COMP20290
Algorithms

Assignment 2

Huffman Compression

Due end of day Sunday May 9th

Objective

In this assignment, you will build your own utility for compressing text - one you could practically use to compress your own files. Your task for this programming assignment will be to implement a fully functional Huffman coding suite equipped with methods to both compress and decompress text files.

Huffman encoding is an example of a lossless compression algorithm that works particularly well on text but can, in fact, be applied to any type of file. Using Huffman encoding to compress a file can reduce the storage it requires by a third, half, or even more, in some situations.

Huffman's algorithm is an example of a **Greedy algorithm**. It's called greedy because the two smallest nodes are chosen at each step, and this local decision results in a globally optimal encoding tree. In general, greedy algorithms use small-grained, or local minimal/maximal choices to result in a global minimum/maximum.

If you encounter any difficulties with this, make sure to consult the lectures, this excellent Wikipedia article or this detailed overview.

Tasks

1. Task 1: Develop a Huffman tree by hand (15%)
2. Task 2: Code a fully-functional Huffman algorithm (50%)
3. Task 3: Test and analyze your Huffman algorithm with various inputs (35%)

What is provided?

Some helper classes are provided so you can focus on building your Huffman algorithm. In addition, various input data is provided to allow you to test your code. You are welcome to add to this.

Deliverables:

A completed assignment should consist of:

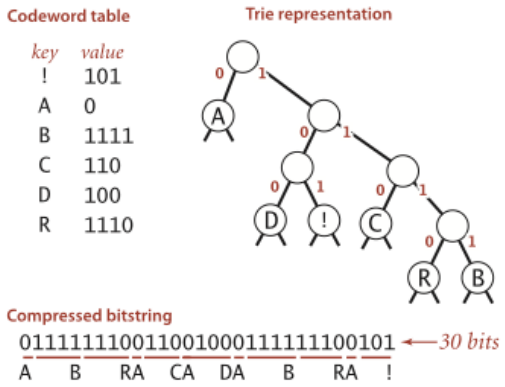
1. a hand-drawn Huffman tree with a codeword table for the given phrase (as pdf)
2. a fully functional Huffman algorithm that can compress and decompress input files
3. a 1 page analysis of your analysis of your algorithm's performance (as pdf)

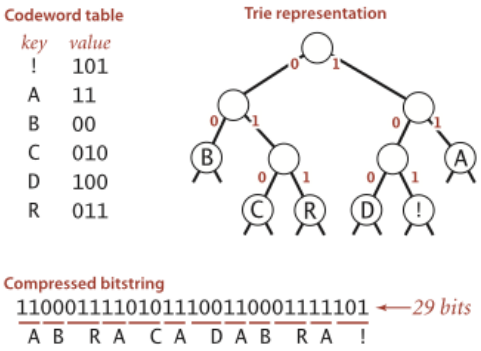
Submission

Submissions are due by 5pm on May 3rd. All of the deliverables should be submitted via GitHub Classroom.

****this assignment can be completed on your own or in groups of 2 or 3. If you are completing this as a group, each person needs to submit the assignment separately with a cover sheet (e.g., pdf) that includes the names and student numbers of the people on the project. This will be taken into consideration when grading your submission.**

Task 1 Code Huffman Tree of phrase by hand





Create a Huffman tree and codeword table for the phrase: “Home is where the heart is”.

You should follow the steps outlined in the lectures to create your Huffman tree by hand which include:

1. Count the characters in the input phrase (including the space character)
2. Build your tree from the bottom up beginning with the two characters with the lowest frequency
3. Merge these into a node / sub-trie with combined weight
4. Continue working through your character frequency table until you reach the root node
5. Encode each character (using either 0 or 1 for left forks in the tree etc.)
6. Write out your codeword table

In summary, the basic idea is that you take the two nodes with the lowest frequency and combine them with an internal-parent-node. The new parent-node takes the combined frequencies of its two sub-children as its frequency and is put back with all of the other nodes. This process is repeated until there is only one node left, which is the root-node. You then create the codes for each character by tracing a path from root to leaf node (aggregating the bits along the way).

Submission: Upload a photo of your hand-drawn character frequency table, your Huffman tree and the resulting codeword table as part of the submission

Task 2: Build your Huffman Compression Suite

For Task 2, you are tasked with building a fully functional Huffman compression algorithm that can compress and decompress input files. Your program should enable a user to compress and decompress files using the standard Huffman algorithm for encoding and decoding.

To complete this task you need to:

1. Create a class called HuffmanAlgorithm which can read in and output files
2. Implement a compress method which will involve:
    a. Building an encoding trie
    b. Writing the trie (as a bitstream) for use in decompression
    c. Use the trie to encode the input byte-stream as a bitstream
3. Implement a decompress method which involve:
    a. Reading in the encoding trie (the one created in the compression stage)
    b. Using this trie then to decompress the bitstream

Compression

The steps you'll take to perform a Huffman encoding of a given text source file into a destination compressed file are:

1. **Read in the input:** you can use the binary input stream helper function.

2**. Count character frequencies**: examine the input file to count the number of occurrences of each character and store them in a data structure (e.g., a map).

3. **Build the Huffman encoding tree**: Build a binary tree where each node represents a character and its count of occurrences in the file. Follow the same process you used for Task 1 to join nodes together. Remember the same text can be encoded in different ways. A **priority** queue can be used to help build the tree along the way.

