Student:
Precious I. Philip-Ifabiyi [philipifabiyiprecious@gmail.com]

Lab3: Graph Search - A* algorithm

1 Introduction

Path-finding algorithms are a crucial component in robotics. They enable robots to navigate efficiently through complex environments by finding the shortest or most efficient path between two points. Breadth-first search (BFS), Depth-first search (DFS), Dijkstra's, and A* algorithms are all path-finding algorithms. Not all the algorithms mentioned use heuristics to find a path.
This lab report delves into exploring and implementing the A* algorithm. The algorithm's key components, such as the open list, *f_score* and heuristics, will be explored, providing a comprehensive understanding of its inner workings. A discrete A* algorithm will also be implemented on a grid map.
The report is organized in the following way: The basic concepts and details of the algorithm are explained in the Methodology section (Section 2). Section 3 shows all the results obtained while implementing the A* and discrete A* algorithms in different environments. Finally, the results and problems encountered during implementation are discussed in the Discussion and Conclusions section (Section 4).

2 Methodology

This section of the report will discuss the programming approach, tools, and methods employed to implement the A* and discrete A* algorithms. Section 2.1 explores the A* algorithm, and Section 2.2 explores the discrete A* algorithm.

2.1 A* algorithm

This subsection will explain the implementation of the A* algorithm.

2.1.1 Class: *Vertex*

The *Vertex* class defines a vertex object of an environment. The nodes of the visibility graph are stored as $Vertex$ objects. The attributes of the *Vertex* class include x, y, *point_id*, and *neighbours*. The *neighbours* attribute is a Python list used to store a node's neighbours. The class also contains a method called *dist*, which calculates the Euclidean distance between two Vertex objects (nodes). The implementation can be seen from Lines 12 to 41.

2.1.2 Function: *load_vertices_from_file*

This function is used to read a *.csv* file containing an environment's vertices. It returns a list of vertices ($Vertex$ objects). The implementation can be seen from Lines 67 to 78.

2.1.3 Function: *load_edges_from_file*

This function is used to read a *.csv* file containing an environment's visibility graph. It returns a list of edges obtained from the visibility graph. The implementation can be seen from Lines 81 to 88.

2.1.4 Function: *A_star*

This function implements the A* search algorithm. It takes the start node, goal node, and heuristics as inputs and returns the path from the start node to the goal node with minimum cost.
In A* algorithm, *g_score* of a node is the path cost of reaching a node from the start node, *h_score* is the heuristic (estimated) cost from a node to the goal node, and *f_score* is the sum of *g_score* and *h_score*.
At the beginning of the algorithm, a list called *openList* is initialized with the start node in it. The *openList* stores the nodes that have been reached but not explored/expanded. Path reconstruction and cost tracking dictionaries are also initialized ($camefrom$, *g_score*, *f_score*). The dictionary $cameFrom$ keeps track of the parent node for each explored node in the graph.
The A* algorithm iteratively selects the node with the lowest *f_score* from *openList*, expands it, and updates the scores of the node's neighbours if a better path to the neighbour is found. This process continues until the goal node is found with the smallest estimated total cost (*f_score*) from *openList* or all nodes have been explored (*openList* becomes empty).
If the goal node is found, the path from the start node to the goal node is reconstructed and returned along with the total cost; otherwise, 0 is returned. The implementation can be seen from Lines 99 to 134.

2.1.5 Function: *heuristics*

For this path-planning problem, the Euclidean distance of the node from the goal node is used as the heuristic function. The *heuristics* function is called by the *A_star* function, and it calculates the Euclidean distance of a node/vertex from the goal node. It uses the already implemented *dist* method in the $Vertex$ class for the calculation. The implementation can be seen from Lines 137 to 139.

