Brigham Michaelis
March 26, 2018
A00364835

Computational Geometry Assn.4

1. Let H be a set of n half-planes. Let Hu be the subset of all upper half-planes of H and let Hl be the subset of all lower half-planes of H. Let Cu be the upper envelope of the bounding lines of Hu and let Cl be the lower envelope of the bounding lines of Hl .

Let C be the common intersection of all half-planes of H. As discussed in class, we can compute C by using Cu and Cl . Suppose we have already computed Cu and Cl (for each of them, you are given a list of their vertices sorted from left to right). Design an O(n) time algorithm to compute C by using Cu and Cl. **(20 points)**

Algorithm Description
The solution to this problem was found while working with John Johnson, Josh Dawson, and Jiyao.

Before we provide the algorithm description we need to give some definitions.
UpperEdgeLine: This is a line that is made from the first two vertices given in Cu extending infinitely in both directions.
LowerEdgeLine: This is a line that is made from the first two vertices given in Cl extending infinitely in both directions.

In the case of the first and last vertices given in Cu and Cl, we treat those as lines in and of themselves.

Now that we have those definitions we present our algorithm.

We are going to process the points in the two lists Cu and Cl one at a time. In this fashion. From the two lists we pick the point with the smallest x-value. And we follow this reasoning:

If this point came from Cl we ask:
    Is this point above the UpperEdgeLine?
        If yes than this point is in the solution set and we mark it as such.
        If not than we remove this point from Cl and repeat.

If this point came from Cu we ask:
    Is this point below LowerEdgeLine?
        If yest than this point in the solution set and we mark it as such.
        If not than we remove this point from Cu and repeat

The idea behind this scheme is that we need to classify each point as either being part of the solution or something that can be discarded. If it is part of the solution we mark it as such, otherwise we remove that point from the set. Either way we are progressing through the vertices in Cu and Cl one at a time, and recalculating UpperEdgeLine and LowerEdgeLine. Also when a point is found to be part of the solution we mark it such and move to the next point in the sets with the smallest x-value. This algorithm has the added benefit of working even when C is unbounded. By doing this UpperEdgeLine and LowerEdgeLine are going to be updated as we process the points one by one.

There are few cases that can arise with C being bounded or unbounded. However the point is that if C is closed(not bounded) Then we know that Cu and Cl intersect twice. With the algorithm given above the intersections can be detected. This is done by noting when a change has taken place. For example, when we start to run our algorithm we are either going to start by processing vertices that are part of the solution set or are not part of the solution set. The key is that when this changes, i.e… when we switch from processing vertices in the solution set to processing vertices that are to be removed we have hit an intersection, and vice versa. Going from vertices that are to be removed to vertices that are in the solution set.

The key is to count the number of intersections that occur and where they occur. If there are two intersections then we know that C is bounded. If there is only one intersection than we know that C is unbounded, and the vertices at which C is unbounded are easily discovered. If there are zero intersections than C is still unbounded.

Psuedocode
Given Above
Correctness
This algorithm works because we are processing all of the points just once and we are doing constant work at each step. As we step through the points we are updating the UpperEdgeLine and the LowerEdgeLine as points are classified and removed appropriately.
Time Analysis
The running time of our Algorithm is O(n) because we are simply processing all the vertices in the two lists Cu and Cl, and doing constant work at each step.

2. On n parallel railway tracks n trains are going with the same direction and with constant speeds v1 , v2 , . . . , vn . At time t = 0 the trains are at positions k1 , k2 , . . . , kn . Design an O(n log n) time algorithm that detects all trains that at some moment in time are leading (i.e., in the front of all other trains).
You may report your solution in the following manner: partition the time into many intervals [ti , ti+1] and associate each interval with a train such that when t ∈ [ti , ti+1] the associated train is leading. **(20 points)**

Algorithm Description
This problem is fun! We have n trains, each with it's own speed, and starting positions at time = 0. To get this problem into a half-planes format, we need to have something like: y = m*x+b. For each train. So let's break this down.

For each train:
y: is the position of the train at any time t.
m: is the velocity of the train also known as v, this is provided.
x: is the time t, this starts at 0 and simply continues.
b: is the intercept or the position of the train at time = 0. Seen above as k.

Now that we have an equation for a line for each train we are going to plot these all these lines on a graph. From this graph we are going to find the upper envelope for all of these lines(trains). The upper envelope of these lines can be found in O(n log n) time, using Graham's scan as discussed in class. Now when it comes to reporting our solution we are going to report, from left to right, all the edges of the upper envelope and the times at which they occur. These edges are the trains. They are simply represented by edge segments made from the vertices of the upper envelope. We then report the edge and the associated time that the edge occupied.

