TCSS 342 - Data Structures
Assignment 3 - Compressed Literature

Guidelines

This assignment consists of programming work with written work as extra credit. Solutions should be a complete working Java program including both your work and any files provided by the instructor used in your solution. These files should be compressed in a zip file for submission through the Canvas link.

This assignment is to be completed on your own or in a group of two. If you choose to work in a group of two this must be clear in your submission. Please see the course syllabus or the course instructor for clarification on what is acceptable and unacceptable academic behavior regarding collaboration outside a group of two.

Assignment

Standard encoding schemes like ASCII are convenient and relatively efficient. However we often need to use data compression methods to store data as efficiently as possible. I have a large collection of raw text files of famous literature, including Leo Tolstoy's War and Peace consisting of over 3 million characters, and I'd like to store these works more efficiently. David Huffman developed a very efficient method for compressing data based on character frequency in a message.

In this assignment you will implement Huffman's coding algorithm in a CodingTree class. This class will carry out all stages of Huffman's encoding algorithm:

- counting the frequency of characters in a text file.
- creating one tree for each character with a non-zero count.
    - the tree has one node in it and a weight equal to the character's count.
- repeating the following step until there is only a single tree:
    - merge the two trees with minimum weight into a single tree with weight equal to the sum of the two tree weights by creating a new root and adding the two trees as left and right subtrees.
- labelling the single tree's left branches with a 0 and right branches with a 1 and reading the code for the characters stored in leaf nodes from the path from root to leaf.
- using the code for each character to create a compressed encoding of the message.
- **(Optional)** provide a method to decode the compressed message.

You are also responsible for implementing a Main controller that uses the CodingTree class to compress a file. The main must:

- Read the contents of a text file into a String.
- Pass the String into the CodingTree in order to initiate Huffman's encoding procedure and generate a map of codes.

- Output the codes to a text file.
- Output the compressed message to a binary file.
- Display compression and run time statistics.

Formal Specifications

You are responsible for implementing the CodingTree class that must function according to the following interface:

- void CodingTree(String message) - a constructor that takes the text of a message to be compressed. The constructor is responsible for calling all private methods that carry out the Huffman coding algorithm.
- Map<Character, String> codes - a **public** data member that is a map of characters in the message to binary codes (Strings of '1's and '0's) created by your tree.
- String or List<Bytes> bits - a **public** data member that is the message encoded using the Huffman codes.
- **(Optional)** String decode(String bits, Map<Character, String> codes) - this method will take the output of Huffman's encoding and produce the original text.

To implement your CodingTree all other design choices are left to you. It is strongly encouraged that you use additional classes and methods and try to use the proper built in data structures whenever possible. For example, in my sample solution I make use of a private class to count the frequency of each character, a private node class to implement my tree, a recursive function to read the codes out of the finished tree, and a priority queue to handle selecting the minimum weight tree.

You will also create a Main class that is capable of compressing a number of files and includes methods used to test components of your program.

- void main(String[] args) - this method will carry out compression of a file using the CodingTree class.
    - Read in from a textfile. You may hardcode the filename into your program but make sure you test with more than one file.
    - Output to two files. Again feel free to hardcode the names of these files into your program. These are the codes text file and the compressed binary file.
    - Display statistics. You must output the original size (in bits), the compressed size (in bits), the compression ratio (as a percentage) and the elapsed time for compression.
- Test files - include at least one test file that is a piece of literature of considerable size.
    - Check out Project Gutenberg an online database of literature in text format.

(Optional) Implement your own MyPriorityQueue<T> class using the array implementation mentioned in lecture. Use it in place of the Java PriorityQueue in your CodingTree class.

Analysis **(Optional)**

In addition to this programming assignment you may also complete a high level worst-case runtime analysis of Huffman's algorithm. You will assume for your analysis that *n* is the length of the message to be encoded, and that *m* is the number of unique characters in the message. Provide a big-oh analysis of each step of Huffman's algorithm:

- Counting the frequency of characters in a message.
- Tree initialization and construction using a priority queue.
- Reading codes from the tree.
- Encoding the message.

Combine these estimates to express the runtime of the algorithm as a whole.

Submission

The following files are provided for you:

- WarAndPeace.txt - the plain text of Leo Tolstoy's novel War and Peace.
- codes.txt - An appropriate set of codes produced by my sample solution. (Note: This is not a unique solution. Other proper encodings do exist.)
- compressed.txt - The compressed text of the novel. It's size is the most relevant feature.

You will submit a .zip file containing:

- Main.java - the simulation controller.
- CodingTree.java - the completed and functional data structure.
- <sample>.txt - a novel or work of art in pure text for that you have tested your program on.
- compressed.txt - the compressed version of your selected text.
- codes.txt - the codes produced on your selected text.
- **(Optional)** MyPriorityQueue.java - an array based implementation of PriorityQueue.
- **(Optional)** Analysis.pdf - a pdf containing the runtime analysis of Huffman's compression algorithm.

Grading Rubric

Each of the following will be awarded **one or more** points toward your assignment grade. Not all points need to be achieved to receive a perfect grade. Excess points contribute to the total points gathered for the quarter. Excess points at the end of the quarter will convert into a bonus assignment grade.

CodingTree

- Counts the characters in the message and stores the count in an appropriate data structure.
- Initialize a single tree for each character.
- Build the Huffman tree using and efficient data structure.
- Recursively extract the codes from the Huffman tree.

- Encode the text.

Main
- Read text from input file.
- Output codes to text file.
- Output encoded text to binary file.
- Component testing or debugging code.
- Display run statistics.
- Extra test files.

Decode **(Optional)**
- Decoder creates uncompressed character string from a string of bits and the codes map.
- Decoder reads the compressed binary file.
- Decoder reads the codes text file.
- Decoder writes the uncompressed message to a text file.

MyPriorityQueue **(Optional)**
- All methods used are implemented.
- All methods are correct.
- All methods are efficient.

Analysis **(Optional)**
- Each step has the correct worst case runtime big-oh bound.
- Each step's estimate is justified with a brief explanation of the runtime.
- The final runtime estimate is correct.

Extra Points
- Compiles first try.
- All and only the files requested are submitted.
- First graded submission. (That is, not a resubmission after a grade awarded.)
- On time.
- No debugging help.

Tips for maximizing your grade:
- Make sure your classes match the interface structure exactly. I will use my own controller (Main.java) and test files on your code and it should work without changes. Do not change the method name (even capitalization), return type, argument number, type, and order. Make sure all parts of the interface are implemented.
- Only zip up the .java files. If you use eclipse these are found in the "src" directory, not the "bin" directory. I can do nothing with the .class files.
- All program components should be in the default package. If you use a different package it increases the difficulty of grading and thus affects your grade.
- Place your name in the comments at the top of every file. If you are a group of two make

sure both names appear clearly in these comments.