2.1.6 Function: *reconstruct_path*

Once the goal node is found by the *A_star* function, it calls the *reconstruct_path* function to obtain the path from the start node to the goal node.
The function takes two parameters: $cameFrom$, a dictionary that maps each node to its parent node in the search, and $current$, the node for which the path needs to be reconstructed. It initializes a list called *total_path* with the current node in it. A $while$ loop that iteratively backtracks from the current node to its parent nodes using the information stored in the $cameFrom$ dictionary is used. The loop stops when it reaches the start node. In each iteration, the current node is updated to its parent, and the current node is added to the beginning of the *total_path* list. Finally, the function returns the *total_path* list, which represents the optimal path from the start node to the goal node based on the search information stored in the $cameFrom$ dictionary.
The implementation can be seen from Lines 91 to 97.

2.1.7 Function: *plot*

In this method, the vertices of the graph, edges of the graph, and path generated from the start node to the goal node by the A* algorithm are plotted. The *matplotlib.pyplot.plot* function is utilized for plotting. The implementation can be seen from Lines 44 to 63.

2.2 Discrete A* algorithm

The discrete A* algorithm implements the A* algorithm for grid maps. The grid maps used in this lab exercise are in *.png* format. The *PIL* library is used to load the grid map, which is converted to a *numpy* array. The *numpy* representation of the grid map is binarized so that 0 represents free spaces and 1 represents occupied spaces. After the grid map has been transformed, it is passed alongside its start and goal positions to the *AStar* function.
The implementation of the *AStar* function is similar with the one explained in Section 2.1.4 (Lines 16 to 50). The main difference between the two implementations is the method of obtaining neighbours of a node. This will be explained in the next subsection.

2.2.1 Function: *neighbours*

This function generates the valid neighbouring positions for a given position (x, y) within a 2D grid map. The function takes three parameters: grid (a 2D NumPy array representing the environment), pos (the current position as a tuple of coordinates), and connectivity (an integer specifying either connectivity 4 or connectivity 8 for neighbouring positions).
All the possible neighbouring positions are obtained depending on the connectivity. Connectivity 4 means that the nodes to the left, right, up, and down of a given position are considered neighbours. Connectivity 8 means that the nodes to the left, right, up, down, and diagonals of a given position are considered neighbours. The valid neighbours are obtained by ensuring that none of the possible neighbours are obstacles or outside the grid boundaries.
The implementation can be seen from Lines 56 to 69.

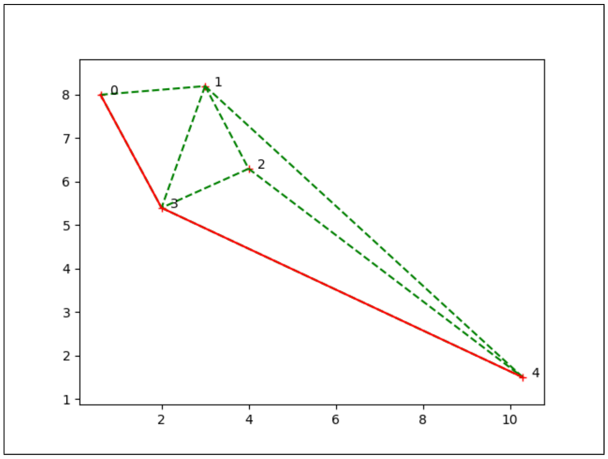
3 Results

3.1 A* Results

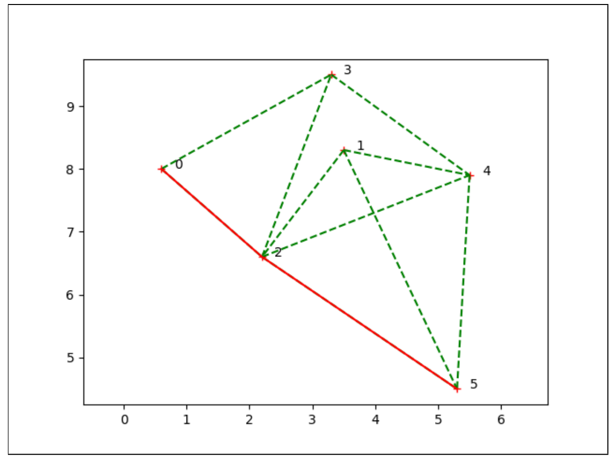
The A* search algorithm was tested in five different environments. Figure 1 shows the environments and paths obtained by applying the A* algorithm. The green dashed line in the sub-figures is the edge of the graph, and the red line is the path with minimum cost to the goal node.

