INF144 Mandatory 2 – CGJ008

How I have solved the task

Lempel-Ziw-Welch compression

- Initialize
    1. Populate the encoding dictionary and the decoding dictionary with the characters of the given alphabet
    2. Find the initial binary code length (block size) needed to represent a text of the given alphabet (e.g. to represent a text made up of an alphabet with 29 distinct characters, we need 5 bits)
    3. Set current block size equal to initial block size
- Compress
    1. Find the first subset of characters in the text which is not in the encoding dictionary
    2. Add this subset to the dictionary at index equal to the size of the dictionary
    3. Add the dictionary index of this subset, minus the last character, to the compression output (as binary, number of bits equal to current block size)
    4. If the dictionary size is now strictly greater than $2^{current\ block\ size}$, increase block size
    5. Find the next subset, which is not in the dictionary, starting with the last character of the previous subset
    6. Repeat from 2. until all the text is compressed
- Decompress
    1. Set current block size equal to initial block size
    2. Read the first block of length equal to current block size, from the compressed source
    3. Add its translation from the decode dictionary to the output
    4. Store the translation, call it **prev**
    5. Read the next code
    6. If it exists in the dictionary
        - Store its translation as **S**

       Else
        - Store **prev** + **C** as **S**
    7. Output **S**
    8. Store the first character of **S** as **C**
    9. Add **prev** + **C** to the dictionary
    10. If the dictionary size is now <u>equal to or greater than</u> $2^{current\ block\ size}$, increase block size
    11. Store the most recently read code as **prev**
    12. Repeat from 5. until all the compressed source is processed

Huffman coding

Encode

- Create a map of each possible code in the source, pointing to how frequent the code occurs in the source
- Create comparable nodes of each possible code (comparable by their frequencies)
- Insert the nodes in a priority-queue
- While the queue contains more than 1 node:
    1. Extract the 2 nodes with the lowest frequency numbers
    2. Create a new node with frequency equal to the sum of the 2 extracted nodes
    3. Set the new node as parent of the 2 extracted nodes
    4. Store a reference to the parent node as **root**
    5. Add the new parent node to the priority-queue
    6. Repeat from 1.
- Create a code map (from source code to Huffman codes)
    1. Start a search of the Huffman tree from **root.**
    2. When recursively going down the tree, add a '0' on left turns, add a '1' on right turns.
    3. When reaching a leaf node, add an entry to the code map, mapping the leaf node's source code to the Huffman code created by left- and right turns.
- Encode each source code using the code map created

Decode

- Decode each Huffman code using the code map created (reverse)

Results

Notes on interpretation and assumptions

I've calculated compression based on an assumption that we consider the initial space requirement of a text to be its number of characters ∗ the bits required to represent its alphabet, and not the usual 8 bits used to store a string character in common programming languages. In this assignment's case, that means all text is initially considered to have a space requirement of 5∗number of characters.

Compression results are calculated as the percentage of size reduction from original.

Testing the implementation on the folk tale

Uncompressed bit-length: 39955
LZW-Compressed bit-length: 27879
LZW compression rate: 30.22%

Bit-length before Huffman-coding: 27879
Bit-length after Huffman-coding: 28421
Huffman-coding compression ratio: -1.94%

Final compression rate from initial bit-length of 39955 to final bit-length of 28421:
28.87%

Am I able to improve the compression rate by applying Huffman coding to the LZW output?

No, not on this type of data with this certain length. Looking at the analysis of the blocks created by LZW compression, it seems most LZW codes are used only once or twice, while few are in the range of 3-10 occurrences. I suspect Huffman coding in this case is ineffective because LZW creates new codes throughout the compression process, while being effective as it codes increasingly longer substrings, it results in a wide array of infrequently used codes. Huffman coding would be more effective if the frequency of codes used were more diverse, and not close to uniform as with the LZW output.

Compression of 100 generated texts from order 0 through 3 of Markov model approximations

Average LZW compression rate for order 0 is 1.06%
Average LZW+Huffman compression rate for order 0 is 3.37%

Average LZW compression rate for order 1 is 20.11%
Average LZW+Huffman compression rate for order 1 is 19.59%

Average LZW compression rate for order 2 is 29.53%
Average LZW+Huffman compression rate for order 2 is 28.27%

Average LZW compression rate for order 3 is 31.62%
Average LZW+Huffman compression rate for order 3 is 30.04%