Alberto Lopez, Romaine Ewan

CSCI-323 Design and Analysis of Algorithms

Semester- Summer 2019

Compressing Data

Introduction:

We will be examining three compression algorithms. Huffman coding, LZW and RLE algorithms will be the algorithms that we will be studying and testing. The expected time complexity for Huffman coding should be O(nlogn) since we will be using a binary heap implementation for it. For LZW, it should be O(n), assuming that we have a data structure that is able to retrieve information in constant time, like a hash table.. For RLE algorithm should be O(n) because will be iterating through each element and encoding it.

Algorithmic Overview:

David A.Huffman published the Huffman code in 1952. The idea of Huffman coding is to be able to compress the number of bits that we would need to store data. It prioritizes the most frequently used letters or numbers and let's it get compressed into the data that will have the least number of bits. Since it's used frequently, then it will make sense to store data, that will be used over and over, into a small amount of data. As for the less frequent data, it is compressed into bigger bits. It utilizes a tree structure, a heap in my case, to be able to assign and traverse data into smaller units. Since we will be using heap to create our tree, our time complexity will be O(nlogn) because we will need to iterate through each element and then heapify everytime we remove an element.

The LZW was discovered by Abraham Lempel, Jacob Viz and Terry Welch as a variation to the original LZ77 algorithm. It utilizes a dictionary by comparing pairs of data that are next to each other and then checks if it exists within the table. If it already exist then we add an extra bit

to the already compressed data but if it doesn't exist then we add it to the table and assign it a value ID. This compression algorithm relies on the retrieval of information from a table and so, this means that the time complexity relies on which data structure we decide to use for the table. The best option is to use a hash table as it has an insertion time of constant and if implemented with a good hash function then we could make insertion be constant as well. Keeping it constant for n elements means that we should expect a time complexity of O(n).

The run-length compression algorithm was patent by Hitachi in 1983. The idea behind it is that it takes a string of data and it then counts the number of repeating symbols and removes all repeating symbols and replaces it with the number of times it appears. It compresses a large string into a much smaller piece that has numbers at the end of each symbol to signify the count of symbols. Running through the algorithms would take us n number of iterations since we have to count every single element. Therefore, giving us a time complexity of O(n).

Implementation Consideration:

- For the Huffman coding, our code will use a binary heap to construct the encoding of each string.
- For our LZW encoding,
- For our Run-Length encoding we will be using a dictionary to be able to keep track of the number of times a letter appears.
- The programming language we coded this in was Python and used a MAC running MAC OS to run our programs. We kept track of the number of comparisons that it did when choosing an optimal path.

Data Consideration:

What we decided to do was create a randomizer that generate a random string for each algorithm. We did this starting off from ten elements and then adding up ten times the amount until a hundred thousand elements. Elements representing the size of the string. We counted the number of operation based off of the number of times we had to put elements into my dictionary. In the Huffman coding, we used a Heap so I counted every element once times Logn for the heapify portion. For the run-length, we counted the amount of counts it had to do when traversing the string. Finally, for LZW, we counted the number of times it had to access the table.

Tracking of Operations:

What we kept track of was the amount of **time** it took for each algorithm to compress and the amount of times it had to perform an **operation** to the elements.

The results are displayed in the **console** when the program is run.

Future Study:

We enjoyed studying about Huffman coding, so studying further on how to create a universal huffman tree for all text documents sounds like a good challenge to work on for Graduate School or during our two weeks off.

Sources:

https://en.wikipedia.org/wiki/Main_Page for background information on the algorithms

https://rosettacode.org/wiki/Huffman_coding#Python for the huffman coding source code

https://www.geeksforgeeks.org/run-length-encoding-python/ for the run-length source code

https://rosettacode.org/wiki/LZW_compression#Python for LZW source code

-Lecture from July 31, 2019 to be able to understand the time complexities.