Experiment 10

Path Planning Algorithm using ROS

Aim:

To learn the Path Planning Algorithms (A* and Dijkstra's algorithms).

Software/ Package Used:

ROS

Programs:

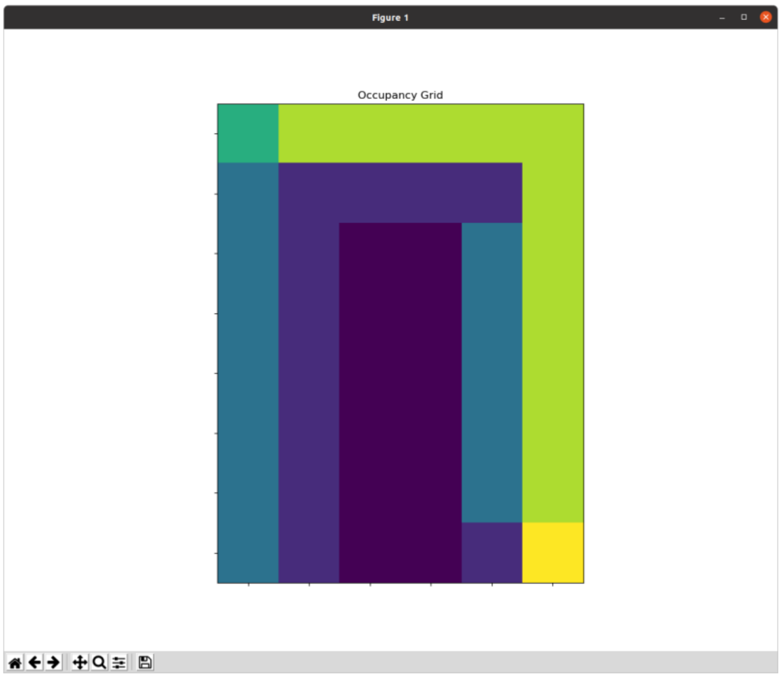
A* algorithm and Dijkstra's algorithm: https://realitybytes.blog/2018/08/17/graph-based-path-planning-a/amp/

https://github.com/SakshayMahna/Robotics-Playground/tree/main/turtlebot3_ws. (simulation)

1.

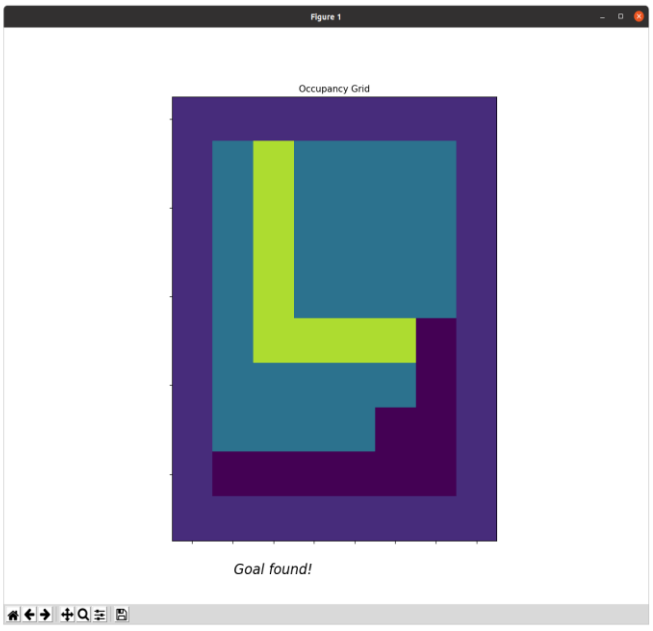
```
git clone https://github.com/atomoclast/realitybytes_blogposts.git

Cloning into 'realitybytes_blogposts'...

remote: Enumerating objects: 70, done.

remote: Counting objects: 100% (5/5), done.

remote: Compressing objects: 100% (5/5), done.

remote: Total 70 (delta 0), reused 2 (delta 0), pack-reused 65

Unpacking objects: 100% (70/70), 26.47 KiB | 392.00 KiB/s, done.

rae@raeCC40:~$ cd realitybytes_blogposts/

rae@raeCC40:~/realitybytes_blogposts$ cd pathplanning/

rae@raeCC40:~/realitybytes_blogposts/pathplanning$ chmod +x a_star.py

rae@raeCC40:~/realitybytes_blogposts/pathplanning$ ls

a_star.py  dijkstra.py

rae@raeCC40:~/realitybytes_blogposts/pathplanning$ python3 a_star.py

Heuristic:

[12, 11, 10, 9, 8, 7]

[11, 10, 9, 8, 7, 6]

[10, 9, 8, 7, 6, 5]

[9, 8, 7, 6, 5, 4]

[8, 7, 6, 5, 4, 3]
```

```
[7, 6, 5, 4, 3, 2]
[6, 5, 4, 3, 2, 1]
[5, 4, 3, 2, 1, 0]
[0, 0] [7, 5]
Found the goal in 20 iterations.
full_path:  [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5)]
['> ', '> ', '> ', '> ', '> ', 'v ']
['  ', '  ', '  ', '  ', '  ', 'v ']
['  ', '  ', '  ', '  ', '  ', 'v ']
['  ', '  ', '  ', '  ', '  ', 'v ']
['  ', '  ', '  ', '  ', '  ', 'v ']
['  ', '  ', '  ', '  ', '  ', 'v ']
['  ', '  ', '  ', '  ', '  ', 'v ']
['  ', '  ', '  ', '  ', '  ', '* ']
```

