Introduction to Mobile Robot Path Planning

Unit 3: A* search algorithm

- Summary -

Estimated time to completion: **2 hours**

In this unit, you will learn about the A* search algorithm. Specifically, you will learn about:

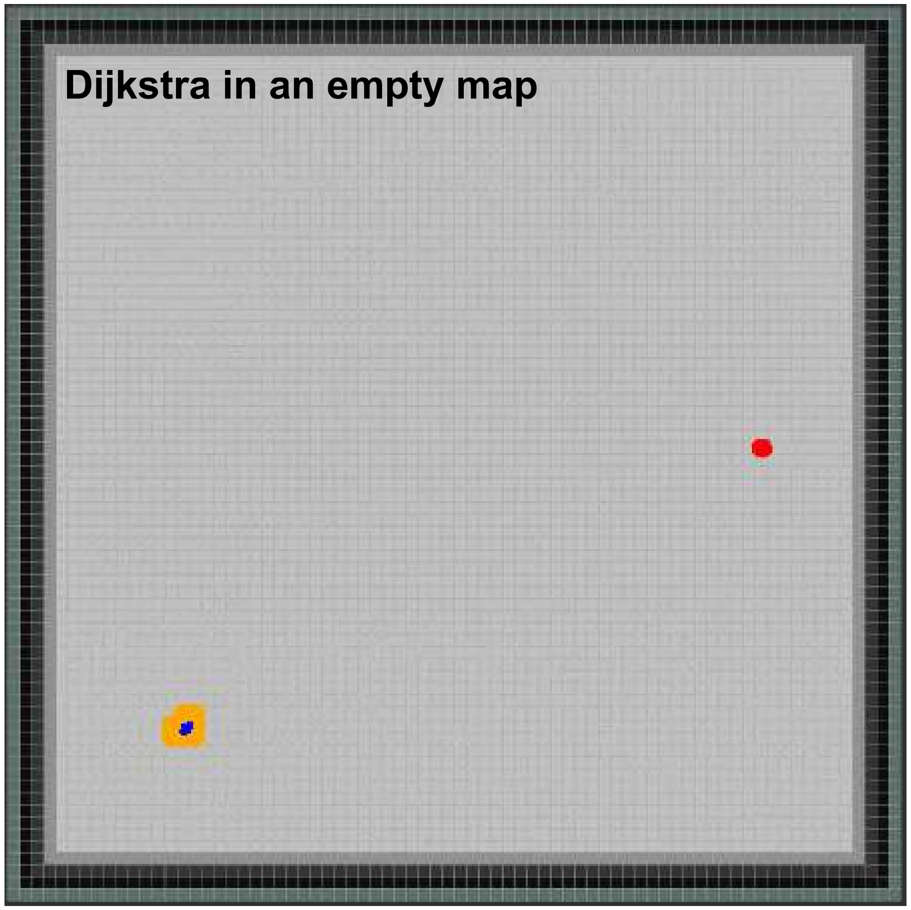
- Informed vs. uninformed search
- What heuristics are and how they can improve path planning speed
- Greedy Best-First Search algorithm
- The A* search algorithm
- Write and test your own implementations through simulation

- End of Summary -

3.1 Introduction

In unit 2, we discussed Dijkstra's algorithm as a way of finding the shortest path between two locations on a map.

If you recall, Dijkstra expands its search to new nodes according to the distance to the starting location. This means that it searches for the goal, expanding outwards from the starting node in all directions, regardless of where the goal node is located.



Nodes in yellow are explored until the path is found.

The image above shows the downside of Dijkstra's Algorithm: the blind search in all directions often consumes a lot of processing time, visiting nodes that are not neccesary because they do not lead to the goal.

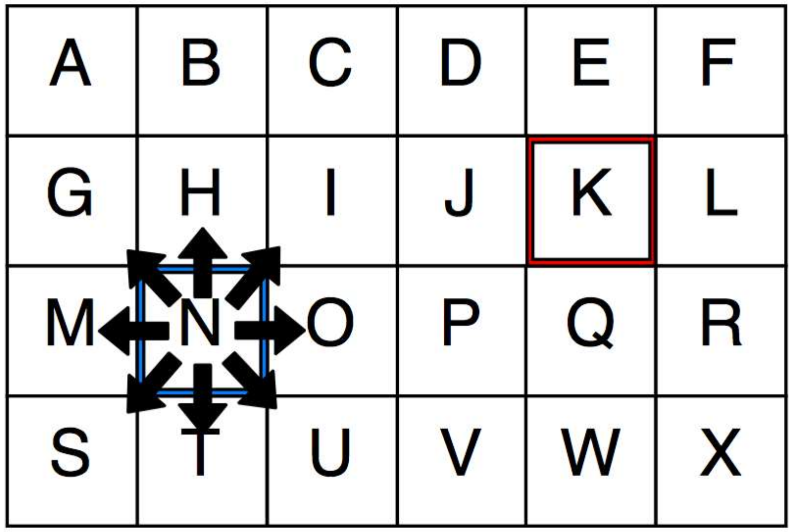
While Dijkstra's algorithm is simple and effective, it is not very efficient. It is, however, a very important algorithm that lays the foundation for other path planning methods. So if you understand Dijkstra, you have already gained a lot of intuition on how the basics of path planning work and are ready to go to the next level!

This unit focuses on the A* (pronounced "A-star"). A* is one of the most popular choices for pathfinding, because it is only a step up from Dijkstra's, but can often find an optimal path much faster.

