CS2710 - Programming and Data Structures Lab

Lab Endsem

Nov 6, 2024

Instructions

- You are expected to solve ALL 4 questions in the lab, using the local computer, C++ language, and g++ compiler. These 4 questions carry equal weight.
- This being a lab Endsem exam:
  - Do not talk with your classmate, do not take code snippets from your other lab assignments, and do not resort to any forms of malpractice.
  - You are only allowed one A4-sized cheat sheet and cppreference.com access.
  - Ask your TA if you need to clarify the question statement; TAs won't discuss any solution.
  - As always, internet access and mobile devices are prohibited in the lab.
- You should submit your code to the course **moodle** on time, i.e., on/before 4.45pm, strictly following the same file name conventions as before. For example, for this Lab EndSem exam consisting of 4 questions, a student with roll number CS23B000 should
  - Name their .cpp files as **CS23B000_ENDSEM_Q1.cpp**, **CS23B000_ENDSEM_Q2.cpp**, **CS23B000_ENDSEM_Q3.cpp**, and **CS23B000_ENDSEM_Q4.cpp**.
  - Put these four .cpp files in a directory named **CS23B000_ENDSEM**
  - Zip this directory into a file named **CS23B000_ENDSEM.zip**
  - Submit only this single .zip file to moodle.

Extra Instructions on Questions:

- Test your code using evaluation scripts before submitting, and **download evaluation scripts in the first 10 minutes** of your lab (after which moodle access may be stopped).
- You are **allowed** to use std/C++ implementation of **any ADT/DS** seen in this course.
- Each question asks for different functionalities of a Huffman Coding System to be implemented. Test cases have been designed to independently test each of the requested functions/commands. Therefore, it is acceptable if some commands are not implemented – **this will fetch partial credit**.
- For each command/function, **write in the answer sheet**, the ADT/DS used and running time complexity. For commands in questions **Q3 and Q4, write also the pseudocode and justify its running time** complexity.

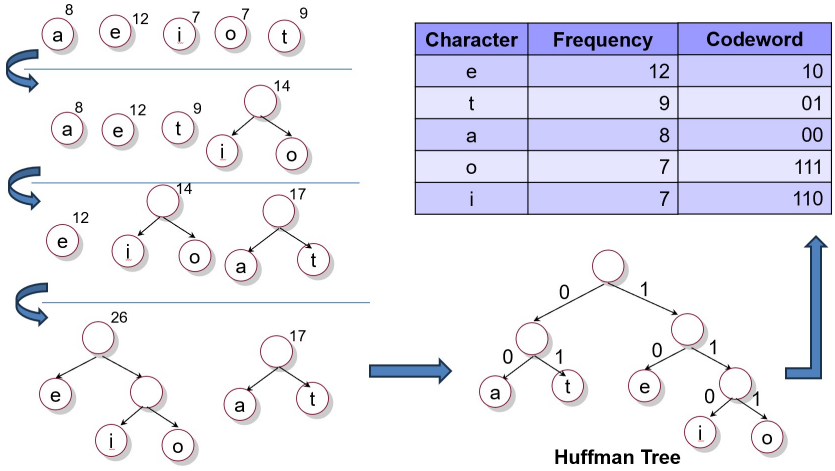
Problem Statement

In this problem, you are tasked with implementing a Huffman Coding System that efficiently encodes text by constructing a binary tree (called Huffman tree) based on character frequencies. Huffman Coding has applications in data compression, as it represents more frequent characters in a text using fewer bits.

Your goal is to write C++ classes/functions that implement Huffman Coding tasks like generating a frequency table, constructing a Huffman Tree, and providing character encodings as well as decodings. The encoder maps each distinct character in the input to a unique **codeword** (a sequence of bits) based on its frequency, with more frequent characters getting mapped to shorter codewords. Let C denote the number of distinct characters in the input text. Then the collection of C codewords is called the **codebook**. Additionally, you should represent the Huffman tree in Newick format to enable conversions between tree structures and the codebook. Note that a **prefix code** is an encoding under which no codeword is a prefix of any other codeword in the codebook. Huffman code is a prefix code.

Background: Huffman (Tree Construction) Algorithm, Optimal Prefix Code, and Tie-Breaking Rule

A binary tree with all C characters at its leaves can be used to derive a (prefix code's) codebook as follows. The codeword of each character c_i at a leaf is found by starting from the root, and recording the path to the leaf, using a 0 for indicating a left branch and 1 for a right branch. Please see figure below for illustration. The **cost of the code** is given by $\sum_{i=1}^{C} d_i f_i$, where d_i is the depth of the leaf with character c_i in the binary tree (i.e., length of the codeword for c_i) and f_i is the number of occurrences (frequency) of c_i in the input text.



