UASLP

Universidad Autónoma
de San Luis Potosí

FACULTAD DE
INGENIERÍA

Robotics A

Programming exercise 4

ROBOTWORKS

José Luis RIVERA BLANCO

Juan Antonio FLORES REYNA

Delivery date: 7 June 2021

Professor: Alejandro GONZALEZ DE ALBA

INDEX

Path Planning

In Robotics, Path Planning is an algorithm or strategy used to find a path for a robot on a map in which we can have (or not) some obstacles.

There exist several ways for solving those paths, but all of them share one principle: starting from a starting point, computing until finding the desired point.

The abstraction of the map depends on the chosen strategy since it can be represented by infinite or finite values. For this work we need to discretize the map into nodes, in a way of representing a finite number of coordinates on a map.

Given path

The objective of this 4th programming exercise is to find a suitable strategy for solving the map shown below.

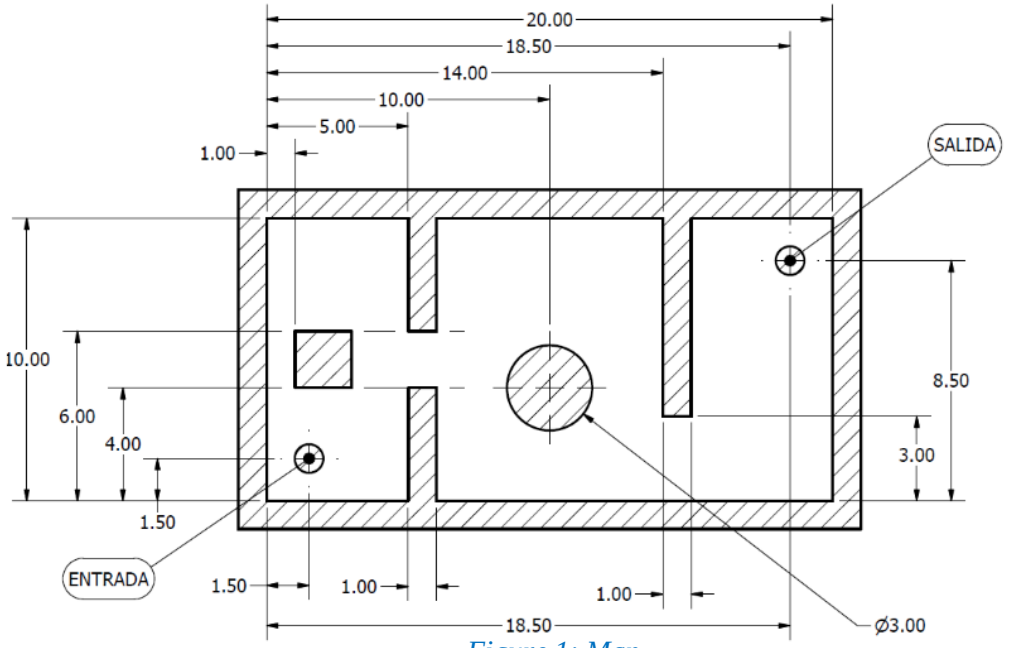
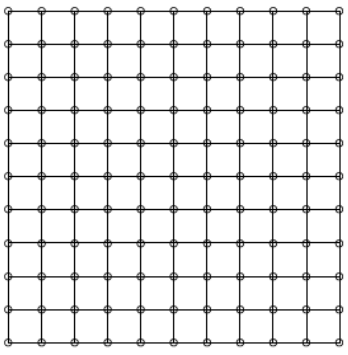


Figure 1: Map

As mentioned before, we are going to represent this map with a finite number of coordinates. It can be seen as a grid where every node represents a coordinate and is linked to the surrounding nodes.

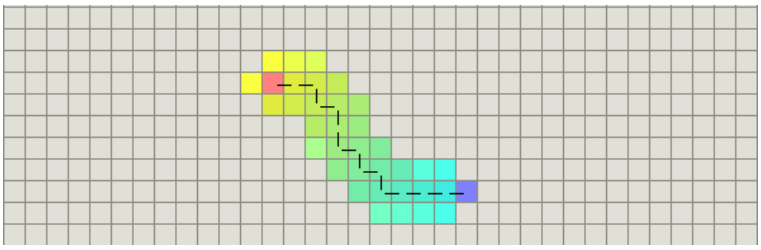


The precision of the strategy will depend on the distance between two nodes.