But before we get much further, let's back up and talk about an important building block that makes faster algorithms possible: **Informed Search**

3.2 Informed Search Algorithms

Informed Search algorithms utilize the information about the goal location to guide the search towards the target and produce a more efficient exploration of the map. For instance, we know that if we want to drive to a location that is east of us, we should search a road that also goes to the east because roads going to the west will most likely take us further away from our target. Have a look at this small grid map below:



In this image, say that we know that we want to go to node 'K'. Traversing through node 'M' is not a good idea because it will take us away from node 'K'. We prefer to explore node 'I' or 'O' over 'M' because we intuitively understand that we will be moving closer to 'K' if we do so. Actually, what we are doing is using the information about the the location of 'K' to steer the direction of the search. This is an example of **informed search**.

In the example, we can easily look at the best nodes to expand our search. However, what if our map is much bigger — let’s say 10,000 nodes? It wouldn’t be as easy for us to decide which node to explore next. How can we quickly find which nodes are the ones that will lead us faster to the goal? And how do we express this in code?

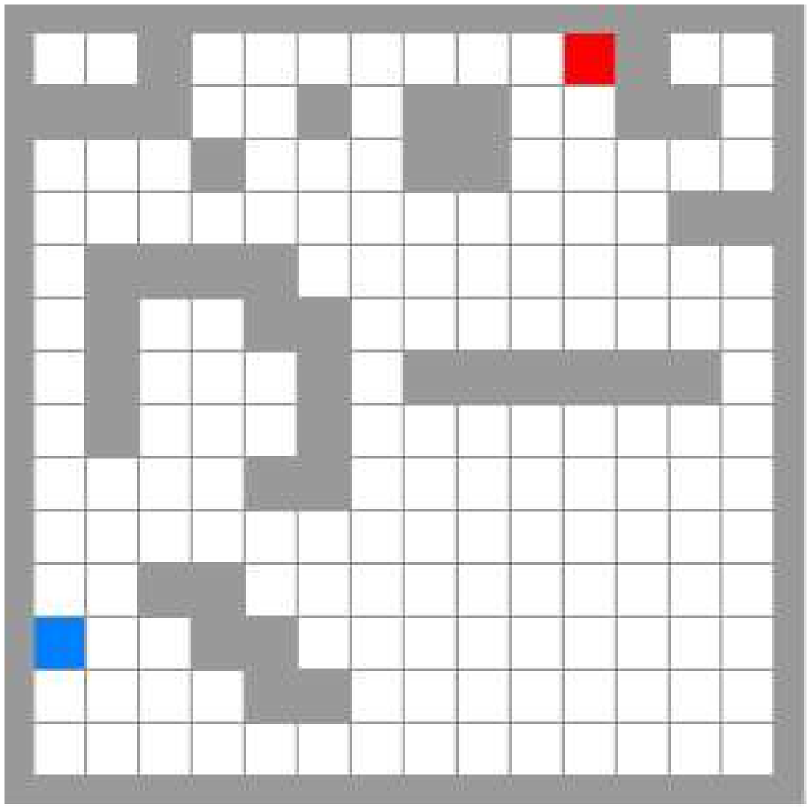
Here is where the concept of **Heuristic** kicks in.

3.3 Heuristics

We can avoid unnecessary exploration in all directions using a **prioritization method** that answers the question "Which node should be explored next?"

To provide an accurate answer to this question, it is neccesary to know the travel distance from each node to the goal node. With this information, we would simply take the node with the shortest travel distance to the goal as the next node to explore. It's as easy as that.

The problem is that at the moment of searching for a path, we just don't know what each nodes's travel distance to the goal is. Why? Because this is exactly the problem we are trying to solve! Just have a look at the image below, and see if you can tell the travel distance from 'blue' to 'red':



Did you notice? You first had to find a path from 'blue' to 'red' before you could come up with the travel distance between both. As a matter of fact, to quickly get the *next node to explore* from and to every possible node, we would need to precompute the shortest path between every pair of nodes in the map. Unfortunately, this not feasible for most maps, as the total number of grid cells with every possible path combination is typically too large to be computed and also stored in memory.

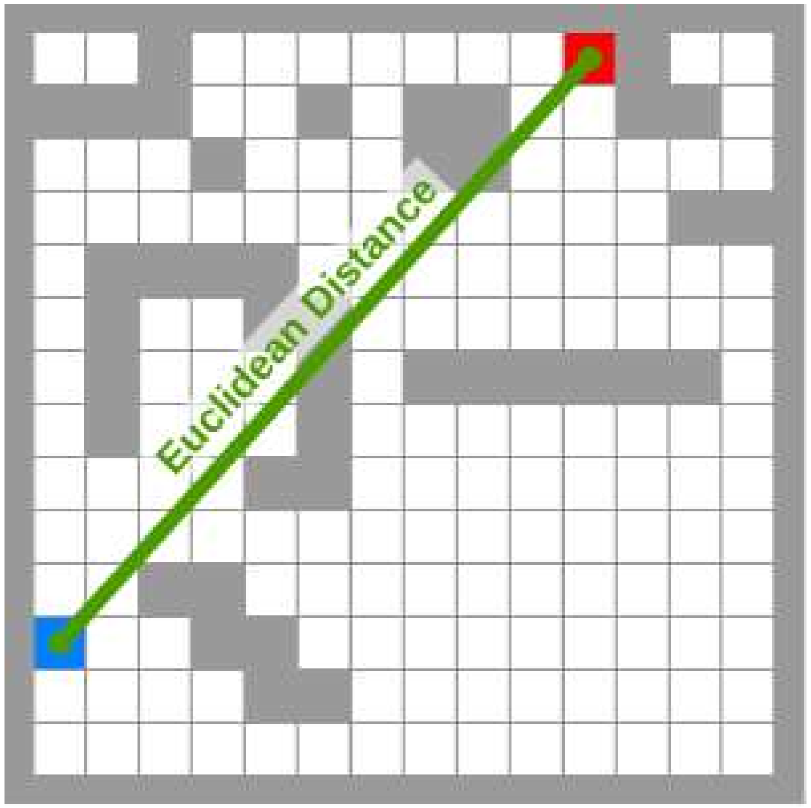
This is why, rather than working with an accurate answer to our question, we work with an **approximation**.

