Andre Shapiro

Professor Tillquist

CSCI 411

04/04/21

LZ78 Research Paper

LZ78 is a lossless compression algorithm that is used in many aspects of our everyday life. A lossless compression algorithm means that no information is lost while the information is being compressed or decompressed. Abraham Lempel and Jacob Ziv created this algorithm in 1978 and it is still used in many current technologies such as ZIP and GIFs.

LZ78, in simple terms, is a dictionary that references itself repeatedly, allowing increasingly large data sets to reference each other. LZ78 uses a basic premise -- Dictionary[ ] = {i, append} -- where i is the index to a previously created entry and append is the data that will be appended to that previously referenced dictionary entry. Every time that a piece of data is entered into the dictionary, we search for a match and enter it as (index of the match, character being appended). Once we determine if the match has been found, we can take action accordingly. We can set the index to the last matching idex, otherwise we can create a new dictionary entry: Dictionary[new index] = (index of last match, character being appended). Once we have completed the dictionary, we will be able to decode it and make sense of all of the data that we were able to store.

While the encoding process is simple to understand, the decompression algorithm takes a little more thought. The idea still consists of creating a dictionary, but in this case we will be building it in the opposite direction. While decoding we will be reading two characters at a time. The first character we read is the code word, or the integer that references the dictionary entry that the current string will be based on. To be able to read this code word we will be converting the character into an 8 bit binary number and converting that into an 8 bit unsigned integer, then we will take the following character which we will append to the dictionary value for the code word we just decoded, Dictionary[ ] = {i, dictionary[code word] + character }, where I is the length + 1 of the dictionary and the string is the previously referenced entry plus the character we are appending. We will continue this process until we have read every character in the file, thus creating a string that is equal to text we first encoded.

example:

Data being compressed: “aababcabcjfkjfkabcjfk”

Dictionary we create vs how we encode it

(1 , a)   We encode as :(0 , a)

(2 , ab)   We encode as :(1 , b)

(3 , abc)   We encode as :(2 , c)

(4 , abcj)   We encode as :(3 , j)

(5 , f)   We encode as :(0 , f)

(6 , k)   We encode as :(0 , k)

(7 , j)   We encode as :(0 , j)

(8 , fk)   We encode as :(5 , k)

(9 , abcjf)   We encode as :(4 , f)

(10 , k)   We encode as :(0 , k)

Once we are finished encoding our compression output should look something like this:

“0a1b2c3j0f0k0j5k4f0k”

Or if we were to convert the integers to characters it would look like this:

“abcjfkjkfkthe”

The compression algorithm is easy to understand we will look through the dictionary until we find a combination of characters that has not been entered, if the letter we are checking is the first letter we write a 0 before it, otherwise we reference the dictionary entry of the previously found string.

The computer would then read the data : “abcjfkjkfkthe” or

“0a1b2c3j0f0k0j5k4f0k”

Information we read vs dictionary we create vs how we decode it:

We read as : (0 , a) we write it as : (1 , a) We decode it as : (+ a)

We read as : (1 , b) we write it as : (2 , ab) We decode it as : (a + b)

We read as : (2 , c) we write it as : (3 , abc) We decode it as : (ab + c)

We read as : (3 , j) we write it as : (4 , abcj) We decode it as : (abc + j)

We read as : (0 , f) we write it as : (5 , f) We decode it as : (+ f)

We read as : (0 , k) we write it as : (6 , k) We decode it as : (+ k)

We read as : (0 , j) we write it as : (7 , j) We decode it as : (+ j)

We read as : (5 , k) we write it as : (8 , fk) We decode it as : (f + k)

We read as : (4 , f) we write it as : (9 , abcjf) We decode it as : (abcj + f)

We read as : (0 , k) we write it as : (10 , k) We decode it as : (+ k)

Once we have decoded every character we are left with

a + ab + abc + abcj + f + k + j + fk + abcjf + k

While decoding is a little more complicated, we can simplify it to the opposite. We will read out the data and if we have a 0 in front of the current character, we will just write the character by itself. Otherwise we will write the dictionary reference to the passed in codeword plus the current character.

LZ78 is very dependent on the data structure that is being used to create the dictionary. Because we need to traverse the entire set of data being compressed we can assume that it will be at least O(n), but depending on what structure we check if the current set of letters is stored in the dictionary can cost very differently. If we assume that we are using a unordered map, the average case will be O(1), but in the worst case it will be O(n), therefore because we can assume that we will look for a dictionary entry every iteration we can assume that LZ78 has a big O(n^2)

LZ78 is a very interesting compression method, due to the fact that the more data that is given into it the more it will be able to compress. When creating dictionary entries we use the previously stored characters to shorten the strings, this helps tremendously because there are only 256 characters, therefore after we have logged every single character we can reference the dictionary over and over allowing us to hold larger and larger strings which will in turn compress our results even more.