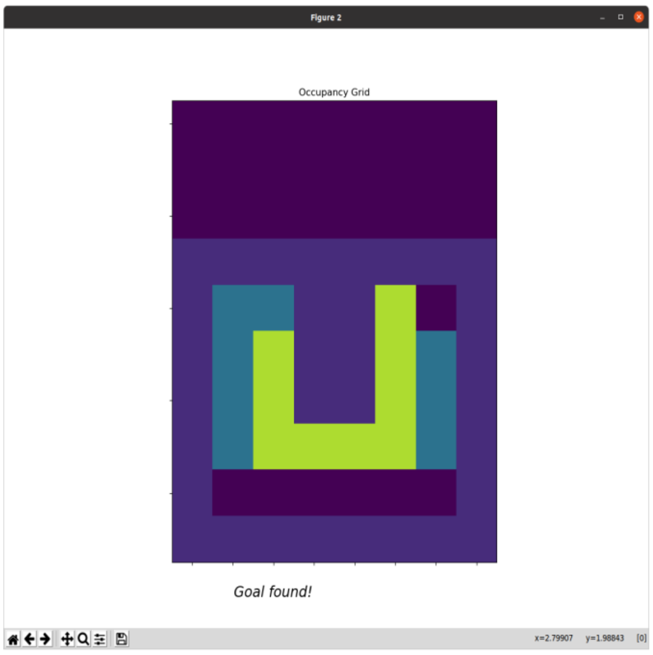


2.

