Image Compression

Gustavo Bizarro
Departamento de Engenharia Informática
Universidade de Coimbra
Coimbra, Portugal
uc2018298933@student.uc.pt

Alex Pinheiro
Departamento de Engenharia Informática
Universidade de Coimbra
Coimbra, Portugal
uc2014227184@student.uc.pt

João Bonifácio
Departamento de Engenharia Informática
Universidade de Coimbra
Coimbra, Portugal
uc2018279685@student.uc.pt

***Abstract*—In this paper we are going to analyze the best possible lossless compression methods for four different monochromatic bmp images. For this, we are going to start by analyzing the state of the art on the subject. Followed by the analyzation of the four images in question, this will allow an informed decision on the algorithms to apply to achieve the best possible compression.**

Data compression is extremely important and will become even more so as data transfer and storage are invaluable in today's world.

Keywords—compression, redundancy, images

I. INTRODUCTION (*HEADING 1*)

Data compression has never been more important than before, in a world where an absurd amount of data is being created and transferred, especially with data storage and bandwidth always being limited, the need to compress data has never been higher. With this need in place, many techniques and algorithms have been created to help tackle the problem. With many algorithms existing already, there is no "best" one, the attributes that are usually used to categorize one are: Compression speed, decompression speed and compression ratio. In this paper, we will focus mostly on compression ratio since the purpose of this article is to achieve the best possible compression ratio on the four images given to us.

The idea behind lossless compression:

To achieve a lossless compression a compressed file must be able to be recovered identically to the decompressed file used. This is done by exploiting redundancy in the data, from representing long chains of symbols with smaller simpler ones to removing redundancy from your code, making it smaller. There are three main types of redundancy in images:

1.Coding redundancy – This type of redundancy can be removed by simply representing a larger code with an equivalent smaller one.

2.Interpixel redundancy – This redundancy arises from having neighboring pixels that are either identical or correlated. This type of redundancy can be removed by adding references to those pixels instead of fully storing the data that describes these pixels individually.

3. Psychovisual redundancy – This type of redundancy comes from data that the human eye cannot see. This type of redundancy is usually just removed from the data since we cannot perceive it.

II. BASIC LOSSLESS COMPRESSION TECHNIQUES

These techniques work because of redundancy in data, more complex algorithms usually use one or more of these techniques together. The use of a combination of these helps achieving a high compression ratio.

2.1 Run Length Encoding (RLE)

This is one of the simplest methods, it relies on heavily on code redundancy. It works by replacing a sequence of identical symbols with a shorter one that represents the same. The notation used is {v, r} where v represents the symbol and r the number of times the symbol appears in the sequence. An example for this would be the sequence "000000", this string would be swapped with a shorter code, in this case, {0,6}.

2.2 Huffman Encoding

This technique, which was created by D.A Huffman, is usually more efficient than RLE. It works by giving all symbols in the data a binary code and subsequently replacing the data symbols with their respective code. These codes are given to the symbols based on their frequency distribution meaning the most frequent symbols get the shortest code and the least frequent ones a bigger code.

2.3 Lempel-Ziv-Welch Encoding (LZW)

This method was developed by Abraham Lempel, Jacob Ziv and Terry Welch. This method exploits spatial redundancy in the data. It works by assigning a code to sequences of symbols, and then replacing all instances of that sequence with its code.

2.4 LZ-77

Abraham Lempel and Jacob Ziv invented this method in 1977. It has a similar concept as LZW, also exploiting spatial redundancy. But instead of assigning a code to sequences it uses a sliding window to replace those sequences with reference pointers to where it last happened, removing the redundancy.

2.2.1 Raising redundancy

Since the techniques used to compress rely on redundancy, one of the methods to help raise compression ratio would be to raise the redundancy on our data and then apply a compression technique. One method to do so is the following:

Burrow Wheeler Technique (BWT):

This technique was invented by Michael Burrows and David Wheeler in 1994. It works by transforming a string into a more compressible one, one with higher redundancy, it uses rotations and sorting to achieve this. Afterwards, the obtained string can be turned into the original one by simply putting it through the inverse transform.

