Path Planning using D* Lite on a Voronoi Diagram

Aleman E., Hulchuk V.

***Abstract*— The present paper describes the implementation of the D* Lite algorithm and the Voronoi Diagram together for solving the path planning problem in robot navigation. The Voronoi Diagram allows us to create a roadmap (graph) given a set of point obstacles, the the D* algorithm is used to solve the graph from a starting node to a goal node while being able to replan if a hidden obstacle was present in one of the vertices of the initial map. The present implementation fuses both algorithms to provide a robust solution for robot navigation in dynamic environments.**

I. Introduction

Voronoi

The Voronoi Diagram is a partition of the plane into regions, so that each region is the closest to the corresponding point obstacle (often called "*site*" in the litereature) as shown in Figure 1.

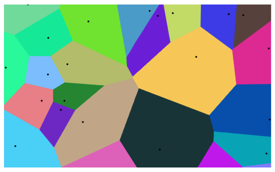


Figure 1. The Voronoi Diagram.

On Figure 1, the sites are the black points. For the present application, the Voronoi diagram is used to create a topological map of the environment that contains obstacles, where the robot will move.

Providing a topological map to a path finding algorithm would lead to a safe path far from the obstacles because each vertex is a middle perpendicular between the sites (obstacles). This map representation can be very useful for applications such as multi robot navigation where a good distribution of the configuration space is required to allow multiple robots to move with agility and without collision.

There are a lot of different algorithms to build the Voronoi diagram. The complexity varies from N^4 to N*log(N).The "Brute-Force" algorithm would check all the possibilities and provide a complexity of N^4. This could only be used to have a toy implementation of the algorithm. The only advantage is the simplicity of implementing it.

An elegant algorithm which will be used in this project is an "iterative algorithm" with complexity N^2. It is based on the idea of building the simple Voronoi Diagram for three sites and afterwards inserting other sites one by one into the existing Diagram. The complexity of the insertion of one site is N, and thus the complexity of the algorithm is $N*N = N^2$.

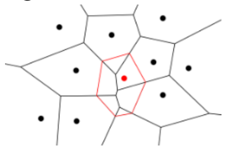


Figure 2. The iterative algorithm for constructing the Voronoi Diagram.

One of the fastest algorithms for building the Voronoi Diagram is the one called "Fortune's algorithm" [3] with complexity N*log(N), and an elegant "Divide and Conquer" algorithm [3] also with complexity N*log(N),. The ideas behind them are effective, nevertheless it could be not the best option for several types of applications. The backdraw of these fast algorithms is that it is not natively suited to adding sites one by one.

Let's say that obstacles (sites) on the map are appearing one by one (dynamic environment). And each time it is needed to recalculate the new Graph for the Voronoi Diagram. In this case, for every new site, complexity of adding this site is N*log(N) - recalculating the entire Graph. But for the iterative algorithm, the complexity of this action is only N.

The whole procedure of building the graph, adding sites one by one, will have the complexity $N^2\log(N)$ for N*log(N) algorithms. The complexity of building all the intermediate graphs is:

$$\sum_{i=1}^{N} i * log(i)$$

The order of this sum will be the same as the order of the following integral:

$$\sum_{i=1}^{N} i * log(i) \approx \int_{1}^{N} x\, log(x) dx \approx N^2 log(N)$$

Table 1 compares the complexities of those algorithms.

Table 1. Comparison of different Voronoi Diagram approaches

Algo\Task	Build a Diagram	Insert an obstacle	Build all intermediate Graphs
Brute-Force	N^4	N^4	N^5
Iterative	N^2	N	N^2
Fortune's	N*log(N)	N*log(N)	N^2log(N)
C&D	N*log(N)	N*log(N)	N^2log(N)

The fact that allows us to estimate the complexity of the algorithm is that the number of edges in the Voronoi Diagram is not more than 3N-6 [1], which is the order of N. And for the iterative algorithm, during the iteration it is needed to find the intersection point with every edge, which gives the complexity of N^2. Execution time T(N) is shown in Figure 3:

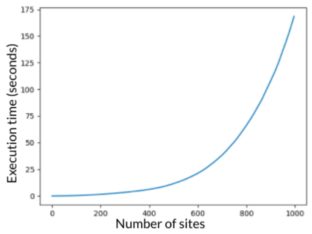


Figure 3. Execution time VS #Sites.

