SOFTWARE 2 SEMINAR

HUFFMAN CODING

Week 5 – Seminar 4

Problem: Huffman Coding Compression Algorithm

Huffman coding (also known as Huffman Encoding) is an algorithm for doing data compression, and it forms the basic idea behind file compression. This seminar addresses the variable-length encoding, uniquely decodable codes, prefix rules, and Huffman Tree construction.

For the reminder of the seminar, the set of distinct characters/punctuation/symbols contained within the text to be encoded/compressed is called the alphabet. We will assume every character is a sequence of 0's and 1's and is stored using 8-bits (e.g. Extended ASCII). This is known as "fixed-length encoding" as each character uses the same number of fixed-bit storage.

Given a text, how to reduce the amount of space required to store a character?

The idea is to use variable-length encoding. We can exploit the fact that some characters occur more frequently than others in a text to design an algorithm that can represent the same piece of text using fewer bits. In variable-length encoding, we assign a variable number of bits to characters depending on their frequency in the text. So, some characters might take a single bit, some might take two bits, some might be encoded using three bits, and so on. The problem with variable-length encoding lies in its decoding.

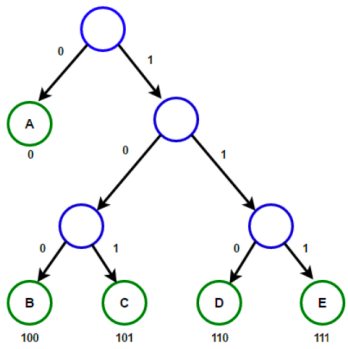
To prevent ambiguities in decoding, we will ensure that our encoding satisfies the "prefix rule", which will result in "uniquely decodable codes". The prefix rule states that no code is a prefix of another code. By code, we mean the bits used for a particular character.

Let's consider the string `aabacdab`. It has 8 characters in it and uses 64–bits storage (using fixed-length encoding). The frequency of characters $\{a = 4, b = 2, c = 1, d = 1\}$. Let's use the encoding {a='0', b='11', c ='100', d='011}, to encode the string `aabacdab`. The result is the encoding `00110100011011` (that is `0|0|11|0|100|011|0|11`).

In the above example, 0 is the prefix of 011, which violates the prefix rule. This breach of the rule leads to possible ambiguity; for example, the resulting string can be decoded as adacdab, aabacadb or finally the expected string aabacdab. Satisfying the prefix rule removes ambiguity. For example, the encoding {a='0', b='10', c ='110', d='111} satisfies the prefix rule and can be used to encode/decode the string without any ambiguity.

Huffman Coding:

The technique works by creating a binary tree of nodes. A node can be either a leaf node (each containing a character) or an internal node (no specific values). As a common convention, bit 0 represents following the left child, and a bit 1 represents following the right child. A finished tree has $n > 1$ leaf nodes and $n - 1$ internal nodes. This means that every internal node has exactly two children. An example of a Huffman Coding tree is given below.



Building the Huffman tree

To build a Huffman tree, we use a priority queue, where the node with the lowest frequency has the highest priority. This is a queue of binary trees, each tree with an associated priority. Below are the complete steps to build the tree:

1. Create a leaf node for each character (this a binary tree with only the root) and add them to the priority queue. At the end of this step, the queue contains n binary trees, and each tree is a single node (the root) containing a character.
2. While there is more than one node in the queue:
    - Remove the two nodes of the highest priority (the lowest frequency) from the queue.
    - Create a new internal node with these two nodes as children and a frequency equal to the sum of both nodes' frequencies.
    - Add the new node to the priority queue. At the end of this step, the queue contains trees of different height. After the first iteration, the queue contains $n - 2$ trees consisting of a single node, and one tree consisting of 1 root and 2 leaves.
3. The remaining node is the root node, and the tree is complete.

Part 1: Building a tree

Using pseudo code, write an algorithm to build a Huffman tree where the input is a dictionary mapping a symbol from an alphabet with its frequency in the text to be encoded. Let's n be the number of characters/symbols in the alphabet (or the size of the dictionary). What is the complexity of your algorithm as a function of n?

Part 2: Decoding

Write an algorithm that takes an encoded string (containing only 0s and 1s) and the corresponding Huffman tree as inputs and returns the decoded string. The algorithm should raise an error if the encoded string is corrupted/invalid; that is, it contains too few bits. For example, given the Huffman tree described earlier, decoding the string 1110 should return EA, whereas decoding 1111 should raise an error (end on an interior node).

The idea behind the decoding:

- Start at the root of the Huffman tree
- Starting at $i = 0$ and while you are not passed the last character in the encoded string,
  - If the character i in the encoded string is 0, go to the left child. If it is 1, go to the right child. If you are at a leaf, add the character stored in the leaf to the decoded text and go to the root of the tree.
  - Go to the next character in the encoded string ($i = i + 1$)
- If the encoded string does not terminate at a leaf, the message has been corrupted.

What is the complexity of your algorithm? For the complexity, we consider s the length of the text to be decoded and n the number of nodes in the Huffman tree.

Part 3

Write an algorithm that takes a Huffman tree as input and returns a mapping of the characters with their respective code (the path from the root to the leaf containing that character). For example, given the tree shown above, the mapping should be:

$$\{a = '0', b = '100'. c = '101', d = '110', e = '111'\}$$

What is the complexity of your algorithm (using Big O notation), where n is the number of characters present in the tree? In the example above, $n = 5$.

Part 4: Encoding

Write an algorithm that takes a string containing the text to compress and the corresponding Huffman tree as inputs, and returns the encoded text. You may want to reuse some of the algorithm you wrote earlier. What is the complexity of your algorithm? For the complexity, we consider s the length of the text and n the number of distinct characters/symbols in the text.