Mobile Robotics
HW 2

Q.2)

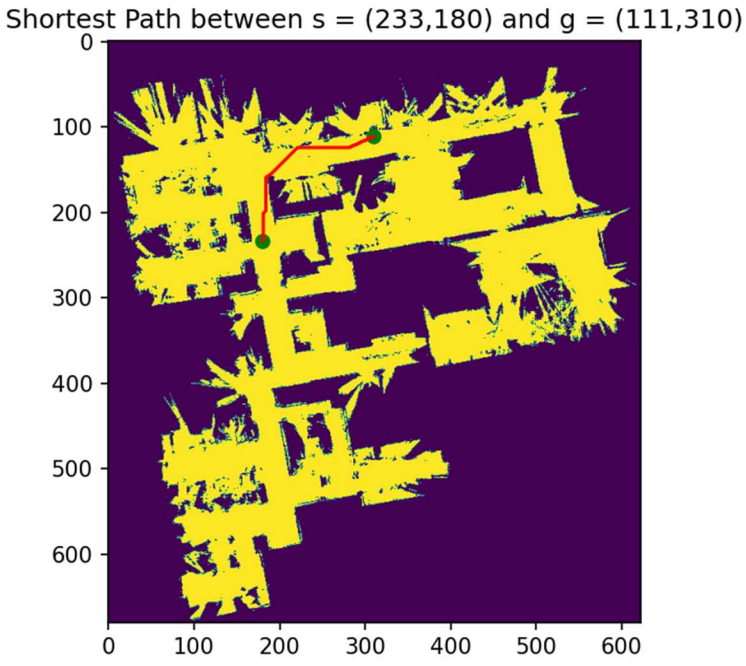
a) i. RecoverPath

```
def RecoverPath(self,pred,s,g):
    # Define empty list
    L1 = []

    # Append the goal vertex to the list
    L1.append(g)
    val = g

    # Loop for back tracking the Path from G to S using pred
    while(pred[val] != s):
        L1.append(pred[val])
        val = pred[val]

    # Append the start vertex to the list
    L1.append(s)

    # Reversing the list to get path from s to g
    L1.reverse()
    return L1
```

```
Path returned by RecoverPath function :
[(233, 180), (232, 181), (231, 181), (230, 181), (229, 181), (228, 181), (227, 181), (226, 181), (225, 181), (224, 181), (223, 181), (222, 181), (221, 181), (220, 181),
 (219, 181), (218, 181), (217, 181), (216, 181), (215, 181), (214, 181), (213, 181), (212, 181), (211, 181), (210, 181), (209, 181), (208, 181), (207, 181), (206, 181),
 (205, 181), (204, 181), (203, 181), (202, 181), (201, 181), (200, 182), (199, 183), (198, 184), (197, 184), (196, 184), (195, 184), (194, 184), (193, 184), (192, 184),
 (191, 184), (190, 184), (189, 184), (188, 184), (187, 184), (186, 184), (185, 184), (184, 184), (183, 184), (182, 184), (181, 184), (180, 184), (179, 184), (178, 184),
 (177, 184), (176, 184), (175, 184), (174, 184), (173, 184), (172, 184), (171, 184), (170, 184), (169, 184), (168, 184), (167, 184), (166, 184), (165, 184), (164, 184),
 (163, 184), (162, 184), (161, 184), (160, 184), (159, 184), (158, 185), (158, 186), (157, 187), (156, 188), (156, 189), (155, 190), (154, 191), (153, 192), (152, 193),
 (151, 194), (150, 195), (149, 196), (148, 197), (147, 198), (146, 199), (145, 200), (144, 201), (143, 202), (142, 203), (141, 204), (140, 205), (139, 206), (138, 207),
 (137, 208), (136, 209), (135, 210), (134, 211), (133, 212), (132, 213), (131, 214), (130, 215), (129, 216), (128, 217), (127, 218), (126, 219), (125, 220), (124, 221),
 (124, 222), (124, 223), (124, 224), (124, 225), (124, 226), (124, 227), (124, 228), (124, 229), (124, 230), (124, 231), (124, 232), (124, 233), (124, 234), (124, 235),
 (124, 236), (124, 237), (124, 238), (124, 239), (124, 240), (124, 241), (124, 242), (124, 243), (124, 244), (124, 245), (124, 246), (124, 247), (124, 248), (124, 249),
 (124, 250), (124, 251), (124, 252), (124, 253), (124, 254), (124, 255), (124, 256), (124, 257), (124, 258), (124, 259), (124, 260), (124, 261), (124, 262), (124, 263),
 (124, 264), (124, 265), (124, 266), (124, 267), (124, 268), (124, 269), (124, 270), (124, 271), (124, 272), (124, 273), (124, 274), (124, 275), (124, 276), (124, 277),
 (124, 278), (124, 279), (124, 280), (124, 281), (124, 282), (123, 283), (123, 284), (122, 285), (122, 286), (121, 287), (121, 288), (121, 289), (120, 290), (120, 291),
 (119, 292), (119, 293), (118, 294), (118, 295), (117, 296), (117, 297), (116, 298), (116, 299), (116, 300), (115, 301), (115, 302), (114, 303), (114, 304), (113, 305),
 (113, 306), (112, 307), (112, 308), (112, 309), (111, 310)]
```

