Preliminary Observations on Restoration of Data from Lossy Compression

Brian G. Maddox[1]
Anthony Jones[2]

Open-File Report

CONTENTS

KEY WORDS

Data restoration lossy compression

ABSTRACT

Traditionally, data that is discarded during lossy compression has been assumed to be lost forever. During decompression, this loss of data leads to visible image distortions known as compression artifacts. Current methods of dealing with these distortions involve using image processing to smooth out the images, leading to further distortions. Mathematical and algorithmic techniques are being developed to actually restore some of the data that was discarded during the lossy compression. This restoration can be used to make the images more visually pleasing and better suited for image analysis tasks.

Any use of trade, product, or firm names is for descriptive purposes only and does not imply endorsement by the U.S. Government.

INTRODUCTION

Lossy image compression techniques have been existence for some time now. These compression methods discard information in order to shrink the size of files for storage or transmission to another system. Their basic operation uses mathematical techniques to approximate what humans are capable of seeing and then discard data that is deemed to be not noticeable to the eyes. Upon decompression, mathematical extrapolations are used to recreate the scene based on the surviving image information.

As some information is discarded, these extrapolations cannot perfectly recreate the original image. As a result, visible image distortions known as compression artifacts appear in the decompressed image. The distortions can appear as visible blocks, as in the case of traditional Joint Photographic Experts Group (JPEG) image compression, or small line segments in the case of wavelet-compressed images. These artifacts typically increase in number as the compression ratio is increased, leading to images that are visually unappealing or unusable for image analysis work.

Current methods of dealing with compression artifacts involve using image processing to smooth and/or blur the image to hide them. In many cases, this process starts with some form of a blurring (typically Gaussian) to soften the edges in the image and smooth out the artifacts. A denoising may also be applied to deal with any speckling that the compression might have caused. Finally, the blurred image can then be sharpened to reduce some of the blurriness that was caused in the initial step. Note that all of these steps actually distort the image even farther from the original uncompressed image, and only operate by trying to enhance the image's visual appearance.

Techniques and algorithms are currently being researched to attempt to actually restore some of the information that was discarded during the compression phase. Computational methods such as genetic algorithms can be used to perform calculations faster than available through a simple brute-force method. Algorithmic variations to routines such as Fourier Transformations can be used to create images that possibly match the uncompressed version as well as determining when to stop restoration attempts.

BACKGROUND

Lossy Compression

The are two methods of data compression available: lossy and lossless. Lossless compression reduces data sizes without discarding any information and is able to decompress the file into an exact copy of the original. This is the type of compression

used by PKWARE's pkzip and Winzip Computing's Winzip utilities. Lossy compression works by discarding information in order to compress data. During decompression, the algorithm works to estimate what information was discarded. This is how image compression such as JPEG and Lizardtech's Mr.Sid operate. As it discards information, lossy compression is usually able to achieve much higher levels of compression than lossless. This paper, and the work it describes, focus on lossy image compression methods.

Lossy compression was developed for two main reasons. The first reason is transmission between systems. The early days of technology featured network communications that were an order of magnitude slower than what can be found today. Images of even a few hundred kilobytes in size could require considerable time to transfer between computers. This became even more of a problem in the early days of the World Wide Web as sites became more graphically oriented. Compressing images to a much smaller size allowed them to be more easily transferred between computers. As the primary purpose was for visual display, lossy compression was well suited for this purpose. The second reason is for storage purposes. Permanent storage was once quite expensive and hard to acquire. Storage in the early days was measured in kilobytes instead of the gigabytes of today. The only way to store images was to compress them down so that they would fit the storage media available.

As technology has progressed, some lossy compression formats have become de-facto standards. JPEG File Interchange Format (JFIF), the file format encapsulating JPEG compression, has become the most popular lossy compression format for images. JPEG compression is based on the Discreet Cosine Transform (DCT) and is representative of the past “state of the art” in image compression. It has also been well studied and verified over the years as many applications depend on JPEG compression. These reasons are also why this project chose JPEG compression as one of the representative examples of image compression methods.

Discreet Cosine Transform

A simple description of the DCT is that it is a signal processing technique that is used to express a signal waveform as a weighted sum of cosines. When applied to images, it operates by converting the two-dimensional spatial representation of the image into frequency space. This conversion is performed by using the values of the pixels along the X and Y axes to represent frequencies of the same signal in two different dimensions (Escena). Once in frequency space, the transform can then express the numerical values of a group of pixels via a series of equations instead of discreet pixel values. The lossy compression occurs when the least significant pixels in the group are dropped during the conversion to equation representations. Upon decompression, the equations are converted back into groups of pixels. However, as the least significant pixels were dropped during the compression phase, they cannot be recovered during

decompression. Instead, the equations provide an estimation of the entire group of pixels from the original image. This is the place where fine detail is normally lost during JPEG compression.

More specifically, DCT compression is a specialized form of *transform coding*. Transform coding is "a technique in which the source output is decomposed, or transformed, into components that are then coded according to their individual characteristics" (Sayood 373). Transform coding uses some advanced concepts from statistics and calculus in order to convert data from one form to another. This method involves selecting a subset of a data sequence of ordered pairs, using a mathematical function to convert it to another representation, and then discarding data from the converted sequence. This process is then reversed to obtain the original subset of ordered pairs from the original data sequence.

Most ordered pairs in a sequence can be described by some function that can fit an average line or curve through the data points. This is known as *curve fitting* in the statistics world. It is generally used to take points from experimental data and fit a function through them that describes the behavior of a specific phenomena. This function allows someone to easily describe and predict the behavior of whatever it was they were observing. It is also useful in order to determine if there is some pattern to the data or if it is truly randomized. Graphically, this is known as plotting a *regression line* through the dataset.

This curve that is fit through the ordered pairs also has other properties that can be used to compress the sequence. The curve fit can be looked at as a function around which the ordered pairs tend to cluster. This allows a transformation to be applied to the sequence that converts it to another form. This new formatting of the sequence will tend to be created so that the ordered pairs will cluster around one axis of a Cartesian plane. If the ordered pairs were plotted on a graph, this can be visualized as rotating the dataset until it lines up with one axis or the other. The transform here would involve rotating the data by the angle formed between the average curve and an axis. Once in the new formatting, one of the elements of the ordered pair will be found to have more of a weight, sometimes called energy, than the other element. In this form, compression can then occur by dropping the element of least weight in the sequence, thereby reducing the number of elements that need to be encoded by fifty percent. In the case of clustering around an axis, one of the elements can be assumed to be at value zero. The remaining data is then encoded into another compressed form.

The data loss in transform encoding occurs when one of the elements of the ordered pair is dropped. The transformed sequence will have elements that cluster around an axis, but that does not mean that all elements will exactly fall on that axis. However, the element in the ordered pair with the highest weight is kept, so that when reversing the

transformation, the transformed ordered pair is close to the original value. This loss of data is usually deemed acceptable when dealing with audio and visual data as the human senses will not normally notice the small differences between the encoded and original data.

A real-world example is in order to explain the techniques of transform coding. It is important to understand transform coding as it is at the core of many modern compression methods. As an example, consider the following table of elements. The ordered pairs are artificially generated and tend to cluster around the function $y=x$. We will look at this example graphically to illustrate the concepts of transform coding.

0	0
5	3
10	11
15	13
20	22
25	24
30	34

Figure 1: Sample Data Points

The plot of these ordered pairs is given in Figure 2. The line $y=x$ is also plotted on the graph for comparison.

