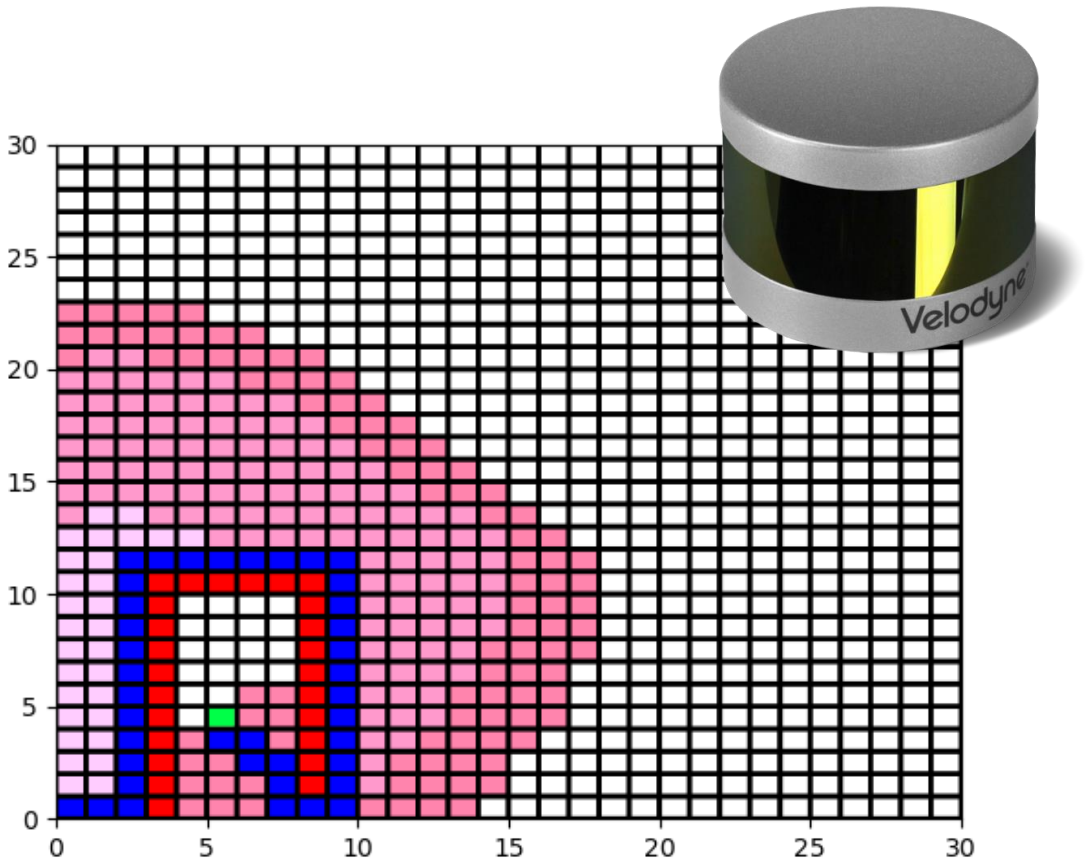
Dijkstra's & A* comparison for visual navigation

Daniel Casado Herráez

14th January 2021



Symbols

A	Unvisited set of nodes
B	Visited set of nodes
$d(v)$	Value of node v, also called tentative distance in Dijkstra's algorithm
e_i	Link, or edge i
$G(V, E)$	Graph of vertices V and edges E
$O(n)$	Number of operations of order n
v_i	Node i

Contents

Chapter 1

Introduction

Visual navigation is an essential task that must be resolved by modern mobile robots in order to be able to move around in their environment, specially in indoor situations in which there is no GPS or other satellite based navigation possibilities. This project will overview the two main steps of visual navigation so that the system can move from a source position to a target avoiding obstacles on its way. Chapter 2 will describe the subject dividing into two stages: exploration and path planning. The former will outline the main sensors used in robots for this type of navigation, while the latter will evaluate two possible approaches of path planning to move to a goal from an initial position. A more detailed view of this will be presented in 3, with the mathematical explanation, algorithm pseudocode, example procedure, and computer program for each of them. Due to the fact that the visual navigation problem might range from the data acquisition to the decision making step, the scope of the computer program will be limited to cover the two possible path planning algorithms, A* and Dijkstra's, including a comparison of them and their results. Finally Chapter 4 will reference the key learnings during the project, as well as future areas of research.

Chapter 2

Subject Description

Indoor visual navigation techniques are needed for a robot to move either in a known or unknown environment. The problem of an autonomous ground robot moving from a starting position and reaching a goal will be studied in this project. The task will be divided into two sub-problems:

- Exploration: Mapless navigation methods will be presented. However, map-based navigation methods will be studied in more detail. This requires having a computed map of the environment. The exploration sensors to build this new map will be analyzed in this stage.
- Path generation: The path going from the starting point to the goal needs to be computed. The most popular algorithms will be presented, and one will be selected in order to perform the computer program evaluation.

2.1 Exploration

One of the first stages of robot navigation is deciding on its dependency on a map. As discussed by [1], visual navigation can be divided into two main categories: mapless, or reactive navigation, which will be briefly introduced, and map-based navigation.

Reactive navigation is based on acquiring visual data without a predefined model of the environment. The main idea is that the robot will be able to move around based on the current data that is being collected. Mapless navigation methods include:

- Optical-flow based approaches: motion difference between pictures is being measured.
- Appearance based navigation: a pretraining stage will adapt the learning model to similar features to those in the environment, and a operational stage that matches these environment descriptors with the ones detected by the sensors

Regarding map based navigation, it is based on defining a model of the environment using sensor data in order to move from a starting point to the target. They

will be divided into metric map-using systems and topological map-using systems. In the following section, common sensors for robot navigation will be studied. This will be followed by the analysis of metric map and topological map navigation approaches.

2.1.1 Sensors

The sensors in map based navigation will be used in order to build the model of the environment and match their data with the existing map. According to [2], based on biological sensing, two main sources of information can be identified, idiothetic, and allothetic sources. The former are related to localizing the robot itself. They are usually dead-reckoning sensors such as odometers and Inertial Measurement Units (IMU). The latter refers to sensing the environment during the operation of the robot either to interact with it or to build a map using visual sensors such as radar, cameras or range-finders.

2.1.1.1 Idiothetic Sensors

Idiothetic information sources refer to those involving speed, acceleration, or wheel rotation. They are referring directly to the robot itself, without taking into consideration its interaction with the environment. As a result of this dead-reckoning measurements, cumulative error will be generated. These sensors are mainly odometers and inertial measurement sensors. These sensors will be reviewed as even though these are not visual-based, they will improve the navigation of the mobile robot inside the visually generated map.

2.1.1.1.1 Odometers

Based on the explanations by [3] odometry data is based on the integration of incremental motion data, therefore odometry encoders are considered dead-reckoning, which will lead to accumulated error over time. The working principle is to measure the angle of rotation of the wheel of the robot since the beginning of the movement, and with this data, attempt to calculate the position of the robot in space. Most common cases of these sensors are optical encoders, that based on a light ray it will detect the position of the wheel in absolute encoders, or count the times the light ray has been interrupted. The values are then transformed into binary so that they can be used by the computer.

2.1.1.1.2 Inertial Measurement Units

The IMUs sensors integrate a combination of accelerometers and gyroscopes and measure the rotation and acceleration of the robot. A first integration of the values will return the velocities for the accelerometer and the position for the gyroscope. A second integral should output the position from the accelerometers. It is of paramount importance to take into consideration the error generated in this procedure. Due to the fact that it does not need any external data apart from the movement of the vehicle, it is useful in combination with odometry to estimate motion and when other sensors are not available. On the negative side, similar to encoder data, IMUs will provide dead-reckoning measurements that can lead to unbounded accumulated error.

2.1.1.2 Allothetic visual Sensors

In the same way idiothetic sensors refer to the vehicle itself, allothetic sensors will capture data from the environment in order to generate the map and interact with it.

2.1.1.2.1 Camera

Camera sensors provide information of the environment detailed enough to be able to identify people, objects, and other obstacles during the navigation of the robot. The camera will mainly be used as a tool for reactive navigation, in which the data is being processed live, and the robot will either match the environment with a precomputed model or detect new obstacles. They are usually implemented with image processing techniques and convolutional neural networks, [4]. Cameras are also used in pairs to measure the distance to the object with stereo vision techniques, based on the distance between them, their properties, and the coordinate points in the image pair.

2.1.1.2.2 LIDAR

Another well-known sensor for autonomous navigation is the Light Detection and Ranging system (LIDAR). The device computes the distance to the objects in the environment measuring the time difference between an emitted light signal and the received reflected wave. The output will be a point cloud of the environment with all the distance information in it. In order to make this data useful, preprocessing of the point cloud should be applied. First the ground and noise should be removed, Himmelsbach ground plane removal algorithm, [5], can be used for this task. Then, to align the point clouds at different timestamps, the Iterative Closest Point algorithm (ICP) is usually computed.