ii. A_STAR_SEARCH

```python
def A_STAR_SEARCH(self,V,s,g,N,w,h):
    null_path = []

    # Defining empty dictionaries for storing
    CostTo = {}
    EstTotalCost = {}
    pred ={}

    # Defining priority Queue in the form of a dictionary
    Q = {}

    # Defining cost of every vertex in the Vertex set (V) as INF
    for vertex in V:
        CostTo[vertex] = math.inf
        EstTotalCost[vertex] = math.inf

    # Defining Cost to Start from Start is 0
    CostTo[s] = 0

    # Defining Total cost to goal as the heuristic distance
    EstTotalCost[s] = h(s,g)

    # Entering the Start Vertex as first element in the list
    Q[s] = h(s,g)

    while(Q):
        v = list(Q.keys())[0]                                   # Storing the smallest element in Q as v
        Q.pop(list(Q.keys())[0])                                # Poping the smallest element in the Q
        if v == g:
            return self.RecoverPath(pred,s,g)                   # Returning o/p of RecoverPath if v is equal to g
        for i in N(v):                                          # Traverse every neighbour of the the current vertex v
            pvi = CostTo[v] + w(v,i)                            # Updating distance of neighbour from start

            if pvi < CostTo[i]:                                 # If path to i is better than previous known path to i
                pred[i] = v
                CostTo[i] = pvi                                 # Update cost of best path to i
                EstTotalCost[i] = pvi + h(i,g)
                Q[i] = EstTotalCost[i]                          # Update Q's priority or add element to Q
                Q = dict(sorted(Q.items(), key=lambda item: item[1]))   # Sort the dictionary according to values

    #return null path if no path exists
    return null_path
```



Path Length: 221.53910524340097

b)

i) N

```
def N(self,v):
    # Define 8 neighbours
    n1 = (v[0]+1,v[1])
    n2 = (v[0]+1,v[1]+1)
    n3 = (v[0],v[1]+1)
    n4 = (v[0]-1,v[1]+1)
    n5 = (v[0]-1,v[1])
    n6 = (v[0]-1,v[1]-1)
    n7 = (v[0],v[1]-1)
    n8 = (v[0]+1,v[1]-1)

    L = [n1,n2,n3,n4,n5,n6,n7,n8]
    L1 = []
    for x in L:
        if(x[0]>=0 or x[1]>=0 or x[0]<680 or x[1]<623):     # Appending the vertex if the vertex is not out of bounds
            if(self.occupancy_grid[x[0]][x[1]] == 255):     # Appending the vertex if the vertex is not occupied
                L1.append(x)                                # Appending the vertex to the list
    return L1
```

```
Output of function N(v)
[(400, 301), (399, 301), (399, 300), (399, 299), (400, 299), (401, 299)]
```

