49274 Space Robotics (Spring 2025)

# Assignment 2: Path Planning for a Planetary Rover

- **Due date:** 11:59pm, Friday 12 September 2025 (week 7) via Canvas
- **Total marks:** 25 (25% of the total mark of the subject)
- **Submission:**
  - ✓ A written report that briefly describes the algorithms and the results achieved.
  - ✓ A short video that shows the algorithms working. Please upload this to YouTube or Google Drive and provide a link in your report.
  - ✓ Your solution source code files will also be submitted to Canvas. Make sure to include all python files that you edited (such as in a zip file). Please note that all code will be checked with a similarity software, and highly similar code can result in zero marks.
  - ✓ You may be asked to demonstrate your understanding to the Subject Coordinator.
- **Specific instructions and marking criteria are provided on the last page of this document.**

## Introduction

Your goal in this assignment is to implement a path planning system for a mobile robot that is navigating a planetary environment. Path planning describes how a robot figures out how to move from the current location to a goal location, while considering several different objectives. Path planning is an essential component of robotic systems since a robot needs to be able to calculate paths for safely and efficiently reaching the next desired location.

The scenario for this assignment is illustrated in the picture below. A mobile ground robot is navigating to a location on the other side of the environment. The robot has access to a 3D terrain map of the area, that may have been generated by an aerial robot or during previous exploration of this area.

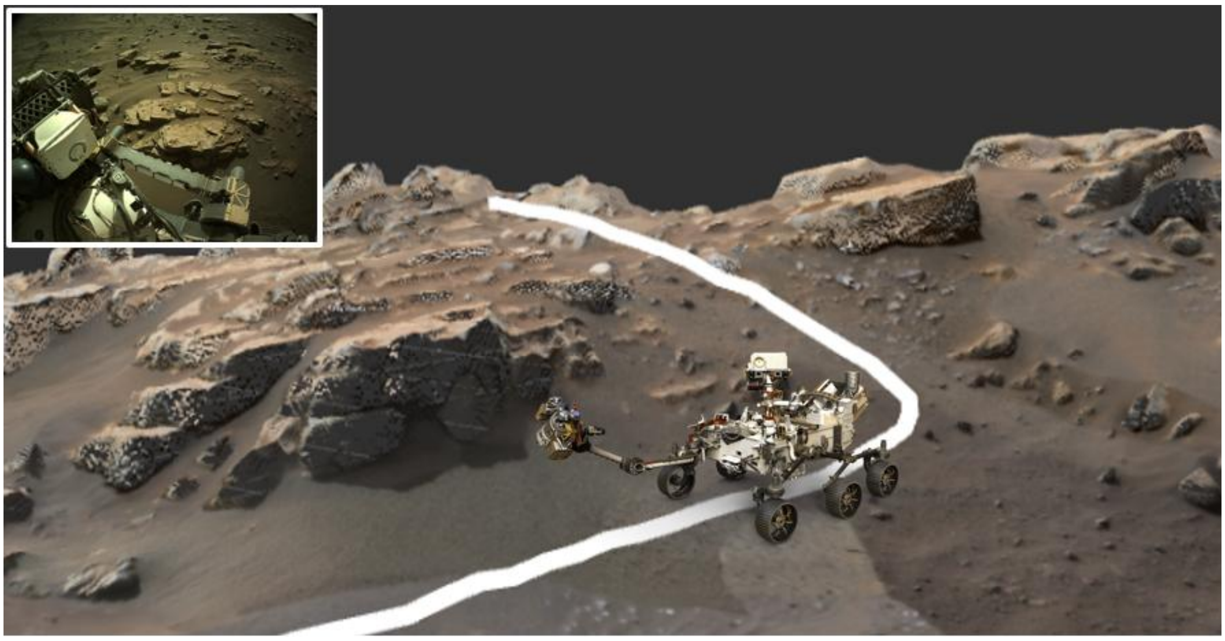


*Figure 1: Illustration of the scenario. The 3D terrain map was generated from actual images recorded by the Perseverance Mars Rover (top left) while investigating the "Bastide rock formation".*
*Dataset: [Sketchfab link], Information about the mission: [NASA link]*

The robot is to plan a path to a specified goal location while satisfying several objectives:

- The robot must **avoid crashing into obstacles**,
- The robot must **avoid traversing very steep terrain**,
- The path is to be the most **energy efficient**, which requires finding paths that have both short travel distance and no unnecessary hill climbing,
- The path planning algorithm needs to be **robust** enough to be able to plan paths to any location in the environment without error, especially narrow passages, and
- The path planning algorithm is to be **computationally efficient**.

## Path Planning

**Path planning** is essential for mobile robots. Given a known map, a start position, and a goal position, the aim of path planning is to find a path from the start to the goal while avoiding obstacles and optimising an objective function. Dijkstra's (pronounced "dike-struh") algorithm and A* ("A star") are two important optimal path planning algorithms.

More generally, Dijkstra's algorithm and A* are known as **graph traversal algorithms**. A graph is a set of nodes (also called vertices), and edges that connect pairs of nodes. Dijkstra's algorithm and A* are used to search through the graph to find the optimal (shortest) path between two nodes.

A navigation graph can be constructed in many different ways. The simplest case is a **grid map**, and a more sophisticated technique is the **Probabilistic roadmaps (PRMs)**; we will try out and compare both of these in this assignment. PRMs involve randomly sampling the free space of the environment and connecting edges between vertices that are "visible" to each other. Compared to grid maps, PRMs are usually sparser (and therefore faster to search over) and more suitable in high-dimensional configuration spaces.

**The figures on the following page show what we are aiming for at different stages.** In all three cases, we are searching for a path through the Mars environment that was generated from images taken by Perseverance.

**In Figure 2**, a uniform grid is generated over the environment, then a path is searched for from the bottom-right to the top-middle. The parts of the environment shaded in red are considered obstacles since there is a large gradient in the terrain, such as where there are rocks. The colours you see in the grid map represent the stages of the A* graph search: blue is the "visited set" that has already been searched, green is the "unvisited set" that lies at the boundary of what has been searched, and red are nodes that have no yet been searched. The green line represents the path found through the graph, and the white line is the same path after some simple path "smoothing".

**In Figure 3** is a PRM graph of the same environment. You can see that the PRM has a similar coverage of the world with much fewer nodes. However, in this example, it is not quite as well connected, meaning that the generated paths might sometimes be longer, or not found at all.

**In Figure 4**, a uniform grid is again generated over the environment, but in this case each edge cost estimates the energy cost of traversing from one location to another. This means that the path planner won't necessarily select the shortest path according to Euclidean distance, but instead will balance this objective with avoiding unnecessary hill climbing.

**Figure 5** introduces a distance transform map, which indicates the distances to the nearest obstacles: red is in an obstacle, black is close to an obstacle, and white is far away from all obstacles. The distance transform reveals features of the geometry of the environment. This is then used in a new PRM sampling method to find a more effective roadmap over the environment, allowing a Mars rover to more reliably navigate through narrow passages.