4. **Build the corresponding codeword table:** Traverse the trie to identify the binary encodings for each character.

5. **Write the trie:** Output the trie you created encoded as a bitstring (Alternatively **w**rite the character count: Write out the count of input characters also encoded as a bitstring)

6. Apply Huffman coding: Re-examine each character in the input and output the encoded binary version of that character to the destination file.

Decompression

Decompressing a file that has been compressed with your Huffman algorithm is a little bit more straightforward and is somewhat the reverse of the compression steps.
You will need to:

1. Read in the input
2. Read the trie that is encoded at the beginning of the bitstream
3. Use this trie to decode the bitstream
4. Output the decompressed characters
5. Use the trie to decode the bitstream

Your finished algorithm should be able to be called from the command line with additional arguments that tell your function whether to compress or decompress, what input file to use for compress and what output file to use to write to e..g., 'java HuffmanCompression compress filename output filename'

Deliverable: Push your Huffman algorithm to GitHub classroom

Task 3: Compression Analysis

Your final task is to test the performance of your Huffman algorithm, benchmark its performance against other algorithms and answer the analysis questions below.

Step 1. Compress the provided text files
Calculate the time to compress and compression ratios for each of the provided files (**and one file of your choosing)**. Include the results in a summary table and upload the resulting compressed files as part of your submission. You can use the tools provided to calculate the compression ratios and time taken to compress them.

Q1. Calculate the compression ratio achieved for each file. Also, report the time to compress and decompress each file.

Step 2. Decompress the files you compressed
Q2. Take the files you have just compressed and decompress them. Report the final bits of the decompressed files and the time taken to decompress each file.

Step 3. Analysis of your results
Assess the results of the above.

Q3. What happens if you try to compress one of the already compressed files? Why do you think this occurs?

Q4. Use the provided RunLength function to compress the bitmap file q32x48.bin. Do the same with your Huffman algorithm. Compare your results. What reason can you give for the difference in compression rates?

Deliverables: A 1-page analysis of your compression / decompression experiments. This can be included as a pdf at the root of your submitted repo or as part of a ReadMe file.

Helper files

The following helper functions have been provided as part of the assignment repo as well as some starter code for your Huffman utility. You do **NOT HAVE TO USE THIS CODE** and are free to choose your own implementations, including different data structures, if you like.

BinaryStdIn - Reads bits from the system

BinaryStdOut - Writes bits to the system

BinaryDump - Allows you to examine the contents of byte-streams and bitstreams while you're testing

HexDump - same as BinaryDump but in more compact Hex form

RunLength - an implementation of RunLength encoding that you can use in Task 3 to benchmark your Huffman algorithm. You can call it from the command line with: **java RunLength - < yourfilename** to compress your chosen file and **java RunLength + < yourfilename** to decompress it.

Helper Data

Several text files have been provided in the github assignment folder that are commonly used to test the effectiveness of compression algorithms. Use them to assess the performance of your implementation of Huffman and benchmark against that of other compression algorithms.

mobydick.txt - Full text of MobyDick

medTale.txt - several sample paragraphs

q32x48.bin - a bitmap

genomeVirus.txt - genetic code

Grading

A Grade

An A grade is appropriate when a project is Excellent in every way. It is expected that a student receiving an A Grade will have fully completed all the assigned tasks to a high degree of perfection, but in places they should also have demonstrated an ability to go beyond the assigned tasks. To obtain an A Grade the student's coding and documentation must be excellent in every respect

B Grade

The B grade is appropriate for a Very Good submission, one provides a thorough and well-organised response to all three tasks. All tasks must be fully completed and the overall submission (report, code and files) should be correct, comprehensive, and professional. The report needs to be very strong and should demonstrate a deep engagement with and understanding of the project. The student must be able to demonstrate a level of analysis that goes above and beyond the average. The submission files should be carefully and properly documented so that the project can be compiled and run without any issues. It is unlikely that a student will merit B grade if there are any material omissions or if there are any coding or compilation or presentation issues. Code should be very well structured and should include detailed and informative comments.

C Grade

The C grade is appropriate for a Good project and should be assigned when the student has submitted an adequate and competent response to the Huffman assignment's three tasks. The student should have implemented all the main tasks correctly and competently. Experiments should be run correctly and suitable data files should have been used. The student should provide a good analysis of the compression tasks and what they have implemented. The submission files should be carefully and properly documented so that the project can be compiled and run without major issues. A student may still be worthy of a C grade even if there are one or two errors or minor omissions. However, if major components are missing or incomplete then a C grade may not be merited. Code should be function correctly to produce a correct and efficient implementation.

D Grade

A D grade is appropriate when a project is Satisfactory in as much as it demonstrates a basic grasp of the subject matter. To obtain a D grade the student needs to have implemented the basic Huffman three tasks but may have done so in a disorganised or inefficient manner. They should have completed most of the tasks but may have turned in some incomplete or incorrect results. Their analysis needs to be satisfactory in the sense that it demonstrates that they have a satisfactory understanding of the technique but may be incomplete or limited in places. If code cannot be compiled or run then a D grade may be appropriate.

E Grade

An E grade denotes are Marginal project submission. Signs that an E grade are appropriate include:

- Incomplete analysis and/or disorganised data.
- An incomplete or very poorly written report with significant typos, poorly presented results, or an inability to demonstrate a clear understanding of the task.
- Incomplete or poorly written code. Code that does not compile and cannot be verified.