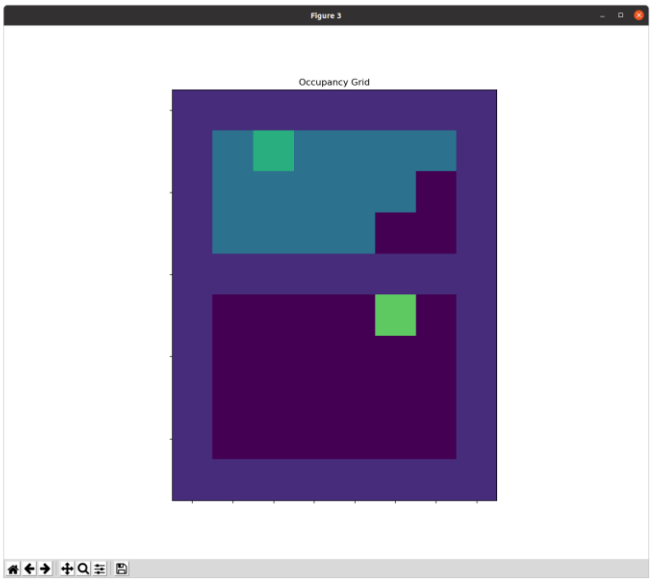
```
rae@raeCC40:~/realitybytes_blogposts/pathplanning$ ls
a_star.py  dijkstra.py
rae@raeCC40:~/realitybytes_blogposts/pathplanning$ python3 dijkstra.py
Start Pose:  2 1
Goal Pose:  5 5
Goal found!
Generating path...
Path:
[[0.3, 0.3],
 [0.325, 0.3],
 [0.325, 0.5],
 [0.325, 0.7],
 [0.325, 0.9],
 [0.325, 1.1],
 [0.455, 1.1],
 [0.585, 1.1],
 [0.6, 1.0]]
Path 1 passes
Start Pose:  2 5
Goal Pose:  5 4
Goal found!
Generating path...
Path:
[[0.5, 1.0],
 [0.5, 1.1],
 [0.5, 1.3],
 [0.5, 1.5],
 [0.7, 1.5],
 [0.9, 1.5],
 [1.1, 1.5],
 [1.1, 1.3],
 [1.1, 1.1],
 [1.1, 0.9]]
Path 2 passes
Start Pose:  2 1
Goal Pose:  5 5
Path 3 passes
```



Goal found!



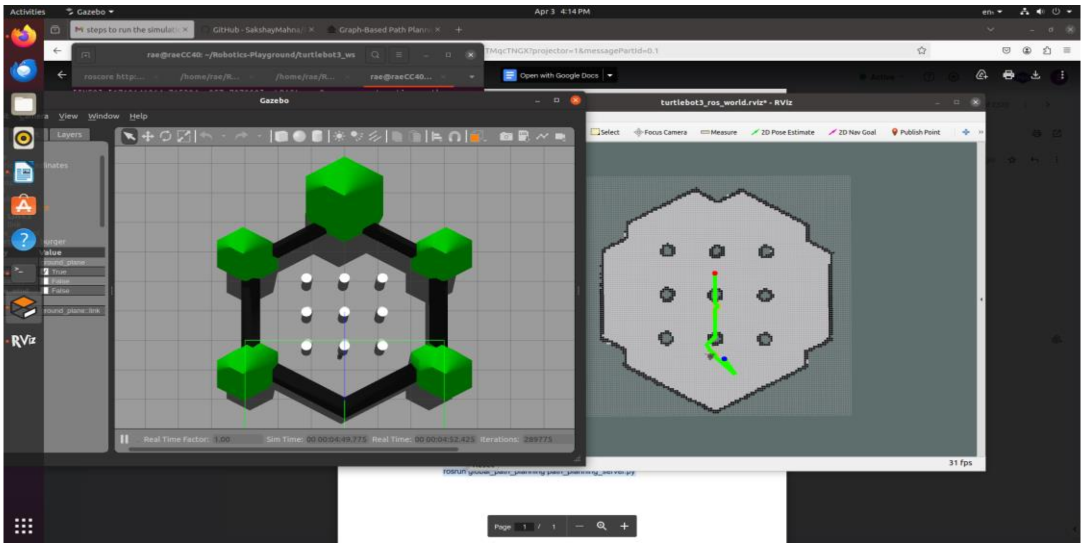
Goal found!