3.2 Discrete A* Results

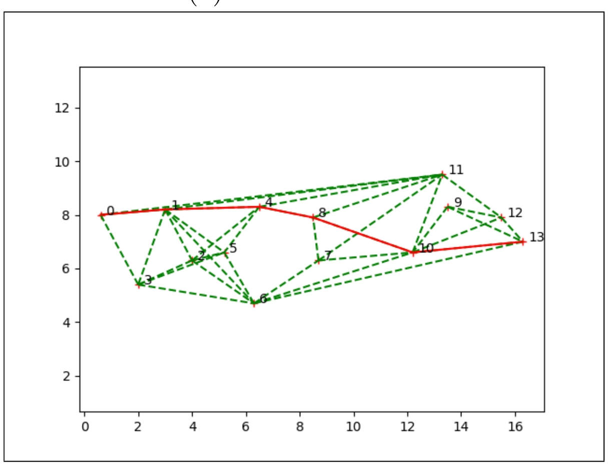
The discrete A* search algorithm was tested in four different environments. Both connectivity 4 and connectivity 8 were used. Figures 2 and 3 show the environments and paths obtained by applying the algorithm. The subplots to the left are paths generated using 4-point connectivity, and those to the right are those generated using the 8-point connectivity. The red line is the path with minimum cost to the goal node.



