A Comparative Analysis of Two-Dimensional Run Length Encoding for Image Compression

Submitted **April 2024**, in partial fulfilment of the conditions for the award of the degree BSc Computer Science.

Student ID: 20334929

School of Computer Science

University of Nottingham

I hereby declare that this dissertation is all my own work, except as indicated in the text:

Signature: DC

Date: 19/04/2024

I hereby declare that I have all necessary rights and consents to publicly distribute this dissertation via the University of Nottingham's e-dissertation archive.

Acknowledgements

I would like to thank my supervisor Dr. Jason Atkin, who not only encouraged me in researching this topic but whose trust and assistance gave me confidence throughout the project.

Abstract

The increasing number of digital images is linked to a growing demand for efficient methods of storage. Solving this without relying on the unsustainable creation of storage media requires further research into alternative image compression techniques. This project creates an application that exploits two-dimensional spatial redundancy through the use of a new 2D Run Length Encoding (2D RLE) algorithm in conjunction with existing encoding methods, such as Deflate, Huffman Coding, and the Burrows-Wheeler Transformation. The proposed 2D RLE algorithm when followed with Deflate compression consistently performed better than PNG for low entropy images and similarly to PNG for high entropy images. The introduction of lossy 2D RLE has been successfully used to compress both high and low-entropy images further than PNG whilst retaining near-perfect visual quality. The implications of the results outlined in this document include improved bandwidth efficiency when transmitting images, reduced storage requirements when archiving images, and consequently a lessened environmental impact stemming from the use of digital images.

Contents

1. Introduction and Motivation **5**
1.1 Project Motivations 5
1.2 Aims and Objectives 5
2. Related Work **6**
2.1 Established Compression Algorithms 6
2.2 Existing 2D RLE Implementations 6
2.3 Post-Processing Methods 7
3. Description of the Work **8**
3.1 Intent 8
3.2 Requirements 8
4. Methodology **9**
4.1 Development Approach 9
4.2 Development Tools 9
5. Design **10**
5.1 2D RLE Algorithm 10
5.2 2D RLE Format and Saving 12
5.3 Huffman Implementation 13
5.4 Deflate Implementation 14
5.5 Burrows-Wheeler Transform (BWT) Implementation 15
5.5 User Interaction 15
6. Implementation **16**
6.1 Platform, Language, and Libraries 16
6.2 Problems Encountered and Consequent Changes 17
6.2.1 Writing to a Binary File 17
6.2.2 Test Images 18
6.2.3 Huffman Coding Inefficiencies 18
6.2.4 Burrows-Wheeler Transformation 18
6.2.5 Memory Leaks 19
6.2.6 Exploiting Entropy 19
6.2.7 Image Clarity and Lossy Compression 20
7. Evaluation **21**
7.1 Testing 21
7.2 Results 21
7.2.1 Lossless Compression 22
7.2.2 Lossy Compression 26
7.2.3 The Effect of the Burrows-Wheeler Chunk Size 28
7.3 Summary 29
8. Progress **29**
8.1 Project Management 29

8.2 Contributions and Reflections 31
8.2.1 Personal Reflection 31
8.2.2 Law, Social, Ethical, and Professional Issues 32
9. Bibliography **33**

1. Introduction and Motivation

1.1 Project Motivations

The rising demand for efficient and fast compression algorithms is a direct result of the continuous increase in the exchange and storage of images; reports state upwards of 3.2 billion images are shared daily (Thompson, Angus and Dootson, 2020). As of September 2020, the United States National Archives and Records Administration stores over 205 million pages of digitised records (National Archives and Records Administration, 2022) and recent innovations in the areas of AI and the accessibility of text-to-image algorithms have resulted in an exponential increase in the number of images created per day. Adobe Firefly generated over 1 billion images within the first three months of its launch (Adobe, 2023). These numbers will only continue to grow over time which is a clear justification for further research into compression methods and effective storage. Improved image compression techniques will result in less bandwidth being needed to transfer these images and will help mitigate the ever-growing need for more storage devices.

Run Length Encoding (RLE) is simple, computationally inexpensive, and effectively addresses local spatial redundancy. If used in combination with existing encoding methods, images with connected blocks of solid colour such as scanned documents, and binary images would benefit greatly from two-dimensional RLE. Existing implementations of 2D RLE differ in method and each has downsides regarding either efficiency of compression, time taken, or wasted computation; these implementations are further discussed in section 2.2.

1.2 Aims and Objectives

This project aimed to explore alternative methods of compression that use a new interpretation of 2D RLE and conduct analysis to determine the most effective amongst them. Certain objectives must have been met to realise the intention of the project. The key objectives include:

- Source a collection of images to test with, this includes the Cifar10 and Fashion-MNIST datasets both of which are commonly used to train machine learning and computer vision algorithms. Other test images should be of the following classifications:
    - Binary Masks
    - Character Sets
    - Medical Scans
    - Landscapes
    - Logos
    - Textures
- Implement the proposed 2D RLE algorithm.
- Implement existing encoding methods to work in concert with 2D RLE. These methods include the following:
    - Deflate
    - Huffman Coding
    - Burrows-Wheeler Transformation

- Test and evaluate each algorithm using compression ratio as a performance metric.
- Compare each 2D RLE algorithm to one another as well as PNG, and BMP to determine the best category of image for 2D RLE compression and the best combination of encoding methods used.
- Provide a user interface that allows the user to compress, decompress, and view images.

2. Related Work

2.1 Established Compression Algorithms

Most existing compression methods fail to fully exploit the intrinsic property of images; being two-dimensional they have both horizontal and vertical spatial redundancy. Bitmap (BMP) (Liesch, 2003) often uses standard Run Length Encoding (RLE) which is limited by the chosen scanline (see section 2.2). A set scanline causes a lack of context sensitivity and as a result BMP is unable to fully make use of correlations between neighbouring pixels. GIF reduces file size through the limitation of an 8-bit colour palette. Whilst GIF is indeed a lossless image format for 8-bit colour images, GIF is restricted from storing true colour - attempting to reduce an image with a bit depth of 24 to only 256 colours will most likely result in colour being lost (Encyclopaedia Britannica, 2023). PNG works by evaluating each row of an image in turn, one of five filter types is assigned to each row. The filter type determines which of the neighbouring pixels - left, above, and upper left - are used when storing each value. PNG compression is complete after the Deflate Compression algorithm is applied. PNG is widely used, has a minimum compression loss, and supports full colour, however, each pixel has the potential to be evaluated multiple times and the 'weighted sum of absolute differences' heuristic used in determining which filter to use is not consistently optimal (Roelofs, 1999). The proposed 2D RLE algorithm has an increased context sensitivity when compared to its 1D counterpart through the use of uniformly coloured rectangles, it supports full colour and does not require the use of a heuristic value to dictate the method of compression.

2.2 Existing 2D RLE Implementations

Different versions of 2D RLE have been explored however their implementations differ. The simplest flattens the image into a single line and performs traditional Run Length Encoding; this method does not exploit vertical inter-pixel redundancies. Similar methods perform RLE along various diagonal scan paths.

A further method defines several shapes, the image is then reconstructed using only combinations of these simple shapes (Wu, Zhou, Wan et al., 2016). Despite the method's impressive compression ratio, it must be performed 2^n times, where n is the number of values present in the grayscale image being compressed, this method is too slow to be practical especially when considering 24-bit colour images.

Another method uses Huffman Coding and is intended for scanned documents and files. It works similarly to traditional RLE except it records consecutive rows and columns rather than individual pixel values (Nuha, 2020). As the image dimensions increase the probability of repeated

identical rows and columns decreases, therefore a mask is introduced to limit the size of the symbols. This method has been shown to work well on binary images with the white space found in documents being mostly responsible for this. Pixels are evaluated more than once as the image is scanned both vertically and horizontally, and an ideal mask size must be found to achieve the best compression.

A different implementation scans the image row by row storing the largest rectangle of uniform colour (Wolfram, 2002). Pixels already belonging to a rectangle are still evaluated and overlapping regions are still considered.

Lastly, the most similar method to the proposed comes from a patent published in 1999 (Wood and Richardson, 1999), in which an image is segmented into rectangles of uniform colour and the starting location to scan for the next evaluating pixel is a corner of the previous rectangle. This implementation does not however have a method to reconstruct the rectangles without storing both the initial rectangle position and its extensions, meaning that a minimum of five values are required per rectangle: x-position, y-position, width, height, and colour value. This is inefficient and solved by the proposed solution.

All current interpretations of 2D RLE have significant downsides, this has been the motivation for exploring and analysing alternative methods of 2D RLE compression.

2.3 Post-Processing Methods

With the intent of increasing the compression ratio achieved by my algorithm, I have researched various existing encoding methods that were anticipated to be useful and have since been implemented in my solution. These methods and how they work are outlined below.

Used in this context Huffman Coding (Huffman, 1952) will result in a binary tree, the leaf nodes of which contain extension and scalar colour values. Beginning traversal at the root node of the tree, representing the left branch as 0 and the right branch as 1, each leaf node will have a unique code depending on the path taken to reach it. 2D RLE must finish running before the use of Huffman coding as all extension-colour pairs from the image and their frequencies must be known to construct the tree.

LZ77 is a form of dictionary encoding, it encodes files into a series of length-distance pairs each of which references data that existed earlier in the file (Ziv and Lempel, 1977) . For this to work the encoder must be able to reference a set amount of the previous data at all times - this is referred to as a sliding window. There have been many iterations on LZ77 such as LZSS which uses a flag to indicate whether the following data is a length-distance pair or whether it is literal data.

Deflate Compression is a combination of LZ77 and Huffman Coding; it performs both in series (Katz, 1991) and is the compression method used by the ZIP file format (PKWARE Inc., 2022). The lengths and literals resulting from LZ77 are encoded into a single tree and the distances into another. As stated before, Deflate compression is the last stage of PNG compression and this project uses it similarly. The implementation covered in section 5.4 first uses 2D RLE to create a stream of data that Deflate will be performed on before saving to a binary file. The use of LZ77 to identify

repeating data throughout the 2D RLE output exploits global redundancy in the image, addressing limitations present in all previously mentioned 2D RLE implementations. If the same combination of extensions and colours are present LZ77 will reference the previous instance rather than duplicating data.

The Burrows-Wheeler Transformation is a reversible transformation that orders data in such a way that neighbouring items are more likely to have the same value (Burrows and Wheeler, 1994). This was expected to be a useful preprocessing method for the deflate algorithm as it will create runs of data that can be further compressed using LZ77. Section 6.2.4 covers the issues encountered when implementing the Burrows-Wheeler Transformation.

Further research into traditional Run Length Encoding has highlighted the use of a flag to indicate when a run of the same colour values is met (AQA, 2020), removing the need to store their lengths. This is very similar to how LZSS differentiates between length-distance pairs and literals using a flag. The 2D RLE algorithm produces a series of x-extensions, y-extensions, and colour values. An early implementation of the software set a flag to indicate that a run of individual pixels has appeared, this had a minimum compression loss as in the worst case of a single (0, 0, value) tuple being encoded it only needed to write the flag, the run length of 1, and the colour value. This was however later removed, the justification for this is covered in section 6.2.4.

3. Description of the Work

3.1 Intent

The intention of the project was to develop compression algorithms that make use of the proposed 2DRLE algorithm in conjunction with existing encoding methods to store images. The desired outcome of this project was a compression algorithm that achieves a better compression ratio than PNG for specific classifications of images.

