Orthogonal Range Queries

Motivation: Data Base queries

Points & S

We assume for non general put sets (distinct x & y coordinates)

Easy start: 1D Range Queries with

Range Trees

Idea: Store the coordinates \((x_1, x_2, x_3, \ldots, x_n)\) in a balanced binary tree

Example:

\[ S = 3, 5, 9, 21, 31, 51, 55, 60 \]
• Every node of the tree encodes a sub
set of \( S \) \( \{ x_i, x_{i+1}, \ldots, x_j \} \) which we name \( R(u) \)
  ordered from \( x_i \) to \( x_j \).
• We store in every node the maximal key
  contained in the subtree of its left child.
• Such a tree can be constructed in
  \( O(n \log n) \) time (first sort) and needs
  \( O(n) \) space.

How to perform a query?

• Query: Find all elements of \( S \) with
  \( a \leq s_i \leq b \).

• Algorithm: (1) Find the "split node" in
  the tree, that is the node
  \( U_{\text{split}} \) with
  \[ R(U_{\text{split}}) \geq [a, b] \]
  \[ R(\text{LC}(U_{\text{split}})) \neq [a, b] \]
  (left child)

  (2) If \( U_{\text{split}} \) is a leaf, check
  if \( R(U_{\text{split}}) \in [a, b] \)

  (3a) Follow the path to a
  and report the right
  Subtrees.
(36) Follow the path to b and report the left subtrees.

**Example**

\[ a = 5, b = 17 \]

![Diagram of a tree structure with labeled nodes and arrows indicating paths.]

"Running" time: It only makes sense to study the running query time "output-sensitive" we denote the length of the output by \( k \).

**Facts:**

- A subtree of size \( n \) can be reported in \( O(n) \).
- The node \( \text{Input} \) can be found in \( \log n \).
- Other than reporting subtrees, we search for \( a \) and \( b \).

\[ \Rightarrow \text{Reporting takes } O(n) \text{ time. Searching for } a, b \text{ is } O(\log n) \]

\[ \Rightarrow \text{Query time } O(k + \log n) \]
Now 2D!

Idea: First do a 1D search (say for the x-coordinates) and then, while reporting, filter for the right y-coordinates.

By Example

\[ S = \{(3,1), (5,18), (9,7), (21,11), (31,8), (51,6), (55,16), (60,4)\} \]

Multi-level DS

- For every node \( v \) in the first level tree (the one for the x-coordinates) we store a balanced binary tree with the y-coordinates of \( R(v) \).
- Every leaf in a second level tree stores \( x \) and \( y \) coordinates.
Storage:

1st level: Storage / node

3rd level

2n

2n

2n

... 3rd level

Preprocessing time: When the y-coordinates are pre-sorted, the 2nd level trees can be built in linear time.

→ Same analysis as for storage

Query time:

For every node on

U→ x &

U→ b

we perform a 1D range query

⇒ Query time = \( \sum_{u \in U : \text{U} \rightarrow a_x \land \text{U} \rightarrow b} O(\log n + k_u) \)

= \( O(\log^2 n + k) \) since \( \sum_{u \in U} k_u = k \)

And the sum runs over \( O(\log n) \) commands
Higher dimensions $\mathbb{R}^d$

- Idea: Apply the idea from 1D to 2D again (and again)

- Data structure with:
  
  Storage: $O(n \log^{d-1} n)$
  
  Preproc: $O(n \log^{d-1} n)$
  
  Query bin: $O(\log^{d-1} n + k)$

An Alternative: kd-trees [Bentley '75]

- Use only 1 tree and subdivide the x and y coordinates alternating

Example
(Construction : Storage : $O(n)$
preprocessing : $O(n \log n)$
(present coordinates simplify the median search)

Queries
Every node $v$ in the tree covers a range region ($v$)

"Scan" the tree for points
As I can stop at $v$ if (i) Region ($v$) contained in the query range or (ii) Region ($v$) and query range are disjoint
or otherwise recurse

Query time
Step (i) can be done in linear time in $k$

→ How often do I have to recurse?

Answer : # of regions containing a part of the query region
Theorem: Every A query region contains at most $O(r^2)$ regions partially proof skipped (not too hard).

Consequence: Query time: $O(r^2 + k)$

Back to Range Trees: Fractional Cascading

Speed up in the query time from $O(\log^2 n + k)$ to $O(\log n + k)$

Basic Idea:

1. Use the range tree as 1st level DS
2. 2nd level DSs are stored as arrays
3. We "wire" the different arrays to reuse information

Example

Rule: Pointer $x \rightarrow y$ s.t. $y \geq x \land \forall y' \in A_1, y' \leq y : y' < x$

The smallest entry in A\_1 greater equal x
How does this help?

Assume we know the "place" of $y_{min}$ in the array:

$y_{min} = 5$ => Search for entry "6" and report all cells from this until $y_{max}$ is reached.

Expensive is the search for $y_{min}$, the best is $O(k)$.

But we can reuse the result of the search in later searches.

That's why we have the pointers!
Analysis

(1) Search for UpSet \( O(\log n) \)

(2a) Follow to the left and "report" all right supers

(2b) Follow to the right and "report" all left supers

The "report" has to scan for the right y-coordinates, but with help of the fractional cascading pointers we have to "search for yMin" only once.

Takes \( O(\log n) \) and the report by only \( O(\log) \)

\[ O(\log^4 + b) \] query time
\[ O(n \log n) \] space
\[ O(n \log^4) \] prepossessing

Chazelle '76 improved this idea to \( O(\log n / \log \log n) \) storage

which is optimal

See also Chazelle for higher dimensions
General Point Sets

Drop to know we assumed the all coordinates are distinct!

How to handle general point sets?

Use composite numbers

\[(x, y) \rightarrow (x', y') \in \text{Con} \subseteq \text{composite number space}\]

\(\text{Con}\) has a natural order (lexicographic)

\[(x_1, y_1) < (x_2, y_2) \iff x_1 < x_2 \text{ or } (x_1 = x_2 \land y_1 < y_2)\]

Lemma: Let \(R = \overline{[x_1, x_2]} \times \overline{[y_1, y_2]}\) a query region then

\[p \in R \quad \implies \quad \hat{p} \in R = (p_x, p_y), (p_3, p_x)\]

with \(\hat{p}\) composite number of \(p\) and

\[\hat{R} = \overline{[x_1, \infty)} \times \overline{[y_1, \infty)} \times \overline{[y_1, \infty)} \times \overline{[y_2, \infty)}\]

Proof:

\[x_1 \leq p_x \leq x_2 \quad \implies \quad (x_1, \infty) \leq (p_x, p_y) \leq (x_2, \infty)\]

\[y_1 \leq p_y \leq y_2 \quad \implies \quad (y_1, \infty) \leq (p_3, p_x) \leq (y_2, \infty)\]