Students:
Fatima Yousif Rustamani [fyousif30@gmail.com]
Syma Afsha [symaafsha.eece@gmail.com]

Lab 3 : Graph Search – A* algorithm

Introduction

The A* algorithm is a method for determining the shortest or most efficient path between nodes in a graph. It begins with the first node and checks the nodes surrounding it, selecting those that appear to be the least expensive to achieve the final goal. The fact that it makes use of heuristics and path cost information when searching classifies it as an informed search algorithm. This paper will guide you through our process of developing the Python A* algorithm, which identifies the best path within a visibility graph.

Additional Functions

Heuristic Function

The heuristic function directs the search by determining how close a node is to the goal point. It calculates the least cost from any vertex n to the goal to predict which neighboring node will most likely lead to the target. As a heuristic function we calculated between the given node and the goal using the **euclidean distance** shown in Figure 1.

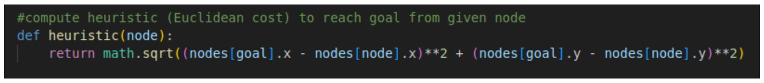
$$euclideandistance = \sqrt{(goal.x - node.x)2 + (goal.y - node.y)2}$$

Distance Calculation Function

The A* method must constantly calculate the weight of the edge from the current node to its neighbor in order to update the gScore. As a result, a distance calculation function is created to compute the Euclidean distance between any two provided nodes, illustrated in Figure 2.

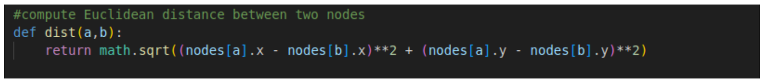
Reconstruct Path Function

When we reach from the start node to the goal node in the A* algorithm, the algorithm ends and displays the optimum path. However, at this stage, we just know the goal node and the node before



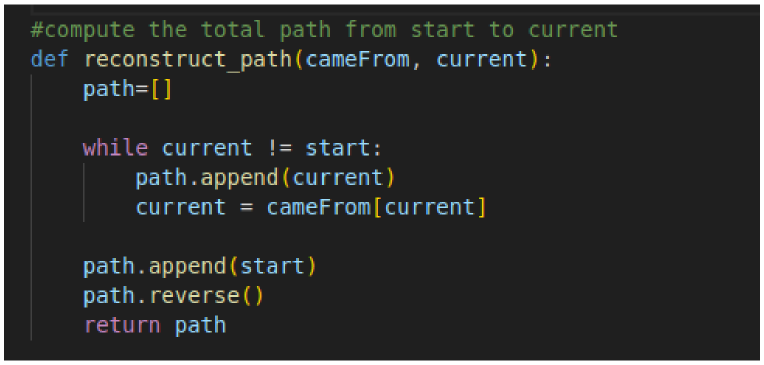
```
#compute heuristic (Euclidean cost) to reach goal from given node
def heuristic(node):
    return math.sqrt((nodes[goal].x - nodes[node].x)**2 + (nodes[goal].y - nodes[node].y)**2)
```

Figure 1: Heuristic Function



```python
#compute Euclidean distance between two nodes
def dist(a,b):
    return math.sqrt((nodes[a].x - nodes[b].x)**2 + (nodes[a].y - nodes[b].y)**2)
```

Figure 2: Distance Calculation Function



```python
#compute the total path from start to current
def reconstruct_path(cameFrom, current):
    path=[]

    while current != start:
        path.append(current)
        current = cameFrom[current]

    path.append(start)
    path.reverse()
    return path
```

Figure 3: Reconstruct Path Function

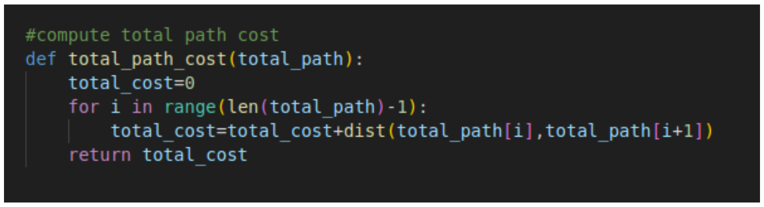
it. We use a special function called reconstructpath(cameFrom, current) to create the entire path from start to the goal. The fundamental idea is that, given the target node, we use the cameFrom dictionary to move up the tree until we reach our start node. We found the shortest path in reverse order by proceeding along the links. So, in the end, we reversed the whole path and returned. The code is demonstrated in Figure 3.