The obstacles will represent the nodes that are not accessible on the grid.

A*

A* is the most popular choice for pathfinding, the algorithm builds on the principles of Dijkstra's shortest path algorithm to provide a faster solution. It achieves this by introducing a heuristic element to help decide the next node to consider.



Principle

The trajectory of the strategy is given by the following expression:

$$f(n) = G \cdot g(n) + H \cdot h(n)$$

Where $g(n)$ represents the distance between the starting point and the proving node (coordinates given by "n"). The heuristic term $h(n)$ represents the distance between the same proving node "n" and the goal.

The constants G and H are weightings which induce a different behavior of the algorithm depending on the preference of one term over another.

Example

Let us consider a simple map where "A" is the starting point and "B" the goal. The black nodes are the obstacles and the white ones those whose $f(n)$ cost has not been calculated.

The A* algorithm calculates iteratively the cost of the surrounding nodes from a reference (starting at the starting point). In the diagram, the $f(n)$ cost of the green nodes has been calculated. At the left we find the $G \cdot g(n)$ cost and at the right the $H \cdot h(n)$.

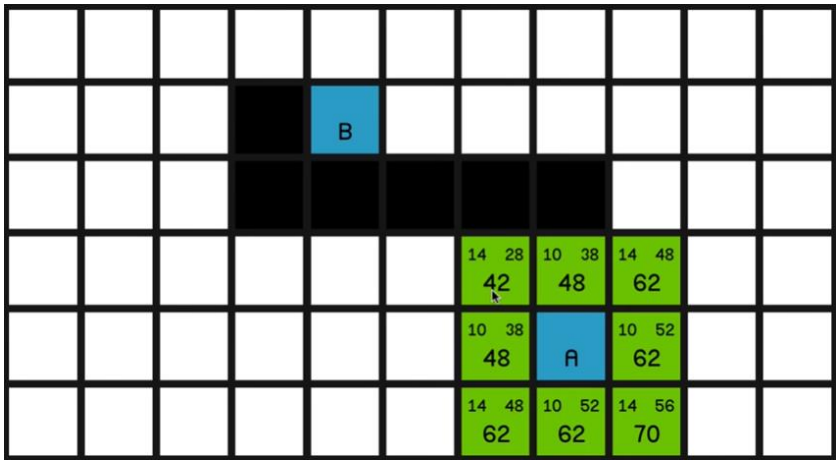


Figure 2: Example (a)

Once we have all the surrounding costs, we select the node with the lowest $f(n)$ cost as a reference and repeat the cycle until having the goal as a reference.

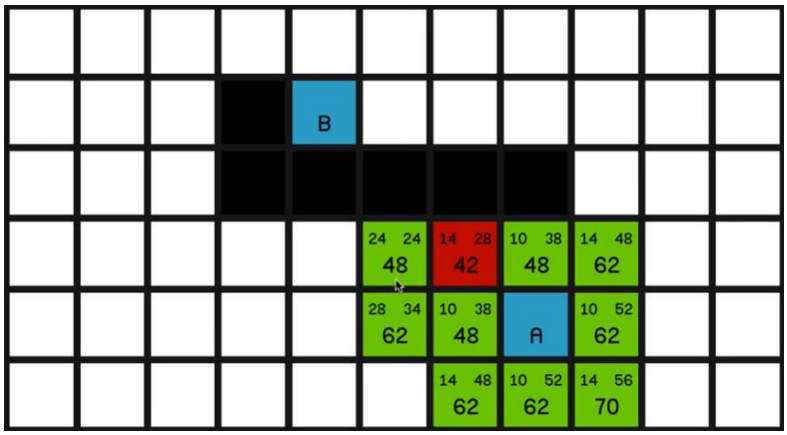


Figure 3: Example (b)

Note that we do not calculate the cost of an obstacle, and it is important to recalculate the cost of the surrounding nodes if the new calculated cost is lower than the previous cost.