**These four figures illustrate some of the different problem variations that you will encounter in this assignment.**

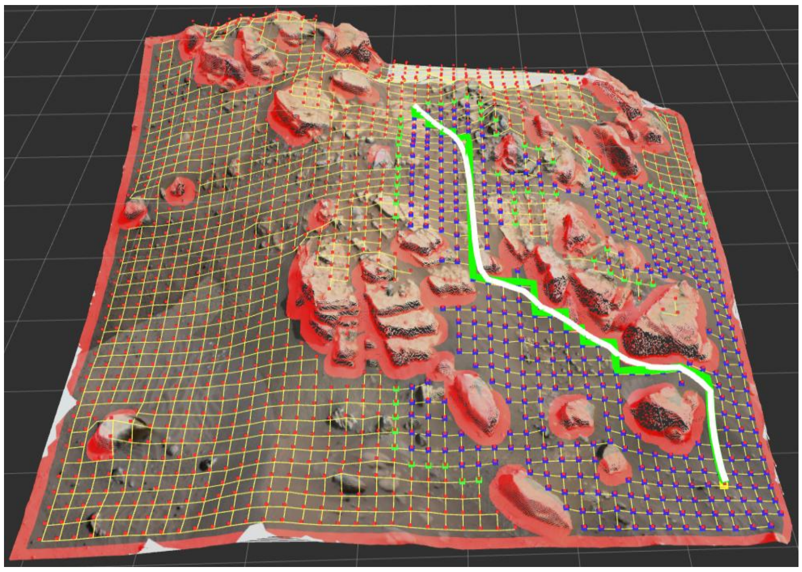


*Figure 2: Grip map with Euclidean distance edge costs.*

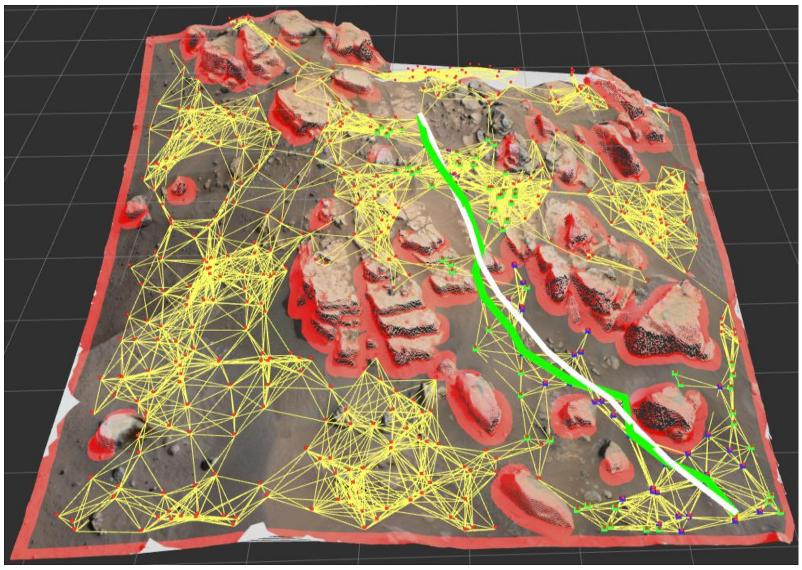


*Figure 3: Probabilistic Roadmap (PRM) with Euclidean distance edge costs.*

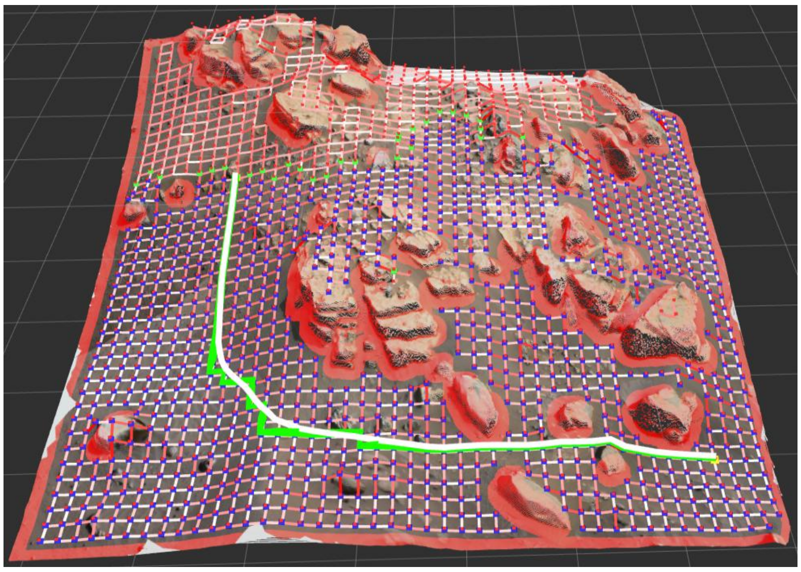


*Figure 4: Grid map with energy-based edge costs. Red edges have higher cost.*

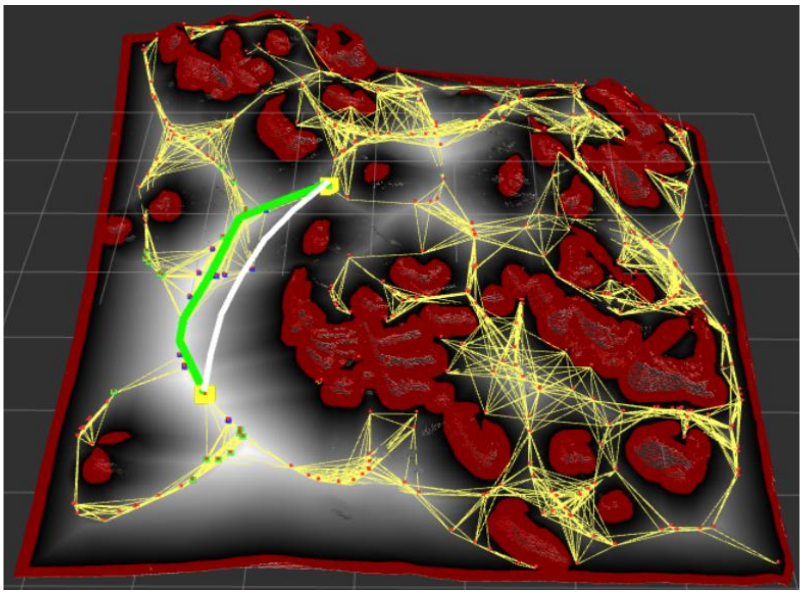


*Figure 5: PRM node generation biased by a distance transform map to find better connections through challenging regions.*

## Dijkstra's algorithm

Dijkstra's algorithm is an optimal algorithm for finding the shortest path between two nodes in a graph. Dijkstra's algorithm can be thought of as an expanding "cloud" that starts at the start node, then expands in all directions equally until the goal node is reached.

An implementation of Dijkstra's maintains two sets of nodes:

- *unvisited set*: nodes get added to this set as they are "discovered". Initially this is just the start node. This is sometimes called the "open set" or "frontier nodes".
- *visited set*: at each iteration of the algorithm, one node is moved from the *unvisited set* to the *visited set*, meaning this node has been "visited". This is sometimes called the "closed set".

Two important variables are associated with each node that gets updated by the algorithm:

- *cost_to_node*: the shortest-path distance from the start node to this node
- *parent_node*: the previous node on the shortest path from the start node to this node.

Initially, the *unvisited set* contains only the start node and the *visited set* is empty. The *cost* of each node is set to $+\infty$ (or a very large number). In each iteration, we remove the node that has the lowest *cost_to_node* from the *unvisited set* and add it to the *visited set*.

For each node that is connected to this new "visited" node, we do *one* of the following:

- If the connected node is not in the *unvisited* set or *visited set* then it is added to the *unvisited set*. The connected node's *cost_to_node* is set to the visited node's *cost_to_node* plus the edge cost from visited node to the connected node. The visited node as saved as the *parent node* of the connected node.
- If a connected node is in the *unvisited set* then we compute the *cost_to_node* using the same formula as above, but we only keep this new *cost_to_node* if it is less than the current *cost_to_node* stored at this node. If the *cost_to_node* is updated, then the *parent node* is also updated to the new parent.
- If a connected node is in the *visited set*, we do nothing as we have already found the shortest path to the connected node.

The search immediately finishes when the goal node is placed into the *visited set*.

The path from the start to goal node is then reconstructed by following the *parent nodes* (that were computed by the algorithm) from the goal node, one by one, back to the start node. This sequence then needs to be reversed so that the path goes from the start to the goal.

The search also finishes if the *unvisited set* is empty. If this occurs, this indicates that there is no possible path from the start node to the goal node.

## A*

A* is a modification of Dijkstra's algorithm that can significantly reduce the computation time by reducing the number of nodes visited. It does this by using a *heuristic*, which is a function that provides an estimated value for the quality of a node.

In mobile robotics each node in the graph represents a location in the map. In this case, a common heuristic that usually performs well is to simply take the Euclidean distance between the location of the current node and the location of the goal node. This heuristic usually works well when there is lots of free space, but can perform poorly if the map is very cluttered with obstacles.

A* is guaranteed to yield the optimal shortest path if the heuristic meets *both* of the following criteria:

- *Admissible*: the heuristic does not *overestimate* the cost of reaching the goal
- *Consistent*: the heuristic is always less than the estimate of any neighbour plus the cost of reaching the neighbours.

The Euclidean distance heuristic mentioned above is both admissible and consistent.

A* is almost identical to Dijkstra's but with one key difference: the heuristic is used when deciding which node to visit next. Instead of selecting the node in the *unvisited set* with the lowest *cost_to_node*, we select the node that has the lowest combined cost:

$$\textit{cost\_to\_node\_to\_goal\_heuristic} = \textit{cost\_to\_node} + \textit{heuristic} \times \text{weight}$$

The weight is a constant between 0 and 1 that can be used to modify the behaviour of the algorithm. A weight of 0 is equivalent to Dijkstra's algorithm, while a weight of 1 is the standard implementation of A*.

It is important to note that *cost_to_node and* *cost_to_node_to_goal_heuristic* need to be computed, since *cost_to_node* is still needed when expanding to new neighbours.

## Example

Let's show a quick example. We have a map that consists of a grid of cells, with obstacles marked as "x", starting location "S" and goal location "G". The robot is only allowed to move in four directions: right, left, up, and down, with each movement having a cost of 1.

