positions on the left.

Given $01011101 \gg 1$ gives 00101110

6.1.5 How to Access the Least Significant Bit of a Given Byte

A bitmask is a pattern of 1s and 0s which is combined with a byte using the AND operation. This results in a 1 in the required position (where the bitmask is set to 1) and a 0 elsewhere. '0x' is conventional notation for a hexadecimal number and precedes the bitmask. It indicates to the compiler that what follows is hexadecimal. In order to obtain the least significant bit of a byte the AND operation is used with the 'bitmask' 0x1. This ensures the last bit is the only bit which can be 1 (if the last bit of the byte before the & with the bitmask is 1) and every other bit is replaced with 0 (since 1 or 0 & 0 is 0). The result of this now gives the least significant bit. E.g. 10110111 becomes 00000001

6.1.6 How to get the second least significant bit

The method here is similar to that for the least significant bit, but this time the bit mask used is 0x2 meaning the only bit which can be 1 is the second least significant bit. In addition to this, the result needs to be shifted to the right by 1 in order to obtain this bit.

So e.g. to get the second least significant bit of 10110110

Do $10110110 \& 00000010 = 00000010$

Then $\gg 1$ gives 00000001 as required.

6.1.7 How to change a bit from a 1 to a 0 and vice versa

In order to hide the bits of the file, the least significant bit (and later the second, third and fourth least significant bit) of each byte of the cover file is replaced with a bit from the file to be hidden. In order to do this, when the next bit of the file doesn't match the bit of the cover file to be replaced, then it should be swapped to match. Thus, if one is trying to hide a 0 and the least significant bit is a 1, the bit to be replaced needs to be changed to a 1 and vice versa. In order to complete this, once more bit manipulation is implemented.

Given a byte which ends in a 1, in order to change to a 0 one must logical and the byte with the complement of 0x1 (called the bitmask)

Thus the operation is expressed as: $\&= \sim 0x1$

To change a 0 to a 1: $\mid= 0x1$

6.1.8 Splitting a byte into its bits and vice versa

In order to implement the hiding of bits, it was necessary to split each byte of the file to be hidden into bits to hide them and then to implement the extraction, the bits needed to be reformed into bytes.

In order to split a byte into bits, it was necessary to access each bit individually. In this case, the bitmask used was $0x2^i$ where i was the i^{th} bit, e.g. bit 0 (the least significant bit) used bit mask 0x1, bit 1 (the second least significant bit) used bit mask 0x2 and so on. The result of this was then shifted to the right by i in order to obtain the required bit. This was a generalisation of accessing the least significant bit and the second least significant bit.

In order to reconstruct the file from the least significant bits of the bytes of the image with the file hidden in it, it was necessary to be able to reconstruct bytes from a list of bits. This was done through a loop which iterated through each of the 8 bits. In order to construct a byte from this, the bit was shifted to the left by i (where i indicates the number of iterations through the loop) and OR'd with the current contents of the variable which contains the byte being constructed. This effectively takes each bit as it is read, shifting it to the left and concatenating the next bit until all 8 bits are added to form the complete byte as required.

6.2 Header Information

The second step was to determine how the program would know when to stop extracting information. That is to say, when the information extracted stopped being the payload. This was overcome by the decision to implement 'header' information before the payload relating to the file to be hidden, thus effectively producing a new payload which consisted of the header information and the file information. The header details were selected to be the size of the file hidden (in terms of the bits stored within the image) and also the extension of the file which

was hidden. The size being hidden allowed the program to extract exactly the correct amount of bits from the file. The extension allowed the correct type of file to be written out upon extraction.

The header information was populated in the FileReader class which had methods which accessed the size and extension of the file, then converted this into bits representing them. The decision was made to use 32 bits to represent the size of the file and 64 to represent the extension. 32 bits for the size allowed for a maximum size of 512MB to be represented which was deemed sufficient for the scope of the project. This was sufficient since the digital images used in the course of the project had a maximum size of 14.4MB (taken using a digital camera), thus an image of size large enough to hide 512MB seemed infeasible. 64 bits allowed for 8 characters (8 bits per character) to be stored as the extension. Normal extensions being .txt, .doc, .mp3, .ppt with the more recent extensions for windows being e.g. .docx. Again, this was felt to leave enough scope for the majority of common file extensions. In order to split the bytes up, bit manipulation was used to shift each byte to the right and obtain the last bit for each bit in the byte in turn. These bits were then ready to be accessed and hidden.

6.3 FileIO Implementation

The next step in implementing the least significant bit algorithm was to access each bit as required in order to hide them within the image. The first bits to be accessed were the bits relating to the header information. This was a key decision, as it allowed for the extraction to be completed successfully. This was so that the size and extension could be extracted before the file bits. By counting the number of bits read in, the program then used the bits extracted to reconstruct the size, which allowed it to stop when the count reached this number. It also meant that the bytes could be written out with the correct extension, allowing them to create the correct format of file.

The next bits to be accessed were the bits pertaining to the file itself. To obtain the bytes of a file the FileReader class was designed. The key to this class was that it implemented FileInputStream objects to read in a file byte by byte, thus allowing the manipulation of each byte as required.

Actual access to the bits of both the header information and the file bits was implemented through ‘hasNext()’ and ‘getNext()’ methods, effectively providing an iterator for access to the bits to be hidden. After accessing the bits of the header information, which were held in lists since they were small and easily iterated through, the bytes of the file were accessed through the FileInputStream and bit manipulation was used to then split these bytes up into bits. The getNext method then retrieved the appropriate bit and returned it to the class calling the method.

The FileReader class was used no matter which algorithm was implemented, and thus payload bit access was independent of the algorithm being implemented. Upon extracting files, it was determined that the complement to FileInputStream would be used, FileOutputStream allowed bytes to be written out to a file.

6.4 Hiding Implementation

Now that all the ground work was in place, it was possible to combine the separate sections to implement the hiding algorithm. The basis of what was required to do this was to first access each byte of the image. This was completed through the use of the `FileInputStream` method as it allowed access to each byte as it was read in. However, this introduced a new problem.

Image files contain information at the start of the file detailing essentially that they are an image (http://www.fastgraph.com/help/image_file_header_formats.html). Thus, this information had to be ‘skipped’ over when altering the bits. The reason for this was that if these bits were altered, the image written out would not open. An error message detailing that it was incorrectly formatted was raised. Alternatively, the image would be written out, but was drastically altered in terms of colour and in some cases, shape.

In order to overcome this problem, it was decided that a size of 54 bytes (http://www.fastgraph.com/help/image_file_header_formats.html) would be skipped over as this is the standard for bitmap images (bmp). Due to this, there was a heavy reliance on bmps in the least significant bit algorithms. Bmp images must be used as the cover image and bmp images are written out as a result of hiding. This is something which could be adapted in the future to determine the type of image file being used, and adjust the header size to be skipped over accordingly. However, due to the time constraints, it was not possible to alter this before completion of the project. This possibility is discussed further in Chapter 11- ‘Future Developments’.

Once the first 54 bytes were skipped over, it was possible to manipulate the bytes to reflect the bits of the image since this has no effect on the formatting of the image file itself. The plan was to access the least significant bit of each byte as it was read in, and perform the swapping methods (see 6.1.7) to ensure it matched the next bit from the `FileReader` (accessed through the `nextBit()` method, see 6.3) . The byte was then written back out to another file, using the `FileOutputStream` class provided by Java. Once all the bits from the `FileReader` had been hidden, the remaining bytes of the image were written out with no alteration, allowing the image to be completed.

In terms of putting the functionality behind the user interface for the least significant bit algorithm, a calculation of the capacity of the image file selected by the user was performed in order to ensure there would be sufficient space to hide the selected payload before allowing the user to proceed with the hiding process. The calculation took into consideration the header of size 54 specific to bmp formatting as well as the header information added in to ensure successful extraction and the number of bytes required to store the bits of information from the payload.

6.5 Extraction Implementation

Once more, the `FileInputStream` class provided by Java was used to read in the cover file with the payload hidden in it. In using this method, the first 54 bytes were skipped over, since it was known that these related to the formatting of the image and thus had no information hidden in them.

The next step was to extract the header information which had been placed in the hiding process to allow for successful extraction. In order to keep code clean and precise, a separate class was implemented to carry out this extraction.

The extraction was a converse of the hiding method in which the least significant bit of each byte (beyond the image header and before the file data) was accessed and added to a list, depending on the number of bits extracted, e.g. in the case of it being within the limit of the file size, the bits were added to a list consisting of all the bits relating to the size. The case was similar for the extension. Once this was completed, methods were written which allowed the lists of bits to be reformed into the bytes which made up the extension (see 6.1.8) and one which reformed the size from the list of bits. Both of these were carried out through bit manipulation. Accessor methods were then implemented in order to allow the main extraction class to access the size and extension before extraction of the payload bits.

The payload bits were extracted in the main ‘LSB_Extractor’ class. This class created a header extractor object which was then used to populate the key variables holding the size and extension of the file hidden. Due to this information having been populated, it was possible to set up a FileOutputStream to write out the bytes which would be formed from the bits extracted from the image, and it was also possible to control when the extracting stopped due to the size being known.

From here, the bits were extracted by accessing the next byte from the FileInputStream and extracting the least significant bit using the bit manipulation described in section 6.1.5. These bits were added to a list which, once fully populated, a method called makeBytes() was used to reform the bits into bytes which could be written out using the FileOutputStream. The bytes were constructed from the bits using the process described in section 6.1.8.

Chapter 7 – Implementation of the Two Least Significant Bits & Four Least Significant Bits Algorithms

The concept of the two least significant bit algorithm was a natural extension of the least significant bit algorithm. The difference being that instead of using one bit to hide payload information, two bits were used. The two bits of each byte were the furthest to the right, considered the least significant e.g. given a byte `10111010` the `1` and `0` to the far right (highlighted by italics) were the least significant and thus were replaced with bits from the payload.

Since the algorithm was an extension of the least significant bit algorithm, it seemed most appropriate to implement a superclass for the least significant bit classes and have the two least significant bits (and later the four least significant bits classes) as subclasses of these, thus only altering classes which would be affected by using two least significant bits instead of one.

The main alterations to the least significant bit algorithm in order to extend it were as detailed in the sections below.

7.1 Two Least Significant Bit Hiding

The two least significant bit hiding algorithm altered the hide method itself and thus the hide file method. The hide method performed the reading in of the image through the `FileInputStream` and determined at which point the payload information was hidden. The payload hiding is performed by the `hideFile()` method which assists the `hide()` method. Thus, this method was also overridden in the subclass to extend it for two least significant bit hiding.

There was one additional method required to implement this algorithm, one that swapped the second least significant bit of a byte with a bit passed in. This was in a similar vein to the least significant bit swapping method used in the superclass. The bit manipulation involved obtaining the second least significant bit (as described in 6.1.6), if the next bit (obtained through the `getNextBit()` method from `FileReader`) matched this bit, then no manipulation was carried out. The alteration if they did not match was similar to that described in section 6.1.7 (swapping a bit from a `1` to `0` and vice versa) with the difference that since the second least significant bit was to be altered, the bitmask changed. The bitmask was altered to be `0x2` to allow access to the second least significant bit, not the first.

Thus, given a byte which ended in a 1, in order to change to a 0 one had to perform a bitwise AND the byte with the complement of 0x2 (the bitmask)

This operation was expressed as: `&= ~0x2`

On the other hand, to change a bit from the image which is a 0 to a 1 to reflect the payload, the operation was the result of a bitwise OR with the bitmask 0x2. This was expressed as: `|=0x2`

The limits when hiding the information also differed, since the amount of space required to hide the payload was cut in half.

7.2 Two Least Significant Bits Extraction

As with the hiding implementation, the extraction was also a natural progression from the least significant bit extraction, as such the extraction classes were designed as subclasses of the related classes in the least significant bit implementation. The sections which follow aim to highlight the key areas which differed in the subclasses when compared with their superclass.

7.2.1 Two Least Significant Bits Header Information Extraction

The extract() method which performed the main logic of extraction (which did so by counting where in the file the bits are being extracted from) needed to be altered to reflect the reduction in space which was used during hiding. The number of bytes which had information extracted from them was split in half, since two bits for every byte was used, doubling capacity.

The methods to extract the bits relating to the extension and also the size also changed, these were extended to allow the extraction of the two least significant bits to be added to the relevant list, as was the case with the superclass albeit with one bit per byte. This was done through the bit manipulation operation described in section 6.1.6, that is the byte has a logical `&` performed with the bitmask 0x2 before shifting the resulting byte one to the right to access the bit required. The populateFileSize() method also needed modification due to the implementation relying on a count, the populateFileSize() method was called at a point in which the two least significant bits also needed to be added to the list of extension bits in order to populate this correctly.

7.2.2 Two Least Significant Bits File Extraction

Similar to the extraction of the header information, the limits for determining what to do with the two bits extracted needed to be halved due to the increased capacity of the two least significant bits hiding algorithm.

The method to populate the header information, which consisted of creating a Header Extractor object was modified to reflect that the header extractor should now be a two least significant bit header extractor, and not one least significant bit.

Finally the `getFileBits()` method was altered to extract two bits instead of one as in the superclass.

7.3 Four Least Significant Bits Algorithm Implementation

In order to demonstrate the result of using too much of the cover image to hide information, the ‘four least significant bits’ algorithm was implemented. This algorithm used the four rightmost bits to hide payload bits. E.g. given a byte `10110101` the four bits highlighted by italic lettering are deemed the four least significant. As with the two least significant bit implementation, the four least significant bits algorithm was implemented by subclasses of the least significant bit hiding and extraction classes.

The methods described above which had to be altered to reflect the second least significant bit also needed to be altered in a similar fashion to reflect four least significant bits.

The notable differences being the limits being quartered instead of halved and the addition of methods to access the third and fourth least significant bits (for extraction) and to swap the third and fourth least significant bits (in hiding).

These methods followed a similar pattern to the least significant and second least significant bits. The bitmasks were altered to reflect the bit required, in the case of the third least significant bit the bitmask was `0x4` and in the case of the fourth least significant bit was `0x8`.

Chapter 8 – Implementation of the BPCS Algorithm

There were several key steps which had to be carried out in order to implement the algorithm in its entirety. As stated by Kawaguchi, Eason (Principle and Applications of BPCS Steganography) the algorithm was:

1. Transform the dummy image from PBC (Pure Binary Code) to CGC (Canonical Gray Code) system.
2. Segment each bit-plane of the dummy image into informative and noise-like regions by using a threshold value (α_0). A typical value is $\alpha_0 = 0.3$.
3. Group the bytes of the secret file into a series of secret blocks.
4. If a block (S) is less complex than the threshold (α_0), then conjugate it to make it a more complex block (S^*). The conjugated block must be more complex than α_0 .
5. Embed each secret block into the noise-like regions of the bit-planes (or, replace all the noise-like regions with a series of secret blocks). If the block is conjugated, then record this fact in a “conjugation map.”
6. Also embed the conjugation map as was done with the secret blocks.
7. Convert the embedded dummy image from CGC back to PBC.

(Kawaguchi et al, 1998)

Due to the time constraints on the project, it was decided from the outset that the transformation from ‘Pure Binary Code’ (PBC) to ‘Canonical Gray Code’ (CGC) would not be implemented. This is discussed in Chapter 11 – ‘Future Developments’.

The processes carried out in the implementation are discussed at length in the sections below.

8.1 Image Manipulation

In the implementation of BPCS steganography, the first task was to ensure that the image could be manipulated as required since the BPCS algorithm made more use of the format of an image than the least significant bit algorithms. That is to say it was more based on the pixels and splitting the image into areas (segments) and bit planes. In order to ensure the necessary operations could be carried out, the ‘Image Manipulation’ package was created. This package included two classes, one called ‘PixelReader’ which read in an image and represented it as an object which could be manipulated as required. The other class deemed necessary was the ‘ImageWriter’ class which allowed for the array of pixels obtained from the PixelReader class (after manipulation) to be written out once more as an image.

The PixelReader class made notable use of two Java classes, the ImageIO class and the PixelGrabber class. The ImageIO class enables a ‘File’ object (which is an image) to be read in and stored as a ‘BufferedImage’ object. This then fed into the use of the PixelGrabber class,

which allowed the pixels to be ‘grabbed’ from the `BufferedImage` object and stored in an array of integers. Each integer within the array was a pixel from the image. The result of using these two classes set class in a position which allowed it to carry out the necessary operations to implement the BPCS algorithm.

The `ImageWriter` class used `ImageIO` to write out a `BufferedImage` object which was constructed from an array of pixels. The `ImageIO` wrote out a `.png` file, but this could be altered to write out a different image type. Thus, this class was used once all image manipulation had been carried out on the `PixelReader` class.

8.2 Decomposition into Bit Planes

The manipulation of the image commenced with the decomposition of the image into bit planes. There was a key assumption in the implementation that a 24-bit image would be used to hide information, since it obtains exactly 24 bit planes at various points within the implementation. Thus, the program has been written for use with 24-bit images, this is discussed further in Chapter 11 – Future Developments.

In order to represent a bit plane, the class ‘Plane’ was created. This class represented a bit plane through a 2D array of integers. Each entry in the array was one bit. Thus, the result was a binary image as expected. The use of a 2D array was deemed as the most appropriate to represent the plane since it represented the height and width as well as the concept of ‘dots’ to create the picture. It was also deemed necessary as later sections of the algorithm required the plane to be split into segments (8x8 regions of pixels) and also required replacement of these segments and the use of a 2D array made these tasks more accessible.