Nevertheless, the algorithm was invented and implemented from scratch and only afterwards was found on the internet. That is why the implementation can differ from the one explained in the source [3].

D* Lite

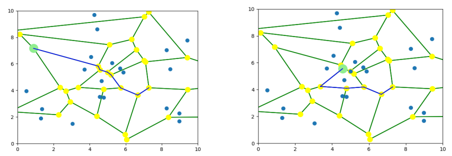


Figure 4. D* Lite Replanning when obstacle is detected

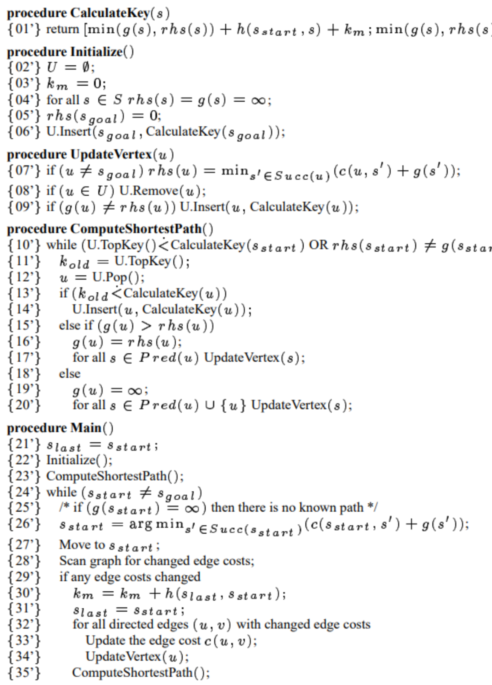
The D* algorithm is an incremental path planning algorithm developed by Sven Koenig that aims to solve the shortest path planning problem for dynamic environments, which are environments that change over time (moving obstacles or obstacles not present when building the prior map). This algorithm falls into the category of incremental search algorithms that make use of prior information for the replanning process (as shown in Figure 4), instead of throwing away the information gathered in the planning process when replanning again (as it is the case in the A* algorithm), which decreases the number of operations and thus becomes a more suitable solution for mobile robotics which usually require real-time and online planning to safely reach a goal in a minimum time.

The D* Lite algorithm can be seen as an improved version of the LPA* (Lifelong Planning A*) which is another incremental path planning also developed by Sven Koenig. Both algorithms share many characteristics between themselves and take some elements from A*, for example the use of a heuristic h(s) (euclidean distance for the present implementation) and the use of the estimated distance to the start g(s). LPA* and D* Lite incorporates the use of the rhs(s) value, which is "a one step lookahead of the g-values that satisfy the following relationship" [2] shown below:

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start} \\ \min_{s' \in Pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases}$$

The main goal of the D* Lite is to use the concept of relaxation to constantly update the different values of the vertices and make them locally consistent, which means to have their rhs(s) values equal to their g(s) values. The locally inconsistent vertices are stored in a priority queue and are the ones to be updated in order to make them locally consistent. The priority queue is ordered considering a key vector k = [k1, k2] for each node, whose first element is the f(s) value commonly used in A* and the second value corresponds to the g(s) value. This formulation of the priority queue is specially useful for solving the problem of two nodes having the same f(s) value. The queue is kept after any replanning process to store useful information for future replanning.

