Lossless Single Pass Image Compression by Interpolation

Dr. Andrey Filippov Nathaniel Callens Jr.
Kelly Chang Bryce Hepner
Elphel, Inc.
1455 W. 2200 S. #205, Salt Lake City, Utah 84119 USA
{andrey,bryce}@elphel.com

June 23, 2022

Abstract

This algorithm operates by scanning through each pixel in a raster pattern, using already scanned pixels to decompress the next pixel's value. By saving the encoded error between the predicted pixel value and the actual value, we were able to losslessly get the compressed images to be less than 41% of their original size. This resulted in files that are approximately 34% smaller than their equivalent PNGs, and 35% smaller than the LZW compression method with TIFF.

1. Introduction

The idea is based off of how images are scanned in originally. Like a cathode-ray tube in a television, the algorithm goes line by line, reading/writing each pixel individually in a raster pattern.

Each pixel, as long as it is not on the top or side boundaries, will have 4 neighbors that have already been read into the machine. Those points can be interpolated and used to predict the next pixel's value. The goal is to encode the error between that value and the original value, save that, and use that to compress and decompress the image. Even though a possibly larger integer may need to be stored, it's more likely that the guess will be correct, or off by a small margin, making the distribution less uniform and better for compression.

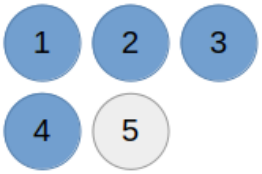

Figure 1: The other 4 pixels are used to find the value of the 5th.

2. Related Work

2.1 PNG

PNG is a compression lossless algorithm that operates using a single pass like ours [**?**]. The image is separated into several blocks of arbitrary size, which are then compressed using a combination of LZ77 and Huffman encoding [**?**]. LZ77 operates by finding patterns in the data and creating pointers to the original instance of that pattern. For example, if there are two identical blocks of just the color blue, the second one only has to make reference to the first. Instead of saving two full blocks, the second one just contains the location of the first, telling the decoder to use that block. Huffman encoding is then used to save these numbers, optimizing how the location data is stored. If one pattern is more frequent, the algorithm should optimize over this, producing an even smaller file[**?**]. The Huffman encoding portion is what separates LZ77 from "deflate", the algorithm summarized here, and the same one used in PNG.

Our algorithm has a similar use of Huffman encoding, but a completely different algorithm than LZ77. LZ77 seeks patterns between blocks while ours has no block structure and no explicit pattern functionality. Ours uses the equivalent block size of 1, and instead of encoding the data it encodes alternate information which is used to compress.

2.2 LZW

LZW operates differently by created a separate code table that maps every sequence to a code. Although this is used for an image, the original paper explains it through text examples which will be done here as well [**?**]. Instead of looking at each character individually, it looks at variable length string chains and compresses those. Passing through the items to be compressed, if a phrase has already been encountered, it saves the

reference to the original phrase along with the next character in sequence. In this way, the longer repeated phrases are automatically found and can be compressed to be smaller. This system also uses blocks like PNG in order to save patterns in the data, but instead by looking at the entire data as it moves along, PNG only operates on a short portion of the text [**?**].

Ours, similarly to PNG, only looks at a short portion of the data, which may have an advantage over LZW for images. Images generally do not have the same patterns that text does, so it may be advantageous to not use the entire corpus in compressing an image and instead only evaluate it based off of nearby objects. The blue parts of the sky will be next to other blue parts of the sky, and in the realm of thermal images, objects will probably be most similar to nearby ones in temperature due to how heat flows.

2.3 A Method to Save Some of the Interpolated Errors

No projects or papers are very similar to the ideas expressed in this paper, especially not for 16 bit thermal images. One paper that comes close is "Encoding-interleaved hierarchical interpolation for lossless image compression" [**?**]. This method seems to operate with a similar end goal, to save the interpolation, but operates on a different system, including how it interpolates. Instead of using neighboring pixels in a raster format, it uses vertical and horizontal ribbons, and a different way of interpolating. The ribbons alternate, going between a row that is just saved and one that is not saved but is later interpolated. In this way it is filling in the gaps of an already robust image and saving that finer detail. It should show an increase in speed but not in overall compression. This will not have the same benefit as ours as ours uses interpolation on almost the entire image, instead of just parts, optimizing over the larger amount of saved error values.

