CS 388R: Randomized Algorithms, Fall 2023 November 6th, 2023

Lecture 21: Low Dimensional Computational Geometry

Prof. Eric Price *Scribe: James Rayman, Jasmeet Kaur*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

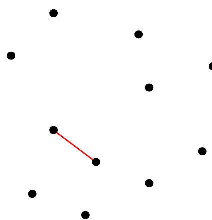
1 Overview

In this lecture, we begin discussing computational geometry. We will look at randomized solutions to a few classic problems, namely closest pair of points, convex hull, and intersecting half spaces.

We will focus specifically on low dimension problems today. That is, we will assume that our dimension d is constant in the analysis of our algorithms.

2 Closest Pair of Points

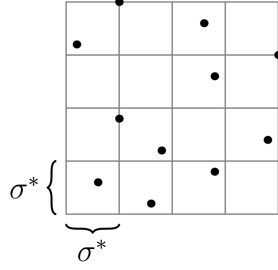
Definition 1. In the closest pair of points problem, we are given n points $p_1, p_2, \dots p_n$ and have to find two points from that list p_i and p_j, $i \neq j$ such that $\|p_i - p_j\|$ is minimized.



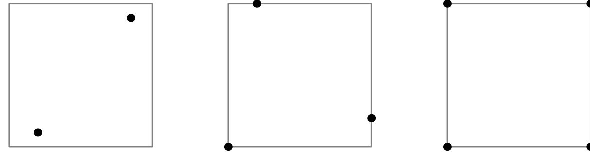
Each pair of points can be checked which takes $O(n^2)$ time.[CLRS] gives an $O(n \log n)$ time divide-and-conquer solution to this problem, but today we will see a randomized $O(n)$ expected time solution.

2.1 Warm Up

Let σ^* be the minimum distance between two points. To find p_i and p_j., we can divide the space into a grid of squares where each cell is $\sigma^* \times \sigma^*$ and use a hash map to quickly look up points are in each cell.



To give an intuition, since we are looking for a pair of points σ^* apart, we only need to check for pairs that are in the same cell or two adjacent cells (including diagonally adjacent ones). Each cell can have a maximum of 4 points. By using the hash map, we are able to determine, for any point p_k, the at most 35 additional points that might be within σ^* of p_k. We can find p_i and p_j in $O(n)$ time.



2.2 Full Solution

If we aren't given σ^*, we can take a random guess σ and run the algorithm above. If our guess is sufficiently close, then our algorithm works just fine. Otherwise, we have the following two cases :

- If $\sigma < \sigma^*$, then the closest pair of points might not be in adjacent cells, in which case we can't calculate the answer.
- If $\sigma > \sigma^*$, then way more than 4 points could end up in a single cell. It takes $O(n^2)$ time. We will see in $O(n)$ time that our guess for σ was too high and we can restart the algorithm with σ equal to the minimum distance we have seen so far.

Modifying this idea a little, we get this algorithm:

1. Randomly shuffle all the points.
2. Set $\sigma = \|p_1 - p_2\|$.
3. Initialize an empty hash map for a $\sigma \times \sigma$ grid.
4. For each p_k of $p_1, p_2, \ldots, p_n$, add p_k to the grid and calculate the distances to all points in the same cell and each neighboring cell. If we find that p_k is less than σ away from another point, set σ to be this new minimum distance and restart the algorithm from step 3.

At the end, we know that $\sigma = \sigma^*$, and if we kept track of what points we used to calculate σ, we know p_i and p_j.

Analysis : Steps 1, 2, and 3 always take $O(n)$ time all together. For step 4, notice that adding a point without restarting takes $O(1)$ time, and a given point can make us restart at most one time. Thus, restarting at the kth point takes $O(k)$ time, so we have:

$$\mathbb{E}[\text{run time}] = \sum_{k=1}^{n} (O(1) + \mathbb{P}[\text{restart at round } k] O(k)) \tag{1}$$

Now:

$$\mathbb{P}[\text{restart at round } k] = \mathbb{P}[\text{closest pair}(\{p_1, p_2, \ldots p_k\}) \text{ contains } p_k]$$

We break ties for closest pair by the index of the second point, since that is the order that we account for the pairs in our algorithm.

