Investigating methods of text and image file compression using the Python programming language.

Department of Physics, Imperial College London

Monday, 26th of June 2017

Abstract

The Python programming language has been used to implement three different compression algorithms. Run-length encoding, Huffman and Lempel-Ziv were successfully produced and were used to compress text files. Bitmap images were also compressed using RLE. RLE achieved a compression rate of 115 for a line image. Huffman was best for literature text files less than 50 kb. Lempel-Ziv had the best compression rate of 2.25 for literature text files and was best for files greater than 50 kb.

Introduction

File compression is the process which reduces the size of a file for storage. There are various methods which are employed depending on the file type and size. The methods of compressing text (ASCII) and Bitmap files which are investigated in this report are Run-length encoding (RLE), Huffman and Lempel-Ziv (LZ). Reducing the size of files for storage frees up space and allows more to be stored. More importantly, it speeds up the process of transferring files through the internet and other networks which may be heavily congested. Data compression has been around since Morse code was developed in 1838. More advanced methods to be used with computers were developed later on. Huffman compression was introduced in 1951 by David Huffman, where he worked on optimising the method of using probability blocks to assign codewords to letters. This method was developed as technology improved, and lead to all data compression being done using the Huffman coding in the 1970s. During the late 1970s, Lempel and Ziv produced the Lempel-Ziv method which used pointer encoding [1].

Theory

Files are stored as binary in computers. Binary is the representation of data at the most basic level. This means the level at which the computer processor can read it. Binary is either a one or a zero representing on/off states, this is because logic gates operate on this principle. Computers store data in segments of 8 bits which is one byte. ASCII (American Standard Code for Information Interchange) is simply a format for text files. ASCII represents letters and symbols used in English text using 8 bits per character. There can be up to 256 characters represented using 8 bits per character. Text files are saved with the ".txt" extension. The number of characters that can be stored, N, is given by

$$N = 2^n, \tag{1}$$

where n is the number of bits per character.

In python strings are a type of class which are used to represent text files. There are many ways to manipulate strings. They can be modified letter by letter which means pieces of text can be split into lists of words or characters.

Bitmap and vector imaging are two methods to produce images. Bitmap uses pixels which are assigned a colour and position. Each pixel uses 24 bits for colour images or 8 bits for black and white. Vector images use mathematical formulas to produce curves and shapes [2]. Only Bitmap images are compressed in this investigation. Lossless compression is carried out and it means that no information is lost during the process of compression and decompression [3]. Lossy is the opposite, where some information is lost but the lost information is not as important[4].

Compression rate, C_r, is given by

$$C_r = \frac{S_u}{S_c}, \tag{2}$$

where S_u is the size of the uncompressed file and S_c is the size of the compressed file. It is the ratio which shows the level of compression achieved.

Method

For each method the algorithm is taken and then transformed into code in the Python programming language. Each algorithm is run using literature text files, and only run-length encoding is run with image files. Some images are taken from the internet and some are produced using Paint, for example line images are made on Paint. Graphs are then plotted to compare the compression rates and the time taken for compression. The text files are also found online and each one is different both in the genre and the actual text. This is so that it can be seen how each algorithm handles these differences.

Run-length encoding (RLE) is done by counting the amount of characters and storing this information as a string. The string has the characters and then the number of characters. To implement this approach a counter was set-up. While loops were used to count and check whether the character was repeated. After the character changes counting the next one starts. When complete the file is saved as a string. This is shown by figure 1. In order to decompress RLE, the compressed file is read and the program reads the letter and the corresponding number, n. Then it prints the letter n times. This was done using a while loop to cycle through the characters. Bitmap images were compressed by turning them into binary files which could then be treated as a list of characters. RLE compression was used for the bitmap images. Images are collected from the internet and saved as bitmap images using the ".bmp" extension. The types of images are different and include fractals, landscapes, text on backgrounds and human faces.

Run-length encoding

```
1 P = first letter
2 WHILE not end of input stream
3     C = next input character
4     set counter = 0
5     IF P=C
6         counter = counter + 1
7     ELSE
8         save C in dictionary with counter as value
9         counter = 0
10 END WHILE
11 output code for P
```

(a) Compression

```
1 P = first character (letter)
2 C = second character (frequency)
4 WHILE not end of input stream
5     output P×C
6     move to next two characters
7     P = next character to be printed
8     C = next character frequency
8 END WHILE
```

(b) Decompression

Figure 1: Shown above is the psuedo-code for the RLE Compression (a) and Decompression (b). Both compression and decompression use while loops to cycle through the files. Compression uses "if" statements whereas decompression does not. The decompression is simpler than the compression.

