Computational Geometry

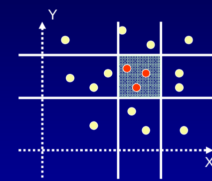
Chapter 5

Range Searching

15.

Orthogonal Range Searching

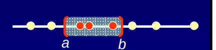
- **Problem:** Given a set of n points in R^d, preprocess them such that reporting or counting the k points inside a d-dimensional axis-parallel box will be most efficient.
- Desired *output-sensitive* query time complexity – $O(k+f(n))$ for reporting and $O(f(n))$ for counting, where $f(n)=o(n)$, e.g. $f(n)=\log n$.



- **Sample application:** Report all cities within 100 mile radius of Boston.

Range Searching – 1D

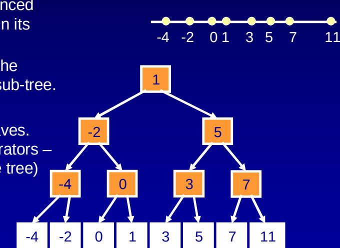
- In a one-dimensional world, points are real numbers and the query is two numbers (a,b).
- Simple $O(\log n)$ algorithm:
  - Preprocessing: Sort points in $O(n\log n)$.
  - Query: (Binary) search for a and b in list in $O(\log n)$.
    List all values in-between.



- Cannot be easily generalized to higher dimensions (why not ?).

Range Searching – 1D Tree

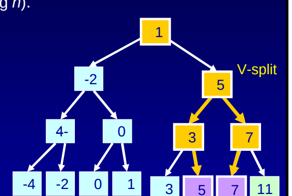
- Range tree solution:
  - Sort points.
  - Construct a binary balanced tree, storing the points in its leaves.
  - Each tree node stores the largest value of its *left* sub-tree.
- (keys are stored only in leaves. Internal nodes stores separators – for navigating search in the tree)



Range Searching in 1D Tree

Input Range: 3.5-8.2

- Required time for finding a leaf: $O(\log n)$.
- Find the two boundaries of the given range in the leaves u and v.
- Report all the leaves in *maximal subtrees* between u and v.
- Mark the vertex at which the search paths diverge as V-split.

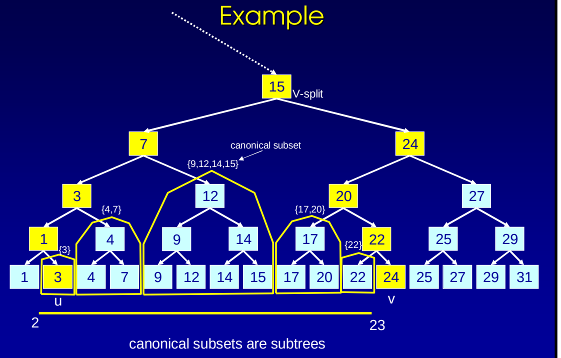


- Continue to find the two boundaries, reporting values in the subtrees:
  When going left (right), report the entire right (left) subtree.
- When reaching a leaf, check it exhaustively.

General Idea

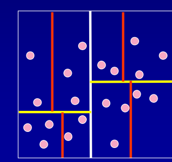
- Build a data structure storing a "small" number of canonical subsets, such that:
  - The c.s. (canonical subset) may overlap.
  - Every query may be answered as the union of a "small" number of c.s.
- The geometry of the space enables this.

Example



2D-Trees

- Given a set of points in 2D.
- Bound the points by a rectangle.
- Split the points into two (almost) equal size groups, using a horizontal or vertical line.
- Continue recursively to partition the subsets, until they are small enough.
- Canonical subsets are subtrees.



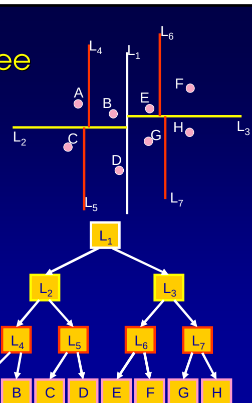
2D-Tree

- Partitions 2D space into axis-aligned rectangular regions.
- Nodes represent partition lines, and leaves represent input points.

construction complexity:

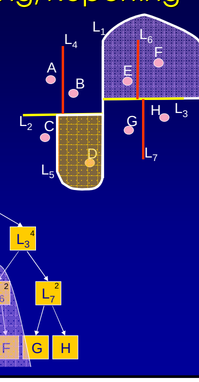
$$T(n)=\begin{cases}O(1) & n=1\\ O(n)+2T\left(\frac{n}{2}\right) & n>1\end{cases}$$

$$T(n)=O(n\log n)$$



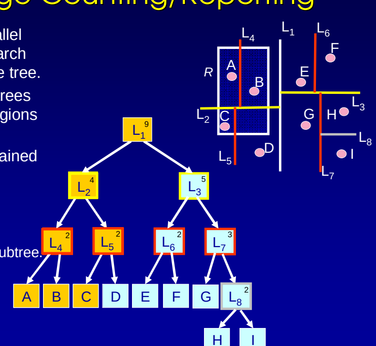
Range Counting/Reporting

- Each node in the tree defines an axis-parallel rectangle in the plane, bounded by the lines marked by this vertex's ancestors.
- Label each node with the number of points in that rectangle.



Range Counting/Reporting

- Given an axis-parallel range query *R*, search for this range in the tree.
- Traverse only subtrees which represent regions intersecting *R*.
- If a subtree is contained entirely in *R*:
  - **Counting**:
    Add its count.
  - **Reporting:**
    Report entire subtree.



Runtime Complexity

- How much time is spent on internal nodes? The nodes visited are those that are **stabbed** by *R* but not contained in *R*. How many such nodes are there ?
- **Theorem**: Every side of *R* stabs O(√*n*) cells of the tree.
- **Proof**: Extend the side to a full line (WLOG - horizontal):
  - In the first level it stabs two children.
  - In the next level it stabs (only) two of the four grandchildren.
  - Thus, the recursive equation is:

$$Q(n)=\begin{cases}1 & n=1\\ 2+2Q\left(\frac{n}{4}\right) & otherwise\end{cases}$$

- Total query time for reporting: O(√*n* + *k*). $=O\left(\sqrt{n}\right)$
- Total query time for counting O(√*n*).
- Total query time for emptiness ????

Kd-Trees – Higher Dimensions

- For a *d*-dimensional space:
  - Construction time: O($d\,n \log n$).
  - Space Complexity: O(dn).
  - Query time complexity: O($dn^{1-1/d}+k$).

Question: Are kd trees useful for non-orthogonal range queries, e.g. disks, convex polygons ?

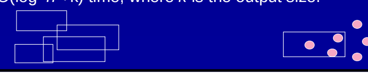
Fact: Using *interval trees* and *segment trees*, orthogonal range queries may be solved in O($\log^{d-1} n+k$) time and O($n \log^{d-1} n$) space.

Segment Trees

Segment trees are structures for storing sets *S* of *n* segments, which support the following operations:

- **insertion** of a segment
- **deletion** of segment
- **stabbing** queries:
  For a given point *p*, report all segments of *S* that contain *p* (that are stabbed by A)

In their 2D version, (2 levels trees) answer queries of the form: Preprocess a set *S* of axis-parallel rectangles, so that for every query point we can report all rectangle containing it in O($\log^2 n$ +k) time, where *k* is the output size.



Segment tree definition

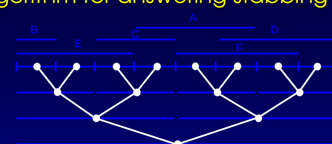
The minimal intervals (separating between segments endpoints) determine elementary intervals.

Construct hierarchy of the intervals: A binary tree of the minimal intervals (sorted), where each internal node *u* corresponds to an interval I_u, which is the union of minimal intervals of its decedent leaves. It also stores a set S_u of S of segments.

A segment *s* is in S_u iff <u>*u*</u> is the first node from the root, such that s contains I_u.

that is, s contains I_u but s does not contain $I_{parent(u)}$

Algorithm for answering stabbing queries



procedure report (*u*: node ; *x*: point):
report all segments of S_u ;
if *u* is leaf then finish else
{ if (*u* has left child *v* and $x \in I_v$)
then report(*v*, x);
if (*u* has left child *w* and $x \in I_w$)
then report(*w*, *x*); } //no need to visit both children

Using the segment tree all segment that contain a query point can be reported in time *O(log n + k)*, where *k* is the number of reported segments.

(a bit overkilling – can be done using an interval tree)

- The output can be expressed as O(log n) canonical sets.

