Computational Geometry

Geometric Data Structures

Vissarion Fisikopoulos

Department of Informatics & Telecommunications
National & Kapodistrian University of Athens

Spring 2025

Contents

Interval Tree

- Stores intervals $[l_i, r_i]$.
- Allows querying all intervals that overlap a query interval or point.
- Built on a balanced BST of midpoints.

Interval Tree: Construction

1. Choose a center point x_{center} (e.g., median of interval endpoints) to ensure balance.
2. Partition the intervals into:
   - S_{left}: intervals completely left of x_{center}
   - S_{right}: intervals completely right of x_{center}
   - S_{center}: intervals overlapping x_{center}
3. Recursively build subtrees for S_{left} and S_{right}.
4. Store S_{center} in two lists:
   - Sorted by interval start
   - Sorted by interval end

Each node stores:

- x_{center}
- Pointers to left/right subtrees
- Intervals overlapping x_{center}, sorted by start and end

Interval Tree: Querying with a Point

Find all intervals overlapping a query point x.

At each node:

- Compare x with x_{center}:
  - If $x < x_{center}$:
    - start enumerating intervals in the list until the startpoint value exceeds x
    - Recurse on the left subtree.
  - If $x > x_{center}$:
    - start enumerating intervals in the list until the endpoint value exceeds x
    - Recurse on the right subtree.
  - If $x = x_{center}$:
    - Report all intervals in S_{center}.

Interval Tree: Querying with an Interval

Find all intervals overlapping a query interval $q = [q_{\text{start}}, q_{\text{end}}]$.

An interval $r = [r_{\text{start}}, r_{\text{end}}]$ overlaps q if:

- $r_{\text{start}} \in q$ or $r_{\text{end}} \in q$; or
- r completely encloses q

Query strategy:

1. Use a search tree on interval endpoints:
   - Perform binary search for q_{start} and q_{end}.
   - Collect all intervals whose start or end lies within q.
   - Mark each interval to avoid duplicates.
2. Handle enclosing intervals:
   - Pick any point $x \in q$ (e.g., midpoint).
   - Use point query to find all intervals overlapping x.
   - Add only those that fully enclose q.

Interval Tree Complexity

Operations:

- Query: $\mathcal{O}(\log n + k)$
- Build: $\mathcal{O}(n \log n)$
- Space: $\mathcal{O}(n)$

Range Search

The problem of finding all points that lie within a given **query range** (interval, rectangle, box, etc.).

Input:

- A set S of n points in $\mathbb{R}^d$
- A query range Q

Output:

- All points $p \in S$ such that $p \in Q$

Applications:

- Database range queries
- Geographic information systems (GIS)
- Computer graphics and CAD

1D Range Search

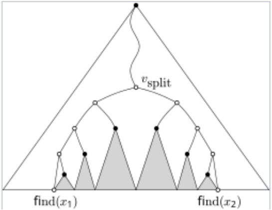
n points on the real line, report all points in interval $[x_1, x_2]$

A balanced binary search tree (BST) where:

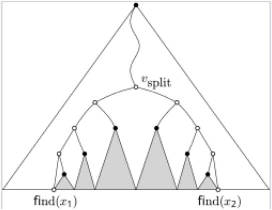
- Leaves store the points in sorted order.
- Internal nodes store the maximum of the left subtree.

Query Algorithm:

- Search for v_{split} the lowest common ancestor of x_1 and x_2.
- Traverse from v_{split} to x_1 and report all points in right subtrees of nodes where the path goes left.
- Traverse from v_{split} to x_2 and report all points in left subtrees of nodes where the path goes right.



1D Range Search



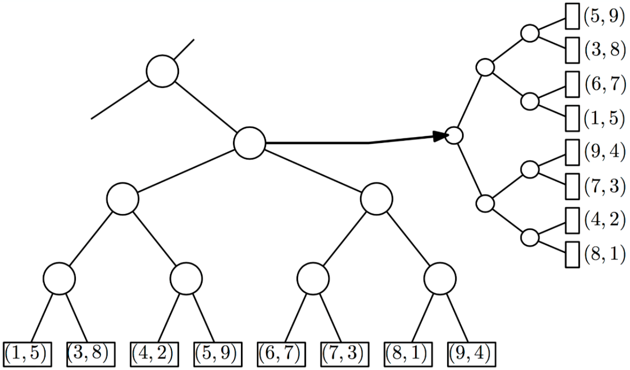
Complexity:

- **Preprocessing:** $\mathcal{O}(n \log n)$
- **Query:** $\mathcal{O}(\log n + k)$, where k is the number of reported points

Range Trees

- Construct a primary BST on the first coordinate.
- For each node v, build an associated $(d-1)$-dimensional range tree on the remaining coordinates of the points in v's subtree.
- Recursively apply this construction until 1D trees are reached.

screen shot from Mark van Kreveld slides, http://www.cs.uu.nl/docs/vakken/ga/slides5b.pdf)



Range Tree: Construction

1D Case:

- Construct a BST on the input points.
- Time complexity: $\mathcal{O}(n \log n)$.

