Introduction to Mobile Robot Path Planning

Unit 4: Rapidly-Exploring Random Tree (RRT)

- Summary -

Estimated time to completion: **2 hours**

This unit covers the fundamentals of the Rapidly-Exploring Random Tree (RRT) algorithm applied to the robotic path planning problem. Specifically, you will:

- Understand the rationale behind sampling-based path planning
- Learn about how RRT works
- Reflect on the properties of the RRT algorithm
- Implement RRT in Python and run it integrated with the ROS Navigation Stack
- Understand the advantages of RRT compared to other pathfinding algorithms

- End of Summary -

3.1 Introduction

A* has been one of the most popular and widely used pathfinding techniques since its introduction. However, one caveat is that it does not scale well in terms of both map size and dimensionality: the larger a map gets, the more resources (computational time, planning time) are needed. Therefore, its use has been mostly limited to 2-D mobile robots and low degree-of-freedom (DOF) robotic arms.

Probabilistic path planning methods form a different family of path planning algorithms, and are very popular for solving complex, high-resolution and high-dimensional path planning problems.

Instead of meticulously examining directly connected grid cells, sampling based algorithms generate candidate waypoints at completly **random** positions in the map and then examine if these positions can be connected without hitting obstacles.

Rapidly-Exploring Random Trees, or RRT for short, is one such algorithm. It is regarded as one of the most efficient tools for robotic path planning in higher dimensions. This unit covers the basics of RRTs, organized into the following sections:

In section 3.2, you will run a demo, so you can have a practical look at what you are going to learn. Section 3.3 presents a step-by-step example of the involved actions, while Section 3.4 shows an outline of the algorithm's logic. In section 3.5, you will follow detailed instructions to code your own RRT implementation, and section 3.6 will guide you on testing your program on a simulated robot. Upon success, your robot will be able to navigate from your current position to a goal position in an environment full of obstacles. The limitations of RRT and variants of RRT are presentend in section 3.7, and section 3.8 closes this unit with a summary and its key take-aways.

3.2 A foretaste of what's to come

As we always do at The Construct, we will be using ROS and Gazebo for developing and testing.

- Exercise 3.2.1 -

Before proceeding, please ensure that the Gazebo simulation at the top left side has completely loaded, as illustrated in the image below:

Note: This unit assumes you have followed the instructions on unit 2 to download the course repository.

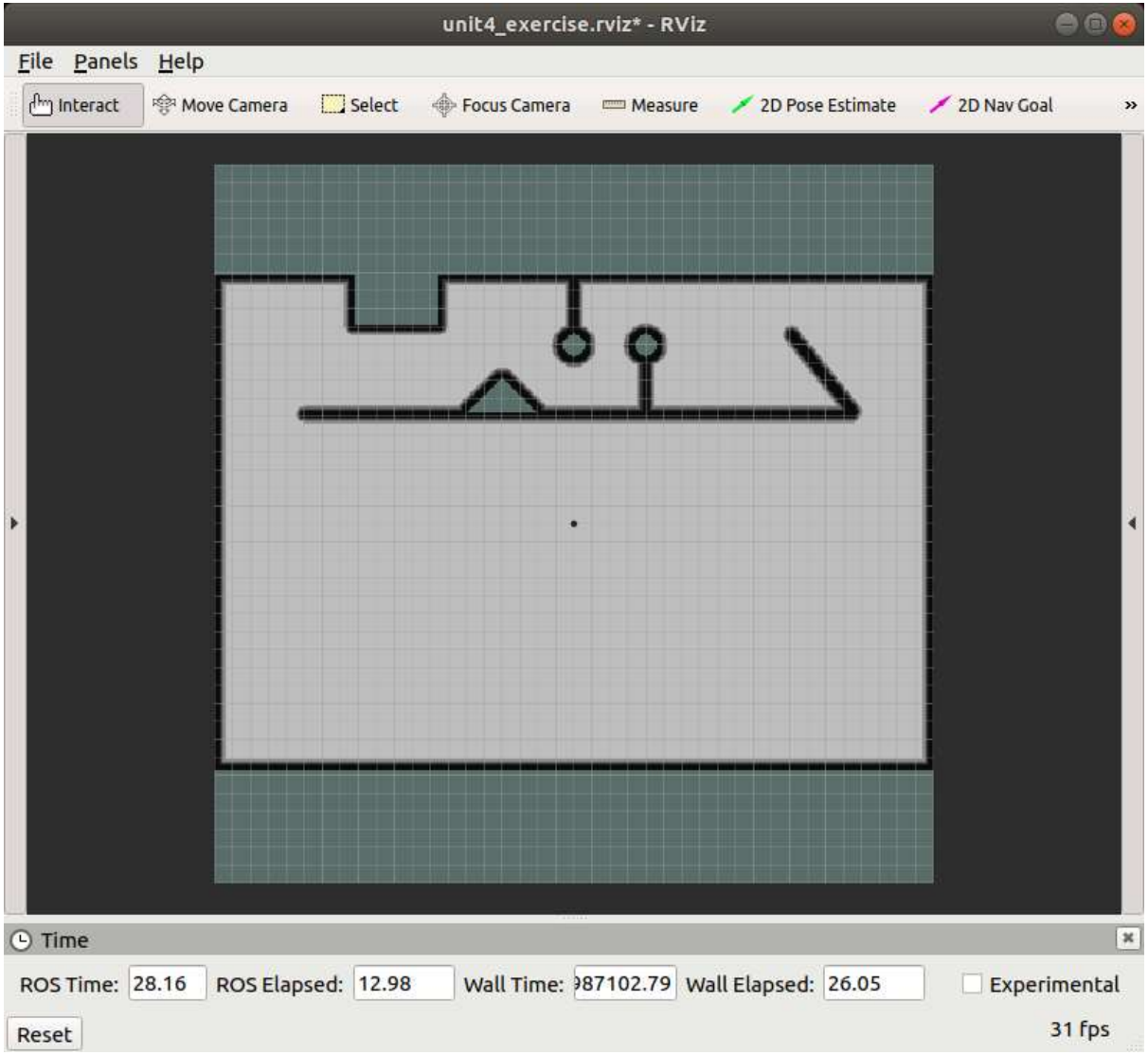
Without further delay, execute the following launch command in order to start an interactive demo program and see what to expect from this unit.