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | | | |
| | | x | x | | | |
| | | x | G | | | |
| S | | x | | | | |

By following Dijkstra's algorithm, the cells get filled in with the *costs* shown in the following picture. Since Dijkstra's algorithm expands nodes in the order of lowest cost, these numbers also represent the order that the nodes are expanded, with ties broken arbitrarily. The counting stops at 10 because the goal is visited with a path of cost 10, and there is no reason to continue the search past this point.

| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 3 | x | x | 8 | 9 | 10 |
| 1 | 2 | x | 10 | 9 | 10 | |
| S | 1 | x | | 10 | | |

As you will see in your results, A* will reach this same result by expanding fewer nodes since it will focus more on expanding nodes that have a lower Euclidean distance to the goal node.

The *parent nodes* that get produced by Dijkstra's or A* can be thought of as "back pointers" along the shortest path from the goal back to the start, as shown in the following picture. The back pointers can be followed to find the optimal path, as shown in blue:

| | | | | | | |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ← | ← | ← | ← | ← |
| ↓ | ↓ | x | x | ↑ | ↑ | ↑ |
| ↓ | ↓ | x | → | ↑ | ↑ | |
| S | ← | x | | ↑ | | |

## Compiling and running the node

**Here are some useful ROS commands for getting started and running your code.**

### Install dependencies

Install some library dependencies by opening a terminal and running:

```
sudo apt install python3-scipy python3-matplotlib
```

### Load template code into workspace

You can re-use your workspace from assignment 1 (or start a new workspace). We will load the new template code into a new package within this workspace.

Download the provided `path_planner.zip` from Canvas and extract the `path_planner` directory into `~/ros_ws/src/` .

Now build your updated workspace from a terminal:

```
cd ~/ros_ws/

colcon build --symlink-install
```

**Tip:** You can also build an individual package at a time using (in case other packages are causing issues):

```
colcon build --symlink-install --packages-select path_planner
```

This should compile without errors.

## Setup the ROS environment

Open a terminal and setup your ROS environment:

```
source /opt/ros/humble/setup.bash

source ~/ros_ws/install/setup.bash
```

You have to repeat the above two commands every time you open a new terminal. A better approach is to add these commands to the "bashrc" file, which runs inside every terminal automatically when the terminal is first opened. To do this, open the bashrc file:

```
gedit ~/.bashrc
```

Scroll down to the bottom and – if they're not already there – add the above two "source" commands to the bottom of this file. Save and exit the file. Close and reopen the terminal.

## Disable ROS networking

By default, ROS2 allows you to subscribe to ROS topics across the local network, such as a wifi network. This is normally very useful, since it allows robots to talk to each other. However, if we are working together in the same classroom on the same network, then this can be very confusing.

To disable this, you need to set an environment variable in every terminal:

```
export ROS_LOCALHOST_ONLY=1
```

You have to repeat the above command every time you open a new terminal. In the same way that we just saw above, a better approach is to add this command to the "bashrc" file, which runs inside every terminal automatically when the terminal is first opened. To do this, open the bashrc file:

```
gedit ~/.bashrc
```

Scroll down to the bottom and – if it's not already there – add the above "export" command to the bottom of this file. Save and exit the file. Close and reopen the terminal.

## Running the code

Run the path planner node and the rviz visualisation with the following launch file.

**Important:** If you have recently changed your code or parameters, make sure to rebuild the package first using the "Building your node" instructions above.

Launch file:

```
ros2 launch path_planner path_planner.launch.py
```

Use *Ctrl+C* to exit a running program in the terminal.

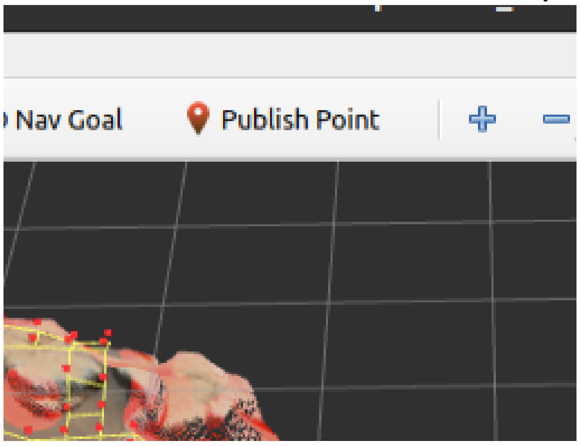
## Overview of the node

Here we provide an overview of the provided code for your reference.

The node starts with the `main` function at the bottom of `controller_node.py`. A `GraphSearchController` object is created, which is a ROS node that takes care of the ROS topics and parameters. The `GraphSearchController` also creates a `GraphSearch` object, which is defined in `graph_search.py` – this is where the path planning is performed. Several other `.py` files are provided in the `path_planner` directory with specific functionalities and are called from `graph_search.py`.

After you've completed the associated tasks, the code will do the following:

1. initialises the ROS node,
2. creates a map from a data file,
3. creates a graph from the map (either grid map or PRM),
4. performs a search over this graph,
5. smooths the path,
6. waits for a new goal from RVIZ: **click "*Publish Point*" in RVIZ then a point in the map**,



7. receives the new goal and repeats from step 4.

Each of the above steps is performed somewhere in the classes and functions of the various `.py` – **your task is to complete several of these python files in this assignment, at the indicated locations.**

## Software Development Tips

You will implement and test PRM-based A* path planner within a ROS workspace. To achieve this, you will begin with the provided template source code to get you started.

**Your implementation will be in Python.** ROS software can be developed with many different languages, but most applications are in Python or C++. We are going to focus on Python as it relatively user friendly, and a good place to start if you are less experienced with programming.

Here are some recommended tutorials:

- Python 3: https://docs.python.org/3/tutorial/index.html
- ROS2: https://docs.ros.org/en/humble/Tutorials.html

- Linux command line: http://www.ee.surrey.ac.uk/Teaching/Unix/
- A separate Python 3 tutorial sheet will be provided in 49274 on Canvas for students who have programming experience but are not familiar with Python.

**I strongly encourage regularly building, running and testing your code. This will make it much faster to find errors as they come up, rather than waiting until everything is complete before testing.**

**If using a virtual machine, I suggest regularly backing up your work *outside* of the virtual machine, in case your virtual machine image ever breaks.**

## Your task

Your task is to complete the provided template code to create and test a working path planning algorithm. The assignment is broken into 10 intermediate tasks, which involve different steps and variations of path planning.

**To score full marks, you need to fully complete all 10 tasks. However, some of the later tasks are quite challenging, so it is okay if you don't finish them all.**

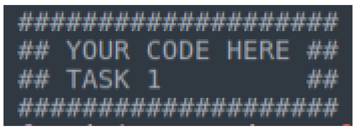
Your task is to complete 10 code snippets, which gradually increase in difficulty:

1. **Select the start and goal graph nodes**
2. **A naïve path planner**
3. **A* graph search**
4. **Extract the planned path from the search tree**
5. **Smooth this path**
6. **Find energy efficient paths**
7. **Create a PRM graph as an alternative to the grid map**
8. **Illustrate the connectivity of the PRM graph (Advanced Task)**
9. **Compute a distance transform map (Advanced Task)**
10. **Bias the PRM using the distance transform map (Advanced Task)**

Each task requires you to insert new code within the template code in one of the following files:

`path_planner/graph_search.py`
`path_planner/graph.py`
`path_planner/path_smoother.py`
`path_planner/map.py`

at a location marked with a comment `## YOUR CODE HERE ##`. It is not necessary to change any code outside of these spots; I suggest not changing any of the other code until you at least have good preliminary results.



```
####################
## YOUR CODE HERE ##
## TASK 1         ##
####################
```

Important parameters are defined in `config/parameters.yaml`, which you will need to change sometimes as you progress through the tasks, as noted below.

Your tasks are described in more detail as follows:

### Task 1: **Find the start and goal nodes**

**Method:** `Graph.get_closest_node`

When you first run the code, you will see that a grid map graph has already been constructed for you. We will use this grid map for the first few tasks.

Your first task is to complete the `get_closest_node` method. This method takes as input a point `xy`, which is in the form of an array `[x, y]`, such that `x=xy[0]` and `y=xy[1]`.

This method needs to return the **index** of the node in `self.nodes_` that is closest to the point `xy`. To measure distance, use the standard Euclidean distance in 2D between each pair of points.

**Hint:** See the definition of the `Node` class near the top of the file to understand how to access the necessary variables.

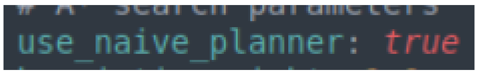
## Task 2: **Naïve path planner**

**Method:** `GraphSearch.naive_path_planner`

Before we implement the A* planner, we are going to implement a simpler planner to get started. This "naïve" planner will start at the current node, then pick the next neighbouring node that has not yet been visited and is closest to the goal. If the path gets stuck, then it is allowed to revisit a node up to 3 times.