At this point, it is important to mention that the D* Lite algorithm starts with a search from the goal state to the start position (position of the robot), and as the robot moves, the algorithm updates the inconsistent vertices as a result of detecting obstacles where there was no prior information about them before. The robot motion happens by moving to the successor node, which happens to minimize c(s,s')+g(s'), (cost to successor from current node + total cost to successor node). A detailed explanation of how the algorithm works is shown after the original pseudocode of D* Lite shown in Figure 5 [2].


```
procedure CalculateKey($s$)
{01'} return [$\min(g(s), rhs(s)) + h(s_{start}, s) + k_m$; $\min(g(s), rhs(s))$];
procedure Initialize()
{02'} $U = \emptyset$;
{03'} $k_m = 0$;
{04'} for all $s \in S$ $rhs(s) = g(s) = \infty$;
{05'} $rhs(s_{goal}) = 0$;
{06'} U.Insert($s_{goal}$, CalculateKey($s_{goal}$));
procedure UpdateVertex($u$)
{07'} if ($u \neq s_{goal}$) $rhs(u) = \min_{s' \in Succ(u)}(c(u, s') + g(s'))$;
{08'} if ($u \in U$) U.Remove($u$);
{09'} if ($g(u) \neq rhs(u)$) U.Insert($u$, CalculateKey($u$));
procedure ComputeShortestPath()
{10'} while (U.TopKey() $\dot{<}$ CalculateKey($s_{start}$) OR $rhs(s_{start}) \neq g(s_{start})$)
{11'}   $k_{old}$ = U.TopKey();
{12'}   $u$ = U.Pop();
{13'}   if ($k_{old} \dot{<}$ CalculateKey($u$))
{14'}     U.Insert($u$, CalculateKey($u$));
{15'}   else if ($g(u) > rhs(u)$)
{16'}     $g(u) = rhs(u)$;
{17'}     for all $s \in Pred(u)$ UpdateVertex($s$);
{18'}   else
{19'}     $g(u) = \infty$;
{20'}     for all $s \in Pred(u) \cup \{u\}$ UpdateVertex($s$);
procedure Main()
{21'} $s_{last} = s_{start}$;
{22'} Initialize();
{23'} ComputeShortestPath();
{24'} while ($s_{start} \neq s_{goal}$)
{25'}   /* if ($g(s_{start}) = \infty$) then there is no known path */
{26'}   $s_{start} = \arg\min_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$;
{27'}   Move to $s_{start}$;
{28'}   Scan graph for changed edge costs;
{29'}   if any edge costs changed
{30'}     $k_m = k_m + h(s_{last}, s_{start})$;
{31'}     $s_{last} = s_{start}$;
{32'}     for all directed edges $(u, v)$ with changed edge costs
{33'}       Update the edge cost $c(u, v)$;
{34'}       UpdateVertex($u$);
{35'}     ComputeShortestPath();
```
Figure 5. D* Lite pseudocode from Sven Koenig as presented in [2]

As shown in Figure 5, the whole execution takes place in the **Main()** procedure which starts by initializing (procedure **Initialize()**) the different values and data structures of the graph such as the priority queue U, the key modifier km which is a way to keep track of the heuristic changes through the robot motion. This is especially useful to compare the changed heuristic to the values stored in the priority queue. Every node receives a rhs(s) and g(s) values equal to ∞ except for the goal node which receives a rhs(s) value equal to 0. The goal node is inserted in the priority queue (because it is inconsistent rhs(s) $\neq$ g(s)).

After the initialization, the procedure **ComputeShortestpath()** is called which basically executes LFA* over the graph from the goal to start position, updating the values of the nodes (applying relaxation since the rhs(s) and g(s) values won't be ∞ anymore) and making them consistent.

In the following the algorithm itself considers the motion of the robot in its core, which makes even more evident the easiness to connect the D* Lite algorithm to a full navigation system by just providing a control signal in line 27 and getting information from an obstacle detection system in line 28.

If there is a new obstacle detected that was not previously considered in the map, the algorithm has to update the different values of the vertices to take into account the new change in the planning process, the procedure **UpdateVertex()** is called which implies one out of three actions:

- To update the rhs(s) value of a given node
- If the node is in the priority queue, remove it from there
- If the node is inconsistent, insert it in the priority queue

Once the vertices have been updated, the **ComputeShortestpath()** procedure is called again to find another shortest path for the robot to continue moving without having to run the algorithm from scratch.

II. Development and Results

Voronoi Diagram

The implemented algorithm executes the main loop, which can be seen on the following diagram shown in Figure 6:

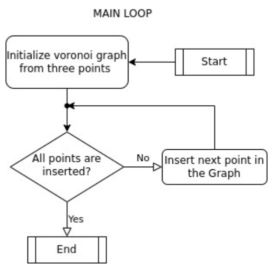

MAIN LOOP
Start
Initialize voronoi graph from three points
All points are inserted?
No
Insert next point in the Graph
Yes
End

Figure 6. Main loop of the iterative algorithm.

As can be seen, the iterative algorithm adds the points (sites) one by one until all the obstacles are added to the Graph. Besides, a separate task is to build an initial Voronoi Diagram.

