Brigham Michaelis
February 26, 2018
A00364835

Computational Geometry Assn.3

1. Let P be a set of n points in the plane. We say that point (x, y) dominates point (x′, y′) if $x \geq x'$ and $y \geq y'$. A point in P that is dominated by no other points of P is said to be maximal. Note that P may contain multiple maximal points. See Fig. 1 for an example. The problem is to compute all all maximal points of P . For simplicity of discussion, we make a general position assumption that no two points of P have the same x-coordinate or y-coordinate.

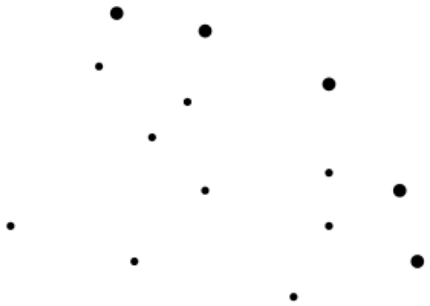

Figure 1: The five larger points are maximal points.

(a) Design an O(n log n) time divide-and-conquer algorithm for the problem. (15 points)

Algorithm Description

Solution for part a:
We begin the solution for this problem by discussing what the answer may look like. If every point had an accompanying boolean value, and we called the boolean value “has_been_dominated”, and if this boolean value was correctly set for each point then the solution could be easily presented in linear time by looping through the set of points P and simply reporting all the points where has_been_dominated is equal to false. These would be all the maximal points in the set P. Now that we have shown how our answer can be obtained by setting this boolean value for every point, we present our divide-and-conquer algorithm to set this value.

Here we show the psuedo code for the algorithm.

```
find_Maximal_points(S)
{
      if |S| == 1
            return;
      if |S| == 2
            // Check if one point dominates another, if it does then set the
            // appropriate bool and then return. Given two points this check can be
            // done in constant time just check the slope that is made by these two
            // points. If it is a positive slope then the the point with the lower
            // y-value has been dominated. If it is a negative slope then nothing
            // happens. On a side note asymptotes are not possible due to our
            // assumption that no two points of P have the same x-coordinates or y-
            // coordinates. Also as we return we sort the two points by their x-
            // coordinate.
            Return;
      // Now we divide the set S into S1 and S2 and we recursively call our
      // algorithm on these two sets.

      find_Maximal_points(S1);
      find_Maximal_points(S2);

      // Now that we have our two distinct sets S1 and S2(We assume that they are
      // solved individually and that they are also sorted by x and now we just
      // have to join the two solution sets together.) to perform the conquer step
      // we first merge the two already sorted sets into one set which is sorted
      // by x-coordinate in linear time. Then after the two sets are merged we
      // perform one more linear scan starting with the point with the highest x-
      // value. We could actually save some time here by only considering the
      // points that have not yet been dominated but to be thorough and show worst
      // case we traverse all the points.

      // We first set some variable called temp_maximal equal to the point with
      // the greatest x-value. Then starting with the point with the greatest x-
      // value and proceeding through the set S  in descending x-order we check
      // every following point against the temp_maximal point that was set
      // previously. If the y-coordinate of the checked point is greater than
      // temp_maximal's y-coordinate then that point becomes our new temp_maximal
      // and we continue as before. However if the y-coordinate of the checked
      // point is less than temp_maximal's y-coordinate then that point has been
      // dominated and we  mark its bool as such and continue this process until
      // we have traversed the entire set or we encounter a point that becomes our
      // new temp_maximal.
      temp_maximal = point in S with largest-x value.
      For (i = largest-x-point to smallest-x-point)
      {
            if(S[i].y-coordinate > temp_maximal.y-coordinate)
            {
                  temp_maximal = S[i];
            }
            else
            {
                  S[i].has_been_dominated = true;
            }
      }
```

```
      // Performing the above merge and linear scan not only allows us to return a
      // set where the solution has been found but it also allows us to return the
      // set to the calling function.
      return;
}
```

The algorithm given above operates in O(n log n) time and when finished the solution can be found by looping through every point and only reporting the points who's boolean value has_been_dominated is equal to false.