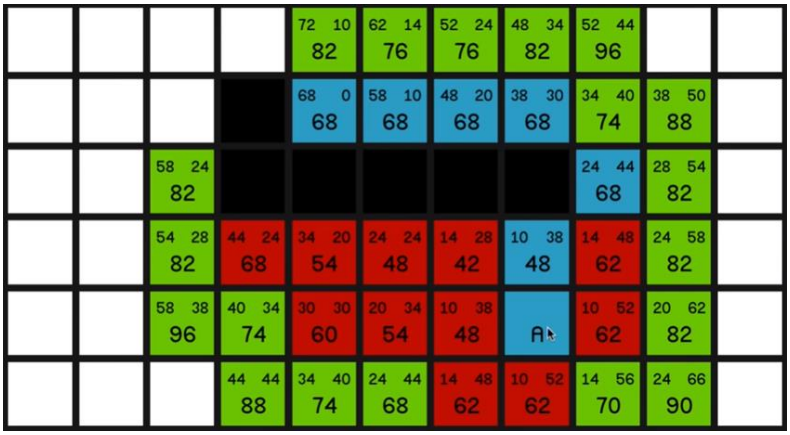


Figure 4: Example (c)

All the nodes considered as the potential minimum value are represented on red in the diagram. However, following this red path is not convenable.

To solve this problem, we need to define a property of each node that tell us which node is his father, or saying it in another way, with respect to which node it has been calculated.

Here is the importance of refreshing the values and the father when we change the reference (if cost is lower).

This will allow us to find the blue path in the diagram (solution).

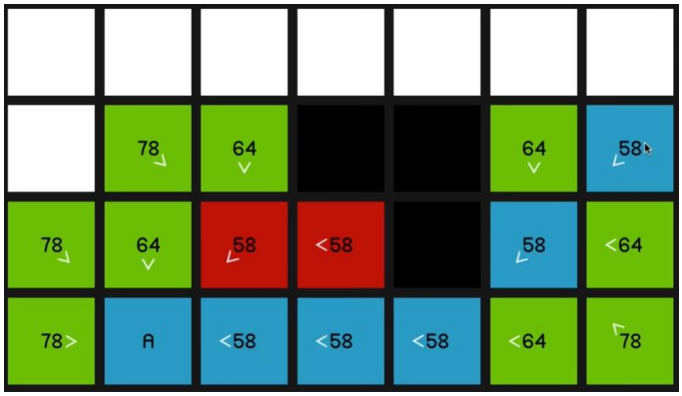


Figure 5: Example (d)

Representation in our programming code

Having introduced the A* algorithm we define our abstraction:

a) *To represent the nodes, we use a matrix whose dimension depends on the desired precision/resolution.*

That means, if the selected precision/resolution is equal to "1" we will only be able to move "1" unit horizontally and vertically. And the matrix's dimension is the same as the map's dimension (12x22).

If the selected precision is "0.5", we can move "0.5" units and the matrix's dimension is the double (24x44). And so on.

b) *Each element of the matrix is a structure that contains the following information:*
   - **lock**: "1" if the node is an obstacle, "0" otherwise.
   - **g**: $g(n)$ cost of the node.
   - **h**: $h(n)$ cost of the node.
   - **father**: coordinates of the father (x, y).

We use a plot function to represent this matrix visually. For a resolution of 0.5 we have:

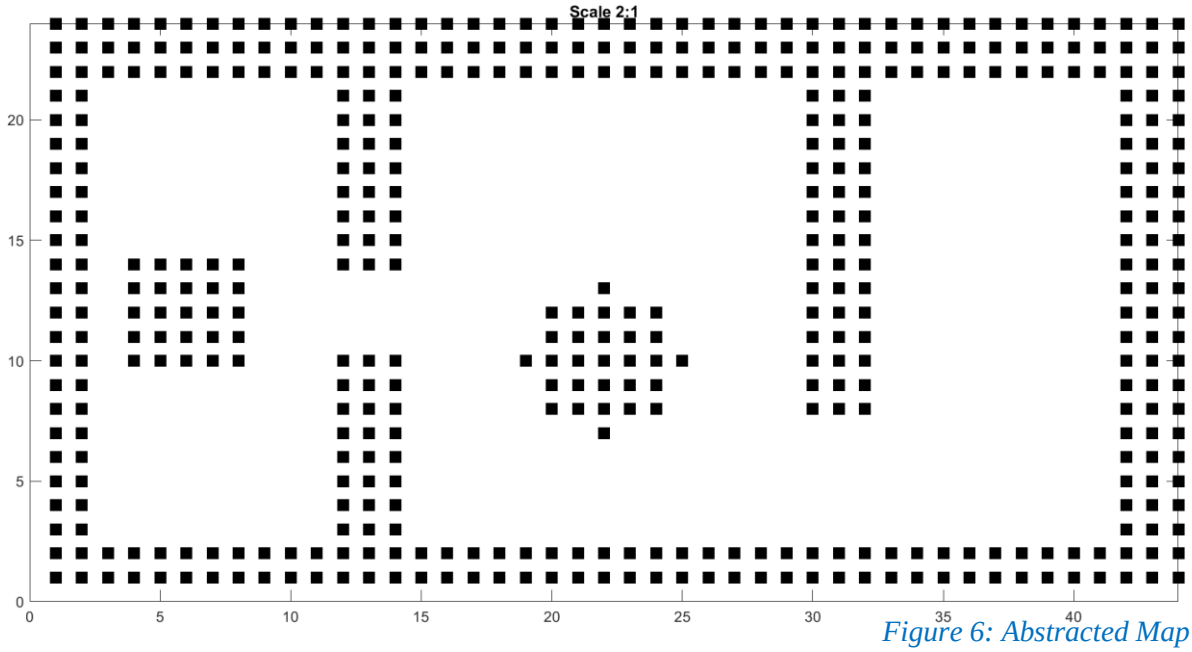


Figure 6: Abstracted Map

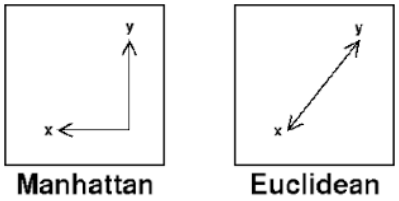
The little black squares are the nodes with property "lock=1" (obstacles).

The lower the resolution, the similar the map is (less discrete).

Considerations

To calculate distances, we use different principles:

- For the $g(n)$ function we use the Euclidean distance.
- For the $h(n)$ function we use the Manhattan distance.



The initial value of $g(n)$ and $h(n)$ for all nodes is "∞" except the initial node, there we should have $g(n) = h(n) = 0$.

Pseudocode

The following pseudocode has been used to implement the A* algorithm:

```
Create the OPEN list          --------- I
Create the CLOSED list        --------- II
Add the start node to OPEN    --------- III

Loop
        Current = node in OPEN with the lowest f_cost     --------- IV
        Remove Current from OPEN                          --------- V
        Add Current to CLOSED                             --------- VI

        If Current is the goal                            --------- VII
                Return

        Foreach Neighbour of Current                                  --------- VIII
                If Neighbour is an obstacle OR Neighbour is in CLOSED --------- IX
                        Skip to the next Neighbour
                If New_cost of Neigbour is lower OR Neighbour is not in OPEN --- X
                        Set f_cost of Neighbour
                        Set father of Neighbour to Current
                        If Neighbour is not in OPEN
                                Add Neighbour to OPEN
```

The Roman Numbers have been used to make the MATLAB code easier to read. Each part of the MATLAB code represents a number on this pseudocode.

Created MATLAB scripts

The MATLAB programming code can be found in the script "`findPath.m`"

```
function [Path, CLOSED, OPEN, Time, Cost] = findPath(Map,start,goal,Ponderation)
```

This function takes the initialized map, the start and goal point and the desired ponderation [G H].

The output is the solution path, the CLOSED list, the OPEN list, the time of computation and the $g(goal)$ cost without ponderation.

Furthermore, it validates the input arguments to guarantee a correct solution.

To test the complete program, execute "`A_star.m`"

Here we use the following initialization and plot functions:

- `function Map = initializeMap()`
- `function plotMap(Map)`
- `function plotPath(Path,Traveled,Evaluated)`

Results

In this section we are going to use first a [1 1] weight for $g(n)$ and $h(n)$. Then we show the effect of using different weights.

Working examples for different paths

The objective mentioned at the beginning of the document has been successfully achieved.

