Anush Sriram Ramesh
Mobile Robotics EECE 5550 Sec 01
HW 3
TSP and Graph search

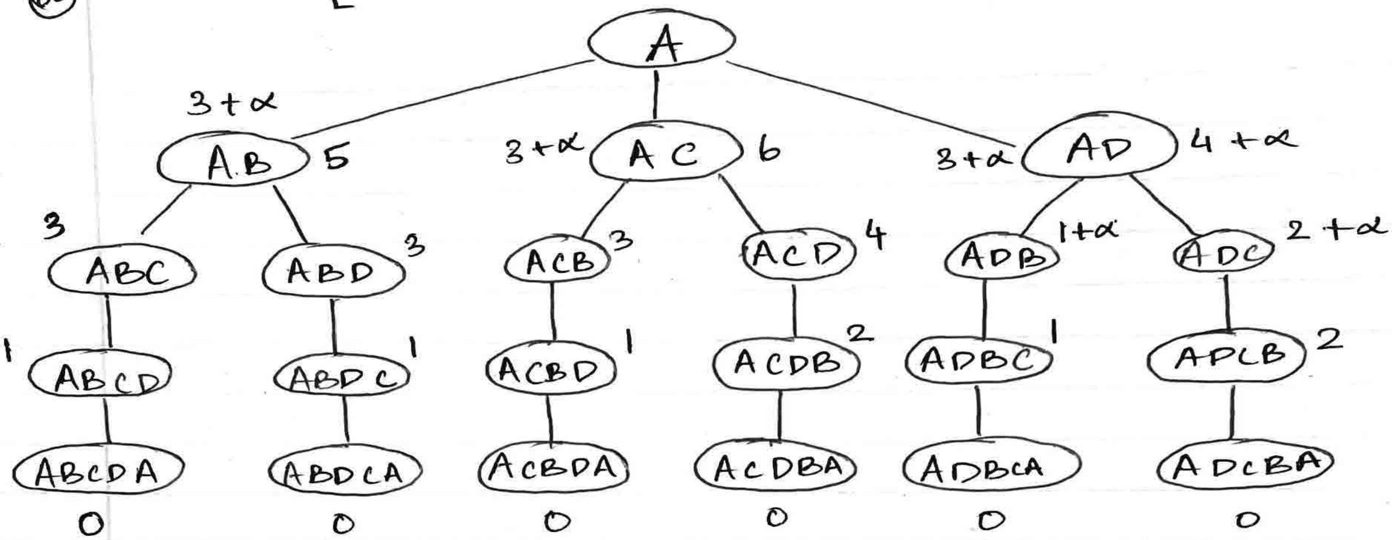
https://github.com/sriram-0311/MR-HW3.git - Github

Question 1.

1.i)

Cost to go of each state –

(a) $[5+\alpha, 7, 4+\alpha, 7, 4+\alpha, 5+\alpha]$



Possible Paths for cycling paths.

ABCDA → $5+\alpha$
ABDCA → 7
ACBDA → $4+\alpha$
ACDBA → 7
ADBCA → $4+\alpha$
ADCBA → $5+\alpha$

Fig: Cost-to-go of each state computed using bottom-up approach

1.ii)

(b) For path, ABCDA = $5+\alpha$ and
ADCBA = $5+\alpha$ will never be optimal
for any value of α.

For $\alpha \in [1, 2]$ paths ACBDA and ADBCA
will be optimal

Paths ABDCA, ACDBA will be optimal for $\alpha > 2$

Fig: Ranges for α for which each path is optimal

Question 2. ***Python***

2. A. i)

Recover path function implementation: Function that takes in the pred map (implemented as a dictionary), reconstructs, and returns the Astar path found.

```
#Recover path from start to goal using predecessor dictionary
#start : start point of the path
#goal : goal point of the path
#pred : predecessor dictionary returned from astar search function
def RecoverPath(self, start, goal, pred):
    current = goal
    retracedPath = []
    retracedPath.append(goal)
    totalCost = 0
    #print("pred = ", pred)
    while current in pred:
        #print("current = ", current)
        prevPath = pred[current]
        retracedPath.append(prevPath)
        current = prevPath
        if current == start:
            break
    if current != start:
        print("no path found")

    for i in range(len(retracedPath) - 1):
        totalCost = totalCost + distance(retracedPath[i], retracedPath[i+1])
    return retracedPath, totalCost
```