Total Cost Calculation Function

We use a different function to figure out the total cost of the final path, and this function is called from the A* algorithm once it finds the best path. The function code is depicted in Figure 4.

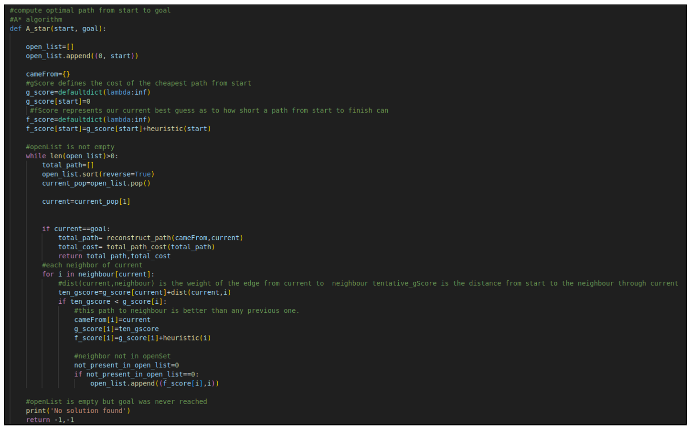
A* Algorithm

The Python code and algorithm for A* graph is shown in Figure 5a and Figure 5b.



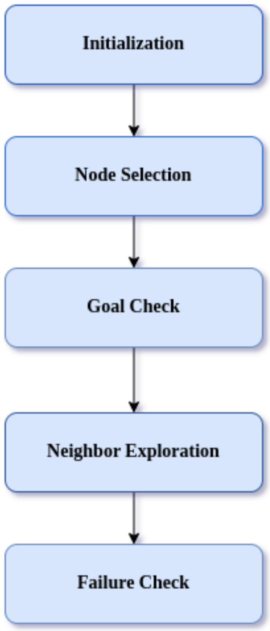
```python
#compute total path cost
def total_path_cost(total_path):
    total_cost=0
    for i in range(len(total_path)-1):
        total_cost=total_cost+dist(total_path[i],total_path[i+1])
    return total_cost
```

Figure 4: Total Cost Calculation Function



```python
#compute optimal path from start to goal
#A* algorithm
def A_star(start, goal):

    open_list=[]
    open_list.append((0, start))

    cameFrom={}
    #gScore defines the cost of the cheapest path from start
    g_score=defaultdict(lambda:inf)
    g_score[start]=0
     #fScore represents our current best guess as to how short a path from start to finish can
    f_score=defaultdict(lambda:inf)
    f_score[start]=g_score[start]+heuristic(start)

    #openList is not empty
    while len(open_list)>0:
        total_path=[]
        open_list.sort(reverse=True)
        current_pop=open_list.pop()

        current=current_pop[1]


        if current==goal:
            total_path= reconstruct_path(cameFrom,current)
            total_cost= total_path_cost(total_path)
            return total_path,total_cost
        #each neighbor of current
        for i in neighbour[current]:
            #dist(current,neighbour) is the weight of the edge from current to  neighbour tentative_gScore is the distance from start to the neighbour through current
            ten_gscore=g_score[current]+dist(current,i)
            if ten_gscore < g_score[i]:
                #this path to neighbour is better than any previous one.
                cameFrom[i]=current
                g_score[i]=ten_gscore
                f_score[i]=g_score[i]+heuristic(i)

                #neighbor not in openSet
                not_present_in_open_list=0
                if not_present_in_open_list==0:
                    open_list.append((f_score[i],i))

    #openList is empty but goal was never reached
    print('No solution found')
    return -1,-1
```

