6.850: Project report
Implementation of algorithms for construction of Voronoi diagram

Timur Garipov

May 13, 2020

Algorithms

For this project, I implemented the following algorithms:

- Beach line algorithm discussed in the class.
- Incremental algorithm.

Implementation details

Voronoi diagram representation

In both algorithms the Voronoi diagram is stored in a doubly linked list structure. The doubly linked lists stores faces, edges, and vertices of the diagram. Face is represented as a list of edges, in each face the edges are stored in the counter-clockwise traversal order. Edges are oriented, there are two versions of each edge going in opposite direction. Each edge stores:

- a pointer to its dual version;
- pointer to the face this edge belongs to, and a pointer to the face the dual edge belongs to;
- pointers to the next and to the previous edges;
- pointers to the two vertices, representing the endpoints of the edge.

The code of both algorithms is build with the assumption that coordinates of the input points are bounded and the points lie in the bounding square $[-100, 100] \times [-100, 100]$ (for a set of input points this assumption can be satisfied by shifting and scaling input points). Also each of the algorithms adds three additional "fake" sites located in the vertices of a equilateral triangle which contains the bounding square in its interior. These sites are added in order to simplify handling of unbounded edges.

Beach line algorithm

I followed the description of the algorithm given in [1].

In my implementation, instead of storing arcs and breakpoints of the beach line in a binary search tree, I used binary search tree to store the breakpoints only. Additionally, each breakpoint stores pointers to the two arcs it is connecting, and each arc stores pointers to the left and right breakpoints of this arc. The links between the arcs and breakpoints can be easily updated when processing beach line events.

Initially, I attempted to use `std::set` in `C++` as a binary search tree for storing breakpoints. The idea was to use a custom comparator object, which compares breakpoints by their x-coordinates for a given position of the sweep line. However, I observed that insertion and deletion of breakpoints from the tree based on comparison of x-coordinates can break the integrity of std::set when breakpoints collide or if the coordinates of the breakpoints are computed with floating point errors.

To address this issue, I switched to a custom implementation of a binary search tree which can maintain the correct order of the breakpoints and does not rely on computation of the positions of the breakpoints. I used Cartesian tree (also known as Treap) — a randomized balanced binary search tree. I used the implementation with implicit keys. Instead of inserting/deleting breakpoints by their x-coordinate, I implemented two operations:

- Insertion of a new breakpoint b_{new} after a given breakpoint b_{prev} (all breakpoints which go after b_{prev} in beachline now will go after b_{new}).
- Deletion of a breakpoint by a pointer to its position in the tree.

I noticed, that these operations are sufficient to maintatin the correct order of the beachline, since a relative order of a two breakpoints in the beachline never changes (unless breakpoints collide, but after collision the breakpoints are deleted). Also, when a new breakpoint appears, we know which breakpoint its should be inserted after. So, my implementation of the binary search tree does not need to compute x-coordinates of breakpoints in order to insert/delete breakpoints in/from beachline. The only stage of the algorithm which relies on x-coordinates of the breakpoints is search query which is performed when processing a site event. The search query finds a leftmost breakpoint in the beachline which has strictly bigger x-coordinate than the current site.

Incremental algorithm

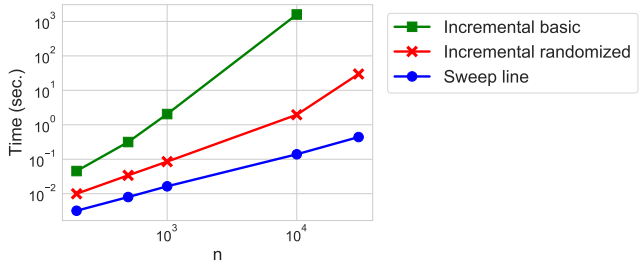
The incremental algorithm constructs the diagram by inserting sites one-by-one and updating the diagram after each insertion. There are two key steps of the algorithm:

- **Point location**. For a new site p_{new} the algorithm finds a cell v of the current version of the Voronoi diagram V_{old} which contains p_{new}.
- **Structural changes**. The algorithm makes structural changes to V_{old} to add the new site p_{new}. The algorithms splits affected faces of V_{old} starting from v, removes or cuts affected edges, and inserts a new face corresponding to p_{new}.

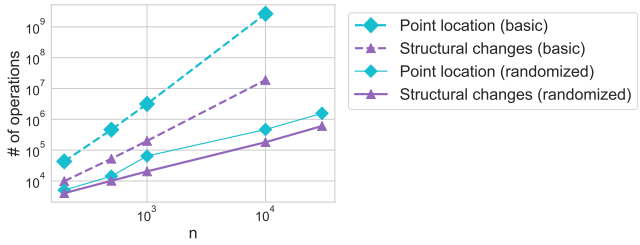
Both of the steps can be performed in linear time per one site with total complexity of algorithm for n sites being $\mathcal{O}(n^2)$. The algorithm can be accelerated by randomization over the order in which sites are inserted [2]. Guibas et al. [2] show that, with randomization, the expected total number of structural changes is linear. Guibas et al. [2] also describe a technique which allows to obtain $\mathcal{O}(n \log n)$ expected time complexity of all point location queries. The idea of the method is to store all versions of Delaunay triangulations on top of each other. When a new site arrives some of the old triangles are replaced with the triangles generated by the new site. If we store the removed triangles and for each removed triangle store pointers to newly generated triangles which have non-empty intersection with the removed triangle, the obtained triangle tree structure can be used for point location. Guibas et al. [2] show that in expectation the total time for processing queries with the described trianglee tree structure is $\mathcal{O}(n \log n)$. I implemented the triangle tree for point location.

