General Framework for Watermarking of Compressed Bitstreams: Data Embedding in Code Space

A Thesis Research Paper
Presented to the Faculty of
The Department of Electrical and Computer Engineering
Villanova University

In Partial Fulfillment
of the Requirements for the Degree of
Master of Computer Engineering

By
Robert J. Berger II

August 31st, 2004

Under the Direction of Dr. Bijan G. Mobasseri
Acknowledgements
This research was supported in part by grant #FA8750-04-1-0185 awarded by the US Air Force Research Laboratory/IFEC, Rome, NY.
This thesis paper is available for research purposes at Falvey Memorial Library.
Abstract

Algorithms that perform data hiding directly in the compressed domain, without the need for partial decompression or transcoding, are highly desirable. Many current watermarking algorithms embed the watermark in the frequency domain which requires at least partial decompression of the cover signal. Watermarking directly in the bitstream allows for fast, real-time applications. Additionally, the algorithm should not increase the file size of the cover signal and removing the watermark should restore the original cover signal exactly. Most digital mediums are already in compressed format prior to watermarking; this implies that file size or bandwidth is important for their use. Thus, any file size increase due to watermarking degrades the usefulness of cover signal. Similarly, if the watermark is not lossless then there is again some degradation to the value of the cover signal. Therefore, a watermarking algorithm that embeds directly in the compressed, is file-size preserving and lossless would be highly regarded with respect to many different applications.

A novel lossless compressed domain watermarking algorithm is proposed. The algorithm itself is general, relying on the concept of entropy encoding which is common to all popular compression algorithms, both for still images and video. This work is based on the idea that only a limited amount of a possible code-space is actually used for any specific code. Through watermarking particular variable length codes (VLCs), it is possible to map these VLCs outside of the given codespace. If the mapping is one-to-one then the watermark can be removed restoring the original cover signal exactly. The watermarking process is done by flipping a single bit in each VLC that is watermarked. Since, no bits are added or subtracted there should not be any file size change in the cover signal. Also, by limiting watermarking to the VLC portion of the compressed bitstream, the overall file structure of the cover signal is unaltered.

To map VLCs outside of a given codespace, the given codespace must not use all possible codes. This is generally the case for a perfectly entropy coded signal. However, very few signals achieve the perfect encoding. This work shows that the algorithm is effective on any codespace that is less than perfectly entropy encoded such as reversible variable length codes (RVLCs). RVLCs are similar to traditional VLCS except that RVLCs are instantaneously decodable in both the forward and backward directions. This guarantees that any RVLC code is not perfectly entropy encoded and the proposed algorithm will be effective on it. A new concept of codeword pairing is also presented in this work which conceptually expands the total codespace allowing for a higher watermark capacity for a given cover signal.

As proof of concept, the general algorithm was applied to the JPEG image standard. Although JPEG is one of the older standards, it is still one of the most widely employed. The algorithm was extremely successful when adapted to JPEG. A little known fact about most JPEG images is that they use example Huffman tables from the JPEG standard instead of creating custom Huffman tables for each individual image. This translates into less efficient entropy encoding which can be exploited by the proposed algorithm. Watermarking the JPEG image was lossless, file size preserving, and format compliant. Additionally, the algorithm was able to mask the watermark so that the watermarked image appeared identical to the original, even when viewed by standard decoders (unaware of the watermark).
# Table of Contents

## 1.0 INTRODUCTION TO WATERMARKING .................................................................1

1.1 BACKGROUND .................................................................................................1

1.2 CHARACTERISTICS OF WATERMARKS............................................................2
  1.2.1 Robustness.................................................................................................3
  1.2.2 Transparency or Fidelity............................................................................3
  1.2.3 Domain.......................................................................................................4
  1.2.4 Tamper Resistance.....................................................................................4
  1.2.5 Computational Cost..................................................................................5
  1.2.6 False Positive/Negative Rate.................................................................6
  1.2.7 Mathematically Lossless Vs. Lossy.........................................................6
  1.2.8 File Size Change.....................................................................................6
  1.2.9 Capacity/Embedding Rate.......................................................................7
  1.2.10 Standard-Compliant.........................................................................7

1.3 COMMON APPLICATIONS ..................................................................................7
  1.3.1 Identification/Authentication.................................................................7
  1.3.2 Security/Steganography..........................................................................8
  1.3.3 Distribution Control...............................................................................8
  1.3.4 Metadata.................................................................................................8

1.4 EXISTING TECHNIQUES .......................................................................................8
  1.4.1 Uncompressed Spatial Domain...............................................................8
  1.4.2 Partially Uncompressed Frequency Domain......................................9
  1.4.3 Fully Compressed VLC Domain..........................................................10

## 2.0 GENERAL LOSSLESS COMPRESSED DOMAIN WATERMARKING..................10

2.1 INTRODUCTION .................................................................................................10

2.2 THEORY .............................................................................................................11
  2.2.1 Entropy Encoding..................................................................................11
  2.2.2 Codespace..............................................................................................11
  2.2.3 Codeword-pairing and Watermarking..............................................12

2.3 CODESPACE .....................................................................................................12

2.4 CODEWORD-PAIRING ....................................................................................14

2.5 BINARY CODE TREE ......................................................................................15

2.6 EXAMPLE .........................................................................................................17
  2.6.1 Form Codeword-pairs and Binary Tree..............................................17
  2.6.2 Determine Watermark Mapping.........................................................17
  2.6.3 Encoding..............................................................................................20
  2.6.4 Decoding................................................................................................21

2.7 RESULTS ..........................................................................................................22

## 3.0 JPEG APPLICATION .........................................................................................23

3.1 INTRODUCTION TO JPEG ..............................................................................23
  3.1.1 Overview..................................................................................................24
  3.1.2 8x8 Pixel Blocks....................................................................................24
  3.1.3 DCT.........................................................................................................24
  3.1.4 Quantization.........................................................................................25
  3.1.5 Entropy Encoding..................................................................................25

3.2 COMPRESSION INEFFICIENCY .......................................................................27

3.3 CODESPACE, VLC MAPPING, AND SYNCHRONIZATION..............................28

3.4 EXPLOITING INEFFICIENCY ...........................................................................28

3.5 JPEG WATERMARKING ..................................................................................29
  3.5.1 Parsing a JPEG image.........................................................................29
  3.5.2 Determine Watermarkable Used VLCS.............................................31
1.0 Introduction to Watermarking

This section is intended to give the reader a general overview of watermarking multimedia content. Also, this section will provide definitions for common watermark characteristics that are used to compare different techniques.

1.1 Background

The term “watermarking” has been used for centuries to describe the method of placing a seal on paper money and documents. The seal was designed to be easy to identify, but difficult to reproduce. The purpose of the seal was to provide protection against counterfeits and similarly to act as identification of authenticity. Watermarking can be generally defined as adding additional information to a cover signal. The cover signal in this example is the money or documents. The phrase “digital watermarking” has gained attention as the cover signal has changed to multimedia content such as digital pictures and video as these have become more prevalent. At first, the goals of digital watermarking were similar to paper watermarking however, these goals have quickly expanded due to the wide range of applications of digital media.

To understand watermarking, knowledge of the general application framework is necessary. While the focus of this work has been on digital images and video, many aspects of watermarking can be, and are, applied to other types of multimedia such as audio. There are four general steps that occur during any useful watermarking process. The first step is identifying the cover signal—what is going to be watermarked, sometimes called medium. This of course is crucial in deciding what method to use to embed the watermark efficiently. The next step is actually embedding the watermark into the cover signal. Generally this step receives the most attention as there is the most flexibility for developing different algorithms that vary on the specific application and cover signal. Also, it is implied in this step that what information is being embedded is decided; i.e. what the watermark is. The third step is transmission. A watermark in most cases is not very useful if the cover signal is not transferred to other users or locations. During this step there is the possibility of various distortions during typical transmissions such as analog-to-digital conversions, rotations, scaling, lossy compression or decompression, etc. Distortions that are expected to occur during transmission should be accounted for in choosing the embedding algorithm. Additionally, there may be intentional attacks to either remove or distort the watermark to the extent that it is unreadable. Finally, the last step is decoding the watermark. At this point it is intended that the watermark survive transmission and be able to be recognized by a decoder.

There are typically three types of decoders representing three different types of users. The most obvious is the user for which the watermark is intended. This user has explicit knowledge of the watermarking algorithm, but may or may not have access to the original, un-watermarked cover signal. The decoder for this user should be able to extract the watermark from any cover signal as long as the watermarked signal
has not been maliciously altered during transmission (any normal distortions during transmission should be accounted for in choosing the embedding algorithm). Another type of user is one that has interest in the cover signal, but not the watermark. Movies currently have watermarks embedded in them for various reasons, including identification of the location the movie is being shown (in the case of theatre viewing). The user in this case would be the patron who simply wants to watch the movie. The decoder in this case would be the projector that is displaying the movie. The importance here is that decoders for this user are unaware of the watermark. This can have various implications, but the most important of which is that the decoder can not take steps to mask visual distortions created by the watermark. Therefore in this example it is the watermark embedders responsibility to ensure that the watermark does not devalue the cover signal through noticeable visual artifacts. The final type of user and decoder is one that wishes to attack the process. Typically this user does not have access to the original, un-watermarked cover signal so their goal may to remove the watermark and obtain a close copy of the original. Other objectives of this user may to either decipher the watermark, because important information is embedded in the watermark or to be able to copy the watermark into a different cover signal to create a forgery. In many cases this user may even have knowledge of the embedding algorithm. The watermark then is commonly protected through encryption or a random key.

Before discussing the characteristics of watermarks, examining the cover signal is necessary. For a cover signal to be watermarked there must be some redundancy inherent within the signal to begin with. In many cases when dealing with images and video, this redundancy comes from the fact that the human visual system (HVS) is less sensitive to certain characteristics of images. For example, the HVS is less sensitive to high frequency information. Therefore, watermarking a cover signal in the frequency domain in the high frequency range is a common approach to watermarking images. This problem becomes more challenging when the image has already been compressed. Typical compression algorithms such as JPEG are aware of the HVS and eliminate much of the high frequency information during compression. The challenge for watermarking algorithms is then to locate any redundancies left in the cover signal.

1.2 Characteristics of Watermarks
Some of the first digital watermarks were simply superimposing a logo or name over the original image for purposes of identification. However, shortly after this use it became evident that this was not enough because a fixed location logo can be easily cropped out, and depending on the location, result in no loss of value to the cover image; i.e. a television station logo is placed in the bottom right corner of the broadcast, but cropping out this corner does not affect the vast majority of the content. With this knowledge, research has been on going into how to embed the watermark, logo in this example, in a manner that may not easily be removed, at least without devaluing the cover image.

Currently the range of uses include broadcast monitoring, embedding secret messages (steganography), tamper resistance, and copy control\(^1\). Broadcast monitoring was accomplished by placing a signature within a watermark that would identify a specific commercial for example. Then, the company paying for
the commercial could have a program monitoring the broadcast channel and count how many times their specific watermark was found. This is essentially an independent verification that the commercial is being broadcast the agreed upon number of times. It is important to note that one of the essential criteria for this type of watermarking is imperceptibility for the viewer. If the watermark degrades the quality of the image, the watermark loses value; therefore the watermark is desired to be transparent. Also, if the broadcast channel is over air, there may be interference such as Gaussian additive white noise to the watermark, but the watermark must still be detectable by the program monitoring the channel. The amount a watermark is affected by general signal processing interference is referred to as robustness. Transparency and robustness are just two of many characteristics inherent to all digital watermarks. Other characteristics include domain, tamper resistance, fidelity, computational cost, false positive rate, lossy, and file size change. Depending on the specific application of the watermark each characteristics importance may vary from irrelevant to critical. In many cases even critical characteristics may be contradictory—meaning that to improve one characteristic it may be necessary to degrade the other. Often times trade-offs have to be made which result in a useful watermark that does not necessarily meet every criteria.

1.2.1 Robustness

The usefulness of a watermark is dependent on it surviving transmission. Robustness is a measure of how well a watermark accomplishes this goal. It should be understood that any watermark can have varying degrees of robustness to different signal processing operations. A watermark may survive translation, but not rotation, for example. What a watermark needs to be robust against is strictly tied to the specific application and cover signal. If the application was designed for JPEG images, then the watermark should be able to survive lossy compression and decompression with respect to the JPEG codec. Also, cropping, scaling, and other common photo editing techniques should be accounted for. Conversely, for a JPEG application robustness against digital-to-analog conversion may not be necessary. Clearly a watermark does not need to be robust against every possible distortion—doing so may result in a loss of efficiency or unnecessary degradation of other important characteristics.

On the opposite side of the spectrum, some applications may actually want a watermark to be fragile to some forms of signal processing. Such applications include legal or medical watermarking where having an untampered cover signal is of the utmost importance. Watermarks for these applications are intentionally designed to break if the cover signal is altered in any way.

1.2.2 Transparency or Fidelity

When viewing a watermarked medium the watermark should not degrade the image too much. Of course deciding what is “too much” is dependent on the application. Typically transparency is defined qualitatively as noticeable to a human viewer, in the cases of images and video, and quantitatively as peak-signal-to-noise-ratio, PSNR. While PSNR is usually proportional to what the HVS perceives as visual artifacts, a high PSNR in itself does not necessarily mean acceptable visual degradation. Both PSNR and
HVS are typically used as relative figures within a given framework. They are useful for comparing the same algorithm on the same image, but different sized watermarks for example. It is expected that the larger the watermark (increased number of bits) the lower the PSNR and the higher the visual impact to the HVS. Another aspect of transparency is statistical transparency. Watermarks are often used in a “secret message” scenario. Here, not only is the watermark required to be transparent to the HVS in display, the cover signal should not have its statistics modified to the point where an analysis would reveal that a watermark is even present.

1.2.3 Domain

Domain refers to the status of the cover signal when the watermark is embedded. The scope of watermarking in this text is digital multimedia; therefore the cover medium is almost always in a compressed form to begin with—such as JPEG, MPEG, and H.263 among others. For watermarking purposes, the cover medium could be decompressed back to pixel values (as when being displayed), partially decompressed, or left as is.

Fully decompressing the cover signal would make inserting the watermark less difficult, because there is so much redundancy in most uncompressed images. However, the watermark would have to be robust enough to survive compression for transmission. The assumption here is that whatever the cover signal started as is how the signal will be transmitted. Also, decompression and recompression may be a slow process, especially if the application were dealing with streaming video.

The solution to these problems would be to watermark directly in the compressed domain. Take the cover medium and simply watermark it as is. This would be extremely fast and in many cases would skip the requirement to be robust against compression since it already is. Of course when dealing with the compressed domain there is very little redundancy left in the cover signal since the compression algorithm squeezed as much redundancy out to minimize the size of the cover signal.

Watermarking in the partially decompressed domain is the most common form of watermarking. Partial decompression is generally fast enough for most applications and when partial decompression results in the frequency domain, redundancies can usually be located.

1.2.4 Tamper Resistance

There are four predominant types of attacks on watermarks: active, passive, collusion, and forgery. An active attack is one in which an attacker’s goal is to remove the watermark. In some cases this means removing all traces of the watermark, but in many instances simply rendering the watermark undetectable is considered a successful attack. A passive attack is defined as an attacker trying to determine if a watermark is present or not in the cover signal. This would only be regarded during covert transmissions in the form of a watermark or some other secret message application.
Collusion attacks often prove the most troublesome to a significant number of watermarking applications. Consider watermarks used to track photographs. It would not be unusual for an artist to place a watermark within the photo as a proof of ownership before selling it. At first it would seem only necessary to have the same watermark in every copy sold. However, what if illegal (meaning copies not sold by the artist) began circulating on the internet? While the watermark would still provide proof of ownership people may be less likely to buy something if they could download it free—even if it were illegal. The solution then is to embed a unique watermark in each copy of the photograph sold. Then if copies begin showing up on the internet the artist could trace which client was illegally distributing the photo based on its unique watermark. A collusion attack then is when an attacker gains multiple copies of the same cover medium, in this example photograph, that contains different watermarks. By comparing the multiple copies, the attacker can locate differences which represent the watermark. If enough copies of the photograph are obtained, the watermark can be removed. Another form of collusion is if the same watermark is used in different cover signals. Continuing the photographer example, the photographer used the same watermark as proof of ownership for all of his photographs. An attacker could then, by obtaining different photographs all with the same watermark, determine what the watermark was and then either remove it from the existing photographs or add the same watermark to a new photograph, a forgery attack. While a photographer may be upset that a photograph that was not his was portrayed as if it were, a forgery attack can become much more serious if the watermark were being used for other forms of identification. A more serious example would be for security badges for a company; forgery attacks could allow an attacker to impersonate an employee.

1.2.5 Computational Cost

Computational cost is simply how much computing is necessary to embed/decode the watermark. There are sometimes two parts to the computational complexity for embedding and as many as three parts for decoding. During embedding there can be what is called offline complexity. Although offline can be somewhat of a misnomer, this refers to analysis of the cover signal prior to actually embedding. With some applications it is acceptable to have a high offline complexity as long as the complexity of embedding the actual watermark is low enough. Without going into too much detail, an application that would fit this description would be one in which video is going to be taken of similar scenery. It may be necessary to examine the first frame of the video to set initial parameters for the watermark embedding process, but examination of each of thousands of frames is not necessary.

For decoding, the three types of processes that may occur are detection, extraction, and removal. Each process could have an independent computational complexity. Detection generally is the fastest—simply determining if a watermark is present. Extraction is determining what the watermark actually is. This could be considered as interpreting what the watermark represents. And finally, removal is the complexity of restoring the cover signal, as best as possible, to originality prior to embedding the watermark.
1.2.6 False Positive/Negative Rate

False positive rate is a measure of how many times a cover signal without an expected watermark is perceived as containing one and false negative rate is a measure of how many times a cover signal that did contain a watermark was not classified as such (synonymous with robustness). Some watermarking algorithms are designed to be a general mathematical algorithm. However, even with a general algorithm it must be applied to a specific cover signal of a specific type. False positive rates for an algorithm applied to different compression techniques (i.e. JPEG, TIFF, GIF, etc.) can give one indication of what the algorithm is best suited for.

In many cases this measure is used in conjunction with the robustness or tamper resistance characteristics. After a cover signal without a watermark undergoes signal processing or an attack of some form, how often is a watermark perceived?

1.2.7 Mathematically Lossless Vs. Lossy

When embedding a watermark, by definition the cover signal is altered. The most common algorithms for video and images replace the high frequency information of the cover signal with the watermark. If the high frequency information can never be recovered this process is termed lossy, referring to the loss of information. In contrast, if the information is displaced, or overwritten, by the watermark but the original cover signal can be recovered exactly, then this is called mathematically lossless, or just lossless. For the example of images, using a lossy algorithm may be acceptable because high frequency changes is not easily noticed by the HVS, however if the image were used in a legal or medical setting even the slightest loss of information or change to the original may not be allowed. In the second case a lossless algorithm must be employed.

