Department of Computer Science

COS 451

Generation of Simple Polygons

Final Project in Computational Geometry

Jan. 14, 2020

Victor Hua (vhua) & Hanson Kang (jaehok)

Professor:
BERNARD CHAZELLE

1. General Approach

The objective of the project was to "design, analyze and implement an algorithm for constructing simple polygons" (per topics page). The only input argument was n, the number of vertices/edges. Furthermore, the polygon was to be as winding and irregular as possible.

In order to achieve the randomness and irregularity expected of the polygon, the algorithm would ideally create a polygon from a randomly generated set of vertices. The challenge was to connect the vertices in a way such that no self-intersections would occur and, later, to achieve this in as efficient a way as possible.

The general approach taken by this project is to first generate a convex hull from the original point set. This guarantees that (1) a valid polygon is constructed and (2) all the remaining points in the point set are within the polygon. Now, at each iterative step, a new point will be added to the polygon, until every point is added to the polygon as a vertex. The point is added by removing one edge, called the removing edge, and adding two new edges from the new point to either vertex of the removing edge (refer to Figure 1).

While there are multiple ways to connect all the vertices this way without having the polygon self-intersect, the easiest method to guarantee a valid simple polygon is to remove the edge that has the shortest point-to-line distance between itself and its closest interior point. Of course, there may be a case in which the edge with the shortest distance to a point cannot be removed (i.e. because it creates a self-intersection). In that case, consider the edge with the next shortest distance to an interior point and repeat until a valid removing edge is found. Using this method, there a polygon will always be formed from a random set of n points.

The bottleneck step in this approach is the finding of the removing edge. We first present a naive solution and then an improved solution that uses dynamic programming to reduce the priority calculations.

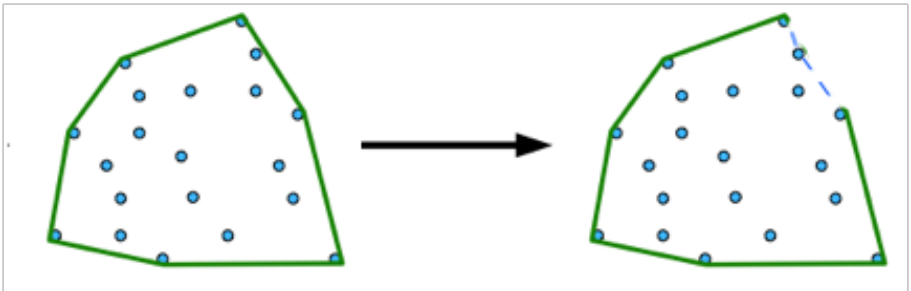


Figure 1

2. Naive Solution

The naive solution for finding the removing edge is, naturally, brute-force. We iterate over all the edges in the current polygon and for each edge, we find its edge to point distance from every interior point. For each pair encountered, we check its validity (i.e. whether the two new edges created would be self-intersection-free) if the distance is less than what we have encountered so far, to ensure it can be inserted into the existing polygon. The valid edge-point pair with the minimum distance is the removing edge-closest point pair that we are looking for.

Pseudocode: naive_simple_polygon_generation(n) {

1. points = generate_random(n), polygon = quickhull(points), interior = points - hull
2. while (interior not empty):
   - min_dist = infinity
   - for (e in polygon):
     - for (p in interior):
       - $e_1 = Edge(e.start, p)$
       - $e_2 = Edge(p, e.end)$
       - curr_dist = distance(e, p)
       - if (curr_dist < min_dist and polygon not intersecting(e_1, e_2)):
         (a) min_dist = curr_dist
         (b) removing_edge = e
         (c) nearest_point = p
   - polygon.insert(p), polygon.insert(e_1, e_2), polygon.remove(e)
3. return polygon

}

For each step, calculating the minimum distance between an edge and a point takes $O(n^2)$ time since we calculate the distances from $O(n)$ edges to $O(n)$ points. For each edge-point pair we encounter, we potentially perform a validity check that takes $O(n)$ time since the two new edges are being checked for intersection against the $O(n)$ edges in the existing polygon, and thus each step takes $O(n^3)$ time. Since there are $O(n)$ total interior points and one step is performed for each point, the algorithm overall takes $O(n^4)$ time. We will discuss an improved algorithm in the next section.

3. Improved Solution

The naive solution's time complexity of $O(n^4)$ is slow. One runtime for $n = 100$ randomly generated points is just 0.72 seconds, but at $n = 500$, it already takes 72.97 seconds. This makes it infeasible to generate random polygons with larger values of n. An improved solution for finding the removing edge must be found.

The naive solution recalculates the point-to-line distances of most edge-point pairs throughout its iteration. Because the distance between an edge and a point will not change while the algorithm is running, we can avoid this repetition by preprocessing the edge-point pairs and storing in sorted order by distance.

