Assignment 4

Shashank Singh
sss1@andrew.cmu.edu
1
April 2, 2012

Minkowski set

(a) For each vertex of P and Q, compute the vector going from that vertex to the vertex immediately clockwise. Merge the two lists into a single list L of vectors, preseving the order of the vectors' slopes (the vectors were initially sorted by slope, because the vertices of P and Q were sorted and P and Q are convex). Add the sum p of the first two vertices in P and Q to a new set S. For each vector in L (proceeding in order of increasing slope), add to S the point produced by adding that vector to the last point added to S. Then, S will contain the sorted vertices of the Minkowski sum of P and Q. ■

(b) Since computing the vector between two points takes constant time, we can compute the lists of vectors in $O(m+n)$ time by simply iterating through each list of vertices. Merging two sorted lists is also a linear ($O(m+n)$) time operation. Finally, adding each point to S is a constant time operation performed $m+n$ times, so that the total runtime of the algorithm is $O(m+n)$. ■

(c) Let $a, b \in M$. Suppose c is a point on the line segment with a and b as endpoints, so that, for some $\theta \in [0,1]$, $c = a + \theta(b-a)$. Since $a, b \in M$, $\exists p_1, p_2 \in P$, $q_1, q_2 \in Q$, such that $a = p_1 + q_1$, $b = p_2 + q_2$. Then,

$$c = p_1 + q_1 + \theta(p_2 + q_2 - (p_1 + q_1)) = p_1 + \theta(p_2 - p_1) + q_1 + \theta(q_2 - q_1).$$

Thus, since P and Q are convex, so that $p_1 + \theta(p_2 - p_1) \in P$ and $q_1 + \theta(q_2 - q_1) \in Q$, $c \in P + Q = M$, and therefore M is convex. ■

Collinearity

(a) Iterate through the $n(n-1)(n-2)$ 3-tuples of distinct points. For each triple, compute whether the three points are colinear (this can be done in constant time by checking whether the slope of the line segment between the first two points is the same as the slope of the line segment between the second and third points). If any 3-tuple is found to be colinear, terminate and return that three points are colinear. Otherwise, return that no three points are colinear.

In the worst case, this requires performing $n(n-1)(n-2) \in O(n^3)$ constant times computations and requires space only for these computations, so that it requires $O(n^3)$ time and $O(1)$ space. ■

(b) For each point p, do the following:

1. Sort the other $n-1$ points radially about p in an array (this takes $O(n \log n)$ time).
2. Initialize 2 pointers f and s to point at the first point in the sorted array (this takes constant time).
3. Define θ to be the clockwise angle between the vector from p to the point poined to by s, and the vector from p to the point pointed to by f (computing θ takes $O(1)$ time).
4. Increment f, iterating through the array until $\theta \geq \pi$. If $\theta = \pi$, then three points are colinear. Otherwise proceed to increment s, iterating through the array until $\theta \leq \pi$.
5. Repeat step 4. until s points to the last point in the array. If $\theta = \pi$, that three points are colinear. Otherwise, no three points are colinear (this takes $O(n)$ time, since it is bounded by the time required to iterate the pointers through the array).

For each point, this process takes $O(n \log n)$ time, (since sorting the points radially is the most expensive part of the process). Therefore, the entire algorithm takes $O(n^2 \log n)$ time. Furthermore, the largest auxilliary data structure is the sorted list of vertices, which is of size $O(n)$. ■

(c) Compute the slopes of the n^2 possible line segments. Hash each line segment, based on its slope, into an array. If there are any collisions, check whether any of the endpoints of the colliding line segments are the same. If they are, then there are there are three collinear points. If all the line segments are hashed without any collisions, then there are no collinear points.

Computing and hashing each line segment takes constant time. In the worst case, this needs to be done n^2 times, so that the algorithm takes $O(n^2)$ time. The n^2 line segments each take constant space, so that the algorithm uses $O(n^2)$ auxilliary space.