3.2 Requirements

Non-Functional Requirements

1. Acceptable compression speeds.
2. Compression ratios that perform better than PNG for certain classifications of image.
3. An interface that is easy to understand and use.

Functional Requirements

1. Take PNG image(s) as input.
2. Output a binary file.
3. Reconstruct an image to be viewable using its compressed binary file.
4. Reconstruct an image to PNG using its compressed binary file.
5. Offer three compression pipelines to users, described in sections 5.3, 5.4 and 5.5.
6. Implement the 2D RLE algorithm detailed in section 5.1.

7. Allow the saving of multiple PNG images into a single 2D RLE binary file.

4. Methodology

4.1 Development Approach

Waterfall methodology was not used despite having a clear direction for development in the Interim Report's project plan (see section 8.1). At each step of development, further improvements presented themselves as a result of regular testing, to accommodate for this an iterative agile approach was used. This provided the flexibility to change the plan in response to new ideas, results, and problems. For example, the initial implementation of Huffman Coding had unexpected inefficiencies (see section 6.2), the agile approach allowed for these to be addressed; the algorithm was redesigned to limit the dimensions of stored rectangles to 255. The adaptability of the project allowed for the easy testing and development of ideas whilst still keeping to a timeline.

The software implementation is modular, the goal of the project was to create various compression pipelines that use a series of encoding methods. These methods function independently of one another and have been implemented as such; the main 'Compress' method uses an enumerator to track the user-selected compression pipeline, and a switch-case statement is then used to determine the sequence of methods that need to be called to compress and save an image. The same is true for decompressing and viewing images.

4.2 Development Tools

The project made use of GitLab. Any time a new feature was to be added an issue was made (Figure 1), a milestone assigned, and an appropriate branch created; this branch was worked on until the feature had been implemented at which point it was merged back into 'dev'. Standard practice was followed when writing commit messages (Figure 2); a prefix describing the type of changes made followed by a short description of those changes. All functions and methods implemented have been commented on with the date they were created, the date they were last edited, and a description of how they work (Figure 3).

The use of GitLab allowed for version control and the use of issues helped keep track of ongoing tasks. Comments were written formally so that the code could be understood by myself or other developers in the future. No other tools were used to keep track of tasks and manage the project, the direction of the project was clear throughout development and using both GitLab and the project plan was enough to keep organised.

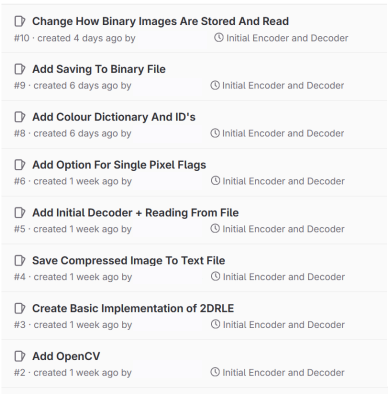


Figure 1

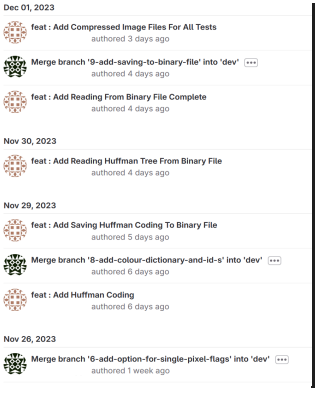


Figure 2

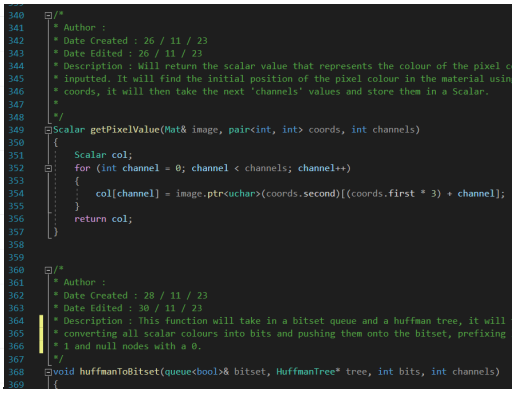


Figure 3

Python was used to assist in the collection and analysis of compression results. Three scripts were created, the first of which uses the TinyPNG API (TinyPNG, n.d.) to compress a directory of images, these images were useful when benchmarking the lossy 2D RLE implementations. The second script compares two directories of images using the Structural Similarity Index Measure (SSIM) (Wang, Bovik, Sheikh, et al., 2004), the first directory contained uncompressed images and the second had images compressed using lossy methods. The final Python script was used to collect various metrics about images including their RAW size and dimensions. The use of these scripts allowed for the accelerated collection of data, without which compiling results would have taken significantly longer. This data was collated in Excel where it is represented using a series of tables and graphs (Appendix. 1).

5. Design

5.1 2D RLE Algorithm

My variation of the 2D RLE algorithm segments an image into rectangles, storing these rectangles as a series of x-extensions, y-extensions, and colours. The design of the algorithm is as follows:

Given an image with width w, height h, and number of colour channels c the implementation loads the image as a matrix m of size $w \times h \times c$. The value of each cell in the matrix corresponds to the colour value of each pixel in the image. $maskSize$ is a constant value of 256 and tolerance t, is a value input by the user. A new matrix b of size $w \times h$ is created and holds 0 in every cell, b is used to maintain which pixels in the image have been explored and to prevent rectangles from overlapping.

Two queues, $topRight$ and $bottomLeft$, are created. These queues hold image coordinates to be evaluated. To begin with, the coordinate $(0,\ 0)$ is pushed to the $topRight$ queue. The algorithm enters a loop, the first step of which is to find a pixel to evaluate, it first checks the $topRight$ queue and if it is empty it checks the $bottomLeft$ queue. If both queues are empty then the algorithm breaks out of the loop and the image has been segmented. Prioritising $topRight$ over $bottomLeft$ means

that the image can be reconstructed using only the extensions of the detected rectangles; their initial positions can be worked out as long as during reconstruction $topRight$ is prioritised over $bottomLeft$.

Once a pixel to be evaluated has been found it's coordinates are stored as $evalPixel$, the next step is to determine the x-extension, x, of the rectangle that stems from $evalPixel$. To do this another loop is used. Each iteration of the loop evaluates a coordinate, stored as $nextPixel$, that is 1 pixel further out horizontally than the last. It will continue to increase the x-extension of the evaluating rectangle until the width of the rectangle exceeds the width of the image, the width of the rectangle exceeds the mask size of 255, $nextPixel$ overlaps with an area of the image that has already been explored, or until the colour of $nextPixel$ is not within the tolerance value (t) of the colour of the evaluating pixel for all colour channels. These conditions are shown below:

- $nextPixel.x < w$
- $b(nextPixel) == 0$
- $nextPixel.x - evalPixel.x < maskSize$
- $m(nextPixel) > m(evalPixel) - t$
- $m(nextPixel) < m(evalPixel) + t$

Once a condition has been violated the x-extension of the rectangle can be calculated as the distance between $nextPixel$ and $evalPixel$. Now that the x-extension has been found the y-extension, stored as y, of the rectangle must be determined. This is done similarly, a loop is entered, within each iteration of the loop y is incremented and it is checked that all pixels from $(evalPixel.x,\ evalPixel.y + y)$ to $(evalPixel.x + x,\ evalPixel.y + y)$ adhere to the same conditions as above. If any of the conditions are violated or if the $nextPixel$ exceeds the height of the image ($nextPixel.y >= h$) then the loop is terminated and y is decremented by 1. The rectangle extensions have been found and to ensure that no future rectangle overlaps this area, all pixels within $b(evalPixel... (evalPixel.x + x,\ evalPixel.y + y))$ are set to 1 to indicate they have been explored.

As justified in section 6.2.7, if an image is lossy ($t > 0$) then the algorithm will check to see whether a colour within $m(evalPixel) + / - t$ is already being stored. If this is the case then $m(evalPixel)$ is set to that colour. The process of searching the existing colour list can take a long time, a full-colour image with a tolerance of 2 could in the worst case produce $621,378$ colours, and so the current implementation will only perform this if $(t >= 5)$ as the colour palette (maximum size of the list) will be in the worst case limited to $(256 / 6)^3 = 77,672$ items for an RGB image.

The entropy of an image can be exploited as discussed in section 6.2.6, the next step in the algorithm is storing the extensions and colour of the detected rectangle. If the image has low entropy x and then y are pushed to a queue called $extensions$, and each colour channel of $m(evalPixel)$ is pushed to a queue called $colours$. If the image has high entropy the difference between the previous x and x is pushed to a queue called $xExtensions$, the difference between the previous y and y to a queue called $yExtensions$, and the differences between each previous colour channel of

$m(evalPixel)$ and the current $m(evalPixel)$ to a queue called $colours$. (Upon the algorithm's completion, if the image has high entropy the output queue $extensions$ is set to be $xExtensions$ followed by $yExtensions$).

Before repeating the process, options for future evaluating pixels must be considered, these are the top right and bottom left coordinates of the rectangle just stored.

- $trAdd = (evalPixel.x + x + 1, evalPixel.y)$
- $blAdd = (evalPixel.x, evalPixel.y + y + 1)$

These coordinates are only suitable and will only be pushed to $topRight$ and $bottomLeft$ respectively if they are within the bounds of the image and unexplored ($b(trAdd) == 0$ or $b(blAdd) == 0$).

Given the output queues of the algorithm, $extensions$ and $colours$, the image can be reconstructed. The origin of each rectangle can be worked out during decompression through the same use of the queues $topRight$ and $bottomLeft$. This solves the problem present in the 1999 patent mentioned in section 2.2 in which five values are required per rectangle: x-position, y-position, width, height, and colour.

5.2 2D RLE Format and Saving

It is important to mention that Bytes are being saved to file as unsigned characters. When individual bits need to be saved, an array of 8 booleans called $bitset$ is first created, in which $bitset[0]$ represents the least significant bit, $bitset$ is then converted into an unsigned character before saving to file. Regardless of the post-processing compression method selected, the same file header information is saved. The format of the file header is this:

- 2 Bytes for Image Width (up to 65,535)
- 2 Bytes for Image Height (up to 65,535)
- 1 Byte for Image Settings / Flags
    - Bitset[0] is set if the image was flagged as High Entropy.
    - Bitset[1] is set if the compression method selected was Deflate.
    - Bitset[2] is set if the compression method selected was Burrows-Wheeler.
    - Bitset[3] is set if the image type is RGB.
    - Bitset[4] is set if the image type is Grayscale.
    - Bitset[5] is set if another image is to be stored after the current.
- 3 Bytes for Number of Quads (up to 16,777,215)

For any 2D RLE file, the header will be contained in the first eight bytes. Storing the image width and height is essential for reversing the 2D RLE algorithm, $topRight$ and $bottomLeft$ pixels that are being considered for evaluation should be discarded if they are outside the bounds of the image therefore width and height must be known prior to decompression.

