# EECE 5550 Mobile Robotics Lab 3 submission

Yash Mewada

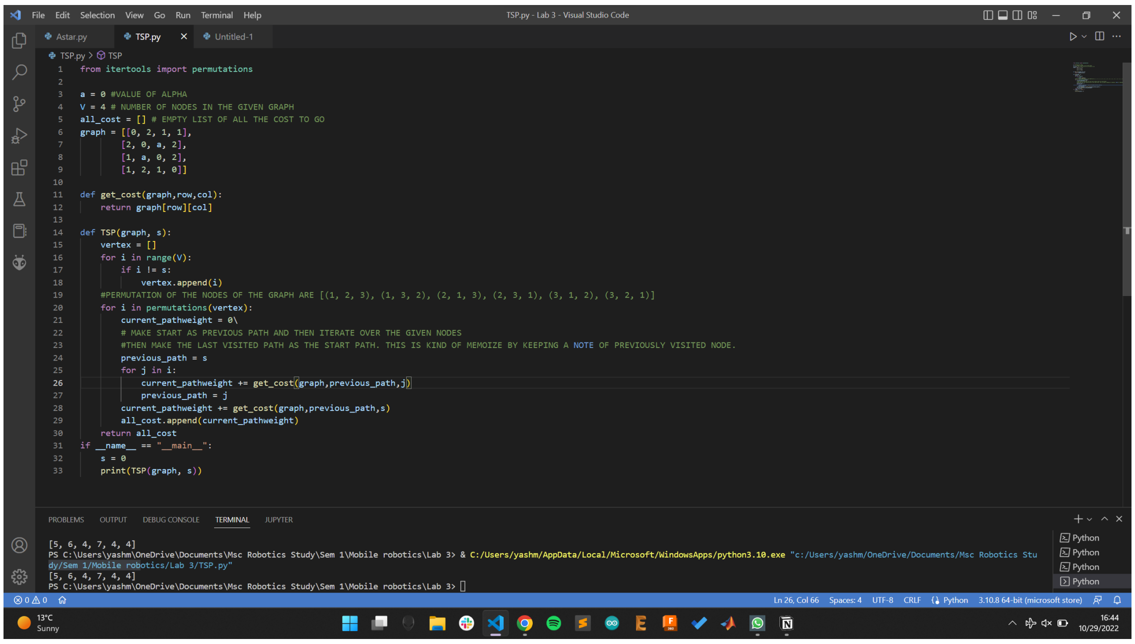
Nov 2,2022

## 1 Solution 1 - DP in Travelling salesman Problem

### 1.1 Computing the cost to go for each stage

The below program outputs the cost-to-go for each stage by using inbuilt permutation library of python. The matrix "Graph" is constructed using the values of the path weight given in the assignment. Assuming the value of $\alpha = 0$ initially the code is been compiled. The below is the cost to go for each stage moving from left to right in order.
Output of the TSP program = [5, 6, 4, 7, 4, 4]

ABCDA = 5 ABDCA = 6 ACBDA = 4 ACDBA = 7 ADBCA = 4 ADCBA = 4 –(1)



```python
from itertools import permutations

a = 0 #VALUE OF ALPHA
V = 4 # NUMBER OF NODES IN THE GIVEN GRAPH
all_cost = [] # EMPTY LIST OF ALL THE COST TO GO
graph = [[0, 2, 1, 1],
        [2, 0, a, 2],
        [1, a, 0, 2],
        [1, 2, 1, 0]]

def get_cost(graph,row,col):
    return graph[row][col]

def TSP(graph, s):
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
    #PERMUTATION OF THE NODES OF THE GRAPH ARE [(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
    for i in permutations(vertex):
        current_pathweight = 0\
        # MAKE START AS PREVIOUS PATH AND THEN ITERATE OVER THE GIVEN NODES
        #THEN MAKE THE LAST VISITED PATH AS THE START PATH. THIS IS KIND OF MEMOIZE BY KEEPING A NOTE OF PREVIOUSLY VISITED NODE.
        previous_path = s
        for j in i:
            current_pathweight += get_cost(graph,previous_path,j)
            previous_path = j
        current_pathweight += get_cost(graph,previous_path,s)
        all_cost.append(current_pathweight)
    return all_cost
if __name__ == "__main__":
    s = 0
    print(TSP(graph, s))
```

```
[5, 6, 4, 7, 4, 4]
PS C:\Users\yashm\OneDrive\Documents\Msc Robotics Study\Sem 1\Mobile robotics\Lab 3> & C:/Users/yashm/AppData/Local/Microsoft/WindowsApps/python3.10.exe "c:/Users/yashm/OneDrive/Documents/Msc Robotics Stu
dy/Sem 1/Mobile robotics/Lab 3/TSP.py"
[5, 6, 4, 7, 4, 4]
PS C:\Users\yashm\OneDrive\Documents\Msc Robotics Study\Sem 1\Mobile robotics\Lab 3>
```

Fig 1: Program for each cost to go

### 1.2 Interval for $\alpha$ that makes the route optimal

By placing different values of $\alpha = 0, 1, 2, 3, 4$ I found that the overall cost of each and every path does-not exceed the maximum value of the cost from the above solution i.e 7 until the value of $\alpha = 0, 1, 2$ after that the overall cost to go for each node goes beyond 7, this means it affects the optimal of the tree. Hence for the interval of $\alpha = 0 - 2$ the overall route remains optimal. The programming explanation is given as below.
Note here that if we consider $\alpha = 0$ then we assume there is no path between B and C, hence the range of $\alpha = 0, 1, 2$ suggests that either there exists no path and if it exists then its value will be either 1 or 2. Use notation (1) for comparison.
$\alpha = 0$ ;Route Cost = [5, 6, 4, 7, 4, 4]

$\alpha = 1$ ;Route Cost = [6, 6, 5, 7, 5, 5]

$\alpha = 2$ ;Route Cost = [7, 6, 6, 7, 6, 6]

