Huffman Codes

https://cs.pomona.edu/classes/cs140/
Outline

Topics and Learning Objectives
• Introduce Huffman Codes for compression

Exercise
• None
Huffman Codes

- This will be our final greedy algorithm / application
- Huffman Codes are used for compression

- In general they can be thought of as:
  - A mapping of some set of characters/symbols to binary strings

- For example: let’s encode the letters [a-z] and {., ?, !, ;, :}.
- How many bits would you use?
- Does this type of encoding sound familiar at all?
Huffman Codes

• In general we use $\Sigma$ to represent the set of characters.
• Let $\Sigma = \{A, B, C, D\}$
• What is one possible binary encoding?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
</tbody>
</table>

• How many bits does it take to store 100 characters?
Huffman Codes

Can we do better than this fixed-length encoding (use fewer bits)?

<table>
<thead>
<tr>
<th>Σ</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Encoding</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>(Bad) Variable Encoding</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

What does the string 001 encode?

- AB: 001
- CD: 101
- AAD: 001
Huffman Codes

The problem with this encoding is called prefixing.

<table>
<thead>
<tr>
<th>$\Sigma =$</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Encoding</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>(Bad) Variable Encoding</td>
<td>0</td>
<td>01</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

- This is **not** a prefix-free encoding.
- Problem: we don’t know where one character ends and the next begins.
- Solution: ensure that the encoding is prefix-free.
### Example Prefix-Free Encoding

<table>
<thead>
<tr>
<th>$\Sigma =$</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fixed Encoding</strong></td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td><strong>Prefix-free Encoding</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Prefix-Free Encoding

Now, we know exactly when one character ends and another starts.

Why would this be a good idea?
• What if we needed to store a bunch of A’s but only a few C’s?

<table>
<thead>
<tr>
<th>$\Sigma =$</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Encoding</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Prefix-free Encoding</td>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
</tbody>
</table>
Example Prefix-Free Encoding

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed</td>
<td>00</td>
<td>01</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>Prefix-free Encoding</td>
<td>0</td>
<td>10</td>
<td>110</td>
<td>111</td>
</tr>
<tr>
<td>Frequency</td>
<td>60%</td>
<td>25%</td>
<td>10%</td>
<td>5%</td>
</tr>
</tbody>
</table>

What are the average bit lengths for these two encodings?
Discovering the Best Encoding

Let’s think of Huffman Codes as trees. $\Sigma = A, B, C, D$

Fixed Encoding
\{00,01,10,11\}

First Variable Encoding
\{0,01,10,1\}

Prefix-Free Encoding
\{0,10,110,111\}
Huffman Codes as Trees

• Go to left child on a ‘0’
• Go to right child on a ‘1’
• For each symbol in $\Sigma$, exactly one node should be labeled $x$
• Prefix-free encoding require all labeled nodes to be leaves
• Trees are just a tool for helping us construct optimal encodings
• Decode: follow the input string until you reach a leaf
• Encode($x$): the path followed from the root to $x$
• The encoding length of $x$ is the same as its depth
Decode the string: 0110111
Huffman Codes

Problem: *how do we choose/design our encodings?*

- Input: a set of symbols $\Sigma$ and their probabilities/frequencies $p_i$
- Notation: if $T$ is a tree with leaves as symbols of $\Sigma$, then let
  \[ L(T) = \sum_{i=1}^{\lfloor \Sigma \rfloor} p_i * \text{depth}_i \]
- $L(T)$ is the average encoding length
- The output of our algorithm will be a binary tree $T$ that minimizes $L(T)$
Huffman’s Algorithm (compression)

Huffman’s approach is the start at the leaves and build the the tree bottom-up
Which Tree is Better?

It depends on the frequencies!
Huffman’s Algorithm

• We’re building from the leaves up.
• How do we know which two symbols we should merge?
• How does the final encoding length of a given symbol in $\Sigma$ relate to the number of merges it experiences?

• Each merge adds one node to the path from the root to $x$!
• So, how do we minimize the weighted average encoding length?
• Huffman’s Greedy Criteria: Merge the least frequent characters first.
How do we compare nodes after a merge?

