Final Report: 6160 Computational Geometry

Fortune's Algorithm Implementation in Python – Alex Liebig

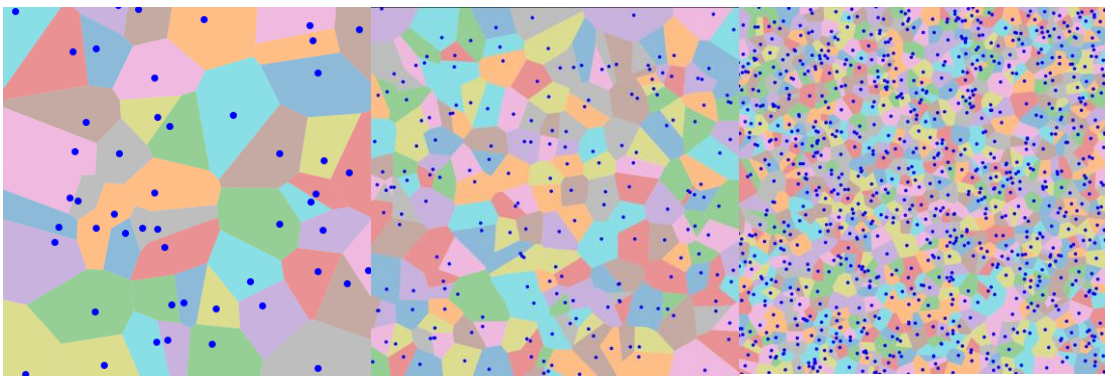
Summary: Overall, implementing Fortune's algorithm in Python was quite challenging. In the beginning, I thought it would be interesting to implement Fortune's algorithm as well as the Delaunay Triangulation, and this is what I originally wanted to do. However, implementing Fortune's algorithm alone proved to be quite a challenge so I decided to focus my efforts on that. This report is going to discuss what Fortune's algorithm is, highlight the challenges I faced, report my implementation, as well as the results of my implementation.

Fortune's Algorithm: Fortune's algorithm is a line-sweep algorithm that generates a Voronoi diagram from a set of points, also known as sites, by sweeping a line from top to bottom. During the sweep, the algorithm creates a beach line that determines the intersections between arcs on the beach line, where Voronoi edges and vertices occur. The algorithm distinguishes two types of events: site events and circle events. Site events occur when new arcs are added to the beach line, while circle events occur when arcs are removed, leading to the creation of Voronoi vertices. Once the algorithm has successfully processed all the events, a complete Voronoi diagram can be generated.

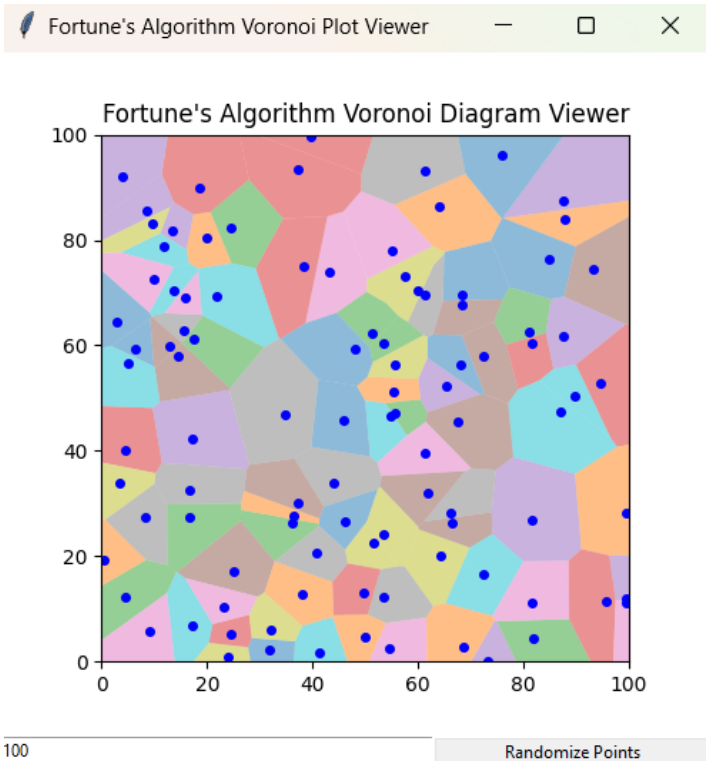
Challenges: Implementing the algorithm itself proved to be very difficult and challenging. To start off, it required remembering how to calculate a parabola based on a focus and directrix. Once this was understood, the beachline in the line-sweep could then be implemented. The single most challenging aspect of this algorithm was implementing the Circle events that occurred. It wasn't clear originally how this was implemented, so I originally considered checking if every arc's site on the beach line was a circle event at every iteration. This alone was an O(n^3) operation since I was comparing a combination of 3 points. Originally this did work, but it was very slow. For a while, the brute force implementation was the only way I knew how to detect circle events. However, I discovered that I only needed to check the 2 surrounding arcs and their neighbors at each site event, as well as the deletion of a circle event. This made my algorithm ultimately function properly.

Implementation: For my implementation, I only used numpy and built-in python libraries to calculate the Voronoi vertices. From the Voronoi vertices, I was able to create the Voronoi edges from that. I used Matplotlib to visualize the Voronoi diagram, and I used Tkinter in order to display the Voronoi diagram.

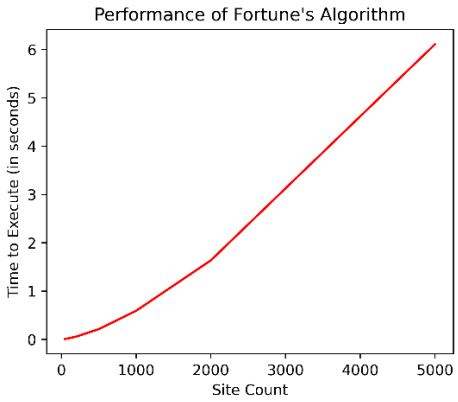
Results: Here is a summary of my results below:



Voronoi Diagram with 50 Sites (left) 200 sites (middle) and 1000 sites (right)



Fortunes Algorithm GUI (default sites = 100)



Performance of Fortune's Algorithm	
Site Count	**Time (in seconds)**
50	0.009495735168457031
100	0.023992300033569336
200	0.058550119400024414
500	0.21218085289001465
1000	0.5923178195953369
2000	1.6361949443817139
5000	6.109100341796875

Benchmarks from 50 sites to 5000 sites graph (left) table displaying raw results (right)

Conclusion: Implementing Fortune's algorithm was very difficult, but ultimately this taught me a lot about Voronoi diagrams. Using Matplotlib and Python proved to be slow in visualizing the Voronoi diagrams though. If I were to re-implement this project, I would use a faster language like C# or C++, and I also would have liked to create a more-involved GUI that would allow users to interactively change points. While it looks correct, I didn't properly handle the edges of the Voronoi diagram either. I introduced points outside of the grid that make it look correct. I also failed to optimally display the algorithm. I sorted each site's Voronoi vertices by angle so that it could properly create the sites (as convex polygons). In doing so, this adds to the time complexity of the overall algorithm.