▶ Execute in WebShell #1

In [ ]:
```
roslaunch unit4 unit4_demo.launch
```

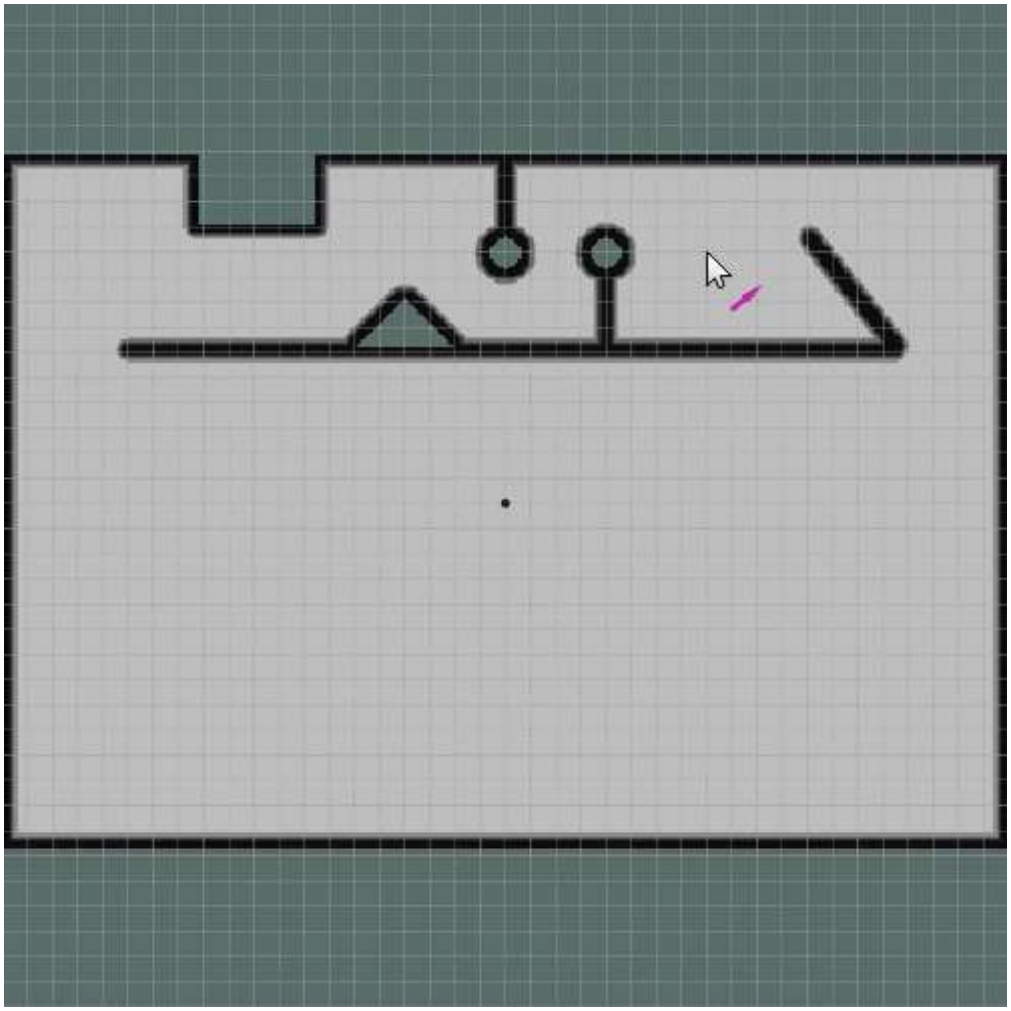
Please wait for a few seconds until **Rviz** opens.

After a loading time you will see something like this:



Now command your robot to any map location using the **2D Nav Goal** button.

Expected Result:



You should be able to see a tree-like structure growing from the robot's position and slowly covering the map's free space. Eventually, the tree will grow until randomly reaching the goal.

When you're done, shut down the demo program by pressing **Ctrl+C** on WebShell #1.

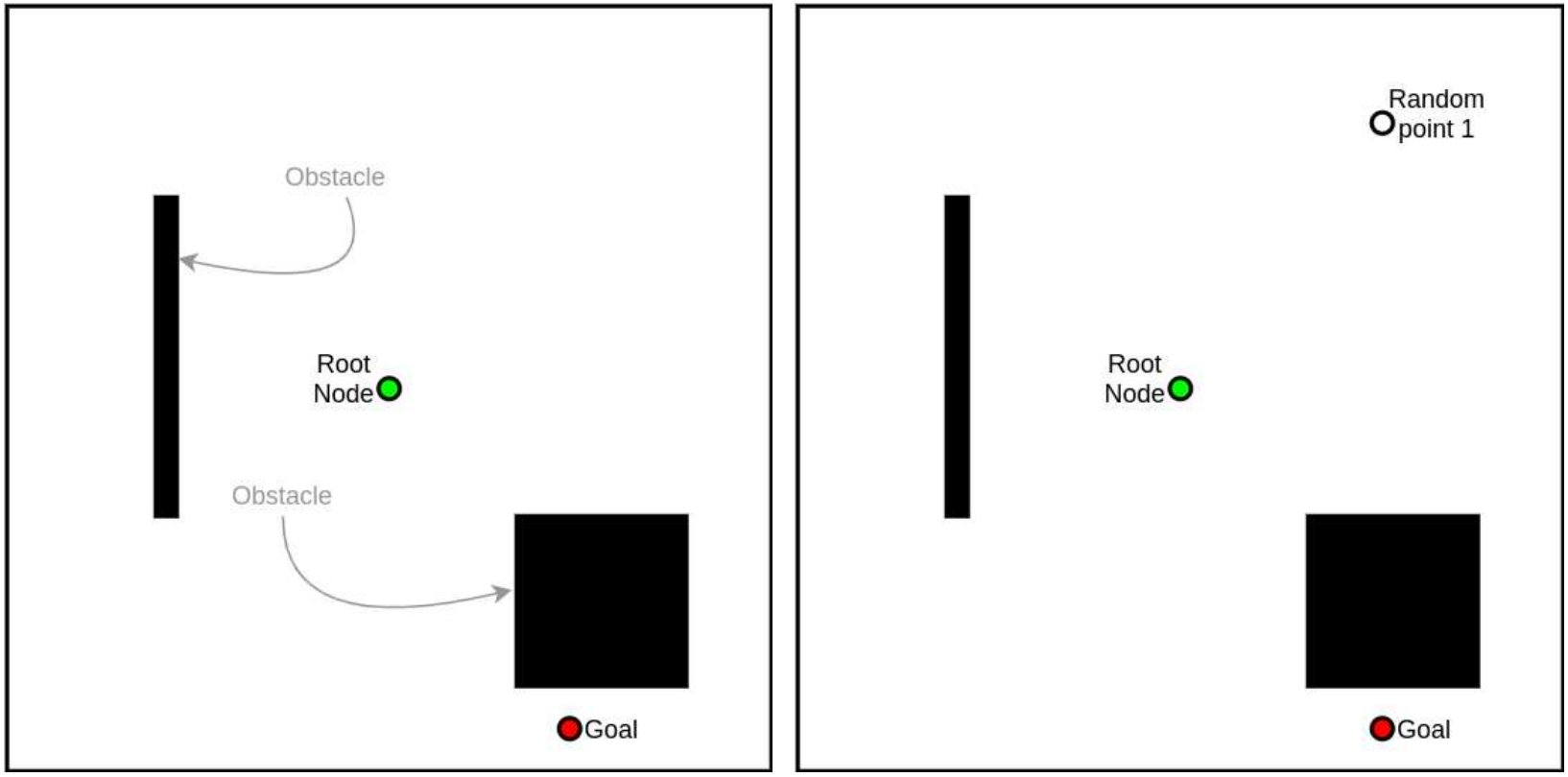
In the following section, you will learn the basic principles behind RRT.

- Exercise 3.2.1 -

3.3 Step-by-step example

This example shows how RRT works in a 2-dimensional grid map. The map shown below displays a white area, the map's free space, which is limited by a surrounding frame, the map boundaries. There are also two obstacles in it represented by two black rectangles. The robot can take on any position in the map's free space and cannot traverse obstacles. RRT initiates from the robot's current pose (let's call that position **root node**), and the goal pose is also explicitly known.

RRT starts by picking a random point anywhere in the map; we call it *Random point 1*.