Huffman encoding is implemented by using a frequency tree with probability blocks. As the probability of a letter is proportional to its frequency, a frequency of each letter is tabled. The tree is produced by assigning the frequency as the probability of the character. Higher probabilities are placed at the top of the tree and lower probabilities towards the bottom of the tree. To build the Huffman tree, the lower frequency letters are used at the bottom. When adding another letter to the tree its probability must be the lowest of all the letters that can be added to the tree but are not in the tree yet. Each character is assigned a binary number, going left along a branch a zero is assigned, and going right along a branch, a one is assigned. To encode the character, the tree branch is followed until the letter is reached as shown in figure 3. This is the key part of the compression because the most repeated letters are represented with the least amount of bits. To implement this into Python code, the function "heapq" is used because it produces a binary tree analogous to the Huffman tree.

A separate function called "encodehuffman" is made to produce this tree. Then another function called "huffman" is produced which takes the tree and then outputs the binary representation of the word. When saving this in binary machine code, the Huffman encoding is split into 8 bit sequences because this is how data is stored in computers.

Huffman encoding and decoding

```
1 Create a list of all characters used
2 Produce a frequency count for all characters
3 Node for each character
4 Extract two nodes with the minimum frequency
5 New node with frequency equal to the sum of the two nodes frequencies
6 Repeat steps 4 and 5 until the tree contains only one node
7 Scan text and replace each character with Huffman tree binary value
```

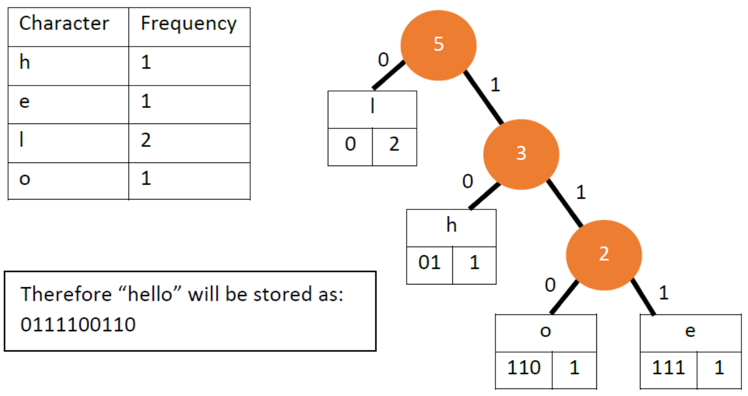
(a) Compression

```
1 Read binary
2 WHILE not end of binary
3     IF 0
4         left branch
5     IF 1
6         right branch
7     IF end of branch
8         translate binary to character
9 END WHILE
```

(b) Decompression

Figure 2: Shown above is the psuedo-code for the Huffman Compression (a) and Decompression (b).

Huffman example: encoding "hello"



Character	Frequency
h	1
e	1
l	2
o	1

Therefore “hello” will be stored as:
0111100110

Figure 3: Shown above is the encoding process for hello. The characters and their corresponding frequencies are tabled. The probability tree is formed and each character is encoded by following the branches. The higher probability characters are at the top of the tree which is the basis of the Huffman compression. "hello" is then saved as its corresponding binary representation from the Huffman tree.

In order to decompress using the Huffman method, the tree needs to be saved and sent alongside the binary encoding. Each binary digit is read and used as a map to find the corresponding character. The directions are left for a 0 and right for a 1. As the encoded binary in saved as 8 bit sequences, the 8 bit sequences are joined together to form a single string when decoding. This is because the Huffman encoding will not necessarily be in 8 bit sequences due to the nature of the Huffman tree. As shown in figure 2, each letter will have a different binary representation and different length. Therefore, when encoding the exact binary representation of the character will not fit exactly 8 bits. A while loop is used to cycle through the binary digits. As each binary digit is read the encoding will lead the computer to the end of a tree branch. At this point the program is told to stop reading binary, and to translate the scanned binary up until this point by using the Huffman tree. Once a translation to text occurs the program continues to scan the binary and produces the rest of the text using the same method.

