Zak Micallef
SM3986 - SMc30381

1st April 2019

Index

Introduction Page 3
Background of techniques used Page 4
Design and Implementation of classes Page 7
Evaluation of results Page 22
Discussion and conclusion Page 24

Introduction

The main idea of this assignment is so that the uses of the Huffman algorithm is used it various ways. Huffman coding was an algorithm developed by David A (HUFFMAN). This algorithm is design so that least bits are used to represent the highest frequency of the same symbol and the second least amount of bit for the most second most represented symbol and so on for all the data set. Thus lowing the amount of bits need to store the representation of information. Huffman is a lossless compression algorithm, this is because the data is not being altered in anyway or no data is being lost, rather the data is changed in how it is represented. Huffman coding considers that some pieces of information occur more often than others unlike it default representation.

The default representation normally considers all type of data to occur equally as much and is built inconsideration of representing the data in compatible formats, it being edible and displaying the variety of the whole data set in its domain. On the other hand when compressing with Huffman technique the data that it being represented in can no longer be read as common data but only represented from a dictionary that it has be created in context of the frequency of that data is being represented. This is made by first creating a table of the frequencies that a unit of data is repeated and then creating a binary tree. The result from the leaves of this binary tree will have prefix code representing the bits in inverse proportion of the frequency it occurred. With the results of the binary tree, the data can be traversed into the new data representation thus making the files size smaller, compression. Then can be traversed back into the normal representation this is the decompression of the file. Therefore not data is being losing but becoming smaller in size.

Compressing the data of course comes in cons. in this state the data is extremely restricted, the data no longer has the properties to be used, this is because when compressed the data is altered in a way that is unique and is tailor made for its own context rather than be ready to be interpreted. Thus such compression is done so that the data is ready to be stored or transferred.

An example of Huffman coding working well is when the occurrence of a few pieces of data is high, such like the english language where the space, a, e, i, o, and t are used a lot and where the rest fall short. In this project we use Huffman coding on text and on gray scale imaging and after it is done we can see the results of such an intelligent method of compression.

Background of techniques used

The technique of the application of Huffman encoding goes as following; A frequency table needed to be producing in order for the binary table to be constructed. This was done by traversing the set of data where compression needed to be applied and keeping tally of what unit of data turned up and inserted into an array. For example a string, “the three turned up” is going to be used. This will result in:

t	h	e	r	u	n	d	p	*space*
3	2	4	2	2	1	1	1	3

After this is produced into an array with a priority feature a similar priority queue empty is made.

e	t	*space*	h	r	u	n	d	p
4	3	3	2	2	2	1	1	1

Priority queue
Empty

The idea is now to use both arrays to concatenate the least frequent together. This will be later used to produce a binary tree. Thus first that need to be is the ‘d’ and the ‘p’

e	t	*space*	h	r	u	n
4	3	3	2	2	2	1

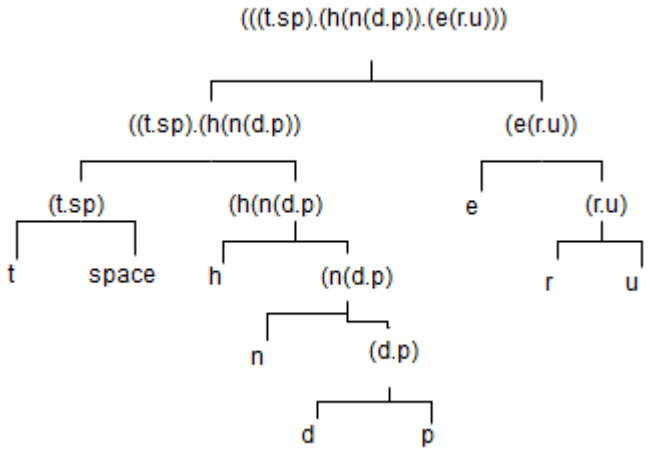
(d.p)
2

Then for the following step using both arrays to compare the frequencies is done repetitively until all units are concatenated into one.

