L19: Greedy Algorithms, Huffman Coding

Alex Steiger
CompSci 201: Spring 2024
3/25/2024
Person in CS: Kathleen Booth

• 1922 – 2022
• British Mathematician, PhD in 1950
• Worked to design the first *assembly language* for early computer designs in the 1950s
• May have been the first woman to write a book on programming
• Early interest in *neural networks*
Logistics, Coming up

• Today: Monday 3/25
  • Project P4: Autocomplete due

• This Wednesday 3/27
  • APT 7 (tree recursion problems) due

• Monday 4/1
  • Nothing due—P5 the week after
Today’s agenda

• Solve HeightLabel APT

• Introduce Greedy Algorithms

• Huffman Coding (P5: Huffman)
  • Uses trees and greedy algorithms
  • Back to more tree-based data structures on Wed
Balance and Trees

**Balanced**

For each node, left and right subtrees have roughly equal number of nodes.

**Unbalanced**

One subtree has many more nodes than the other.
Recurrence relation and runtime for traversing a **balanced** tree

- \( T(n) \) time for \( \text{count}(\text{tree}) \) with \( n \) nodes (balanced)

```java
public int count(TreeNode tree) {
    if (tree == null) {
        return 0;
    }
    return 1 + count(tree.left) + count(tree.right);
}
```

- \( T(n) = 2T(n/2) + O(1) \)
- \( = O(n) \)
Recurrence relation and runtime for traversing *unbalanced* tree

- \( T(n) \) time for \( \text{count(tree)} \) with \( n \) nodes (unbalanced)

```
public int count(TreeNode tree) {
    if (tree == null) {
        return 0;
    }
    return 1 + count(tree.left) + count(tree.right);
}
```

- \( T(n) = T(1) + T(n-1) + O(1) \)
- \( = O(1) + T(n-1) + O(1) \)
- \( = O(n) \)
Balance Binary Search Tree
Runtime (add, contains)

Balanced

Unbalanced

\[ T(n) = T(n/2) + O(1) = O(\log(n)) \]

\[ T(n) = T(n-1) + O(1) = O(n) \]

We will return to this problem later!
HeightLabel APT

https://www2.cs.duke.edu/csed/newapt/heightlabel.html

• Create a new tree from a tree parameter
  • Same shape, nodes labeled with height
  • Use **new TreeNode**. With what values ...

Note that this APT 1-indexes height/depth. We introduced it 0-indexed.
Solving Height Label in Pictures

Base case: when null, 0

Recursive case: height of node is 1 + max(height of node.left, height of node.right)

When null? 0?

5 = 1 + max(4, 2)

2 = 1 + max(1, 1)

1 = 1 + max(0, 0)
Live Coding HeightLabel
Solving HeightLabel in Code

```java
private int height(TreeNode t) {
    if (t == null) return 0;
    return 1 + Math.max(height(t.left), height(t.right));
}

public class HeightLabel {
    public TreeNode rewire(TreeNode t) {
        // replace with working code
        return null;
    }
}

public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
    return new TreeNode(height(t),
                         rewire(t.left),
                         rewire(t.right));
}
```

- **Base case:** when null, 0
- **Recursive case:** height of node is $1 + \text{height of node.left} + \text{height of node.right}$
- Method doesn’t just calculate height, is supposed to create and return new tree with new nodes...
- Using height helper method, get height, create new node, return.
Rewire runtime?

- Recurrence of this correct code? \( T(n) = \)

- \( 2T(n/2) + O(n) \)
  - Balanced tree

- \( T(n-1) + O(n) \)
  - Unbalanced

```java
public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
    return new TreeNode(height(t),
                         rewire(t.left),
                         rewire(t.right));
}

private int height(TreeNode t) {
    if (t == null) return 0;
    return 1 + Math.max(height(t.left),
                         height(t.right));
}
```
HeightLabel Complexity

• Balanced?
  • \( T(N) = 2T(n/2) + O(n) \Rightarrow O(N \log N) \)

• Unbalanced,
  • \( T(N) = T(N-1) + O(N) \Rightarrow O(N^2) \)

• Doable in \( O(N) \) time? Yes, if we don't call height
  • Balanced: \( T(N) = 2T(N/2) + O(1) \)
  • Unbalanced: \( T(N) = T(N-1) + O(1) \)
HeightLabel in $O(N)$ time

- If recursion works, subtrees store heights!

