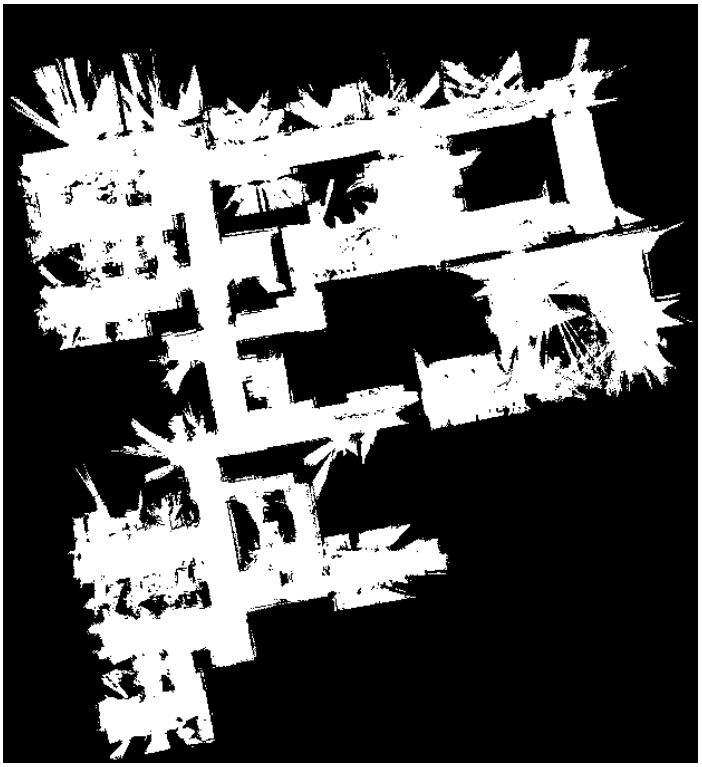
ROUTE PLANNING IN OCCUPANCY GRID MAPS USING A* SEARCH



Occupancy Grid Map

Files associated with this work :

- main.py : Main file
- A_star_search.py
- AuxillaryFunctions.py
- PRM.py
- PRM_Functions.py

*Run the **main.py** file to run the code.*

A* Search:

The code is broken into modules :

- **A_star_search** module contains the A* function itself which takes in a set of vertices, a neighbor dictionary and an occupancy grid matrix as an input.
- **AuxillaryFunctions** module contains the N (neighbor function), w(weight function), RecoverPath function and Pathlength function (which gives the length of the path found by A*). This module is imported by the A* function and also by the PRM modules later.

RecoverPath Function

```
def RecoverPath(s,g,pred): # Takes in the predecessor dictionary
    list = [g]
    v = g   # Setting v as the goal node.
    while v!=s:
        list.insert(0,pred[v])
        v = pred[v]   # Setting v as the predecessor
    return list
```

```
Recovering path ...
Path Recovered : A* successful
[(635, 140), (636, 140), (637, 140), (638, 140), (639, 140), (640, 139)]
Length of path for A* on grid map is :
5.414213562373095
Number of nodes on the path found by A*
6
```

Shown above is a sample demonstration of A* and the recover path function between nodes (635,140) and (640,139).

A* function

```
import AuxillaryFunctions

def A_star(V,neighbor,s,g):
    path = []     # Initializing an empty path
    EstTotalCost = dict()
    CostTo = dict()
    h = dict() # Dictionary to store heuristic
    EstTotalCost[s] = AuxillaryFunctions.w(s,g)   # Cost for start node to reach goal
    pred = dict() # Dictionary to store the predecessors.
    for v in V:
        CostTo[v] = float('inf')
        EstTotalCost[v] = float('inf')
        h[v] = AuxillaryFunctions.w(v,g)
    CostTo[s] = 0  # Cost to s is zero
    EstTotalCost[s] = h[s]    # Setting heuristic for start node as the estimated total cost for the node.
    Q = dict()    # Initializing the queue dictionary
    Q[s] = EstTotalCost[s]
    while len(Q)>0 :
        v = list(Q.keys())[0]    # Getting the vertex with the smallest estimated total cost. The dictionary is sorted by value of est total cost.
        del Q[v]  # Removing that vertex from the dictionary
        if v==g:
            print('Recovering path ... ')
            path = AuxillaryFunctions.RecoverPath(s,g,pred)
            print("Path Recovered : A* successful")
            print(path)
            return path
        for i in neighbor[v]:
            pvi = CostTo[v] + AuxillaryFunctions.w(v,i)  # Represents the total cost to get TO THE NODE i
            if pvi < CostTo[i] :
                pred[i] = v
                CostTo[i] = pvi
                EstTotalCost[i] = pvi + h[i]
                Q[i] = EstTotalCost[i]    # Updating the estimated total cost of the vertex in the dictionary
                dict(sorted(Q.items(), key=lambda item: item[1]))   # Sorting the dictionary.

    return path
```