$\alpha = 3$ **;Route Cost = [8, 6, 7, 7, 7, 7]**

$\alpha = 4$ **;Route Cost = [9, 6, 8, 7, 8, 8]**

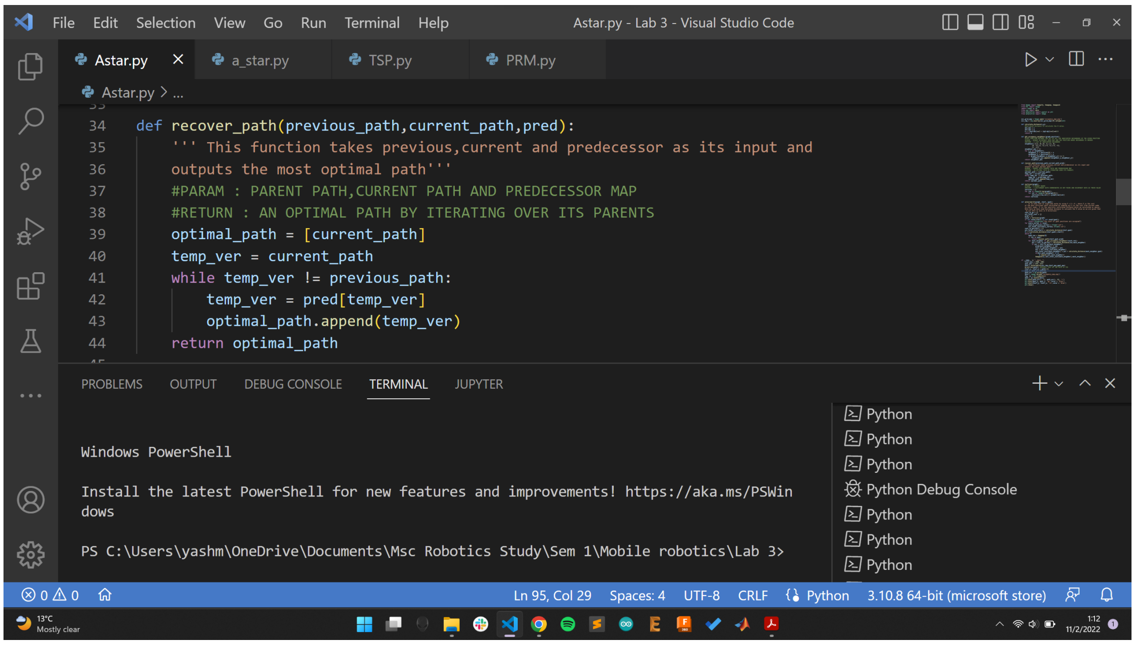
## 2 Solution 2 - Route Planning in Occupancy Grid Map

### 2.1 (a) A* Search algorithm implementation

Given the occupancy grid and threshold it to the binary values gives the binary map into 1's and 0's of the whole map.

#### 2.1.1 (i) Implement the Recover Path Function

The below snippet of the code is the implementation of the recover path function which takes the visited path, current path and Predecessor map as its input and outputs the optimal path list by retracing it from goal to start. When we plot a line over this optimal path coordinates we can see the shortest path drawn. This will be seen in further answers.



File Edit Selection View Go Run Terminal Help
Astar.py - Lab 3 - Visual Studio Code
Astar.py
a_star.py
TSP.py
PRM.py
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL JUPYTER
Python
Python Debug Console
Ln 95, Col 29 Spaces: 4 UTF-8 CRLF Python 3.10.8 64-bit (microsoft store)

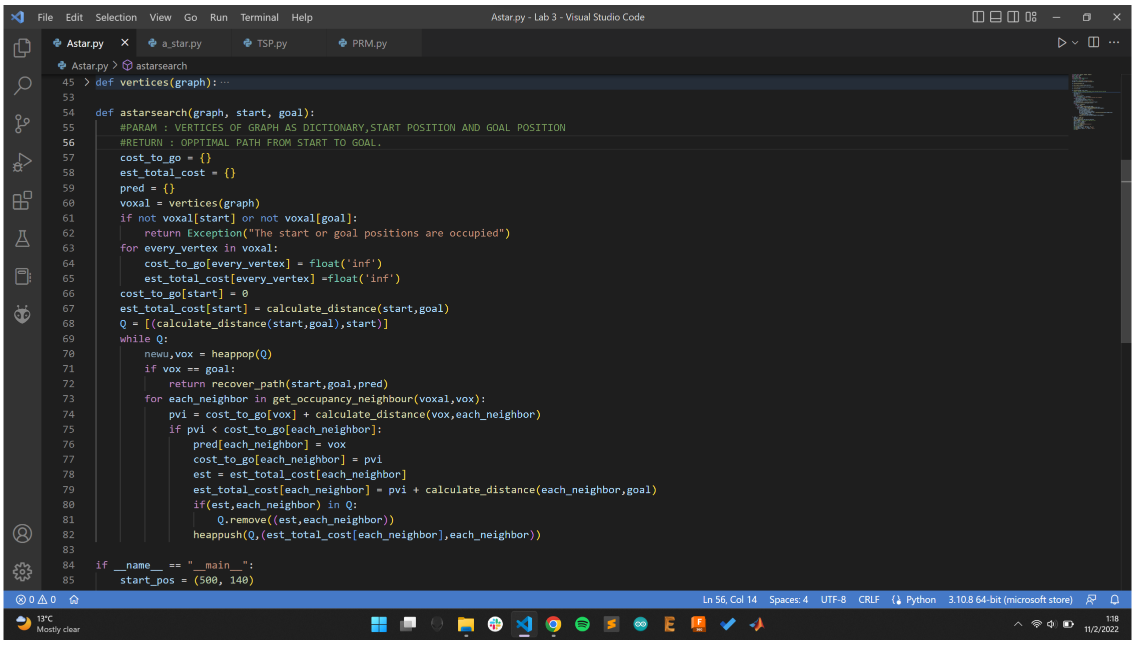
```python
def recover_path(previous_path,current_path,pred):
    ''' This function takes previous,current and predecessor as its input and
    outputs the most optimal path'''
    #PARAM : PARENT PATH,CURRENT PATH AND PREDECESSOR MAP
    #RETURN : AN OPTIMAL PATH BY ITERATING OVER ITS PARENTS
    optimal_path = [current_path]
    temp_ver = current_path
    while temp_ver != previous_path:
        temp_ver = pred[temp_ver]
        optimal_path.append(temp_ver)
    return optimal_path
```

```
Windows PowerShell

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWin
dows

PS C:\Users\yashm\OneDrive\Documents\Msc Robotics Study\Sem 1\Mobile robotics\Lab 3>
```

Fig 2: A* Reconstruct path program

#### 2.1.2 (ii) Implement the whole A* search algorithm

Using the pseudo code of the algorithm given in the file and changing few of the logic the below is the implementation of the A* search algorithm. Note here I have used Euclidean Formula to calculate the output of Heuristic function as it is evident that the occupancy graph is using 8 connected graphs and it is allowed to move in all 8 directions rather than just moving in 4 direction (Like the Manhattan distance formula).