Evaluation

I evaluated the algorithms on two types of data. The first dataset is the worst case input for the incremental algorithm described in [2]. On this data incremental algorithm without randomization reaches quadratic complexity. Figure 1 shows dependence of execution time and number of point location operations / structural changes for inputs of different size n. We observe that basic version of the incremental algorithm is the slowest. Randomized incremental algorithm and sweep line algorithm demonstrate nearly linear performance. The sweep line algorithm is the fastest. On 1 (b) we observe that randomization decreases the number of point location operation and structural changes performed in incremental algorithm.



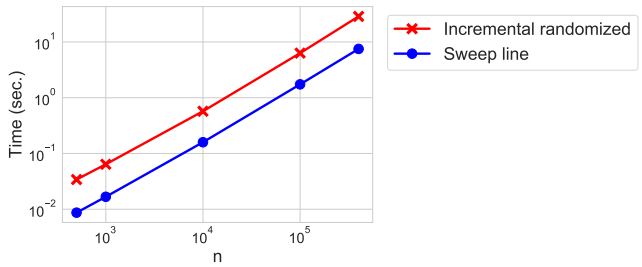
(a) Execution time as a function of n



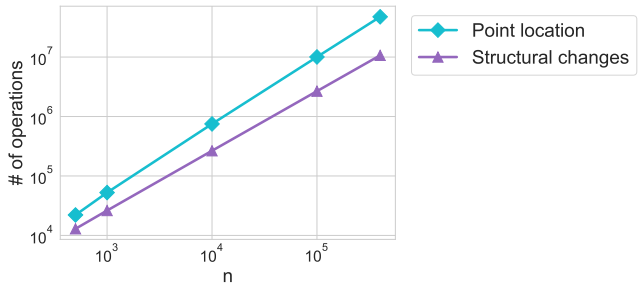
(b) Number of operations as a function of n

Figure 1: Evaluation of the algorithms on the worst case example for basic incremental algorithms.

The second type of data I used for evaluation is datasets with sites uniformly distributed in the square $[-100, 100] \times [-100, 100]$. Figure 2 shows results of the evaluation on the uniformly distributed points. Both sweep line and randomized incremental algorithm have nearly linear performance rate. Again, the incremental algorithm performs slower than the sweep line by a constant factor. We also observe that number of point location operations and structural changes in the incremental algorithm grow almost linearly with n.



(a) Execution time as a function of n



(b) Number of operations as a function of n

Figure 2: Evaluation of the algorithms on the examples with uniformly distributed sites

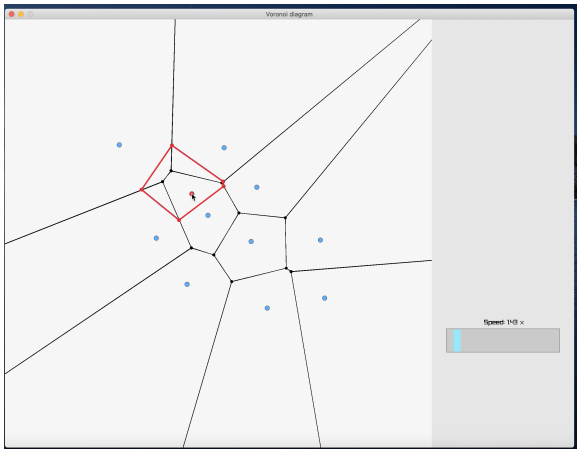
Code and Demonstration

For both algorithms I implemented visualization tools which show the workings of the algorithms. Deomnstration videos with visualizations can be accessed via the links below

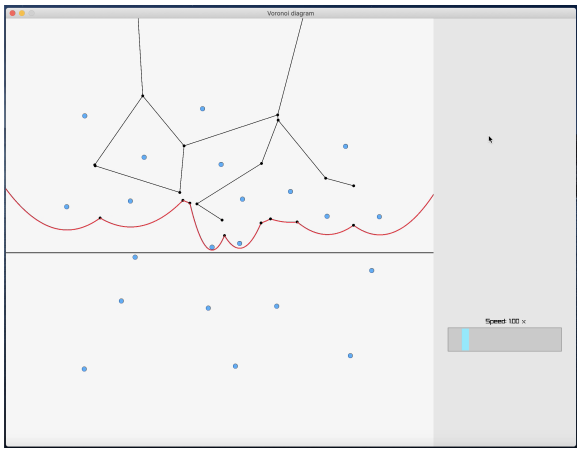
- **Beach line algorithm:** https://drive.google.com/file/d/1PifI-gkobYKBbXkYN-fmRGAGx1Vex5Hp/view?usp=sharing
- **Incremental algorithm:** https://drive.google.com/file/d/1kDGoCbE7ie-xImuJ5tNwz9kzWfjlKvWt/view?usp=sharing

For implementation of the graphical routines and user interface, I used `raylib`[1] open-source library.

The code for the project can be found at https://drive.google.com/file/d/1wSloSTOucMkxCuRY-y0ZhH14lrsTN1hQ/view?usp=sharing



(a) Incremental algorithm visualization tool



(b) Sweep line algorithm visualization tool

References

[1] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. Computational geometry. In *Computational geometry*, pages 1–17. Springer, 1997.

[2] Leonidas J Guibas, Donald E Knuth, and Micha Sharir. Randomized incremental construction of delaunay and voronoi diagrams. *Algorithmica*, 7(1-6):381–413, 1992.

[1] https://www.raylib.com