An Introduction To Range Searching

Jan Vahrenhold
Department of Computer Science
Westfälische Wilhelms-Universität Münster, Germany.

Overview

1. Introduction: Problem Statement, Lower Bounds
2. Range Searching in 1 and 1.5 Dimensions
3. Range Searching in 2 Dimensions
4. Summary and Outlook
**Problem Setting**

**Given:** Collection $S$ of $n$ points in $d$ dimensions ($S \subset \mathbb{R}^d$).

**Wanted:** Algorithm for *efficiently* reporting all $k$ points in $S$ falling into a given axis-parallel query range $D \subset \mathbb{R}^d$.

**Applications:** Geographic Information Systems; Databases having relations in which the keys can be totally ordered.

---

**A First Approach**

- Assume that $S = \{p_0, \ldots, p_{n-1}\}$ is stored in an array.
- Scan though the array and test for each $p_i$ whether $p_i \in D$.

- Need to scan the whole array, regardless of how many points are reported. Complexity: $\Theta(n)$ time and space.
Lower (and Upper) Bounds

Change the model to also include $k$ (the number of points reported) as a parameter.
- Algorithm on previous slide has complexity $O(n + k) = O(n)$.

Time complexity: preprocessing time $\Leftrightarrow$ query time

Can disregard preprocessing time for many applications (one-time operation).

Query time composed of two components:
- **Search time**: Time to locate the first element to be reported.
- **Retrieval time**: Time to fetch and report all $k$ elements to be reported.

Space requirement (lower bound for preprocessing time).

---

Lower Bounds [Bentley & Maurer, 1980]

Parameters: $n$ points, $k$ points reported, $d$ dimensions.

- **Space requirement**: $\Omega(n)$.
- **Retrieval time**: $\Omega(k)$.
- **Search time**: Using binary decision tree ($\rightarrow$ sorting lower bound).

Lower bound construction:
- $(n =) 2ad$ points, each with exactly one unique non-zero integer coordinate taken from $[-a, a] \setminus \{0\}$.
- $D = [b_1, \ldots, b_d] \times [c_1, \ldots, c_d]$, with $b_i \in [-a, -1]$, $c_i \in [1, a]$, $1 \leq i \leq d$.
- Query ranges not-empty, each produces a different answer.
- Overall: $a^{2d} = (n/(2d))^{2d}$ different answers.
- Depth of decision tree: $\Omega\left(\log (n/(2d))^{2d}\right) = \Omega(d \cdot \log n)$.
- Lower bound not tight for all $d$. 

---

Jan Vahrenhold
Range Searching
4

Jan Vahrenhold
Range Searching
5
One-Dimensional Range Searching

- Point set $S = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}$, stored in an array.
- Query range $D = [x_1, x_2]$.
- Scanning is sub-optimal; lower bound: $\Omega(1 \cdot \log_2 n + k)$.

Preprocessing:
- Sort the points, e.g., using heapsort in $O(n \log_2 n)$ time.

Query: Binary search for smallest $p_i \geq x_1$ \(\mathcal{O}(\log_2 n)\)

\[\ldots\text{scan forward until first } p_i < x_2 \text{ (or end of array).}\] \(\mathcal{O}(k + 1)\)
Does Sorting Help in Two Dimensions?

- There is no total order on points in two dimensions sorting according to which guarantees $\Theta (2 \cdot \log_2 n + k)$ query time for range searching.

Recap: One-Dimensional Range Searching

- Key ingredient: binary search (bisection).
- Replace (sorted) array by binary search tree.

- **Time Complexity:**
  - Preprocessing time: $O(n \log n)$
  - Query time: $O(\log n + k)$

- **Space Complexity:** $O(n)$.
- Inserts/Deletes possible.
Three-sided (1.5-dim.) Range Searching

Given: Point set $S = \{p_0, \ldots, p_{n-1}\} \subset \mathbb{R}^2$, stored in an array.

Wanted: Method to efficiently retrieve all $p \in S$ that, for given $(x_1, x_2, y)$, fall into $[x_1, x_2] \times ] - \infty, y]$.

Look at two subproblems:
- Report all points in $[x_1, x_2] \times \mathbb{R}$ using, e.g., a threaded binary search tree.
- Report all points in $\mathbb{R} \times ] - \infty, y]$ using, e.g., a heap:
  - Almost complete binary tree.
  - $\text{key}(v) \leq \min\{\text{key}(\text{LSON}(v)), \text{key}(\text{RSON}(v))\}$.

Combining the best of both worlds(?)

Binary search tree with heap property:
- Binary search tree unique w.r.t. inorder-traversal.
- No (direct) way of incorporating heap property.

Heap with search tree property:
- Heap not unique.
- More precisely: Children of a node may be switched.