2.A.ii)

Astar search algorithm implemented as a member function of the class Astar which has the occupancy grid G, Distance function d, weight, and heuristics (w and h) function handles.

```
#Astar search algorithm
#graph : occupancy grid
#start : start point of the path
#goal : goal point of the path
#Function is implemented as a member function of the class AStar
def a_star(self):
    while self.Q != []:
        v = heapq.heappop(self.Q)[1]
        del self.Qhelper[v]
        if v == self.goal:
            print("goal found")
            return self.RecoverPath(self.start, self.goal, self.pred)
        #print("neighbors of ", v, " = ", Neighbours(self.graph, v))
```

```
        for n in Neighbours(self.graph,v):
            #print("n = ", n)
            pvi = self.costTo[v] + distance(v, n)
            if pvi < self.costTo[n]:
                #The path to i through v is better than the previously-known best path to i,
                # so record it as the new best path to i.
                self.pred[n] = v
                self.costTo[n] = pvi
                self.estTotalCost[n] = pvi + distance(n, self.goal)
                if n in self.Qhelper:
                    tmpVal = self.Qhelper[n]
                    self.Q.remove((tmpVal, n ))
                    heapq.heappush(self.Q, (self.estTotalCost[n], n))
                    self.Qhelper[n] = self.estTotalCost[n]
                else:
                    heapq.heappush(self.Q, (self.estTotalCost[n], n))
                    self.Qhelper[n] = self.estTotalCost[n]
    return []
```

2.B.i)

N(v) : Neighbours function implemented as a global function that takes in occupancy grid as graph and the current vertex V and returns all the free neighbors of the vertex.

```
#Neighbour function : takes in occupancy grid and current vertex as input and return
the list of neighbors of #the current vertex as a list of tuples. [(N1),(n2),(n3),...]
#graph : occupancy grid
#v : current vertex
def Neighbours(graph, v):
    neighbors = []
    for x,y in neighbor_map:
        search_point = (v[0]+x,v[1]+y)
        if graph[search_point] == 1:
            neighbors.append(search_point)
    return neighbors
```

2.B.ii)

d(v1, v2) : Distance between two vertices function implemented as global function that takes in two vertices and returns the Euclidean distance between the vertices.

```
#distance(v1,v2) : returns the distance between two vertices v1 and v2
#v1 : vertex 1
#v2 : vertex 2
def distance(v1, v2):
    return math.dist(v1, v2)
```

2.B.iii)

Using the functions implemented above, Astar search is performed between Start (635,140) and Goal (350,400). The path is plotted using matplotlib superimposed on the thresholded occupancy grid map given.