Since we shuffled the points at the beginning, if there are no ties, every point is equally likely to be in the closest pair. There is a $2/k$ chance that p_k is in the closest pair. If there are ties for closest pair :

- All the closest pairs share a common point. We restart with probability $1/k$.
- No point belongs to every closest pair and we never restart.

In all, we have:

$$\mathbb{P}[\text{closest pair}(\{p_1, p_2, \ldots p_k\}) \text{ contains } p_k] \leq 2/k$$

Substituting into (1), we get that our expected run time is $O(n)$.

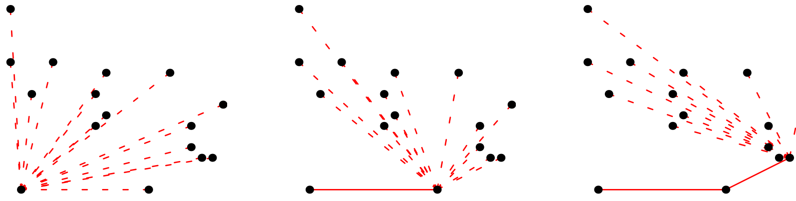
3 2D Convex Hull

Definition 2. A set of points is in general position if no three points are collinear.

Definition 3. In the convex hull problem, we are given n points $p_1, p_2, \ldots p_n$ in general position. We have to identify a subset of those points $p_{i_1}, p_{i_2}, \ldots p_{i_k}$ that form a convex polygon which encloses all n points.

[CLRS] describes two algorithms for convex hull:

1. Gift wrapping : $O(nk)$ (k is the number of points on the hull). Starting with the point with the lowest y-coordinate (any point on that is guaranteed to be on the hull works), given any current point, pick the next point that makes the largest angle with the current point in counterclockwise direction. In worst case, runs in $O(n^2)$ time when all the points are on the hull.

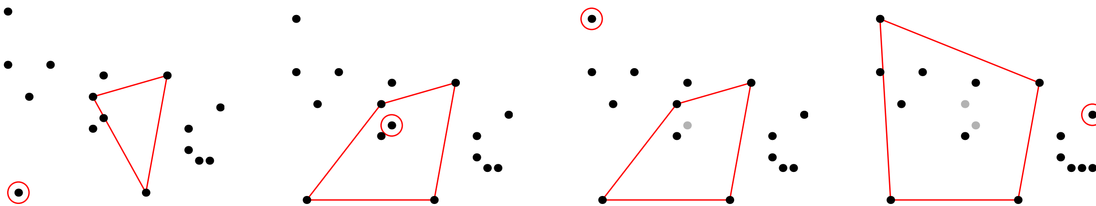


2. Graham's scan : $O(n \log n)$. Start by sorting all the points by angle with the starting point. This takes $O(n \log n)$. Walk around this sorted array to pick points that lie in the hull which takes $O(n)$. We will present a similar algorithm later in this lecture.

Today, we will look at an expected $O(n \log n)$ time algorithm.

3.1 A Simple Approach

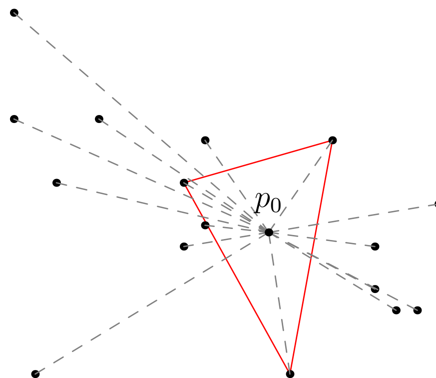
Let's consider the following general algorithm: Keep a list of candidate points for the hull, initially $\{p_1, p_2, p_3\}$. Add the rest of the points one by one, and after each addition, compute the new hull.