1.2.8 File Size Change

File size change is what it sounds like, a measure of the difference in the number of bits of the cover signal prior to watermark insertion and after. With cover signals that deal with transfer rates as opposed to a fixed file size, this characteristic can be represented as a percent of the bandwidth or change in the transfer rate, again before and after watermark insertion.

In practice, since many cover signals are already compressed there is the assumption that minimal file size is important. Therefore, algorithms that dramatically increase file size, relative to the original file, are considered below standard. A small fixed size increase regardless of the actual cover signal is considered acceptable. While a small percent increase is sometimes overlooked for certain applications, because multimedia files are already quite large even a 1% increase could result in unacceptable file size increase, especially in cases dealing with a restricted bandwidth. Of course preserving the original file size is the goal.
1.2.9 Capacity/Embedding Rate
Capacity refers to how many watermark bits can be embedded in a cover signal, and similarly for transmission applications how many watermark bits per second can be embedded in the cover signal. Whenever discussing capacity for an algorithm, a context must be given. Even for the same algorithm applied to the same compression standard, different capacities can occur based on the specific cover signal. Consider JPEG images for instance. If the image is a natural image or computer generated may have an affect on capacity as well as the size of the image. Additionally, context for capacity is the affect on other key characteristics of the watermark such as robustness and transparency. It may be possible to watermark every pixel of an image, but then the image may be completely unrecognizable. Alternatively, embedding a single bit in an image would have little effect. Capacity must take into account all of the criteria that is important to the specific application.

1.2.10 Standard-Compliant
Watermarking refers to embedding data within a cover signal. Cover signals, in the case of multimedia, typically are already contained within a standard such as MPEG or JPEG. If watermarking the cover signal renders the signal unreadable by standard-compliant viewers, then watermarking can be thought of nothing more than a form of encryption. The majority of applications of watermarking video and images require that the underlying standard of the cover signal not be altered by the watermark. This requirement only refers to viewing the cover signal as it was originally intended, and does not refer to decoding or dealing with the watermark in any way. Of course there are exceptions; such as applications that are only intended for “in-house” operation. In this case, it may be inferred that any viewer may be customized to deal with the effects of the watermark on the cover signal.

1.3 Common Applications
There are a myriad of useful ways watermarking can be applied to multimedia content. As mentioned earlier, this work will focus on images and video. This section is intended to give the reader a sense of a few broad areas where watermarking is currently being employed; for more in depth information on a specific topic the reader is referred to the increasing amount of literature available.

1.3.1 Identification/Authentication
One of the earliest applications of watermarking was as proof of ownership. A popular (within the field of watermarking) image known as “Lena” had a logo in the bottom right corner, but could easily be cropped out without altering the rest of the image which did not decrease the value of the image. A solution to this problem is to embed the watermark in a manner where it can not easily be identified and removed. Watermarks have since been used as verification within identification cards (often embedded in the person’s image), tracking in cinema movies, ownership of photographs, and representing unaltered surveillance images.
1.3.2 Security/Steganography
Steganography has been greatly focused on because of its military and espionage applications as well as having mathematical roots already in place. Steganography refers to hiding the watermark in a cover signal such that the watermark is not even observed as present to an attacker. Often times this requires that even the statistics of the cover signal not be significantly altered. Applications here are clearly of the “secret message” type—meaning that a cover signal such as broadcast television has a message secretly embedded in it so that information can be covertly transmitted. Similar use has been seen in transmitting JPEG images containing hidden messages over the internet.

1.3.3 Distribution Control
Watermarking has been used by the movie industry to successfully reduce pre-release versions of movies. Often times, production staff receives a copy of a movie prior to its general release. If it is made known that each copy has a unique watermark, it serves as a deterrent for the staff member to sell the copy since it could be traced back to them.

1.3.4 Metadata
Metadata refers to embedding data in the cover signal, which can be in the form of a watermark. This can be useful for combining data streams; for example in the case of space probes. Space probes are outfitted with a camera as well as numerous other instruments reading velocity, location, temperature, etc. The video from the camera and the other data must be sent separately, unless the data is embedded directly into the video. Any useful data can be embedded in video such as the location and date that the video was shot—thus avoiding the necessity to place this information in the form of font over the image as currently done for most hand held video cameras.

1.4 Existing Techniques
Current existing multimedia watermarking techniques are found in three domains. The first is in the uncompressed medium. For images and video this would be the RGB, or equivalent, color space as displayed. The second is in the frequency domain, such as taking the discrete cosine transform (DCT) or fast Fourier transform (FFT) of the spatial domain. The third is the compressed form of the medium. This would be how the medium is transmitted or stored before viewing. While there are many different watermarking algorithms, the following will give the reader a brief introduction to some of the basic methods in each domain. Most other techniques are similar or based on the ones described here.

1.4.1 Uncompressed Spatial Domain
The first watermarking techniques began in the spatial domain. Of course the most obvious way of watermarking is simply to replace pixel values in an image with new values that represent the watermark. This is still done to place logos for TV channels, for example. However, these can be removed through
simple image processing. So for any medium that is expected to be attacked, a more robust method was required.

While still in the spatial domain a more subtle method for watermarking is to modify the least significant bits (LSB) of pixels in the image. The image will not be distorted to a large degree and it will be more difficult for an attacker to locate the watermark. Unfortunately, statistical analysis will usually reveal that a watermark is present and an attacker would only need to randomize the lowest bit of every pixel to destroy the watermark. While not widely used currently in the spatial domain, LSB watermarking has reappeared in use in the fully compressed domain.

Another technique is to add a pseudo random noise pattern to an image. Then, to detect the pattern the correlation between a watermarked image the random pattern is calculated. If the correlation is higher than some threshold, a watermark is detected.

1.4.2 Partially Uncompressed Frequency Domain

Probably one of the most widely recognized watermarking algorithms is spread spectrum watermarking. This method watermarks in the frequency domain. Many compression algorithms, such as JPEG, convert an image to the frequency domain using DCT or some other transform so that high frequency information can be removed from the image. Thus only partial decompression of the image is necessary to get to the frequency coefficients. Removing high frequency information is less noticeable to human viewers than the low frequencies. Spread spectrum places the watermark by changing some of the frequency coefficients. In this case changing a frequency coefficient will spread the watermark over the entire image in the spatial domain. This makes cropping out the watermark in the spatial domain impossible and the watermark will also be resilient to rotation and scaling. One of the main choices for this algorithm is deciding which frequencies to watermark. If the high frequencies are watermarked then the visual impact to the image will be minimized, but someone could remove or randomly alter these frequencies, destroying the watermark, and still not affect the ability to view the image too much. Watermarking the low frequencies is more robust but will have a greater visual impact. Typically, some middle frequencies will be used to try and strike a balance between being robust and not distorting the image.

There have been many variations on watermarking in the frequency domain. One variation is to do frequency transforms on small blocks within the image as opposed to the entire image. This has the benefit of being able to possibly locate tampering within an image, but also could have parts of the watermark easily cropped out. In this case it may be necessary to place the same watermark in every block of the image. Major drawbacks of current algorithms applied JPEG images such JSTEG, OutGuess, and F5, are that they are lossy algorithms and since they require at least partial decompression of the signal, they are slower for embedding a watermark.
1.4.3 Fully Compressed VLC Domain

Out of the three domains, the compressed domain has probably seen the fewest algorithms applied to it. The reason for this is that watermarking requires some redundancy in the cover signal to exploit. The whole point of compression algorithms is to remove all of this redundancy, thus it is very difficult to watermark successfully in the fully compressed domain. One method is a form of LSB watermarking of the variable length codes (VLC)\(^8\). This method was designed to be applied to MPEG video streams. The algorithm finds pairs of VLCs that have the same level (referring to categories of amplitude for AC coefficients) and length. Then the LSB of the VLC that is part of a pair is forced to be identical to the watermark bit that is desired to be embedded. Another similar method that has been proposed for compressed domain is done by modulating DCT coefficient levels\(^9\). The largest drawback of both of these algorithms is that they are lossy.

Besides the algorithm proposed in the next section of this paper, there is only one other completely compressed domain watermarking scheme that boasts the lossless characteristic. This algorithm is designed specifically for JPEG and is a modified version of the LSB watermarking technique\(^{10}\). Certain AC DCT coefficients are paired and assigned parity’s based on whether the coefficient value is even or odd (essentially LSB of the appended bits to the VLC pairs). A list of the valid pairs and the corresponding parities is stored in a vector. This vector is then compressed and the extra room gained through compression allows for additional data to be stored in the vector to make it back to its original length. The extra data is what the LSBs were originally (allowing for the lossless quality of removing the watermark) and any additional room after that can have the watermark stored in it. Then the LSBs of the VLCs are replaced by the new vector. For JPEG, this may cause significant visual degradation, so based on certain criteria (run/size combinations) only certain VLCs are considered valid for watermarking. The drawbacks of this algorithm are that there is a low capacity and even only watermarking certain VLCs still results in perceptual visual artifacts in the watermarked image.

2.0 General Lossless Compressed Domain Watermarking

This section lays out the framework for a new watermarking algorithm that is designed to be applied directly in the compressed domain.

2.1 Introduction

The goals for this new watermarking algorithm are that it be applied in the fully compressed domain, be mathematically lossless, be file-size preserving, and the watermarked signal remain format compliant. Many of these features are seemingly contradictory, such as lossless watermarking in the compressed domain. Compression algorithms remove redundancy and to create a lossless algorithm, there must be more redundancy inherent in the cover signal than for a lossy algorithm. This problem could be side stepped if the file size of the cover signal could be increased. Then it would only be necessary to put in markers to indicate what or where the watermark was and then what the original signal contained.
However, the file size was intended to remain constant. Finally, the format of the compressed signal has to remain unchanged. In principal, this is probably the easiest criteria to meet. Achieving this is done by limiting the watermarking to the VLC portion of the cover signal. However, also in this criterion is implied that visual degradation of the signal not be too extreme. Since the method will be lossless, it will always be possible to restore the original signal for display, so less emphasis will be placed on visual quality of displaying the signal while the watermark is embedded.

It is important to note that there is not a specific compression method that this algorithm is designed for. This is because most modern compression algorithms for still images and video all employ VLCs at the lowest level. Thus, the overall compression method is not crucial for designing the theory for this watermarking algorithm. Of course, if the algorithm is being applied to a specific codec (coding and decoding method), then this should be accounted for by modifying the proposed algorithm accordingly. However, the necessity for modification should be minimal.

2.2 Theory
The method for the proposed watermarking algorithm is laid out in this section. A summary of entropy encoding is presented, followed by a paradigm for how to think of VLCs, and finally how watermarking would actually take place.

2.2.1 Entropy Encoding
As described throughout this paper, the fundamental necessity for watermarking is that there is redundancy in the cover signal that is being exploited. VLCs are designed to be as close to theoretical entropy encoding as possible, such as traditional Huffman coding. When this is done, the average codeword length is as short as possible given a set of symbols and their corresponding occurrence probabilities. This corresponds to a maximum compression if the number of symbols being encoded remains constant. This also means that if there is any bit changes to a codeword in the bit stream that it will likely become another valid codeword. Valid codeword refers to a codeword that has been assigned to a symbol.

2.2.2 Codespace
The set of all valid codewords for a specific cover signal makes up the VLC table. This VLC table can be thought of as defining a codespace. Any VLC that is found in the VLC table is therefore part of the codespace, and any that does not is outside of the given codespace, with one additional clause. Since VLCs are designed to be instantaneously decodable, a valid codeword cannot be a prefix to another valid codeword. The clause is then that for a codeword to be outside of the codespace it must not be a prefix to a valid codeword, or have a valid codeword be a prefix to it.
2.2.3 Codepairing and Watermarking

Watermarking of a compressed bit stream can be achieved by changing a codeword within the codespace to one that is outside of the codespace. If this mapping is one-to-one then the algorithm will be lossless since any codeword outside of the codespace can be mapped exactly back to the original. If the mapping is achieved by flipping bits, instead of adding or subtracting bits, then the file size of the cover signal will be preserved. Of course, if the VLCs are being modified, then the overall format of the cover signal will remain unchanged. The degree to which the watermark will impact the visual quality of the cover signal must be tested, but will likely be large if the number of bits that are changed is large.

Since, any useful compression algorithm will have VLCs that are close to entropy encoding, there may not be a large number of valid VLCs that can be successfully mapped outside of the codespace by flipping bits. Codeword pairing may help deal with this problem, by conceptually expanding the codespace by squaring the number of valid codewords (if pairs of codewords are now considered to be the valid ones). This is done by pairing every possible valid codeword with all other ones, including themselves. Now, instead of trying to map a single codeword outside of the codespace, a codeword pair is attempted to be mapped outside of the codespace defined by all of the pairs. This will give a higher percentage of pairs that can be watermarked when compared to the percentage of non-paired codewords that could have been watermarked.

If the original VLC table was in fact Huffman coded, then even pairing may not achieve a high enough capacity for watermarking to be useful. To guarantee that there will be enough room to watermark, VLCs that have error correcting ability built in could be used. Error correcting ability is intentionally leaving in some extra redundancy so that bit changes can be detected and possibly corrected. This would not only allow for watermarking, but the error correction may help in covering up any visual distortions caused by the watermarking. One type of VLCs that have error correction built in is reversible variable length codes (RVLCs). RVLCs are two way decodable; meaning that they are instantaneously decodable in both the forward and backward directions. With this added feature they are farther away from entropy encoding, and hence will have a larger watermarking capacity.

2.3 Codespace

For fixed length codewords, total code-space refers to all possible codewords of the given length. For a fixed length code of three bits there is a total of $2^3$, or eight possible codewords. However, for a specific application there may be less than eight valid codewords. The valid codewords make up the valid codespace. The occupancy for a given code is the percent of total code-space taken up by the valid code-space—the ratio of valid codewords to total possible codewords. Extending the definition of occupancy to variable length codes requires first choosing a length to determine total code-space. The obvious choice is to choose the length of the longest valid codeword. Occupancy in this context is then defined as ratio of the
number of valid and illegal codewords with length equal to the longest valid codeword to the total number of codewords (valid, unused, and illegal) of the same length.

For conceptual purposes, we first propose representing total code-space by a large circle, with valid VLCs represented by smaller circles within the larger one, as shown in Figure 1; in later sections we will present a more practical view of code-space in the form of code trees. The different sizes of the smaller circles represent different lengths of the codewords. So far, we have mentioned three types of codewords: valid, unused, and illegal. Valid codewords are smaller circles that initially exist, and by definition will never be overlapping since by definition they do not violate the prefix condition. Unused codewords are represented by space within the larger circle not occupied by a smaller circle. A violation of the prefix condition within the visual of Figure 1 would occur if a circle were to overlap another circle, so any circle that overlaps another circle would represent an illegal codeword. Watermarking changes a single bit of a valid codeword, essentially mapping an existing small circle from one region of the large circle to another. We would like to watermark a codeword by changing a valid codeword to an unused one and avoid changing it to an illegal codeword—essentially mapping a small circle to an open area of the larger circle. For valid code-spaces that are close to theoretical entropy encoding, the small circles nearly fill the larger circle as shown in Figure 1. Occupancy for such cases is nearly 100% since moving any of the smaller circles would result in overlapping another circle which means for this valid code-space there is no room for watermarking using our algorithm to this point.

The solution is to create more room by increasing the size of the large circle, or conceptually expanding the total code-space, seen in Figure 2. This is done by pairing all valid codewords with all other valid codewords including themselves, which we represent paired codewords as ovals in Figure 2. It is important to note that this is a conceptual pairing; a decoder with pairs of valid codewords can decode the same

![Figure 1- Example of an entropy coded code-space. Each circle represents a variable length code. Any vector will overlap another one, if moved. The overlap violates the prefix condition. This code-space is not suitable for watermarking.](image)
stream as a decoder with individual codewords (up to the last codeword if there is an odd number in the stream). By pairing, the total code-space is squared, represented by squaring the area of the large circle. Of course the number of smaller circles (and area they take up) is squared as well, but if there was even a little unused area of the large circle, the unused area will grow as well allowing for the possibility of watermarking. However, if the original occupancy was exactly 100%, then even pairing will not create any more unused area since the occupied region and total code-space were the same size to begin with and will grow at the same rate.

With the expanded total code-space, there are enough unused areas to begin watermarking. Since we are watermarking a codeword by changing exactly one bit, then the number of new codewords, or locations, the codeword can be mapped to is equal to its length. In Figure 2, Vij can be assumed to be at least three bits long since it is shown being mapped to three different regions. Following arrow 1, in Figure 2, Vij is mapped to V*ij. In this case, the result is an illegal mapping since V*ij overlaps Vpq corresponding to a violation of the prefix condition. Thus we can not watermark the bit of Vij that caused this mapping. Watermarking a different bit of Vij will map the codeword pair along arrow 2, which also causes an illegal condition. Finally, watermarking the bit of Vij that causes a mapping along arrow 3 results in a valid watermarking since V*ij does not overlap any other existing valid codeword pair.

![Diagram](image)

**Figure 2-** By pairing VLCs, it is possible to create redundancies that did not exist in the original code-space. Each ellipse represents a VLC pair. It is now possible move pairs to locations with no overlaps.

### 2.4 Codeword-pairing

Define an expanded code-space by pairing the original VLCs as follows:

\[
U = \{u_{ij}\}, i, j \in 1, ..., N
\]

\[
u_{ij} = \{v_i, v_j\}
\]  \hspace{1cm} \text{(1)}
U now consists of $N^2$ codeword-pairs ranging in length from $2l_1$ to $2l_N$. To illustrate what code-pairing brings to the table, consider simple block code \{00,11\}. Out of possible 4 codes, two are taken resulting in 50% occupancy of the code-space. However, if looked at in pairs the code space is \{0000 0011 1100 1111\}. Out of possible 16 codewords, only 4 are taken resulting in 25% occupancy. If the original code-space is 100% occupied, then pairing will not help. Examples of such cases are Huffman coded stream used in Baseline JPEG. However, space occupancy of streams using reversible VLCs (RVLC) is much less. RVLCs are used in error-resilient streams, such as MPEG-4 and H.263, and provide two-way decoding capability.