When I started the code for this algorithm I realized that I was having trouble compressing anything over about 700 characters. After much thought and debugging I realized that I was trying to write 16 bit unsigned integers into 8 bit characters, this resulted in anything after 256 entries being scrambled due to the nature of only being to turn a 8 bit character into an 8 bit unsigned integer. To solve this problem I modified the code to be able to compress in 1 byte blocks meaning that my dictionary was only able to store 256 entries before I had to destroy it and create a new one. As discussed previously LZ78 performs better when there is more data that can be referenced in the dictionary thus 256 entries limiting the amount I was able to compress To show this example I created 4 test cases to show how this affected my results:

Name original File	Size of original file	Size of compressed file	Compression ratio
The bible	9.6 mb	7.6 mb	Compressed by : 17.8257%
Game of Thrones	1.7 mb	1.4 mb	Compressed by : 14.4645%
File of all letter A’s	27.1 mb	425 kb	compressed by : 98.431%
Our textBook	2.5 mb	2.2 mb	Compressed by : 12.7553%

As seen in the results above, having a 256 dictionary entry limitation was extremely restrictive for compressing anything that is not comprised of pure repetition as seen in the file comprised of all A’s. seeing the failure in my code I was able to modify it into taking two characters as a codeword, thus allowing me to encode code words with a size up to 2^16 or 65536. Having a dictionary with 65536 entries allowed me to increase my compression by a drastic amount, even if I had to encode two characters for every code word instead of only one. After having to write 33% more characters I was able to get these results:

Name original File	Size of original file	Size of compressed	Compression ratio

		file	
The bible	9.6 mb	4.7 mb	Compressed by : 50.7466%
Game of Thrones	1.7 mb	897 kb	Compressed by : 45.8099%
File of all letter A’s	27.1 mb	22 kb	compressed by : 99.9185%
Our textBook	2.5 mb	1.4 mb	Compressed by : 46.9417%

As seen in these two examples when increasing the dictionary size from 256 to 65536 entries the compression ratio increased by almost 35% for every single test case, even if I had to write 33% more characters to show what entry I had to use to decode. These test cases were able to demonstrate the power of lz78, showing how repetition allows for massive compression ratios as shown in the example of a file of all A’s even in the dictionary table of only 256 entries. I was able to compress the information by 98% which is incredible. on the other hand, increasing the dictionary entry size was a clear demonstration of how lz78 rewards length. By increasing the amount of information that I can reference to the dictionary the compression drastically improved. I was able to almost triple my compression rate for every single test.

Pseudo Code:

Compression:

```
Func compress( fileToRead, fileToOutput)
    dataToCompress = readfile(fileToRead)
    compressedData = “”
    currentChar = ‘’
    prevChars = “”
    dictionary = [pair<string, int>]
    Counter =0
    For i from 0 to dataToCompress.lengt()-1
        currentChar = dataToCompress[i]
        If dictionary.contains(currentChar)
            prevChars += currentChar
        Else
            If prevChars = “”
                compressedData += ‘0’
            Else
                compressedData+= convertBinary(dictionary[prevCharacters])
             compressedData += curChar`
             counter +=1
             dictionary.add( prevchars + curChars, counter)
             prevchars = “”
    writeToFile(compressedData)
```

Decompression:

```
Func Decompress( fileToRead, fileToOutput)
    decompressedData
    dataToDecompress = readFile(fileToRead)
    character
    codeword
    dictionary
    dictionary.add (0, "")
    For i from 0 to dataToDecompress.lengt()-1  i =1+2
        codeword = convertToDecimal(dataToDecompress[i])
        character = dataToDecompress[i+1]
        dictionary.add(dictionary.size()+1, dictionary[codeword] + character)
        DecompressedData += dictionary[codeword] + character
    writeToFile(decompressedData)
```

Feedback for presentation:

Reviewer 1 Chris Pugh:

- **Slides look nice and make it easy to follow.**
- **Good explanation on run-time and why the algorithm is effective.**
- **Duncan would be glad to see he has pseudocode   :D -dh**

Reviewer 2 Kevin Kahn:

- **Nice tie-in with the topics we have been discussing in class.**
- **Good detail explaining the algorithm and the pros and cons associated with the algorithm and your implementation of it.**

Reviewer 3 Lukas Pecson:

- **Pretty interesting, seems like a continuation of the algs we talked about in class**
- **Good examples**
- **Compressing a bible is pretty sick and the all a’s one being compressed by 98% is a great example to show the difference repetition makes**

Reviewer 4 Duncan Hendrickson:

- **Like the visual example!**
- **I have pretty much just good things to say! Nice!**
- **Very useful algorithm**

After listening to the comments from my Classmates, I realized I was on the right track, I was really glad that the algorithm I chose was also interesting for them.

Works Cited

Goemans, Michel. *Lempel-Ziv Codes - MIT Mathematics*.
math.mit.edu/~goemans/18310S15/lempel-ziv-notes.pdf.

Long, Darell Assignment 7 Lempel-Ziv Compression - CSE 13S - UC Santa Cruz–
Winter 2020. PDF

“LZ77 And LZ78.” *Wikipedia*, Wikimedia Foundation, 14 Mar. 2021,
en.wikipedia.org/wiki/LZ77_and_LZ78.

ojas1542. “ojas1542/Ojas-Code.” *GitHub*,
github.com/ojas1542/Ojas-Code/tree/master/LZ-78Compression.