The process of accessing a bit plane was implemented in the `PixelReader` class. The method `getBitPlane (int i)` returned a plane object representing the i^{th} bit plane. It created a new 1D array with the same length as the array of pixels for the image, since the bitplane had this many entries (one bit per pixel). The method then iterated through each pixel of the image, and accessed the i^{th} bit of each pixel. This was then stored in the 1D array of bits until each pixel had been used. The result was a 1D array of all the relevant bits corresponding to the bit plane. This was then passed into a constructor for a `Plane` object along with the height and width of the image it related to.

Bit planes could therefore be accessed as needed by using the correct integer relating to the bitplane to be obtained.

8.3 Complexity Calculation

In order to segment a bit plane to allow for hiding of the payload information, a complexity calculation had to be carried out. This section discusses the implementation of the border complexity calculation as defined by Kawaguchi, Eason (1998).

The definition of the complexity (α , α) was:

$$\alpha = \text{border} / \text{maximum changes}$$

Thus, the first task was to calculate the border. The border was defined as

$$\text{border} = \text{number of row changes} + \text{number of column changes}$$

In terms of the border, a change was classified as a change from a 1 to a 0 or vice versa within the segment. Thus, as mentioned previously, given the image in figure 8.3.1 the border is of size 4 (2 changes in the columns, and two in the rows)

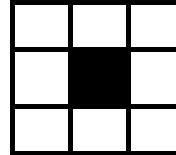


Figure 8.3.1

This calculation was automated by two methods. One of the methods iterated through the rows, counting the number of changes by storing the ‘previous’ value and incrementing a count if the next bit was different from the ‘previous’ value. The other performed a similar task, but iterated through the columns counting the changes. The results of these methods were then summed to give the value for the border.

The next step was to calculate the maximum number of changes possible for the plane (or segment). This was completed by a method called ‘getMaxChanges’. The method first calculated the maximum number of column changes by asserting that if the number of rows was exactly divisible by 2, then the maximum number of column changes was the number of rows divided by 2 multiplied by the number of columns. If however, the number of columns was not exactly divisible by 2, then the result was that the maximum number of column changes was the number of rows divided by 2 plus one, since there was the possibility of one more change, again this was multiplied by the number of columns to get the total maximum column changes.

A similar calculation was completed for the maximum row changes, asserting if the number of columns was exactly divisible by 2 then the maximum number of row changes was the number of columns divided by 2 multiplied by the number of rows. If however, the number of columns was not exactly divisible by 2, then the result was that the maximum number of row changes is the number of columns divided by 2 plus one, since there was the possibility of one more change, again this was multiplied by the number of rows to get the total maximum row changes. The overall maximum number of changes was then the sum of these two figures. The maximum complexity for an 8x8 segment is therefore 64. This is demonstrated in figure 8.3.2 where the number of row changes is 4, multiplied by 8 (one for each row) and similarly for the columns. The sum of these two figures gives the maximum number of changes of 64.

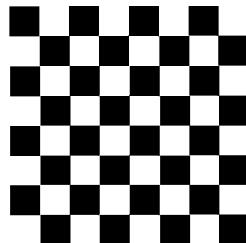


Figure 8.3.2

The complexity was then calculated by inserting the resulting figures into the complexity formula

$$\alpha = \text{border} / \text{maximum changes}$$

It can be seen from this, in order for a segment to be complex (the segment has $\alpha > 0.3$), the number of changes must have been greater than 19. This calculation was then used to categorise the segments of a bit plane to determine if a segment is complex enough to hide information in it, and also to determine if the information to be hidden was complex enough such that it would not cause noticeable alterations to the image if hidden.

8.4 Segmentation of Bit Planes

As previously mentioned, each bit plane was split into segments. The most efficient way to do this was deemed to be accessing each segment as required. In order to do this, within the Plane class two global variables were set up. One of the variables was a placeholder for the position in the plane of the current segment row position, the other was a placeholder for the current segment column position. These were then be incremented to allow traversal of the plane from left to right, through each row until the plane had been completely traversed.

In the segmentation process, it was decided that if the number of rows was not exactly divisible by 8 (the height of a segment) then any excess rows were ignored. In a similar fashion, if the number of columns was not exactly divisible by 8 (the length of a segment) then any excess was ignored. This is demonstrated in the example provided in figure 8.4.1. In this diagram, one can see that the area which can be used for segments is highlighted by a dotted pattern; the excess in terms of rows and columns is shown in white. These areas were ignored as they would not constitute a full 8x8 segment.

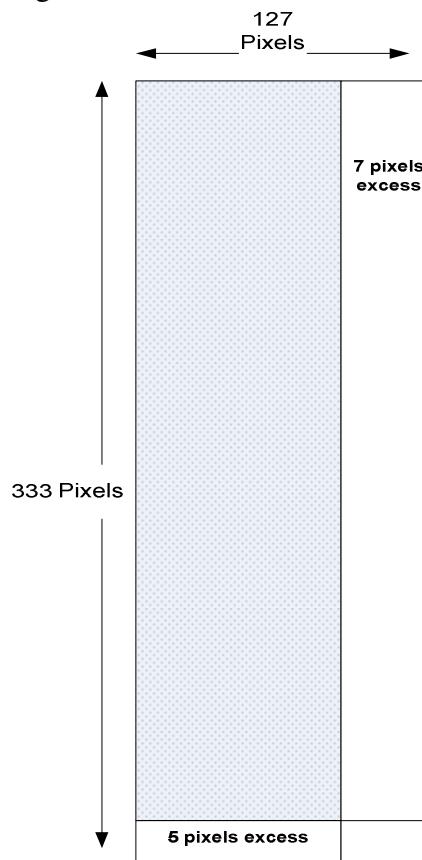


Figure 8.4.1

The actual incrementing of the position indicators was carried out in the `getNextSegment` method of the `Plane` class. It should be noted that this was always be preceded by the check ‘`hasNextSegment`’ which determined if there was another segment, either to the right of the current segment or beneath it. The traversal went from left to right and top to bottom. The `getNextSegment` method then incremented the positions by 8 (as appropriate) to move the counters onto the start of the now current segment, and effectively copied the 8x8 region into a new `Plane` object which is returned by the method. As a result, the complexity for the segment could now be accessed to determine if it was ‘noisy’ enough to hide information within the segment.

8.5 Splitting of the Payload Information

In order to hide payload information (which was accessed from the `FileReader` object by the methods `hasNextBit` and `getNextBit`) it had to be formatted into 8x8 regions which would replace the noisy segments of the bit plane. These regions were called ‘blocks’ and were represented by the `Plane` class in order. This was to allow their complexity to be examined to ensure they are complex enough such that they would not appear artificially informative.

Splitting the payload information into ‘blocks’ was carried out by the `FileReader` class. For this purpose, the class had two methods, `hasNextBlock` and `getNextBlock`. The `hasNextBlock` method returned a boolean value indicating whether there was another block of payload information to be hidden. This was then followed by the method ‘`getNextBlock`’ which returned a `Plane` object consisting of a block of payload information.

In order for ease of hiding and extraction, the blocks of information were populated left to right and top to bottom using the `hasNextBit` and `getNextBit` methods within the same class. It was decided that the first bit should be the conjugation map. This bit indicated whether the information in the block had been conjugated to make it more complex and thus suitable for hiding. As a default, upon creation of a block the conjugation map is 0 to denote no conjugation. This was altered at the point when the block is conjugated (if it was conjugated at all). The remaining 63 bits were the relevant bits of the payload, consisting of the header information and the bits relating to the file to be hidden. It was decided that if there was less than 63 bits to form a full block, the remainder of the block was padded out with 0s.

8.6 Replacement of Segments and Bit Planes

In order to implement the hiding BPCS algorithm, it was necessary to be able to replace segments of a bit plane with blocks of information from the payload, and subsequently replace the bit plane with the new plane holding the blocks of payload data.

The segments were replaced using a method ‘`replaceSegment`’ within the `Plane` class. This method took as parameters the block which will replace the segment and the position of the segment to be replaced in terms of its starting row and column positions within the plane. Thus, before accessing each segment to determine if it was complex enough, the row and column

starting position were accessed through accessor methods in order to be able to replace the segment if it was deemed complex enough.

The replaceSegment method then iterated through the segment within the plane replacing bits of the segment with corresponding bits of the block. This is demonstrated in figure 8.6.1 which shows how an 8x8 block is mapped into an 8x8 segment by highlighting that each bit from the block replaces the corresponding bit on the segment e.g. the (1,2) bit in the block replaces the (1,2) bit in the segment.

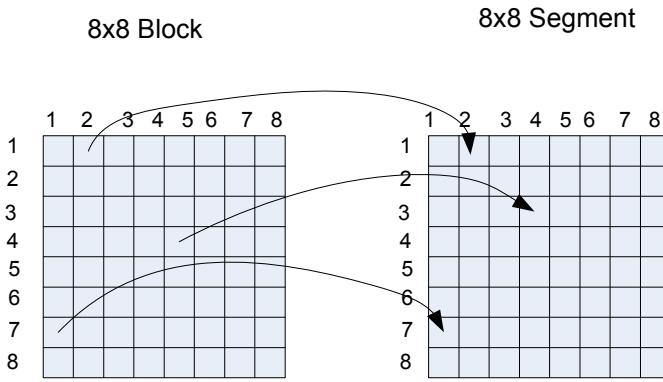


Figure 8.6.1

Once all the replacements had been made for the segments within the plane, this new plane replaced the old one so it could be written out with the information hidden in it. In order to do this, a method ‘replaceBitPlane’ was implemented within the PixelReader class. The method took two parameters, the first being an integer which provided an indication of the plane number which should be replaced. The second parameter was the replacement plane. The plane was then changed back into a 1D array which was iterated through alongside the pixels array. Each bit in the pixels array corresponding to the plane number was then altered to match the bit in the replacement plane array. This was carried out through bit manipulation operations as detailed in section 6.1.7 with a bitmask corresponding to the plane number passed in. Once the array had been fully exhausted, the replacement was complete.

8.7 The Hiding Implementation

The hiding implementation itself made use of each of the processes detailed above in sections 8.1 through to 8.6. The steps altogether were as follows:

While there was more information to hide (i.e. another block of payload data), each bit plane was accessed in turn and segmented. As each segment was accessed, a calculation was performed to determine if the segment was complex enough for hiding. This was done by determining if the complexity was greater than the threshold value of 0.3 as established by Kawaguchi et al (1998). If the segment was complex enough, the current block of payload data would replace the current segment only if it was noisy enough, if it was not, then a conjugation operation was performed to make it so (as detailed in section 3.3). The process of performing the conjugation operation involved performing the exclusive or operation on the bits of the block with the corresponding bits of the white checkerboard pattern (again, detailed in section 3.3) and resulted in setting the first bit of the block to 1, which indicated the operation had been performed so as to allow successful extraction.

If the segment was not complex enough to hide information in, then the current block was not hidden and the next segment was checked until a complex enough segment was found to hide the information in. If at any time there were no segments left on a plane, but there were remaining unused planes and blocks of unhidden information, the current plane was replaced using the PixelReader class and the next plane was accessed. The planes were accessed from plane 0 through to plane 23. It was later discovered that a more effective method would have been to access the least significant plane of each colour (the 0th plane, then the 8th plane, then the 16th) and then moving on to the next least significant plane for each colour. This was implemented as a result and improved the overall effectiveness of the algorithm. The process was repeated for this plane and subsequent planes until all information had been hidden.

Once all the hiding had been completed, all the bit planes had been replaced as appropriate and the PixelReader object then held an altered array of pixels which held the payload information within the complex segments. An accessor method was then used to access this array of pixels, this then allowed for ImageIO to be used to write out the new pixels into an image file.

8.8 The Extraction Implementation

The BPCS extraction was the reverse process of the hiding algorithm. The idea of extraction was to access each bit plane, and extract the segments which had been used to hide information. The bits extracted were then reconstructed to form the header information and the bytes of the file, which were written out to reform the file.

The first step involved creating a PixelReader object in order to access the planes and therefore segments of the cover image. Bit planes and segments could then be accessed as within the hiding algorithm.

The first bit plane was accessed, from there, segments of the plane were iterated through to determine if they were complex or not (by the same standard as the hiding algorithm). If the segment was complex, then it must have had information hidden in it, thus the segment bits were retrieved. The bits of the segment were read one by one with a count of the number of bits being kept to allow the program to populate the size and the extension of the payload as well as the payload itself at the correct times.

If a segment was deemed complex enough, the bits were extracted from it and a count of the number of segments was incremented. The purpose of the count of the segments used was to allow for an indication of when to stop extracting information from complex bits. This worked by allowing the size of the file to always be extracted by setting an initial ‘segmentsUsed’ variable to 2. Segments were accessed as long as the count of complex segments was less than the number of segments used. The number of segments used was updated upon extraction of the size, which was completed when a count of the number of bits extracted was equal to the number of bits used to store the size. It was possible to calculate the number of segments required to hide the file given the size, thus the iteration would stop once this number was reached.

Once a segment was identified as being complex, the segment’s conjugation map was checked to see if the segment had been conjugated. If the map was set to a 1 then it was conjugated,

thus the conjugation operation was applied once more before the bits were extracted to reverse the initial conjugation.

In order to extract the bits from the segment, two methods were implemented in the plane class. These methods were ‘hasNextBit’ and ‘getNextBit’. The getNextBit method returned a -1 to indicate the bit accessed was the conjugation map. This allowed the conjugation bit to be ignored upon extraction. Two variables were used to indicate where in the segment the next bit was to come from. This bit was then accessed as necessary.

The iteration of the segments in a plane was continued until either the count of the number of segments accessed reached the number of segments required to hide the file or the plane ran out of segments. If the plane ran out of segments, but there was still more to be extracted, then the next plane was retrieved and the process repeated until the number of segments from which information was extracted was equal to the number of segments it took to hide the information in the first instance.

When the size bits and extension bits were extracted, they were added to a list as with the extraction in the least significant bit algorithms. These lists could then be reconstructed as detailed previously (section 6.5) in order to access the header information.

In order to increase efficiency and reduce dependency on Lists, for the file bits, the reforming of the bytes was carried out in a different manner. A variable was set up to hold the current byte (“myByte”) as well as a counter to indicate the number of bits which had been concatenated to the end of the byte. Once a byte had been formed, it was written out and the counter reset to 0. This continued until all bytes had been written out. Upon conclusion of this, the extraction was complete.

Chapter 9 – GUI Evaluation

In order to ensure that the user interface was usable, several evaluations were carried out at different points in the implementation. The first evaluation was carried out after implementation of the original design, with limited functionality behind it. The second user evaluation was carried out after alterations had been made to the first design, and the full functionality had been implemented. Final user evaluations were carried out during the evaluation phase of the project, to determine how easily the system could be used. The sections below report the findings of these evaluations and the resulting actions taken.

9.1 GUI Interim Evaluation

The first evaluation, as with the others, was conducted through a think aloud empirical evaluation. The user was provided with a list of tasks. The tasks were as follows:

1. Hide a file in an image
2. Select a different file or image instead of the ones initially selected
3. Extract a file from an image

The original GUI provided for this evaluation was as shown in figure 9.1.1

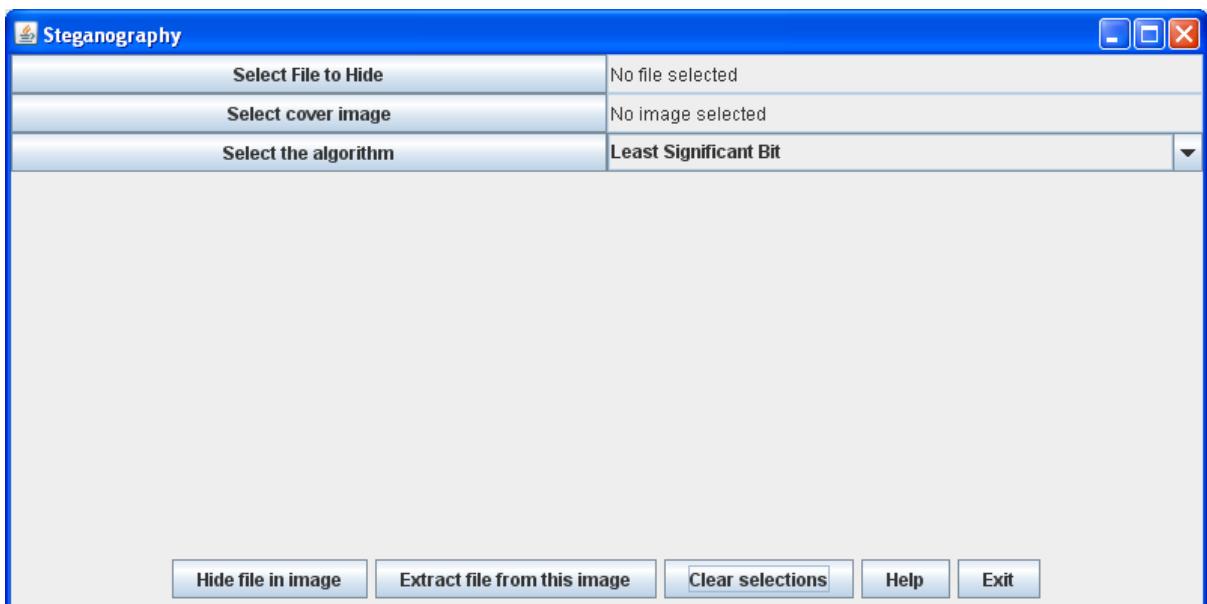


Figure 9.1.1

Hiding was straightforward and intuitive, the user managed to easily select a file and an image to hide it in. However, they found that the algorithm selection was not entirely intuitive. They were inclined to believe that if they hit the “Select the algorithm” another window would appear allowing them to then select the algorithm from there. Thus, the select algorithm section of the GUI was adapted to become a set of 3 radio buttons (as shown in figure 9.1.3).