ii) d

```
def d(self,v1,v2):
    return math.dist(v1,v2)
```

```
Output of function d(v1,v2):
180.27756377319946
```

iii) w and h

```
def w(self,v1,v2):
    return self.d(v1,v2)

def h(self,v1,v2):
    return self.d(v1,v2)
```

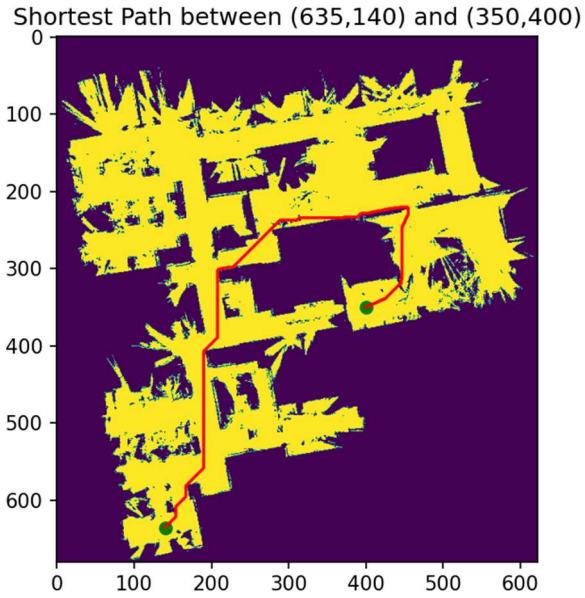
```
Output of function w(v1,v2):
180.27756377319946
```

```
Output of function h(v1,v2):
180.27756377319946
```

iv) A_STAR_SEARCH for given Start and Goal

A Star Search for s = (635, 140) to g = (350,400)

```python
def A_STAR_SEARCH(self,V,s,g,N,w,h):
    null_path = []

    # Defining empty dictionaries for storing
    CostTo = {}
    EstTotalCost = {}
    pred ={}

    # Defining priority Queue in the form of a dictionary
    Q = {}

    # Defining cost of every vertex in the Vertex set (V) as INF
    for vertex in V:
        CostTo[vertex] = math.inf
        EstTotalCost[vertex] = math.inf

    # Defining Cost to Start from Start is 0
    CostTo[s] = 0

    # Defining Total cost to goal as the heuristic distance
    EstTotalCost[s] = h(s,g)

    # Entering the Start Vertex as first element in the list
    Q[s] = h(s,g)

    while(Q):
        v = list(Q.keys())[0]                                   # Storing the smallest element in Q as v
        Q.pop(list(Q.keys())[0])                                # Poping the smallest element in the Q
        if v == g:
            return self.RecoverPath(pred,s,g)                   # Returning o/p of RecoverPath if v is equal to g
        for i in N(v):                                          # Traverse every neighbour of the the current vertex v
            pvi = CostTo[v] + w(v,i)                            # Updating distance of neighbour from start

            if pvi < CostTo[i]:                                 # If path to i is better than previous known path to i
                pred[i] = v
                CostTo[i] = pvi                                 # Update cost of best path to i
                EstTotalCost[i] = pvi + h(i,g)
                Q[i] = EstTotalCost[i]                          # Update Q's priority or add element to Q
                Q = dict(sorted(Q.items(), key=lambda item: item[1]))   # Sort the dictionary according to values

    #return null path if no path exists
    return null_path
```