Character	Frequency	Codeword
e	12	10
t	9	01
a	8	00
o	7	111
i	7	110

Figure 1: Huffman Algorithm (with tie-breaking rule) and Code Illustration

The above binary tree based encoding yields a prefix code, and the prefix code with optimal (minimum) cost can be obtained using the greedy Huffman algorithm as follows. Huffman algo-

rithm maintains a forest of trees (illustrated in the left of above figure). The weight/frequency of a tree is equal to the sum of the frequencies of its leaves. In each of $C - 1$ iterations, select the two trees, T_1 and T_2, of smallest weight, breaking ties as mentioned below, and form a new tree with T_1 and T_2 as left and right subtrees. At the beginning of the algorithm, there are C single-node trees – one for each character, **constructed in a lexicographic order** (e.g., the single node for **'a' is constructed earlier than for 't'**). At the end of the algorithm, there is one tree, and this is the Huffman tree corresponding to the optimal prefix code.

To break ties when two or more trees in the forest have the same weight, we **choose the tree whose root was constructed earlier in the above algorithm**. Also, T_1 and T_2, the trees in the forest with the two smallest weights in the above algorithm are merged such that – **the tree with the lower of these two weights is set as the left subtree**, and the other tree the right subtree of the merged tree (if both T_1 and T_2 has the same weight, the ties are again broken by setting the tree whose root was constructed earlier as the left subtree).

Summary of Tasks, and Expected Running Time

You are tasked with implementing various aspects of a Huffman Coding System, with each functionality triggered by a specific command and its appropriate inputs. Your implementation will handle 4 main categories of tasks and have expected running time complexity as indicated:

- **Q1: Encoder** — Commands to build frequency tables ($O(M \log C)$ time), and construct Huffman trees ($O(C \log C)$ time).
- **Q2: Decoder** — Commands to derive Huffman trees from codeword tables ($O(CH)$ time) and decode encoded messages ($O(NH)$ time).
- **Q3: Enhanced Encoding** — Commands to further optimize the tree construction ($O(C)$ time) and count distinct Huffman trees ($O(C \log C)$ time).
- **Q4: Length-limited Encoding** — Command to limit maximum codeword length.

In the expected running time complexity indicated above,
C is the number of unique characters in input text/message,
M is the length of the input message in characters,
N is the length of the encoded message in bits, and
H is the height of the Huffman tree.

Constraints on the inputs/parameters

Each command is triggered by its **"Command Name"**, followed by its appropriate **inputs**, and then terminated by **"Quit"** command. For the parameters defined above,

- $2 \leq C \leq 63$ (i.e., 2 to 63 possible characters in message, all of which will be either **lower-case or upper-case alphabets** or **digits (0-9)** or '_' (underscore).
- $2 \leq M \leq 10^7$ characters.
- $2 \leq N \leq 10^5$ bits.

1. [Q1: ENCODER COMMANDS] Implement the 2 commands given below.

Command 1a: `BuildFrequencyTable`

- **Description**: Construct a frequency table from an input string.
- **Input**:
  - A single string containing the text/message.
- **Output**: Each character and its frequency, space-separated, in lexicographical order, with one row of the frequency table per line.

Sample Input:

```
BuildFrequencyTable
ohelloo
Quit
```

Sample Output:

```
e 1
h 1
l 2
o 3
```

Command 1b: `BuildTreeFromFrequencyTable`

- **Description**: Construct a Huffman Tree from a frequency table.
- **Input**:
  - Integer C, the number of rows in the frequency table.
  - Next C lines: Each line contains a character and its frequency, space-separated.
- **Output**: Print the constructed Huffman Tree in Newick format as a single line.
  Beware: Follow the exact tie-breaking rule in Background above to pass test cases.

Sample Input:

```
BuildTreeFromFrequencyTable
4
e 1
h 1
l 2
o 3
Quit
```

Sample Output:

```
(o,(l,(e,h)));
```

2. [Q2: DECODER COMMANDS] Implement the 2 commands given below.

Command 2a: `DeriveHuffmanTree`

- **Description**: Constructs a binary (Huffman) Tree consistent with a given codebook of a prefix code.
- **Input**:
  - Integer C, the number of rows in the codebook of a prefix code.
  - Next C lines: Each line contains a character and its codeword, space-separated.
- **Output**: Newick-formatted binary (Huffman) tree reconstructed from the codebook.

Sample Input:

```
DeriveHuffmanTree
4
e 110
h 111
l 10
o 0
Quit
```

Sample Output:

```
(o,(l,(e,h)));
```

Command 2b: `DecodeMessage`

- **Description**: Decode a given binary message using a Huffman Tree, producing the original message string.
- **Input**:
  - A single string representing the Newick-formatted Huffman Tree.
  - Next line: A binary string representing the encoded message to be decoded.
- **Output**: Outputs the decoded message as a string in a single line.