The second task was to undo the selections made. The user managed this with complete ease, indicating understanding of both the ‘clear selection’ button and the ability to just click on the ‘select’ buttons again.

The user had problems with the extraction of a file indicating it was “not intuitive at all”, demonstrating a gulf of execution which had to be rectified. The problems noted were mainly due to the labelling of the buttons, thus it was decided that the GUI would be updated with an initial screen allowing the user to select either to hide a file or extract one. The GUI was redesigned accordingly, giving the user the option to hide or extract at the start, and subsequently appearing with different screens shown in figures 9.1.2, 9.1.3 and 9.1.4



Figure 9.1.2

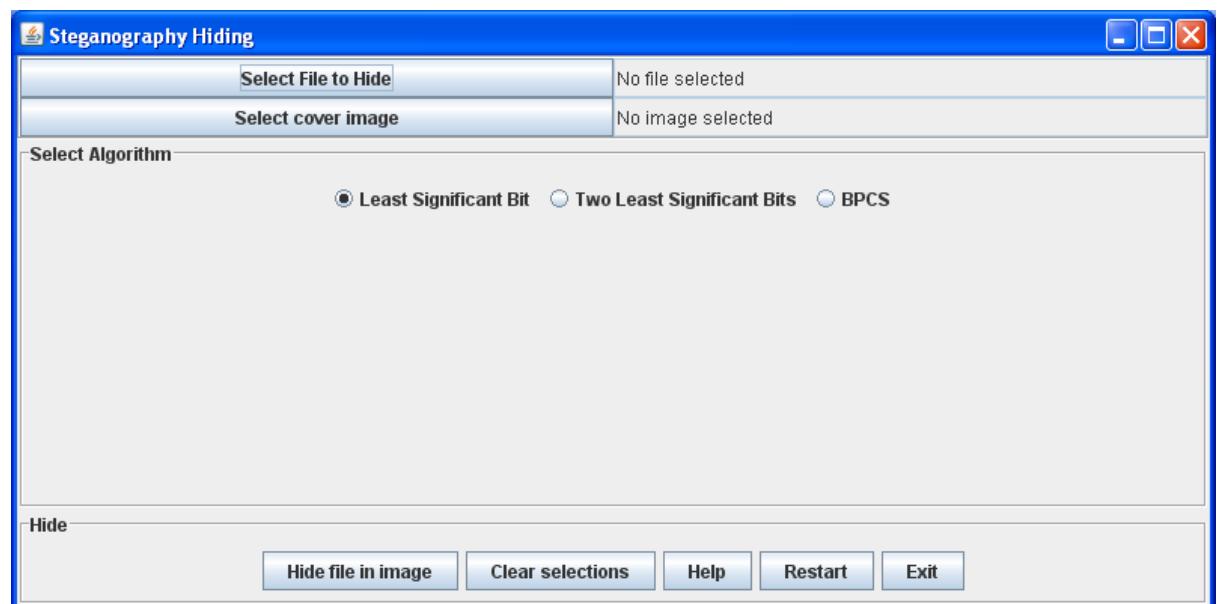


Figure 9.1.3

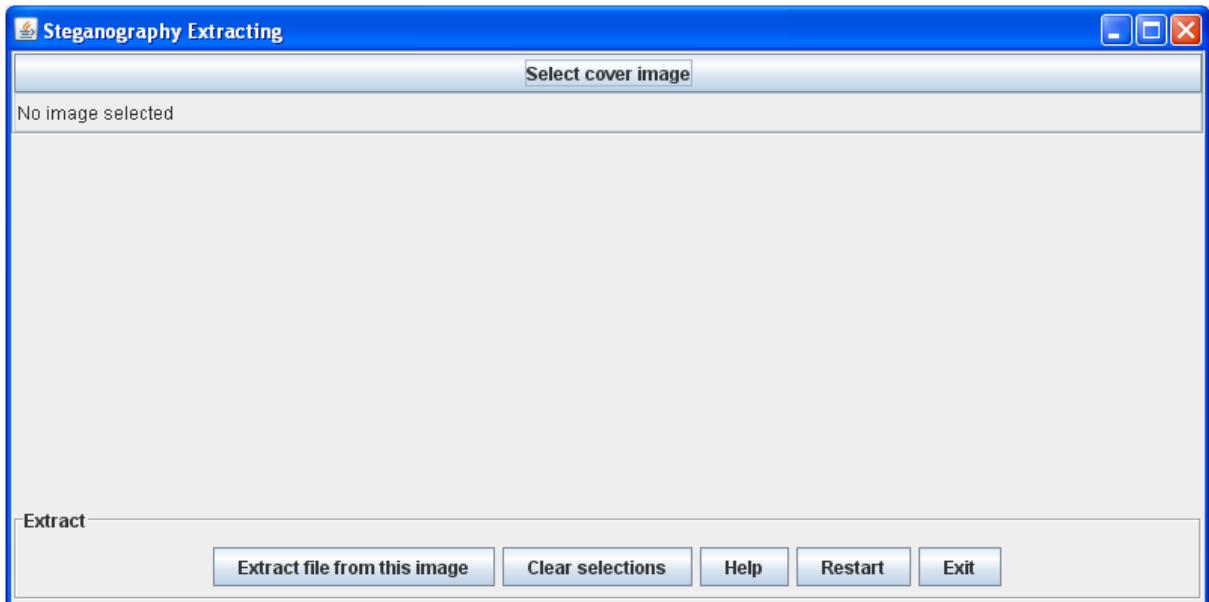


Figure 9.1.4

9.2 GUI Second Evaluation

The second evaluation was carried out once the functionality had been completely implemented. The purpose of the evaluation was to ensure all aspects of the GUI were simple and accessible. The approach taken was similar to the first evaluation. The user was provided with the two tasks of hiding a file within an image and extracting the file from the image.

The process was carried out in a ‘think aloud’ manner. The results were that tasks were carried out with a small amount of comments on usability. The first issue was that the user was unaware precisely what was required in the ‘Name of file to be output’ field. They were unsure of whether they needed to add in the extension or not. This was rectified by the addition of a label next to the field detailing that no extension was required. However, it should be noted that the concept of entering an extension may have been specific to users with a technical background in computing as later evaluations with non technical users indicated this was not a problem.

The second issue raised was that upon extraction the label ‘select cover image’ was not helpful, it left a gulf of execution since the user was not sure of what image should be selected. To rectify this, the label was altered to say ‘select the image to extract from’.

There was also a concern raised that there was no description next to the JComboBox to select the algorithm which had been used to hide the file within the image. Again, this was resolved by the addition of a JLabel which indicated that the user should select the algorithm which had been used to hide the file.

The last small issue, which it was possible to remedy, was the lack of indication that extraction had commenced. Despite a message appearing upon completion, the user was unsure if the process had started. This was corrected by adding in a message when the extraction was starting.

The main issue, which could not be resolved due to lack of available time, was related to the length of time it took to complete the operations. Specifically, in terms of the least significant bit algorithms which took a long time to complete. The ideal solution would have been to implement a progress bar, indicating to the user that the work was being carried out, and at which point it would complete. However, due to the limitations of time available, it was not feasible to implement this. This matter is discussed further in Chapter 11- Future Developments.

9.3 GUI Final Evaluations and Review

The final review of the GUI was completed by providing the same tasks as previously used (hiding a file and subsequently extracting it). The tasks were given to a cross section of users covering varying levels of technical ability. There were no issues with the use of the program by this point, with the exception of the time taken to write out the image. In order to minimise the effects of this concern, messages indicating the commencement of the operations were updated to include a message that the hiding may take some time. As well as this, within the help function, it was also detailed that hiding operations may take a while to complete.

The final alteration to the GUI came in the form of implementing one main screen, and to have two screens called from it for hiding and extracting. The reason for this was due to difficulty switching between ‘hide’ and ‘extract’ modes when prompting the user for the mode required using an option pane. As well as this, it was thought that the user may wish to access the help function before entering either of these modes, this was achieved through the implementation of the separate screen. The final GUI was as shown in figures 5.2.2, 5.2.3, and 5.2.4 (Chapter 5).

It should be noted that despite users not detailing many deficiencies of the program, there is the possibility of the ‘Hawthorne’ effect in which users alter their behaviour since they are aware they are being watched. They may have veered towards being less critical of the system than they otherwise would have been.

10. Testing and Evaluation

After the implementation of all the algorithms, the next step of the project was to perform the testing and evaluation stages. These stages comprised of testing (both black and white box), user evaluations and analysis of the relative effectiveness of algorithms implemented. The full details of these stages are presented in the sections following.

10.1 Testing

It was decided at an early stage, in order to perform the white box testing, unit testing would be implemented through the use of ‘JUnit’. The result of this decision was an extra package being created which held 16 test classes. Each class provided a test for each class of the main program. This was with the exception of the GUI classes and the Main class which initiated the GUI. The testing was carried out by writing test methods within the testing class which tested the methods of the relevant class. Specific cases were used, where the outcome was known. E.g. in the calculation of the complexity and border values the image shown in figure 3.3.2 was used as input and the outcome of the algorithm used to calculate the complexity was compared with the expected result. All the tests written were written with the intention of passing, which is to say that no tests were written which were meant to fail.

Upon running the classes, all tests passed. However, it should be noted that this only increases confidence in the correctness of the program and its algorithms; it does not ensure the program is complete and correct. Given more time, more tests would be written testing each aspect of the program with more than one expected outcome.

In order to test the hiding and extraction methods, a class was written which compared two files byte by byte. The files passed into the class were the file which was hidden, and the file as extracted from the cover image after hiding. The class iterated through each byte of each file by using the Java class FileInputStream. Each byte of the file hidden was compared to the corresponding byte of the file extracted, if at any point they differed, then the test failed. If the files were successfully traversed with no differences between corresponding bytes, then the test was successful.

Black box testing was also carried out; this was to test the functionality of the GUI as well as the functionality of the different algorithms in terms of hiding and extracting using different algorithms and files. The results of which are detailed in Appendix E.

10.2 Hiding Capacity Analysis

For the least significant bit algorithms, it was possible to calculate the precise capacity. It should be noted that despite the theoretical capacity of 12.5% for the least significant bit

algorithm, it was not feasible to achieve this as it was necessary to insert information at the start of the payload which indicated to the program where to stop upon extraction, and which type of file to write out. Given more time, the amount of space used for a header could possibly be reduced by the implementation of magic numbers. A ‘magic number’ is a constant number which represents a particular attribute (indicating, e.g. the type of extension). The header size could also have been condensed by reducing the number of bits used to represent the size by storing the size of the hidden file in terms of bytes rather than bits. The second least significant bit effectively doubled the capacity of the least significant bit algorithm, and the four least significant bit algorithm doubled the capacity once more. The capacity for each algorithm is summarised in figure 10.2.1. The header size consists of the number of bits used to hide the size (32) plus the number of bits used to hide the extension (64), which totaled to 96.

Algorithm	Capacity (in bits)
least significant bit	image file size (in bytes) – header size
two least significant bits	$2 * (\text{image file size (in bytes)} - \text{header size})$
four least significant bits	$4 * (\text{image file size (in bytes)} - \text{header size})$

Figure 10.2.1

Determining the capacity of the BPCS algorithm was directly linked to the complexity of the image. The more complex the image, the higher the capacity of the image. It became apparent through evaluation, that within each of the colours (red, green, blue) there was a least significant plane (the lower numbered plane) and a most significant plane (the higher numbered plane). This can be seen by observing the result of splitting an image into its composite bit planes as provided in Appendix D. The 0th, 8th and 16th bit planes are the least significant of red, green and blue respectively whilst the 7th, 15th and 23rd planes are the most significant.

The capacity of the image could be determined by accessing each plane, and subsequently each segment within the plane to determine if it was complex. If the segment was complex, a counter was incremented. For each complex segment, 63 bits of information could be hidden (64 bits per segment, less one bit for the conjugation map). Thus, the capacity of an image could be determined precisely by carrying out this computation. However, this was not done within the project as the process threw ‘OutOfMemory’ errors and there was insufficient time to refactor code to negate this.

To demonstrate the capacity, and investigate the claim the 50% of the size of the image could be used, an evaluation was carried out on 10 of the testing images. The result of the evaluations is detailed in figure 10.2.2. The effectiveness of these capacities is discussed in section 10.3. Appendix F holds the original testing images.

Image Used	Reported Capacity (as a % of cover image size)
Wild Flowers	79
Castle	31
Canoe	56
Cliff	55
Houses	54
More Houses	63
Ruins	54
Pebbles	69
Speedboats	57
View	52

Figure 10.2.2

In order to test the result of using the maximum capacity, a file of size 10.6MB was hidden in the image shown in figure 10.2.3. The resulting image is shown in figure 10.2.4. It appeared noticeable, but not significantly so. The extraction was successful, thus proving the capacity claimed in the paper (Kawaguchi et al, 1998) of >50%. A file equating to just over 50% of the image size was also hidden in the image shown in figure 10.2.3, the resulting image is shown in figure 10.2.4, showing that not only can more than 50% be hidden, but it is not immediately noticeable. The alteration is clear, however, when the image is ‘zoomed’ in. This is shown in figure 10.2.5. For closer observation, these images are available on the accompanying CD.



Figure 10.2.3 – Original Image



Figure 10.2.3 – 73% Hidden



Figure 10.2.4 – 50% Hidden



Figure 10.2.5 – 73% Hidden Close up

10.3 Effectiveness Analysis

In order to determine the effectiveness of the different algorithms user evaluations were carried out. The effectiveness was measured in terms of how noticeable any alterations to the image were. If the user looked at the image, but could not perceive anything “wrong” with it, then the algorithm was deemed effective. If, on the other hand, there were obvious problems with the image, then the algorithms were flagged as ineffective.

The initial evaluations were carried out with a range of 15 images which used a cross-section of each of the algorithms, different capacities used for hiding and unaltered images. This evaluation was carried out with 11 users, 3 of these users were given the evaluation online in order to determine if the use of different screens and settings affected the results. The use of the online facility to perform the evaluations also ensured as large a number of evaluations as possible.

In order to assess the users perception, a likert scale form was used. This was to determine not only if they suspected something wrong with the image, but also to place this on a scale for a more complete view of their interpretations. A 5 point scale was used, allowing the users the extreme choices of ‘No alterations’ and ‘Definitely Altered’ with the middle scales being probably no alteration, can’t tell and a little alteration. It was noted by a small number of users that the ‘probably none’ and ‘can’t tell’ values were a little ambiguous. However, in order to maintain comparable results, the scale was kept as designed. Given an opportunity to re-evaluate, the scale could have been changed to a 3 point scale with the extremes as stated before, and a middle value of ‘Can’t tell’. If the majority of the results lay on the lower three values (no alteration, probably none, can’t tell) they were deemed effective. If the majority lay on the side of ‘a little alteration’ or ‘definitely altered’ they were deemed ineffective.

Another notable observation from the evaluations was that all users, after being told the purpose of the project and the basis of the evaluation, were highly skeptical of each image. E.g. there were several users who commented on people within pictures looking ‘photoshopped’ and the sky in some images being ‘too white’. In these cases, there was nothing wrong with the image in this sense. It became apparent that people were expecting alterations as one would see for digitally enhanced images. However, when an image was used which had too much information hidden within it, almost all users picked up on it.

After the evaluations had been completed, it became apparent that the BPCS algorithm was not implemented as effectively as the program implemented by the authors of the algorithm. After some investigation, it was discovered that the reason for this was the order in which the planes were used for hiding information. Initially, each plane was accessed in order from 0,1,2,...,23. The reason this was not as effective was because the planes have varying complexity within each of the colours red, green and blue. As detailed before, this can be seen in Appendix D. This was altered so that the least significant plane of each set of 8 bit planes (one for each colour) was used before the more significant of each set of 8 planes. The result of this alteration was that the BPCS became more effective. Thus, a second evaluation was carried out, with a small selection of users and images (due to the limit on time). A selection of images including the original image, and several using very large limits of the improved BPCS implementation were presented to users who indicated if they perceived anything wrong with the image.

A summary of the effectiveness of each algorithm is provided below, full results of the evaluations are provided in Appendix G. The images used for the evaluation are provided in Appendix H.

In general, the least significant bit algorithm was effective as was the two least significant bits algorithm. Four least significant bits algorithm was noticed by most of the users, thus ineffective. BPCS was noticeable, but again this was before the improvement of the algorithm. Of the 5 unaltered images, only one was misinterpreted by the majority as altered.

The second evaluation reported a significant improvement in the BPCS algorithm effectiveness. A summary of results is provided here with full details in Appendix G. In terms of capacity, up to 50% was unnoticed by users. The majority of users were unsure of complex images which had between 51 and 72% hidden within them, but were positive of alterations to uncomplex images which had more than 50% hidden in them. This evaluation consisted of images which had the estimated maximum capacity hidden in them, as reported by the estimation class alongside images which had the true maximum capacity hidden in them.

