Assignment 2 - Report

Armela Ligori
aligori18@epoka.edu.al
Epoka University - Theory of Computation (CEN 350)

Abstract

This assignment focuses in using a lossless compression algorithm such as LZ77 to achieve compression of some text files into a string of tuples. The tuples are then encoded into bits representation using Huffman Coding to achieve a much better compression. The compressed files are then decompressed with the goal of having no loss of information. Regarding the whole procedure, some statistics and comparisons are provided such as the number of encoded tuples and the compression ratio for each file. Based on these results, the efficiency and some drawbacks of this LZ77 implementation are discussed.

Keywords – LZ77, Compression, Decompression, Sliding Window, Huffman Encoding, Compression Ratio

1 Introduction

Data Compression Data compression is the art or science of representing information in a compact form [1]. It refers to the process of encoding information using fewer bits than the original representation, in attempt to save space and transmission time [2]. The Lempel–Ziv (LZ) algorithm family comprises of dictionary-based algorithms that are among the most popular for lossless data compression [3]. A lossless compression, involves no loss of information and data compressed can be recovered completely [1]. An important area of application for lossless compression is text compression. In text compression, it is crucial to keep all the information intact, otherwise the reconstructed data will be meaningless. Considering LZ algorithm family, this assignment focuses specifically on implementing and testing the Lampel-Ziv 77 compression algorithm on a series of text files, aiming a lossless result.

LZ77 Different from the other part of the family, LZ77 maintains a sliding window during compression. This is equivalent to a dictionary only when the entire data is intended to be decompressed. It achieves compression by replacing repeated occurrences of data with references to a single copy of that data existing earlier in the uncompressed data stream. A match is encoded by a triple called a length - distance - mismatch character pair.

2 Dataset

This experiment is performed individually on a set of files containing random generated protein sequences. The datasets are downloaded from Link, converted into text files and uploaded on repl.it. The initial goal was to compress files with a size of 50 MB, but due to the complexity of the implementation, the size of the files was reduced to a maximum size of 1 MB.

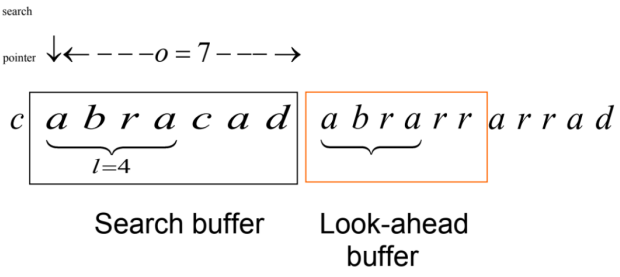

Figure 1: **LZ77** - Search buffer and look-ahead buffer

2.1 LZ77 Compression Algorithm

LZ77 compression algorithm comprises of two parts:

1. The encoder
2. The decoder

Encoding The encoder examines the input sequence through a sliding window which is partitioned into:

- Search buffer : contains portion of most recently encoded sequence - the search buffer is used as a dictionary to the recently encoded sequence.
- Look-ahead buffer : contains next portion of sequence to be encoded.

The process of compression can be divided in 3 steps:

1. Find the longest match of a string that starts at the current position with a pattern available in the search buffer.
2. Output a triple (d, l, d) where:
   - d: distance or offset, represents the number of positions that we would need to move backwards in order to find the start of the matching string.
   - l: length, represents the length of the match.
   - ch: mismatch character,represents the character that is found after the match.
3. Move the cursor length + 1 positions to the right.

Decoding Decoding/Decompression In LZ77 starts from the initial triple. Decoder keeps the same dictionary window as encoder. Each triple (d, l, ch) is examined and *l* characters are copied in the reconstructed string starting from

d positions behind the current index, then followed by the mismatch char *ch*.

