Denmark Technical University (DTU)
34240 – Data Compression

Applying the Lempel-Ziv-Welsh Lossless Compression Algorithm

Anit Gandhi
Student Number: s136203
May 19, 2014

Abstract

As the amount of data stored on the world's computers increases exponentially, the need for efficient data compression algorithms becomes increasingly apparent and important. In an effort to explore the real-world aspects of this, I sought to apply the existing LZW algorithm to bitmap (BMP) images.

In this report, I explore my findings regarding the implementation, quality, compression ratios, and speed of the LZW algorithm as it was applied to text, binary files, and images.

Table of Contents

Introduction

The amount of data in the world's computers is increasing exponentially every minute. Even disregarding the falling costs of storage media, it can be argued that the need for an all-purpose compression algorithm is stronger than ever.

For the course project, I chose to implement a universal, lossless, dictionary-based encoder and decoder known as the Lempel-Ziv-Welsh algorithm. Terry Welch published a modified scheme, based on Lempel and Ziv's famous "LZ" family of lossless compression schemes. This "LZW" scheme has one main distinction: the dictionary is not sent with the encoded data. In this lies the advantage of using LZW in computing environments. Already, it has been implemented for use in TIFF and GIF image storage formats, and the unix command line utility *compress*.

I wanted to do more than just write LZW. Instead of reinventing the wheel, I sought to create a new (relatively simple) scheme that would use LZW, but first apply some aspects of other simple schemes on special data types such as images and audio.

Original Plan

I had hoped to implement a few aspects of LZW in different forms.

1. Basic Text – LZW_Text
2. Generic byte data – LZW_Byte
3. Images: indexing and tolerance-based lossy compression - indexingTolerance
4. Images – pixels for the source alphabet in the LZW dictionary – LZW_BMP
5. A combination of 3 and 4
6. Audio – removing outlier frequencies that were not being used (perhaps with a tolerance), and those frequencies that humans cannot hear.
7. 12-bit codes
8. Variable width codes

Unfortunately, the last few steps regarding audio files and code lengths could not be implemented due to two reasons: time restrictions, and inability to code it properly. Based on my limited experience, I'm not yet able to code the custom-bit-width and variable width code lengths without more time.

Overview of the LZW Algorithm

Let's briefly look at an overall view of the LZW compression scheme. Here is an outline of the steps involved in the encoding and decoding processes. Some pseudo-code is included, as it would be applied to text based compression. Python code for LZW text compression can be seen in Appendix C, which is easy to read and understand given the simple syntax.

Encoding

```
1. Create the initial dictionary with the source alphabet. For example, for human languages,
   we can use the 256 ASCII characters.
2. Start with an empty string w.
3. For each character in the uncompressed input string (call the character c):
      a. Make a new string wc equal to the concatenation of w and c.
      b. If wc is in the dictionary:
            i. w = wc
      c. If it’s not found in the dictionary:
            i. Output the dictionary code for w
           ii. Add wc to the dictionary
          iii. w = c
      d. Go to the start of step 3
4. Output the dictionary code for w
```

Decoding

```
1. Create the initial dictionary, in the same manner as it was done for encoding
2. Start with strings w and result both with an initial value of the first input code, and entry is a
   new empty string
3. For each code in the compressed date (call it k):
      a. If k is in the dictionary:
            i. entry becomes the value of the dictionary at index k
      b. If it’s not in the dictionary:
            i. entry becomes w + w’s first letter
      c. Else
            i. Bad k value
      d. Add w + entry’s first letter to dictionary
      e. Increment the size of the dictionary
      f. w = entry
```

Advantages over LZ

There are a few key advantages of LZW over other lossless schemes such as LZ78, on which it was based.

The most important aspect is that the dictionary isn’t sent over. Instead, the initial dictionaries for the encoder and decoder are both the same. This is absolutely crucial, for if the two don’t match on their source alphabets and therefore dictionaries, there will be constant errors during runtime. The first character in the output and input are the same. Both the encoder and decoder create the same dictionary as it progresses, with the decoder always just one entry behind the encoder.