Priority Search Tree:
- Binary tree $\mathcal{H}$ storing a two-dimensional point at each node s.t. the heap property w.r.t. the $y$-coordinates is fulfilled.
- Additional requirement: $\forall v \in \mathcal{H} : \exists x_v \in \mathbb{R} : l \leq x_v < r \quad \forall l \in \text{LSUBTREE}(v), \forall r \in \text{RSUBTREE}(v)$. 
Building a priority search tree

Use recursive definition [McCreight, 1985]:

- Build priority search tree $\mathcal{H}(S)$ for a given set $S$ of points in the plane. Assume w.l.o.g. that all coordinates are pairwise distinct.
- If $S = \emptyset$, construct $\mathcal{H}(S)$ as an (empty) leaf.
- Else let $p_{\text{min}}$ be the point in $S$ having the minimum $y$-coordinate.
- Let $x_{\text{mid}}$ be the median of the $x$-coordinates in $S \setminus \{p_{\text{min}}\}$.
- Partition $S \setminus \{p_{\text{min}}\}$:
  
  \[
  S_{\text{left}} := \{ p \in S \setminus \{p_{\text{min}}\} \mid p.x \leq x_{\text{mid}} \}
  \]
  
  \[
  S_{\text{right}} := \{ p \in S \setminus \{p_{\text{min}}\} \mid p.x > x_{\text{mid}} \}
  \]
- Construct search tree node $v$ storing $x_{\text{mid}}$ and set $p(v) := p_{\text{min}}$.
- Recursively compute $v$'s children $\mathcal{H}(S_{\text{left}})$ and $\mathcal{H}(S_{\text{right}})$.
- Complexity: $\mathcal{O}(n)$ space; $\mathcal{O}(n \log n)$ time (why?).

Querying a priority search tree

Query range $[x_1, x_2] \times [-\infty, y]$:

- Queries for $x_1$ and $x_2$ result in two search paths in $\mathcal{H}$.
- Check all points on these paths.
- All subtrees “embraced” by these paths contain points in $[x_1, x_2] \times \mathbb{R}$.
- Query these subtrees as follows:

$\text{SearchInSubtree}(v, y)$

\[
\begin{align*}
\text{if } & v \text{ not a leaf and } p(v).y \leq y \text{ then } \\
& \text{Report } p(v); \\
& \text{SearchInSubtree}(\text{LSO}(v), y); \\
& \text{SearchInSubtree}(\text{RSO}(v), y);
\end{align*}
\]

Query time: $\mathcal{O}(1 + k_v)$.
Summary

Missing Components:

- A more detailed description of the query algorithm.
- Proof of correctness.

⇒ [de Berg et al., 2000]

Theorem 2.1
Priority search trees allow for answering three-sided range queries on points in $\mathbb{R}^2$ with time and space complexities as follows:

Preprocessing time: $\Theta(n \log n)$

Query time: $O(\log n + k)$

Space requirement: $\Theta(n)$

Overview

1. Introduction: Problem Statement, Lower Bounds
2. Range Searching in 1 and 1.5 Dimensions
3. Range Searching in 2 Dimensions
4. Summary and Outlook
- Extend the concept of binary search by \textit{bisection} to higher dimensions.
- Instead of intervals, partition (hyper-)rectangles; do the partitioning \textit{alternating} parallel to the coordinate axes.
- $R_i$ is partitioned into $R_j$ and $R_k \Rightarrow |R_j| \approx |R_k| \approx \frac{1}{2}|R_i|$.
- Structure corresponding to partitioning: balanced binary tree ($kD$-tree [Bentley, 1975]).
- Node $v$ corresponds to hyperrectangle $R(v)$, $R(\text{root}) = \mathbb{R}^d$; children correspond to sub-hyperrectangles.
- Each node $v$ is augmented to store:
  - $S(v)$: points contained in $R(v)$ (implicitly).
  - $\ell(v)$: representation of split axis.
  - $P(v)$: median of $S(v)$ w.r.t. $\ell(v)$.

\textbf{Example}

Alternating partitioning along the coordinate axes.
void search(node v, rectangle D, list(point)& result)

double left, median, right;
if v.type == “vertical” then
    left = D.x1; right = D.x2;
    median = v.P.x;
else
    left = D.y1; right = D.y2;
    median = v.P.y;
if left ≤ median ≤ right and D.contains(v.P) then
    result.append(v.P);
if !isLeaf(v) then
    if left < median then
        search(leftSon(v), D, result);
    if median < right then
        search(rightSon(v), D, result);
return;

Complexity of a 2D-tree

Space requirement:
- \( p \in R(v) \iff p = P(v) \lor p \in R(q) \) for any descendant \( q \) of \( v \).
- \( \mathcal{O}(1) \) space requirement per node, exactly one point stored at each node \( \Rightarrow \mathcal{O}(n) \) overall space requirement.