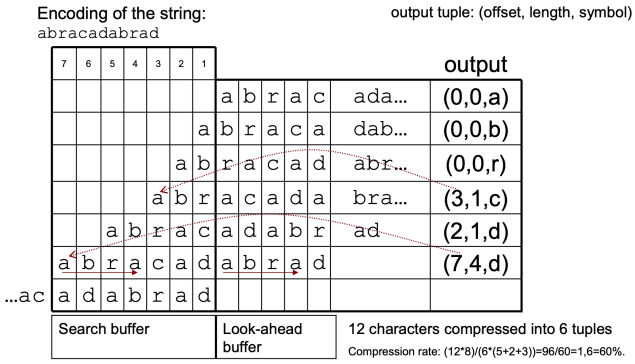

Figure 2: **LZ77**– Encoding flow illustration [4]

2.2 Python Implementation

This implementation is based on this Java LZ77 program contributed by R.G.Baldwin: Java Implementation. I modified the code using python. The program did not work for some specific cases such as when the searching from the window size expands into the look-ahead buffer, therefore I made changes to the encoding function based on what we have discussed in class. This is a straightforward implementation of LZ77 Compression Algorithm and therefore no optimization is taken into account. View in Repl.it

Listing 1: LZ77 Implementation

```
class LZ77:
    def __init__(self, rawData):
      self.rawData = rawData
      self.encodedtuples = []
      self.maxSearchWindow = 100
      self.lookAheadWindow = 50

    def encode(self):
      '''Proccess data one char at a time (or skip chars if matches found)'''
      charCnt = 0

      while(charCnt < len(self.rawData)):
        '''Get a reference position, length of the longest match'''
        distance, length = self.getLongestMatch(charCnt)
        '''Find character of mismatch'''
        mismatchChar = self.rawData[charCnt + length]
        '''Append tuple'''
        self.encodedtuples.append((distance,length,mismatchChar))
        '''Shift the window (length + 1) positions along'''
        charCnt += (length + 1)



    def getLongestMatch(self, charCnt):
        ''' Establish the size of the search window '''
        searchWindowSize = min(charCnt, self.maxSearchWindow)
        '''Establish the end of lookAheadWindow'''
        lookAheadWindowEnd = min(len(self.rawData) - charCnt,self.lookAheadWindow)

        '''Search the buffer window for a match to the next character in the lookAhead window.'''
        distance = 0
        longestLength = 0

        for d in range(1, searchWindowSize + 1):
          length = 0
          while(self.rawData[charCnt-d+length] == self.rawData[charCnt+length] and length <= lookAheadWindowEnd):
            length += 1
            if(length > longestLength):
              longestLength = length
              distance = d
            elif(d < distance and length == longestLength):
              distance = d
        return distance, longestLength


    def decode(self, tuples):
      reconstructedData = ''

      for distance, length, char in tuples:
        if (length == 0):
          # There was no match before this mismatch char, just concatenate it
          reconstructedData += char
        else:
          for _ in range(length):
            currentChar = reconstructedData[len(reconstructedData) - distance]
            reconstructedData += currentChar
          reconstructedData += char

      return reconstructedData


    def writeToFile(self, filename):
      with open(filename, 'w') as file:
        for d, l, ch in self.encodedtuples:
          file.write(str(d) + ' ' + str(l)+ ' ' + ch)
          file.write('\n')
```

2.3 PseudoCode

Encoding
while lookahead buffer is not empty **do**
  go backwards in search buffer and get a reference $(position, length)$ to longest match of the lookahead buffer
  if $length > 0$ **then**
    output$(position, length, nextsymbol)$
    shift the window length + 1 positions along
  else
    output(0,0, first symbol in the $lookAheadBuffer$
    shift the window 1 position along
  end if
end while

Algorithm 1: Encoding in LZ77

Decoding
Get the stored data
Start with an empty reconstructed text
while pointer is not at the end of the stored data **do**
  if distance and length is 0 **then**
    reconstructed text += mismatch char
  else
    copy the phrase which has the stored distance and length to the end of the reconstructed text
    reconstructed text += mismatch char
  end if
end while

Algorithm 2: Decoding in LZ77

Complexity Although the time complexity of LZ77 and LZSS encoding in practice is O(M) for a text of M characters, straightforward implementations are very slow [5]. The implementation used in here is a straightforward one, therefore it is really time consuming if the input text becomes very large. Encoding is the most time-consuming process. We

will see below in the experiments that the buffer sizes play a major role in the time-complexity. If few/no matches are found close together (in the limit of search buffer) then the skips of characters will not be as great as compared to the text length, and for each current character the search should start for each distance within the range of the search window. There are many factors to consider actually, but the time complexity of the encoding part can be as close to $O(n^2)$. On the other hand, decoding is fast since no searching is required, only the copying of characters as presented in the computed tuples.