The 5th byte is very important in the reconstruction of an image:

- As explained in sections 5.1 and 6.2.6 the user-defined entropy of an image changes the order in which data is saved and whether literals or differences are saved, therefore storing the first bit of the 5th byte is crucial.
- The next two bits are important as to reverse the compression, the algorithms used must be known, 10/11 maps to Deflate, 01 maps to Burrows-Wheeler, and 00 to Huffman Encoding.
- The image type must also be known to anticipate how many values should be read per colour, 10/11 maps to RGB and therefore three values, 01 maps to Grayscale and 00 maps to Binary both of which read one value.
- The last 'flag' bit (the 6th bit of byte 5) indicates that once all current image data has been read another image is stored. This allows the compression of entire image directories into one file. For an individual image or the last image in a directory, this bit will not be set so that data that does not exist is not attempted to be read and interpreted.

The final value stored in the header is the number of quads that 2D RLE segmented the original image into. This is valuable as when reading the file it must be known how many extensions and colours to expect, and it can be confirmed that an image has been reconstructed successfully if the same number of quads that are saved have been drawn.

5.3 Huffman Implementation

Once 2D RLE has been completed the data is further processed before saving to the file, if the user selected Huffman Encoding as their post-processing algorithm the following will occur:

The output $extensions$ and $colours$ queues will be combined in series into one single queue before being passed, by value, to a method responsible for encoding these values into a Huffman Tree.

To begin with, an array called $frequencies$ of size 256 is initialised; $maskSize$ restricts the range of values possible from a rectangle extension or colour channel to between 0 and 255 hence the size 256. The input queue is iterated through and the value $frequencies[value]$ is incremented. Once each input value has been processed the list $frequencies$ will be a frequency distribution for the 2D RLE output data.

A structure $HuffmanTree$ is defined as having two integers as contents, the value held and its frequency, as well as two pointers to other $HuffmanTree$ objects, these act as the left and right child nodes. A $HuffmanTree$ object is created for each input value that has a non-zero frequency, the child node pointers are initialised to NULL, and the value and frequency are both set.

A loop is entered and during each iteration of the loop, the two $HuffmanTree$ objects with the lowest frequencies are assigned as children to a new $HuffmanTree$ object whose frequency is

set to the sum of the two child node frequencies. This process is repeated until a single $HuffmanTree$ remains.

The file header is then saved as per the specification detailed in section 5.2, this is then followed by writing the main body of the file. Both the constructed Huffman Tree and the encoded 2D RLE output data must be saved.

To save the Huffman Tree it is first converted into a series of bits, a queue of boolean values $bitset$ is created and the tree is navigated using post-order traversal, pushing 0 to $bitset$ when evaluating branch nodes and 1 when evaluating leaf nodes (Raghunathan, n.d.). When a leaf node is met each bit (from left to right) of the 8-bit binary value held there is also pushed to $bitset$. This has been implemented using a loop, each iteration uses binary shift operators to move the bit to be saved to the first position and $\& \, 0x01$ is used to select only this bit. Once the entire tree has been encoded into a series of bits, the method outlined in section 6.2.1 is implemented to pack these bits into a series of bytes before saving them to the file.

A list of 256 Bitsets, $codemap$, is created, and it is responsible for mapping values to their corresponding Huffman Codes, this is necessary as all values output from 2D RLE must be encoded before saving. To populate $codemap$ post-order traversal is once again used, as nodes are explored, codes are assigned to them, if a left branch was traversed to reach the current node then a 0 is pushed to that node's code, and if a right branch was traversed then a 1 is pushed instead. When a leaf node is evaluated the node's code is added to the $codemap$ at the index held by the node's value. Once $codemap$ is filled it is simple to retrieve a code, for example, the code for value 4 is returned by $codemap[4]$.

Each 2D RLE output value's code is retrieved in order, this results in a large series of bits which are once again saved to file using the method outlined in section 6.2.1.

5.4 Deflate Implementation

For LZ77 to exploit redundancy resulting from the texture and noise of an image the output data of 2D RLE varies; how the output data is formatted is specified in section 6.2.6.

Before performing Deflate the size of the input data in bytes is found, the input data is converted into one continuous stream of characters, and memory for the compressed data is allocated. Calling the 'ZLIB' function 'compress2' performs the algorithm and returns an integer to indicate whether it was successful or not.

If Deflate was successful then the file header is saved as explained in section 5.2 and then the Deflate data is saved. The Deflate data consists of four bytes to store the number of bytes the compressed data occupies, four bytes to store the number of bytes the uncompressed data occupies, and the actual compressed data which is written one byte at a time.

Storing the compressed size is necessary so that when the file is read the correct amount of data is sent to be inflated, and storing the uncompressed size is necessary as the 'ZLib' function that handles inflation requires an array, with enough memory for the uncompressed data, be passed in.

5.5 Burrows-Wheeler Transform (BWT) Implementation

The Burrows-Wheeler compression pipeline is a straightforward attempt at increasing the coding redundancy in the output data of 2D RLE that can be exploited by the LZ77 component of Deflate compression. As such this pipeline is identical to that of the Deflate pipeline except it performs the Burrows-Wheeler Transformation (BWT) on the extensions data (section 6.2.4 justifies the exclusive use of the Burrows-Wheeler algorithm on extensions data) before converting it into one continuous stream of bytes, as well as using a $maskSize$ of 255 instead of 256 when performing 2D RLE (the change in mask size reserves the value 255 to be the unique identifier used when performing BWT).

The extension data is segmented into chunks of length $chunkSize$, once encoded these chunks are reassembled. Encoding begins by inserting the value 255 at the end of the chunk. A table of size $chunkSize + 1 \times chunkSize + 1$ is created, each row of this table is populated by the input chunk data rotated cyclically by 1 cell further than the previous row. The table is then sorted lexicographically and the transformed data can be found to be the last column.

The data output from the BWT is 1 byte larger per encoded chunk than the data input due to the unique identifier 255 being inserted, for BWT to be worthwhile it must result in at least 2 similar values that were not grouped previously to be grouped now whilst retaining any previous coding redundancies. This becomes much more plausible when using a larger $chunkSize$. However this comes with complications regarding time, proof of which can be seen in section 7.2.3.

No additional data needs to be stored when using this compression pipeline compared to using pure Deflate; the only data required to reverse the Burrows-Wheeler Transformation is the number of chunks, the number of bytes storing BW values, the unique identifier, and the $chunkSize$. The unique identifier will always be 255 and the $chunkSize$ has been set to 64 and is not editable outside of the code. The number of chunks can be worked out as $(((noQuads * 2) - 1) / chunkSize) + 1$ and the number of bytes storing BW data as $(noQuads * 2) + noChunks$.

5.5 User Interaction

As this is primarily a research project no attempt at an interactive user interface has been made, therefore the user interacts with the software via the console. The user is asked a series of questions, and switch-case statements determine the behaviour of the program depending on the answers to those questions. The user is first asked whether they are compressing, viewing, or decompressing an image.

If the user selects compressing it will ask whether they are compressing a single image or directory. They will then be prompted for the image or directory filepath, its existence will be verified and the compress method will be called (if a directory is being compressed it will also ask the user whether they would like to use the same compression rules for all files). The user then needs to choose the rules of compression, this consists of which compression pipeline to use, what tolerance value they would like, and whether or not the image has large areas of uniform colour (entropy of the image). Each time a new process begins the user is updated on the progress of compression via text output. At the end of compression, the user is provided the following information: uncompressed file size, compressed file size, compression time, and compression ratio.

There is only one difference between viewing and decompressing 2D RLE files, viewing will present the reconstructed image(s) to the user, and decompressing will instead save the image(s) to file as a PNG. Upon selecting either option the user will be prompted for a filepath. The header of the file is read and the inverse algorithm to each post-processing method mentioned in the above sections will be performed, each one will output a queue of $extensions$ and a queue of $colours$. These queues are used to reconstruct the image using the inverse function of 2D RLE. If the image header indicates that another image follows, the user will be asked whether they would like to continue, and if so the process repeats, starting by reading the header of the next image. Upon completion, it will show the user the time taken to decompress the file and ask the user if they would like to process another item.

6. Implementation

6.1 Platform, Language, and Libraries

The resolution of target images varies wildly; millions of pixels may need to be processed and as such any potential increase in speed and efficiency should be utilised. Therefore, this project has been coded in C++ with Visual Studio 2019. The direct memory access offered by C++ through pointers allows for more control over memory and therefore, pixel values. The debugging tools offered by Visual Studio further assisted in the development process.

The library ‘OpenCV2’ (OpenCV, 2023) was initially used to load and manipulate images; it automatically does the conversion from image data into a matrix allowing for the easy access and processing of pixels. For reasons discussed in section 6.2.6 ‘OpenCV2’ was not suitable and as such replaced with the library ‘CImg’ (Tschumperle, 2024). Unlike ‘OpenCV2’ ‘CImg’ does not support opening any image format other than BMP, to remedy this the library ‘ImageMagick’ (Cristy, 2024) is used in conjunction with ‘CImg’. If a user wants to compress an image of any other type than BMP they cannot without the use of ‘ImageMagick’ which can be easily installed, however, the user may decompress and view any image without installing additional libraries.

As mentioned in section 4.2 Python was used to assist the collection and analysis of compression results. Python libraries used include, ‘Pillow’ (Clark et al., 2024), ‘Glob’, ‘SKImage’ (Van Der Walt, Schönberger, Nunez-Iglesias et al., 2014), ‘Tinyfy’ (TinyPNG, n.d.), and ‘Natsort’ (Morton, 2023). ‘Pillow’ was used to open images and calculate raw image size in bytes. ‘Glob’ was used to gather all image file paths within a directory. The structural similarity (SSIM) function

provided by ‘SKImage’ has been used to evaluate image quality and ‘SKImage’ has been used to calculate average image entropy. ‘Tinyfy’ has been used to interact with the Tiny PNG API, this API allows for the lossy compression of images. Windows orders files numerically however when file paths are read using ‘Glob’ they are ordered alphabetically. When comparing the SSIM of entire directories ‘Natsort’ is used to ensure compressed images match up with their respective uncompressed counterparts.

The C++ libraries ‘fstream’ and ‘iostream’ are used to save and load the resulting compressed images to and from the file. This library has been chosen as it can be used to create binary files. ‘fstream’ also uses a buffered output to reduce the number of writes and consequent I/O overhead.

Libraries ‘Visual Leak Detector’ (Shapkin, 2017) and ‘CRT’ have been used throughout development to detect memory leaks.

The last notable library used was ‘ZLib’ (Gailly and Adler, 2024), which provides Deflate and Inflate methods within a single header file.

6.2 Problems Encountered and Consequent Changes

6.2.1 Writing to a Binary File

The minimum size that can be written to a binary file is 1 byte. Save data can be categorised into two groups, data where $(data\ size\ in\ bits)\ mod\ 8 == 0$ and $(data\ size\ in\ bits)\ mod\ 8 != 0$. Which group data belongs to can be ignored when assessing the endianness of the system as both methods of storing data write the most significant byte first and then each following byte sequentially, therefore 2D RLE files are stored in big endian format.

To store the first category of data an unsigned character is created, packed with 8 bits using bitwise shift operators, and written to file. This is repeated until all bytes in the value to be stored have been written, an example of this is shown below.