The images true maximum capacity was calculated by the creation of a class which iterated through the bit planes counting the number of truly complex segments. From here, the number of bits which could be hidden were calculated as:

$$63 * \text{no. of complex segments.} \quad (63 = 64 \text{ bit segment less conjugation bit})$$

It should be noted that the hiding using this capacity was achieved by removing the section of code within the GUI which stops any hiding beyond the reported estimated capacity. The limit is in place with the intention of stopping users using more of the capacity than they should and obtaining unfavorable results. Thus, if an individual wished to perform a “true” maximum capacity hiding operation, they would need to alter the code to remove this check and run the class ‘ComplexityAnalysis’ for the desired cover image.

10.4 Image Analysis

In order to further analyse the images, several approaches were taken. Firstly, selections of ‘post-hiding’ images were ‘subtracted’ from the original images in order to obtain the ‘difference’ image. The next step was to perform histogram analysis on a selection of the original images, post hiding images and difference images. The results of the difference analysis are detailed in this section, the histograms are provided in Appendix J.

10.4.1 Difference Images

A difference image is the result of subtracting the pixel values of the image before the alterations, and after the alterations. If the result of subtracting the values of the pixels was negative, 256 (the number of possible colours) was added to ensure the resulting pixel to be written out was in the correct range (0 – 255). The resulting pixels were then written out to an image file. These images then provided a visual map of where the image had been altered, and

an indication of the extent to which they had been altered. The resulting images are shown in figures 10.4.1 to 10.4.3. Figure 10.4.1 shows the result of subtracting the image with 50% hidden using the BPCS algorithm (Image 4 from the second evaluation in Appendix H) from the original ‘Wild Flowers’ image (as in Appendix F).

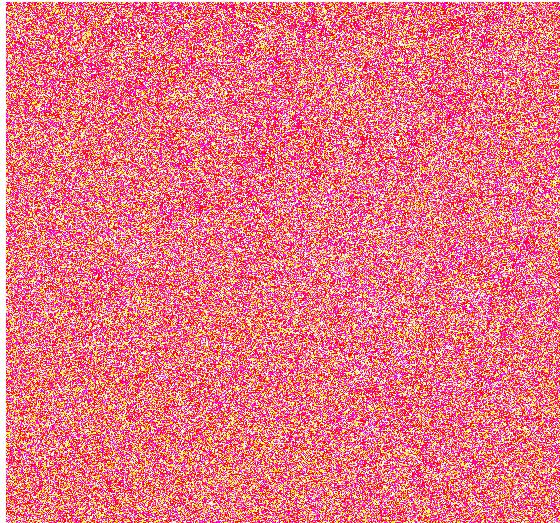


Figure 10.4.1

Figure 10.4.2 shows the result of subtracting the image with 72% hidden using the BPCS algorithm (Image 5 from the second evaluation in Appendix H) from the original ‘Wild Flowers’ image (as in Appendix F).

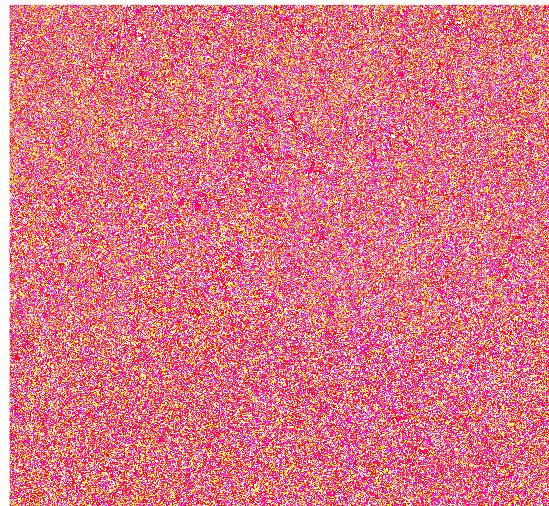


Figure 10.4.2

Finally, figure 10.4.3 shows the result of subtracting the image with the maximum capacity hidden using the least significant bit algorithm (Image 5 from the first evaluation in Appendix H) from the original ‘Canoe’ image (as in Appendix F).

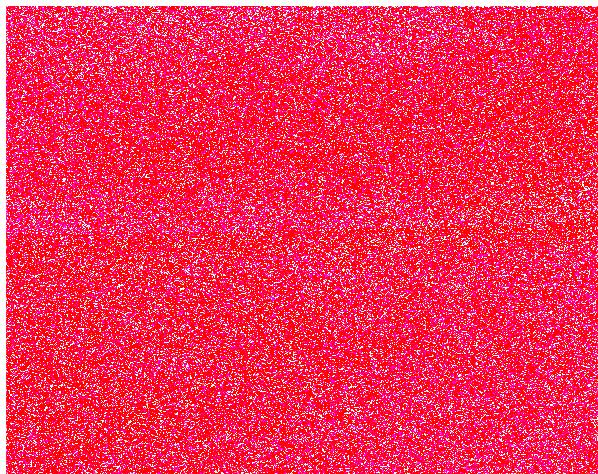


Figure 10.4.3

Chapter 11 -Future Developments

As mentioned before, there were substantial limitations in terms of the time allowed to complete the project. The result of this was that there were a number of aspects of the program which could be improved upon, and functionality which could be added given additional time for their implementation. This chapter aims to discuss some of the possibilities of future developments of the program.

It was detailed by Kawaguchi et al that implementing the transformation from pure binary code bit planes to canonical gray code bit planes increased the hiding capacity significantly. Thus, given more time, it would have been ideal to implement this transformation. The process takes the binary code plane (as implemented in this project) whose m^{th} bit plane is a collection of all the m^{th} bits and transforms them in the following way:

The m -bit gray code $g_{m-1}, \dots, g_2, g_1, g_0$ can be computed from

$$g_i = a_i \text{ xor } a_{i+1} \quad 0 < i \leq m-2$$

$$g_{m-1} = a_{m-1}$$

(xor denotes the exclusive or operation)

(Joshi, 2006)

This could be implemented by using the bit manipulation exclusive or operation on the corresponding bits.

Another aspect which could have been improved was the lack of indication of how long the hiding and extraction processes take. This would be avoided by the implementation of a progress bar. Java Swing provides three classes to allow the implementation of this, these are JProgressBar, ProgressMonitor and ProgressMonitorInputStream. JProgressBar is a visual component which allows the user to see the progress on screen. ProgressMonitor is not visual, but monitors the progress of a task. ProgressMonitorInputStream is an input stream with a progress monitor attached, which could have been used on reading in of the images to allow progress to be determined more easily. An implementation of these classes would allow progress to be easily monitored and as a result, effectiveness of the program improved.

In the proposal of the project, it was decided that had there been sufficient time, different complexity measures (as opposed to the border complexity measure) would have been implemented. However, time was insufficient and this did not occur. If there were more time available, these measures would have been implemented in order to compare results. These measures involve using a DF expression (a one dimensional representation of the quadtree of a binary picture) to count the number of leaf nodes and then divide this by an integer calculated from the size of the image.

In addition to the above, the removal of the dependency on bmp files as the cover image for least significant bit algorithms would be desirable. The dependency comes from the ‘skipping’

of the header size, which is specific to bitmap images. The proposed approach would involve determining the type of image file used, and using this to populate the correct image file header size which should be skipped over before altering bytes to hide payload information. This would result in the same format of image being written out, thus, upon extraction, the header size of the image format could be determined in the same fashion, and the unaltered bytes could be ignored before extraction.

A similar dependency is apparent in the BPCS algorithm, this is due to the assumption that there are 24 planes to be used and extracted from at various points of the algorithm. In order to improve this, it would have been desirable to alter this in a similar fashion to the way described above. This would be by checking the extension, and populating the number of planes which could be used/extracted from by having a list of extensions and their corresponding bit depth. The type of image written out should also match the type of image used to hide, so that the correct number of planes can be used upon extraction.

In terms of the capacity estimation for the BPCS algorithm, this could also be improved by in effect implementing a ‘dummy’ hiding where the number of complex segments is counted. This would mean iterating through the bit planes and segments as one would with hiding, until all complex segments were counted. This would then provide a precise capacity for hiding, and not just an estimation.

Had time not been an issue, there was a desire to refactor some areas of code. In particular, it would have been beneficial to refactor the ‘Plane’ class so that Plane was a superclass, and have ‘Segment’ and ‘Block’ as subclasses, this would allow for common functionality to remain in common, but improve the class cohesion in terms of having one class contributing to one job.

Despite being outwith the project’s scope, it would have been interesting to determine how susceptible to steganalysis the algorithms implemented were. Also, in order to increase usability within the area of security, implementing an option of encryption or password protection before hiding and upon extraction could have been an advantage.

Similarly, it would have been interesting to allow the user to vary the complexity threshold (which was implemented as 0.3 as per Kawaguchi et al) in order to examine the results of both a higher and a lower value.

A final item which would have been desirable to rectify, would be the onus on the user to select the algorithm which was used to hide the file when using the extraction function. This could potentially be completed through use of a ‘magic number’ to indicate the algorithm used. This would be placed at the start of the image written out, and should be accessed in the same way no matter which algorithm is being used for extraction.

Chapter 12- Conclusion

The aims of the project were to implement three steganography algorithms using Java, and compare the results in terms of capacity and effectiveness. In respect of these aims, the project was successful. All three algorithms (least significant bit, two least significant bits, BPCS steganography) were implemented successfully, allowing both the hiding and extraction of a file within a cover image. A comparison of the effectiveness and capacities of the algorithms was carried out, with the results being summarised in sections 9.5 and 9.6.

It became apparent upon comparison of the different algorithms, that the most successful algorithm was BPCS steganography. This is due to its hiding capacity. It has been shown in this project that the claim made by Kawaguchi et al (1998) that up to 50% of the image can be used, is correct. (See figures 9.5.3, 9.5.4 and 9.5.5) This provides a lot more scope for hiding information in plain sight as it guarantees more than mere watermarking of the image. The algorithm is also a lot quicker in terms of processing time since it is based more on bit manipulation and ImageIO than FileInputStream and FileOutputStream (used in the least significant bit algorithms) which take significantly more time. In general, if one uses a complex image, at least 50% of that image size can be used to hide a file with no noticeable effects, as determined through user evaluation (see section 9.6).

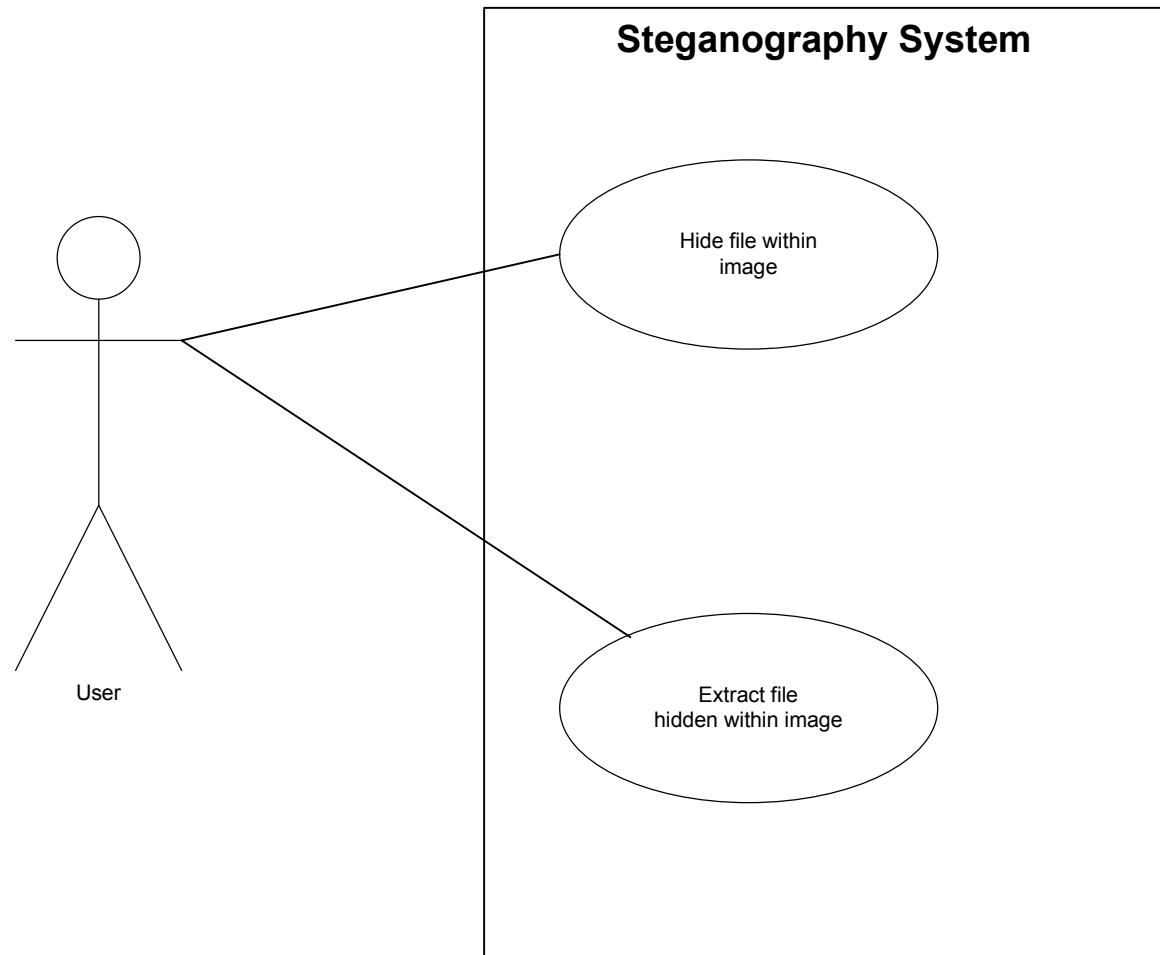
The least significant bit and two least significant bit algorithms were deemed effective in terms of their output images having unnoticeable alterations. This provides the user with up to 12.5% or 25% less the file header size (96 bits) hiding capacity. These algorithms were much more straight forward in terms of their implementation. Therefore, if the user's needs were to hide only this amount and they had to implement one of the algorithms, the least significant bit or the two least significant bit approach would be the most appropriate choice.

In order to demonstrate the significant difference between hiding 50% using the least significant bit approach, and the BPCS approach, a four least significant bit algorithm was implemented. In each evaluation case (detailed in section 9.6), the resulting image had noticeable alterations. The significance of implementing the BPCS algorithm was that despite providing the algorithm within the authoring paper, no implementation details were provided at all. The source code for the software implemented alongside the paper is not available either. This implementation provides an insight into how one might approach this.

Overall, the project was highly successful in achieving its aims, and implementing an additional algorithm to place the effectiveness of the other algorithms in a better context. There were some areas which could have minor improvements made, given sufficient time. These were detailed in Chapter 11.

Appendix A – Requirements Capture Diagrams

UseCase



Use Case Description

Hide file within image

Rationale – This is the basis of the concept of steganography

Actors - User

Priority – Must have (according to MoSCoW approach)

Preconditions – Both an image and a file must be selected by the user

Post conditions – The image with the file hidden in it is written out

Basic Flow of Events (the "Happy Days" scenario) – follows the activity diagram ‘hiding’ route where there are no problems with hiding.

Insufficient Hiding Space Scenario –

Follows the activity diagram where the hiding fails

Non-functional requirements

Accessibility – java, platform independent, do it in packages which can be imported like in the java stuff e.g. java.swing etc

Usability requirements: no indepth knowledge of the workings/details of steganography required, must be simple to use

Portability – java

Robustness- must be robust

Effectiveness – the result of the hiding should not be discernable to the naked eye, if it were noticeable, this would undermine the entire purpose of steganography.

Risks – low, the hiding is essential to the steganography concept

Hiding Capacity – the program should be able to hide files equating to a significant proportion of the image size

Extract File from Image

Rationale – This is the reverse of the hiding, again this is quintessential to the concept of steganography.

Actors- User

Priority – Must have

Preconditions – The user must select an image in which a file has been hidden using the same software.

Post conditions – The file is extracted and written out, it can then be accessed by the user.