**I describe this method as "naïve" because not only does it not guarantee to find the shortest path, but sometimes it might get stuck and not even reach the goal at all! The A* algorithm in the following task will overcome these issues.**

To run this task, first ensure that the `use_naive_planner` parameter in the `parameters.yaml` file is set to `true`.



```
use_naive_planner: true
```

You must complete this task by implementing the following pseudocode:

**Algorithm 1** Naive path planning algorithm.

```
function NAIVEPATHPLANNER($start\_idx$, $goal\_idx$)
    $path \leftarrow []$

    ▷ Add the current node to the path
    $current \leftarrow nodes[start\_idx]$
    $path.append(current)$

    ▷ Move to a neighbouring node and add it to the path until goal is reached
    while $current.idx != goal\_idx$ do

        ▷ Find the neighbour node that is closest to the goal and hasn't been visited yet.
        $best\_neighbour \leftarrow None$
        $best\_neighbour\_distance \leftarrow \infty$
        for $neighbour \in current.neighbours$ do
            $count\_visits \leftarrow path.count(neighbour)$
            if $count\_visits < 3$ then

                ▷ Euclidean distance plus a large penalty for revisiting a node.
                $distance\_to\_goal \leftarrow distance(neighbour, goal) + 100 \times count\_visits$
                if $distance\_to\_goal < best\_neighbour\_distance$ then
                    $best\_neighbour \leftarrow neighbour$
                    $best\_neighbour\_distance \leftarrow distance\_to\_goal$
        if $best\_neighbour == None$ then

            ▷ If no valid neighbours, exit.
            break
        else

            ▷ Otherwise, move to the best neighbour.
            $current \leftarrow best\_neighbour$
            $path.append(current)$
    return $path$
```

**Self-check:** At this point, you should be able to plan paths to new locations by specifying the next location in rviz. It should *usually* find the goal, but some of the paths near obstacles may be quite inefficient, as shown in the screenshot below – that's okay!

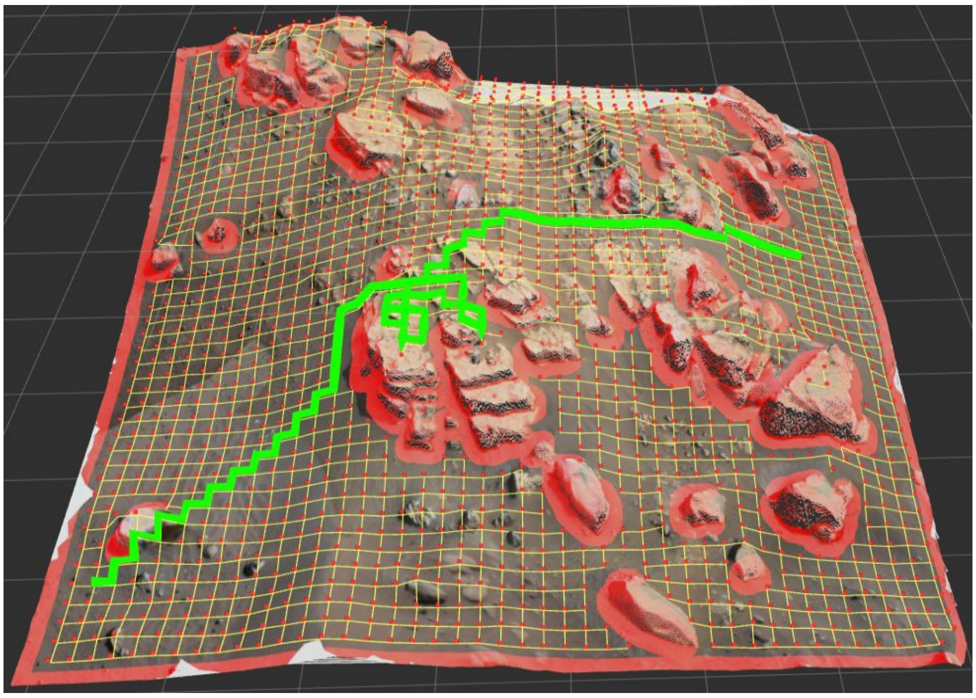


*Figure 6: Example path planned using the naive planner in Task 2.*

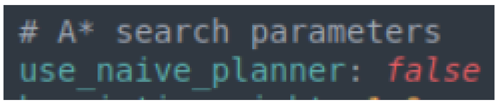
## Task 3: **Implement A***

**Method:** `GraphSearch.astar_path_planner`

You will now implement the A* search algorithm, which should be a much more reliable planner than the solution in Task 2.

Your A* implementation will expand a tree that covers the graph, starting from `self.graph_.nodes_[start_idx]` until it reaches the goal `self.graph_.nodes_[goal_idx]`.

To run this task, first ensure that the `use_naive_planner` parameter in the `parameters.yaml` file is set to `false`.



```
# A* search parameters
use_naive_planner: false
```

See the hints inside the template code, as well as the description at the start of this assignment document and in the seminar slides, for how to implement A*. You will need to fill in code at **six points** within this method, each marked with a *"YOUR CODE HERE"* comment, and each point should be approximately 1 to 3 lines of code.

**Hint:** Once you understand the logic of the algorithm, the hardest part of the implementation is likely to be keeping track of the different variables. Use logical variable names that help you keep track of their meaning. For example, use variable names such as `node_idx` to help you remember that this variable is an array `index` rather than a node.

**Hint:** The function `get_minimum_cost_node` is likely to be useful. Note that this function finds the node in the `unvisited_set` with the minimum cost, and returns its index in the `unvisited_set` array, **not the index in the `graph.nodes_` array.**

**Self-check:** If your code is working correctly, you should see the colours change in the graph as the A* search expands over the graph until it reaches the goal. It should look like the following screenshot.

Select a few more goal locations using the "Publish Point" button to test your code. It does not yet construct a path – we will do that in the next Task.

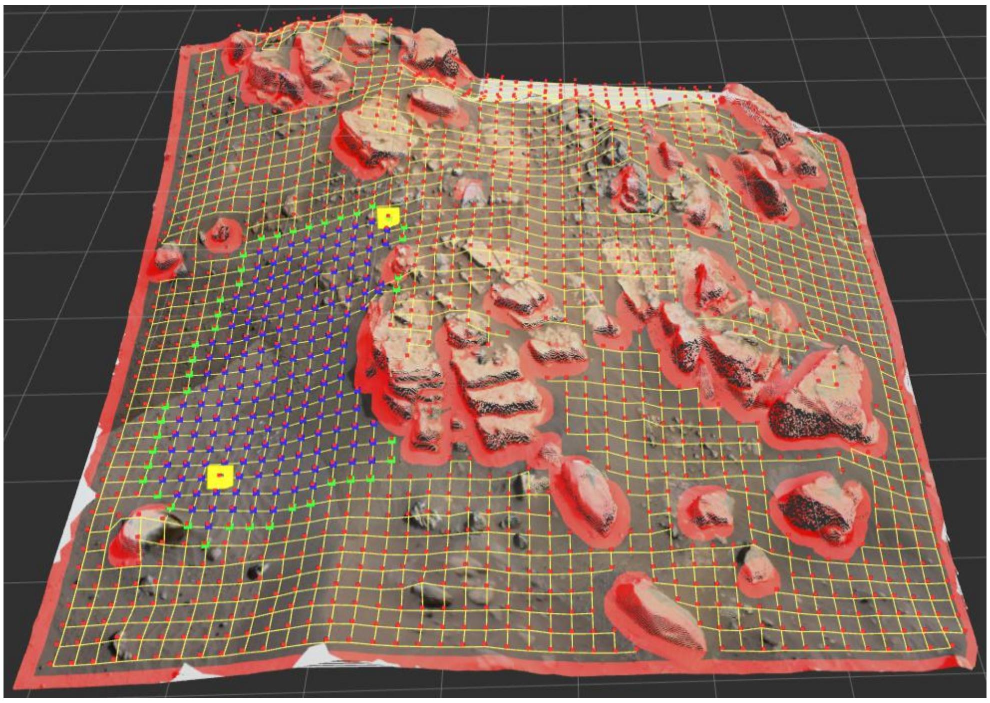


*Figure 7: Example output from Task 3. The blue coloured nodes are the visited set, and the green nodes are the unvisited set, at the point that the search reaches the goal node.*

## Task 4: **Generate the path**

**Method:** `GraphSearch.generate_path`

Your A* code from Task 3 creates a search tree over the graph, but does not directly provide a path from the start node to the goal node. The `generate_path` method will extract this path from the search tree.

Use the `node.parent_node` variables that were filled in during the search. Follow these parents from the goal node until you reach the start node. This will produce the path in reverse, so you will then need to **reverse the path**.