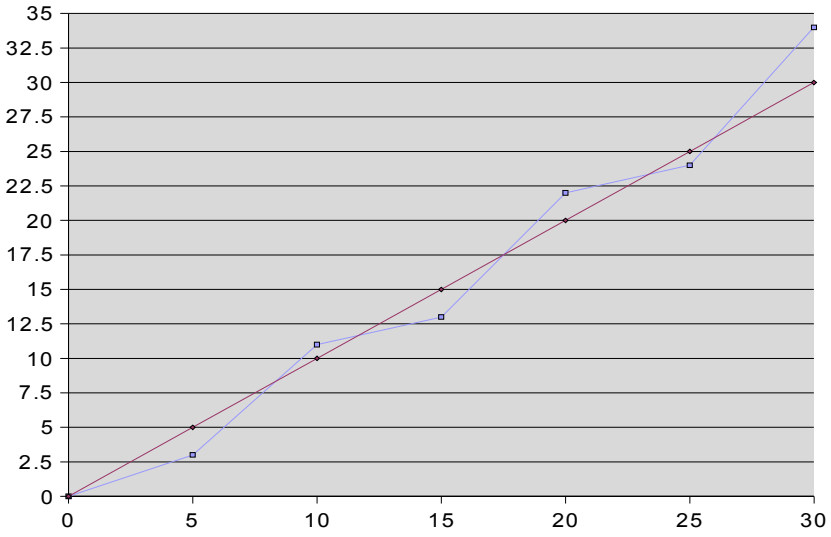


Figure 2: Graph of Sample Ordered Pairs

As this dataset clusters around the line $y=x$, we see that we can transform the dataset by rotating it 45° so that it lines up with the x-axis. The value of 45° can be derived either from geometry (the line $y=x$ forms the diagonal of a square, which then makes a right triangle) or computationally by taking $\arctan(a)$ based on the equation

$y = a \cdot x$. In order to do this, we must use matrix algebra. The ordered pairs can be considered to be a vector consisting of the *x* and *y* elements so that they take the form of $\begin{bmatrix} x \\ y \end{bmatrix}$. To rotate these values, we must multiply them by a transformation matrix based on the angle *θ* that takes the form of $\begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix}$. *Θ* is the angle that is made between the regression line and the *x-axis*. In this example, *θ* is 45°. Multiplying the matrices together then gives the new equation $\begin{bmatrix} x_{new} \\ y_{new} \end{bmatrix} = \begin{bmatrix} \cos(45) & \sin(45) \\ -\sin(45) & \cos(45) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$. The values in Figure 3 represent the ordered pairs in Figure 1 with the rotation applied and Figure 4 is the new values plotted out on a graph.

0.00	0.00
5.65	-1.44
14.85	0.64
19.79	-1.51
29.70	1.28
34.64	-0.87
45.27	2.62

Figure 3: Rotated Values

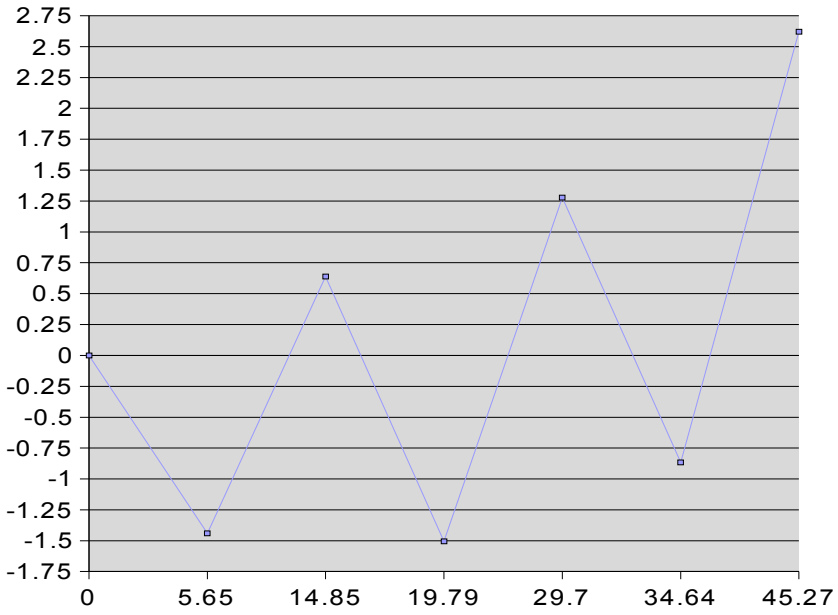


Figure 4: Plot of Rotated Values

We can see both numerically and graphically that the transformed ordered pairs vary by a small amount in the y-axis, from -1.75 to 2.75, and that most of the variation is contained in the x-axis which ranges from 0 to 45.27. Therefore, the transformed x-values contain most of the weight in each ordered pair. As the x-axis contains most of the energy, we can drop the y-axis values whenever we encode the sequence, thus

achieving a fifty-percent compression. The data loss in this case comes from the discarding of the second elements of each ordered pair in the sequence.

To reverse the compression, we would start off by reading in the encoded sequence that only contained the first values of each ordered pair. A sequence would be generated containing the actual first values of each transformed ordered pair and 0 substituted as the second value for every ordered pair. We then must rotate the sequence back so that we can undo the original transform. To do this, the generated sequence must be multiplied by the inverse of the original rotation matrix, which results in the following matrix: $\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$. Applying this reverse transformation to the sequence yields the reconstructed original sequence given in Figure 5. The deltas between the original sequence and the reconstructed sequence are given in Figure 6.

0.00	0.00
3.98	4.01
10.45	10.55
13.93	14.06
20.91	21.10
24.38	24.61
31.86	32.16

Figure 5: Reconstructed Sequence

0.00	0.00
1.02	-1.01
-0.45	0.45
1.07	-1.06
-0.91	0.90
0.62	-0.61
-1.86	1.84

Figure 6: Deltas between reconstructed and original

Figure 6 demonstrates how one can transform an original sequence of values, drop fifty-percent of them, and then later construct a sequence that is extremely close to the original uncompressed sequence. The deltas above can be seen to be very close to the original values, with an average numeric error of -0.07 for the first elements and 0.07 for the second elements. For pure numeric data such as financial transactions, this error is unacceptable. However, for image and audio applications, such a small loss would be barely noticeable to the observer and represent a good trade-off between storage sizes and quality.

This trade-off is the basis for the DCT compression used in JPEG compression. It gets its name as each element in the transform matrix that DCT uses is made up of functions of cosines. The DCT in a single dimension is given by the equation

$$G_f = \sqrt{\frac{2}{n}} C_f \sum p_t \cos\left[\frac{(2t+1) f \pi}{2n}\right]$$ where $C_f = \frac{1}{\sqrt{2}}$ for $f = 0$ and $C_f = 1$ for $f > 0$ (Salomon 290). DCT compression focuses most of the input energy into the first few coefficients of the transform. When looking at image data in frequency space, the transformation used by DCT is good at isolating various frequencies of the image data. The outputs from this transform consist of mostly zero or very small values with only a few large (thus requiring more storage space) numeric values. These first coefficients contain the low-frequency information of the image while the last portion of the coefficients contain the high-frequency information. DCT compression seeks to focus more attention on preserving the low-frequency information while discarding much of the higher-frequency data. The low-frequency information is the most noticeable portion of the image, thus requiring more care to maintain it as it represents the overall structure of the image. The higher-frequency data contain the fine detail of the image, much of which would not normally be perceptible to the human eye. Thus, the higher frequencies are not as important to restore properly.

The compression of the DCT is done by quantizing the transform coefficients. The smaller coefficients, usually the ones associated with high-frequency information, are quantized rather coarsely and sometimes dropped to zero. The larger coefficients that represent the low-frequency information are usually quantized carefully and rounded to the nearest integer. This quantization represents the last step of the compression, and can be reversed by taking the inverse of the DCT to decompress. However, as the transform coding example demonstrated, this data will not be identical to the original dataset.

DCT compression applications typically will break the data up into individual elements and then apply compression to them instead of the entire dataset at once. The reason for this is that there is a trade-off between the transform coefficients based on each data block. Small data blocks can result in a small number of small-valued coefficients. In this case, all of the data could be lost as it would be dealt with as consisting entirely of high-frequency information. On the other hand, large blocks can lead to a case where all of the information is treated as low-frequency information and the compressed data ends up unnecessarily large. For JPEG compression, the block size of eight by eight pixels is normally used.

A before and after example of DCT-based JPEG compression can be found in the next two figures. Note that the fine detail in the compressed image has been lost, which correlates to the compression focusing more energy into preserving the low-frequency information at the expense of the high-frequency (higher detail) information. However, the overall structure of the image (the low-frequency data) is still preserved quite well. In fact, when viewed at the proper size and zoom ratio, the absence of the high-frequency information is hardly noticeable.

Figure 7: Uncompressed Image

Figure 8: DCT Compressed Image

Wavelet Compression

Just as the DCT compression used in JPEG is representative of past state-of-the-art compression techniques, wavelet compression represents more modern leading-edge compression technologies. It is the compression behind the new JPEG2000 standard as well as other packages being sold for compression of large raster datasets. Wavelet compression is a direct descendant of the mathematics of the Fourier Transform. The Fourier Transform is a method of describing a continuous function in terms of sines and cosines. Wavelet transforms are able to deal with non-continuous functions, which makes them very useful for image compression.

In general, Wavelet compression works by first decomposing a signal into various components. The core idea is to analyze the signal by scale. The decomposition is accomplished through an array of *filter banks*, where each bank subsamples the original sequence into various subsequences. The subsequence can be, for example, a part of the original sequence where each sequence contains the even-numbered elements and the other the odd-numbered elements. Once the original sequence is decomposed, each subsequence is then down sampled based on a technique similar to that described above. The subsequence is then quantized to reduce the amount of data that it has to carry even further. The final step in the process is that actual encoding that creates the compressed information.

