Delaunay triangulation

Andrea Frigo
mat: 223813
andrea.frigo@studenti.unitn.it

ABSTRACT

This paper describes an implementation of the Delaunay triangulation using the incremental approach described in the book "Computational Geometry - Algorithms and Applications, 3rd Ed.". The formal definitions and demonstrations are not stated in this document, but are present in the book. In this document it is described the implementation of the algorithm in a Python program.

1 INTRODUCTION

Delaunay triangulation is useful in the spatial domain, in particular when representing a map using a set of points in the 2d space, having for each of them also the height information, in this way it is possible to get a polyhedral terrain represented by 3d triangles. The challenge described in this document resides in the 2d triangulation, in fact, given a set of points, there are different possible triangulations, the goal is to create the triangulation that represents better the terrain. In the book it is demonstrated that the best triangulation is the one that maximises the minimum angle over all the possible triangulations, this triangulation is the Delaunay triangulation. From this statement an important consideration can be made: the Delaunay triangulation is unique given a set of points.

The Delaunay triangulation can be obtained using two different approaches: it can be derived from the Voronoi diagram, or it can be computed using the random incremental approach, in the project described by this document the second approach is used.

2 ALGORITHM DESCRIPTION

Given a set P of 2d points all different from each other, where a point Pi is composed by two coordinates (PiX>=0, PiY>=0), the general algorithm is the following.

The algorithm begins creating a large triangle that contains all the points P, in particular the three vertices of the triangle are: the highest point P0 of P and two special points P-1 and P-2. A point P1 is greater than an other point P2 if and only if the y-coordinate of P1 is greater than the one of P2 or if they are equal and the x-coordinate of P1 is greater than the one of P2. The two special points are represented in the described implementation as negative points P-1=(-1,-1) and P-2=(-2,-2), but are not real points in the plane, but just a representation of immaginary points, and they are treated differently. This is done in order to avoid the usage of points with large coordinates that would make the algorithm slower. P-1 represents a point below the set P and to the right of the set, P-2 instead represent a point above P and to the left of the set. Both P-1 and P-2 must be outside any circle defined by three points of P for the algorithm to work, this is why the special points are created, also all the set P has to be contained inside the triangle (P0, P-1, P-2).

As stated before, the algorithm is random incremental, so the execution can be divided in P steps, where each step is the triangulation of the previous one with a new point.

The general algorithm presented in the book can be seen in fig.1, the implementation follows this algorithm, but with some adjustments, and it is described in the next sessions.

Algorithm DELAUNAYTRIANGULATION(P)
Input. A set P of $n+1$ points in the plane.
Output. A Delaunay triangulation of P.
1. Let p_0 be the lexicographically highest point of P, that is, the rightmost among the points with largest y-coordinate.
2. Let p_{-1} and p_{-2} be two points in $\mathbb{R}^2$ sufficiently far away and such that P is contained in the triangle $p_0p_{-1}p_{-2}$.
3. Initialize $\mathcal{T}$ as the triangulation consisting of the single triangle $p_0p_{-1}p_{-2}$.
4. Compute a random permutation $p_1, p_2, \ldots, p_n$ of $P \setminus \{p_0\}$.
5. **for** $r \leftarrow 1$ **to** n
6. **do** (∗ Insert p_r into $\mathcal{T}$: ∗)
7. Find a triangle $p_ip_jp_k \in \mathcal{T}$ containing p_r.
8. **if** p_r lies in the interior of the triangle $p_ip_jp_k$
9. **then** Add edges from p_r to the three vertices of $p_ip_jp_k$, thereby splitting $p_ip_jp_k$ into three triangles.
10. LEGALIZEEDGE($p_r, \overline{p_ip_j}, \mathcal{T}$)
11. LEGALIZEEDGE($p_r, \overline{p_jp_k}, \mathcal{T}$)
12. LEGALIZEEDGE($p_r, \overline{p_kp_i}, \mathcal{T}$)
13. **else** (∗ p_r lies on an edge of $p_ip_jp_k$, say the edge $\overline{p_ip_j}$ ∗)
14. Add edges from p_r to p_k and to the third vertex p_l of the other triangle that is incident to $\overline{p_ip_j}$, thereby splitting the two triangles incident to $\overline{p_ip_j}$ into four triangles.
15. LEGALIZEEDGE($p_r, \overline{p_ip_l}, \mathcal{T}$)
16. LEGALIZEEDGE($p_r, \overline{p_lp_j}, \mathcal{T}$)
17. LEGALIZEEDGE($p_r, \overline{p_jp_k}, \mathcal{T}$)
18. LEGALIZEEDGE($p_r, \overline{p_kp_i}, \mathcal{T}$)
19. Discard p_{-1} and p_{-2} with all their incident edges from $\mathcal{T}$.
20. **return** $\mathcal{T}$