Frequency table
Empty

(((t.sp).(h(n(d.p)).(e(r.u)))
19

Now using the concatenation of units a binary tree can be created. Due to the joining of two parts, the splitting nature when unwrapping makes it easy to build the tree. The resulting of the experiment becomes;



A binary tree is created so that a prefix code can be found on the ends of a branch, where the most frequency gets placed at the shallowest part of the three thus having the shortest binary word. Then using a breadth first search to find the binary value is applied counting each left child node with a ‘1’ bit value and ‘0’ bit value on the right node child. This would result in the following;

e	t	*space*	h	r	u	n	d	p
01	111	110	101	001	000	1001	10001	10000

And thus make the string represented in a shorter way rather than using something like ASCII code. This is the compressed result: 111 101 01 110 111 101 001 01 01 111 000 001 1001 01 10001 110 000 10000. Rather the 8-bits for each letter making it 144 bit instead of 54 bits.

When coming to compress a gray scale image on the other hand. There were way too many dissimilar variations of gray and thus the Huffman coding does not function. Huffman coding need repeated data pieces in order for it to be effective. To overcome this problem the finding of the different in grays will result in more common values thus making Huffman code efficiency once again. In order to get this done the image need to be made into a matrix. After this has been done a residual matrix is applied to the pixel to the left. Thus let’s say if A is a matrix then *R[i,j]=A[i,j]-A[i,j-1]* where R is the residual matrix. Below is an example;

The Image

The Matrix of gray values (not real values)

1	2	3	4	5	6
1	1	1	2	3	4
1	1	1	2	3	4
1	1	1	2	3	4
1	2	3	4	5	6

Residual matrix

1	1	1	1	1	1
1	0	0	1	1	1
1	0	0	1	1	1
1	0	0	1	1	1
1	1	1	1	1	1

As you can see, there are more in common values making it efficient to compress with Huffman coding. Thus the next step is to create a frequency table, then create the binary tree and find the compressed prefix code.

Decompression on the other hand need to be first decompressed and then the residual matrix needs to be reversed, *A[i,j]=R[i,j]+R[i,j-1]*

Design and Implementation of classes

`Element` class

The `Element` class is used to store string values and it frequency.

```
new Element = Element(String unit, int frequency)
```

It is to store an unit/Symbol with the amount of time it occurred this is used of the frequency table

Or;

```
new Element = Element(String unit, String bits)
```

Its is to store an unit/Symbol with a bit word that represents itself. This is used to store the results from searching the tree for the compressed prefixed code.

Also you can assign bit value using.

```
tableElement.setBits(String bits)
```

The `Element` class extends the `el` class due to some features shared among other classes

Constructor Summary

getUnit() Returns **String** value assigned to the **class**
getFrequency() Returns frequency as an **int** value
setBits(**String** b) Used to set the bit value of the **class**
getBits() Returns the bit assigned to the **class**

Method Summary

toStringWithFrequency() Returns a **String** of the unit and the frequency
toStringWithBits() Return a **String** of the unit and the bit word that represents the unit/symbol
toString() Return a **String** of the unit

`FrequencyTable` class

Used to create the Frequency table. On creating a frequency table it can either accept a string

```
new FrequencyTable = FrequencyTable(String str)
```

Or a two dimensional array for images or datasets that are store in such architecture

```
new FrequencyTable = FrequencyTable(int[][] residueMatrix)
```

On initialization these are stored in a priorityQueue. Using `isEmpty()`, `peek()` and `pop()` the `priorityQueue` the array can be accessed.