2.1.1.2.3 RADAR and SONAR

Radio Detection and Ranging sensors provide measurements based on the Doppler effect, calculating the apparent change in frequency of an electromagnetic wave from the source with respect to the observer. When applied to robot mobile navigation, the signal emitting source will be the robot instead of the control station, and the reflections of the wave will come from the obstacles. These devices are mostly used in bigger autonomous platforms such as driverless cars as they are not greatly influenced by weather conditions, light variations, and they can provide long range mesaurements. [6] mentions these main advantages, as well as the classification into three main categories, Long Range Radars (LRR), Short Range Radars (SRR) and side-blind zone radars (SBZR). Their performance is based on their field of view (FOV), the larger this value, the shorter the range will be. However, some disadvantages of these devices are also mentioned in the report referring to signal interference, such as self-interference, cross interference from other radars on the same vehicle, and interference from other vehicles' radars. Regarding the SONAR, or Sound Navigation and Ranging sensors, they are similar to RADAR and LIDAR, but instead of being light-based or electro magnetic wave-based they are ultrasonic wave-based. The time is measured between the emitted wave and the reflected wave to calculate distances to the objects and generate data from a map. Early applications of SONAR sensors include underwater navigation and mapping, however researchers have also implemented them for mobile robot navigation, [7].

2.1.2 Map based navigation

Explanations of map based navigation have been studied from [8] and [2]. Two main categories of map can be identified. Metric maps, which are data-based, and topological maps, that are graph-based.

2.1.2.1 Metric maps

Metric maps for robot navigation are based on a coordinate representation of the space. Metric models of the environment are constructed based on accurate metric data. Sensors models will enable the combination of allothetic and idiothetic data to capture data from the objects in the environment. Once the robot has the map it can navigate matching real time acquired data with its database data. The main advantage of using this map representation is that it leads to accurate data, that is easily understandable by humans. The main tradeoff for this accuracy is the computational cost, as well as the need of having good sensor models that consider the properties of the environment. Furthermore, the quality of idiothetic data is key for metric map building, hence drift being a major issue in this method.

There are two main techniques for building metric maps, based on feature extraction, or based on using free-space representation. The first one relies on sensor data

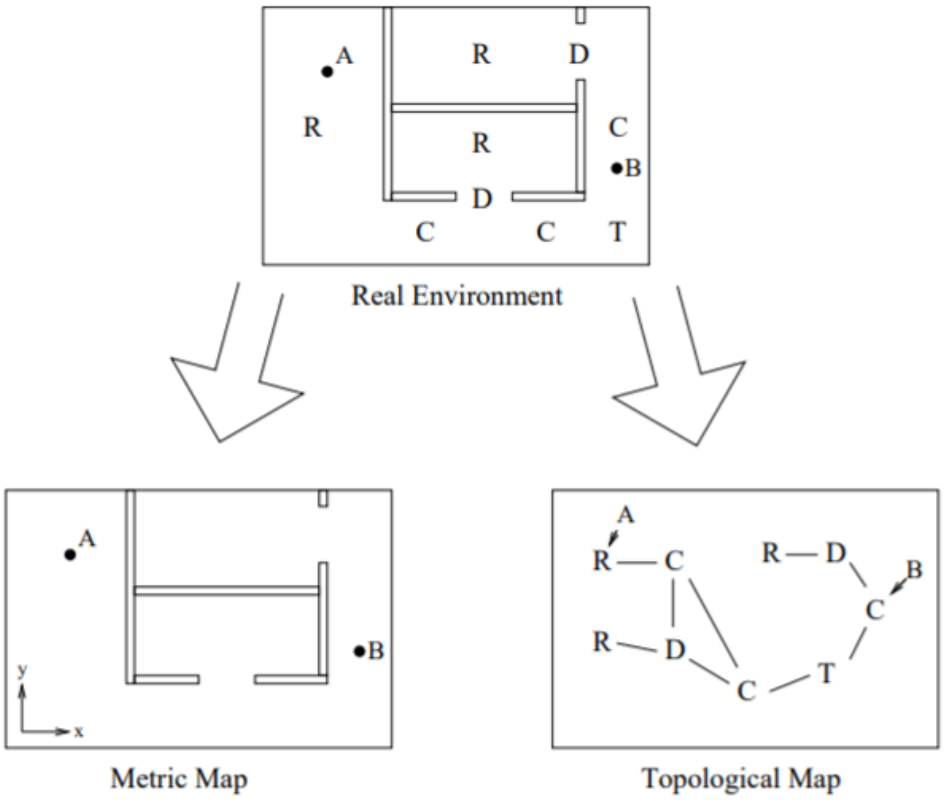


FIGURE 2.1: Map based navigation types, [2]

for identifying features from the objects and setting them in the map to be avoided by the robot. The second one is more popular, and instead of considering the objects in the environment, it uses the empty space where the robot can navigate. The most popular way of empty space representation is with occupancy grids described by [9], which do not require feature extraction from the objects. Its main idea is to discretize the space into regular boxes that correspond to a probability of being occupied by the robot, and that state whether there is an obstacle in it. This heuristic approach solves the problem of map generation using noisy sensor data.

2.1.2.2 Topological maps

The alternative to metric map representation of the environment are topological maps. They are built by nodes and edges. Each node corresponds to a characteristic feature or zone of the environment defined by allothetic information, and can be associated with an action, such as turning, crossing a door, stopping, or going straight ahead. The edges will contain idiothetic data. On the one hand, topological maps are simple and compact, take up less computer memory, and consequently speed up computational navigation processes. They also don't need precise sensor models. Topological representation is an alternative to occupancy grids, that can be inefficient in complex environments. On the other hand, building a topological map

needs a more exhaustive feature description and might lead to navigation errors between environments that look very similar.

2.2 Path Generation

After capturing the data from the sensors and deciding which kind of map will use the robot to navigate in the environment, in order to move from the starting point to the goal a path must be generated avoiding the obstacles. These can be of two types, static and dynamic. The former will be included in the map information, while the robot's response to the latter will be reactive. Only static obstacles for topological maps are in the scope of this project. The possible methods for path planning in visual navigation will be studied in the following sections with help of the computer program.

Chapter 3

Computer Program in Python

3.1 Problem description and assumptions

The subject of visual navigation has been studied, and divided into the components of obtaining the data using sensors, generating the map, and navigating through it using path planning. The computer program will work on the problem of obtaining a feasible path from the origin to the target avoiding obstacles obtained from visual sensors. This will be done using the topological map implementation.

The main assumption is that the sensors have been able to retrieve the information relative to the objects, generating a map with obstacles. This map is assumed to be a grid with empty and blocked space, and with the same path cost to move from one cell to another. Moreover, movement will be limited to vertical and horizontal. Diagonal steps will be not allowed in any of the experiments, with exception of 3.4.4.

The goal of the computer program will be to arrive to the solution of leading the robot from the source to the target through a feasible path along the free space in the map. For that, the Dijkstra's and A* algorithm will be used, tested and compared in two different cases, for small maps, and for bigger maps.

3.2 Algorithm

In this section the proposed solution to the problem is presented, using two well-known algorithms in the field of path planning, A* and Dijkstra. Their mathematical explanation will be presented, along with the algorithm pseudocode, and a useful example demonstrating the working principle.

3.2.1 Dijkstra's Algorithm

Dijkstra's shortest path algorithm is one of the most basic path planning algorithms. Developed by Edsger Dijkstra in 1956, the main idea is to find the shortest path between two nodes in a graph based on the weight of the edges. First, the mathematical explanation will be reviewed, followed by the algorithm pseudocode 1 and a representative example.

3.2.1.1 Mathematical explanation

Given a weighted graph $G(V, E)$ where V is a set of nodes $\{v_1, v_2, ..., v_n\}$ and E is a set of links $\{e_1, e_2, ..., e_n\}$. The cost between two nodes will be $w(u, v)$ where u and v belong to the node set. The shortest path between the starting s point to the goal v is denoted as the tentative distance, $d(v)$. That will be the value of the distance assigned to each of the nodes. Dijkstra's Algorithm time complexity is defined by $O(|E| + |V| * log(|V|))$ where E is the number of edges (links between nodes) and V is the amount of nodes.

Before the start of the algorithm, two sets are created. A will contain all unvisited nodes, and B will contain all nodes that have already been visited, and a reference to their previous node.

Initially, all the node values are set a tentative distance value of ∞ except the initial one, that will be assigned a value of $d(s) = 0$. This first one is set as current node.

The distances (link weights) to the neighboring unvisited nodes are computed. The minimum distance between their corresponding value and their previous value (∞ in this case) is assigned to them, they are referenced to their previous node (s in this case), and they are included in the B set. After this, the node with the smallest distance is visited and removed from the unvisited set A.

For this second node, the distances (link weights) to the neighboring unvisited nodes are computed. The minimum distance between their corresponding value and their previous value is assigned to them, they are referenced to their previous node (this second node), and they are included in the B set. After this, the node with the smallest distance is visited and removed from the unvisited set A.

Process is repeated until all of the nodes from the set V have been visited or the goal is reached. Once this condition is fulfilled, the target node will already have been reached and it will be contained in set B.

Its reference to the previous node can be extracted, and the solution will be back-propagated until reaching the source node s.

As it has been observed, the algorithm goes through each of the neighboring nodes until finding a solution, without considering whether it is getting closer or farther away from the goal. This leads to inefficiencies when the graph $G(V, E)$ is composed of many nodes and edges. This problem is addressed by an improved algorithm called A*.