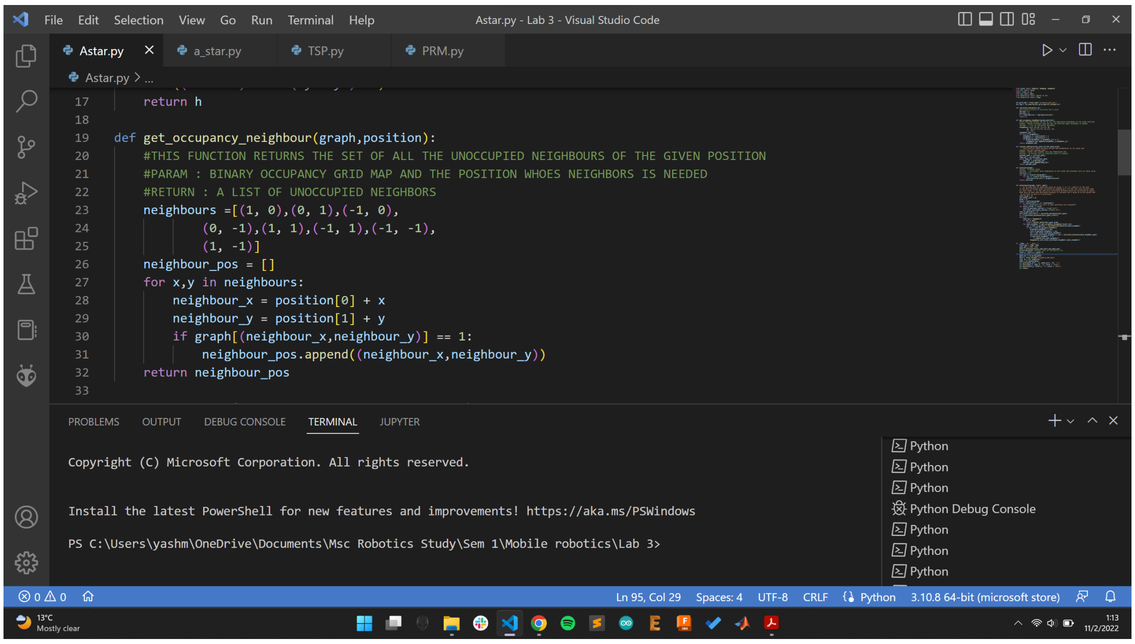
```python
def vertices(graph): ···

def astarsearch(graph, start, goal):
    #PARAM : VERTICES OF GRAPH AS DICTIONARY,START POSITION AND GOAL POSITION
    #RETURN : OPPTIMAL PATH FROM START TO GOAL.
    cost_to_go = {}
    est_total_cost = {}
    pred = {}
    voxal = vertices(graph)
    if not voxal[start] or not voxal[goal]:
        return Exception("The start or goal positions are occupied")
    for every_vertex in voxal:
        cost_to_go[every_vertex] = float('inf')
        est_total_cost[every_vertex] =float('inf')
    cost_to_go[start] = 0
    est_total_cost[start] = calculate_distance(start,goal)
    Q = [(calculate_distance(start,goal),start)]
    while Q:
        newu,vox = heappop(Q)
        if vox == goal:
            return recover_path(start,goal,pred)
        for each_neighbor in get_occupancy_neighbour(voxal,vox):
            pvi = cost_to_go[vox] + calculate_distance(vox,each_neighbor)
            if pvi < cost_to_go[each_neighbor]:
                pred[each_neighbor] = vox
                cost_to_go[each_neighbor] = pvi
                est = est_total_cost[each_neighbor]
                est_total_cost[each_neighbor] = pvi + calculate_distance(each_neighbor,goal)
                if(est,each_neighbor) in Q:
                    Q.remove((est,each_neighbor))
                heappush(Q,(est_total_cost[each_neighbor],each_neighbor))

if __name__ == "__main__":
    start_pos = (500, 140)
```

Fig 3: Implementation of A* Search Algorithm

## 2.2 Route Planning in Occupancy Grid Map with A* Search

### 2.2.1 (i) Implement the function N(v) that returns the set of unoccupied neighbors of a vertex v

The below code snippet shows the implementation of N(v) function which returns the set of free space/ unoccupied neighbors of the input vertexes. Also if the input points or the neighbors are greater than the size of the map it will throw an exception error. See the output of the below snippet in the terminal.



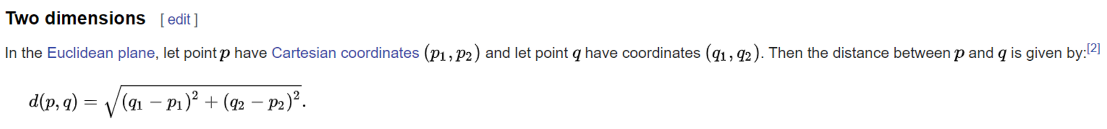
```python
    return h

def get_occupancy_neighbour(graph,position):
    #THIS FUNCTION RETURNS THE SET OF ALL THE UNOCCUPIED NEIGHBOURS OF THE GIVEN POSITION
    #PARAM : BINARY OCCUPANCY GRID MAP AND THE POSITION WHOES NEIGHBORS IS NEEDED
    #RETURN : A LIST OF UNOCCUPIED NEIGHBORS
    neighbours =[(1, 0),(0, 1),(-1, 0),
            (0, -1),(1, 1),(-1, 1),(-1, -1),
            (1, -1)]
    neighbour_pos = []
    for x,y in neighbours:
        neighbour_x = position[0] + x
        neighbour_y = position[1] + y
        if graph[(neighbour_x,neighbour_y)] == 1:
            neighbour_pos.append((neighbour_x,neighbour_y))
    return neighbour_pos
```

```
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\yashm\OneDrive\Documents\Msc Robotics Study\Sem 1\Mobile robotics\Lab 3>
```

Fig 4: Program for getting the occupied neighbor of the vertex of the graph.

