Introduction to Mobile Robot Path Planning

Unit 2: Dijkstra Algorithm

- Summary -

Estimated time to completion: **2 hours**

In this unit, you will learn about Dijkstra's algorithm for robotic path planning. Specifically, you will:

- Learn basic graph theory concepts
- Understand how Dijkstra's algorithm works behind the scenes with a step-by-step example
- Put Dijkstra's algorithm into code using Python
- Test your implementation on real map data using ROS

- End of Summary -

2.1 Introduction

In this unit, we will get into Dijkstra's super-famous path planning algorithm and apply it to motion planning of a mobile robot. By the end of this unit, your algorithm will guide your robot in a simulated indoor environment while avoiding any collision with obstacles. Along with a theoretically grounded explanation on how the algorithm work, you will implement it yourself, from scratch in Python. Your implementation will even be replacing the default path planning algorithm that comes with the ROS Navigation Stack! This means that you can, if you wish, configure and use the code that you will develop in this unit on a real robot as well.

Okay, well, let's jump right in!

2.2 The problem to solve

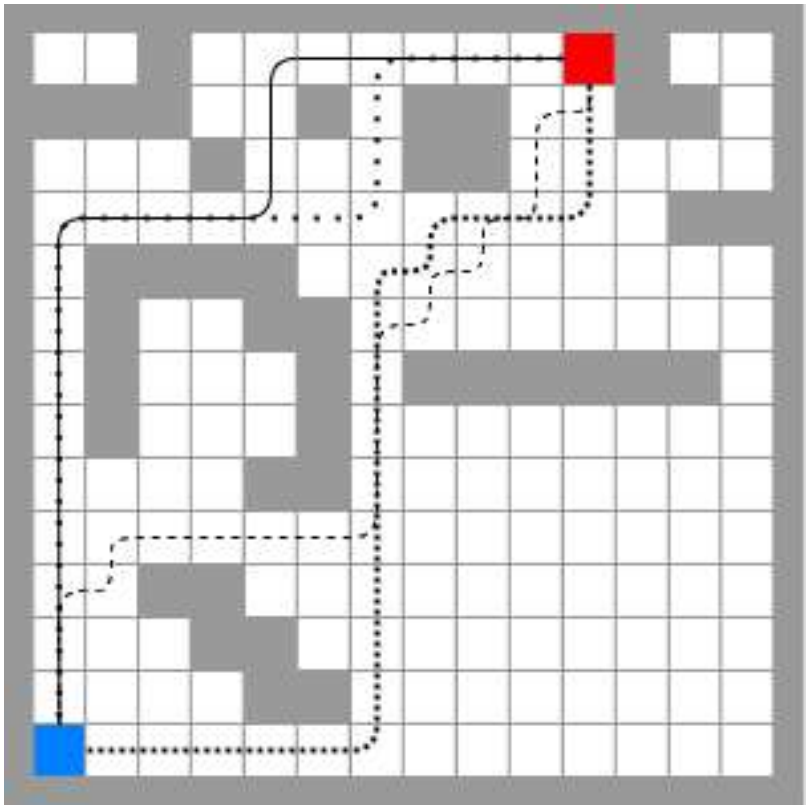
To start, assume our robot has a **occupancy grid map** in advance and is able to self-localize with no error. Recall that an occupancy grid map stores data on free and occupied areas of the robot's environment.

Additionally, the robot's **start position** (where the robot is situated) is known and a **goal location** (where the robot has to go) is also given.

The image below shows an occupancy grid's white cells that represent free regions, and dark obstacles.

From any given grid cell, the robot can only move to adjacent free grid cells and cannot go outside of the map boundaries.

The question we are facing is: what is the sequence of connected waypoints a robot has to follow in order to reach a particular goal location?



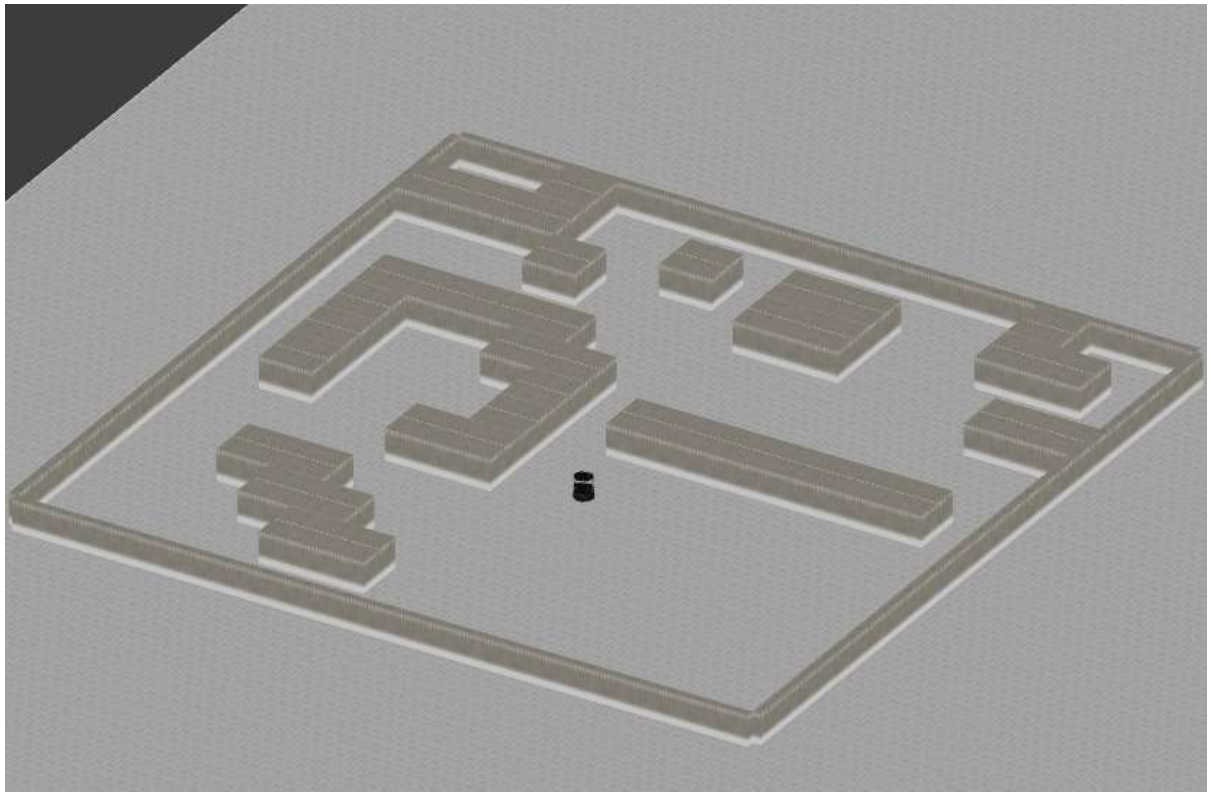
As you can see, there are many ways a robot can reach the red cell from the blue cell in this 14 × 14 grid map. We want the robot to figure out the best way, that is, the shortest path to move to a chosen destination.

Important assumptions:

- The robot has a map
- The robot knows its location in it
- The environment does not change
- The robot can move up, down, left, or right, as long as the space is free

2.3 Let's get started with a preview

In this section, you will get familiar with the simulated environment where you will be testing your implementation of Dijkstra's algorithm.



Notice that Gazebo is already running in the top right corner of your screen.

You can orbit the view by dragging the mouse wheel, which you have pressed. You could also left-click drag while holding the Shift key.