(a) Python code



(b) Main steps of the A* algorithm

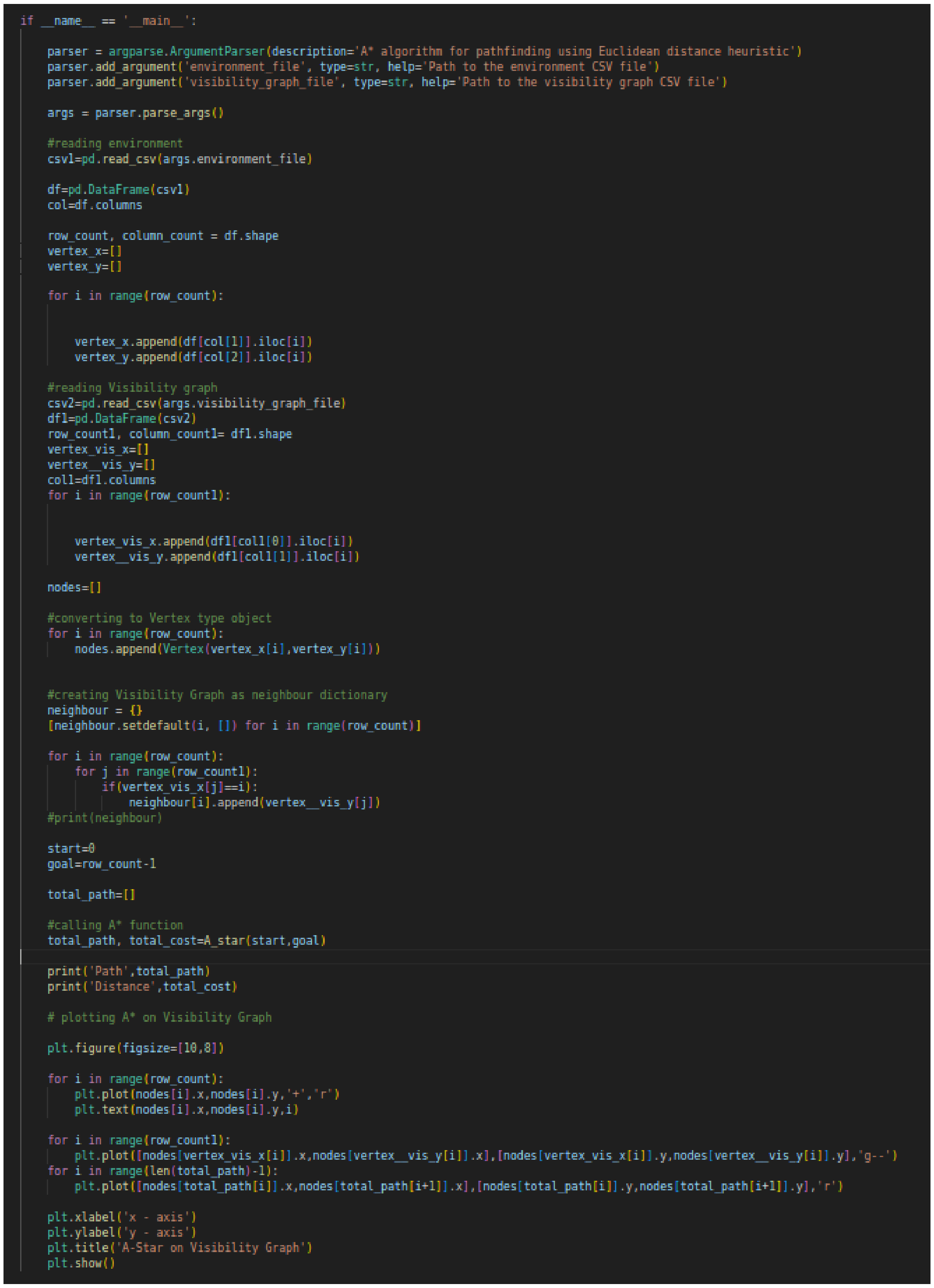
Figure 5: A* Algorithm

Program Execution

Firstly, we loaded the two CSV files that were provided; the first one gave us a complete environment, and the second one included a visibility graph, into the main function. For simplicity of execution, we converted all the vertex of the environment file to point type objects. We then converted all the edges of the second file as a dictionary where keys are nodes and values are their corresponding all neighbours. The points and neighbour dictionary are both used as global variables. We then called the A* algorithm with start and goal node as follows, where the start node is the first object and the goal node is the last object of the environment file.The algorithm returns the optimal path from start to goal and the length of the optimal path. The python code is illustrated in Figure 6.