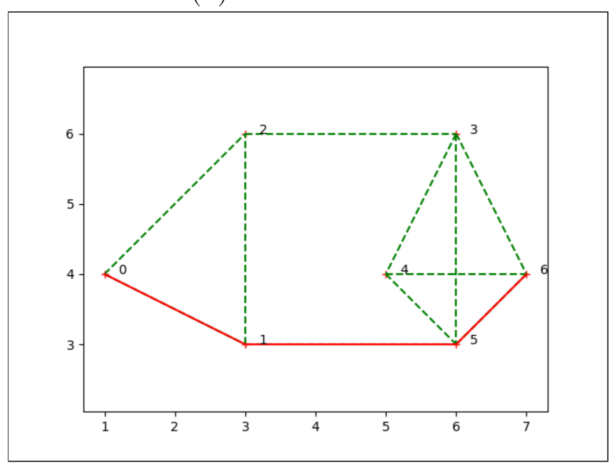
(a) Environment 1



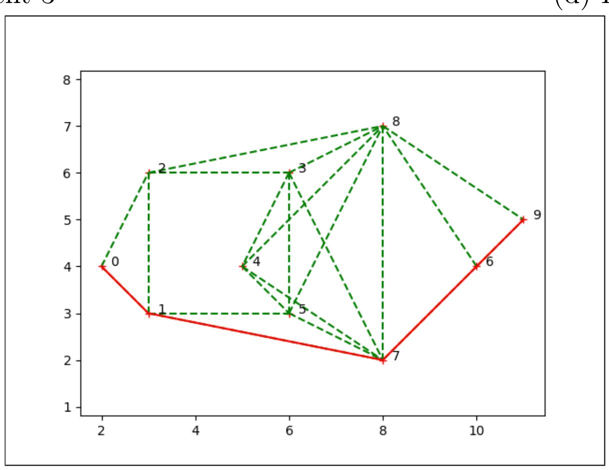
(b) Environment 2



(c) Environment 3

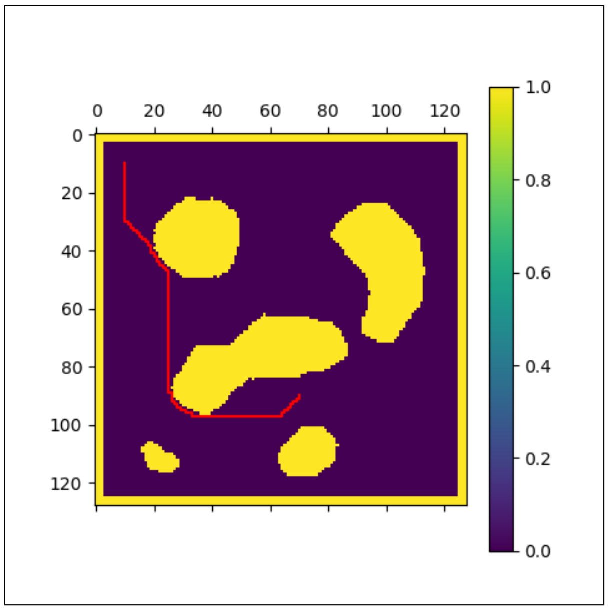


(d) Environment 4

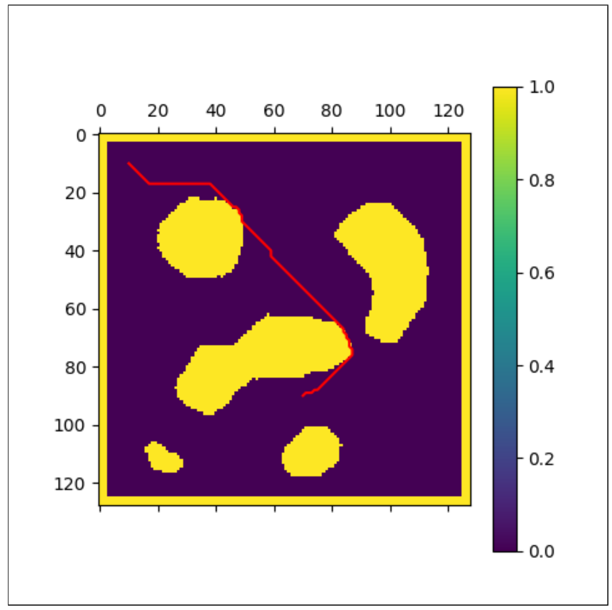


(e) Environment 5

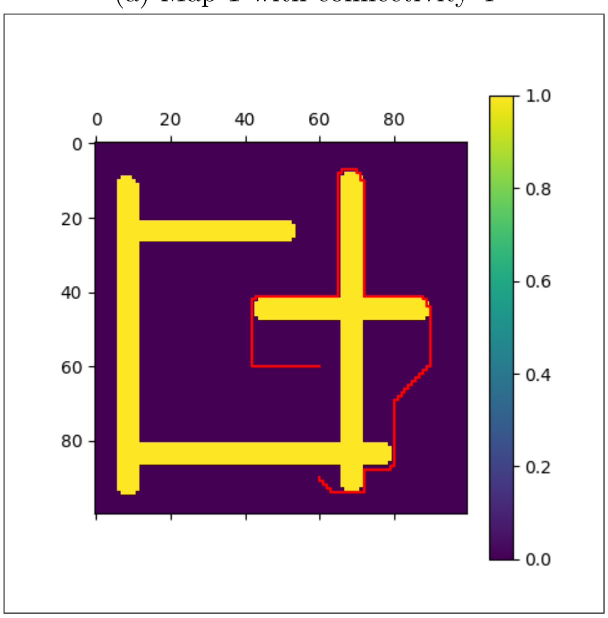
Figure 1: A* search in different environments