3.SIMULATION:

a)PATH PLANNING

```
#!/usr/bin/env python
import rospy
from pp_msgs.srv import PathPlanningPlugin, PathPlanningPluginResponse
from geometry_msgs.msg import Twist
from gridviz import GridViz
from algorithms.dijkstra import dijkstra
from algorithms.astar import astar
from algorithms.greedy import greedy
from algorithms.q_learning import q_learning
from algorithms.lpastar import lpastar

previous_plan_variables = None
def make_plan(req):
  Callback function used by the service server to process
  requests from clients. It returns a msg of type PathPlanningPluginResponse
  global previous_plan_variables

  # costmap as 1-D array representation
  costmap = req.costmap_ros
  # number of columns in the occupancy grid
  width = req.width
  # number of rows in the occupancy grid
  height = req.height
  start_index = req.start
  goal_index = req.goal
  # side of each grid map square in meters
  resolution = 0.05
  # origin of grid map
  origin = [-10, -10, 0]

  viz = GridViz(costmap, resolution, origin, start_index, goal_index, width)

  # time statistics
  start_time = rospy.Time.now()

  # calculate the shortes path
  path, previous_plan_variables = lpastar(start_index, goal_index, width, height, costmap, resolution,
origin, viz, previous_plan_variables)

  if not path:
    rospy.logwarn("No path returned by the path algorithm")
    path = []
  else:
    execution_time = rospy.Time.now() - start_time
    print("\n")
    rospy.loginfo('++++++++ Path Planning execution metrics ++++++++')
    rospy.loginfo('Total execution time: %s seconds', str(execution_time.to_sec()))
    rospy.loginfo('++++++++++++++++++++++++++++++++++++++++++++')
    print("\n")
    rospy.loginfo('Path sent to navigation stack')
  resp = PathPlanningPluginResponse()
  resp.plan = path
  return resp

def clean_shutdown():
  cmd_vel.publish(Twist())
  rospy.sleep(1)

if __name__ == '__main__':
  rospy.init_node('path_planning_service_server', log_level=rospy.INFO, anonymous=False)
  make_plan_service = rospy.Service("/move_base/SrvClientPlugin/make_plan", PathPlanningPlugin,
make_plan)
  cmd_vel = rospy.Publisher('/cmd_vel', Twist, queue_size=5)
  rospy.on_shutdown(clean_shutdown)
  while not rospy.core.is_shutdown():
    rospy.rostime.wallsleep(0.5)
  rospy.Timer(rospy.Duration(2), rospy.signal_shutdown('Shutting down'), oneshot=True)
```