Lempel-Ziv compression is based on pointer encoding. This means that it stores data and has a method to find it. Initially, a python dictionary of ASCII values up to 256 is produced. Input text is looped over. Each character is checked to see whether it is in the dictionary already. If the character is in the dictionary then it is not added but instead the next character is taken and the two are joined to form one set. This new set of characters is then checked against the dictionary. If it is not in the dictionary then it is added to the dictionary in a new slot outside the defined ASCII range. The previous character ASCII value is added to a list. For the first few words there will be little compression as there is little repetition. This means the first few values in the list will be actual ASCII defined values, which is an important point for the decompression. Once a pair of characters has been checked the process begins again for the next pair as shown in figure 3. At the end of this process the data is stored as binary. For this investigation 16 bits were used per character because after running a long piece of text such as "Hamlet" the dictionary size increases to around 44000. This mean that 16 bits would allow all these values to be represented. From equation 1, n=16 so N = 65536 which covers the range for large files. To save these 16 bit characters, they are written two bytes at a time using the "array" function.

Lempel-Ziv encoding and decoding

```
1 Initialize table with single character strings
2 P = first input character
3 WHILE not end of input stream
4     C = next input character
5     IF P + C is in the string table
6         P = P + C
7     ELSE
8         output the code for P
9         add P + C to the string table
10        P = C
11 END WHILE
12 output code for P
```

(a) Compression

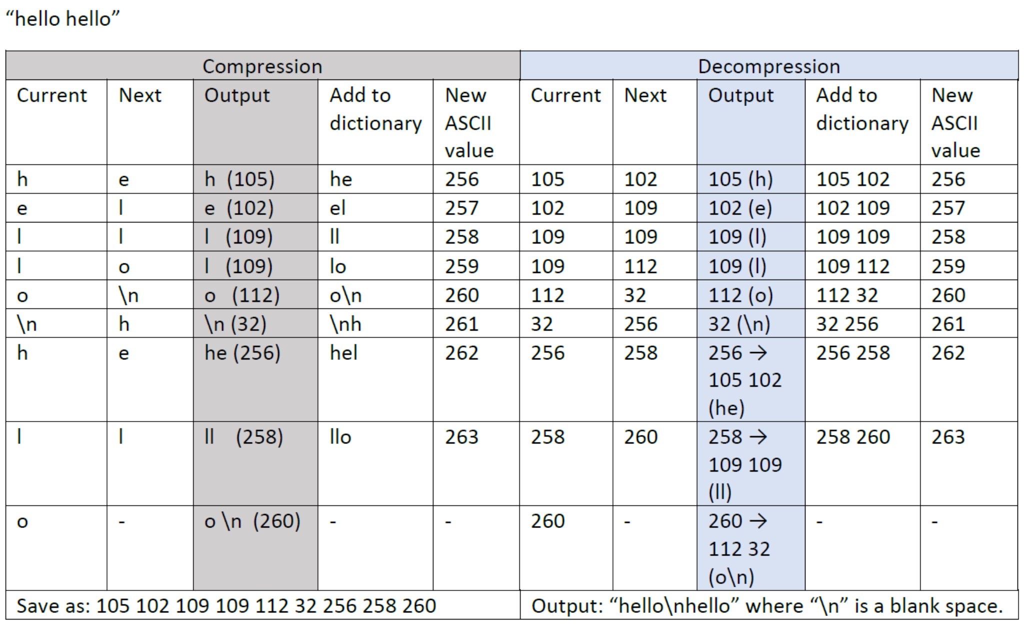
```
1 Initialize table with single character strings
2 OLD = first input code
3 output translation of OLD
4 WHILE not end of input stream
5     NEW = next input code
6     IF NEW is not in the string table
7         S = translation of OLD
8         S = S + C
9     ELSE
10        S = translation of NEW
11        output S
12        C = first character of S
13        OLD + C to the string table
14        OLD = NEW
15 END WHILE
```

(b) Decompression

Figure 4: Shown above is the psuedo-code for the LZ Compression (a) and Decompression (b).

To decompress using the Lempel-Ziv method, the same approach is taken as with the compression. There is no need to send the new dictionary along side the compressed file to decompress. When decompressing, an ASCII dictionary is produced however this time the key and values are flipped. Now the dictionary key is the ASCII value and the dictionary value is the character string. The decompression process is then implemented in the exact same way as with the compression but now it is run with numbers as the input. A potential problem with not sending the dictionary is that there will be undefined numbers as they are outside the ASCII range. This works without the compressed dictionary because the first few characters will be saved as defined ASCII values. So when the numbers are run through the code, the new dictionary formed will produce pairs in the dictionary which will be saved as the value outside the ASCII range. Therefore, when the code comes across the value which is outside the ASCII range it will be pointed to the value stored in the new dictionary which will translate to the defined ASCII values (see figure 5).

