Readings

- Lecture Notes Chapter 15: Huffman Coding
- Lecture Notes Chapter 16: Graph Traversals: BFS

Review: Huffman Coding

The motivation behind Huffman Coding is to encode and decode characters as bits, minimizing the average bits per letter (ABL). Furthermore, we seek a prefix-free code, where no encoding is a prefix of another — implying that a bit sequence can be parsed and decoded without any ambiguity.

The Huffman algorithm is a greedy algorithm that does this. Given a set of characters and their frequencies, the algorithm outputs an encoding by repeatedly merging the 2 nodes with the smallest frequency values until only one node remains. This one node is the root of the Huffman tree, whose leaves are characters and each root-to-leaf path is an encoding of that character. Furthermore, this tree is a full binary tree, where each internal node has exactly 2 children. Therefore, the Huffman algorithm produces an optimal and prefix-free encoding that minimizes the ABL.

The running time of the Huffman algorithm is $O(n \log n)$ if we utilize a min-heap to find the 2 nodes with minimum frequency in each step, as seen in the pseudocode. This is because at each step, we perform a constant number of EXTRACT-MIN and INSERT operations, which take $O(\log n)$ time, and we repeat this for $O(n)$ iterations.

Review: Graph Representations

Let $G = (V, E)$, where $|V| = n$ and $|E| = m$. In other words, $G$ contains $n$ vertices and $m$ edges.

Adjacency Matrix

We can represent $G$ with an $n \times n$ matrix $A$, where $A[i][j] = 1$ if there is an edge $(i, j)$ and 0 otherwise. The primary advantage of an adjacency matrix is that we can check whether or not there is an edge between two vertices in $O(1)$ time since all we have to do is index into the matrix. On the other hand, the matrix uses $\Theta(n^2)$ space, and it takes $\Theta(n)$ time to enumerate the neighbors of a vertex $v$ since we must iterate through an entire row in the matrix.

Adjacency List

We can also represent $G$ as an adjacency list, where we have an array $A$ of $|V|$ lists such that $A[u]$ contains a LinkedList of vertices $v$, such that for every vertex $v_i$ in $v$, $(u, v_i) \in E$. (Note that if we have a directed graph, we can store two LinkedLists at index $u$, one for $u$’s in-neighbors and one for $u$’s out-neighbors.) The primary advantage of an adjacency list is that we only use $\Theta(n + m)$ space, which is better than the $\Theta(n^2)$ space usage of an adjacency matrix especially if $m \ll n^2$. Furthermore, it only takes $O(\text{deg}(v))$ time to enumerate the neighbors of a vertex $v$, but on the other hand, checking whether $(u, v) \in E$ also takes $O(\text{deg}(v))$ time since we have to traverse the LinkedList. In CIS 121, we usually use adjacency lists.
Review: BFS (Breadth First Search)

At a high-level, in BFS, we explore the graph in “layers,” so we explore “wide” before exploring “deep.” We begin at a node \( v \) (layer 0); then explore the children of \( v \) (layer 1); then the children of the nodes in layer 1 (which make up layer 2); and so on. In other words, we explore all nodes at layer \( L_i \) before exploring any nodes at layer \( L_{i+1} \). Because we explore the graph in layers, as seen in the pseudocode, we use a queue to implement this algorithm since it is FIFO. Note that the output of BFS is a BFS tree, and that as written, BFS is not guaranteed to visit every node in the graph.

The running time of BFS is \( O(n + m) \). This is because we add and remove each node from the queue at most once, and for each node, we only examine its neighbors.

Problems

Problem 1

Construct an optimal Huffman coding for the following alphabet and frequency table \( S \):

<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 ABL, or average bits per letter, for this encoding?

Solution

The following tree \( T \) would be produced:

\[
\begin{array}{c}
/ \\
A / \backslash \\
B / \backslash \\
C / \backslash \\
E D
\end{array}
\]

\[
\text{ABL}(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x) = 0.4 \cdot 1 + 0.3 \cdot 2 + 0.15 \cdot 3 + 0.1 \cdot 4 + 0.05 \cdot 4 = 2.05
\]

Problem 2

What are the pros and cons of Huffman Coding?

Solution

Pros: Huffman codings represent only the characters contained in the text, without wasting space in the encoding length for characters not present. The simplest method to encode a text of \( n \) characters would be to use \( \lceil \lg n \rceil \)-bits, resulting in a fixed number of bits per character. Although we cannot reduce the number of bits used per encoding, Huffman optimizes this method because characters that occur more frequently take less bits to encode, minimizing the average bits per letter (ABL).

Cons: Since storing the symbol table takes space, Huffman codings may not be appropriate for small files where this size would be significant. Furthermore, encoding a text requires pre-processing to generate the table, which is ineffective given a data stream. Similarly, Huffman codings are not adaptive compression schemes. For example, given an alphabet with 4 characters and a text where the first half contains 2 characters equally frequently and the second half only contains the other 2 characters equally frequently, Huffman would assign 2 bits per letter. Instead, we could “adapt” by starting out with an encoding that represents just the 2 characters in the first half, which would only require 1 bit per character, and then...
change the encoding to represent just the 2 characters in the second half, which would again only require 1 bit per character. In other words, with some overhead, adaptive compression schemes could lead to an improvement over Huffman by essentially viewing the first half and second half as separate texts.