Basic Flow of Events (the "Happy Days" scenario) – follows the flow on the activity diagram for “extraction” without any problems

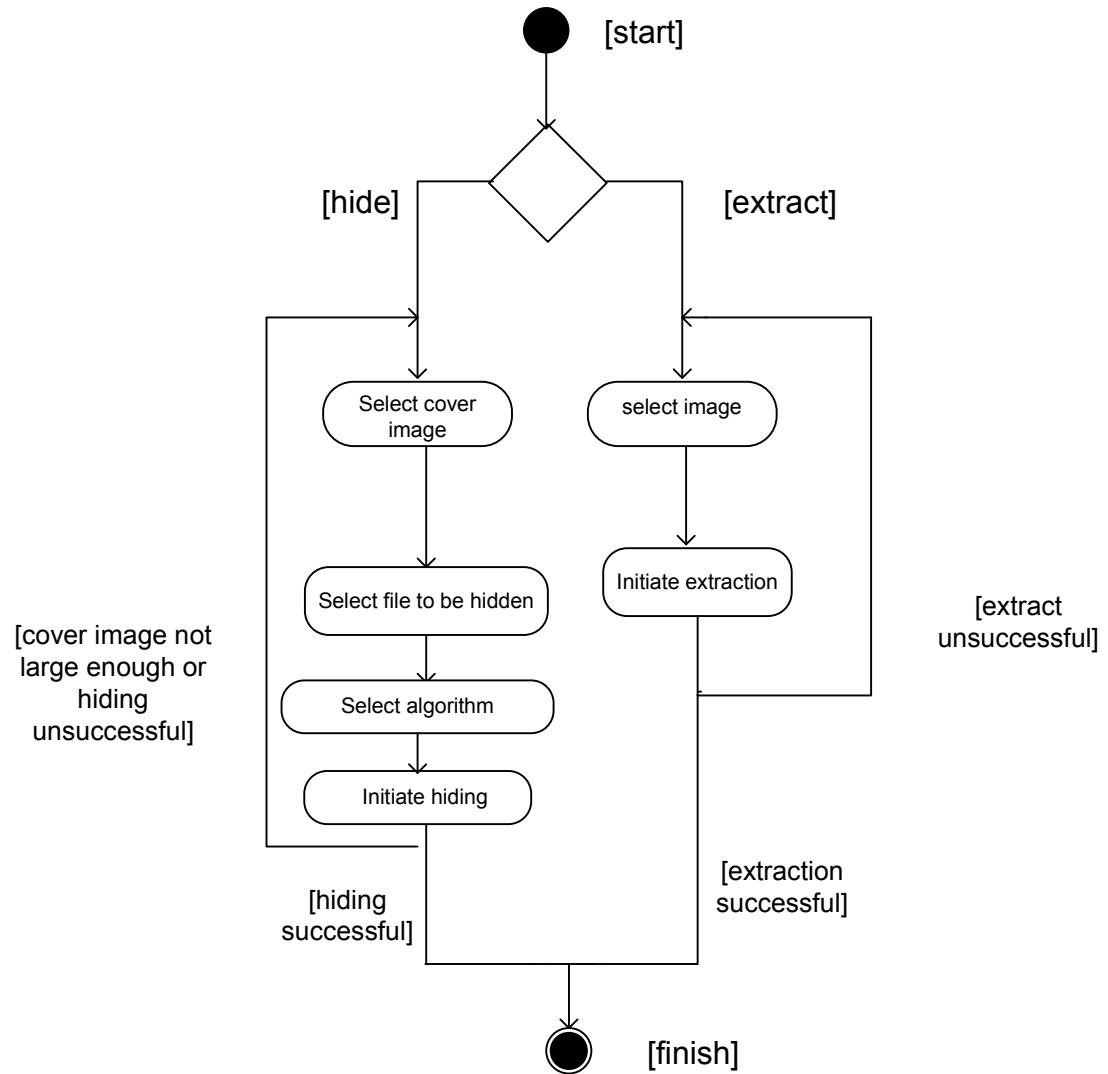
Failure Scenario – The file fails to write out, follows the activity diagram extraction flow with extract unsuccessful route.

Non-functional requirements –

As for hiding.

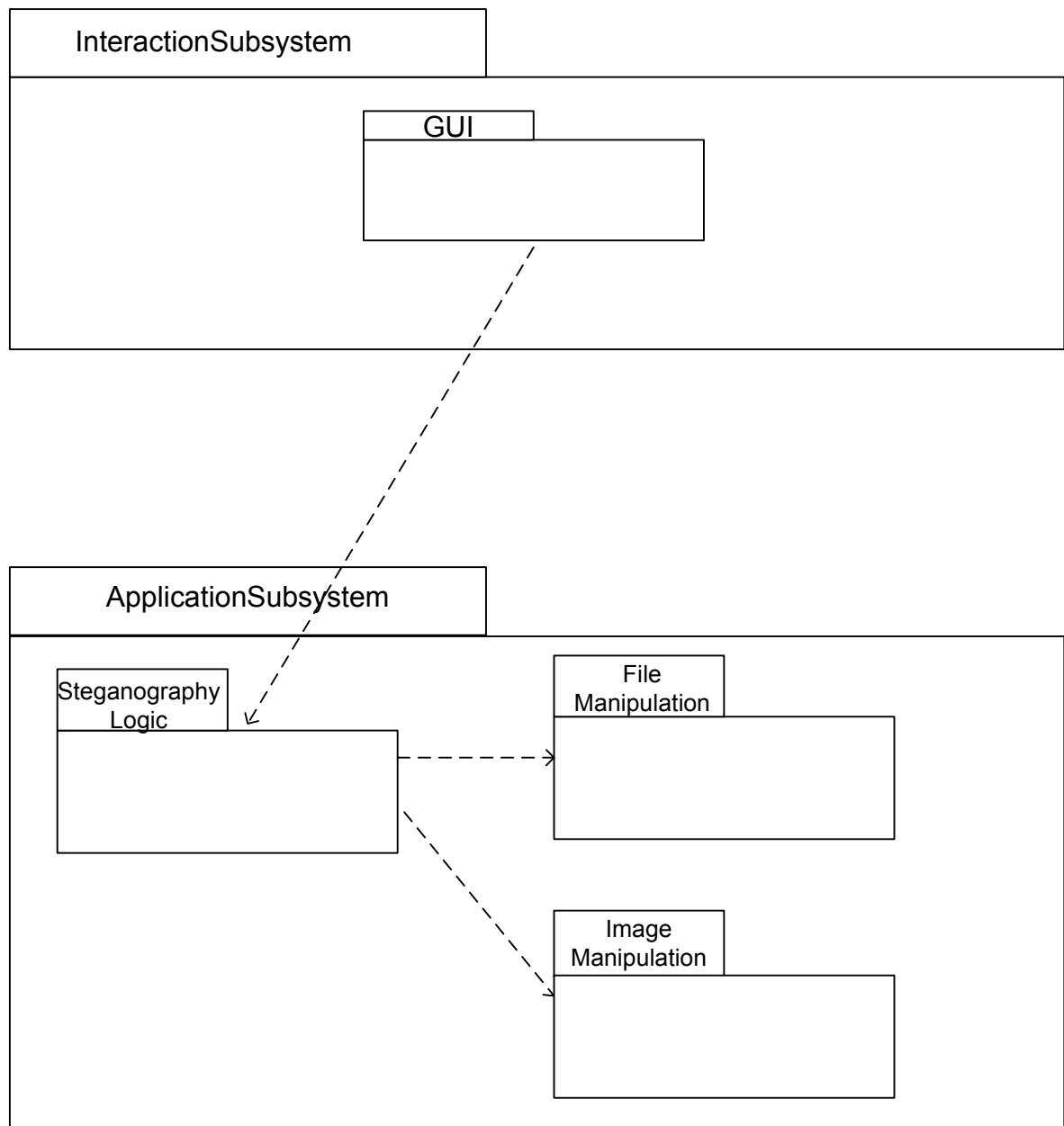
Risks – Low, this use case was determined very early in the project and discussed at length with the client.

Activity Diagram



Appendix B- Design Documents

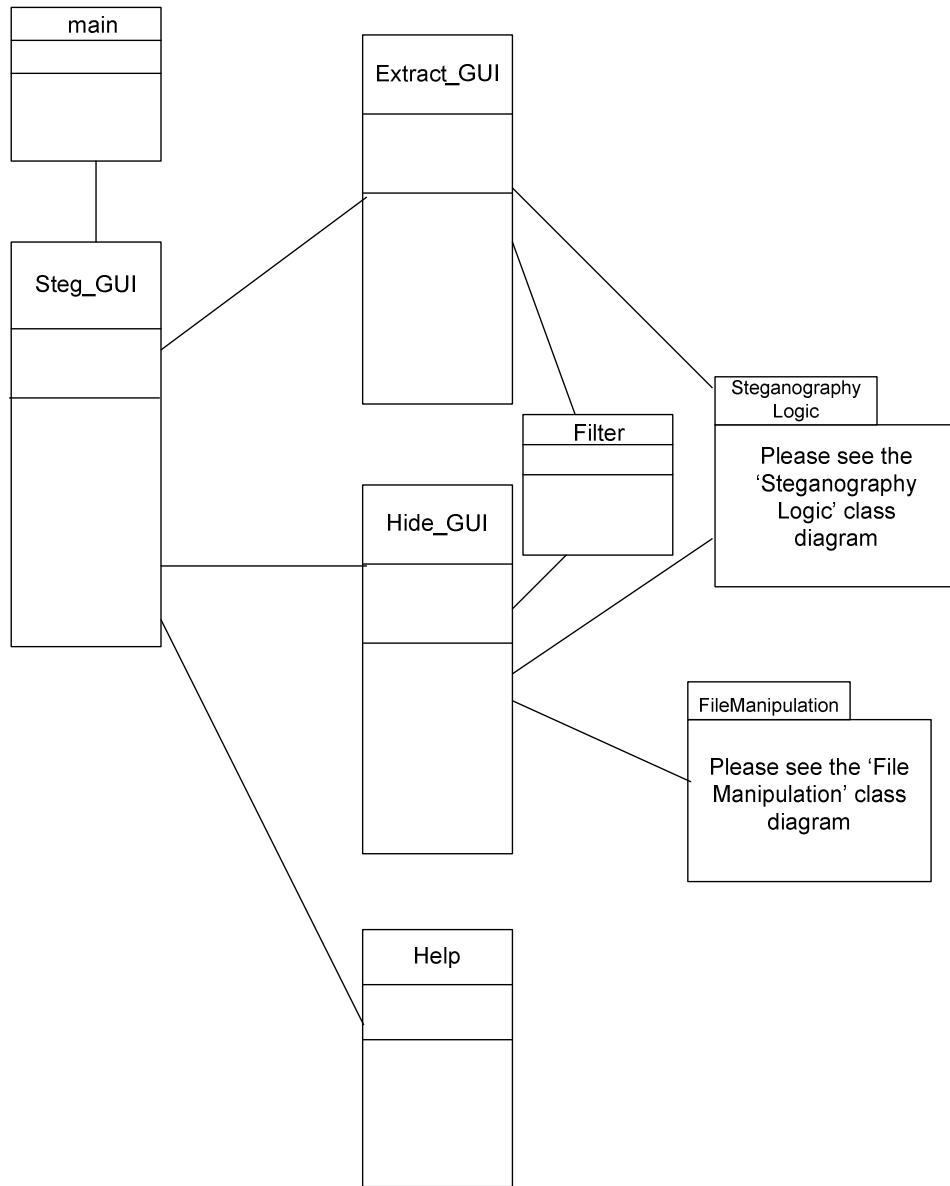
Package Diagram



Class Diagram

Due to the volume of classes, the class diagram was split into packages. In addition, all methods and variables are detailed in the tables which follow instead of within the diagram as it was unfeasible to have them within the diagram and to remain legible.

GUI Class Diagram



Steganography Logic Class Diagram

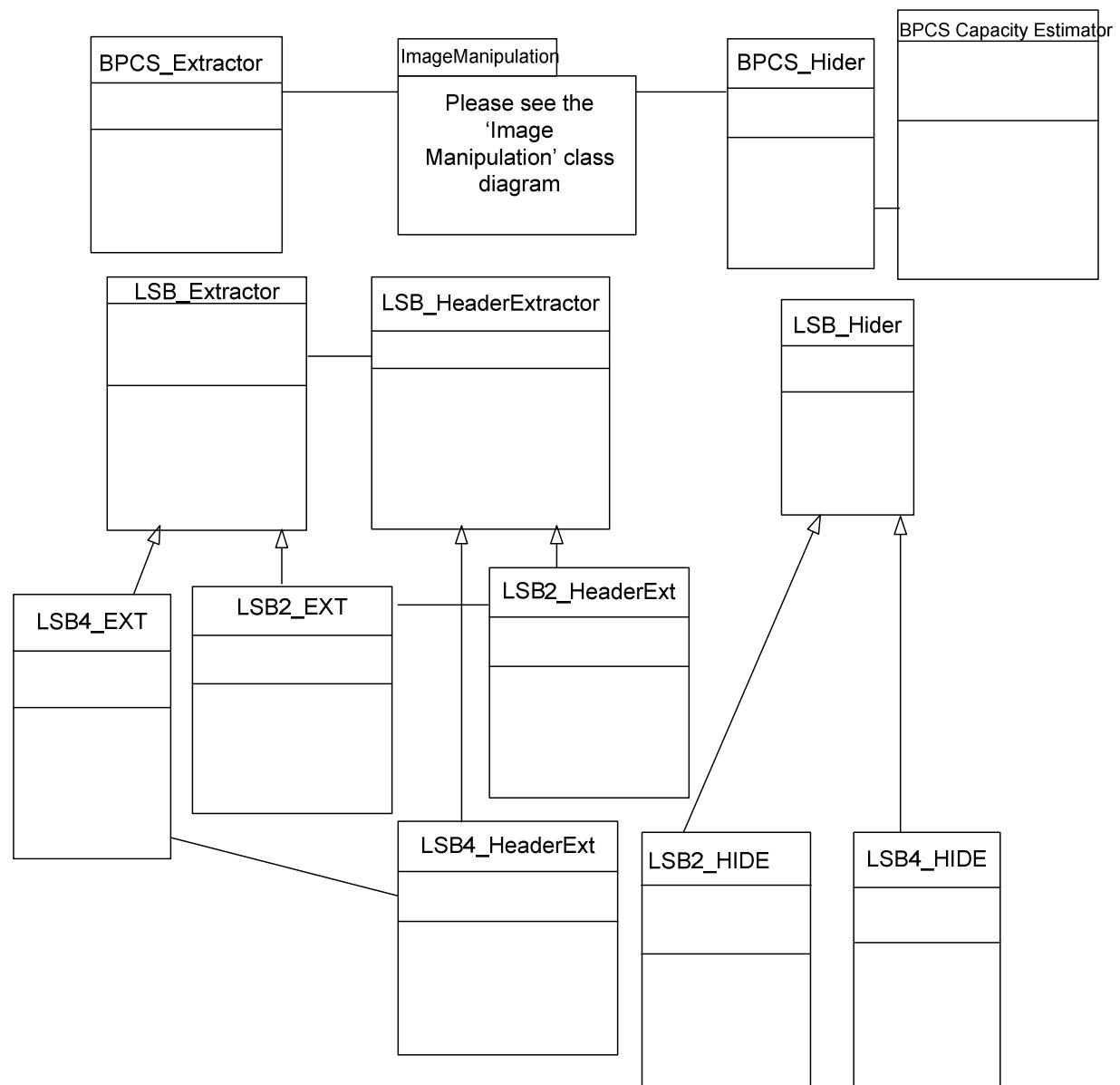
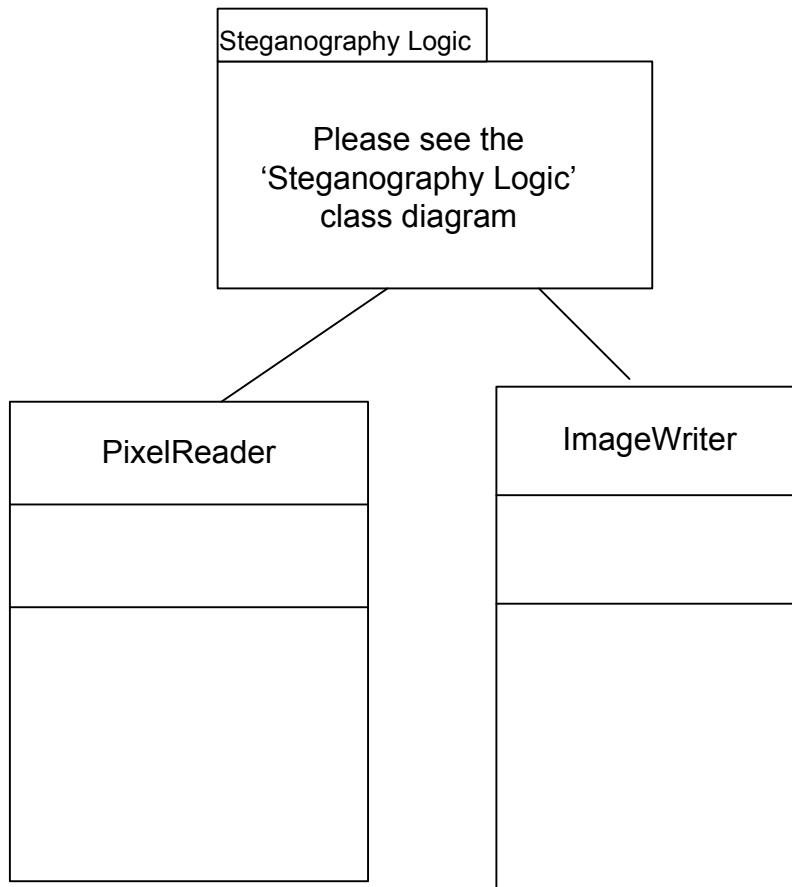
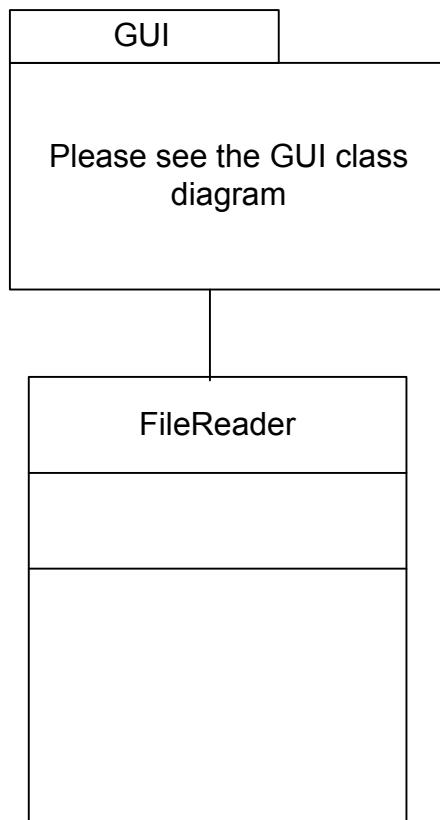