Each point is added once and removed at most once, so our run time is $O(n)$ plus the time it takes to decide which points to remove. Naïvely, we can get $O(n)$ per point ($O(n^2)$ total), but being more clever lets us achieve $O(\log n)$ per point ($O(n \log n)$ total).

3.2 A Fast Deterministic Approach

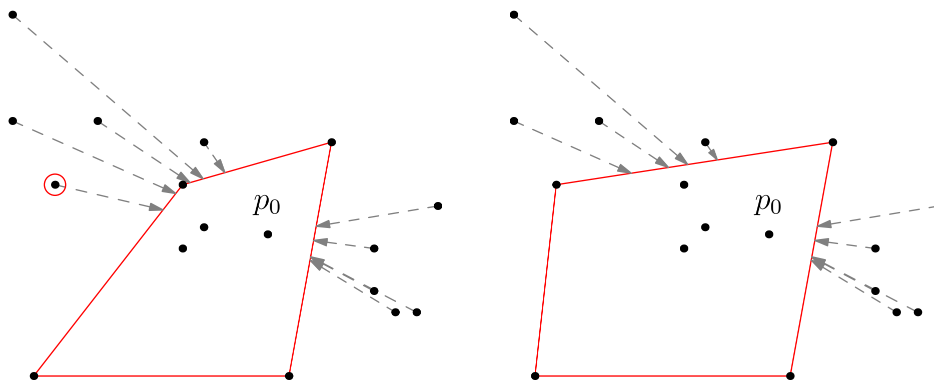
First, take p_0 to be any point on the interior of the hull ($(p_1 + p_2 + p_3)/3$ suffices) and sort the remaining points by angle θ relative to p_0.



Now when we add a point p_i, we need to find what edge $\overline{p_i p_0}$ intersects, if any. We can do this by binary searching on θ, comparing to points on the current hull. Since we are adding and removing points to the hull, we will need to store the hull as a binary search tree. This gives us the $O(\log n)$ time per point we mentioned above.

3.3 A Fast Randomized Approach

Instead of sorting, we instead store pointers from each point p_i to the edge that $\overline{p_0 p_i}$ intersects (if there is no intersection point, we can remove p_i). Every round, edges could be removed and added, so we need to update some edge pointers. If we randomly pick which point to add in each round, we can show that our expected run time is $O(n \log n)$.



3.3.1 Analysis

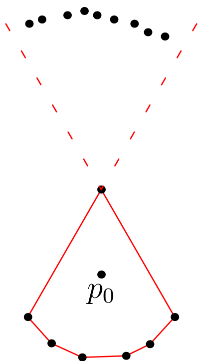
Since we can keep a list of backpointers on each edge and at most two edges are added each round, determining which edges pointers to update and what to update them to is $O(1)$ per edge pointer update. Determining which edges to remove is also fast since we always remove a contiguous subset of edges. Thus, our run time is proportional to the total number of edge pointer updates.

3.3.1.1 A Wrong Approach

We might hope for an inequality like this:

$$\text{On round } i,\ \mathbb{P}[\text{the pointer for } p_j \text{ updates}] \lesssim \frac{1}{i}$$

However, this is not true in all cases, as demonstrated by the following counterexample:



Here, no matter which point we choose to add, we will have to remove two edges and update all the edge pointers, so this approach doesn't work.

