Geometric Range Searching
Kinetic Data Structures
Clustering Mobile Nodes

Leonidas J. Guibas
Stanford University
Geometric Range Searching

1. Suppose we want to know all employees with Salary $\in [40K, 50K]$

   ➔ Scan records & pick out those in range *slow*

Adapted from N. Amato
## Motivation

- **Database**

<table>
<thead>
<tr>
<th>employee</th>
<th>Database Record</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td></td>
</tr>
<tr>
<td>salary</td>
<td></td>
</tr>
<tr>
<td>start date</td>
<td></td>
</tr>
<tr>
<td>city address</td>
<td></td>
</tr>
</tbody>
</table>

2. Suppose we want to know all employees with Salary $\in [40K, 50K]$ AND Age $\in [25, 40]$
   
   ➔ Scan records & check each one *slow*
Motivation, Cnt’d.

- Alternative: View each employee is a point in space
  - age range $[15, 75]$
  - salary range $[0K, 500K]$
  - start date $[1/1/1900, today]$
  - city/address $[College Station, Bryan, Austin, ...]$
Motivation, Cnt’d.

Query #2 (age 25-40, salary 40K-50K)

Orthogonal Range Query (Rectangular)
- Want all points in the orthogonal range
- Faster than linear scan, if good data structures are used.
- Query time $O(f(n)+k)$; where $k =$ # of points reported
Range searching desiderata

Because many queries will be made, it pays to preprocess the data and build an index. We desire:

- Low index storage cost
- Fast index construction
- Low query overhead \([f(n)]\)
- Reasonably efficient dynamic DB modifications (insertions/deletions)
1-D Range Searching

- **Data:** Points \( P = \{p_1, p_2, ..., p_n\} \) in 1-D space (set of real numbers)
- **Query:** Which points are in 1-D query rectangle (in interval \([x, x']\))

**Data structure 1: Sorted Array**

- \( A = 3 \ 9 \ 27 \ 28 \ 29 \ 98 \ 141 \ 187 \ 200 \ 201 \ 202 \ 999 \)
- **Query:** Search for \( x \) & \( x' \) in \( A \) by binary search \( O(\log n) \)
  - Output all points between them. \( O(k) \)
  - Total \( O(\log n + k) \)
- **Update:** Hard to insert points. Add point \( p' \), locate it in \( A \) by binary search. Shift elements in \( A \) to make room. \( O(n) \) on average

- **Storage Cost:** \( O(n) \)
- **Construction Cost:** \( O(n \log n) \)
1-D Range Searching Contd.

Data structure 2: Balanced Binary Search Tree

- Leaves store points in P (in left to right order)
- Internal nodes are splitting values. $x_v$ used to guide search.
  - Left sub tree of $V$ contains all values $\leq x_v$
  - Right sub tree of $V$ contains all values $> x_v$
- Query: $[x, x']$
  - Locate $x$ & $x'$ in $T$ (search ends at leaves $u$ & $u'$)
  - Points we want are located in leaves
    - In between $u$ & $u'$
    - Possibly in $u$ (if $x=u_v$)
    - Possibly in $u'$ (if $x=u'_v$)

Leaves of sub trees rooted at nodes $V$ s.t. parent ($v$) is on search path root to $u$ (or root to $u'$)
1-D Range Searching Ctd.

- Look for node $V_{\text{split}}$ where search paths for $x$ & $x'$ split
  - Report all values in right sub tree on search path for $x'$
  - Report all values in left sub tree on search path for $x$
- Query: $[18:77]$
1-D Range Searching Ctd.

- Look for node $V_{\text{split}}$ where search paths for $x$ & $x'$ split
  - Report all values in right sub tree on search path for $x'$
  - Report all values in left sub tree on search path for $x$
- Query: [18:77]
1-D Range Searching Ctd.

- Look for node $V_{\text{split}}$ where search paths for $x$ & $x'$ split
  - Report all values in right sub tree on search path for $x'$
  - Report all values in left sub tree on search path for $x$
- Query: [18:77]
1-D Range Searching Contd.

- Look for node $V_{split}$ where search paths for $x$ & $x'$ split
  - Report all values in right sub tree on search path for $x'$
  - Report all values in left sub tree on search path for $x$
- Query: [18:77]
1-D Range Searching Ctd.