A **heuristic** is a method that serves us to solve a problem with an approximate solution. This solution is not optimal, but has the advantage that it can be calculated very quickly and helps to solve the main problem at hand.

Euclidean distance

The most common heuristic method for approximating the travel distance from one point to another is the Euclidean distance.

The Euclidean distance is really just a fancy name for the imaginary straight line between two points.



Uhh, wait a second...what about the obstacles? Are we going to totally ignore the fact that the line is crossing over obstacles? The plain and simple answer is yes! All we need to guide our search is some kind of value that tells us where to explore next, and we need to calculate that value fast, otherwise we won't get much benefit from it.

So this is how we calculate the Euclidean distance between two points (x1,y1) and (x2,y2):

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

The distance between any two points in a coordinate plane (x,y) is the difference between each x-value squared, plus the difference between each y-value squared, and finally take the square root of that.

- Exercise 3.3.1 -

Exercise

Implement the function `euclidean_distance()` that takes in two points as two lists of [x,y] coordinates and returns the Euclidean distance between them.

For example, if the nodes' coordinates are `point_a = [5.5, 10.2]` and `point_b = [-8.0, 4.6]` , your function call might look like this:
`h = euclidean_distance(point_a, point_b)` ,
where we saved the output of the function to the variable `h` .

Tasks

- Go to `catkin_ws/path_planning_course/unit3/scripts/`
- Open this file `unit3_gbfs_exercise.py` .
- Add an `euclidean_distance()` function with a function signature as presented below

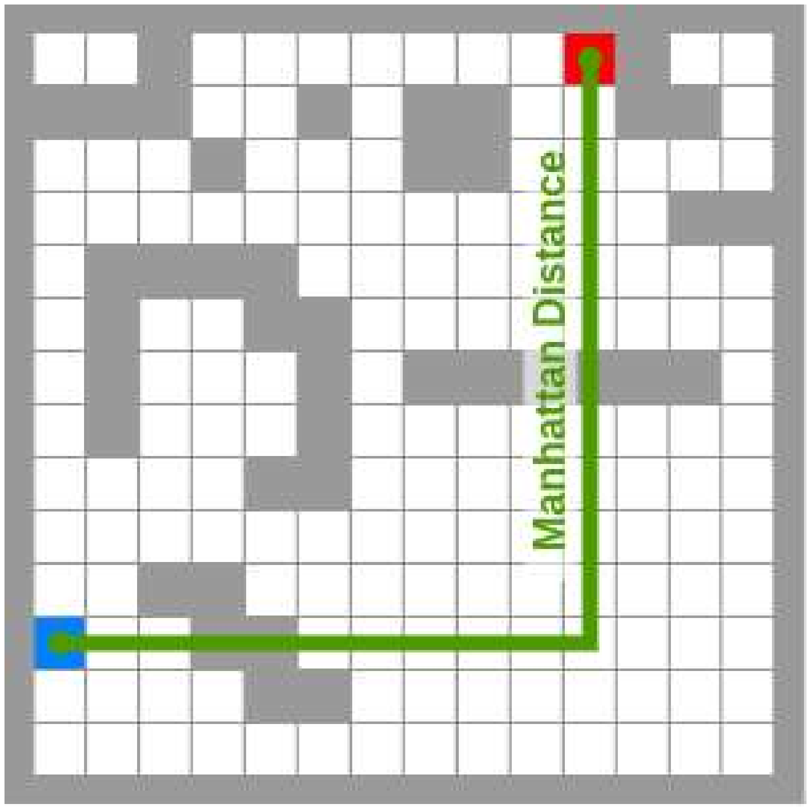
A good idea is to write your code first into a separate python script and test it with the provided values.
Compare the results of the function to the value we expect, which is: 14.615402834
Afterwards, you can add the function to the existing `unit3_gbfs_exercise.py` file.

In [ ]:
```python
# two x,y points for testing your code
point_a = [5.5, 10.2]
point_b = [-8.0, 4.6]

def euclidean_distance(a, b):
    # implement the body of this function
    pass

# statements to test the functions return value
h_value = euclidean_distance(point_a, point_b)
print(h_value)
```

- End of Exercise 3.3.1 -

Manhattan Distance

The Manhattan Distance is the distance between two points measured along the axes at right angles. It derives its name from the fact that the distance is calculated allowing only left/right and up/down movements, just as one would travel through the streets of Manhattan, New York.



The Manhattan Distance is shown by the green line. We see that the path includes both horizontal or vertical segments, and the angle between these segments is 90 degrees.

In a 2-d space, the Manhattan Distance is defined as the sum of absolute differences between two points (x1,y1) and (x2,y2):

$$d = |x_1 - x_2| + |y_1 - y_2|$$

Manhattan Distance is a way to get very, very fast distance approximations without requiring square root calculations!!

For instance, say: What is the Manhattan Distance between the coordinates (1, 0) and (10, 0)?
See if you can do mental math on the spot.

Solving the same exercise in your head using Euclidean distance can take you much longer.. and this applies to a computer as well.

- Exercise 3.3.2 -

Exercise

Implement the function `manhattan_distance()` . This function should take in two lists in the form [x, y], where [x,y] is one pair of coordinates associated with a point, and return the Manhattan Distance.