### 2.2.2 (ii) Implement a distance function which returns Euclidean Distance

The euclidean distance is considered as the shortest distance between two points if it is permissible to move diagonally which is in our case. The below snippet is the implementation of the formula using the Fig 5 as reference.

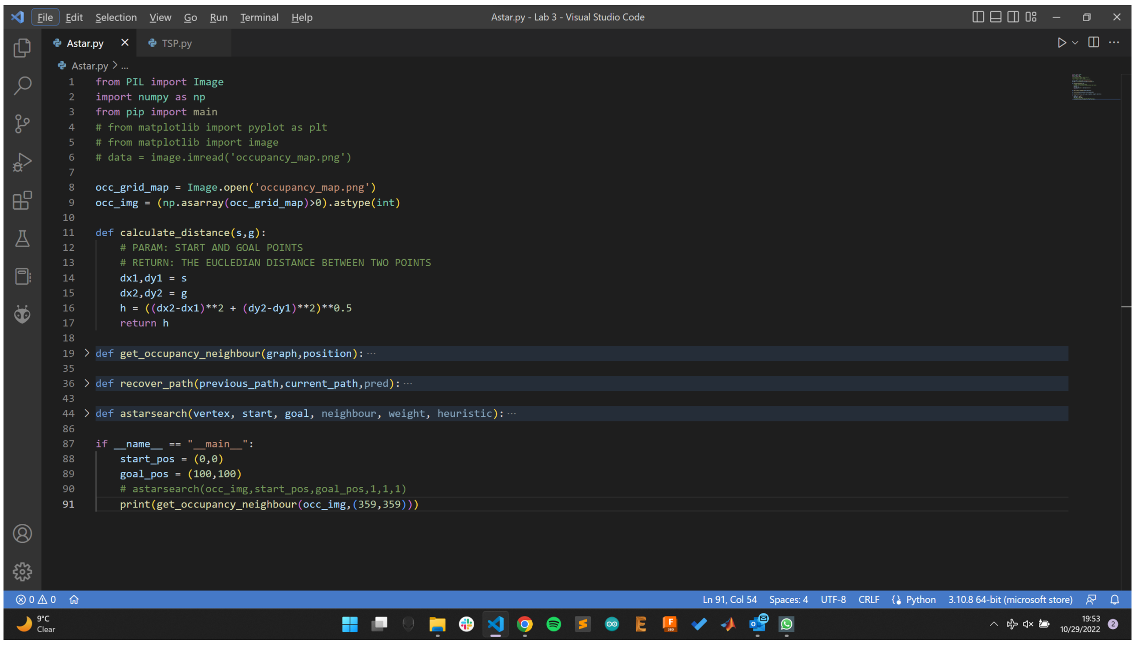


**Two dimensions** [ edit ]

In the Euclidean plane, let point $p$ have Cartesian coordinates $(p_1, p_2)$ and let point $q$ have coordinates $(q_1, q_2)$. Then the distance between $p$ and $q$ is given by:[2]

$$d(p,q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}.$$

Fig 5: Euclidean distance formula.



```python
from PIL import Image
import numpy as np
from pip import main
# from matplotlib import pyplot as plt
# from matplotlib import image
# data = image.imread('occupancy_map.png')

occ_grid_map = Image.open('occupancy_map.png')
occ_img = (np.asarray(occ_grid_map)>0).astype(int)

def calculate_distance(s,g):
    # PARAM: START AND GOAL POINTS
    # RETURN: THE EUCLEDIAN DISTANCE BETWEEN TWO POINTS
    dx1,dy1 = s
    dx2,dy2 = g
    h = ((dx2-dx1)**2 + (dy2-dy1)**2)**0.5
    return h

def get_occupancy_neighbour(graph,position): ···

def recover_path(previous_path,current_path,pred): ···

def astarsearch(vertex, start, goal, neighbour, weight, heuristic): ···

if __name__ == "__main__":
    start_pos = (0,0)
    goal_pos = (100,100)
    # astarsearch(occ_img,start_pos,goal_pos,1,1,1)
    print(get_occupancy_neighbour(occ_img,(359,359)))
```

File Edit Selection View Go Run Terminal Help
Astar.py - Lab 3 - Visual Studio Code
Astar.py
TSP.py
Ln 91, Col 54 Spaces: 4 UTF-8 CRLF Python 3.10.8 64-bit (microsoft store)
9°C Clear
19:53 10/29/2022

Fig 6: Program for getting the Euclidean distance between two points.

### 2.2.3 (iii) Implementation of A* search with given start and goal positions

The below is the output of the A* search algorithm for given start position as (635,140) and goal as (350,400). Also the overall cost of the path is 725m (assuming the scales of the graph is in meters.)

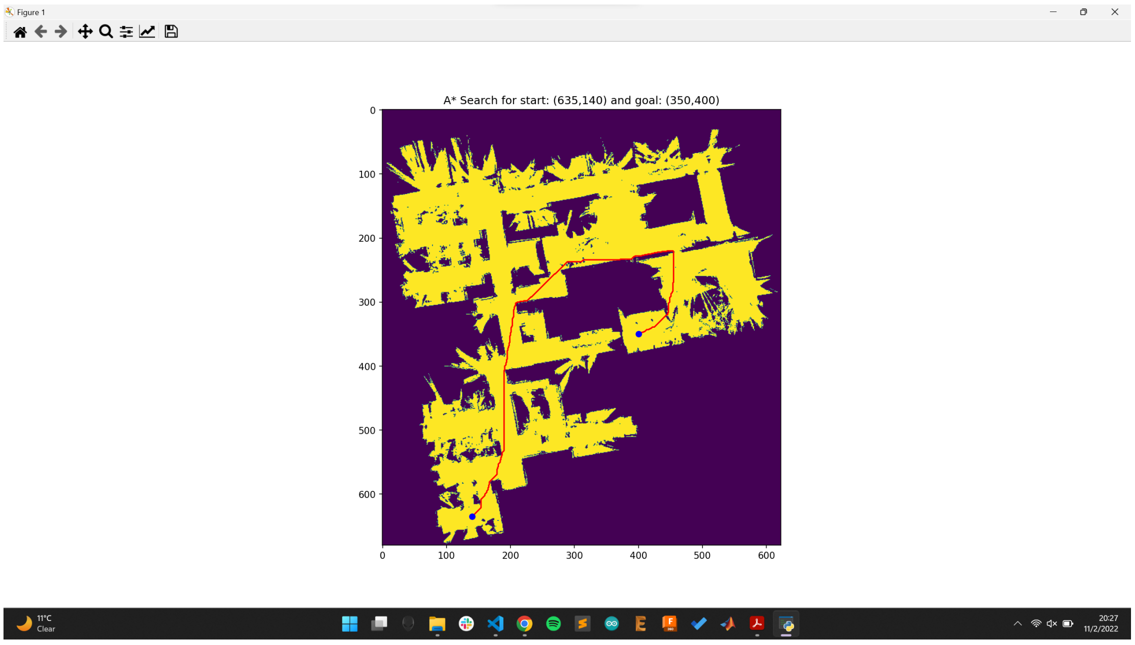


Fig 7: Output of A* search.