2.4 A Method of Interpolation by Clustering

The closest method is "Near-lossless image compression by relaxation-labelled prediction" [**?**] which has similarity with the general principles of the interpolation and encoding. The algorithm detailed in the paper uses a clustering algorithm of the nearby points to create the interpolation, saving the errors in order to retrieve the original later. This method is much more complex, not using a direct interpolation method but instead using a clustering algorithm to find the next point.

This could potentially have an advantage by using more points in the process, but the implementation becomes too complicated and may lose value. It also has a binning system based off of the mean square prediction error, but which bin it goes into can shift over the classification process adding to the complexity of the algorithm. The use of more points could be implemented into ours too, although it would not help the temporal complexity.

3. Background

The images that were used in the backing of this paper are all thermal images, with the values from the sensors ranging from 19197 to 25935. Total possible values can range from 0 to 32768. Most images had ranges of at most 4,096 between the smallest and the largest pixel values. The original purpose was for the development of an algorithm that works in raster format that can be used to load the images individually. This algorithm has support for 16 bit images, as well as 8 bits with some slight modification. For other uses of this algorithm, such as being used on multiple channel color images, only slight modifications would be necessary. The camera being used has 16 forward facing thermal sensors creating 16 similar thermal images every frame.

4. The Approach

To begin, the border values are encoded into the system. There are not many values here and the algorithm needs a place to start. Once the middle points are reached, the pixel to the left, top left, directly above, and top right have already been read in. Each of these values is given a point in the x-y plane, with the top left at (-1,1), top pixel at (0,1), top right pixel at (1,1), and the middle left pixel at (-1,0), giving the target (0,0). Using the formula for a plane in 3D ($ax + by + c = z$) we get the system of equations

$$-a + b + c = z_0$$

$$b + c = z_1$$

$$a + b + c = z_2$$

$$-a + c = z_3$$

.

These complete the form $Ax = b$ as

$$A = \begin{bmatrix} -1 & 1 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

$$b = \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix}$$

Due to there being 4 equations and 4 unknowns, this is unsolvable.

This can be corrected by making

$$A = \begin{bmatrix} 3 & 0 & -1 \\ 0 & 3 & 3 \\ 1 & -3 & -4 \end{bmatrix}$$

and

$$b = \begin{bmatrix} -z_0 + z_2 - z_3 \\ z_0 + z_1 + z_2 \\ -z_0 - z_1 - z_2 - z_3 \end{bmatrix}$$

.

The new matrix is full rank and can therefore be solved using `numpy.linalg.solve` [**?**]. The x that results corresponds to two values followed by the original c from the $ax + by + c = z$ form, which is the predicted pixel value.

Huffman encoding performs well on data with varying frequency [**?**], which makes saving the error numbers a good candidate for using it. Most pixels will be off by low numbers since many objects have close to uniform surface temperature or have an almost uniform temperature gradient.

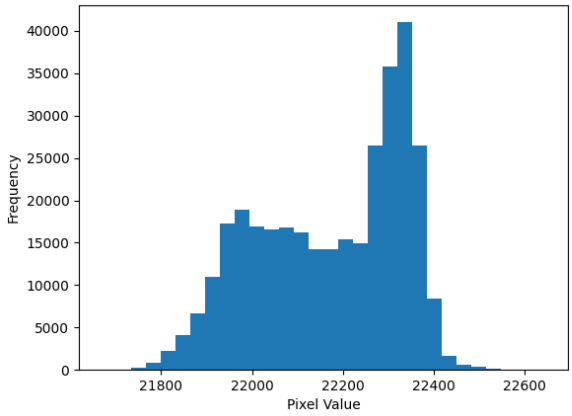

Figure 2: Encoding the Pixel Values

In order to control for objects in images that are known to have an unpredictable temperature (fail the cases before), a bin system is used. The residuals from `numpy.linalg.lstsq` [**?**] are used to determine the difference across the 4 known points, which is then used to place it in a category. This number is the difference between trying to fit a plane between 4 different points. If a plane is able to be drawn that contains all 4 points, it makes sense that the error will be much smaller than if the best fitted plane was not very close to any of the points. Something more certain in this case is likely to