3.3.1.2 Backwards Analysis

Instead, let's try working backwards. Suppose we are given S, the set of i points that our current hull encloses after $i-3$ rounds. If we remove a random element of S (i.e. step one round backwards in the algorithm), what is the number of edges pointers we need to update? Well:

$$\mathbb{E}[\text{number of pointer updates}] \leq n\,\mathbb{P}[\text{the pointer for } p_j \text{ updates}]$$

The edge pointed to p_j only has two endpoints, and we only update the pointer if one of those points is removed, so:

$$\mathbb{P}[\text{the pointer for } p_j \text{ updates}] = \frac{2}{|S|} = \frac{2}{i}$$

Combining these two results and summing over all rounds, we get:

$$\mathbb{E}[\text{total number of pointer updates}] \leq n \sum_{i=4}^{n} \frac{2}{i} \lesssim n \log n$$

Thus, the expected running time is $O(n \log n)$. This isn't any better than Graham's scan, which is deterministic, but our algorithm has the benefit that it can generalize to higher dimensions.

4 Intersecting Half Spaces in $\mathbb{R}^3$

Definition 4. A half space in $\mathbb{R}^3$ is a plane together with every point on one side of that plane.

If the intersection of a set of half spaces is bounded and has positive volume, then this intersection is a polyhedron.

Definition 5. A set of half spaces is in general position if its intersection is a polyhedron where every vertex is incident to 3 edges.

Definition 6. In the intersecting half spaces problem, we are given n half spaces in general position and have to find the polyhedron that defines the intersection.

Using Euler's characteristic formula ($V - E + F = 2$) and the fact that $2E = 3V$ from the general position constraint, we get that the resulting polyhedron has $O(n)$ edges and $O(n)$ vertices.

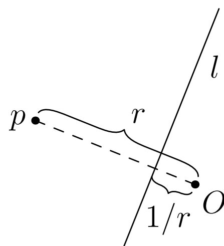
To solve this, we can employ a similar algorithm to 2D convex hull: Add the half spaces to the intersection one by one, and for each half space we haven't added, keep a pointer to a vertex that is not in the half space. After each addition, we will need to update vertex pointers, and our total time is proportional to the number of vertex pointer updates.

4.1 Backwards Analysis

The analysis is very similar to that of convex hull. Suppose our current set of considered half spaces is S and we step backwards in the algorithm, removing a half space from S. Since each vertex is incident to only 3 half spaces, there is at most a $3/|S|$ chance that a given vertex pointer updates. The rest of the analysis is almost the same as for convex hull, and we again get a $O(n \log n)$ expected time bound.

4.2 Duality

Definition 7. A point p and a line l are dual to each other with respect to the origin O if l is perpendicular to $\overleftrightarrow{pO}$ and the distance from l to O is $1/pO$.



This notion of duality extends to any dimension (replacing l with a plane or hyperplane), and many properties are preserved by taking the dual. For example a set of points is in general position if and only if the dual of the set is in general position. Another important property is that the dual of convex hull is intersecting half spaces, so we can use the algorithm we just described to compute 3D convex hull.

4.3 Delaunay Triangulation

One application of 3D convex hull is computing the 2D Delaunay triangulation of a set of points.

Definition 8. The Delaunay triangulation of a set of points $p_1, p_2, \ldots, p_n$ in general position is the triangulation T of the set of points that maximizes the minimum angle of any triangle in T.

To calculate the Delaunay triangulation of $p_1, p_2, \ldots, p_n$, we can map each $p_i = (x_i, y_i)$ to $(x_i, y_i, x_i^2 + y_i^2)$, compute the 3D convex full of the resulting set of points, and project the bottom of the hull back to $\mathbb{R}^2$.

5 Closing Remarks

Notice that in all the algorithms we discussed today, the run time is exponential in the dimension d. In closest pair, we need to check a neighborhood of 3^d cells for every point. For intersecting half-spaces (and convex hull), the number of vertices in the resulting polytope is $n^{O(d)}$. This is fine for spatial applications where the dimension is 2 or 3, but for other applications, such as reverse image search, d could be in the thousands or millions.

Next lecture, we will discuss the nearest neighbor search problem in high dimensions, which has reverse image search as an application.

References

[CLRS] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*, Third Edition (3rd. ed.). The MIT Press.