

COMPUTATIONAL GEOMETRY

Optional Projects

By **November 20, 2007**, you must notify me of which project you are doing and submit a brief (1-page) description of the project plan. Some projects can be done in small teams (maximum 3 people), but if more than one person is involved, the project plan must include a detailed breakdown of who will do what part of the project.

Projects are due **December 6, 2007**, but don't wait until the last minute to do it!! Submission should be by email. In addition, it is usually useful for you to demo the project in person; please email me to schedule the demo. If you do the project in Java, please also create an applet and webpage, so that particularly educational projects can be easily featured in future classes.

Note that some projects may lead to publishable research, depending on how successful the investigation is.

All software should be written in standard C, C++, or Java. (Check with me before using another language.) It should be easily compilable under Unix (e.g., a PC running Linux) or Windows. Please document your code! Also, provide a README, and bundle all files in one directory. You may find that C or Java is the most convenient: you are allowed to use C code fragments provided in the O'Rourke textbook (and available on the web, both in C and in Java – follow pointers to O'Rourke's home page, from our course home page). You may be asked to give a short demonstration of your code to me. Try to handle degenerate cases and to make your implementation as robust as you can.

You will find it most convenient to have a graphical display to output and “animate” your algorithm. You may also want to be able to mouse in data.

A rough breakdown of points assigned to the project: Correctness (including degenerate cases): 50 points; robustness: 25 points; interface (ideally graphical) and ease of input/output: 10 points; documentation and readability of code: 15 points.

(1). (See Problem 2.10, Chapter 2, BKOS) Let S be a planar subdivision with n vertices, stored as a DCEL. Let P be a set of m points. Implement a plane-sweep algorithm to locate each point of P in a face of S . In order to be able to verify correctness (and to debug), you should use a graphical interface and animate the algorithm. Be creative! (It may be useful to have a mouse-based editor to input S and build a DCEL representation of it.)

(2). Implement a method to compute a minimum-area triangulation of a set of points in 3D: Each point $p_i = (x_i, y_i, z_i)$ sits above the xy -plane by distance $z_i > 0$ and our goal is to triangulate the projections of the points onto the xy -plane so that the corresponding set of triangles in 3D has the smallest possible surface area. This problem arises in surface reconstruction from point cloud data. I have some ideas for how to do this with heuristic methods – see me, and I can give details. You are encouraged to come up with your own ideas too. I think local search methods may be good. There is a possibility that a good project leads to a publishable paper.

I am also very interested in theoretical results: Can you *approximate* optimal in polynomial time? Can you show that the exact solution is NP-hard?

(3). Implement a method to place a “small” number of guards in a simple polygon, P . The goal is to design a reasonable heuristic to do this, making sure that the entire polygon is seen. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

The implementer of this algorithm may work in partnership with the implementer of the witness point algorithm below.

(4). Implement a method to place a “large” number of independent witness points in a simple polygon. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

(5). Given a simple polygon P , compute its minimum weight triangulation, using dynamic programming. The input is assumed to be a text file whose first line gives the x and y coordinates of p (separated by a space), and whose

remaining lines give a clockwise listing of the (x, y) coordinates (floating point) of the vertices (two floating point numbers, separated by a space). It will be useful to have the option of mouse-clicking (and saving) input too. You may assume that the polygon is definitely simple.

(6). Implement the incremental linear programming algorithm of Section 4.3 for the case of two dimensions ($d = 2$). The input is assumed to be a text file of floating numbers, with the first line being $c_1 c_2$ (separated by a space), then the next n lines being $a_{i,1} a_{i,2} b_i$ (with spaces separating the numbers). The output should print the optimal objective function value (with the option to return “NOT FEASIBLE” or “UNBOUNDED”), along with the point (x, y) that achieves the optimal value. You should animate the algorithm, showing each constraint line as it is inserted, along with the currently optimal vertex, etc.

There is no need to randomize the order of the input; you may assume that the constraints arrive in random order already.

(7). Implement the deterministic incremental convex hull algorithm (sketched in class) in 2D for points $\{(x_1, y_1), \dots, (x_n, y_n)\}$, inserting each point in time $O(\log n)$ (amortized) using appropriate data structures. The input is assumed to be either graphical (e.g., allowing the user to mouse-click enter points) or read from a simple text file, with each line consisting of two floating point numbers (x_i and y_i) separated by a space. You should animate the algorithm to show each step as each point is inserted. Try to make the code be as educational as possible in illustrating how the algorithm works.