- Exercise 2.3.1 -

First of all, you will need to download and compile the course repository. You can do so by executing the following command sequence:

▶ Execute in WebShell #1

In [ ]:
```
cd ~/catkin_ws/src
```

In [ ]:
```
git clone https://bitbucket.org/theconstructcore/path_planning_course.git
```

In [ ]:
```
cd ~/catkin_ws
```

In [ ]:
```
catkin_make; source devel/setup.bash
```

This repo contains all the required files that you will use during the course.

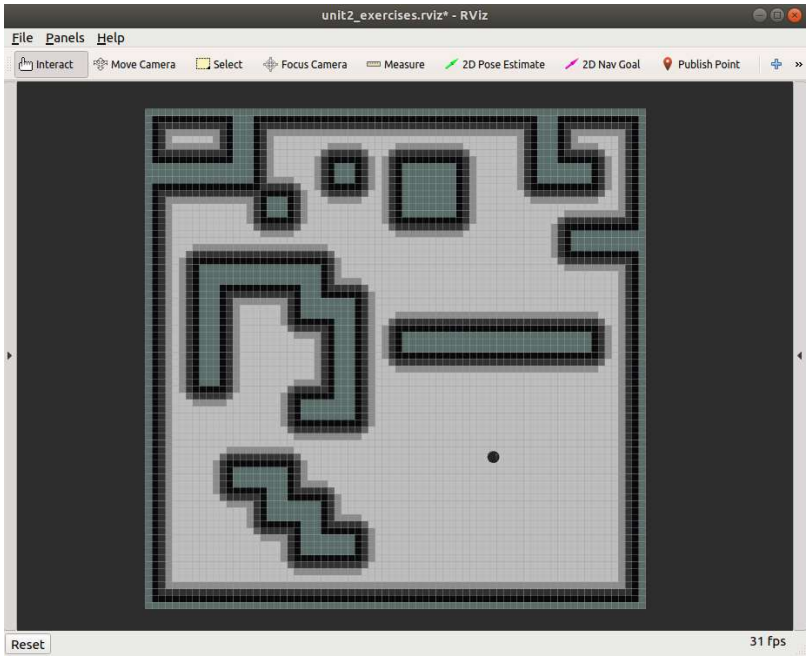
During the rest of this unit, you will execute your path planning code by typing in the command shown below. Go ahead and test it!

▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit2 unit2_exercise.launch
```

After a few seconds a new window with **Rviz** should pop up. If it doesn't, click on the icon with the screen at the bottom bar. As you already know, Rviz allows us to directly interact with the visualized robot and display the planned path.

Please wait for some additional few seconds until you see something like this:



Goal Assignment

Press the **2D Nav Goal** tool from the toolbar, then click anywhere on the map to set the the robot's target position.

Almost immediately, you should see a warning telling you that no path was found.