Image Manipulation Class Diagram



FileManipulation Class Diagram



Class Descriptions

Class: Main

Variables	
Methods	main(String[] args)

Class: Steg_GUI

Variables	<code>private JButton hide;</code> <code>private JButton extract;</code> <code>private JButton exit;</code> <code>private JButton help;</code>
Methods	<code>public Steg_GUI()</code> <code>public void layoutTop()</code> <code>public void actionPerformed(ActionEvent e)</code> <code>public void exitPressed()</code>

Class: Extract_GUI

Variables	<code>private JButton selectImage;</code> <code>private JButton extract;</code> <code>private JButton clear;</code> <code>private JButton help;</code> <code>private JButton close;</code> <code>private JTextField imageNameField;</code>
-----------	---

	<pre>private JTextField outputName; private String imageName; private String lsb1; private String lsb2; private String lsb4; private String bpcs; private File imageFile; private String algorithmUsed; private JComboBox algorithmUsedBox; private String invalidCharacter private boolean isBlank</pre>
Methods	<pre>public Extract_GUI() public void layoutTop() public void layoutMiddle() public void layoutBottom() public void actionPerformed(ActionEvent e) public void helpPressed() public void clearPressed() public void selectImagePressed() public void extractPressed() public void extract() public boolean nameIsValid(String name)</pre>

Class: Hide_GUI

Variables	<pre>private JButton selectFile; private JButton selectImage; private JButton clear; private JButton close; private JButton help; private JButton hide; private JButton getCapacity; private JFileChooser chooser; private JTextField imageNameField; private JTextField fileNameField; private JTextField outputName; private String imageName; private String fileName; private String lsb1; private String lsb2; private String lsb4; private String bpcs; private String algorithmChosen; private File imageFile; private File hideFile; private boolean sizeOk; private int headerSize; private final long byteLength; private JRadioButton lsb1Button; private JRadioButton lsb2Button; private JRadioButton lsb4Button; private JRadioButton bpcsButton; private String invalidCharacter; private boolean isBlank</pre>
Methods	<pre>public Hide_GUI() public void layoutTop() public void layoutMiddle() public void layoutBottom() public void actionPerformed(ActionEvent e)</pre>

	<pre>public void getCapacityPressed() public void helpPressed() public void clearPressed() public void selectFilePressed() public void selectImagePressed() public void selectAlgorithmPressed() public void hidePressed() public boolean checkSizes(String alg) public void hideSelectedFile() public int calculateSegments() public boolean nameIsValid(String name)</pre>
--	---

Class:Help

Variables	<pre>private JPanel top; private JPanel middle; private JPanel bottom; private JButton introduction; private JButton hiding; private JButton extracting; private JButton algorithms; private JButton close; private Font font; private JTextArea details;</pre>
Methods	<pre>public Help() public void layoutTop() public void layoutMiddle() public void layoutBottom() public void actionPerformed(ActionEvent ae) public void introductionPressed() public void hidePressed() public void extractPressed() public void algorithmsPressed() public void closePressed() public String hidingDetails() public String introductionDetails() public String extractionDetails() public String algorithmDetails()</pre>

Class:Filter

Variables	
Methods	<pre>public Filter() public boolean accept(File f) public String getDescription()</pre>

Class: ImageWriter

Variables	<pre>private int [] pixs; private int width; private int height; private String imageOut; private boolean success=true;</pre>
Methods	<pre>public ImageWriter(int [] pixels, int w, int h, String name) public void write() public boolean getSuccessBool()</pre>

Class:PixelReader

Variables	<pre><code>private File imageFile; private BufferedImage picture; private int height; private int width; private int [] pixels; private boolean success=true; private String error;</code></pre>
Methods	<pre><code>public PixelReader(File f) public void readImage() public int[] grabPixels() public void setHeight(int h) public int getHeight() public void setWidth(int w) public int getWidth() public int[] getPixels() public Plane getBitPlane(int i) public void replaceBitPlane(int planeNo,Plane plane) public boolean getSuccessBool() public String getErrorMessage()</code></pre>

Class: BPCSCapacityEstimator

Variables	<pre><code>private double totalMostSigComplexity; private double totalLeastSigComplexity; private int segCountMS; private int segCountLS; private double medianValue; private double capacity; private double averageForLSBP; private double averageForMSBP;</code></pre>
Methods	<pre><code>public BPCS_CapacityEstimator(File coverImage) public void analyse(File image) public void calculateCapacity() public double getCapacity() public double getMostConservativeEstimate()</code></pre>

Class: BPCS_Extractor

Variables	<pre><code>private File cover; private FileOutputStream out; private final int byteLength private String ext private int [][] wCheck private Iterator<Integer> sizeIt private final int segmentSize private int size private int width; private int height; private final int bitsPerSegment; private int segPerPlane private int segmentsUsed private PixelReader read; private List<Integer> sizeBits private List<Integer> extBits private String fileOutName private final int numSizeBits private final int numExtBits private final double alphaBase</code></pre>
-----------	---

	<pre> private boolean success; private List<Integer> extBytes; private List<Integer> sizeBytes; private int planesUsed; </pre>
Methods	<pre> public BPCS_Extractor(File im, String name, Extract_GUI gui) public void close() public Plane conjugateBlock(Plane block) public void checkWhite() public void populateSize() public void populateExtension() public List <Integer> makeBytes(List <Integer> bits) public void extract() public int getSize() public String getExtension() </pre>

Class: BPCS_Hider

Variables	<pre> private File coverImage; private File toHide; private int [][] wCheck private double alphaBase private int height; private int width; private FileReader reader; private PixelReader pixReader; boolean needAnotherPlane private int segmentsRequired; private Plane currentBlock; private Plane hidingPlane; int rowStart; int colStart; int planeCount boolean currBlockHidden private final int segmentSize private boolean success private String hidingProblem Plane currSeg; Plane currBlock; boolean done=false; private int planesUsed; </pre>
Methods	<pre> public BPCS_Hider(File image , File hide, int numSegments, String name, Hide_GUI gui) public void writeOut(String name) public void hide() public void hideInCurrentPlane() public void replace(Plane block, int r, int c) public boolean isNoiseLike(Plane segment) public Plane conjugateBlock(Plane block) public void checkWhite() public void initiateReader() public void initiatePixReader() public int getHeight() public int getWidth() public FileReader getFileReader() public PixelReader getPixelReader() public Plane getCheckPattern() </pre>

Class: LSB Extractor

Variables	<pre><code>protected FileInputStream in; private FileOutputStream out; protected File image; protected String fName; protected File fout; protected List<Integer> bits private List<Integer> bytes protected String ext protected int byt; protected int fileSize private final int byteLength protected final int headerSize protected final int sizeBitsLength protected final int extBitsLength protected boolean success</code></pre>
Methods	<pre><code>public LSB_Extractor(File im, String name, Extract_GUI gui) public void createStreams() public void extractBits() public void populateHeaderInfo() public List <Integer> makeBytes public void getFileBits() public void writeOut(List<Integer> bytes) public String getExtension() public int getSize() public List<Integer> getFileBitsList()</code></pre>

Class: LSB_HeaderExtractor

Variables	<pre><code>private File coverImage; protected FileInputStream in; protected int count protected final int headerSize protected final int sizeBitsLength protected final int extBitsLength protected String ext protected int size protected int byt; protected Iterator<Integer> sizeIt; protected List<Integer> sizeBits protected List<Integer> extBits private final int byteLength private boolean success;</code></pre>
Methods	<pre><code>public LSB_HeaderExtractor(File im) private void initiateStream() public void extract() public void getExtBits() public void getSizeBits() public void populateFileSize() protected void populateExtension() private List <Integer> makeBytes(List <Integer> bits) public String getExtension() public int getSize() public boolean getSuccessBool() public List<Integer> getSizeBitsList() public List<Integer> getExtBitsList()</code></pre>

Class: LSB_Hider

Variables	<pre><code>protected FileInputStream in; protected FileOutputStream out; protected FileReader fReader; protected File image; protected File fout; protected int currByte; protected int fileSize; protected final int headerSize protected final int numSizeBits protected final int numExtBits protected final int byteLength protected boolean success</code></pre>
Methods	<pre><code>public LSB_Hider(FileReader reader , File im, String outName, Hide_GUI gui) public void createStreams() public void hide() public void closeStreams() public int swapLsb(int bitToHide,int byt) public void hideFile() public int getCurrByte() public FileInputStream getInputStream()</code></pre>

Class: LSB2_Ext

Variables	
Methods	<pre><code>public LSB2_Ext(File im, String name, Extract_GUI gui) public void extractBits() public void populateHeaderInfo() public void getFileBits() public List<Integer> getFileBitsList()</code></pre>

Class: LSB2_HeaderExtractor

Variables	
Methods	<pre><code>public LSB2_HeaderExtractor(File im) public void extract() public void getExtBits() public void getSizeBits() public void populateFileSize()</code></pre>

Class: LSB2_Hide

Variables	
Methods	<pre><code>public LSB2_Hide(FileReader hideFileReader, File im, String name, Hide_GUI gui) public void hide() public int swap2Lsb(int bitToHide,int byt) public void hideFile()</code></pre>

Class: LSB4_Ext

Variables	
Methods	<pre><code>public LSB4_Ext(File im, String name, Extract_GUI gui) public void extractBits() public void populateHeaderInfo() public void getFileBits()</code></pre>

LSB4_HeaderExt

Variables	
Methods	<pre>public LSB4_HeaderExt(File im) public void extract() public void getExtBits() public void getSizeBits() public void populateFileSize()</pre>

LSB4_Hide

Variables	
Methods	<pre>public LSB4_Hide(FileReader reader, File im, String outName, Hide_GUI gui) public void hide() public void hideFile() public int swap2Lsb(int bitToHide, int byt) public int swap3Lsb(int bitToHide, int byt) public int swap4Lsb(int bitToHide, int byt)</pre>

Plane

Variables	<pre>private int[][] plane; private int rowCount=0; private int colCount=0; private int currSegRowStart=0; private int currSegColStart=0; private int height; private int width; private double border; private double alphaBase; private final int segmentsSize private int segRowPlace; private int segColPlace; private boolean isConjugationBit</pre>
Methods	<pre>public Plane(int[][] pl) public Plane(int[] pl1, int h, int w) public boolean hasNextSegment() public Plane getNextSegment() public int getCurrSegRowStart() public int getCurrSegColStart() public int[] get1DPlane() public int[][] get2DBitPlane(int [] pl1) public void print() public void analysePlane() public double getAlpha() public int getValue(int row,int col) public double getMaxChanges() public double getRowChanges() public double getColChanges() public Plane replaceSegment(Plane currentBlock, int rowPlace, int colPlace) public void setValues(int r, int c, int value) public double getBorder() public boolean wasConjugated() public boolean hasNextBit() public int getNextBit() public boolean isConjugationBit() public void setSegRowStart(int r) public void setSegColStart(int c)</pre>

Appendix C- Evaluation of Other Steganography Software

Steghide - Capacity Testing

Cover Image Size	Maximum Hiding Capacity Reported	Hiding Capacity (as a percentage of cover image size)
2.21MB	126KB	5%
14.4MB	615.1KB	5%
2.33MB	129.7KB	5%

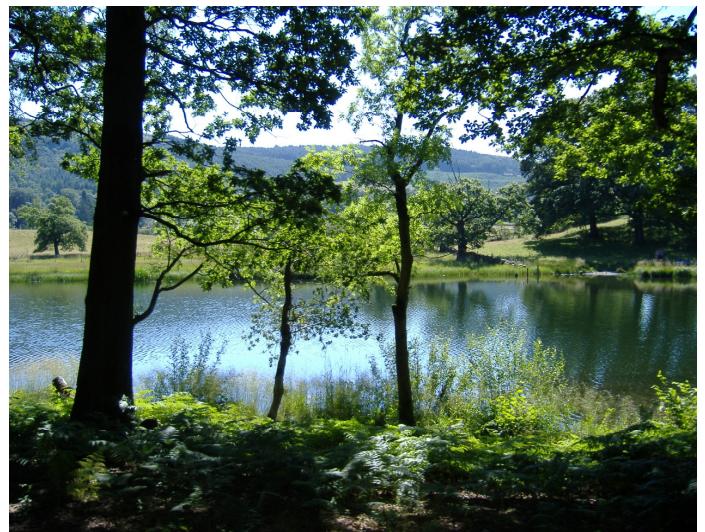
Steghide - Effectiveness Testing

Below are two images showing the comparison of the vessel image before and after embedding.

BEFORE



AFTER



Vecna - Effectiveness Testing

Below are two images showing the comparison of the image embedded, and extracted after embedding.

BEFORE



AFTER



Open Stego - Effectiveness Testing

Below are two images showing the comparison of the vessel image before embedding, and after embedding.

BEFORE



AFTER



QTech Hide and View – Capacity Testing

Cover Image Size	Maximum Hiding Capacity Reported	Hiding Capacity (as a percentage of cover image size)
1.37MB	458.75KB	33%
426KB	191.82KB	45%

QTech Hide and View – Effectiveness Testing

Below are two images showing the comparison of the vessel image before embedding, and after embedding.