2.5 Binary Code Tree

Code-space occupancy can be defined and quantified using codetrees. The codetree for variable length codes are binary trees where VLCs occupy leaf nodes. The root of the tree for $\mathbf{V}$ is at level 0 and consists of $l_N$ levels. Level $l$ may consist of up to $2^l$ nodes. The maximum number of leaf nodes in the codetree for $\mathbf{V}$ is $2^{l_N}$. All of the possible leaf and branch nodes from $v_i$ at levels from $l_l$ to $l_N$ define a subtree. To satisfy the prefix condition, none of the codes in $\mathbf{V}$ can reside on this subtree. As a result $v_i$ removes a total of $2^{l_N-l_l}$ nodes in the tree. As an example, let $\mathbf{V} = \{00,010,0110\}$. The tree corresponding to $\mathbf{U}$ is shown in Figure 3. The solid nodes represent elements of $\mathbf{U}$. The nodes shown in dashed circles are unavailable as valid codewords as they are preempted by the parent node. The entire right side of the tree is unoccupied. The maximum number of codes, or leaf nodes, that do not violate any prefix conditions in a 4 level tree is 16—all VLCs occurring at the 4th level. In Figure 3 7 out of 16 nodes are either occupied or are denied as codewords due to violation of prefix rule. In total, 43.75% of the maximum 16 leaf nodes are considered occupied. We can derive a simple expression for tree occupancy of a general variable length codebook. Since watermarking a codeword is equivalent to mapping of the codeword to a free node in the tree, the number of free nodes controls watermarking capacity. For code-pair $\mathbf{U}$ consisting of $2N$ codewords of length $\{l_1,l_2,...l_{2N}\}$, percent occupancy of the corresponding codetree is given by,

$$\text{% occupancy} = \frac{\sum_{i=1}^{2N} 2^{2l_N-l_i}}{2^{2l_N}}$$

Evaluating this expression for the codetree in Figure 3 the percent occupancy is 19.14% This is a substantial reduction from 43.75% when pairing is not used. Occupancy is 100% for perfectly entropy coded streams, with or without pairing.

Watermarking of a codeword can now be defined in the context of the codetree. Let $u_{ij}^k$ define the watermarked version of $u_{ij}$ by flipping the $k$th bit. This action will move $u_{ij}$ to another node on the same level. The node the code moves to depends on $k$. The watermarked codeword must meet the following conditions where $\notin$ means “not a prefix of”:
1. $u^k_{ij} \notin u_{mn}$

2. $u_{mn} \notin u^k_{ij}$

3. $u^k_{ij} \notin u^p_{mn}$

4. $u^p_{mn} \notin u^k_{ij}$

Conditions 1 and 2 assure that no watermarked codeword violates the prefix condition. Conditions 3 and 4 assure that no collision takes place when two different valid codewords are mapped to the same tree node as a result of watermarking. Mapping a codeword to another node alters the structure of the tree. In most cases, more than one option is available. The choice depends on many factors. Moving one codeword to another node denies the entire subtree of the new node from being used for watermarking thus affecting capacity. How far the code is moved from its original location also impacts quality.

The total possible number of nodes in the codetree shown in Figure 3 is 16. Each leaf node removes the entire subtree under it thus reducing percent occupancy of the tree. In this case, only 7 out of 16 nodes are occupied. The entire right side of the tree is free. Watermarking is defined by mapping one codeword to another node at the same level. The new node must neither lie on a path to an existing codeword nor be the child of an existing codeword. For decoding, the mapping must also be unique, i.e. no two codewords shall be watermarked to the same node or violate the prefix condition for each other.

Figure 3 - The codetree for variable length codes $V = \{00, 010, 0110\}$. 

16
2.6 Example

This section will present a complete methodology of our algorithm on the same three initial codewords from the previous section. So for this example, \( N \) is three and \( V = \{00, 010, 0110\} \). These codewords happen to be symmetric RVLCs; however, our algorithm can be applied to any type of VLC. Analysis of the codewords will be presented based on the binary tree structure, followed by proposed watermarking, and then how encoding and decoding would work on a cover medium containing the codewords.

2.6.1 Form Codeword-pairs and Binary Tree

The first step in our algorithm is to exhaustively pair each codeword with every other codeword, including itself. The pairs represent any possible combination of two successive codewords that could be found in the cover signal. Pairing the three codewords in our example results in nine codeword-pairs as follows: \( U = \{0000, 00010, 000110, 01000, 010010, 0100110, 011000, 0110010, 01100110\} \).

The binary tree structure formed from \( U \) is shown in Figure 4. Note that since no codeword-pairs begin with a ‘1’, that the entire right half of the tree is empty. In this example only branches necessary to place a leaf node (containing a codeword-pair) are drawn.

![Figure 4-Binary tree for U](image)

2.6.2 Determine Watermark Mapping

We would like to watermark every codeword-pair. This is done through analysis of the paired VLC table. Both the pairing and analysis can occur offline if the original unpaired VLCs are known prior to receiving the cover signal that is desired to be watermarked. Our algorithm calls for watermarking exactly one bit of each pair, if possible. With our example this will be possible because the original code only has an
occupancy of 43.75%. While having a low occupancy does not guarantee that every pair can be watermarked, it increases the probability that this is the case. To determine which bit of each codeword-pair will be changed to watermark that pair, we begin by flipping the least significant bit. If we were checking the codeword-pair, ‘0000’, then we would modify it to ‘0001’. Unfortunately, it turns out that this would cause a collision with codeword-pairs {00010 000110} since this violates the prefix condition. This can also be seen clearly from Figure 5 as ‘0001’ falls on an already existing branch node along the path to valid codeword-pairs.

Using the tree structure allows for quick determination if a watermarked codeword-pair is placed on an unused or illegal location. For a location to be unused, at least one branch must be created in creating the path in the binary tree. This is true if we were to watermark the second least significant bit of ‘0000’ resulting in ‘0010’. Two new branches must be created to place the watermarked version in the binary tree, which is therefore an unused location and does not cause any collisions. This is shown in Figure 6.
When a valid codeword-pair is found to be watermarkable, the watermarked version is immediately added to the tree. For our algorithm to be successful at being lossless, it is necessary to be able to map a watermarked pair back to its unwatermarked form. For this to be true, no two watermarked pairs can violate the prefix condition, otherwise during decoding it may be impossible to tell which codeword-pair should be decoded. By immediately adding watermarked pairs to the tree, any subsequent watermarked pairs will be checked against existing ones for collisions. When watermarked pairs are added to the tree, the original and watermarked versions are tied together so that when either encoding or decoding, adding a watermark or removing it is done by following the mapping. When all watermarked pairs are added to the original tree, the resulting tree will be referred to as the watermark tree. A portion of the watermark tree for our example is shown in Figure 7.

![Figure 7-Part of Watermark tree](image)

The entire list of codeword-pairs and their watermarked versions are in Table 1. This table will be used as a reference since the watermark tree with all eighteen terminating nodes is difficult to display clearly. In this example, all codeword-pairs have the ability to be watermarked.

**Table 1-Complete list of valid codeword-pairs and watermarked versions.**

<table>
<thead>
<tr>
<th>Codeword-pair</th>
<th>Watermarked Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0010</td>
</tr>
<tr>
<td>00010</td>
<td>00110</td>
</tr>
<tr>
<td>000110</td>
<td>000111</td>
</tr>
<tr>
<td>01000</td>
<td>01010</td>
</tr>
<tr>
<td>010010</td>
<td>010110</td>
</tr>
<tr>
<td>0100110</td>
<td>0100111</td>
</tr>
<tr>
<td>011000</td>
<td>011010</td>
</tr>
<tr>
<td>0110010</td>
<td>0110110</td>
</tr>
<tr>
<td>01100110</td>
<td>01100111</td>
</tr>
</tbody>
</table>
2.6.3 Encoding

Once all of the codeword-pairs have been evaluated on whether or not they can be watermarked it is time to encode our watermark in a cover signal. The cover signal as well as the watermark that we will be using is shown in Figure 8.

```
Watermark bits: 101011
Cover signal bitstream:
000001100001000010011001001100001000
```

Figure 8-Example compressed cover signal bitstream and watermark bits.

The cover signal VLC bitstream is parsed one bit at a time until a valid codeword-pair is recognized. For each bit read we move one level down the binary tree. Since we are assuming there are no transmission errors in the cover signal, when we reach a terminating node, the codeword-pair represented by that terminating node is recognized. For encoding, there are two possibilities for a terminating node, either it is a codeword-pair that can be watermarked or one that can not be.

Embedding a watermark bit of one requires finding the next codeword-pair in the bitstream that can be watermarked (in general not all pairs will be able to be watermarked) and mapping that codeword-pair to its watermarked version as described by the watermark tree. Embedding a zero requires no change of the bitstream. This is done by finding the next codeword-pair that still could be watermarked, but then we simply skip it without changing it.

As we parse our example bitstream, we first find a ‘0’ bit. From this we take the first left branch and find another branch node so we continue. We parse three additional ‘0’ bits, each causing us to take a left branch until we finally reach a leaf node. The path is shown in Figure 9.
Reaching a leaf node signals the encoder that a valid codeword-pair has been found, in this case ‘0000’. Next the encoder must determine if this pair can be watermarked. This is specified by the watermark tree created during our offline analysis. In this case ‘0000’ can be watermarked to ‘0010,’ so a watermarked version of the cover signal is created accordingly with ‘0010’. Next we continue parsing, beginning at the first bit that follows our previously decoded pair. Our parsing discovers that ‘011000’ is the next pair. From Table 1 we see that ‘011000’ can be watermarked to ‘011010,’ however since the second watermark bit we wish to embed is a ‘0,’ we do not modify the codeword-pair, but simply add it to our existing watermarked bitstream. This results in a watermarked bitstream which currently is ‘0010011000.’ If we continue in this manner we decode the following pairs: ‘01000’, ‘0100110’, ‘010011’, and ‘00010’. Watermarking as appropriate results in the watermarked cover signal shown in Figure 10.

Watermark bits: 101011
Watermarked Cover signal bitstream: 0010011000 010100100110 010011 10011000

Figure 10-Watermarked cover signal.

At the end of the bitstream we parse ‘00’ and then the stream terminates. Since we did not reach a terminating node (this is only one VLC, not a pair) we can not watermark and the encoding process is finished.

2.6.4 Decoding
Decoding is carried out similarly to the encoding process. The decoder receives the watermarked version of the cover signal as well as the list of original, unpaired VLCs. The original VLCs are necessary so that the decoder can create the watermark tree using the same process as the encoder. With the watermark tree
and the watermarked bitstream, the decoder simply parses the stream in the same manner as the encoder. The difference is that when a terminating node is discovered during parsing there are now three possibilities instead of two. The first case is that the codeword-pair is not one that could be watermarked. The second case is that it is a pair that has been watermarked. Finally, the third case is that the codeword-pair could have been watermarked, but was not.

One goal of the decoder is to extract the watermark bits. If case one is encountered, the decoder does not record any watermark bit as being embedded. For case two the decoder records a ‘1’ bit as being encoded and for case three a ‘0’ bit as being encoded in the current pair.

The decoder begins by parsing the bitstream by traversing the watermark tree, shown in Figure 11. The first terminating node reached is after ‘0010’.

From the watermark tree and Table 1, we find that ‘0010’ is the watermarked version of ‘0000’. Since this is case two we record a ‘1’ watermark bit and if desired we can remove the watermark partially restoring the original cover signal by converting ‘0010’ back to ‘0000’. Continuing with our parsing, we find ‘011000’ as the next decoded codeword-pair. This pair could have been watermarked to ‘011010,’ but was not. This indicates a case three and therefore we record a ‘0’ watermark bit. Since every codeword-pair in this example can be watermarked, we will never encounter a case one. If we continue with our decoding we will recover all of the watermark bits, ‘101011,’ and have the ability to exactly restore the original cover signal.

2.7 Results

The proposed algorithm was tested on asymmetric RVLCs for the English alphabet shown in Table 3. After running the code, an estimated 4% of the cover signal could be watermarked. The reason that this is an estimate is that this is the expected percent based on how often each letter occurs in the English
alphabet. If run on specific text the actual capacity could be higher or lower. The results are shown in Table 2.

Table 2 Results from applying algorithm to English alphabet

<table>
<thead>
<tr>
<th>RVLC Codeword-pairs</th>
<th>Good Table</th>
<th>Great Table</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>676</td>
<td>613</td>
<td>434</td>
<td>4%</td>
</tr>
</tbody>
</table>

Table 3 RVLCs for the English alphabet.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>Letters</th>
<th>Asymmetric RVLC</th>
<th>Frequency</th>
<th>Letters</th>
<th>Asymmetric RVLC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E</td>
<td>001</td>
<td>14</td>
<td>F</td>
<td>11111</td>
</tr>
<tr>
<td>2</td>
<td>T</td>
<td>110</td>
<td>15</td>
<td>M</td>
<td>111101</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>0000</td>
<td>16</td>
<td>C</td>
<td>101101</td>
</tr>
<tr>
<td>4</td>
<td>O</td>
<td>0100</td>
<td>17</td>
<td>W</td>
<td>011101</td>
</tr>
<tr>
<td>5</td>
<td>R</td>
<td>0101</td>
<td>18</td>
<td>G</td>
<td>111011</td>
</tr>
<tr>
<td>6</td>
<td>N</td>
<td>1000</td>
<td>19</td>
<td>Y</td>
<td>01110011</td>
</tr>
<tr>
<td>7</td>
<td>H</td>
<td>1010</td>
<td>20</td>
<td>B</td>
<td>11101011</td>
</tr>
<tr>
<td>8</td>
<td>I</td>
<td>10010</td>
<td>21</td>
<td>V</td>
<td>111010011</td>
</tr>
<tr>
<td>9</td>
<td>S</td>
<td>01100</td>
<td>22</td>
<td>K</td>
<td>011110011</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>00010</td>
<td>23</td>
<td>X</td>
<td>0111100111</td>
</tr>
<tr>
<td>11</td>
<td>L</td>
<td>00011</td>
<td>24</td>
<td>J</td>
<td>1110101011</td>
</tr>
<tr>
<td>12</td>
<td>U</td>
<td>10111</td>
<td>25</td>
<td>Q</td>
<td>11101010011</td>
</tr>
<tr>
<td>13</td>
<td>P</td>
<td>11100</td>
<td>26</td>
<td>Z</td>
<td>1110101000111</td>
</tr>
</tbody>
</table>

3.0 JPEG Application

Up until this point a general watermarking algorithm has been presented and a binary tree structure has been shown to be an efficient tool in applying the algorithm. However, the algorithm has only been shown to effective on example VLC tables. This section will demonstrate how the general algorithm can be adapted to a specific compression method, JPEG. In addition to the previously stated criteria of lossless, file-size preserving, compressed domain, and format compliant; the algorithm will be shown to be able to visually mask the watermark while viewing watermarked JPEG images.

Watermarking JPEG images presents a number of challenges. The first challenge is that the watermarking is going to occur directly in the compressed domain so the JPEG file format must be fully understood. This includes knowledge of all headers and how to decode and encode a JPEG image. Another challenge is to find redundancy in the standard that no one else has taken advantage of. As with most compression standards, JPEG attempts to remove as much redundancy from an image as possible to minimize the file size. A final challenge is to be able to display the image without any noticeable visual degradation while the watermark is still embedded—by any standard viewer (without knowledge that a watermark has even been embedded).
3.1 Introduction to JPEG

The following sections will provide a detailed introduction into the JPEG standard itself, in preparation for further discussion about watermarking JPEG images.

3.1.1 Overview

The JPEG compression standard is one of the older standards yet is still one of the most widely used. For example, many images found on the World Wide Web are stored in the JPEG format. JPEG compression can be broken up into the following four key steps: first the image is divided into 8x8 pixel blocks, then each block is transformed to the frequency domain using the discrete cosine transform (DCT), followed by quantization, and finally entropy encoding.

3.1.2 8x8 Pixel Blocks

The reason that the image is divided into 8x8 pixel blocks is to try to maximize compression in the frequency domain. Most images have regions of similar texture. If groups of similar texture can be stored separately then only the most dominant frequencies in the region need to be stored, thus compressing the image. If an entire image were transformed into the frequency domain, there would likely be many different texture regions and therefore many different frequencies present. It would also be possible to divide the image into 4x4 or 2x2 pixel blocks. In these cases the texture of each block is more likely to be uniform; however texture regions are typically larger in size. Through experimentation, it was determined by the JPEG standards committee that 8x8 pixel blocks seemed optimal for most images, especially natural images. For color images, the luminous, or grayscale component, and the other two color components are treated separately. Each component is divided into 8x8 blocks for the remaining processes. Then each block is level shifted by subtracting 128 from each pixel value, so that pixel values which normally range from 0 to 255 now are on a scale of -128 to 127.

3.1.3 DCT

To take advantage of frequency redundancies in the spatial domain, each block of the image is transformed using the DCT. Other transforms could have been used, such as wavelet transform that is used in the more recent JPEG2000 standard. However, DCT has its advantages such as being well known and having a fast DCT version which dramatically reduces computation time.

When the DCT is performed on an 8x8 block, the result is a new 8x8 matrix with each of the 64 values representing a coefficient for a different frequency. The lowest frequency, which is known as the DC coefficient, is stored in the upper left corner of the matrix. The DC value is equivalent to the average pixel value. Horizontal and vertical frequencies increase to the right and down, respectively, in the matrix. All of the other 63 coefficients other than the DC coefficient are referred to as the AC coefficients.
For natural images, many 8x8 blocks will have uniform texture regions so most of the block information will be stored in only a few of the 64 coefficients, most likely in the lower frequencies. Many of the smaller valued coefficients will most likely be rounded to zero during the quantization step.

### 3.1.4 Quantization

Quantization is a rounding process that will hopefully reduce many AC DCT coefficients to zero, which is how much of the compression is achieved, in conjunction with entropy encoding. Every JPEG image must define a quantization table that consists of an 8x8 matrix of quantization values. Each coefficient in every 8x8 DCT block is divided by the corresponding value in the same matrix location of the quantization table, rounding down. The JPEG standards committee developed example quantization tables for grayscale as well as color components through experimentation on what is perceptible to the human vision system (HVS). Typically the lower frequencies have smaller values in the quantization tables and the values increase for higher frequencies.

JPEG allows for varying compression for a given image by sacrificing visual quality. By scaling the quantization table, more or less DCT coefficients will be rounded to zero to give a higher or lower compression rate. A quality factor of 50, usually specified in the encoding software, typically employs the example quantization tables provided in the JPEG standard. Increasing the quality factor by a given percentage will reduce the quantization table values by the same percentage.

