Range Search (chapter 5)

Elisha Sacks
Range Search

Find the points in an axis-aligned box.
1D Range Search

- Find the keys in a range \([x : x']\), for example \([18 : 77]\).
- Use a binary tree with keys at the leaves.
- Searches for \(x\) and \(x'\) end at leaves \(\mu\) and \(\mu'\).
- The leaves between \(\mu\) and \(\mu'\) are in \([x : x']\).
- Need to test if \(\mu\) and \(\mu'\) are in \([x : x']\).
Algorithm

1. Set \( \nu \) to the root of the tree.
2. while \( \nu \) is not a leaf and \( x' \leq x_\nu \) or \( x > x_\nu \):
   - if \( x' \leq x_\nu \), replace \( \nu \) with its left child, else its right child.
3. Search for \( x \) in the \( \nu_{\text{split}} \) subtree.
4. Report right children where the search goes left.
5. Search for \( x' \) in the \( \nu_{\text{split}} \) subtree.
6. Report left children where the search goes right.
A 2D range consists of two 1D ranges.

kd-trees treat the two ranges in the same way.

Range trees make one primary and the other secondary.

Assume for now that points have unique $x$ and $y$ coordinates.
A kd-tree has split lines at internal nodes and points at leaves.

Split the points with a vertical line through their $x$ median.

Split each subset with a horizontal line through its $y$ median. (Median points go left and down.)

Repeat until every region contains one point.
Complexity of kd-tree Construction

- **Algorithm details**
  - Sort the points on $x$ and on $y$.
  - Obtain the $x$ median $x_m$ from the $x$ sorted list.
  - Split both lists at $x_m$.
  - Pass the left/right halves to the left/right recursive call.
  - Likewise for $y$ splits.

- **Complexity**
  - Sorting $n$ points takes $O(n \log n)$ time.
  - Excluding sorting, $T(n) = 2T(n/2) + O(n)$ for $n > 1$.
  - $T(n) = O(n \log n)$.
  - Space complexity is $O(n)$ because every leaf stores a point.
Each node defines a rectangular region.

The region of the root is the plane.

The regions of the children of a node are the intersections of its region with the half planes of its splitting line.

They are open to the right or above the splitting line.
Search a tree with root \( v \) for points in \( Q \) (not shown).

If \( v \) is a leaf, report its point if it is in \( Q \).

else

    Split the region of \( v \) to compute the regions of \( lc(v) \) and \( rc(v) \).
    if the region of \( lc(v) \) is contained in \( Q \), report it.
    else if the region intersects \( Q \), search \( lc(v) \).
    if the region of \( rc(v) \) is contained in \( Q \), report it.
    else if the region intersects \( Q \), search \( rc(v) \).
Range Search Example

- This kd-tree is *not* constructed with our algorithm.
- The grey nodes are visited when $Q$ is the grey rectangle.
- The region of the * node (dark grey) is contained in $Q$. 
The $k$ points in the regions contained in $Q$ take $O(k)$ time.

The regions of the other visited nodes intersect a side of $Q$.

The number that intersect a side is bounded by the number that intersect its supporting line.

We bound the number of $x$ split nodes $V(n)$ whose regions intersect a vertical line $l$. The $y$ split case is analogous.

The bound is pessimistic because $Q$ is usually small.
The \( x \) split node \( r \) has \( n \) points in its region.

- The vertical line \( l \) intersects the region of one child of \( r \).
- It intersects the regions of both children of this child.
- Each grandchild is an \( x \) split node with \( n/4 \) points.
- The recurrence is \( V(n) = 2V(n/4) + 2 \) for \( n > 1 \).
- The solution is \( O(\sqrt{n}) \), so the time complexity is \( O(\sqrt{n} + k) \).
Higher Dimensional kd-trees

- kd-trees generalize to 3 dimensions.
- The split lines become planes.
- The construction cycles between $x$, $y$, and $z$ splitting planes.
- Likewise in any dimension $d$.
- The worst-case query time is $O(n^{1-1/d} + k)$.
- They work well for small $d$ in practice.
Points are stored in a binary tree with $x$-coordinate keys.

Each node $v$ has an associated tree with $y$-coordinate keys.

