Students:
Mir Mohibullah Sazid [sazidarnob@gmail.com]
Lisa Paul Magoti [paullisa051@gmail.com]

Lab3 : A Star Algorithm

Introduction

The most effective path between nodes or within graphs is found using the A* method. This algorithm uses heuristics and cost information from the path and integrates it into its solution, making it 'informed'. This report will give a thorough, step-by-step explanation of how we used Python programming to use the A* method to find the optimal path on a visibility graph.

1 Implementation

We used the loadcsv.py file provided in the lab data to do the load the csv file and the provided functions needed for further implementation in the code explained below.
The code is implemented using three functions. At first, the path cost is calculated using the following codes. It a path of the system estimating the cost of the most effective path from node n to the goal. Figure1 shows the code of this function.

Another function is used to add a node either in open dictionary or closed list. This Include the node in either the open dictionary or the closed list depending on its heuristic cost. This function controls the current node's expansion by examining its neighbors, modifying path costs, and monitoring the open and closed sets to guide the search in the direction of the best paths. The coding is shown is Figure 2 The third function is used to perform the A star Algorithm. At first, the data is loaded for nodes and edges from the files. Then heuristic (optimization cost) for each node to the goal is calculated. The path is initialized as well as the search structure. After that, A* search was performed. Then reconstructing the path, the output graph is generated using the pre-existing plot function. Figure 3a and 3b shows the picture of the function.

```
def calculate_past_cost(node, parent, edges, costs):
    if node == 0:
        return 0

    cost = 0
    for edge in edges:
        if node in edge and parent[node] in edge:
            try:
                cost = costs[edge] + calculate_past_cost(parent[node], parent, edges, costs)
                break
            except KeyError:
                return 0

    return cost
```

Figure 1: Path Cost Function

```python
def update_open_closed(node, edge_costs, closed_set, open_set, parents, path_costs, heuristic_costs, edges):
    node_costs = edge_costs[node, :]
    for neighbor in range(len(node_costs)):
        if node_costs[neighbor] > 0 and neighbor not in closed_set:
            temp_parents = parents.copy()
            parents[neighbor] = node
            updated_cost = calculate_past_cost(neighbor, parents, edges, edge_costs)
            if path_costs[neighbor] > updated_cost:
                path_costs[neighbor] = updated_cost
            else:
                parents = temp_parents

            estimated_total_cost = path_costs[neighbor] + heuristic_costs[neighbor]
            open_set[neighbor] = estimated_total_cost

            # Sorting the open set by costs
            sorted_open_set = sorted(open_set.items(), key=lambda item: item[1])
            open_set = dict(sorted_open_set)

    open_set.pop(node, None)  # removes node from open_set if it exists
    closed_set.append(node)

    return open_set, closed_set, parents, path_costs
```

Figure 2: function to add a node either in open dictionary or closed list

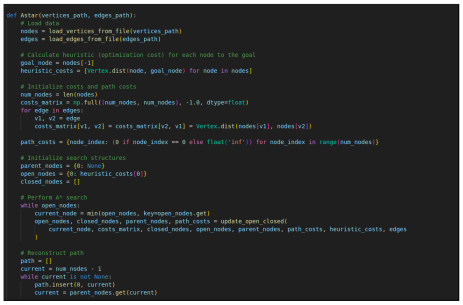
```python
def position(position, grid):
    rows, cols = grid.shape
    return 0 <= position[0] < rows and 0 <= position[1] < cols

def check_obstacle(position, grid):
    return grid[position] != 1

def is_valid_node(node, grid):
    return position(node, grid) and check_obstacle(node, grid)

def get_distance(pos1, pos2):
    return np.linalg.norm(np.subtract(pos1, pos2))
```

Figure 4: Helper function for Grid map environments



(a)

(b)