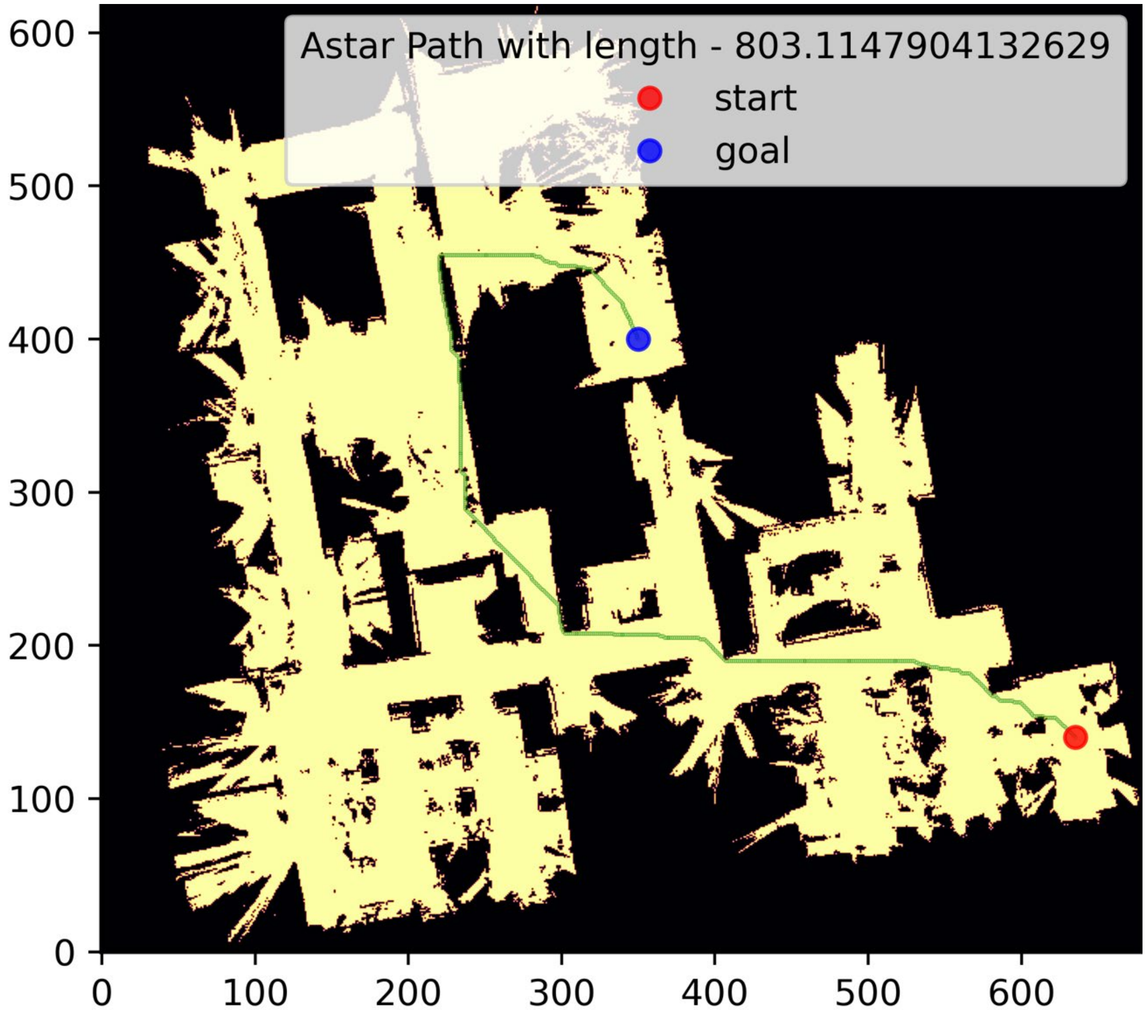


Fig: Astar Path calculated by the above implementation and path plotted with matplotlib. Total length of the path is 803.114

2.C.i)

Rejection Samples function – implemented as a global function that takes in a 2D NumPy array – the occupancy grid and returns one sampled vertex from the free space.

```
#rejectionSamples(occupancyGrid) : returns a uniformly randomly sampled point from
the free space in the occupancy grid
#occupancyGrid : 2D NumPy array of the occupancy grid
#Returns : a tuple of the form (x,y) where x and y are the coordinates of the sampled point
def rejectionSampler(occupancyGrid):
   while True:
      x = int(np.random.uniform(0, occupancyGrid.shape[0]))
      y = int(np.random.uniform(0, occupancyGrid.shape[1]))
      if occupancyGrid[x][y] > 0:
         return (x,y)
      else:
         continue
```

2.C.ii)