Tasks:

- You should still have the file from the previous exercise `unit3_gbfs_exercise.py` open. If you don't go ahead and open it now.
- Have a look at the function shown below
- Implement the body of the function
- You can run using the Python interpreter with different values for `a` and `b` to confirm that it works as intended
- Finally add this function to the file `unit3_gbfs_exercise.py`

In [ ]:
```python
# two x,y points for testing your code
point_a = [5.5, 10.2]
point_b = [-8.0, 4.6]

def manhattan_distance(a, b):
    # implement the body of this function

    pass

# statements to examine the functions return value
h_value = manhattan_distance(point_a, point_b)
print(h_value)
```

- End of Exercise 3.3.2 -

Euclidean distance and Manhattan distance are two of the most widely used heuristics methods used for approximating travel distances between nodes. There are, however, many other well known heuristic methods. For instance, you could use a spherical approximation for better estimating the distance between two distant points on Earth's surface. Just remember to check that you are using an **admissible heuristic**. What is an admissible heuristic? An admissible heuristic is one who's estimated distance is always lower or equal to the real distance, i.e. it never overestimates it.

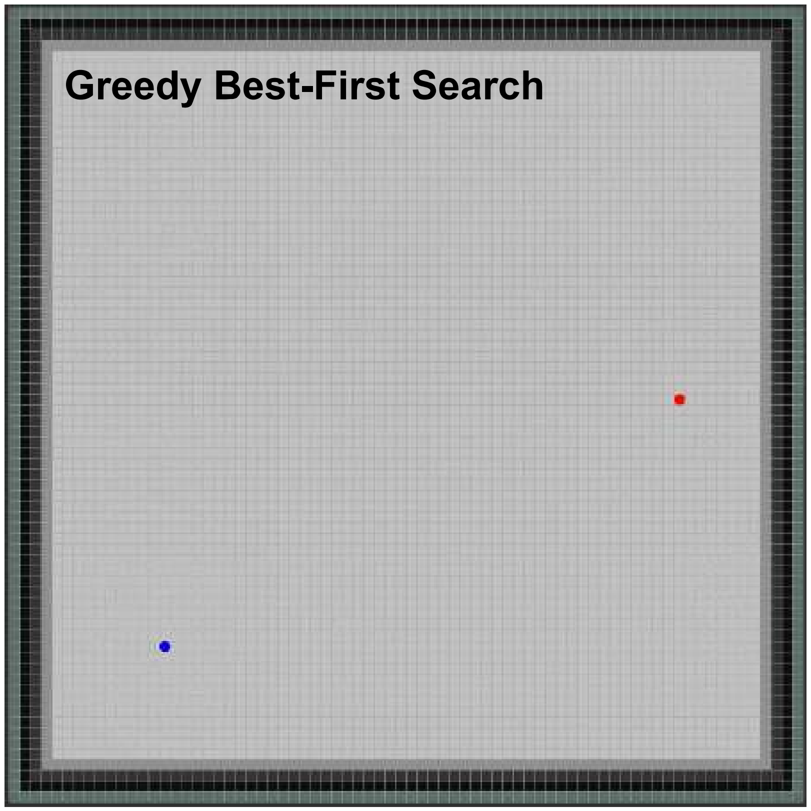
Before moving on, let me repeat why an heuristic is important once again: we can guide our search towards the goal position, which can **speed up the path search by a noticeable amount**.

So, what if we rewrite Dijkastra's algorithm to explore nodes according to their heuristic value? Let's find out in the next section!

3.4 Greedy Best-First Search

The Greedy Best-First Search (Greedy BFS) algorithm shares most of its code with Dijkstra's shortest path algorithm, except that it expands its way by selecting the node closest to the goal. To estimate how far from the goal any node is, an heuristic function is used.

Due to this change, Greedy BFS can run much quicker than Dijkstra! For example, if the goal is to the upper right of the starting position, Greedy BFS will tend to focus on nodes that lead to the upper right. In the image below, orange represents nodes in the **open list** (candidates to be selected as the current node) and yellow represents **closed list** nodes. It shows that Greedy BFS can explore significantly fewer nodes than Dijkstra:



Greedy BFS takes into account the goal position to guide its search efforts towards the goal and find a path very quickly.

Now, let's put theory into practice and see Greedy BFS in action!

- Exercise 3.4.1 -

From Dijkstra to Greedy Best-First Search

In this exercise, you will modify Dijkstra's code from the previous unit. Your goal is to convert it into a Greedy Best-First Search algorithm.

Open `unit3_gbfs_exercise.py` , located inside `catkin_ws/path_planning_course/unit3/scripts/` .

Before moving forward, please make sure you have added the heuristic functions to your script as described on Exercise 3.3.1 and Exercise 3.3.2 above. You will need them for this exercise.

All right, here are a few tips to help you get started:

- Basically, you need to change the criteria used to select `current_node`
- Instead of picking the node with the shortest travel distance (**g_cost**), extract the node with the lowest heuristic value (call it **h_cost**)
- You can choose which heuristic function to use
- Notice that the map nodes are kept as indices of a *1-D flat map*. What's important here is to know that it does NOT make any sense to calculate a distance between indices except their corresponding x,y coordinates! In order to do this, the function `indexToWorld()` is provided:

```python
def indexToWorld(flatmap_index, map_width, map_resolution, map_origin = [0,0]):
    """
    Converts a flatmap index value to world coordinates (meters)
    flatmap_index: a linear index value, specifying a cell/pixel in an 1-D array
    map_width: number of columns in the occupancy grid
    map_resolution: side lenght of each grid map cell in meters
    map_origin: the x,y position in grid cell coordinates of the world's coordinate origin
    Returns a list containing x,y coordinates in the world frame of reference
    """
    # convert to x,y grid cell/pixel coordinates
    grid_cell_map_x = flatmap_index % map_width
    grid_cell_map_y = flatmap_index // map_width
    # convert to world coordinates
    x = map_resolution * grid_cell_map_x + map_origin[0]
    y = map_resolution * grid_cell_map_y + map_origin[1]
    return [x,y]
```

Tasks:

- Modify the `greedy_bfs()` function which currently implements a Dijkstra's algorithm. It should select the next node to explore using an heuristic function to calculate and use the *h_cost* instead of the *g_cost*.

