CS 603 Project

Karanam Lokesh (210050081)
Varada Mahanth Naidu (210050161)

April 20, 2025

Problem Statement

Let P be a set of n points in the plane.

(a) Find a unit square(axis-aligned of side length 1) tht contains the maximum number of points from P.
Input: A set $P \subset \mathbb{R}^2$ of n points.
Output: An axis-aligned unit square that contains the largest possible subset of points from P.
Time Complexity: $O(nlogn)$ time.

(b) Maintain the above structure dynamically: a point is inserted into or deleted from the point set at each step.
Input: A dynamic sequence of point insertions and deletions.
Output: After each update, report a unit square that contains the maximum number of current points.

Part A : Static Version

Approach

We solve the problem by reducing it to a geometric problem involving square depth:

- Around each point $p_i \in P$, we place a square of side length 1 centered at p_i.
- We define the **depth** of a point $q \in \mathbb{R}^2$ to be the number of such unit squares that contain q.
- A point q with depth k is equivalent to saying an unit square centered around q has k points in it. For a point r to lie inside a square of side length 1 centered at point s, the condition is:

  $$|s_x - r_x| \leq \frac{1}{2} \quad \text{and} \quad |s_y - r_y| \leq \frac{1}{2}$$

  Suppose k squares centered at points $p_1, p_2, \ldots, p_k$ each contain a common point q. Then, for all $i = 1, 2, \ldots, k$, the following must hold:

  $$|p_{i,x} - q_x| \leq \frac{1}{2} \quad \text{and} \quad |p_{i,y} - q_y| \leq \frac{1}{2}$$

  This is equivalent to:

  $$|q_x - p_{i,x}| \leq \frac{1}{2} \quad \text{and} \quad |q_y - p_{i,y}| \leq \frac{1}{2}$$

  Therefore, all points p_i (for $i = 1, 2, \ldots, k$) lie inside the square of side length 1 centered at point q.
- By placing a unit square centered at the point with maxium depth, we ensure it encloses all the original points whose squares overlapped at q.
- Thus, finding the point with the maximum depth directly gives us the optimal unit square.

This transformation is visualized in the figure below (as provided), where overlapping squares contribute to the depth at each point.

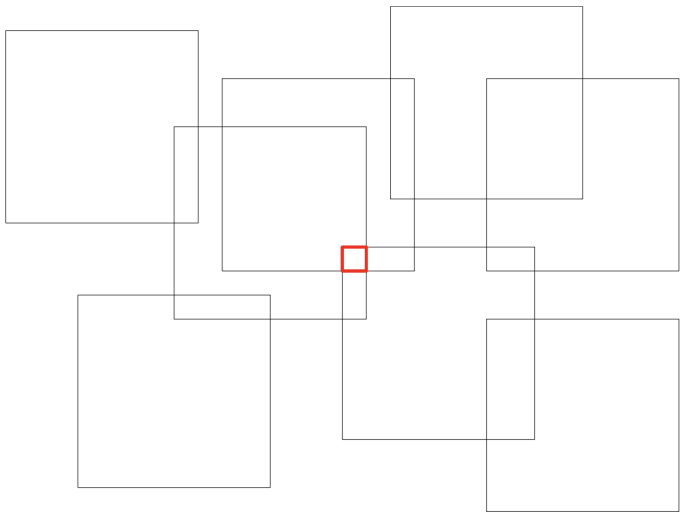

Figure 1: An arrangement of squares. The red area has depth 3, the maximum in the arrangement.

Algorithm Overview

We use a sweep-line approach combined with a segment tree with lazy propagation to efficiently track the overlapping intervals:

Algorithm 1: FindMaxPointsInUnitSquare

Input: A list P of n 2D points
Output: The unit square (side length 1) that contains the maximum number of points

Initialize empty list Y to store y-interval endpoints
foreach *point* (x, y) *in* P **do**
    Add $y - 0.5$ and $y + 0.5$ to Y
Sort Y and remove duplicates (coordinate compression)
Initialize empty list of events E
foreach *point* (x, y) *in* P **do**
    Add event at $x - 0.5$ with interval $[y - 0.5, y + 0.5]$ and type $+1$
    Add event at $x + 0.5$ with interval $[y - 0.5, y + 0.5]$ and type -1
    Convert y-intervals to compressed indices using Y
    Append both events to E
Sort all events in E by x-coordinate (breaking ties by type)
Initialize a Segment Tree ST over the compressed y-values
$maxCount \leftarrow 0$, $bestX \leftarrow 0$, $bestY \leftarrow 0$
foreach *event* e *in* E **do**
    ST.`rangeAdd`$(e.y_1, e.y_2, e.\texttt{type})$
    $(curMax, yIndex) \leftarrow ST$.`getMax`$()$
    if $curMax > maxCount$ **then**
        $maxCount \leftarrow curMax$
        $bestX \leftarrow e.x$
        $bestY \leftarrow Y[yIndex]$
return $maxCount$ and axis aligned square centered at $(bestX, bestY)$ with side length 1