Size of a Segment Tree

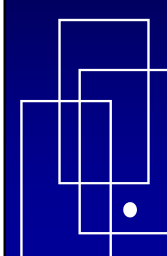
In each level of the tree, a segment *s* can be stored at at most two nodes.

Each segment of *I* appears in at the most *O(log n)* nodes.

Construction of a segment tree with *n* intervals is possible in time *O(n log n).*

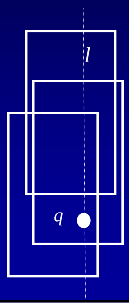
Applications for 2D – Query set of rectangles

Given a set *S* of n axis-para rectangles in 2D, preprocess *S* so that given a query point, we can find all points of S inside R in O(log *n*+*k*), where *k* is the output size



Query set of rectangles (cont)

Given a set S of n rectangle in 2D, all intersecting a vertical line l, preprocess S so that given a query point q on l, we can find all rectangle of S containing q in O(log n+k),

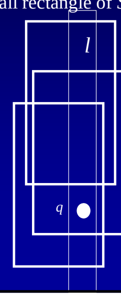


Answer:
Build a 1D segment tree on the intersection of each rectangle with l

Perform a query with q in this tree.

Query set of rectangles (cont)

Given a set S of n rectangle in 2D, all crossing a vertical strip l *from left to right* (their x-projection contains the x-projection of l), preprocess S so that given a query point q inside l, we can find all rectangle of S containing q in O(log n+k),

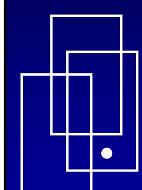


Answer:
Build a 1D segment tree on the intersection of each rectangle with l

Perform a query with q in this tree.

Applications for 2D

Given a set S of n axis-para rectangles in 2D, preprocess S so that given a query point, we can find all points of S inside R in O(log n+k), where k is the output size



Answer:
Build a 1D segment tree T on the projections of each rectangle on the x-axis.

Each node u in T corresponds to an interval I_u on the x-axis, and to a set S_u of rectangles whose x-projection contains I_u

Build a tree T_u on the y-projections of the rectangles in S_u

To answer a query $q=(x_0,y_0)$, find a set of O(log n) nodes of T whose interval I_u, contains x_0.
Query each tree T_u with y_0.

2D Segment tree -Analysis

Space – the size of T is O(n log n)
Each tree T_u has size O($|S_u|$ log $|S_u|$) = O($|S_u|$ log n)

Total space O(n $\log^2$ n)

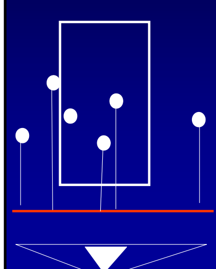
Query time: We perform a search in T, giving O(log n) nodes, and perform a search in each such node u,
(in O(log $|S_u|$)= O(log n))

Total O($\log^2$ n)

Construction – O(n $\log^2$ n)

Range space for points sets

Given a set S of n points in 2D, preprocess S so that given a query vertical strip R, we can find all points of S inside R in O(log n+k), where k is the output size

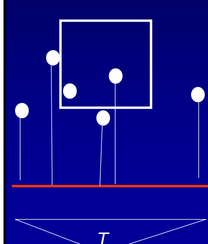


Since we care only about the x-axis of the coordinate, we construct a range tree on their x-coordinates. Once the query strip is given, express the answer as a union of O(log n) subtrees. (that is, canonical sets).

The range of reported canonical subset is fully contained inside $X(R)$ the X-projection of R

Applications for 2D

Given a set S of n points in 2D, preprocess S so that given a query axis-parallel rectangle R, we can find all points of S inside R in O($\log^2$ n+k), where k is the output size



First project the points on the x-axis, and built a range tree T.

Each node of T also points to a range tree of the points in these subset, sorted by their Y-coordinate.

Back to 1D – interval trees

Given a set *S* of *n* segments in 1D, preprocess *S* so that given a query point *q*, we can find all segments of *S* containing *q* in O(log *n*+*k*), where *k* is the output size.

Note – segment tree answers this query, but needs θ(n log n) space

We use interval trees:
The root is associated with the set $S_{root(T)}$ of all segments of S containing the point m, where *m* is the median of the endpoints of all segments of S.
Also store lists $L_{root(T)}$ and $R_{root(T)}$ of all endpoints sorted.

The left (resp. right) subtree is constructed recursively of all segment completely to the left (right) of *m*.