- $w = 1980$ (the 2-byte binary representation of this is 0000011110111100)
- unsigned character $byte$ = 0
- $byte = w >> 8$ (representing the first 8 bits of w 00000111)
- $byte$ is then written to file
- $byte = w$ (representing the remaining 8 bits of w 10111100)
- $byte$ is once again written to file

To store the other category of data (data of variable bit length) it is first converted into a Bitset (an array of boolean values), an unsigned character is again created and a loop begins, each iteration of the loop packs the character (from the most significant bit to the least) with 1 bit from the Bitset using bitwise shift operators . Once the unsigned character is full it is written to the file as a byte. This continues until all bits of the Bitset have been written to file. An example use case is the saving of the Huffman Coding Tree as it is always of variable length.

6.2.2 Test Images

Initially, the dataset of test images consisted of 10 images from each classification - Binary Masks, Medical, Scanned Documents, Landscape, and Textures, results showed promise when compressing scanned documents however more test data was required to definitively conclude performance when compared to PNG. The work plan was adjusted to allow time for further images to be collected. Ultimately 20,212 images were collected and used for testing. These images are from a total of 8 classifications outlined in section 1.2. The collected images are of a wide variety, some are grayscale, some are binary, some are textured full colour etc. To source test images a list of properties was first created: Binary, Grayscale, RGB, Textured, Uniform, Large, Small. Combinations of these properties were used to identify useful categories of image, for example Scanned Documents falls under Grayscale and Uniform. Once a category of image had been identified, websites such as Kaggle were used to find datasets that distributed the desired images in the PNG file format.

6.2.3 Huffman Coding Inefficiencies

Performing Huffman Coding on an image with bit-depth 24 and assigning codes to each RGB value is very computationally expensive, slow, and ineffective; possible 2^{24} colours could be encoded. Evidence of this inefficiency can be found in section 7.2.1 by the compression ratio of images classified under ‘landscape’. To address this each channel value is encoded separately.

Previously two Huffman Trees were saved, one that encoded quadrilateral extensions and one that encoded colours. The result of encoding each colour channel separately meant the size of the colour Huffman Coding Tree could at maximum have 256 leaf nodes. Most images will not contain uniform areas of width or height larger than 255 and therefore the decision was made to limit the possible extension size of a saved quadrilateral to 255. Introducing this mask size allowed for both extensions and colours to be encoded using the same tree.

These changes have resulted in faster, more efficient compression, proof of this can be found in section 7.2.1.

6.2.4 Burrows-Wheeler Transformation

The Burrows-Wheeler Transformation requires a unique identifier to be inserted at the end of the input data so that the transformation may be reversed. The value called $maskSize$ in the 2D RLE algorithm describes the maximum width and height of evaluated rectangles, the $maskSize$ used always limits extension values to 255 except for when the BWT pipeline is used. Considering the mask size limits extensions to 255 and the maximum colour value stored is 255 the unique identifier must be the number 256 or greater. However, due to the Deflate and Inflate methods provided by ‘ZLib’ taking in and outputting data as a stream of bytes, when reconstructing the image there would be no way of discerning the following 2 bytes 00000001 00000000 as 1 followed by 0 or as 256. The solution to this problem was to limit the mask size of 2D RLE to 254, reserving 255 to be the unique identifier and then performing the BWT on the extension values exclusively. The intention of implementing BWT was to reorder data to increase coding redundancy before performing Deflate

compression. As BWT was unable to be used on colour values, the coding redundancy of the 2D RLE colour output is increased by exploiting the properties that entropy provides, these methods are further discussed in section 6.2.7.

Time and memory were further problems that presented themselves whilst implementing the Burrows-Wheeler Transformation. The algorithm requires that a table of size x^2 where x is the length of the input data, be created and sorted lexicographically x times. An 8-bit image of size 512x512 may in the worst case present itself as a series of 262,144 quadrilaterals each consisting of an x-extension, y-extension, and colour. To perform BWT on a list of this size will take an impractical amount of time as well as $(512 * 512 * 3)^2$ Bytes of memory which equates to roughly 618 Gigabytes. This issue was solved by dividing the input data into chunks and performing BWT on each chunk independently. The chunk size has been set to 64 however this can be changed depending on the needs of the user. The effects of chunk size on compression ratio and compression time can be found in section 7.2.3.

6.2.5 Memory Leaks

The library 'OpenCV2' had been used to load and manipulate images however after testing with 'Visual Leak Detector' and 'CRT' it was apparent that 'OpenCV2' was responsible for a number of memory leaks. Despite following instructions provided by 'OpenCV2', the memory occupied by loaded images would not be released. To fix this issue the library used to read in images was changed to 'CImg' in conjunction with 'ImageMagick'. This change significantly reduced the amount of memory leaks detected, and those that remain are the fault of the 'CImg' library. 'Visual Leak Detector' and 'CRT' both detect similar leaks when analysing compiled 'CImg' example programs provided by the library themselves, and each time memory is allocated for an image the appropriate function is always called to clear and release it, this has led me to believe that these tools are falsely detecting leaks. Switching the library used to interact with images, whilst reducing the detected memory leaks, also allows for the project to be compiled without the need to dynamically link 'OpenCV2'.

6.2.6 Exploiting Entropy

Compression makes use of redundancy within data, images with significant randomness hold redundancy in different ways than uniform images. Noise and texture within an image means that more often than not the extensions output from 2D RLE are small and neighbouring quadrilaterals are of similar size. The output of 2D RLE has been changed to take advantage of this property. Extensions are stored in the format xxx… yyy… and rather than store literal values the initial extension values are stored and then only the differences between each following extension are stored. Storing the differences over literals combined with the property that neighbouring quadrilaterals will be of similar sizes results in a higher concentration of lower numbers being stored. The increased concentration of lower numbers output from 2D RLE introduces coding redundancy that is exploited by LZ77 and Huffman Coding whose algorithms use repeated sequences and frequency respectively. The same approach is applied when encoding colours, for images with noise, colours are stored by channel rrr…

ggg… bbb…, the logic being that individual channel values will be similar to those of their neighbours.

When storing images that have large areas of uniform colour the opposite is true. The extensions are stored in the format xyxyxy… and colours rgbrgbrgb…. The literal values of both extensions and colours are also stored. If there are both large and small extensions being stored then storing the differences is negligible. Images that have large areas of uniform colour tend to lack gradients and consequently have harsher transitions between colours, this behaviour is the justification for storing the colour literals over the differences and for formatting colours not by channel. Evidence justifying these decisions is presented in section 7.2.1.

The motivation for storing the file as extensions then colours rather than as a series of extension colour pairs is similar. The values of neighbouring extensions are likely to be much more similar when compared to the value of an extension and its neighbouring quadrilaterals' colour. It is this change that caused the initial removal of the previously mentioned (section 2.3) flag indicating a run of single pixels. For this flag to work the file must be a series of extension colour pairs otherwise there is no indication of how many single pixels follow.

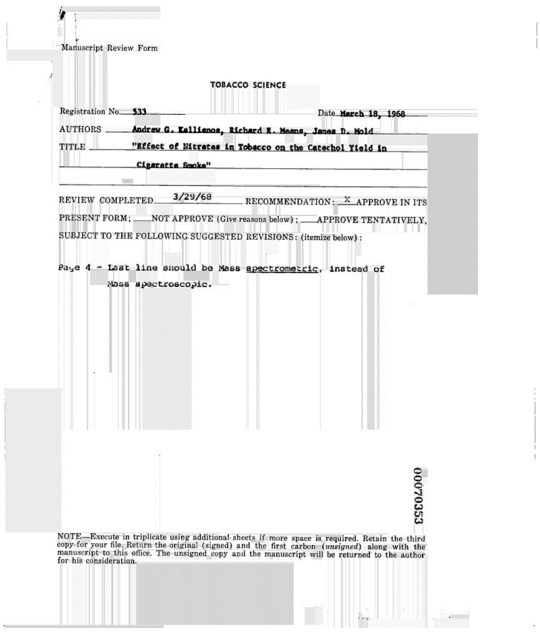
The reason a flag in the form of 256 followed by the number of individual pixels in the run was not implemented instead is because, as discussed in section 2.3, LZ77 stores data as length-distance pairs which ultimately exploits any redundancy that may occur as a result of repeated sequences.

A summary of the output format of 2D RLE data is as follows:

- High Entropy - xxx…yyy…rrr…ggg…bbb… (Differences)
- Low Entropy - xyxyxy… rgbrgbrgb… (Literals)

6.2.7 Image Clarity and Lossy Compression

The original implementation of lossy compression simply checked whether all colour channels of an evaluating pixel were within a tolerance value of the first. However, this would lead to neighbouring quadrilaterals having very contrasting colours, especially at higher tolerance values. This can be seen in Figure 4 and as shown it is extremely noticeable. To fix the blocky texture a method, suggested by my supervisor, determines whether a colour within the tolerance value of the first pixel to be evaluated is already being saved and if so, save the current quadrilateral as having that colour. This method not only reduces the blocky texture present (Figure 5) it also restricts the image's colour palette. In the worst case, the colour palette will be of size $(256 \,/\, t + 1)^c$ where t is tolerance and c is number of channels, and in the best case the colour palette will be reduced to size $(256 \,/\, 2t + 1)^c$. A grayscale image with a tolerance of 20 will store between 7 and 13 unique colours. The quantisation of stored colours also reduces file size.



Manuscript Review Form

TOBACCO SCIENCE

Registration No. 533 Date March 18, 1968

AUTHORS Andrew G. Kallimos, Richard E. Means, James D. Mold

TITLE "Effect of Nitrates in Tobacco on the Catechol Yield in Cigarette Smoke"

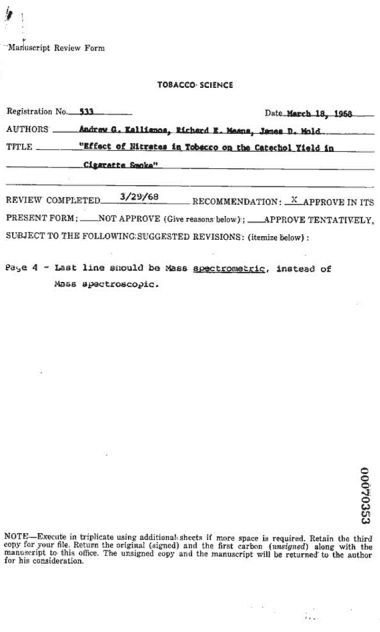
REVIEW COMPLETED 3/29/68 RECOMMENDATION: X APPROVE IN ITS PRESENT FORM; ___NOT APPROVE (Give reasons below); ___APPROVE TENTATIVELY, SUBJECT TO THE FOLLOWING SUGGESTED REVISIONS: (itemize below):

Page 4 - Last line should be Mass spectrometric, instead of Mass spectroscopic.

00070353

NOTE—Execute in triplicate using additional sheets if more space is required. Retain the third copy for your file. Return the original (signed) and the first carbon (*unsigned*) along with the manuscript to this office. The unsigned copy and the manuscript will be returned to the author for his consideration.

Figure 4



Manuscript Review Form

TOBACCO SCIENCE

Registration No. 533 Date March 18, 1968

AUTHORS Andrew G. Kallimos, Richard E. Means, James D. Mold

TITLE "Effect of Nitrates in Tobacco on the Catechol Yield in Cigarette Smoke"

REVIEW COMPLETED 3/29/68 RECOMMENDATION: X APPROVE IN ITS PRESENT FORM; ___NOT APPROVE (Give reasons below); ___APPROVE TENTATIVELY, SUBJECT TO THE FOLLOWING SUGGESTED REVISIONS: (itemize below):

Page 4 - Last line should be Mass spectrometric, instead of Mass spectroscopic.