Figure 3: A star Function

For the grid map environment, three helper function and Astar function is developed. The position function checks if a given position is within the bounds of the grid. The check-obstacles function check if a given position in the grid is not an obstacle. The is-valid-node, uses the above two functions to check if a node is a valid, non-obstacle position on the grid. And lastly the get distance function computes the Euclidean distance between two points. The functions are shown in figure 4 For Astar function, the values are initialized. Then the Astar search is done. At last, the path reconstruction is done. The following function shows in figure 5

```python
def a_star_search(start, goal, grid):
    movements = [(1, 0), (-1, 0), (0, 1), (0, -1), (1, 1), (-1, 1), (1, -1), (-1, -1)]
    open_set = []
    heappush(open_set, (0 + get_distance(start, goal), start))
    g_scores = defaultdict(lambda: float('inf'))
    g_scores[start] = 0
    f_scores = defaultdict(lambda: float('inf'))
    f_scores[start] = get_distance(start, goal)
    parent_set = {start: None}

    while open_set:
        current_f_score, current_position = heappop(open_set)

        if current_position == goal:
            path = []
            while current_position:
                path.append(current_position)
                current_position = parent_set[current_position]
            return path[::-1], f_scores[goal]

        for dx, dy in movements:
            next_position = (current_position[0] + dx, current_position[1] + dy)
            tentative_g_score = g_scores[current_position] + get_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 + get_distance(next_position, goal)
                heappush(open_set, (f_scores[next_position], next_position))

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

Figure 5: A star function for Grid map environments.

Results

