Project Ideas – SMD156 Computational Geometry

Håkan Jonsson

October 2, 2006

1 General instruction

Each of the suggested projects below asks for an implementation of a program that solves a problem with geometric flavour using geometric algorithms and data structures. For each problem there are explicit upper bounds on the time and space a solution might consume. In many cases it is possible to find solutions with better upper bounds.

Each program should have a graphical user interface (GUI) by which actions can be controlled and results are displayed. In particular, and in addition to the possibility to enter input manually (by clicking, dragging, typing etc), the GUI should support the automatic generation of fairly large sets of input, like sets of points, line segments, convex polygons etc.

There are no restrictions on the programming environment and the programming language the project is implemented in as long as all geometric computations are programmed explicitely; it is not allowed to call geometric routies in libraries that happen to be present in the chosen environment/language.

Maybe needless to say, cheating by copying geometry code from someone else (the Internet, a book or paper, a fellow student etc) is not allowed. This also goes for help; don't do someone else's project for them and don't have someone else do your project. Don't cheat.

As a first step, you should try to come up with a solution on your own. You typically spend a day or so thinkning about the problem. Then you discuss the problem with friends and eventually, if you like, with me. I will give you enough hints so that you end up with a fairly good solution.

1.1 To finnish up the project

There are three things that must be completed to pass the project.

1. Have the source code graded by me. You pack everything together in a single file and mail it to me or, if the file is more than 2 megabytes in size, you instead mail me an URL or unix path on Sigma where I can find the file.

   To pass the grading I should be able to read and understand your code. Give the "big picture" and then the details. If you write it so that all your fellow classmates would understand it when browsing through the code once and without your help, I will also understand it. I like to stress that this is an important criteria.

   It should be clear from the program structure, the names on variables and methods etc, and comments

   - WHAT is done,
   - HOW it is done, and
   - WHY it works and does not consume more time and space than allowed.

   I will focus on the geometric parts. I am not particularly interested in the GUI part (it should, however, still be in the file so everything compiles but I will not pay much attention to it during my grading).

   If I can't understand your program when browsning through it you will be asked to explain it to me face to face. If this turns out to be a long, boring (yes), and complicated explaination, you will be asked to re-write the program and re-submit it. If this still does not improve the program you fail and will also fail the entire course.

2. Give a short presentation ($\approx$ 15 min) of your project for the class with

   (a) a description of the problem you have solved, usually easy if you prepare a slide with an example of a typical problem instance,

   (b) a (high-level) description of the geometric algorithms and data structures you have used in your solution, easy if you prepare a bullet-list with the main steps, and

   (c) a justification that your solution keeps within the required upper bounds, where you relate to the bullets above and explain.

3. Demonstrate your program in front of the class. This is done in connection to your presentation.

2 Projects

2.1 Point location in triangulations

Implement a data structure and operations for point location in triangulations (Fig. 1).

Upper bounds: $O(n)$ space for the point location structure and queries should take $O(\log n)$ time, where n is the number of points in the triangulation. Building the structure: $O(n^2)$ time.
More: The only requirement on the triangulation part is that it should be practical so it is possible to run a demo in front of the class. Keep this part simple.

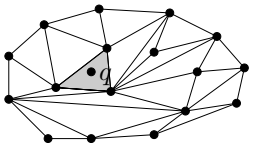

Figure 1: A triangulation of a point set. The query point q lies in the shaded triangle.

2.2 Contour of a set of rectangles

Given a set of rectangles, their contour is the set of boundaries that bounds their union (Fig. 2).

Implement an algorithm that computes[1] a contour.

Upper bounds: $O(n^2)$ time, where n is the number of rectangles.
More: —

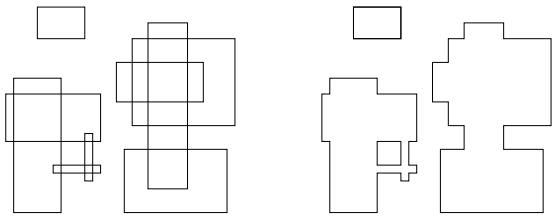

Figure 2: a) Input rectangles. b) Their contour.

2.3 Smallest bounding box of a simple polygon

Implement an algorithm that computes the smallest perimeter bounding box of a simple polygon (Fig. 2).

Upper bounds: $O(n \log n)$ time and $O(n)$ space, where n is the number of vertices in the polygon.
More: —

[1]The program should not just visualize the contour. It should also store it in some internal data structure.

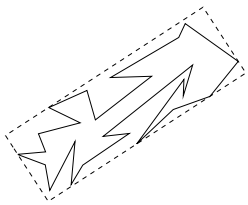

Figure 3: A smallest bounding box of a simple polygon.

2.4 Facility location

Write a program that, given a set of points in the plane and a number r, computes where a disc with radius r should be placed in order to maximize the number of input points covered by the disc (Fig. 4)[2].