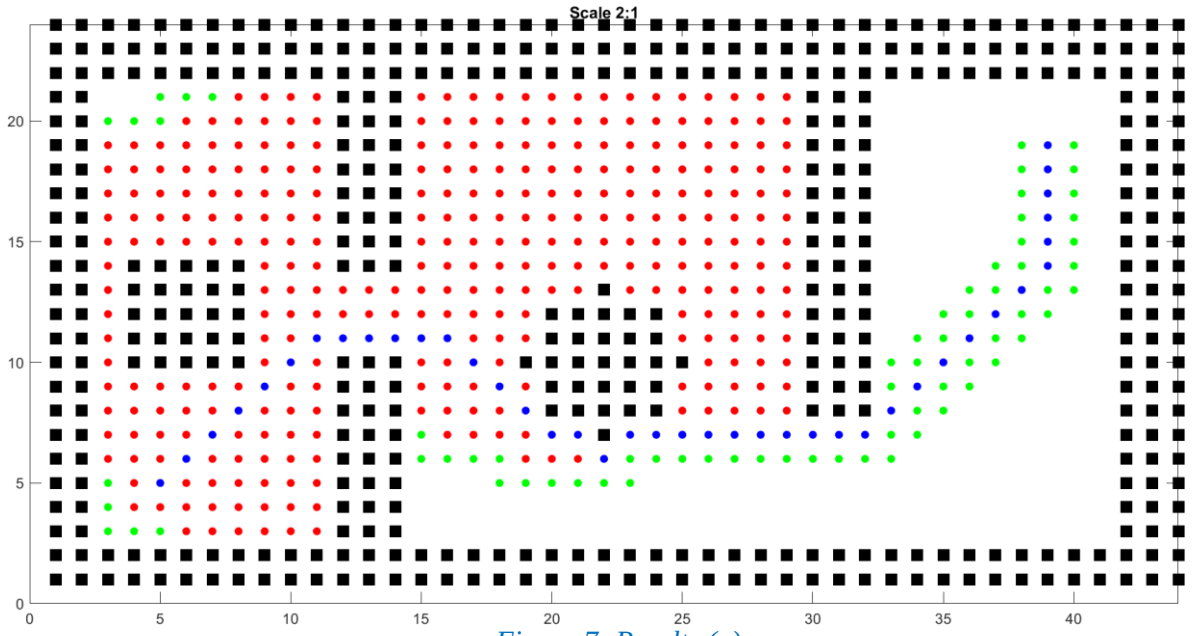


Figure 7: Results (a)

The colors of the circles represent the same as the example principle. The red ones are the supposed nodes with the minimum $f(n)$ cost; The green ones, the evaluated neighbours; and on blue, the solution path.

Trying others initial/goal points we obtain:

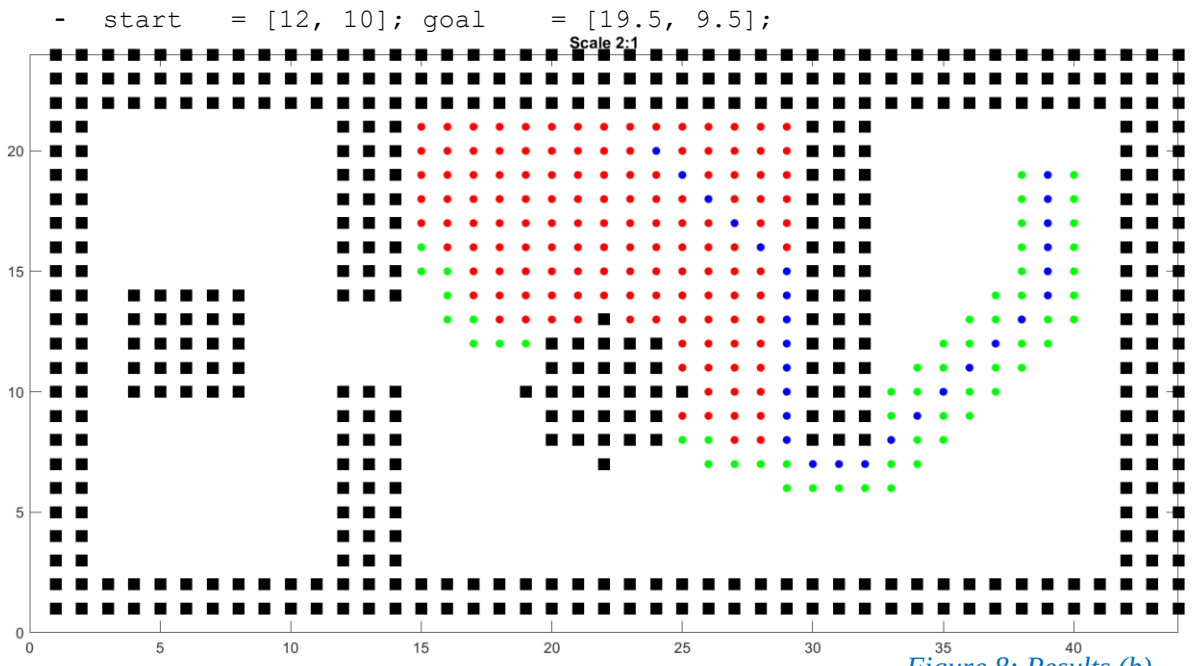


Figure 8: Results (b)

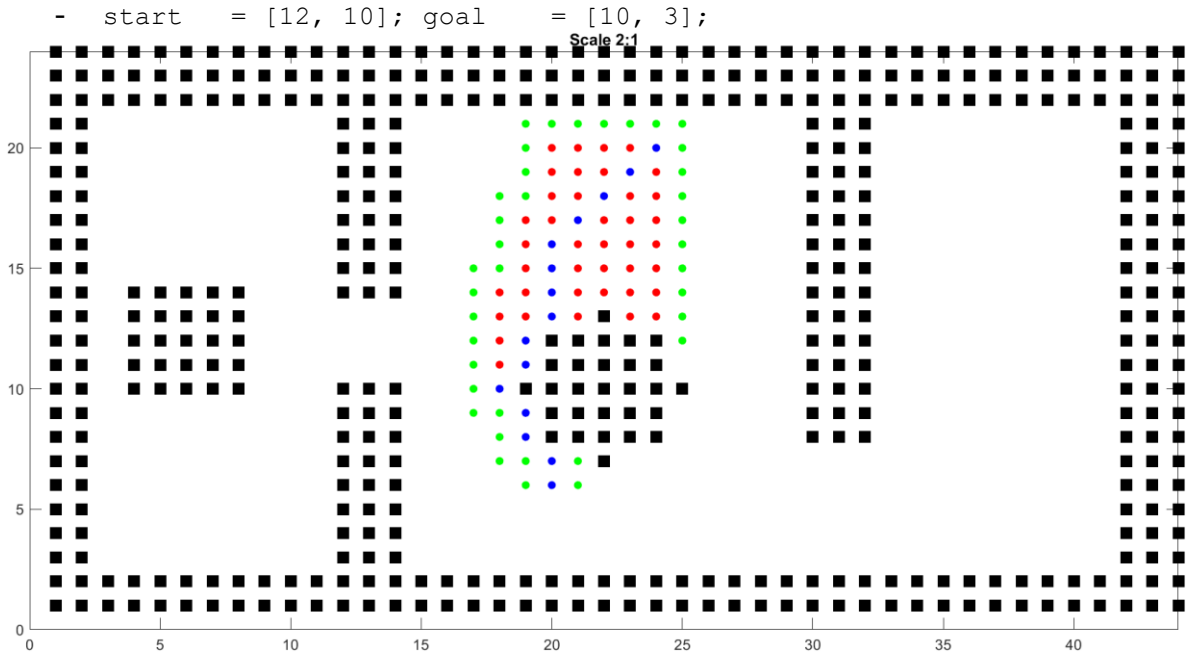


Figure 9: Results (c)

Role of weighting (Time/Cost)

The weighting of the $f(n)$ cost function will determine whether the algorithm follows a Dijkstra's behavior or a A* solution.

When the weight of $h(n)$ is greater than $g(n)$ we decrease the number of nodes evaluated, so the algorithm will be faster, but the path longer (Figure 10a, A*).

