605.727 Course Project

Marc Johnson mjohn351@jh.edu

JHU Whiting School of Engineering — February 12, 2021

Introduction

Implement an algorithm for constructing two-dimensional Delaunay triangulations, either divide-and-conquer algorithms or incremental insertion, described by Leonidas J. Guibas and Jorge Stolfi. Feel free to skip Section 3, but read the rest of the paper. See also this list of errors in the Guibas and Stolfi paper, and Paul Heckbert, Very Brief Note on Point Location in Triangulations, December 1994. (The problem Paul points out can't happen in a Delaunay triangulation, but it's a warning in case you're ever tempted to use the Guibas and Stolfi walking-search subroutine in a non-Delaunay triangulation.)

Your implementations must use Guibas and Stolfi's quad-edge data structure (with appropriate simplifications, if you wish). You are responsible for reading and understanding the Guibas and Stolfi paper, except Section 3, which may be safely skipped. It's a long paper, so start early.

The purpose of this project is to burn an understanding of Delaunay triangulations and planar subdivision data structures into your brain.

Which of the options should you choose? Well, the divide-and-conquer algorithm is faster, and will satisfy you better if you're a speed junkie. The incremental algorithm is more flexible because it's incremental, and therefore can be used in an on-line manner by mesh generators and other algorithms that choose vertices based on the state of the triangulation. I think the difficulty of the two implementations is equal. (Guibas and Stolfi's divide-and-conquer pseudocode is substantially more complicated than their incremental pseudocode, but the extra stuff you must devise yourself is more complicated for the incremental algorithm than for the divide-and-conquer algorithm.)

Info: The quad-edge data structure is useful when constructing geodesic space-frames since it encapsulates nonorientable 2-manifolds. This is not crucial to understand, what is important is to know that the doubly-connected edges list (DCEL) is a variant of the quad-edge data structure. Each quad-edge record groups together four directed edges corresponding to a single edge in the subdivision. At each vertex, a record maintains how faces connect to each other around that vertex. This is key to understand from the texts like here.

1 Divide-and-Conquer Algorithm

You have different choices of algorithms from which to choose.

1.1 Guibas and Stolfi Pseudocode: Divide-and-Conquer

Implement the divide-and-conquer algorithm for which Guibas and Stolfi give pseudocode.

1.2 Dwyer Implementation

Implement a second version of the divide-and-conquer algorithm whose recursion alternates between using horizontal cuts (at even depths of the recursion tree) and vertical cuts (at odd depths) to divide the points into two subsets. In other words, you bisect the entire set of points by x-coordinate, then bisect each half-set by y-coordinate, then bisect each quarter-set by x-coordinate, and so on, alternating between directions. (This alternation is motivated by a paper by Rex Dwyer.) The original algorithm of Lee and Schachter (discussed by Guibas and Stolfi) uses vertical cuts only.

To bisect the sets quickly, I suggest using the standard $O(n)$-time quick select median-finding/partitioning algorithm. It's not a good idea to fully sort the points at every level of the recursion, because if you do, you'll have a $\Theta(n \log^2 n)$ Delaunay triangulation algorithm, which defeats the speed advantage of alternating cuts.

You should find that you can use the same code to merge hulls regardless of whether the cuts are vertical or horizontal, but before/after you merge, you'll need to readjust the positions of the convex hull "handles" called ldi, rdi, ldo, and rdo in the Guibas-Stolfi paper.

1.3 Guibas and Stolfi Pseudocode: Incremental Insertion Algorithm

Implement the incremental insertion algorithm for which Guibas and Stolfi give pseudocode. Use their suboptimal "walking" method for point location.

1.4 Fast Point Location Incremental

Implement fast point location, based on either conflict lists or a history DAG $\mathcal{D}$ (whichever you prefer). Since you are using an edge-based data structure, rather than a triangle-based data structure, you will need to modify the point location data structure slightly.

Each quad-edge has four Data fields. Two of these are used to reference the vertices of the quad-edge. For point location based on conflict lists, the other two Data fields may be used to reference the lists of uninserted vertices associated with the triangles on each side of the edge. For point location based on $\mathcal{D}$, the other two Data fields may be used to reference the leaves of $\mathcal{D}$ associated with the triangles on each side of the edge.

For conflict lists, each uninserted vertex should maintain a reference to an oriented edge of the triangle that contains the uninserted vertex. For a history DAG, each leaf of the DAG should maintain a reference to an oriented edge of the triangle that the leaf represents.

2 Implementation

If you write in any language other than Python or Java, you are required to give me very complete instructions on how to compile and run your code. The instructions do not have to be in a Readme, but you must include the instructions for compiling and then running the program.

An example of a program written in Python is listed below.

hello.py
```
#! /usr/bin/python

import sys
sys.stdout.write("Hello␣World!\n")
```

For the previous code, example instructions would tell you to go to the terminal and go to the directory containing hello.py.

Command Line
```
$ chmod +x hello.py
$ ./hello.py

Hello World!
```

2.1 Interface

Your code should take as input as .csv file that contains the (x, y) coordinates of the points that are to be triangulated. You are free to use whatever packages you desire for InCircle tests and CounterClockwise (CCW) tests, but please use floating-point operations to be as accurate as possible.

Notice: Borrowed code: You are welcome to use publicly available libraries or implementations of the following, so long as none of them was produced by any of your classmates: sorting, selection (aka median finding or partitioning), trees, other fundamental non-geometric data structures, command-line switch processing, file reading/writing, and geometric primitives like the InCircle and CCW predicates. You must write the quad-edge implementation and geometric algorithms all by yourself..

2.2 Submission

Submit your assignment in the submission portal of Module 10 as a tar or zip file. You should also submit a report (on paper) that includes the following:

- Instructions for compiling and running your code. If you use C/C++ and Unix, relatively rudimentary instructions will probably do. Be sure to document how to specify the input file, and how to choose between different algorithms.
- A table containing timings for each of your algorithms or point location methods on random points sets (top of this page) of 10,000, 100,000, and 1,000,000 points. (If the “walking” method of point location takes longer than thirty minutes on large point sets, just note that you couldn’t wait for it to terminate.) Time only the randomized versions of the incremental insertion algorithm. Try to exclude all file I/O from your timings if possible. (If using a timer within your program isn’t possible, the Unix time command will do, although file I/O time will be included.) Why do you think the fastest algorithm is fastest?
- Try to answer one of the questions below depending on your type of algorithm you used to triangulate.
- If you borrowed any code, please give full credit.

Question 1

For the divide-and-conquer algorithm only: Can you create a point set for which the vertical cuts algorithm is notably faster than the alternating cuts algorithm? What do you think accounts for the discrepancy?

Question 2

For the incremental algorithm only: create an orderly point set, like a square grid whose vertices are given in a structured order. Does failing to randomize the order of the vertices significantly change the running time of the incremental insertion algorithm with the fast point location method?