Algorithmics I – Assessed Exercise

Status and Implementation Reports

Karlis Siders
2467273S

November 15, 2021

Status report

The program is working correctly as the output it gives is the same (file size- and ratio-wise) as given on Microsoft Teams.

Implementation report

Compression ratio for the Huffman algorithm

For the Huffman algorithm, first, a dictionary of character frequencies is made and the whole text given is parsed to fill this dictionary. Then, a Priority Queue is used to build a queue of all parentless Nodes. The queue's internal comparator is based on the Nodes' weights. Finally, the weighted path length (WPL) is calculated by going through this Priority Queue, making a parent Node for all parentless Nodes with its weight as the sum of the weights of its children, and adding the parent to the parentless Nodes queue as well, until there are no parentless Nodes in the queue anymore. The WPL is the sum of the weights of the internal Nodes, while also being the size of the compressed file.

Efficiency is gained by using Java's java.util.PriorityQueue, which uses a min-heap as its base. The minimum heap reduces the time to find the nodes with the least weight.

Compression ratio for the LZW algorithm

For the LZW algorithm, first, a dictionary is initialised as a trie with all first 128 ASCII characters in it. Then, going character by character through the input file, the longest word already in the dictionary is found, inserted into the dictionary along with the next character, and the position in the file changes to the index of this new character, when the search is started again. The Node class stores its codeword length as a field in order for the last word in the input file, most likely already in the dictionary, to be added to the length of the compressed file.

Efficiency is gained by using a Trie class and search/insert methods which are used for searching for the longest word and inserting a new word into the dictionary. A polymorphic search() method is added to search for a character as a child of a given Node to avoid going from the root every time and repeating the search path for a string with an added character. Future improvements could involve adding a single character at the last Node of the longest word (instead of adding the whole word), and using a faster lookup (O(1)) data structure, like an array or HashMap, for storing the children of a Node and finding the relevant child.

Empirical results

`Input file small.txt Huffman algorithm`

```
Original file length in bits = 46392
Compressed file length in bits = 26521
Compression ratio = 0.57167184
Elapsed time: 43 milliseconds

Input file medium.txt Huffman algorithm

Original file length in bits = 7096792
Compressed file length in bits = 4019468
Compression ratio = 0.5663782
Elapsed time: 290 milliseconds

Input file large.txt Huffman algorithm

Original file length in bits = 25632616
Compressed file length in bits = 14397675
Compression ratio = 0.56169355
Elapsed time: 535 milliseconds


Input file small.txt LZW algorithm

Original file length in bits = 46392
Compressed file length in bits = 24832
Compression ratio = 0.53526473
Elapsed time: 40 milliseconds

Input file medium.txt LZW algorithm

Original file length in bits = 7096792
Compressed file length in bits = 2620426
Compression ratio = 0.3692409
Elapsed time: 541 milliseconds

Input file large.txt LZW algorithm

Original file length in bits = 25632616
Compressed file length in bits = 9158986
Compression ratio = 0.35731766
Elapsed time: 1562 milliseconds
```