Algorithm 1 : Dijkstra's algorithm

Initialization:

$G(V, E)$ - Graph that will be studied

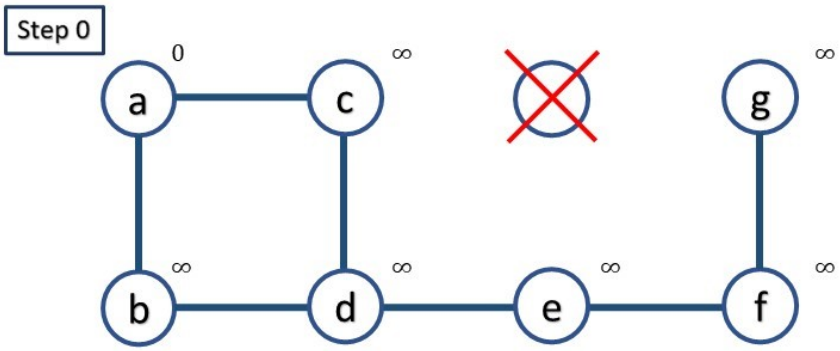
A - Set containing all the nodes

B - Empty set of nodes

1: **for** `each node in` V **do**
2: `distance(V) =` ∞`;`
3: **end for**
4: Set $d(s) = 0$, add s to B and set s as current node.
5: **while** A is not empty, or goal is not reached **do** ▷ Check all the nodes
6: $v_n \leftarrow$ Is the current node with value val_n
7: $N, \{n_1, n_2..., n_m\} \leftarrow$ Is the set of neighbor nodes of v_n
8: **for** `each node in` N **do**
9: $d(n_m) = min\{currentvalue, val_n + w\}$ Compute for all neighboring nodes.
10: **end for**
11: Set the node in N with smallest $d(n_m)$ as v_n, reference it to v_{n-1} and add it to B
12: **end while**

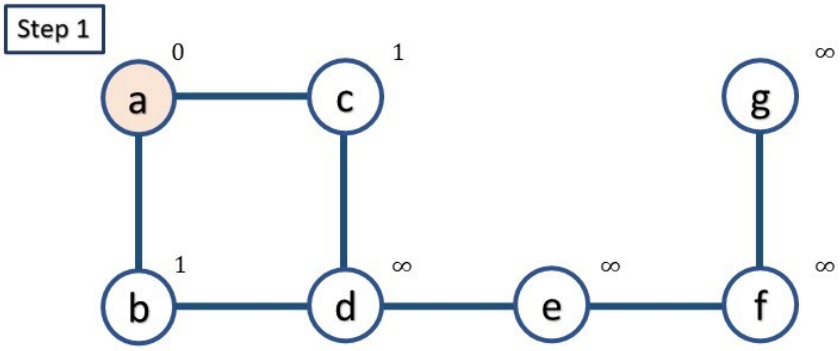
3.2.1.2 Dijkstra's Algorithm Example

The following part will include a simple practical example of how the computer program will resolve the shortest path problem using the Dijkstra's algorithm.



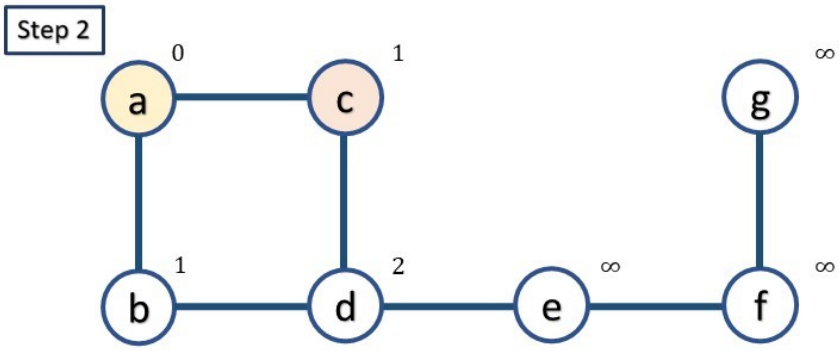
$$A = \{a(0), b(\infty), c(\infty), d(\infty), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{\}$$

FIGURE 3.1: Step 0: Firstly, the graph is built, ignoring the selected obstacle nodes so that there are no links between a free node and an osbstacle one. The initial node values is ∞ for all except the source node. The unvisited and visited sets, A and B, are initialized.



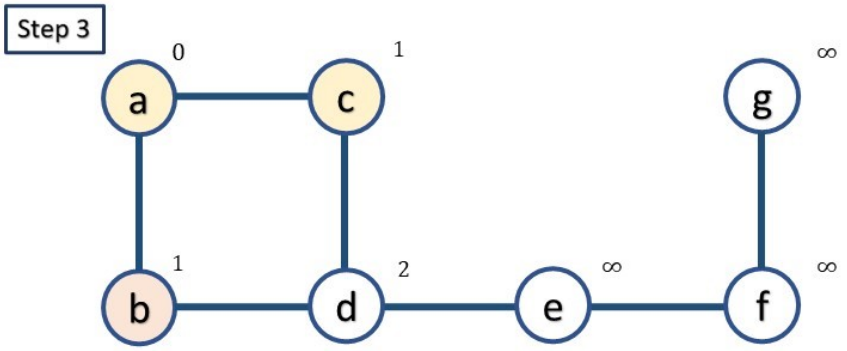
$$A = \{c(\min(1, \infty)), b(\min(1, \infty)), d(\infty), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{a(0)\}$$

FIGURE 3.2: Step 1: The source node a is selected (orange), removed from the unvisited set A, and added to the visited set B. It's value is a zero constant. The neighboring node values are also computed by calculating the minimum between the previous node value, and the addition of the previous node value plus the edge value. This value is updated in the unvisited set A, and the set is sorted from lowest value to the highest.



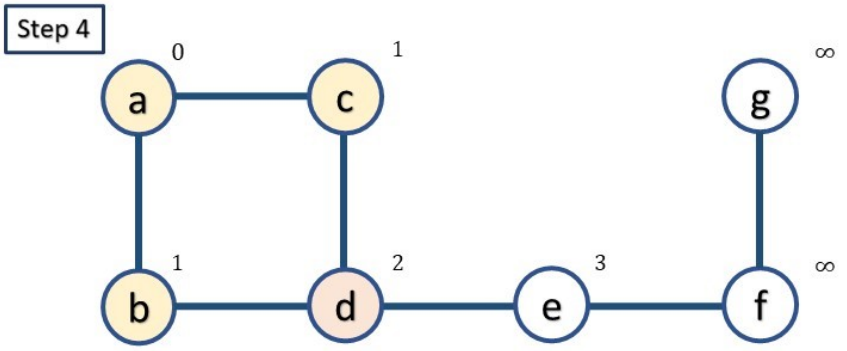
$$A = \{b(1), d(\min(2, \infty)), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{a(0), c(1)_a\}$$

FIGURE 3.3: Step 2: The node with smallest value from A will be selected now. As both nodes b and c have the same value, one is randomly selected, c, and added to the visited set B being linked to its predecessor node. Then c is removed from the unvisited set A. The neighboring node values are computed.



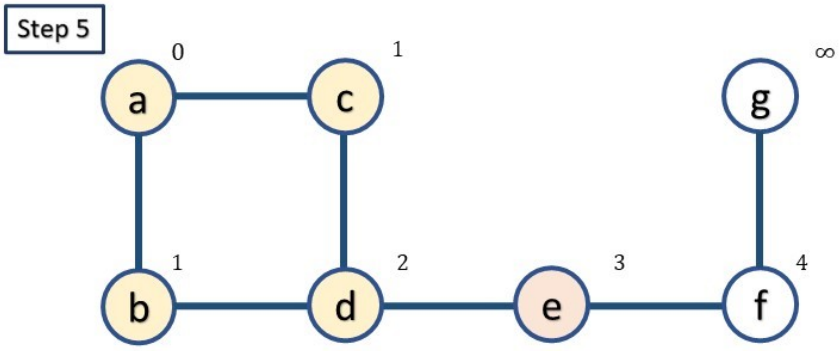
$$A = \{d(1), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{a(0), c(1)_a, b(\min(1, 2+1))\}$$

FIGURE 3.4: Step 3: The next node in the A set will be b. It is possible to get to b directly from a or going through c and d, so the path with least cost is selected. Moreover, it can be observed that the value of node d is computed from two different nodes , the minimum of both should be selected. However in this graph the value is the same and it will have no influence in the cost of the path.



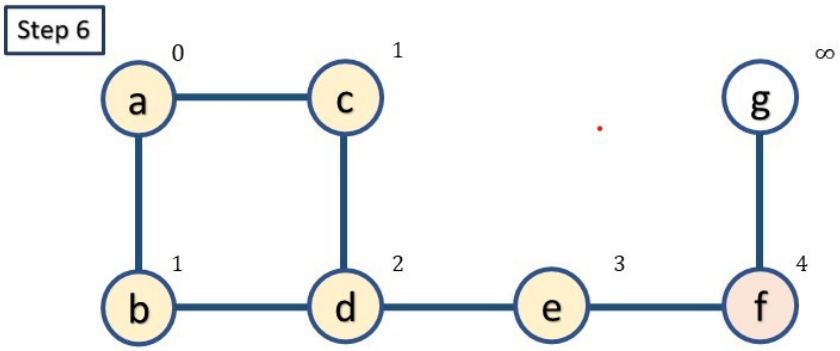
$$A = \{e(\min(3, \infty)), f(\infty), g(\infty)\}$$
$$B = \{a(0), c(1)_a, b(1)_a, d(\min(2,2))\}$$

FIGURE 3.5: Step 4: The next node in A will be d, and the previous procedures are repeated, which involve the computation of the value of its neighbor e, removing d from A and adding it to B.