N – Neighbor function : This function considers 8 neighbors for each node and discards those nodes that lie within obstacles.

```
def N(v,occ_grid):
    neighbor_set = set()

    i = v[0]
    j = v[1]
    for p in range(-1,2): # three rows it will check. The upper row, the lower row and the current row
        #print(p)
        for q in range(-1,2):
            #print(q)
            if p==0 and q==0:
                continue
            elif i+p>=0 and i+p<=occ_grid.shape[0]-1 and j+q>=0 and j+q<=occ_grid.shape[1]-1 :  # Inside the square.
                #print("Inside square IF")
                if occ_grid[i+p,j+q] == 1 :
                    neighbor_set.add((i+p,j+q))
            else : continue
    return neighbor_set
```

```
Output of neighbor function N :
{(236, 454), (235, 454), (234, 456), (236, 456), (234, 455), (236, 455), (235, 456), (234, 454)}
```

Shown above is the output of the neighbor function for the node (235,455).

Weight Function w: This function calculates the Euclidian distance between two nodes which is later used as a weight of the edge.

```
def w(v1,v2):
    return LA.norm([v2[0]-v1[0],v2[1]-v1[1]])
```

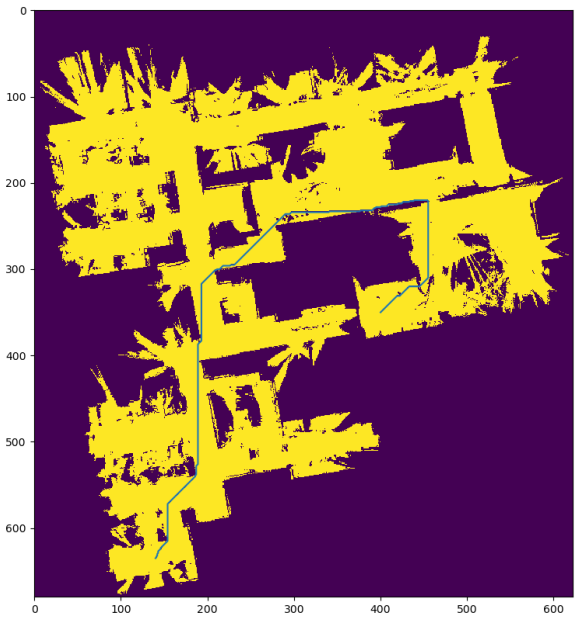
```
Output of the w function for the following nodes
(532, 116)
(485, 270)
161.01242188104618
```

A separate function for the heuristic was not written. Instead, the w function is used in the A* function whenever the heuristic needs to be calculated.

Path length function :

```
def pathlength(path):
    length = 0
    for i in range(0,len(path)-1):
        length = length + w(path[i],path[i+1])
    return length
```

A* search between (635,140) and (350,400) nodes



ROUTE PLANNING WITH PROBABILISTIC ROADMAPS (PRM)

Just like A*, here too the code is broken into modules.

- **PRM** module : Contains the PRM function which takes input as number of sample points, dmax and occupancy grid maps. It returns a set of vertices and a dictionary of neighbors of each vertex.
- **PRM_Functions** module : This contains the local planner (executes the Bresenham algorithm to get pixels between two points), check_feasibility function (which returns a 0 or 1 depending on whether the path by local planner hits an obstacle or no) and the add_vertex function.

Local Planner :

This contains three functions : planLineLow, planLineHigh and planLine. The third one calls the first two based on different scenarios. The local planner returns a list of pixels.

```
def planLine(v0,v1) :
    pixels = []
    if abs(v1[1]-v0[1])<abs(v1[0]-v0[0]):
        if v0[0]>v1[0] :
            pixels = planLineLow(v1,v0)
        else :
            pixels = planLineLow(v0,v1)
    else :
        if v0[1]>v1[1] :
            pixels = planLineHigh(v1,v0)
        else :
            pixels = planLineHigh(v0,v1)
    return pixels
```

```
def planLineLow(v0,v1):           # Takes in tuples of vertices.
    pixels = []
    dx = v1[0]-v0[0]
    dy = v1[1]-v0[1]
    yi = 1
    if dy<0:
        yi = -1
        dy = -dy
    D = (2 * dy) - dx
    y = v0[1]
    for x in range(v0[0],v1[0]+1):
        pixels.append((x,y))  # Adding the tuple to the list
        if D>0 :
            y = y + yi
            D = D + (2*(dy-dx))
        else :
            D = D + 2*dy
    return pixels
```

```
def planLineHigh(v0,v1):          # Takes in tuples of vertices.
    pixels = []
    dx = v1[0]-v0[0]
    dy = v1[1]-v0[1]
    xi = 1
    if dx<0:
        xi = -1
        dx = -dx
    D = (2 * dx) - dy
    x = v0[0]
    for y in range(v0[1],v1[1]+1):
        pixels.append((x,y))  # Adding the tuple to the list
        if D>0 :
            x = x + xi
            D = D + (2*(dx-dy))
        else :
            D = D + 2*dx
    return pixels
```