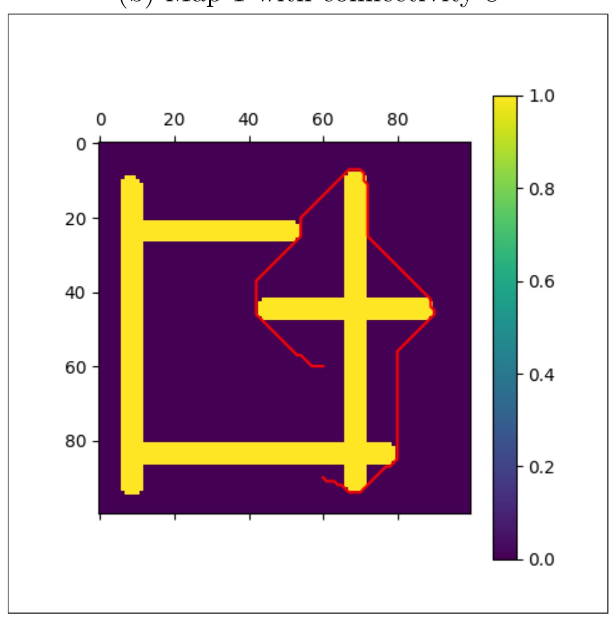
(a) Map 1 with connectivity 4



(b) Map 1 with connectivity 8

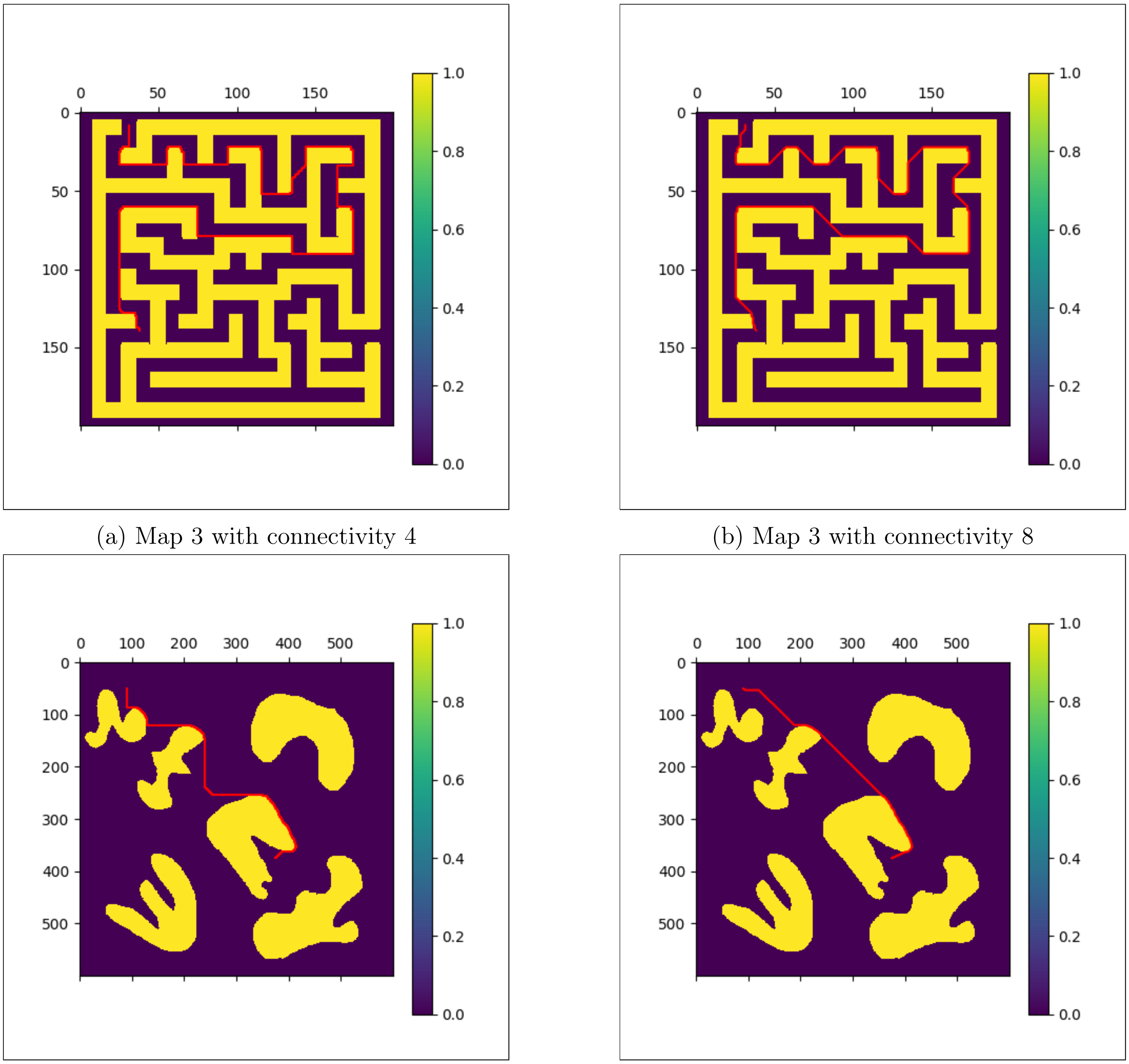


(c) Map 2 with connectivity 4



(d) Map 2 with connectivity 8

Figure 2: Discrete A* search in different environments



(a) Map 3 with connectivity 4

(b) Map 3 with connectivity 8

(c) Map 4 with connectivity 4

(d) Map 4 with connectivity 8

Figure 3: Discrete A* search in different environments

4 Discussion & Conclusions

The results shown above have revealed the robustness and efficiency of the A* algorithm in solving path-finding problems and finding optimum paths. From grid-based environments to maps and graphs, A* consistently demonstrates versatility. Furthermore, we observed that connectivity-8 always produces paths with lower cost than connectivity-4. No problems were faced during this lab exercise.

In conclusion, this lab has provided a comprehensive exploration of the A* algorithm, shedding light on its effectiveness and versatility in solving path-finding problems. By considering both the cost of traversal and heuristic estimates, A* strikes a balance between optimality and computational efficiency.