00070353

NOTE—Execute in triplicate using additional sheets if more space is required. Retain the third copy for your file. Return the original (signed) and the first carbon (*unsigned*) along with the manuscript to this office. The unsigned copy and the manuscript will be returned to the author for his consideration.

Figure 5

7. Evaluation

7.1 Testing

The collection of an image dataset at the start of development has allowed for testing to occur using the same data for the duration of the project. Each iteration either changes what image data is stored or how the image data is stored. To ensure that changes to the codebase result in compression that retains enough information for images to be reconstructed the encoders and decoders for each algorithm have been developed in parallel; each time a change is made to the compress function, the decompress function reflects it. This method of development aids in finding issues with compression as decompressing an image allows for visual evaluation. To ensure that lossless compression is truly lossless the SSIM scores for the compressed and decompressed images have been calculated and are all 1.0.

As mentioned in section 6.1 libraries ‘Visual Leak Detector’ and ‘CRT’ have been used throughout development to test for memory leaks. This regular testing was the reason that the memory leaks associated with ‘OpenCV2’ were caught with enough time before the end of the project to substitute the library with ‘CImg’.

7.2 Results

As presented in section 1.2 this project aimed to explore alternative methods of compression that use a new interpretation of 2D RLE and determine the most effective amongst them. To meet the objectives of the project various metrics needed to be gathered and certain behaviours observed, for

each algorithm implemented data collected includes the compression ratio of various classifications of image, the compression time, the effect of tolerance on compression ratio and structural similarity, the effect of chunk size on compression ratio, and the effect of image entropy on compression ratio. All data was collated into a spreadsheet (Appendix. 1) before being presented below as a series of graphs.

7.2.1 Lossless Compression

The primary metric for evaluating image compression is compression ratio; one aim of the project is to produce an algorithm that outperforms PNG in terms of compression ratio for a definable classification of images. As such, the compression ratio of all lossless implementations of the proposed 2D RLE algorithm for twelve image datasets has been collected.

The image datasets were separated by folder, and each folder was compressed separately. Due to how the 2D RLE format stores multiple images in the same file, a directory in which each file is compressed and saved separately will be no larger then if the directory itself is compressed into a single file.

To aid in the calculation of compression ratio a Python script was created, it multiplies each image's width, height, and bit depth and divides this value by eight. The sum of these values makes the raw size (in bytes) of a directory of images. Once a directory has been compressed the raw size of the directory is divided by the size (in bytes) of the newly created 2D RLE file, this is the compression ratio.

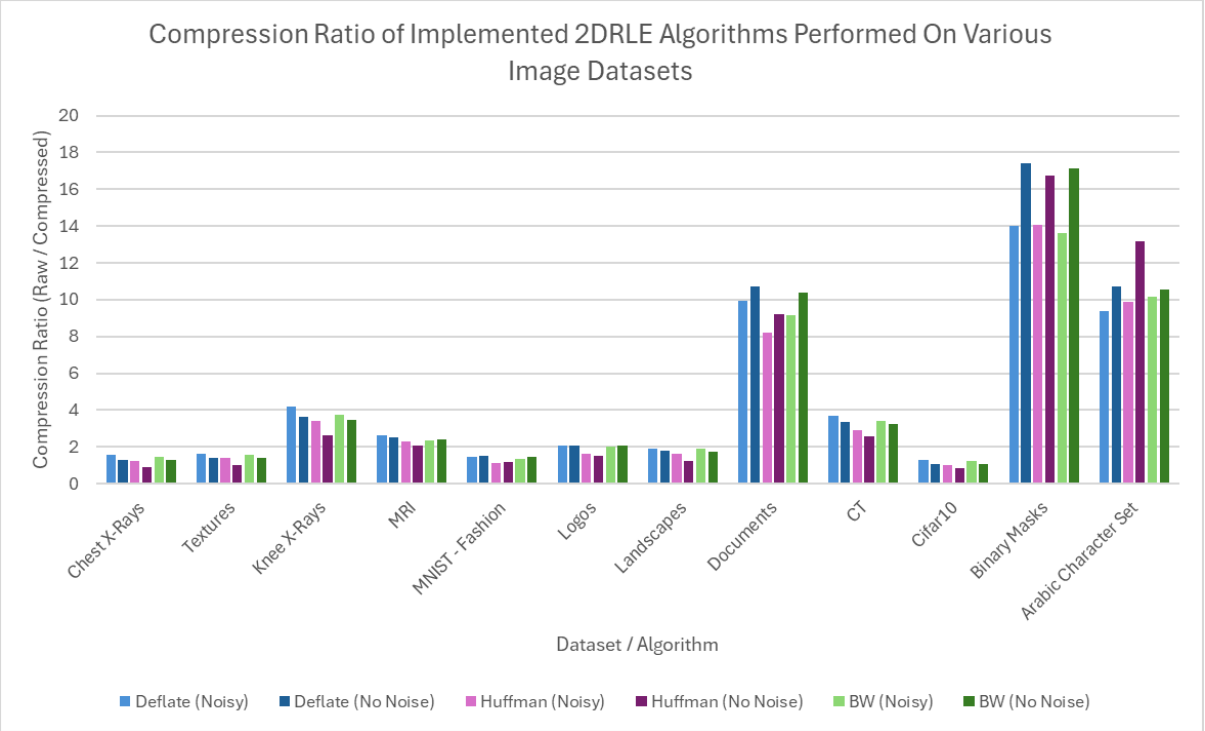


Figure 6

Figure 6 shows the compression ratio achieved by each 2D RLE implementation for each dataset, a compression ratio of 1 means that the compressed file is the same size as the raw image (raw image size is calculated as $width * height * bit\ depth$). Immediately from Figure 6 it can be observed that the algorithm performs significantly better for the Scanned Documents, Binary Masks, and Character Set datasets. Figure 6 also shows that the implementation using the Deflate algorithm has the highest average compression ratio; it has a higher compression ratio for every dataset except for the Character Set for which it is outperformed by the Huffman Implementation. It can also be seen that the Burrows-Wheeler implementation consistently performs worse than its Deflate counterpart except for the Character Set dataset, for which the BW Noisy (high entropy) algorithm outperforms the Deflate Noisy algorithm.

The notable property of the Arabic Character Set dataset and the probable reason for Huffman Coding outperforming Deflate is the resolution of the images. Each image within the dataset is $32x32$ pixels, with less data also comes less coding redundancy that can be exploited by LZ77 so the importance of code length increases.

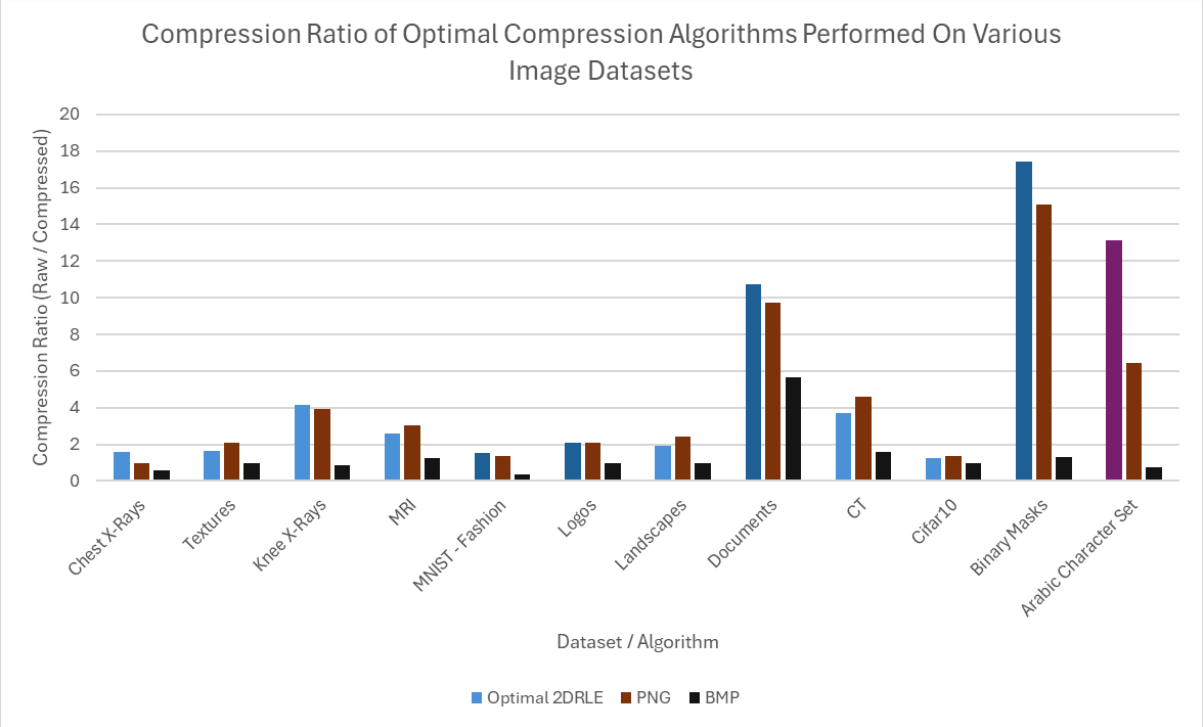


Figure 7

To compare the potential of 2D RLE to existing lossless compression methods, PNG and BMP compression ratios have been gathered for the same datasets. These have been plotted against the optimal 2D RLE implementation for each dataset (Figure 7).

All images collected were stored in the PNG format so the compression ratio was simple to determine, the previously calculated raw size of each dataset was divided by the sum of PNG sizes within the directory. To determine the compression ratio of BMP the library 'ImageMagick' was used to convert each PNG image to BMP before performing the same calculations.

As can be seen from Figure 7, 2D RLE outperforms BMP for every dataset and PNG for both X-Ray datasets, the MNIST-Fashion, Scanned Documents, Binary Masks, and Arabic Character Set datasets. These results show that my implementation of 2D RLE has wider applications when compared to methods discussed in section 2.2 such as the paper, which due to its dependence on repeated rows and columns, exclusively tested its algorithm on binary generalisations of scanned documents (Nuha, 2020).

2D RLE performs best on images classed as low entropy, these are images with large areas of uniform colour and little randomness. Figure 8 shows the relationship between image entropy and compression ratio for both 2D RLE and PNG. Each point on the graph represents a dataset that has been compressed using either 2D RLE or PNG.