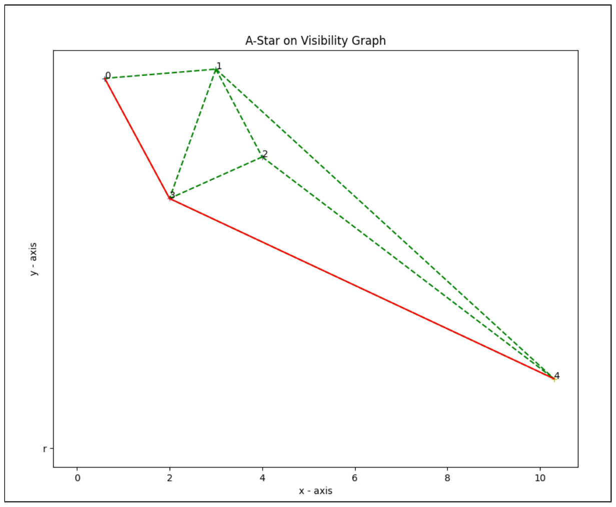
Results and Plotting

Finally, we plotted to visualize the environment and the shortest path. When we tested two csv files, The output for first two csv files, we got the following results: The final path is **[0, 3, 4]**, with a distance of **12.12356982653498**,as shown in Figure 7a, the result for the second two csv files, the final path is **[0, 2, 5]**, distance is **5.870358207916741**,as depicted in Figure 7band the outcome for the third csv files is final path **[0, 1, 4, 8, 10, 13]** and distance is **15.990555296232605**, as illustrated in 7c.

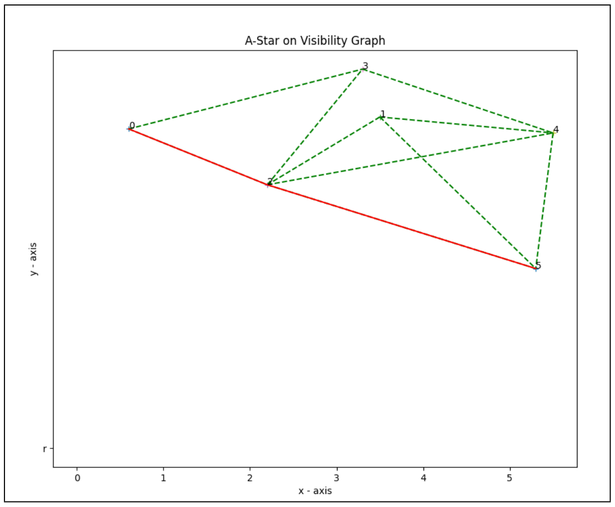


```
if __name__ == '__main__':

    parser = argparse.ArgumentParser(description='A* algorithm for pathfinding using Euclidean distance heuristic')
    parser.add_argument('environment_file', type=str, help='Path to the environment CSV file')
    parser.add_argument('visibility_graph_file', type=str, help='Path to the visibility graph CSV file')

    args = parser.parse_args()

    #reading environment
    csv1=pd.read_csv(args.environment_file)

    df=pd.DataFrame(csv1)
    col=df.columns

    row_count, column_count = df.shape
    vertex_x=[]
    vertex_y=[]

    for i in range(row_count):


        vertex_x.append(df[col[1]].iloc[i])
        vertex_y.append(df[col[2]].iloc[i])

    #reading Visibility graph
    csv2=pd.read_csv(args.visibility_graph_file)
    df1=pd.DataFrame(csv2)
    row_count1, column_count1= df1.shape
    vertex_vis_x=[]
    vertex__vis_y=[]
    coll=df1.columns
    for i in range(row_count1):


        vertex_vis_x.append(df1[coll[0]].iloc[i])
        vertex__vis_y.append(df1[coll[1]].iloc[i])

    nodes=[]

    #converting to Vertex type object
    for i in range(row_count):
        nodes.append(Vertex(vertex_x[i],vertex_y[i]))


    #creating Visibility Graph as neighbour dictionary
    neighbour = {}
    [neighbour.setdefault(i, []) for i in range(row_count)]

    for i in range(row_count):
        for j in range(row_count1):
            if(vertex_vis_x[j]==i):
                neighbour[i].append(vertex__vis_y[j])
    #print(neighbour)

    start=0
    goal=row_count-1

    total_path=[]

    #calling A* function
    total_path, total_cost=A_star(start,goal)

    print('Path',total_path)
    print('Distance',total_cost)

    # plotting A* on Visibility Graph

    plt.figure(figsize=[10,8])

    for i in range(row_count):
        plt.plot(nodes[i].x,nodes[i].y,'+','r')
        plt.text(nodes[i].x,nodes[i].y,i)

    for i in range(row_count1):
        plt.plot([nodes[vertex_vis_x[i]].x,nodes[vertex__vis_y[i]].x],[nodes[vertex_vis_x[i]].y,nodes[vertex__vis_y[i]].y],'g--')
    for i in range(len(total_path)-1):
        plt.plot([nodes[total_path[i]].x,nodes[total_path[i+1]].x],[nodes[total_path[i]].y,nodes[total_path[i+1]].y],'r')

    plt.xlabel('x - axis')
    plt.ylabel('y - axis')
    plt.title('A-Star on Visibility Graph')
    plt.show()
```