In [1]:
```
# Open unit3_gbfs_exercise.py and modify the greedy_bfs() function to utilize a heuristic value instead of th
```

Click HERE for a hint.

- End of Exercise 3.4.1 -

3.5 Testing Greedy Best-First Search

- Exercise 3.5.1 -

Launch your script with the following command:

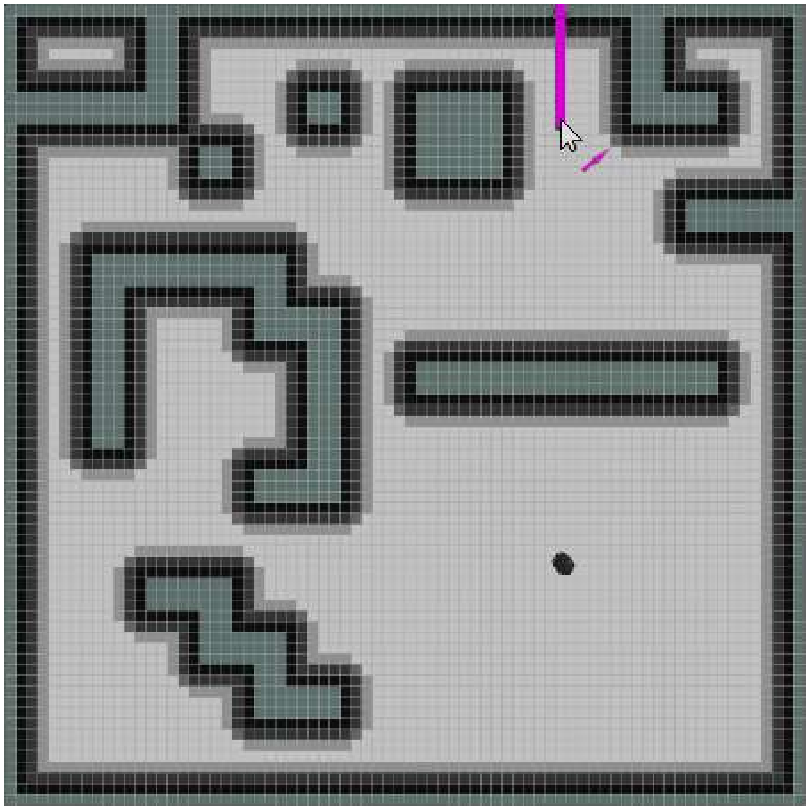
▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit3 unit3_gbfs_exercise.launch
```

You should see the **Graphical Tools** screen open. Please wait until your robot appears at the center of the grid map in RViz.

Command your robot to different goal locations and watch it plan and execute serveral different paths and find its way around obstacles of different complexity.

Expected result



Stop the execution of Greedy BFS by pressing **Ctrl + C**.

Now launch **Dijkstra** with the following command:

▶ **Execute in WebShell #1**

In [ ]:
```
roslaunch unit3 unit3_dijkstra.launch
```

Switch between **Dijkstra** and **Greedy BFS** and compare both algorithms.

Can you confirm:

- Who finds the better path?
- What algorithm is faster?
- What about the number of path bends?

See if you can spot its strong and weak points.

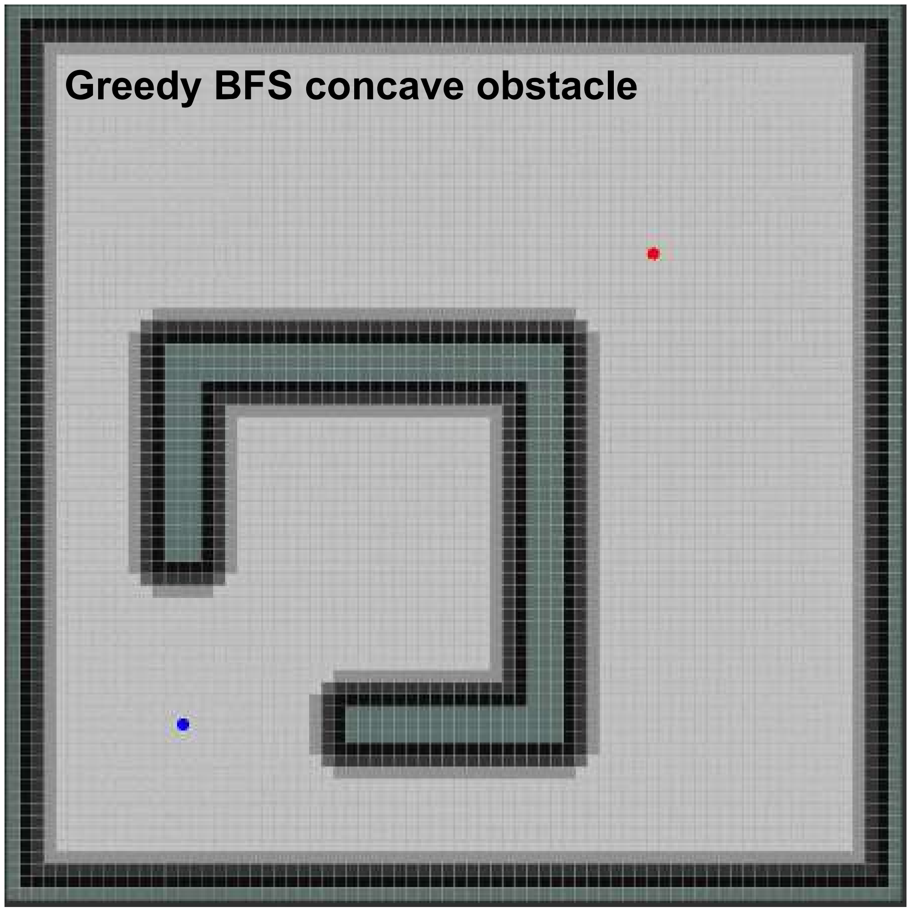
Don't forget to stop the current path planner by pressing **Ctrl + C** when you finish.

- End of Exercise 3.5.1 -

3.6 Exposing Greedy BFS's Behavoir

During your tests perhaps you observed the fact that the produced paths can sometimes be far from optimal, meaning that they are not always the shortest paths.

The problem with Greedy BFS is that it always expands to the nodes that *appear* to be closest to the goal. This behavoir can mislead the algorithm to expand to nodes that look promising, but ultimately are not due to obstacles further down the road. As a result, paths generated by Greedy BFS can become unnecessarily long. For instance, let’s consider a concave obstacle shown in the image below:



As we can see, the path above (green line) follows promising nodes into a dead end and then has to move out again. This is because Greedy BFS only focuses on following nodes that are closer to the goal, at any cost, even if the path becomes really long. This problem becomes specially noticeable if obstacles are large or concave.

As we can see, **every search method has its advantages and disadvantages**. Keeping track of the distance to the starting point, like Dijkstra does, is slower, but it produces the shortest path; using an heuristic function like Greedy BFS is faster, but we can produce long paths that are not optimal.

