Outdoor path finding from a map and a list of obstacles

In this session, we will simulate a path finding in a simulated outdoor environment containing a high density of obstacles of small size (tree, electric pole, road signs…). We will study and develop several ways to find the shortest path between two given points of the map, considering the following constraint: we must stay as far as possible from obstacles.

I Images Binarization

1) After downloading the file containing the outdoor map, available on *https://github.com/gaysimon/TP_Voronoi* , you will write a program that load this image and display it.

2) The loaded image has several problems that makes it difficult to automatically analyze it: lossy compression, icons and color text… Write a program to binarize this image to only keep walls on a black and white binary image. The walls have a color of (217, 208, 201) in RGB space (the color may however be slightly altered by image compression).

II Adding obstacles

1) We now add a set of random punctual obstacles on the map. Write a program to add 200 random points on the map, considering these two constraints: an obstacle can only be added in an empty space, and two obstacles must be separated by a minimum of 20 pixels. You will display these points on the map.

III Voronoï Diagram

We defined a constraint that imply moving as far as possible from obstacles. Thus, when we will move between two obstacles, we must move in the middle of these obstacles.

In order to define possible paths, we will construct a *Voronoï diagram.* This diagram segments a surface containing points (called 'seeds') into separated cells, each cell containing a seed and all points that are closer to this seed than to the others.

Borders between cells will correspond to points of space that are equidistant between the two closest seeds. These borders are thus good candidates to define our paths.

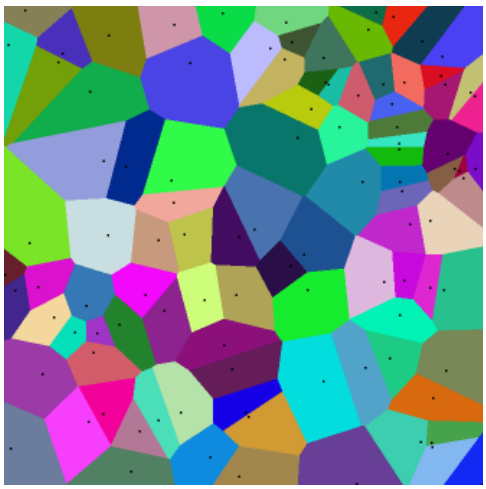

Figure 1 : a Voronoï diagram: the diagram divides space into cells, each cell containing a 'seed' (black points) and all points of spaces that are closer to this seed than to the others.

1) Write a program that constructs a Voronoï diagram from the set of points previously defined.

2) Write a program that detects paths and nodes of your diagram:
- a pixel will be considered as a path if a neighboring pixel is a part of another cell.
- a pixel will be considered as a node if it has two neighboring pixels from two different cells.
For each node that you find, you will store the id number of the cells that it connects.

You can use a region of interest of 2x2 pixels. Some examples of configuration of paths and nodes are given below:

11	12	11	12		12	11	12	12	12
22	12	12	22		13	23	33	32	34
Paths					Nodes				

3) Construct a graph from detected nodes. You can link two nodes when they are related to two common cells.

4) You now have to remove paths that cross walls indicated by the map.

To achieve this, you can exploit the path map previously defined: if there is a 'path pixel' such as this pixel is on a wall, you can get the two cells separated by this path and remove the corresponding path from the graph (as there is only one path that separate each couple of cells). Your graph is now ready to be used.

IV Application of a path finding algorithm on the graph

A path finding consists in finding the shortest path between two points. These algorithms are often used in robotics, but also in video games to move NPCs. We will use an algorithm called A* to find a path between two points on the map:

1) Implement A* algorithm:

```
Initialization :
define starting_node
define ending_node

create an empty list list
initialize each node node_i with a very high (INFINITY) value V_i
initialize each node node_i with a null predecessor P_i

set the value of starting_node to 0
add starting_node to list

while list is not empty
        node_0 ← list[0]
        for each neighbor node_i of node_0 do
                if V_0+d(node_i,node_0) < V_i then
                        V_i ← V_0+d(node_0,node_i)
                        add node_i to list
                        save node_0 as predecessor P_i of node_i
                end if
        end for
        remove list[0] from list
end while

if V_ending==INFINITY then write 'destination is unreachable'
else
        clear list
        node_0 ← ending_node
        add node_0 to list
        while node_0!=starting_node do
                add predecessor P_0 of node_0 to list
                node_0 ← P_0
        end while
end if
```

At the end of the program execution, the list contains the sequence of nodes giving the optimal path between starting_node and ending_node. Your program must display the map and the path with the color of your choice. The distance d(n1,n2) is the geometrical distance between n1 and n2.

2) We can note that after the execution of the program, each accessible node has a predecessor. It is thus possible to compute the shortest path from starting_node to any accessible node without redefining the predecessors. Modify your program to first compute predecessors of each node when selecting a starting_node, then compute the shortest path to a node when clicking on it. You can use left/right click to select starting and ending nodes.

3) The A* algorithm that you previously implemented has a major inconvenient: it is long to execute, especially on large scale graphs, which makes it unsuitable for applications that require a high reactivity. It is however possible to optimize the algorithm by adding heuristics, i.e. simplification assumptions. Here, these heuristics consists in defining a priority when exploring neighbor nodes, such as selecting the nodes that are closer to the destination, to reduce the number of node to compute. Note that in this case, the path may not be the optimal path.

The Dijkstra algorithm is a variant of the A* algorithm allowing to find the shortest path between two points of a graph, often used to compute routes (e.g. GPS routing). Its particularity is that it gives a higher priority to nodes that are closer to starting node: node are thus ordered in the list according to this distance. This heuristic increases the chances of finding the destination node, and avoid computing unnecessary nodes.
Implement the Dijkstra algorithm to compute the shortest path between two given nodes:

```
Initialization :
define starting_node
define ending_node

create an empty list list
initialize each node node_i with a very high value V_i (INFINITY)
initialize each node node_i with a null predecessor P_i (null)

set the value of starting_node to 0
add starting_node to list

while list is not empty AND node_0!=ending_node
        node_0 ← list[0]
        for each neighbor node_i of node_0 do
                if V_0+d(node_i,node_0) < V_i then
                        remove node_i from list
                        V_i ← V_0+d(node_0,node_i)
                        insert node_i in list accordind to V_i
                        save node_0 as predecessor P_i of node_i
                end if
        end for
        remove list[0] from list
end while

if (list is empty AND V_ending==INFINITY) then write 'destination is unreachable'
else
        clear list
        node_0 ← ending_node
        add node_0 to list
        while node_0!=starting_node do
                add predecessor P_0 of node_0 to list
                node_0 ← P_0
        end while
end if
```