```
[WARN] [1613380034.148830, 165.496000]: No path returned by Dijkstra's shortes
 path algorithm
[ERROR] [1613380034.546033239, 165.695000000]: Aborting because a valid plan c
ould not be found. Even after executing all recovery behaviors
```

Don't worry, this error message is expected, as we have not implemented our path planning algorithm yet. Our robot is in urgent need of an algorithm able to provide it with a safe, collision-free path. Lucky for the robot, that is exactly what you will develop in the remainder of this unit.

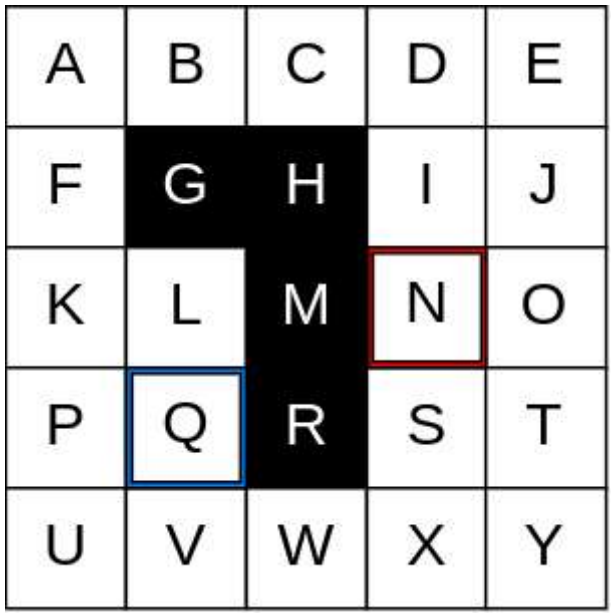
There's no time to waste here, so let's jump right in!

Note: Please shut down the program now by pressing **Ctrl+C** on WebShell #1.

- End of Exercise 2.3.1 --

2.4 Step by step example

Let's walk through Dijkstra's shortest path algorithm with an explained example that will give you some key fundamentals before you start to code your own implementation. For simplicity, this example is shown on a 5 × 5 grid map, with letters that identify each of its cells. Have a look:

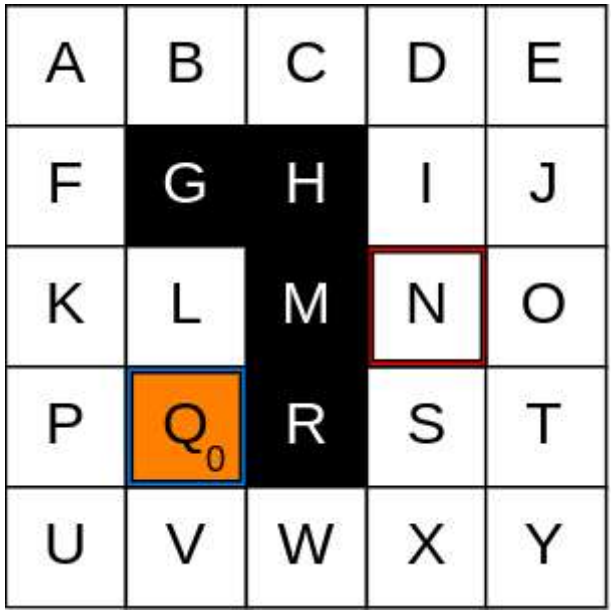


We are interested in finding the shortest path that starts from the starting point Q, marked with a blue outline, to the goal N, marked with a red outline. Black cells are considered to be occupied cells and cannot be traversed.

The algorithm starts

A key component of Dijkstra's algorithm is that it **always keeps track of the shortest distance from the start node to each individual cell**. We will refer to this value as the the **g_cost** of a node and display it on each grid cell's lower right corner. Before the search process starts, the starting point (node Q) gets a **g_cost** of 0 (as its distance to itself is 0). Later, as the algorithm progresses, these values will be updated to the actual shortest distance from the start node (more on that will follow soon).

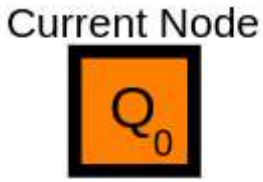
Aditionally, we put node Q into a so-called **open list**. Nodes that are inside the **open list** are the color orange.



Iteration steps:

1) Pick a current node

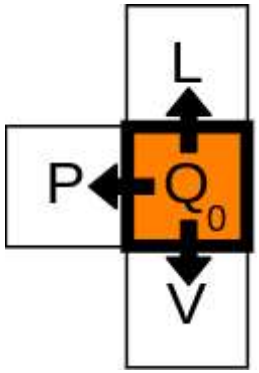
The iterative search process starts by picking the node inside **open list** with the lowest **g_cost**. We call this node **current node**. At this moment, **open list** only contains node Q. We will use a bold black outline to identify the current node in the diagrams.



2) Neighbours of the current node

We continue by examining adjacent nodes that are direct neighbours of the **current node**. In the example, we will only consider traversable grid cells above, below, to the right, and to the left of the current node to be neigbours.

Neighbours of our current node can be identified by the arrows pointing from the current node towards neighbour grid cells that are not an obstacle (grid cells L, V, and P in this example).



3) Update travel distance values, store parent node

Once we have identified all neighbour nodes in free space, we update their **g_cost**. Then, we set the **parent node** of each neighbour. The parent is the node from which the update of the **g_cost** came from (which is always the current **current node**). On all diagrams, we will show each grid cell's **parent node** at the bottom of a grid cell, to the right of its **g_cost**. To conclude, we put each neighbour inside **open list**.

This is a detailed description of how we would proceed in the case of the neighbours of node Q:

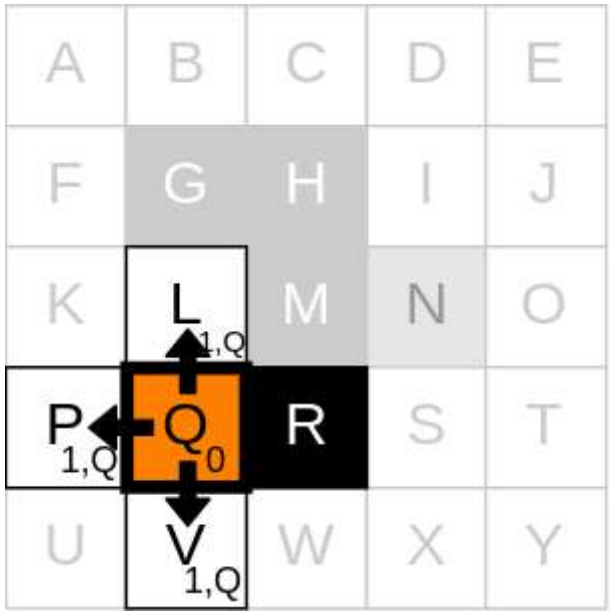
Let's begin with neighbour L, as there is no reason to prefer one neighbour node over another:

- We add the *g_cost* of the current node (in this case, 0) with the **step-cost**. The **step-cost** is the cost incurred when moving to a neighbor cell. In this example, we assume that each grid cells side length is 1, therefore, all step-costs will be equal to 1. We set 1 as the **g_cost** of L.
- We set its parent node, which is Q.
- Now we can put this neighbour inside **open list**.

Brilliant! We continue with adjacent cell V:

- We add 0 (the *g_cost* of Q, our current node) with 1 to obtain 1. Therefore, we set the *g_cost* of V to 1.
- We set as parent node Q.
- Finally, we add V to open list.

OK. Apply the same steps again for P.



Awesome! We are done considering all neighbours of Q, therefore, we can mark node Q as visited. Visited nodes will be kept in a separate list called **closed list** and will be shown in yellow.

After the above steps, Dijkstra's algorithm has completed one full iteration cycle.

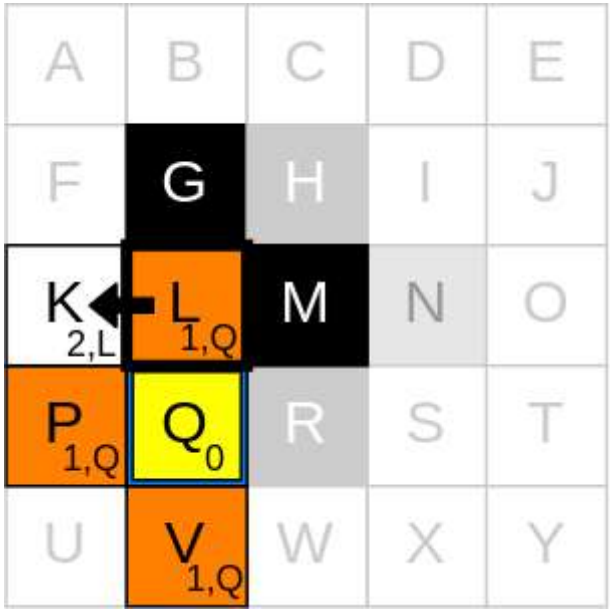
New cycle, repeat 1) pick a new current node, 2) check neighbours, 3) update g_cost and parent node