Wouldn’t it be nice to combine the best of both methods? How lucky that this is exactly what A* does!

3.7 A*'s special secret

A* is arguably the most popular choice for pathfinding, and for a reason. A* has proven to be both efficient and effective as it is quick and also finds the shortest paths. So you might ask, what's A*'s special secret?

The key factor to the success of A* is that it combines the distance from the start node to the current node, like Dijkstra does, and the estimated distance from the current node to the goal node, like Greedy Best-First Search does.

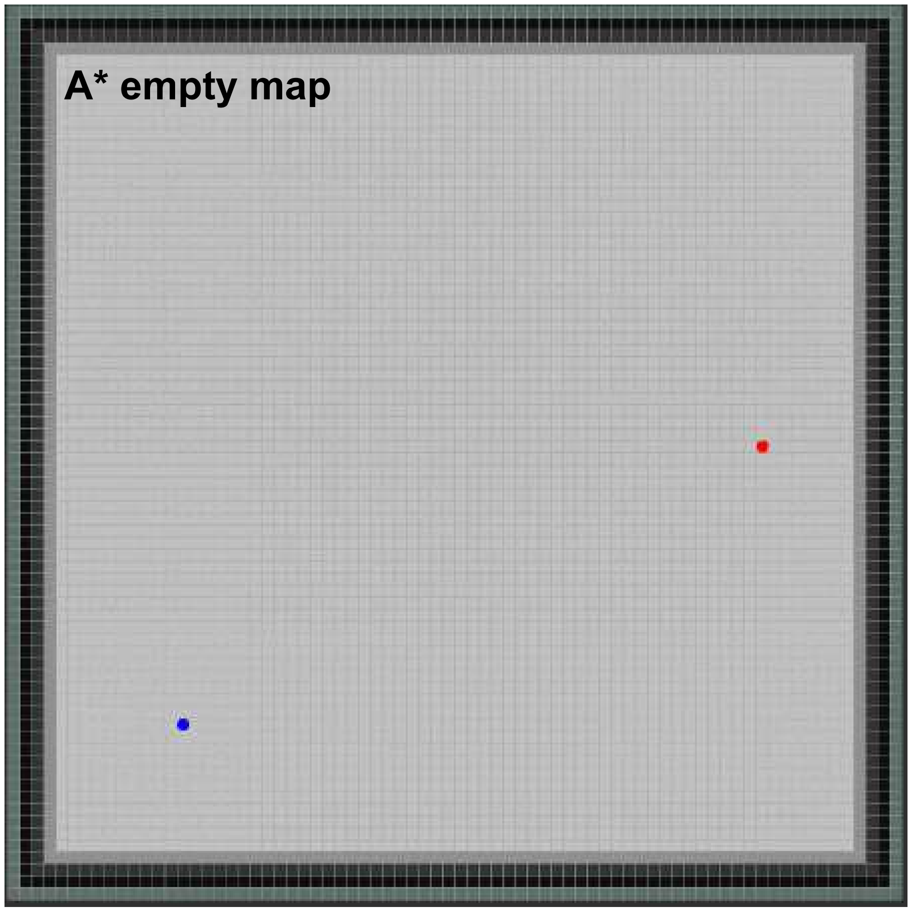
These two pieces of information are combined to create a value known as the **total cost of the node**, which we will denote as `f_cost` .

The total cost `f_cost` equals `g_cost + h_cost` where:

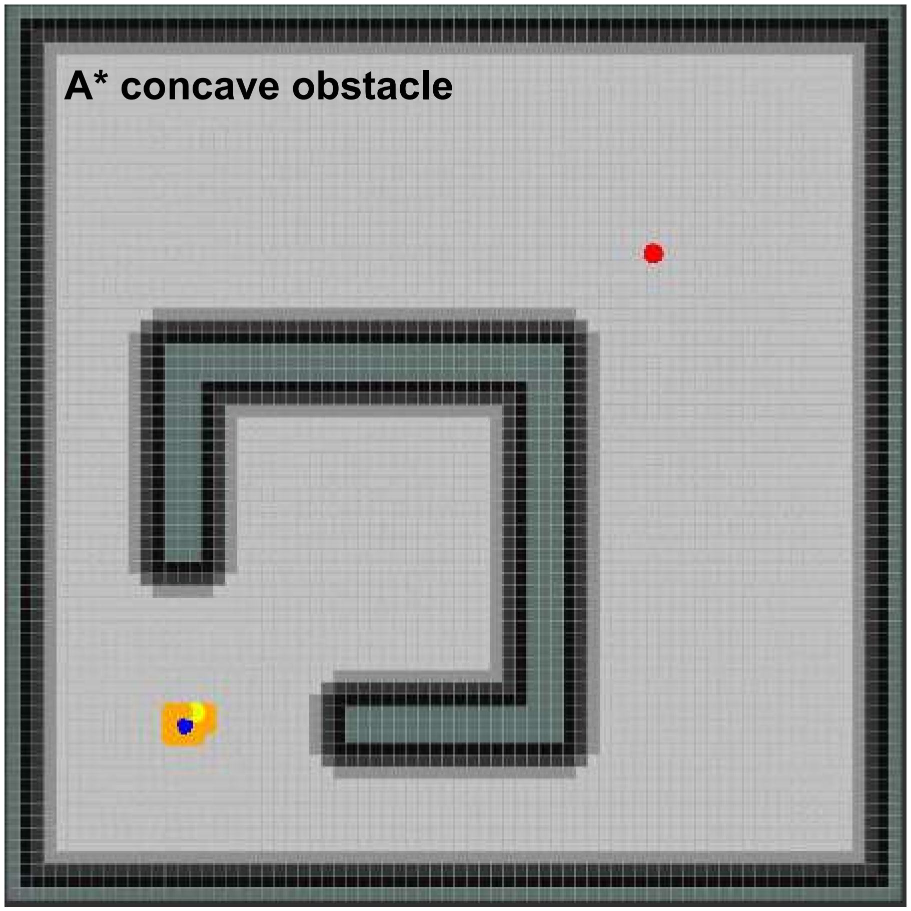
- `g_cost` represents the exact travel distance from the starting point to any node n
- `h_cost` represents an heuristic distance from a node n to the goal location

Basically, whenever A* expands its search to a new node, each candidate is evaluated according to its total cost, given by `f_cost` .
The node with the smallest `f_cost` is selected as the next node to be explored.

Let's take a look at the animation below as it is a perfect showcase of how the algorithm explores the grid:



Let's go back to the example with the concave obstacle to see how A* manages such a situation:



Examine the path found (green line). As you can see, A* is able to work around concave obstacles like the one in this example. Thanks to its `f_cost` , it can completely avoid paths into dead ends and **effectively find the shortest path from start to goal**.

3.8 A* search, full details

Below you will find a complete implementation of A*. As already noted, A* shares many common steps with Dijkstra. If you've been paying attention, these steps shouldn't be hard to follow:

- Start by creating: `open_list` , `closed_list` , `parents` , `g_costs` , `f_costs` and `shortest_path` .
- Set the **g_cost** value for the start node, and determine the heuristic value **h_cost** for start node.
- Put the starting node into **open_list** with its corresponding **f_cost** value.

Enter a condition-controlled loop that runs the instructions below and exits when `open_list` is empty.

- Pick the node with the lowest `f_cost` from `open list` . We refer to it as `current_node` .
- `current_node` gets deleted from `open_list` and added to `closed_list` .
- If `current_node` is the goal, exit the search loop