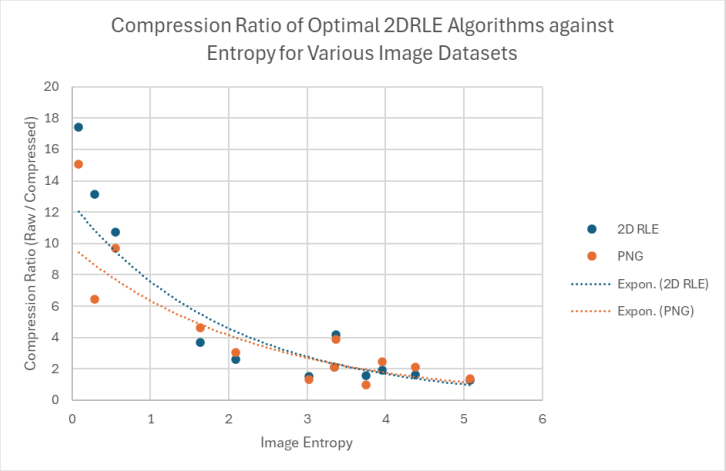


Figure 8

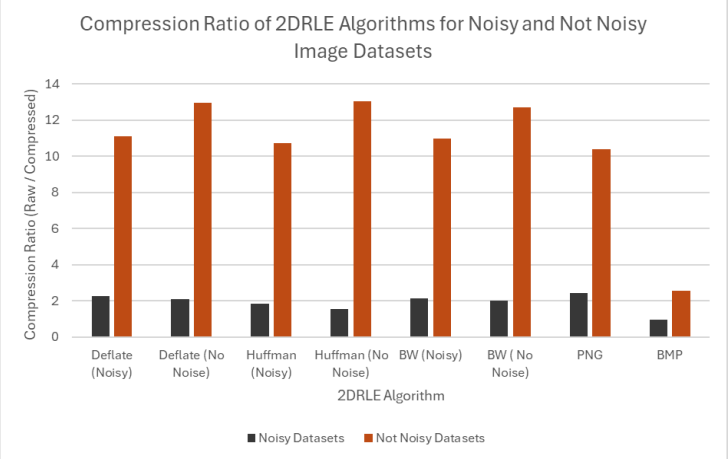


Figure 9

Entropy has been found using a Python script. For each image in a directory it loads the image with 'OpenCV2' converts the image to grayscale and uses the 'SKImage' function 'entropy'. Each returned value is summed and once all images have been evaluated the average is calculated by dividing by the number of images in the dataset.

As explained in section 6.2.6 data output from 2D RLE is organised differently depending on how the user classes the entropy of the input image(s). Figure 9 shows how each algorithm performs against low and high-entropy datasets.

The twelve datasets previously encoded have each been classified as either low or high entropy depending on whether the average image entropy of the dataset is greater than or less than 1.

Figure 9 shows that for low-entropy (Not Noisy) datasets the 2D RLE

low-entropy (No Noise) implementations outperform the 2D RLE (Noisy) implementations and that for high-entropy datasets the 2D RLE (Noisy) implementation outperform the 2D RLE (No Noise) implementations. This is confirmation that the methods used to reformat data output from 2D RLE depending on entropy are successful in increasing coding redundancy.

Figure 9 also shows that similar to PNG, all implementations of 2D RLE perform better on datasets with low entropy. This is to be expected as 2D RLE exploits spatial redundancy which is more prevalent in images with less randomness.

Another important metric when assessing compression algorithms is compression time. The compression time of each 2D RLE implementation is measured using the C++ library 'Chrono'. When compressing an image or directory the time between loading the input image and saving the 2D RLE output is recorded in milliseconds. The average compression time for a dataset has been calculated as the sum of each image's compression time divided by the number of items in the dataset.

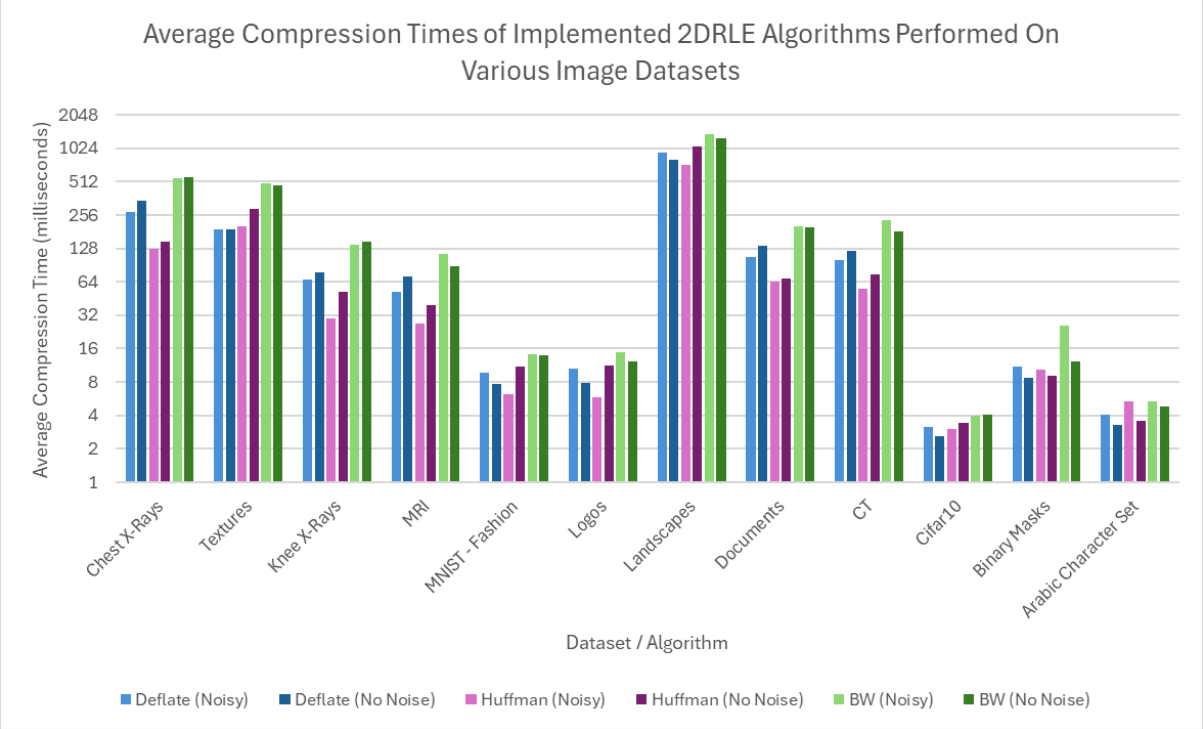


Figure 10

The average compression time of each 2D RLE implementation for each test dataset has been recorded and shown in Figure 10. It can be observed that the average compression time for the Burrows-Wheeler implementation is consistently the highest, then Deflate, and finally Huffman. It should be noted that the y-axis of Figure 10 uses a logarithmic scale to better fit the data. The graph shows that even for high-entropy, high resolution, full colour images such as those present in the

Landscape dataset, when using the Deflate variation, the average compression time does not exceed 1 second. When comparing compression times to the dictionary of predefined shapes method (Wu, Zhou, Wan et al., 2016), outlined in section 2.2, my 2D RLE implementation is much more practical for a much wider range of images (in particular colour images).

7.2.2 Lossy Compression

As described in sections 5.1 and 6.2.7 the current implementation of 2D RLE supports lossy compression. The intensity of compression is controlled by a tolerance variable. To summarise, the higher the value of tolerance the more colours are accepted into each rectangle.

The relationship between tolerance and compression ratio can be seen in Figures 11 and 12. Figure 11 uses the Scanned Documents dataset to represent low entropy images and Figure 12 uses the Landscape dataset to represent high entropy images. Rather than calculate the compression ratio using the raw image sizes like in section 7.2.1 the PNG image sizes are used; a compression ratio greater than 1 means that 2D RLE produced a file smaller than its PNG equivalent. Both Figure 11 and Figure 12 show the average compression ratio of the popular online lossy compression algorithm 'TinyPNG' for their respective dataset. For the Scanned Documents dataset, a tolerance of 5 is required to outperform 'TinyPNG' and for the Landscape dataset, a tolerance of 10 is required.

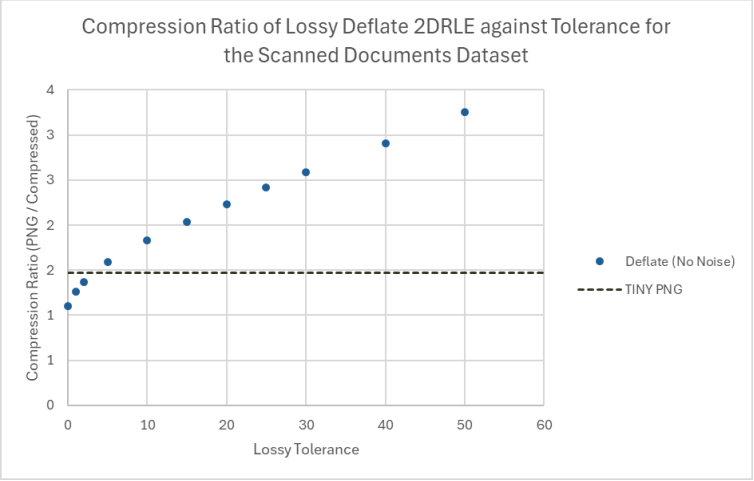


Figure 11

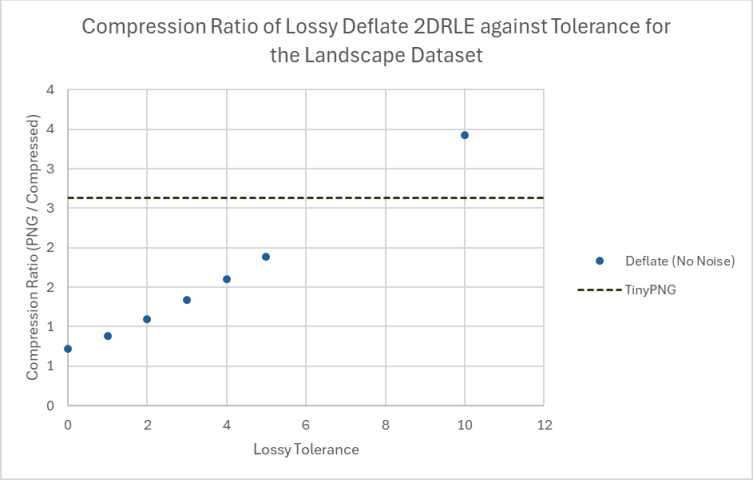


Figure 12

From both diagrams, it can also be observed that tolerance has a greater effect on high-entropy images. High entropy images by nature have more texture and noise, thus providing a greater chance for neighbouring pixels to have similar colours which can be grouped.

Lossy compression is only useful if only the least perceptual details and information are discarded. The Structural Similarity Index Measure

(SSIM) is a measure of image quality, it evaluates an image with respect to another, the higher the output value the more similar the two input images are, an output of 1 indicates perfect similarity. A 2013 experiment concluded that 'the point at which a human observer cannot determine that compression has been used hovers around an SSIM value of {0.}95' (Flynn, Ward, Abich IV et al., 2013).

Figures 13 and 14 show the relationship between SSIM and tolerance for the Scanned Documents and Landscape image datasets. The average SSIM value of the images output from 'TinyPNG' is marked on each graph for comparison.