Now we are at the beginning of a new iteration cycle, so we need to select a new **current node** to explore. We pick a node from inside **open list**. It has to be one of the nodes with the smallest **g_cost** value. That could be node L, V, or P since all have a *g_cost* value of 1. We can pick any of them, so let's pick L and mark it with a bold black outline to identify it as the **current node**.

Now we repeat the process by updating the neighbours of L. We ignore nodes inside **closed list** (in this case, we ignore node Q) and occupied cells (G and M), therefore, we can only process neighbour node K.

This is how we update **g_cost** and **parent node** of node K:

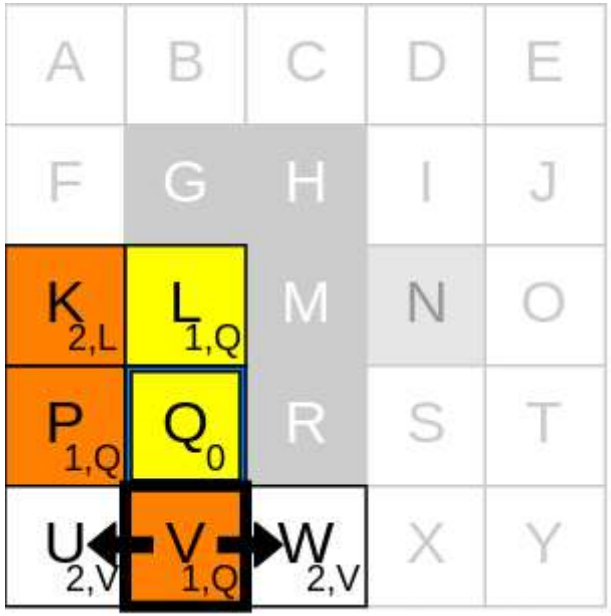
- Add 1 (the *g_cost* of L, our current node) with 1 (the step-cost when moving from L to K) to obtain 2.
- Set the *g_cost* of K to 2.
- Set L as parent node of K.
- We add K to **open_list**.



To wrap up this iteration cycle and prepare for the next one, add L to **closed list**. In the next iteration cycle, L shows up yellow to represent that it has been "visited".

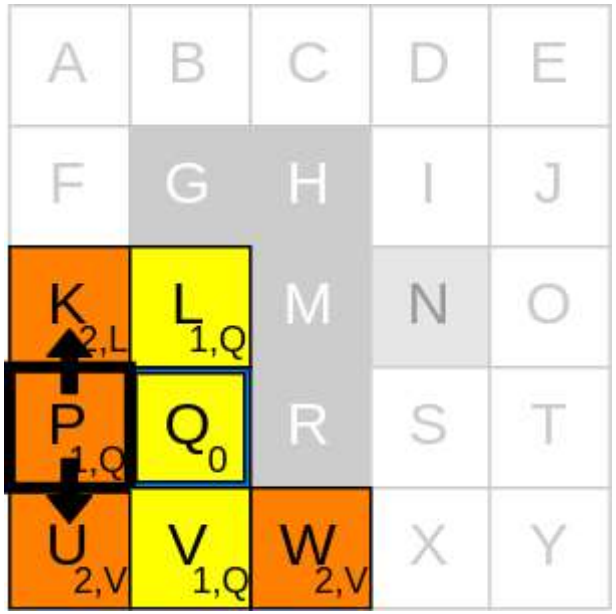
Iteration cycle 3

Pick a new current node to analyze its neighbours nodes. We can take either P or V; both nodes are inside **open list** and have the smallest *g_cost* (1). Take V, for instance, and set the *g_cost* of its neighbours U and W. Both get a *g_cost* of 2 (make sure you understand why) and V as a parent node. Add both U and W to **open list** (they show up orange on the next iteration cycle). Add V to **closed list** (it will show up yellow on the next iteration).



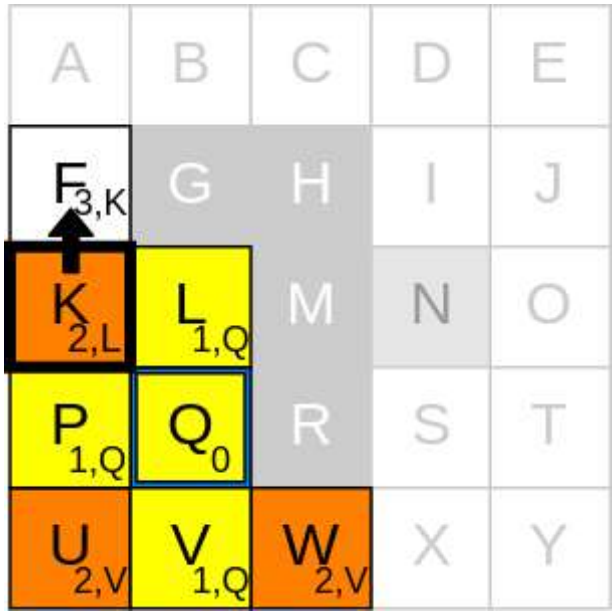
Iteration cycle 4

- Select a **new current** node, in this case, only P is eligible (node in **open list** with the smallest **g_cost** value).
- Notice that both neigbours, K and U, already have a **g_cost** value. In this case, we have to compare if the new **g_cost** is lower or not. As the new *g_cost* is also 2, it is not neccesary to updates K and U's previous *g_cost*. Likewise, the parent node is not updated and remains as it was.
- Finally, mark P as visited by adding it to **closed list**.



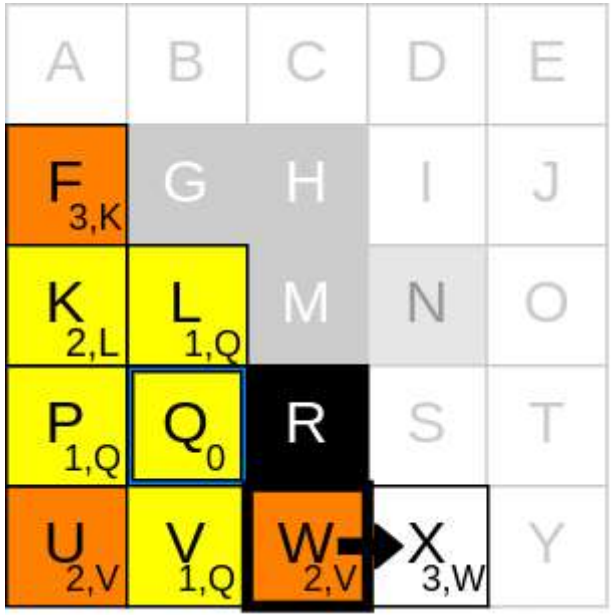
Iteration cycle 5

- Get a new **current node**, in this case K.
- Get the neighbour nodes of K: F is the only non-visited neigbour cell.
- Set F's **g_cost** value: Which we get by adding 2 + 1 = 3.
- Set F's **parent** node (K).
- Add F to **open list**
- Add K to **closed list**.


A
B
C
D
E
F3,K
G
H
I
J
K2,L
L1,Q
M
N
O
P1,Q
Q0
R
S
T
U2,V
V1,Q
W2,V
X
Y

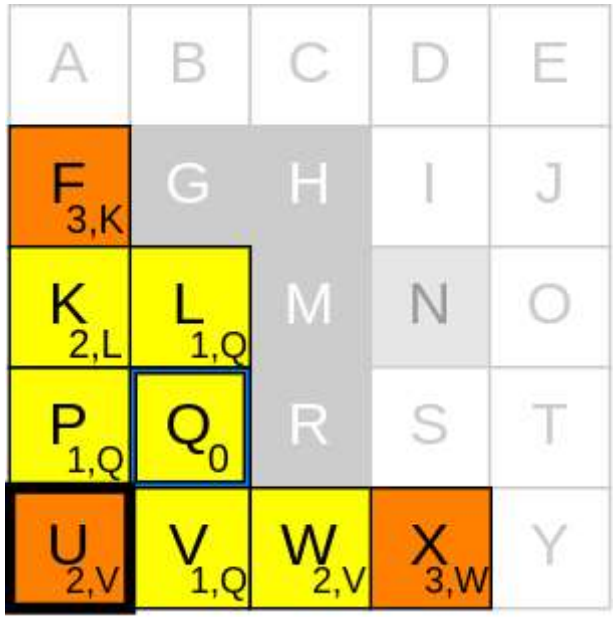
Iteration cycle 6

- Now, the **current node** is W.
- Neighbours: From W we can only visit X, as V has already been marked as visited and, therefore, there is no need to visit it ever again.
- Update the current **g_cost** value of X.
- Set the **parent** node of X, which is W.
- Add X to **open list**.
- Put W into **closed list**.



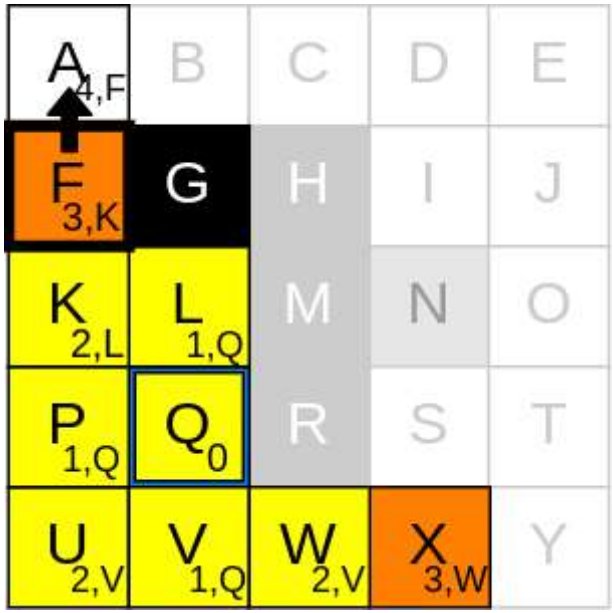
Iteration cycle 7

- Pick a new **current node**: node U is the only candidate node eligible.
- Since U doesn't have any non-visited neighbours, there are no further steps other than adding it to **closed list**.


A
B
C
D
E
F 3,K
G
H
I
J
K 2,L
L 1,Q
M
N
O
P 1,Q
Q 0
R
S
T
U 2,V
V 1,Q
W 2,V
X 3,W
Y

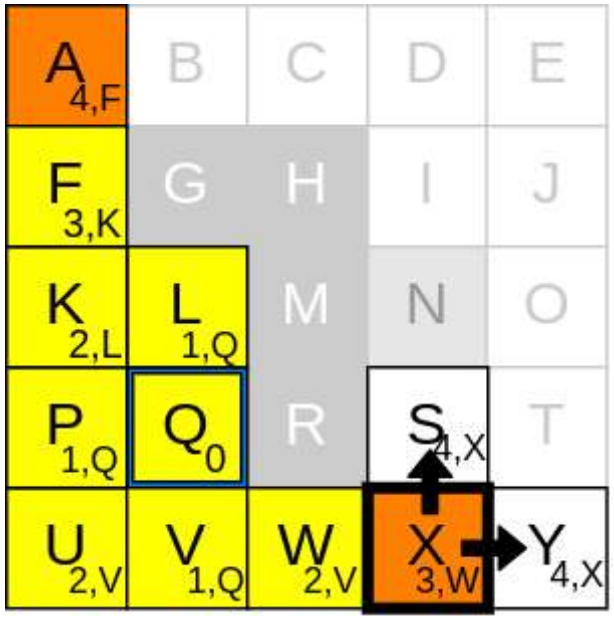
Iteration cycle 8

- Select a new **current node** from among the candidate nodes F and X.
- Say we pick F, from which we can only access neighbor A.
- Update A's current **g_cost** to 4.
- Set the **parent** of A, which is F.
- Add A to **open list** and F to **closed_list**.



Iteration cycle 9

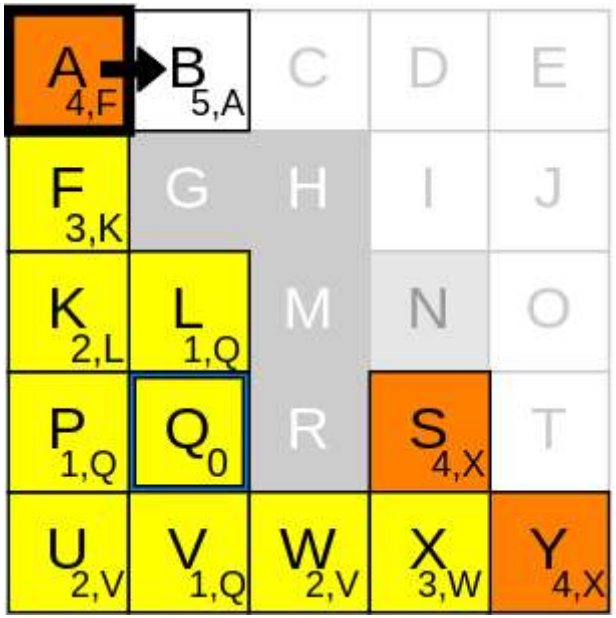
- Pick a new current node: we are now at X.
- Find the neigbours (S and Y) and set their corresponding **g_cost** and **parent**.
- Mark X as visited and put S and Y into **open list**


A 4,F
B
C
D
E
F 3,K
G
H
I
J
K 2,L
L 1,Q
M
N
O
P 1,Q
Q 0
R
S 4,X
T
U 2,V
V 1,Q
W 2,V
X 3,W
Y 4,X

Iteration cycle 10

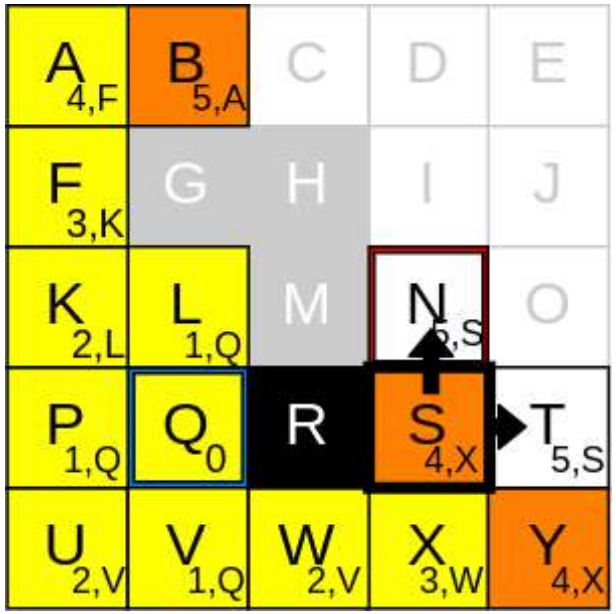
Almost there. We could pick for the next node A, S, or Y.

- Take A and set B's **g_cost** to 5 and its **parent** node A.
- Then mark A as visited and put B into **open list**



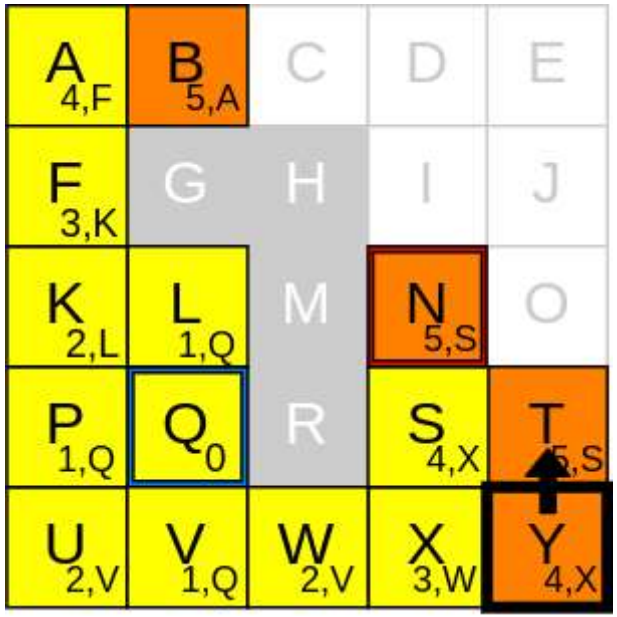
Iteration cycle 11

- Set S as the **current node**.
- Check each neighbour of S, and for each one, set the **g_cost** and **parent**.
- Add S to **closed list** and its neigbours into **open list**.


A 4,F
B 5,A
C
D
E
F 3,K
G
H
I
J
K 2,L
L 1,Q
M
N
O
P 1,Q
Q 0
R
S 4,X
T 5,S
U 2,V
V 1,Q
W 2,V
X 3,W
Y 4,X

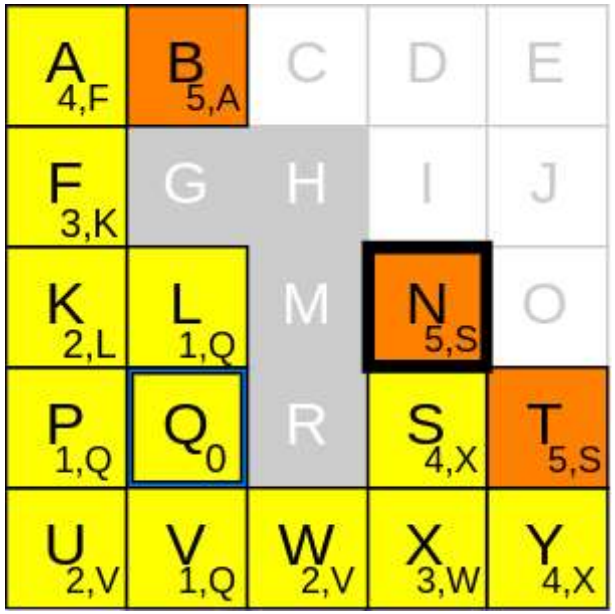
Iteration cycle 12

- Set Y as the **current node**.