The improved solution uses a minimum priority queue that stores the edge-point pairs and is sorted by distance. We can preprocess the unconnected points such that the distances between each edge on the convex hull and all of the interior points can be mapped to edge-point pairs and pushed into the priority queue. After every addition of a vertex to the polygon, we push to the priority queue all the edge-point pairs between the remaining interior points and the two new edges.

Pseudocode: improved_simple_polygon_generation(n) {

1. minPQ = minPQ(), distances_dict = dictionary()
2. points = generate_random(n), polygon = quickhull(points), interior = points - hull
3. for (e in polygon):
   - for (p in interior):
     - curr_dist = distance(e, p)
     - minPQ.push(curr_dist)
     - distances_dict[curr_dist] = [e, p]
4. while (interior not empty):
   - curr_dist = minPQ.pop()
   - e, p = distances_dict[curr_dist]
   - $e_1 = Edge(e.start, p)$, $e_2 = Edge(p, e.end)$
   - if (polygon not intersecting(e_1, e_2):
     (a) polygon.insert(p), polygon.insert(e_1, e_2), polygon.remove(e)
     (b) for (p in interior):
       - d_1 = distance(e_1, p), d_2 = distance(e_2, p)
       - minPQ.push(d_1), minPQ.push(d_2)
       - distances_dict[d_1] = [e_1, p], distances_dict[d_2] = [e_2, p]
5. return polygon

}

The preprocessing takes $O(n^2)$ time since we calculate the distances from $O(n)$ edges to $O(n)$ points. The size of the minimum priority queue is also $O(n^2)$, and for each element in the queue we do a validity check which takes $O(n)$ time. Thus, the runtime of our improved algorithm is $O(n^3)$. We will compare the results of our two solutions in the next section.

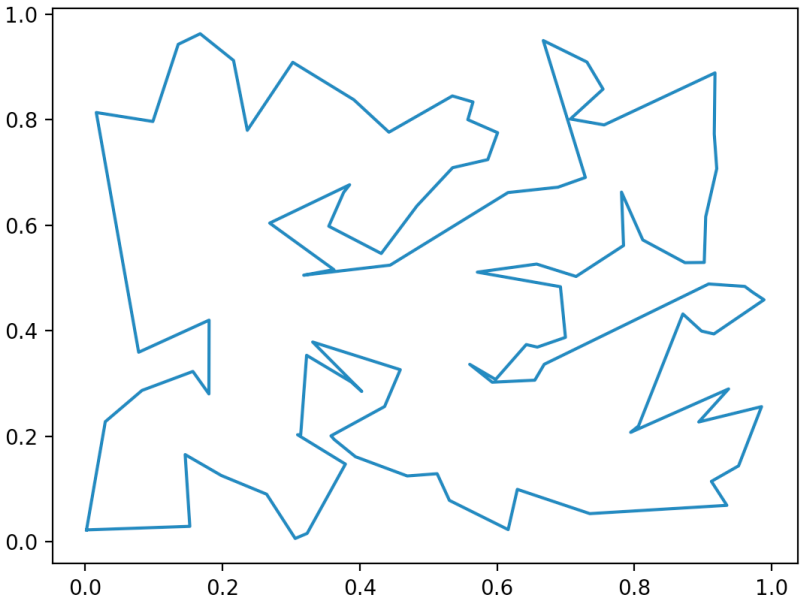
4. Results and discussion

Both algorithms are correct as they both generate simple polygons that are random and intersection-free. As the number of vertices increases, the polygons become increasingly more complex, meaning that for a random point within the boundary, it is hard to determine whether it lies on the interior or exterior of the polygon. The polygons are also irregular and winding.

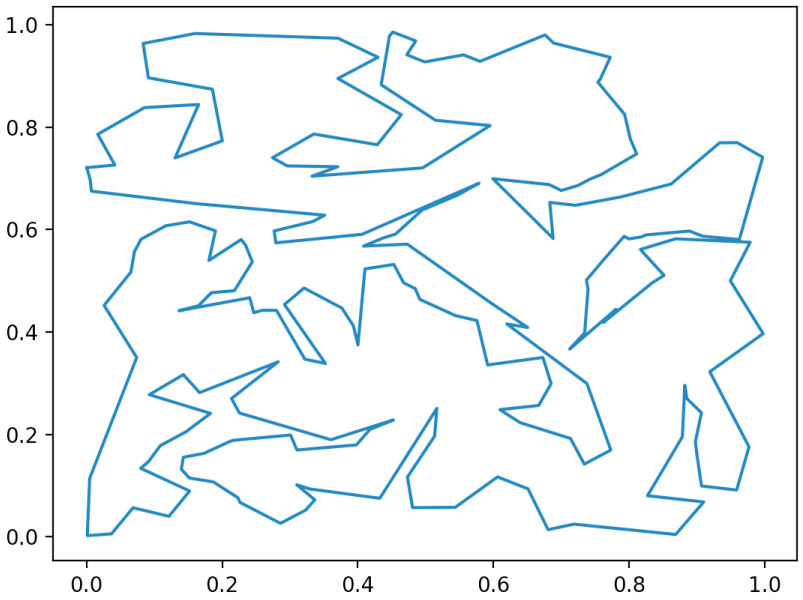
We found that our improved algorithm utilizing a minimum priority queue significantly reduced the time complexity. For $n = 100$ points, our naive algorithm was slow at 0.72 seconds while the improved algorithm finished in just 0.06 seconds (as shown below). The difference becomes even more clear at $n = 500$, in which the naive algorithm took 72.97 seconds and the improved algorithm took just 2.57 seconds. Running the naive solution on anything greater than $n = 1,000$ became infeasible, whereas the improved solution could produce random polygons for inputs as high as $n = 10,000$.