(8). Implement the Bentley-Ottmann sweep to DETECT if a polygonal chain $C = (p_1, p_2, \dots, p_n)$ is *simple* (it is not simple if it properly crosses itself, at a point interior to two edges, or if it crosses itself at a vertex, in the degenerate case). Report a witness to the crossing if the chain is not simple.

The input is assumed to be either graphical (e.g., allowing the user to mouse-click enter points) or read from a simple text file, with each line consisting of two floating point numbers (the x - and y -coordinates of a vertex) separated by spaces. You should animate the algorithm to show each step during the sweep.

(9). Implement an algorithm that runs in $O(n)$ time to triangulate a y -monotone simple polygon having n vertices. The input is assumed to be a text file with a clockwise listing of the (x, y) coordinates (floating point) of the vertices; each line of the text file consists of two floating point numbers, separated by a space. (Again, it will be useful to have the option of mouse-clicking input too.) You may assume that the polygon is simple and that it is y -monotone (you need not check to confirm this in your code), but you do not know which vertex comes first (e.g., it need not be the topmost or the bottommost). Again, there may be degeneracies (duplicated points, collinear points, etc).

The output should be a list of pairs of points that define the diagonals of the triangulation. (You need not build a DCEL or other data structure for the triangulation.)

Conduct experiments to determine the in-practice efficiency of the algorithm for large inputs.

(10). Implement the point location data structure (DAG) of Chapter 6. The input consists of a list of segments $S = (s_1, s_2, \dots, s_n)$, which are to be inserted in the order given. Show graphically the trapezoidal diagram as each one is inserted. (Basic animation of the algorithm using simple graphics will greatly ease the debugging.)

(11). Implement the algorithm `VISIBLEVERTICES(p, S)` on page 312 of the text. You may assume that each simple polygon in the set S is given by a list of its vertices in order (ccw) around its boundary, that they are known to be pairwise disjoint (and not touching), and that p is necessarily disjoint from S . Draw the connections from p to each visible vertex.

(12). Implement a gift-wrapping (Jarvis march) algorithm to compute the convex hull of a set of convex polygons, $C = \{P_1, P_2, \dots, P_k\}$, in the plane. The input may be given by allowing the user to mouse-click polygons P_i (you may assume the user always enters them ccw, and they must be convex), or read from a file.

The output should be shown graphically, and a list of the vertices in ccw order about the hull should be written to a file. Try to animate the algorithm to make it educational. (You will need to be able to control the speed of animation.)

(13). Implement kd-trees for points in the plane. The input should be a set of points, $S = \{p_1, p_2, \dots, p_n\}$, given either by mouse clicks or by reading a text file, with each line consisting of floating point numbers x_i and then y_i , separated by a space. (The first line of the file should be the number, n , of points.) The program should then allow one to specify a rectangle, $[x, x'] \times [y, y']$, by text input from the user and/or mouse input from the user. The output, then, should be a listing of the points that are within the query box. (If you use graphical output, change the color of the points that lie within the query box.)

Perform the following experiment with your code. For various values of n (e.g., $n = 100, 200, 300, \dots, 1000$, or other interesting range of values), do the following. Generate a set S of n random points in the unit square (each x_i and y_i is uniformly distributed between 0 and 1, using the random number generator of the language in which

you wrote the code). Build the kd-tree for each S , and compute the average time, over, say, 20 samples, it takes to answer a query Q given by a box that is, say, 0.1-by-0.1. (A simple way to do this is to generate the query box as $[x, x + 0.1] \times [y, y + 0.1]$, where x and y are chosen uniformly at random from the interval $(0,0.9)$.) Compare the speed of the kd-tree method to the “brute force” method of simply testing all n points, one by one, to see which ones are in the query box. It would be nice to see a plot of the two times, as a function of n . For what value of n is it worth using the more sophisticated data structure of the kd-tree?

(14). This project is just like project (13), except that instead of using a kd-tree to do orthogonal range queries, use a 2d range tree. (You may do it without the use of fractional cascading, if you want.)

(15). Implement either the Lawson edge swap algorithm (LegalizeTriangulation) or the randomized incremental method for Delaunay triangulations of points in the plane.

(16). Implement Timothy Chan’s convex hull algorithm in 2D.

(17). A set S of n line segments in the plane is said to have a Type 1 degeneracy if all segments lie on a common line. They are said to have a Type 2 degeneracy if some subset of 3 or more endpoints are collinear. It is easy to see that a Type 1 degeneracy can be detected in $O(n)$ time. Using duality and building a line arrangement, you can see that a Type 2 degeneracy can be detected in $O(n^2)$ time. (Do you see how?) Implement a degeneracy-tester that takes as input a set of line segments in the plane and reports if there is a Type 1 or Type 2 degeneracy.