Constructor Summary

add(**Element** te) Adds element to priority queue
add(**String** unit, **int** frequency) Creates an object type **Element** and then adds it to the queue

Method Summary

isEmpty() Returns a **boolean** if the priority queue is empty
peek() Return a **Element** next in the priority queue
pop() Return a **Element** next in the priority queue and removes it
reNew() Repopulates the priority queue
print()

Print in console what the **FrequencyTable** stores
getTotalFrequency() Returns the total frequency of letters in the table

`Container` class

This class stores two `Element` objects. This is when two values are concatenated due to their frequencies, in order to create the binary tree. (*see the Background of techniques used*) A container implement the `el` class as it has similar features as the `Element` class. An Container normally contains two Elements.

```
Container container = new Container(el element1, el element2)
```

In some instances in may only store one. Only an element can be stored on its own.

```
Container container = Container(Element element)
```

Each Container has it own frequency that can be obtained using `getFrequency()`

Other Constructor Methods

getEl1() Return the first object stored in the container which is either an **Element** or a **Container** object
getEl2() Return the second object stored in the container which is either an **Element** or a **Container** object
getFrequency() Returns the total frequency of the **Elements** enclosed in the container

Method Summary

toString() Returns the container as a string using brackets and ‘.’

`el` interface

This is made to reduce ambiguity between object `Elements` and `FrequencyTable`

`ContainerTable` class

This class is the second priority queue to complete the huffman coding. (*see the Background of techniques used*) The `ContainerTable` is filled with containers and then flushed to get the Binary tree results.

```
ContainerTable containerTable = new ContainerTable(FrequencyTable frequencyTable)
```

The `ContainerTable` uses a recursive function so that it is populated accordingly. Each time there is the next container to made `next()` is called.

Other Constructor Methods

add(**el** e) Used to add containers to the table

Method Summary

isEmpty() Returns a **boolean** if the priority queue is empty
peek() Return a **Container** next in the priority queue
poll() Return a **Container** next in the priority queue and removes it
size() Return a size of the table as **int**
print() Print in the console the the **Containers**/**Elements** in the table
getTotalFrequency() Gets the total frequency of the table

`BinaryTree` class

This class does all the binary tree operations. This works hand in hand with the `Node` class.

To create a `BinaryTree` it requires a `FrequencyTable` this is done in ored to create the table depending on the frequencies.

```
BinaryTree binaryTree = new BinaryTree(FrequencyTable frequencyTable)
```

The `BinaryTree` class includes a breadth first search to find the value of its leaves `bfsTree()` and returns them into a `dictionary`.

Since the `containerTable` has flushing properties, a `containerTable` is populated and then later flushed to create the tree.

the breadth first search is done in a classical way using `Queue<Elements>.`

Also the class has also a draw feature that draws the tree that it produces

Method Summary

makeTree(**el** el) Sets the root and calls for the left and right nodes to be constructed
setLeft(**Node** p, **el** c) **Node** p is set as the parent node and **el** c which is a **Element** or a **Container** is stored in the current node. It then recursively calls for the left and right nodes to be constructed
setRight(**Node** p, **el** c) **Node** p is set as the parent node and **el** c which is a **Element** or a **Container** is stored in the current node. It then recursively calls for the left and right nodes to be constructed
bfsTree() First calls the **createContainerTable**() so that the the containers are assembled. Then calls **makeTree**() so that the tree is manufactured. After engaged in a breadth first search to find the Compressed values of the frequency table.
bsf()

This is part of **bfsTree()** recurse true the tree in order to complete the search
draw()/draw(**Graphics** g, **Node** node, **int** widthPos, **int** height, **int** widthDis) Uses swing and awt to draw the tree in a window. This is done recursively.

`Huffman` class

Is basically a class that combines the use of the two main classes `frequencyTable` and `binaryTree` so that huffman algorithm works in a classical way. A main method is implemented to display and run the huffman algorithm.

`Images` class

This class uses `frequencyTable` and `binaryTree` to adapt the huffman algorithm to work with images. A constructor is used to get the image.

```
Images image = new Images(String fileName)
```

Once the image is obtained it is turned to a grayscale image matrix stored in a two dimensional array. Once this is made the matrix is reduilzed using `residueMatrix(int[][] pixelMatrix)`

This class also does the reverse so that it can be decompressd. This means referring to the dictionary created by the tree, reverse the reduilzed matrix and recostructing the image.