In the other hand, when $g(n)$ is greater than $h(n)$, or $h(n)$ does not exist, the algorithm will increase considerably the number of nodes evaluated. Nevertheless, we can guarantee the lowest cost (Figure 10b, Dijkstra).

Graphically:

*Figure 10a: A**

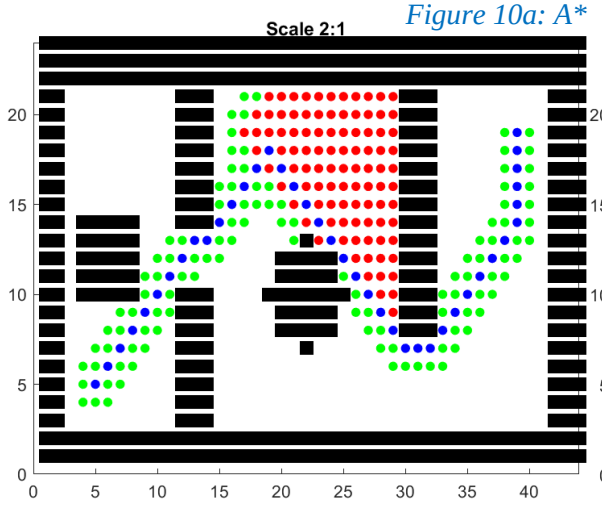
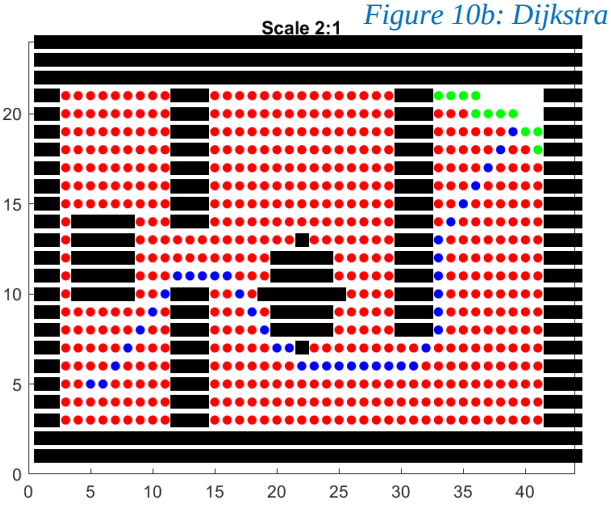


Figure 10b: Dijkstra



The "`A_star.m`" script contains a test in which we evaluate different ponderations. Our objective is to see the evolution of cost and time.

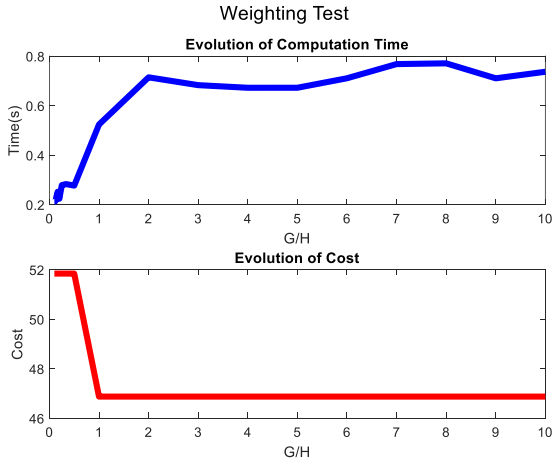


Figure 11: Weighting Test

The "x" axis represents the rate G/H, and we can verify the increase on time when the impact of $g(n)$ is more important. At the same time, we get the lowest cost. When $h(n)$ is greater (G/H<1) the algorithm runs faster with a higher cost.

Note that for a certain combination the costs converge.

References

Theory:

https://theory.stanford.edu/~amitp/GameProgramming/

https://isaaccomputerscience.org/concepts/dsa_search_a_star

Animated example:

https://www.youtube.com/watch?v=-L-WgKMFuhE&list=LL&index=6

Working examples:

Experimental Comparison of A* and D* Lite Path Planning Algorithms for Differential Drive Automated Guided Vehicle, Pandu Sandi Pratama, Pusan National University.

Pseudocode:

https://theory.stanford.edu/~amitp/GameProgramming/ImplementationNotes.html