Upper bounds: $O(n^2 \log n)$ time and $O(n)$ space, where n is the number of points.
More: —

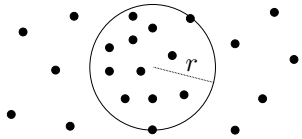

Figure 4: A placement of a disc width radius r that contains a maximum number of points.

2.5 Clustering

Implement a program that splits a set S of n points in the plane into two sets such that the distance of the convex hulls of the two sets is maximized (Fig. 5).

Upper bounds: $O(n^3)$ time.
More: —

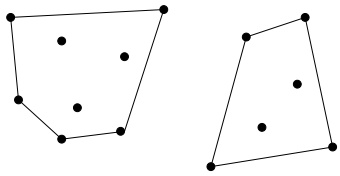

Figure 5: The shortest path between p and q among obstacles (polygons).

[2]If points are customers and r is the longest distance a customer is willing to go to buy an item, the midpoint of the disc is where the store should be placed to maximize the number of potential customers.

2.6 Shortest paths among obstacles

Write a program that, given a set of disjoint polygons, a start point p, and an end point q, computes the shortest path between p and q that does not enter into the interior of any polygon (Fig. 6).

Upper bounds: $O(n^2 \log n)$ time, where n is the total number of edges in the obstacles.
More: See also Chapter 15 in the course book [1].

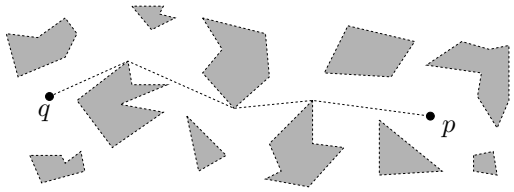

Figure 6: The shortest path between p and q among obstacles (polygons).

2.7 Convex hull of line intersections

There are $O(n^2)$ intersections among n lines. Write a program that, given just the lines, computes the convex hull of these intersections in $O(n \log n)$ time(!) (Fig. 7).

Upper bounds: $O(n \log n)$ time, where n is the number if lines.
More: —

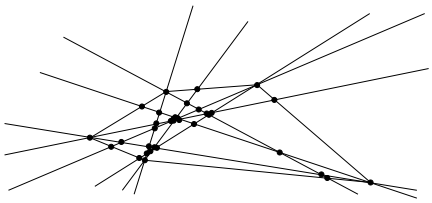

Figure 7: The convex hull of intersections among lines.

2.8 Closest pair

Given a set of points, compute a pair of points whose distance is smaller than or equal to all other distances between points (Fig. 8).

Upper bounds: $O(n \log n)$ time, where n is the number of points.
More: —

2.9 Shortest paths in weighted regions

Implement a program that computes a *fastest* path among triangles in a triangulation with different "thickness" (Fig. 9).

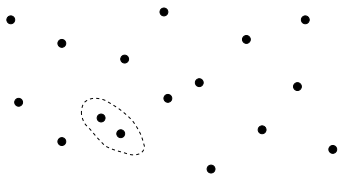

Figure 8: A closest pair.

Let $\mathcal{T}$ be a triangulation in which each triangle have been assigned a positive weight that represents how "thick" the triangle is. Let S be a polygonal path that connects two points p and q and lies entirely within $\mathcal{T}$. Let $\Delta_1, \Delta_2, \ldots, \Delta_k$ denote the triangles that S intersects from p and q, let w_i be a weight associated with Δ_i, and d_i the length of $\Delta_i \cap S$, the part of S that lies in Δ_i.

Then, given $\mathcal{T}$ the program should compute a path S such that

$$\sum_i w_i d_i$$

is minimized.

Upper bounds: —
More: —

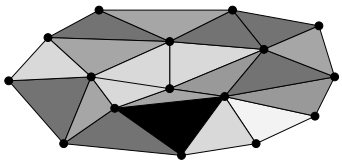

Figure 9: Weighted regions.

2.10 Collision Detection

Write a program that, given two simple and disjoint polygons **P** and **Q**, where **P** lies strictly to the left of **Q**, computes the first points on the polygons that will collide if **P** is translated horizontally and in the positive x-direction, or determines that they do not collide (Fig. 10).

Upper bounds: $O((n+m)\log(n+m))$ time, where $n = |\mathbf{P}|$ and $m = |\mathbf{Q}|$.
More: —

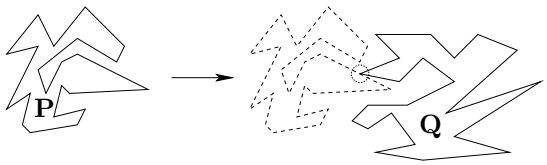

Figure 10: The circle shows where **P** collides with **Q** when translated horizontally to the right.

2.11 Translating Rectangles for Maximal Overlap

Implement an algorithm that translates a set $\mathcal{B}$ of rectangles (all rectangles are translated in the x- and y-directions exactly the same) onto another set of rectangles $\mathcal{R}$ such that their common intersection $\mathcal{B} \cap \mathcal{R}$ is maximized (Fig. 11).