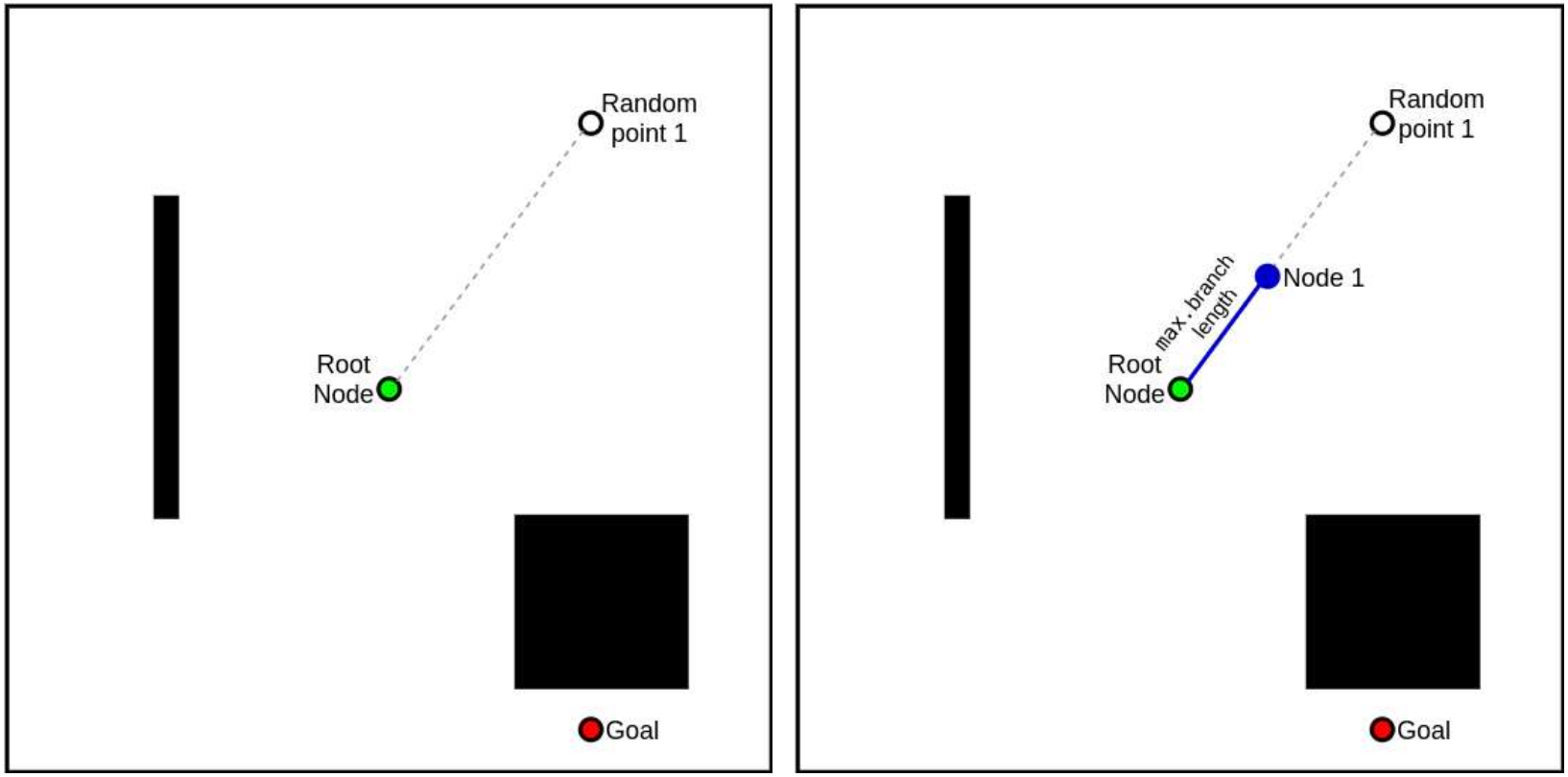


Then we want to connect *Random point 1* to the root node. But we cannot just plug it in, as we don't want to add a node that is too far away!

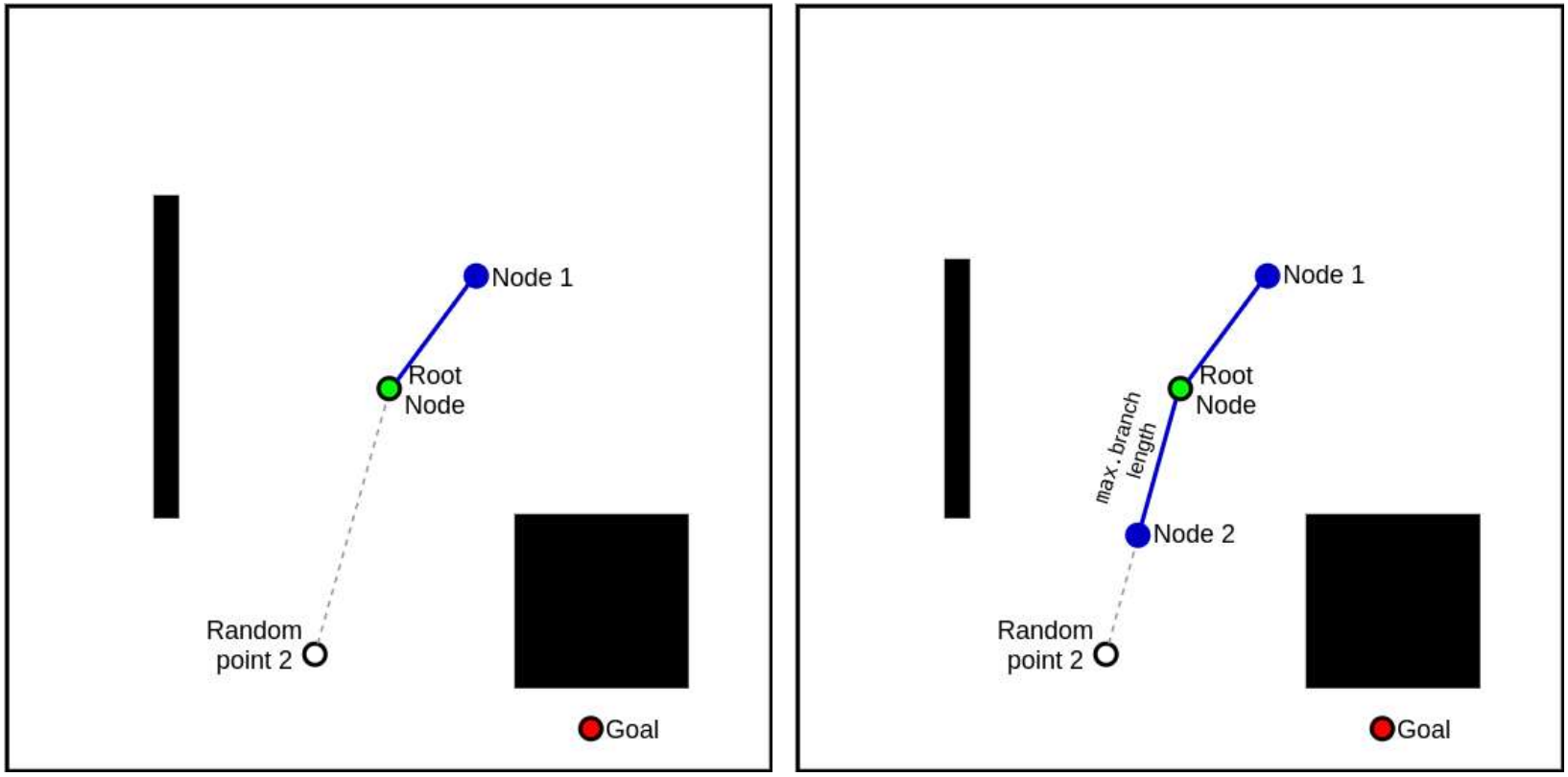
Why? Because the chance of crossing an obstacle or just travelling too far in the wrong direction is greater with a longer distance. Therefore, we specify a maximum distance, indicated in the diagram below as *max. branch length*, by which a new node can be away from an existing one. Intuitively, this maximum length limits the growth rate of the tree.

All right, since *Random point 1* is further away than the maximum allowed branch distance, we create a node called "Node 1" pointing in the direction of *Random point 1*, but at a distance equal to the **maximum branch length**.

And now we can connect it to the **root node**.