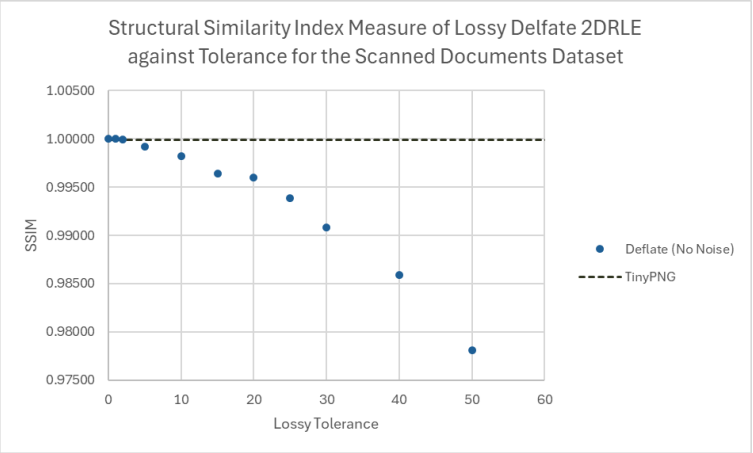


Figure 13

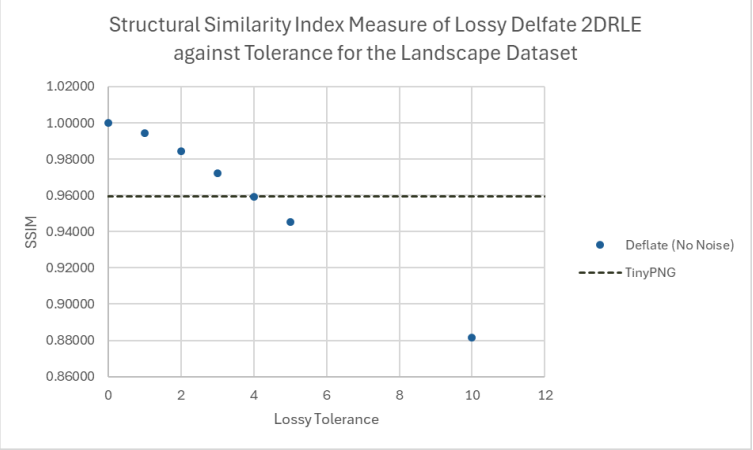


Figure 14

To graph the effect of the tolerance value on SSIM both datasets were compressed with 2D RLE using a range of tolerance values and then decompressed and saved as a series of PNGs. A Python script using 'SKImage' iterates through both the uncompressed and compressed directories calculating the SSIM for each corresponding item. The average SSIM is then calculated using the total SSIM divided by the number of items in the dataset. This is repeated for both datasets using the following tolerance values: 1, 2, 3, 5, 10, 15, 20, 25, 30, 40, 50.

Figure 13 shows that for the Scanned Document dataset, the image quality is preserved even at high tolerance values. Figure 14 shows that for the Landscape dataset, the image quality rapidly decreases as tolerance increases; it passes the SSIM threshold of 0.95 when tolerance is greater than 4.

Figure 15 and Figure 16 show the compression ratios achieved, for the Scanned Documents and Landscape datasets, by the 2D RLE algorithm when using the highest possible tolerance value that retains an SSIM of 0.95 or above.

The compression ratios of PNG, BMP, and 'TinyPNG' are plotted for comparison. It is shown that for Scanned Documents lossy 2D RLE can accomplish a compression ratio significantly larger

than PNG, BMP, and ‘TinyPNG’, whereas for the Landscape dataset, 2D RLE can outperform PNG and BMP but not ‘TinyPNG’.

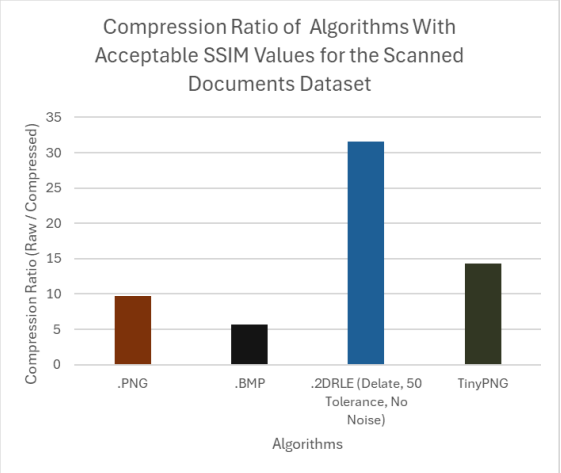


Figure 15

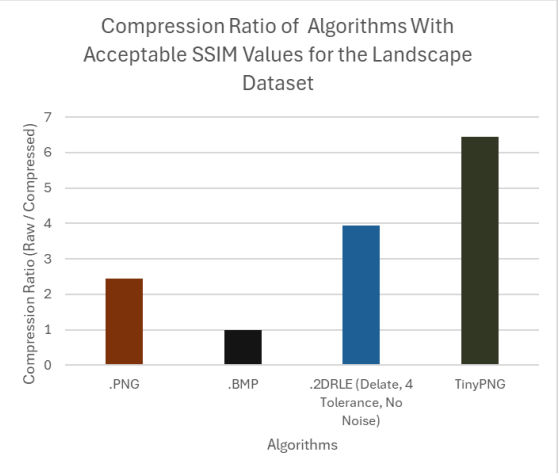


Figure 16

7.2.3 The Effect of the Burrows-Wheeler Chunk Size

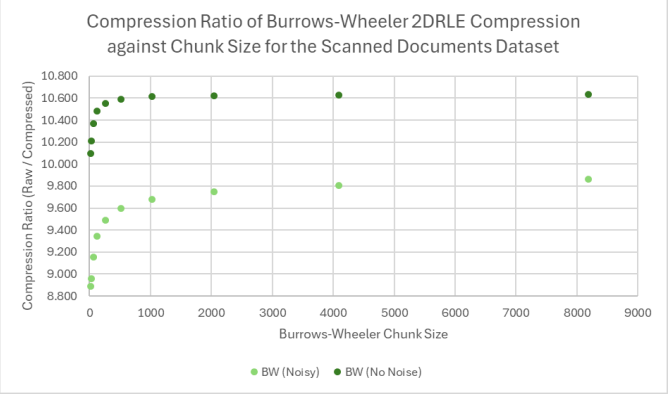


Figure 17

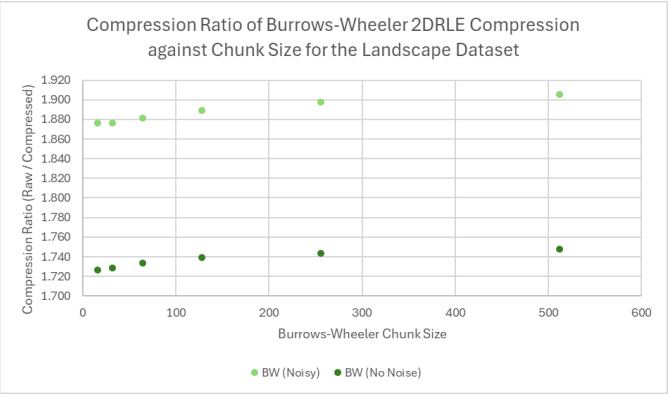


Figure 18

Section 6.2.4 details how splitting the extension output from 2D RLE into a series of chunks helps reduce memory and time when performing the Burrows-Wheeler transformation. The downside of splitting the data into chunks is that for every chunk a unique identifier must be inserted into the data. The relationships between compression ratio and chunk size for both the Scanned Documents and Landscape datasets have been recorded (Figures 17 and 18).

As shown, the larger the chunk size the larger the compression ratio, however as chunk size increases the compression ratio increases in smaller increments. With a chunk size of 8192, the compression ratio of the Scanned Documents dataset having applied the BW No Noise (low entropy) algorithm is 10.637 which is still lower than if the Deflate No Noise algorithm was used (10.734).

Section 6.2.4 highlighted the need to implement chunk size to get acceptable compression times. Figures 19 and 20 show the relationship between compression time and chunk size for both the Scanned Documents and Landscape datasets. As expected, as chunk size increases so does compression time.

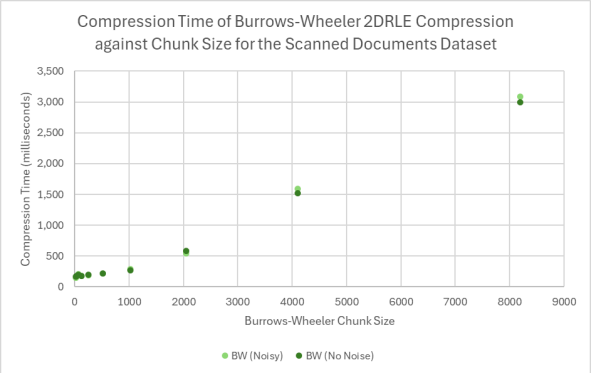


Figure 19

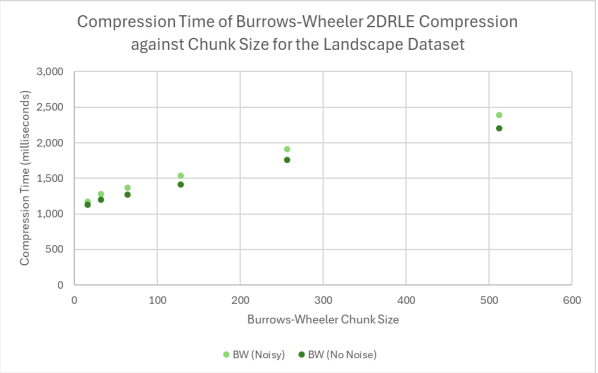


Figure 20

7.3 Summary

The most notable results are the following:

- The implementation that uses Deflate compression provides the best compression ratio for most datasets.
- All implementations of 2D RLE compress low-entropy images further than high-entropy images.
- The methods employed to reformat the output data of 2D RLE depending on the entropy of the image are justified.
- The lossless 2D RLE implementation that uses Deflate compression consistently performs better than PNG for low entropy images (<1) and similarly to PNG for high entropy images (>1).
- Increasing the chunk size used when compressing with the Burrows-Wheeler implementation of 2D RLE will not result in a better compression ratio than the Deflate implementation, however, it will significantly increase compression time.
- Lossy 2D RLE can be successfully used to compress both high and low-entropy images further than PNG whilst retaining visual quality (an SSIM value >0.95).

8. Progress

8.1 Project Management

Two project plans were created to aid in development, these were the initial plan and the revised plan. The initial project plan assigned a week for the collection of test images, this amount of

time was not necessary and as a result, the initial Huffman Coding implementation was completed sooner than expected. The initial project plan was incredibly valuable when managing time and was, for the most part, followed without deviation. Results gathered mid-way through development reaffirmed the choices of encoding methods and helped identify three compression pipelines to be implemented. This alongside an unexpected extension in time allowance in January caused for a slight revision of the project plan, this can be seen in the list below and in Figure 21.

Revised Project Plan

A. Revise and finish the Project Proposal and Data Management Plan.
B. Perform further research into existing lossless image compression methods and their implementations.
C. Gather a set of test images.
D. Improve upon the design of the proposed algorithm.
E. Implement the 2D RLE algorithm encoder and decoder in parallel.
F. Implement Huffman Coding for colours.
G. Write the interim report.
H. Gather a larger set of test images.
I. Improve the Huffman Coding implementation with reference to section 6.2.3 (This completes algorithm one).
J. Implement LZ77 to run on data output from the 2D RLE algorithm.
K. Alter the existing Huffman Coding algorithm to run on the output of LZ77 and save to file (This completes algorithm two).
L. Implement Burrows-Wheeler (BW) transformation to run on data output from the 2D RLE algorithm.
M. Alter the existing Deflate code to run on the output of BW and save to file (This completes algorithm three).
N. Evaluate the algorithms' performances noting any inefficiencies.
O. Allow time to understand, fix and improve upon discovered inefficiencies.
P. Research C++ UI libraries.
Q. Create a UI that allows users to choose between the three algorithms to compress, decompress, and view images.
R. Evaluate the performance of the algorithms against one another, determining the best, and determining the categories of image that 2D RLE outperforms existing solutions for.
S. Write the dissertation.
T. Create a demonstration video and presentation.