If you are still unsure, have a closer study of how this example works that was introduced earlier in this assignment document:

| | | | | | | |
|---|---|---|---|---|---|---|
| ↓ | ↓ | ← | ← | ← | ← | ← |
| ↓ | ↓ | x | x | ↑ | ↑ | ↑ |
| ↓ | ↓ | x | → | ↑ | ↑ | |
| S | ← | x | | ↑ | | |

**Self-check:** You should now have a functional A* algorithm working! It should perform like in the screenshot below. Compare its performance to the naïve planner from Task 2 – it should be much better!

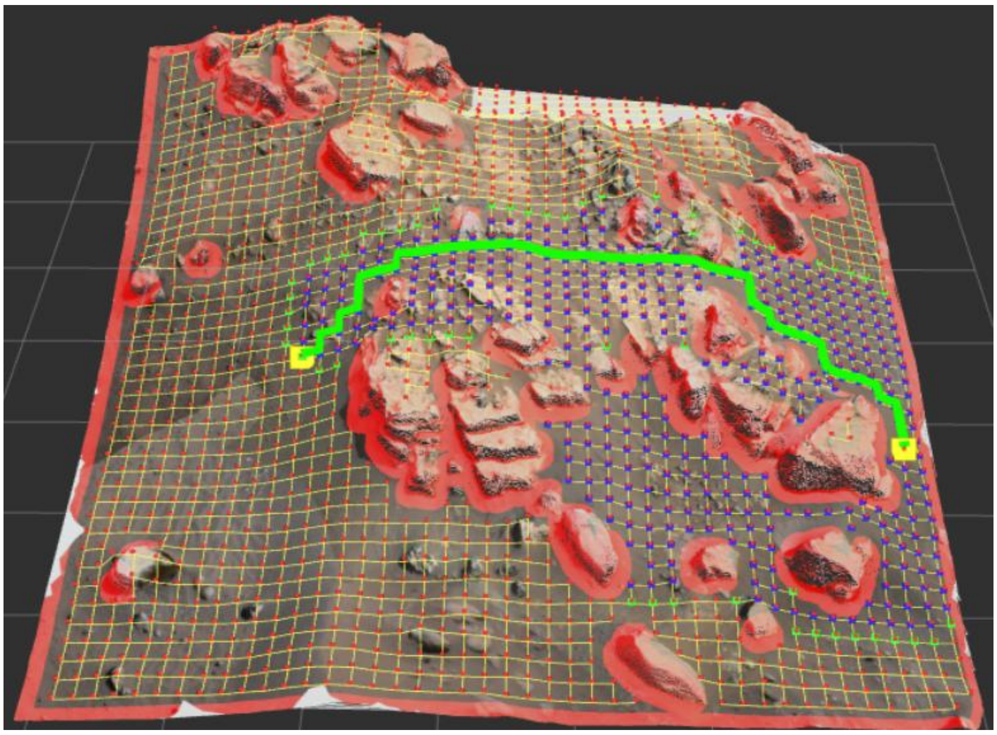


*Figure 8: Example output for Task 4.*

## Task 5: **Smooth the path**

**Method:** `PathSmoother.smooth_path`

You will probably notice that sometimes your planner produces unnecessarily sharp turns. These turns can be difficult for a robot to execute. Now your task is to smooth this path using a simple gradient descent path smoothing method.

There are many ways to achieve path smoothing, but here we introduce a simple algorithm. Suppose the original (un-smoothed) path is defined as the sequence of points $(p_1, p_2, \dots, p_N)$ and the new (smoothed) path is defined as $(s_1, s_2, \dots, s_N)$. We want to optimise the smoothed path by minimising the following objective function:

$$\alpha \sum_{i=1}^{N} \|s_i - p_i\|_2^2 + \beta \sum_{i=1}^{N-1} \|s_i - s_{i+1}\|_2^2$$

where $\|\cdot\|_2^2$ denotes the square of the Vector L2 Norm (i.e. Euclidean distance), and α and β are constant parameters that you can tune. The first term is the sum of the squared distances between the smooth and original points. The second term is the distance between consecutive smooth points.

It is important to realise that if we only minimise the first term then we will get the original path. If we only minimise the second term then we will get a straight line linking the start and goal. Therefore, we must simultaneously minimise both by minimising a weighted sum of the two terms, weighted by α and β.

An example smoothing is shown in the following picture.

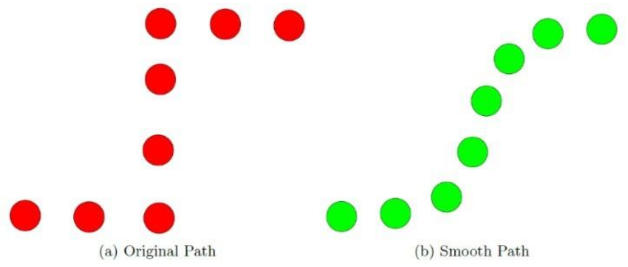


*Figure 9: Illustration of the smoothing concept.*

You will implement this smoothing through the following steps:

1. Initialise the list of smooth waypoints $s_i$ as the original waypoints $p_i$. Note that $s_i$ and $p_i$ are points with elements *s_i.x* and *s_i.y*.
2. At each iteration, loop through all of the waypoints, except the **first** and the **last**, and update the smooth path using the following equation, where the parameters are defined in variables `alpha` and `beta` in your code:

$$s_i^{new} = s_i - (\alpha + 2\beta)s_i + \alpha p_i + \beta s_{i-1} + \beta s_{i+1}$$

3. Each time you perform the above update, make sure that the **two path segments** that have been adjusted **do not go through an obstacle**. If the updated path does go through the obstacle, then discard this update and revert back to the previous path. The collision check can be performed with the `is_occluded` function using something similar to the following, which returns true if there is an obstacle in the map image between two points:

```
blocked = is_occluded(self.graph_.map_.obstacle_map_, [x1, y1], [x2, y2])
```

4. Finally, keep iterating until within this loop your new smooth path changes very little from your previous smooth path. With a small tolerance value of $\epsilon = 0.001$, keep updating the path until the following condition is met:

$$\sum_i (s_i^{new}(x) - s_i(x))^2 + (s_i^{new}(y) - s_i(y))^2 < \epsilon$$

**Self-check:** In RVIZ, you should now see a white path and a green path, like in the screenshot below. The green path is the path through the graph and the white path is the new smoothed path. You should see that they follow a similar path, but the white path should have smooth corners that are likely to be easier for a robot to execute. Both the white and green paths should not go through obstacles.

Try changing the `alpha` and `beta` parameters in the `parameters.yaml` file to get a different balance between smoothness and matching the original path.

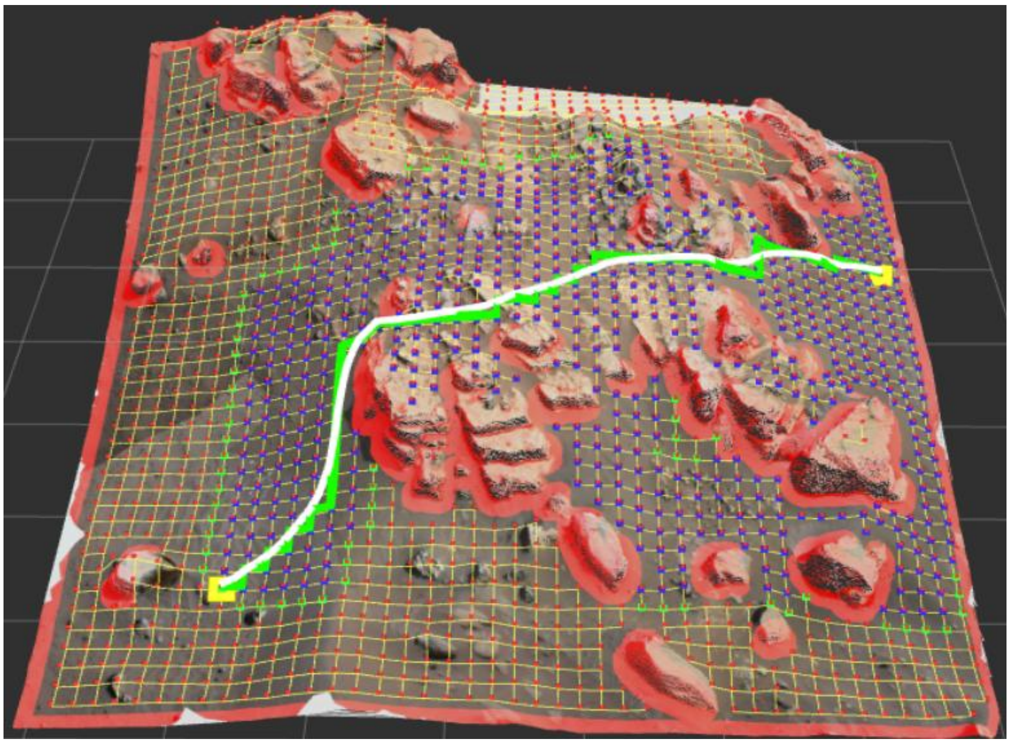


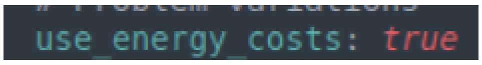
*Figure 10: Example output from Task 5. The white path is a smoothed version of the original path (green).*

## Task 6: **Plan energy efficient paths**

**Method:** `Graph.create_grid`

So far, we have been assuming that the cost to travel from one node to another is simply the 2D Euclidean distance between the two points. In this task we are going to incorporate of estimates of energy consumption into the edge costs. Our robot should now favour flat and down-hill routes over the 3D terrain, and avoid steep up-hill sections where possible.