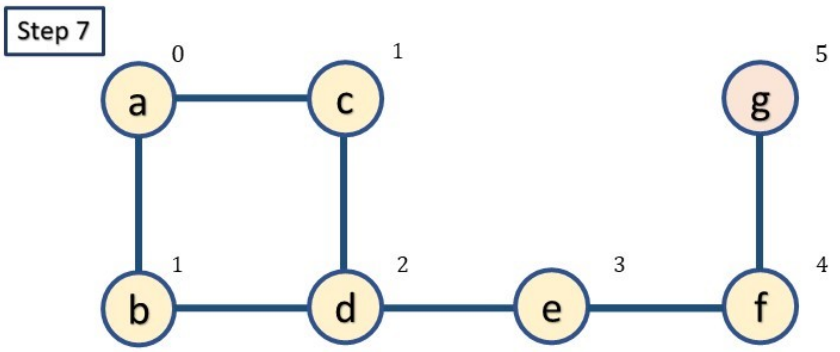
$$A = \{f(\min(4, \infty)), g(\infty)\}$$
$$B = \{a(0), c(1)_a, b(1)_a, d(2)_c, e(3)_d\}$$

FIGURE 3.6: Step 5: Procedure is repeated trying to reach the goal.



$$A = \{g(\infty)\}$$
$$B = \{a(0), c(1)_a, b(1)_a, d(2)_c, e(3)_d, f(4)_e\}$$

FIGURE 3.7: Step 6: Procedure is repeated trying to reach the goal. There are two stop conditions in the program. Whether all of the nodes have been visited, or whether the goal has been reached.



$$A = \{\}$$
$$B = \{a(0), c(1)_a, b(1)_a, d(2)_c, e(3)_d, f(4)_e, g(5)_f\}$$

FIGURE 3.8: Step 7: If the goal has been reached, the path is back-propagated by observing the predecessor of each node that leads to the goal. In this case the order will be: $g > f > e > d > c > a$. That will be the shortest path computed for the selected map, goal and initial nodes.

3.2.2 A* Algorithm

A* (Pronounced A star) shortest path algorithm is widely known for navigation strategies, as it is an upgraded version of Dijkstra's. The main idea is to compute the lowest cost path by measuring the cost of the edge weights. The main difference lays in the fact that it introduces a distance component to the loss function. This will be the distance to the goal, which in the case of the grid can be measured by the euclidean distance between the two nodes. Similar to Dijkstra's, the maths of the problem will be briefly introduced, followed by the algorithm and a practical example with the same graph.

As it will be observed, the difference in the procedure is very similar for small graphs, and the chance that the shortest path coincides with Dijkstra's shortest path is very high. This occurs mainly because of the size of the map. The main advantages of A* are observed in maps with a larger amount of tiles.

3.2.2.1 Mathematical explanation

Given a weighted graph $G(V, E)$ where V is a set of nodes $\{v_1, v_2, ..., v_n\}$ and E is a set of links $\{e_1, e_2, ..., e_n\}$. The cost between two nodes will be $w(u, v)$ where u and v belong to the node set. The shortest path between the starting s point to the goal v is denoted as the tentative distance, $d(v)$. That will be the value of the distance assigned to each of the nodes. A* worst case performance is defined by $O(|E|)$ operations.

Before the start of the algorithm, visited A set and unvisited set B are created. Node values are initially set to a tentative distance value of ∞ except the initial one, that will be assigned a value of $d(s) = 0$. This first one is set as current node.

Two costs will be assigned to each node. The first one, $g(v)$, is the cost of the path that has been followed, and it coincides with the cost used in the Dijkstra's algorithm. The second cost function, $h(v)$ corresponds to the distance between the currently selected node and the target node, equation 3.1. In the computer program, as it is a grid map, a useful option is the euclidean distance between them. The total node value will be the sum of both cost functions: $J(v) = h(v) + g(v)$.

$$h(v) = \sqrt{(x_v - x_{goal})^2 + (y_v - y_{goal})^2} \tag{3.1}$$

The distances to the neighboring unvisited nodes are computed, as well as their $g(v)$ cost. The minimum total cost $J(v)$ between their corresponding value and their previous value is assigned to them, they are referenced to their previous node, and they are included in the B set. After this, the node with the smallest value is visited and removed from the unvisited set A.

For this second node, the distances (link weights) to the neighboring unvisited nodes are J cost is computed. The minimum distance between their corresponding

value and their previous value is assigned to them, they are referenced to their previous node (this second node), and they are included in the B set. After this, the node with the smallest value is visited and removed from the unvisited set A.

As in Dijkstra's, the process is repeated until the goal has been reached, or all of the nodes from the set V have been visited. Once this condition is fulfilled, the target node will already have been reached and it will be contained in set B.

Its reference to the previous node can be extracted, and the solution will be back-propagated until reaching the source node s.

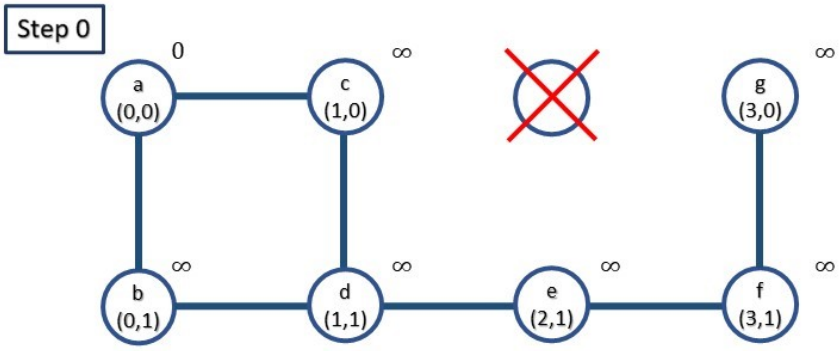
Algorithm 2 : A* algorithm

Initialization:
$G(V, E)$ - Graph that will be studied
A - Set containing all the nodes
B - Empty set of nodes

1: **for** `each node in` V **do**
2: `distance(V) =` ∞`;`
3: **end for**
4: Set $d(s) = 0$, add s to B and set s as current node.
5: **while** A is not empty or goal is not reached **do** ▷ Check all the nodes
6: $v_n \leftarrow$ Is the current node with value val_n
7: $N, \{n_1, n_2..., n_m\} \leftarrow$ Is the set of neighbor nodes of v_n
8: **for** `each node in` N **do**
9: $g(n_m) = val_n + w$
10: $h(n_m) = \sqrt{(x_{n_m} - x_{goal})^2 + (y_{n_m} - y_{goal})^2}$
11: $J(n_m) = g(n_m) + h(n_m)$
12: $d(n_m) = min\{currentvalue, J(n_m)\}$ Compute for all neighboring nodes.
13: **end for**
14: Set the node in N with smallest $d(n_m)$ as v_n, reference it to v_{n-1} and add it to B
15: **end while**

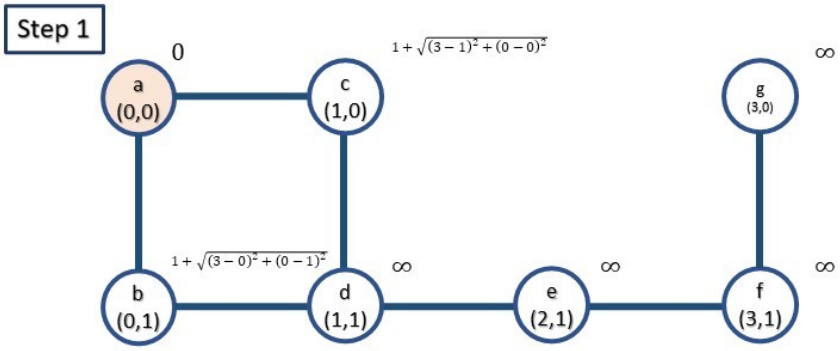
3.2.2.2 A* Algorithm Example

The following section will include a practical example of how the computer program will resolve the shortest path problem using the A* algorithm.



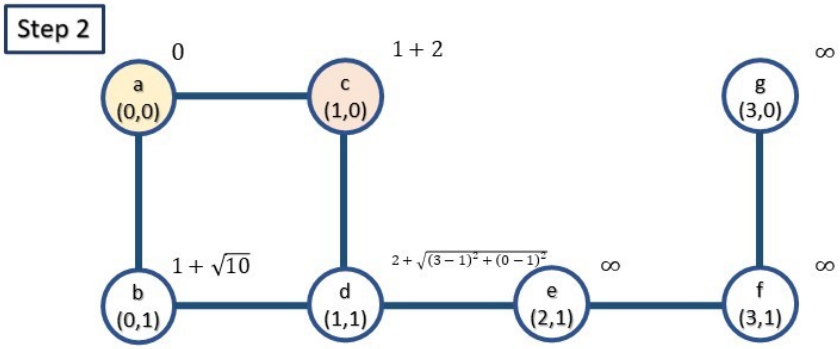
$$A = \{a(0), b(\infty), c(\infty), d(\infty), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{\}$$

FIGURE 3.9: Step 0: Firstly, the graph is built, ignoring the selected obstacle nodes so that there are no links between a free node and an osbstacle one. The initial node values is ∞ for all except the source node. The unvisited and visited sets, A and B, are initialized.