5 Appendix

```python
#!/usr/bin/python3
import math
from matplotlib import pyplot as plt
import csv
from matplotlib import pyplot as plt
import numpy as np
import math
from typing import List
import sys


Class Vertex:
    """
    Vertex class is defined by the x and y coordinates.
    """
    # constructor or initializer of vertex class

    def __init__(self, x=0, y=0, id=None):
        self.x = x
        self.y = y
        self.id = id
        self.neighbours = []

    def dist(self, p: "Vertex"):
        """
        Return distance between vertices
         Parameters:
            p: input vertex to calculate distance to.
         Returns:
            Distance to vertex from this vertex object
        """

        return math.sqrt((self.x - p.x)**2 + (self.y - p.y)**2)
```

```

    # method to define print() function of object vertex
    def __str__(self):
        return "({}, {})".format(np.round(self.x, 2), np.round(self.y, 2))

    # method to define print() function of list[] of object vertex
    def __repr__(self):
        return "({}, {})".format(np.round(self.x, 2), np.round(self.y, 2))


def plot(vertices, edges, path=None):
    #Plot the vertices
    for v in vertices:
        plt.plot(v.x, v.y, 'r+')
    #Plot the edges of the visibility graph
    for e in edges:
        plt.plot([vertices[e[0]].x, vertices[e[1]].x],
                 [vertices[e[0]].y, vertices[e[1]].y],
                 "g--")

    #Plot the path obtained from A*
    if path != None:
        points=np.zeros((len(path),2))
        for i in range(len(path)):
            points[i,0]=vertices[path[i]].x
            points[i,1]=vertices[path[i]].y
        plt.plot(points[:,0],points[:,1],'r')
    for i, v in enumerate(vertices):
        plt.text(v.x + 0.2, v.y, str(i))
    plt.axis('equal')



def load_vertices_from_file(filename: str):
    # list of vertices
    vertices: List[Vertex] = []
    current_id = 0
    with open(filename, newline='\n') as csvfile:
        v_data = csv.reader(csvfile, delimiter=",")
        next(v_data)
        for row in v_data:
            vertex = Vertex(float(row[1]), float(row[2]), id=current_id)
            vertices.append(vertex)
            current_id += 1
    return vertices


def load_edges_from_file(filename: str):
    edges = []
    with open(filename, newline='\n') as csvfile:
        reader = csv.reader(csvfile, delimiter=",")
```

```python
        next(reader)
        for row in reader:
            edges.append((int(row[0]), int(row[1])))
    return edges


def reconstruct_path(cameFrom, current):
    #Backtrack and get the path from start node to goal node
    total_path=[current]
    while current in cameFrom.keys():
        current=cameFrom[current]
        total_path=[current]+total_path
    return total_path

def A_Star(start, goal, heuristics):
    # Initialization of data structures
    openList = [start]  # List of nodes to be evaluated
    camefrom = {}       # Dictionary to reconstruct the path
    g_score = {start: 0}  # Dictionary to store the cost of the shortest path from start to each node
    f_score = {start: 0 + heuristics(start, goal)}  # Dictionary to store the estimated total cost from start to goal through each node

    # Main A* loop
    while len(openList) != 0:
        # Sort the open list based on f_score
        sorted_openlist = sorted(openList, key=lambda vertex: f_score[vertex])
        smallest_point = sorted_openlist[0]  # Select the node with the smallest f_score

        # Check if the selected node is the goal
        if smallest_point == goal:
            return reconstruct_path(camefrom, smallest_point.id), f_score[smallest_point]

        openList.remove(smallest_point)  # Remove the selected node from the open list

        # Explore neighbors of the current node
        for neighbour in smallest_point.neighbours:
            g_score_neighbour = math.inf if neighbour not in g_score else g_score[neighbour]
            tent_score = g_score[smallest_point] + smallest_point.dist(neighbour)

            # Check if the tentative score is better than the current score for the neighbor
            if tent_score < g_score_neighbour:
                camefrom[neighbour.id] = smallest_point.id
                g_score[neighbour] = tent_score
                f_score[neighbour] = tent_score + heuristics(neighbour, goal)

                # Add the neighbor to the open list if it's not already present
                if neighbour not in openList:
                    openList.append(neighbour)

    # Return 0 if the goal is not reachable
```

```
    return 0


def heuristics(vert, goal):
    #Return euclidean distance of the vertex from the goal node
    return vert.dist(goal)

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage:\n ${} env.csv visibility_graph_env.csv".format(sys.argv[0]))
    else:
        vertices=load_vertices_from_file(sys.argv[1])
        #print(vertices)

        edges = load_edges_from_file(sys.argv[2])
        vertex_list = []

        #Fill the neighbour attribute for all the vertices
        for p1_index, p2_index in edges:
            p1 = vertices[p1_index]
            p2 = vertices[p2_index]
            p1.neighbours.append(p2)
            p2.neighbours.append(p1)

        #Get path from start to goal using A*
        path, dis=A_Star(vertices[0],vertices[-1],heuristics)
        print("Path: ", path)
        print("Distance: ", dis)

        plot(vertices, edges, path)
        plt.show()
```

