Project 5

In this project, you will create a huffman coding tree implementation alongside a knuth-morris-pratt (KMP) string search implementation. We first explore huffman encoding followed by KMP.

Huffman Encoding

Huffman encoding is a particular type of optimal algorithm for lossless data compression. It uses frequency of occurrence (weight) of each possible value of the source character and maps itr to a variable-length binary encoding. This strategy is very effective for compression as more common symbols are represented using fewer bits than less common symbols.

Theory

The algorithm starts by using the weights of each character to construct a huffman tree in a bottom up manner. The two lowest frequency nodes with no parent node are combined by creating a new node that serves as their shared parent and has the combined weight of both child nodes. We iteratively perform this merging step until there are no more nodes left to merge and we thus end up with a complete huffman tree. A sample run of the tree construction is shown below:

[Figure tree Construction]

For our implementation, we always place nodes with lower weight as left child and higher weight as right child while performing the merge step. Ties can be resolved arbitrarily. Each new node created during merging always holds the null character (‘\0’).

Once the tree has been constructed, we can traverse the tree to find encodings and decodings of given strings based on the path they match on the tree. But, to make this more efficient, we traverse the tree before-hand to construct the encoding and decoding mappins. The encoding map maps characters to binary strings and the deciding map maps binary strings to characters. Instead of repeatedly traversing the tree, we will instead use these two maps to encode and decode input strings.

Coding

You will complete the HuffmanCode class implementation in HuffmanCode.java to construct a fully functioning Huffman encoding implementation. This class involves methods for both encoding (compress) and decoding (expand). A Node class representing each node in the huffman tree has already been provided to you. Fill in the //TODO blocks to complete the implementation and use it in the HuffmanCode class. Within the HuffmanCode class, complete all the //TODOs to help support the compress and expand operations correctly.

TODO

- Node.isLeaf : Complete method to check if current node is a leaf node
- Node.compareTo : Complete method that compares two nodes and returns an integer comparison value
- HuffmanCode constructor: Complete the constructor that needs to set up variable text with the input text, variable huffmanTree with the root of the huffman tree after construction, and the encoding and decoding maps that store required compression information.
- HuffmanCode.compress: Complete method to support string compression using the encoding map
- HuffmanCode.expand: Complete method to support string decompression using the decoding map

Notes

- Lower weight child is always the left child in the huffman tree
- Each new node formed during tree construction hold the null character
- In case encoding does not exist, return an IllegalArgumentException with the message “Encoding Not Found!”
- In case decoding does not exist, return an IllegalArgumentException with the message “Decoding Not Found!”

KMP

Theory
Coding