Path Length: 803.1147904132627

b. PRM

i. sample_new_point

```python
def sample_new_point(self):
    while(True):
        # Sample new point
        new_point = (rn.randint(0,self.occupancy_grid.shape[0]-1), rn.randint(0,self.occupancy_grid.shape[1]-1))

        # Return newly sampled point if it lies in unoccupied space and is not already present as a node
        if self.occupancy_grid [new_point[0]] [new_point[1]] == 255 and new_point not in self.G.nodes():
            return new_point
```

```
Output of sample_new_point() : (215, 336)
```

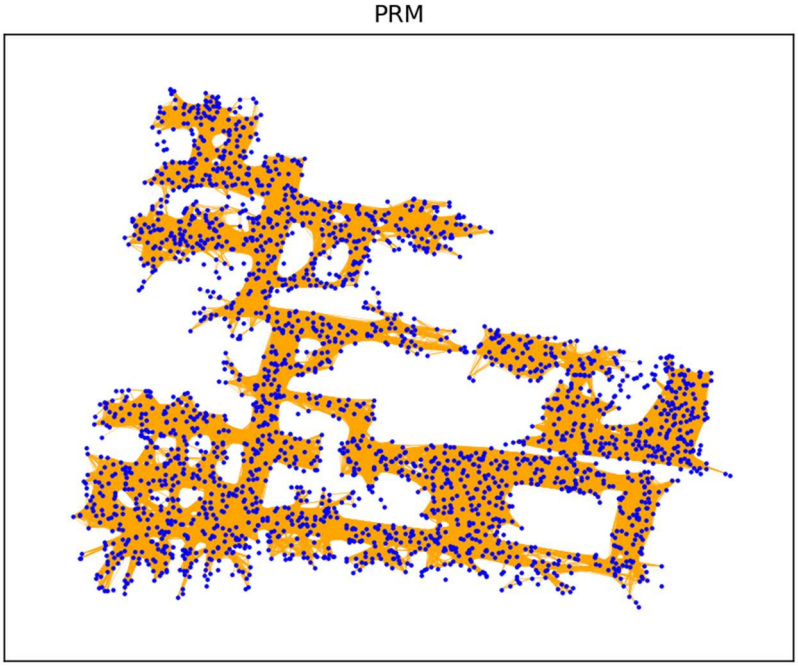
ii. Reachability_check

```
def reachability_check(self,v1,v2):
    x1, y1 = v1[0],v1[1]
    x2, y2 = v2[0],v2[1]

    # If Slope is ND
    if x1 == x2:
        if y2>y1:
            for y in range(y1,y2+1):
                x = 0
                if (self.occupancy_grid[x][y] == 0):
                    return 0

        elif y2<y1:
            for y in range(y2,y1+1):
                x = 0
                if (self.occupancy_grid[x][y] == 0):
                    return 0
    else:
        slope = (y2 - y1)/(x2 - x1)

        # If line is tilted towards y-axis
        if slope>1 or slope<-1:
            if y2 > y1:
                for y in range(y1,y2+1):
                    x = ((y - y1)/slope) + x1
                    if (self.occupancy_grid[int(x)][y] == 0):
                        return 0
            else:
                for y in range(y2,y1+1):
                    x = ((y - y1)/slope) + x1
                    if (self.occupancy_grid[int(x)][y] == 0):
                        return 0

        # If line is tilted towards x-axis
        else:
            if x2 > x1:
                for x in range(x1,x2+1):
                    y = (slope * (x - x1)) + y1
                    if (self.occupancy_grid[x][int(y)] == 0):
                        return 0
            else:
                for x in range(x2,x1+1):
                    y = (slope * (x - x1)) + y1
                    if (self.occupancy_grid[x][int(y)] == 0):
                        return 0
    return 1
```