Figure 1: General algorithm from the book

An important part is the function to legalize the edge, that checks if the edge is legal and, if it is not, swaps it. The algorithm for this function can be seen in fig.2. The function is recursive because each time that there are some modifications in the triangulation the edges involved could become illegal (and so should be legalized).

LEGALIZEEDGE($p_r, \overline{p_ip_j}, \mathcal{T}$)
1. (∗ The point being inserted is p_r, and $\overline{p_ip_j}$ is the edge of $\mathcal{T}$ that may need to be flipped. ∗)
2. **if** $\overline{p_ip_j}$ is illegal
3. **then** Let $p_ip_jp_k$ be the triangle adjacent to $p_rp_ip_j$ along $\overline{p_ip_j}$.
4. (∗ Flip $\overline{p_ip_j}$: ∗) Replace $\overline{p_ip_j}$ with $\overline{p_rp_k}$.
5. LEGALIZEEDGE($p_r, \overline{p_ip_k}, \mathcal{T}$)
6. LEGALIZEEDGE($p_r, \overline{p_kp_j}, \mathcal{T}$)

Figure 2: Legalize edge algorithm

3 IMPLEMENTATION

In this section it is described the actual implementation of the algorithm, showing the different parts of the code and explaining what choices have been made.

3.1 Main algorithm

After reading the points and creating the first triangle (P0, P-1, P-2), the main part of the algorithm described in the previous sections and showed in fig.1 has been implemented in the following way:

```
1  while(len(input)>0):
2      i=i+1
3      p = input.pop(0)
4      if pointGreater(minPoint, p): minPoint = p
5      triangles = findTrianglesPoint(p, dag, list(dag.keys())[0])
6      edge = None
7      if len(triangles)>1:
8          t1 = None
9          t2 = None
10         k = 0
11         while (k<len(triangles) and not edge):
12             edge = pointOnTriangle(p, triangles[k])
13             k+=1
14         if edge:
15             t1 = triangles[k-1]
16             while (k<len(triangles)):
17                 if edgeOfTriangle(edge, triangles[k]):
18                     t2 = triangles[k]
19                     k += len(triangles)
20                 k+=1
21             pk = None
22             for point in t1:
23                 if(point != edge[0] and point != edge[1]):
24                     pk = point
25             pl = None
26             for point in t2:
27                 if(point != edge[0] and point != edge[1]):
28                     pl = point
29             for t in [t1,t2]:
30                 triangulation.remove(t)
31                 if pk in t:
32                     dagAppend(dag, t, (edge[0], pk, p))
33                     dagAppend(dag, t, (edge[1], pk, p))
34                     addTriangle((edge[0], pk, p), triangulation)
35                     addTriangle((edge[1], pk, p), triangulation)
36                 else:
37                     dagAppend(dag, t, (edge[0], pl, p))
38                     dagAppend(dag, t, (edge[1], pl, p))
39                     addTriangle((edge[0], pl, p), triangulation)
40                     addTriangle((edge[1], pl, p), triangulation)
41             dagLeaf(dag, (edge[0], pk, p))
42             dagLeaf(dag, (edge[1], pk, p))
43             dagLeaf(dag, (edge[0], pl, p))
44             dagLeaf(dag, (edge[1], pl, p))
45             legalizeEdge(p, (edge[0], pl), dag, triangulation, minPoint,
                   maxPoint)
46             legalizeEdge(p, (pl, edge[1]), dag, triangulation, minPoint,
                   maxPoint)
47             legalizeEdge(p, (edge[1], pk), dag, triangulation, minPoint,
                   maxPoint)
48             legalizeEdge(p, (pk, edge[0]), dag, triangulation, minPoint,
                   maxPoint)
49     if edge==None:
50         t = triangles[0]
51         triangulation.remove(t)
52         addTriangle((p, t[0], t[1]), triangulation)
53         addTriangle((p, t[1], t[2]), triangulation)
54         addTriangle((p, t[2], t[0]), triangulation)
55         dagAppend(dag, t, (p, t[0], t[1]))
56         dagAppend(dag, t, (p, t[1], t[2]))
57         dagAppend(dag, t, (p, t[2], t[0]))
58         dagLeaf(dag, (p, t[0], t[1]))
59         dagLeaf(dag, (p, t[1], t[2]))
60         dagLeaf(dag, (p, t[2], t[0]))
61         legalizeEdge(p, (t[0], t[1]), dag, triangulation, minPoint, maxPoint)
62         legalizeEdge(p, (t[1], t[2]), dag, triangulation, minPoint, maxPoint)
63         legalizeEdge(p, (t[2], t[0]), dag, triangulation, minPoint, maxPoint)
```