For a more in-depth explanation of wavelet compression, we must first briefly look at wavelet's ancestor, the *Fourier Transform*. Fourier's theorem, the basis for the

transform, states that “it is possible to form any one-dimensional function f(x) as a summation of a series of sine and cosine terms of increasing frequency. The Fourier transform of the function *f(x)* is written *F(u)* and describes the amount of each frequency term that must be added together to make *f(x).*” (Russ 285). The standard equation of the Fourier Transform is $F(u)=\int f(x)\mathrm{e}^{-2\pi iux}dx$. This use of the integral relies on *Euler's Formula*, which states that $\mathrm{e}^{-2\pi iux}=\cos(2\pi ux)-\sin(2\pi ux)$. The integration in this case serves to perform an averaging operation for the entire time interval of the signal. This can tell us what frequencies are present, but not when they occurred.

To determine when (and therefore where) a particular portion of a signal occurred, we must bracket the signal and perform a Fourier Transform on each individual piece. The problem with this method is that sine and cosine functions are continuous and smooth functions. When given a non-continuous or non-smooth function, they will have trouble approximating the original function. To make matters worse, there is some uncertainty as to what the original signal actually was due to approximation errors and lack of information as to where a particular frequency occurred.

In order to deal with some of these problems, wavelet compression performs a multi-resolution analysis on the image data in a signal processing context. One of the problems with the previously described bracketed Fourier Transform method (also known as the *short-term Fourier Transform*) is that the bracket size is fixed. This causes problems in that it can sometimes be too small or large, resulting in the wrong types of signal information being filtered out. Wavelets solve this problem by operating on the signal at multiple resolutions.

The function for a windowed Fourier Transform is given by the equation $F(\omega,\tau)=\int f(t)g*(t-\tau)$, where the function *g(t)* defines the size of the window (Sayood 457). In wavelet terms, this function is known as the *mother wavelet*. By varying the window sizes, we can create various basis functions to perform the multi-resolution analysis. In the example above, if we look at the first interval, where τ is zero, the first basis functions are *g(t), g(t)*$e^{j\omega 0\tau}$*, g(t)*$e^{j2\omega 0\tau}$*,* and so on. In this way, as the frequency of the signal increases, we decrease the size of the window so that we can increase the resolution of that signal.

The advantage of processing a signal at multiple resolutions is that temporal analysis can be performed on the signal by using a small high-frequency version of the mother wavelet, while a frequency analysis can be done by examining a larger low-frequency version. This multi-resolution analysis also works well for compression as it is able to handle sharp spikes or discontinuities in the input function. For example, if we look at

an image as a two-dimensional grid where the pixel value is a height, we can see how we can create a function that passes through the input points. A Fourier Transform will work, but not work well as a photographic scene does not always contain smooth graduations between color values. Wavelets, however, are better able to map an image in this case as the multi-resolution analysis can handle the sharp spikes between color values in an image.

Problems with Compression Artifacts

As both of the above-mentioned methods are a form of transform coding, they each suffer the accuracy problems of this encoding method. As demonstrated above, transform coding seeks to encode the more energetic portion of a sequence. However, even then the most energetic members are approximated, while the least-energetic are typically dropped and have a higher margin of error when restored. In terms of images, these approximations can lead to visible distortions in an image that are also known as *compression artifacts*. In DCT compression, these artifacts are manifested by "blockiness" in the image. Wavelet compression has artifacts that appear as small line segments scattered throughout the image. The following figures demonstrate these compression artifacts.

Figure 9: DCT Example: Uncompressed image

Figure 10: DCT Example: Compressed JPEG

Figure 11: Wavelet Example: Uncompressed image

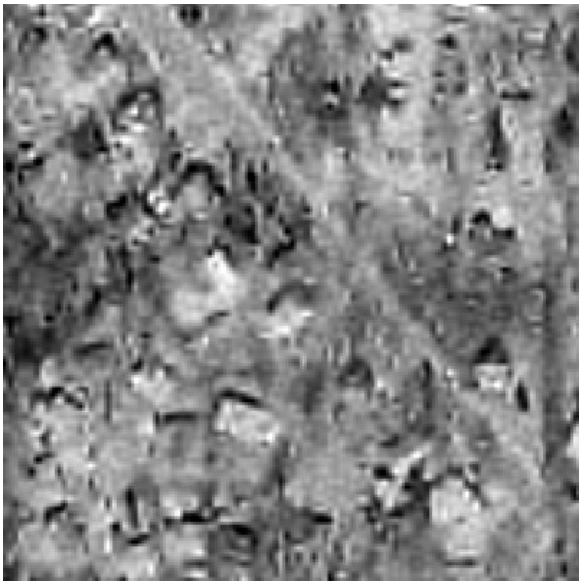


Figure 12: Wavelet Example: Compressed image

In the DCT case, one can see that not only do the block artifacts distort the objects in the image, but they also distort the colors in the image as well. The DCT compressed image was compressed by roughly a 10:1 compression ration. In the wavelet case, the compression artifacts distort the image by erasing certain features. The wavelet example is taken from data provided by the Missouri Spatial Data Information Service. While these may be considered extreme cases, historical data is also likely to be highly compressed as it would have had to fit on the storage media available in the past.

In addition to the visible image degradation, compression artifacts also hinder other analysis of the images. For example, feature extraction can be near impossible when performed on a highly compressed image. Edge detection, a component often used in feature extraction, becomes difficult as discreet edges can disappear in a compressed image. False edges introduced by compression artifacts can also make edge detection difficult. Change detection can also be very difficult as features can be distorted or completely lost after compression.

A New Theory of Compression Restoration

Once lossy compression is performed, the original data is for all intents and purposes lost in the compressed output image. If the original image is retained, one can simply go back to the original to get the uncompressed data. However, if the original uncompressed sample is lost, all that remains of the original data is the compressed output. This is the case with many geospatial data holdings, where the data had to be compressed in the past as storage technology was very limited.

Modern processing capabilities and advancements in computer science have provided much more computational power than previously available. Distributed processing has become common with the advent of Beowulf clusters (Beowulf.org). Processors are approaching a stage where a single processor can contain multiple execution cores. Computational techniques such as genetic algorithms provide quicker methods to solve very complex problems. All of these techniques are coming together to provide greater processing capabilities than previously imagined.

It is these advances that allow a new theory to be developed about addressing the problem of restoring data that was discarded during lossy compression. There is an old joke that 10,000 monkeys with 10,000 typewriters can eventually write *War and Peace*. In theory, there is a grain of truth in this joke. By randomly creating combinations of letters to form words, one in theory could eventually stumble across the combination of words that would make up a great novel. However, those 10,000 monkeys may all have to randomly type a few million years to come up with *War and Peace*.

The same idea holds true for images. Consider a given input image with a specific width, height, and photometric. One can start with a blank image of the same dimensions and photometric and perform a linear search of pixels to eventually reproduce the original image. In this method, one would have to try every possible value for a given pixel, and every possible combination of those values across the image. Eventually, such an exhaustive search would find the combination of pixel values to exactly match any given sample.

This works because any image is simply a combination of pixel values in a particular order. To humans, this combination produces a picture. A computer, however, sees an image simply as an array of pixel values. The array of pixel values is nothing more than a sequence of numbers with certain bounds imposed (i.e., RGB imagery has three components per pixel and each component can only have values of 0 to 255). This known bounds means that searching through each component value can eventually recreate the original sequence.

This idea can then be applied to compressed imagery. Compression operations tend to be repeatable when the same parameters are applied to the same image. We can then extend the idea of a linear search to say that given a compressed output and the compression parameters used to create that output, we can create test input images and perform a linear search to find the original uncompressed image. To do this, we must first create an input image with the same dimensions and photometric as the compressed output, compress our sample input, and then compare the generated compressed image to the compressed sample. As will be discussed later in this paper, this in practice is not quite as easy as the uncompressed case as we must deal with compression artifacts and the incredibly long run times associated not only with creating

the input images, but also with compressing them.

METHOD AND TESTING

Data Collection

Early on in the project, data collection was deemed to be a time consuming, yet important, activity. The images produced during various stages of testing needed to be stored so they could be analytically as well as subjectively examined.

The first step in data collection was storing the array data that produced a matching image. The medium used for this was a TIFF image file. It was also deemed necessary to output the compressed image produced by the match into a JPEG file. Comparing the input compressed seed image file to the match's JPEG will show just how close the array data for the match should be to the original uncompressed image.