LZW has good throughput in embedded situations. Because the source code is relatively simple, it's easy to use in embedded/software-for-hardware applications with much higher processing speeds as compared to other schemes. Most importantly, it can be applied to many different situations, including text, raw binary data, images, etc.

Disadvantages

A clear negative aspect of LZW is that, many times, it will end up creating a massive dictionary, large portions of which might never be used in creating output codes. This can use up memory, which can be limited in embedded and other applications.

The major disadvantage here is the drastic running time that can occur when searching the dictionary for an existing entry. This is especially apparent in LZW_BMP, where pixels have to be compared individually. There are, however, a few tricks to speed this up. Namely, if we know we're only searching for one pixel, we can just get the index my calculating its integer value (using bit shifts to speed it up). If we're looking for "strings" of pixels, we can search only in the extended dictionary, and only those whose size match. By ignoring the rest, we quickly cut down on the total number of dictionary entries to search through. These were used in LZW_BMP.

Block Diagram

The generic block diagram (Transformation/Decorrelate → Quantization → Entropy Coding) does not strictly apply to lossless coding. However, a rough estimate can be given for LZW coding.

The transformation and decorrelation phase is similar to taking in the data and converting it to the "characters" that we are working with. This separates them from the surrounding characters, making them independent for the next phase.

Quantization is especially difficult for LZW, besides to say that each character has an equivalent quantification, or code. It is much clearer for lossy compression, such as the indexingTolerance implemented. Both the indexing aspect and tolerance aspect have a part in quantization. Since quantization can be seen as losing information when you go from a larger data set to a smaller. When we index the colors in an image, we are purposely disregarding the colors that aren't used. Further, with the tolerance, we are reducing the similarity between pixels when we check them against the index. Both of these lead contribute to a loss in quality.

Entropy coding's analog for LZW is simply checking the dictionary and finding the appropriate code.

Implementation

I chose to implement all of the source code in C and C++, as this is a language that is common to nearly every platform out there. Further, it is simple to port it to another language, such as Java on Android or Objective-C for iOS, for applications on mobile devices, and even scripting languages like Python, JavaScript, or PHP.

Further, minimum dependencies were used. For example, many existing implementations choose to use a map data structure for the dictionary. Instead, I chose to use a vector, which is a dynamic array. This kept the overhead of the data in memory to a minimal while still allowing the use of helpful functions and changing the array as needed. This was crucial for the dictionary, which expands as the program progresses.

For all aspects except LZW_BMP, the code lengths were fixed at 16 bits, using the *uint16_t* data type. For LZW_BMP, because of the 24 bits required to code the base dictionary, the code lengths were fixed at 32 bits, using *uint32_t* data type.

The following tests were run: subjective quality based on my interpretation (on lossy images only), size of the compressed file (compression ratio), entropy and code lengths (for the lossless parts), and running time. The entropy tests were done using a Python script from an Internet source, which can be seen in Appendix C.[12].

Steps Taken

I started by implementing LZW on only raw text – strings of characters. This made the initial dictionary a fixed size of 256, which is the number of characters in the ASCII standard.

Next, I made two “sub-programs” to help build up to indexing tolerance. These were indexing and tolerance, each coded independent of each other. These are not included, because they, together, formed indexingTolerance. The combined indexingTolerance program indexes the image’s pixels used. As it indexes them, it checks whether the current pixel is within a tolerance range of the index’s pixels. The first one it finds within the range is what is used in the indexed image. This creates a lossy coding, where the quality of the image is directly related to the tolerance set by the user.

From here, I shifted to implementing the LZW algorithm on generic binary data, looking at bytes as “characters.” This was a little more difficult to implement as compared to LZW on text, because of the use of vectors to replace strings, which have convenient functions already programmed. An important thing to note is that a txt file interpreted in binary form is basically the same as the txt file in text form, since ASCII characters are 1 byte each.

