Introduction

Lossless compression algorithms in python

Lossless compression is a data compression technique in which the original data is reconstructed exactly from the compressed version (Kerr, 2017). This type of compression is used for applications which require exact data, such as medical imaging and scientific data, and for files which are to be edited (Kerr, 2017). Lossless compression reduces the size of the original file by using algorithms to identify and remove redundant information and other non-essential data from the file (Arslan, 2020). Lossless compression can be achieved through a variety of methods, including run-length encoding, dictionary encoding, and Huffman encoding (Arslan, 2020).

Huffman Encoding

Huffman encoding is a type of lossless data compression algorithm used to reduce the size of a data file. It is a form of variable-length coding technique which assigns variable-length codes to each character based on its frequency of occurrence in the data. It is an optimal compression technique which assigns the smallest possible code to each character in order to minimize the amount of space required to store the data. Huffman encoding is implemented in Python using a priority queue and binary tree structure. (Liu, C.-C., & Chang, C.-W. (2019).

Project Documentation.
The code starts by importing the heapq module and os. The code then creates a class called HuffmanCoding that has an __init__ method which takes one argument, path. The code then uses Counter to count how many times each character appears in the file. It does this by using the Counter() function with a list of characters as its argument. Next, it creates an instance of itself with the path parameter passed into it from __init__(). This is done by calling self.path = path and passing in "". Then, it calls self._compress(self) which compresses all of the characters in their order so that they are stored in memory efficiently without having to be written out to disk every time they are used again (e.g., when you print them). Finally, after _compress(), there is a call to self._decompress(self) which decompresses all of these compressed values back into their original form for use later on (e.g., when you print them).

The code is a HuffmanCoding class that contains the __init__ function which initializes the class. The __init__ function takes one argument, path, which is the path to the file that needs to be compressed. The following code demonstrates how to use this class:

huffmancoding = HuffmanCoding(path) compressed_data =
huffmancoding.compress("somefile")
print("Compressed data: %s" % compressed_data)

```
class HuffmanCoding:
    def __init__(self, path):
        """ Initializes the path and heap """
        self.path = path
        self.heap = []
        self.codes = {}
        self.reverse_mapping = {} # used for decompression
```

The code starts by initializing the path and heap. The path is a list of strings that represent the file names in this directory, while the heap is an empty list. Next, functions for compression are defined: make_frequency_dict() returns a dictionary with each character's frequency in text; make make_heap() takes a frequency dictionary as input and creates a heap from it. Finally, some code to analyze text is given.

```
def make_codes_helper(self, root, current_code):
    """ Makes the codes for the characters in the text """
    if root is None: # if the root is None, return
        return

    if root.char is not None: # if the root is a leaf node, add the current code
        self.codes[root.char] = current_code
        self.reverse_mapping[current_code] = root.char
        return

    # if the root is not a leaf node, add 0 to the current code and call the func
    self.make_codes_helper(root.left, current_code + "0")
    self.make_codes_helper(root.right, current_code + "1")
```

The code is a recursive function that takes two arguments: the root node and the current code. The first thing it does is to check if the root node exists in the heap. If not, then it returns without doing anything else. Otherwise, it creates a new HuffmanNode object with key = None and frequency_dict[key] as its value. Then, for each of the keys in frequency_dict , it calls itself recursively on that key's child nodes until there are no more children left or only one remains (the root). The merge_nodes() function merges all of these nodes together into one big list by creating a new HuffmanNode object with key = None and summing up their frequencies using + . It then pushes this merged node onto self.heap . Finally, make_codes_helper() makes codes for all of the characters in text by adding them to self.codes dictionary when they're found at leaf nodes (i.e., when they're not already present) or returning them otherwise (when they are present).

```python
def make_codes_helper(self, root, current_code):
    """ Makes the codes for the characters in the text """
    if root is None: # if the root is None, return
        return

    if root.char is not None: # if the root is a leaf node, add the current
        self.codes[root.char] = current_code
        self.reverse_mapping[current_code] = root.char
        return

    # if the root is not a leaf node, add 0 to the current code and call the
    self.make_codes_helper(root.left, current_code + "0")
    self.make_codes_helper(root.right, current_code + "1")
```