The below is the output of A* search if the start and goal positions are changed to start = (500, 140) goal = (350, 500). In this case the overall cost of the path becomes 577m.

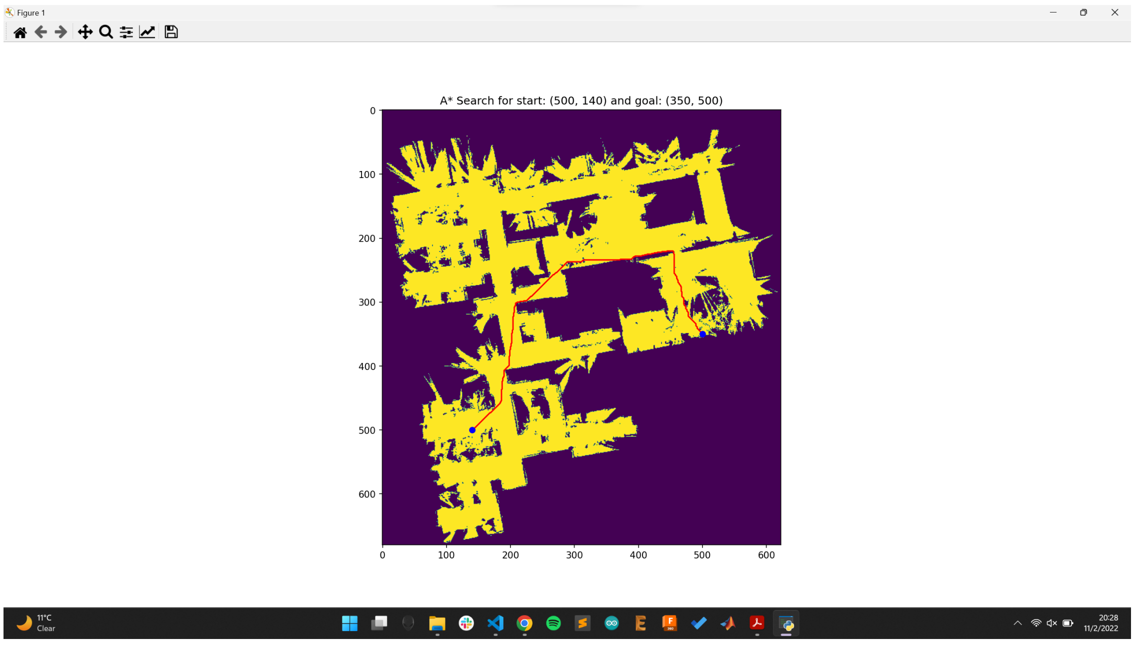
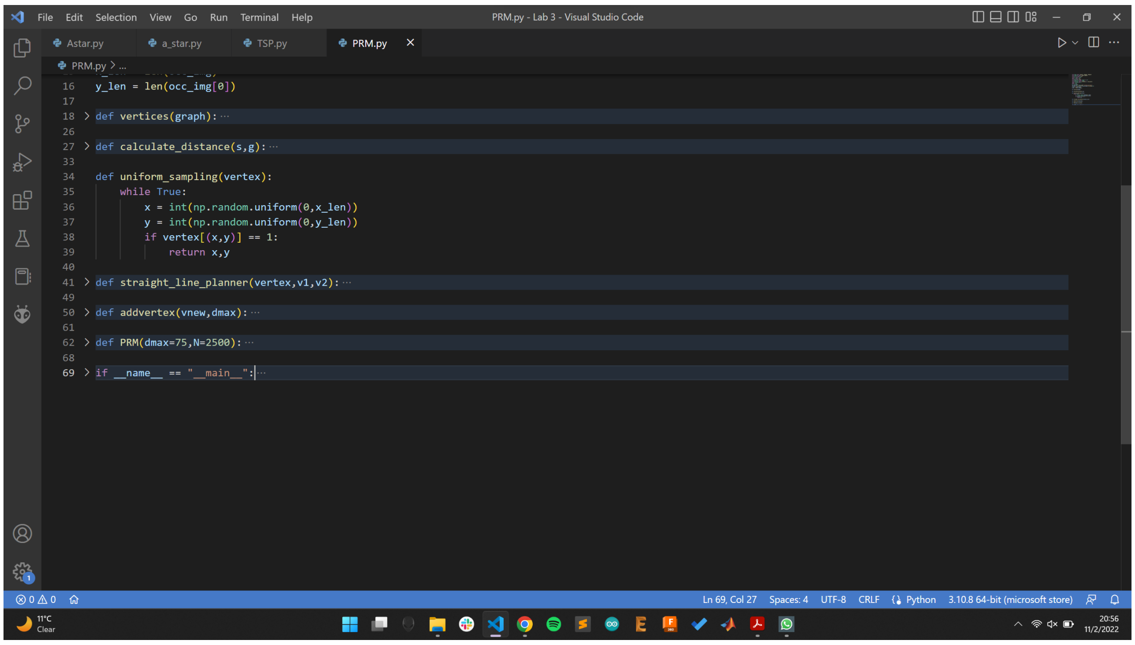


Fig 8: Output of A* search for different positions.

## 2.3 Route Planning with Probabilistic Road maps

### 2.3.1 (i) Implement a function which outputs a uniformly sampled unoccupied vertex

As it is given that the sampling must be uniform and random, I used the inbuilt function of generating a random number. The below is the implementation of that.