Time Complexity

Step	**Time Complexity**
Coordinate Compression (sorting $2n$ y-endpoints)	$O(n \log n)$
Event Creation (2 events per point)	$O(n)$
Sorting Events by x-coordinate	$O(n \log n)$
Segment Tree Initialization	$O(n)$
Segment Tree Updates and Queries (per event)	$O(\log n)$ per event
Processing All Events ($2n$ events)	$O(n \log n)$
Overall Time Complexity	$\boldsymbol{O(n \log n)}$

Output

After processing all events, the point of maximum depth gives the maximum number of points enclosed and the center of the optimal axis aligned unit square.

Part B : Dynamic Version

Changes from the Static Version

Compared to the original static implementation, the modified code introduces support for **dynamic insertion and deletion** of points. The key differences are outlined below:

- **Event Management:** Events are maintained using a `map<double, multiset<Event>>` called `x_event_map`, which supports efficient insertion and deletion of events associated with each point.
- **Point Tracking:** A `set<pair<double, double>> active_points` is used to track currently active points in the plane.
- **Dynamic Segment Tree:** Instead of a static array-based segment tree, a pointer-based recursive segment tree is used, with each node supporting lazy propagation and dynamic rebuilding.
- **Interactive Commands:** The code accepts runtime commands in the form `insert x y` and `delete x y` to update the current point set and recompute the result accordingly.
- **Output:** After each update, the program writes the list of active points, the maximum overlap count, and the coordinates of the optimal axis-aligned unit square to `out.txt`.

Dynamic Nature of the Algorithm

The modified algorithm supports real-time updates to the point set, recomputing the axis-aligned unit square of side length 1 that covers the maximum number of points after each change.

Dynamic Point Management and Data Structure Enhancements

The updated code introduces support for dynamic insertion and deletion of points, building upon the previous static implementation. Below are the key changes made to enable dynamic behavior and how the underlying data structures have been modified:

1. Dynamic Behavior via User Commands

Unlike the earlier static version that computed the optimal unit square once for a fixed set of points, the updated implementation allows:

- `insert x y`: Inserts a new point into the set.
- `delete x y`: Deletes an existing point.

After each command, the system recomputes the axis-aligned unit square that contains the maximum number of points.

2. Event Management with Coordinate Mapping

To support dynamic updates, we replaced the static list of events with a `map<double, multiset<Event>> x_event_map`. This enables:

- Efficient insertion and deletion of sweep-line events based on the x-coordinate.

- Grouping of events by x-coordinate to allow sweep-line processing during recomputation.

Additionally, we maintain a global vector `x_vals` to keep track of all x-coordinates where events occur. This vector is re-sorted during each recomputation.

3. Active Points Set

We introduced a `set<pair<double, double>> active_points` to:

- Track all currently active points in the plane.
- Prevent duplicate insertions or deletions of non-existent points.
- Output the current point set state for debugging or logging.

4. Segment Tree Upgrade to Pointer-Based Nodes

To better support rebuilding the segment tree during each recomputation, the tree structure was changed:

- From an array-based implementation with fixed size.
- To a pointer-based recursive implementation using a `Node` struct.

Each node stores:

- Interval bounds `[l, r]`.
- Maximum value `maxv` and its position `maxpos`.
- Lazy propagation value `lazy`.
- Child pointers `left` and `right`.

This allows the entire tree to be reconstructed cleanly and safely using the destructor when the point set changes.

5. Recompute Function

The core of the dynamic behavior is handled in the `recompute()` function:

- It sorts the `x_vals`.
- Updates the segment tree using the current set of y-intervals.
- Processes sweep events in sorted x-order.
- Updates the segment tree using `range_add()` based on event type.
- Identifies the unit square with the maximum overlapping points.

This mechanism allows for exact recalculation of the best axis-aligned unit square after every point update while maintaining correctness.

Conclusion

The dynamic version significantly enhances the usability of the original algorithm by supporting runtime updates to the dataset. With careful use of maps, sets, and a pointer-based segment tree, it maintains efficient performance while allowing for interactive geometric queries. This makes the solution suitable for applications requiring real-time spatial analysis.