Lempel-Ziv example:



"hello hello"

Compression					Decompression				
Current	Next	Output	Add to dictionary	New ASCII value	Current	Next	Output	Add to dictionary	New ASCII value
h	e	h (105)	he	256	105	102	105 (h)	105 102	256
e	l	e (102)	el	257	102	109	102 (e)	102 109	257
l	l	l (109)	ll	258	109	109	109 (l)	109 109	258
l	o	l (109)	lo	259	109	112	109 (l)	109 112	259
o	\n	o (112)	o\n	260	112	32	112 (o)	112 32	260
\n	h	\n (32)	\nh	261	32	256	32 (\n)	32 256	261
h	e	he (256)	hel	262	256	258	256 → 105 102 (he)	256 258	262
l	l	ll (258)	llo	263	258	260	258 → 109 109 (ll)	258 260	263
o	-	o \n (260)	-	-	260	-	260 → 112 32 (o\n)	-	-
Save as: 105 102 109 109 112 32 256 258 260					Output: "hello\nhello" where "\n" is a blank space.				

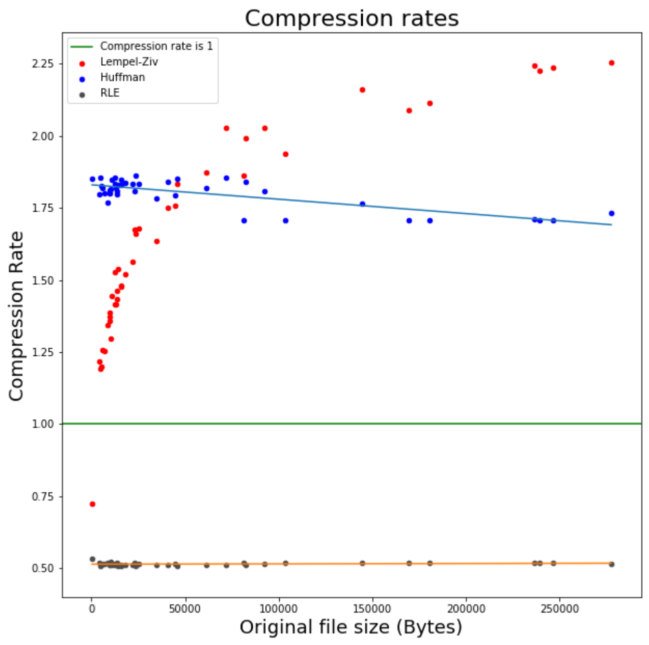
Figure 5: Shown above is the Lempel-Ziv encoding and decoding process for "hello hello".

Results and Discussion

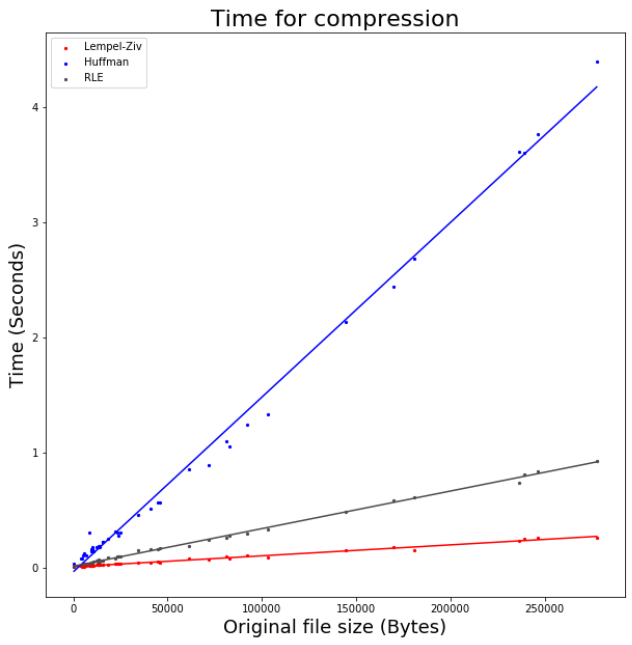
All three algorithms were implemented and did compress the size of files. Each algorithm preformed better than the others for different types and sizes of text. Run-length encoding (RLE) worked best for repeated characters, e.g "aaaaabbbbbcccc", but not for any literature. Huffman produced the best compression rates for small text which is not repeated. From figure 6(a), Huffman was shown to compress best when the file size was under 50 kb. Lempel-Ziv outperformed RLE compression for all literature. LZ did best when the file size exceeded 50 kb as shown in figure 6(a). Lempel-Ziv compression produced the highest compression rate of 2.25 for literature text files.