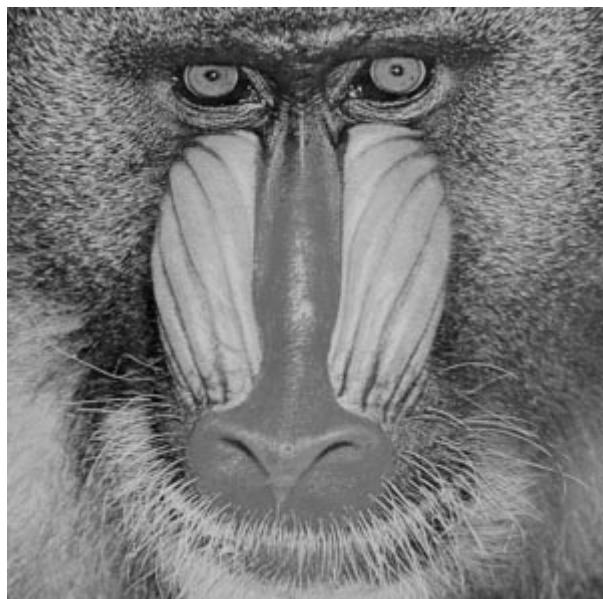
BEFORE



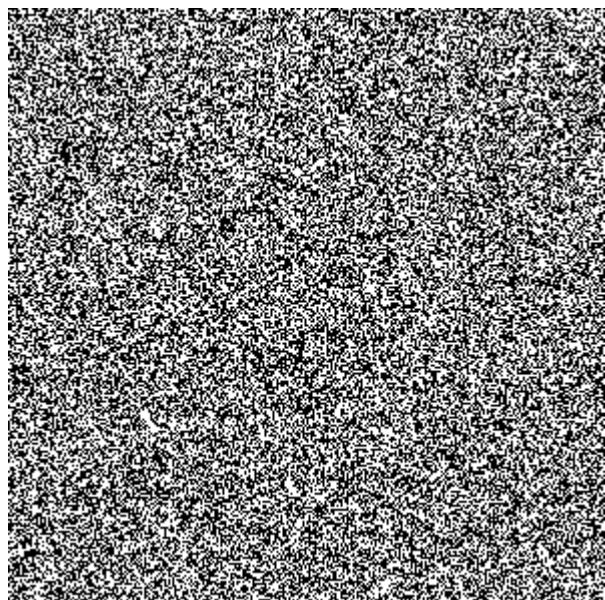
AFTER



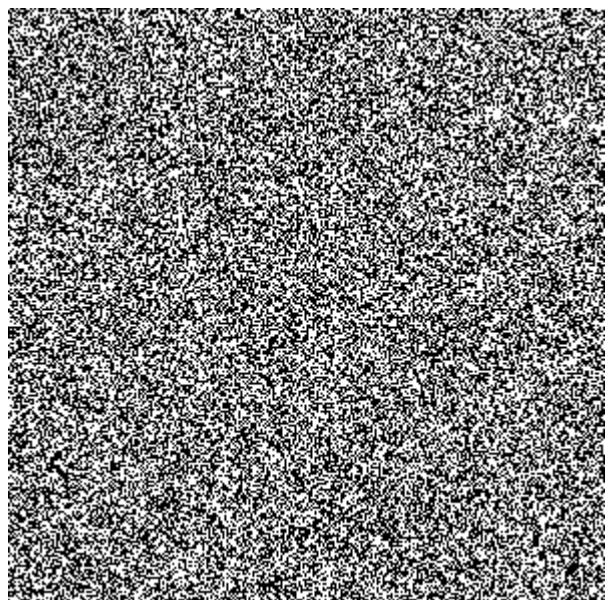
Appendix D – Bit Plane Decomposition



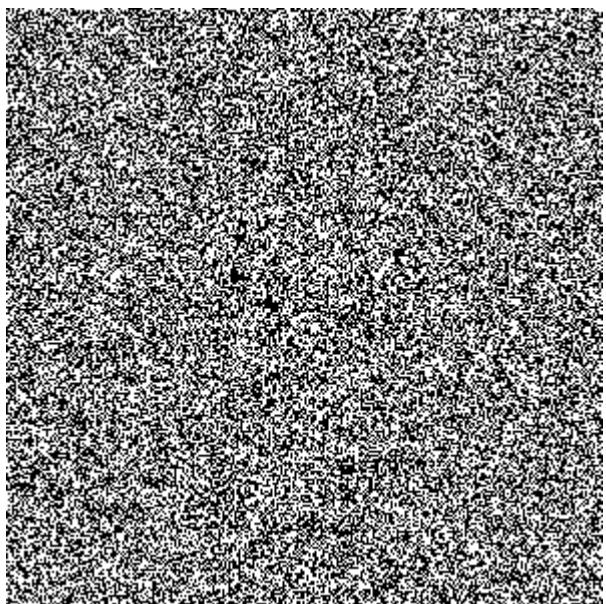
Original Image



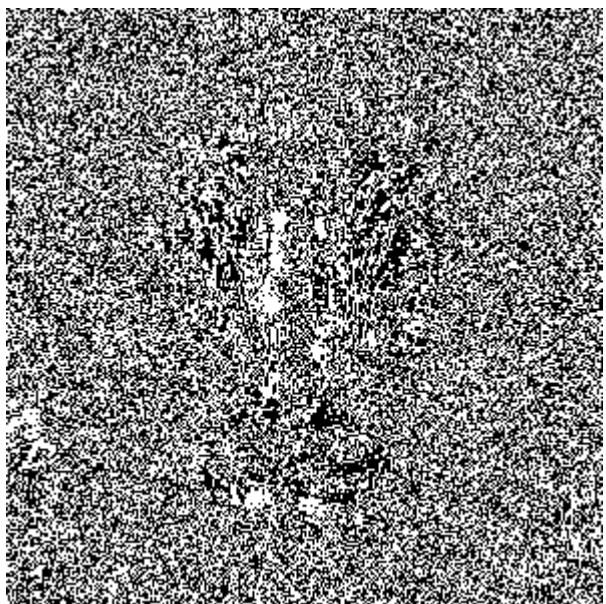
Plane 0



Plane 1



Plane 2



Plane 3



Plane 4



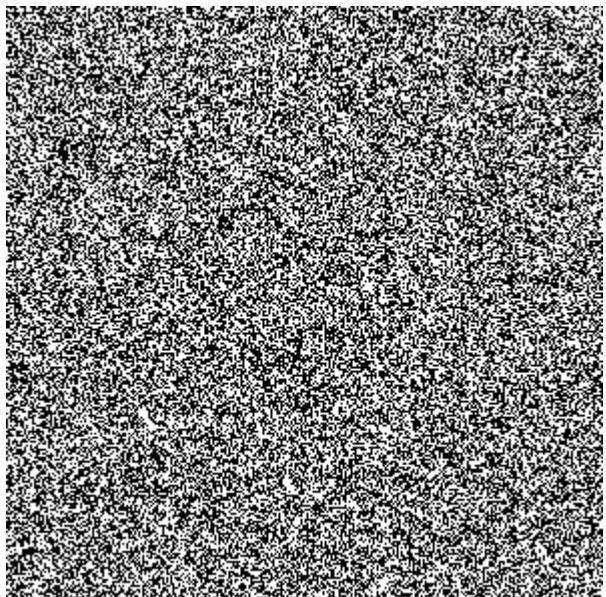
Plane 5



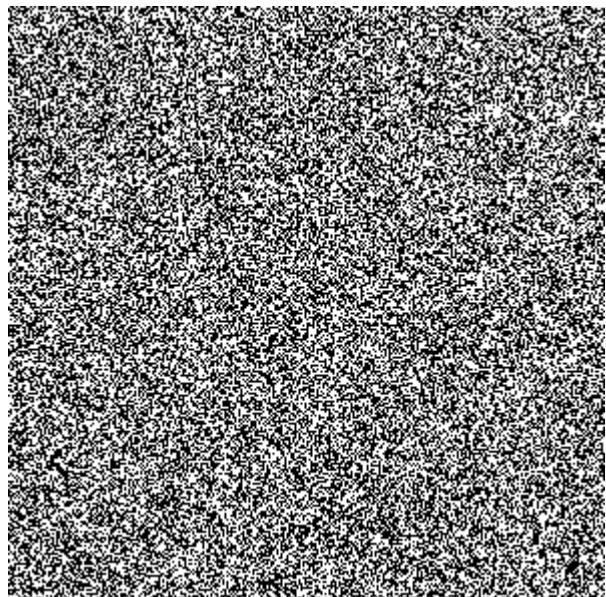
Plane 6



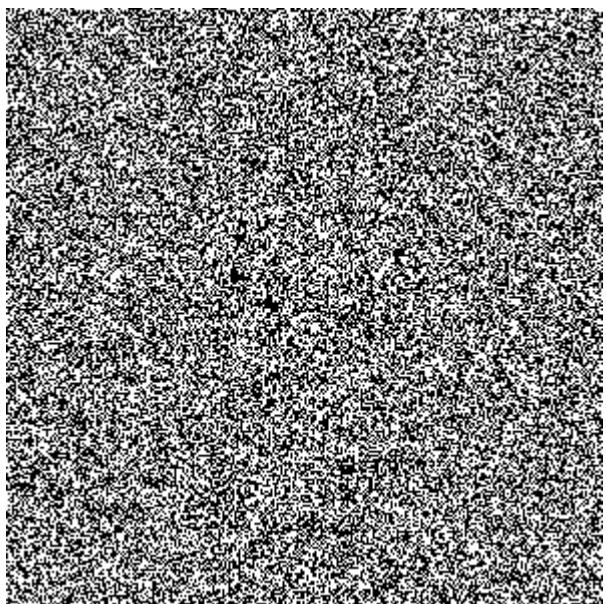
Plane 7



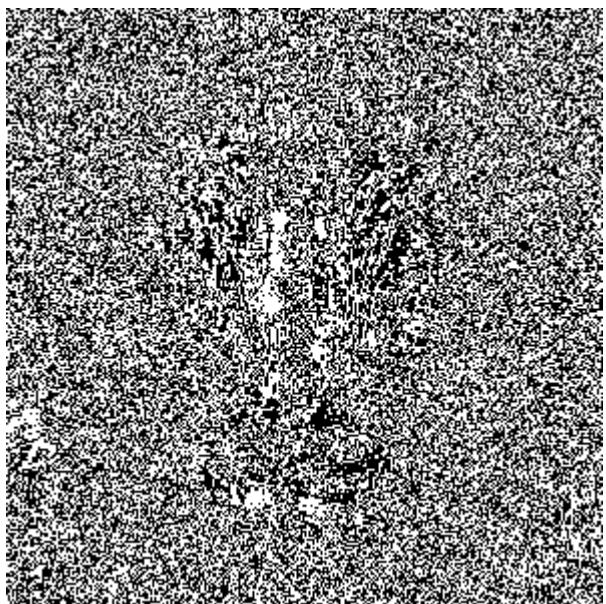
Plane 8



Plane 9



Plane 10



Plane 11



Plane 12



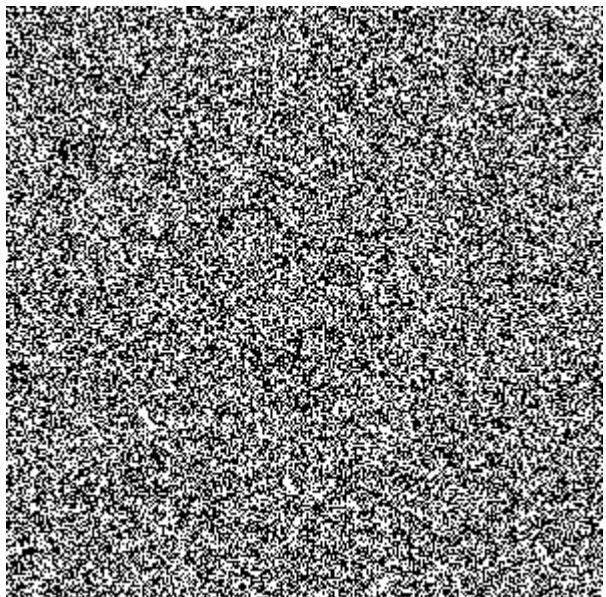
Plane 13



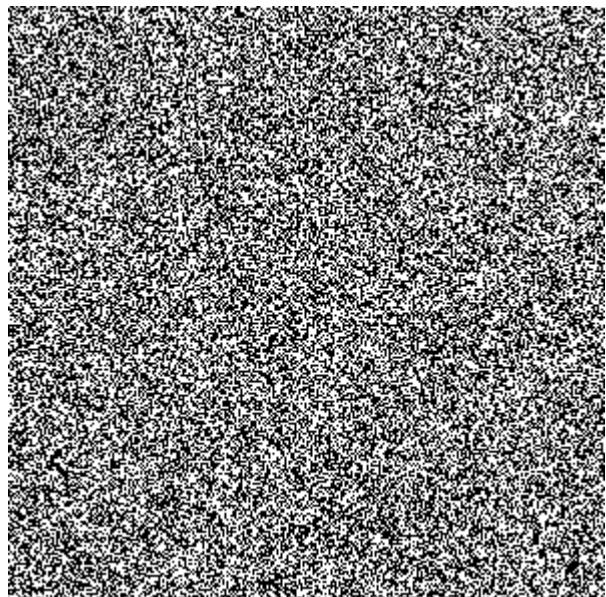
Plane 14



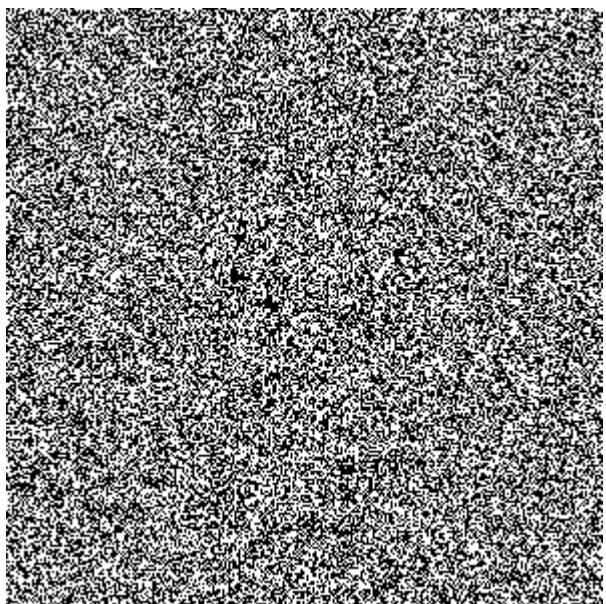
Plane 15



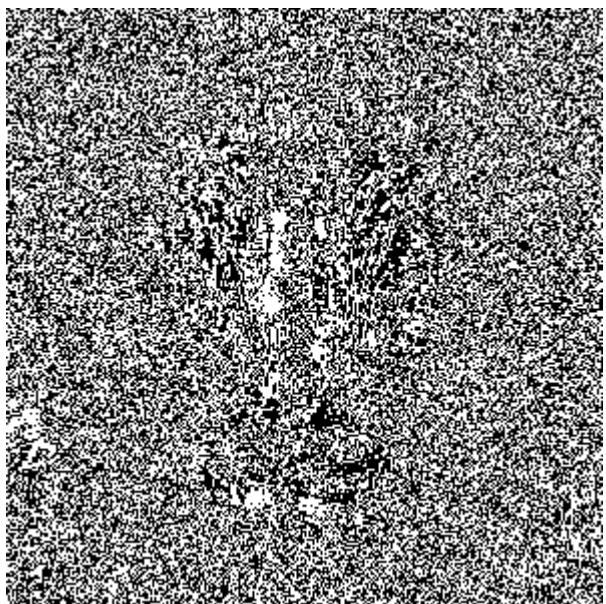
Plane 16



Plane 17



Plane 18



Plane 19



Plane 20



Plane 21



Plane 22



Plane 23

Appendix E – Testing Results

JUnit Test Cases Summary

In this section each test class is given (highlighted by italic lettering), under which the test class methods are detailed. Each test passed using the testing files provided on the attached CD.

TestBPCSExtractor

TestBPCSExtractor()
TestBPCSExtractor(String)
testConjugate()
testExtraction()
testMakeBytes()
testPopulateExtension()
testPopulateSize()

TestBPCSHider

TestBPCSHider(String)
testHide()
testInitialiseFileReader()
testInitialisePixelReader()
testIsNoiseLike()

TestCapacityEstimatorClass

TestCapacityEstimatorClass(String)
testCapacityCorrectness()

Test.FileReaderClass

Test.FileReaderClass(String)
testGetCurrentByte()
testGetExtention()
testGetFileSize()
testGetNextBit()
testGetNextBlock()
testGetSuccessBoolean()
testHasNextBit()
testHasNextBlock()
testHasNextByte()
testPopulateExtensionBits()
testPopulateSizeBits()
testSetCurrentByte()

testSetUpStream()

TestImageWriterClass

TestImageWriterClass(String)
testWrite()

TestLSB2Extractor

TestLSB2Extractor(String)
testExtraction()
testGetFileBits()

TestLSB2HeaderExtractor

TestLSB2HeaderExtractor(String)
testExtractExtensionBits()
testExtractSizeBits()
testPopulateExtension()
testPopulateSize()

TestLSB2HiderClass

TestLSB2HiderClass(String)
testHiding()
testSwap2LSB()

TestLSB4Extractor

TestLSB4Extractor(String)
testExtraction()
testGetFileBits()

TestLSB4HeaderExtractor

TestLSB4HeaderExtractor(String)
testExtractExtensionBits()
testExtractSizeBits()
testPopulateExtension()
testPopulateSize()

TestLSB4Hider

TestLSB4Hider(String)
testHiding()
testSwap3LSB()
testSwap4LSB()

TestLSBExtractor

```
TestLSBExtractor(String)
testExtraction()
testGetFileBits()
```

TestLSBHeaderExtractor

```
TestLSBHeaderExtractor(String)
testExtractExtensionBits()
testExtractSizeBits()
testPopulateExtension()
testPopulateSize()
```

TestLSBHiderClass

```
TestLSBHiderClass(String)
testCreateStream()
testHiding()
testSwapLSB()
```

TestPixelReaderClass

```
TestPixelReaderClass(String)
testGetBitPlane()
testGrabPixels()
testReadImage()
testReplaceBitPlane()
```

TestPlaneClass

```
TestPlaneClass(String)
testAnalysePlane()
testGet1DPlane()
testGet2DPlane()
testGetColChanges()
testGetMaxChanges()
testGetNextBit()
testGetNextSegment()
testGetRowChanges()
testHasNextBit()
testHasNextSegment()
testIsConjugationBit()
testReplaceSegment()
testWasConjugated()
```

Black Box Testing Results

GUI Functionality Test	Expected Result	Actual Result
Hide button pressed	A new window appears with the hiding options	As expected
Extract button pressed	A new window appears with the extraction options	As expected
View Help button pressed and within this, the various options are selected	A new window appears with the hiding details, upon pressing of each button, the appropriate text appears	As expected
Exit button pressed	Confirmation dialogue appears to ask the user if they wish to exit, if the user says yes, then the program exits.	As expected
Within hiding GUI – select cover image pressed	A JFileChooser appears which allows the user to select only an image file	As expected
Within hiding GUI- file select pressed	A JFileChooser appears which allows the user to select any type of file	As expected
Within hiding GUI- enter nothing for the name of the file and press hide	An error message appears indicating to enter a file name before proceeding	As expected
Within hiding GUI- enter a valid name out and press hide	Hiding proceeds and writes out a file with the name entered and extension .bmp or .png depending on the algorithm selected	As expected
Within hiding GUI- enter an extension for the file out	An error message appears prompting the user to try again	As expected
Within hiding GUI- enter a name out with an invalid character and press hide	An error message appears prompting the user to try again	As expected
Within hiding GUI- Select algorithm changed to different algorithms and press hide	Hiding is performed with different algorithms	As expected
Within hiding GUI- show capacity button is pressed	A message appears with the capacity for each of the algorithms for the image selected	As expected
Within hiding GUI- try to hide a file using an image format other than .bmp and press hide	An error message appears asking the user to select a .bmp file instead	As expected
Within hiding GUI- clear selections pressed	The image file and file to hide are cleared	As expected

Within hiding GUI- view help pressed	The help screen appears with complete functionality	As expected
Within hiding GUI- hide pressed with no file selected	An error message appears prompting the user to select a file	As expected
Within hiding GUI- hide pressed with no image selected	An error message appears prompting the user to select a file	As expected
Within hiding GUI- close hiding window pressed	The hiding GUI is closed, but the program does not exit	As expected
Within extraction GUI- select image pressed	A JFileChooser appears which allows the user to select only an appropriate image file	As expected
Within extraction GUI - enter nothing for the name of the file and press extract	An error message appears indicating to enter a file name before proceeding	As expected
Within extraction GUI - enter a valid name out and press extract	The extraction process completes and a file is extracted with the name as entered in the GUI	As expected
Within extraction GUI - enter an extension for the file out and press extract	An error message appears prompting the user to try again	As expected
Within extraction GUI - enter a name out with an invalid character and press extract	An error message appears prompting the user to try again	As expected
Within extraction GUI – different algorithms used can be selected	Extraction can be performed for different algorithms	As expected
Within extraction GUI – extract button pressed	extraction does not work if the wrong algorithm is selected	As expected
Within extraction GUI – Clear pressed	The image selected is cleared	As expected
Within extraction GUI – Help pressed	The help GUI appears with all related functionality	As expected
Within extraction GUI – close extraction window pressed	The extraction window closes, but the program does not exit	As expected

Algorithms functionality testing – the following table details the result of hiding and subsequently extracting files of the specified parameters with each of the algorithms. The purpose of this was to demonstrate the programs functionality. Some of the images resulting doubled up for use within the effectiveness section. The files and images used are provided on the attached CD. Varying image sizes were used in order to check that varying the size would not have a detrimental effect on the functionality of the system.