Reachability check (check_feasibility function):

Returns a flag which is either 0 or 1.

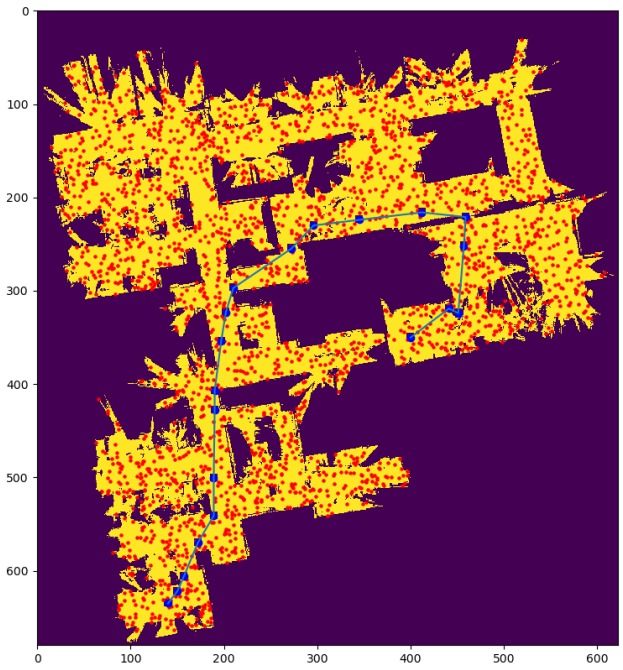
```python
def check_feasibility(v0,v1,occ_grid):
    pixels = planLine(v0,v1)
    flag = 1
    for v in pixels:
        if occ_grid[v[0],v[1]] == 0:
            flag = 0
            break
    return flag
```

Add Vertex function :

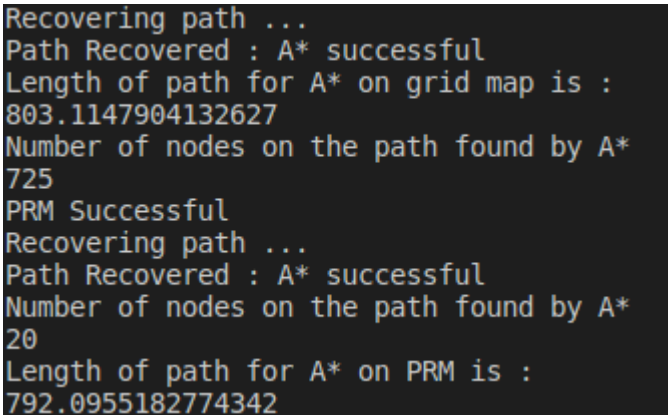
Adds a sampled vertex to the neighbor dictionary is (a) It passes the feasibility test (b) if it lies within dmax. The sampled point is added to the set of vertices if it is in the free region. Addition of a vertex to the set of vertices does not need to satisfy conditions (a) and (b).

```python
def add_vertex(V,neighbor,vnew,dmax,occ_grid):
    flag = 0
    nv = set()     # Set of neighbors for vnew
    for v in V:
        dist = LA.norm([v[0]-vnew[0],v[1]-vnew[1]])

        if dist<=dmax :
            if check_feasibility(v,vnew,occ_grid) == 1:
                nv.add(v)
                if v in neighbor.keys():
                    neighbor_set = neighbor[v]
                    neighbor_set.add(vnew)
                    neighbor[v] = neighbor_set
                else :
                    neighbor[v] = {vnew}       # Creating a new neighbor entry in the dictionary if the vertex didn't exist before in the dictionary.
                if flag==0:
                    flag=1
    V.add(vnew)
    if flag==1:
        neighbor[vnew] = nv

    return V,neighbor,flag      # Returning V, neighbors with the appropriate changes.
```

Final Output of PRM and subsequent A* :



Final output in the console for both the tasks after running the main :



As can be observed, the path length returned by A* search on the PRM is shorter than the path length on the path planned on the occupancy grip map itself.