### 3.1.5 Entropy Encoding

After quantization, many of the AC coefficients are zero, however these occur more to the lower right of the 8x8 block. To maximize the number of zero coefficients in a row the block is zig-zag scanned to convert the matrix into a one-dimensional vector. The zig-zag scan order of coefficients is shown in Figure 11\(^1\)\(^2\).
DCT coefficients are then considered to be in two categories, either zero or non-zero. Zero coefficients are run-length encoded to increase compression. Run-length encoding is achieved by associating the number of preceding zeros with each non-zero coefficient. Non-zero coefficients are placed in categories (SSSS) representing ranges of values, as shown in Figure 11. Thus each non-zero coefficient has two values associated with it: the run (or number of preceding zero coefficients) and the category, or size.

<table>
<thead>
<tr>
<th>SSSS</th>
<th>AC coefficients</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1,1</td>
</tr>
<tr>
<td>2</td>
<td>-3,-2,2,3</td>
</tr>
<tr>
<td>3</td>
<td>-7,-4,4,7</td>
</tr>
<tr>
<td>4</td>
<td>-15,-8,8,15</td>
</tr>
<tr>
<td>5</td>
<td>-31,-16,16,31</td>
</tr>
<tr>
<td>6</td>
<td>-63,-32,32,63</td>
</tr>
<tr>
<td>7</td>
<td>-127,-64,64,127</td>
</tr>
<tr>
<td>8</td>
<td>-255,-128,128,255</td>
</tr>
<tr>
<td>9</td>
<td>-511,-256,256,511</td>
</tr>
<tr>
<td>10</td>
<td>-1 023,-512,512,1 023</td>
</tr>
</tbody>
</table>

Figure 11 Size category for AC DCT coefficients
The entropy encoding method can either be Huffman or arithmetic encoding. Most images use Huffman encoding so that will be the focus of the discussion here. Runs of zeros can range from 0 (two non-zero coefficients next to each other) to 15. If more than 15 zeros occur in a row then a run of 15 and a size of 0 will be encoded followed by a new run/size combination with the remaining zeros. The DC coefficients are coded separately from the AC coefficients. The first coefficient in every block is the DC coefficient and each is difference encoded from the previous block (the difference between the previous block’s DC coefficient and the current block’s determines the value to be encoded). There is a DC size category table similar to the AC one shown in Figure 11, except there is a zero category for when two consecutive blocks have the same DC values (difference value is then zero). Also, for DC coefficients there are no zero runs, so only the size is Huffman encoded.

For AC coefficients, there are 10 different size categories (from 1 to 10) and runs of zeros from 0 to 15. This makes 160 different run/size combinations. Additionally there is an end of block (EOB) code (also known as run 0 size 0) and a run of 15 with size of 0 (ZRL) to allow for zero runs of more than 15 consecutive zeros. Therefore, there are 162 total different possible run/size combinations. The JPEG standard provides example tables for both luminance and chrominance Huffman tables that assign variable length codes (VLCs) to each run/size. However, the JPEG standard allows for custom tables to be created based on the run/size combinations that actually occur in a given specific image. A custom table would maximize the compression of the image without any additional visual degradation. A custom Huffman table would only assign VLCs to run/size combinations that actually occur in the image.

The final step in the entropy encoding is adding additional appended bits to each VLC. The VLC will specify the run and the size category, but not the actual value within that category. To specify the specific value within a category, a number of bits equal to the category are appended to the VLC. For example, a category 2 VLC will have two bits appended to it. Positive values are coded in the appended simply as the binary equivalent of the value. Negative values are the two’s compliment value minus one and then all but the most significant bit. For example, with category 2 value of -3, the two’s compliment of -3 is 101 (positive 3 is 011, subtract one 010, and invert all the bits 101), then subtract one from 101 to arrive at 100. Since only two bits are appended the low two bits of 100 are appended, which is just 00. Note that all negative valued appended bits begin with a 0.

### 3.2 Compression Inefficiency

Since watermarking occurs in the AC VLCs, that will be the focus of this discussion and all statements can be assumed to be about AC Huffman tables, however, most statements are generally true about DC VLCs and Huffman tables as well. The JPEG standard does not require a specific Huffman table to be used for every image. Instead, to maximize compression it recommends customizing the Huffman table based on the specific run/size occurrences for each individual image that is desired to be compressed. However, many applications bypass this customization step and instead use example tables that are provided in the standard itself.
These standard tables include VLC assignments for all 162 possible run/size combinations, which mean that the tables can be used for any image. The VLC assignment to each of the 162 possible run/sizes was determined based on a large library of images, approximately 100,000 according to the standard. This means that if the example table is examined and a run/size such as 0/2 has a shorter VLC than 1/2, then for most images it should be expected that 0/2 combinations occur more often than 1/2 combinations. It should be clear that there is the possibility that for a specific image that 1/2 might happen to occur more often than 0/2 which means that the image is not compressed as well as could have been.

It is typical to find natural images that use between 40 and 60 of the possible 162 run/size combinations. If such an image is using the example table than this guarantees that there is plenty of watermarking capacity. Using less than half of the possible run/size combinations would at first appear to be a gross neglect of efficiency. In reality most of the combinations used are the ones with the shortest VLCs anyway, and only a few of the used VLCs are longer than they need to be, and these are ones that occur infrequently in the image. So forming a custom table for image may only improve compression a few percent, which is why many applications bypass this step.

Even though creating custom tables would only improve compression slightly which may not be important to the average user (unless thousands of images were being stored), a few percent capacity in watermarking may be highly desirable to store copyright, date, location, names, or other information about the image.

### 3.3 Codespace, VLC Mapping, and Synchronization

A valid codespace consists of all of the codes for a given Huffman table, which are referred to as valid VLCs or codewords. By default a bit string of any length that is not a prefix to a valid VLC, or vice versa, is considered to be an invalid codeword. Typical Huffman codes are formed so that changing a single bit of any existing VLC will result in another valid VLC (in the standard JPEG table the only invalid codewords are 16 bits or longer beginning with 16 one bits). This means that it is impossible to find an invalid codeword. In a JPEG stream this can be thought of as a loss of synchronization because an error would not be detected. The decoder would simply misinterpret one VLC for another (which is likely a different length) and continue decoding, thus propagating the error. The only positive aspect of such an error is that it only takes one location that propagating errors just happen to randomly realign with the correct beginning of a VLC for the decoder to regain synchronization. Although the length of the “burst” error (meaning a single error that is propagated, simulating a string of errors) can not be predicted, and realignment is not guaranteed, the number of VLCs in most JPEG images is so high that it is likely that resynchronization will eventually occur. However, even a single bit error in this case can cause dramatic visual degradation of the image. Therefore avoiding loss of synchronization in a JPEG watermarking scheme is crucial.
3.4 Exploiting Inefficiency

In many JPEG images, the example AC Huffman table is used, which includes all 162 possible run/size combinations even though less than half actually occur in the image. This means that the valid codewords can now be broken up into two subsets: used and unused (referring to whether or not they actually occur in the image, respectively). The used codewords are then examined to determine which of these can be watermarked. For a codeword to be successfully watermarked, exactly one bit must be changed and the result must not be a prefix to another used codeword, or vice versa. For the example Huffman table it is not possible to find invalid codewords which means that any successfully watermarked used VLC will be an unused VLC (or prefix to one or vice versa). To maintain lossless quality of watermarking then once a VLC is watermarked mapping it to an unused VLC, the unused VLC must be considered as “used” for determining if any subsequent VLCs can be watermarked. Essentially this keeps two used VLCs from being mapped to the same unused VLC (it would be impossible to know which was the original in this case). The only other requirement is that the original used VLCs are passed on to the decoder in an additional Huffman table.

3.5 JPEG Watermarking

These sections will layout the specifics for taking a JPEG image and inserting a watermark. This process can be broken into three key steps. The first step is parsing the JPEG image and determining which VLCs actually occur in the image. The next step is to take the used VLCs and determine which VLCs can be successfully watermarked. The final step is actually embedding the watermark in the watermarkable VLCs. There are currently two methods for embedding the watermark that emphasize different characteristics. This section will assume that a grayscale JPEG image is being processed. The only difference for color images is that they have two additional chrominance components which are sampled at a lower rate than the luminance (grayscale) component and these additional samples are interlaced in the scan. Also, the JPEG image is assumed to be the baseline version (which is the most common version).

3.5.1 Parsing a JPEG image

The JPEG file structure is formatted so that each section has a header to identify the start of that section. All JPEG headers are made up of two bytes: the first is always 0xFF (hex for eight ones) followed by a second byte that designates what type of section will follow the header. Headers always occur after an integer number of bytes. Usually following each header, regardless of which section follows, is additional information about the section, such as length (either of the entire section or just the additional header information prior to the start of data for the section) and variable definitions.

JPEG headers allow for easy and fast transversal of the file to locate specific sections. For watermarking purposes, many sections are ignored as they are not necessary for the watermarking process. For example, the quantization tables are ignored.
3.5.1.1 Huffman Tables
The first sections that need to be considered when parsing a JPEG image are the Huffman table sections. These can be located by scanning the JPEG file from the beginning looking for the Huffman header 0xFFC4. The JPEG standard allows for different ways of inserting multiple Huffman tables. One way is that each additional Huffman table has its own header, and the other way is that multiple Huffman tables follow a single header.

For grayscale images there are two Huffman tables, one for the AC and one for the DC DCT coefficients. The JPEG standard allows up to four Huffman tables for any given image. The additional tables are typically used for AC and DC color components.

The Huffman tables dynamically create all VLCs and assign each to a unique run/size combination. The watermarking algorithm proposed here takes advantage of the fact that while, the VLCs are dynamically created by each image, most images are coded so that the VLCs that are created are those found in the example table in the JPEG standard.

VLCs are created using a list of the number of VLCs of each length, ranging from one bit long to a maximum of sixteen bits long. This list is referred to as \( L_i \)'s. There are always sixteen \( L_i \)'s each of which is exactly one byte, with the first indicating how many length one VLCs there will be. Using these lengths and a flow chart (C.1) found in the JPEG standard, all the VLCs for a Huffman table are created. The VLCs are ordered from shortest to longest when first created. If the example table from the JPEG standard is being created there will be 162 VLCs.

Next, the VLCs are each paired with a unique run/size combination. Run/size combinations are listed after the \( L_i \)'s and are referred to as \( V_{ij} \)'s. Each of the \( V_{ij} \)'s is one byte long with the first four bits explicitly stating the run, and the second four bits explicitly stating the size. Runs range from 0x0 to 0xF and sizes range from 0x1 to 0xA. Additionally there is a run/size of 0x0/0x0 which is the end of block (EOB) marker and a run/size of 0xF/0x0 which is the ZRL marker indicating fifteen zeros. ZRL is only used when there is a run of more than fifteen zeros in a row.

After all VLCs are paired with a run/size, this list is the Huffman table. Defined in the header for each Huffman table is whether the table is for AC or DC coefficients. For grayscale images, the whole process is repeated at least twice so that are two complete Huffman tables, one for AC and one for DC.

3.5.1.2 Zero Padding
As previously described, every header begins after an integer number of bytes and then is designated by 0xFF. If at any point in the file data happens to result in an 0xFF after an integer number of bytes, then a byte of eight zeros is placed after the 0xFF data. This indicates to a JPEG decoder that the 0xFF is data and not the beginning of a header.
Zero padding can cause lots of problems when attempting to watermark an image. For example, the algorithm proposed here is designed to preserve the original file size of the JPEG image by not adding bits; watermarking is achieved simply by flipping bits. However, by flipping a bit it is possible to change what was not 0xFF in data to 0xFF. This would mean that a decoder would interpret it as a header instead of data. To counteract this it would be necessary to insert a 0x00 byte. This can create a small file size increase depending on how many times this occurs. This also has the potential of decreasing the file size since if 0xFF in data are ever watermarked it would mean removing the following 0x00 byte (else the 0x00 would be interpreted as data). Some of these concepts and possible solutions will be discussed later when examining modifications to inserting the watermark.

To avoid unnecessary logic in implementation, all zero pads are removed from the VLC portion of the JPEG file during parsing. This is achieved by finding the start of scan (SOS) header and the end of image (EOI) header and ensuring that there are no additional headers in between. In this case, all 0xFF are part of data and the following 0x00 are simply removed. After watermarking, the VLC portion of the JPEG file is rescanned and any necessary 0x00 (including possible new ones) are reinserted.

### 3.5.1.3 Reading in VLCs

After the zero pads are removed, and both an AC and DC VLC table have been decoded then the last step is to read the VLCs. If there are multiple DC and/or AC Huffman tables in the image, the SOS header includes information to determine which Huffman tables will be used during this scan. Baseline grayscale images should have only two Huffman tables and a single scan.

The first VLC is always a DC VLC so the DC VLC table must be used to identify it. Once identified, the category (or size) determines how many additional appended bits follow the VLC. These appended bits are skipped and the decoder now knows where the beginning of the next VLC is located. Following the DC VLC are up to 63 AC VLCs (if all are non-zero). After the DC VLC is decoded the AC VLC table is used to decode the next AC VLC and then determine how many appended bits follow it; again based on the size. This process is continued using the AC VLC table for decoding until and EOB VLC is found (which is part of the AC VLC table) or 63 AC coefficients (not VLCs) have been decoded. Coefficients include runs of zeros, so it is possible to reach 63 AC coefficients with fewer than 63 AC VLCs present.

An EOB VLC indicates that all of the remaining coefficients in the current block are zero. After an EOB or 63 AC coefficients have been decoded the next VLC must be a DC VLC, so the DC VLC table must be used to decode it. This process is repeated until the end of the image is found. To keep the EOI header after an integer number of bytes, ‘1’ bits are placed after the final VLC so that this occurs (if the final VLC lines up perfectly no ‘1’ bits are necessary).

While reading in all of the VLCs, the number of times each VLC occurs in the image is stored for later use during the watermarking process.
3.5.2 Determine Watermarkable Used VLCs

One of the results of parsing the image is a list of VLCs that actually occur in the image. This list is used to create a binary tree structure. To determine which VLCs are watermarkable, each node is checked individually, beginning with the node that is furthest left. The least significant bit of the VLC at this node is flipped and the resulting VLC is checked to see if it falls at valid node location. A valid node location is one that does not already have an existing node above or below it along the same path through the binary tree. If this were the case, then the new VLC would be a prefix to another VLC or vice versa, making it impossible for a decoder to interpret it correctly. If the least significant bit change does not result in a valid node, then the next most significant bit is checked. As soon as a single bit is found to be successful then the new valid node is added to the tree, the bit that was changed for the current VLC node is saved (this will be the bit that is watermarked), and the next node is checked. If there are no valid nodes found by checking all of the bit positions for a VLC node, then that VLC cannot be watermarked. Additionally, since every time a VLC is found to be watermarkable a valid node is placed in the tree, it is more difficult for each subsequent VLC to find a valid node (there is a finite number of available nodes). Following this manner, all of the original VLC nodes are checked.

The process described so far refers to individual VLCs. The same process is used when considering pairs of VLCs, except each of the original nodes in the binary tree is a VLC pair. The pairs are created by pairing every VLC that existed in the image with every other VLC that occurred in the image, including itself.

It is important to note that the order in which VLCs are checked can change the final outcome. Therefore, depending on the order, it may be possible to increase the final watermark capacity. Additionally, the current method requires changing exactly one bit to achieve watermarking. There is no reason that more than one bit could be changed.

3.5.3 Inserting and Removing Watermark

After the binary tree determines which VLCs can be watermarked, and which bit to change of each, the final step is to actually insert the watermark. From parsing, there is a list of all VLCs that occurred in the image, and how often each occurred. The number of times watermarkable VLCs occurred in the image determines the maximum watermark capacity. This section describes will describe two methods for inserting the watermark. The key difference is that how a standard JPEG decoder would interpret the watermark is not considered in the basic method, while it is considered in the appended bits method. This means that it may not be possible to view the JPEG image in a standard decoder using the basic method while the watermark is still embedded (the process is lossless so the watermark can always be removed and the original image then displayed). The appended bit method will allow for viewing the image while the watermark is embedded with some visual distortion, a dramatic increase in capacity, but also an increase in file size.
3.5.3.1 Basic Method
After the maximum capacity of the image is known, a bit string is chosen for the watermark that is less than or equal to the maximum capacity. The only difference between pairs of VLCs or individuals being watermarked is how many VLCs are read in at a time. This section will describe the watermarking process for individuals, but the same would be true for pairs.

3.5.3.1.1 Encoding
The first AC VLC is read in from the image, and is checked to see if it is watermarkable. If it is not watermarkable, the VLC is skipped and the next AC VLC is read. Only AC VLCs are considered for watermarking. If the VLC is watermarkable then the first bit of the watermark binary string is embedded. To embed a zero bit the VLC is left unchanged. To embed a one bit the VLC has one bit flipped according to the bit specified after using the binary tree structure. After the watermark bit is embedded the next VLC is read.

Decoding requires knowledge of which VLCs were used in the original image. Currently this information is encoded in an additional Huffman table, although alternative means are being explored.

3.5.3.1.2 Decoding
First, the added Huffman table containing the original VLCs is decoded. With these the binary tree structure can be created and watermarkable VLCs can be determined exactly the same way as was done on the encoding side. The resulting binary tree is then used for decoding. Every AC VLC is read in and traced on the binary tree. Each node in the tree is labeled as either an original VLC or as a watermarked VLC. Original VLCs are further marked in that they are either watermarkable or unwatermarkable. If a VLC is unwatermarkable it is skipped and no Meta Data is recorded. If the VLC is watermarkable, but is an original, then a zero Meta Data bit is recorded, and if a VLC is a watermarked VLC than a one bit is recorded.

All watermarked VLCs are a one-to-one mapping so during decoding any VLCs that were watermarked by a one Meta Data bit can be returned exactly to their original form. Therefore a JPEG viewer aware of the watermarking would be able to display the image properly, however, any standard viewer would most likely not be able to display the image at all due to loss of synchronization caused by the watermarking.

3.5.3.2 Additional Appended Bits Method
This method is exactly the same as the basic method except that it takes into account how a standard JPEG viewer would try to display the watermarked image. Additional bits are added to maintain synchronization with a standard viewer. This method will not work if pairs of VLCs are considered due to the fact that additional appended bits would have to be added both in the middle of the pair and at the end.
3.5.3.2.1 Encoding

Encoding is the same as the basic method except that VLCs that are watermarked with a one Meta Data bit are checked to see what the new run/size combination a standard viewer will interpret the watermarked VLC as. The size determines how many appended bits are expected after a VLC. The length of the interpreted VLC plus the interpreted size is how many bits the decoder expects to be there. Since the both the interpreted VLC length and interpreted size may be different than the original (unwatermarked VLC), than the expected number of bits may be different than the number that is actually there. In most cases the standard viewer will expect more bits than are actually there. In this case, the number of bits that are different are appended after the original appended bits. Any added appended bits can be considered Meta Data bits.