- Get all neighbour cells of `current_ node` and put them inside a list called `neighbours` .

Iterate over each `neighbour` :

- If `neighbour` is inside `closed_list` , ignore it and get the next neighbour. Otherwise, do the following:
    - If it isn’t part of `open_list` :
        - Add it to `open_list` .
        - Make the `current_node` the `parent` of this neighbour.
        - Keep a record of the **f_cost** and **g_cost** of the node.
    - If it is inside `open_list` :
        - Check to see if this new path is better, using `g_cost` as the measure.
        - If so, change the parent of the node to be `current_node` , and recalculate the `g_cost` and `f_cost` of the node.

When the algorithm exits the main loop:

- Check if the target was found.
- If so, reconstruct the path. Work backwards from the target node, and go from each node to its parent until you reach the starting node.

3.9 Your own A* implementation

- Exercise 3.9.1 -

In this exercise, you will implement A* search and get to see it work.

Go ahead and open the file you will be working on now:

a) Using the IDE, go to the **unit3** package directory.
b) In the `scripts` directory, open the file **unit3_astar_exercise.py**.

Fortunately, we can reuse the code that we wrote for **Greedy Best-First Search**. In fact, this exercise only requires you to tweak the code from the previous exercise to calculate A*'s total cost or **f_cost**. This should be pretty straightforward up to this point as we currently have the **h_cost** from the heuristic function from Greedy Best-First Search and you know how to calculate the **g_cost** from Dijkstra's exercise.

Task:

- Adapt the function `a_star()` from **unit3_astar_exercise.py** in such a way that it performs A* search

Points to consider:

- Use `h_cost` and `g_cost` to calculate `f_cost` .
- Keep track of each node's `g_cost` and `f_cost` values.
- During the algorithm's execution, always pick the node with the lowest `f_cost` as the `current_node` .
- If reaching a neighbour through `current_node` results, if a shorter `f_cost` than the current `f_cost` is known to it, update it's `f_cost` , `g_cost` , and `parent` .

In [2]:
```
# Adapt the function a_star() from unit3_astar_exercise.py in such a way that it performs A* search
```

Click HERE for a hint.

- End of Exercise 3.9.1 -

3.10 Testing A* Search

- Exercise 3.10.1 -

Launch your script with the following command:

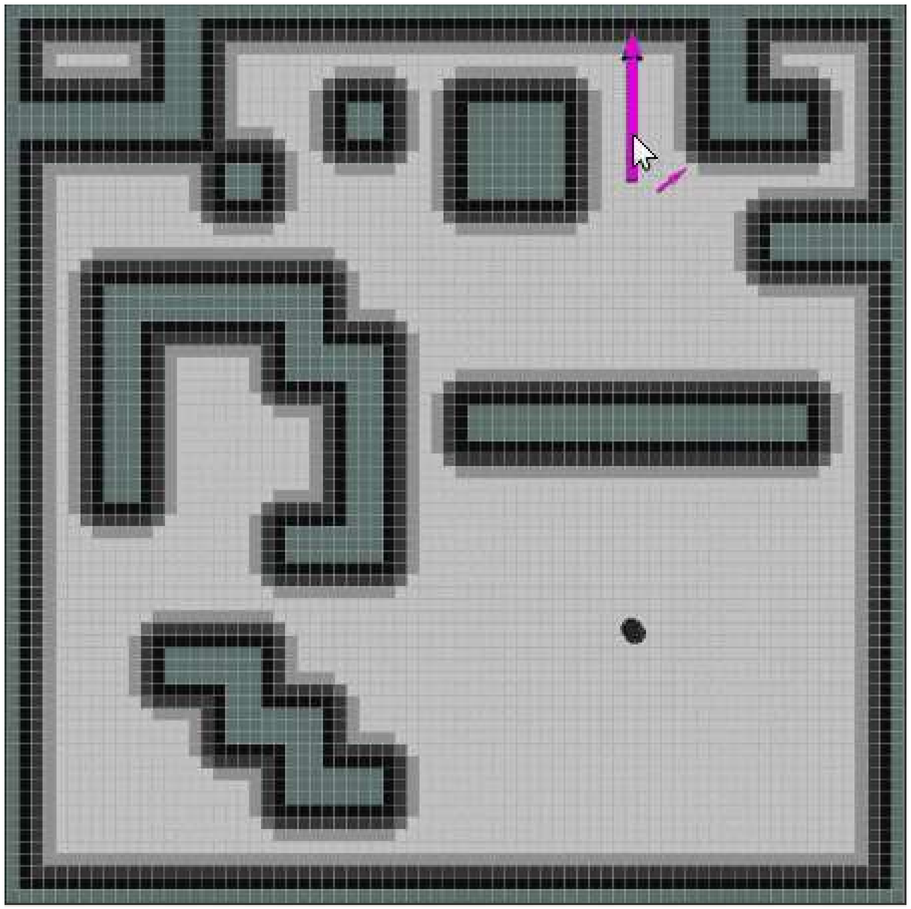
▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit3 unit3_astar_exercise.launch
```

