Algorithms and Data Structures for Data Science
Nearest Neighbor Search

CS 277
Brad Solomon

March 29, 2023
Learning Objectives

Review BST implementations

Discuss applications of BSTs

Introduce nearest neighbor search using images

An overview of the KD-Tree (You will not be implementing!)

An overview of the Huffman tree
def find_helper(node, key):
    if not node:
        return None
    if node.key == key:
        return node
    if node.key > key:
        return find_helper(node.left, key)
    if node.key < key:
        return find_helper(node.right, key)
def find_helper(node, key):
    if not node:
        return None
    if node.key == key:
        return node
    if node.key > key:
        find_helper(node.left, key)
    if node.key < key:
        find_helper(node.right, key)
def insert_helper(node, key, value):
    if node == None:
        return bstNode(key, value)
    if node.key > key:
        node.left = insert_helper(node.left, key, value)
    if node.key < key:
        node.right = insert_helper(node.right, key, value)
    return node
def insert_helper(node, key, value):
    if node == None:
        return bstNode(key, value)
    if node.key > key:
        insert_helper(node.left, key, value)
    if node.key < key:
        insert_helper(node.right, key, value)
    return node

insert(5)
def insert_helper(node, key, value):
    if node == None:
        return bstNode(key, value)
    if node.key > key:
        node.left = insert_helper(node.left, key, value)
    if node.key < key:
        node.right = insert_helper(node.right, key, value)
def remove_helper(node, key):
    if node == None:
        return None
    if node.key > key:
        node.left = remove_helper(node.left, key)
    if node.key < key:
        node.right = remove_helper(node.right, key)
    if node.key == key:
        if node.left == None and node.right == None:
            return None
        elif node.left == None:
            return node.right
        elif node.right == None:
            return node.left
        iop = findIOP(node)
        node.key = iop.key
        node.val = iop.val
        node.left = remove_helper(node.left, iop.key)
    return node
## BST Analysis – Running Time

<table>
<thead>
<tr>
<th>Operation</th>
<th>BST Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>delete</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>traverse</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

*Diagram of a binary search tree showing nodes 1, 5, 7, 6, 9.*
When would we use a tree?

Pretend for a moment that we always have an optimal BST.

What is the running time of \texttt{find}?

What is the running time of \texttt{insert}?

What is the running time of \texttt{remove}?

Is there a data structure with a \textit{better} running time for all of these?
Advantages of trees

The running time for a balanced tree is always $O(\log n)$

The structure of a tree can have underlying meaning

Ex: Huffman Trees for Huffman encoding

Trees can be used to find the nearest neighbor
Nearest Neighbor Find

FNN (70)
Nearest Neighbor Find

FNN(27)
Nearest Neighbor Find

FNN(14)
Nearest Neighbor Find

```
def fnn_helper(node, key):
    if not node:
        return None
    if node.key == key:
        return node
    if node.key > key:
        temp = fnn_helper(node.left, key)
    if node.key < key:
        temp = fnn_helper(node.right, key)
    # Nearest neighbor is either node.val (curr node)
    # OR the nearest neighbor found in the subtree
```
Real World Use Case: Nearest neighbor search

Given the collection above, what is the closest match to the color below?
Real World Use Case: Nearest neighbor search

Given the collection above, what is the closest match to the color below?

(90, 75, 50)
Euclidean Distance

The distance between two points is the length of a line between them

1D: \[ d(p, q) = \sqrt{(p - q)^2} \]

2D: \[ d(p, q) = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2} \]

3D: \[ d(p, q) = \sqrt{(p_0 - q_0)^2 + (p_1 - q_1)^2 + (p_2 - q_2)^2} \]
Real World Use Case: Nearest neighbor search
Real World Use Case: Nearest neighbor search
We can reduce the total number of calculations by **averaging colors**
Real World Use Case: Nearest neighbor search
Real World Use Case: Nearest neighbor search
Naive Nearest Neighbor Search

1. Create a method of getting the Euclidean distance between points
   \[ \text{exactColorDist}(c_1, c_2) \]

2. Create a method of getting the average color for a subset of the image
   \[ \text{getAverageColor}(\text{numArray}, \ rstart=0, \ cstart=0, \ rlen=None, \ clen=None) \]

3. For each sub-image of a large image, get the closest matching tile
   \[ \text{getClosestColor}(\text{inlist}, \ \text{query}) \]
Naive Nearest Neighbor Search

Pros:

Cons:
BST Nearest Neighbor Search

Rather than compare every sub-image to every tile, we want to build a BST!