The points are generated randomly, so for each execution the first point of the input gets removed and inserted in the triangulation. Both the highest and the lowest points are kept, they are useful for legalizing the edges (see later).

The findTrianglesPoint() function returns a list of triangles that contain the point p, the number of triangles that are given from this function is important, because if there is only a triangle then the point is inside this one, otherwise if it gives two triangles then the point lies in the common edge of the two triangles. If the condition of line 7 is satisfied, the algorithm finds the common edge and the vertexes of the two triangles that are needed to update the triangulation (line 21-28). The two triangles are then divided in four new triangles, the triangulation is updated and the legalization can begin.

If the point is inside just one triangle instead the triangle is divided in three new triangles, the triangulation is updated and the legalization can begin.

This part of the algorithm works exactly as the general algorithm stated before, the different functions and structures used are described later in this document.

3.2 Useful functions

In this section are described some functions that are used in the main part of the algorithm or to legalize the edges. The code of most of these functions is not described because it is simple to understand.

- **pointGreater(p1,p2)**: checks whether the point p1 is greater than point p2, using the definition of "greater" previously defined;
- **pointLinePosition(point, line)**: this function is needed in many parts of the algorithm, it returns the position of the point w.r.t. the line (right, left or on the line), the point has to be a real point, the line can also be made of one or two special points;
- **pointInTriangle(point, triangle)**: using the *pointLinePosition* function it checks whether a point is inside a triangle (also on the edge);
- **pointOnTriangle(point, triangle)**: it's similar to the previous one, but checks only if a point is on the edges of the triangle;
- **isInsideCircle(p, triangle)**: this is the function used to check if an edge is legal when both the points and the edge are real, as stated in the book an edge is legal if the point p lies outside the circle made by the three points of the triangle;
- **findTrianglesPoint(p, dag, start, ret=None)**: using the *pointInTriangle* function it returns all the triangles that have p inside or on the border, at most two triangles should be returned (in case the point is on the edge in common), this function uses a special structure to search for the point, this is described better later when describing the DAG structure;
- **edgeOfTriangle(edge, triangle)**: checks whether the edge is an edge of the triangle;
- **findTrianglesEdge(edge, triangulation)**: gives the triangles that have the edge, as before should return at most two triangles;
- **isSameTriangle(t1, t2)**: checks whether the two triangles are the same (are composed of the same points, but maybe stored in different order);
- **addTriangle(triangle, triangulation)**, **dagAppend(dag, node, triangle)**, **dagLeaf(dag, triangle)**: functions to add a triangle in the triangulation, they are described better when describing the structure used.

