Readings

- Lecture Notes Chapter 15: Huffman Coding
- Lecture Notes Chapter 16: Graph Representations & BFS

Problems

Problem 1
What are pros and cons of Huffman Coding?

Solution

Pros: Huffman encodings represent only the characters that are contained in the text, without wasting space in the encoding length for characters that are not present. Additionally, characters that occur most take less bits to encode each time.

Cons: It does not help with longer patterns in the text. Storing the symbol table takes up additional space. Therefore it isn’t appropriate for very small files where the table size would be significant. Encoding a text requires a preprocessing step to generate the table, does not work with a data stream.

Problem 2

Construct an optimal Huffman coding for the following alphabet and frequency table:

<table>
<thead>
<tr>
<th>Character</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>0.4</td>
<td>0.3</td>
<td>0.15</td>
<td>0.1</td>
<td>0.05</td>
</tr>
</tbody>
</table>

What is the average encoded character length for the above encoding?

Solution

The following tree would be produced:

```
    /
   /\n  A /\B
  C /\E
    D
```

\[
l_{avg} = \sum_{c \in R} d(c) \times freq(c) = 0.4 \times 1 + 0.3 \times 2 + 0.15 \times 3 + 0.1 \times 4 + 0.05 \times 4 = 2.05
\]

Problem 3

You have an alphabet with \( n > 2 \) letters and frequencies. You perform Huffman encoding on this alphabet, and notice that the character with the largest frequency is encoded by a 0. In this alphabet, symbol \( i \) occurs with probability \( p_i \), where \( p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n \).

Given this alphabet and encoding, prove that there does not exist an assignment of probabilities to \( p_1 \) through \( p_n \) such that \( p_1 < 1/3 \).
Solution

We can use a simple proof by contradiction. Assume that there exists an assignment such that \( p_1 < \frac{1}{3} \). Look at the last step of the Huffman algorithm, that is the step when our two final nodes are merged into one node. Let these two final nodes be called \( x \) and \( y \). Because character 1 has an encoding length of 1, it must have been included in this step. WLOG let \( x \) be our single character 1. This is the final step of Huffman, so we know that \( p_x + p_y = 1 \). Given our assumption that \( p_1 < \frac{1}{3} \), this tells us that \( p_y = 1 - p_x \to p_y > \frac{2}{3} \).

We know that because \( n > 2 \), \( y \) must be a group containing at least 2 characters. So, examine the time when \( y \) was created. Let the nodes which combined to \( y \) be called \( a \) and \( b \). We know that \( p_a + p_b > \frac{2}{3} \), which implies that \( \max\{p_a, p_b\} > \frac{1}{3} \). Here we have reached contradiction. We know Huffman uses the smallest two nodes at all steps, but at the step \( y \) was created, node \( x \) was still available where \( p_x < \frac{1}{3} \). Thus, it would have been chosen instead of \( \max\{a, b\} \). This contradiction then proves the original claim.

Problem 4

Design an algorithm to determine whether or not an undirected graph has a cycle.

Solution

We can perform a BFS traversal and just keep track of which elements have been seen. For example, we can store vertices we have seen in a set and just track whether or not any previously seen node is encountered again by checking if it is in the set. Since we are simply doing a BFS, this algorithm runs in linear time.

Additional Problems

Problem 5

Design an algorithm to determine whether or not a connected undirected graph has a cycle in \( O(n) \) time.

Solution

Perform a BFS traversal and terminate early if you explore at least \( n \) edges. Recall that a undirected connected acyclic graph is by definition a tree. We know that a connected graph on \( n \) nodes is a tree iff it has exactly \( n - 1 \) edges. An additional edge would signify that two nodes are connected by two independent paths, implying the existence of a cycle. This algorithm will take \( O(n) \) time to check each vertex and \( O(n) \) to check each edge (since we are checking at most \( n \) edges). Thus, the running time is \( O(n) \).