- We compare the new **g_cost** of T with its previous value, and since the new value is not lower, there is nothing to update other than putting Y into **closed list**.



Iteration cycle 13

- Set N as the **current node**.
- Voilà! We have found node N and our goal!


A 4,F
B 5,A
C
D
E
F 3,K
G
H
I
J
K 2,L
L 1,Q
M
N 5,S
O
P 1,Q
Q 0
R
S 4,X
T 5,S
U 2,V
V 1,Q
W 2,V
X 3,W
Y 4,X

Now you think you are done? Still not yet.

Dijkstra's algorithm is divided into two phases:

1. Explore the map to collect data
2. Utilize the collected data to build the shortest path

At the moment, we have finished phase 1.

By examining neighbors and deciding which nodes are parent to which other nodes, we build a representation of the connectivity of the free space. We also confirmed that a path to the goal location exists. However, we have not yet built the sequence of nodes that a robot would need to visit to move from start to goal, so let's do that now.

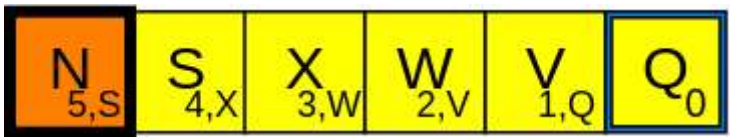
Building the shortest path

Once we have reached the target node, we can extract the shortest route by following every single node's parent node until we reach the start node.

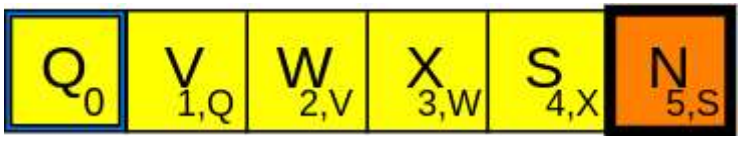
We first take the target node N and add it to a new list. Then we look up the parent for that node. We add node's N parent to the list, which is S. In the same way, we pick the parent's parent node and store it, too. In this case, the parent of S is X, and we add X to the list. We keep iterating this process until we get the start node Q, which has no parent node.

Can you see why it is important during Phase 1 to keep track of each node's parent node?

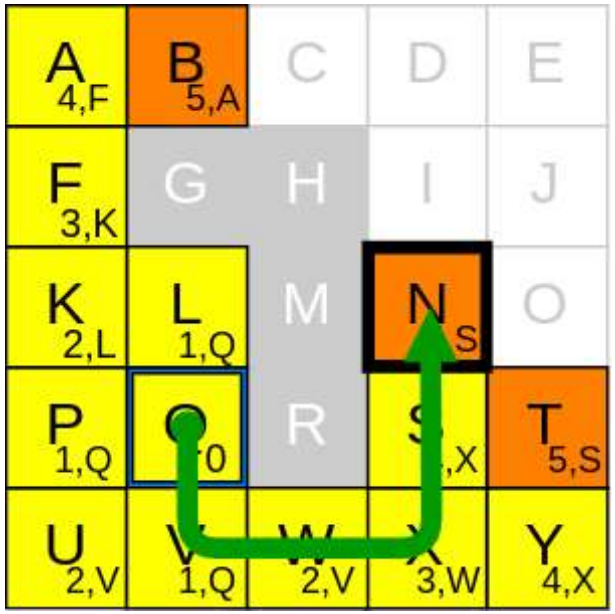
Once the backtracking is complete, the newly created list will contain the shortest path as a sequence of grid cells to visit, but in the wrong order:



To have the path from start to end, so all we need to do is reverse it:



Now we are done with the path finding process!


A 4,F
B 5,A
C
D
E
F 3,K
G
H
I
J
K 2,L
L 1,Q
M
N S
O
P 1,Q
R
T 5,S
U 2,V
1,Q
2,V
3,W
Y 4,X

2.5 Reflection on Dijkstra's core elements

Before continuing, let me highlight a few key aspects and terminology of the process that we just went through.

Path planning is an exploration process. It solves the shortest path problem by searching through connected nodes.

During this search, the algorithm mantains one node as the **current node**, which changes as the search progresses. The algorithm expands out from the start node into newly discovered nodes that are put in a list of nodes that can be explored next. This step is usually called **find neighbours**. Another quite common way of referring to the same concept is *expansion*, *propagation*, or *generation of successors*. Basically, **neighbours** (successors or children) are all nodes that are accessible from the current node.

Lists are also important. There are two lists that we need as bookkeeping tools:

- Open list (also known as *unvisited nodes* or *frontier*)
- Closed list (or visited nodes)

Nodes on **open list** have been *discovered* as **neighbours**, but have not yet been selected as the **current node**.

Nodes on **closed list** have been completely processed. They sometimes are called *visited nodes* because they were once inside *open list*, have completed all the process steps as the *current node*, and are then pushed to *closed list*. A *closed list* is used to avoid getting trapped into loops or repeating work, which causes an excessive search time overhead.

Don't worry if you get lost a little bit at the beginning when confronted with these new concepts. Hopefully, you should start to follow along as we go through the coding exercise and explain how they are applied. Now let's review a few more terms:

To keep track of the shortest path, we use two variables: **g_cost** and **parent node**.

The **g_cost** is the cost of moving from the start to a particular node. It is computed by adding the **g cost** of a **parent** to the cost of moving from the parent to the node, the **step-cost**. In the example, we used a fixed step-cost of 1 because we assumed that each grid cell's side length is 1. A lower **g_cost** means a better path. Dijkstra uses the lowest **g_cost** as the decisive criteria to pick a new **current node** out of **open list**.

Each node on the open list keeps a record of its **parent node**. A parent node is the one who first *discovered* the node in question. Each time a newly discovered node is added to *open list*, it also gets the current node as its parent. Parent nodes are required to build the final path, but of course, this is done after the search has finished.