Finally, I implemented “native” LZW on 24-bit bitmap images using the EasyBMP C++ library (this made it easier to deal with the pixels themselves). This required the creation of a source alphabet for the initial dictionary, because pixels act as the "characters." I chose to fill the dictionary with all 16, 777, 216 possible 24-bit TrueColor pixels. While this is certainly excessive, it allows for some interesting effects and compression ratios. Unfortunately,

It's very important to note that, because of the sheer number of pixels stored in memory during execution, the RAM usage of LZW_BMP will remain around 500MB for the duration of its execution. This is usually regardless of the image size.

File Storage

The output file of the encoder was chosen as binary format, because this would be the easiest way of outputting different variable types together. For example, my program uses a mixture of integers, shorts (2 byte integers), chars, etc.

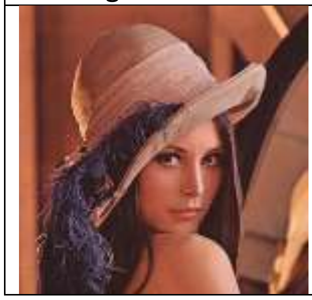
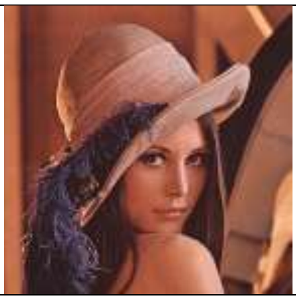
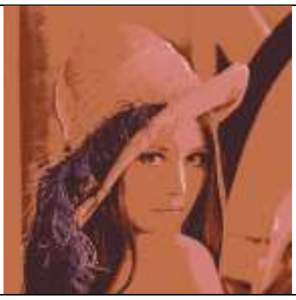
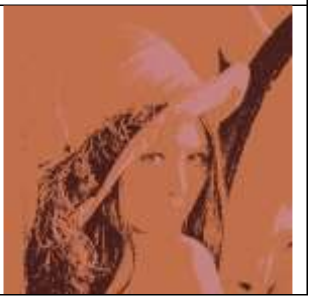
Also, using binary files have space benefits over other formats. Because the codes are saved continuously, one right behind another, there is no padding necessary in between codes. They can save any format, as long as you know how to read them back in. Since I was using vectors and arrays, and the native C commands for reading and writing binary files take in arrays, this was especially helpful.

Results – Quality

indexingTolerance

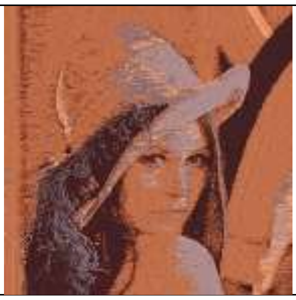
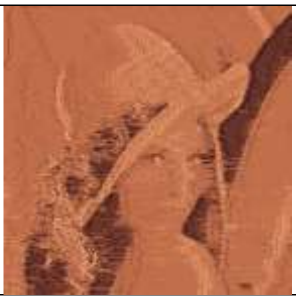
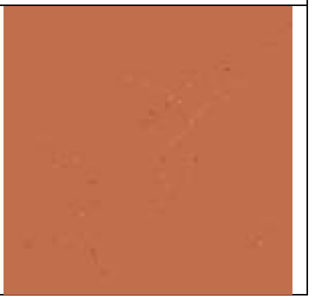
Here are some of the results of indexingTolerance, as done on the "lena.bmp" image, 512x512. As you can see, the image quality degrades sharply at tolerance=0.5. However, at t=0.05, the difference is almost imperceptible.

One interesting thing to note is that with t=1, the entire image almost becomes the same color: that of the first pixel in the image. This is predicted by the way the tolerance is used, since it looks at (1-t)*pixel to (1+t)*pixel. This means it looks from 0 to 2*pixel values, allowing it a large range to get its values from.

Original "Lena"	t = 0.05	t = 0.5	t = 0.7
			

<u>LZW_BMP with Tolerance</u>

The same thing as indexingTolerance was done on LZW_BMP with tolerance. It's pretty clear that the quality degrades rather quickly relative to the tolerance.