3 Experiment and Results

Test Testing the correctness of this implementation

In some implementations of LZ77 that I have tested, even though the decompressed text was the same as the original, the (d, l, char) tuples were not as they would be if the algorithm was applied by handwriting. Much more tuples were computed than there was necessary, for a fixed window size. Therefore, before calculating the number of tuples, I will check the correctness of this implementation in computing the minimal required distance-length tuples. To do so, I chose a short string for which I know what tuples LZ77 should compute.

Text: *aacaacabcababac*
Tuples: (0,0,a) (1,1,c) (3,4,b) (3,3,a) (2,2,c)

```
----------------------LZ77Compression---------------------

Filename: test.txt

Starting Compression...

Tuples successfully created with LZ77 and stored into tuples/tupl
es_test.txt
[(0, 0, 'a'), (1, 1, 'c'), (3, 4, 'b'), (3, 3, 'a'), (2, 2, 'c')]

Number of encoded tuples:  5
```

Figure 3: **Testing Results**

This string was tested for a search window size of 6 and look-ahead buffer size of 3. The results are as expected. With this specific text it was also tested that the Search Window expands into the look-ahead Buffer, when characters continue to match and encoding reaches the look-ahead Buffer.

Task 1 - Calculating the number of computed tuples
When using **encode** method in LZ77, the computed tuples are stored into a list of tuples as an instance variable. To find the number of tuples, I simply found the length of this list using the python **len()** built-in function.

```
1    with open('input-files/' + filename + '.txt', 'r') as inputFile:
2        text = inputFile.read()
3        lz77 = LZ77(text)
4        lz77.encode()
5        tuplesFilename ='tuples/tuples_'+ filename+'.txt'
6        lz77.writeToFile(tuplesFilename)
7        print('Number of encoded tuples:',len(lz77.encodedtuples))
```

In order to compute the number of tuples efficently, it is important to consider:

1. The size of the Search Buffer and Look-Ahead Buffer
2. The size of the input file

I started with a text file of 50 MB but the program would crash due to the high complexity of the straightforward implementation. So I reduced the size to 1449983 Bytes = 1.3828 MB (in binary). Initially I chose *Search Window* = 400 and a *Look-ahead Window* = 200.

```
----------------------LZ77Compression---------------------

Filename: protein1.txt

Starting Compression...

Tuples successfully created with LZ77 and stored into tuples/tuples_p
rotein1.txt

Number of encoded tuples:  454786
```

Figure 4: **Tuples Calculation** – Text File of 1.3828 MB, Search-Window = 400, Look-Ahead Window = 200

The result is that 454786 tuples are computed from a file of 1449983 Bytes. However, it took approximately 3-4 minutes to arrive at this result. Assuming that the size of the dataset is the key factor in this computation time, I reduced again the size of the dataset to 895269 Bytes. Then I tried to compute the tuples for different values of Search window size and Look-ahead Buffer size. I also calculated the time needed for the compression into tuples using the python **time** module, by encapsulating the compress operation between the starting and finishing time.

```
1    start = time.time()
2    lz77 = LZ77(text)
3    lz77.encode()
4    print('Compression Time ',time.time() - start)
```

The results are stored in the table:

Filename: protein1.txt **Size:** 895269 B			
Search Window	Look-ahead Buffer Size	Number of Tuples	Compression Time
100	50	376355	36.053212
200	100	329633	83.79336
300	150	303270	85.26601
400	200	285292	95.52431
500	250	270692	129.68412