Reachability check : implemented as a global function taking in input as occupancy grid, point 1 and point2 and returns True if there is a line of sight between the 2 points, False otherwise.

```
#reachabilityCheck(occupancyGrid, point1, point2) : checks if there is a line of
sight between point1 and point2
#occupancyGrid : 2D NumPy array of the occupancy grid
#point1 : tuple of the form (x,y) where x and y are the coordinates of the first point
#point2 : tuple of the form (x,y) where x and y are the coordinates of the second point
#Returns : True if there is a line of sight between point1 and point2, False otherwise
def reachabilityCheck(occupancyGrid, v1, v2):
    # print("v1: ", v1)
    # print("v2: ", v2)
    current = v1
    while current != v2:
        neighbors = Neighbours(occupancyGrid, current)
        current = min(neighbors, key=lambda x: math.dist(x, v2))
        if occupancyGrid[current] == 0:
            return False
        else:
            continue
    return True
```

2.C.iii)

Build PRM Graph : Implemented as a member function that accepts the number of samples to be generated as input and builds the graph as a member variable of the class PRM.

```
#build_graph() : builds the PRM graph using the rejection sampling algorithm for the
given number of samples and maximum distance
#number_of_samples : number of samples to be generated
#max_distance : maximum distance between two nodes in the graph
#returns : None
#Updates the graph, vertices, and edges attributes of the class PRM
def build_graph(self, n):
    for i in range(n):
        sampleVertex = rejectionSampler(self.occupancy_grid)
        self.addVertex(sampleVertex, i)



#addVertex(vertex, index) : adds a vertex to the graph
#vertex : tuple of the form (x,y) where x and y are the coordinates of the vertex to be added sampled from the free
space
#itr : index of the vertex to be added
#returns : None
def addVertex(self, sampleVertex, itr):
```

```
    self.graph.add_node(itr, pos=sampleVertex)
    if itr > 1:
        for v in self.graph.nodes:
            if (self.graph.nodes[v]['pos'] != sampleVertex) and (math.dist(self.graph.nodes[v]['pos'], sampleVertex) <=
self.maxDistance):
                if (reachablity_check(self.occupancy_grid, self.graph.nodes[v]['pos'], sampleVertex)):
                    self.graph.add_edge(v,itr, weight=math.dist(sampleVertex, self.graph.nodes[v]['pos']))
                    self.vertices.append(sampleVertex)
                else:
                    continue
            else:
                continue
        return
    else:
        return
```

2.C.iv)

PRM Graph generated for 2500 samples generated using rejection sampling. Maximum distance between two nodes is 75.