Original "Lena"	t = 0.25	t = 0.5	t = 0.75
			

The same as above applies to LZW_BMP for t=1. Many times, the entire image will, in fact, become just one color, depending on resolution and colors used in the original.

All results can be seen in Appendix B – Results, and the attached Excel file.

Results - Entropy

For text, entropies were calculated using the Shannon entropy formula, by first calculating the relative frequencies of the characters. For binary data and images, it was similar: instead of characters, the relative frequencies of the bytes was calculated, then the same formula was used. For finding the total size using this entropy, it first has to be rounded up, to mimic real world scenarios. Here is a table of some of the values found for text and binary input.

Source	Input Size (Bytes)	Entropy (bits/symbol)	Compressed Size at Entropy (bytes)
txt	500	4.2100	313
txt	2500	4.2100	1563
txt	309026	5.0900	231770
Guidelines.pdf	13167	6.9830	11521
Proposal.docx	14550	7.2770	14550
A10.jpg	80205	7.9770	842468

Notice the sharp increase in Shannon's entropy for binary files that aren't text. This stems from the fact that they don't have typical repetition and redundancy like language based text does, as seen above.

All results can be seen in Appendix B – Results, and the attached Excel file.

Results - Compression Ratios

Here is a sample of some compression ratios in the various programs.

Source	Method Used	Input Size (Bytes)	Compressed Size (Bytes)	Compression Ratio
txt	LZW_Text	500	626	0.7987
txt	LZW_Text	2500	2256	1.1082
txt	LZW_Text	309026	137836	2.2420
pdf	LZW_Byte	13167	17836	0.7382
docx	LZW_Byte	14550	21714	0.6701
jpg	LZW_Byte	80205	116246	0.6900
BMP	indexingTol. (t=0.01)	90054	110847	0.8124
BMP	indexingTol. (t=0.5)	3524630	2349091	1.5004
BMP	LZW_BMP (t=0.01)	90054	119356	0.75
BMP	LZW_BMP (t=0.5)	786486	41916	18.76

There are a few interesting things here. For text, LZW seems to require around 2000 characters of input text to achieve >1 compression ratios. Binary files seem to have a hard time getting compression at all (except for text files).

It's clear that even with a low tolerance, and high quality, there was a good compression ratio for indexingTolerance. As you'll see in the full results, this algorithm doesn't go above 1.5 for compression ratio, regardless the tolerance level.

LZW_BMP can have massive compression ratios based on tolerance, for a serious degradation of quality, of course. In the full results, it goes as high as 75.

All results can be seen in Appendix B – Results, and the attached Excel file.

Results - Speed

When running all the various versions of my code, it's clearly and quickly noticeable how little time it takes for the decoder to run in comparison to the encoder. There is one dominant reason for this: the decoder does not have to search and check if the current code is already in the dictionary.

Of course, the raw byte and text compression simply took longer based on the size of the file. Text compression happened very quickly, and any difference between different input sizes was barely perceptible.

Unfortunately, any time images were used for compression (based on the pixels), an extremely long time was needed to do the calculations. Anytime the dictionary needed to be checked to see if the code existed in it, a sequential/linear search algorithm was used. In the worst case (which was the case many times), the running time for checking the dictionary time would have been *O(n)*. Since the LZW_BMP base dictionary has over 16 million entries, this had very poor performance.

In the case of indexingTolerance, the speed largely depended on the tolerance input. If the tolerance was 1, then compression happened instantaneously. If it was near 0, it took upwards of 10 minutes for a 512x512 image. However, even if the tolerance was as low as 0.05, the time drastically came down to the order of a few minutes. A graph of these running times is given in Appendix B.

All results can be seen in Appendix B – Results, and the attached Excel file.

Applications

LZW as described above can be extremely useful in multimedia applications, especially over the Internet. This is because decoding takes significantly less time than encoding. As cloud computing increasingly becomes widely used, the need for less (therefore cheaper) computing power on the client-side becomes clear.

The cloud (server-side) computers have much more processing power, and can compress a file quickly. But, if the client has minimal power, this becomes useless in traditional schemes, as it would take a long time to decompress. However, with LZW, a server can send a compressed file, and a client-side script can quickly compress it with minimal resources.