As we said, the best way to familiarize yourself with these terms is to get your hands dirty. But first, let's have a quick look at a more formal algorithm description, before we move on to our friend the code editor.

2.6 The generic Dijkstra algorithm

Here's a short description of the process:

1. Mark your initial node with a **g_cost** of 0, and add your initial node to **open_list**

Phase I:

Repeat the following while **open_list** is not empty:

1. Extract the node with the smallest **g_cost** from **open_list** and call it **current_node**
2. Mark it as visited by adding it into **closed_list**
3. Check if **current_node** is the target node, and if so, go to phase II; otherwise, continue with step 5
4. Find the **neighbours** of the **current_node**

For each node in the list of **neighbours** of **current_node**:

1. If a neighbour is inside **closed_list**, skip it and pick the next neighbour

1. If a neighbour is inside **open_list**:

    - If the new **g_cost** value is smaller than the current **g_cost** value:
        - Update its **g_cost**
        - Update its **parent** node

1. If a neighbour is not inside **open_list**:

    - Set its **g_cost**
    - Set its **parent** node
    - Add it to **open_list**

When we are done considering all **neighbours** of **current_node**, go to step 2.

Phase II:

Build the path from start to end.

1. Trace back from the target node to the start node, using each node's parent node.

2.7 Dijkstra's Algorithm in Python

We started this unit with a walk through the inner workings of Dijkstra and then described important concepts. Now we will turn that theory into practice. To get you started, we have provided you with the code necessary to integrate your own global path planner into the ROS Navigation Stack.

You can find the file where you will write your code here:

1. Use the code editor window to go into the folder `catkin_ws/src/path_planning_course/unit2`
2. Open `/scripts` and then open the file **unit2_exercise.py**.

Your task will be to complete the `dijkstra()` function.

Once completed, this function will return a list containing the indices of all nodes that connect the shortest path from start to goal or return an empty list if the path does not exist.

In order to split the implementation into smaller parts, we have organized the exercises into the following individual fragments:

- Algorithm start
- Main loop over open list
- Secondary loop over neighbouring nodes
- Reconstruction of shortest path

In all exercises that follow, insert your code directly into **unit2_exercise.py**. Make sure you save your progress often.

All right, let's start!

- Exercise 2.7.1 -

Algorithm start

The first step is to set the variables and data containers that our algorithm will require. These are the following:

- `open_list` : an empty list that will store open nodes as a sublist using this format: [index, g_cost]
- `closed_list` : an empty set to keep track of already visited/closed nodes
- `parents` : a blank dictionary for mapping children nodes to parents .
- `g_costs` : a dictionary where the key corresponds to a **node index** and the value is the associated **g_cost**.

At this point, we only have one node that we can work with: *start_index*.

- Set the start's node **g_cost**: update `g_costs` using *start_index* as key and 0 as the value (as the travel distance to itself is 0.
- Put *start_index* along with it's **g_cost** into **open_list**.

  Note that besides of a node's **index** we are also adding it's corresponding **g_cost** into **open_list**, but why? We do this because we want to be able to sort all nodes inside **open_list** by their lowest **g_cost** value. You will find more details on how we do that in the next exercise!

  Ok, but why are we storing the **g_cost** again on a separate dictionary? After all, it is already stored inside **open_list**. Answer: to make the algorithm more efficient. With a dictionary, we can get a node's **g_cost** very fast, without having to iterate over **open_list**, which is pretty time consuming.

- To conclude this exercise, create `shortest_path` as another empty list that we will use to hold the result of the search.
- Finally, create the variable name `path_found` and assign `False` to it. We will use this boolean flag later on to know if the goal was found or the goal was not reachable.
- If you want, you can finish the initialization step with a message on the console using this syntax: `rospy.loginfo('Dijkstra: Initialization done!')` .

Also, if you want to test your code for any errors, replace the `pass` statement and make the function return the empty list `shortest_path` .

Click HERE for a hint.

In [1]:

```
def dijkstra(start_index, goal_index, width, height, costmap, resolution, origin):
    '''
    Performs Dijkstra's shortest path algorithm search on a costmap with a given start and goal node
    '''

    #### To-do: add required variables/data containers below ####



    pass
```

- End of Exercise 2.7.1 -

- Exercise 2.7.2 -

Main loop over open_list

Now let's add some of the core functionality of Dijkstra, and begin implementing the main loop.

To complete this exercise:

- Create a loop that will continue until `open_list` is empty

  Note that if `open_list` is empty, then we will have traversed all nodes, meaning that the target node is not reachable from the starting node.

Now we want to select the node with the lowest `g_cost` as **current node**. We proceed in two steps:

- Sort `open_list` ordering the nodes by `g_cost` in an ascending order, meaning smallest to largest.
  You can use the syntax below in order to sort according to the second element of each sublist inside `open_list` :

  ```
  open_list.sort(key = lambda x: x[1])
  ```

- Now that `open_list` is sorted, pop the first node from `open_list` and set it as the **current node**.
  Use this sytanx to pop the first element inside *open_list* and get the node index (which is the first element in the sublist):
  `open_list.pop(0)[0]`

  Now let's hit pause for a second to pay closer attention to the last two tasks.
  One of the most fundamental characteristics of Dijkstra is that it expands its search considering the nodes that are closer to the start node first. In this implementation, we sort `open_list` each time we want to pick a new **current_node**. With *open_list* sorted, the next node to explore will always be its first element. There are other, faster ways that optimize this process, but here we stick to a version based on sorting as we think it simplifies the understanding of the underlying process. All right, let's continue, shall we?

- Now add `current_node` into the set `closed_list` to prevent it from being visited again.

- Optional: visualize closed nodes.
  To add a graphical representation on the path searching process, add this line of code:
  `grid_viz.set_color(current_node,"pale yellow")`

- Then add a conditional statement to exit the main loop if the goal node has been reached. In such a case, set `path_found` to `True` , otherwise, the main loop execution should continue.

Okay, we need one more line of code to finish this exercise:

- As the last task, call the provided function `find_neighbours()` and save the result into a variable named `neighbours` which we also need for the next exercise. `find_neighbours()` identifies neighbor nodes in free space. It takes in the index of the current node, the map width and height, the costmap, and the map's resolution as arguments. It returns a list with neighbour nodes as [index, step_cost] pairs.

In [1]:
```
# Tasks:
# create a loop that executes as long as there are still nodes inside open_list
# sort open_list in order to get the nodes ordered by the lowest 'g_cost' value
# get the first element of open_list and call it current_node
# add current_node into closed_list
# add a conditional statement to exit the main loop if the goal node has been reached
# get the g_cost corresponding to current_node and assign it to a variable named 'g'
# call the provided function find_neighbours() and save the result into a variable named 'neighbours'
```

- End of Exercise 2.7.2 -

- Exercise 2.7.3 -

Secondary loop over neighbouring nodes

So far, we have picked a **current_node** but have not yet iterated through each of its neighbors. In this exercise, you will write a second loop to do just that.

Proceed as follows:

- Invoke a loop over all elements in `neighbours`
- Unpack each neighbor into `neighbor_index` and `step_cost` for later use
- Check if `neighbor_index` is in `closed_list` , and if so, skip the current loop iteration and proceed with the next neighbour

- Next, determine the cost of travelling to the neighbor. We need it for the subsequent steps.
  Name `g_cost` the travel cost required to get to `current_node` plus `step_cost` (the cost of moving to the neighbor).

- Now verify if `neighbor_index` is currently part of `open_list` . If found, keep track of the position where the [index, g_cost] element appears as you will need it to update it as described in Case 1 below:

- Case 1, if the neighbor is already inside `open_list` :
  - Verify that `g_cost` is better than the neighbor's current `g_cost` value, if so:
    - Update the node's *g_cost* inside `g_costs`
    - Modify its *parent* node, set it to `current_node`
    - Update the node's *g_cost* inside `open_list`

- Case 2, if the neighbor is not part of `open_list` :
  - Set the node's *g_cost* inside `g_costs`
  - Set `current_node` as its parent node
  - Add the neighbor into `open_list`
  - Optional: add this line of code to visualize the frontier:
    `grid_viz.set_color(neighbor_index,'orange')`

  If you have any doubts why we have two cases that need to be treated differently:

  A neighbor that is in `open_list` has already been visited earlier. Therefore, we have to compare the new calculated *g_cost* to the *g_cost* we already have. We update the *g_cost* and the *parent* only if we discovered a lower *g_cost* value.

  A neighbor that is not in `open_list` gets a *g_cost* and *parent* for the very first time. It also must be added to `open_list` in order to be part of the possible candidates the next time we pick a new `current_node` .

To close this exercise, you can print a message to the console when the main loop has exited (optional):
`rospy.loginfo('Dijkstra: Done traversing nodes in open_list')` .

Thats all for the path search process. You have done a magnificent job!

Click HERE for a hint.

- End of Exercise 2.7.3 -

- Exercise 2.7.4 -

Reconstruction of the shortest path

You have now reached the final exercise!

When the main loop exits, either:

- The search failed to find a path to the goal, or
- You have found the goal node

Task 1:

- If the goal node was not found, exit `dijkstra()` by returning an empty list.
  Optional: add a quick message to alert the user using this syntax: `rospy.logwarn('Dijkstra: No path found!')`

If the goal node was found, we still need to build the path from start to goal by using the information collected during the search.

The path reconstruction process is quite straightforward: Start with the target node and add it to the path. Find its parent node and add it to the path. Then add the parent's parent node, its parent, etc. until we reach the start node, which is also added to the path. Note that since we added nodes from the target moving to the start node, the order will be reversed, so you will have to reverse the order before returning the path.

Tasks 2:

- If the goal node was found:
  - Declare a variable named `node` and assign the target node to it
  - Enter a loop that keeps going until `node` is equal to `start_index`
  - At each iteration step:
    - Append `node` to `shortest_path`
    - Get the next node by re-assigning `node` to be the node's `parent` node
  - Once the loop exits, reverse `shortest_path` to get the correct order
  - Optional: add a message to inform the user : `rospy.loginfo('Dijkstra: Done reconstructing path')`

In [ ]:
```
# If the goal node was found:
# - declare a variable named 'node' and assign the target node to it
# - Enter a loop that keeps going until 'node' is equal to 'start_index'
# - At each iteration: append `node` to `shortest_path`, re-assign `node` to be its `parent`
# - Once the loop exits, reverse `shortest_path` to get the correct order
```

Click HERE for a hint.

- End of Exercise 2.7.4 -

Great work! You've implemented your first path planning algorithm. So, let's wait no more and get ready to test it on real map data!

2.8 Testing

Now it is time to test your own implementation using ROS and Gazebo.

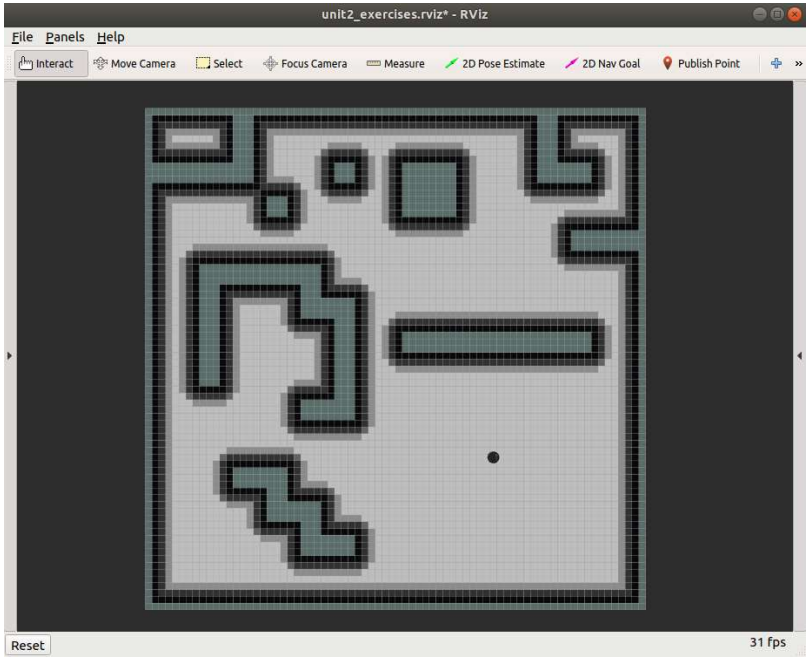
Launch the file you created with the following command:

▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit2 unit2_exercise.launch
```