3.5.3.2.2 Decoding

Decoding is accomplished the same as the basic method except that all additional appended bits are considered Meta Data bits and are recorded appropriately. Additional appended bits can be read by decoding what a watermarked VLC was originally and finding the difference in interpreted length from original length and the difference in interpreted size from the original size (size referring to category). The total sum difference is then the number of appended bits. These appended bits are considered Meta Data (watermark) bits.

3.6 Results

The proposed algorithm was applied to a JPEG image downloaded off of the World Wide Web. The image was originally color and was converted to grayscale using Microsoft Photo Editor. There were no other image manipulations performed on the image prior to watermarking. The algorithm was performed both with and without pairing of the VLCs. The results are shown in Table 4.

<table>
<thead>
<tr>
<th>Table 4 Watermarking JPEG, Unpaired VLCs vs. Paired VLCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unpaired</td>
</tr>
<tr>
<td>VLCs used</td>
</tr>
<tr>
<td>Occupancy</td>
</tr>
<tr>
<td>Watermarkable VLCs</td>
</tr>
<tr>
<td>Total AC VLCs in image</td>
</tr>
<tr>
<td>Watermarkable VLCs in image</td>
</tr>
<tr>
<td>Watermark capacity</td>
</tr>
</tbody>
</table>

3.7 Possible Modifications

There are a number of possible modifications to the algorithm to further enhance the characteristics of the JPEG watermarking algorithm. The first modification is to remove the added Huffman table. The added Huffman table increases the file size 79 bytes. It is possible to tell the decoder which VLCs were the
originals by altering the existing Huffman tables. Any VLC that was not an original should have its run/size altered from the standard table. This would indicate to the decoder that it is a watermarked VLC.

Another modification is to only watermark one bits to zero bits. Another reason for a small increase in file size is due to the fact that when watermarking, 0xFF are created. For JPEG anytime there is a 0xFF after an integer number of bytes this indicates the beginning of a header. To tell the JPEG decoder that this is not the case a 0x00 byte must be placed after any 0xFF. By watermarking only one bits to zero bits there will be no possibility of having to add any zero bytes and there is a possibility of removing some, thus there is potential to actually decrease the file size by adding watermark bits.

A third modification is to have the binary tree structure be aware of the JPEG run/size VLC structure. Currently the binary code tree's only input is the set of VLCs that are used in the image. This could greatly improve statistics when applying watermark masking as described in the next section.

### 3.8 Minimizing Visual Distortions

If watermarking is performed as described so far, the result will be that some, if not all, of the used VLCs can be watermarked, or mapped, to some of the unused VLCs (from the total 162). If a watermark were inserted according to this mapping, it would be enough for a watermark aware decoder to be able to extract the watermark and/or return the image to its original state. On the other hand, how would a watermark unaware, or standard, viewer handle the watermarked image? The best scenario would be that the standard viewer would be able to display the image and there would be no noticeable difference between the watermarked version and the original image. As the algorithm is described so far, this would not be the case. By mapping a used VLC to an unused VLC this is also mapping the actual run/size of the used VLC to a new run/size of the unused VLC. It should be clear that if no watermarking was performed on the image then modifying the Huffman table run/size combinations for unused VLC will not alter the image in anyway. So if a VLC is watermarked to a new run/size then changing the value of the run/size will only affect how the watermarked VLC is interpreted by standard decoders.

At this point the algorithm must consider that there are appended bits associated with each VLC. Any decoder must be able to decode the current VLC and then correctly identify how many appended bits follow it to be able to find the start of the next VLC and keep the correct synchronization. The appended bits for a VLC are equal to the size of the run/size combination that corresponds to the decoded VLC. When a used VLC is watermarked it can be mapped to an unused VLC that is either the same length, longer, or shorter (based on the prefix condition).

If the watermarked and unwatermarked VLCs are the same length then the number of appended bits that are actually there should be how much a standard decoder expects to keep synchronization. However, the run/size of the watermarked VLC will be interpreted as different than the original in the run, size, or both the run and size. If the size is different, than the number of appended bits that the decoder expects will be different than what is actually there. This will cause synchronization to be lost. If the run is different than
this can cause significant visual artifacts in the particular block that the watermarked VLC occurs in (usually visual artifact in this case will be blockiness where the edges of the block are distinguishable).

The solution to this problem is to modify the run/size of what watermarked VLCs will be interpreted as. Since watermarked VLCs are only mapped to unused VLCs, then modifying the run/size that corresponds to the unused VLC will only affect how watermarked VLCs are displayed. Ideally the run/size should be changed to match the original run/size prior to watermarking. However, this is constrained by making sure that the decoder expects the same number of bits as that are actually there. This becomes further complicated by the fact that watermarked VLCs may be interpreted as different lengths than the original VLC. This results in three possibilities: interpreted length of the watermarked VLC is equal to the actual length of the original VLC, the interpreted length is less than the actual length, or the interpreted length is greater than the actual length. These three cases will be illustrated in the following examples.

### 3.8.1 Examples

At this point, it is already known which bit of which VLCs can be watermarked. Which bits that can be watermarked were determined based solely on the VLCs that actually occurred within the JPEG image. This watermarking will map a VLC that is used in the image to another VLC that is unused in the image. However, both used and unused VLCs are part of the standard AC VLC table that consists of 162 VLCs that include all possible run/size combinations.

For a watermark unaware JPEG decoder to maintain synchronization the number of bits for each VLC plus corresponding appended bits must be known. Choosing which bit to watermark guarantees that the result will be a prefix to an unused VLC or that an unused VLC will be a prefix to the result. This is because in choosing a bit to watermark the resulting watermarked VLC can not be a prefix to another used VLC or vice versa. Ideally the lengths of the used VLC and the watermarked version it was mapped to, would be desired to be the same, but this is not guaranteed because of the prefix condition of unused VLCs (see figure 1)—allowing for the result to be longer or shorter than the unused VLC the decoder will interpret it as. However, the run can always be modified to be the same as the original, but the size must be adjusted to account for length differences between the numbers of bits that are actually there and how many the decoder would expect to be there to be. The run can always be modified since the run is for an unused VLC. Therefore changing the run only affects what a watermarked VLC will be decoded as and the run does not determine the number of appended bits. The size indicates the number of appended bits to follow a decoded VLC so must be equal to the number of bits prior to the start of the next VLC.
Figure 12- VLC is watermarked by flipping bit 11 from ‘0’ to ‘1’. The result is a prefix to two unused VLCs with run/size equal to 5/5 and 5/6 respectively.

To be able to modify the size, the total length of the original VLC plus actual appended bits must be equal to the interpreted VLC length of the watermarked VLC plus the modified size.

If the lengths of both the used VLC and the interpreted length are the same then the number of appended bits required should be the same. In this case it will always be possible to replace the run size of an unused VLC with the original run/size. This will make watermarking this particular VLC completely invisible since nothing is changed for display purposes (see Table 5).

To reiterate, a VLC that occurs in the image has an actual run/size assigned to it. This VLC is mapped to an unused VLC, one that does not occur in the image, through watermarking (flipping exactly one bit). The unused VLC has a run/size that is initially assigned to it in the standard VLC table since all 162 used and unused VLCs appear on the table. The run/size for the watermarked VLC will only be used for displaying the watermarked VLC by watermark unaware decoders. Therefore to minimize the visual impact of watermarking the run/size for the modified VLC should be changed to the run/size of the original VLC. The constraint is that no appended bits can be added or subtracted to maintain file-size preservation and lossless reversibility.
Another possibility is that the interpreted VLC length is longer than the actual length. If this is the case then there is the possibility that a single VLC when watermarked could be decoded as two, or more, different VLCs. Since the length of the actual VLC and the decoded watermarked versions will be different, the run/size of the decoded versions can not be made to match the original exactly. The run can always be made identical but the size will have to be used to keep synchronization. This means that if the length difference is one, then the size must be modified to be one less than the original. This will result in a difference in the image, but should be far less than what the human visual system can detect. Also, since the decoder could interpret the watermarked version as multiple different VLCs, each of the possible interpreted VLC’s run/size combinations must be altered. They will all be modified to the same new run/size. Since the length of the watermarked versions are longer than the actual VLC bits, what will determine how the watermarked versions are decoded as will be the appended bits. If the length is one bit longer than the first appended bit will determine which of two unused VLCs the watermarked VLC will be decoded as.

Table 5-Interpreted length and actual length are equal.

<table>
<thead>
<tr>
<th></th>
<th>Actual</th>
<th>Watermarked</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLC</td>
<td>1111111000</td>
<td>1111111001</td>
</tr>
<tr>
<td>Run/Size</td>
<td>4/2</td>
<td>11/1</td>
</tr>
<tr>
<td>Length</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>Size</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>New Size = 2, New Run/Size = 4/2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6-Interpreted length longer than actual length

<table>
<thead>
<tr>
<th></th>
<th>Actual</th>
<th>Watermarked</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLC</td>
<td>1111111100000</td>
<td>1111111110100000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>or 1111111110100001</td>
</tr>
<tr>
<td>Run/Size</td>
<td>8/2</td>
<td>5/5 or 5/6</td>
</tr>
<tr>
<td>Length</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>Size</td>
<td>2</td>
<td>5 or 6</td>
</tr>
<tr>
<td>Total</td>
<td>17</td>
<td>21</td>
</tr>
<tr>
<td>New Size = 1, New Run/Size = 8/1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The final, and most problematic, possibility is that the interpreted VLC length will be shorter than the actual length. In this case it is possible that two or more actual VLCs when watermarked will be interpreted as the same VLC. It is important to note that the watermarked VLCs are distinct however the first N bits of each are the same. Also, the first N bits of each happen to be an unused VLC. Therefore, a watermark unaware decoder will interpret the first N bits of any of these watermarked VLCs as the same valid VLC. Since multiple VLCs are interpreted the same it is likely that multiple different runs and/or sizes will be mapped to the same VLC. Therefore it is impossible to mask these well since it will only be possible to modify one run/size for all of them. This causes the most problems because the VLC lengths plus appended bits can be different making it impossible to keep synchronization.

### Table 7-Interpreted length shorter than actual length

<table>
<thead>
<tr>
<th>VLC</th>
<th>Actual</th>
<th>Watermarked</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1111110111 or 1111110110</td>
<td>111111010</td>
</tr>
<tr>
<td>Run/Size</td>
<td>5/2 or 1/5</td>
<td>12/1</td>
</tr>
<tr>
<td>Length</td>
<td>11</td>
<td>10</td>
</tr>
<tr>
<td>Size</td>
<td>2 or 5</td>
<td>1</td>
</tr>
<tr>
<td>Total</td>
<td>13 or 16</td>
<td>11</td>
</tr>
</tbody>
</table>

New Size = 3, New Run/Size = 5/3

3.8.2 Results

The algorithm was applied to the Lena.jpg image. This image was 512x512 pixels and in color when downloaded off of the World Wide Web. The only image manipulation was converting the image to grayscale using Microsoft Photo Editor. Through analysis the image was determined to be baseline JPEG with a quality factor of 90 and entropy encoded using the example Huffman tables provided in the JPEG standard.

As shown in Table 8, the Lena image had only 58 VLCs occur in the image, although the standard Huffman table defines 162. This allowed for watermarking and masking the watermarked VLCs. Only 31 were watermarked, however, 34 could have been watermarked but were not due to collisions. Three pairs of VLCs when watermarked would have been interpreted as the same new VLC. In this case it is desired to replace the run/size of the new VLC with the run/size of the old VLC. However, since two different VLCs, each with different run/size was watermarked to the same new VLC (for each pair) it is impossible to redefine the run/size to mask the watermark. Thus, only one VLC in each pair could be watermarked to maintain the watermark masking. This resulted in a loss of about 400 possible watermark bits.
### Table 8 Statistics for Lena.jpg image with watermark masking algorithm

<table>
<thead>
<tr>
<th></th>
<th>Lena.jpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLCs in image</td>
<td>58</td>
</tr>
<tr>
<td>Watermarkable VLCs</td>
<td>31</td>
</tr>
<tr>
<td>File size of image before watermarking</td>
<td>46,733 bytes</td>
</tr>
<tr>
<td>File size of image after watermarking</td>
<td>46,820 bytes</td>
</tr>
<tr>
<td>Number of watermark bits embedded (random bit sequence)</td>
<td>2300</td>
</tr>
<tr>
<td>Watermark capacity</td>
<td>0.61%</td>
</tr>
</tbody>
</table>

There is a slight increase in file size caused by the watermarking, of 87 bytes. 79 bytes of these is due to adding an additional Huffman table specifying the original VLCs. This makes decoding extremely easy since the decoder has the same information as the encoder without having to parse the whole image. Through analysis of the watermarked image, the decoder should be able to determine the original VLCs without need for the table, and this concept is being examined for future versions of the algorithm.

![Figure 13 Original (left) and watermarked image (right) with 2300 watermark bits](image-url)
Appendix

1.0 Results
This section of the appendix has tables and figures for all results found throughout the paper and is intended as a quick reference. For descriptions and explanations of the results please see the corresponding section in the body of the paper.

1.1 English Alphabet
Table 9 Results from applying general algorithm to English alphabet

<table>
<thead>
<tr>
<th>RVLC Codeword-pairs</th>
<th>Good Table</th>
<th>Great Table</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>676</td>
<td>613</td>
<td>434</td>
<td>4%</td>
</tr>
</tbody>
</table>

1.2 JPEG
Table 10 Applying algorithm to JPEG Lena image (without Watermark Masking)

<table>
<thead>
<tr>
<th>Unpaired</th>
<th>Paired</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLCs used</td>
<td>58</td>
</tr>
<tr>
<td>Occupancy</td>
<td>99.85%</td>
</tr>
<tr>
<td>Watermarkable VLCs</td>
<td>39</td>
</tr>
<tr>
<td>Total AC VLCs in image</td>
<td>44638</td>
</tr>
<tr>
<td>Watermarkable VLCs in image</td>
<td>4292</td>
</tr>
<tr>
<td>Watermark capacity</td>
<td>1.15%</td>
</tr>
</tbody>
</table>

1.3 JPEG with Watermark Masking
Table 11 Watermarking of JPEG Lena image with watermark masking

<table>
<thead>
<tr>
<th>Lena.jpg</th>
</tr>
</thead>
<tbody>
<tr>
<td>VLCs in image</td>
</tr>
<tr>
<td>Watermarkable VLCs</td>
</tr>
<tr>
<td>File size of image before watermarking</td>
</tr>
<tr>
<td>File size of image after watermarking</td>
</tr>
<tr>
<td>Number of watermark bits embedded (random bit sequence)</td>
</tr>
<tr>
<td>Watermark capacity</td>
</tr>
</tbody>
</table>
2.0 Code

Code for running the algorithm on JPEG images is included for reference in this section of the appendix. The code can be broken up into three steps (see Figure 15 below). The first step is parsing a JPEG image using MATLAB code. This outputs a list used VLCs in plain text format. This is passed onto the C code which creates a binary tree structure based on the VLCs. From the binary tree, watermarkable VLCs are determined. A list of watermarkable VLCs paired with their watermarked versions is output from the program in plain text format. This new list is then read in by additional MATLAB code which also receives stored variables from the initial parsing of the JPEG image. The MATLAB code will create a binary string to act as the watermark (currently just a random bitstring, but easily modified to embed metadata), and embed the watermark in the JPEG image. The final result is a watermarked image in JPEG format. Currently the code is set to run with watermark masking so that the watermarked image and original appear the same.

Parsing the JPEG image (currently reading in file name LenaG.jpg) is carried out using jpeg_main.m. This will write out the VLCs used in the JPEG image (set as a variable in the program) to a new text file (currently aht_Lena.txt). aht_Lena.txt must be then passed onto the C code. All variables created from this code should be saved in order to be used by watermark.m when inserting the watermark.

After compiling the C code, it is run by typing ./test aht_Lena.txt. aht_Lena.txt could be replaced by any VLC table and the code will output a list of watermarkable VLCs paired with the watermarked versions. The input file must be plain text format with one VLC per line and no spaces before or after any VLCs. The output will be in the same format with the watermarked version of a VLC appearing on the line following the original.

The text file containing the watermarkable VLCs is passed onto watermark.m. Currently the file name being read in is LenaG_ws.txt. Additionally, the variables from initially running jpeg_main.m are loaded.
under the name LenaG_visual_6_22.mat. Both file name and saved variable name appear in the beginning of watermark.m and can be changed to match current names. The output from watermark.m is a watermarked version of the read in JPEG image (currently LenaG.jpg) to a new jpeg file currently named LenaG_w.jpg.