Upper bounds: $O(n^4)$ time, where n is the total number of rectangles in $\mathcal{B}$ and $\mathcal{R}$.
More: —

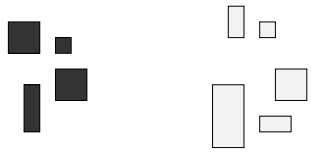

Figure 11: How should the set of dark rectangle be translated onto the grey ones to maximize the overlap?.

2.12 Shortest path in a Simple Polygon

Implement an algorithm that computes the shortest path between two given points in a given simple polygon (Fig. 12).

Upper bounds: $O(n)$ time once the polygon has been triangulated, where n is the size of the polygon.
More: There is no bound on the triangulation, as long as it works in practice (at the demonstration and for polygons with, say, at least 30-40 vertices); keep this part simple.

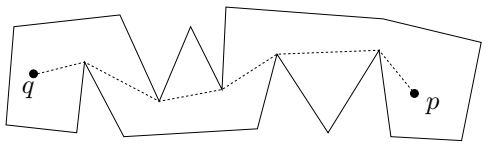

Figure 12: The shortest path between p and q.

2.13 Largets Empty Square

Implement a program that, given a set of n points in the plane, computes a data structure to answer queries about largest empty squares efficiently: Given a query point q, find the largest empty square centered at q (Fig. 13).

Upper bounds: Your query algorithm should run in $O(\log^2 n)$ time and $O(n \log n)$ space.
More: —

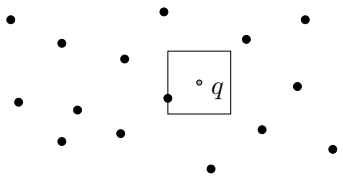

Figure 13: The largets empty square centered at the query point q.

2.14 Monotone Polygons

Implement a program that checks if there is a direction in which a simple polygon is monotone and, in that case, reports such a direction (Fig. 14).

Upper bounds: $O(n)$ time, where n is the size of the polygon.
More: —

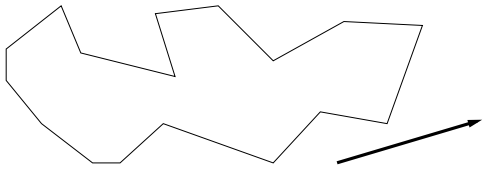

Figure 14: A simple polygon and a direction in which it is monotone.

2.15 Triangulation using Divide-and-Conquer

Implement a program that computes a triangulation of a set of points in the plane by Divide-and-Conquer, that is the algorithm recursively divides the set in two halves, triangulates the halves, and stich them together into a single triangulation by adding triangles between them. (Fig. 15).

Upper bounds: $O(n \log n)$ time, where n is the number of points.
More: —

2.16 Kernels

The kernal of a simple polygon is the subset of points that can see[3] the whole polygon. Implement a program that take a simple polygon as input and compute its (possibly empty) kernel. (Fig. 16).

[3] A point can see another if the line of sight does not go outside the polygon.

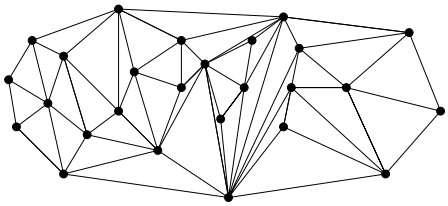

Figure 15: A triangulation of a point set.

Upper bounds: $O(n)$ time and space, where n is the size of the polygon.
More: —

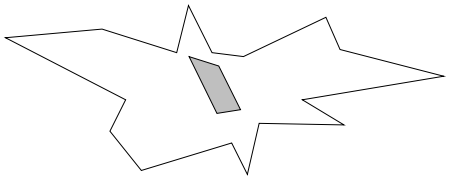

Figure 16: The kernel of a simple polygon.

2.17 Convex layers

The convex layers of a set of points is the set of convex hulls one gets by computing a convex hull, removing it, computing a convex hull of the remaining points, removing it, computing yet one convex hull of the remaining points, etc until there are no points left. Implement a program that computes the convex layers of a set of points (Fig. 17).

Upper bounds: $O(n^2)$ time and $O(n)$ space, where n is the number of points.
More: —

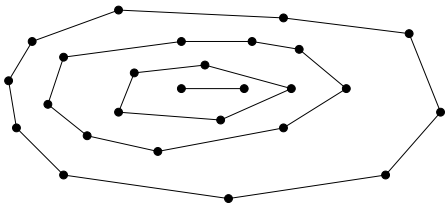

Figure 17: Convex layers.

Acknowledgement

Some of the project ideas were inspired by similar material produced by Bettina Speckmann and David Mount.

References

[1] Mark de Berg, Otfried Schwarzkopf, Marc van Kreveld, Mark Overmars, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2000. 2nd rev., ISBN: 3-540-65620-0.