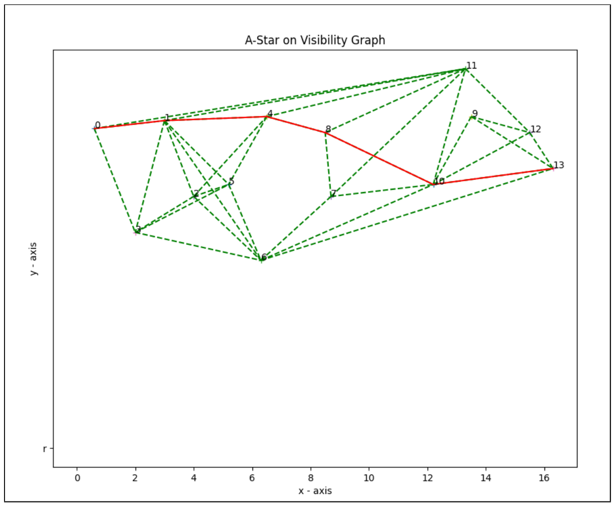
Figure 6: Program Execution



(a) First Graph



(b) Second Graph



(c) Third Graph

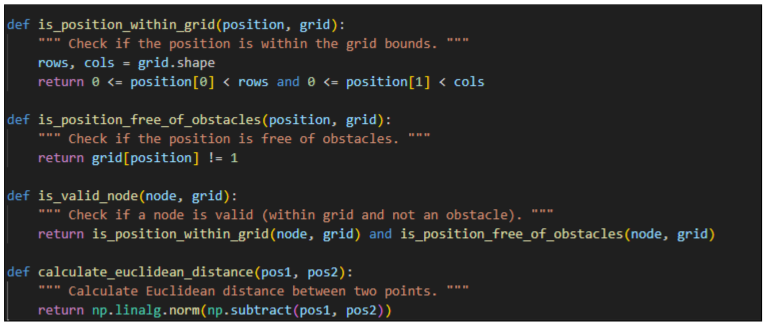
Figure 7: Shortest Path with A* Algorithm

Discrete A* algorithm

In this section, we provide our work on the Discrete A* algorithm on grid map environments with 8-point connectivity. The results are the same as of find_path done previously in lab 1, however here the A* algorithm is used which is different than the Euclidean distance used to find the path previously for grid maps. This section allowed us to test the results with different sets of path-finding algorithms on the grid maps which can be as close to the real-life environment sets with obstacles. The process is detailed in the below-mentioned code snippets.

0.1 Map bounds

We first make the necessary functions to check the bounds of the position of path points, and that they are not obstacles to check node validity. Moreover, the get_distance() function performs the Euclidean distance explained above for the visibility graph.

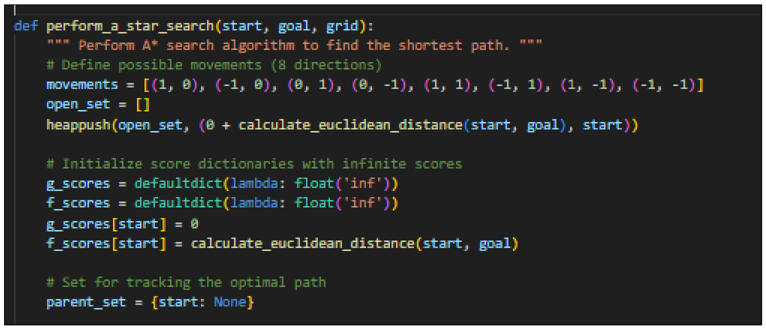


```
def is_position_within_grid(position, grid):
    """ Check if the position is within the grid bounds. """
    rows, cols = grid.shape
    return 0 <= position[0] < rows and 0 <= position[1] < cols

def is_position_free_of_obstacles(position, grid):
    """ Check if the position is free of obstacles. """
    return grid[position] != 1

def is_valid_node(node, grid):
    """ Check if a node is valid (within grid and not an obstacle). """
    return is_position_within_grid(node, grid) and is_position_free_of_obstacles(node, grid)

def calculate_euclidean_distance(pos1, pos2):
    """ Calculate Euclidean distance between two points. """
    return np.linalg.norm(np.subtract(pos1, pos2))
```

Figure 8: Map validity and Euclidean distance

0.2 A star initialization

Secondly, we initialize the variables required for the A* algorithm, and these variables are detailed in section 1 for the visibility graph environments above.