Delta Filter:

This method replaces every value in an image with the result of the expression (value-previous_value). This helps raise redundancy because we will end with sequences of similar values. Example, the sequence [0 1 2 3] would be turned into [0 1 1 1].

III. Existing lossless image compression algorithms

Now we are going to look at existing compression algorithms, these types of algorithms are usually the essence of more complex ones. These normally also take use of basic compression techniques.

3.1 Spatial Domain Algorithms

These algorithms take advantage of spatial redundancy on the data.

1. Variable block Size Compression

This algorithm was invented by Ragnathan, it works by exploiting spatial redundancy, either global or local. It works by dividing the data in blocks. When coding a block, it scans all other blocks to find if an identical block exists to use as reference. It ends by coding the results with RLE.

2 FELICS

FELICS stands for Fast, Efficient, Lossless Image Compression System. It works by scanning a pixel's two closest neighbors to create an approximate probability distribution for the pixel's values. In essence, it combines error and prediction modeling. Afterwards, it picks the closest of a set of error models, all error models are assigned a simple prefix. Finally, it replaces the values with the prefix.

3 Lossy + Residual

This type of technique usually yields the best compression ratio. The idea behind it is to create a lossy image and a residual part which can be combined back into the original image. Both the lossy image and the residual part are really compressed compared to the original image, which is why this technique works well.

3.2 Context Based Algorithms

1. Context Based Adaptive Lossless Image Coder (CALIC)

This technique was made by Wu and Menon. It uses a Lossy + Residual approach. It starts by using a gradient based nonlinear prediction to obtain its lossy and residual image and ends with coding the results arithmetically.

2. Low Complexity Lossless Compression for Images (LOCO-I)

This technique combines simplicity with the compression potential of context models. By using a fixed context model which almost matches the capability of more complex universal techniques in capturing high order dependencies. After applying the model, it codes everything with a Golomb-type code, it chooses the best one to use adaptively. It also has an embedded alphabet extension to code low-entropy regions.

3.3 Transformed Based Algorithm

1. S+P Transformed Based Algorithm

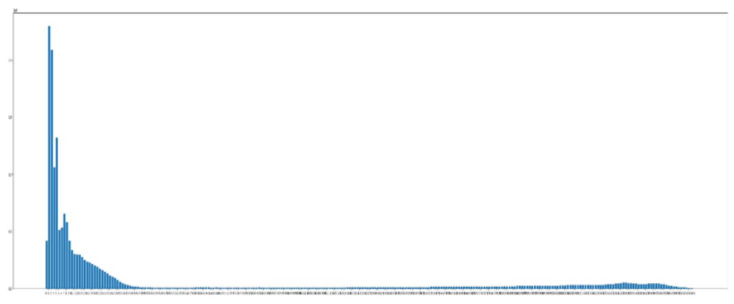
This technique was made by Said and Pearlmen. It works only using integer additions and bit shift operations on the data, the result is less space occupied. It works well for either lossy or lossless compression. It has good compression ratio.

IV. Analyzing our images

The four images that we are going to analyze are the following: Egg.bmp, Landscape.bmp, Pattern.bmp and Zebra.bmp. We are going to look at each picture's entropy and it's counting of symbols histogram.

