CSE332: Data Abstractions
Lecture 23: Wrapping Up

James Fogarty
Winter 2012

Including slides developed in part by
Ruth Anderson, James Fogarty, Dan Grossman, Richard Ladner, Steve Seitz
The Good News

• You have learned a set of tools that allow you to think about, and talk about, a wide variety of computing problems
CueFlik: Learning Image Similarity

$1 - \left( \frac{12}{12 + 48} \right) = 75\%$ likely a ‘product’ image

AAAI 2011, Amershi et al.
CHI 2010, Amershi et al.
UIST 2009, Amershi et al.
CHI 2008, Fogarty et al.
How Can We Decide Which are Positive?
**KNN Graph**

- Compute pairwise distance between every pair (using domain knowledge to determine the distance metric)
- Preserve edges corresponding only to the k nearest neighbors of each vertex in the graph
- Run a search from your positive and negative examples, classify each based on whichever is closer
- KNN greatly reduces $|E|$, from $|V^2|$ to $k|V|$ (i.e., dense to sparse)
- The classification strategy is also semi-supervised, respecting the distribution of your data
  - Imagine two interlocking spirals
How About Now?
Disconnected Graph

• The KNN transformation does not necessarily preserve a path between your labels and all of your data

• You have to decide what this means and what to do about it

• What kinds of tools can you work with?
How Can We Choose a Representative Set?
The Bad News

- We do not know how to efficiently select the items which are "the most representative subset of these elements"

- Our implementation is greedy
  - Choose single most representative item
  - Given that choice, choose another
  - Repeat until desired number of items

- It gets worse, there are many such problems
  - Learn about P and NP in CSE 312
Some Better News

• We have lots of cool algorithms, not just those you have seen
Amortized Algorithms

- Recall our stack implemented as an array
  - Doubles its size if it runs out of room
  - How can we claim \texttt{push} is $O(1)$ time if resizing is $O(n)$ time?
  - We \textit{cannot}, but we \textit{can} claim it’s an $O(1)$ amortized operation

We will just do a simple example
  - There are entire families of data structures based on this
  - The text has more complicated examples and proof techniques
  - The \textit{idea} of how amortized describes average cost is essential
Amortized Complexity

If a sequence of $M$ operations takes $O(M f(n))$ time, we say the amortized runtime is $O(f(n))$.

- The worst case time per operation can be larger than $f(n)$
  - For example, maybe $f(n) = 1$, but the worst-case is $n$

- But the worst-case for any sequence of $M$ operations is $O(M f(n))$

- Amortized guarantee ensures the average time per operation for any sequence is $O(f(n))$
  - This is a stronger guarantee than “average case” $O(f(n))$
Amount of Copying in a “Doubling” Stack

After M operations, we have done < 2M total element copies.

Let n be the size of the array after M operations.

– Then we’ve done a total of:
  \[\frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots\] \(\text{INITIAL\_SIZE} < n\) element copies.

– We must have done at least enough push operations to cause resizing up to size n:
  \[M \geq \frac{n}{2}\]

– So
  \[2M \geq n > \text{number of element copies}\]
Other Approaches to Growing / Shrinking

• If array grows by a constant amount (say 100), operations are not amortized $O(1)$
  – After 1000 operations, you may have done $900 + 800 + 700 + \ldots + 300 + 200 + 100$ copies (i.e., $N^2$)

• If array shrinks when 1/2 empty, operations are not amortized $O(1)$
  – pop and shrink, push and grow, pop and shrink, …

• If array shrinks when 3/4 empty, it is amortized $O(1)$
  – Proof is more complicated, but basic idea remains: by the time of a fan expensive operation, many cheap operations
Splay Tree Basic Idea

If you are forced to make a deep access:

Since you are down there, fix up a lot of deep nodes!

All the way to the root!

No height “bookkeeping”

Amortized $O(\log n)$ operations
Usefulness of Amortized Algorithms

• Proofs are complicated, with “potential functions” to describe how cheap operations “pay for” later expensive operations
  – But this has nothing to do with complexity of the code
  – Often simple, with better constant factors

• When the average cost per operation is all we care about (i.e., sum over all operations), amortized is perfectly fine
  – This is a very common situation

• If every operation must finish quickly, amortized bounds are weak
  – Real-time software
  – Concurrency setting, where you are holding the lock
Range Queries

• Project 3 Grouped Census Data into Blocks

• What if you wanted to keep the original data and efficiently answer queries at arbitrary precision

• Balanced trees can allow accessing on one dimension
  – “Give me all blocks between longitudes x and y”
  – “Give me all blocks between latitudes x and y”

• But what about access on both dimensions?
  – “Give me all blocks in a rectangle”
Range Queries

Rectangular query

Circular query
Nearest Neighbor Search
A Challenging Case for 1D Structure
Quad Trees

Recursively divide up the space as needed to have only one item at each leaf.
A Really Bad Case
**k-d Trees**

- Jon Bentley, 1975, while an undergraduate
- Tree used to store spatial data.
  - Nearest neighbor search.
  - Range queries.
  - Fast look-up
- k-d tree are guaranteed \( \log_2 n \) depth where \( n \) is the number of points in the set.
  - Traditionally, k-d trees store points in \( d \)-dimensional space which are equivalent to vectors in \( d \)-dimensional space.
**k-d Tree Construction**

- If there is just one point, form a leaf with that point

- Otherwise, divide the points in half on one dimension
  - Book uses round-robin division
  - Could also divide on dimension with greatest spread

- Recursively construct k-d trees for the two sets of points
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
$k$-d Tree Construction

At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
At each step divide perpendicular to the widest spread.
Rectangular Range Query

• Recursively search every cell that intersects the rectangle.
Rectangular Range Query
Rectangular Range Query
Rectangular Range Query
Rectangular Range Query
Rectangular Range Query

- Rectangular range query is illustrated with a graph and a corresponding tree structure.
- The query is represented by a shaded region on the left side, which is matched by the tree structure on the right.
- The tree structure shows how the query is evaluated through a series of comparisons and traversals.

