

COT 5405 Midterm 2 Solutions

April 1, 2009

1 Problem 4 Graded by Hale

GEOMETRIC DATA STRUCTURES AND ALGORITHMS [30 = 7+8+7+8 Points]

You may assume that no three points are colinear in the following problem.

(a) Given a convex polygon P with k points as a doubly-linked list (in clockwise and counterclockwise order) and a query q , describe an algorithm which finds whether q is inside P in $O(k)$ time.

If the polygon is convex as in our case, then one can choose one of the directions (clockwise or counterclockwise) and consider the polygon edges as a path from the first vertex. Notice that, a point is on the interior of this polygon if it is always on the same side of all the line segments making up the path. In other words, we should go through every edge in the same direction and execute an orientation test with the query point. If the relative orientation is always the same, then that means the point q is inside the polygon, otherwise it is outside. Orientation test will take $O(1)$ time for each edge of the convex polygon, making the total running time $O(k)$.

The orientation test can be summarized simply as follows: Given a line segment between $P_0 = (x_0, y_0)$ and $P_1 = (x_1, y_1)$, another point $q = (x, y)$ has the following relationship to the line segment. Compute $(y - y_0) * (x_1 - x_0) - (x - x_0) * (y_1 - y_0)$; if it is less than 0 then q is to the right of the line segment, if greater than 0 it is to the left, if equal to 0 then it lies on the line segment.

Hence, the algorithm will simply check whether the query point is always right of the edges or left of the edges. Also, if it is on the edge you should handle that case as neutral and ignore, since the other edges will reveal if the point is outside or not.

(b) Describe a more efficient structure that could be used in addition to doubly linked list maintaining a convex polygon such that the query in part (a) can be answered in $O(\log k)$ time.

Given a query point q , let us choose the counterclockwise order for the vertices from the given doubly-linked list. Since the convex polygon P is given in counterclockwise, the lines from the starting point s to the points in the array are sorted by angle (from the x-axis). We can now do a binary search, using orientation tests to decide if the query point q is to the left of next segment and right of the current segment. The binary search ends when: $orientationTest(s, a, q) > 0$ and, $orientationTest(s, a.next, q) < 0$.

Then, if q is inside the polygon, it is inside the triangle $(s, a, a.next)$. The node a , after the binary search must have: $orient(s, a, q) > 0$ and $orient(a.next, s, q) > 0$. So q is inside the polygon if $orientationTest(a, a.next, q) > 0$, and q is outside the polygon if $orientationTest(a, a.next, q) < 0$, and q is on the polygon boundary if $orientationTest(a, a.next, q) = 0$.

Before the binary search we can check two additional special cases. (i) If $(q == s)$, all orientation tests will be zero. (ii) If $orientationTest(s, s.next, q) < 0$, then q is outside the polygon. (iii) If $orientationTest(s.prev, s, q) < 0$, then q is outside the polygon

Hence, time for checking special cases $O(1)$, for binary search $O(\log n)$ and for checking last triangle edge $O(1)$ making total time $O(\log n)$. So, in addition to doubly-linked list we can use a binary search tree on vertices as the second data structure.

(c) Describe how to perform an insert operation on this data structure, i.e., INSERT takes a convex polygon P and a point p outside of P and maintains the convex hull P' of p and the points of P . The running time of INSERT should be $O(k)$, where k is the number of points on P .

When inserting a new point, two cases arise: (i) either the existing convex hull can be extended to include the new point or (ii) the new point requires the existing convex hull to be modified. Informally, we can distinguish the two cases by noticing that in case (i) the new point causes a right turn in the hulls boundary, whereas in case (ii) the new point causes a left turn. This can be determined mathematically by finding the angle between the right most line segment of envelope and the line segment of the right most envelope point and the new point. If the angle is less than 180 degrees then we have a case (i), otherwise we have a case (ii).

We have already mentioned how to find the closest point to the new point q in part (a) which takes $O(n)$ with doubly-linked list.

- (i) This case does not happen for this question, because in the problem q is already given as an outside point.
- (ii) In this case, the new point cannot be safely added to the existing convex hull without violating its convexity, therefore, the existing convex hull must be modified to accommodate the new point. This can be done by the following procedure. We know the closest points to the new point q . So, perform a convexity check on the predecessor to the new point. If the convexity check fails, delete the point from the convex hull which means updating the doubly-linked list which is only a matter of pointer changes. Now the new point has a new predecessor. Repeat the convexity check and deletion on the predecessor of the new point until the convexity check is satisfied, at which point we will have a valid upper convex hull again. This is basically the three-pennies algorithm you have seen in the class.

Analysis: Let there be k points in the problem. Convexity checks take a constant amount of algebra and therefore take $O(1)$. Simply updating a linked list is $O(1)$ and may occur a maximum of k times, which implies an $O(k)$ runtime. In particular, to find the runtime contribution of case (ii), the total number of deletions that may be made from the convex hull is less than k . Therefore, the cost of case (ii) is $O(k)$. The total cost of the insert operation will be $O(k)$.

(d) Consider an algorithm that uses this data structure for computing the convex hull of a set $S = \{s_1, s_2, \dots, s_n\}$ of n points incrementally. At the first step, it computes the convex hull of $\{s_1, s_2, s_3\}$. Then, the algorithm iteratively checks whether s_{i+1} is inside the convex hull C_i of the first i points and computes the convex hull C_{i+1} of the first $i + 1$ points using the INSERT operation (if necessary). A straightforward analysis of this algorithm would lead to an $O(n^2)$ time bound as each INSERT operation might cost linear-time in the worst-case. Give an amortized analysis showing that the running time of this incremental convex hull algorithm is $O(n \log n)$.

The basic observation is every point can be deleted from the convex hull once. So, once a point is deleted from the convex hull it will not be checked again. This will bound the number of points deleted throughout the algorithm by n . Since you need to check whether a point is inside or outside in $O(\log n)$ time from part (b), it will make the overall running time $O(n \log n + n) = O(n \log n)$. You can also think in terms of an accounting method to charge the vertices beforehand for deletion. When a point is inserted you can charge extra for the cost when it is deleted. That way, you charge a linear amount of money in total; \$1 for insertion and one more for possible deletion.

Note: Solutions given are not the only possible solutions, there can be many different solutions.