Figure 15 Flowchart for JPEG watermarking code

2.1 Binary Tree Code (in C)

2.1.1 pairtree.c
/*
 * RJ Berger
 * 2/27/04
 *
 * pairtree.c
 *
 */

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "tree.h"
#include "pair.h"
#include "goodtree.h"
#include "textfile.h"

#define maxNUMvlcs 200
#define maxLength 20
int main(argc, argv)
    int argc;
    char *argv[];
{
    char *textfile;
    node *current, *singleRoot, *pairRoot, *goodRoot;
    VLCtable myVLCtable, pairTable;
    VLC myVLC;
    int i, j, numVLCS, *index, numGreat, myIndex=0;
    double *numLeaves, leaves=0;
    char *VLCS[maxNUMvlcs];
    char *path;
    FILE *infile;
    int *totalVLCS, mytotalVLCS, *totalLength, mytotalLength, test;

    if(argc != 2)
    {
        printf("USAGE: ./test filename.txt
"); return(0);
    } else {
        textfile=argv[1];
    }

    goodRoot=addNode(NULL, TREEROOT);
    pairRoot=addNode(NULL, TREEROOT);

    numLeaves=&leaves;
    index=&myIndex;
    totalVLCS=&mytotalVLCS;
    totalLength=&mytotalLength;

    for(i=0; i<maxNUMvlcs; i++)
    {
        VLCS[i]=malloc((maxLength+1)*sizeof(char));
    }

    infile=fopen(textfile, "r");
    if(infile==NULL) printf("error: reading file
");
    test=read_VLCs(infile, VLCS, totalVLCS, totalLength);
    if(test!=1) printf("read_VLCs error\n");
    fclose(infile);

    path=GetBlock((maxLength*2-1)*sizeof(char));
    for(i=0; i<maxLength*2-1; i++)
    {
        path[i]=NULL;
    }
myVLCtable=New(VLCtable);
numVLCs=0;

for(i=0;i<(*totalVLCs);i++)
{
  myVLC=New(VLC);
  (char *)myVLC=VLCs[i];
  myVLCtable->VLClist[numVLCs]=myVLC;
  numVLCs++;
}
myVLCtable->VLClist[numVLCs]=NULL;

/********** CREATES VLC TREE (not paired) **********/

singleRoot=createOriginalTree(myVLCtable);
current=countLeaves(singleRoot, numLeaves);
/*printf("Single numLeaves=%f\n",*numLeaves);*/
(*numLeaves)=0;
/*printTree(singleRoot);*/

myVLCtable->VLClist[numVLCs]=NULL;

pairTable=New(VLCtable);
numVLCs=0;
for(i=0;i<(*totalVLCs);i++)
{
  for(j=0;j<(*totalVLCs);j++)
  {
    myVLC=New(VLC);
    myVLC=VLCpair(myVLCtable->VLClist[i],myVLCtable->VLClist[j],myVLC);
    pairTable->VLClist[numVLCs]=myVLC;
    numVLCs++;
  }
}
pairTable->VLClist[numVLCs]=NULL;

/********** CREATE PAIRED VLC TABLE (original) **********/

pairRoot=createOriginalTree(pairTable);
current=countLeaves(pairRoot, numLeaves);
/*printf("Paired numLeaves=%f\n",*numLeaves);*/
(*numLeaves)=0;

myVLCtable->VLClist[numVLCs]=NULL;

/********** CREATE GOOD TREE **********/

goodRoot=copyTree(singleRoot,goodRoot);
current=countLeaves(goodRoot, numLeaves);
(*numLeaves)=0;
goodRoot=createGoodTree(goodRoot, path, index);
current=countLeaves(goodRoot, numLeaves);

/*printTree(goodRoot);*/
/*printf("Good numLeaves=%f",*numLeaves);*/
(*numLeaves)=0;

/* ***********************************************
*********** CHECK STATS **************************/
numGreat=uniqueGoodCount(goodRoot,(*totalVLCs)*(*totalVLCs));
printf("number for watermark=%d\n",numGreat);

return(1);
}

2.1.2 watermark.c
/*
 * RJ Berger
 * 2/27/04
 * * pair.c
 * *
 */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "tree.h"
#include "pair.h"
#include "goodtree.h"
#include "textfile.h"
#define maxNUMvlcs 200
#define maxLength 20

/*****************  Main  *******************/

int main(argc,argv)
  int argc;
  char *argv[];
{
  if(argc != 2)
  {
    printf("USAGE: ./test filename.txt\n");
  }
return(0);
} else {
    textfile=argv[1];
}
return(1);
}

2.1.3 goodtree.h

/*
 * RJ Berger
 * 3/07/04
 *
 * goodtree.h
 *
 */

#ifndef _goodtree_h
#define _goodtree_h

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>
#include "tree.h"
#include "pair.h"

/******* Function Prototypes ***********/

node *createOriginalTree(VLCtable pairTable);
node *createGoodTree(node *originalRoot, char *path, int *index);
node *createGoodNodes(node *original, char *path, int *index);
node *checkPath(node *original, char *path);
int uniqueGoodCount(node *root, int numOriginal);
node *goodCount(node *root, int orginalAddress[], int *index);
int checkAddress(int nodeAddress, int originalAddress[]);
void printVLC(node *current);
void printPath(char *path, int *index);

/******* Functions *******************/

node *createOriginalTree(VLCtable pairTable)
{
    node *root;
    int i=0,status;

    root=addNode(NULL,TREEROOT);

    while(pairTable->VLClst[i]!=NULL) {
        status=placeVLC((char *)pairTable->VLClst[i], root, ORIGINAL);
        i++;
    }
node *createGoodTree(node *originalRoot, char *path, int *index)
{
    node *current=originalRoot;

    if(current==NULL)
    {
        printf("Empty Tree\n");
        return(NULL);
    } else {
        if(current->left!=NULL || current->right!=NULL)
        {
            if(current->left!=NULL)
            {
                path[(*index)]=LEFTBRANCH;
                (*index)++;
                current=createGoodTree((node *)current->left, path, index);
                (*index)--;
                path[(*index)]=NULL;
            }
            if(current->right!=NULL)
            {
                path[(*index)]=RIGHTBRANCH;
                (*index)++;
                current=createGoodTree((node *)current->right, path, index);
                (*index)--;
                path[(*index)]=NULL;
            }
            if(current->parent!=NULL)
            {
                return((node *)current->parent);
            } else {
                return(current);
            }
        } else {
            if((int)current->list==ORIGINAL)
            {
                current=createGoodNodes(current, path, index);
            }
            if(current->parent!=NULL)
            {
                return((node *)current->parent);
            } else {
                return(current);
            }
        }
    }
}

node *createGoodNodes(node *original, char *path, int *index)
{
    node *current;
    int branch,originalBranch,testbit=(*index-1);
    char tempPath[(*index)];

    int i;

    /*printf("in goodtree:createGoodNodes{ index=%d, path= ",*index);
    printPath(path,index);*/
current=GetBlock(sizeof(node));

strncpy(tempPath,path,(*index));
tempPath[(*index)]=NULL;

while(tempPath[testbit]>=0)
{
    if(tempPath[testbit]==LEFTBRANCH)
    {
        originalBranch=LEFTBRANCH;
        tempPath[testbit]=RIGHTBRANCH;
    } else {
        originalBranch=RIGHTBRANCH;
        tempPath[testbit]=LEFTBRANCH;
    }
    current=checkPath(original,tempPath);
    if(current!=NULL){
        break;
    }
    tempPath[testbit]=originalBranch;
    testbit--;
}
return(original);

node *checkPath(node *original, char *path)
{
    node *current=original, *root;
    int i=0;

    /******** Find Root **********/

    while(current->parent!=NULL)
    {
        current=(node *)current->parent;
    }

    i=0;
    while(path[i+1]!=NULL)
    {
        if((int)current->list==ORIGINAL || (int)current->list==GOOD)
        {
            return(NULL);
        }

        if(path[i]==LEFTBRANCH)
        {
            if(current->left==NULL)
            {
                current=addNode(current,LEFTBRANCH);
            } else {
                current=(node *)current->left;
            }
        } else {
            if(current->right==NULL)
            {
                current=addNode(current,RIGHTBRANCH);
            } else {
                current=(node *)current->right;
            }
        }
    }
}
current=addNode(current,RIGHTBRANCH);
} else {
    current=(node *)current->right;
}
}
i++;
}
if((int)current->list==ORIGINAL || (int)current->list==GOOD)
{
    return(NULL);
}

if(path[i]==LEFTBRANCH)
{
    if(current->left==NULL)
    {
        current=addNode(current,LEFTBRANCH);
        setList(current,original,GOOD);
        return(current);
    } else {
        return(NULL);
    }
} else {
    if(current->right==NULL)
    {
        current=addNode(current,RIGHTBRANCH);
        setList(current,original,GOOD);
        return(current);
    } else {
        return(NULL);
    }
}

int uniqueGoodCount(node *root, int numOriginal)
{
    int originalAddress[numOriginal];
    int i,j=0,numUniqueGood=0;
    node *current=root;
    int *index;
    index=&j;

    for(i=0;i<numOriginal+1;i++)
    {
        originalAddress[i]=NULL;
    }

    current=goodCount(current,originalAddress,index);
    i=0;
    while(originalAddress[i]!=NULL)
    {

numUniqueGood++; 
i++; 
} 
return(numUniqueGood); 
}

node *goodCount(node *root, int originalAddress[],int *index)
{
    node *current=root;
    node *print;
    int test;

    if(current==NULL)
    {
        printf("Error: Empty Tree\n");
        return(NULL);
    }
    else
    {
        if(current->left!=NULL || current->right!=NULL)
        {
            if(current->left!=NULL)
            {
                current=goodCount((node *)current->left,originalAddress, index);
            }
            if(current->right!=NULL)
            {
                current=goodCount((node *)current->right,originalAddress, index);
            }
            if(current->original!=NULL)
            {
                print=(node *)current->original;
                printVLC(print);
                print=current;
                printVLC(print);
                test=checkAddress((int)current->original,originalAddress);
                if(test==1)
                {
                    originalAddress[*index]=(int)current->original;
                    (*index)++;
                }
            }
        } else {
            return((node *)current->parent);
        }
    } 
    else {
        if(current->original!=NULL)
        {
            print=(node *)current->original;
            printVLC(print);
            print=current;
            printVLC(print);
            test=checkAddress((int)current->original,originalAddress);
            if(test==1)
            {
            
            }
        }
    } 
}
originalAddress[*index]=(int)current->original;
(*index)++;}
}
if(current->parent!=NULL){
return((node *)current->parent);
} else {
return(current);
}
}

int checkAddress(int nodeAddress, int originalAddress[])
{
    int i=0;
    while(originalAddress[i]!=NULL)
    {
        if(originalAddress[i]==nodeAddress) return(0);
        i++;
        return(1);
    }
}

void printVLC(node *original)
{
    char VLC[50];
    int i;
    node *temp, *current=original;
    for(i=0;i<51;i++){
        VLC[i]=NULL;
    }
    i=0;
    while(current->parent!=NULL)
    {
        temp=(node *)current->parent;
        if((node *)temp->left==current)
        {
            VLC[i]='0';
        } else if((node *)temp->right==current)
        {
            VLC[i]='1';
        } else
        {
            printf("error in printVLC\n");
        }
        current=temp;
        i++;
    }
    i--;
    while(i>=0)
    {
        printf("%c",VLC[i]);
        i--;
    }
}
void printPath(char *path, int *index)
{
    int i;

    for(i=0;i<*index;i++)
    {
        printf("%c", path[i]);
    }
    printf(" }\n");
}

#endif

2.1.4 tree.h
/*
 * RJ Berger
 * 3/7/04
 *
 * tree.h
 *
 */
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#ifndef _tree_h
#define _tree_h

/***************** PERMANENT CONSTANTS ****************/
#define LEFTBRANCH 48
#define RIGHTBRANCH 49
#define TREEROOT 3
#define NULL 0
#define ORIGINAL 1
#define GOOD 2
#define GREAT 3

/***************** Structures *************/

/* node */
typedef struct {
    struct node *parent, *left, *right, *original, *watermark;
    int length, list;
} node;

/**************** Function Prototypes ********/
node *addNode(node *current, int branch);
node *printTree(node *root);
node *freeTree(node *root, int branch);
node *copyTree(node *root, node *newRoot);
node *countLeaves(node *root, double *numLeaves);
node *setRoot(node *root);
void setList(node *current, node *originalNode, int list);

/******************* Tree Functions **************/
node *createNode(void)
{
    node *newNode;
    newNode = (node *)malloc(sizeof(node));
    newNode->parent=NULL;
    newNode->left=NULL;
    newNode->right=NULL;
    newNode->original=NULL;
    newNode->watermark=NULL;
    newNode->length=0;
    newNode->list=NULL;
    return(newNode);
}

node *addNode(node *current, int branch)
{
    node *cn=current;
    if(branch==LEFTBRANCH)
    {
        current->left=(struct node *)malloc(sizeof(node));
        current=(node *)current->left;
        (node *)current->parent=cn;
        current->left=NULL;
        current->right=NULL;
        current->original=NULL;
        current->watermark=NULL;
        current->length=(cn->length)+1;
        current->list=NULL;
        return(current);
    } else if(branch==RIGHTBRANCH) {
        current->right=(struct node *)malloc(sizeof(node));
        current=(node *)current->right;
        (node *)current->parent=cn;
        current->left=NULL;
    }
node *printTree(node *root){

node *current=root;

if(current->left!=NULL || current->right!=NULL){
  if(current->left!=NULL){
    current=printTree((node *)current->left);
  }
  if(current->right!=NULL){
    current=printTree((node *)current->right);
  }
  printf("length = %d\n",current->length);
  if(current->parent!=NULL){
    return((node *)current->parent);
  } else {
    return(current);
  }
} else {
  if(current==NULL){
    printf("Empty Tree\n");
    return(NULL);
  } else {
    printf("length = %d\n",current->length);
    if(current->parent!=NULL){
      return((node *)current->parent);
    } else {
      return(current);
    }
  }
}
}

node *freeTree(node *root, int branch){

node *current=root;

if(current->left!=NULL || current->right!=NULL){
  if(current->left!=NULL){
    current=freeTree((node *)current->left,LEFTBRANCH);
  }
  if(current->right!=NULL){
    current=freeTree((node *)current->right,RIGHTBRANCH);
  }
} else {
  if(current==NULL){
    printf("Empty Tree\n");
    return(NULL);
  } else {
    printf("length = %d\n",current->length);
    if(current->parent!=NULL){
      return((node *)current->parent);
    } else {
      return(current);
    }
  }
}
if (current->parent != NULL) {
    current = (node *) current->parent;
    if (branch == LEFTBRANCH)
    {
        free(current->left);
    } else
    {
        free(current->right);
    }
    return (current);
} else {
    free(current);
    return (NULL);
}
}

node * copyTree(node * root, node * newRoot)
{
    node * current = root;
    node * curr_copy = newRoot;

    if (current->left != NULL || current->right != NULL) {
        if (current->left != NULL) {
            curr_copy = addNode(curr_copy, LEFTBRANCH);
            current = (node *) current->left;
            (node *) curr_copy->watermark = (node *) current->watermark;
            setList(curr_copy, (node *) current->original, (int) current->list);
            current = copyTree(current, curr_copy);
            curr_copy = (node *) curr_copy->parent;
        }
        if (current->right != NULL) {
            curr_copy = addNode(curr_copy, RIGHTBRANCH);
            current = (node *) current->right;
            (node *) curr_copy->watermark = (node *) current->watermark;
            setList(curr_copy, (node *) current->original, (int) current->list);
            current = copyTree(current, curr_copy);
            curr_copy = (node *) curr_copy->parent;
        }
    }
}
if (current->parent != NULL) {
    return ((node *)current->parent);
} else {
    return (newRoot);
}
}
else {
    if (current == NULL) {
        printf("Empty Tree\n");
        return (NULL);
    } else {
        if (current->parent != NULL) {
            return ((node *)current->parent);
        } else {
            return (current);
        }
    }
}
}

node *countLeaves(node *root, double *numLeaves) {
    node *current = root;
    if (current == NULL) {
        printf("Empty Tree\n");
        return (NULL);
    } else {
        if (current->left != NULL || current->right != NULL) {
            if (current->left != NULL) {
                current = countLeaves((node *)current->left, numLeaves);
            }
            if (current->right != NULL) {
                current = countLeaves((node *)current->right, numLeaves);
            }
            if (current->parent != NULL) {
                return ((node *)current->parent);
            } else {
                return (current);
            }
        } else {
            (*numLeaves)++;
            if (current->parent != NULL) {
                return ((node *)current->parent);
            } else {
                return (current);
            }
        }
    }
}
node *setRoot(node *root)
{
    printf("setRoot: length=%d\n", root->length);
    while (root->parent!=NULL)
    {
        root=(node *)root->parent;
        printf("setRoot: length=%d\n", root->length);
    }
    return(root);
}

void setList(node *current, node *originalNode, int list)
{
    if(list==ORIGINAL)
    {
        (node *)current->original=NULL;
        (int)current->list=ORIGINAL;
    }
    else if(list==GOOD)
    {
        (node *)current->original=originalNode;
        (node *)originalNode->watermark=current;
        (int)current->list=GOOD;
    }
    else
    {
        (node *)current->original=NULL;
        (int)current->list=NULL;
    }
}

#endif

2.1.5 textfile.h

/* RJ Berger * 3/7/04 */
* textfile.h *
*/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#ifndef _textfile_h
#define _textfile_h

#endif
#define _textfile_h
/*************** Function Prototypes ***************/
int read_VLCs(FILE *FP, char *VLCs[],int *totalVLCs, int *maxLegnth);

/*************** Text file functions ***********/
int read_VLCs(FILE *Fp, char *VLCs[],int *totalVLCs, int *maxLength)
{
    int tempLength=0,tempNum=0;
    char ch;
    int index=0;

    *totalVLCs=0;
    *maxLength=0;

    while ((ch=getc(Fp))!=EOF)
    {
        if(ch!='1' & ch!='0')
        {
            VLCs[(*totalVLCs)][index]=NULL;
            (*totalVLCs)++;
            if(index>(*maxLength))(*maxLength)=index;
            index=0;
        } else {
            VLCs[(*totalVLCs)][index]=ch;
            index++;
        }
    }
    VLCs[(*totalVLCs)][index]=NULL;
    (*totalVLCs)++;
    return(1);
}

#error

2.1.6 pair.h
/*
 * RJ Berger
 * 2/27/04
 *
 * pair.h
 *
 */

#ifndef _pair_h
#define _pair_h

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <string.h>

#endif
/************************ MODIFIABLE CONSTANTS ****************************/
#define MAXNUMVLCS 20000
#define MAXVLCLENGTH 30

/******************** Structures ********************/

typedef struct {
  char currentVLC[MAXVLCLENGTH*2+1];
} *VLC;

typedef struct {
  VLC VLClist[MAXNUMVLCS];
} *VLCtable;

/******************** Function Prototypes ********************/
VLC VLCpair(VLC myVLC, VLC myVLC2, VLC mypair);
int placeVLC(char VLC[], node *root, int list);
void *GetBlock(size_t nbytes);
void FreeBlock(void *ptr);
#define New(type) ((type) GetBlock(sizeof *((type) NULL)))
#define NewArray(n,type) ((type *) GetBlock((n)*sizeof(type)))

/******************** Functions ********************/

VLC VLCpair(VLC myVLC, VLC myVLC2, VLC mypair) {
  int length=0;
  strcpy((char *)mypair,(char *)myVLC);
  strcat((char *)mypair,(char *)myVLC2);
  length=strlen((char *)mypair);
  mypair->currentVLC[length]=NULL;
  return(mypair);
}

int placeVLC(char myVLC[], node *root, int list) {
  node *current=root;
  int index=0;
  char bit, temp;
  while(myVLC[index+1]!=NULL) {
    bit=myVLC[index];
    }
if(bit==LEFTBRANCH)
{
    if(current->left==NULL)
    {
        current=addNode(current,LEFTBRANCH);
    } else {
        current=(node *)current->left;
    }
} else if(bit==RIGHTBRANCH)
{
    if(current->right==NULL)
    {
        current=addNode(current,RIGHTBRANCH);
    } else {
        current=(node *)current->right;
    }
} else {
    printf("Error in pair.h : placeVLC\n");
    index++;
}

bit=myVLC[index];
if(bit==LEFTBRANCH)
{
    if(current->left==NULL)
    {
        current=addNode(current,LEFTBRANCH);
    } else {
        printf("ERROR in pair.h, couldn't add original node\n");
    }
} else if(bit==RIGHTBRANCH)
{
    if(current->right==NULL)
    {
        current=addNode(current,RIGHTBRANCH);
    } else {
        printf("ERROR in pair.h, couldn't add original node\n");
    } else {
        printf("ERROR in pair.h placeVLC 2\n");
    }
    setList(current, NULL, list);
    return(1);
}

void *GetBlock(size_t nbytes)
{
    void *result;

    result=malloc(nbytes);
    if(result==NULL) printf("error in GetBlock\n");
    return(result);
}

void FreeBlock(void *ptr)
{   free(ptr);
}

#endif

2.1.7 makefile
CC=gcc
test_file=LenaG
PROGS=pairtree.c
test:
   $(CC) -o test $(PROGS)
run:
   ./test aht/aht_$(test_file).txt > results/$(test_file)_p3.txt
clean:
   rm *~ test