Sample Input:

```
DecodeMessage
(o,(l,(e,h)));
0111110101000
Quit
```

Sample Output:

```
ohelloo
```

3. [Q3. ENHANCING THE ENCODER] Implement the 2 commands given below.

Command 3a: `GenerateTreeEnhanced`

- **Description**: Construct in **linear time** a Huffman tree using the non-decreasingly **sorted** character frequencies provided as input. Your code should take advantage of the sorted order to achieve a time complexity of $O(C)$, improving upon the usual $O(C \log C)$ running time on unsorted input. To claim non-zero marks for this question, linear running time of the code is a must, and you should also write your pseudocode and justify its linear running time in your answer sheet.
- **Input**:
  - Integer C, the number of rows in the frequency table.
  - Next C lines: Each line contains a character and its frequency, space-separated (Note: Assume that the input frequency table is sorted, so that lower frequency characters appear first, with ties broken by lexicographic order, as in Example below).
- **Output**: Newick-formatted Huffman Tree, reconstructed from sorted frequency table in $O(C)$ time.
  Beware: Continue to follow the exact tie-breaking rule in Background above to pass test cases.

Sample Input

```
GenerateTreeEnhanced
4
e 1
h 1
l 2
o 3
Quit
```

Sample Output

```
(o,(l,(e,h)));
```

Command 3b: `CountHuffmanTrees`

- **Description**: It is possible that many different Huffman Trees of the same optimal cost can be constructed from a given frequency table of characters (if we abandon our tie-breaking rules described in Background above). A Huffman tree is different from another Huffman tree if both trees differ in the codeword assigned to at least one character. This command prints the number of distinct Huffman trees in expected time complexity $O(C \log C)$.
- **Input**:

  - Integer C, the number of rows in the frequency table (i.e., the number of unique characters).
  - Next C lines: Each line contains a character and its frequency, space-separated.
- **Output**: Number of distinct Huffman Trees (**modulo** $(10^9 + 7)$, to avoid integer overflow), each of which yields an optimal prefix code for the same input frequency table.

Sample Input1

```
CountHuffmanTrees
4
e 1
h 1
l 2
o 3
Quit
```

Sample Output1

```
8
```

Explanation1

`(o,(l,(e,h)));` `(o,(l,(h,e)));` `(o,((e,h),l));` `(o,((h,e),l));` are four distinct Huffman trees with the same optimal cost of the code. The other four follow simply from these trees by swapping the left and right subtrees of the root.

Sample Input2

```
CountHuffmanTrees
7
h 1
e 1
l 3
o 2
r 1
w 1
d 1
Quit
```

Sample Output2

```
5760
```

4. [Q4. LENGTH-LIMITED HUFFMAN CODE] Huffman algorithm provides the binary tree corresponding to the optimal cost prefix code, but its height can be quite large (even close to C) in some real-world scenarios. We may want to limit the tree height or equivalently length of the codewords in such settings to a maximum of D. Can you find a binary tree based prefix code with optimal (minimum) cost, subject to the constraint that no codeword is longer than D bits?

Command 4: `GenerateLengthLimitedHuffman`

- **Description**: Generate a length-limited Huffman tree based codebook, which optimizes the prefix code cost while ensuring that no codeword exceeds a given length of D bits.
- **Input**:
  - Input messsage/text given as a string (using which the frequency table of characters and cost of code can be calculated).
  - Integer D, the maximum possible codeword length.
    (Note: You can assume that $\lceil \log_2 C \rceil \leq D \leq D_{Huff}$, where D_{Huff} is the height of the optimal Huffman tree without any codeword length constraint, and $\lceil \log_2 C \rceil$ is a lower bound on the number of bits to represent the C unique characters in the text.)
- **Output**:
  - First C lines - length-limited codebook, with each line being the space-separated character and its codeword (expected to be the optimal length-limited Huffman code).
  - Cost of your output codebook, as shown in the example below.
- **Evaluation:** If you **cannot design an efficient algorithm** to exactly solve this problem, you are welcome to **implement heuristics** that get you as close to the length-limited optimal cost as possible (the closer you get, the more marks you will score, after we verify that your codebook is a valid prefix code). Note: Your vs. test-cases' codebook needn't match due to the heuristics and multiple codebooks with same cost.

Sample Input

```
GenerateLengthLimitedHuffman
abbccccddddddddeeeeeeeeeeeeeeee
3
Quit
```

Sample Output

```
e 0
a 100
b 101
c 110
d 111
cost: 61
```

Explanation
Constructing a Huffman tree with the usual greedy (length-unconstrained) algorithm results in $D_{Huff} = 4$ and cost 56. But we are constrained here by $D = 3$. Shown above is a possible codebook with length-limited optimal cost 61 (it satisfies length limit and prefix property).