This tree contains the points $P(v)$ in the subtree of $v$. 
Query Algorithm

- Search the primary tree for the $x$ interval.
- The result is the primary nodes $v$ with $P(v)$ in the $x$ interval.
- Search their associated trees for the $y$ interval.
Space Complexity

- A range tree for $n$ points uses $O(n \log n)$ space.
- The primary tree has $O(n)$ nodes.
- A point is stored in the associated trees of the nodes on the primary tree path from the root to its leaf.
- This yields $O(n \log n)$ associated tree nodes.
Construction Algorithm

buildRT(xpts, ypts)

1. Build the associated tree using ypts.
2. If xpts contains one point, the primary tree is a leaf.
3. Else
   3.1 Let point $p$ have the median $x$ coordinate.
   3.2 Split xpts into xpts1 and xpts2 at $p_x$.
   3.3 Split ypts into ypts1 and ypts2 at $p_x$.
   3.4 Set $v_1$ to buildRT(xpts1, ypts1).
   3.5 Set $v_2$ to buildRT(xpts2, ypts2).
   3.6 The primary tree has key $p_x$, left child $v_1$, and right child $v_2$.

➤ xpts are presorted by $x$; ypts are presorted by $y$.
➤ Splitting maintains these orders.
➤ The associated tree is built bottom up.
Step $i$ constructs level $i$ above the leaves.

Complexity: $n + n/2 + n/4 + \cdots < 2n$
Bottom Up Tree Construction

step 2

step 3
Time Complexity

- **Construction**
  - $O(n \log n)$ for sorting the points on $x$ and on $y$.
  - $O(k)$ to build a $k$-point associated tree.
  - Time for primary node $v$ is linear in the size of $P(v)$.
  - Construction time is linear in sum of $P(v)$.
  - It is $O(n \log n)$ because space is $O(n \log n)$.

- **Search**
  - $O(\log n)$ for the primary tree.
  - $O(\log n + k_v)$ for an associated tree with $k_v$ outputs.
  - $O(\log^2 n + k)$ for $k$ outputs.
  - Fractional cascading reduces this to $O(\log n + k)$. 
Higher Dimensional Range Trees

- Approach generalizes to dimension $d$.
- Time complexity is $O(\log^d n + k)$.
- Programming looks painful.
- Fractional cascading removes a log $n$ factor.
The distinct coordinates assumption is easily removed.

Define a composite number as a pair of numbers \((a|b)\).

Define \((a|b) < (c|d)\) using lexicographic order.

Replace a point \(p\) with \(\hat{p} = ((p_x|p_y), (p_y|p_x))\).

The resulting points have distinct coordinates.

Form a \(kd\)-tree or a range tree for these points.

Replace a query rectangle \([a, b] \times [c, d]\) with \([((a|\infty), (b|\infty)] \times [(c|\infty), (d|\infty))\].

The data need not be changed; just the comparison code.

The method works in any dimension.
Fractional Cascading

A1

\[
\begin{array}{cccccccccccc}
3 & 10 & 19 & 23 & 30 & 37 & 59 & 62 & 70 & 80 & 100 & 105 \\
\end{array}
\]

A2

\[
\begin{array}{cccccccccccc}
10 & 19 & 30 & 62 & 70 & 80 & 100 \\
\end{array}
\]

- Remove a log \( n \) factor from searches of related ordered sets.
- \( A_1 \) and \( A_2 \) are sorted arrays with \( A_2 \) a subset of \( A_1 \).
- Each element of \( A_1 \) points to its least upper bound in \( A_2 \).
- The pointers are set in \( O(n) \) time via array traversal.
- Search a range \([y : y']\) in \( A_1 \) and \( A_2 \) in \( O(\log n + k) \) time.
  1. Find the first index \( a \geq y \) in \( A_1 \) by binary search.
  2. Follow the \( a \) pointer to the first index \( b \geq y \) in \( A_2 \).
  3. Traverse \( A_1 \) and \( A_2 \) until the indices are greater than \( y' \).
- Example: \([20 : 65]\), \( a = 23, b = 30 \), grey elements returned.
Fractional Cascading in a 2D Range Tree

- Replace associated trees with sorted arrays.
- The array of primary node $v$ stores pointers to its $y$ greatest lower bounds in the arrays of $lc(v)$ and $rc(v)$.
- Search the primary tree as before.
- Compute the first array index with $p_y \geq y$ at the split node.
- Update this index at later nodes using the array pointers.
- For each node whose primary set is in the $x$ range, traverse the associated array starting from its first array index.