After these initial steps, many more variables were added to the algorithms and needed to be logged as well. Both the modified brute force and the genetic algorithms output their own Comma Separated Values (.csv) file. A CSV file can be imported into most any spreadsheet program for easy post in-depth analysis. The CSV file contains all variables and inputs used by the algorithm, as well as data relevant to the matches found. This makes it very easy to see what settings were used, recreate test runs, and post analysis.

A differencing image program was created to compare two images and give statistics on how closely related the images were. The two images compared generally are the compressed seed image and the compressed match. The differencing program outputs such data as the maximum positive difference, the maximum negative difference, and the minimum difference between two corresponding pixels in the two images. It also gives the mean, standard deviation, and similarity percentage match of the two images being compared. These calculations are done for each color sample. For example, a RGB image would have calculations with statistics for the red, green, and blue channels separately. The differencing program also outputs all this data and more to a CSV file.

Brute-Force Method

The classic definition of the Brute-Force method is that it is a "*programming style that does not include any shortcuts to improve performance, but instead relies on sheer computing power to try all possibilities until the solution to a problem is found*" (webopedia.com). This method is computationally complex as there is no optimization of the algorithm used. However, it has the advantage that it always returns the correct

result, even though it may take large amounts of computational resources and long processing times.

The initial testing method was to closely follow the theory and perform a brute-force search of all of the possible combinations of input images. This consisted of examining the compressed sample, creating a sample data area that matched the dimensions and photometric of the compressed sample, and performing a linear search by looping each pixel through all possible values. The linear search ran by starting each color component of each pixel in the image at the minimum value, and searching through until the maximum pixel component values were reached. Each generated image was then compressed using the same parameters and compression format that was used on the compressed sample. The generated compressed output was then compared to see if it matched the compressed input. If not, the linear search continued.

It quickly became obvious that a true brute-force method was completely unworkable for this type of situation. A simple linear search of all possible pixel values can be done fairly quickly on modern hardware. However, each generated sample image had to be compressed, and the compression step is a non-trivial operation. In addition, each compressed test image had to be compared to the compressed sample image on a per-pixel basis. These additional steps added a large amount of processing time to the problem. In fact, a worst-case analysis estimated that for a 64x64 pixel image, the number of pixel combinations that would have to be tested exceeded the number of atoms in the known universe. The run-time of these combinations run on a 3.0 gigahertz Pentium 4 was estimated to take 1.2 million years. Clearly, a better solution was needed.

Modified Brute-Force Method

In order to solve the run-time problems of a true brute-force method, a new way of looking at generated images was introduced. If we look at a linear search of pixel values, there is some commonality between successive images as only a few pixels are different between those images. We can visualize this by taking each image generated by a linear search and stacking them one on top of each other from the first (empty) image generated to the final image where each pixel component is at its maximum value. If we pick any given image in this stack, then the images that surround our chosen image will be very similar to what we picked.

This method can then be extended to develop an improvement upon the brute-force algorithm. Compression algorithms may modify an image by introducing artifacts and some other modifications, but the compressed image still looks like the original. Compression will not take an image of a car and turn it into a picture of an elephant. So if we go back to the image stack idea, we can take an image out of the stack, compress

it, and roughly end up with an image that is near the original within that stack. In this sense, compression does not result in the same image, but one that is spatially similar to the original.

By taking this point of view, the brute-force method can be greatly improved. Instead of searching the entire stack, suppose that we take our compressed image and we want to find the original. We can find where in the stack the compressed image fits, and then based on the previous discussion, we know that the original image is spatially near our compressed sample. So, the exhaustive linear search suddenly becomes a small subset search of the entire image space. This is a huge improvement as it greatly reduces the number of images that must be tested, although it does still leave a large number left. This process begins by using the original compressed sample image and creating a *feedback loop*. The idea behind this is that the compressed input image primes the new loop. The subset is then bracketed by a lower bound where each pixel in the input image is subtracted by a set value and an upper bound where each pixel has a value added to it. Once we know this subset, we can begin searching it.

With the new algorithm and theory in place, testing then began by running the experimental algorithms on a 3.0 gigahertz Pentium 4 workstation. It was quickly obvious that while the problem size was magnitudes upon magnitudes smaller, it was still too massive to accomplish within any human lifetime. An example of this can be seen in a very small 8x8 pixel image in 8 bit gray scale. Testing every possible combination of images would require $256^{(8*8)}$ compressions. This method limits the 256 possibilities for a pixel color to a specified range. In example, suppose a range of -20 to +20 around each pixel is used. This would produce rather than 256^{64} combinations of images, but a much smaller $(40+1)^{64}$ image combinations. The amount of combinations is then about $1*10^{51}$ times smaller. Yet again, this number of combinations is still not feasible with today's fastest processors. One attempt to fix the sheer amount of combinations is to limit the range to very small numbers, such as -5 to +5. This would make only $(10+1)^{64}$ combinations of images. But even at a rate of 10,000 compressions a second, this would take $1.4*10^{55}$ years.

Finally it was realized that doing any type of linear or complete problem search is impossible, and the Monte Carlo approach was integrated in the algorithm. The definition of the Monte Carlo algorithm from the National Institute of Standards and Technology is, “A *randomized algorithm* that may produce incorrect results, but with bounded error probability. (Black)” This randomization was used to not search the entire stack of possible image combinations, but to randomly pick images out of that stack to test. The idea behind testing random images is that most of the images in the entire problem size are very similar. Using this method we can test much more variety, and hope for an image match much quicker. However since the Monte Carlo algorithm is not guaranteed to find the exact answer, a measurement is needed of how close an image guess is to the exact solution. This is explained in much more detail in the

comparison work section.

After including all these modifications, the modified brute force algorithm currently works as follows. First, all desired variables and parameters are decided upon before execution. The compressed image (seed image) that we want to find the original uncompressed image of is loaded. A random image is generated during each iteration based off the compressed image's original pixel values modified by a random number between the negative range and positive range. Once every pixel has had a random modifier added or subtracted from it, a new “guess” image of the original uncompressed image is generated. This guess image then needs to be tested to see if it is indeed what we think the original uncompressed image should look like. This is done by compressing the guess image using the same compression parameters used to create the seed image in the first place. The output from this is then compared to the seed image to determine if it is indeed what we think the uncompressed image should look like (explained in next section). If a match is determined, then the randomly generated image guess , the matching compressed image guess, and information about the match is outputted in file form. The algorithm then randomly generates another guess image to test for the next iteration. This continues until the desired number of iterations has been completed.

Comparison Work

Some of the most important work in this project is determining when a generated compressed image is “close” to the original compressed sample, when the processing is done, and when the image that best matches the original uncompressed image is found. It is simple for a human to look at a series of images and pick the best of the group. However, it is much more difficult to teach a computer how to look at images in the same way that people do. In addition, people may miss minute details that are important to restore. Several methods were used in order to try to find a good balance between fast and accurate comparison algorithms.

The first and simplest method used to test if generated compressed image is close to the original compressed sample image was a simple pixel value check. This worked by going through all values in both images and if there was a discrepancy, the generated image was deemed to not be a match and discarded. The ramifications of using an exact match criteria meant that the only exact matches found were those for highly compressed images. In general, the higher the image compression, the larger amount of different images there are that can create that same compressed image. As the compression lessens, the amount of matching images goes down quickly.

In order to get any image matches for medium to low compressions, a less strict criteria was needed. The idea of a set tolerance that could hold a value for the maximum

acceptable absolute error for a particular pixel came up. This tolerance would work much like a range around the original compressed sample image that the generated compressed image could have its pixel values fall within. Here is a simple example of how this works:

$$Original\ Compressed\ Sample\ Image\ of\ 3\ pixels = [22, 127, 240]$$

$$Generated\ Compressed\ Image\ of\ 3\ pixels = [24, 124, 247]$$

$$Tolerance\ for\ maximum\ absolute\ value\ error\ allowed = 3$$

The pixel values from the two images would be subtracted and the absolute value between them determined. For the first pixel this would be the value 2. Since $2 \leqslant 3$ the first pixel is acceptable. For the second pixel the value would be 5, and since $3 \leqslant 3$, this is also an acceptable pixel. For the third pixel the difference value is 7. Since $7 > 3$ this is not an acceptable pixel and the generated image would not be a match.

This method using a tolerance to allow more possible image matches with minimal error worked much better for getting any matches for non-highly compressed images. However, the flaw was still the same as before with an exact match. What if an image was a perfect match for every pixel value, except one that was over the tolerance? This almost perfect match, minus the single pixel, is discarded. This leads to the next improvement in image comparison criteria.