Construction time (preprocessing):
- Linear-time median finding per partitioning step, i.e., recurrence:
  \[
  T(n) = 2 \cdot T(\lceil n/2 \rceil) + \mathcal{O}(n) \in \mathcal{O}(n \cdot \log n)
  \]
- Alternative: Replace median-finding by pre-sorting (copies of) the point by their \( x \)- and \( y \)-coordinates, respectively.
  - Can find median w.r.t. \( x \)-coordinate in \( \mathcal{O}(1) \) time.
  - Can construct sorted \( y \)-arrays to be passed to the children in linear time.
Analysis of worst-case query time

- Query time proportional to number of nodes visited.
- \( v \) productive \( \iff \) \( P(v) \in D \).
- Nodes visited: productive and unproductive nodes.

**Definition 3.1**

Let \( R(v) \) be a rectangle and let \( 0 \leq i \leq 4 \). \( D \) and \( R(v) \) form a type-\( i \) situation \( \iff \) \( i \) sides of \( R(v) \) intersect the interior of \( D \).

- Type-4 situation always productive, all other situations may be unproductive.

**Constructing a worst-case situation—I**


**Recurrence for worst-case query time:**

\[
T(h) = \frac{1}{A} + \frac{1}{B} + \frac{1}{C} + T(h - 2) + T'(h - 2) + \frac{1}{D} + T'(h - 3)
\]
A closer look at situation “subtree rooted at node $D$”.

Recurrence for this situation:

$$T'(h) = \frac{1}{D} + \frac{1}{X} + \frac{1}{Y} + 2 \cdot T'(h-2)$$

Children of $X$ and $Y$

The following recurrence holds for $T'(h)$:

$$T'(h) = 2 \cdot T'(h-2) + 3$$

with $T'(0) = 0$ and $T'(1) = 1$.

Solve recurrence for $T'(h)$, w.l.o.g. $h = 2 \cdot i$, $i \in \mathbb{N}$.

$$T'(2 \cdot i) = 3 + 2 \cdot T'(2(i - 1))$$

$$= 3 + 2 \cdot (3 + 2 \cdot T'(2(i - 2)))$$

$$= \sum_{j=0}^{i-1} 3 \cdot 2^j = 3 \cdot 2^i - 3$$

Similarly: $T'(2 \cdot i + 1) = 4 \cdot 2^i - 3$. 

Jan Vahrenhold
Range Searching

22

23
The following recurrence holds for $T(h)$:

\[
T(h) = T(h - 2) + T'(h - 2) + T'(h - 3) + 4
\]

\[
T'(h) = \begin{cases} 
4 \cdot 2^i - 3 & \text{for } h = 2i + 1 \\
3 \cdot 2^i - 3 & \text{for } h = 2i 
\end{cases}
\]

with $T(0) = T'(0) = 0$ and $T(1) = T'(1) = 1$.

Solve recurrence for $T(h)$, w.l.o.g. $h = 2i$, $i \in \mathbb{N}$.

\[
T(2i) = 4 + T(2(i - 1)) + 3 \cdot 2^{i-1} - 3 + 4 \cdot 2^{i-2} - 3
\]

\[
= T(2(i - 1)) + 5 \cdot 2^{i-1} - 2
\]

\[
= 5 \cdot (2^{h/2} - 1) - h
\]

Similarly: $T(2i + 1) = 7 \cdot (2^{\lfloor h/2 \rfloor} - 1) - h + 2$.

Overall (for $n \leq 2^h - 1$): $T(n) \in O\left(2 \cdot n^{1/2}\right)$.

**Summary**

- Worst-case query time independent of the number of points reported.
- kD-tree very relevant in practice!
- Extension to higher dimensions (points in $\mathbb{R}^d$): Do partitioning in a round-robin manner of the coordinate axes $x_1 \to x_2 \to \ldots \to x_d \to x_1 \to \ldots$

**Theorem 3.2**

Multidimensional search trees (kD-trees) allow for answering four-sided range queries on points in $\mathbb{R}^d$, $d \geq 2$ with time and space complexities as follows:

- Preprocessing time: $\Theta(d \cdot n \log n)$
- Query time: $O\left(d \cdot n^{1-1/d} + k\right)$
- Space requirement: $\Theta(n)$
1. Introduction: Problem Statement, Lower Bounds

2. Range Searching in 1 and 1.5 Dimensions

3. Range Searching in 2 Dimensions

4. Summary and Outlook

Summary

Lower bounds:

- $\Omega (d \cdot \log_2 n + k)$ time, $\Omega (n)$ space.

Results:

- One dimension: optimal $O (\log_2 n + k)$ algorithm, $\Theta (n)$ space.
- 1.5 dimensions: optimal $O (\log_2 n + k)$ algorithm, $\Theta (n)$ space.
- Two dimensions: sub-optimal $O (\sqrt{n} + k)$ algorithm, $\Theta (n)$ space.
- $d$ dimensions: sub-optimal $O (n^{1-1/d} + k)$ algorithm, $\Theta (n)$ space.

Outlook:

- Optimal query time possible of one is willing to spend superlinear space [Chazelle, 1990]. Beware: choosing the adequate model of computation is crucial.
Bibliography