You should see the **Graphical Tools** screen open. If it doesn't, click on the icon with the screen at the bottom bar.

Please wait a few seconds until you can graphically visualize the robot model and the occupancy grid map.



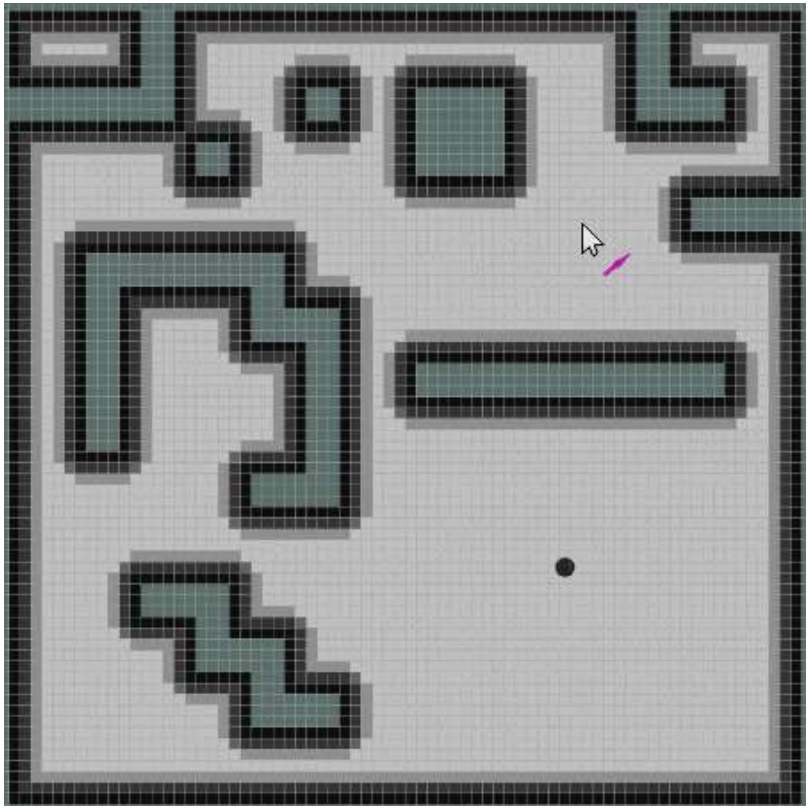
Go ahead and use the **2d Nav Goal** tool to watch your robot plan and follow a path to the commanded goal position.

Here are some ideas you may want to try out:

- Try target locations far away from the robot's current position
- Set a target very close to your robot's current position
- See what happens when your goal position is behind your robot
- Set a new goal before the navigation to the previous goal has finished
- Click on an obstacle to start the path finding
- Select a target point in free space but outside of the robot's reachable space

Happy testing!

Expected result



If the planner failed to find a path due to the goal being set on an obstacle or outside of the map, you will see an error message written to the console, which is the expected behavior.

When finished testing, press **Ctrl+C** to shut everything down.

2.9 Limitations

Dijkstra's algorithm is an **uninformed search** algorithm, also known as blind search. It means that during the expansion process, it doesn't know if one grid cell is better than the other. This causes the algorithm to search in all directions in a radial pattern originating from the starting node. This makes it generally slower than informed search strategies (which you will learn in the next unit), especially as the map size goes up.

Because Dijkstra expands to a very large number of nodes, it is a very **cpu-intensive** algorithm. This can result in a path search process that is slow or fails to find a path in time.

Additionally, the further the search expands, the more memory is needed. This can sometimes be a problem since **memory requirements** increase exponentially as the map size or space dimensionality increases. One important source of memory usage is the open and closed list.

Adding to that, Dijkstra assumes that the environment is **static** and not dynamic. This means that Dijkstra produces a rigid path. As a result, if the environment changes, the robot could be moving along a path that is potentially obsolete. One possible solution to this issue is to continuously recalculate the path with an updated map.

Also keep in mind that possible constraints, such as starting orientation, minimum turning radius, velocity, and acceleration limits, among others, have not been taken into account.

2.10 Summary

I hope you found this unit helpful and were able to understand the mechanics of Dijkstra.

Before continuing, let's review some of its key aspects one last time:

Dijkstra is one of the most important algorithms to solve the path planning problem. One of its charateristics is that it always finds the shortest path if that path exists. By gradually expanding amd exploring adjacent grid cells, it is able to create a structure of connected free space from the start node outwards.

However, this process requires some care in order to keep a correct registry of parent nodes and travel distances, as well as to avoid visiting already explored nodes twice. This is why the usage of an open and closed list is critically important. Additionally, we took care to keep the open list sorted by the travel distance to each node, to easily retrieve the node with the lowest travel distance, one node at a time.

Then after the ground structure is built, we need to build a path, that is, extract the sequence of nodes that comprise the shortest path. This is done by working backwards, from the target node until the start node, and then reversing the sequence found.

Now, Dijkstra's shortest path algorithm is really great, but it also has some limitations, such as the processing speed, and in robotics performance, this is also really important. In the next unit, you will learn how Dijkstra can be modified to make it faster, thus, transforming it into A* (A-Star).

See you in the next unit!