RLE increased the size of all literature texts. This is seen in figure 6(a) where the RLE points lie on a straight line at a compression rate of 0.5. This is because literature text does not have many repeated characters in a row. As RLE counts the number of consecutive character repeats and then stores the character with the number of its repeats, it will be expected to increase the file size. This number is expected to be around double the original size and so the compression rate should be 0.5 which it is. The reason for this is because on average characters are repeated once in long pieces of literature. Therefore RLE saves each character as the character and a "1". And so each letter is now stored using twice as much space so overall it double the size of the file for text.

Huffman compression maintained a stable compression rate across the large range of file sizes. The line for Huffman compression shown in figure 6(a) shows it decreases with increasing file size. The decrease is not very steep and nearly flat. This is expected because the Huffman algorithm is based on assigning probabilities to characters in the text. As for long pieces of text the frequency of characters reaches a equal level because of the nature of words. Words consist of characters, and are unchangeable entities which means they form the fundamental building blocks of sentences and so the whole text. As the amount of connectives, verbs and nouns used is proportionate in general literature the words used will eventually be repeated in equal amounts. This leads to the equal probability of finding a character for long pieces of text. Therefore, when building the probability tree for long pieces of text the difference in probabilities for each character decreases so the methods effectiveness decreases. Huffman was better than Lempel-Ziv for the smaller files because Lempel-Ziv relies on repetitions of combinations of characters in the text. There is little repetition for small files of literature text and so the Lempel-Ziv algorithm is not optimised for small files.



(a) Compression Rate

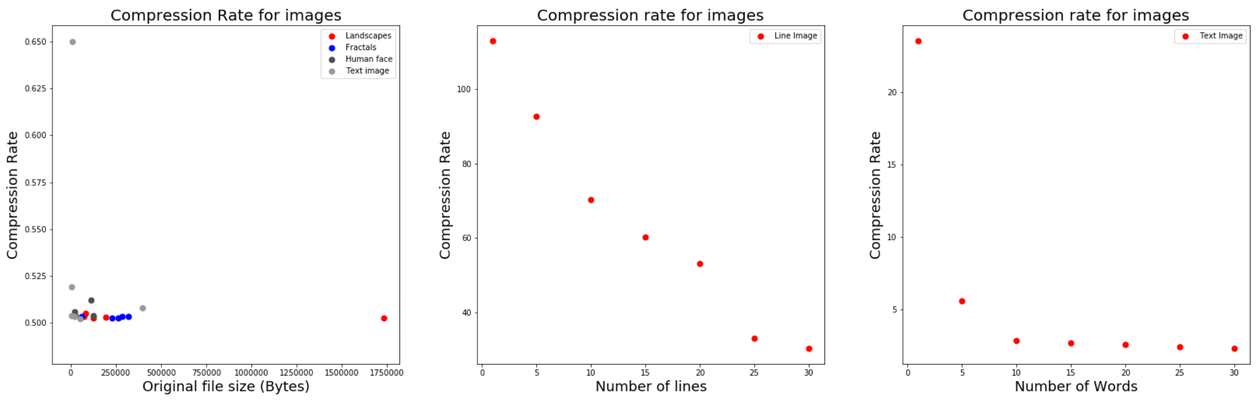


(b) Time

Figure 6: (a) shows the compression rate plotted against the original file size. All three algorithms are plotted. RLE (orange) is shown to have compression rate less than 1 and so it is not compressing but rather increasing the size of the file. Huffman (blue) is linear and slowly decreases and is the best for file sizes under 50 kb. Lempel-Ziv (red) follows an asymptotic curve. It is the best for files greater than 50 kb and does not compress small files(«1 kb). (b) shows the time taken for each compression. Huffman is the least time efficient as it takes more than four seconds to compress the largest file, compared to Lempel-Ziv and RLE which require less than one second.

For very small files («1 kb), LZ compression actually increases the size as the compression rate is less than 1. This is due to the lack of repetition in small files. On the other hand, for text which is more than a paragraph it does compress. In fact, Lempel-Ziv compression performed best for files above 50 kb because the amount of repetitions increasing as the file size increase for literature text. Repetition increase due to the same reason which was explained earlier. This means that the dictionary formed by the LZ algorithm becomes more effective as the file becomes larger and more repetitive. The LZ curve in figure 6(a) shows that it may reach a limit as the graphs gradient decreases at a lower rate after the 50 kb point. A limit must be reached because the dictionary will eventually include very large chunks of the text and so it will make little difference if another character is added to these large sets of characters.