Listing 1: A_star.py

```
#!/usr/bin/python3
import math
import numpy as np
from matplotlib import pyplot as plt
from PIL import Image
import sys

def reconstruct_path(cameFrom, current):
    # Backtrack to get path from start node to goal node
    total_path=[current]
    while current in cameFrom.keys():
        current=cameFrom[current]
        total_path=[current]+total_path
    return total_path

def AStar(gridmap, start, goal, connectivity):
    # Initialization of data structures
```

```python
    openList = [start]  # List of nodes to be evaluated
    camefrom = {}       # Dictionary to reconstruct the path
    g_score = {start: 0}  # Dictionary to store the cost of the shortest path from start to each node
    f_score = {start: 0 + heuristics(start, goal)}  # Dictionary to store the estimated total cost from start to goal through each node

    # Main A* loop
    while len(openList) != 0:
        # Sort the open list based on f_score
        sorted_openlist = sorted(openList, key=lambda vertex: f_score[vertex])
        smallest_point = sorted_openlist[0]  # Select the node with the smallest f_score

        # Check if the selected node is the goal
        if smallest_point == goal:
            return reconstruct_path(camefrom, smallest_point), f_score[smallest_point]

        openList.remove(smallest_point)  # Remove the selected node from the open list

        # Explore neighbors of the current node
        for neighbour in neighbours(gridmap, smallest_point, connectivity):
            g_score_neighbour = math.inf if neighbour not in g_score else g_score[neighbour]
            tent_score = g_score[smallest_point] + heuristics(neighbour, smallest_point)

            # Check if the tentative score is better than the current score for the neighbor
            if tent_score < g_score_neighbour:
                camefrom[neighbour] = smallest_point
                g_score[neighbour] = tent_score
                f_score[neighbour] = tent_score + heuristics(neighbour, goal)

                # Add the neighbor to the open list if it's not already present
                if neighbour not in openList:
                    openList.append(neighbour)
    # Return 0 if the goal is not reachable
    return 0

def heuristics(x,y):
    # Euclidean distance
    return math.sqrt((x[0] - y[0])**2 + (x[1] - y[1])**2)

def neighbours(grid, pos, connectivity):
    x, y = pos
    x_max, y_max = grid.shape

    # Determine possible neighboring positions based on connectivity type
    if connectivity == 4:
        possible = [(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y)]
    else:
        possible = [(x, y - 1), (x, y + 1), (x - 1, y), (x + 1, y),
                    (x - 1, y - 1), (x - 1, y + 1), (x + 1, y - 1), (x + 1, y + 1)]

```

```
    # Filter valid neighboring positions that are within the grid boundaries and have a grid value of 0
    valid_neighbours = [(i, j) for i, j in possible if 0 <= i < x_max and 0 <= j < y_max and grid[i, j] == 0]
    return valid_neighbours


if __name__ == "__main__":
    if len(sys.argv) != 6:
        print("Usage:\n ${} path_to_grid_map_image start_x start_y goal_x goal_y".format(sys.argv[0]))
    else:
        # Load grid map
        image = Image.open(sys.argv[1]).convert('L')
        grid_map = np.array(image.getdata()).reshape(image.size[0], image.size[1])/255
        # binarize the image
        grid_map[grid_map > 0.5] = 1
        grid_map[grid_map <= 0.5] = 0
        # Invert colors to make 0 -> free and 1 -> occupied
        grid_map = (grid_map * -1) + 1
        # Show grid map

        # A* using 4 point connectivity
        path_4,cost_4=AStar(grid_map,(int(sys.argv[2]),int(sys.argv[3])),(int(sys.argv[4]),int(sys.argv[5])),4)
        print("Path cost for connectivity 4 is: ",cost_4)
        print("Path for connectivity 4: ", path_4)

        plt.matshow(grid_map)
        plt.colorbar()
        if path_4!=0:
            xax=[i for i,_ in path_4]
            yax=[j for _,j in path_4]
            plt.plot(yax,xax, c='r')

        # A* using 8 point connectivity
        path_8,cost_8=AStar(grid_map,(int(sys.argv[2]),int(sys.argv[3])),(int(sys.argv[4]),int(sys.argv[5])),8)
        print("\n\nPath cost for connectivity 8 is: ",cost_8)
        print("Path for connectivity 8: ", path_8)

        plt.matshow(grid_map)
        plt.colorbar()
        if path_8!=0:
            xax=[i for i,_ in path_8]
            yax=[j for _,j in path_8]
            plt.plot(yax,xax, c='r')

        plt.show()
```

Listing 2: A_star_discrete.py