The use of a null file demonstrates functionality in terms of there being no bytes in the file, it is a special case scenario.

The small, large and maximum capacity parameters demonstrate that the program is able to function with varying levels of information hidden in an image. That is to say, it does not only function for small files.

The .doc and .jpg hiding operations were used to demonstrate that different types of file can be hidden and extracted successfully.

Within each file parameter, the same image was used to further demonstrate the resulting image using different algorithms. It was necessary in some instances to obtain files from the web in order to test the maximum capacity, as it was not possible to locate such files out with this resource.

File Hidden / Algorithm	least significant bit	two least significant bits	four least significant bits	BPCS
null file (hidden in Baseball.bmp, taken from freeimages.co.uk)	Hiding and extraction successful	Hiding and extraction successful	Hiding and extraction successful	Hiding successful, extraction unsuccessful
small capacity (19.5K .doc file used with 'Ferry' image, taken from freeimages.co.uk)	Hiding and extraction successful	Hiding and extraction successful	Hiding and extraction successful	Hiding and extraction successful
maximum capacity (wild flowers image used)	1.34MB gif file hidden – hiding and extraction successful	3.51MB .doc hidden-hiding and extraction successful	6.78MB .bmp hidden-hiding and extraction successful	10.6MB mp3 hidden - hiding and extraction successful

The most notable issue with the testing was that due to the way the BPCS algorithm uses FileOutputStream it will throw a 'FileOutputStream was null' error, and thus will not extract a null file.

Some images used were from www.freeimages.co.uk, these were the 'Ferry' image and the 'Baseball' image.

Appendix F - Original Images



Wild Flowers



Castle



Canoe



Cliff



Houses



More Houses



Ruins



Pebbles



Speedboats



View

Appendix G- Image Perception Evaluation Results

First Evaluation Results

Picture	Actual Hiding Details	User 1 (Online)	User 2 (Online)	User 3 (Online)	User 4	User 5	User 6
1	Max LSB	Probably none	No alteration	Probably none	A little alteration	No alteration	Probably none
2	Unaltered	No alteration	No alteration	Probably none	No alteration	Definitely altered	Probably none
3	Max BPCS	Definitely Altered	Definitely Altered	Definitely Altered	Definitely Altered	Definitely altered	Definitely altered
4	Unaltered	Probably none	Probably none	Probably none	A little alteration	No alteration	A little alteration
5	Unaltered	No alteration	No alteration	Can't tell	No alteration	Can't tell	Probably none
6	Small BPCS	No alteration	No alteration	No alteration	No alteration	Probably none	A little alteration
7	Small LSB	No alteration	No alteration	Can't tell	No alteration	Probably none	Can't tell
8	Max LSB4	A little alteration	No alteration	Definitely Altered	A little alteration	Probably none	Definitely altered
9	Unaltered	Probably none	Probably none	No alteration	No alteration	Probably none	Probably none
10	Small LSB2	No alteration	No alteration	Probably none	No alteration	Probably none	Can't tell
11	Max BPCS	A little alteration	Can't tell	A little alteration	Definitely Altered	A little alteration	Definitely altered
12	Max LSB2	Probably none	No alteration	Probably none	No alteration	None	Can't tell
13	Unaltered	Probably none	No alteration	No alteration	No alteration	None	A little alteration
14	High BPCS	Definitely Altered	Definitely Altered	Definitely Altered	Definitely Altered	Definitely altered	Definitely altered
15	Small LSB4	Probably none	Probably none	No alteration	Can't tell	None	Can't tell

Picture	Actual Hiding Details	User 7	User 8	User 9	User 10	User 11
1	Max LSB	A little alteration	A little alteration	No alteration	A little alteration	Probably none
2	Unaltered	Probably none	No alterations	No alteration	Probably none	Probably none
3	Max BPCS	Definitely altered				
4	Unaltered	Definitely altered	A little alteration	A little alteration	A little alteration	A little alteration
5	Unaltered	No alterations	A little alteration	A little alteration	No alteration	No alteration
6	Small BPCS	Can't tell	Probably none	A little alteration	No alteration	Probably none
7	Small LSB	Can't tell	No alterations	No alteration	No alteration	No alteration
8	Max LSB4	A little alteration	Definitely altered	Definitely Altered	Definitely Altered	Definitely Altered
9	Unaltered	Can't tell	Probably none	Probably none	Definitely Altered	Probably none
10	Small LSB2	A little alteration	A little alteration	Can't tell	A little alteration	Probably none
11	Max BPCS	Definitely altered	No alterations	Definitely Altered	Definitely Altered	Can't tell
12	Max LSB2	A little alteration	A little alteration	Can't tell	A little alteration	Probably none
13	Unaltered	Can't tell	A little alteration	A little alteration	Probably none	Probably none
14	High BPCS	Probably none	Definitely altered	Definitely Altered	Definitely Altered	Definitely Altered
15	Small LSB4	A little alteration	Probably none	Definitely Altered	Probably none	Probably none

Second BPCS Evaluation Results

Image	Actual Hiding Details	User 1	User 2	User 3	User 4	User 5
1	63% BPCS	Altered	Unsure	Altered	Unaltered	Unaltered
2	43% BPCS	Unaltered	Unaltered	Unaltered	Unaltered	Unaltered
3	56% BPCS	Altered	Altered	Altered	Altered	Altered
4	50% BCPS	Unaltered	Unaltered	Unaltered	Unaltered	Unaltered
5	72% BPCS	Altered	Unsure	Altered	Unaltered	Unaltered
6	Unaltered	Unaltered	Unaltered	Unaltered	Unaltered	Unaltered

Appendix H – Perception Evaluation Images

First Evaluation Images



Image 1



Image 2



Image 3



Image 4



Image 5



Image 6



Image 7



Image 8



Image 9



Image 10



Image 11



Image 12



Image 13



Image 14



Image 15

Second Evaluation Images



Image 1



Image 2



Image 3

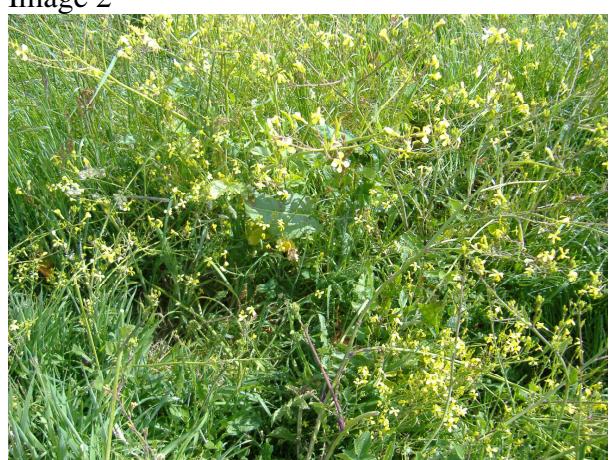


Image 4



Image 5



Image 6

Appendix I – List of Figures

FIGURE 2.1.1	4
FIGURE 2.2.1	5
FIGURE 2.3.1	6
FIGURE 2.3.2	7
FIGURE 2.4.1	8
FIGURE 2.4.2	9
FIGURE 2.4.3	9
FIGURE 2.5.1	10
FIGURE 2.5.2	10
FIGURE 3.3.1	13
FIGURE 3.3.2	14
FIGURE 5.1	18
FIGURE 5.2.1	22
FIGURE 5.2.2	23
FIGURE 5.2.3	23
FIGURE 5.2.4	23
FIGURE 8.3.1	35
FIGURE 8.3.2	35
FIGURE 8.4.1	36
FIGURE 8.6.1	38
FIGURE 9.1.1	41
FIGURE 9.1.2	42
FIGURE 9.1.3	42
FIGURE 9.1.4	43
FIGURE 10.2.1	46
FIGURE 10.2.2	47
FIGURE 10.2.3 – ORIGINAL IMAGE	47
FIGURE 10.2.3 – 73% HIDDEN	48
FIGURE 10.2.4 – 50% HIDDEN	48
FIGURE 10.2.5 – 73% HIDDEN CLOSE UP	48
FIGURE 10.4.1	51
FIGURE 10.4.2	51
FIGURE 10.4.3	52
FIGURE I1	96
FIGURE I2	96

Appendix I – Image Histograms

Histograms of images show the distribution of the colour components within the image. (Auer, 2007) The two histograms which follow were taken to show the effect that hiding information could have on the original image colour distribution. Figure I1 provides the distribution before hiding 50% within the image and figure 10.4.5 shows the histogram after 50% had been hidden using the BCPS algorithm.

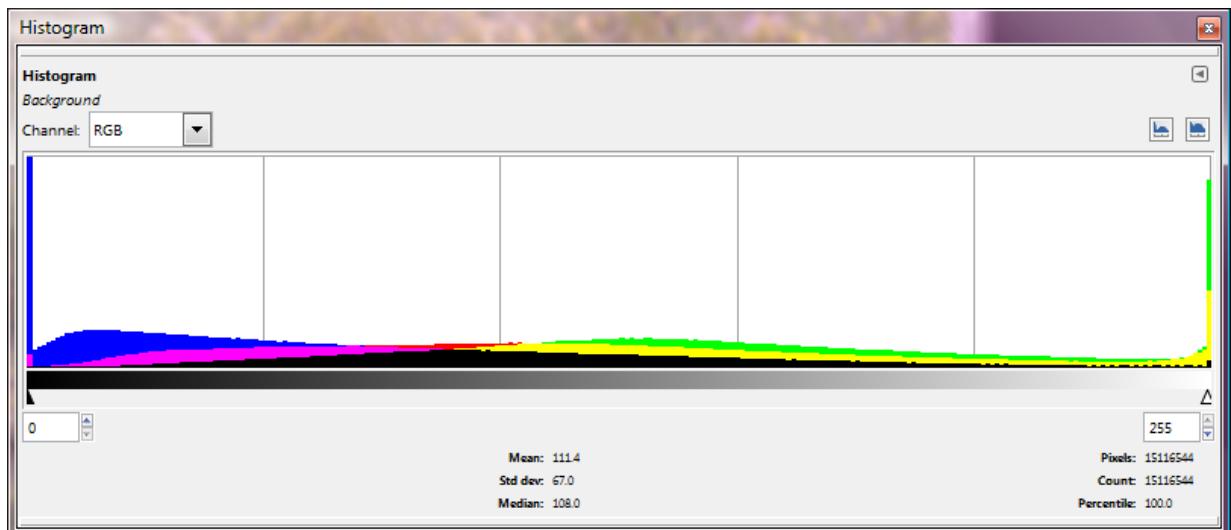


Figure I1

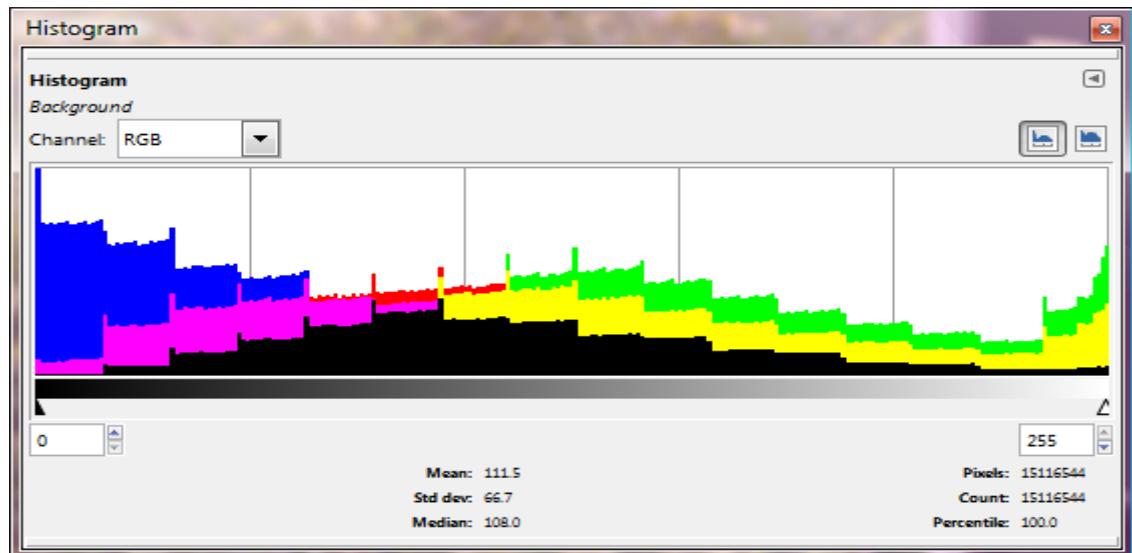


Figure I2

Bibliography

- Auer, B. , 2007, *Working with Image Histograms [Online]* , Available at:
<http://blog.epicedits.com/2007/04/14/working-with-image-histograms/> [Accessed 7/09/2009]
- Flanagan, D. , 2005, *Java in a Nutshell* , 5th Edition, O'Reilly
- Grossbart, Z., 2007, *Hakito Ergo Sum Steganography Software [Online]*, Available at:
<http://www.zackgrossbart.com/hackito/2007/11/30/how-to-keep-a-secret-secret/>
[Accessed 12/03/2009]
- Hempstalk, K. , 2005, *Digital Invisible Ink Tool Steganography Software [Online]*, Available at: <http://diit.sourceforge.net/> , [Accessed 12/03/2009]
- Hetzl, Stefan, 2003, StegHide Software [Online], Available at: <http://steghide.sourceforge.net/>
[Accessed 01/02/2009]
- Johnson, N., Jajodia, S, 1998, Exploring Steganography: Seeing the Unseen, *Computer (published by IEEE)*, Volume 31, Issue 2, pages 26-34
- Joshi, Madhuri A., 2006, *Digital Image Processing: An Algorithmic Approach*, Prentice-Hall
- Kamata, S. et al, 1995, Depth-First Coding for Multivalued Pictures Using Bit-Plane Decomposition, *IEEE Transactions on Communications*, Volume 43, No. 5
- Kawaguchi, Eiji et al, 1998, A Concept of Digital Picture Envelope for Internet Communication, *8th European-Japanese Conferences on Information Modelling and Knowledge Bases*, pages 343-350
- Kawaguchi, E., Eason, R., 1998, Principles and applications of BPCS-Steganography, *SPIE's International Symposium on Voice, Video, and Data Communications*
- Kawaguchi, Eiji, 2008, *Principle of BPCS-Steganography [Online]* Available at:
<http://www.datahide.com/BPCSe/principle-e.html> [Accessed 10/1/2009]
- Kawaguchi, E., 2009, *QTech Steganography Software [Online]*, Available at:
<http://www.datahide.com/BPCSe/QtechHV-download-e.html> [Accessed 12/03/2009]
- Kipper, G., 2004, *Investigator's Guide to Steganography*, CRC Press
- Makabe, S. et al, 2007, Implementation of Modified Level Transformation for Bit-Plane Watermarking, *TENCON 2007 - 2007 IEEE Region 10 Conference*, pages 1-4

Niimi, M. et al, 1997, An Image Embedding in Image by Complexity Based Region Segmentation Method, *Proceedings of the 1997 International Conference on Image Processing (ICIP '97) Volume 3*, pages 74-77

Niimi, M. et al, 1999, Steganography Based on Region Segmentation With a Complexity Measure, *Systems and Computers in Japan*, Volume 30, No. 3

Nozaki, K. et al, 1998, A Large Capacity Steganography Using Color BMP Images, *Lecture Notes in Computer Science; Vol. 1351*, pages 112-119

Rebah, K. , 2004, Steganography – The Art of Hiding Data , *Information Technology Journal*, Journal 3, pages 245-269

Stallings, William, 1998, Cryptography and Network Security: Principles and Practice, 2nd edition, Prentice Hall

Summary: This book details cryptography and security related topics. It briefly discusses steganography and provides a small overview of it.

Sommerville, I. , 2006, *Software Engineering* , 8th Edition, Addison Wesley

Strauss, *Vecna Steganography Software [Online]*, Available at: <http://www.uni-koblenz.de/~strauss/vecna/> [Accessed 12/03/2009]

Vaydia, Samir, 2009, *OpenStego Steganography Software [Online]*, Available at: <http://openstego.sourceforge.net/> [Accessed 10/03/2009]

Wang, H., Wang, S., 2004, Cyber Warfare: Steganography vs. Steganalysis, *Communications of the ACM*, Volume 47, No. 10, pages 76-82

Westfield, A., Pfitzmann, A., 1999, Attacks on Steganographic Systems, *Lecture Notes in Computer Science; Vol. 1768*, pages 61 - 76