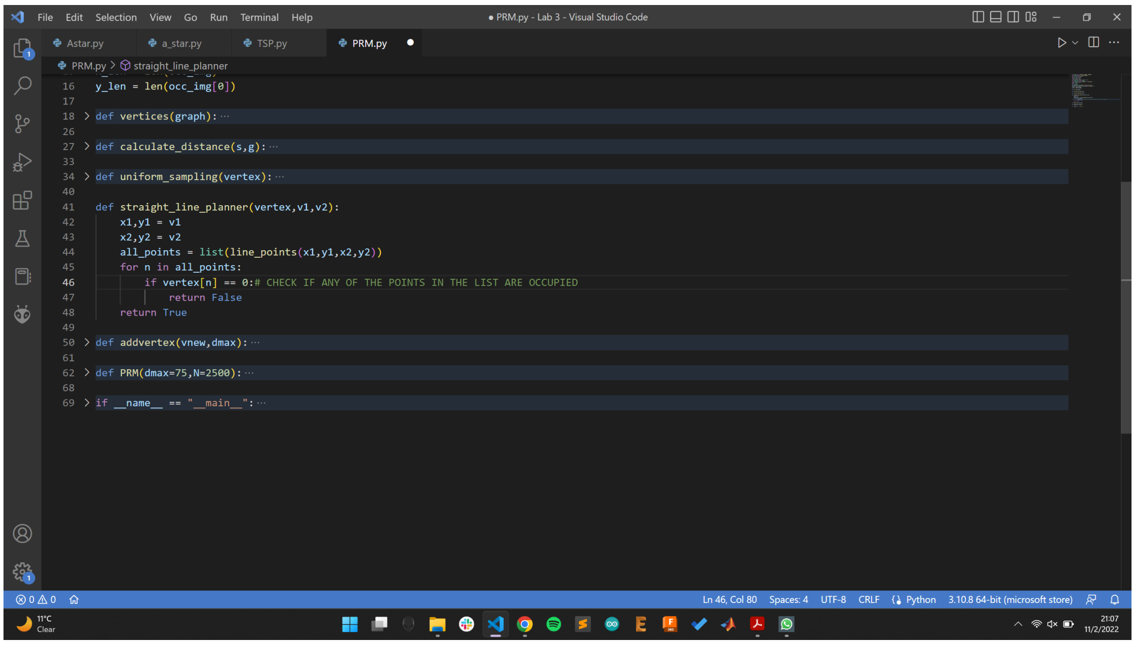


```python
y_len = len(occ_img[0])

def vertices(graph):···

def calculate_distance(s,g):···

def uniform_sampling(vertex):
    while True:
        x = int(np.random.uniform(0,x_len))
        y = int(np.random.uniform(0,y_len))
        if vertex[(x,y)] == 1:
            return x,y

def straight_line_planner(vertex,v1,v2):···

def addvertex(vnew,dmax):···

def PRM(dmax=75,N=2500):···

if __name__ == "__main__":···
```

Fig 9: Uniform randomly sampled vertex.

### 2.3.2 (ii) Reachability check algorithm

After each randomly generated vertex is obtained we need to find all the points on the line between the points and return true only if all the points on that line are unoccupied. For that I have used Bresenham Library in python which outputs all the points.
Below is the implementation of the same.

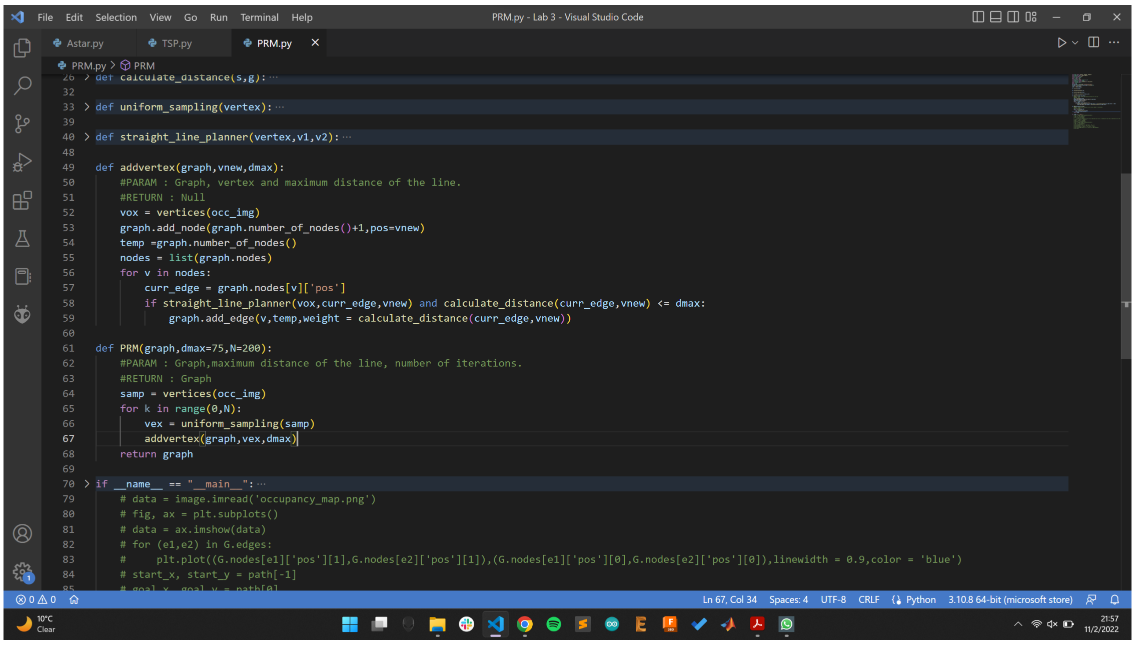


```python
y_len = len(occ_img[0])

def vertices(graph):···

def calculate_distance(s,g):···

def uniform_sampling(vertex):···

def straight_line_planner(vertex,v1,v2):
    x1,y1 = v1
    x2,y2 = v2
    all_points = list(line_points(x1,y1,x2,y2))
    for n in all_points:
        if vertex[n] == 0:# CHECK IF ANY OF THE POINTS IN THE LIST ARE OCCUPIED
            return False
    return True

def addvertex(vnew,dmax):···

def PRM(dmax=75,N=2500):···

if __name__ == "__main__":···
```

Fig 10: Attempt a safe and unoccupied path.