The three algorithms also were timed and LZ was shown to be the fastest for all files sizes (figure 6(b)). All three lines for the time taken to compress followed a linear relationship with respect to the file size, and all of them increased the time to compress as the file size increased. Lempel-Ziv has the least steep gradient line and the change in time is around 0.1 seconds. This shows that LZ compression is very time efficient for a large range of files sizes. RLE is similar to LZ with respect to time and it has an increase in time of about 0.8 seconds. Huffman has the worst time efficiency as it takes the longest for all file sizes. LZ has a dynamic dictionary and does not keep a count of all characters in the text file like Huffman. This means it spends more of its time compressing rather than counting and organising. As Huffman compression requires a tree to be built and the frequency of all characters in the text to be counted it takes the longest to compress. The tree building process requires analysis of all characters and use of another algorithm to sort out the probabilities of each character. RLE is faster because it is a much simpler process.



(a) Mix of images **(b)** Line Images **(c)** Text Images

Figure 7: Run-length encoding of images from internet (a), images produced on paint (b) and (c). (a) shows that the images taken from the internet are not compressed but in fact are increased in size. There is one file which has a much higher compression rate than the average and it is one of the text images. (b) shows that RLE is compressing. The graph decreases as the number of lines increases. The highest compression ratio achieved is 115. (c) shows that text images are also compressed. The compression rate drops much faster than for the lines images. Reaches a constant compression value of just above 1.

Image bitmap compression with RLE was not successful with the images from the internet which used fractals, text background, human faces and landscapes. They all had a compression ratio under one. This was not expected because with bitmap images the pixels are repeated. And so RLE should be able to compress these images as it can simply count the number of pixels which are repeated, and then store their frequency and their value as it would with other files. Compression was not achieved because the images were high quality and complex. To achieve compression for these images lossy compression would have been the solution as it removes some pixels which will not be noticed by the human eye [3]. However, when ran for very simple line and text images compression was achieved. Figure 7 (b) shows that the highest achieved compression rate is for the smallest amount of lines. This is because there is only one line and it is the same colour so the pixels are identical, which is the optimal situation for RLE. The compression rate for the lines reduces slower than the words because the words are more complex. Every thing can be thought to be made up of small shapes which are made by small lines. So words are more complex than simple lines and harder to compress. Also, this explains the fact that the highest compression rate for text images is 29 which is less than that for lines.

RLEs limitation is that it is very simple and so cannot handle more complicated information structures. Huffman encoding is held back by the fact that for large text the frequency and so the probability of each character showing up becomes very similar. LZ encoding is not good for very small files as it relies on repetition of character combinations.

Conclusion

Lempel-Ziv encoding was the best for larger files (>50 kb) and it was the most time efficient algorithm. It took 0.2 seconds to complete the largest file compression. Huffman was the best for smaller files (<50 kb) and it was the least time efficient algorithm. It took 4 seconds to complete the largest file compression. RLE was best for simple images like lines and text, and is was nearly as time efficient as Lempel-Ziv. It took 0.9 seconds to compress the largest file. When attempting to compress complex images such as fractals and landscapes using RLE there was no compression. This was due to the images complexity. However, when simple images such as lines and text with a plain background were run through the RLE code there was compression. The highest compression ratio being 115 for a single line image. Improvements include combining RLE, Huffman and Lempel-Ziv to optimise the compression for all files. This would be implemented by creating a program to scan the file that need to be compressed and looking for distinct features which would be optimal for each algorithm. The encoding would be done with pointers so that for each compression part it is clear which decompression algorithm to use.

Bibliography

[1] Stephen Wolfram, A New Kind of Science, Notes for Chapter 10: Processes of Perception and Analysis, Section: Data Compression

[2]http://etc.usf.edu/techease/win/images/what-is-the-difference-between-bitmap-and-vector-images/

[3]Khalid Sayood, "Introduction to data compression", Third Eidition, page 179. ISBN 13: 978-0-12-620862-7.

[4]Khalid Sayood, "Introduction to data compression", Third Eidition, page 4. ISBN 13: 978-0-12-620862-7.