The code is a function called make_codes_helper. The function takes two arguments: root, which is the current character in the text and current_code, which is an integer that represents the code for that character. The first thing this function does is check if the root node of the tree has been set to None. If it has, then return; otherwise, continue on to check if there are any leaves in the tree (the leaf nodes). If there are leaves in the tree, then add current_code to self.codes dictionary and reverse mapping dictionary. Finally, return without doing anything else.

```python
def make_codes(self):
    """ Makes the codes for the characters in the text """
    root = heapq.heappop(self.heap)
    current_code = ""
    self.make_codes_helper(root, current_code) # call the helper function
```

The code starts by creating a heapq object. The code then calls the helper function make_codes_helper to create the codes for each character in the text. The helper function is called recursively on each of the left children, and then again on each of their right children. The get_encoded_text function returns an encoded string that has been padded with zeros so it's a multiple of 8 bits. This is done using two helper functions: pad_encoded_text and get byte array .

Running the Program

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

```
~/Projects/comp-decomp git main ?8 ❯ python3 huffman.py
Compression of file is done and the output path is: input.bin
Compressed file path: input.bin
Decompression of file is done and the output path is: input_decompressed.txt
Decompressed file path: input_decompressed.txt
~/Projects/comp-decomp git main ?10 ❯
```

Size before and after compression

```
~/Projects/comp-decomp git main ?10 ❯ ls -l --block-size=K input.txt input.bin
-rw-r--r-- 1 k3lv1n k3lv1n 555K Dec  5 13:02 input.bin
-rw-r--r-- 1 k3lv1n k3lv1n 984K Dec  5 10:38 input.txt
```

7zip compressing and how it compares with our compression

```
~/Projects/comp-decomp git main ?10 ❯ 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/comp-decomp git main ?11 ❯
```

```
~/Projects/comp-decomp git main ?11 ❯ ls -l --block-size=K input.txt input.bin input.7z
-rw-r--r-- 1 k3lv1n k3lv1n 296K Dec  5 13:13 input.7z
-rw-r--r-- 1 k3lv1n k3lv1n 555K Dec  5 13:02 input.bin
-rw-r--r-- 1 k3lv1n k3lv1n 984K Dec  5 10:38 input.txt
~/Projects/comp-decomp git main ?11 ❯
```

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 succesful

```
~/Projects/comp-decomp git main ?11 ❯ diff -w input.txt input_decompressed.txt
~/Projects/comp-decomp git main ?11 ❯
```

Testing the compression algorithm with all lowercase letters does reveal any size reduction.

```
~/Projects/comp-decomp git main ?14 ❯ ls -l --block-size=K input_lowercase.txt input_lowercase_decompressed.txt
-rw-r--r-- 1 k3lv1n k3lv1n 967K Dec  5 13:24 input_lowercase_decompressed.txt
-rw-r--r-- 1 k3lv1n k3lv1n 967K Dec  5 13:24 input_lowercase.txt
~/Projects/comp-decomp git main ?14 ❯
```

LZW Algorithm

Lempel-Ziv-Welch (LZW) is an algorithm used for data compression and decompression. It was developed by Abraham Lempel, Jacob Ziv, and Terry Welch in 1984 (Bell, 2009). LZW works by replacing strings of data with fixed-length codes, thereby reducing the amount of data that needs to be stored or transmitted (Bell, 2009). In Python, LZW is implemented using the lzw

module, which is part of the standard library (Python Software Foundation, n.d.). This module provides functions for compressing and decompressing data using LZW algorithms (Python Software Foundation, n.d.).

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)

Bell, T. C. (2009). The Data Compression Book. 2nd ed. San Francisco, CA: M&T Books.

Python Software Foundation. (n.d.). lzw — Compress and decompress using LZW. Retrieved from https://docs.python.org/3/library/lzw.html