A number needed to be attached to the generated compressed image on how close overall the image was. A percentage match was then developed to say how close the generated compressed image matches the compressed image sample. This percentage is calculated by summing up the total absolute values of the error in pixels. This sum divided by the amount of pixels in the image gives the average pixel error number. The average pixel error can range from the minimum to the maximum allowable pixel value (or 0 to 255 in the case of standard 8 bit images). This average pixel error divided by the maximum minus the minimum allowable pixel value and then multiplied by 100 gives the percentage difference. Subtracting this number from 100 gives the percentage of how similar the two images are. The similarity percentage makes more logical sense when comparing two images closely related, and that is why it is used. The equations for this is as follows:

$$Image\ difference\ percentage = \frac{\sum |(samplepixel_i - generatedpixel_i)|}{(numberpixels * (maxpixelvalue - minpixelvalue))} * 100\%$$

$$Image\ similarity\ percentage = 100\% - Image\ difference\ percentage$$

Now with the image similarity percentage, many more images were able to be matched to the compressed sample image. A parameter was added before the algorithm runs for the required minimum similarity percentage to have a match. If the calculated image's similarity percentage is greater than this minimum, a match is found and outputted before the next iteration.

What was found when using both the tolerance and similarity percentage criteria, was that the tolerance had a negative impact on higher quality compressions, and a neutral impact on low quality compressions. This was because low quality compressed images already had many exact matches, and using the modified brute force algorithm, exact matches could already be found within a small amount of time. For higher quality compressed images there are far fewer image matches. Using a tolerance in addition to a minimum percentage match produced no matches until the tolerance was either raised absurdly high, or the minimum acceptable similarity percentage lowered. With too flexible a match criteria, the “matches” become much more grainy, and very incorrect when compared to the compressed sample image. The tolerance can be a beneficial criteria for very low to medium quality compressions, where there are too many exact image matches.

The tolerance is useless and sometimes harmful for any comparison criteria other than that “sweet spot” between low and medium quality compressions. For these reasons, the tolerance's use was discontinued as a match criteria. However, the total number of pixels violating a set tolerance is still logged in order to research a possible trend in the future.

Building upon the similarity percentage criteria, the idea of comparing the edge detections of two images came up. This method would work just like comparing the compressed sample image to the generated compressed image, only instead it would compare the outputs after running each image through an edge detection algorithm. As an example of what these edge detections look like can be seen in the following figures. The idea behind this comparison is that the compressed sample edge detection should be close to the original uncompressed image's edge detection. This would mean that if the generated compressed image's edge detection matched the compressed sample, the generated image could be declared a match. A similarity percentage was also used for this comparison method in the same way as before.

Figure 13: DCT Example: Uncompressed image

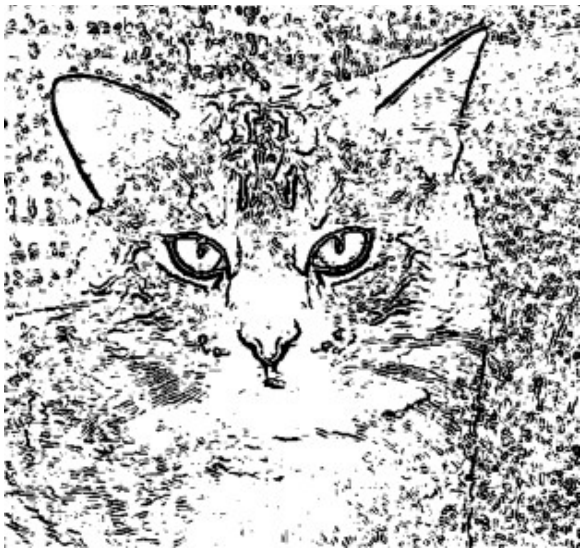


Figure 14: Sobel Edge Detection of Uncompressed

Figure 15: DCT Example: Compressed image

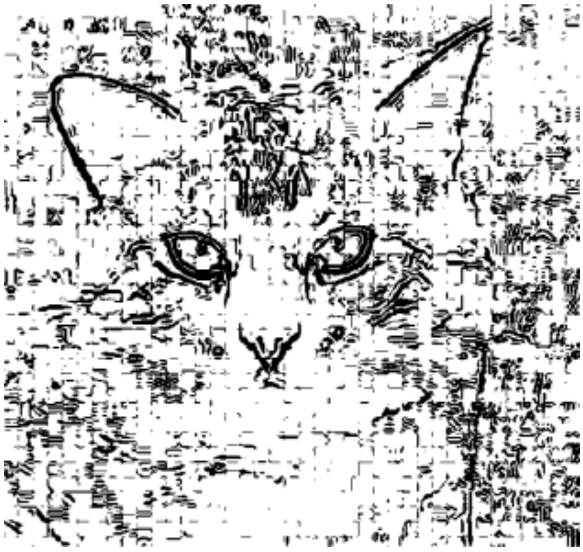


Figure 16: Sobel Edge Detection of Compressed

However, using the edge detection percentage match criteria did not help produce image matches that were not already found using the image similarity percentage. This is based on the fact that edge detection algorithms focus on using the pixel color values to produce an edge detection. So if an image's pixel values matches another image's pixel values (they are similar), the edge detections will also be similar proportionally. The added cost of processing that the edge detection algorithms added was also very costly. The number of iterations per second was one hundred times smaller when using these calculations. Therefore, the use of edge detection as a match criteria was discontinued.

Genetic Algorithms

Even with the modified brute-force method, the run-time of the algorithm still took an exceptionally long time. While only a subset of all possible images was being tested, there can still be a large number of input images that must be generated. When the overhead of the comparison and convergence criteria are applied, the processing runtime for a restoration algorithm is still very large. It became clear that another method was needed in order to find the best match to the original uncompressed image.

Genetic algorithms have evolved as more intelligent ways to find a best solution instead of relying on the brute-force method. They work by combining ideas taken from evolution. In the general sense, genetic algorithms start off by selecting members of the first generation, either at random from the data space or based on some other algorithm. Some processing then takes place that modifies the members in some way. Then, a fitness function is run that determines which data members will survive to become the next generation. Out of the remaining members, additional data elements are created by *breeding* them together. Other *mutations* are created by slightly modifying existing data members. Finally, some new data members are generated at random to finish out the next generation. These additions to the surviving members are used to ensure that the global best possible value is found instead of some localized best value. This continues for multiple generations, sometimes going for a set number of generations, and other times going until the program determines that the best possible value has been found.

The restoration method uses a genetic (also known as evolutionary) algorithm that simulates the principle of Natural Selection and Darwin's theories on Evolution. The program takes an original JPEG image as input and generates a certain number of slightly modified random copies for the first population of images. The purpose in this case being to determine what the uncompressed image of the original JPEG would look like. Mutation and crossover are then applied to each population resulting in a new batch of images for the next generation.

There are many variables in the evolutionary algorithm restoration program such as mutation rate, mutation range, and crossover rate. The mutation rate has possible values of 0% to 100%. Mutation rate refers to the amount of images in each trial that are mutated with random pixel changes. The mutation range can have values of generally 0 to 255 in a standard 8 bits per sample image. The mutation range is how much the program can change the number value of the pixel. For example, a given pixel has a color value of 125 in this hypothetical situation and the range is set at 5. This means that the program can set the transformed color value anywhere between

120 and 130 (depending on the random number selected). The crossover rate can range from 0% to 100%, and determines how many images in the population will randomly "breed" with another image. Crossover rate is a percentage of how images are combined with another image to produce a new "child" image for the next generation of images. The program mutates and does crossover randomly respective to their set percentages until 100 new images are filling spots for the next generation.

For each generation of images completed, a summary of information is outputted to the screen as well as the log file. This includes the best image match that generation, its fitness value (similarity percentage), and the average fitness for all the images in the generation. The log file is also sent additional information such as the worst image in the population for each generation, and the all the settings of the variables to produce the results. The final result of the algorithm is a complete log file, a TIFF of the best image, the JPEG of the best image that matched the compressed seed image, and the percentage similarity of the best image JPEG compared to the compressed seed image. All of the variables involved in the algorithm have an effect on how well the final image matches the compressed seed image.

Many other variables exist in an genetic algorithm such as this. Those variables are: quality, range, population size, number of generations, mutation number maximum, and elitism. Quality is simply the quality of the compression of the JPEG image in question. Photoshop and other programs allow you to change this from a default of 75 or so. Range is used for setting up the evolutionary algorithm's first population of images and the value bounds to modify them by. Population size is how many images are in each generation available for mutation and crossover. Number of generations is how many times the mutation and crossover will take place on the images to produce new slightly modified images. Mutation number maximum is a cap on how many randomly selected pixels can be modified per image. Elitism is a setting to keep the best image in the population from generation to generation unchanged with no mutation or crossover to mess it up.

