Generating random simple polygons with N vertices

Nuno Neto
up201703898

June 7, 2021

Advanced Topics in Algorithms
Masters in Computer Science

1 Introduction

This report discusses the implementation, complexity and correctness of the algorithm that was implemented to generate random simple polygons with N vertices in a MxM square grid. To represent the generated polygon we use a doubly connected edge list that will store all vertices, edges and faces in a manner that allows us to efficiently access and alter the polygon.

2 Doubly connected edge list

2.1 Proof of correctness

In this assigment, it is necessary to maintain two basic properties, that assert the triangulation is well maintained by the DCEL:

- Each internal face represents 1 triangle from the triangulation of the polygon and as such has 3 edges.
- There is only 1 external face that contains the outer half-edges of the polygon.

The base case for these properties:

- When there are no half-edges, there are no external or internal faces, so the properties are valid.
- When there's an open polygon, the DCEL only has one face that contains all half-edges. Since the polygon is open all half-edges are outer edges, so the properties are valid as there are no internal faces and the outer face contains all of the edges.

The operations described in Section 2.3 maintain these properties when closing open polygons and when generating new open polygons as we will see further on.

2.2 Data representation

The data in the DCEL is represented by 3 types:

- Vertexes
- Half Edges
- Faces

2.2.1 Vertex

A vertex represents a vertex and stores the geometric point data and an half-edge that starts at that node.

2.2.2 Half Edge

A half edge is, as the name states, half of the data needed to represent a full edge between two vertexes. These two half edges are *twins* and one cannot exist without the other (all half edges must have a twin). The twin of a half edge will always have the opposite direction (its target is the origin of the twin and vice versa).

Each half edge stores a target vertex (the vertex the half edge is incident on), the incident face (the face the half edge belongs to), a reference to its twin and a reference to the previous and next half edges.

Half edges do not need to store their origin vertex as to get their origin we only need to go to their twin and get its target vertex.

2.2.3 Face

A face only has to store a reference to a half-edge that is part of it, as the edges will always form a circular linked list if the data represented is a closed polygon.

2.3 Operations

In order to edit the information that is stored in a DCEL we use the following functions:

- add-vertex - Add a vertex to the DCEL.
- add-edge - Add an edge to the DCEL (can also be referred to as split-face).
- join-face - Remove an edge from the DCEL and subsequently joins the faces that it separates.
- split-edge - Splits an edge into two by inserting a vertex half way through it.

In this assignment we only used the *add-vertex* and *add-edge* functions, so we will focus on those.

2.3.1 add-vertex

This function takes as an argument the geometrical coordinates of the new vertex v represented as the green vertex in Figure 1 and the half edge that is incident $incidentOnU$ on the vertex u, both represented as blue, that we want to connect our new vertex v to.

It will then create the half edges that connect v to u, create the vertex object for v and add them to the vertex and half-edge lists. This operation will never alter a face as it is not possible to create a closed polygon by just adding a vertex (to close a polygon we have to connect two already existing vertices) and has a complexity of $O(1)$.

The two created half-edges, $halfE1$ and $halfE2$ are represented as red and yellow, respectively, in the following images.

We now reach a forking point, where we have three possible situations.

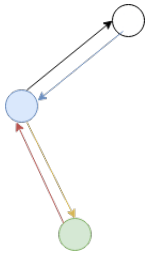

Figure 1: Adding a new vertex to a DCEL, situation 1

Situation 1 To handle the situation represented in Figure 1:

In this situation we only have one face for the entire polygon so it is known that the $incidentFace$ of the $incidentOnU$ half-edge has to be the same as the face of it's $twin$.

Here we connect $halfE1$ to the $twin(incidentOnU)$, by setting the $next(halfE1) \leftarrow twin(incidentOnU)$. This has a complexity of $O(1)$.