Other Constructor Details

```
Images(String imageName)
This constructor is longer there the typical constructor. it was designed
to store most of the values for later use such as; pixelMatrix this stores
the image as a gray scale two dimensional array, the height, and the
width, the image reduilzed as residueMatrix, the frequency table
frequencyTable with the residueMatrix values, the binaryTree, and two two
dimensional arrays to store the compressed image compressedImage and
decompressed image decompressedImage.
```

Method Summary

```
readImageGrayScale(String fileName)
Reads the image converts it to grayscale by adding the red green and blue
together and dividing it by three and returning a two dimensional array of
the pixels
```

```
residueMatrix(int[][] pixelMatrix)
Return the pixel Matrix such that R[i,j]=A[i,j]-A[i,j-1] and R is return
```

```
getDecomprressedImage(String[][] compressedImage)
This two dimensional array gets the compressed bit words and converts it
to the residual matrix values in reference to the dictionary produced by
```

the breadth first search. This also reverses the residual matrix values in order to convert it back to the original image. The final step is to construct the image so that a full decompression is done.
saveImage() This save the bufferedimage on local storage as a jpeg.
displayImage() This displays the image in a window
getCompressedPixelMartix() This return a two dimensional array of all the compressed values in reference to the dictionary produced by the breadth first search.
printCompressedResults() This prints the dictionary produced by the breadth first search.

`Evaluation` class

This class has no getters and setter. It predominantly offers methods that work with evaluation of the compression algorithm. Most of this class perform basic maths in order to put the performance of the algorithm on some sort of metrics.

Method Summary

compressionRatio(**int** fileSizeAfter, **int** fileSizeBefore) This Calculates the compression ratio with the file size before and the file size after
compressionFactor(**int** fileSizeAfter, **int** fileSizeBefore) This Calculates the compression factor with the file size before and the file size after
savedPercentage(**int** fileSizeAfter, **int** fileSizeBefore) This Calculates the the percentage of how much the size got reduced by using the file size before and the file size after.
sizeOfFile(**String** path) Finds and returns the size of the image it bits.
sizeOfCompressionFromString(**String**[] bitChunks) Finds and returns the size of and array of strings that are assumed to be a binary representation, it bits.
sizeOfCompressionFromString(**String**[][] pictureBits) Finds and returns the size of a two dimensional array of strings that are assumed to be a binary representation, it bits.
entropy(**String**[][] pictureBits)/entropy(**int**[][] pixelMatrix) Finds the entropy of the image either expressed in a binary representation or in there pixel values.
FrequncyOfBitWord(**String** word,ArrayList<**String**> bitWords) This is to find the frequency of binary represented word, from a list of binary represented words.Then returns the number of times it shows up.
removeBitWord(**String** word, ArrayList<**String**> bitWords) This is to find the binary represented word, and remove it from a list of binary represented words if it occurs. Then returns the list.

```
avarageCodeLength(String[][] compressedImage)
Find the average length of binary represented words in a two dimensional
array.
```

`Main` class

This class performs the compression of an image using `Images` class and also does an evaluation using the `evaluation` class. This class truly show the performance of the compression that is going to be discussed later. In this class there is nothing to it that makes have function except the fact that it uses the classes as there supposed to in a main method.

Evaluation of results

In context of the assignment, overall the results were very successful. At first there were lots of nieve mistakes done. Such as the compression of the image was done without apply a residual matrix to the two arrays of pixels. This resulted in have the image compressed into a larger file. Secondly the program was design at first in a really inefficient method where there were lots of recursive functions getting the simplest of thing, this loaded up the ram and made thing slower than it should have. And finally there was a lot of clumsy code written where there was lots of unnecessary casting until a simple interface was introduced.

What makes the results so successful is not only an 8-bit BMP image of classically used photo of Lenna was compressed save about 40% of the original size but also how the code was written.

The idea of the code was that there were going to be two main parts. One part producing the frequency table, the second part so for handling the tree. This was decided was decided as both parts can be later used and changed to further change and improve the compression and not hard coded for the short term. Then both parts came together to form the image and string compressor.