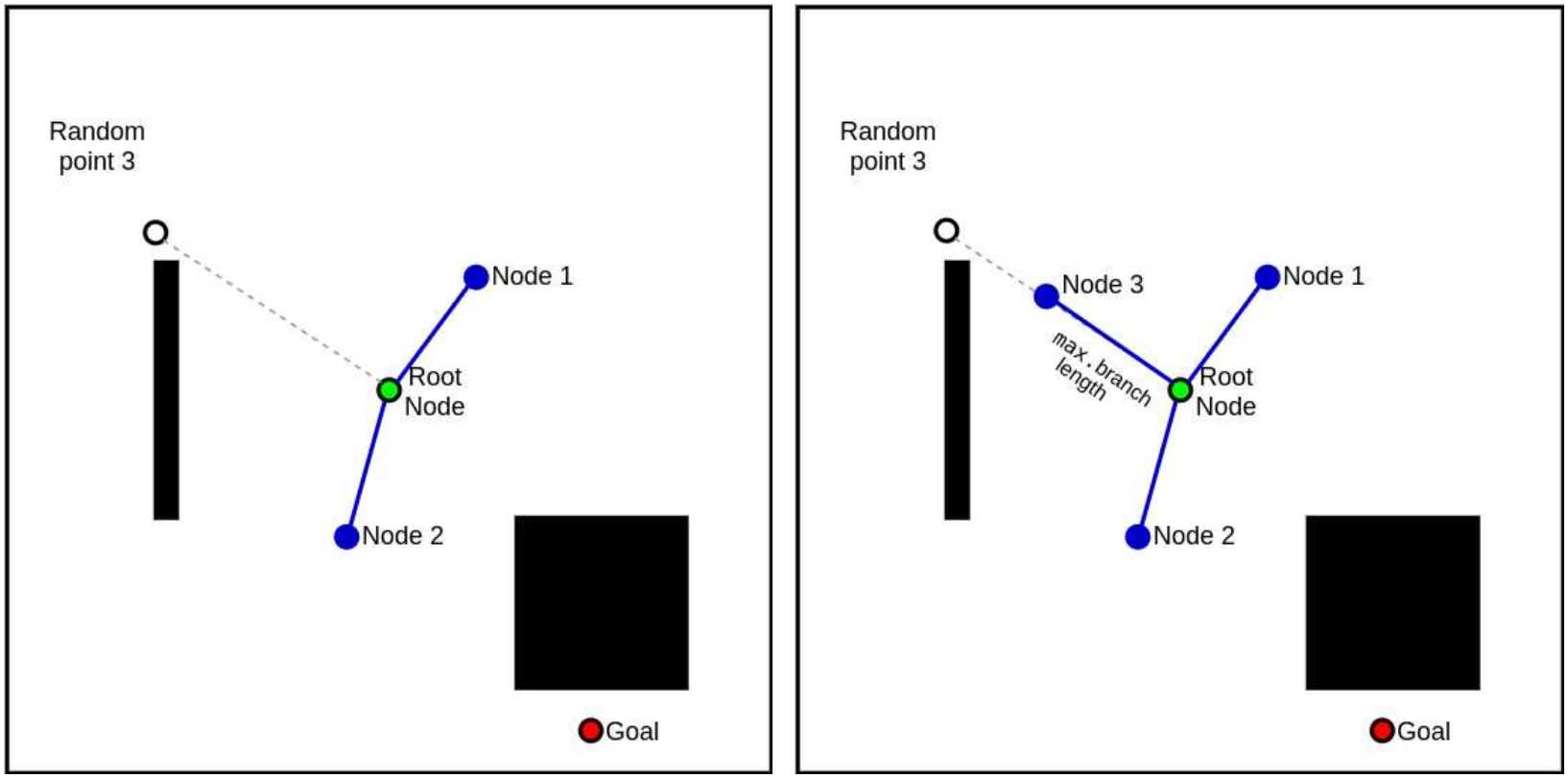


The algorithm continues by generating a new random point that we call "Random point 2". Notice that now we have two existing nodes: "Root Node" and "Node 1". To connect the new node to the tree, we have to find out which of the existing nodes is closer to it. In this example "Root Node" is closer to "Random point 2". We use "Root Node" as the basis to calculate the position of the new node based on the maximimum branch length distance and the direction given by "Random point 2". Then, "Node 2" and "Root Node" get connected as shown by the image below:



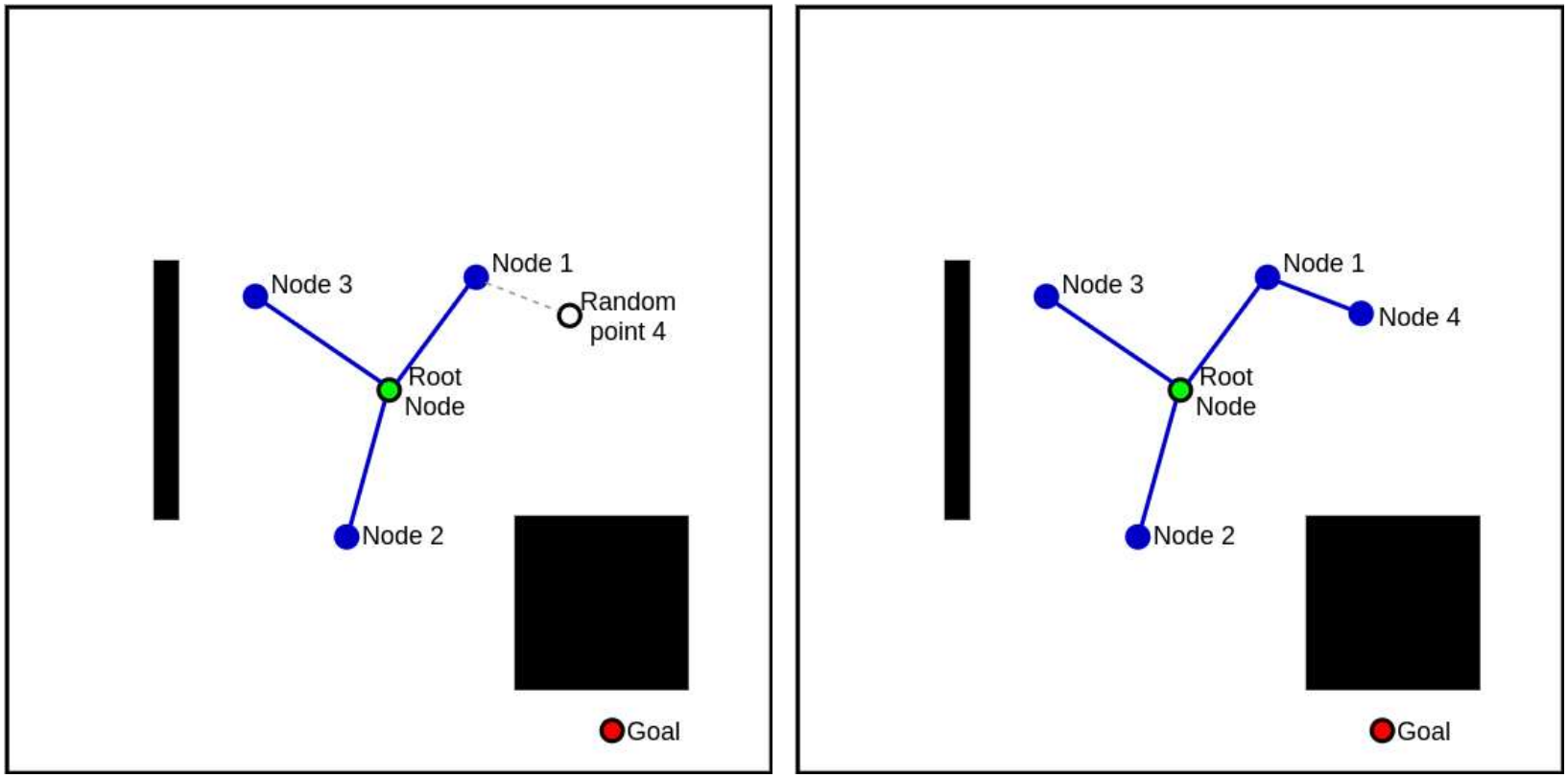
Following the same steps, over and over, the structure grows, adding more and more branches. The more iterations, the more it grows. As you will see, this structure will start to resemble a tree.

Below, we apply the same steps again, generating "Random point 3", finding the closest node, and taking into account the **maximum branch length** distance.