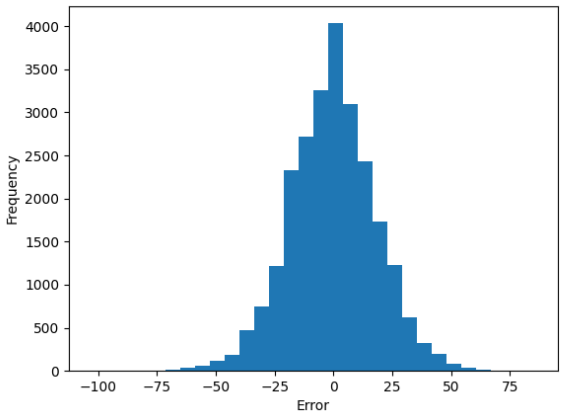

Figure 3: Encoding the Error Values

be more correct. 5 bins were used with splits chosen by evenly distributing the difference numbers into evenly sized bins. Many of the images had several different bin sizes ranging from 11 in the first category to a difference of 30 as the first category. An average number between all of them was chosen, since using the average versus specific bins had an effect on compression of less than half a percent.

5. Results

We attained an average compression ratio of 0.4057 on a set of 262 images, with compression ratios ranging from 0.3685 to 0.4979. Because the system as it stands runs off of a saved dictionary, it is better to think of the system as a cross between individual compression and a larger archive tool. This means that there are large changes in compression ratios depending on how many files are compressed, despite the ability to decompress files individually.

When the size of the saved dictionary was included, the compression ratio on the entire set only changed from 0.4043 to 0.4057. However, when tested on just the first image in the set, it went from 0.3981 to 0.7508. This is not a permanent issue, as changes to the system can be made to fix this. These are detailed in the discussion section below.

We are using it on a set of at least 16 images, so this does not affect us as much. When tested on a random set of 16 images, the ratio only changed from 0.3973 to 0.4193.

Compression Rates			
Original	LZW	PNG	Ours
100%	61.94%	61.21%	40.57%

The created file system together created files that are on average 33.7% smaller than PNG and 34.5%

smaller than LWZ compression on TIFF.

6. Discussion

The files produced through this method are much smaller than the others tested but at great computational costs. PNG compression is several orders of magnitude faster than the code that was used in this project. Using a compiled language instead of python will increase the speed but there are other improvements that could be made. Part of the problem with the speed was the `numpy.linalg.solve` [**?**] function, which is not the fastest way to solve the system. This method operates in $O(N^3)$ [**?**] for an $N \times N$ matrix, while more recent algorithms have placed it at $O(n^{2.37286})$ [**?**] Using an approximation could be helpful. Although it is potentially lossy, it would greatly improve computational complexity. The least squares method mentioned in this project also has the same shortcoming. It runs in $O(N^3)$ for a similar $N \times N$ matrix [**?**].

This compression suffers greatly when it is only used on individual images, which is not a problem for the project it was designed for. The camera that this compression was built for has 16 image sensors that work simultaneously. They work in 100 image increments and therefore create large packets that can be saved together, while still having the functionality of decompressing individually. This saves greatly on the memory that is required to view an image. It was therefore not seen necessary to create a different system to compress individual files as individual images are not created.

A potential workaround for this problem would be to code extraneous values into the image directly instead of adding them to the full dictionary. This has the downside of not being able to integrate perfectly with Huffman encoding. A leaf of the tree would have to be a trigger to not use Huffman encoding anymore and use an alternate system to read in the bits. We chose not to do this but it would be a simple operation for someone with a different use case.