To run this task, set the `use_energy_costs` parameter to `true` in the `parameters.yaml` file:



```
use_energy_costs: true
```

The energy cost $E$ you are to implement is described by the following equation:

$$E = |\, (\mu m g \ \cos\theta + m g \ \sin\theta) \mathrm{d}x \,|$$

where the terms have the following meanings:

5. $\mu = 0.1$ is the rolling resistance coefficient. For simplicity, we set this as a constant, but in general this is a property of the terrain type, which might change across the environment.
6. $m = 1025\text{kg}$ is the mass of the Perseverance rover.
7. $g = 3.71\ m/s^2$ is the acceleration due to gravity on Mars.
8. $\mathrm{d}x$ is the 3D Euclidean distance from the start to goal node.
9. $\theta$ is the gradient of the slope in radians relative to the horizontal. A value of 0 indicates flat terrain. Positive values indicate up-hill, while negative values indicate down-hill. For simplicity, assume that the slope is constant between the two points, and therefore can be computed as:

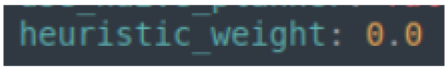
$$\theta = \tan^{-1} \frac{\text{change in elevation}}{\text{horizontal distance}}$$

- Note that the model for $E$ takes the **absolute value**. This is because without this, steep downhill gradients would generate energy rather than consume energy, according to this model. The idea of using an absolute value here is to model braking required to maintain a constant velocity. Also, A* unfortunately breaks for negative costs, so it is convenient to keep all costs positive.

**Important note:** The distance and slope calculations mentioned above should be measured in **3D world coordinates** (i.e., the coordinates in meters relative to a fixed origin). Note that the $(x, y)$ coordinates stored within a Node are in **2D pixel coordinates** (i.e., the pixel location in the underlying image representing the terrain). The following function will help you do this **transformation**; you will need to work out how to use it correctly:

```
[world_x, world_y, world_z] = Map.pixel_to_world(pixel_x, pixel_y)
```

**Hint:** Since we haven't designed a good A* heuristic that aligns with this new cost function, we should disable the A* heuristic. You can do this by setting the heuristic weight to 0:



```
heuristic_weight: 0.0
```

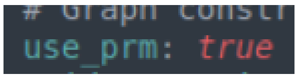
**Self-check:** Test your path planner again, and you should now observe that the path won't always be the shortest path, but instead avoid steep uphill sections. The colouring of the edges in the graph have now changed to reflect edge costs: white is lower costs and red is higher costs. Note that for each line we actually have two edges on top of each other – one for each direction. The colours only indicate the higher of these two costs (i.e., the **up-hill costs**).

## Task 7: **Create a PRM graph**

**Method:** `Graph.create_PRM`

So far, you've seen the graph search performed over a grid. However, the graph search works on *any* type of graph. This task is to replace the grid with a Probabilistic Roadmap (PRM).

To run your code in this task, change the `use_prm` parameter in `parameters.yaml` to `true`:



```
use_prm: true
```

You can switch back to the default grid at any time but setting it back to `false`.

When `use_prm` is set to `true`, then the `create_PRM` method is called instead of `create_grid`.

Fortunately, creating a PRM is very similar to creating a grid. In fact, our edge connection approach can remain the same, just with a higher connection distance threshold, so this code is already provided for you (except I have removed the energy cost code from Task 6).

Your task is to fill in the part of `create_PRM` where the nodes get created. The nodes should be created as `num_nodes` number of nodes **randomly** placed in the free space of the environment. Take a close look at how `create_grid` works, since `create_PRM` node generation should be similar.

**Self-check:** You should now see a PRM graph instead of a grid. How does the path planning compare to when using a grid? Try it with the energy costs (Task 6) and the default costs.

## Task 8: **Graph connectivity (Advanced Task)**

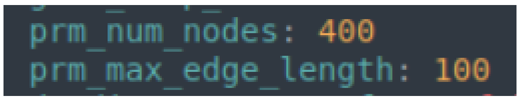
**Methods:** `Graph.find_connected_groups` **and** `GraphSearch.find_connected_nodes`

**The followed tasks are extension tasks that will require you to *design* and *implement* a solution algorithm that addresses each problem below. Make sure all of the previous tasks are working well before attempting the following tasks.**

You may have noticed with your PRM that sometimes the graph is **disconnected.** In some regions of an environment, this may be okay if it has regions that are unreachable, such as behind a closed door or through a very narrow corridor. However, sometimes disconnections are incorrect, which can occur if the PRM fails to adequately sample nodes and connect edges in a region of the environment.

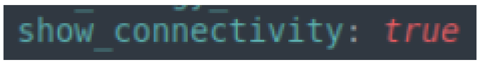
Your task here is to **analyse the connectivity of the PRM graph** by writing code that **finds the connected components of the graph**.

You can experiment with the following `parameters.yaml` file parameters, which is going to vary the connectivity of the PRM:



```
prm_num_nodes: 400
prm_max_edge_length: 100
```

To begin with, change the `show_connectivity` parameter to `true.`



```
show_connectivity: true
```

Changing this parameter will disable the replanning and change some of the visualisation so it will be easier to see the results for this task. **Remember to switch back this parameter to** `false` when switching to other tasks.

**The below RVIZ visualisation shows what you are aiming for in this task. See how that the nodes are now coloured. All nodes that are connected to each other are given the same colour, while nodes with different colours are disconnected from each other.**

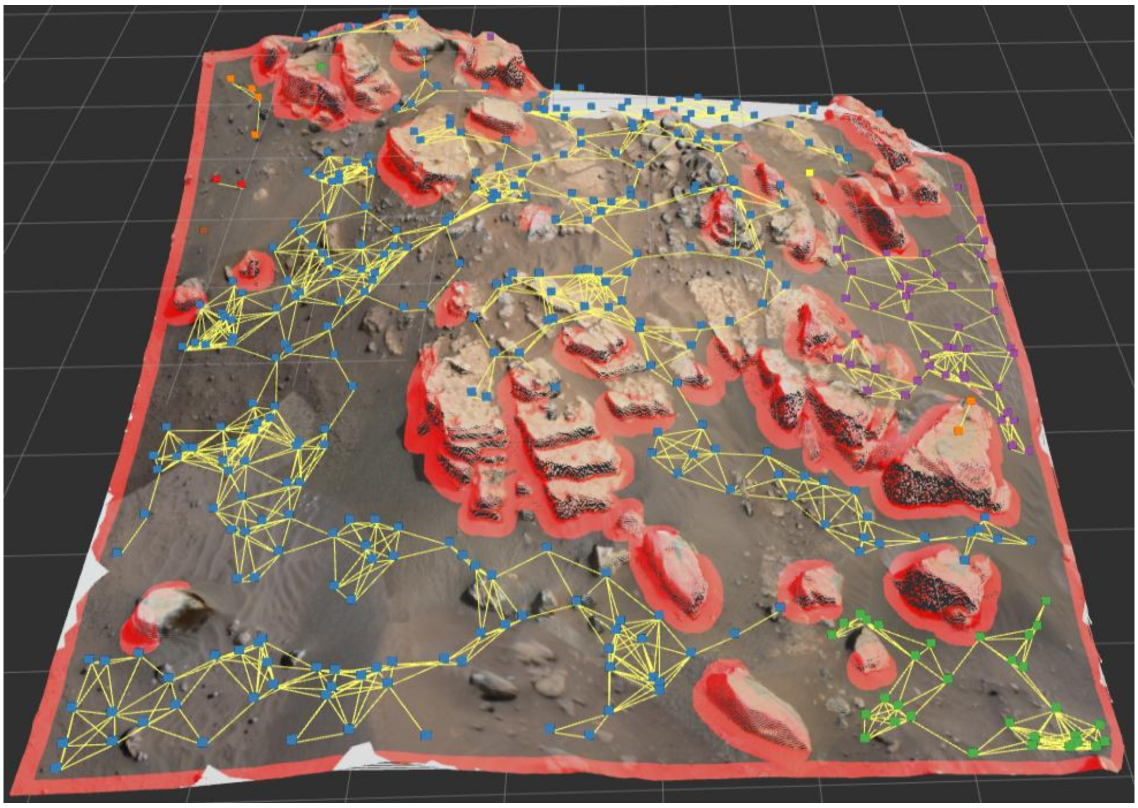


*Figure 11: What you are aiming for in Task 8. The different colours of the nodes indicate connected groups of nodes.*

In this example, you can see there are many blue nodes that are connected through the centre of the environment. But there are also smaller disconnected groups of nodes, such as the green nodes at the bottom-right, purple nodes in the middle-right, and many smaller groups in the top-left. There are even a few nodes that are completely isolated and disconnected from the rest of the graph.