Further, it can be implemented in any programming language very easily, so it’s easy to use on both server-side languages like PHP and client-side like JavaScript.

Finally, LZW can be implemented easily in embedded firmware situations. This might be useful for things such as microcontrollers that could collect data, compress it, and then send it over a network.

Platform Portability Issues

There is potential for this program to not be fully portable across platforms and architectures. That is, the source code cannot be compiled directly for another language or computing platform without taking into the account some of the differences between them.

For example, the endian-ness of the computer’s memory must be factored in. Desktop computers are little-Endian, so they store their memory contents “backwards,” while mobile devices may be big-Endian. While this may not seem like an issue, it presents a big issue when carrying the encoded files from one device to another. The program directly writes the proper memory contents to a binary output file. If this output file was created on a desktop, then decoded on a mobile device, the decoded output would likely make no sense. There may even be a run-time error involved.

Also, some different variable types were used that may behave differently when the program is compiled, depending on operating system, architecture, compiler, etc. However, some steps were taken to minimize this issue as much as possible, like using uint8_t instead of char.

Room for Improvement

Arguably the best thing that could have made my program better was the implementation of variable width codes.

The next major improvement would have been reduction in computing time. The largest bottleneck was the algorithm searching in the dictionary for existing entries. Some sort of data structure, such as a hash table or binary search tree, can be employed here to minimize the time required to search to *O(log n)* or *O(1)* running time, rather than the worst case *O(n)* that the current linear search requires. The issue with this is choosing or creating a hash function with absolute minimum collisions and uniformity.

The transparency (alpha) values can, effectively, be ignored. Since I explicitly dealt with 24-bit BMP images, which do not contain transparency values in the pixels, there is no point in running operations on them. This not only increases running time, but also the size of the compressed file.

Finally, for better ease of use, there can be some polishing done to the source code and interface when actually being used. This includes adding header files and removing duplicate code from the different sub-programs.

Conclusion

Although not everything was accomplished, this project gave a good foundation for future advancements in LZW and its applications in computing today. It showed that a simple lossless algorithm like LZW can be used in many different situations in a native form. While it performs especially well on text because of its repeating nature, LZW still has use for other data in the real world.

Appendix A – Source Code and Attached Files

Source Code (.cpp) Files: Please find the following documented files attached. The usage is explained when running the compiled file. Note that all files were compiled using g++ on a MinGW installation on Windows 8, 64 bit. Other platforms and compilers may have issues with these files. The Windows executables are also attached.

- LZW_Text.cpp
    - Simple string and character based text compression and decompression.
    - Input
        - Compression Mode: .txt file
        - Decompression Mode: binary file
    - Output
        - Compression Mode: binary file containing the codes
        - Decompression Mode: .txt file
    - Code length was fixed to 16 bits.

- LZW_Byte.cpp
    - LZW applied on the raw bytes, where the "characters" are the bytes. This can be used on any file.
    - Input and output files are always binary or interpreted as binary
    - Code length was fixed to 16 bits

- indexingTolerance.cpp
    - Indexes the pixels of an image. As it indexes, instead of doing every unique one, it checks to see if it's within a tolerance (user input).
    - Input
        - Compression Mode: BMP File
        - Decompression Mode: binary file containing the index and indexed image
    - Output
        - Compression Mode: binary file containing the index and indexed image
        - Decompression Mode: BMP file
    - The output can be used as the input to LZW_Byte for further compression.

- LZW_BMP.cpp
    - Runs the full LZW algorithm on a BMP file, by using a base dictionary of all 16,777,216 24-bit TrueColor pixels. Also applies tolerance, like above.
    - Input
        - Compression Mode: BMP file
        - Decompression Mode: binary file
    - Output
        - Compression Mode: binary file
        - Decompression Mode: BMP file

Appendix B – All Results

The rest of the results are in the attached Excel file named "Results.xlsx". Note some of the time and speed results may be inaccurate due to the method of measurement and amount of system resources being used by other running programs. This outliers or abnormalities are highlighted in red in the file.