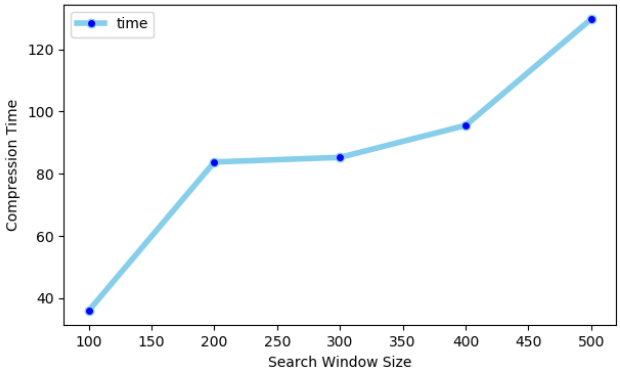

Figure 5: **Compression Time VS Window Size**

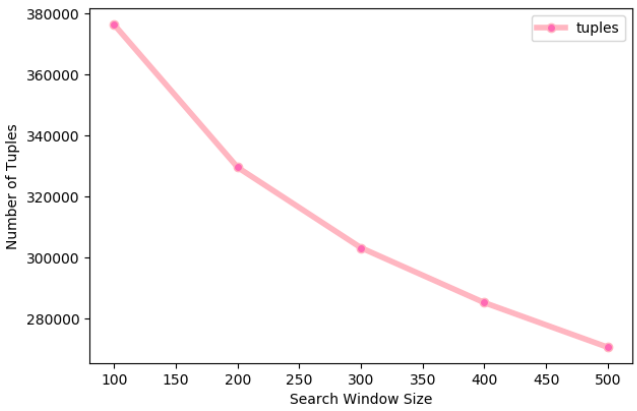

Figure 6: **Number of tuples VS Window Size**

Sizes of these buffers are parameters of the implementation and this is a drawback of LZ77. LZ77 assumes that matches will occur close together. Any match that recurs in a distance longer than that covered by the search window will not be considered. It is impossible to find a fixed search window size for which the algorithm will perform much better. Just for a buffer size of + 1 character, the number of tuples could be smaller.

As it can be seen from the table and the plots, increasing the size of the search-buffer and look-ahead buffer resulted in a decrease of the number of tuples. This is great because the smaller the number of computed tuples, the higher the compression rate. However, the greater the buffer sizes, the more time is required to finish the compression, because there are more characters inside the search window that need to be checked. Even though we are interested in saving space by compressing the data, trading space with time in this implementation cannot be considered an efficient solution. For an algorithm to be efficient, both space and time must be considered.

In order to have less computation time, for the next tasks I am going to consider a search window size of 100 B, and a look-ahead window size of 50 B. I will also reduce the size of the datasets to a maximum of 500000 B = 500 KB. These datasets are stored in files protein2.txt ... protein6.txt:

Filename	Size (B)	Number of Tuples
protein2.txt	201294	84847
protein3.txt	218366	92290
protein4.txt	418343	175880
protein5.txt	380506	157706
protein6.txt	104842	47067

Task 2 - Checking for lossless compression

To decompress the tuples, I simply read the tuples from the file and passed the list of tuples as an argument to **lz77.decode()**. Decompression returns a string that is the reconstructed text from the compressed data. This result is written into a file inside the directory *decompressed/*.

As mentioned even before, LZ77 is a lossless compression algorithm. This involves no loss of information and data compressed can be recovered. This could be checked by bare eye if the size of the text was too small. However, since the files have a size of KB to MB, to check that the reconstructed text is the same as the original one, I used a simple string comparison, and returned the result as a boolean:

```
----------------------LZ77Compression--------------------

Filename: protein6.txt

Starting Compression...
Tuples successfully created with LZ77 and stored into tuples/tupl
es_protein6.txt

Number of encoded tuples:  47067

Data is decompressed and stored into decompressed/decompressed_pr
otein6.txt

Is the reconstructed text the same as the original?  True
```

Figure 7: **Decompression**

The result of this comparison was True for all the input files. Thus, the reconstructed text is identical to the original one, no loss was detected.

Task 3 - Bits' representation, Compression Ratio

To represent the tuples in bits' representation, I used Huffman Encoding Algorithm. Huffman Coding is also a lossless data compression technique. It assigns variable-length codes to the input characters, based on the frequencies/probabilities of their occurrence. The python code for Huffman Algorithm is based on this website Code [6].

To find the code-words for each distinct tuple encoded by LZ77, the frequency for each tuple is calculated and stored in a dictionary. Using these freqeuencies, huffman encoding is applied and the prefix code tree is generated. After the prefix codes for each tuple are found, the list of tuples is encoded into a series of 01's. To store the result into a compressed file, a text file cannot be used, because '1' or '0' is considered as an 8 bit character. For this reason, the final compressed result is stored into a binary file as a byte array.