Note: The lower is a file's entropy the higher its redundancy and compression potential

4.1 Egg.bmp

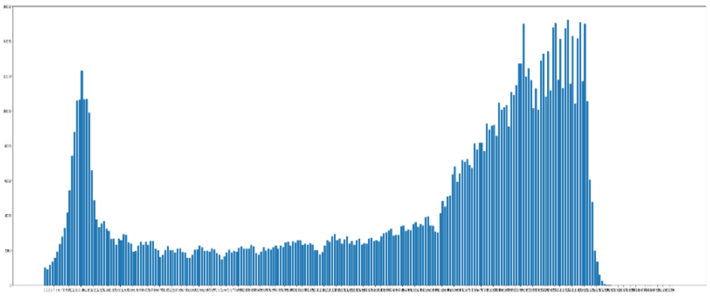


Entropy: 5.724225083014125

As we can see, the picture's entropy is high, but the diversity of symbols is low. To lower the entropy, we could apply a grouping of the alphabet symbols. With this data, we can take that a Huffman encoding would be useful here however, we should look at ways to raise the image's redundancy, like applying a delta filter on the data.

Visually, we can see that there is a possibility to rate the image 90º left or right to take advantage of long black lines of pixels. We can also conclude that Huffman encoding will take advantage of the fact that some intensities are much more predominant than others.

4.2 Landscape.bmp

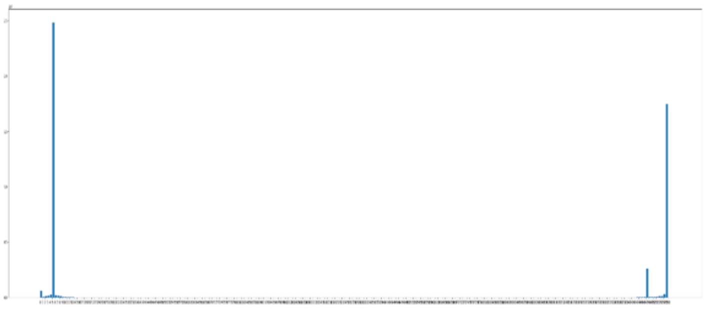


Entropy: 7.420548809289037

This Image has a high entropy, a high diversity of symbols and low redundancy. We will have to find ways to lower the entropy and raise the redundancy to be able to achieve a good compression ratio.

Looking at the image we can see the difference between the intensity of the pixels in this image is small, going from white at the top to black at the bottom. From this, we think we can raise the redundancy in this file by applying delta encoding to it.

4.3 Pattern.bmp

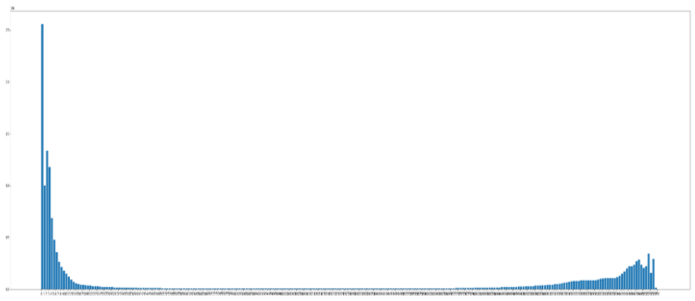


Entropy: 1.8292232406555082

As we can see, this picture has a low entropy, a low diversity of symbols and high redundancy, meaning that RLE could be efficient here since there will be many identical symbols together and also that Huffman Encoding could work since it'll be able to code every symbol with a relatively small code.

Looking visually at the image, we think we might be able to use LZ-77 or another similar technique to take advantage of repeating patterns on the image, exploiting spatial redundancy.

4.4 Zebra.bmp



Entropy: 5.8312399986899

This picture has high entropy, a low diversity of symbols and high redundancy. This tells us that RLE will not be that efficient here since there will be many less frequent symbols preventing long chains of identical symbols. Huffman encoding could work because as we can see in its histogram there are symbols that are very frequent in comparison to others. However, we must lower this picture's entropy to achieve a good compression ratio.

We also discussed the idea of rotating this image 90º left or right to further improve the compression potential of RLE. We think this could work because the rotated image would have several long white and black lines, this could help raise redundancy.