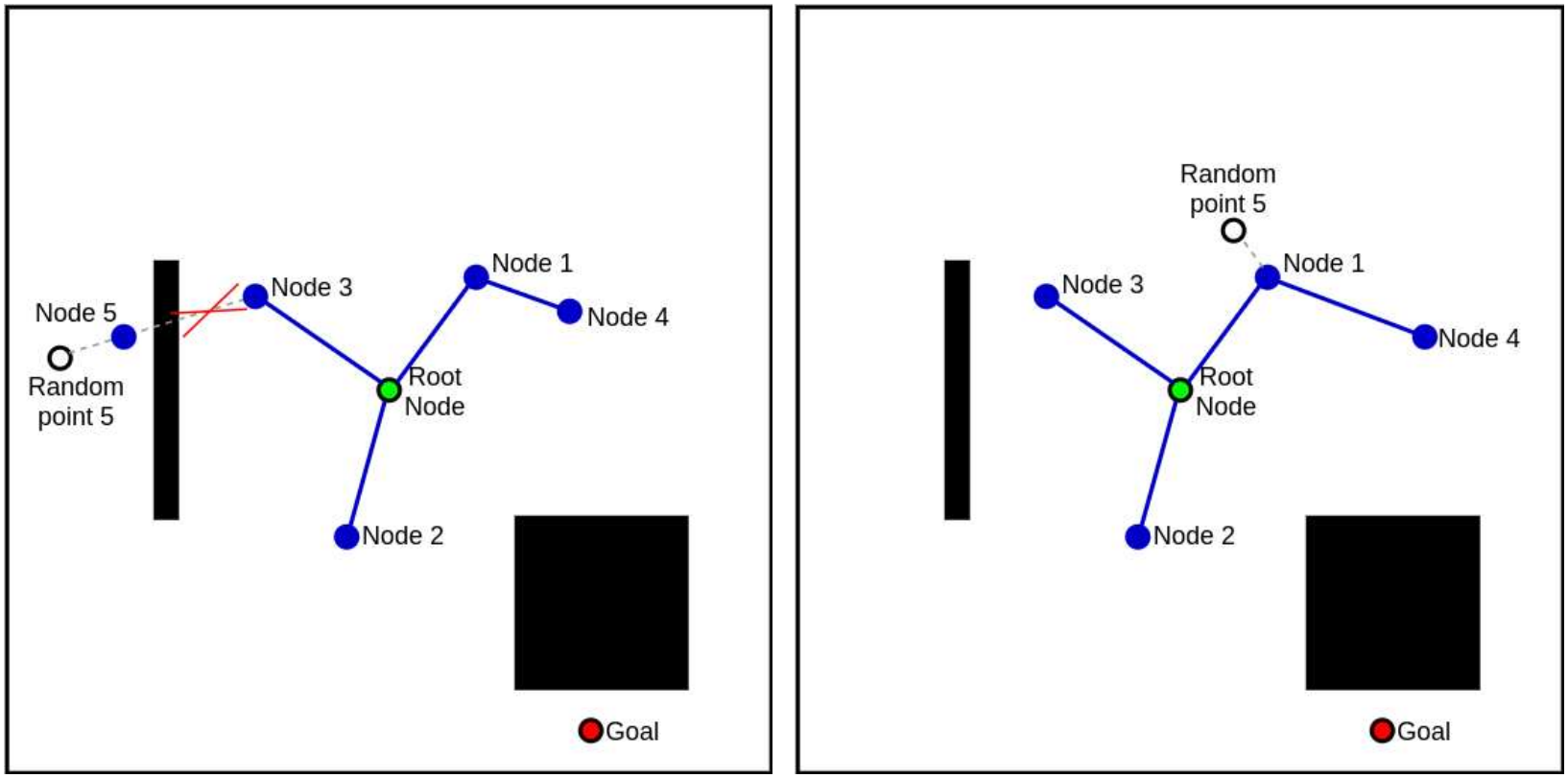
Fantastic! We continue with "Random point 4".

Note that the distance between "Random point 4" and the closest existing node in the tree (Node 1) is *less* than the maximum branch length. In this case, we just place the new node exactly where the random point was generated.



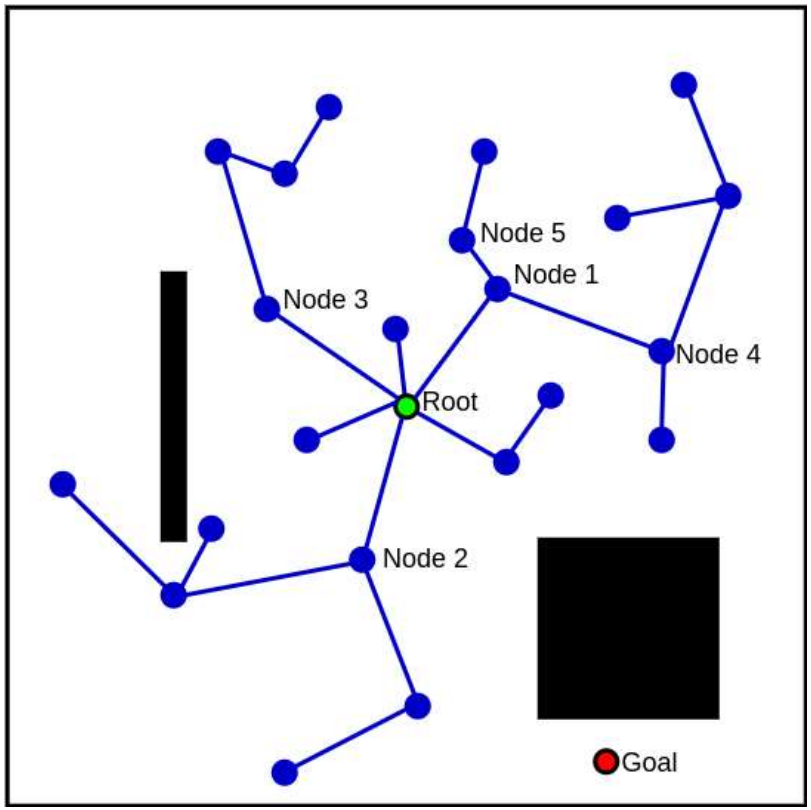
We repeat the process with "Random point 5".
Here is a new situation: what if the line that connects a new random point with its closest node crosses an obstacle? Obviously, it is very important that every segment that we add to the tree **does not cause our robot to bump into obstacles**. Therefore, the algorithm has to verify that every newly connected node has a **collision free branch**.

So this is an imporant rule: If a new branch crosses an obstacle, the new node is discarded and the current iteration cycle cancelled. A new iteration cycle can start with the generation of a new random point. In this example, we generate a new "Random point 5" as shown below right:



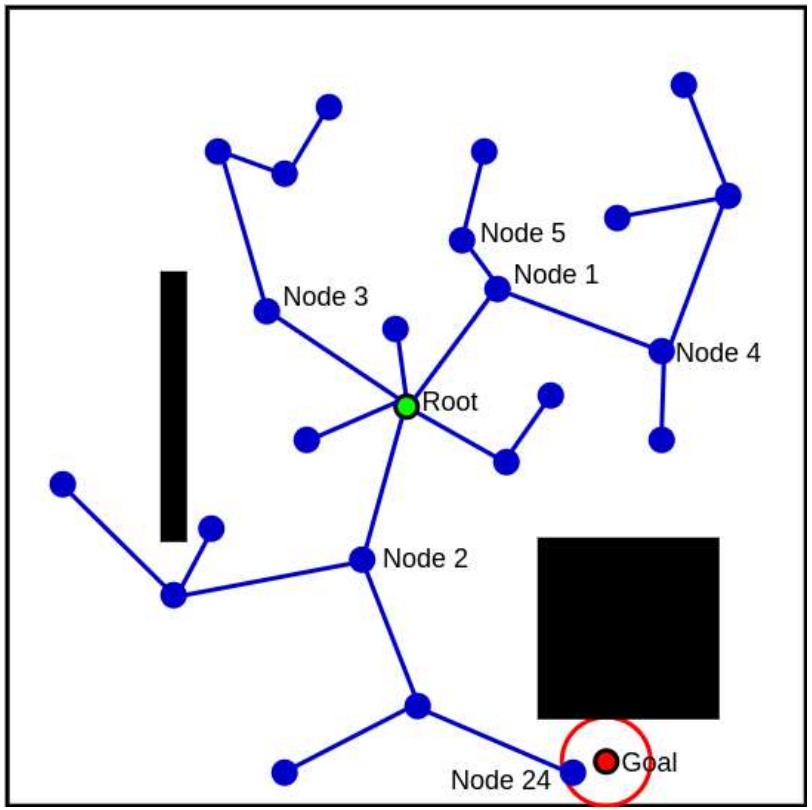
The name Rapidly Exploring Random Tree, or RRT, comes from the fact that the generation of random samples is very effective in biasing the extension of the tree towards unexplored areas of the map, thus causing the algorithm to *rapidly explore* the map.

Let's fast forward the process and look at the tree with 23 nodes:



The expansion of the tree continues until a connection to the goal node is found or a maximum number of iterations is exceeded.

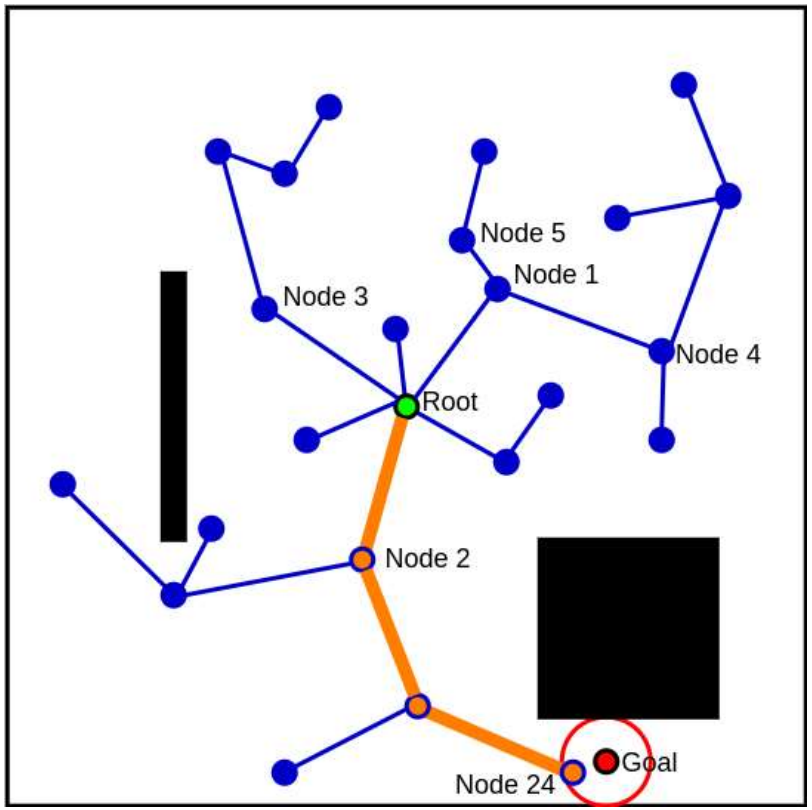
Finally, there is more thing to consider: since it's very unlikely that a random point gets generated exactly at the goal location, we need to define a certain distance to the goal, called **tolerance**, that is sufficient for the algorithm to declare success and stop the search. This tolerance is shown below as a red circle around the target position:



In our example, the algorithm succeeds in finding a path to the goal location with the creation of "Node 24".

The final step is to reconstruct the path between the start and the goal locations. We have seen this before: We start with the target node, find its parent node and put it on an output list. Then find the parent's parent node, its parent, etc. until we reach the start node, which is also put on the output list. Finally, we reverse it to get the right order before continuing.

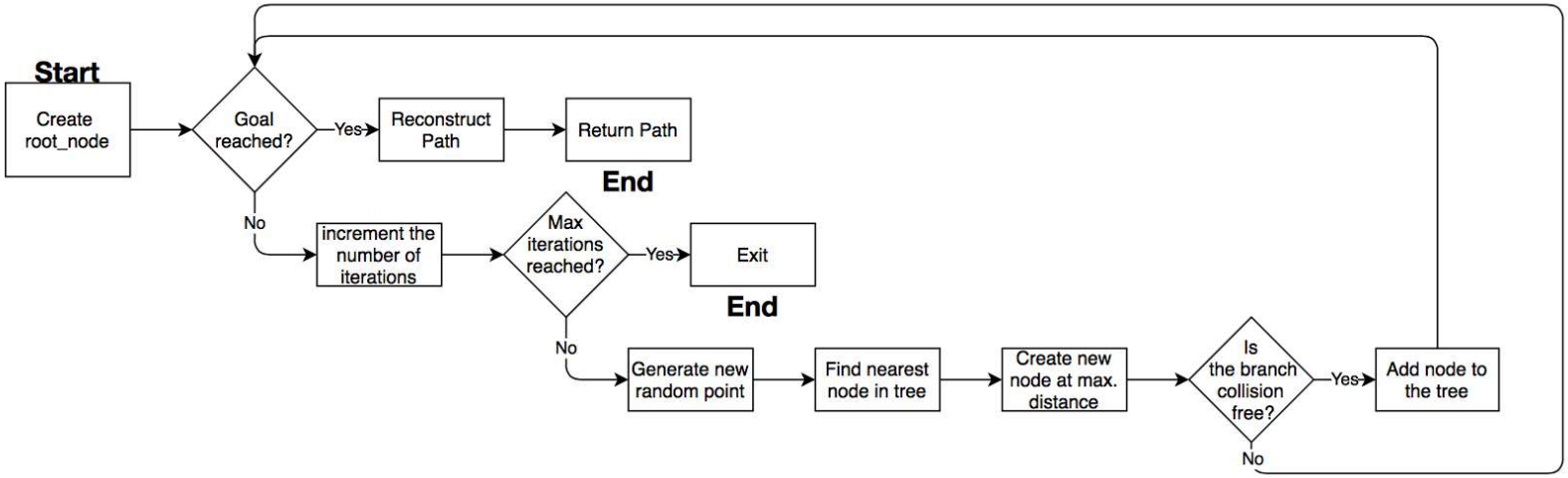
Below we show the final path with orange.



3.4 The basic RRT algorithm

Here's a short description of the process:

- Initialize a tree with the current robot position as the root node
- Generate a random point in space
- Find its nearest node in the existing tree
- Create a new node at the maximum branch length from the nearest node in the tree in the direction of the random point
- If the connecting edge does not intersect with an obstacle, attach the new node to the tree and run the same process again
- Else, try with a different random point
- Terminate the search when a newly added node is created within a tolerance distance to the goal or when a maximum number of iterations has been reached
- If the goal was found, reconstruct the path from root node to goal



3.5 RRT in Python

In this section, you'll be implementing RRT in Python. Once you have finished, you will be testing your script in Gazebo.

What is new is that in this exercise, we will have a class. This class is called *Node* and it is very simple. When we create a Node object, we can pass in the node's x,y coordinates, and a parent node, both of which are assigned to the member variables `coordinates` and `parent` respectively. Note that `coordinates` keeps a list and `parent` keeps another instance of `Node` .

```
# Node class
class Node:
  def __init__(self, coordinates, parent):
    # coordinates: list with [x,y] values of grid cell coordinates
    self.coordinates = coordinates
    # parent: Node object
    self.parent = parent
```

- Exercise 3.5.1 -

See the following example below:

- First, the class `Node` is defined
- Then, a Node object called `root_node` is created with coordinates [1,1] but without a parent node
- Next, a new instance of Node called `node1` is created; it has coordinates [5,5] and `root_node` as parent
- Finally, the coordinates of `root_node`, `node1`, and `node1.parent` are printed out

Note that for `root_node`, we omit the parent node (set it to `None`) as we can't set a parent node for the root node.