Compression Ratio To calculate the relative reduction in size of data, I needed to calculate the size of the original text file and the compressed binary file. Then, the data compression ratio is defined as the ratio between the uncompressed size and the compressed size:

$$Compression\ Ratio = \frac{Uncompressed\ Size}{Compressed\ Size}$$

In python the sizes of the files can be found directly with: **os.stat(file).st_size**. The results of compression ratio for each file when compressing with LZ77, search-buffer = 100, look-ahead buffer = 50, and then applying Huffman encoding, are stored in the table:

Filename	Original Size	Compressed Size	Compression Ratio
protein2.txt	201294 B	119256 B	1.6879150
protein3.txt	218366 B	129679 B	1.6838963
protein4.txt	418343 B	248685 B	1.6822204
protein5.txt	380506 B	222821 B	1.7076756
protein6.txt	104842 B	65007 B	1.6127801

As we can see from the results, the compression ratio of the files when using the same search buffer size is approximately the same, even though they have different sizes.

In order to compress and decompress the files, 4 steps were performed:

1. Text is compressed with LZ77 into a list of tuples
2. Tuples are encoded using Huffman codes
3. The compressed file is decoded into the list of tuples using Huffman Decoding
4. The list of tuples is decompressed into the original text using LZ77 decompression

Here are some more detailed statistics of the whole procedure in file protein1.txt which has a greater size than the other files, with a search-window = 100 and look-ahead buffer = 50:

```
----------------------LZ77Compression-----------------

Filename: protein1.txt

Starting Compression...
Tuples successfully created with LZ77 and stored into tuples/tup
les_protein1.txt

Starting Huffman Encoding...
Data is compressed and stored into compressed/compressed_protein
1.bin

Starting Decompression...
Data is decompressed and stored into decompressed/decompressed_p
rotein1.txt

Is the reconstructed text the same as the original?  True

--------------------------STATISTICS--------------------------

Size of input data:  895269  bytes =  7162152  bits
Number of encoded tuples in LZ77:  376355
Size of compressed file:  533375  bytes =  4267000  bits
Compression Ratio:  1.6784982423248185
```

Figure 8: **Compression Statistics** – protein1.txt

Checking compression ratio for different window-sizes
As explained in task 1, when increasing the buffer sizes, the number of computed tuples decreases. This implies, that the compression ratio will increase.

After applying Huffman encoding to the tuples, I wanted to visualize this behaviour of compression ratio on a file with a changing window size. To this end, I considered the file protein6.txt that has the smallest size among all the other files. The results are stored in the table and plotted below.

Filename: protein6.txt **Size:** 104842 B			
Search Window	Look-ahead Buffer Size	Compressed Size (B)	Compression Ratio
100	50	65007	1.6127801
200	100	63951	1.6394114
300	150	63214	1.6585250
400	200	62562	1.6758096
500	250	62025	1.6903184
1000	500	59953	1.7487365
2000	1000	57700	1.8170191
3000	1500	56409	1.8586041

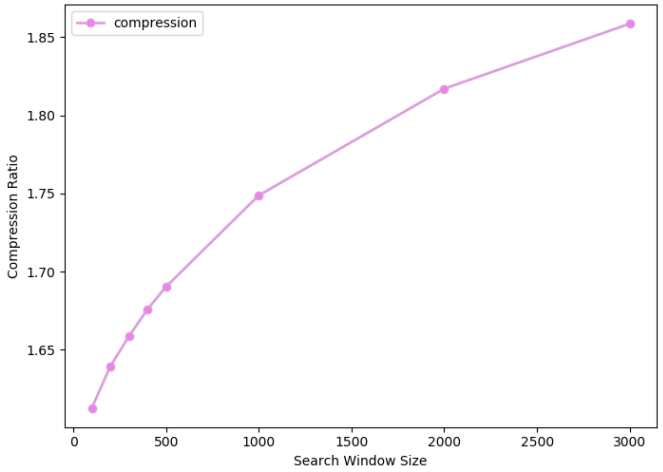

Figure 9: **Compression Ratio VS Window Size**

Task 4 - Proceeding in the reverse order

Performing the algorithm in the reverse order can be done simply by reversing the text after it is read from the file, and reversing it again whenever finishing decompressing the compressed result. String reversing in python can be achieved with a simple line of code:

```
1  text = text[::-1] # Start at the end of the string and end at ↩
       position 0, move with a step of -1 (one step backwards).
```