Psuedo Code
Given above
Correctness
This algorithm works because we made a modification to the closest pair algorithm that was given in class and because the changes we made only added two O(n) operations in each subset then the overall runtime of the algorithm becomes O(n log n)
Runtime
The runtime is O(n log n).

(b) Suppose all points of P have already been sorted from left to right by their x-coordinates. Design an O(n) time algorithm for the problem.

We have already presented a solution that uses this idea in part a, but to recap:

```
// We first set some variable called temp_maximal equal to the point with
// the greatest x-value. Then starting with the point with the greatest x-
// value and proceeding through the set S in descending x we check every following
// point in descending x-order,
// against the temp_maximal point that was set previously. If y-coordinate
// of the checked point is greater than temp_maximal's y-coordinate then
// that point becomes our new temp_maximal and we continue as before.
// However if the y-coordinate of the checked point is less than
// temp_maximal's y-coordinate then that point has been dominated and we
// mark its bool as such and continue this process until we have traversed
// the entire set or we encounter a point that becomes our new temp_maximal.
      temp_maximal = point in S with largest-x value.
      For (i = largest-x-point to smallest-x-point)
      {
            if(S[i].y-coordinate > temp_maximal.y-coordinate)
            {
                  temp_maximal = S[i];
            }
            else
            {
                  S[i].has_been_dominated = true;
            }
      }
```

Using the above algorithm the solution can be found in linear time. This is of course true under the condition that the set has already been sorted by the x-coordinate.

2. (20 points) We have studied the Euclidean closest pair problem in class, where the distance of two points in the plane are measured by their Euclidean distance. In this exercise, we consider the rectilinear closest pair problem. Let P be a set of n points in the plane. For any two points p 1 = (x 1 , y 1) and p 2 = (x 2 , y 2), their rectilinear distance is defined to be |x 1 − x 2 | + |y 1 − y 2 |. The rectilinear closest pair problem is to find a closest pair of points of P with respect to the rectilinear distance.

Modify the O(n log n)-time algorithm for the Euclidean closest pair problem we covered in class to solve the rectilinear closest pair problem in O(n log n) time.

Remark: Euclidean distance is also called L 2 -distance, and rectilinear distance is also called L 1 -distance or Manhattan distance.

Algorithm Description

This problem can be solved by using the same algorithm that was given in class for the closest-pair with the following changes:

1. Instead of computing the Euclidean distance we are going to compute the rectilinear distance. This is still just a number that can be computed in constant time.

However we need to make one more modification.

The modification is when we are looping through every 8 points and checking for a closest pair by rectilinear distance we instead need to loop through every 10 points instead of the 8, this is because in a box that is δ x 2δ there could be up to 10 points that are contained within such a box.

Here we present the modified closest-pair algorithm for rectilinear distance.

```
closest-pair(S)
{
      // Here we introduce the base cases
      if(|S| == 1)
      {
            return infinity;
      }
      if(|S| == 2)
      {
            return the rectilinear distance of the two points in S and sort the two
            points by their y-coordinate;
      }
      // Now in linear time we are going to split the set S into S1 and S2 by the
      // median x-coordinate xm. This operation will take linear time using
      // selection.
      // Now we are going to recursively call closest-pair on the two sets S1 and
      // S2. Remember these two sets are sorted by their y-coordinate.
      δ1 = closest-pair(S1);
      δ2 = closest-pair(S2);
      δ  = min(δ1 , δ2);
```

```
	// From here we are going to assume that we have the solutions for the sets
	// S1 and S2. Now we just have to conquer or merge the two sets together. We
	// begin this idea of conquering with a discussion of what these two sets
	// may look like. First we know that we have the solution for each set, and
	// we have the smallest δ from each set respectively. Now we are going to
	// compute the smallest δ between the two sets such that: δ = min(δ1 , δ2),
	// and much like we did in class we need to consider all the points that are
	// within one δ of xm on either side. Here is the idea. In linear time we
	// are going to isolate all the points that are within one δ of xm and merge
	// them into a single set.
	A1;  // All the points of S1 that are within one δ of xm.
	A2;  // All the points of S2 that are within one δ of xm.
	// We can form the two sets (A) in linear time because because the two sets
	// S1, and S2, are already sorted. Now we merge the two sets A1 and A2 into
	// one list A.
	A = mergeSort(A1, A2);
	// This operation can be done in linear time because A1 and A2 are already
	//sorted.

	// Now we have set A, and if we could just discover the smallest rectilinear
	// distance between two points in A we would be done. Remember that A is
	// sorted by y coordinate, this helps us in solving the problem.

	for(i = 1 to |A|)
	{
		for(j = i+1 to i+9)
		{
			if(rectilinear_distance_between_Pi and_Pj < δ )
			{
				δ = |Pi - Pj|;
			}
		}
	}
	// This works because the maximum number of points that can fit in a
	// δ x 2δ area is 10. Consider how this algorithm works for Euclidean
	// distance. We merge-sort A1 and A2 into a single set A, loop through the set
	// and consider, for every point the following 8 points in the set. In this
	// example we are simply incrementing by that number by two because of how
	// points can be grouped together when rectilinear distance is the desired
	// value.

	// Now that we have found the smallest δ from conquering the two sets S1 and
	// S2 together, we simply merge the two sets together into a single sorted
	// list and return δ. This operation takes linear time.
}
```