The addition of the sites follows the following diagram shown in Figure 7:

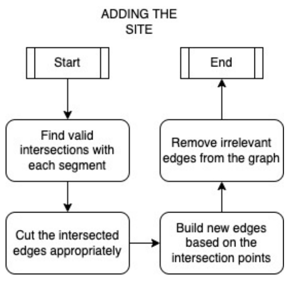

ADDING THE SITE
Start
Find valid intersections with each segment
Cut the intersected edges appropriately
Build new edges based on the intersection points
Remove irrelevant edges from the graph
End

Figure 7. Schema of the iteration.

The process consists of the four main procedures:

- Find valid intersections of each segment;
- Cut the intersected edges;
- Build new edges;
- Remove irrelevant edges;

To use the algorithm, there are several structures used:

- For every edge, points that it separates are stored (2 points for every edge).
- Edges can be of two types: finite and infinite
- For an infinite type, one point is a voronoi vertex, and another is a second point in the direction of the beam. Criteria to distinguish : the second point is not a voronoi point.
- List of all vertices
- List of which vertices are voronoi (not the second point of the infinite beam)

Required Procedures.

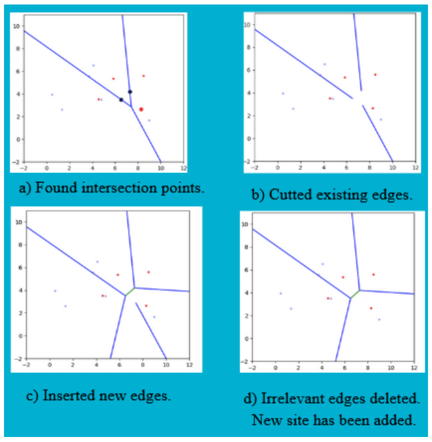


Figure 8. Procedures results

- Find Valid Intersections:

With a given new site P' let's call the middle perpendicular between P1 and P' MP1, between P2 and P': MP2. So if there are new voronoi edges that intersect E, in vertex V, the V would be an intersection of the MP1 and E. Moreover, if they intersect, the intersection point would be the same as for MP2 and E. The main steps are summarized in the scheme shown in Figure 9.

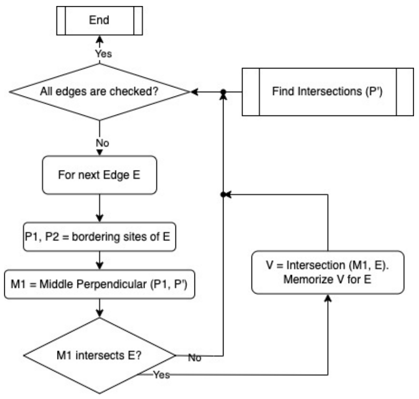

End
Yes
All edges are checked?
Find Intersections (P')
No
For next Edge E
P1, P2 = bordering sites of E
M1 = Middle Perpendicular (P1, P')
V = Intersection (M1, E). Memorize V for E
M1 intersects E?
No
Yes

Figure 9. Schema of Finding Valid Intersections.

After executing the procedure, the valid intersections are found as shown in Figure 8a.

- Cut the intersected edges.

Basically, the idea of this step is to cut the intersected edges as shown in Figure 10:

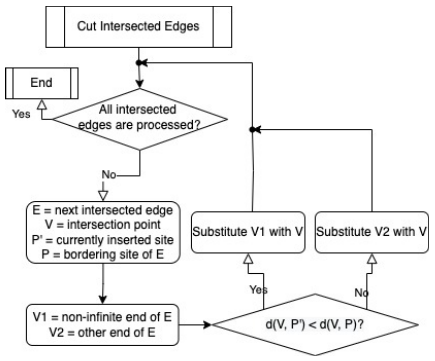


Figure 10. Schema of Cutting the Intersected Edges.

Following the presented algorithm, the result is shown in Figure 8b.

- Insert the new edges.