2-**Dimensional Case:**

- Naïve construction time: $\mathcal{O}(n \log^2 n)$.
- Optimized: Two sorted lists of points: x and y-coordinate.
- Linear time to construct an associated tree on a node.
- Improved time: $\mathcal{O}(n \log n)$.

Optimized dD Construction: $\mathcal{O}(n \log^{d-1} n)$

2D Range Query Using Range Tree

Given a set S of n points in $\mathbb{R}^2$, report all points inside a query rectangle $[x_1, x_2] \times [y_1, y_2]$.

Query Algorithm:

- Find the split node v_{split} for $[x_1, x_2]$ in the primary tree.
- Traverse to x_1 and x_2 as in 1D range search.
- For each visited subtree rooted at node v:
  - Perform a 1D range query on the associated y-structure of v.

Complexity:

- $\mathcal{O}(\log^2 n + k)$
- Can be improved to $\mathcal{O}(\log n + k)$ using *fractional cascading*
- General dimension: $\mathcal{O}(\log^{d-1} n + k)$

k-d Tree

- Binary tree
- Each node splits space using a hyperplane *orthogonal to one axis*.
- The splitting axis cycles between x and y axis
- Left subtree: points with smaller coordinate along the axis.
- Right subtree: points with larger coordinate along the axis.

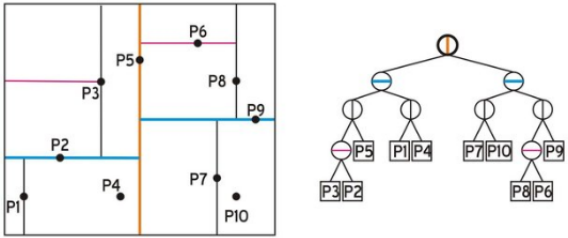


figure from "Registration of 3D Points Clouds for Urban Robot Mapping"

k-d Tree: Construction

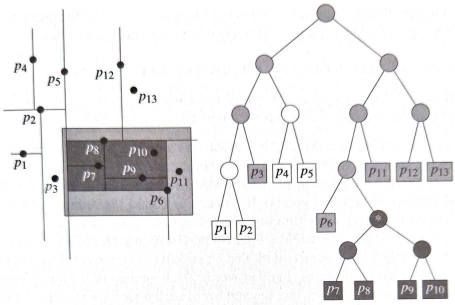
- At each level, choose a splitting axis (cycle through dimensions).
- Select the median point along that axis to balance the tree.
- Create a node storing the median point.
- Recursively construct left and right subtrees:
  - Left subtree: points with smaller coordinate.
  - Right subtree: points with larger coordinate.
- Keep one sorted list of points per dimension

Time: $\mathcal{O}(n \log n)$ **Space:** $\mathcal{O}(n)$

k-d Tree: Range Searching

Report all points inside a rectangular query region R

- At each node:
  - Check if the point at the node lies in R.
  - Compare the splitting coordinate with R:
    - If R lies completely on one side of the hyperplane, recurse only on that subtree.
    - If R straddles the hyperplane, recurse on both subtrees.



Complexity: $\mathcal{O}(\sqrt{n} + k)$

2D k-d Tree: Query Complexity

Key Idea: How many regions are intersected by a vertical (horizontal) line?

Recurrence Relation (2 levels):

$$Q(n) = 1, \ (n = 1)$$
$$Q(n) = 2 \cdot Q(n/4) + 2, \ (n > 1)$$

- After 2 levels, input size reduces to $n/4$ in each subproblem.
- Line may intersect up to 2 such subregions.