Psuedo Code

Shown above.

Correctness

This algorithm works because we are only slightly modifying the code provided in class, and all of our modifications do not alter the runtime.

Runtime

The runtime is O(n log n) this is just a modification of the algorithm given in class.

3. (20 points) In this exercise, we consider the following problem we left in class.
Suppose we have a planar subdivision P represented by the doubly-connected edge list (DCEL). Let v be a vertex of P . Let m be the number of edges of P that are incident to v. By using the DCEL data structure, give an O(m)-time algorithm to report all edges of P that are incident to v.

Algorithm Description
This algorithm is fun and easy. We have a vertex “v”. By definition we are given an incident edge “e” to this vertex, so first we are going to do a check to see if the vertex p is the origin of the given incident edge.

```
if(origin(e) == v)
{
      starting_edge = e;
}
else
{
      starting_edge = twin(e);
}

original_edge = starting_edge;

// Now we take one step forward.

report(starting_edge);
starting_edge = twin(starting_edge);
report(starting_edge);

while(next(starting_edge) != original_edge)
{
      starting_edge = next(starting_edge);
      report(starting_edge);
      starting_edge = twin(starting_edge);
      report(starting_edge);
}
```

And with that we are done. We are basically going in an out-in-out-in direction and traversing all the edges of v as we move around v.

Psuedo Code
Psuedo-Code is shown above.
Correctness
This algorithm runs in O(m)-time because we are traversing nothing except the incident edges to v and we are stopping when we are done making the loop.
Runtime
O(m)-time is achieved because we are only considering the incident edges to v and reporting them as we traverse those edges.

4. (20 points) Let P be a set of n points in the plane. We make a general position assumption that no three points of P lie on the same line. Let T be a triangulation of P . Let nt be the number of triangles of T . Let n e be the number of edges of T . Let h be the number of edges of the convex hull of P . See Fig. 2 for an example.

Prove that ne = 3n − h − 3 and nt = 2n − h − 2.

Hint: If we consider T as a planar subdivision, then there are n vertices, ne edges, and nt + 1 faces (each triangle is a face and the outer space is also a face). According to Euler's formula, we have n + (nt + 1) = ne + 2.

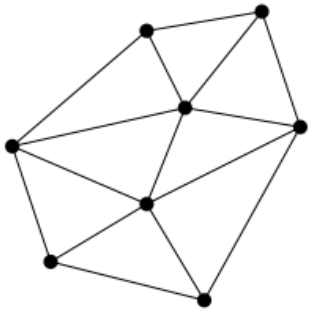

Figure 2: Illustrating a triangulation of $n = 8$ points. We have $h = 6$, $n_e = 15$, and $n_t = 8$.

Algorithm Description
There is a bit that needs to be done here. First solve the two for h and then we arrive at eulers. So we have shown that those two equations make up eulers. Now we just have to prove that those are accurate for h, we can do this by indcuction. And it is done by removeing 1 edge from the CH and showing how the h is updated, then we remove 2..3..n..n+1 basically show that these equations hold for stuff.
Psuedo Code
Not Required
Correctness
Not Required
Runtime
Not Required