A Quick Introduction to Greedy Algorithms

Throughout the rest of the course, we will be discussing a fundamental paradigm called greedy algorithms. Much of these notes are adapted from CLRS Chapter 16.

**Definition** (Greedy Algorithms). A greedy algorithm obtains an optimal solution to a problem by making the choice that seems ‘the best’ at the moment. It is a heuristic strategy that does not work all of the time, yet for certain problems, it produces an optimal solution.

Greedy algorithms show up in many parts of computer science. We will see this week how we can use greedy algorithms to perform optimal data compression (Huffman’s Algorithm) and we will soon see how greedy algorithms can be used to find unique graph properties (Dijkstra’s Algorithm for shortest path and Prim’s/Kruskal’s Algorithms to find the minimum spanning tree).

**Greedy-choice Property**

The key ingredient to greedy algorithms is the greedy-choice property. This properties states that we can assemble a globally optimal solution by making locally optimal choices. This means that when we are considering a choice in our problem, we will always make the choice that is the best in our current situation without considering any future problems that we may encounter.

You can think of this as a ‘bottoms up’ approach. Greedy algorithms will solve sub problems one by one, choosing what is best at the current iteration, until it finds a globally optimal solution for the entire problem. For any greedy algorithm to be valid, we need to show that a greedy choice at each step yields a globally optimal solution. We can do this with the exchange argument.

**Definition** (The exchange argument). We first examine some globally optimal solution to our problem. We want to show how to modify this solution to substitute a greedy choice for some other choice in the problem that results in a similar but smaller sub problem. If we can show that the optimal solution to our problem includes our greedy choice along with the same optimal solution to a smaller subproblem, then we can ensure our greedy solution is correct.

If you want to learn more about greedy algorithms, please read CLRS Chapter 16.1 and 16.2 for a more in depth analysis.
Huffman Encoding

Huffman coding is a common technique used to losslessly compress text. It uses relative character frequency to encode text such that the least possible space is taken on average. In this lab we will be focusing on encoding text into binary, though later you will learn other methods.

Encoding Implementation

Characters are encoded by runs of bits of non-constant length. The encoding is prefix free, which means that there doesn’t exist a character with a binary representation that is the prefix of another character’s representation. We represent an encoding as a trie, with each leaf representing a character, and the path from the root to the leaf describing its encoding. Each edge in the tree is assigned either 1 or 0. By convention, each left edge is 0. We can construct such a tree as follows:

As input, assume that we have access to the relative probabilities with which each character appears. For a given text to compress, this can be generated in a preprocessing step before the compression. Treat each character as a tree of one node, with weight equal to its frequency, and construct a min-heap of the trees. While the tree is not empty, we preform the following steps.

1. Remove the lowest frequency item from the heap (tree A), then remove the new lowest frequency item (tree B).
2. Construct a new tree by creating a root node with two children: A as the left child, and B as the right child.
3. Add this tree back into the heap, with weight equal to the combined weights of A and B.

This process is complete when only one tree is left in the heap, our final Huffman coding tree.

Encoding Length

When transmitting an Huffman coded file, both the trie-backed encoding and the compressed contents have to be transmitted.

Relative to single character encoding schemes, Huffman’s algorithm produces an optimal encoding, that is, the compressed contents of the text will be the shortest length possible. Using the frequency with which each character appears, and the number of bits needed for each compressed character, we can find the length of the compressed text.

Definition 1. The number of bits to represent a character $c$ is equal to depth of the leaf representing it, $d(c)$ in the trie. The length, or number of bits to represent the entire text is therefore equal to the following expression.

$$L = \sum_{c \in R} d(c) \ast freq(c)$$

Loosely speaking, Huffman Coding makes the term $d(c) \ast freq(c)$ vary less between characters than it otherwise would, and therefore minimizes this summed over all characters. Each character is represented with brevity consistent with its relative importance, or frequency, in the text.

Decoding Implementation

Because the representation of the encoding is included, decoding is simple. While processing the text, follow bit by bit through the trie until a leaf node is reached, then output that character.

1. This direction of this step is convention; the reverse is just as valid, but for this course we will expect you to abide by this standard.
Problems

Problem 1. What are pros and cons of Huffman Coding?

Solution.

Pros: Huffman codings represents 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:

```
    /
   / \
  /   \
 /     \
 |   C   |
 |   /   |
 | / D   |
 E
```

Thus, the average encoded character length is:

\[
    l_{avg} = \sum_{c \in R} d(c) \ast freq(c) = .4 \ast 1 + .3 \ast 2 + .15 \ast 3 + .1 \ast 4 + .05 \ast 4 = 2.05
\]

Problem 3. Construct an alphabet \( A \) with frequencies such that in an optimal Huffman coding there exist at least two encodings of length exactly \( (n - 2) \), where \( n \) is the size of the alphabet. \( n \) must be at least 5.

Solution. Multiple solutions exist. One possible is:

<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>0.6</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
<td>0.1</td>
</tr>
<tr>
<td>Encoding</td>
<td>1</td>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
</tr>
</tbody>
</table>

Represented as a tree as:

```
    /
   / \
  /   \
 /     \
|   C   |
|   /   |
| / D   |
B
```

Problem 4. 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 \( p_i; p_1 \geq p_2 \geq p_3 \geq \ldots \geq p_n \).

Given this alphabet and encoding, is it true or false that there exists an assignment of probabilities to \( p_1 \) through \( p_n \) such that \( p_1 < \frac{1}{3} ? \)
Solution. This claim is false. 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 $p_y = 1 - p_x \rightarrow p_y > \frac{2}{3}$.

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 and unpaired. $p_x < \frac{1}{3}$, so it would have been chosen instead of $\max\{a, b\}$. This contradiction then proves the original claim.

Problem 5. Your task is to connect $n$ ropes with a minimum cost. You are given $n$ ropes of different lengths, and you need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths. You need to connect the ropes with minimum cost.

For example if we are given 4 ropes of lengths 4, 3, 2 and 6. We can connect the ropes in following ways.

1. First connect ropes of lengths 4 and 3. Now we have three ropes of lengths 2, 7, 6.
2. Now connect ropes of lengths 2 and 7. Now we have two ropes of lengths 6 and 9.
3. Finally connect the two ropes and all ropes have connected.

Total cost for connecting all ropes is $7 + 9 + 15 = 29$. Give an algorithm which always finds the minimum cost of attaching the rope.

Solution. If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in total cost. Therefore, the idea is to connect smallest two ropes first and recur for remaining ropes. This is a very similar principle as in Huffman encoding. We put smallest ropes down the tree so that they can be repeated multiple times rather than the longer ropes.

Following is complete algorithm for finding the minimum cost for connecting $n$ ropes. The total runtime is the same as Huffman, so $\mathcal{O}(n\lg(n))$. Let there be $n$ ropes of lengths stored in an array $\text{len}[0...n-1]$

1. Create a min heap and insert all lengths into the min heap.
2. Do following while number of elements in min heap is not one.
   (a) Extract the minimum and second minimum from min heap
   (b) Add the above two extracted values and insert the added value to the min-heap.
   (c) Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
3. Return the value of this total cost