In [ ]:
```
# Node class
class Node:
    def __init__(self, coordinates, parent=None):
        self.coordinates = coordinates
        self.parent = parent

# Create a 'root_node' instance
root_node = Node([1,1])
# Create a 'node1' instance
node1 = Node([5,5],root_node)

# print coordinates for 'root_node'
print(root_node.coordinates)
# print coordinates for 'node1'
print(node1.coordinates)
# print coordinates for 'node1.parent'
print(node1.parent.coordinates)
```

Optional: You can test the code above by placing it into a python file if you want. Save the file with a name of your liking and make sure it is executable. Then using a WebShell, navigate to the directory and run `python <name_of_your_script.py>` to see it work.

- End of Exercise 3.5.1 -

In order to support you with the algorithm implementation, we will also provide some helper functions that you can use when you write your code. These will be introduced to you where appropriate.

In this unit, you will work on this file: **unit4_exercise.py**

You can open that file now:

1. Inside the code editor window, open the **unit4** folder located inside `catkin_ws/src/path_planning_intro`
2. Open `/scripts` and then the file **unit4_exercise.py**.

Your task is to complete the `rrt()` function. Good to start? Let's go!

- Exercise 3.5.2 -

The algorithm's initialization

In this exercise, you will create the names and data containers that the algorithm requires:

Tasks:

- Start by creating a root node: instantiate a new `Node` object by passing in `initial_position` and naming it `root_node` .
- Create a list named `nodes` and put `root_node` inside it. This list will hold all the nodes in the tree.
- Include a counter variable named `iterations` that will get incremented every time the main loop runs. Set it to `0` .
- Add a parameter `max_iterations` to define the maximum number of iterations and assign it a value, for instance `10000` .
- Add a parameter `max_branch_lenght` for setting the maximum length of each branch, give it a numeric value of `20` .
- Name a variable `goal_tolerance` and set a value of `20` to be used as tolerance for reaching a position goal.
- Add an empty list to hold the output path from start to goal, and name it `path` .

Add your code to the body of the function `rrt()` and save it afterwards.

In [ ]:
```
def rrt(initial_position, target_position, width, height, map, map_resolution, map_origin, tree_viz):
    '''
    Performs Rapidly-Exploring Random Trees (RRT) algorithm on a costmap with a given start and goal node
    '''
    ## Add your code ##
    pass
```

- End of Exercise 3.5.2 -

- Exercise 3.5.3 -

Main loop and new random point

In this exercise, you will add a loop that will repeat the core of the RRT search algorithm. This loop will repeat until the robot reaches its goal or the maximum number of iterations has been exceeded.

Tasks:

- Add a loop that repeats until a break statement is found at any point during its execution

Inside the body of the loop:

- Increment the number of iterations by one
- Check if the number of iterations has exceeded the parameter value of `max_iterations`
  - Terminate the loop if the maximum number of loop cycles is reached, return `path`
- Otherwise, generate a random point in the map. It should be a list that contains x and y coordinate values
- Assign it to a variable called `random_point`

Add your code immediately below your previous exercise and before the end of the RRT function.

- Save

In [ ]:
```
# implement main loop and new random point
```

Click HERE for a hint.

To generate a random integer number, use the `rand` function by passing in the range for the random value, like so: `rand(1000)`
This will output a random number between 0 and 1000.

- End of Exercise 3.5.3 -

- Exercise 3.5.4 -

Find the closest node in the current tree

In this exercise, you will code a function that searches for the closest node in the existing tree.

Tasks

- Complete the function `find_closest_node()` shown below
- It should take as arguments one x,y point (as a list with x,y coordinates values) and one list containing all Node instances connected to the tree
- When you call `find_closest_node()`, it should return the Node that is closest to the x,y point that was passed in
- If you wish, you can use the provided function `calculate_distance()`. It takes in two x,y positions formatted as lists, such as [10,22] and [2,3], and outputs the distance between those two points
- Add your code to the body of the function `find_closest_node()`, which is part of the provided file skeleton code
- Finally, inside the main loop, call `find_closest_node(random_point, nodes)` and assign the return value to the variable `closest_node`
- Save

In [1]:

```
def find_closest_node(random_pt, node_list):
    """
    Finds the closest node in the tree
    random_pt: a [x,y] point (as a list)
    node_list: list that keeps all nodes in the tree
    returns: Node instance that is the closest node in the tree
    """
    ## Add your code ##

    pass
```

Click HERE for a hint.

- End of Exercise 3.5.4 -

- Exercise 3.5.5 -

Create a new node towards the random point, at the maximum branch length allowed

In this exercise, you will have to add a function that returns a new pair of x,y coordinates at the maximum distance from the tree along the line to the randomly-chosen point.

Complete `create_new_branch_point()` , a function that takes in two sets of x,y coordinates and a maximum distance, and returns x and y values of the new point as a list.

Tasks

- Write a `create_branch()` function which takes a list with the x,y coordinates to go from, the node to go to (again as a list containing x and y values), and the maximum branch distance as arguments
- Make the function return a list containing the x and y coordinates correponding to the position of the new node
- You can use the `calculate_distance()` and `calculate_angle()` functions, which are provided to you
- Add your implementation to the body of the provided function `create_new_branch_point()`
- Then, inside the main loop, call `create_new_branch_point()` and assign the return value to the variable `candidate_point`
- Save

In [ ]:

```
def create_new_branch_point(p1, p2, max_distance):
    """
    Creates a new point at the max_distance towards a second point
    p1: the point to go from (as a list containing x and y values)
    p2: the point to go to (as a list containing x and y values)
    max_distance: the expand distance (in grid cells)
    returns: new point as a list containing x and y values
    """
    ## Add your code ##

    pass
```

Click HERE for a hint.

- End of Exercise 3.5.5 -

- Exercise 3.5.6 -

Verify that the new branch is collision free

In this exercise, you will add a call to the provided function `collision_detected()` and use it to determine whether to add the new node to the tree or not.

`collision_detected(p1, p2, map, map_width)` takes in two points as lists with x,y values, a map and the width of the map. It returns `True` if a collision is detected and `False` if the line between both points is collision free.

Tasks:

- Call the function `collision_detected()` by passing in the coordinates of the closest node in the tree, the newly created candidate point, the map, and the map width.
- If the line is collision free, create a new node named `latest_node` and add it to the tree
- Otherwise, do nothing and the main loop should start a new iteration
- Add your code to the main loop, after the value assignment to the variable `candidate_point` , which you performed in the previous exercise
- Optional: visualize the tree graph in Rviz
  To add a graphical representation of the RRT tree as it expands, add this line of code:
  `tree_viz.append(latest_node)`
- Save

Click HERE for a hint.

- End of Exercise 3.5.6 -

- Exercise 3.5.2 -

Check if the goal has been reached

Continuing with the implementation of RRT, add a function to test if the latest added node is within the tolerance distance of the goal position.

Tasks:

- Complete the body of the function `test_goal()` shown below and add your implementation to the program file
- This function should return a Boolean value, i.e. `True` when the goal is within the tolerance distance or `False` otherwise
- Use the function `test_goal()` from inside the RRT function to verify if the latest node added to the tree is within the tolerance distance of the goal
  - Exit the loop if the goal is within reach
  - If you want, you can inform the user with a message on the console using this syntax:
    `rospy.loginfo('RRT: Goal reached')`
- Save

In [2]:

```
def test_goal(p1, p_goal, tolerance):
    """
    Test if the goal has been reached considering a tolerance distance
    p1: a [x,y] point (as a list) from where to test
    p_goal: a [x,y] point (as a list) corresponding to the goal
    tolerance: distance margin (in grid cells) allowed around goal
    returns: True goal is within tolerance, False if goal is not within tolerance
    """
    ## Add your code ##

    pass
```

- End of Exercise 3.5.2 -

You've made good progress, but you're not done yet!
Once the goal is found, we use the tree structure created to extract a list of waypoints that connect the start position with the goal position.