There is one important function that is used inside the legalize edge function, in this case also the code is shown, because this is a function that was not present in the algorithm described in the book, but it is needed in order to avoid to swap some edges when it can not be done.

```
1  def isConvex(p0, p1, e):
2      e0 = e[0]
3      e1 = e[1]
4      if (p0[0]<0 or p1[0]<0) and (e0[0]>= 0 and e1[0] >= 0):
5          return True
6      if (e0[0]<0 or e1[0]<0) and (p0[0]<0 or p1[0]<0):
7          if e0==(-2,-2) and p0[0]<0:
8              return pointGreater(p1, e1)
9          if e0==(-2,-2) and p1[0]<0:
10             return pointGreater(p0, e1)
11         if e1==(-2,-2) and p0[0]<0:
12             return pointGreater(p1, e0)
13         if e1==(-2,-2) and p1[0]<0:
14             return pointGreater(p0, e0)
15         if e0==(-1,-1) and p1[0]<0:
16             return pointGreater(e1, p0)
17         if e0==(-1,-1) and p0[0]<0:
18             return pointGreater(e1, p1)
19         if e1==(-1,-1) and p1[0]<0:
20             return pointGreater(e0, p0)
21         if e1==(-1,-1) and p0[0]<0:
22             return pointGreater(e0, p1)
23     if (e0[0]<0 or e1[0]<0) and (p0[0]>=0 and p1[0]>=0):
24         line = (p0, p1) if pointGreater(p1,p0) else (p1,p0)
25         if e0[0]>=0:
26             if e1==(-2,-2) and pointLinePosition(e0, line)==1: return False
27             if e1==(-1,-1) and pointLinePosition(e0, line)==-1: return False
28         elif e1[0]>=0:
29             if e0==(-2,-2) and pointLinePosition(e1, line)==1: return False
30             if e0==(-1,-1) and pointLinePosition(e1, line)==-1: return False
31         return True
```

The **isConvex(p0, p1, e)** function checks whether the quadrilateral composed by the four given points is convex. This is useful because, as stated in the book, if the quadrilateral is not convex the edge can not be swapped, because it would create two new triangles, one inside the other (this is not allowed in triangulation), with a common edge, this can be seen in the example of fig.3. The left part of the figure shows the two triangles that compose the non-convex quadrilateral (ABC and ACD), if the common edge (AC) is considered illegal it would be swapped, and the result can be seen on the right (ABD and BCD). In order to avoid this, inside the legalize edge function the control on the quadrilateral has to been made. If all the points are real this control is not needed, because the standard algorithm works, the problem arises when one point of the edge is special, in this example the point A could be P-2.

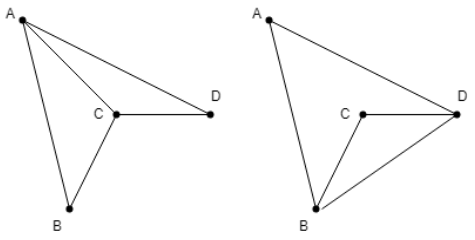

Figure 3: Swap edge with non convex quadrilateral

As said before, at least one point is a special point (the situation with no special points is already handled, also the case where the edge is made by two special point is handled, and it is not possible that both the points are special because at least one is the point added in that execution), so the condition in line 4 checks if the special point is not one of the edge, in this case the edge is legal, because by definition any special point lies outside the circle made by three real points.

In line 6 there is the case where one point of the edge and one of the two points are special, in this case it is enough to check which point between the real points is the greatest. This control is not perfect, it is not true that, if the function returns true, the quadrilateral is convex, but it is true that, if the function returns false, the quadrilateral is not convex. The function is used only negating it, so the output in this case is correct.

The condition in line 23 handles the case where both the points are real, so one point of the edge is special, it is enough to check the position of the real point of the edge w.r.t. the line made by the two points to understand if it lies on the same side of the special point (not convex) or not (convex).