The idea of this step is to iterate through the affected sites. If the site's borders are intersected twice, that means that one new edge should be inserted: between the intersection points. If it is intersected once, it means that the halfline should be inserted, starting at the intersection point. The procedure is represented by the following diagram in Figure 11:

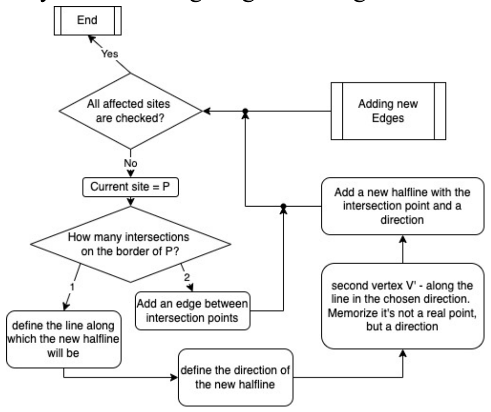

End
Yes
All affected sites are checked?
Adding new Edges
No
Current site = P
Add a new halfline with the intersection point and a direction
How many intersections on the border of P?
1
2
Add an edge between intersection points
second vertex V' - along the line in the chosen direction. Memorize it's not a real point, but a direction
define the line along which the new halfline will be
define the direction of the new halfline

Figure 11. Schema of Inserting New Edges.

The result of inserting new edges is expected to be as in Figure 8c.

- Delete the irrelevant edges.

At this point, the only thing left is to delete the irrelevant edges that are left from the previous graph.The idea is to iterate through all the affected sites. For every affected site S, it's border edges should be checked: if they are closer to the new site then to S, it is irrelevant. The process is summarized in the diagram below of Figure 12.

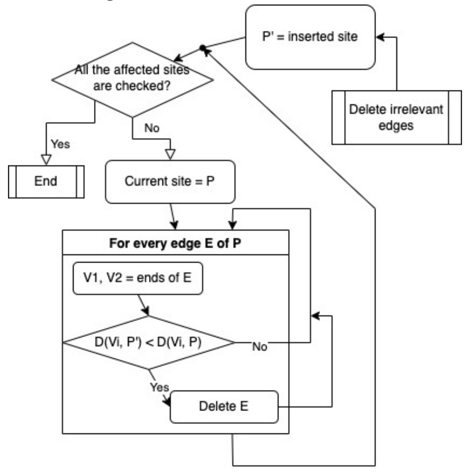


Figure 12. Schema of Deleting Irrelevant edges.

After executing the procedure above, irrelevant edges will be deleted as shown in Figure 8d.

The full implementation was structured in the class Voronoi. The class uses some additional functions for the computational geometry defined in the beginning of the voronoi module. The module is complete and can be exported as a library by the end-user. In the module, if run as a script, an example of using the diagram can be found. The class would generate a Voronoi diagram's class given a set of points. In the module there is a code provided for generating a given number of random sites (point obstacles). Also it plots the result (Figure 13), and plots the cumulative amount of time needed to build every intermediate step (Figure 3). The implementation's complexity perfectly fits the dynamic environment, better than other known algorithms.

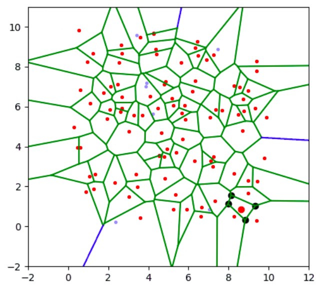


Figure 13. Built Voronoi Diagram for 200 sites (Click for demo)

D* Lite

For the present implementation, Python 3 was the programming language selected, thanks to the built in dictionaries, the implementation of the different parts of the algorithm was trivial, specially the computeShortestPart() procedure shown in Figure 5, which is very similar to a A* algorithm but focuses mainly in updating the different values of the nodes (states) of the graph.

The main procedures shown in Figure 5 of the original algorithm, were programmed as a class in a different file to import as a library and were defined as method functions to call in the main file where the motion of the robot takes place. These functions are:

- calculateKey()
- Initialize()
- computeShortestPath()
- searchpath()
- updateVertex()

The **searchPath()** procedure is mainly in charge of checking that the condition in line 10 of the pseudocode in Figure 5 is met for the search to continue.

The most challenging part in the implementation of the algorithm was to simulate dynamic environments (obstacles that were not considered initially in the map), this problem was solved by using a function called obstacle_simulation(g, current_node, hidden_obs) where g is the full graph, the current node is were the robot's current location on the graph, and hidden_obs are a set of unordered nodes in the graph.