The A* method determines which path should be taken by combining the real cost from the start node to the current node with a heuristic estimate of the cost from the current node to the goal. It has reached the goal by moving from node 0 to node 1, then node 3, and lastly node 4. Given the planning of the graph and the expenses involved in relocating from one node to another, the algorithm's decision-making process created the solid red path, which represents the most efficient way from the start node to the goal node. The optimal path's separation between nodes 0 and 4 is 12.12356982653498. Figure 6 shows the output of graph.
The output for env-1.csv and visibility-graph-1.csv are: Final path [0, 2, 5], total cost 5.870358207916741 and the graph is depicted in figure 7. Finally, The results for env-2.csv and visibility-graph-2.csv are: Final path [0, 1, 4, 8, 10, 13], total cost 15.990555296232605 and the graph is depicted in figure 8 The output of map0 is shows the 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), (57, 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)] and the Path cost: 133.5807358037434. The following figure 9shows the output. The output for map1 is 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, 43), (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),

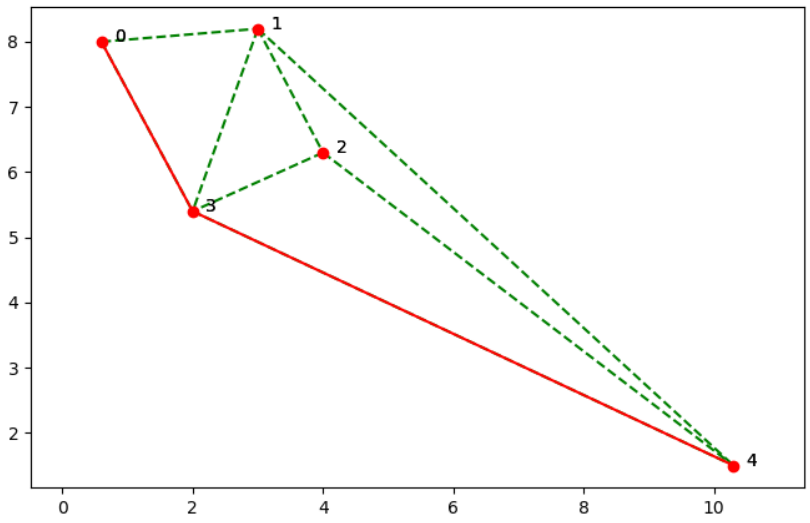


Figure 6: Optimal path from env-0.csv file

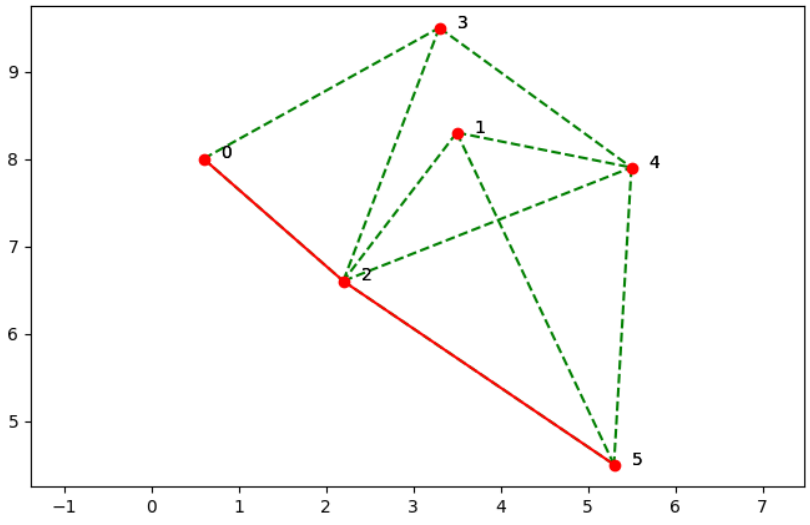


Figure 7: Optimal Path from env-1.csv

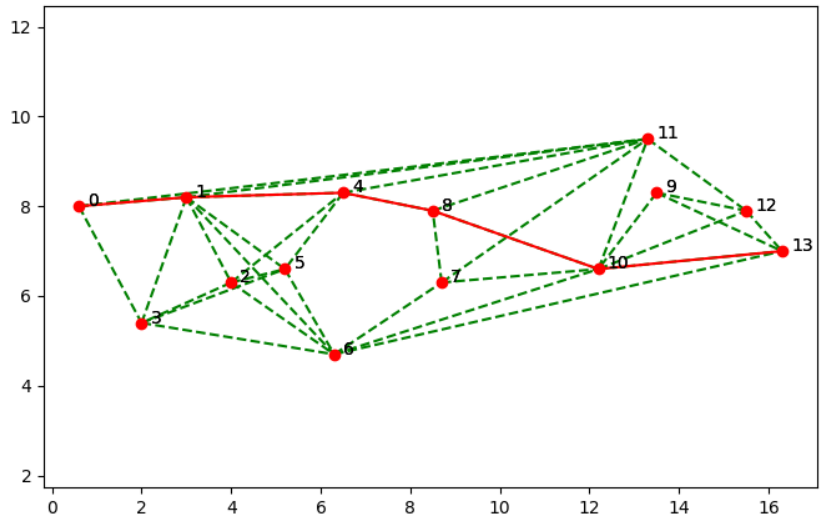


Figure 8: Optimal Path from env-2.csv

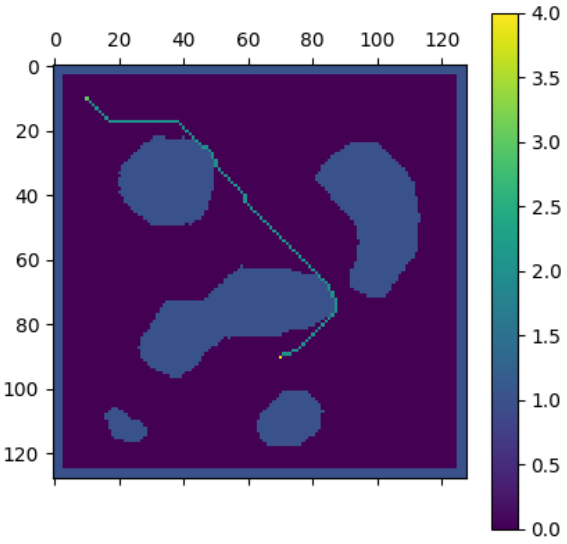


Figure 9: Using 8 point connectivity Ploted path

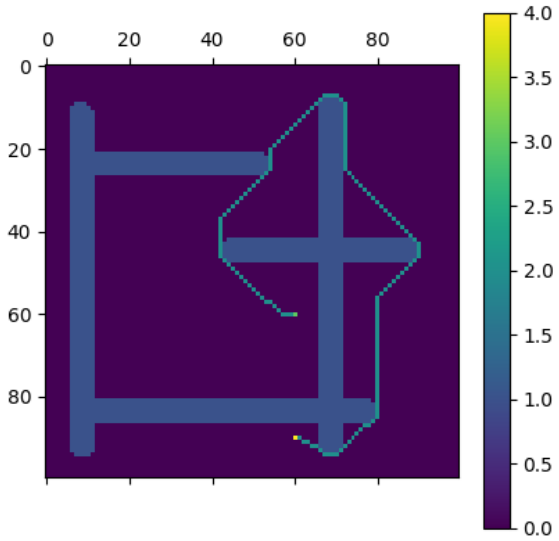
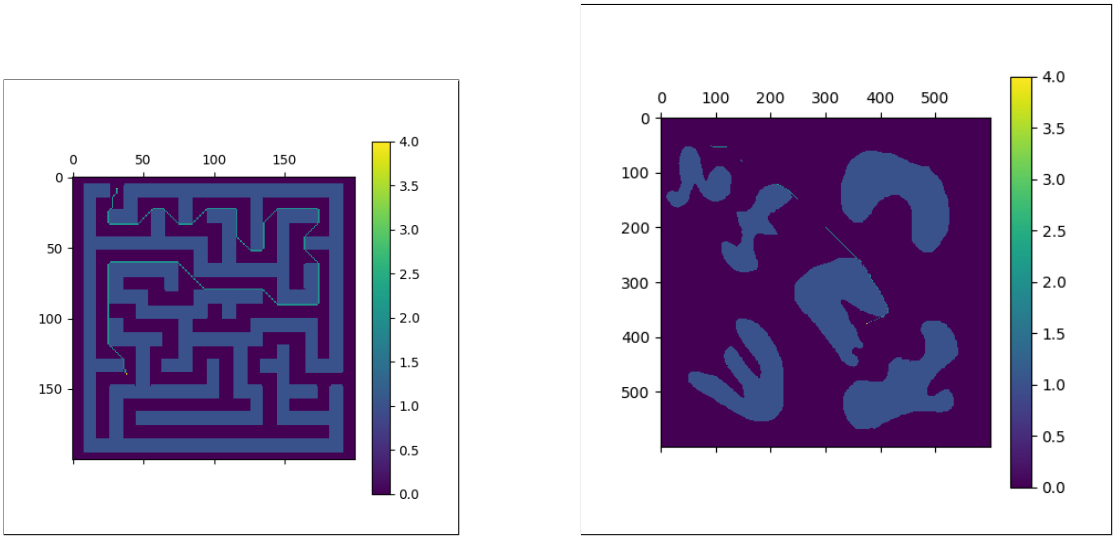


Figure 10: For map1

(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 The following figure 10 shows the output. For the map2 and map3 the Path cost: 555.8477631085033 and Path cost: 523.3940110268419 respectively. The figure shows the output:



(a) (b)

Figure 11: For map2 and map3

Problem Faced

During the gScore updates in each iteration, we addressed the absence of actual edge costs by treating the edges as unit length, leading to diverse paths. Subsequently, we applied the A star function with Euclidean distance, resulting in optimal paths.

Conclusion

The main objective of the lab was to understand the Astar algorithm and implement it which is done successfully.