We then connect $halfE2$ to $incidentOnU$, by setting the $next(incidentOnU) \leftarrow halfE2$. This has a complexity of $O(1)$.

We then set the $face(halfE1)$ and $face(halfE2)$ as $incidentFace$. This has a complexity of $O(1)$.

Finally, we add the half-edges and the vertex to the edge and vertex list, respectively. This has a complexity of $O(1)$.

In total, this operation has a worst case complexity of $O(1)$.

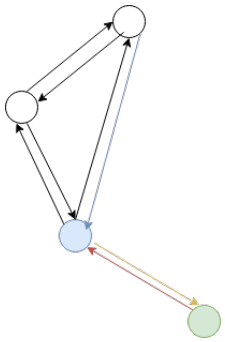

Figure 2: Adding a new vertex to a DCEL, situation 2

Situation 2 In this situation there is already a closed polygon so we have to maintain the properties.

We connect $halfE1$ to the $next(incidentOnU)$ by setting the $next(halfE1) \leftarrow next(incidentOnU)$. This has a complexity of $O(1)$.

We then connect $halfE2$ to $incidentOnU$, by setting the $next(incidentOnU) \leftarrow halfE2$. This has a complexity of $O(1)$.

We then set $face(halfE1)$ and $face(halfE2)$ as the $face(incidentOnU)$. This has a complexity of $O(1)$.

In total, this operation has a worst case complexity of $O(1)$.

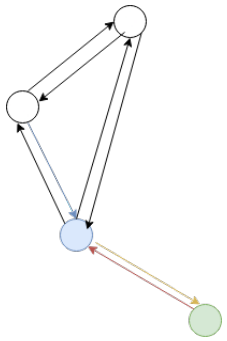

Figure 3: Adding a new vertex to a DCEL, situation 3

Situation 3 This situation is never possible in our use case, as we only choose outer half-edges, from the outer face when we want to add a new vertex (This property is assured by the way we select half-edges to add new vertexes to).

Since this situation is never possible from our use case, it is not accounted for.

Induction step on the add-vertex operation In this operation we do not create new faces (as we do not close any polygons) so by an induction step, we know that property 1 is maintained. Since this operation is not able to close polygons, we know that all the newly added edges are going to belong to the outer face. By an induction step on property 2, since we add these newly created half-edges to the outer face property 2 is maintained.

2.3.2 add-edge

This function creates an edge between two existing vertices that are not yet connected. It effectively splits the face that they're part of into two new faces.

The arguments that are accepted are: A half edge that is incident on the first vertex u we want to connect and a vertex v which is the vertex we want to connect u to.

It will then create the new half-edges and the new faces.

Creating the half edges and the new face objects takes $O(1)$ time.

Now, to actually connect the half edges to their correct targets and vice-versa. To do this, we have to take into account the current condition of the DCEL. In this assignment, since we are always creating new triangles there are only two possible situations:

Situation 1 In this situation, we are trying to add an edge that connects the vertex in blue to the vertex in green. The arguments provided were the half edge $incidentOnU$ (marked as blue) that is incident on the blue vertex u and the vertex that is marked green, v.

In this image, $halfE1$ is represented in red and $halfE2$ is represented in yellow. We first connect $halfE2$ to the $twin$ of the $incidentOnU$ edge by setting $previous(twin(incidentOnU)) \leftarrow$

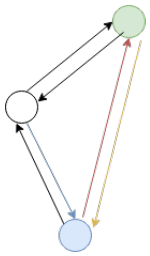

Figure 4: The first possible situation when adding an edge

$halfE2$ and consequently $next(halfE2) \leftarrow twin(incidentOnU)$. We then connect $halfE1$ by setting $previous(halfE1) \leftarrow incidentOnU$ and consequently $next(incidentOnU) \leftarrow halfE1$. This has a complexity of $O(1)$.