**To compute connectivity**, it turns out that we can also use Dijkstra's graph search. The *visited set* at the end of the search will indicate all nodes that can be reached from the start node. Note that this works irrespective of the goal node, so you should **not** specify a goal node in this case. This means that your Dijkstra's algorithm should not halt at a goal node, and we **cannot use the A* heuristic** since this relies on the distance to a goal node.

**You need to think about the best way to approach this task.**

Write your solution code in the method `find_connected_groups`. This method should return a list that has length equal to the number of nodes in the graph. This list contains integer numbers, with each number associated to the node with the same index in the graph. If a pair of nodes **are connected, meaning that a path exists from the first node to the second node**, then these two nodes should **have the same number** associated with it. It doesn't matter what the number is, so long as this number is the same for all nodes in a connected group, and a different number to all nodes that are not connected. For example, the group of green nodes up the top in the example were all arbitrarily given the number "2".

Your `find_connected_groups` method should rely on Dijkstra's graph search, but as hinted above, this implementation needs to be a bit different to the `GraphSearch.search` method that you implemented in Task 3. Instead, you should call the `GraphSearch.find_connected_nodes` method. **You will need to fill in this method.** It will likely be very similar to the `search` method, but you will need to think about what changes need to be made. **There's a couple of hints provided at the top of the `find_connected_nodes` method in the template code.**

**Once your code is working, the RVIZ markers have already been setup to colour the nodes according to the `Graph.groups_` variable, which you should set in your `find_connected_groups` method. Since the selected colourmap has only 8 colours for simplicity, some colours may be repeated for multiple groups – this is okay.**

**In your report, briefly describe your approach to this task. Also, use this new tool you developed to analyse the connectivity of your PRM graphs under different conditions. What parameters of the PRM are most important for ensuring good connectivity? You can also compare the connectivity of the PRM to the grid map.**

## Task 9: **Distance transform map (Advanced Task)**

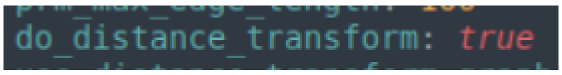
**Method:** `Map.generate_distance_transform_map`

A **distance transform map** is a special type of map that describes the distance to nearest obstacles. Specifically, the value defined at a location in the map denotes the distance to the nearest obstacle. A grid cell with a value of 0 is inside an obstacle, small values are close to obstacles, and large values are far away from obstacles.

**Distance transform maps have a variety of use cases in robotics, including:**

- **Enabling safer navigation by adjusting the behaviour of the robot (such as speed) based on queries of the distance transform map**
- **Faster retracing (this was used in the template code for Assignment 1!)**
- **Identifying challenging parts of an environment for navigation – we explore this capability further in Task 10!**

To enable running this task, set the following configuration parameter:



```
do_distance_transform: true
```

There are various ways of computing a distance transform map; we are going to implement the method outlined as follows – your task is to translate this algorithm concept into functioning python code. Note that we'll be measuring distance as number of pixels using Manhattan distance – you don't need to convert these distances into units of meters.

1. Define an empty matrix, the same size as the occupancy map that describes the obstacles
2. **First pass:** Iterate through all cells from the top-left to the bottom-right. To avoid an out-of-bounds error, skip the first and last rows and columns. For each cell:
    a. If the cell is inside an obstacle, then give it a `distance=0`
    b. Otherwise, look at the cell immediately to the **left** of this one, and the cell immediately **above** this one. Add 1 to each of these distances. Pick the smallest out of these two distances. Store this new distance in this cell.
3. **Second pass:** Iterate through all cells in the reverse order (i.e. from bottom-right to top-left). Same as above, to avoid an out-of-bounds error, skip the first and last rows and columns. For each cell
    a. If the cell is inside an obstacle, then give it a `distance=0`
    b. Otherwise, look at the cell immediately to the **right** of this one, and the cell immediately **below** this one. Add 1 to each of these distances. Pick the smallest out of these two distances. **If this distance is less than the current distance in this cell** (from the first pass), then store this new distance in this cell.

That's it!

When working correctly, the distance transform can be visualised in rviz, with the obstacles (distance=0) shown as red, low distances as black, and higher distances as white (scaled by the highest distance in the map).

The following screenshots show what you are aiming for. Note that I disabled the graph and the paths in rviz for these screenshots, since they are not directly relevant to this task (we'll return to these in Task 10). The first screenshot is after implementing the **first pass only**, which is incomplete, but a good starting point. The second screenshot shows the result of the full algorithm.

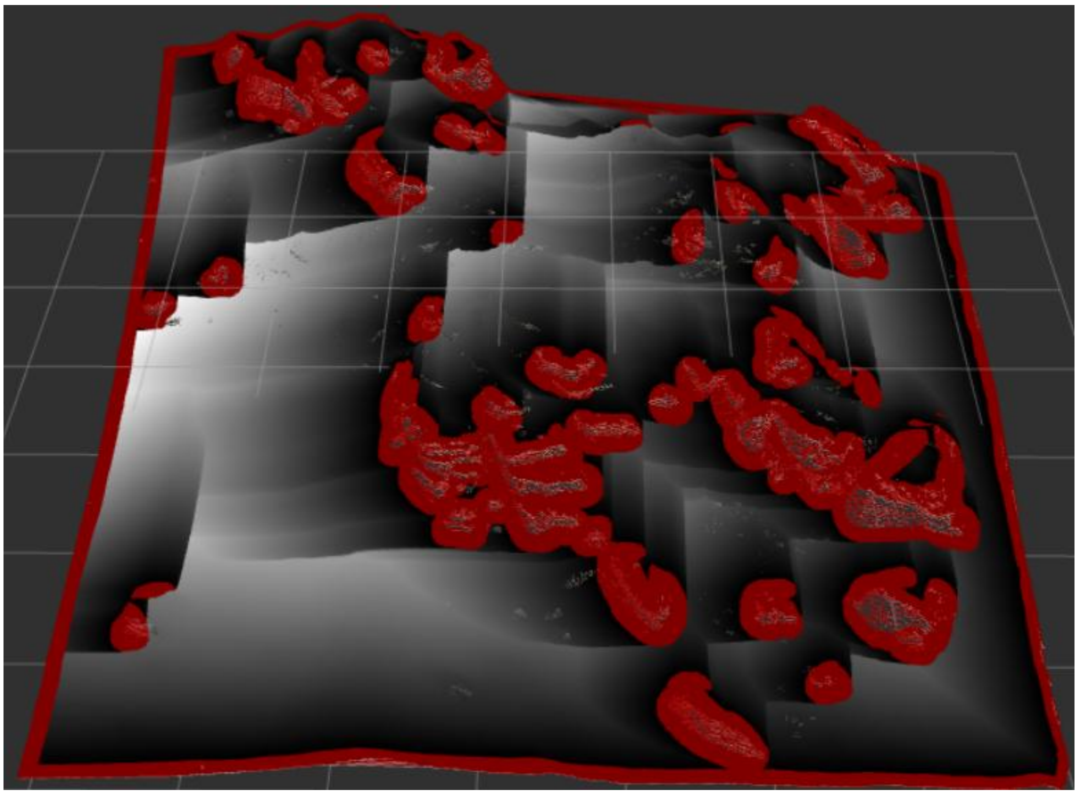


*Figure 12: **Partial solution** for Task 9, just showing the **first pass only**.*