Appendix

Compression Code:

```
import time as time
import array as ar
import os
from heapq import heappush, heappop, heapify
import pickle
import numpy as np
def characters(file):
    with open(file, 'r') as f:
        return [character for line in f.readlines() for character in line]
def characterset(file): #lists all characters used in text
    characterset=set(characters(file))
    return(characterset)
def character_dictionary(file): # dictionary of characters
    characterdictionary={}
    for i in characterset(file):
        characterdictionary.update({i:characters(file).count(i)})
    return(characterdictionary)
def encodehuffman(symb2freq):#frequency count and builds tree
        """Huffman encode the given dict mapping symbols to weights"""
        heap = [[wt, [sym, ""]] for sym, wt in symb2freq.items()]
        heapify(heap)
        while len(heap) > 1:
            lo = heappop(heap)
            hi = heappop(heap)
            for pair in lo[1:]:
                pair[1] = '0' + pair[1]
            for pair in hi[1:]:
                pair[1] = '1' + pair[1]
            heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
        return sorted(heappop(heap)[1:], key=lambda p: (len(p[-1]), p))
def huffman(file): # uses tree to encode files
    time_initial=time.time()
    huffman_info=[]
    code=encodehuffman(character_dictionary(file))
    finaldict={}
    codedict={}
    j=0
    while j<len(code):
        codedict.update({code[j][0]:code[j][1]})
        j=j+1
    k=0
    while k<len(code):
        finaldict.update({code[k][1]:code[k][0]})
        k=k+1
    final=''
    for i in characters(file):
        final=final+str(codedict[i])
    finallist=[]
    for i in final:
        finallist.append(i)
    final=str(final)
    k=0
    anotherlist=[]
    while k<len(finallist)-8:
       anotherlist.append(str(''.join(finallist[k:k+8])))
       k=k+8
    else:
        number=(str(''.join(finallist[k:len(finallist)])))
        extrabits=8-len(number)
        while len(number)!=8:
            number+='0'
        anotherlist.append(number)
    with open('decompressionkey.pickle', 'wb') as handle:
        pickle.dump(finaldict, handle, protocol=pickle.HIGHEST_PROTOCOL)
    bin_array =ar.array('B')
    for i in anotherlist:
        bin_array.append(int(i,2))
    size=(len(bin_array))
    f = open('newfile.txt','wb')
    bin_array.tofile(f)
    f.close()
    f1=open('filesize.txt','w')
    f1.write((str(size)+","+str(extrabits)))
    f1.close()
    time_final=time.time()
    time_difference=time_final-time_initial
    size_huffman=os.path.getsize("newfile.txt")
    huffman_info.append(size_huffman)
    huffman_info.append(time_difference)
    return(huffman_info)
def runlengthcode(file):
    run_length_info=[]
    time_initial=time.time()
    characterlist=characters(file)
    lettersdictionary=[]
    i=0
    n=1
    while i<len(characterlist)-n:
        while i+n<len(characterlist) and characterlist[i]==characterlist[i+n]:
            n+=1
        else:
            lettersdictionary.append(str(len(characterlist[i:i+n])))
            lettersdictionary.append(characterlist[i])
            i=i+n
            n=1
    final=''.join(lettersdictionary)
    run_length_file = open("runlengthcompressed.txt","w")
    run_length_file.write(final)
    run_length_file.close()
    size_run_length=os.path.getsize("runlengthcompressed.txt")
    time_final=time.time()
    time_difference=time_final-time_initial
    run_length_info.append(size_run_length)
    run_length_info.append(time_difference)
    return(run_length_info)
def lempelziv(file):
    time_initial=time.time()
    lempelziv_info=[]
    dictionary_lempel_ziv = {}
    ascii = 256
    i = 0
    while i < ascii:
        dictionary_lempel_ziv.update({chr(i):i})
        i += 1
    current = ""
    text = ar.array('H')
    for i in ''.join(characters(file)):
        currentnext = current + i
        if currentnext in dictionary_lempel_ziv:
            current = currentnext
        elif currentnext not in dictionary_lempel_ziv:
            text.append(dictionary_lempel_ziv[current])
            dictionary_lempel_ziv[currentnext] = ascii
            ascii = ascii + 1
            current = i
    text.append(dictionary_lempel_ziv[current])
    lempel_ziv_file = open('compressed_text.txt','wb')
    text.tofile(lempel_ziv_file)
    lempel_ziv_file.close()
    size_lempel_ziv=os.path.getsize("compressed_text.txt")
    time_final=time.time()
    time_difference=time_final-time_initial
    lempelziv_info.append(size_lempel_ziv)
    lempelziv_info.append(time_difference)
    return(lempelziv_info)
#IMAGES
import time as time
import os
def brl(file): # contains code for compression of images
    time_initial=time.time()
    ddict=bytearray()
    bufsize = 255
    f = open(file, 'rb')
    buf = bytearray(f.read(bufsize))
    while len(buf):
        i=0
        n=1
        while i<len(buf)-n:
            while i+n<len(buf) and buf[i]==buf[i+n]:
                n+=1
            else:
                ddict.append(len(buf[i:i+n]))
                ddict.append(buf[i])
                if i+n == len(buf) - 1:
                    ddict.append(1)
                    ddict.append(buf[i+n])
                i=i+n
                n=1
        buf = bytearray(f.read(bufsize))
    run_length_file = open("brlcmp.bin","wb")
    #print(ddict)
    run_length_file.write(ddict)
    run_length_file.close()

    org_size=os.path.getsize(file)
    size_run_length=os.path.getsize("brlcmp.bin")
    time_final=time.time()
    time_difference=time_final-time_initial

    run_length_info = {}
    run_length_info['original size']    = org_size
    run_length_info['compressed size']  = size_run_length
    run_length_info['compression rate'] = org_size / size_run_length
    run_length_info['time difference']  = time_difference
    print(run_length_info)
```

