Module 8: Data Structures for Multi-Dimensional Data

CS 240 - Data Structures and Data Management

Mark Petrick

Based on lecture notes by many previous cs240 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Fall 2017

Multi-Dimensional Data

- Various applications
  - Attributes of a product (laptop: price, screen size, processor speed, RAM, hard drive, ...)
  - Attributes of an employee (name, age, salary, ...)

- Dictionary for multi-dimensional data
  A collection of $d$-dimensional items
  Each item has $d$ aspects (coordinates): $(x_0, x_1, \cdots, x_{d-1})$
  Operations: insert, delete, range-search query

- (Orthogonal) Range-search query: specify a range (interval) for certain aspects, and find all the items whose aspects fall within given ranges.
  Example: laptops with screen size between 11 and 13 inches, RAM between 8 and 16 GB, price between 1,500 and 2,000 CAD

Multi-Dimensional Data

- Each item has $d$ aspects (coordinates): $(x_0, x_1, \cdots, x_{d-1})$
- Aspect values ($x_i$) are numbers
- Each item corresponds to a point in $d$-dimensional space
- We concentrate on $d = 2$, i.e., points in Euclidean plane

![Diagram showing a range-search query](image-url)
One-Dimensional Range Search

- **First solution**: ordered arrays
  - Running time:
  - Problem: does not generalize to higher dimensions
- **Second solution**: balanced BST (e.g., AVL tree)

```
BST-RangeSearch(T, k1, k2)
T: A balanced search tree, k1, k2: search keys
Report keys in T that are in range [k1, k2]
1. if T = nil then return
2. if key(T) < k1 then
3. BST-RangeSearch(T.right, k1, k2)
4. if key(T) > k2 then
5. BST-RangeSearch(T.left, k1, k2)
6. if k1 ≤ key(T) ≤ k2 then
7. BST-RangeSearch(T.left, k1, k2)
8. report key(T)
9. BST-RangeSearch(T.right, k1, k2)
```

Range Search example

`BST-RangeSearch(T, 30, 65)`

Nodes either on boundary, inside, or outside.

```
52
35
15
9 27
42
39 46
74
65
60 69
97
86 99
```

Note: Not every boundary node is returned.

One-Dimensional Range Search

- **P1**: path from the root to a leaf that goes right if \( k < k_1 \) and left otherwise
- **P2**: path from the root to a leaf that goes left if \( k > k_2 \) and right otherwise
- Partition nodes of \( T \) into three groups:
  - **boundary nodes**: nodes in \( P_1 \) or \( P_2 \)
  - **inside nodes**: non-boundary nodes that belong to either (a subtree rooted at a right child of a node of \( P_1 \)) or (a subtree rooted at a left child of a node of \( P_2 \))
  - **outside nodes**: non-boundary nodes that belong to either (a subtree rooted at a left child of a node of \( P_1 \)) or (a subtree rooted at a right child of a node of \( P_2 \))
- **k**: number of reported items
- Nodes visited during the search:
  - \( O(\log n) \) boundary nodes
  - \( O(k) \) inside nodes
  - No outside nodes
- Running time \( O(\log n + k) \)
2-Dimensional Range Search

- Each item has 2 aspects (coordinates): \((x_i, y_i)\)
- Each item corresponds to a point in Euclidean plane
- Options for implementing \(d\)-dimensional dictionaries:
  - Reduce to one-dimensional dictionary: combine the \(d\)-dimensional key into one key
    Problem: Range search on one aspect is not straightforward
  - Use several dictionaries: one for each dimension
    Problem: inefficient, wastes space
  - Partition trees
    - A tree with \(n\) leaves, each leaf corresponds to an item
    - Each internal node corresponds to a region
    - quadtrees, kd-trees
  - multi-dimensional range trees

Quadtrees

- We have \(n\) points \(P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}\) in the plane
- How to build a quadtree on \(P\):
  - Find a square \(R\) that contains all the points of \(P\) (We can compute minimum and maximum \(x\) and \(y\) values among \(n\) points)
  - Root of the quadtree corresponds to \(R\)
  - Split: Partition \(R\) into four equal subsquares (quadrants), each correspond to a child of \(R\)
  - Recursively repeat this process for any node that contains more than one point
  - Points on split lines belong to left/bottom side
  - Each leaf stores (at most) one point
  - We can delete a leaf that does not contain any point

Quadtrees

- Example: We have 13 points \(P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{12}, y_{12})\}\) in the plane
Quadtree Operations

- **Search**: Analogous to binary search trees
- **Insert**:
  - Search for the point
  - Split the leaf if there are two points
- **Delete**:
  - Search for the point
  - Remove the point
  - If its parent has only one child left, delete that child and continue the process toward the root.

Quadtree: Range Search

```
QTree-RangeSearch(T, R)
T: A quadtree node, R: Query rectangle
1. if (T is a leaf) then
2. if (T.point ∈ R) then
3. report T.point
4. for each child C of T do
5. if C.region ∩ R ≠ ∅ then
6. QTree-RangeSearch(C, R)
```

- **Spread factor** of points $P$: $\beta(P) = d_{max}/d_{min}$
- $d_{max}(d_{min})$: maximum (minimum) distance between two points in $P$
- **Height** of quadtree: $h \in \Theta\left(\log_2 \frac{d_{max}}{d_{min}}\right)$
- Complexity to build initial tree: $\Theta(nh)$
- Complexity of range search: $\Theta(nh)$ even if the answer is $\emptyset$