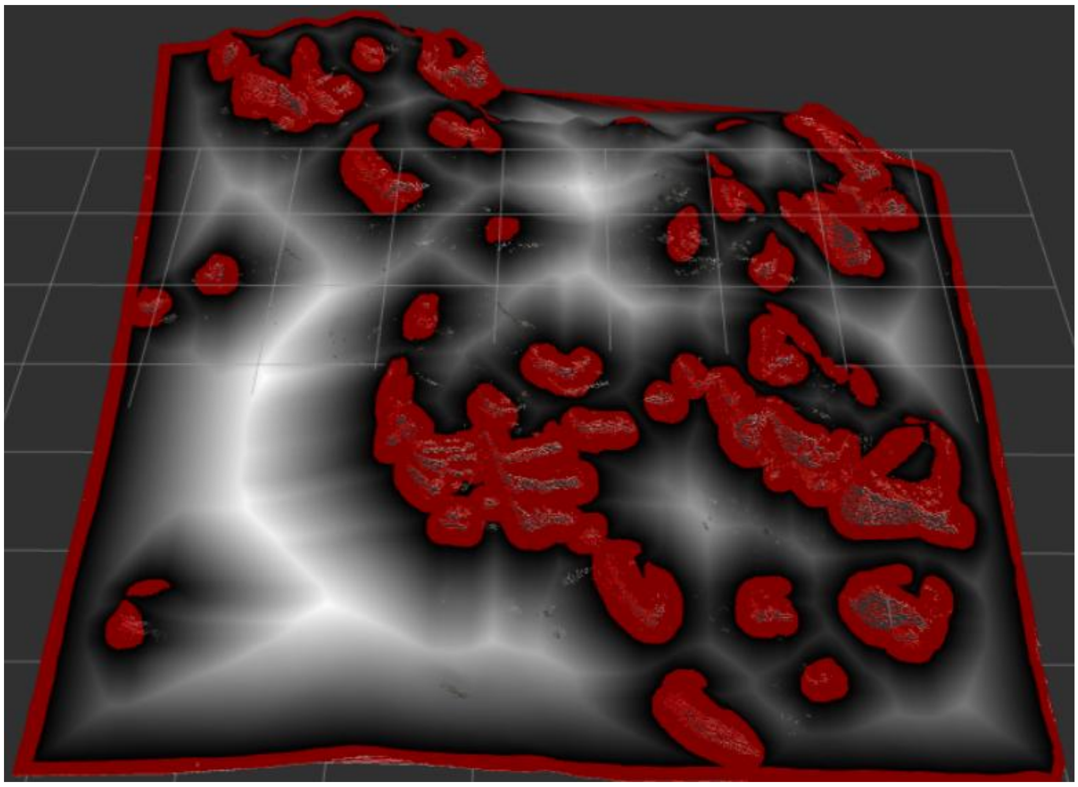


*Figure 13: The complete distance transform map. The red areas are obstacles, while the shade of white is proportional to the distance to the nearest obstacle.*

## Task 10: **Distance transform-guided PRM (Advanced Task)**

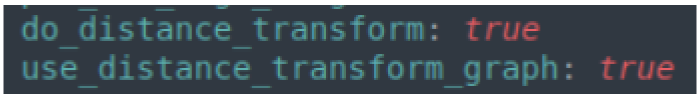
**Method:** `Graph.create_distance_transform_graph`

**We’re now going to use the distance transform from Task 9 to assist with generating a navigation roadmap. Make sure your solution to Task 9 is working correctly before attempting this task.**

What you should see when studying the visualisation in Task 9 is that the distance transform seems to discover the geometry of the environment. If you follow the brighter regions, this resembles a navigation roadmap that naturally finds paths through the various passages of the environment.

In this task, we are going to update our PRM generation from Task 7 to use this geometry revealed by the distance transform.

To enable running this task, set the following parameters:



```
do_distance_transform: true
use_distance_transform_graph: true
```

Begin by copying your solution from Task 7 in the function `create_PRM` to the new function `create_distance_transform_graph`.

Modify the PRM node generation step to only sample in regions that are deemed to be important according to the distance transform map. You can use *rejection sampling* which involves randomly sampling, then discard samples that don’t meet your criteria.

A relatively simple criteria we can implement here is to look for local maxima in the distance transform map. Look in a small neighbourhood around the sampled location (say an 11x11 grid around the sample location), and compare the distance values in the distance transform map. If the distance in the sampled location is greater than or equal to some fraction of the distance of the neighbouring cells (say 70%), then keep this sample. Otherwise, discard the sample, and repeat until `num_nodes` nodes have been added.

Perform the edge connection as before, but you may need to update the `prm_max_edge_length` parameter to refine the results.

While not expected for this assignment, various other criteria could be used on the distance transform map. A common one is to use properties of the Hessian matrix, which can be used to perform a **medial axis transform**, which reveals a “skeleton” of the environment. Again, this is not expected from this assignment, but I encourage exploring this in your own time if you are interested.

Here’s a screenshot of an example run, which shows roughly what you are aiming for. You should observe that this new PRM variation does a better job of connecting edges through narrow passages, since the sampling has now been biased towards these more challenging parts of the environment. This would allow our Mars rover to more reliably find paths through the challenging narrow passages!

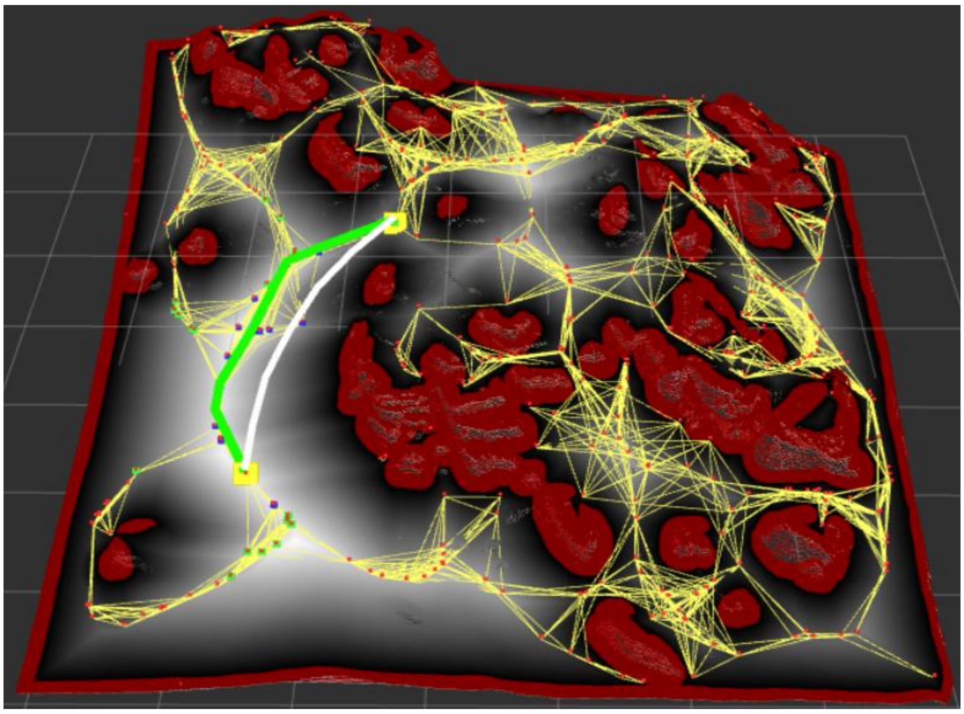


*Figure 14: Example roadmap from Task 10, where the PRM node sampling has been biased towards local maxima of the distance transform map.*

## Your report

You will submit a brief report about what you have achieved. I expect reports to be between 2 to 10 pages and can include pictures. I encourage being concise in what you are trying to communicate – long reports will not necessarily score higher.

**Your report should follow this structure:**

- Your name and student number up the top.
- A very brief introduction that includes a **summary of what you achieved**. Did it work? If not, what went wrong?
- A brief **reflection on your development methodology**. Did everything work the first time? If not, how did you go about identifying and addressing any challenges?
- A link to your **video** (uploaded to YouTube or Google Drive) – **please make sure the marker has permission to view the video**. Your video should be a screen recording (e.g. with Kazam in Ubuntu) of your path planner working. You should include short clips that demonstrate each of your tasks that you completed working. Aim for a 2 to 5 minute video. There is no need to add audio or any description within the video – this goes in your report. Here is a sample video:
    - https://youtu.be/yUMLExkABeg . Note this example is from a previous year, so there are small differences, and it doesn't include the advanced tasks.
- **For each task**, provide one short paragraph that describes **what you did for each of the 10 tasks** and includes a **screenshot of your code snippets**. No need to explain in detail. Optional: Include additional screenshots/pictures if it helps communicate your message.
- A brief conclusion that **compares the different approaches** that you implemented and summarises **how you would improve** the methods if you had more time.

**Your overall mark for the assignment will be based on the following criteria:**

- Implementation of all 10 tasks. Each task is weighted approximately equally – see rubric on Canvas.
- How well your path planner works for the various tasks.
- **Self-reflection:** Your report clearly and **accurately** describes what works, what doesn't work, how you overcame any challenges, and how things may be improved in the future.
- Good coding style, such as keeping things simple and having adequate commenting. We won't be too strict on this criteria, but unnecessarily complicated code that's not well motivated in your report may be penalised.
- You may be asked to demonstrate your understanding verbally to the Subject Coordinator.