So, let's finish it then!

- Exercise 3.5.7 -

Path reconstruction

Last but not least, we need to build the final path. Your task is to fill in the empty list `path` with the [x,y] coordinates that form the route from start node to target node.

Tasks:

- Place the goal coordinates into `path`
- Create a loop to work backwards from the target node to the root node
  - Start by adding the coordinates of `latest_node` to the list
  - Follow the parent nodes, adding them to the resulting path, until you reach the root node
  - Don't forget to add the coordinates corresponding to `root_node`
  - Reverse the path to get the correct order
- Make `rrt()` return `path`
- Save

In [ ]:
```
# Reconstruct path by working backwards from target
```

- End of Exercise 3.5.7 -

Great work, you made it! Your first probabilistic path planning algorithm is complete!

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

3.6 Testing

In this section, we will verify that RRT is as intended.

Open the **Graphical Tools** screen, then launch the file you created with the following command:

▶ Execute in WebShell #1

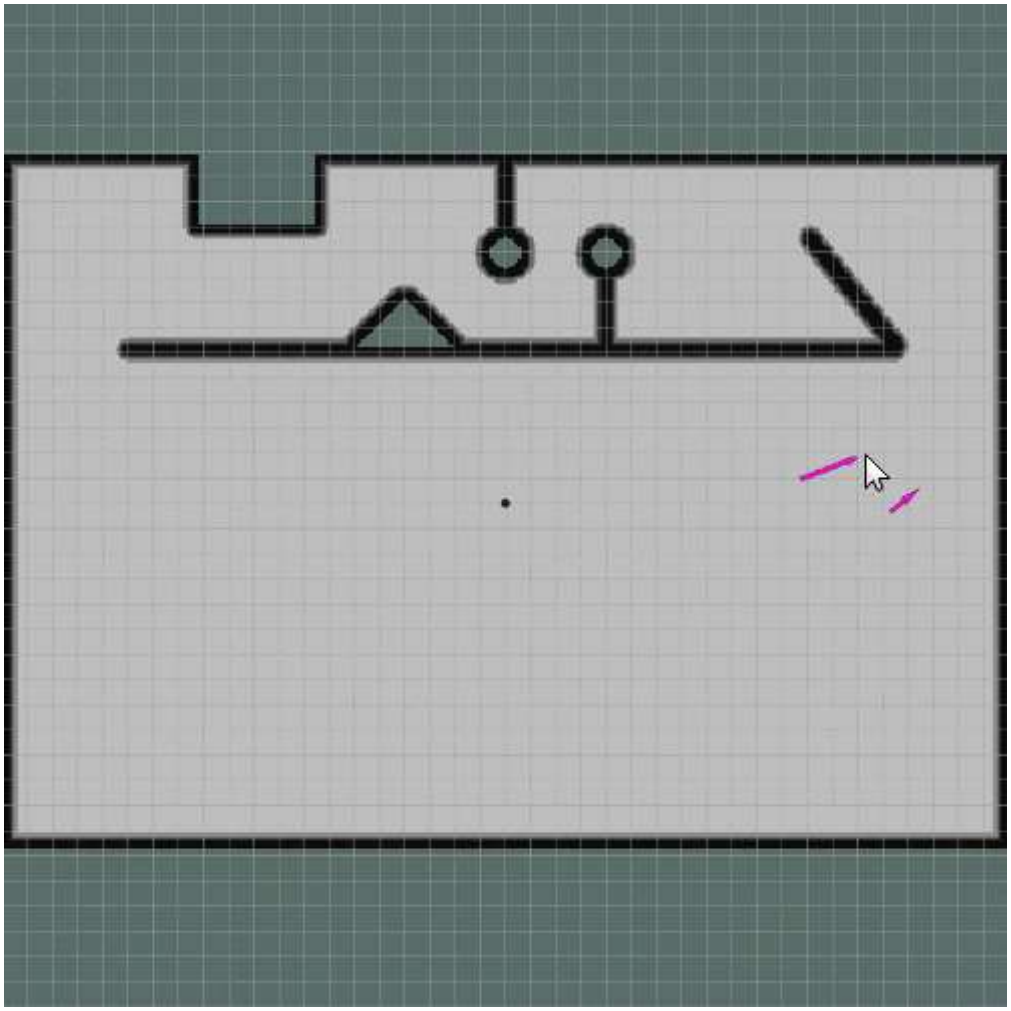
In [ ]:
```
roslaunch unit4 unit4_exercise.launch
```

Wait a few seconds until you can visualize the map and the robot model.

Make sure your script runs without any errors.

Use the **2D Nav Goal** button to give a goal position to the robot's path planner through RViz.

Expected Result



Set a few different goal positions to test the path planner thoroughly.

Experiment with a variety of values for `max_branch_lenght` , `max_iterations` , and `goal_tolerance` .

As you can see, this method is good for quickly finding a path. However, the path doesn't look anything close to what a shortest path is, right?

Go back to the original terminal and stop the execution using **Ctrl+C**.

3.7 Benefits, Challenges, and Variants of RRT

RRT has advantages in that it is fairly quick compared to other path planning algorithms. In some cases, this characteristic is very important, such as in real-time path planning within high-dimensional dynamic environments.

Additionally, RRT can be applied to both discrete maps (grid maps) and continuous map representations, allowing high resolution path planning with little computation.

One thing you surely noticed from testing RRT is that paths generated are jagged, often with redundant waypoints and sub-optimal in terms of the length (they are evidently longer compared to the paths generated by Dijkstra or A*). As a consequence, the time of traversal of a path or the time a robot takes to reach the goal is longer.

This outcome is characteristic of the random process that is used to generate new nodes. Expressed in formal terms, we say that RRT, at least in its traditional implementation, is **not optimal**.

An additional drawback is that even if a path exists, that path may not be found in finite time. In academic language, this is formally referred to as being an **probabilistically–complete** algorithm (it guarantees that the path is found only if time approaches infinity).

You must also consider the lack of repeatability of results. In other words, with the same start and goal location, the path computed by RRT greatly varies from one execution to another.

Most of these limitations are exclusively the result of the random nature of the tree expansion process. In this course we introduced you to the basic RRT algorithm, however, since its appearance, many other variations and optimizations have been published.

Many of these versions are related with the different strategies used for randomly placing nodes.
Some of them include:

- Create more nodes near obstacles
- Take into account non-holonomic contsraints
- Bias new nodes on the path towards the goal
- Have two or more root nodes or goal nodes
- ...and many others

And there are just as many creative names out there as RRT implementations, for instance:

- RRT Connect
- RRT*
- RRT#

- RRTX
- RRT*–Smart
- MP-RRT
- Informed RRT*
- ...among many others (https://en.wikipedia.org/wiki/Rapidly-exploring_random_tree#Variants_and_improvements_for_motion_planning)

An amazing area of expertise in robotics, isn't it? As they say, learning is a never-ending journey.

3.8 Summary

Ok, let's wrap this unit up by looking at its essential elements one last time:

- Sampling-based algorithms solve the path planning problem by generating random sample points on the map
- RRT works by incrementally constructing a tree from the start point towards such random sample points until the goal location is reached
- New nodes are added to the tree at a maximum distance, and not at the original position of the random point
- Additionally, new nodes are only connected if the direct line to them is collision free
- The search is successful if a new node is added within a certain tolerance distance of the goal
- Furthermore, a maximum number of iterations ensures that the algorithm does not run forever searching for the goal
- Several modifications have been made to the original RRT algorithm to enhance its capabilities