… what is the smallest point in a 2D plane?
BST Nearest Neighbor Search

Rather than compare every sub-image to every tile, we want to build a BST!

… what is the smallest point in a 2D plane?

Problem: There’s no obvious order in multi-dimensional space!

We can reduce the dimensions to create an arbitrary order, but lose precision
BST Nearest Neighbor Search

Because our input set is *colors*, there is at least one dimensional reduction
BST Nearest Neighbor Search

Because our input set is colors, there is at least one dimensional reduction. Instead of a 3D RGB value, we can store a 1D luminance value:

\[ L = 0.299R + 0.587G + 0.114B \]
BST Nearest Neighbor Search

1. Create a method of comparing 1D ‘sizes’ of 3D objects
   \[\text{getLum}(c1)\]

2. Build a luminance BST that stores 3D objects based on their 1D size
   \[\text{lum}\_\text{tree}\_\text{insert}(\text{root, key, value})\]

3. Implement a nearest neighbor search on the luminance BST
   \[\text{lum}\_\text{tree}\_\text{find}(\text{root, key})\]
Luminance Nearest Neighbor Search

Pros:

Cons:
The k-dimension tree (KD-tree)

Imagine we have a set of two dimensional points…
The k-dimension tree (KD-tree)

We can build a k-dimension BST by comparing one dimension at a time.

Depth: Split Dimension:
The k-dimension tree (KD-tree)

We can build a k-dimension BST by comparing one dimension at a time.

Depth:  
Split Dimension:
The k-dimension tree (KD-tree)

We can build a k-dimension BST by comparing one dimension at a time.

Depth:                      Split Dimension:
The k-dimension tree (KD-tree)

We can build a k-dimension BST by comparing one dimension at a time.

Depth: Split Dimension:
The k-dimension tree (KD-tree)

At every level, we essentially partition our tree in half:
KD-Tree Construction Example

Imagine I wanted to build a KD-tree with the following points in order:
(7, 2), (5, 4), (9, 6), (2, 3), (8, 1), (4, 7)
At each level we want to compare *only* the relevant search dimension.
KD-tree Search

At each level we want to compare *only* the relevant search dimension.

\[
\text{search}((6, 3))
\]
KD-tree Search

At each level we want to compare only the relevant search dimension.
KD-tree Search

The leaf we find only tells us the maximum radius we need to search, \( \text{search}((6, 3)) \).
KD-tree Search

As we go back up the tree, we decide if our point is closer to our query

search((6, 3))
Since our splitting point was within our search radius, we also have to check if there’s a closer point in the other subtree.
KD-tree Search

We repeat this process all the way up to the root...

search((6, 3))
KD-tree Search

... and in the other subtree if the root was in our radius

search((6, 3))
KD-tree Search

… and in the other subtree if the root was in our radius

search((6, 3))
KD-tree Search

... and in the other subtree if the root was in our radius

search((6, 3))
KD-tree Search

This strategy is worst case $O(n)$ but on average $O(\log n)$.

search((6, 3))
KD-Tree Nearest Neighbor Search

Pros:

Cons:
KD-Tree in CS 277

You do not need to know how to implement a KD-tree

```python
from scipy.spatial import KDTree
l = [(255, 0, 0), (255, 255, 0), (0, 255, 0), (0, 255, 255), (0, 0, 255), (255, 0, 255)]
kdt = KDTree(l)
```

You should know that it is the optimal solution to 100% accuracy NNS

You should understand conceptually how a KD-tree is built
Friday: A different tree application
Next Week: Addressing the ‘height’ problem

<table>
<thead>
<tr>
<th>Operation</th>
<th>BST Worst Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>find</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>delete</td>
<td>$O(h)$</td>
</tr>
<tr>
<td>traverse</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>