Decompression:

```
#HUFFMAN
import time as time
import pickle as pickle
import array as ar
time_initial=time.time()
f=open('newfile.txt','rb')
ra=ar.array('B')
sizefile=open('filesize.txt','r+')
sizefileline=sizefile.readline()
size=''
while len(sizefileline)!=0:
    for i in sizefileline:
        size=size+str(i)
    sizefileline=sizefile.readline()
size=size.split(',')
sizeoffile=int(size[0])
extrabits=int(size[1])
sizeoffile=int(size[0])
ra.fromfile(f,int(sizeoffile))
result=[]
for c in ra:
    result.append('{0:08b}'.format(c))
binarycode=result
result=''.join(result)
with open('decompressionkey.pickle', 'rb') as handle:
    b = pickle.load(handle)
decompressionkey=b
binary={}
for i in decompressionkey.keys():
    binary.update({i:len(i)})
maxlength=max(binary.values())
minlength=min(binary.values())
finaltext=''
p=0
k=minlength
while p+k<=len(result)-extrabits:
    if result[p:p+k] not in binary:
#        print('result is ', result[p:p+k])
        k+=1
    else:
        finaltext+=str(decompressionkey[result[p:p+k]])
        p=p+k
        k=minlength
file=open("decompressed_huffman.txt","w")
file.write(finaltext)
file.close()
```

```
#RLE
file=open("testfile.txt","r+")
mystring=file.readline(100)
mystringlist=[]
for i in mystring:
    mystringlist.append(i)
letters=[]
numbers=[]
combinations={}
i=0
n=1
while i<len(mystring):
     try:
        ''.join(mystringlist[i:i+n])==int(''.join(mystringlist[i:i+n]))
        n+=1
     except ValueError:
        combinations.update({''.join(mystringlist[i:i+n-1]):mystringlist[i+n-1]})
        i+=n
        n=1
decompression=[]
for key in combinations:
    y=int(key)
    while y>0:
        decompression.append(combinations[key])
        y=y-1
decompression=''.join(decompression)
print(decompression)
#Lempel-Ziv
import array as ar
from cStringIO import StringIO
f=open('compressed_text.txt','rb')
ra=ar.array('H')
while True:
    try:
        ra.fromfile(f,100)
    except EOFError:
        break
text =ra.tolist()
dictionary = {}
ascii = 256
i = 0
while i < ascii:
    dictionary.update({i:chr(i)})
    i += 1
decompressed = StringIO()
w = chr(text.pop(0))
decompressed.write(w)
for k in text:
    if k in dictionary:
        entry = dictionary[k]
    elif k == ascii:
        entry = w + w[0]
    decompressed.write(entry)
    dictionary[ascii] = w + entry[0]
    ascii += 1
    w = entry
file = open("checkthis.txt","w")
file.write(decompressed.getvalue())
file.close()
#IMAGES
import time as time
time_initial=time.time()
file=open("brlcmp.bin","rb")
```

```
decompression=bytearray()
buff=bytearray(file.read(1024))
while len(buff)!=0:
    i=0
    n=1
    while i<len(buff)-1:
        counter=int(buff[i])
        while counter!=0:
           decompression.append(buff[i+1])
           counter=counter-1
        i+=2
    buff=bytearray(file.read(255))
file=open("decompressed.bmp","wb")
file.write(decompression)
file.close()
```