### 2.3.3 (iii) Implement PRM algorithm

The below snippet depicts the implementation of PRM algorithm and also the implementation of adding a new vertex under the condition that the distance must be less than 75 and edge must be unoccupied.



```python
def calculate_distance(s,g): ···

def uniform_sampling(vertex): ···

def straight_line_planner(vertex,v1,v2): ···

def addvertex(graph,vnew,dmax):
    #PARAM : Graph, vertex and maximum distance of the line.
    #RETURN : Null
    vox = vertices(occ_img)
    graph.add_node(graph.number_of_nodes()+1,pos=vnew)
    temp =graph.number_of_nodes()
    nodes = list(graph.nodes)
    for v in nodes:
        curr_edge = graph.nodes[v]['pos']
        if straight_line_planner(vox,curr_edge,vnew) and calculate_distance(curr_edge,vnew) <= dmax:
            graph.add_edge(v,temp,weight = calculate_distance(curr_edge,vnew))

def PRM(graph,dmax=75,N=200):
    #PARAM : Graph,maximum distance of the line, number of iterations.
    #RETURN : Graph
    samp = vertices(occ_img)
    for k in range(0,N):
        vex = uniform_sampling(samp)
        addvertex(graph,vex,dmax)
    return graph

if __name__ == "__main__": ···
    # data = image.imread('occupancy_map.png')
    # fig, ax = plt.subplots()
    # data = ax.imshow(data)
    # for (e1,e2) in G.edges:
    #     plt.plot((G.nodes[e1]['pos'][1],G.nodes[e2]['pos'][1]),(G.nodes[e1]['pos'][0],G.nodes[e2]['pos'][0]),linewidth = 0.9,color = 'blue')
    # start_x, start_y = path[-1]
    # goal_x, goal_y = path[0]
```

Fig 11: PRM algorithm with adding edge.

### 2.3.4 (iv) Implement PRM algorithm with given distance and number of sampling

Below is the output of PRM with both 200 and 2500 samples. Note here the blue portion is the PRM Graph while the yellow is the free space of the graph.
For 2500 sampled nodes it adds edges in range of 85000 to 1,00,000.

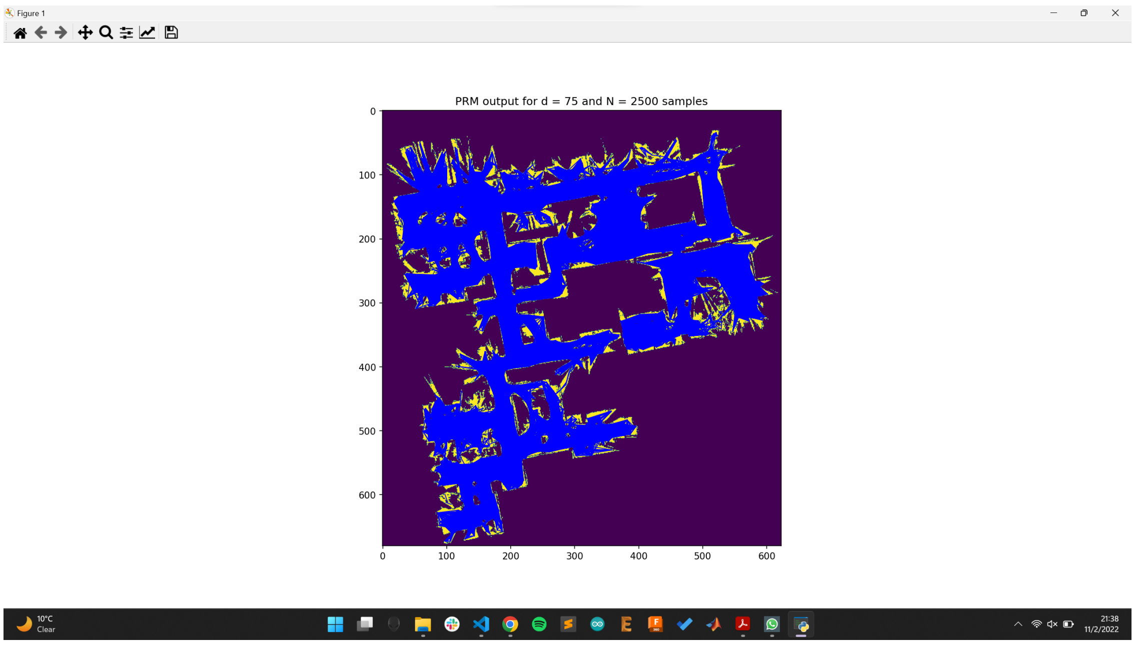


Fig 12: PRM algorithm with 200 nodes and 836 edges.

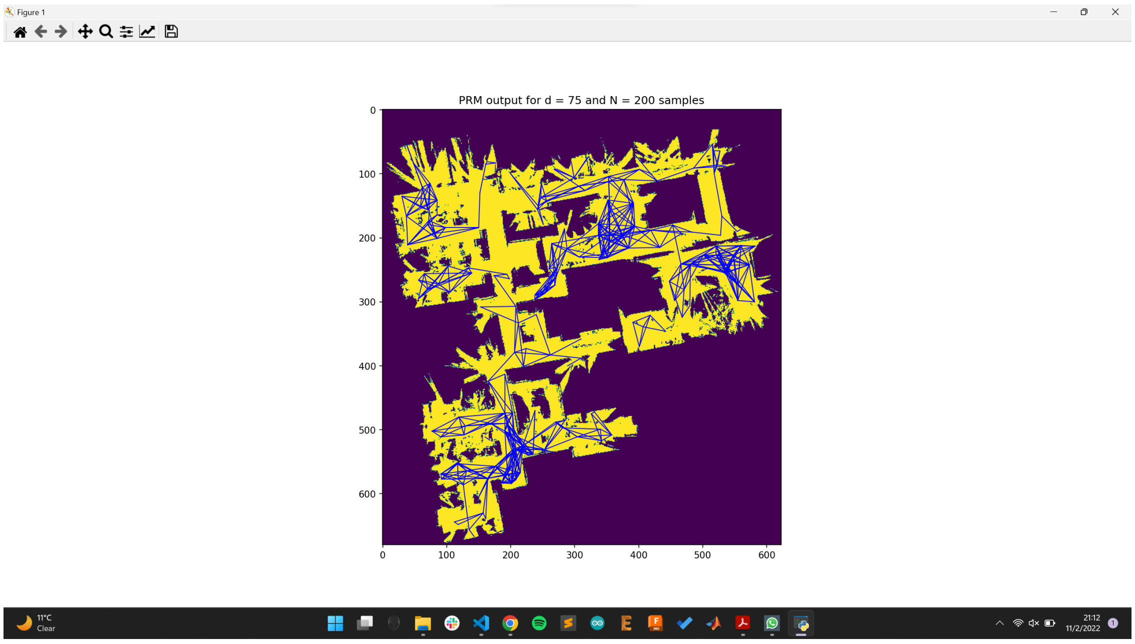


Fig 13: PRM algorithm with 200 nodes and 836 edges.

