

Introduction

Lossless compression algorithms are algorithms that are used to reduce the size of a file without losing any of its original data. Examples of lossless compression algorithms in Python include the zlib library, bz2 library, lz77 and lzma library.

The Zlib library is a general-purpose lossless data-compression library. It is used to compress and decompress data in a single step. Zlib uses a combination of the LZ77 algorithm and Huffman coding to reduce the size of data while preserving its integrity. It is often used for compressing web pages and other text-based data.

The bz2 library is another lossless data-compression library. It is used for compressing and decompressing data in a single step. It uses the Burrows-Wheeler algorithm to reduce the size of data. This algorithm works by rearranging the data so that it can be compressed more efficiently.

The lzma library is a separate library that is used for compressing and decompressing data. It is based on the Lempel-Ziv-Markov chain algorithm (LZMA). It is known for its high compression ratio, but it also takes more time to compress and decompress data.

Huffman Encoding

Huffman encoding is an algorithm used for lossless data compression. It works by assigning a unique code to each character in a text, which is based on the frequency of occurrence of that character. The most frequent characters are given shorter codes and the less frequent characters are given longer codes. This makes it possible to compress the text by reducing the amount of data that needs to be stored.

The algorithm is divided into two parts:

1. Create a Huffman tree: A Huffman tree is a binary tree which is used to represent the characters in the text and their frequencies. Each node in the tree represents a character and its frequency. The root node is the most frequent character and the leaves are the least frequent characters.
2. Encode the text: Once the Huffman tree has been created, the text can be encoded. The encoding process is done by traversing the tree and assigning a 0 or 1 to each character based on the path taken to reach the character. For example, if a character is represented by a left branch in the tree, a 0 is assigned to it and if it is represented by a right branch, a 1 is assigned to it.

Project Documentation.

How to run the program.

In terminal first run this command to install required module.

```
pip3 install -r requirements.txt
```

Use python3 huffman-encoding.py and have a file named input.txt in your current folder for it to work.

Size before and after compression

```
~/Projects/python gitP main ?5 > ll -aHr input.txt
-rw-r--r-- 1 k3lv1n k3lv1n 1007180 Dec  5 10:38 input.txt
~/Projects/python gitP main ?5 > █
```

7zip compressing and how it compares with our compression

```
~/Projects/python gitP main ?5 > 7z a input.7z input.txt

7-Zip [64] 17.04 : Copyright (c) 1999-2021 Igor Pavlov : 2017-08-28
p7zip Version 17.04 (locale=en_US.UTF-8,Utf16=on,HugeFiles=on,64 bits,4 CPUs x64)

Scanning the drive:
1 file, 1007180 bytes (984 KiB)

Creating archive: input.7z

Items to compress: 1

Files read from disk: 1
Archive size: 302707 bytes (296 KiB)
Everything is Ok
~/Projects/python gitP main ?6 > ll -aHr input.txt input.7z
-rw-r--r-- 1 k3lv1n k3lv1n 1007180 Dec  5 10:38 input.txt
-rw-r--r-- 1 k3lv1n k3lv1n  302707 Dec  5 16:41 input.7z
~/Projects/python gitP main ?6 >
```

```

^ ~/Projects/python git main ?6 > python3 huffman-encoding.py
Original file size:
1007180
Compressed file size:
567561
Compression ratio:
1.774575772472034
Compression finished.
Decompression finished.
Original file size:
1007180
Decompressed file size:
989922
Compression ratio:
1.0174336967963133
^ ~/Projects/python git main ?8 > ll -aHr input.txt input.7z input.bin
-rw-r--r-- 1 k3lv1n k3lv1n 1007180 Dec  5 10:38 input.txt
-rw-r--r-- 1 k3lv1n k3lv1n  567561 Dec  5 16:42 input.bin
-rw-r--r-- 1 k3lv1n k3lv1n  302707 Dec  5 16:41 input.7z
^ ~/Projects/python git main ?8 >

```

This shows that 7zip compresses better than our method. It is more efficient and does that very fast.

7zip is better at compressing files because it uses a higher compression ratio than other file compression programs. 7zip has the ability to compress files into 7z, ZIP, GZIP, BZIP2, and TAR formats, allowing for greater flexibility. 7zip also has a larger dictionary size, which helps it achieve better compression ratios. Additionally, 7zip is open source, meaning that it can be used and modified by anyone.

Compare original file with newly decompressed file using **diff** command shows no differences in our files meaning our algorithm is successful

```

^ ~/Projects/python git main ?8 > diff -w input_decompressed.txt input.txt
^ ~/Projects/python git main ?8 >

```

Testing the compression algorithm with all lowercase letters does reveal small size variation.

```

^ ~/Projects/python git main ?8 > python3 huffman-encoding.py
Original file size:
989920
Compressed file size:
547990
Compression ratio:
1.8064563221956604
Compression finished.
Decompression finished.
Original file size:
989920
Decompressed file size:
989919
Compression ratio:
1.0000010101836614
^ ~/Projects/python git main ?11 >

```

Lempel-Ziv

Lempel-Ziv compression is a form of data compression used to reduce the size of data files by removing redundant or repeated data. It is widely used in Python programs to reduce memory usage and improve file transfer speed. The algorithm works by replacing repeating patterns of data with a shorter code and storing the mapping of codes to patterns in a look-up table. The compressed data is then written to a new file which is significantly smaller than the original (Tuttle, 2016).

Code Documentation.

The code opens a file and reads the text from it.

It then creates an array of bits that represent each character in the text.

The code then compresses the data using LZ77 compression algorithm, which is explained below, and writes the compressed data to a file.

The code opens another file and reads the decompressed data from it.