In hindsight the code could have been written in more of an affect way where the tree handler would be extended to expect an array of values this would be the complete Huffman code and then extended once again accept and residual a matrix. Although this didn't bother me as it didn't affect the final outcome.

The outcome of the compression has a little issue when I come to keeping the saturation of the image. It seems to become more satiated then it should. Although no solutions found for such a small issue, I suspect that it happens when translating the image from BMP to data and the data to jpeg, not the compressor itself. This is disappointing because the compression is lossless but still information is being altered.

The affectivity of applying a residual matrix can be seen clearly from the significant drop of the entropy. We are interested in lowering the entropy as much as possible as this work is a theoretic limit for the Huffman code to compress. The idea is that the average code length of the file is lowered by the encoding of Huffman algorithm but can lower it then its entropy. In any case Huffman does not work at high levels of entropy and that why my algorithm didn't work before I realised that the residual matrix needed to be applied.

When working with compressing the classically used photo of Lenna formatted as an 8-bit BMP image the results we found where the following;

Results	
Size Before	2105792 bits
Size After	1258680 bits
Space saved	847112 bits
Space saved in per cent	40.22 %
Entropy of the image in grey scale	6.7591367
Entropy of residual matrix	4.033282
Average code length of compressed image	4.801483

As we can see for the results table is that although the image become 40.22 % smaller theoretically there could still been more compression. As we compare the entropy of the residual matrix and the average code length of compressed image we can find out that the Huffman code is 84% optimal. This leaves us with 7.66 % more space to compress theoretically. When it comes to comparing the outcome of my algorithm to high end products or work we can quickly realise that that the image can be compressed much more. This is because the algorithm is not designed of optimality but rather as an exercise of competence. Otherwise if we comparing level of compression to other file compression application you can quickly realize that there is room for improvement. A image compression algorithm that works with lossless compression converting images it to jpeg named JPEGmini clams to compress jpeg 3 to 6 times smaller than they already are (website optimization, 2011). On the other hand the compression algorithm designed almost makes it to halfway to 2 times smaller when compressing a BMP as a in grey scale image. This shows how much more effect compression can become. Such efficiency can be reach by taking full advantage of the nature of images, filtering, applying effective pixel predictive techniques and not being restricted into using Huffman in its classical format.

All in all the algorithm was built is basic but build in such a way that future work can extend the code in place to excrement in compressing in other ways. Also this Huffman algorithm is surly a good instruction into Entropy encoding.

Discussion and conclusion

In general we can conclude that such an algorithm is the basic to more efficient data handling. In general we can conclude that in this assignment we need to build a stock version of Huffman coding for strings and images. This is an interesting build as this only covers the surface of entropy encoding. This algorithm can and has be adapted for different situations. This can go deep and wide, as there are many alterations that can be applied to improve the efficiency and compression rate, tailoring for different file types and situations.

Entropy encoding are all lossless data compression that is some way share similarities with huffman coding. One example is arithmetic coding. Arithmetic coding adopts and improves the idea of extend the alphabet and encoding multiple symbols from the technique of extended Huffman. Arithmetic coding is the algorithm best used when there is a lot of repetition of the same few letters.

Huffman can also be built into a version of itself where it is built for transmitting data. In order to make huffman efficient for transmitting data, it need to be able to encode data in a stream and output that data in a stream. Thus the solution for this is to compresses and then build the binary tree one unit at a time and then output the values as soon as it is done. This is called Adaptive Huffman.

Fortunately true this exercise the basic have already been covered and that building an Arithmetic encoder or an Adaptive Huffman applied to a stream of data would be interesting and an appropriate increment in challenges so that skills in the Entropy encoding improve and become wiser in the domain.

When it comes to compressing images we can use Huffman coding for effective compression. This also include making an altered Huffman algorithm so that higher and more efficient compression can take place. Unlike string of data a lossy compression can be applied to images. This is because the quality of the photo may not be important and it may be that some data simply does not making a difference to the naked eye.