Key elements:
- **Rectangular Range**: The shaded region on the left represents the range query, highlighting the area of interest.
- **Nodes and Labels**: Different nodes are labeled with letters (a, b, c, d, e, g, h, i) and numbers (s1, s2, s3, s4, s5, s6, s7, s8) to identify specific elements and their relationships within the query.
- **Tree Structure**: The tree on the right side mirrors the query by having nodes that correspond to the shaded region, indicating how the query is broken down into smaller, manageable parts for efficient evaluation.

The diagram effectively visualizes how data points are queried and matched in a structured format, making it easier to understand the process of a range query.
Rectangular Range Query
Rectangular Range Query
Rectangular Range Query
Rectangular Range Query
k-d Tree Nearest Neighbor Search

query point

x

y

s1

s2

s3

s4

s5

s6

s7

s8

a

b

c

d

e

f

g

h

i

query point

x

y

s1

s2

s3

s4

s5

s6

s7

s8

a

b

c

d

e

f

g

h

i
k-d Tree Nearest Neighbor Search

query point
**k-d Tree Nearest Neighbor Search**

- *query point*

![Diagram](attachment:image.png)
**k-d Tree Nearest Neighbor Search**

- **Query Point**: Red dot
- **Tree Structure**:
  - **Root**: `x` (s1)
  - **Left Child**: `y` (s2)
    - **Left Child**: `x` (s3)
      - **Left Child**: `a`
      - **Right Child**: `b`
    - **Right Child**: `y` (s4)
      - **Left Child**: `x` (s5)
      - **Right Child**: `g`
        - **Left Child**: `d`
        - **Right Child**: `e`
  - **Right Child**: `y` (s6)
    - **Left Child**: `y` (s7)
    - **Right Child**: `y` (s8)
      - **Left Child**: `c`
      - **Right Child**: `f`
      - **Right Child**: `h`
      - **Right Child**: `i`
**k-d Tree Nearest Neighbor Search**

- **query point**

Diagram:
- A k-d tree is shown with nodes and points.
- The tree is partitioned into nodes labeled with 'x', 'y', and subscripts 's1' to 's8'.
- Points are marked with distinct colors and labels such as 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', and 'i'.
**k-d Tree Nearest Neighbor Search**

![Diagram of k-d Tree and nearest neighbor search]

- **Query point**
- **Nodes and Splitting Planes**
  - x (root)
  - y
  - s1
  - s2
  - s3
  - s4
  - s5
  - s6
  - s7
  - s8

- **Points**
  - a
  - b
  - c
  - d
  - e
  - f
  - g
  - h
  - i

- **Decision Process**
  - Starting at the root (x), adjust based on the query point's position relative to the splitting planes (y, s1) and continue recursively until the nearest point is found.
k-d Tree Nearest Neighbor Search
k-d Tree Nearest Neighbor Search

- **query point**

```
<table>
<thead>
<tr>
<th>y</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
<th>s6</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>s1</th>
<th>s2</th>
<th>s3</th>
<th>s4</th>
<th>s5</th>
<th>s6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- **Query point**

```
x
s1
y
s2
y
s4
y
s6
x
s3
g
x
s5
d
e
```
**k-d Tree Nearest Neighbor Search**

![Diagram](image)
k-d Tree Nearest Neighbor Search

query point
k-d Tree Nearest Neighbor Search

query point
**k-d Tree Nearest Neighbor Search**

- **Query Point**: Red square
- **Nodes**: Green circles
- **Leaves**: Purple squares

Diagram shows a k-d tree structure with nodes and leaves. The query point is located in the top-left quadrant, surrounded by nodes and leaves indicating the search process.
k-d Tree Nearest Neighbor Search

query point
Prefab:
What if Anybody Could Modify Any Interface

CHI 2012, Dixon et al.
CHI 2011, Dixon et al.
CHI 2010, Dixon et al.
Prefab CHI 2011 Video
Decomposing Widgets into Their Parts

Windows Vista Steel Button Prototype
Decomposing Widgets into Their Parts

Mac Slider Prototype
Problem

- Efficiently match a large library of “pixel patches” in images
- Can break this down as a dictionary matching problem
  - Match a dictionary of strings in text
Aho-Corasick Algorithm

- Pre-process dictionary to create a finite state machine
  - Follow an edge for every ‘character’
  - Output any ‘strings’ when arriving at a node

- Linear in the length of the ‘text’ we examine
  - Ignoring pre-processing (which is fine in this application)
Aho-Corasick Algorithm

- Matching a dictionary of pixel rows:
  \{blue, red\}, \{blue, green, green\},
  \{blue, red, green, green\}, \{green, blue, red\}

\[ \neq \{blue, green\} \]
Moral of the Story