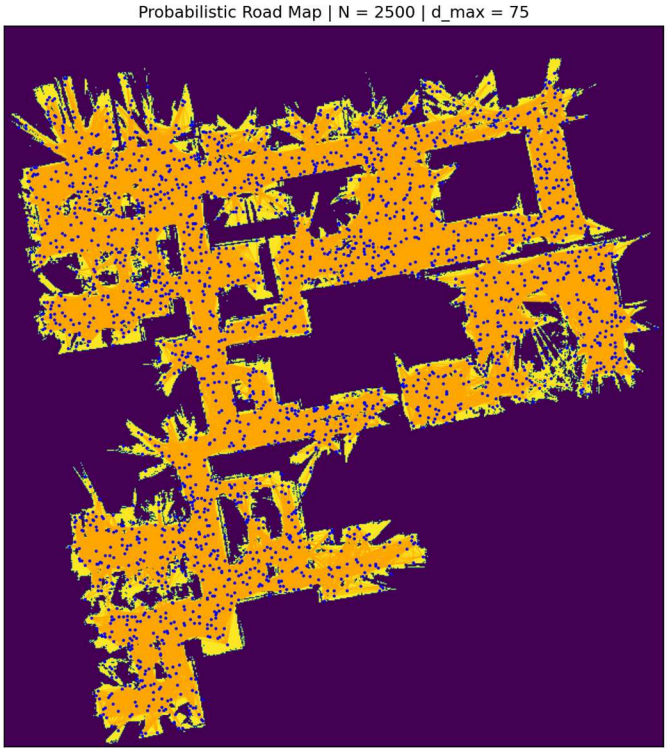
Output is Boolean (0 or 1)

iii. Construct PRM

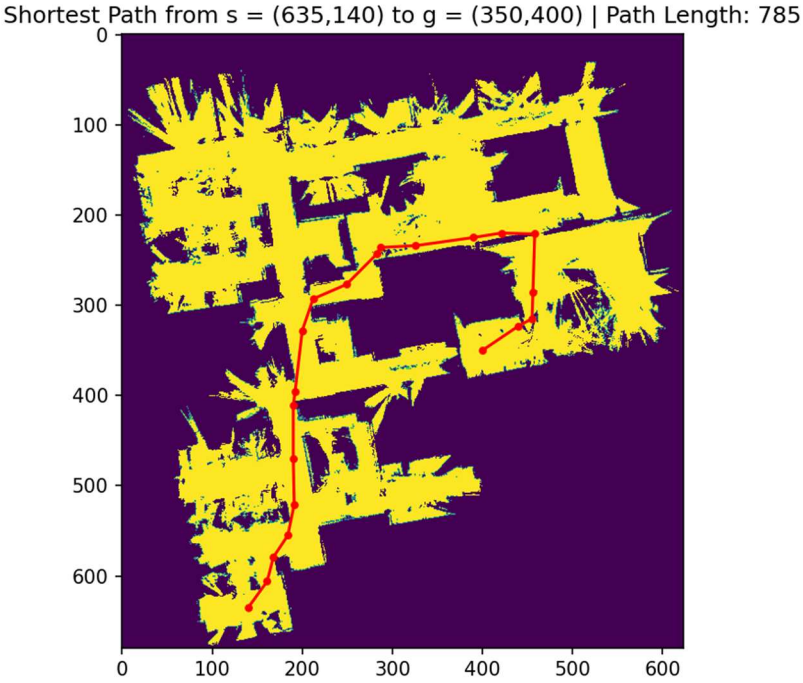
```python
def add_vertex(self,v_new,d_max):
    self.G.add_node(v_new)                                  # Add new node to G
    for v in self.G.nodes():
        if v!=v_new and self.d(v,v_new) <= d_max:
            if(self.reachability_check(v,v_new)):           # Check if node is reachable from v
                self.G.add_edge(v,v_new,weight = self.d(v,v_new))   # Add edge

def construct_prm(self,N,d_max):

    while(self.G.number_of_nodes() < N):
        v_new = self.sample_new_point()
        self.add_vertex(v_new,d_max)

    return self.G
```



iv. PRM



v. Shortest Path using A star



Path Length: 785.7263498231738