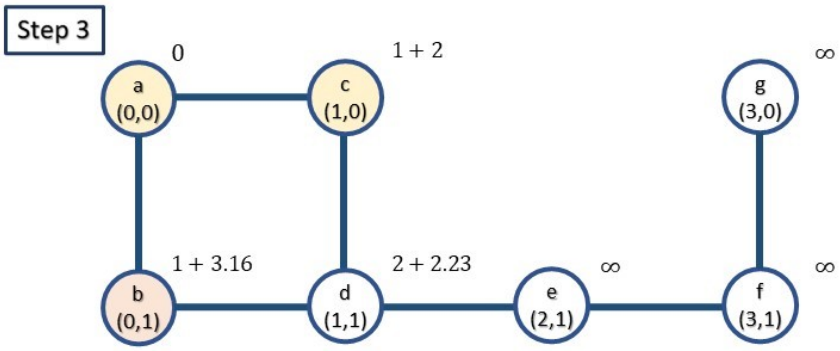
$$A = \{c(\min(3, \infty)), b(\min(4.16, \infty)), d(\infty), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{a(0)\}$$

FIGURE 3.10: Step 1: The source node a is selected (orange), removed from the unvisited set A, and added to the visited set B. It's value is a zero constant. The neighboring node values are also computed by calculating the minimum between the previous node value, and the addition of the previous node value plus $f(v) = g(v) + h(v)$. This value is updated in the unvisited set A, and the set is sorted from lowest value to the highest.



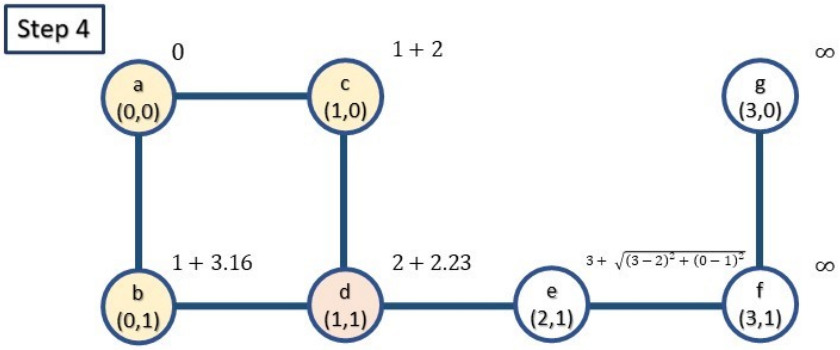
$$A = \{b(4.16), d(\min(4.23, \infty)), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{a(0), c(3)_a\}$$

FIGURE 3.11: Step 2: The node with smallest value from A will be selected now, c. It is added to the visited set B being linked to its predecessor node. Then c is removed from the unvisited set A. The neighboring node values are computed.



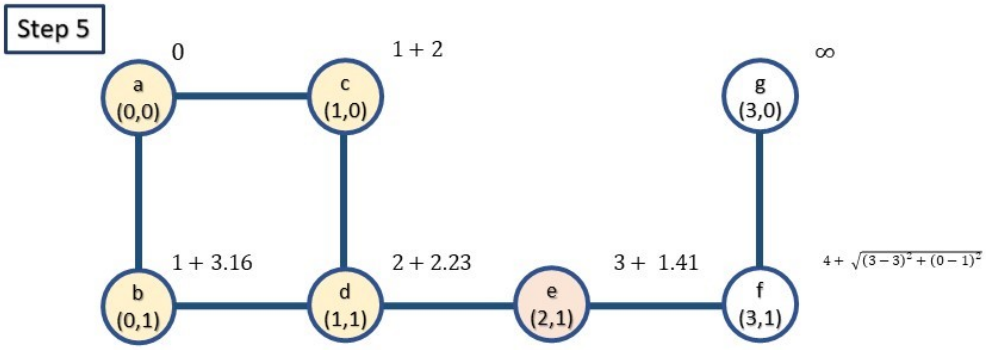
$$A = \{d(4.23), e(\infty), f(\infty), g(\infty)\}$$
$$B = \{a(0), c(3)_a, b(\min(4.16, 4.23 + 1))\}$$

FIGURE 3.12: Step 3: The next node in the A set will be b, reachable directly from a or by going through c and d. The one with smallest cost is selected. and the procedure in Step 1 Figure 3.10, is repeated. It can be observed that the value of node d is computed from two different nodes, the minimum of both should be selected.



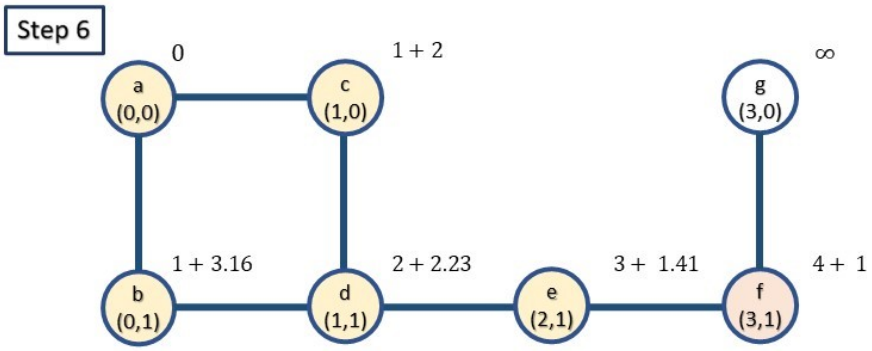
$$A = \{e(\min(4.41, \infty)), f(\infty), g(\infty)\}$$
$$B = \{a(0), c(3)_a, b(4.16)_a, d(\min(4.23, 5.16))\}$$

FIGURE 3.13: Step 4: The next node in A will be d, and the previous procedures are repeated, which involve the computation of the value of its neighbor e, removing d from A and adding it to B.



$$A = \{f(\min(5, \infty)), g(\infty)\}$$
$$B = \{a(0), c(3)_a, b(4.16)_a, d(4.23)_c, e(4.41)_d\}$$

FIGURE 3.14: Step 5: Procedure is repeated trying to reach the goal.



$$A = \{g(\infty)\}$$
$$B = \{a(0), c(3)_a, b(4.16)_a, d(4.23)_c, e(4.41)_d, f(5)_e\}$$

FIGURE 3.15: Step 6: Procedure is repeated trying to reach the goal. The two stop conditions are still whether all of the nodes have been visited, or whether the goal has been reached.

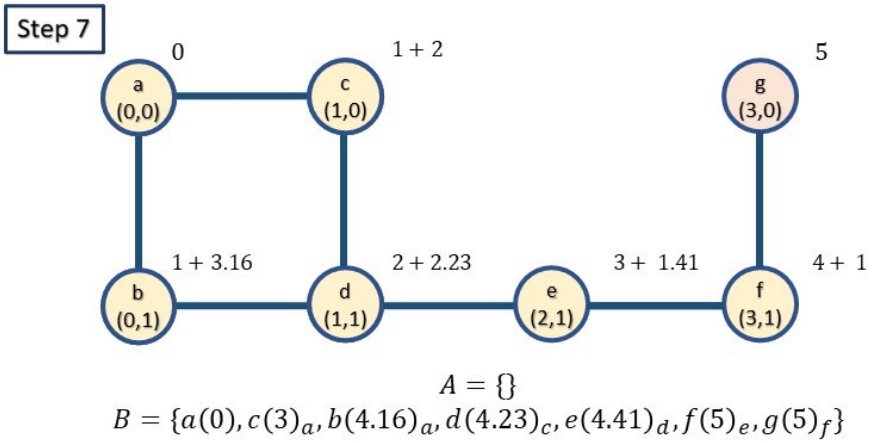


FIGURE 3.16: Step 7: If the goal has been reached, the path is back-propagated by observing the predecessor of each node that leads to the goal. In this case the order will be: $g > f > e > d > c > a$. That will be the shortest path computed for the selected map, goal and initial nodes. The path is apparently the same as Dijkstra's because of the small size of the map and limited path option.

3.3 Program description

This section of the report will describe the code of the visual navigation computer program that has been used to implement the Dijkstra's and A* algorithm.

It is important to recall that the kind of graph that will be used in the procedure will be a grid map. Movement will only be allowed in the vertical and horizontal directions, and each cell will correspond to one node, while the connections between cells correspond to the edges. The weight of these is set to 1.

The first lines in the code will define the size of the map, the weights of the edges, the position of the obstacles, and the start and target node.

```
# Limits of the size of the grid map
xlim, ylim = 30, 30

# Weight for the edges
w = 1

# Set up the positions of the obstacle
x_obs = [*[3]*11, 4, 5, 6, 7, *[8]*11]
y_obs = [*range(0, 10+1), 10, 10, 10, 10, *range(0, 10+1)]

# Given in map coordinates
start_node_name = [0, 0]
target_node_name = [5, 4]
```

These variables will be used as the input to the functions that create the graph and compute the solution with the algorithms. This procedure comes defined in Dijkstra and A_star functions.

```
node_map, t_value = Dijkstra(xlim, ylim, w, x_obs, y_obs,
                             start_node_name, target_node_name)

node_map, t_value = A_star(xlim, ylim, w, x_obs, y_obs,
                           start_node_name, target_node_name)
```

Before analyzing how these functions operate, it is convenient to define some utility methods and classes.