2.1.8 aht.LenaG.txt
1010
00
01
100
1011
11010
1111000
11111000
1111110110
111111111000010
1100
11011
1111001
111110110
11111110110
1111111110000100
1111111110001001
1111111110001010
1111111110001011
111010
111110111
2.2 JPEG Application (in MATLAB)

2.2.1 jpeg_main.m

% RJ Berger
%
% this file will create statistics from given jpeg image; including
% which vlc's used and how often
%
% image is the file to be read in, must be in local directory

close all
clear
cle

image='lenaG.jpg';
table_number=1;
[jpeg_text,size]=jpeg_read(image);
[BITS,Huffval,end_last_huffman,begin_AC_huffman]=jpeg_parse(table_number,jpeg_text,size);

[Huffsize,lastK]=C1(BITS);
Huffcode=C2(Huffsize);
[EHUFCO1, EHUFSI]=C3(Huffval, Huffcode, Huffsize, lastK);
EHUFCO1=EHUFCO1(1:12,:);
EHUFCO1(12,1)=0;
EHUFCO1(12,2)=11;

table_number=2;
[jpeg_text,size]=jpeg_read(image);
[BITS,Huffval,end_last_huffman,begin_AC_huffman]=jpeg_parse(table_number,jpeg_text,size);

[Huffsize,lastK]=C1(BITS);
Huffcode=C2(Huffsize);
[EHUFCO2, EHUFSI]=C3(Huffval, Huffcode, Huffsize, lastK);

%% modified so wouldn't crash with only 2 tables

table_number=1;
[jpeg_text,size]=jpeg_read(image);
[BITS,Huffval,end_last_huffman,begin_AC_huffman]=jpeg_parse(table_number,jpeg_text,size);

[Huffsize,lastK]=C1(BITS);
Huffcode=C2(Huffsize);
[EHUFCO3, EHUFSI]=C3(Huffval, Huffcode, Huffsize, lastK);

%% changed for specific image
\[ \text{[Huffsize, lastK]} = C_1(\text{BITS}); \]
\[ \text{Huffcode} = C_2(\text{Huffsize}); \]
\[ \text{[EHUFCO4, EHUFSI]} = C_3(\text{Huffval, Huffcode, Huffsize, lastK}); \]

\%
rem
oves zero pads in VLC portion of jpeg_test
\[ \text{[no_zero_pads, num_zero_pads]} = \text{remove_zero_pads(jpeg_test, size)}; \]

\[ \text{new_size} = \text{length(no_zero_pads)}; \]

\[ \text{[DC_VLCs_used, AC_VLCs_used, zero_pad, AC_VLC_vector, actual_huffman_table, trailing_ones]} = \text{jpeg_parse_no_zero_pads(no_zero_pads, new_size, EHUFCO1, EHUFCO2, EHUFCO3, EHUFCO4)}; \]

\[ \text{DC_num_bits} = 0; \]
\[ \text{DC_num_VLCs_used} = 0; \]
\[ \text{different_DC_num_VLCs_used} = 0; \]

\text{for} \quad i = 1:12
\quad \text{if DC_VLCs_used(i,3) = 0}
\quad \quad \text{DC_num_bits} = \text{DC_num_bits} + \text{DC_VLCs_used(i,3)} * (\text{EHUFCO1(i,3)} + \text{DC_VLCs_used(i,2)});
\quad \quad \text{different_DC_num_VLCs_used} = \text{DC_num_VLCs_used} + \text{DC_VLCs_used(i,3)};
\quad \quad \text{DC_num_VLCs_used} = \text{DC_num_VLCs_used} + \text{DC_VLCs_used(i,3)};
\quad \text{end}
\text{end}

\[ \text{AC_num_bits} = 0; \]
\[ \text{AC_num_VLCs_used} = 0; \]
\[ \text{different_AC_num_VLCs_used} = 0; \]

\text{for} \quad i = 1:162
\quad \text{if AC_VLCs_used(i,3) = 0}
\quad \quad \text{AC_num_bits} = \text{AC_num_bits} + \text{AC_VLCs_used(i,3)} * (\text{EHUFCO4(i,3)} + \text{AC_VLCs_used(i,2)});
\quad \quad \text{different_AC_num_VLCs_used} = \text{AC_num_VLCs_used} + 1;
\quad \quad \text{AC_num_VLCs_used} = \text{AC_num_VLCs_used} + \text{AC_VLCs_used(i,3)};
\quad \text{end}
end

num_bits=AC_num_bits+DC_num_bits;
num_bytes=ceil(num_bits/8)+zero_pad;

%table=

%[vlc_table,vlc_list,good_list,great_list]=vlc_main(table(:,3:4),1);

2.2.2 jpeg_parse.m
function [BITS,Huffval,end_last_huffman,begin_AC_huffman]=jpeg_parse(table_number,jpeg_text,size)

num_tables=0;

i=1;
while i<size
    if jpeg_text(i)==255
        if (jpeg_text(i+1)==196)
            num_tables=num_tables+1;
            if(num_tables==2)
                begin_AC_huffman=i;
            end
            end_last_huffman=i+2+jpeg_text(i+3);
        end
        i=i+1;
    end
end
if num_tables==1
    multiple_tables_exist=0;
else
    multiple_tables_exist=1;
end
found_my_table=0;

while 1
i=1;
current_table=0;

while i<size
    if jpeg_text(i)==255
        if (jpeg_text(i+1)==196)
            current_table=current_table+1;
            if multiple_tables_exist
                if current_table == table_number
                    i=i+2;
                    found_my_table=1;
                    break
                end
            else
                i=i+2;
                found_my_table=1;
                break
            end
        end
    end
    end
    i=i+1;
end

if found_my_table & multiple_tables_exist==1
    temp=bit2vlc([vlc2bit(jpeg_text(i),8) vlc2bit(jpeg_text(i+1),8)]);
    Lh=temp(1);
    temp=vlc2bit(jpeg_text(i+2),8);
    Tc=temp(4);
    Th=temp(8);
    Li(1:16)=jpeg_text(i+3:i+18);
    num_Vs=sum(Li);
    end_Vs=i+18+num_Vs;
    %j=1;
    %while j<num_Vs+1
    Vij(1:num_Vs)=jpeg_text(i+19:end_Vs);
    % j=j+1;
    %end

    BITS=Li;
    for j = 1:num_Vs
temp1=vlc2bit(Vij(j),8);

temp2=bit2vlc(temp1(1:4));

temp3=bit2vlc(temp1(5:8));

Huffval(j,1:2)=[temp2(1) temp3(1)];

end

break;

else

while current_table~=table_number

if current_table==1

    temp=bit2vlc([vlc2bit(jpeg_text(i),8) vlc2bit(jpeg_text(i+1),8)]);

    Lh=temp(1);

else

    i=i-2;

end

temp=vlc2bit(jpeg_text(i+2),8);

Tc=temp(4);

Th=temp(8);

Li(1:16)=jpeg_text(i+3:i+18);

num_Vs=sum(Li);

end_Vs=i+18+num_Vs;

j=1;

while j<num_Vs+1

    Vij(1:num_Vs)=jpeg_text(i+19:end_Vs);

    j=j+1;

end

BITS=Li;

for j = 1:num_Vs

    temp1=vlc2bit(Vij(j),8);

    temp2=bit2vlc(temp1(1:4));

    temp3=bit2vlc(temp1(5:8));

    Huffval(j,1:2)=[temp2(1) temp3(1)];

end

i=end_Vs+1;

current_table=current_table+1;

end

if current_table==1

    temp=bit2vlc([vlc2bit(jpeg_text(i),8) vlc2bit(jpeg_text(i+1),8)]);

    Lh=temp(1);
else
    i=i-2;
end

temp=vlc2bit(jpeg_text(i+2),8);
Te=temp(4);
Th=temp(8);
Li(1:16)=jpeg_text(i+3:i+18);
num_Vs=sum(Li);
end_Vs=i+18+num_Vs;

for j = 1:num_Vs
    temp1=vlc2bit(Vij(j),8);
    temp2=bit2vlc(temp1(1:4));
    temp3=bit2vlc(temp1(5:8));
    Huffval(j,1:2)=[temp2(1) temp3(1)];
end
break
end

2.2.3 remove_zero_pads.m

function [no_zero_pads, num_zero_pads]=remove_zero_pads(jpeg_text,size)

jpeg_text=jpeg_text';

i=1;
while i<=size
    if jpeg_text(i)==255
        if (jpeg_text(i+1)==218)
            Ls1=vlc2bit(jpeg_text(i+2),8);
        end
    end
    i=i+1;
end
Ls2=vlc2bit(jpeg_text(i+3),8);
Ls=[Ls1 Ls2];
Ls=bit2vlc(Ls);
j=i+Ls(1)+2;
break;
end
end
i=i+1;
end

num_zero_pads=0;
k=1;
index=1;
while j<size
  if jpeg_text(j)==255 & (jpeg_text(j+1)==217)
    fprintf('%s %d
','remove_zero_pads reached: EOF,j);
    no_zero_pads=jpeg_text;
    return
  else
    zero_position=0;
    if jpeg_text(j)==255 & jpeg_text(j+1)==0
      temp=jpeg_text(1:j);
      temp2=jpeg_text(j+2:size);
      jpeg_text=[temp temp2];
      size=size-1;
      num_zero_pads=num_zero_pads+1;
      j=j+1;
    elseif jpeg_text(j)==255
      'error line 47 of remove_zero_pads: header not recognized(J)
    else
      j=j+1;
    end
  end
end

2.2.4 actual_huffman.m

function actual_huffman_table=actual_huffman(AC_VLCs_used,AC_EHUFCO)
% RJ Berger
% 3.24.04
% actual_huffman.m

[rows,cols]=size(AC_EHUFCO);

temp=AC_VLCs_used;
temp2=AC_EHUFCO;
temp2(:,5)=AC_VLCs_used(:,3);
temp2(:,6)=(1:rows)';
[rows,columns]=size(AC_VLCs_used);
i=1;
index=rows;
while(i<=index)
    if(temp(i,3)==0)
        temp2(i:index-1,:)=temp2(i+1:index,:);
        temp(i:index-1,:)=temp(i+1:index,:);
        index=index-1;
        i=i-1;
    end
    i=i+1;
end
actual_huffman_table=temp2(1:index,:);

[rows,columns]=size(actual_huffman_table);
fid=fopen('actual_huffman.txt','w');
for(i=1:rows)
    vlcs=vlc2bit(actual_huffman_table(i,4),actual_huffman_table(i,3));
    fprintf(fid,'%d',vlcs);
    if(i<rows)
        fprintf(fid,'
');
    end
    vlcs=[];
end
fclose(fid);
2.2.5 bit2vlc.m

function x=bit2vlc(m)
%BIT2VLC Convert bit vector to decimal and length pair
%
% Example
%   bit2vlc([0 0 1]) returns [1 3]
%
% Copyright 2003 RJ Berger
% $Revision: 1.00$  $Date: 2003/04/16 12:04:32$

temp=0;
L=length(m);
m=fliplr(m);
for n=1:L,
    temp=temp+2^(n-1)*m(n);
end

x=[temp,L];

2.2.6 C1.m

function [Huffsize, lastK] = C1(BITS)

% RJ Berger
% JPEG code
%
% implement flow chart C1, page 51.

k=1;
i=1;
j=1;

while 1
    while j > BITS(i);
i = i + 1;
j = 1;
if i > 16
    break
end
end
if i <= 16
    Huffsize(k) = i;
k = k + 1;
j = j + 1;
else
    break
end
end
Huffsize(k) = 0;
lastK = k;

2.2.7 C2.m

function Huffcode = C2(Huffsize)

% RJ Berger
% JPEG code
%
% implement flow chart C2, page 52.

k = 1;
code = [0 2];
si = Huffsize(1);
Huffcode = [ ];

while 1
    Huffcode(k, 1:2) = code;
    code(1) = code(1) + 1;
k = k + 1;
    while Huffsize(k) ~= si
        if Huffsize(k) == 0
            break
        end
        code(1) = code(1) * 2;
        code(2) = code(2) + 1;
    end

end
si = si + 1;
if Huffsize(k) == si
    break
end
end
if Huffsize(k) == 0
    break
end
end

### 2.2.8 C3.m

function [EHUFCO, EHUFSI] = C3(Huffval, Huffcode, Huffsize, lastK)

% RJ Berger
% JPEG code
%
% implement flow chart C3, page 53.

k = 1;
while k < lastK
    tempEHUFCO(k, 1:4) = [Huffval(k, 1:2), Huffcode(k, 1:2)];
    tempEHUFSI(k) = Huffsize(k);
    k = k + 1;
end

EHUFCO = zeros(162, 4);
EHUFSI = zeros(1, 162);

i = 2;
while i < 152
    EHUFCO(i, 2) = mod(i - 1, 10);
    if EHUFCO(i, 2) == 0
        EHUFCO(i, 2) = 10;
    end
    EHUFCO(i, 1) = floor((i - 2) / 10);
    i = i + 1;
end

EHUFCO(152:162, 1) = 15;
EHUFCO(152:162, 2) = (0:10)';
k=1;

while k<lastK
    run=tempEHUFCO(k,1);
    size=tempEHUFCO(k,2);
    if run<15
        index=run*10+size+1;
    else
        index=run*10+size+2;
    end
    EHUFCO(index,3)=tempEHUFCO(k,4);
    EHUFCO(index,4)=tempEHUFCO(k,3);
    EHUFSI(index)=tempEHUFSI(k);
    k=k+1;
end

2.2.9 insert_zero_pads.m

function [with_zero_pads, num_zero_pads]=insert_zero_pads(jpeg_text,size)

i=1;
while i<=size
    if jpeg_text(i)==255
        if (jpeg_text(i+1)==218)
            Ls1=vlc2bit(jpeg_text(i+2),8);
            Ls2=vlc2bit(jpeg_text(i+3),8);
            Ls=[Ls1 Ls2];
            Ls=bit2vlc(Ls);
            j=i+Ls(1)+2;
            break;
        end
    end
    i=i+1;
end

num_zero_pads=0;

k=1;
index=1;
while j<size
    if jpeg_text(j)==255 & (jpeg_text(j+1)==217) & j==size-1
fprintf('%s %d\n','jpeg_parse_with_zero_pads reached: EOF',j);
with_zero_pads=jpeg_text;
return
else
    zero_position=0;
    if jpeg_text(j)==255
        temp=jpeg_text(1:j);
        temp2=jpeg_text(j+1:size);
        jpeg_text=[temp 0 temp2];
        size=size+1;
        num_zero_pads=num_zero_pads+1;
        j=j+1;
    else
        j=j+1;
    end
end

2.2.10 jpeg_read.m

function [jpeg_text,size]=jpeg_read(file)

    fid=fopen(file);
    F=fread(fid); % reads the file into F as a binary
    %s=char(F'); % converts binary ascii into character array

    jpeg_text=F;

    size=length(F); % total number of characters read

    fclose('all');

2.2.11 match_vlc.m

function [match,new_index]=match_vlc(bits,index,EHUFCO)

    %EHUFCO=current_EHUFCO;

    bits_length=length(bits);
if bits_length==4
    bits1=vlc2bit(bits(1),8);
    bits2=vlc2bit(bits(2),8);
    bits3=vlc2bit(bits(3),8);
    bits4=vlc2bit(bits(4),8);
    stream=[bits1 bits2 bits3 bits4];
    max_length=33-index;
elseif bits_length==3
    bits1=vlc2bit(bits(1),8);
    bits2=vlc2bit(bits(2),8);
    bits3=vlc2bit(bits(3),8);
    stream=[bits1 bits2 bits3];
    max_length=25-index;
elseif bits_length==2
    bits1=vlc2bit(bits(1),8);
    bits2=vlc2bit(bits(2),8);
    stream=[bits1 bits2];
    max_length=17-index;
else
    bits1=vlc2bit(bits(1),8);
    stream=[bits1];
    max_length=9-index;
end