### 2.3.5 (v) Implement A* Search based on the Network graph from PRM Output

The below is the snippet showing the output of the PRM graph and implementing the A* Search on top of that networked graph.

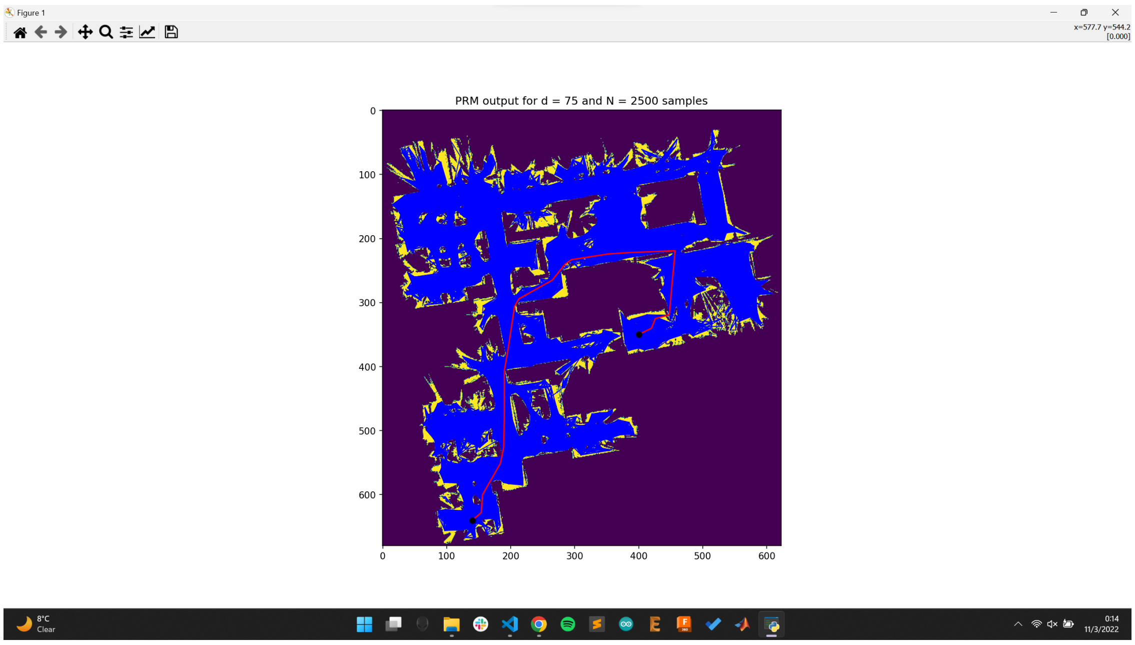


Fig 14: A* and PRM with 2500 nodes.

The difference in number of nodes sampled drastically changes the path length and path cost but significantly increases the computational time. Here the overall cost of the path is 20 Nodes for 2500 samples and 30 Nodes for 400 samples. This cost is less if the number of samples are more. Cost of Path = 790.
Here **cost of path = Number of Nodes * Distance;** $(0 < distance < 75)$
400 Nodes sampling is the optimum minimum for the given start and goal positions. Notice in the below figure how the path changes when number of nodes changes.

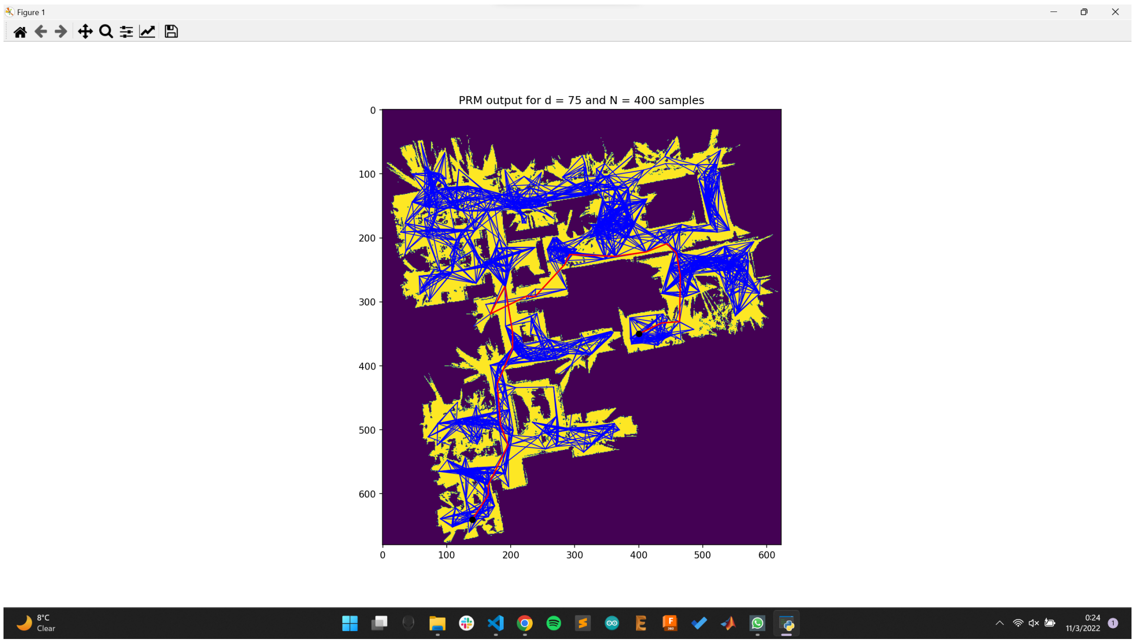


Fig 15: A* and PRM with 400 nodes.