Quadtree Conclusion

- Very easy to compute and handle
- No complicated arithmetic, only divisions by 2 (usually the boundary box is padded to get a power of two).
- Space wasteful
- Major drawback: can have very large height for certain nonuniform distributions of points
- Easily generates to higher dimensions (octrees, etc.).
kd-trees

- We have \( n \) points \( P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\} \) in the plane.
- Quadtrees split square into quadrants regardless of where points actually lie.
- kd-tree idea: Split the points into two (roughly) equal subsets.
- How to build a kd-tree on \( P \):
  - Split \( P \) into two equal subsets using a vertical line.
  - Split each of the two subsets into two equal pieces using horizontal lines.
  - Continue splitting, alternating vertical and horizontal lines, until every point is in a separate region.
- More details:
  - Initially, we sort the \( n \) points according to their \( x \)-coordinates.
  - The root of the tree is the point with median \( x \)-coordinate (index \( \lfloor n/2 \rfloor \) in the sorted list).
  - All other points with \( x \)-coordinate less than or equal to this go into the left subtree; points with larger \( x \)-coordinate go in the right subtree.
  - At alternating levels, we sort and split according to \( y \)-coordinates instead.
- Complexity: \( \Theta(n \log n) \), height of the tree: \( \Theta(\log n) \).

kd-tree: Range Search

\[
\text{kd-rangeSearch}(T, R, \text{split}[\leftarrow 'x'])
\]

- \( T \): A kd-tree node, \( R \): Query rectangle.
- 1. if \( T \) is empty then return.
- 2. if \( T\.point \in R \) then
  - report \( T\.point \).
- 3. for each child \( C \) of \( T \) do
  - if \( C\.region \cap R \neq \emptyset \) then
  - \( \text{kd-rangeSearch}(C, R) \) if \( \text{split} = 'x' \) then
  - if \( T\.point\.x \geq R\.leftSide \) then
  - \( \text{kd-rangeSearch}(T\.left, R, 'y') \)
  - if \( T\.point\.x < R\.rightSide \) then
  - \( \text{kd-rangeSearch}(T\.right, R, 'y') \)
  - if \( \text{split} = 'y' \) then
  - if \( T\.point\.y \geq R\.bottomSide \) then
  - \( \text{kd-rangeSearch}(T\.left, R, 'x') \)
  - if \( T\.point\.y < R\.topSide \) then
  - \( \text{kd-rangeSearch}(T\.right, R, 'x') \)
kd-tree: Range Search Complexity

- The complexity is $O(k + U)$ where $k$ is the number of keys reported and $U$ is the number of regions we go to but unsuccessfully.
- $U$ corresponds to the number of regions which intersect but are not fully in $R$.
- Those regions have to intersect one of the four sides of $R$.
- $Q(n)$: Maximum number of regions in a kd-tree with $n$ points that intersect a vertical (horizontal) line.
- $Q(n)$ satisfies the following recurrence relation:
  $$Q(n) = 2Q(n/4) + O(1)$$
- It solves to $Q(n) = O(\sqrt{n})$.
- Therefore, the complexity of range search in kd-trees is $O(k + \sqrt{n})$.

kd-tree: Higher Dimensions

- kd-trees for $d$-dimensional space
  - At the root the point set is partitioned based on the first coordinate.
  - At the children of the root the partition is based on the second coordinate.
  - At depth $d - 1$ the partition is based on the last coordinate.
  - At depth $d$ we start all over again, partitioning on first coordinate.
- Storage: $O(n)$
- Construction time: $O(n \log n)$
- Range query time: $O(n^{1−1/d} + k)$
  (Note: $d$ is considered to be a constant.)

Range Trees

- We have $n$ points $P = \{(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})\}$ in the plane.
- A range tree is a tree of trees (a multi-level data structure).
- How to build a range tree on $P$:
  - Build a balanced binary search tree $\tau$ determined by the $x$-coordinates of the $n$ points.
  - For every node $v \in \tau$, build a balanced binary search tree $\tau_{assoc}(v)$ (associated structure of $\tau$) determined by the $y$-coordinates of the nodes in the subtree of $\tau$ with root node $v$. 

**Range Trees: Operations**

- **Search**: trivially as in a binary search tree
- **Insert**: insert point in $\tau$ by $x$-coordinate
- From inserted leaf, walk back up to the root and insert the point in all associated trees $T_{\text{assoc}}(v)$ of nodes $v$ on path to the root
- **Delete**: analogous to insertion
- **Note**: re-balancing is a problem!

**Range Trees: Range Search**

- **A two stage process**
  - To perform a range search query $R = [x_1, x_2] \times [y_1, y_2]$:
    - Perform a range search (on the $x$-coordinates) for the interval $[x_1, x_2]$ in $\tau$ ($BST\text{-RangeSearch}(\tau, x_1, x_2)$)
    - For every outside node, do nothing.
    - For every "top" inside node $v$, perform a range search (on the $y$-coordinates) for the interval $[y_1, y_2]$ in $T_{\text{assoc}}(v)$. During the range search of $T_{\text{assoc}}(v)$, do not check any $x$-coordinates (they are all within range).
    - For every boundary node, test to see if the corresponding point is within the region $R$.
  - Running time: $O(k + \log^2 n)$
  - Range tree space usage: $O(n \log n)$
Range Trees: Higher Dimensions

- Range trees for $d$-dimensional space
- Space/time trade-off
  - **Storage:** $O(n (\log n)^{d-1})$
  - **Construction time:** $O(n (\log n)^{d-1})$
  - **Range query time:** $O((\log n)^d + k)$

(Note: $d$ is considered to be a constant.)