Following are some example printouts of the polygons generated by the improved algorithm, along with the execution time.

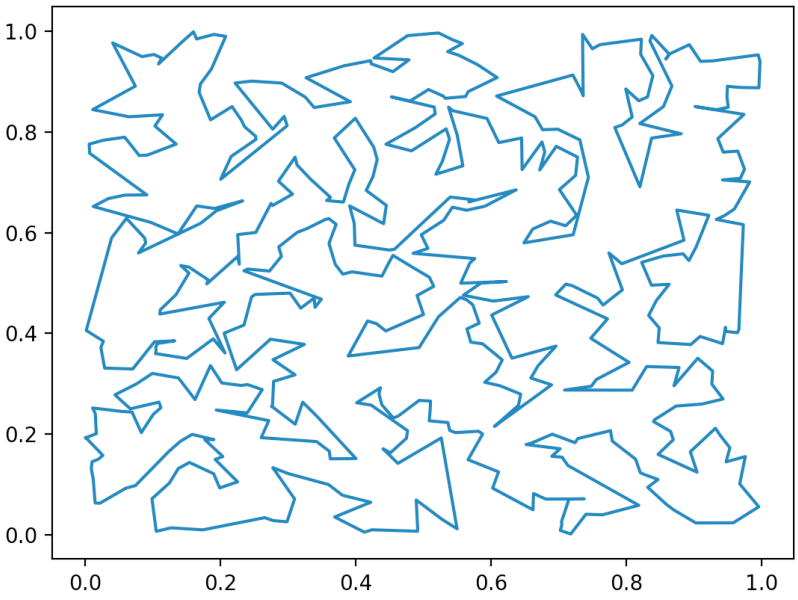
Example Polygons



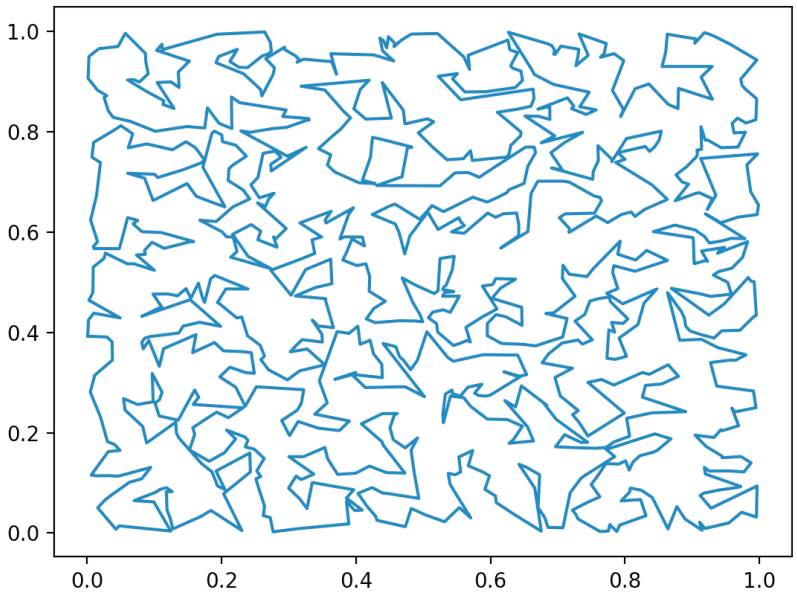
Number of vertices: 100
Execution time: 0.06363081932067871 seconds