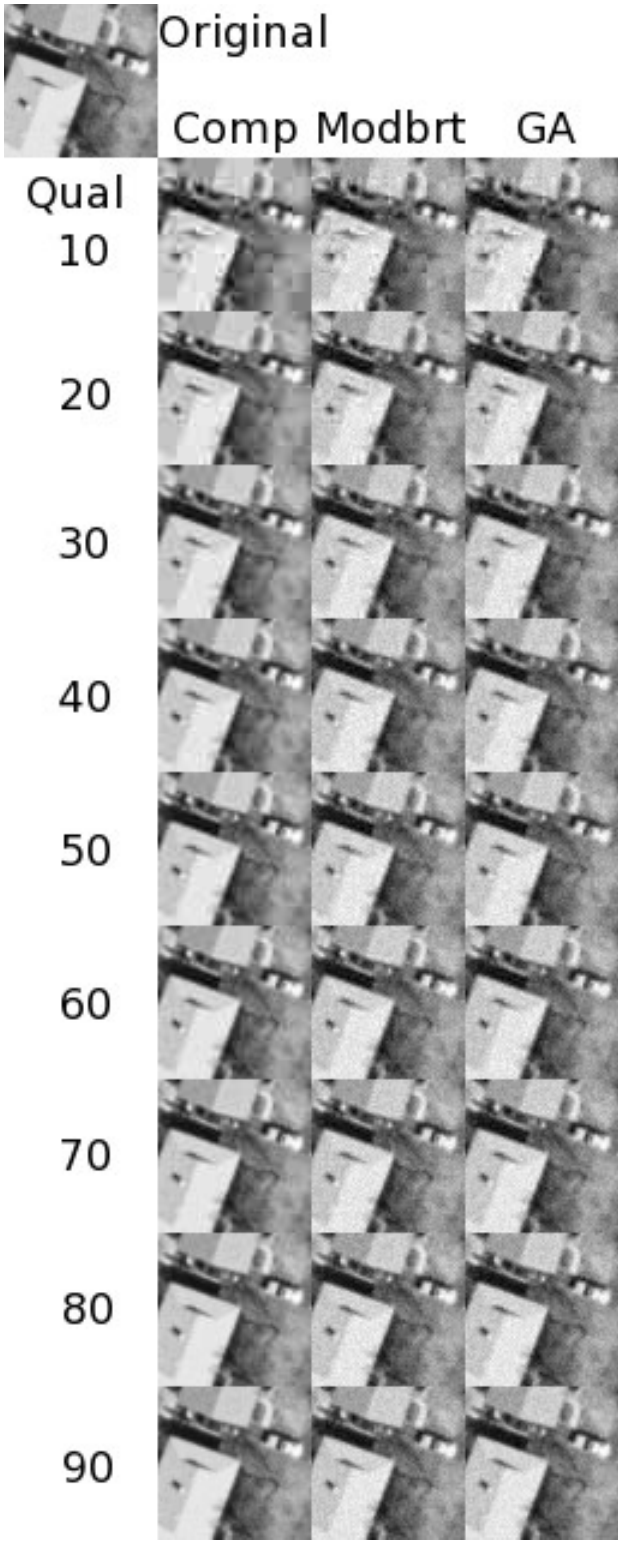
The results of using this new genetic algorithm to recover lost image detail was promising. In the vast majority of tests run, the genetic algorithm could acquire the same restoration quality as the modified genetic algorithm in about half the time. However, due to using the same comparison criteria for both of these algorithms the genetic algorithm did not obtain a higher quality image restoration, it just obtained an equal one faster.

Initial Restored Images

The following images illustrate some of the work to date and the success this project has had in restoring some of the data lost during compression.

Figure 17: Original uncompressed image sample.

This is a collection of images showing a direct comparison of the compressed image sample on the left column (starting image), the best match from the modbruterand algorithm in the center column, and the best match from the genetic algorithm on the right column. In every row, the JPEG quality increases.


Original
Comp
Modbrt
GA
Qual
10
20
30
40
50
60
70
80
90

Modified Brute Force Value Combinations Algorithm

This algorithm provides a fast and computationally simple way to generate the linear search images from a limited number of combinations. The basic idea behind this algorithm is that it mimics binary number counting in order to quickly generate the combinations. To accomplish this, the two-dimensional image must first be "unrolled" so that it becomes a one-dimensional array, where each row in the 2-D image is placed in order next to each other in the array. This array is then copied and becomes the working array. This copy is necessary as the original array is used to calculate the bounds for each pixel based on the range around each pixel value that must be searched. Each element is set to its minimum value based on the specified range.

For the main part of the algorithm, the idea is that it will loop through the values for each pixel and roll them over much like an odometer, although the values are not counted in a "standard order". The first array element is designated as the anchor element. This anchor drives the main part of the algorithm as it spins through the range of values specified. Once this element reaches its maximum value (original value + range up to the maximum possible value for that pixel), it then rolls over back to the minimum value. The roll over then triggers a walk through the array to find the next value to increment. If the next element in the array has not reached its maximum value, it is incremented and the main algorithm runs again with the anchor element spinning from minimum to maximal values. If the next element is at its maximum value, however, it is then set to its minimum and the rest of the array is walked until a value can be incremented.

To illustrate how this algorithm works, consider the following table. The array contains three values and was initialized to {100, 236, 58} with a range one (three surround values from value -1, value, and value + 1).

99 235 57	99 235 58	99 235 59
100 235 57	100 235 58	100 235 59
101 235 57	101 235 58	101 235 59
99 236 57	99 236 58	99 236 59
100 236 57	100 236 58	100 236 59
101 236 57	101 236 58	101 236 59
99 237 57	99 237 58	99 237 59
100 237 57	100 237 58	100 237 59
101 237 57	101 237 58	101 237 59

Figure 18: Sample MBF Counting output

The values in this table range from top to bottom, left to right. As can be seen, the algorithm does generate all possible combinations of values in the array based on the range and the initial values.

A slight problem with this method is that it can sometimes generate duplicate combinations if the input values are already at their boundary conditions for the specific data type. A solution to this would be to create another array to store the old combination and then test it against the newly generated combination. However, the added processing complexity of doing this would likely be greater than simply dealing with an occasional duplicated combination.

DISCUSSION

Restoration of discarded image data is a long and complex process, and is only now possible due to advances in computer processors and computer science. The techniques used to restore this information are computationally complex, and as stated previously, take some time to run.

True restoration, where an exact pixel-for-pixel match of the original images, is likely not currently possible. There are several reasons for this, the first being that computational resources do not currently exist to test every possible input image in a reasonable amount of time. A form of the brute-force method is necessary in order to test all possible images, yet a brute-force method applied to this problem would take far to long to be useful. Instead, the best means currently available is to use a genetic algorithm and converge towards an image that is better than the compressed input yet still not a perfect recreation of the original uncompressed image. This image would have some of the detail that was lost due to the compression method used, making the image more suitable for computational and visual display applications.

Many of the techniques used were selected because they are compression-neutral. This means that they are not tied to any particular compression format. The advantage of this is that once the techniques are developed, they can be applied to different lossy compression methods by plugging in the appropriate compression codec. The disadvantage to this, however, is that it does not take advantage of any signal processing techniques that could be applied to a specific compression method.

Research is ongoing in both the compression-neutral method and with the signal processing methods specific to a compression format. Doing both simultaneously will find general methods that apply to multiple formats as well as specific methods that can improve the quality of the restoration work. Working on both methods in parallel also helps in that there is some commonality between the two, and different approaches have created techniques that are useful to both research paths. For example, some of the convergence work on the generic compression restoration path may be applicable to help out the signal processing path determine when to stop processing.

The software libraries chosen for this project were the Independent JPEG Group's libjpeg for DCT JPEG compression and Jasper for JPEG-2000 wavelet compression. These libraries are Open Source, thus allowing the research to be done without paying a large licensing fee to a proprietary vendor. They also allow the source code to be examined so that a thorough understanding of the libraries and the codecs could be obtained.

The biggest problem that arose from this research is how to determine convergence towards the original uncompressed image. Lossy compression has several caveats that can make the problem very difficult. For low levels of compression, convergence approaches more of a one-to-one ratio of input image to compressed image. This makes it fairly easy to find a close match when a generated image is compressed and matched to the original compressed image. At higher compression levels, convergence skews away from a one-to-one ratio to more of a many-to-one ratio. The primary reason for this is the distortions introduced from the lossy compression can cause multiple images to generate similar compressed images. In this case it becomes difficult to decide which image to choose.