3.3 Legalize edge

This is one of the most important part of the algorithm, it would be trivial without special points, but those made this part more tricky. In this implementations there are two main additions w.r.t. the algorithm proposed in the book. The code is the following.

```
1  def legalizeEdge(p, e, dag, t, minPoint, maxPoint):
2      firstTriangle = list(dag.keys())[0]
3      triangles = findTrianglesEdge(e,t)
4      if edgeOfTriangle(e, firstTriangle):
5          return
6      k = None
7      for triangle in triangles:
8          for point in triangle:
9              if(point != p and point != e[0] and point != e[1]):
10                 k = point
11     if p[0] >= 0 and e[0][0] >= 0 and e[1][0] >= 0 and k and k[0] >= 0:
12         if isInsideCircle(p, [e[0], e[1], k]):
13             for triangle in triangles:
14                 t.remove(triangle)
15                 dagAppend(dag, triangle, (p,k,e[0]))
16                 dagAppend(dag, triangle, (p,k,e[1]))
17             addTriangle((p,k,e[0]), t)
18             addTriangle((p,k,e[1]), t)
19             dagLeaf(dag, (p,k,e[0]))
20             dagLeaf(dag, (p,k,e[1]))
21             legalizeEdge(p, (e[0], k), dag, t, minPoint, maxPoint)
22             legalizeEdge(p, (e[1], k), dag, t, minPoint, maxPoint)
23         return
24     if k:
25         if not isConvex(p, k, e):
26             return
27         if (e[0][0]<0 or e[1][0]<0):
28             pswap = e[0] if e[0][0]>=0 else e[1]
29             if (minPoint==pswap or maxPoint==pswap):
30                 backuptriangulation = t[:]
31                 for triangle in triangles:
32                     backuptriangulation.remove(triangle)
33                 addTriangle((p,k,e[0]), backuptriangulation)
34                 addTriangle((p,k,e[1]), backuptriangulation)
35                 p_1 = False
36                 p_2 = False
37                 for triangle in backuptriangulation:
38                     if pswap in triangle:
39                         if (-1, -1) in triangle:
40                             p_1 = True
41                         if (-2, -2) in triangle:
42                             p_2 = True
43                 if p_1 == False or p_2 == False:
44                     return
45         if(min(p[0], k[0]) >= min(e[0][0], e[1][0])):
46             for triangle in triangles:
47                 t.remove(triangle)
48                 dagAppend(dag, triangle, (p,k,e[0]))
49                 dagAppend(dag, triangle, (p,k,e[1]))
50             addTriangle((p,k,e[0]), t)
51             addTriangle((p,k,e[1]), t)
52             dagLeaf(dag, (p,k,e[0]))
53             dagLeaf(dag, (p,k,e[1]))
54             legalizeEdge(p, (e[0], k), dag, t, minPoint, maxPoint)
55             legalizeEdge(p, (e[1], k), dag, t, minPoint, maxPoint)
56         return
57     return
```

The first step is to get the triangles containing the edge that has to be checked and in case legalized, this is done using the *findTrianglesEdge* function described before.

If the edge is an edge of the first triangle then it is legal by construction. If it is not the case, in lines 6-10 it is found the second point for the algorithm (the vertex of the second triangle

containing the edge, the vertex of the first triangle is given in input from the main algorithm).

If all the points are real, the normal approach is used: if one of the two point lies inside the circle made by the other three (the other point and the two of the edge) then the edge is illegal and is flipped, changing the structures to update the triangulation and calling recursively the legalize edge on the new triangles. This step is the same for any point chosen (if p lies inside the circle defined by edge and k, the same holds for k and the circle defined by the edge and p).

At this point both the cases of two special point in the edge and of all real points have been handled, so the only cases that could happen are having just one special point, or two: one of the edge and one between the two points. This is the perfect situation to use the *isConvex* function, if the quadrilateral is non-convex then the edge can not be flipped.

If the quadrilateral is convex maybe the edge is illegal, so the algorithm has to continue. The next condition is the following: both the highest and the lowest point of the triangulation have to be connected to both P-1 and P-2. This condition can be found in the book, but is not inserted in the algorithm. The condition has to be maintained even after swapping an edge, and the only way to break it is if one point of the edge is a special point, and the other is the highest or lowest point. If this is the case the algorithm simulates a swap without modifying the data structures and checks if the condition has been broken, if it is the case then the edge has to be legal, otherwise nothing happens and the algorithm continues its execution. This part can be seen in lines 27-44.