Execution example:

Start: (2,4), Goal: (0,0), Hidden Obstacle: (2,2)

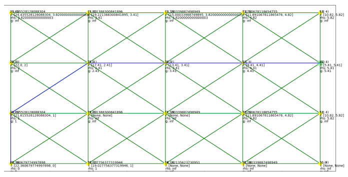


Figure 14. D* Lite Example (1 hidden obstacle) (Click for demo)

In the execution setting shown in Figure 14, an initial map is known, and an objective is set: Move robot from the initial position to the goal, however the map didn't take into account a hidden obstacle in position (2,2), so the robot had to replan in order to accomplish its objective.

Voronoi + D* Lite

Once both algorithms were programmed and tested separately, they were fused in order to get a full solution for the path planning problem as shown in Figure 15. This section will present 3 examples of the full execution of the D* Lite algorithm on a Voronoi diagram.

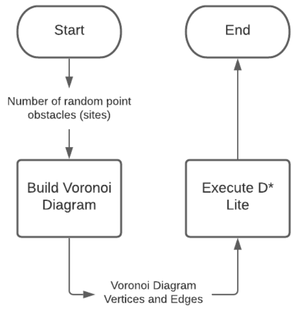

Start
End
Number of random point obstacles (sites)
Build Voronoi Diagram
Execute D* Lite
Voronoi Diagram Vertices and Edges

Figure 15. Voronoi Diagram + D* Lite procedure

Table 2. Parameters Used for Each Test					
Figure	Point Obstacles	Seed	Goal node	Start node	Hidden Obstacles nodes
16	10	53	9	1	4,12
17	20	53	0	4	16,1,18
18	50	53	75	52	65,4,83,36

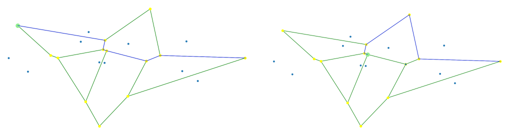


Figure 16. Initial plan (left). Replanning after an obstacle (right). (Click for demo)

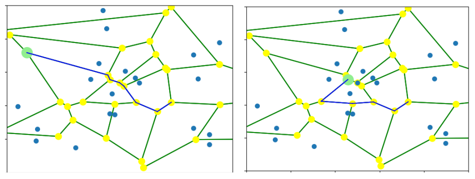


Figure 17. Initial plan (left). Replanning after an obstacle (right). (Click for demo)

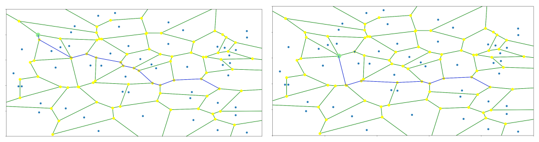


Figure 18. Initial plan (left). Replanning after an obstacle (right). (Click for demo)

III. Conclusion

The present paper described the implementation of two crucial algorithms for solving the path planning problem for mobile robots. The Voronoi Diagram by itself is a computational geometry problem that allows finding a safe topological map for a robot to navigate given a set of point obstacles (sites). On the other hand the D* Lite algorithm is a very efficient algorithm for path planning and replanning on graphs. The combination of both algorithms ensures safe navigation for mobile robots, since the Voronoi Diagram makes sure to compute a proper topological map for the robot to move without lateral collision and the D* Lite allows it to quickly replan in case the sensor on the robot detects an obstacle on a vertex of the map.

The project presented multiple challenges, specially when selecting a proper implementation for the Voronoi diagram. The literature shows a wide range of algorithms that have solved the problem in different ways, at the end the one implemented to be the most efficient for the robotics problem. The D* Lite followed the original structure of the algorithm; however, selecting a way of detecting new obstacles (simulating an obstacle detection system) was a challenge and a fundamental part of the task to show the real effectiveness of the algorithm. At the end the goal was reached, the problem was solved in a modular way, since the interaction between both algorithms works in a systematic way where one depends on the other to solve the path planning problem.

References

[1] "Comp 163: Computational Geometry", Professor Diane Souvaine, Michael Horn, Julie Weber (2004), Tufts University, Spring 2005. Link

[2] Sven Koenig and Maxim Likhachev. 2002. D*lite. In *Eighteenth national conference on Artificial intelligence*. American Association for Artificial Intelligence, USA, 476–483.

[3] 4"Algorithms for constructing Voronoi Diagram", Vera Sacristan (Slides). Link