I loved this problem!
Psuedocode
Graham's Scan for finding the upper envelope of a set S of n lines in the plane.
Correctness
This algorithm works by taking the problem above, translating it into a half-planes problem, sorting the lines that are derived from the problem description, and outputing the solution based on the results of finding the upper envelope.
Time Analysis
The running time of this algorithm is O(n log n) and comes from Graham's scan for finding the upper envelope of a set of lines, as discussed in class.

3. Let S be a set of k convex polygons. Let n be the total number of vertices of all these convex polygons. Denote by C the common intersection of all convex polygons of S.

(a) Give an O(n log k) (not O(n log n)) time algorithm to compute C. **(15 points)**

Algorithm Description
We can merge two convex hull in linear time. For this problem we are just going to divide the set of convex hulls in half until we have split up the entire set into pairs and then continuously merge the pairs and each successive pair. In essence we are doing n work log k times. For this problem we would use the algorithm discussed in class for merging two convex hulls in linear time, but the trick is to not do any unnecessary work. We do this by splitting the set S into pairs and then we merge each pair. Then we take all the resulting convex polygons again and split them into pairs. Essentially each point is going to be looked at log k times(in the worst case), where k is the number of polygons.
Psuedocode
Not Required
Correctness
This works because no vertex will be processed more than log k times, and because there are n points we are doing “n work” log k times. This is very similar to many other problems that can be solved in similar time by using divide and conquer appropriately.
Time Analysis
The running time of our algorithm is O(n log k) because we can merge two polygons in linear time and as long as we are reducing the number of polygons in our set by half after every merge step then we can achieve a runtime of O(n log k).

(b) It is possible that C is empty. Give an O(n) time algorithm to determine whether C is empty. **(10 points)**

Algorithm Description
As discussed in class we have an algorithm that can find p* in linear time given a set H of n half-planes, but to use this algorithm we have to first make the half-planes from the set of convex polygons. This can be done by going through all the convex polygons and for every two consecutive points in each polygons we make a half-plane. Now we just apply the algorithm discussed in class to find p* in linear time. If a valid p* value is returned then we know that C is not empty. If nothing is returned then we know that C is empty.
Psuedocode
Not Required
Correctness
This works because we are making a set of n half-planes from every two consecutive points in the k convex polygons. Once we have these half-planes we can then apply the Lowest point algorithm discussed in class. If nothing is returned then p* does not exist and C is empty. Otherwise a point will be returned and C is not empty.
Time Analysis
This works because we are building the n half-planes in in linear-time from the set S of the k convex polygons. We then are applying a linear time algorithm on the set of half-planes to see if P* exists or not. The total runtime is O(n).

4. In this exercise, we consider the following one-dimensional one-center problem.
Let $P = \{p1 , p2 , \ldots , pn \}$ be a set of points (not necessarily sorted) on a line L. Each point pi has a weight $wi > 0$. For any point q on L, the weighted distance between q with any point pi of P is defined as $d(q, pi) = wi \cdot |q - pi |$, where $|q - pi |$ is the distance between q and pi on L. The one-center problem is to find a point q, called the center, such that the maximum weighted distance from q to all points of P (i.e., $\max 1 \le i \le n\ d(q, pi)$) is minimized.

(a) Suppose all points of P have the same weights (i.e., $w1 = w2 = \cdots = wn$). Design an O(n) time algorithm to compute the center. **(5 points)**

(b) Suppose the weights of the points of P are not necessarily the same. Design an O(n) time algorithm to compute the center. **(20 points)**

Hint: Reduce the problem to finding the lowest point in a set of O(n) upper half-planes and then simply apply the linear-time algorithm discussed in class.

Remark: In facility location theory in Operations Research, the points of P are usually called customers or clients (the weights may represent their importance) and the center is called the facility. The weighted distance between the facility and each customer is called service cost. The one-center problem is thus to choose a location on the line L to build a facility to serve all customers such that the maximum service cost is minimized.

We are going to give an algorithm that will work for both cases. Not just for a. Since we are able to solve case b, there is no need to solve case a. The same algorithm can be applied to both problems for a linear runtime.

Algorithm Description
The main idea of this problem is how to transform the problem statement into a plane of half-planes. Once we have a plane of half-planes we can then apply the lowest point algorithm to all the half-planes in the plane. The result from running the lowest point algorithm is that p* will be the point on the line that will result in the minimum service cost.
So how do we transform the statement into a plane of half-planes?

Psuedocode

Not Required

Correctness

This algorithm works because of the way that we are able to translate the problem statement into a set of half-planes on a plane. All of these half planes are upper half-planes and the slopes of these lines represent the weighted-distance of each weight at any given point on the line. The fact that the various points may have different weights makes no difference in the overall runtime. Now that we placed into a plane all of the half-planes we can simply use the Lowest Point algorithm given in class which can find p* in O(n) time given a set of half-planes.

But to be concise all you really need is the slope of each line, and you need to know the bounds of the line L and only consider the half-planes that are inside that length.

Time Analysis

The running time of this algorithm is O(n).