Algorithmics I – Assessed Exercise

Status and Implementation Reports

Ciara Losel
2438870L

November 15, 2021

Status report

I believe that my Huffman algorithm is working correctly due to the fact that I get exactly the same results as the example results provided by results.txt. The only result that differs is the elapsed time, which is expected due to laptop specifications. My laptop is quite old and slow, so I believe this is the root of the problem.

My LZW algorithm seems to be not correctly calculating the new file size. If one compares it to results.txt, the new file sizes for small.txt, medium.txt and large.txt are 4 bits smaller, 2 bits larger and 3 bits larger than they should be, respectively. I believe that this might be due to an incorrect calculation of the codeword length. Unfortunately, I ran out of time to debug this issue. Again, the elapsed time differs greatly to what is expected but, as discussed above, I think this is due to my laptop's abilities.

Implementation report

Compression ratio for the Huffman algorithm

My Huffman implementation runs in O(n+k) time, where n is the total number of characters in the inputted text file and k is the number of unique characters in the text file. It uses the weighted path length in order to calculate the size of a Huffman compressed version of the inputted text file. The weighted path length is equal to the new file size since Huffman encoding usually involves storing the file as a binary tree. However, one can find the weighted path length without having to create the tree, which is discussed below.

I have used a Java dictionary (Hash Map) in order to represent the ASCII characters and their corresponding frequencies in the inputted text file. With a Scanner, the algorithm reads each line of the input file and adds a new line character to it so that this is included in the calculations. For each line in the file, it iterates through the characters, updating the frequency dictionary accordingly. If the character is new, it is added to the dictionary. If it is already in the dictionary, the character's frequency is incremented by 1. It also keeps track of the total amount of characters in the file, which gets multiplied by 8 in order to find the original file size.

Once the file has been read, the frequencies are then all stored in a Java list. A list works well here because the algorithm requires the ability to remove and add elements to the list frequently. As stated previously, my implementation calculates the weighted path length in order to calculate the compression ratio. The weighted path length is calculated by repeating the following steps... 1. Remove the two smallest frequency values from the list 2. Sum these values 3. Add the summation to the list 4. Add the summation to the weighted path length until there is only one value left in the list. This value would be the weight of the root node if the compression was represented as a tree.

In order to make my Huffman algorithm efficient I have implemented a number of steps to ensure this. Firstly, once I create the list of frequency values, I use Java's sort function to sort it. This means retrieving the two smallest values is easy, as it's just the first two values. Originally, I continued to sort the list every time a new element was added. However, I have made it more efficient by simply finding the position at which the new element should go. This is possible since the list is already sorted. Lastly, one would normally implement Huffman encoding using a binary tree. However ,a binary tree representation of the file isn't required because, as described above, the weighted path length can be found by simply knowing what the weights of the nodes would be. We know that all the character frequencies are represented by leaf nodes and that their parent nodes are the sum of the two child weights. Hence why, with this information, the weighted path length can be found from the list of frequency values.

The compression ration is found by calculating the ratio of the new file size to the original file size.

Compression ratio for the LZW algorithm

My LZW implementation runs in O(n) time, where n is the total number of character in the inputted text file. It uses the length of codewords in order to calculate the size of an LZW compressed version of the inputted text file. Since in LZW we compress the original text into a series of character sequences represented by codewords, it is sufficient to use the length of these codewords in order to calculate what the size of the new file would be.

My implementation uses a trie in order to represent the dictionary, with adjusted search and insert functions which will be discussed now. The insert function takes as parameters a character to insert and the node that should be the parent node of this new node. It gets the first child of the new node and sets this as the new node's sibling. It then sets the new node, representing the character to be inserted, as the parent node's first child. This inserts the new character in the correct position in the trie. The search function takes as parameters a character to find and a node to start the search at. Having this start node as a parameter makes the algorithm more efficient as the LZW algorithm can specify a node to search at. This is discussed further below. The search function iterates through the trie, starting at the start node, until it can return an appropriate node, whether that be the node at which the character exists, or the node at which it should get inserted because it doesn't exist.

Firstly, the trie is filled with the first 128 ASCII characters, all as children of an empty root node. This prepares the dictionary to be built upon when the text file gets read.

As in my Huffman implementation, each line from the file is read with a scanner and a new line character is added. It iterates through each character in each line and updated the dictionary accordingly. Whilst the file is read, there are a few numbers that my implementation keeps track of. First is the codeword length. Since 128 characters were added to the trie, we start the text file reading off with a codeword of 128, which gives a codeword of length 8. However, this codeword length will increment by 1 every time the trie gets to it's maximum capacity. Hence, it also keeps track of the maximum capacity, which changes when the codeword length is incremented. To know when the trie has reached it's maximum capacity, it also keeps track of the trie size which increments every time a node is added. It also keeps track of the total amount of characters in the file, which gets multiplied by 8 in order to find the original file size. The last number it keeps track of is the new file size. Every time a character is inserted into the dictionary, the new file size is added to with the current codeword length since we know that's how long the new character's codeword is.

For every character in the text file, the implementation searches for the character in the dictionary in a certain area, as a trie that is, it searches for the character in the child nodes of a certain node. This node is given by the result of the previous character's search. For the first character, it calls the search function with the root node, which will return the node representing the first character of the file, recalling that the dictionary already contains all the first 128 ASCII characters. For the second character in the file, it then searches for it with in the child nodes of the first character's node. Evidently this won't exist, hence it returns the first character's node to indicate that this is where the second character should get inserted. The implementation will call the insert function on the second character and the node representing the first character. It will also increment the trie size, update the new file size and set the next node for the next iteration. This next node should be the child node of the root that represents the second character, which starts off the next prefix. This is the procedure that occurs for each character in the file.

Before a new node is added, the trie size is checked to see if it has reached it's maximum capacity. If it has, the codeword length is incremented and the maximum capacity is adjusted accordingly.

The compression ratio is found by calculating the ratio of the new file to the original file size.

Empirical results

This section is part of the marking scheme ”Outputs from test data: 2 marks”.

If the program fails to terminate in, say, two minutes, simply report non-termination. To print your outputs you can use the verbatim environment:

```
Input file small.txt Huffman algorithm

Original file length in bits = 46392
Compressed file length in bits = 26521
Compression ratio = 0.5717
Elapsed time: 173 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: 485 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: 880 milliseconds

Input file small.txt LZW algorithm

Original file length in bits = 46392
Compressed file length in bits = 24828
Compression ratio = 0.5351785
Elapsed time: 159 milliseconds

Input file medium.txt LZW algorithm

Original file length in bits = 7096792
Compressed file length in bits = 2620428
Compression ratio = 0.3692412
Elapsed time: 588 milliseconds

Input file large.txt LZW algorithm

Original file length in bits = 25632616
Compressed file length in bits = 9158989
Compression ratio = 0.35731778
Elapsed time: 1101 milliseconds
```