Balanced? $O(N)$, $2T(n/2) + O(1)$

Unbalanced, $O(N)$, $T(N-1) + O(1)$

```java
public TreeNode rewire(TreeNode t) {
    if (t == null) { return null; }
    TreeNode leftOfMe = rewire(t.left);
    TreeNode rightOfMe = rewire(t.right);
    int lHeight = 0;
    int rHeight = 0;
    if (leftOfMe != null) { lHeight = leftOfMe.info; }
    if (rightOfMe != null) { rHeight = rightOfMe.info; }
    return new TreeNode(
        x: 1+Math.max(lHeight, rHeight),
        leftOfMe,
        rightOfMe);
}
```
Hi, Alexander. When you submit this form, the owner will see your name and email address.

* Required

1

NetID *

solutions

2

Here is the recursive height helper method we wrote (1 indexes height). Which recurrence relation best describes the runtime complexity of height as a function of $N =$ the number of
nodes in the tree \( t \) **assuming the tree is balanced**? *  

```java
private int height(TreeNode t) {
    if (t == null) return 0;
    return 1 + Math.max(height(t.left), height(t.right));
}
```

- \( T(N) = 2T(N/2) + O(1) \)
- \( T(N) = 2T(N/2) + O(N) \)
- \( T(N) = 2T(N-1) + O(1) \)
- \( T(N) = T(N-1) + O(1) \)

Here is another version of the height helper method. Will it work correctly? *  

```java
private int height(TreeNode t) {
    int value = 1 + Math.max(height(t.left), height(t.right));
    if (t == null) return 0;
    return value;
}
```
Yes, this is equivalent to the previous approach

No, this will not compile

No, this computes different height values

No, this generates a runtime exception

4

Here is an equivalent version of the rewire method that uses the height helper method. Which recurrence relation best describes the runtime complexity of rewire as a function of $N =$ the number of nodes in the tree $t$ **assuming the tree is balanced**? You may assume that the height method is $O(N)$. * 

```java
public TreeNode rewire(TreeNode t) {
    if (t == null) return null;
    int myHeight = height(t);
    TreeNode lchild = rewire(t.left);
    TreeNode rchild = rewire(t.right);
    return new TreeNode(myHeight, lchild, rchild);
}
```

- $T(N) = 2T(N/2) + O(1)$
- $T(N) = 2T(N/2) + O(N)$
Suppose you run this recursive method on some tree t. During the execution of the program, what is myHeight for the **first** (in terms of execution of the code) TreeNode that gets returned by line 13 on any of the recursive invocations of the method? *

```
8   public TreeNode rewire(TreeNode t) {
9       if (t == null) return null;
10      int myHeight = height(t);
11      TreeNode lchild = rewire(t.left);
12      TreeNode rchild = rewire(t.right);
13      return new TreeNode(myHeight, lchild, rchild);
14   }
```
Greedy Algorithms for Discrete Optimization
Optimization

- Find the solution that maximizes or minimizes some objective

**Example: Knapsack**
- Find the bundle of items with maximum value without exceeding a budget.
- What should you buy if you have $10?
- (Only one of each item)

<table>
<thead>
<tr>
<th>Items</th>
<th>Value</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>2</td>
<td>$1</td>
</tr>
<tr>
<td>Banana</td>
<td>1</td>
<td>$1</td>
</tr>
<tr>
<td>Pizza</td>
<td>12</td>
<td>$10</td>
</tr>
</tbody>
</table>
Greedily Searching for Optima

• Start with a partial solution. In each iteration make a step toward a complete solution.

• **Greedy principle**: In each iteration, make the step that “best improves” the solution (e.g., the lowest cost or highest value step).

• Knapsack example:
  • Partial solution is a set of items you can afford
  • Greedy step: Add the item with best value per cost ratio that you can afford with remaining money
Local Optima vs Global Optima?

Greedy algorithms do not always guarantee to find the best overall solution, called global optima.

0. Start with $10
1. Buy apple, best value/cost. $8 remaining
2. Buy banana (can’t afford pizza). $7 remaining
3. Done: Can’t afford any more items. Total value of items = 3

<table>
<thead>
<tr>
<th>Items</th>
<th>Value</th>
<th>Cost</th>
<th>Value/Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apple</td>
<td>2</td>
<td>$1</td>
<td>2</td>
</tr>
<tr>
<td>Banana</td>
<td>1</td>
<td>$1</td>
<td>1</td>
</tr>
<tr>
<td>Pizza</td>
<td>12</td>
<td>$10</td>
<td>1.2</td>
</tr>
</tbody>
</table>

But just buying the pizza has value 12, which is the (only) global optimum
Why Learn Greedy Algorithms?