First of all, it is necessary to define the classes for the Dijkstra and A* nodes and edges. Nodes will only differ in the part that A* nodes will include the *value* as the total value, and the g and h cost values.

```
# Dijkstra's classes
class Node:
    def __init__(self, name, value, free, previous):
        self.name = name
        self.value = value
        self.free = free
        self.previous = previous


class Edge:
    def __init__(self, name, value):
        self.name = name
        self.value = value
```

```
# A* classes
class Node:
    def __init__(self, name, value, free, previous, gvalue,
hvalue):
        self.name = name
        self.free = free
        self.previous = previous
        self.value = value
        self.gvalue = gvalue
        self.hvalue = hvalue


class Edge:
    def __init__(self, name, value):
        self.name = name
        self.value = value
```

The procedure will be very similar in both techniques, A* and Dijkstra, with the main variation being defined only in the node properties and in the main algorithm, algorithms 1 and 2. Thus only the A* case will be explained.

After the classes, the first function is *create_graph*, which will receive as inputs the user specified map size, and the obstacles. With that information, it will be able

to create the map with the free space ($free = true$) and the obstacles ($free = false$). Then, the links between free nodes will be generated in the map. The function returns the node list with all the free nodes, and the links between them referred to as [current node, neighboring node].

```
def create_graph(xlim, ylim, x_obs, y_obs):
    node_list = []
    # Creation of all the nodes in the grid
    for y in range(ylim):
        for x in range(xlim):
            # Set the "free" value of the nodes that are taken by an obstacle to False
            for i in range(len(x_obs)):
# Check if node coincides with any of all obstacles
                if ((x == x_obs[i]) and (y == y_obs[i])):
                    print("Obstacle detected")
                    # If it coincides set the free value to false
                    current_node = Node([x, y], 10000, False, [], 0, 0)
                    break
                else:
                    current_node = Node([x, y], float(np.inf), True, [], 0, 0)

            node_list.append(current_node)

    # Finding all links between nodes
    edge_list = []
    for node in [i for i in node_list if i.free]:
        for neighbor in neighbors(node, node_list):
            edge_list.append(
                Edge([node.name, neighbor.name], 1))
    return edge_list, node_list
```

The next function is *neighbors*, whose objective is to determine the neighbors to the input node, in the node list V. It does this by moving in the four directions: up, right, down and left.

```
def neighbors(node, V):
    # 4 possible directions, up-down-left-right
    dirs = [[1, 0], [0, 1], [-1, 0], [0, -1]]
    result = []  # stores the neighbors
    for dir in dirs:
        neighbor_name = [node.name[0] + dir[0], node.name[1] + dir[1]]
        # If the neighbor belongs to the node list, it will be added to the neighbor list
        # List only containing the names of the nodes
        name_list = [x.name for x in V]
        if neighbor_name in name_list:
            # Get the index of the neighbor node in the list of names
            index = name_list.index(neighbor_name)
            result.append(V[index])  # Add the neighbor node to the result list
    return result
```

Function *print_map* will generate a grid map and assign values to it in order to

be able to visualize the result. If the value is still infinite, which means that the node has not been visited yet, it will be assigned a 0, and if it has another value, it will be assigned that value, rounded.

```
def print_map(node_list, xlim, ylim):
    # Create a matrix map to visualize the results
    node_map = np.zeros((ylim, xlim))

    for node in node_list:
        x = node.name[0]
        y = node.name[1]
        if node.value < float(np.inf):
            node_map[y][x] = round(node.value, 2)
        else:
            node_map[y][x] = 0
    return node_map
```

The final internal function that has been defined is called *backpropagate*. It will find the path that the robot should take by going from the goal node, and following the predecessors of each node that lead to it until the source node is reached. The path nodes are stored in the list variable called *path*. The output will have this path, and the updated graph with the values plus a big number (500 in this case) so that the color map shows the path followed.

```
def backpropagate(V, target_node):
    # Get the path followed by backrpopagation
    path = []
    # V will be used instead of V_free for representation
    node_names_full = [x.name for x in V]
    current_index = node_names_full.index(target_node.name)
    current = V[current_index]
    path.append(current.name)

    while current.previous:
        current_index = node_names_full.index(current.previous)
        current = V[current_index]
        path.append(current.name)
        V[current_index].value = V[current_index].value+500
    return path, V
```

After explaining the auxiliary functions, the *A_star* algorithm function can be defined. This is based on the pseudocode presented in 2, taken to the Python programming language. The comments in the code will help understand how the code works.

The first part of the function will initialize the nodes, the visited and unvisited set, and create the graph.

```
# Start and target node are defined according to the classes
    start_node = Node(start_node_name, 0, True, [], 0, 0)
    target_node = Node(target_node_name, 0, True, [], 0, 0)

    # Generate the graph and output edges and nodes G(E,V)
    E, V = create_graph(xlim, ylim, x_obs, y_obs)
    # Ignore all nodes with obstacles, keep V for representation
    V_free = [x for x in V if x.free]

    # Create a matrix map to visualize the results
    gridMap = print_map(V, xlim, ylim)
    print_map(V, xlim, ylim)

    # Create an array containing all the free node names in the graph
    node_names = [x.name for x in V_free]
    if (start_node.name not in node_names) and (target_node.name not in
 node_names):
        print("ERROR: Please select valid nodes")

    start_node_index = node_names.index(start_node.name)
    target_node_index = node_names.index(target_node.name)

    V_free[start_node_index].value = 0

    # Empty set B is initialized
    B = []
    # Start node is set as current
    v_n = V_free[start_node_index]
    # Set A initially contains all free nodes

# Error: A = V In Python, setting a variable equal to a list just points it to t
hat list object

    A = V_free.copy()
```

The second part of A_star refers to the main loop, in which the iteration between the current node is selected, the iteration along the neighbors is performed, and their values are assigned.

```
# Iterate through all free nodes
start_time = time.time()  # start the timer to measure performance
while A:
    # If the target is reached:
    if (v_n.name == target_node_name) and (v_n.value < float(np.inf)):
        # print("Target reached!")
        # Compute the path using backpropagation
        path, V = backpropagate(V, target_node)

        # To show the map uncomment this section
        # ======================================
        node_map = print_map(V, xlim, ylim)
        print(node_map)
        #  ======================================
        break

    # If all the nodes have been visited and the target has not been found
    elif (len(A) == 0):
        print("No target found :(")
        break

    node_index = node_names.index(v_n.name)
    val_n = v_n.value
    # Function neighbors takes the current node and the node list
    v_n_neighbors = neighbors(v_n, V_free)
    # Check the neighbors of the current node and compute their value
    for n in v_n_neighbors:
        neighbor_index = node_names.index(n.name)
        gdist = val_n+w
        hdist = np.sqrt(
            (target_node_name[0] - n.name[0]) ** 2 + (target_node_name[1
] - n.name[1])**2)
        fdist = gdist+hdist

# If the new value is smaller, assign that value to the node and set the curren
t node as previous

        if (n.value > fdist):
            n.value = fdist
            n.previous = v_n.name
        V_free[neighbor_index].value = n.value

    # Remove current node from unvisited set and add to the visited set
    B.append(v_n)
    A.remove(v_n)
    A.sort(key=lambda x: x.value)

    if A:
        v_n = A[0]

    #  ======================================
    node_map = print_map(V, xlim, ylim)
    # print(node_map)
    #  ======================================
# Stop timer and get the time of the computation
t_value = time.time() - start_time
print("--- %s seconds ---" % (t_value))
return node_map, t_value
```

3.4 Algorithm comparison

Test	Section	Goal
Small map	3.4.1	Compare the algorithms to the explanations in the report for small maps and measure its performance.
Medium map	3.4.2	Evaluate the performance in medium sized maps to compare it with bigger and smaller grids.
Big map	3.4.3	Compare the performance of the algorithms for more real-sized scenarios that are not limited only to a few nodes.
Big map with diagonal movement	3.4.4	Evaluate the performance drop due to the number of computations, as well as the performance of both algorithms in a more realistic scenario.

TABLE 3.1: Tests performed for comparison of Dijkstra's algorithm and A*

Once the algorithms have been explained step by step, it is convenient to test them on the computer program and compare their performance.

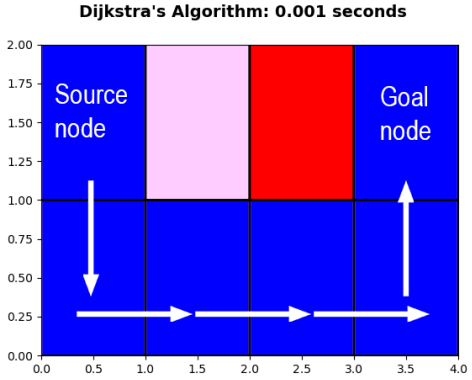
Four representative experiments were performed, described in 3.1.