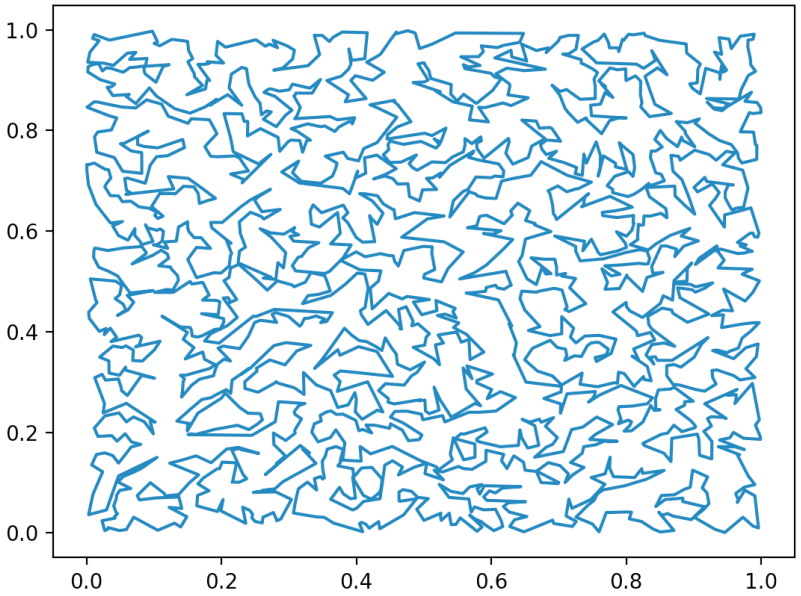
Number of vertices: 200
Execution time: 0.2813560962677002 seconds



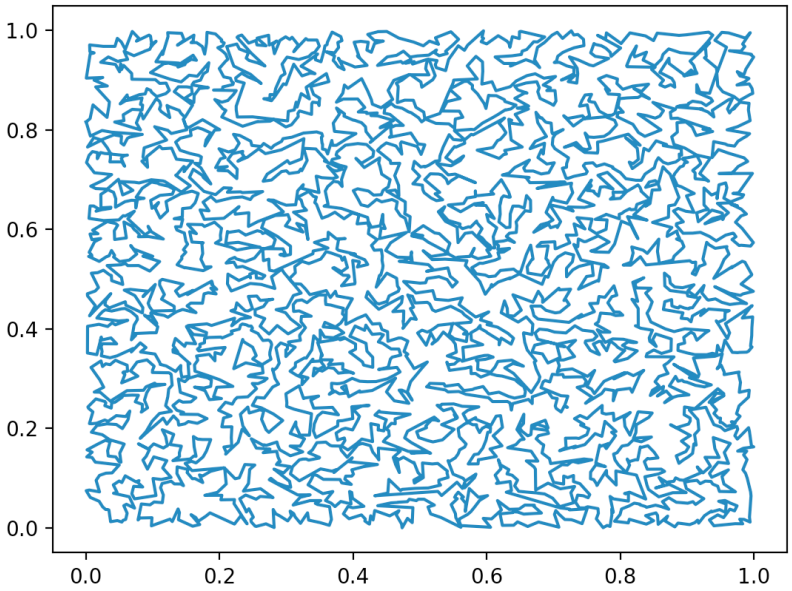
Number of vertices: 500
Execution time: 2.5730531215667725 seconds



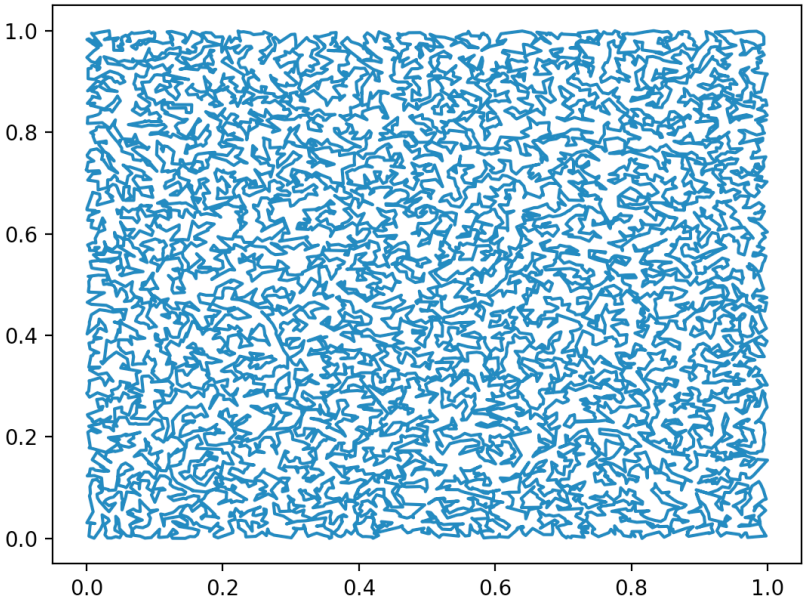
Number of vertices: 1,000
Execution time: 12.147171020507812 seconds



Number of vertices: 2,000
Execution time: 33.56451201438904 seconds



Number of vertices: 4,000
Execution time: 150.69127893447876 seconds



Number of vertices: 10,000
Execution time: 2761.4851989746094 seconds