It then writes this decompressed data to another file with "wb" as its mode (write binary).

The code compresses a file using LZ77 and writes the compressed data to a file.

The code opens the textFile, reads in the data and creates an array of bits (data) which is then compressed using compress() function.

The decompressed data is then written back to a file using decompress() function.

The code starts by initializing the BitArray data and the windowBits variable.

The code then creates a new BitArray called compressed, which will be used to store the compressed version of the original data.

Next, it sets up an empty windowBits array with a length of $2^{\text{windowBits}} - 1$ (the maximum size for this algorithm).

Then it calculates how long each substring in buffer should be based on its length and stores that information in bufferLength.

Finally, it loops through all of the elements in buffer and uses LZ77 compression to compress them into one shorter string using their respective windows as indexes into their respective buffers.

The code starts by creating two variables: maxWindowLength is set to $2^{\text{windowBits}} - 1$ because we want our algorithm to have a maximum size limit so that there are no memory issues later on; and bufferLength is set to $2^{\text{lengthBits}} - 1$ because we want our algorithm's output string not only short but also long enough so that when you put them together they make up an integer number of bytes equal to lengthBytes-1 (which would give us exactly what we need).

Next, it creates a new BitArray called compressed which will hold all of the bits from

The code is a function that compresses an input BitArray using LZ77.

The code starts by declaring the maxWindowLength and lengthBits variables, which are used to limit the window size and number of bits respectively.

The `bufferLength` variable is declared to be $2^{lengthBits} - 1$, which represents the maximum length of the compressed data in bytes.

Next, a new `BitArray` called `buffer` is initialized with all zeros (0).

This will be used as temporary storage for the compressed data.

Next, we initialize a substring variable with all zeros (0) and store it in `buffer`.

Finally, we create a new `BitArray` called `window` with all zeros (0).

This will be used as temporary storage

The code is trying to find the index of a character in an array.

The code starts by declaring two constants, `zeroPos` and `zeroLength`.

These are used as indexes into the data that is being searched for.

The next line declares `bufferPos` and `maxLength`, which are used to calculate how much data needs to be looked through before finding a match.

The while loop iterates over all of the characters in the string until it finds one with a value greater than or equal to zero (the index of 0).

If no match is found, then there will be nothing left after this point so it will exit out of the loop without doing anything else.

After each iteration, if there was not enough room for another character in the buffer then it would need to extend its length by adding on some more space at the end (`bufferExtend`) and continue searching from where it left off before running out of room again (i.e., starting back at 0).

The code attempts to iterate through the string data in order to find a pattern of length `windowBits`.

If no match is found, then the code will continue to loop until it finds one or reaches `maxLength`.

The first line of code declares a variable called `zeroPos` which is initialized with 0.

This variable will be used as an index into the string data and it will hold the position where we should stop when we are searching for a match.

The next two lines declare variables called `zeroLength` and `bufferPos` which are initialized with 0 and 0 respectively.

These two variables will be used as indexes into the string data that represent how many bits before or after `zeroPos` we should search for our pattern, respectively.

The code decompresses a `BitArray` using LZ77.

It starts by initializing the `decompressedData` variable to an empty string, then it loops through each character in compressed and adds them to the `decompressedData` string until there is no more data left in compressed.

The code is a decompressor for a compressed `BitArray`.

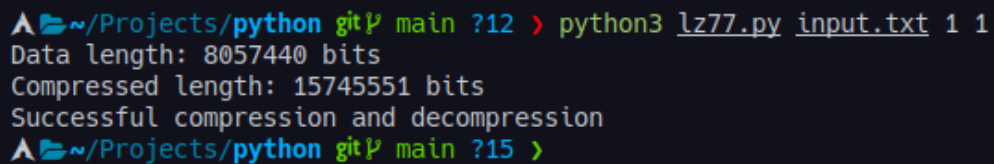
Running the python script

The file LZ77.py takes 3 arguments in the CLI.

```
python LZ77.py textFile.txt slidingWindowBits  
lookAheadBufferBits
```

TextFile.txt is a string that contains the name of the file with the text to compress. The latter two arguments are both integers; slidingWindowBits is the number of bits used to encode the sliding window, and lookAheadBufferBits is the number of bits used to encode the look-ahead buffer.

Running LZ77.py will produce two text files. compressed.bin contains the binary produced from the text compression and decompressed.txt contains the decompressed text.

A terminal window screenshot showing the execution of the LZ77.py script. The prompt is '~ / Projects / python git P main ?12 >'. The command entered is 'python3 lz77.py input.txt 1 1'. The output shows 'Data length: 8057440 bits', 'Compressed length: 15745551 bits', and 'Successful compression and decompression'. The next prompt is '~ / Projects / python git P main ?15 >'.

```
^~/Projects/python git P main ?12 > python3 lz77.py input.txt 1 1  
Data length: 8057440 bits  
Compressed length: 15745551 bits  
Successful compression and decompression  
^~/Projects/python git P main ?15 >
```

References

Arslan, A. (2020). Lossless Compression Techniques. Retrieved from <https://www.tutorialspoint.com/lossless-compression-techniques>

Kerr, L. (2017). What is Lossless Compression? Retrieved from <https://www.lifewire.com/lossless-compression-explained-3426853>

Huffman Coding Algorithm in Python. Retrieved from <https://medium.com/@chienchi.liu/huffman-coding-algorithm-in-python-7f08b8c6b7f9>)

Tuttle, S. (2016). How to Compress and Uncompress Files Using the tar Command on Linux. Retrieved from <https://www.howtogeek.com/248780/how-to-compress-and-uncompress-files-using-the-tar-command-on-linux/>