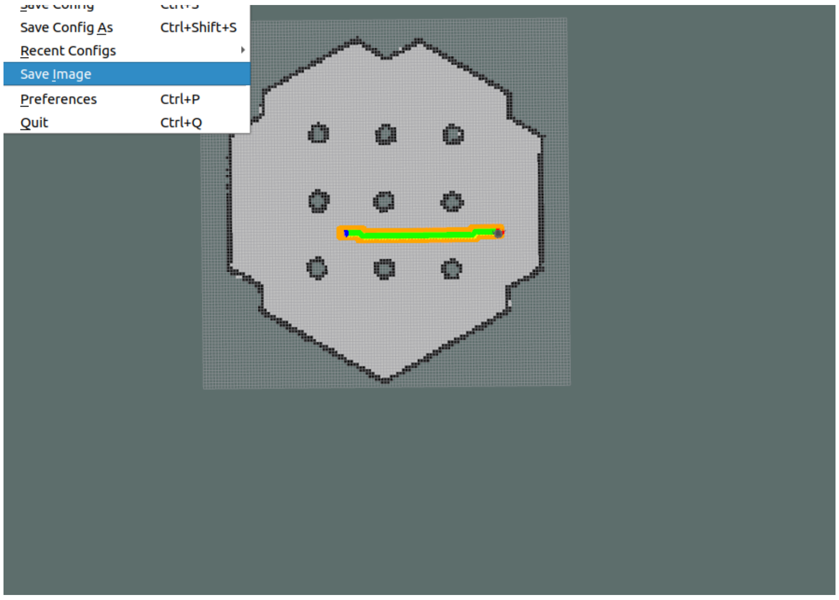


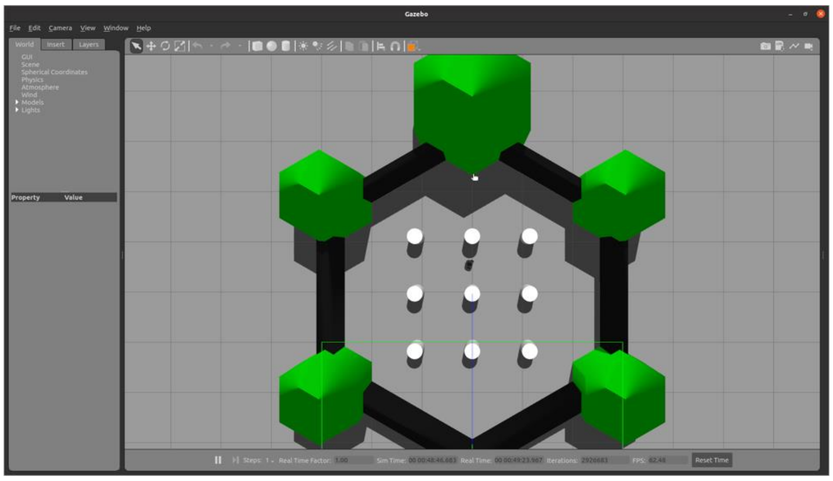
b)A* ALGORITHM

```
#! /usr/bin/env python3
import rospy
from math import sqrt
from algorithms.neighbors import find_neighbors
def euclidean_distance(index, goal_index, width):
  """ Heuristic Function for A Star algorithm"""
  index_x = index % width
  index_y = int(index / width)
  goal_x = goal_index % width
  goal_y = int(goal_index / width)
  distance = (index_x - goal_x) ** 2 + (index_y - goal_y) ** 2
  return sqrt(distance)


def astar(start_index, goal_index, width, height, costmap, resolution, origin, grid_viz,
previous_plan_variables):
  '''
  Performs A Star shortest path algorithm search on a costmap with a given start and goal node
  '''
  # create an open_list
  open_list = []
  # set to hold already processed nodes
  closed_list = set()
  # dict for mapping children to parent
  parents = dict()
  # dict for mapping g costs (travel costs) to nodes
  g_costs = dict()
  # dict for mapping f costs (heuristic + travel) to nodes
  f_costs = dict()
  # set the start's node g_cost and f_cost
  g_costs[start_index] = 0
  f_costs[start_index] = 0
  # add start node to open list
  start_cost = 0 + euclidean_distance(start_index, goal_index, width)
  open_list.append([start_index, start_cost])
  shortest_path = []
  path_found = False
```

```
rospy.loginfo('A Star: Done with initialization')
# Main loop, executes as long as there are still nodes inside open_list
while open_list:
  # sort open_list according to the lowest 'g_cost' value (second element of each sublist)
  open_list.sort(key = lambda x: x[1])
  # extract the first element (the one with the lowest 'g_cost' value)
  current_node = open_list.pop(0)[0]

  # Close current_node to prevent from visting it again
  closed_list.add(current_node)
  # Optional: visualize closed nodes
  grid_viz.set_color(current_node,"pale yellow")
  # If current_node is the goal, exit the main loop
  if current_node == goal_index:
    path_found = True
    break
  # Get neighbors of current_node
  neighbors = find_neighbors(current_node, width, height, costmap, resolution)
  # Loop neighbors
  for neighbor_index, step_cost in neighbors:
    # Check if the neighbor has already been visited
    if neighbor_index in closed_list:
      continue
    # calculate g_cost of neighbour considering it is reached through current_node
    g_cost = g_costs[current_node] + step_cost
    h_cost = euclidean_distance(neighbor_index, goal_index, width)
    f_cost = g_cost + h_cost
    # Check if the neighbor is in open_list
    in_open_list = False
    for idx, element in enumerate(open_list):
      if element[0] == neighbor_index:
        in_open_list = True
        break
    # CASE 1: neighbor already in open_list
    if in_open_list:
      if f_cost < f_costs[neighbor_index]:
        # Update the node's g_cost and f_cost
      g_costs[neighbor_index] = g_cost
        f_costs[neighbor_index] = f_cost
        parents[neighbor_index] = current_node
        # Update the node's g_cost inside open_list
        open_list[idx] = [neighbor_index, f_cost]
    # CASE 2: neighbor not in open_list
    else:
      # Set the node's g_cost and f_cost
      g_costs[neighbor_index] = g_cost
      f_costs[neighbor_index] = f_cost
      parents[neighbor_index] = current_node
      # Add neighbor to open_list
      open_list.append([neighbor_index, f_cost])
      # Optional: visualize frontier
      grid_viz.set_color(neighbor_index,'orange')
rospy.loginfo('AStar: Done traversing nodes in open_list')
if not path_found:
  rospy.logwarn('AStar: No path found!')
  return shortest_path
# Reconstruct path by working backwards from target
if path_found:
    node = goal_index
    shortest_path.append(goal_index)
    while node != start_index:
        shortest_path.append(node)
        # get next node
        node = parents[node]
```

```
# reverse list
shortest_path = shortest_path[::-1]
rospy.loginfo('AStar: Done reconstructing path')
```