- Update Cost $O(\log n)$
- Query Overhead $O(\log n)$
- Storage Cost $O(n)$
- Construction Cost $O(n \log n)$

![1-D Range Tree](image)

1-D Range Tree
Key ideas

- Pre-store the answer to certain queries, in a hierarchical fashion
  - the binary tree defines canonical intervals
- Assemble the answer to the an actual query by combining answers to pre-stored queries
  - any other interval is the disjoint union of canonical intervals
- How many answers to canonical sub-problems do we pre-store? Storage vs. query-time trade-off.
KD-Trees
(Higher dimensional generalization of 1D-Range Tree.)

- e.g. for 2-dimensions

  idea: first split on x-coord (even levels)
  next split on y-coord (odd levels)
  repeat

  levels : store pts

  internal nodes : splitting lines (as opposed to values)
Build KD-Tree
Build KD-Tree
Build KD-Tree
Build KD-Tree
Build KD-Tree
Build KD-Tree
Build KD-Tree
Build KD-Tree
Build KD-Tree
Complexity

Construction time

- Expensive operation: determining splitting line (median finding)
  - Can use linear time median finding algorithm \( \Rightarrow O(n \log n) \) time.
    \[
    T(n) = O(n) + 2T\left(\frac{n}{2}\right) = O(n \log n)
    \]
  - but can obtain this time without fancy median finding
    \( \Rightarrow \) Presort points by x-coord and by y-coord \( (O(n \log n)) \)
      Each time find median in \( O(1) \) and partition lists and update x and y ordering by scan in \( O(n) \) time
    \[
    T(n) = O(n) + 2T\left(\frac{n}{2}\right) = O(n \log n)
    \]
Complexity

Storage

- Number of leaves = n (one per point)
- Still binary tree $\Rightarrow O(n)$ storage total

Queries

- each node corresponds to a region in plane
- Need only search nodes whose region intersects query region
- Report all points in subtrees whose regions contained in query range
- When reach leaf, check if point in query region
Algorithm: Search KD-Tree \((v, R)\)

- Input: root of a subtree of a KD-tree and a range \(R\)
- Output: All points at leaves below \(v\) that lie in the range

1. If \((v = \text{leaf})\)
2. then report \(v\)’s point if in \(R\)
3. else if (region \((\text{lc}(v))\) fully contained in \(R\))
4. then ReportSubtree \((\text{Rc}(v))\)
5. else if (region \((\text{lc}(v))\) intersects \(R\))
6. then SearchKdTree \((\text{lc}(v)), R\) 
7. if (region \((\text{rc}(v))\) fully contained in \(R\))
8. then ReportSubtree \((\text{rc}(v))\)
9. else if (region \((\text{rc}(v))\) intersects \(R\))
10. then SearchKdtree \((\text{rc}(v)), R\) 
11. Endif

Note: need to know region \((v)\)
- can precompute and store
- Compute during recursive calls, e.g.,
  \[
  \text{region}(\text{lc}(v)) = \text{region}(v) \cap I(v)_{\text{left}}
  \]

\(L(v)\) is \(v\)’s splitting line and \(I(v)_{\text{left}}\) is left halfpland of \(I(v)\)
**Lemma** A query with an axis parallel rectangle in a Kd-tree storing $n$ points can be performed in $O(\bar{c} + k)$ time where $k$ is the number of reported points.
Query time: Generalization to Higher dimensions

Construction is similar: one level for each dimension
Storage: $O(d \cdot n)$
Time: $O(d \cdot n \log n)$

Query time: $O(n^{1-\frac{1}{d}} + k)$
Range trees

- For each internal node \( v \) in tree \( T_x \) let \( P(v) \) be set of points stored in leaves of subtree rooted at \( v \).

Set \( P(v) \) is stored with \( v \) as another balanced binary search tree \( T_y(v) \) (second level tree) on y-coordinate. (have pointer from \( v \) to \( T_y(v) \))
Range trees

**Lemma:** A 2D-range tree with n points uses $O(n \log n)$ storage.

**Lemma:** A query with axis-parallel rectangle in range tree for n points takes $O(\log^2 n + k)$ time, where $k = \#$ reported points.
Higher Dimensional Range Trees

- 1\textsuperscript{st} level tree is balanced binary search tree on 1\textsuperscript{st} coordinate
- 2\textsuperscript{nd} level tree is (d-1) dimensional range tree for P(v)
  - restricted to last (d-1)-coordinates of points
    - this tree constructed recursively
  - last tree is 1D balanced binary search tree on d\textsuperscript{th} coordinates