LZW_Text

Source	Input Size (Bytes)	Entropy (bits/symbol)	Compressed Size at Entropy (bytes)	Code Length (Bytes)	Compressed Size (Bytes)	Compression Ratio	Compression Time (ms)	Decompression Time (ms)
lorem	500	4.2100	313	2	626	0.7987	178	15
lorem	1000	4.2500	625	2	1106	0.9042	41	15
lorem	1500	4.2400	938	2	1506	0.9960	60	26
lorem	2000	4.2100	1250	2	1898	1.0537	79	16
lorem	2500	4.2100	1563	2	2256	1.1082	101	34
world95	309026	5.0900	231770	2	137836	2.2420	289449	234

LZW_Byte

Source	Input Size (Bytes)	Entropy (bits/symbol)	Compressed Size at Entropy (bytes)	Code Length (Bytes)	Compressed Size (Bytes)	Compression Ratio	Compression Time (ms)	Decompression Time (ms)
pdf	13167	6.9830	11521	2	17836	0.7382	4055	44
docx	14550	7.2770	14550	2	21714	0.6701	5518	46
jpg	80205	7.9770	842468	2	116246	0.6900	190940	231
bmp	196662	7.5040	786486	2	208750	0.9421	663340	710
txt	309026	5.0860	231770	2	139078	2.2220	230540	269

Compression Ratio

This is best seen in the full file.

Speed

This is best seen in the full file.

Appendix C – Miscellaneous

Python Code for LZW text compression and decompression[1]

```
def compress(uncompressed):
    # Build the dictionary.
    dict_size = 256
    dictionary = dict((chr(i), chr(i)) for i in xrange(dict_size))
    w = ""
    result = []
    for c in uncompressed:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            # Add wc to the dictionary.
            dictionary[wc] = dict_size
            dict_size += 1
            w = c
    # Output the code for w.
    if w:
        result.append(dictionary[w])
    return result

def decompress(compressed):
    dict_size = 256
    dictionary = dict((chr(i), chr(i)) for i in xrange(dict_size))
    w = result = compressed.pop(0)
    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError('Bad compressed k: %s' % k)
        result += entry
        # Add w+entry[0] to the dictionary.
        dictionary[dict_size] = w + entry[0]
        dict_size += 1
        w = entry
    return result
```

Python Code for Finding the Entropy of any File[12]

This will give the same entropy for text files as if the formula was used directly on the string.

```
# read the whole file into a byte array
f = open(sys.argv[1], "rb")
byteArr = f.read()
f.close()
fileSize = len(byteArr)
print('File size in bytes:')
print(fileSize)

# calculate the frequency of each byte value in the file
freqList = []
for b in range(256):
        ctr = 0
        for byte in byteArr:
                if byte == b:
                        ctr += 1
        freqList.append(float(ctr) / fileSize)

# Shannon entropy
ent = 0.0
for freq in freqList:
        if freq > 0:
                ent += freq * math.log(freq, 2)
ent = -ent
print('Shannon entropy (min bits per byte-character):')
print(ent)
print('Min possible file size assuming rounded entropy:')
ent = math.ceil(ent)
print((ent * fileSize), 'in bits')
print((ent * fileSize) / 8, 'in bytes')
```

Resources Used

1. http://rosettacode.org/wiki/LZW_compression
2. http://www.cise.ufl.edu/~sahni/cop3530/powerpoint/lec18.ppt
3. http://urlstai.com/ELEG867/lz78.html
4. https://www.cs.duke.edu/csed/curious/compression/lzw.html
5. http://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
6. https://courses.cs.washington.edu/courses/cse326/05au/lectures/lecture19.pdf
7. https://sites.google.com/site/datacompressionguide/lzw
8. http://www.cplusplus.com/articles/iL18T05o/#Version6
9. http://marknelson.us/2011/11/08/lzw-revisited/
10. http://www.imagemagick.org/Usage/anim_opt/
11. http://code.activestate.com/recipes/577476-shannon-entropy-calculation/