As always, you need to wait until the **Graphical Tools** screen opens for visualizing what is happening.

Once the costmap gets loaded, use the **2d Nav goal** tool a few times to see how A* behaves.

See if you can spot A* characteristics in action. Look either at the statistics or the resulting path.



Also, try to trick your robot into a concave obstacle by placing the robot in front of the concave trap and the goal directly behind it. Confirm that A* avoids bending the path when this is not necessary.

When you're done, you can **Ctrl+C** to stop the execution of the program.

- End of Exercise 3.10.1 -

3.11 A* search limitations

A* is a **complete algorithm**, which is good because it means that it will always find that solution if a solution exists. A* is also an **optimal algorithm**, good again, because this means that it will always find the shortest path if one exists. But these features come at a cost. To find a **complete and optimal** solution, a so-called deterministic algorithm (https://en.wikipedia.org/wiki/Deterministic_algorithm) is required. **A* is such a deterministic path planning algorithm**, which means that it always produces (on a given start, goal and map input) the exact same path, following the exact same computation steps.

Unfortunately, deterministic algorithms do not scale well with the map size. The more nodes to process, the more difficult it becomes to keep up with the planning time requirements. Also, large maps require a lot of memory since each node discovered has to be be accounted for.

Below you will find situations that would normally result in a large map:

- A large area to cover
- A high resolution map
- A high-dimensional space

And all of the reasons above combined.

Want a concrete example: path planning for robotic arms. Normally, we want very precise movements (high resolution!) and because the configuration space has many dimensions (typically higher than 4), we require very large maps.

So, is it possible to create an algorithm that is faster and more memory-efficient than A*?

Yes, but we will have to give up on optimality and completeness to win in computational efficiency. In some cases, it can be convenient, in other cases we have no choice but to make that sacrifice. The next algorithm we look at is called **RRT**, which belongs to a famlily of algorithms called **probabilistic algorithms**. Continue with the next unit to check it out!

3.12 Summary

All right, let's walk through the core concepts of this unit one more time before we move on!

- Uninformed search methods do not take into account how close a node is to the target in order to select where to expand next
- Dijkstra is one example of uninformed search: it picks the node closest to the start as next node, without considering how close it is to the goal
- Informed Search algorithms take into consideration the distance of a node to the goal and use this information when expanding their search
- Informed Search algorithms cannot exist without a function that evaluates the distance to the goal: the Heuristic function
- Greedy BFS uses such an heuristic function; it picks the node with the lowest heuristic value at each iteration, and processes that node
- In general, Greedy BFS is not optimal, that is, the path it finds is sometimes not the shortest one
- A* combines the search strategy of Greedy BFS with Dijkstra's: it always expands to nodes with lowest total cost first
- The total cost value of A* is defined as the sum of the real travel distance and the heuristic value
- A* is optimal, that is, it always finds the optimal path between the starting node and the goal node (given an admissible heuristic function)
- A* is complete, that is, it will always find a solution if one exists (in finite maps)