Some research was done to deal with the many-to-one convergence problem. Once the multiple matches to the input compressed were selected together, they were averaged together into a single estimate of the original image in an attempt to try to smooth out the minor differences between images. This will work to a slight degree and can produce an image that is more visually appealing in some cases. Averaging tends to have the effect of removing noisy areas in the restored image. However, it still may not be useful for analytical image processing on the restored image depending on what level of detail is needed. There is also the drawback that the more image matches you average, the closer the image reverts to the input compressed, and the artifacts return.

Figure 19: Original uncompressed sample image.

Figure 20: Sample image compressed with JPEG at 10 quality (small file size).

Figure 21: Example of an exact image match from modbruterand algorithm.

Figure 22: Average of 3 quality 10 image matches.

Convergence to a single solution is also difficult simply in determining when the processing is finished. In addition to the many-to-one issue, there is also the problem of not being able to computationally test the entire input space. This means that there is more of a likelihood of finding an image that is close to the original uncompressed image instead of the exact uncompressed image. So the problem then becomes one of determining when an image that is a “good enough” match to the original has been found.

Humans are good at this type of determination as our brains have developed to process visual stimuli. Computers, however, must be trained and even then it is difficult to develop an automated system with a high success rate. Using an exact pixel-by-pixel match can be harmful as it generates so few results that it misses images that might visually be valid yet suffer a minor difference from the original (such as having an overall histogram that differs from the compressed input image). On the other hand, widening the criteria can result in such a large number of matches that they contaminate the valid image results.

Some signal processing algorithms could be used here to help determine the matches by processing the images in frequency space. However, these methods can be highly computationally intensive. One of the problems with this work is finding ways to quickly scan enough images so that a best guess of the original uncompressed image can be found in a reasonable amount of time. Signal processing in this case would add to the runtime of the application and actually slow things down during processing. So far the above mentioned percentage method has shown a good balance between finding enough matches to find a better approximation of the original uncompressed image.

Genetic algorithms give us another way to determine when to stop processing. As previously mentioned, genetic algorithms converge to a solution more quickly than the other methods studied. Using this, we could stop processing and consider a solution to be found after a certain number of generations have been processed through the genetic algorithm.

One of the interesting theories that has come out of this work is that restoration will restore the detail lost during compression, but the restored image might not have the same histogram as the original uncompressed image. There are several reasons that can cause this to happen. After lossy compression is applied, the compressed image can exhibit a slightly different histogram than the original uncompressed image. The reason here is that some pixels are “smoothed out” so that an area that might originally have had pixels of different colors might be compressed into an area of a single average color. This then leads to the next case, where generated images from the original compressed sample can recreate the detail that was in the original image but comprised of different colors than in the original.

To understand this, one has to look at images as a series of edges. Each object in an image, be it a face, road, or what not, is made up of different edges that are created by the gradient between adjacent color areas. When the uncompressed image is lossy compressed, there is a reduction of colors as part of the compression process. This reduction of colors means that some of the extremely fine detail in the image might be

lost and that the remaining detail is comprised of edges with a slightly different color gradient than the original. The algorithms to generate test images from the original compressed image are then already starting with data whose histogram differs slightly from the original. Some of these images can then generate the same detail but with some color gradients that differ from both the compressed seed and the uncompressed original image. This is also one of the reasons that determining when an image match has been found since the histograms between test image and compressed seed may be slightly or even highly different. The previously-mentioned edge-detection experiments were done in an attempt to quickly scan an image to see if the edges matched up.

There are several ways to address this issue. The first is to computationally modify the histogram of the guessed image so that it matches the compressed sample. While the histogram would still differ slightly from the uncompressed original, matching it to the compressed sample would bring it much closer to the look of the original image. This method may have some unintended side-effects that could cause some of the detail to be lost again. Another method could be to only modify certain areas of the test image to match the similar area in the compressed sample. Here, the histogram of subsets of the image would be modified instead of the entire image.

The biggest gains in image quality thus far have been observed within the highly compressed images. These images have very noticeable artifacts and poor color variance. Also, as previously stated there are many more image matches that will compress exactly to the input compressed sample image. This guarantees an exact match solution has been found, and when compared with the uncompressed image sample is more visually appealing than the compressed sample. This has been the best accomplishment thus far with the ability to see detail has been restored despite a slight graininess to the match image.

However, the tables turn when the image is not highly compressed and is a good approximation to the original uncompressed image. This narrows down the possible exact image matches to such small a number our algorithms cannot compute enough image guesses to ever find it. It was estimated that for a default quality 75 JPEG image, the percentage of image matches out of 20^{4096} combinations that would compress to create the same JPEG was at the very least less than 0.0000000001%. This meant that out of all the image combinations possible, there was only a very small chance a randomized image would ever match. Either increased computing power to get through more iterations, or intense optimization of algorithms will be necessary to get exact image matches for less compressed images. For the algorithms using randomized image guesses, finding an image match to a lightly compressed image is similar to guessing six numbers and winning the lottery, only in this case those six numbers are enlarged to the number of pixels in the image.

The genetic algorithm may be very useful for the less compressed images. Since our algorithms rely on detecting a similarity percentage between images to determine when an image is a match, this could lend itself nicely for lower compressions that are much more sensitive to small mutations in pixel changes. Using this method the genetic algorithm could slowly creep towards a higher and higher quality match.

FUTURE WORK

There are still research topics that must and will be investigated in order to truly develop and optimize this process. In reality, some of these topics could continue on for years as they are dependent on developing technology and image science. However, they are interesting and important topics that can do much to advance the state of the art in this area of computer science.

Genetic Algorithm Refinements

One of the first future research topics is a refinement of the genetic algorithm method used thus far. There are several ways to perform genetic algorithms, and some additional methods must be researched to determine their effects upon performance. Also, more research must be done on the fitness function in determining what elements survive into the future generations. Some additional work is needed in determining when to stop processing, either when a certain fitness function is reached or after a fixed number of iterations.

An issue observed with running very large generation amounts (i.e. 1,000,000) was that after the first 500 to 10,000 generations, there is very little improvement if any in fitness. This is probably due to a genetic algorithm's tendency to get stuck at local maxima. Once a better comparison method is found, combined with some additions to avoid local maxima, the results should continue improving and come closer to the uncompressed image sample as the number of generations increases. This would allow the genetic algorithm to have more of a linear relationship to image quality gains rather than the logarithmic behavior currently.

“Large Scale Fourier or Wavelet Analysis”

A new way of using Fourier or Wavelet analysis will also be researched to determine its effects upon generating sample images and determining when processing is complete. Traditionally, Fourier and descendant mathematical analysis techniques start from a fixed point on an image, generally a corner, and process the entire image. Fourier analysis in this usage can produce a mathematical model of the image. It is currently theorized that performing this analysis on a larger scale can result in better

approximations of the image. Here, instead of starting at a single fixed position, each pixel of the image would be used as a starting position so that the analysis would be done many times and iterate through different patterns across the image. The idea is that doing this across the entire image will provide a series of mathematical equations that can be used to describe the image. Performing the analysis from one starting point can lead to situations where some pixels are missed during the construction of the transform. The missing pixels can be important, either being an image artifact or some detail in the image. Skipping over these pixels can effect the quality of the transform that is constructing for an image. By performing a large number of transforms, these equations can then be averaged together to provide an average mathematical model of the image. This average model might be a better approximation since it would be mathematically constructed of various models that together touch each pixel of the image.

Once this average is determined, the large scale analysis of the image could be used for some different purposes. The first purpose is in determining when processing is complete. The current theory is that as the image artifacts are removed, the standard deviation of the image equations will slowly decrease. In layman terms, this means that the image distortions would be decreasing as more of the original image is restored. Also, this technique could be used to provide additional inputs to the genetic algorithm as it could be run on the surviving members of a generation to create addition test images within the next generation. The downside of this method is that creating each transform is a compute-intensive process, and many transforms would be generated for each image.

Continuous Random Number Values

Upon preliminary analysis of trends in JPEG images, there seems to be a pattern in the differences between the lossy compressed pixel value, and the original image. The pattern roughly follows a cosine waveform as the Discrete Cosine Transform is used in JPEG. An example showing this behavior is below:

Image Type	1st Pixel	2nd Pixel	3rd Pixel	4th Pixel	5th Pixel	6th Pixel	7th Pixel	8th Pixel
Original	65	79	66	90	54	68	89	101
Compressed	70	83	68	90	52	65	84	95
Difference	+5	+4	+2	0	-2	-3	-5	-6

Observe the Difference row and you might see a slight sinusoidal pattern to the numbers. This can be a key to optimizing the random numbers used to modify pixels in the Modified Brute Force Method as well as the Genetic Algorithm Method. In order to

verify this occurs reliably, many images will need to be processed and tested to verify the pattern occurs in most images, and to what extent. The collected data on many different images would then be analyzed for patterns and similarities they all contain to develop an algorithm. This can then be used to strengthen the random numbers modifying pixels with a much more narrow range of values to randomly select, as opposed to a much larger range. The modification to the random numbers would then act as a limiter, preventing very unlikely random numbers to be tested, and hence reducing the problem size by a magnitude for each single pixel value it excludes.

