CONVEX HULL IMPLEMENTATION

a)Chan's Algorithm for Convex Hull

1 Introduction

This project implements **Chan's Algorithm** for computing the convex hull of a set of 2D points. The algorithm combines the advantages of Graham Scan and Jarvis March to achieve an optimal output-sensitive time complexity of $O(n \log h)$, where n is the number of input points and h is the number of points on the convex hull.

2 Overview of the Implementation

The implementation is done in Python and visualized using `matplotlib`. The main steps of the algorithm are as follows:

- Partition the input set of n points into groups of size at most m.
- Compute the convex hull of each group using Graham Scan.
- Merge the hulls using an enhanced version of Jarvis March by finding tangents via binary search.
- Repeat the process with an increased value of m (exponentially) until the complete convex hull is found.

3 Graham Scan for Sub-Hulls

The function `graham_scan` constructs the convex hull for a subset of points by maintaining the upper and lower hulls separately. It works in $O(m \log m)$ time per group, where m is the group size.

4 Binary Search for Tangent Finding

A critical step in Chan's Algorithm is efficiently finding the tangent from a point to a convex polygon. Instead of scanning the polygon linearly, this implementation uses a **binary search** approach to achieve $O(\log m)$ time per group.

Binary Search Procedure

- The polygon is assumed to be sorted in counter-clockwise (CCW) order.
- For a given external point p, we search for a point q on the polygon such that the line segment $\overline{pq}$ is tangent to the polygon.
- The function `is_tangent` checks whether a given vertex satisfies the tangent property by evaluating cross products with neighboring vertices.
- Binary search proceeds by comparing the cross product $\texttt{cross}(p, curr, next)$ to guide the search.

This approach reduces the tangent-finding step from $O(m)$ to $O(\log m)$, making the overall merging step efficient.

5 Main Algorithm: Chan's Hull Construction

- The `chan_hull` function orchestrates the process of computing group hulls and incrementally building the global convex hull.
- For each step, the best candidate point is selected based on the angle formed with the previous two points.
- If the hull closes (returns to the starting point) within m steps, the result is returned.
- Otherwise, m is increased exponentially and the process repeats in `full_chan_hull`.

6 Visualization

The program uses `matplotlib` to:

- Draw input points and label them.
- Show each group in a different color.
- Display group hulls and the final merged convex hull.

Output

The final convex hull is printed in clockwise (CW) order. The plot shows the hull with line segments and labeled points.

7 Key Functions and Logic

- `draw_points, draw_polygon`: For plotting.
- `find_angle, cross`: For geometric calculations.
- `binary_search, is_tangent`: Core of the tangent-finding mechanism.
- `graham_scan`: Computes convex hull for each group.
- `chan_hull, full_chan_hull`: Main algorithm functions.

Time Complexity

- Graham Scan per group: $O(m \log m)$.
- Binary Search for tangents: $O(\log m)$.
- Merging phase: $O(h \log m)$, repeated $\log \log n$ times.
- Overall: $O(n \log h)$.

8 Conclusion

This implementation of Chan's Algorithm uses binary search to optimize tangent finding, making the convex hull construction efficient for large datasets. The visualization helps in understanding the grouping, local hulls, and final merged result clearly.

b)Dynamic Convex Hull using Segment Tree

1 Abstract

This report presents a Python implementation of a dynamic convex hull using a segment tree data structure. The convex hull of a set of 2D points is the smallest convex polygon enclosing all the points. Supporting insertions, deletions, and queries efficiently on a dynamic set of points is essential in computational geometry, robotics, and motion planning. Our implementation uses a hierarchical approach inspired by the Overmars and van Leeuwen technique, and supports interactive updates with convex hull visualization.

2 Introduction

The convex hull problem is a classical problem in computational geometry. While static algorithms like Graham scan and Andrew's monotone chain are well-known, maintaining the convex hull dynamically under insertions and deletions is more complex. This report implements a segment-tree-based structure for dynamic convex hulls that supports:

- Insertion of a point.
- Deletion of a point.
- Query whether a given point lies on the convex hull boundary.
- Displaying and visualizing the convex hull.

3 Overview of the Algorithm

3.1 Segment Tree Construction

A balanced segment tree is built over the x-coordinate range $[-10^6, 10^6]$. Each node in the tree corresponds to a range of x-values and stores:

- `points` list: all points with x-values in the node's range.
- `hull` list: the convex hull of points in the node.

3.2 Merging Phase

When inserting or deleting a point, we update all segment tree nodes whose range includes the x-coordinate of the point. The convex hull for each node is rebuilt using the `build_hull` function. This function merges the upper and lower hulls for a set of sorted points using the cross product method.

3.3 Binary Search (Bisect)

Insertion and deletion maintain sorted order in each node using Python's custom `bisect_left` and `insort` functions. These use binary search to achieve efficient insertions ($O(\log n)$ per node).

4 Point Location Query

To check whether a point lies on the convex hull, we perform a linear scan on the edges of the root node's hull. For each edge (a, b), we check:

- Whether the point lies on the line formed by a and b using the cross product.
- Whether the point lies between a and b using coordinate comparisons.

Future Optimization

Currently, point-on-hull queries take $O(h)$ time, where h is the number of points on the convex hull. Using binary search on hull segments sorted by angle or slope can reduce this to $O(\log n)$ time.

5 Time Complexity

Let n be the total number of inserted points.

- **Insert/Delete:** $O(\log^2 n)$
  Each update affects $O(\log n)$ nodes in the segment tree, and each node rebuilds its hull using $O(k \log k)$ where k is the number of points in that node. On average, this yields $O(\log^2 n)$ amortized time.
- **Query (Point on Hull):** $O(h)$
  Currently implemented using a linear scan over all hull edges. This can be optimized to $O(\log n)$ using binary search on the convex hull segments.
- **Space:** $O(n \log n)$
  Each point may be stored in $O(\log n)$ segment tree nodes.

6 Visualization

The implementation includes a plotting function using `matplotlib` to visualize the current state of the convex hull and its points. It labels all points and plots the hull in counter-clockwise order.

7 Sample Run

```
Options:
1) Insert point
2) Delete point
3) Query if point on hull
4) Print & Plot convex hull
5) Exit
```

```
Enter choice: 1
Enter x y: 2 3
Inserted (2, 3)
```

References

1. Overmars, M. H., & van Leeuwen, J. (1981). "Maintenance of Configurations in the Plane." *Journal of Computer and System Sciences*, 23(2), 166–204.

2. Preparata, F. P., & Shamos, M. I. (1985). *Computational Geometry: An Introduction*. Springer-Verlag.

3. Chan, T. M. (1996). "Optimal output-sensitive convex hull algorithms in two and three dimensions." *Discrete & Computational Geometry*, 16(4), 361–368.

Contributors

Submitted by:
Sanku Venkatesh (24M2002)
Narayana S S (24M2010)