3.4.1 Small map experiment

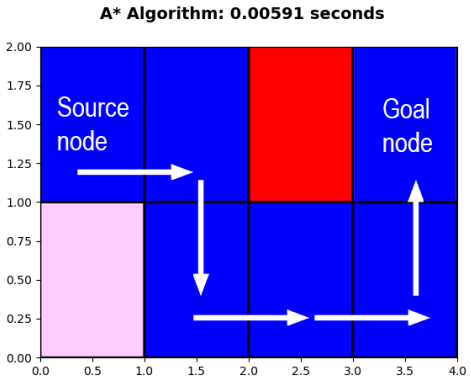
This first tests demonstrates the procedure of the previous sections for a simple grid map. The source node is in this case the top left corner cell, the obstacle is indicated in red, and the target node is the top right corner.

3.2a shows how the Dijkstra's algorithm randomly selects either node b or c, as it does not consider the distance to the goal. Meanwhile, A* does go through the path that is closer to the goal, node c.

It can also be observed the time values for the computation of the solution. The Dijkstra's algorithm apparently takes less time to resolve the problem than A*. However this is due to the fact that the A* algorithm has to compute two loss functions, and in very small problems (maps).



(A) Dijkstra's algorithm on a 2x4 map. Blue indicates the followed path, while red indicates obstacles (non-free space)



(B) A* algorithm on a 2x4 map.

3.4.2 Medium map experiment

The second test is performed in a 8x8 grid map with a diagonal obstacle. The time taken for the Dijkstra's algorithm to reach the goal is almost half of the time for A*, being 0.01518 for the former and 0.03408 for the latter. It can also be observed that in the case of the A* algorithm, not all nodes are being visited, leaving the unvisited ones blank.

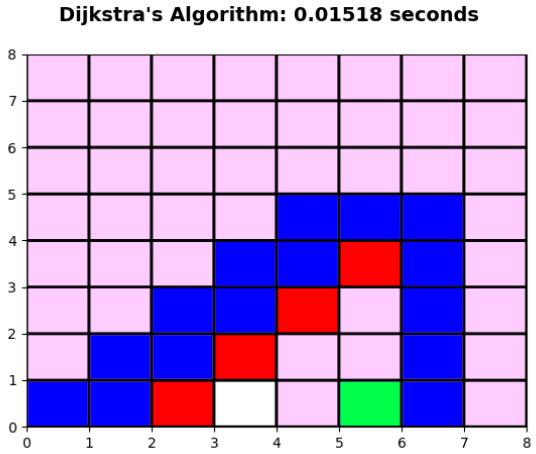


FIGURE 3.17: Dijkstra's algorithm tested on a 8x8 grid map. Source node is located at [0,0], and target node is colored green and in position [5,0]. Followed path is marked with dark blue color and obstacles are dark red.

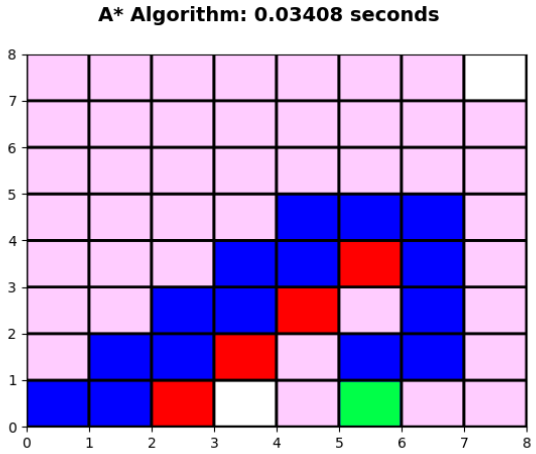


FIGURE 3.18: A* algorithm tested on a 8x8 grid map. Source node is located at [0,0], and target node is colored green and in position [5,0]. Followed path is marked with dark blue color and obstacles are red. Darkness levels of pink show the scale of the grid cell values.

3.4.3 Big map experiment

The second test, referring to a 30x30 grid shows how the A* algorith gets a computatino time of 0.68939s, clearly outperforming the Dijkstra's algorithm that takes a computation time of 1.10064s. The number of nodes visited by A* is noticeably lower, resulting in a great increase in performance and computation time for bigger maps.

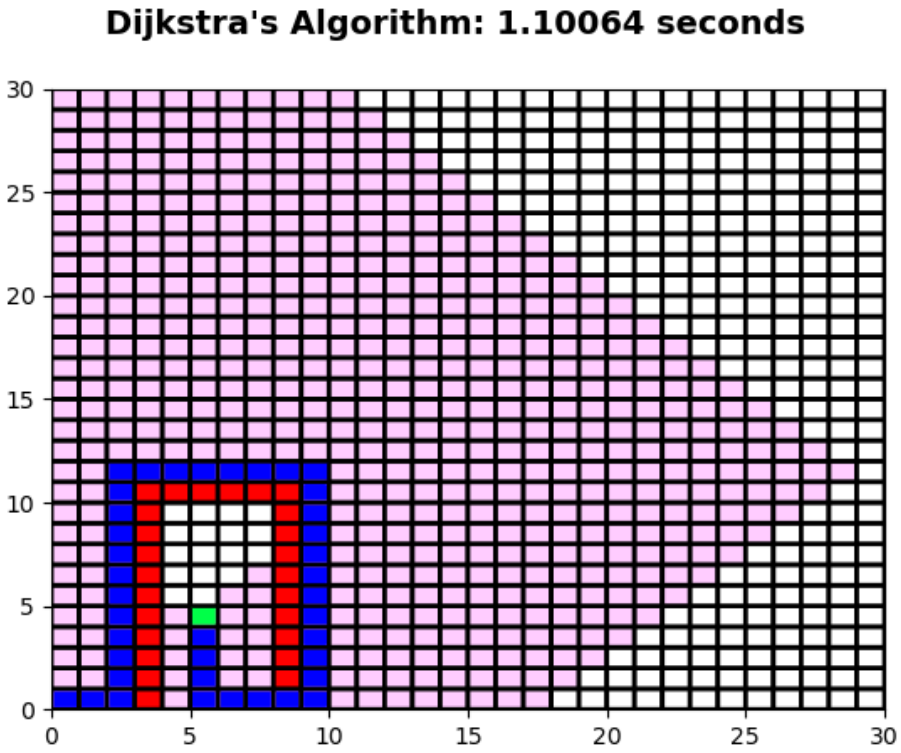


FIGURE 3.19: Dijkstra's algorithm tested on a 30x30 grid map. Source node is located at [0,0], and target node is colored green and in position [5,4]. Followed path is marked with dark blue color and obstacles are dark red.

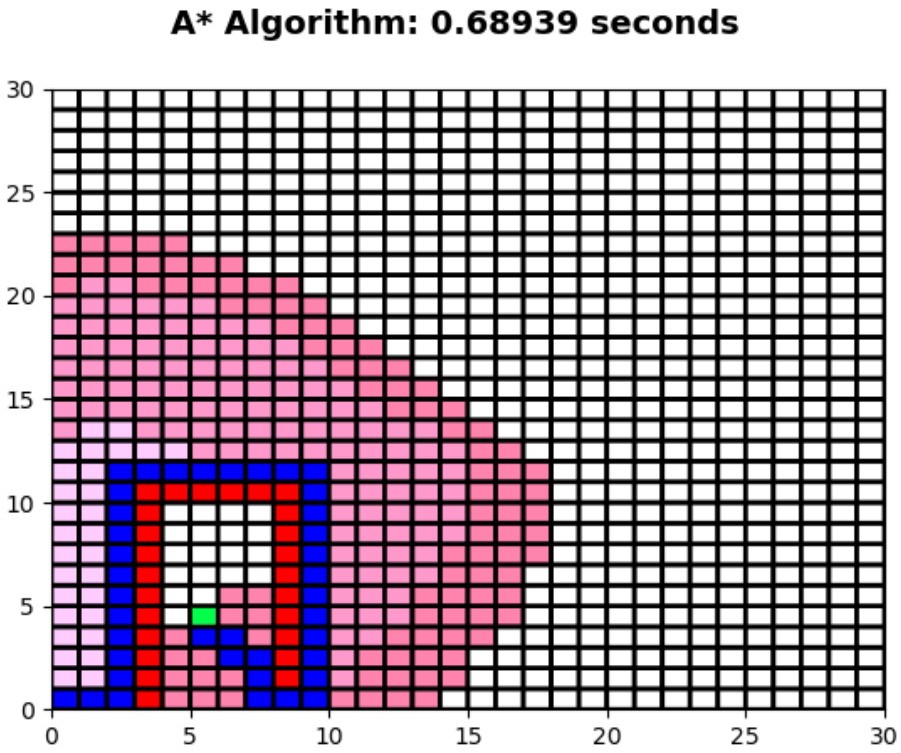


FIGURE 3.20: A* algorithm tested on a 30x30 grid map. Source node is located at [0,0], and target node is colored green and in position [5,4]. Followed path is marked with dark blue color and obstacles are red. Darkness levels of pink show the scale of the grid cell values.

3.4.4 Big map experiment with diagonal movement

The last experiment was carried out with the aim of creating a more realistic scenario, where diagonal movement is permitted in a big map. This was achieved by adding the neighbors to be visited in the grid map in the diagonal positions of the tiles.

For both algorithms, due to the need of performing more operations, the execution time increases by 271% in the Dijkstra's Algorithm, and 206% with A*. The difference between both path finding techniques is still visible, with the A* algorithm performing noticeably better and visiting less nodes thanks to its goal-oriented cost function.