For a 64x64 pixel image (4096 pixels total), using a range of 40 different values for a random number would create 40^{4096} distinct images to randomly guess. Using a limiter to prevent unlikely values within the range could reduce this to 10 different values. This would create a problem size of 10^{4096}, roughly 4^{4096} times smaller than the original problem space.

The problems that could arise from this optimization would be the possibility that the original image has large amounts of variance in its pixel values, and therefore the limiter may not allow image combinations with high variability that are in fact suitable matches. Edge detection can be used to minimize this problem. In most cases, the highest variability in images occurs at edges. These are the parts of the image with the highest contrast differences. Edge detection algorithms are trained to find those high contrast areas and create a resulting image outlining them. Edge detection can be run on the compressed JPEG image to create an outline image as a guideline for when not to use the limiter algorithm. This would allow high contrast areas to receive a greater range of pixel values that may match the original image more accurately. Another issue using this method is the specific sinusoidal compression required to find this sort of trend. Currently this could only be applied to mostly JPEG compression.

Edge Detection Post Processing

Edge detection might also be used as a post processing tool for image cleanup. Current methods for cleaning image artifacts involve slight blurs and sharpening as stated before. However, these techniques lose the small detail in an image and while making it more visually appealing, makes it less accurate. None of these techniques employ edge detection to limit the blur and sharpening algorithms. The use of edge detection can be used to determine edges that are important to the image and areas that are not. This can be used to create a “smart” blur and sharpen technique that would lose less detail while removing unpleasing artifacts.

There are of course cases that are too close to determine if they are an important contrast pixel, or just a bad artifact. In these cases, the pixel could be reverted back to the compressed seed image rather than risk the guessed or blurred value that could be

very inaccurate. This technique can be used in any of the methods, but especially lends itself to the Genetic Algorithm method. The technique can allow for some pixels to gradually alter for the better, while others in question would not drift to a worse value. This would eliminate the noise seen in the preliminary restored image samples, and could result in a much more visually appealing and accurate match.

Neural Network for Artifact Reduction

Neural networks are used to detect patterns within a given dataset. They are good at this because pathways through the network are created as data is run through it. At the end, patterns within the data can be identified by examining the pathways inside the network. The more a pattern appears, the more a pathway within the network is "etched in stone", so to speak. This also allows for modeling purposes, as a network can be trained on a certain mathematical model and then used for prediction purposes by seeing what the output would be given certain inputs.

Research into this area should be done to see if neural networks can be used to reduce artifacts in lossy compressed images. It might be possible to train a neural network on an image, either using the above-mentioned analysis or some new algorithmic method. Once this is done, it should be possible to use a network to detect where exactly the image distortions are within a given image. This can help lead to localized processing, where instead of processing the entire image, it is only processed where the artifact impacts the image. This could lead to a significant reduction in processing times since the entire image would not need to be analyzed.

Another possible area where neural networks could help is in creating guess images based on the compressed sample or generated test images. Here, neural networks could be combined with the large scale analysis technique to try to predict what the original image should look like. This image could be used as an input into the genetic algorithm and could possibly help speed that algorithm up, depending on how well the prediction is done.

Recompression Study

Testing was done to determine that no matter what program, what implementation, and what optimizations used in a JPEG compression, the input image still produced the same output compressed image. The only technicalities with this testing came up for programs or implementations lacking the options to set sub-sampling or integer/float precision. With all the same options however, the output image was pixel value equal (despite a few optimization differences in file sizes from different implementations).

This same testing was applied to JPEG2000, although not at the same scale or accuracy as JPEG due to a lack of quality open source implementations and programs with the ability to create JPEG2000 images.

More JPEG2000 Studies

Preliminary observations on JPEG2000 testing has shown an interesting difference from the old DCT JPEG compression. Since JPEG2000 is a newer and more computationally complex compression, it does take longer to do iterations than JPEG. However, JPEG2000's wavelet basis provides it a much closer approximation to the vast majority of pixels. This means the range required to randomly modify pixels by can be smaller, and hence have magnitudes less images to actually compute.

The randomized methods used so far also may not be suited for JPEG2000 due to its tendency to create grainy image matches, as well as JPEG2000's pickier algorithm's used when compressing pixel values. With JPEG, one could get away with changing a few pixels by small amounts and creating the same output compressed image. JPEG2000, however, is more sensitive to small changes, and somewhat unpredictable. Once this sensitivity is understood better, it could be exploited well with the genetic algorithm by making small changes and progressively increasing image quality.

CONCLUSION

Historically, restoring data that was discarded during lossy compression has been considered impossible. Modern technology and computer science, however, have recently evolved to a point where it is possible to attempt this problem. Restoration is still difficult and can require an enormous amount of computing power.

The best way to ensure a complete match is to use some form of a brute-force method. This technique tries every possible combination of inputs to try to match a desired output. When it comes to restoration of data, this is also the method that takes the most computing time and the runtime of such a program would likely last longer than our species. A compromise can be made that attempts to get a better image that exists between the quality of the uncompressed original image and the compressed image. This compromise is important in that, for many purposes, the exact original image is not necessary.

The other method to accomplish the compromise involves generating algorithms and techniques to create a general framework that will work with any type of lossy compression. In this case, a compression codec can be used with the framework and some form of restoration process can be run. This is useful in that it allows current

compression techniques to be restored, and it can work with any future lossy compression methods that may be developed.

Accomplishing these techniques requires advances in modern computer science. Genetic algorithms are an intelligent way of performing a brute-force task. However, they can accomplish the task in a much shorter time than what a brute-force method requires. Additionally, multiple-core processors are becoming available, and processors in general have become much faster to the point where most computers are actually overpowered for their tasks at hand. The advances in processing hardware and in the algorithms that can run on that hardware can make restoration possible where even just a few years ago it was for all intents and purposes impossible.

The ability to restore this lost data is useful for several purposes. It can take data that has been previously compressed and restore it to a better quality so that it is more usable for various purposes. One such purpose is for use in *The National Map*, where data that does not meet certain requirements due to it being highly compressed could be restored to a point where it can be included. Data in this case can go from barely acceptable for simple visual display to being usable through *The National Map* for cartographic and high end visual purposes.

Another area where this technology would be useful is in transferring extremely large datasets. Currently, the datasets that comprise the 133 Urban Areas Project range in size from seven to 240 gigabytes in size. At these file sizes, it could take a month of continual and uninterrupted downloading over a telephone modem. However, lossy compressing this data, downloading it, and then restoring it on the client computer can reduce this download time from weeks to hours. This can make data available to people who might otherwise not be able to access it. It can also allow data to be more easily stored by those who would choose to compress it and restore it later when they need it.

Several interesting discoveries have been made so far by this research. One is that while the detail of the data can be restored, this detail might not necessarily match the histogram of the original image. This is due to how lossy compression works by discarding some data and smoothing areas of an image out so that it compresses better. While the detail can be restored, the histogram must be matched to the compressed image sample to try to match the actual histogram.

REFERENCES

Beowulf.org: The Beowulf Cluster Site. <http://www.beowulf.org>. Sycld Software. 2004.

Black, Paul E, *Randomized Algorithm*, <http://www.nist.gov/dads/HTML/randomizedAlgo.html>. Dictionary of Algorithms and Data Structures. National Institute of Science and Technology. February 2005.

brute force – Webopedia.com. <http://itmanagement.webopedia.com/ *TERM/B/brute_force.html*>. Jupitermedia Corporation. 2004.

DCT. <http://www.escena.de/british.php?sitekey=tech_komm_transform>. ESCENA. December 2004.

Fan, Zhigang, and de Queiroz, Ricardo L., *Identification of Bitmap Compression History: JPEG Detection and Quantizer Estimation*, IEEE Transactions on Image Processing, Vol 12, No. 2, February 2003, pp. 230-235.

Russ, John C., 1995, *The Image Processing Handbook.* 2nd ed. CRC Press.

Salomon, David, 2004, *Data Compression The Complete Reference*. 3rd ed. Springer.

Sayood, Khalid, 2000, *Introduction to Data Compression*. 2nd ed. Morgan Kaufmann.