V. Our final two algorithms

After some testing we came up with two final algorithms for our images. The first was used exclusively on the landscape image since it was the only one which this algorithm gave promising results, the algorithm is:

Apply a delta filter to the data -> Apply burrow Wheeler transform on the data -> Encode the result with RLE -> Encode the final data with Huffman encoding

This algorithm gave us a final size of 4 402 KB on landscape.bmp (including the Huffman code table), down from its original 10 400 KB.

The algorithm is tailored specifically for this image and probably wouldn't work for the others. This is because the delta filter only worked on landscape (as expected), for other images it didn't work because it made the redundancy go a lot down, which would make them have a size larger than the original image.

Some tweaks worth mentioning that aren't referred above are that we had to turn the data from integers to the corresponding ASCII symbols, we had to sum 67 to all values before turning them into ASCII symbols because we could only work with positive numbers, we had to break the data into blocks since the BWT (Burrow Wheeler Transform) has a exponential complexity and finally we had to replace certain of the ASCII symbols to other unique symbols since the BWT algorithm we had used two special characters to mark the beginning and the end of each block. Some of these tweaks make this algorithm not viable for general purpose use and makes it specific to this image.

The second algorithm is:

Apply burrow Wheeler transform on the data -> Encode the result with RLE -> Encode the final data with Huffman encoding

The only difference to this one is that we decided not to apply a delta filter to the image since it wasn't suitable for the rest of the images. The results we're as follows:
Egg.bmp -> 9 610 KB from 16 900 KB
Pattern.bmp -> 2 892 KB from 45 700 KB
Zebra.bmp -> 9 845 KB from 15 900 KB

VI. Testing

We tested several combinations of techniques, but none gave us better results than the ones above, however, the following sizes I will be mentioning won't be to be trusted since we suspect there are faults in our implementation of the code. Some of these combinations are as follows (tested on "egg.bmp"):

RLE -> Arithmetic encoding
This yielded us a final encoded size of 12 300 KB.

RLE -> BWT -> Arithmetic encoding
We stopped the process once the size of the encoded file was getting to upwards of 40 MB. This should be a big pointer to why some values shouldn't be trusted like the ones we have for our main algorithms.

BWT -> RLE v2 -> Huffman encoding
This gave us a final encoded size of 12 112 KB. The reason we call this RLE a different version is because we decided to separate the count and the symbol instead of grouping them as one single symbol, this is important for the Huffman part of the algorithm.

Worth mentioning:
We spent some time trying to implement an LZ77 algorithm for the image pattern since we thought it might be especially good on it. But we eventually found out it is not an easy one to implement and involved some techniques we didn't know how to use. Even forcing us to use two separate files for the encoder and so on and so forth, it felt like we we're forcefully trying to make it work even thought it was clearly not made for this and that it was hard to mold it to our needs, we eventually scrapped the idea but not without (un)successfully having it run once, it yielded us an encoded size of 49 700 KB.

VII. Our biggest mistake

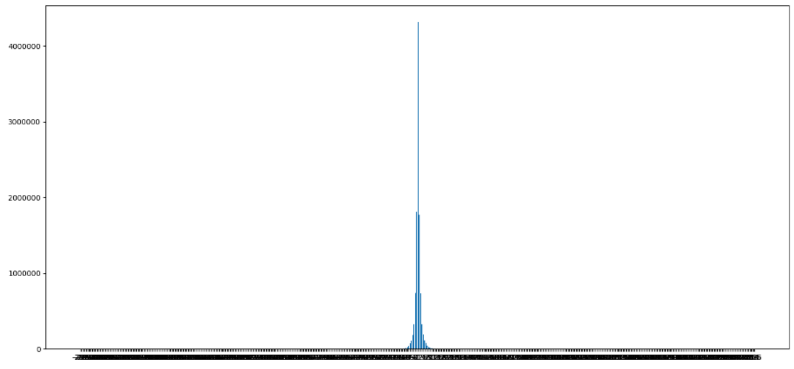
Our biggest mistake was that we jumped straight to trying to implement final algorithms from ideas we came up with during the state of the art instead of testing individual techniques on each image, this made the quality of this paper go down because we lost most of our time. But after some feedback we decided to take some conclusions from simpler tests, unfortunately, by this point our final algorithms have been set and we had no time to improve them(*).