I repeated the same steps, for the same files by calculating the number of encoded tuples and the final compression ratio when proceeding in reverse order (Search Window = 100, Look-ahead Window = 50). Results:

Filename	Number of Tuples	Compressed Size	Compression Ratio
protein1.txt	376559	533319 B	1.6786744
protein2.txt	84940	119399 B	1.6858935
protein3.txt	92187	129553 B	1.6855341
protein4.txt	175847	248525 B	1.6833034
protein5.txt	157646	222642 B	1.7090486
protein6.txt	47065	65047 B	1.6117883

Task 5 - Comparing the results
The above results regarding the number of tuples and the compression ratio are merged in the tables below:

Number of tuples		
Filename	Direct Order	Reverse Order
protein1.txt	376355	376559
protein2.txt	84847	84940
protein3.txt	92290	92187
protein4.txt	175880	175847
protein5.txt	157706	157646
protein6.txt	47067	47065

Compression Ratio		
Filename	Direct Order	Reverse Order
protein1.txt	1.6784982	1.6786744
protein2.txt	1.6879150	1.6858935
protein3.txt	1.6838963	1.6855341
protein4.txt	1.6822204	1.6833034
protein5.txt	1.7076756	1.7090486
protein6.txt	1.6127801	1.6117883

As it can be seen, the number of tuples and hence the compression ratio, are approximate but not the same, when proceeding in direct vs reverse order. Ideally they would be the same, but in practice this is not the case. I am assuming this happens because the computed tuples are not identical especially at the beginning and at the end of the file. When proceeding in the normal direction, when we encounter some elements at the end of the file, we might have seen them before and therefore the number of chars that we skip might be much greater and less tuples may be computed. However, when reversing the text, these last characters of the file become the first, and there are no occurrences of those yet, therefore we don't have the same tuples computed as in the direct case. The same thing happens at the other end of the text (the beginning of the normal one, and the end of the reversed one). These kind of details, are reflected into the computed number of tuples and the compression ratio. However, the majority of the text (the 'middle' part) is compressed into the same tuples. This makes the number of tuples, not identical, but very approximate.

4 Conclusion

This experiment proved once again the lossless behaviour of LZ77 compression algorithm, by reconstructing texts that are identical to the uncompressed results. As a sliding-window technique, LZ77's efficiency depends a lot on the buffer sizes. As evidenced, selecting the size of the search buffer becomes a trade-off between the compression time and the compression ratio. A small search buffer will result in less time needed to complete the compression phase, but the resulting encoding will require more memory. On the other hand, a large search buffer will generally increase the time needed for compression, but it will be more efficient in terms of memory usage. This implementation used the straight-forward approach, but in real life applications of LZ77 such gzip, further optimizations are taken into account to make the compression as optimal as possible.

References

[1] K. Sayood, *Introduction to data compression*. Morgan Kaufmann, 2017.

[2] O. A. Mahdi, M. A. Mohammed, and A. J. Mohamed, "Implementing a novel approach an convert audio compression to text coding via hybrid technique," *International Journal of Computer Science Issues (IJCSI)*, vol. 9, no. 6, p. 53, 2012.

[3] R. Naqvi, R. Riaz, and F. Siddiqui, "Optimized rtl design and implementation of lzw algorithm for high bandwidth applications," *Electrical Review*, vol. 4, pp. 279–285, 2011.

[4] J. Müller, "Data compression, lz77." http://jens.jm-s.de/comp/LZ77-JensMueller.pdf, 2008.

[5] T. Bell, "Better opm/l text compression," *IEEE Transactions on Communications*, vol. 34, no. 12, pp. 1176–1182, 1986.

[6] B. Srivastava, "Huffman coding." https://github.com/bhrigu123/huffman-coding, 2016.