Complexity: $Q(n) = \mathcal{O}(\sqrt{n} + k)$

Nearest Neighbor Problem

Given a set of points $P \subset \mathbb{R}^d$ and a query point $q \in \mathbb{R}^d$, *find the point* $p^* \in P$ *closest to* q *according to a given distance metric (usually Euclidean).*

Data Structures for NN Search:

- **kd Tree:** Space partitioning via axis-aligned hyperplanes
- **Voronoi Diagrams:** Partition space into nearest neighbor regions

Nearest Neighbor Search in 2D using kd-trees

- **Traverse down the tree:** At each node, compare q with the node's splitting coordinate to decide which subtree to explore first.
- **Leaf node:** Record the point as the current best candidate.
- **Backtracking:** As recursion unwinds, check if the hypersphere around q with radius equal to the best distance found so far intersects the splitting line at the current node.
  - If yes, explore the other subtree as it may contain closer points.
  - If no, prune that subtree.
- **Update best candidate:** At each node visited, update the current best candidate.

Complexity: Worst-case: $O(n)$, in practice: $O(\log n)$.

Nearest Neighbor Search using Voronoi Diagrams

Reduce nearest neighbor search to a point location problem in the Voronoi diagram of the input point set.

Preprocessing:

- Given a set P of n points in $\mathbb{R}^2$, construct the Voronoi diagram $\mathrm{Vor}(P)$.
- Build a point location data structure (e.g., trapezoidal map) on top of $\mathrm{Vor}(P)$.
- Time complexity:
  - Voronoi diagram: $O(n \log n)$
  - Trapezoidal map (for point location): $O(n \log n)$ preprocessing

Query:

- Given a query point q, locate the Voronoi cell containing q using the trapezoidal map.
- Query time: $O(\log n)$

Range Tree and kd Tree

Structure	Construction	Space	Range Query	NN Query
k-d Tree	$O(n \log n)$	$O(n)$	$O(\sqrt{n} + k)$	$O(\log n)$ (avg)
Range Tree	$O(n \log n)$	$O(n \log n)$	$O(\log n + k)$	-

Notes:

- n: number of input points.
- k: number of reported points in range queries.

R-Tree

Height-balanced tree used for indexing spatial objects via their *Minimum Bounding Rectangles* (MBRs).

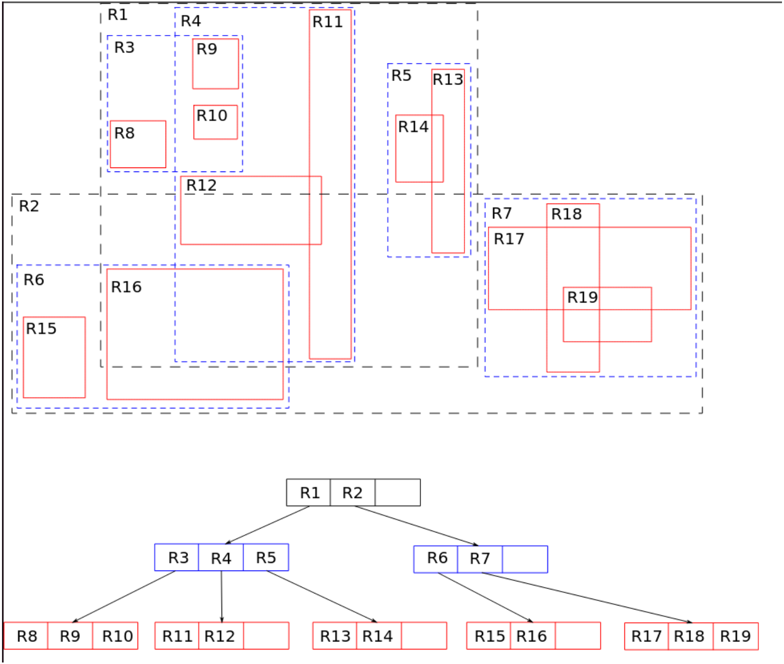
Structure:

- **Leaf Nodes:** Store actual data or pointers to data.
  - Point data: store the point and its ID.
  - Polygon data: store the polygon's MBR and a reference/ID.
- **Non-Leaf Nodes:** Store entries of the form:
  - `(MBR of child subtree, pointer to child node)`
- Bounding boxes in parent nodes tightly enclose all MBRs in their children.

Usage:

- Range search, nearest neighbor, intersection.
- Widely used in spatial databases and GIS systems.

R-Tree


R1
R4
R11
R3
R9
R5
R13
R10
R8
R14
R12
R2
R7
R18
R17
R6
R16
R19
R15
R1 R2
R3 R4 R5
R6 R7
R8 R9 R10
R11 R12
R13 R14
R15 R16
R17 R18 R19