We then traverse the polygon, starting at $halfE2$ (So we will go in clock wise order) until we reach the vertex v and set all of those edges as part of the new face we are creating $face2$. After we reach v, we connect the edge that is incident on v to $halfE2$ as its previous. This has a complexity of $O(|H|/2)$.

In this situation, to finish connecting $halfE1$ we only have to connect it to the $twin$ of the edge that is incident on v. This has a complexity of $O(1)$.

After we have connected $halfE1$, we have to traverse the face that has now been created and set the $incidentFace$ of all edges that are a part of it as $face1$. This has a complexity of $O(1)$, as we are guaranteed by the first property that all internal faces have 3 half-edges.

We then delete the previous face from the face list, which has a complexity of $O(|F|)$.

We are now done creating the new edge. This operation has a worst case complexity of $O(|H| + |F|)$.

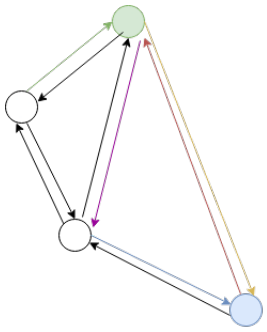

Figure 5: The second possible situation when adding an edge

Situation 2 Again, the arguments provided are the half edge $incidentOnU$ (marked as blue) that is incident on the blue vertex u and the green vertex v.

In this image, $halfE1$ is represented in red and $halfE2$ is represented in yellow. We first connect $halfE2$ to the $twin$ of the $incidentOnU$ edge by setting $previous(twin(incidentOnU)) \leftarrow halfE2$ and consequently $next(halfE2) \leftarrow twin(incidentOnU)$. We then connect $halfE1$ by setting $previous(halfE1) \leftarrow incidentOnU$ and consequently $next(incidentOnU) \leftarrow halfE1$. This has a complexity of $O(1)$.

Again, like in situation 1, we traverse the polygon starting at $halfE2$ until we reach the vertex v and set all of those edges as part of the new face we are creating, $face2$. After reaching v the edge that is incident on v (marked as green) is connected to $halfE2$ as its previous. This has a complexity of $O(|H|/2)$.

However, we cannot just connect $halfE1$ to the $twin(incidentOnV)$ as it's not part of the new internal face we are creating. So we now have to traverse the polygon in counter clock-wise order (through the inside of the polygon) until we reach the twin of the half edge that is part of the new face we are creating. This half edge is marked in the image as pink. When we reach pink half-edge's twin, we stop and we connect $halfE1$ to it. In the case represented in Figure 5 we only have to traverse a single inner face, however there are situations where we might have to traverse more than one inner face. To move from inner face to inner face, when we reach the final point of the inner face we are traversing (the next half-edge is where we started) we switch to it's *twin* and keep iterating. This has a complexity of $O(|H|)$ as we might have to traverse all the inner faces of the polygon.

We then traverse the face starting in the half-edge $halfE1$, and set the incident face of all the edges to the new face being created, $face1$. This has a complexity of $O(1)$ as we know that all internal faces have at most 3 half-edges which is garanteed by the first property.

Now we remove the previous external face from the face list, which has a complexity of $O(|F|)$.

We are now done creating the new edge. This operation has a worst case complexity of $O(|H| + |F|)$.

Induction step on the add-edge operation We don't alter any of the existing inner faces, so we only have to guarantee that the newly created inner face maintains the property. Since we always generate triangles by performing an *add_vertex* followed by an *add_edge*. That is, we never perform two *add_vertex* or two *add_edge* in a row. Since we always intercalate, we know that the resulting polygons will always have 3 edges (the edge we are adding everything to, the edges we create when we add the vertex and the final edge created by the *add_edge*) and therefore the newly created inner faces will always have 3 edges. So by an induction step on property 1 and since we have shown that our newly created inner face will always have 3 edges, we have shown that property 1 is maintained.