**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 just a 0. In this alphabet, symbol \( i \) occurs with probability \( p_i; p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n \).

Given this alphabet and encoding, does there exist an assignment of probabilities to \( p_1 \) through \( p_n \) such that \( p_1 < \frac{1}{3} \)? Justify your answer.

**Solution**

There does not exist an assignment of probabilities to \( p_1 \) through \( p_n \) such that \( p_1 < \frac{1}{3} \). Assume for the sake of contradiction that there exists an assignment such that \( p_1 < \frac{1}{3} \). Consider the last step of the Huffman algorithm when our two final nodes are merged into one node. Let these two final nodes be \( x \) and \( y \). Because character 1, which has the highest frequency, has an encoding length of 1, it must have been merged via this step. WLOG, let \( x \) be this character 1. Since this is the final step of Huffman, we know \( p_x + p_y = 1 \). From our assumption that \( p_1 < \frac{1}{3} \), we know \( p_y > \frac{2}{3} \).

Because \( n > 2 \), \( y \) must be a node representing at least 2 characters. Consider the time when \( y \) was created. Let the nodes that were merged to become \( y \) be \( a \) and \( b \). We know \( p_a + p_b > \frac{2}{3} \), which implies that \( \max\{p_a, p_b\} > \frac{1}{3} \). Here, we have reached a contradiction. Huffman always merges the two smallest frequency nodes, but when \( y \) was created, node \( x \) with \( p_x < \frac{1}{3} \) was still available and unmerged. Hence, \( x \) would have been chosen instead of \( \max\{a, b\} \). Therefore, via contradiction, we have proved that the original claim is false and thus that there does not exist an assignment of probabilities as specified.

**Problem 4**

Design an \( O(n + m) \) algorithm to find the shortest path between nodes \( u \) and \( v \) in a connected, unweighted graph.

**Solution**

**Algorithm:** Run BFS starting from \( u \); for each node \( x \) that is visited, maintain a “parent” pointer to its parent node (the node we visit \( x \) from). Once we visit \( v \), stop and backtrack through these parent pointers until we reach \( u \), outputting this path as the shortest path. For example, by backtracking, we could see that \( v \)’s parent was \( d \), \( d \)’s parent was \( c \), and \( c \)’s parent was \( u \), so the path outputted would be \( u - c - d - v \).

**Proof of Correctness:** The correctness of this algorithm follows from the lemma that BFS outputs the shortest paths in an unweighted graph. That is, for a vertex \( v \) in layer \( L_i \), \( i \) represents the length of the shortest path from the source \( u \) to \( v \). We prove this lemma via strong induction on \( i \):

**Base Case:** If \( i = 0 \), then by design, \( L_0 \) only contains the source vertex \( u \). The shortest path from \( u \) to \( u \) has length 0, so the claim holds.

**Induction Hypothesis:** Assume that for all vertices \( v \) such that \( v \) is in layer \( L_k \), where \( k < i \), that \( k \) is the length of the shortest path from \( u \) to \( v \).

**Induction Step:** We want to show that the claim holds for some arbitrary vertex \( v \) in layer \( L_{k+1} \). By design, \( v \) was discovered from another vertex \( w \), such that there is an edge between \( w \) and \( v \) and \( w \) is in layer \( L_k \). By IH, we know that the shortest path from source \( u \) to vertex \( w \) has length \( k \), so by adding the edge from \( w \) to \( v \), we get a path of length \( k + 1 \) from source \( u \) to vertex \( v \). We want to show that this path is a shortest path.
from $u$ to $v$. Assume for the sake of contradiction that some other path $u - \cdots - w' - v$ is the true shortest path, meaning that it must have length $l \leq k$ and that $w'$ is reachable from $u$ by a shortest path with length less than or equal to $k - 1$. Thus, we know that vertex $w'$ is in some layer $L_{l-1}$, where $l-1 \leq k-1$. By design, this means that $w'$ was removed from the BFS queue before $w$, implying that when we processed $w'$, we would have discovered $v$. However, this means we would have added $v$ to layer $L_l$, and since $l \leq k$, we have reached a contradiction because we stated that $v$ was an arbitrary vertex in layer $L_{k+1}$. Therefore, we have shown that the length of the shortest path from $u$ to $v$ is in fact $k + 1$, completing our Induction Step and thus our proof.

From the lemma, we know that BFS correctly computes the shortest paths from the source in an unweighted graph. In our algorithm, note that once we visit $v$, we stop and backtrack through its parent pointers until we reach $u$, so we just backtrack through the layers to yield the shortest path from $u$ to $v$. Hence, outputting this path correctly yields the shortest path from $u$ to $v$.

**Runtime Analysis:** Running BFS takes $O(m + n)$ time, and to output the shortest path, we backtrack no more than $O(n)$ times since the longest path in a graph has $n - 1$ edges. Therefore, our algorithm runs in $O(m + n)$ time.