%index

table_length=length(EHUFCO);
match=0;
i=1;
j=1;

while j<=max_length-1
    while i<=table_length
        temp=stream(index:index+j);
        temp_length=length(temp);
        if EHUFCO(i,3)==temp_length
            if vlc2bit(EHUFCO(i,4),EHUFCO(i,3))==temp
                match=i;
                new_index=index+temp_length;
            end
        end
        i=i+1;
    end
    j=j+1;
end
break
end
end
i=i+1;

if i>162
    stream
    index
end

end
if match ~= 0
    break
end
i=1;
j=j+1;
end

if match==0
    match=-1;
    new_index=-1;
end

### 2.2.12 vlc2bit

```
function x=vlc2bit(v,L)
%VLC2BIN Convert decimal and length pair to bit string
%
% Example
%   vlc2bit(1,3) returns [0 0 1]
%
% Copyright 2003 RJ Berger
% SRevision: 1.00  SDate: 2003/04/16 12:04:32

bits=[];
for n=1:L,
    if v==0
        bits=[bits,0];
    else
        if(v/2==round(v/2))
bits=[bits,0];
else
    bits=[bits,1];
end
v=floor(v/2);
end

x = fliplr(bits);

2.2.13 vlc_format.m

function vlc_table=vlc_format(string,file_format)
%VLC_FORMAT reads in file and converts to useful vlc format
%
% Example
%   vlc_table=vlc_format(file, file_format);
%
% file is a string name of a .txt file to open and file_format is an integer
% either 1 or 0, if 1 then file is already in vlc_format separated by commas and new lines,
% if 0 then file is bit format separated by new lines ; vlc_format will
% return a table of vlc's in vlc format
%
% Copyright 2003 RJ Berger
% $Revision: 1.01 $  $Date: 2003/06/12 11:38:10 $

fid=fopen(string);
F=fread(fid);                   % reads the file into F as a binary
s=char(F');                     % converts binary ascii into character array
vlc_table=[];

len=length(s);                  % total number of characters read

% this will read in the text file in vlc format (decimal value, length)
% and place the vlc list in variable vlc_table
currB=0;
currE=0;
if file_format==1
    j=1;
    index=[1 1]; % index into vlc_table
    while j<len
        currB=j;
        while (s(j) ~= ',' & double(s(j))~=13 & j<len)
            j=j+1;
        end
        if j==len
            currE=j;
        else
            currE=j-1;
        end
        curr_len=currE-currB+1;
        temp=str2num(s(currB:currE));
        if index(2)==1
            temp1=temp;
        else
            vlc_table(index(1),:)=[temp1,temp];
        end
        if s(j)==','
            index(2)=index(2)+1;
        elseif double(s(j))==13
            index(1)=index(1)+1;
            index(2)=1;
            j=j+1;
        end
        j=currE+2;
    end
elseif file_format==0
    % this part will do the same as the first except
    % assumes that the file is in binary(bit) format
    currB=0;

currE=0;  
j=1;  
index=1; % index into vlc_table  
while j<len  
    currB=j;  
    while (double(s(j))~=13 & j<len)  
        j=j+1;  
    end  
    if j==len  
        currE=j;  
    else  
        currE=j-1;  
    end  
    curr_len=currE-currB+1;  

    bit_string=[]; % temporarily holds bit string  
    for k=currB:currE,  
        bit_string=[bit_string, str2num(s(k))];  
    end  
    temp=bit2vlc(bit_string); % converts bit string to vlc_format  
    vlc_table(index(1),:)=temp;  
    if double(s(j))==13  
        index=index+1;  
        j=j+1;  
    end  
    j=currE+2;  
end  
end  
fclose('all');

2.2.14 watermark.m
%function watermarked_image=watermark(file)

clear  
clc
%%%%%%%%%%%% Creates watermark_bits variable %%%
% watermark_bits is a list of VLC_pairs in VLC format with which
% bit can be watermarked
%
file='LenaG_ws.txt';

%load 'LenaG_watermark_parse.mat'
load 'LenaG_visual_6_22.mat'

fid=fopen(file);
F=fread(fid); % reads the file into F as a binary
C_code_file=F;
size=length(F); % total number of characters read

i=1;
original=[];
wmrk=[];
watermark_bits=[];

first=1;
bit=0;
index=1;
num_watermarkable=0;
while i<=size
    if C_code_file(i)=='1'
        if first==1
            original=[original 1];
        else
            index=index+1;
            wmark=[wmark 1];
            if(original(index)~=wmark(index))
                bit=index;
            end
            num_watermarkable=num_watermarkable+1;
        end
    end
    i=i+1;
    elseif C_code_file(i)=='0'
        if first==1
            original=[original 0];
        else
            index=index+1;
            wmark=[wmark 1];
            if(original(index)~=wmark(index))
                bit=index;
            end
            num_watermarkable=num_watermarkable+1;
        end
    end
end
else
    index=index+1;
    wmark=[wmark 0];
    if(original(index)~=wmark(index))
        bit=index;
        num_watermarkable=num_watermarkable+1;
    end
end
i=i+1;
else
    if(i>size)
        break
    end
    while(c_code_file(i)~='1' & c_code_file(i)~='0')
        i=i+1;
        if(i>size)
            break
        end
    end
    if first==1
        first=0;
        index=0;
    else
        vlcs=bit2vlc(original);
        watermark_bits(num_watermarkable,:)=[vlcs bit];
        first=1;
        wmark=[];
        original=[];
    end
end
end

fclose('all');

%%%% Adds VLC number for each VLC in pair to variable watermark_bits %%%%
[temp_r, temp_c]=size(watermark_bits);
[rows_EHUFCO2, cols_EHUFCO2]=size(EHUFCO2);

i=1;

while(i<=temp_r)
    temp_vlc_pair=vlc2bit(watermark_bits(i,1), watermark_bits(i,2));
    match=0;
    j=1;
    while(match~=1)
        temp_vlc=temp_vlc_pair(1:j);
        temp_vlc_length=j;
        k=1;
        while(k<=rows_EHUFCO2)
            if(temp_vlc_length==EHUFCO2(k,3))
                temp_huff_vlc=vlc2bit(EHUFCO2(k,4), EHUFCO2(k,3));
                if(temp_vlc==temp_huff_vlc)
                    if(match==0)
                        temp_index=j+1;
                        match=1;
                        first=k;
                        break
                    end
                end
            end
            k=k+1;
        end
        j=j+1;
    end
    watermark_bits(i,4)=first;
    i=i+1;
end

%%% Visual Variable
% col 1 2 3 4 5 6 7 8
% RUN SIZE VALUE LENGTH WAT.LOC. ORG. #OCCUR WAT.VAL.

% col 9 10 11 12 13 14
% INTRP.AS KEY# INTERPRET LEN APPEND.NEDED UNUSED REF #PAD BITS
% visual is watermark_bits plus number of times vlc occurs in image

visual(:,3:6)=watermark_bits;
i=1;
while(i<=temp_r)
    visual(i,1:2)=actual_huffman_table(find(watermark_bits(i,4)==actual_huffman_table(:,6)),1:2);
    visual(i,7)=actual_huffman_table(find(watermark_bits(i,4)==actual_huffman_table(:,6)),5);
    i=i+1;
end

visual=-(sortrows(-visual,7));

% adds watermarked version (value) to column 8 of visual

i=1;
while(i<=temp_r)
    current_vlc=vlc2bit(visual(i,3),visual(i,4));
    current_vlc(visual(i,5))=xor(current_vlc(visual(i,5)),1);
    temp=bit2vlc(current_vlc);
    visual(i,8)=temp(1);
    i=i+1;
end

% finds what unaware decoder will interpret watermarked vlc as and stores it in visual column 9

index=1;
i=1;
while(i<=temp_r)
    bits=[0 0 255];
    temp=vlc2bit(visual(i,8),visual(i,4));
    if(visual(i,4)<8)
        temp2=[temp(1:visual(i,4)) zeros(1,8-visual(i,4))];
        temp2=bit2vlc(temp2);
        bits(1)=temp2(1);
    else
        temp2=temp(1:8);
        temp2=bit2vlc(temp2);
    end
    i=i+1;
end
bits(1)=temp2(1);
temp3=[temp(9:visual(i,4)) zeros(1,16-visual(i,4))];
temp3=bit2vlc(temp3);
bitems(2)=temp3(1);
end
visual(i,9)=match_vlc(bits, index, EHUFCO2);
if(visual(i,4)<EHUFCO2(visual(i,9)))
  % unaware decoder interpretts as multiple run/size
  %i
  i=i+1;
end

% determines key#:=original VLC length plus appended bits add to col. 10 %

i=1;
while(i<=temp_r)
  visual(i,10)=visual(i,4)+visual(i,2);
  i=i+1;
end

% determine interpreted length col. 11 %

i=1;
while(i<=temp_r)
  visual(i,11)=EHUFCO2(visual(i,9), 3);
  i=i+1;
end

% determine needed append bits col. 12 %

i=1;
while(i<=temp_r)
  visual(i,12)=visual(i,10)-visual(i,11);
  i=i+1;
end
% creates table of unused vlcs 
% col 1 2 3 4 5 6 7 
% CUR RUN CUR SIZE LENGTH VALUE INT.AS YES/NO ORIG. RUN ORIG.SIZE 
% 
% i=1; 
% unused_index=1; 
% while(i<=162) 
% temp=find(i==actual_huffman_table(:,6)); 
% temp=[temp 12345]; 
% if(temp(1)==12345) 
% unused(unused_index,:)=EHUFCO2(i,:); 
% unused_index=unused_index+1; 
% end 
% i=i+1; 
% end 
% unused(:,5)=0; 
% 
% i=1; 
% while(i<unused_index) 
% unused(i,6)=unused(i,1); 
% unused(i,7)=unused(i,2); 
% i=i+1; 
% end 
%
% % finds best unused to switch run/size with and adds reference to col. 13% 
% 
% i=1; 
% [unused_r unused_c]=size(unused); 
% no_same_run=[]; 
% while(i<=temp_r) 
% cur_run=visual(i,1); 
% append_needed=visual(i,12); 
% current_unused=find(unused(:,3)==EHUFCO2(visual(i,9),3) & unused(:,4)==EHUFCO2(visual(i,9),4)); 
% if (unused(current_unused,5)==1 & visual(i,12)<=unused(current_unused,2)) 
% else 
% unused(current_unused,5)=1; 
% if(append_needed==EHUFCO2(visual(i,9),2)) 
% visual(i,13)=current_unused; 
% end 
% end 
%
% %unused(current_used,1)=visual(i,1);
% else
%    j=1;
%    found_unused=0;
%    while(j<=unused_r)
%      if(unused(j,5)==0),
%        if(cur_run)==unused(j,1)
%          if(unused(j,2)>=append_needed)
%            %unused(j,5)=1;
%            visual(i,13)=current_used;
%            temp1=unused(current_used,1);
%            temp2=unused(current_used,2);
%            unused(current_used,1)=unused(j,1);
%            unused(current_used,2)=unused(j,2);
%            unused(j,1)=temp1;
%            unused(j,2)=temp2;
%            % determine number of bits that need to be added for padding col. 14 %
%            visual(i,14)=unused(current_used,2)-append_needed;
%            found_unused=1;
%            break
%          end
%        end
%      end
%      j=j+1;
%    end
%    if found_unused==0;
%      %'error in watermark.m no same run found'
%      no_same_run=[no_same_run i];
%    end
% end
% end
% i=i+1;
% end
% %
%
% %%%%%%%%%%%%%%%%%%%%% % REPEATS LAST STEP FOR ANY THAT DIDNT FIND SAME RUN %%%%%%%%%%%%%%%%%%%%
% % i=1;
% length_no_run=length(no_same_run);
while(i<=length_no_run)
    current_unused=find(unused(:,3)==EHUFCO2(visual(no_same_run(i),9),3) &
    unused(:,4)==EHUFCO2(visual(no_same_run(i),9),4));
    unused(current_unused,5)=0;
    i=i+1;
end


i=1;
while(i<=length_no_run)
    cur_run=visual(no_same_run(i),1);
    append_needed=visual(no_same_run(i),12);
    current_unused=find(unused(:,3)==EHUFCO2(visual(no_same_run(i),9),3) &
    unused(:,4)==EHUFCO2(visual(no_same_run(i),9),4));
    if (unused(current_unused,5)==1 & visual(no_same_run,12)<=unused(current_unused,2))
    else
        unused(current_unused,5)=1;
        if(append_needed==EHUFCO2(visual(no_same_run(i),9),2))
        visual(no_same_run(i),13)=current_unused;
        else
            j=1;
            found_unused=0;
            while(j<=unused_r)
                if(unused(j,5)==0),
                %if(append_needed==EHUFCO2(visual(no_same_run(i),9),2))
                visual(no_same_run(i),13)=current_unused;
                else
                %j=1;
                % found_unused=0;
                % while(j<=unused_r)
                % if(unused(j,5)==0),
                % if(append_needed==EHUFCO2(visual(no_same_run(i),9),2))
                % visual(no_same_run(i),13)=current_unused;
                % temp1=unused(current_unused,1);
                % temp2=unused(current_unused,2);
                % unused(current_unused,1)=temp1;
                % unused(current_unused,2)=temp2;
                %
                % determine number of bits that need to be added for padding col. 14 %
                visual(no_same_run(i),14)=unused(current_unused,2)-append_needed;
                %found_unused=1;

break
end

end
j=j+1;
end
if found_unused==0;
'error in watermark.m no unused found'
no_same_run=[no_same_run i];
end
end
end
i=i+1;
end

%%%% Corrects for number of pad bits when multiple watermarked VLCs are
%%%% interpreted as same run/size

i=1;
while(i<=temp_r)
visual(i,14)=unused(visual(i,13),2)-visual(i,12);
i=i+1;
end

%%%% ORIGINAL RUN/SIZE and what they should be changed to are stored in
%%%% variable UNUSED   %%%  ORIGINAL col. 6,7   NEW col. 1,2

%% Number of additional pad bits (can be watermark bits) for each VLC is
%% stored in variable VISUAL column 14
%% use this section to find watermark capacity (if less than MAX bits)
% i=1;
% MAX=10000;
% while(i<=MAX)
%   watermark=[];
%   watermark=[watermark 0];
%   i=i+1;
% end
% num_watermark_bits=MAX;

%watermark contains the bits to be inserted
%watermark=[0 0 0 1 0 0 1 1 0 0];
%watermark=[0 0 0 0 0 0 0 0 0 0];
%num_watermark_bits=10;
% num_watermark_bits=length(watermark);

%create random watermark string
num_watermark_bits=2300;
watermark=[];
i=1;
rand('state',sum(100*clock));
while(i<=num_watermark_bits)
  if(rand(1)<.5)
    rand_bit=0;
  else
    rand_bit=1;
  end
  watermark=[watermark rand_bit];
  i=i+1;
end
watermark_index=1;
temp_file=no_zero_pads;
temp_file_length=length(temp_file);
[temp_rows VLC_vector_length]=size(AC_VLC_vector);

binary_file=dec2bin(temp_file(1:temp_file_length-2)); % removes end of image marker
[binary_file_r binary_file_c]=size(binary_file);
binary_file=reshape(binary_file',1,binary_file_r*8);
binary_file_length=length(binary_file);
binary_file=binary_file(1:binary_file_length-trailing_ones); % removes 1's padding at end of image
binary_file_length=length(binary_file);

i=1;
while(i<=VLC_vector_length)
    if(watermark_index<=num_watermark_bits)
        if(AC_VLC_vector(1,i)==0)
            current_VLC=0;
        else
            current_VLC=AC_VLC_vector(1,i);
            j=1;
            while(j<=num_watermarkable)
                if(current_VLC==visual(j,6) & j~=4 & j~=5 & j~=6)
                    %%%%%%%%%%%%%%%%%%%%%%%% placing watermark %%%%%%%%%%%%%%%%%%%%%%%%%
                    if(watermark(watermark_index)==1)
                        VLC_start=AC_VLC_vector(3,i);
                        byte_location=AC_VLC_vector(2,i);
                        bit_to_watermark=visual(j,5)+VLC_start-1;
                        current_VLC_length=EHUFCO2(current_VLC,3);
                        change_bit=8*(byte_location-1)+bit_to_watermark;
                        temp_bit=binary_file(change_bit);
                        if(temp_bit=='1')
                            binary_file(change_bit)='0';
                        else
                            binary_file(change_bit)='1';
                        end
    end
end

92
end
watermark_index=watermark_index+1;
end
j=j+1;
end
end
i=i+1;
else
break
end
end
fprintf(1,'%d watermark bits placed \n',watermark_index-1);

even_bytes=floor(binary_file_length/8);
byte_remainder=8-(binary_file_length-even_bytes*8);
i=1;
ones_padding=[];
while(i<=byte_remainder)
    ones_padding=[ones_padding 49];
    i=i+1;
end
ones_padding=char(ones_padding);
EOI_dec=[255 217];
EOI_bin=dec2bin(EOI_dec);
EOI_lin=reshape(EOI_bin',1,16);
temp_binary_file=[binary_file ones_padding EOI_lin];
temp_binary_file=reshape(temp_binary_file,8,even_bytes+3);
byte_format=bin2dec(temp_binary_file');
temp_file=byte_format';
size=length(temp_file);
table_number=1;
[BITS,Huffval,bad_end_last_huffman,begin_AC_huffman]=jpeg_parse(table_number,temp_file,size);

% find beginning of AC Vijs

AC_huffman_length=temp_file(begin_AC_huffman+3);
Vij=begin_AC_huffman+21;

k=1;
changed=[];
while(k<=num_watermarkable)
    original_run=visual(k,1)
    original_size=visual(k,2)

    interpret_run=EHUFCO2(visual(k,9),1)
    interpret_size=EHUFCO2(visual(k,9),2)

    original_length=visual(k,4)+original_size
    interpret_vlc_length=EHUFCO2(visual(k,9),3)

    new_run=original_run;
    new_size=original_length-interpret_vlc_length
    replace_value=new_run*16+new_size;

    i=Vij;
    while(i<=AC_huffman_length+begin_AC_huffman+1)
        curr_Vij=jpeg_text(i);
        curr_run=floor(curr_Vij/16);
        curr_size=curr_Vij-curr_run*16;

        if(curr_run==interpret_run & curr_size==interpret_size & length(find(changed==i))==0)
            temp_file(i)=replace_value;

        i=i+1;
changed=[changed i];
break
end
i=i+1;
end
k=k+1;
end

%%% Create Huffman table to specify original VLCs %%%
%%% assumes number of original VLCs > 37 %%%

[number_VLCs_used temp_cols]=size(actual_huffman_table);

new_huffman_table_length=19+number_VLCs_used;

L_one=length(find(actual_huffman_table(:,3)==1));
L_two=length(find(actual_huffman_table(:,3)==2));
L_three=length(find(actual_huffman_table(:,3)==3));
L_four=length(find(actual_huffman_table(:,3)==4));
L_five=length(find(actual_huffman_table(:,3)==5));
L_six=length(find(actual_huffman_table(:,3)==6));
L_seven=length(find(actual_huffman_table(:,3)==7));
L_eight=length(find(actual_huffman_table(:,3)==8));
L_nine=length(find(actual_huffman_table(:,3)==9));
L_ten=length(find(actual_huffman_table(:,3)==10));
L_eleven=length(find(actual_huffman_table(:,3)==11));
L_twelve=length(find(actual_huffman_table(:,3)==12));
L_thirteen=length(find(actual_huffman_table(:,3)==13));
L_fourteen=length(find(actual_huffman_table(:,3)==14));
L_fifteen=length(find(actual_huffman_table(:,3)==15));
L_sixteen=length(find(actual_huffman_table(:,3)==16));

sort_actual_huffman_table=sortrows(actual_huffman_table,5);
i = number_VLCs_used;
Vij = [];
while(i >= 1)
    temp = sort_actual_huffman_table(i, 1) * 16 + sort_actual_huffman_table(i, 2);
    Vij = [Vij temp];
    i = i - 1;
end

new_huffman_table = [255 196 0 new_huffman_table_length 17 L_one L_two L_three L_four L_five L_six L_seven L_eight L_nine L_ten L_eleven L_twelve L_thirteen L_fourteen L_fifteen L_sixteen Vij];

%%% Insert Created Huffman table into watermarked image %%%

temp = [temp_file(1:end_last_huffman-1) new_huffman_table temp_file(end_last_huffman:size)];
temp_file = temp;

size = length(temp_file);

[with_zero_pads, num_zero_pads] = insert_zero_pads(temp_file, size);

watermark_file = fopen('LenaG_w.jpg', 'w');
fwrite(watermark_file, with_zero_pads, 'uint8');
fclose('all');
References

3 Provos, Niels Defending Against Statistical Steganalysis Center for Information Technology Integration, University of Michigan, provos@citi.umich.edu
7 J.Fredrich et al., “Attacking the OutGuess”, ACM 1-58113-000-0/00/0000, 2000.