As mentioned above the human visual system has limitation of how much information in can take in. We can see this in colour. There is a limmate in how much of colour once can conceive. Interestingly we don't perceive all the colours equally as shown in figure 1.1. This clearly show that the eye is not so sensitive to lower wavelength colour such as blue and is way more sensitive to green and red. This perception of colour can be use to compress or get rid of unnecessary data.

In this assignment we only worked with grayscale images which made it easier to compress. The grayscale image processed in this assignment was no more than the average color depth of each pixel. In the case of producing color for each pixel we need 8 bits to representing each of the primary color. If we consider the three colour streams that will make 16,777,216 levels of colour making to large for hoffman to stay efficient thus using grayscale 256 levels of gray. In

order to compress image in colour successful the data of the image need to be organized in a different way. An example of this that is not too far from what we did in the assignment is to divide the colours into three images and compress them separately. This is an example not too far off from what we have made in the assignment and there are more effective ways compressing color.

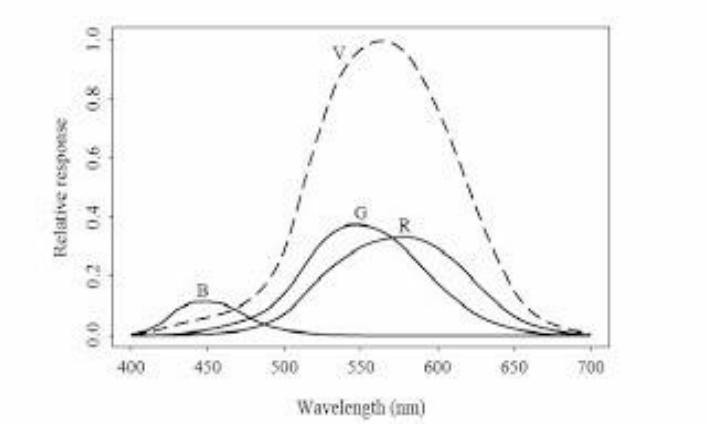


Figure 1.1 Response of eye towards different wavelength

The resolution needed is also a factor important into getting the most efficient system. Resolution is the amount of pixels used in the image. It is important to understand that the resolution of the image is what make the size of the image and that the high the resolution more of the pixels it need to traverse.

The format of the image is also important to consider when compressing the image. Let's say if we have a raw image that means there is no compression done thus all the data is embedded in the image and it is at the highest resolution. This mean that as mentioned before there is variation of colour in this image where it can't be see thus can be removed. This will not affect the overall quality of the image. Also a resolution of considerate size should assess and applied.

Doing such task already reduced the size of an image constable. To take it one step further filters and predictive pixel vaulting can make compressing even more compressible. As we mentioned in the evaluation, when applying a residual matrix the pixels became more alike thus Huffman worked more effectively. Filters and residual matrix in different patters can create even more alike values make the compression even more successful (Madhavan, 2007).

To conclude, such an assignment has shown that It covers basics in compressing both images and text. Although these are very restricted in use this can be a foundation of make new and more specific to compresses files of different types. This making us more of a versatile and wiser coder and to have insight of what compression is.

Bibliography

HUFFMAN, D. A. (n.d.). A Method for the Construction ofMinimum-Redundancy Codes*. A Method for the Construction ofMinimum-Redundancy Codes*. PROCEEDINGS OF THE I.R.E.

Madhavan. (2007, May 08). Image Coding Fundamentals. Retrieved April 01, 2019 from Video Code CS: http://videocodecs.blogspot.com/2007/05/image-coding-fundamentals_08.html

Seychell, D. B. (n.d.). Huffman Coding . Retrieved March 20, 2019 from St Martins Moodle: https://moodle.stmartins.edu/pluginfile.php/3464/mod_resource/content/2/Chapter%2005%20-%20Huffman%20coding.pdf

website optimization. (2011, August 29). JPEGmini: More Efficient Image Compression. Retrieved March 29, 2019 from Websiteoptimization: http://www.websiteoptimization.com/speed/tweak/jpegmini/