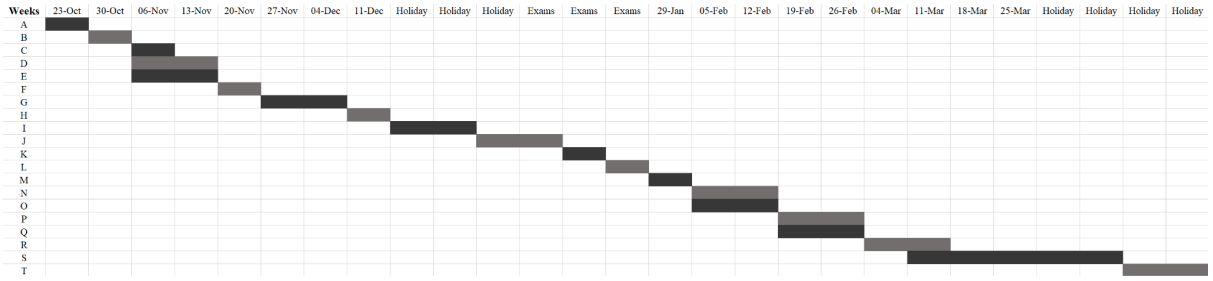


Figure 21

Throughout December work unrelated to this project took longer than expected to complete and to accommodate for this, two revisions to the project plan were made. Rather than implement Deflate compression from scratch ‘ZLib’ was used, ‘ZLib’ is an open-source library which is used in PNG (Gailly and Adler, 2024), and as such it was thought to be an acceptable way to save development time. As the project is primarily a research project the second revision to the work plan was to focus less on UI and instead have the user interact via the console. These two changes made it possible to successfully implement, test, and analyse all three desired compression pipelines and achieve all key objectives of the project described in section 1.2.

Initially, the project had no intention of implementing lossy compression, however, due to the easy implementation of a 2D RLE tolerance value to accept similar colours into the evaluating rectangle, lossy compression could be introduced with little disruption to the project plan. Lossy compression has since provided valuable results, as can be seen in section 7.2.2, and has become an integral part of 2D RLE compression. This was made possible due to the clarity of the codebase, the organisation of the project, and the flexibility of the development methodology.

The revised project plan provided a clear series of steps needed to progress towards the project aims and objectives, whilst allowing time for testing and subsequent results to guide iterative development at each stage.

8.2 Contributions and Reflections

8.2.1 Personal Reflection

The intent of this project as expressed in section 3.1 was to develop a compression algorithm in conjunction with existing encoding methods to achieve a compression ratio better than PNG for certain classifications of image. With regard to this statement, the project has objectively been successful. Each implementation of each algorithm has been tested on 20,212 images spread over 12 distinct datasets and results, as can be seen throughout section 7, conclude that for low-entropy images such as Scanned Documents and Binary Masks 2D RLE outperforms PNG whilst performing similarly to PNG for high-entropy datasets such as Textures and Landscape images. Lossy 2D RLE has also been shown to compress both high and low-entropy images further than PNG whilst retaining acceptable visual quality.

I am extremely happy with the outcome and results of the project. As mentioned in section 8.1, the organisation of the project and flexibility of the development methodology allowed for the iterative testing and development of the implemented algorithms, this has been invaluable. The clear direction provided by the project plans also allowed for constant progress to be made. Upon reflection, I should have anticipated the amount of work I would be required to do outside of the Dissertation over the Christmas break. This mistake caused me to be overly ambitious about the UI I could create in the given time.

If I were to develop the project further I would implement the following:

- The evaluation of entropy upon an image being input so that it automatically uses the most optimal compression algorithm.
- A separate app used exclusively for viewing images that can be linked to the Windows Registry so that files can be opened from Windows Explorer without the need to use the console.
- A checksum mechanism to verify the integrity of compressed files.

8.2.2 Law, Social, Ethical, and Professional Issues

The method of area encoding, specifically the use of queues, so that only extension-colour pairs have to be stored is patentable, as demonstrated by the 1999 patent (Wood and Richardson, 1999) of a method outlined in section 2.2. I do not plan on applying for any patents and I do not plan to commercially exploit the work outlined in this document.

The project's intentions have been realised and it could be argued that there is an ethical responsibility to make it open source. A reduction in the storage requirements of digital images can help take action towards the United Nations Sustainable Development Goals (United Nations General Assembly, 2015). Mitigating the impact that the manufacturing of storage media has on the climate addresses SDG 13: Climate Action, and making information easier to access through the reduction of bandwidth required to transmit images addressed SDG 10: Reduced Inequalities.

None of the work involved in this project uses human participants or data subjects. The only data being used are image datasets and as some have unknown licences, rather than distribute the images, links to where they were initially sourced can be found attached (Appendix. 2). Only two libraries not belonging to the C++ Standard Library are being distributed via the project source code these are ‘CImg’ and ‘ZLib’, whose licences are CeCILL-C and ZLib respectively. Both licences allow for redistribution as long as it is acknowledged that I did not author them, any changes that have been made by me have been addressed, and that the code for these libraries is made available. The project will not collect any user data. Therefore, this project has no ethical or data protection concerns.

9. Bibliography

1. Thomson, T.J., Angus, D., Dootson, P., 2020, '3.2 billion images and 720,000 hours of video are shared online daily. Can you sort real from fake?', Available at: https://theconversation.com/3-2-billion-images-and-720-000-hours-of-video-are-shared-online-daily-can-you-sort-real-from-fake-148630
2. National Archives and Records Administration, 2022, 'National Archives by the Numbers', Available at: https://www.archives.gov/about/info/national-archives-by-the-numbers
3. Adobe, 2023, 'Adobe Firefly expands globally, supports prompts in over 100 languages', Adobe, Available at: https://news.adobe.com/news/news-details/2023/Adobe-Firefly-Expands-Globally-Supports-Prompts-in-Over-100-Languages/default.aspx
4. Liesch, N., 2003, 'THE BMP FILE FORMAT', Available at: http://www.ece.ualberta.ca/~elliott/ee552/studentAppNotes/2003_w/misc/bmp_file_format/bmp_file_format.htm?source=post_page-----ed6fbbf1c54----------
5. Britannica, T. Editors of Encyclopaedia., 2023, 'GIF.' Encyclopaedia Britannica, Available at: https://www.britannica.com/technology/GIF
6. Roelofs, G., 1999, 'Chapter 9. Compression and Filtering' in 'PNG: The Definitive Guide', Available at: https://www.oreilly.com/library/view/png-the-definitive/9781565925427/18_chapter-09.html
7. Wu, P., Zhou, S., Wan, B., Fang, F., and Zhou, Sha., 2016, 'An Improved Two-Dimensional Run-Length Encoding Scheme and Its Application', International Journal of Signal Processing, Image Processing and Pattern Recognition, vol.9, no.4, pp.339-346. Available at: http://article.nadiapub.com/IJSIP/vol9_no4/30.pdf
8. Nuha, H., 2020, 'Lossless Text Image Compression using Two Dimensional Run Length Encoding', Available at: https://www.researchgate.net/publication/341125061_Lossless_Text_Image_Compression_using_Two_Dimensional_Run_Length_Encoding
9. Wolfram, S., 2002, 'Chapter 10: Processes of Perception and Analysis' in 'A New Kind of Science', Available at: https://www.wolframscience.com/nks/notes-10-5--2d-run-length-encoding/
10. Wood, K., and Richardson, T., 1999, '2D(area)-run length encoding', Available at: https://worldwide.espacenet.com/publicationDetails/biblio?FT=D&date=19990728&DB=EPODOC&CC=GB&NR=2333655A#
11. Huffman, A., 1952, 'A Method for the Construction of Minimum-Redundancy Codes' in 'Proceedings of the IRE', vol. 40, no. 9, pp. 1098-1101, Available at: https://doi.org/10.1109/JRPROC.1952.273898
12. Ziv, J., Lempel, A., 1977, 'A Universal Algorithm for Sequential Data Compression' in 'IEEE Transactions On Information Theory', vol. IT-23, no. 3, pp. 337-343, Available at: https://courses.cs.duke.edu/spring03/cps296.5/papers/ziv_lempel_1977_universal_algorithm.pdf
13. Katz, P., 1991, 'String Searcher, and Compressor Using Same', Available at: https://worldwide.espacenet.com/patent/search?q=pn%3DUS5051745A

14. PKWARE Inc., 2022 '.ZIP File Format Specification', section 4.1.3, available at: https://pkware.cachefly.net/webdocs/casestudies/APPNOTE.TXT
15. Burrows, M., Wheeler, D., 1994, 'A Block-sorting Lossless Data Compression Algorithm', Available at: https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html
16. AQA, 2020, 'Teaching guide: Run-length encoding (RLE)', Available at: https://filestore.aqa.org.uk/resources/computing/AQA-8525-TG-RLE.PDF
17. TinyPNG, Available at: https://tinypng.com/
18. Wang, Z., Bovic, A.C., Sheikh, H.R., Simoncelli, E.P., 2004, 'Image quality assessment: from error visibility to structural similarity', Available at: https://doi.org/10.1109/TIP.2003.819861
19. Raghunathan, V., n.d., 'ECE264: Huffman Coding', Advanced C Programming, Purdue University, Available at: https://engineering.purdue.edu/ece264/17au/hw/HW13?alt=huffman
20. OpenCV, 2023, 'OpenCV (Version 4.8.1)', Available at: https://opencv.org/releases/
21. Tschumperle, D., 2024, 'CImg: C++ Template Image Processing Library (Version 3.3.5)', Available at: https://cimg.eu/
22. Cristy, J., 2024, 'ImageMagick (Version 7.1.1-29)', Available at: https://imagemagick.org/index.php
23. Clark, J.A., 2024, 'pillow (Version 10.3.0)', Available at: https://pypi.org/project/pillow/
24. Van Der Walt, S., Schönberger, L., Nunez-Iglesias, J., Boulogne, F., Warner, J.D., Yager, N., Gouillart, E., Yu, T., and scikit-image contributors, 2014, 'scikit-image: Image processing in Python', PeerJ 2:e453, Available at: https://doi.org/10.7717/peerj.453
25. Morton, S., 2023, 'Natsort (Version 8.4.0)', Available at: https://pypi.org/project/natsort/
26. Shapkin, A., 2017, 'Visual Leak Detector (Version 2.5.1)', Available at: https://marketplace.visualstudio.com/items?itemName=ArkadyShapkin.VisualLeakDetectorforVisualC
27. Gailly, J., Adler, M., 2024, 'ZLib (Version 1.3.1)', Available at: https://zlib.net/
28. Flynn, J. R., Ward, S., Abich IV, J., Poole, D., 2013, 'Image Quality Assessment Using the SSIM and the Just Noticeable Difference Paradigm', Engineering Psychology and Cognitive Ergonomics. Understanding Human Cognition, pp.23-30, Available at: https://link.springer.com/chapter/10.1007/978-3-642-39360-0_3
29. United Nations General Assembly., 2015, 'Transforming our World: The 2030 Agenda for Sustainable Development'. https://sdgs.un.org/2030agenda