At this point it is known that one or two of the four points are special and that the possible swap would not break any condition, so the control in line 45 is made, in the book it is stated that, if min(p, k) < min(i,j) than the edge is legal, where p and k are the index of the two points and i and j are the index of the points of the edge. This condition has been reversed in order to catch the illegal edges and the indexes have been replaced by the x-coordinates of the points. This replacements can be justified with these statements:

- only one of the two points p and k can be special, because the other is the point inserted during the current execution;
- at most one of the point of the edge is special, because the condition with both special has already been handled
- the x-value of the special points is the same as their index (-2 for P-2 and -1 for P-1)
- the index of the real points is not important, the important information is that they are real and not special, so that their index is not negative, but the same holds for the x-coordinate, because all the points generated have x-coordinate >=0

The last statement requires more justification, because using x-coordinates instead of indexes for real points could change their order. In the book the index are assigned basing on which point is greater, so Pi > Pj if i>j, but it could be that the x-coordinate of Pj is greater than the one of Pi. In this case this order does not matter, because if both the points are real, their order is not relevant, because one point of the edge would be special, so min (p,k)>min(i,j) if p>k and also if p<k. The important checks are for the non real points, and the condition using the x-coordinates is the same as the one using the indexes for them. If the condition on line 45 is satisfied the edge is illegal, so it is swapped, the structures are changed and the function gets called recursively.

If none of the conditions stated brought to an edge swap, the edge is legal and the function terminates.

3.4 Structures

Two structures are created and maintained for all the execution of the program: a directed acyclic graph (DAG) and a list for triangulation.

The DAG is stored as a Python dictionary, where the keys are the nodes of the DAG and the content of the dictionary for a specific key is the children of that node. Leafs are represented by nodes without children. The DAG is created at the beginning as an empty dictionary, then the first triangle (P0, P-1, P-2) is inserted and its value is set to an empty list (at the beginning the triangle has no children so it is a leaf). If a triangle T1 gets split in 3 (a new point is inside T1) or 4 (a new point is on an edge shared between T1 and another one) smaller triangles, these smaller triangles are inserted in the list of T1 that represents its children and for each smaller triangle a node is created as a leaf (with an empty list). In case an edge shared between T1 and T2 is flipped creating two new triangles, both T1 and T2 have the new triangles as children.

The leafs of the DAG represent the current triangulation, and the structure can be used to find the triangles containing a point (operation that has to be done for each execution) in an effective way. This is because a point can be inside at most two triangles for each layer of the DAG, reducing the number of triangles that need to be checked.

The other structure is simpler, it is just a list containing the current triangulation. This structure does not keep any information on the previous executions (the DAG can also be useful for debugging the code) and it is not used to find the triangles containing a point because it is inefficient compared to the DAG. The list is instead better when the entire triangulation is needed, for example for the final output, because in any case all the triangles have to be read. Also for finding the triangles that contain a specific edge the list is more useful, because to do so the entire DAG should be checked, increasing complexity.

For both the DAG and the triangulation list some function to add elements have been created, these performs the adding of the elements and also avoid inserting duplicates in the structures.

3.5 Complexity

The algorithm has been implemented following the steps presented in the book, only the conditions for *isConvex* and for keeping the edges between the highest and lowest point with both P-1 and P-2 have been added. Given n points in the set P. The first function performs only one comparison each time it is called, while the second one performs two complete reads on the triangulation list.
Both these additions have temporal complexity less then $O(n \log n)$, so the temporal complexity is $O(n \log n)$ as stated in the book. While the first functions does not use any additional structure, the second condition simulates an edge swap, generating a copy of the triangulation, but this copy can be deleted every time the legalize edge function finishes its execution, so the space complexity remains $O(n)$ as stated in the book.

The temporal and spatial complexity analysis is stated in the book, the proof is not reported here.

4 PROJECT STRUCTURE AND EXECUTION

It is possible to create an input file using the following command:
python generate_input.py np max
where np is the number of points and max is the maximum x and y-coordinate. The script generates a text file "input.txt" containing np points generated with coordinates (random(0, max), random(0, max)).

The Delaunay triangulation algorithm can be executed using the following command:
python main.py input.txt
where input.txt is the input file. The program creates the image *plots/triangulation.png*.

In the same way also the algorithm that generates the Delaunay triangulation using scipy can be executed:
python compare.py input.txt.
The program creates the image *plots/compare.png*. This can be used to check the correctness of the algorithm.