Visibility

(a) Sort the endpoints of the line segments radially around p (in $O(n \log n)$ time). Initialize an empty priority queue of line segments ordered by the distance of their clockwise-most vertex. Find the first line segment under this ordering (in linear time), and add it to the priority queue. Iterate through the endpoints of the lines clockwise from this point. For each endpoint, add the line segment of which that point is an endpoint to the priority queue. If the line is already in the priority queue, remove it (this can be done in $O(\log n)$ time by hashing the location of each line segment in the priority queue). At each vertex, record the first element of the priority queue, as well as the angle of the point being considered in a list L. Iterating through all of the endpoints takes $O(n \log n)$ time ($2n$ add and $2n$ remove operations in the priority queue, each of which take worst case $O(\log n)$ time). Initialize an empty set S of line-segments. Iterate through L; for each (line-segment, angle) pair, add to S the line segment from the previous point to the point on the line-segment at the recorded angle (if they are part of the same line segment). Then, S is the list of visible line segments desired. L is of length at most $2n$, this takes $O(n \log n) + O(n) = O(n \log n)$ time, as desired. ■

(b) I suspect this is impossible, since there are, in the worst case, $\frac{n(n-1)}{2} \in \Theta(n^2)$ possible intersections for which any algorithm would have to account. ■

Connecting Dots, aka. Geometric Quick Sort

(a) Consider having red dots at $(0,0)$ and $(1,1)$, and blue dots at $(0,1)$ and $(1,0)$. Then, since the only way to create a spanning tree of each color is to place an edge between the two red dots and an edge between the two blue dots, which would cause line segments in the trees to intersect, no spanning tree with the desired proberties exist. ■

(b) Iterate through the vertices from top to bottom and connect each (except the first) vertex to the previous vertex. Then, connect the vertices of the triangle to each other or to the tree of interior points, as based on the colors of the vertices and the interior points. ■

(c) Suppose we have a nice triangle such that the interior of that nice triangle includes dots of both colors. Since the triangle is nice, it has either two red vertices and one blue vertex, or it has two blue vertices and one red vertex. In the first case, let p be some blue point in the interior of the triangle. Consider the triangular partitions created by connecting p to the vertices of the nice triangle. The triangle with both of the original nice triangle's red vertices as vertices has two red vertices and one blue vertex, so that it is also nice triangle. The other two triangles have two blue vertices (p and the blue vertex of the original triangle) and one red vertex, so that they are also nice triangles. Therefore, in this case, we can partition the original nice triangle into three nice triangles by connecting a point in the interior to the three vertices of the original nice triangle. In the latter case, let p be some red point in the interior of the triangle. An argument similar to that in the first case shows that connecting p to the vertices of the triangle partitions the nice triangle into three nice triangles. Since the interior of the triangle contains dots of both colors, this can always be done. ■

(d) Suppose we have a triangle with n interior points. Using the median-of-medians algorithm we can, in $O(n)$ time, find the p_1 and p_2, respectively the $(n/8)^{th}$ highest and lowest points (i.e., the points p_1 and p_2 such that, $n/8$ of the points have y-coordinates at most that of p_1 and $n/8$ of points have y-coordinates at least that of p_2). Similarly, we can, in $O(n)$ time, find p_3 and p_4, respectively the $(n/8)^{th}$ left-most and right-most points (i.e., the points that would be p_1 and p_2 if we rotated our coordinate system clockwise by $\frac{\pi}{2}$). Let S be the closed square with sides parallel to the x- and y-axes whose sides go through p_1, p_2, p_3, p_4. By choice of p_1, p_2, p_3, p_4, this square must contain at least at $n/2$ of the points; in particular, there is at least one point p in this square. Suppose we partition the triangle into three smaller triangles by connecting each of the vertices to p. Then, since there are at least $n/8$ points to left of, to the right of, above, and below p, none of these three triangles can contain more than $n - n/8 = 7n/8$ of the points. ■

(e) Consider the following recursive algorithm given a set of red and blue points whose convex hull in a nice triangle.

1. If the interior points of the triangle are all of the same color, use the algorithm from part (b) to return an appropriate tree and terminate.
2. Otherwise, let c be the color which is in the minority among the vertices of the nice triangle which is the convex hull of the set of points. Using the algorithm from part (d), partition the triangle into three smaller nice triangles, each of which has at most $7(n+m)/8$ points. Recursively perform the same algorithm on these three triangles.
3. Now that we have the desired trees on the points in each of the three smaller triangles, we can connect the vertices of each small triangle to the vertices of the other small triangles that are of the same color, thereby constructing a tree with the desired properties on the set of points in the original triangle.

The worst case run time of this algorithm occurs in the case that some triangle in the partition has exactly $7n/8$ points, so that the minimum possible number of points are "removed" in the recursive step. The runtime of the algorithm in this case is given by the recurrence

$$T(1) = O(1); T(k) = 2T(k/16) + T(7k/8) + cn,$$

in the case $k = n + m$, for some constant $c > 0$. Using the tree method, it can be seen that the work done in each level of recursion is at most ck, and that there are at most $\log_{8/7} k$ levels of recursion, giving the runtime $ck \log_{8/7} k = c(n+m) \log_{8/7}(n+m) = O((n+m)\log(n+m))$. ■