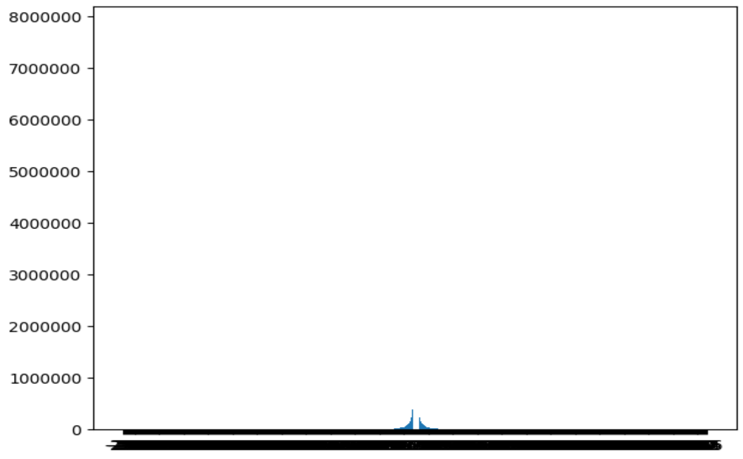
(* We in fact had time to make better ones as you'll see)

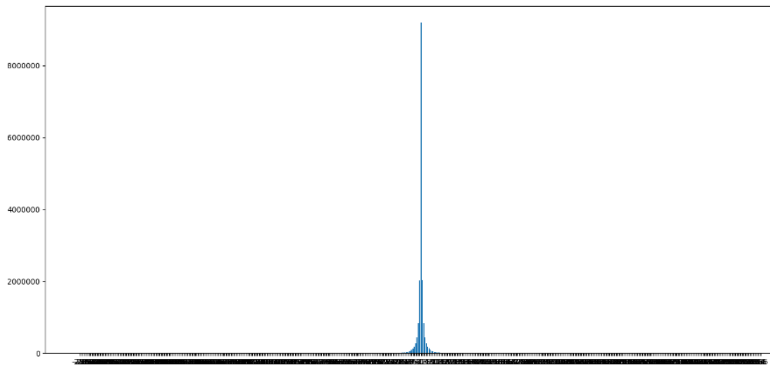
VIII. A better look at delta filter

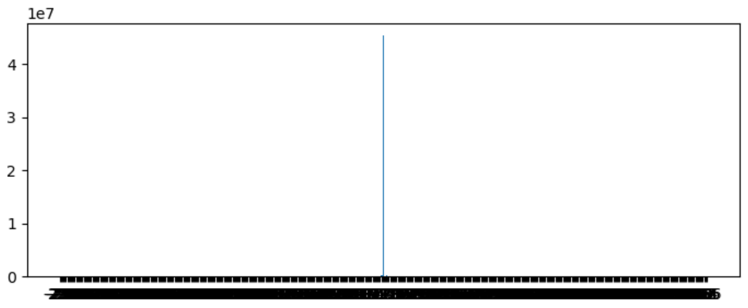
In this chapter, we take a better look at what the delta filter does to the images. This is what happened to the data:
Egg.bmp:
Entropy went from 5.72422 to 2.70204
Landscape.bmp:
Entropy went from 7.42054 to 2.82420
Pattern.bmp:
Entropy went from 1.82922 to 0.61024
Zebra.bmp:
Entropy went from 5.83123 to 3.22314

Unlike what we expected, all the entropies went down a significant amount, meaning this technique would have been good for all four images if followed by a technique that takes advantage of this reduction, techniques like









These histograms represent the statistical distribution of each symbol after the delta filter. And as we can see, they all follow a similar distribution. This type of distribution can be taken advantage of with algorithms that exploit statistical distributions, like RLE.