1. Sometimes a greedy algorithm is optimal (always returns global optima). Examples:
   • Huffman Compression (today, Project 5)
   • Computing shortest paths in networks/graphs

2. Sometimes the greedy algorithm is not optimal, but still works well in practice

3. A greedy algorithm is typically easy to start with for optimization problems.
Aside: What is Machine Learning?

which are designed for one use-case, the API today provides a general-purpose “text in, text out” interface, allowing users to try it on virtually any English language task. You can now request access in order to integrate the API into your product, develop an entirely new application, or help us explore the strengths and limits of this technology.

response = openai.Completion.create(model="davinci", prompt=prompt, stop="\n", temperature=0.9,

users to try it on virtually any English language task. You can now request access in order to integrate the API into your product, develop an entirely new application, or help us explore the strengths and limits of this technology. The road to making AI safe and challenging, but with the support of the community we expect to get there.
Aside continued – How do you “learn a model” greedily?

• Often (in deep learning) represent a model with a neural network.

• Learn model: optimize parameters of network on data.

• How to optimize the parameters?
  • Greedy algorithm called gradient descent
  • At each step, make a small change that best improves model performance
Huffman Coding

Topic of Project 5: Huffman
Huffman Compression

Representing data with bits: Preferably fewer bits

- Zip
- Unicode
- JPEG
- MP3

Huffman compression used in all of these and more!
Encoding

- Eventually, everything stored as bit sequence: 011001011...

- **Fixed length encoding**
  - Each value has a unique bit sequence of the same length stored in a table.
  - With $N$ unique values to encode, need $\lceil \log_2(N) \rceil$ bits per value.
  - E.g., with 8 characters, need 3 bits per character.

<table>
<thead>
<tr>
<th>ASCII coding</th>
<th>char</th>
<th>ASCII</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
<td></td>
</tr>
<tr>
<td>space</td>
<td>32</td>
<td>1000000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3-bit coding</th>
<th>char</th>
<th>code</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>0</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>1</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>2</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>3</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>4</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>r</td>
<td>5</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>6</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>space</td>
<td>7</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>
Optimizing Encoding?

• Suppose we have three characters \{a, b, c\}:
  • a appears 1,000,000 times
  • b and c appear 50,000 times each

• Fixed length encoding uses 2,200,000 bits:
  • \(\lceil \log_2 (3) \rceil = 2\) bits per character
  • 2 bit/char * 1,100,000 chars = 2,200,000 bits

• **Variable length encoding**: Use fewer bits to encode more common values, more bits to encode less common values.
  • What if we encode: a = 1, b = 10, c = 11?
  • Only uses 1,200,000 bits.
Decoding Fixed Length

- Fixed Length with length \( k \)
  - Every \( k \) bits, look up in table
- 001 001 010 110
  - 001 \( \rightarrow o \)
  - 001 \( \rightarrow o \)
  - 010 \( \rightarrow p \)
  - 110 \( \rightarrow s \)
Decoding Variable Length

• What if we use
  • a = 1
  • b = 10
  • c = 11

• How would we decode 1011?
  • “baa” or “bc?”

• Problem: Encoding of a (1) is a prefix of the encoding for c (11). Ambiguous!
Prefix Property: Encoding as a Tree

Convention: 0 for left and 1 for right

Encoding is the sequence of 0’s and 1’s on root to leaf path

Values you want to encode are leaves: Ensures prefix property.

Values deeper in tree encoded with more bits than those earlier in the tree.
Huffman Coding

• **Greedy** algorithm for building an optimal variable-length encoding tree.

• High level idea:
  • Start with the leaves/values you want to encode with weights = frequency. Then repeat until all leaves are in single tree:
    • **Greedy** step: Choose the lowest-weight nodes to connect as children to a new node with weight = sum of children.

• Implementation? Priority queue!
Visualizing the Algorithm

Encoding the text “go go gophers”

```
<table>
<thead>
<tr>
<th>char</th>
<th>binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>'g'</td>
<td>00</td>
</tr>
<tr>
<td>'o'</td>
<td>01</td>
</tr>
<tr>
<td>'p'</td>
<td>1110</td>
</tr>
<tr>
<td>'h'</td>
<td>1101</td>
</tr>
<tr>
<td>'e'</td>
<td>101</td>
</tr>
<tr>
<td>'r'</td>
<td>1111</td>
</tr>
<tr>
<td>'s'</td>
<td>1100</td>
</tr>
<tr>
<td>'r'</td>
<td>100</td>
</tr>
</tbody>
</table>
```
P5 Outline

1. Write Decompress first
   • Takes a compressed file (we give you some)
   • Reads Huffman tree from bits
   • Uses tree to decode bits to text

2. Write Compress second
   • Count frequencies of values/characters
   • Greedy algorithm to build Huffman tree
   • Save tree and file encoded as bits
Diameter Problem

[leetcode.com/problems/diameter-of-binary-tree](leetcode.com/problems/diameter-of-binary-tree)

Calculate the *diameter* of a binary tree, the length of the longest path (maybe through root, maybe not, can’t visit any node twice).