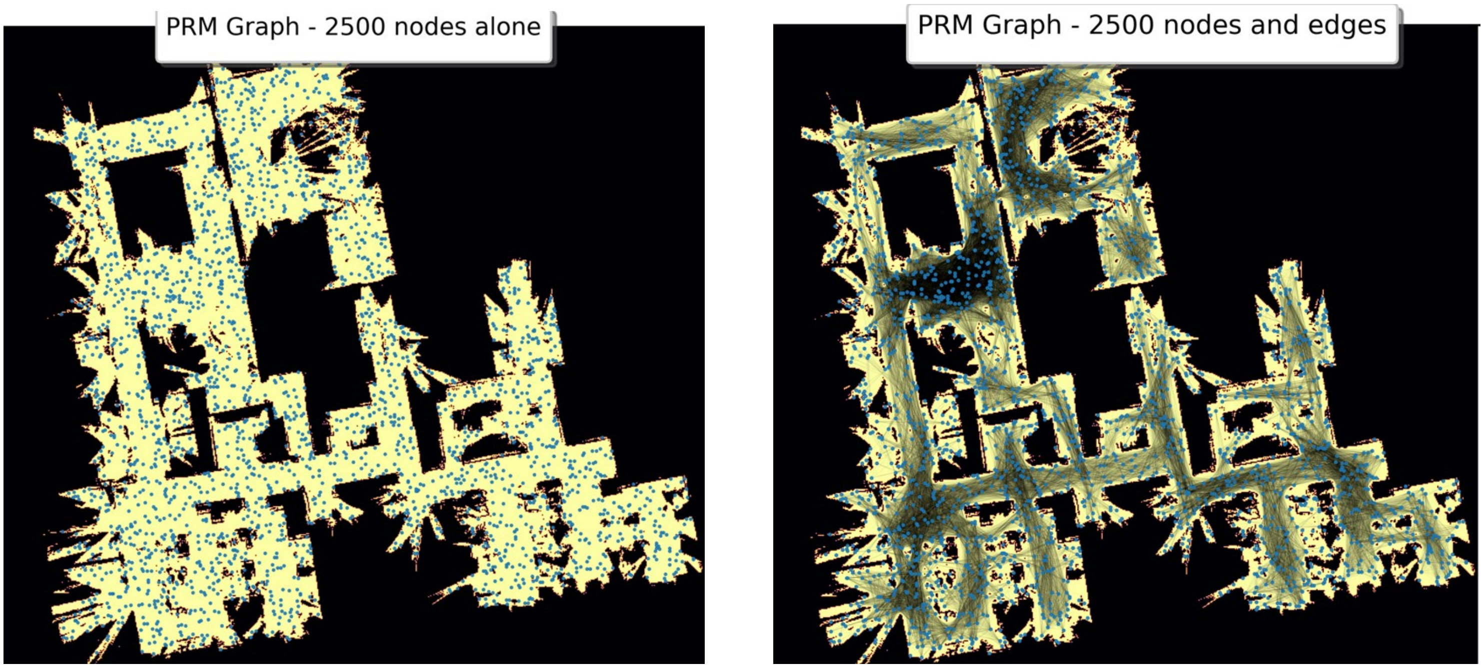


Fig: 2500 nodes sampled from free space in occupancy grid and added to graph. Plotted with Matplotlib.

2.C.v)

Astar search performed on the graph generated above between 2 points, **Start (635,140)** and **Goal (350,400)**. The path is plotted with matplotlib with the edges in the graph removed for better visibility of the path.

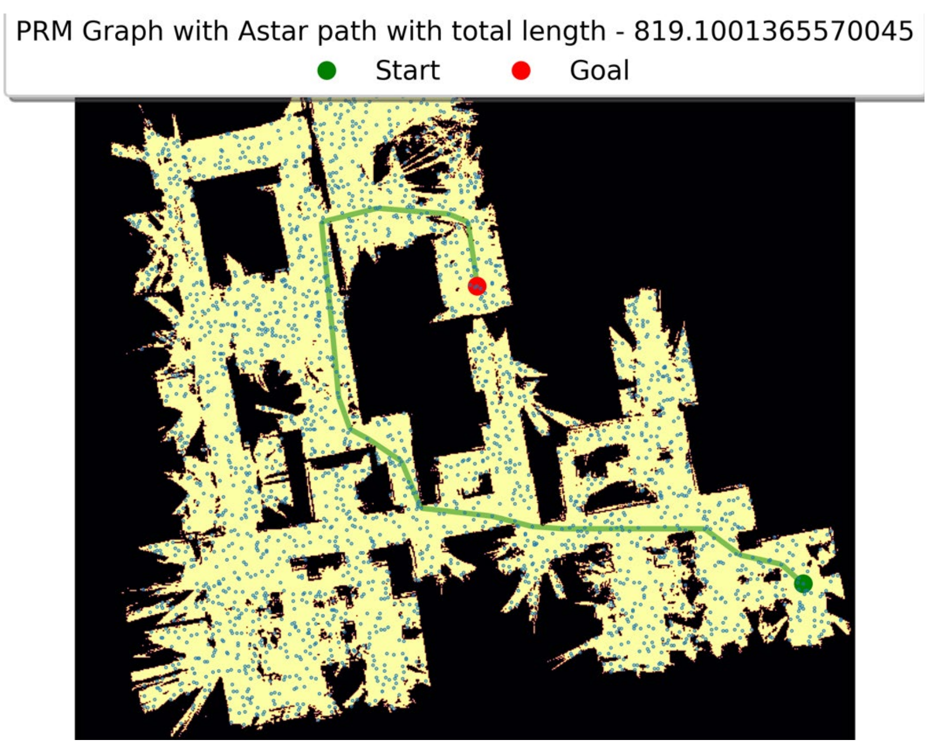


Fig: Astar path found by the networkx's astar_search function call and plotted. Total length of the path is 819.100