This test definitely showed us we were taking the wrong approach, the algorithm should have been similar for all four images(*), namely similar to the first one I mentioned in the V chapter.

*It maybe wouldn't be so efficient for pattern.bmp since it has long lines of identical intensities, making RLE not so efficient. In other words, [5 5 5 5] would become [5 0 5 0] which is worse if thinking of applying RLE (this is theoretical and not founded in tests yet).

IX. Run length encoding, Burrow wheeler technique and delta filter

To better understand the effects these techniques on the images given to us we decided to test them all, separately and together to find out which gave us a better result.

Size in bytes Entropy(grouping count and symbol) Entropy(separate count and symbol)	Original	Only RLE	RLE with delta filter	RLE with BWT	RLE with BWT and delta filter
Egg	17738362	17219478 8.14904 5.16971	18892554 4.68825 3.40308	19936189 7.87490 4.99742	19437368 4.75470 3.44699
Landscape	11005344	13406309 8.94839 5.48363	14381795 4.38343 3.24616	15860129 8.68770 5.35242	15272197 4.36495 3.23590
Pattern	48004864	5727503 6.89883 4.61669	6584224 6.49182 4.40803	7120762 6.93031 4.60702	7372987 6.57825 4.44314
Zebra	16737132	17992890 7.98001 5.09190	19537902 5.26191 3.70050	21102212 7.63412 4.87798	20993522 5.14620 3.65577

When looking at the results of BWT we have to keep in mind it's adding 100000 symbols to the end result since it has to add a start and end symbol to every block of data it processes, when using the transform we broke the data in 50000 blocks since it has an exponential complexity.

To also have a better view of what grouping the symbols mean we also made the following table showing the total number of symbols to encode when grouping and not grouping.

Number of Symbols to encode	Only RLE		RLE with delta filter		RLE with BWT		RLE with BWT and delta filter	
	G	Not G	G	Not G	G	Not G	G	Not G
Egg	8542792	17085584	9385504	18771008	9745323	19490646	9643083	19286166
Landscape	6685459	13370918	7178012	14356024	7749574	15499148	7619943	15239886
Pattern	2641113	5282226	3070597	6141194	3147217	6294434	3472518	6945036
Zebra	8930683	17861366	9712195	19424390	10357773	20715546	10441729	20883458

Firstly, the entropy when grouping symbols is always higher than not grouping, however, we must keep in mind the total size decreases by about half. So, theoretically, if the entropy does not increase by at least double, it might be worth it to group the symbols.

The results we got when using BWT we're almost always worse than not using it, this is because of the added symbols it adds. To fix this, we would need to divide the data in less but much bigger blocks, however, we would need a more efficient algorithm to achieve this.

Let's look at these results individually.

Egg:

The method that yielded the best results was RLE with BWT. This technique gave us the lowest entropy and size, also, grouping the symbols would be the better choice since the entropy wouldn't increase by a significant amount.

However, RLE didn't seem that efficient in this image since the size in bytes actually increased.

Landscape:

From these results we can see that using RLE with BWT and delta filter would have been the best choice since it gave us the lowest entropy, however, since the blocks weren't big enough the entropy difference will not outweigh the added symbols. In this case, grouping the symbols would probably be the better choice since the entropy increase would not outweigh the smaller number of symbols to encode.

However, using just the delta filter followed by Huffman encoding would probably yield similar results to including RLE (with grouping) since the entropy increased by about double its original one (with a delta filter) and actually raised it's size in bytes by a bit.

Pattern:

The results for this image we're the least conclusive, if we're trying to compare the use of methods in conjunction with each other, since they were close and with some exceling at some things by a little bit. Only using RLE gave us the smallest number of symbols to encode but its entropy is slightly higher than all other methods. Using RLE in conjunction with a delta filter gave us the lowest entropy, however, this is by a small margin and the size increased by close to one million bytes.