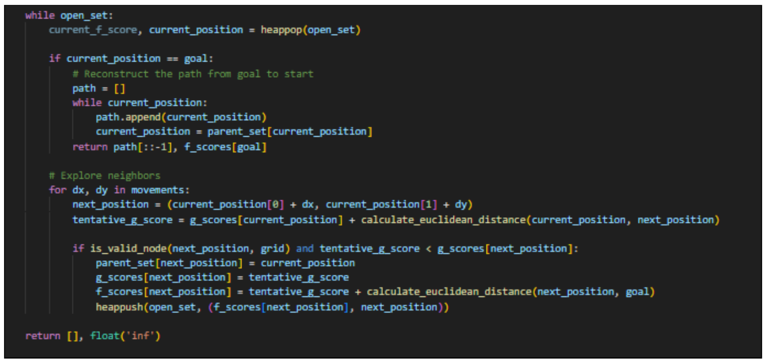


```
def perform_a_star_search(start, goal, grid):
    """ Perform A* search algorithm to find the shortest path. """
    # Define possible movements (8 directions)
    movements = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
    open_set = []
    heappush(open_set, (0 + calculate_euclidean_distance(start, goal), start))

    # Initialize score dictionaries with infinite scores
    g_scores = defaultdict(lambda: float('inf'))
    f_scores = defaultdict(lambda: float('inf'))
    g_scores[start] = 0
    f_scores[start] = calculate_euclidean_distance(start, goal)

    # Set for tracking the optimal path
    parent_set = {start: None}
```

Figure 9: A star initialization

0.3 Discrete A star algorithm

In this A* algorithm implementation, a while loop works the same for visibility graph environments. However, here the movements are taken into account for grid map path-finding with A*. Then, for each possible movement in 8-connectivity, the algorithm calculates the scores for the neighboring nodes. If a neighbor is valid (within the map boundaries and not an obstacle), the algorithm checks whether it is in the open or closed set. If it is in the open set with a lower score, or the closed set, the neighbor is skipped. Otherwise, the scores are updated, and the neighbor is added to the open set. Finally, the open set is sorted based on scores.

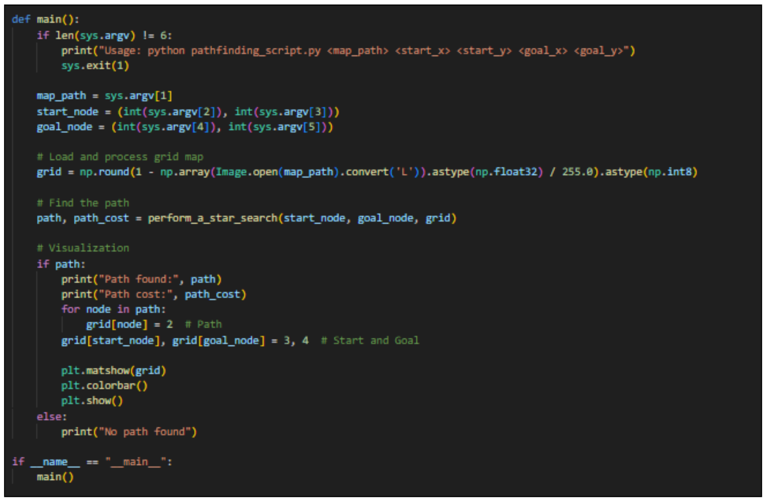


```python
while open_set:
    current_f_score, current_position = heappop(open_set)

    if current_position == goal:
        # Reconstruct the path from goal to start
        path = []
        while current_position:
            path.append(current_position)
            current_position = parent_set[current_position]
        return path[::-1], f_scores[goal]

    # Explore neighbors
    for dx, dy in movements:
        next_position = (current_position[0] + dx, current_position[1] + dy)
        tentative_g_score = g_scores[current_position] + calculate_euclidean_distance(current_position, next_position)

        if is_valid_node(next_position, grid) and tentative_g_score < g_scores[next_position]:
            parent_set[next_position] = current_position
            g_scores[next_position] = tentative_g_score
            f_scores[next_position] = tentative_g_score + calculate_euclidean_distance(next_position, goal)
            heappush(open_set, (f_scores[next_position], next_position))

return [], float('inf')
```

Figure 10: Discrete A star implementation

0.4 Main script

In this part of the code, as shown in the code snippet below, We will use gray-scale images to define our grid map environments with already provided code in the lab guide. To further calculate path and path values as required for part 2 of this lab.