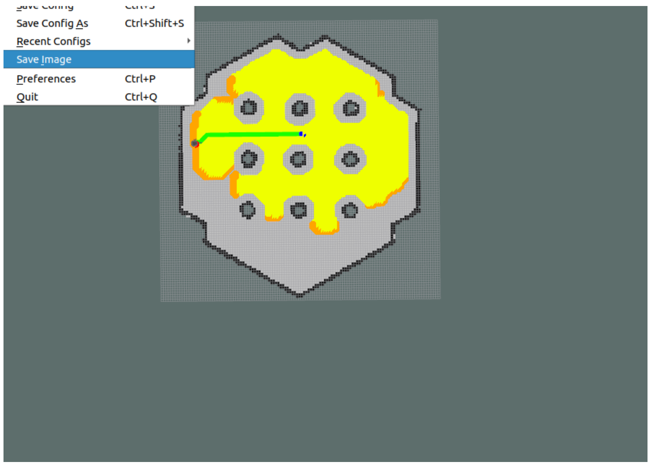
c)DIJIKSTRA

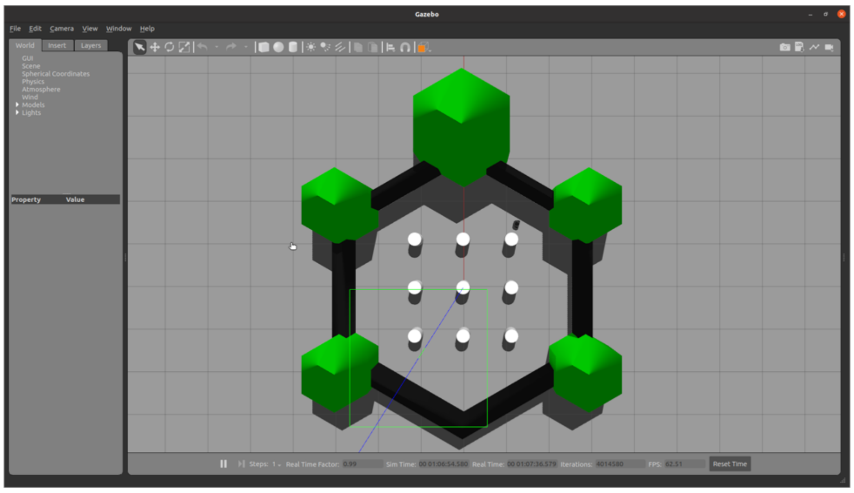
```
#! /usr/bin/env python3
"""
Dijkstra's algorithm path planning exercise solution
Author: Roberto Zegers R.
Copyright: Copyright (c) 2020, Roberto Zegers R.
License: BSD-3-Clause
Date: Nov 30, 2020
```

```
Usage: roslaunch unit2_pp unit2_solution.launch
"""
import rospy
from algorithms.neighbors import find_neighbors
def dijkstra(start_index, goal_index, width, height, costmap, resolution, origin, grid_viz, previous_plan_variables):
  '''
  Performs Dijkstra's shortes path algorithm search on a costmap with a given start and goal node
  '''
  # create an open_list
  open_list = []
  # set to hold already processed nodes
  closed_list = set()
  # dict for mapping children to parent
  parents = dict()
  # dict for mapping g costs (travel costs) to nodes
  g_costs = dict()
  # set the start's node g_cost
  g_costs[start_index] = 0
  # add start node to open list
  open_list.append([start_index, 0])
  shortest_path = []
  path_found = False
  rospy.loginfo('Dijkstra: Done with initialization')
  # Main loop, executes as long as there are still nodes inside open_list
  while open_list:
    # sort open_list according to the lowest 'g_cost' value (second element of each sublist)
    open_list.sort(key = lambda x: x[1])
    # extract the first element (the one with the lowest 'g_cost' value)
    current_node = open_list.pop(0)[0]
```

```
# Close current_node to prevent from visting it again
closed_list.add(current_node)
# Optional: visualize closed nodes
grid_viz.set_color(current_node,"pale yellow")
# If current_node is the goal, exit the main loop
if current_node == goal_index:
  path_found = True
  break
# Get neighbors of current_node
neighbors = find_neighbors(current_node, width, height, costmap, resolution)
# Loop neighbors
for neighbor_index, step_cost in neighbors:
  # Check if the neighbor has already been visited
  if neighbor_index in closed_list:
    continue
  # calculate g_cost of neighbour considering it is reached through current_node
  g_cost = g_costs[current_node] + step_cost
  # Check if the neighbor is in open_list
  in_open_list = False
  for idx, element in enumerate(open_list):
    if element[0] == neighbor_index:
      in_open_list = True
      break
  # CASE 1: neighbor already in open_list
  if in_open_list:
    if g_cost < g_costs[neighbor_index]:
      # Update the node's g_cost inside g_costs
      g_costs[neighbor_index] = g_cost
      parents[neighbor_index] = current_node
```

```
        # Update the node's g_cost inside open_list
        open_list[idx] = [neighbor_index, g_cost]

    # CASE 2: neighbor not in open_list
    else:
      # Set the node's g_cost inside g_costs
      g_costs[neighbor_index] = g_cost
      parents[neighbor_index] = current_node
      # Add neighbor to open_list
      open_list.append([neighbor_index, g_cost])
      # Optional: visualize frontier
      grid_viz.set_color(neighbor_index,'orange')
rospy.loginfo('Dijkstra: Done traversing nodes in open_list')
if not path_found:
  rospy.logwarn('Dijkstra: No path found!')
  return shortest_path
# Reconstruct path by working backwards from target
if path_found:
    node = goal_index
    shortest_path.append(goal_index)
    while node != start_index:
        shortest_path.append(node)
        # get next node
        node = parents[node]
# reverse list
shortest_path = shortest_path[::-1]
rospy.loginfo('Dijkstra: Done reconstructing path')
return shortest_path, None
```


Save Config Ctrl+S
Save Config As Ctrl+Shift+S
Recent Configs
Save Image
Preferences Ctrl+P
Quit Ctrl+Q


Gazebo
File Edit Camera View Window Help
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Atmosphere
Wind
Models
Lights
Property Value
Steps: 1
Real Time Factor: 0.99
Sim Time: 00 01:06:54.580
Real Time: 00 01:07:36.579
Iterations: 4014580
FPS: 62.51
Reset Time

Department of RAE			
Criteria	Excellent (75% - 100%)	Good (50 – 75%)	Poor (<50%)
Preparation (30)			
Performance (30)			
Evaluation (20)			
Report (20)			
Sign:		Total (100)	

Result:

The Path Planning algorithms were learnt using ROS.