Property two might be broken by this operation, as the outer polygon is changed by the newly added edge. This new face is composed of the edges we get by traversing the incident face of the edge $halfE2$, which is connected to an edge of the outer face. By an induction step on property 2, we know that beggining a traversal on a half-edge that belongs to the previous outer face, we will traverse the entire outer face, starting on $halfE2$ and ending on $halfE2$, which is exactly what we want and proves that our newly created outer face also contains all of the outer edges.

2.4 Proof of correctness finalized

Since we have proven that performing *add_vertex* followed by *add_edge* n times, we maintain the properties. Therefore, the DCEL is provably correct for any number of vertices n, created by triangulation.

3 Polygon generation

To start generating a polygon we begin with 3 random points that are not collinear and that form a valid triangle.

3.1 Assuring randomness when generating polygons

To decide where to add a new vertex, we first generate a random point in the interval $(0, 0)$ to (M, M) and verify whether it is inside the currently generated polygon.

If it is inside the currently generated polygon, then it is not a valid point and is discarded.

If the point is not contained in the polygon, we will then get a random half-edge from the outer face.

When we get this edge, we will then verify whether the connections that will have to be created intersect with the polygon, if they do intersect, we discard the point and the face and try again, if not we add the point to the polygon by calling *add_vertex* followed by *add_edge*.

3.2 Checking whether a point is contained in the polygon

To verify whether a point is contained in the polygon, we use the property 1, assured by the DCEL structure. We traverse the list of faces (excluding the outer face), which represents the triangulation of the polygon. For each of these triangles, we check whether the point is contained in the triangle. If it is, then it's part of the polygon if it's not then we keep checking until we have checked all of the triangles in the data structure.

If the point does not belong to any of the triangles, then it is not contained in the polygon.

We can check whether a point is contained in a triangle with a time complexity of $O(1)$, by calculating the triangle area formula.

So, since there are $N-2$ triangles in the polygon as all triangles share 2 vertices with other triangles we know that verifying whether a point is contained in the polygon takes $O(N)$ time.

3.3 Checking whether a segment intersects with edges of the existing polygon

To check whether a given segment intersects with the polygon, we traverse the edges that compose the outer face (We know that this outer face contains all the outer edges, guaranteed by the property 2 of the DCEL). We only have to check the outer face because we know that any point that we want to add is guaranteed to be outside of the polygon so for the segment created by joining a vertex from the polygon with a new random vertex, for it to intersect, it has to intersect with the outer edges of the polygon.

To check whether a given segment intersects with another segment, we use vector mathematics which has a complexity of $O(1)$.

We have to check the two new segments against all the segments that belong to the outer face. Since we have $N-1$ segments, the total time to check whether a segment intersects with the polygon is $O(N)$.

3.4 Total complexity for point generation

For each newly generated point we have to:

- Check whether the generated point is inside the polygon. Complexity: $O(N)$.
- Select a random face from the outer face. Complexity: $O(N)$.
- Check whether the two new segments we have to create intersect with the outer face. Complexity: $O(N)$.
- Add the new vertex to the DCEL. Complexity: $O(1)$.
- Add the new edges to the DCEL. Complexity: $O(|H| + |F|)$, however since we know the number of edges is $N-1$ (so $|H|$ is $2*(N-1)$) and the number of faces is $N-2$, we know that the complexity of this operation is also $O(N)$.

So in total to check if a given vertex can be added to the DCEL, choosing the face to add it to, checking whether the segments intersect and actually adding and connecting the new vertex has a complexity of $O(N)$, where N is the number of vertexes.

4 Polygon visualizer

The program outputs a *result.json* that contains all the faces of the polygon (so it contains the triangulation and the outer face). This file serves as an input to the *visualizer.py* script, which takes the triangulation and produces an image of the polygon.

To use this visualizer do the following command:

```
$ python3 visualizer.py M result.json
```

Note that the last argument is the path to result.json, so if the visualizer is not on the same folder as the result.json the path to it must be provided.