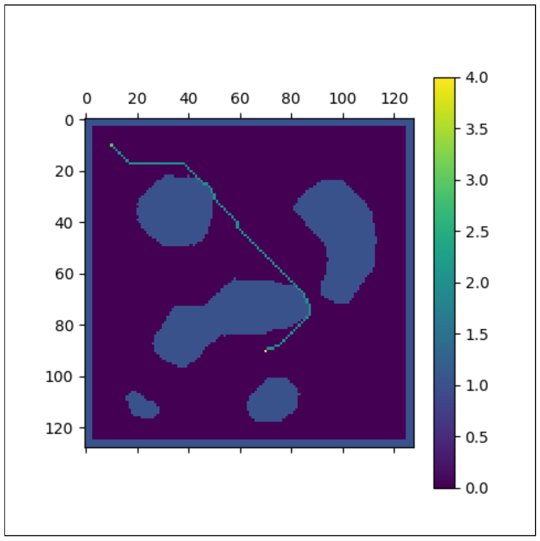


```python
def main():
    if len(sys.argv) != 6:
        print("Usage: python pathfinding_script.py <map_path> <start_x> <start_y> <goal_x> <goal_y>")
        sys.exit(1)

    map_path = sys.argv[1]
    start_node = (int(sys.argv[2]), int(sys.argv[3]))
    goal_node = (int(sys.argv[4]), int(sys.argv[5]))

    # Load and process grid map
    grid = np.round(1 - np.array(Image.open(map_path).convert('L')).astype(np.float32) / 255.0).astype(np.int8)

    # Find the path
    path, path_cost = perform_a_star_search(start_node, goal_node, grid)

    # Visualization
    if path:
        print("Path found:", path)
        print("Path cost:", path_cost)
        for node in path:
            grid[node] = 2  # Path
        grid[start_node], grid[goal_node] = 3, 4  # Start and Goal

        plt.matshow(grid)
        plt.colorbar()
        plt.show()
    else:
        print("No path found")

if __name__ == "__main__":
    main()
```

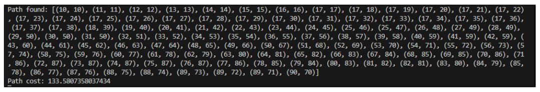
Figure 11: Main script

0.5 Results

The results for 8-point connectivity path-finding with Discrete A* algorithm (path and path cost are also included) are given below tested for map0.png and map1.png:



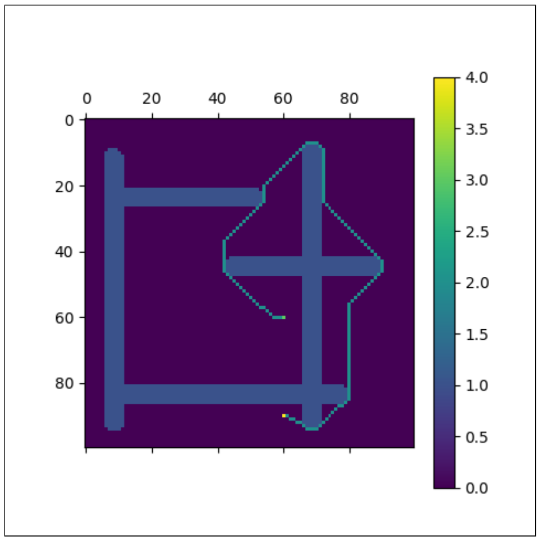
(a) Plotted path



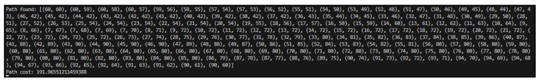
```
Path found: [(10, 10), (11, 11), (12, 12), (13, 13), (14, 14), (15, 15), (16, 16), (17, 17), (17, 18), (17, 19), (17, 20), (17, 21), (17, 22)
, (17, 23), (17, 24), (17, 25), (17, 26), (17, 27), (17, 28), (17, 29), (17, 30), (17, 31), (17, 32), (17, 33), (17, 34), (17, 35), (17, 36),
 (17, 37), (17, 38), (18, 39), (19, 40), (20, 41), (21, 42), (22, 43), (23, 44), (24, 45), (25, 46), (25, 47), (26, 48), (27, 49), (28, 49),
(29, 50), (30, 50), (31, 50), (32, 51), (33, 52), (34, 53), (35, 54), (36, 55), (37, 56), (38, 57), (39, 58), (40, 59), (41, 59), (42, 59), (
43, 60), (44, 61), (45, 62), (46, 63), (47, 64), (48, 65), (49, 66), (50, 67), (51, 68), (52, 69), (53, 70), (54, 71), (55, 72), (56, 73), (5
7, 74), (58, 75), (59, 76), (60, 77), (61, 78), (62, 79), (63, 80), (64, 81), (65, 82), (66, 83), (67, 84), (68, 85), (69, 85), (70, 86), (71
, 86), (72, 87), (73, 87), (74, 87), (75, 87), (76, 87), (77, 86), (78, 85), (79, 84), (80, 83), (81, 82), (82, 81), (83, 80), (84, 79), (85,
 78), (86, 77), (87, 76), (88, 75), (88, 74), (89, 73), (89, 72), (89, 71), (90, 70)]
Path cost: 133.5807358037434
```

(b) Path values

Figure 12: map0 - Discrete A* results



(a) Plotted path



```
Path found: [(60, 60), (60, 59), (60, 58), (60, 57), (59, 56), (58, 55), (57, 54), (57, 53), (56, 52), (55, 51), (54, 50), (53, 49), (52, 48), (51, 47), (50, 46), (49, 45), (48, 44), (47, 4
3), (46, 42), (45, 42), (44, 42), (43, 42), (42, 42), (41, 42), (40, 42), (39, 42), (38, 42), (37, 42), (36, 43), (35, 44), (34, 45), (33, 46), (32, 47), (31, 48), (30, 49), (29, 50), (28,
51), (27, 52), (26, 53), (25, 54), (24, 54), (23, 54), (22, 54), (21, 54), (20, 54), (19, 55), (18, 56), (17, 57), (16, 58), (15, 59), (14, 60), (13, 61), (12, 62), (11, 63), (10, 64), (9,
65), (8, 66), (7, 67), (7, 68), (7, 69), (7, 70), (8, 71), (9, 72), (10, 72), (11, 72), (12, 72), (13, 72), (14, 72), (15, 72), (16, 72), (17, 72), (18, 72), (19, 72), (20, 72), (21, 72), (
22, 72), (23, 72), (24, 72), (25, 72), (26, 73), (27, 74), (28, 75), (29, 76), (30, 77), (31, 78), (32, 79), (33, 80), (34, 81), (35, 82), (36, 83), (37, 84), (38, 85), (39, 86), (40, 87),
(41, 88), (42, 89), (43, 90), (44, 90), (45, 90), (46, 90), (47, 89), (48, 88), (49, 87), (50, 86), (51, 85), (52, 84), (53, 83), (54, 82), (55, 81), (56, 80), (57, 80), (58, 80), (59, 80),
 (60, 80), (61, 80), (62, 80), (63, 80), (64, 80), (65, 80), (66, 80), (67, 80), (68, 80), (69, 80), (70, 80), (71, 80), (72, 80), (73, 80), (74, 80), (75, 80), (76, 80), (77, 80), (78, 80)
, (79, 80), (80, 80), (81, 80), (82, 80), (83, 80), (84, 80), (85, 80), (86, 79), (87, 78), (87, 77), (88, 76), (89, 75), (90, 74), (91, 73), (92, 72), (93, 71), (94, 70), (94, 69), (94, 68
), (94, 67), (93, 66), (92, 65), (92, 64), (91, 63), (91, 62), (90, 61), (90, 60)]
Path cost: 191.96551211459388
```

(b) Path values

Figure 13: map1 - Discrete A* results

Video submission

Click to watch on YouTube

Problem Faced

The only problem we encountered while implementing the A* star algorithm was the reconstructed path function that returned the path from goal to start while we wanted the results from start to goal, this is where we got stuck. However, we solved the same problem by reversing our output path list as required.

Discussion & Conclusions

The objective of the lab has been successfully achieved as we were able to understand and implement the A* algorithm in different environment settings from visibility graphs to grid maps (as the discrete environments), giving us a broader scope of this powerful and efficient path planning algorithm giving us optimal/minimum path.