Algorithmics I – Assessed Exercise

Status and Implementation Reports

Iain McLean
2308499M

November 18, 2019

Status report

Huffman and LZW compression both successfully implemented. Both algorithms produce the correct expected outputs for *small.txt*.

Implementation report

Compression ratio for the Huffman algorithm

Implemented the Huffman through implementing 4 separate classes: HuffMain, HuffTree, HuffComparator and Node.
HuffMain: Reads in the entire text using the *Files* API, using the *readAllBytes* function, and then convert that file into a string, which is then passed into the HuffTree class. .
HuffTree: Builds a HashMap to calculate the character frequencies of every character in the text, and then creates a Node array which is appended into a priority queue. While there is still 2 nodes in the array, create a parent equal to the weight of the two nodes combined, and add that value to a running total of the Weighted Path Length.
HuffComparator: Used for the priority queue.
Node: Data structure for storing characters and their weights.

Main efficiency improvements were through implementing the Files API to read in the files, and avoiding the use of *scanner*. Another improvement was to implement a priority queue instead of using a MinHeap and a list, which improved efficiency by >20%. Implementing the Files API to read in the file instead of using the scanner and reader classes increased efficiency by 33%. A further improvement to improve Huffman would be to remove the use of Nodes.

Compression ratio for the LZW algorithm

Implemented LZW through implementing 5 separate classes: LZWMain, LZW, Node, Trie and codeNode.
LZWMain: Passes file path to LZW class.
LZW: Initialises trie with all the ascii characters. Using a *FileReader*, reads in char by char and calls build on each character. Each character is appended to a *StringBuilder*. A current word variable is used to check if the characters exists in the trie, and is reset whenever a new string is appended to the trie. Codeword length dynamically grows as the trie size increases. File Length is calculated by appending the codeword length to a variable whenever a word is added to the trie.
Node: Provided Node implementation.
Trie & codeNode: Modified Trie implementation to return the codeword length of the last word in the text.

Main efficiency improvements were through using StringBuilder which massively improved string addition time. Another improvement was to avoid reading the whole file all at once and then run LZW, more efficient to perform building the trie during reading. Efficiency could have also been improved by modifying the trie, so that when searching the trie for the new characters, it begins searching at the previously found node.

Empirical results

```
Tested on Lab machines.

Input file small.txt Huffman algorithm

Original file length in bits = 46392
Compressed file length in bits = 26521
Compression ratio = 0.5717
Elapsed time: 140 milliseconds

Input file medium.txt Huffman algorithm

Original file length in bits = 7096792
Compressed file length in bits = 4019468
Compression ratio = 0.5664
Elapsed time: 110 milliseconds

Input file large.txt Huffman algorithm

Original file length in bits = 25632616
Compressed file length in bits = 14397675
Compression ratio = 0.5617
Elapsed time: 171 milliseconds

Input file small.txt LZW algorithm

Original file length in bits = 46392
Compressed file length in bits = 24832
Compression ratio = 0.5353
Elapsed time: 47 milliseconds

Input file medium.txt LZW algorithm

Original file length in bits = 7096792
Compressed file length in bits = 2620428
Compression ratio = 0.3692
Elapsed time: 328 milliseconds

Input file large.txt LZW algorithm

Original file length in bits = 25632616
Compressed file length in bits = 9158986
Compression ratio = 0.3573
Elapsed time: 1156 milliseconds
```