BWT could maybe have been useful here, however, the small blocks made its impact seem small and negligible.

However, we can take that RLE in general would be good for this image since the its size went down by a very significant amount.

Zebra:

Firstly, we can see that using a delta filter gave us the smallest entropy compared to the other methods, however, the total size in bytes actually increased, but if we look at the grouping of symbols we can see that the entropy is still lower than the original and the number of symbols to encode would be around half.

X. Encoding the images with diferent techniques

In this chapter we will encode the images with different techniques to find out which algorithm gives the best result for each image. Let's look at the compressed size of the resulting files.

Egg:

Only Huffman: 12 440 KB
RLE with Huffman (not grouped): 10 939 KB
RLE with Huffman (grouped): 8 595 KB
Delta with Huffman: 5 896 KB
Delta followed by RLE with Huffman (not grouped): 7 927 KB
*Delta followed by RLE with Huffman (grouped): 5 416 KB

Landscape:

Only Huffman: 10 017 KB
RLE with Huffman (not grouped): 9 064 KB
RLE with Huffman (grouped): 7 424 KB
*Delta with Huffman: 3 857 KB
Delta followed by RLE with Huffman (not grouped): 5 791 KB
Delta followed by RLE with Huffman (grouped): 3 866 KB

Pattern:

Only Huffman: 11 301 KB
RLE with Huffman (not grouped): 3 058 KB
*RLE with Huffman (grouped): 2 361 KB
Delta with Huffman: 7 661 KB
Delta followed by RLE with Huffman (not grouped): 3 390 KB
Delta followed by RLE with Huffman (grouped): 2 504 KB

Zebra:

Only Huffman: 11 989 KB
RLE with Huffman (not grouped): 11 258 KB
RLE with Huffman (grouped): 8 786 KB
Delta with Huffman: 6 640 KB
Delta followed by RLE with Huffman (not grouped): 8 934 KB
*Delta followed by RLE with Huffman (grouped): 6 289 KB

The ones marked with a '*' are the ones that resulted in the lowest encoded size and are the ones we will use to create the final algorithm for the corresponding image.

XI. Our algorithms vs PNG

How did our final algorithms stand versus the modern-day PNG algorithm? Here are the ratios:

Egg:
Original: 17,744,598 bytes
PNG: 4,632,955 bytes -> 3.83 compression ratio
Our: 5,545,939 bytes -> 3.20 compression ratio

We compressed 19.5% less

Landscape:
Original: 11,006,422 bytes
PNG: 2,283,344 bytes ->4.82
Our: 3,948,848 bytes -> 2.79

We compressed 72.94% less

Pattern:
Original: 48,005,942 bytes
PNG: 2,283,344 bytes -> 21.02
Our: 2,417,022 bytes -> 19.86

We compressed 5.8% less
Compared to PNG this was the closest we came to one of them, this is also the one which we achieved the highest compression ratio.

Zebra:
Original: 16,738,210 bytes
PNG: 5,470,858 bytes -> 3.06
Our: 6,439,837 bytes -> 2.60

We compressed 17.7% less

XII. Conclusions

From VI chapter onward we started taking a different and much more effective approach, this paper could be even better if we started with this method from the beginning, we could have more and more varied tests and so on and so forth, however, due to this we can make comparisons to what we once though was true, this is, comparing to what we once though would have been the better algorithm for the images.

The major comparisons being:

-We were sure RLE would have given significant results in the image landscape in conjunction with the delta filter, when in fact, it was actually worse, even if by a small margin.

-We thought the delta filter would have only been good for the landscape image, since visually, it looked like the only image in which its pixels had a small difference from its neighboring ones. But in fact, it had good results in all images, entropy wise.

This brings us to the end of our paper. Image compression is and will only become more and more important in these days of immense data transmission.