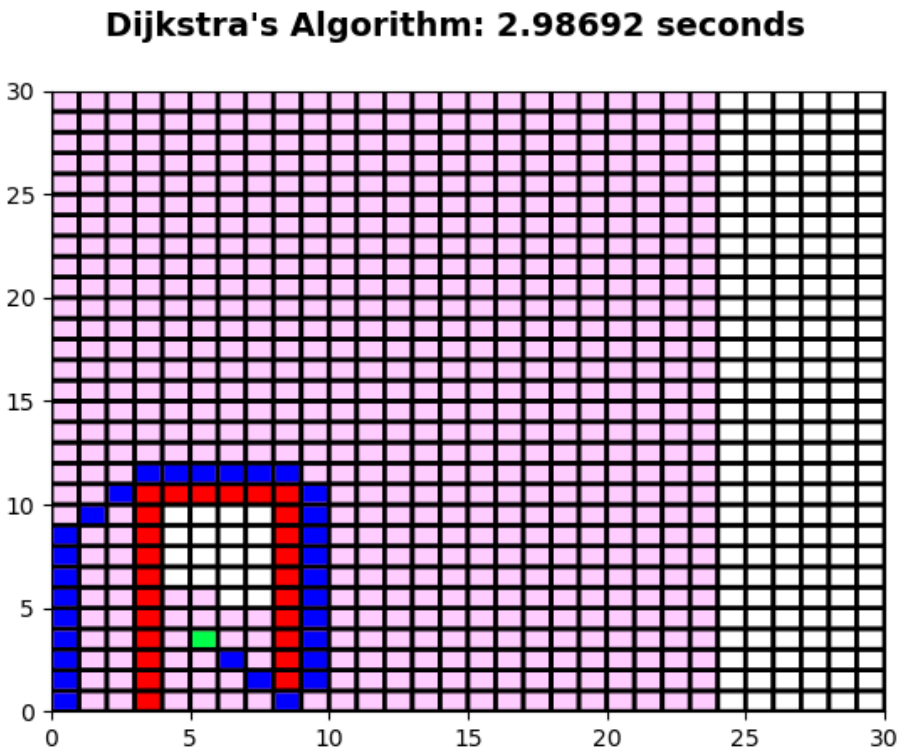


FIGURE 3.21: Dijkstra's algorithm tested on a 8x8 grid map. Source node is located at [0,0], and target node is colored green and in position [5,0]. Followed path is marked with dark blue color and obstacles are dark red.

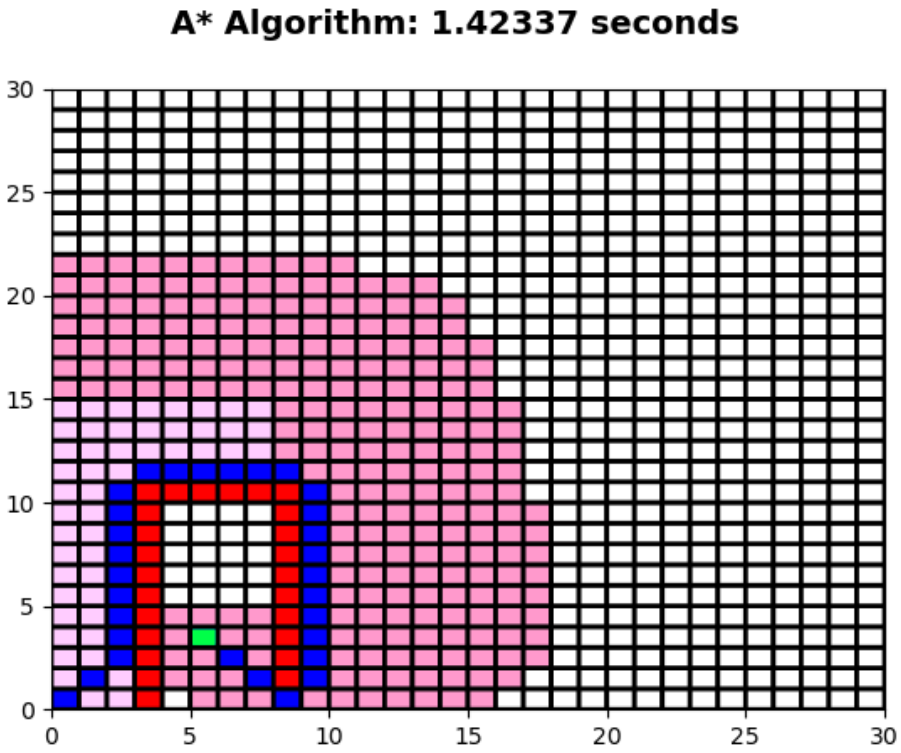


FIGURE 3.22: A* algorithm tested on a 8x8 grid map. Source node is located at [0,0], and target node is colored green and in position [5,0]. Followed path is marked with dark blue color and obstacles are red. Darkness levels of pink show the scale of the grid cell values.

3.5 Conclusion of the computer simulation

After both algorithms converging to a solution in all the tests, the following conclusions can be extracted:

- Both algorithms will converge to a solution if available and obtain the path avoiding the obstacles successfully.
- The time difference is minimal for very small maps, with Dijkstra's algorithm obtaining slightly better computation time due to the reduced number of operations.
- For medium sized maps, the number of operations will affect performance more than the number of nodes visited, and again, the Dijkstra's algorithm will outperform A*.
- For large maps, the goal oriented cost function of A* influences positively on the path search. Less nodes are visited, and the target is reached in much less time than Dijkstra's technique.

- Diagonal movement will simulate more realistic situations, however, the number of operation duplicates and the computation time doubles. Moreover, the performance drop is less in the A* algorithm than in Dijkstra's.

Chapter 4

Conclusion

This project studied the problem of visual navigation of mobile robots. The following key learnings have been extracted from the project:

- The first stage in visual navigation refers to the process of exploration. Sensors are used with the purpose of obtaining the data from the environment an creating a map.
- The map can be metric if high accuracy is required, having a tradeoff with the computer power required. Or it can be topological if advanced feature extraction is used and less accuracy is admitted.
- Two of the main classical algorithms for path generation and obstacle avoidance are the Dijkstra's algorithm and A*, whose performance might vary depending on the size of the map.
- Dijkstra's algorithm is based on finding the shortest path in a graph by covering all nodes until goal is reached.
- A* algorithm is an improved version of Dijkstra's in the way that it has a goal-oriented cost function that will help it to reaching a solution visiting less nodes.
- In grid or tile maps, Dijkstra's algorithm will perform better in reduced size maps while A* will be better with larder grids.
- The more realistic the simulation scenario is, the higher the computation time will be, as it has been the case with diagonal movement.

Further research on this topic could be carried out by evaluating other path planning algorithms and comparing them to the current ones. Examples of these are probabilistic road maps, recursive random trees and first breath search.

In addition, tests could be performed in real robots to select the optimal method for visual navigation of mobile robots, expanding the topic to reactive navigation and avoiding dynamic obstacles for a more realistic environment.

Bibliography

[1] Francisco Bonin-Font, Alberto Ortiz, and Gabriel Oliver. "Visual Navigation for Mobile Robots: A Survey". en. In: *Journal of Intelligent and Robotic Systems* 53.3 (Nov. 2008), pp. 263–296. ISSN: 0921-0296, 1573-0409. DOI: 10.1007/s10846-008-9235-4. URL: http://link.springer.com/10.1007/s10846-008-9235-4 (visited on 12/04/2020).

[2] David Filliat and Jean-Arcady Meyer. "Map-based Navigation in Mobile Robots:: I. A Review of Localization Strategies". In: *Cognitive Systems Research* (Jan. 2003), pp. 243–282.

[3] J. Borenstein et al. "Mobile Robot Positioning: Sensors and Techniques". In: *Journal of Robotic Systems* 14.4 (1997), pp. 231–249. DOI: https://doi.org/10.1002/(SICI)1097-4563(199704)14:4<231::AID-ROB2>3.0.CO;2-R.

[4] Daniel Casado Herráez. "Self-Driving Car Autonomous System Overview-Industrial Electronics Engineering-Bachelors' Thesis". In: (2020).

[5] M. Himmelsbach, F. v. Hundelshausen, and H. . Wuensche. "Fast segmentation of 3D point clouds for ground vehicles". In: *2010 IEEE Intelligent Vehicles Symposium*. 2010, pp. 560–565. DOI: 10.1109/IVS.2010.5548059.

[6] I. Bilik et al. "The Rise of Radar for Autonomous Vehicles: Signal Processing Solutions and Future Research Directions". In: *IEEE Signal Processing Magazine* 36.5 (2019), pp. 20–31. DOI: 10.1109/MSP.2019.2926573.

[7] John Leonard and Hugh Durrant-whyte. *Directed Sonar Sensing for Mobile Robot Navigation*. May 2004. ISBN: 978-1-4613-6625-6. DOI: 10.1007/978-1-4615-3652-9.

[8] E. Zalama et al. "Concurrent mapping and localization for mobile robots with segmented local maps". In: *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Vol. 1. 2002, 546–551 vol.1. DOI: 10.1109/IRDS.2002.1041447.

[9] A. Elfes. "Using Occupancy Grids for Mobile Robot Perception and Navigation". In: *Computer* 22.6 (1989), pp. 46–57. DOI: 10.1109/2.30720.