Iteration 1

A (0.6)
B (0.2)
C (0.15)
D (0.05)

Iteration 2

A (0.6)
B (0.2)
CD (??)

a) $p_c + p_d$
b) Min[$p_c$, $p_d$]
c) Max[$p_c$, $p_d$]
d) $p_c \times p_d$
FUNCTION Huffman(symbols, frequencies)

forest = [(f, s) FOR f, s IN Zip(symbols, frequencies)]
heapifyMin(forest)

WHILE forest.length ≥ 2
    treeA = extract_min(forest)
    treeB = extract_min(forest)
    treeMerged = merge(treeA, treeB)
    heap_add(forest, treeMerged)

# Only one tree remaining in forest
RETURN forest[0]
FUNCTION Huffman(symbols, frequencies)

    forest = [(f, s) FOR f, s IN Zip(symbols, frequencies)]
    heapifyMin(forest)

    WHILE forest.length ≥ 2
        treeA = extract_min(forest)
        treeB = extract_min(forest)
        treeMerged = merge(treeA, treeB)
        heap_add(forest, treeMerged)

    # Only one tree remaining in forest
    RETURN forest[0]

Σ = | A | B | C | D | E | F |
---|---|---|---|---|---|---|
P = | 3 | 2 | 6 | 8 | 2 | 6 |
FUNCTION Huffman(symbols, frequencies)

forest = [(f, s) FOR f, s IN Zip(symbols, frequencies)]
heapifyMin(forest)

WHILE forest.length ≥ 2
    treeA = extract_min(forest)
    treeB = extract_min(forest)
    treeMerged = merge(treeA, treeB)
    heap_add(forest, treeMerged)

# Only one tree remaining in forest
RETURN forest[0]

What is the running time?
Note: faster algorithms do exist for this problem
Correctness Proof

**Theorem**: Huffman’s algorithm computes a binary tree that minimizes the average encoding length of all symbols

\[ L(T) = \sum_{i=1}^{|\Sigma|} p_i \times \text{depth}_i \]

**Strategy:**
- Induction
- Exchange argument

**Proof by induction that** \( P(n) \) **holds for all** \( n \)
  - **Base Case**: \( P(1) \) holds because ...
  - **Inductive Hypothesis**: Let’s assume that \( P(k) \) holds, where \( k < n \)
  - **Inductive Step**: \( P(n) \) holds because of \( P(k) \) and ...
  - Thus, by induction, \( P(n) \) holds for all \( n \)
Inductive Proof

Proof by induction that $P(n)$ holds for all $n$
- **Base Case:** $P(1)$ holds because ...
- **Inductive Hypothesis:** Let’s assume that $P(k)$ holds, where $k < n$
- **Inductive Step:** $P(n)$ holds because of $P(k)$ and ...
- Thus, by induction, $P(n)$ holds for all $n$

**Base Case:**
- If $n = 1$ or $n = 2$ there is only one option for average encoding length
- Thus the base cases are trivially true

**Inductive Hypothesis:**
- Huffman’s algorithm produces the optimal coding with $\leq k$ symbols where $k < n$

**Inductive Step...**
Main Ideas for Inductive Step

Let symbols $\emptyset$ and $\pi$ be the symbols with the smallest and second smallest frequencies, respectively

1. Huffman’s Algorithm outputs the optimal tree in which $\emptyset$ and $\pi$ are siblings
   - Out of all possible trees where $\emptyset$ and $\pi$ are siblings

2. The optimal tree is the one in which $\emptyset$ and $\pi$ are siblings
   - Out of all possible trees in general
Part 1

Huffman’s outputs the optimal tree in which $\phi$ and $\pi$ are siblings

• After combining symbols $\phi$ and $\pi$ into a single “$\phi\pi$” symbol we have reduced the total number of symbols by 1

• Given our inductive hypothesis, we know that Huffman’s algorithm outputs the optimal tree for $k$ symbols where $k < n$

• Thus, Huffman’s outputs the optimal tree after combining $\phi$ and $\pi$
Part 2

The optimal tree is the one in which $\phi$ and $\pi$ are siblings

• Consider the case where $\phi$ and $\pi$ are not siblings
• And we then exchange $\phi$ and $\pi$ with two nodes that are siblings
• The average encoding length goes down (or stays the same)!
Summary

• Prefix-free, variable-length binary codes have smaller average encoding lengths (per symbol) than fixed-length codes

• These Huffman Codes can be visualized as a binary tree

• Huffman’s Algorithm works by greedily combining trees in the forest until you are left with a single tree in $O(n \lg n)$ time

• We proved correctness with induction and an exchange argument