Students:
Loc Thanh Pham [u1992429@udg.campus.edu]
Eliyas Kidanemariam Abraha [eliyaskidane@gmail.com]

Lab3:A* Algorithm

Introduction

This lab report presents the implementation of A* search algorithm which is related to path planning algorithm for searching a graph efficiently.A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm).Compared to Dijkstra's algorithm, the A* algorithm only finds the shortest path from a specified source to a specified goal, and not the shortest-path tree from a specified source to all possible goals.A* achieves better performance by using heuristics to guide its search.Particularly this lab we implement A* algorithm to find optimal path in visibility graph. Additionally, we implement also finding an optimal path in discrete environment such grid map.

Methodology

The A* algorithm is a widely used path finding algorithm that efficiently finds the optimal path from a start node to a goal node in a graph or grid. It combines the benefits of Dijkstra's algorithm and Greedy Best-First Search by incorporating both the cost of reaching a node and a heuristic estimate of the remaining cost to the goal. The algorithm maintains two lists, the open list, and the closed list, to systematically explore the search space.

Key Points:

- **Cost Function(g(n))**: The cost function represents the cumulative cost of reaching a particular node from the start node. In each iteration, the algorithm updates the cost of reaching neighboring nodes by considering the cost of the current node and the cost of the edge to the neighbor.

- **Heuristic Function (h(n)):** The heuristic function provides an estimate of the remaining cost from a node to the goal. It guides the algorithm by favoring nodes that are expected to lead to a more optimal solution. The heuristic should be admissible, meaning it never overestimates the true cost to reach the goal.There are different of heuristic calculation method , here we approximated the heuristic using euclidean distance

$$h(n) = \sqrt{(x_{\text{goal}} - x_n)^2 + (y_{\text{goal}} - y_n)^2}$$

- **Evaluation Function (f(n)):**The evaluation function combines the cost and heuristic to prioritize nodes for exploration. It is defined as f(n) = g(n) + h(n). Nodes with lower values

of f(n) are considered first, allowing the algorithm to explore paths that appear promising based on both the actual cost and the heuristic

- **Open List:**The open list contains nodes that are candidates for exploration. Initially, it only includes the start node. During each iteration, the algorithm selects the node with the lowest f(n) from the open list for expansion. The open list is updated as new nodes are discovered
- **Closed List:**The closed list contains nodes that have already been visited and expanded. It prevents the algorithm from revisiting nodes and getting stuck in cycles. Nodes are moved from the open list to the closed list once they have been expanded.

The algorithmic steps employed for the implementation of the A* algorithm to determine the optimal path in both graph-based and discrete environments are outlined as follows:

Algorithm Flow:

1. **Initialization**:
   - Add the start node to the open list with initial cost of 0
   - Calculate the heuristic value for the start node
   - Set the closed list to empty
2. **Main Loop**:
   - While the open list is not empty:
     - Select the node with the lowest '**f(n)**' value from the open list.
     - If the selected node is the goal , the optimal paths has been found. Terminate the algorithm.
     - Otherwise, move the node to the closed list and expand its neighbors.
3. **Node Expansion:** For each neighboring node do the following:
   - Calculate the tentative g(n) value (cumulative cost from the start node).
   - If the node is not in the open list, calculate its heuristic value.
   - f the node is not in the closed list or the new g(n) value is lower than the existing one:
     - Update the node's g(n) value.
     - Calculate the node's f(n) value (f(n) = g(n) + h(n)).
     - If the node is not in the open list, add it; otherwise, update its position based on the new f(n) value.
4. **Termination :** The algorithm terminates under the following conditions:
   - The goal node is reached. The optimal path has been found
   - The open list is empty, indicating that no path exists to the goal.

- If the node is not in the closed list or the new g(n) value is lower than the existing one:

5. **Path Reconstruction :** Once the goal node reached , reconstract the optimal path:
   - Trace back from the goal node to the start using the information stored in each node's parent pointer.

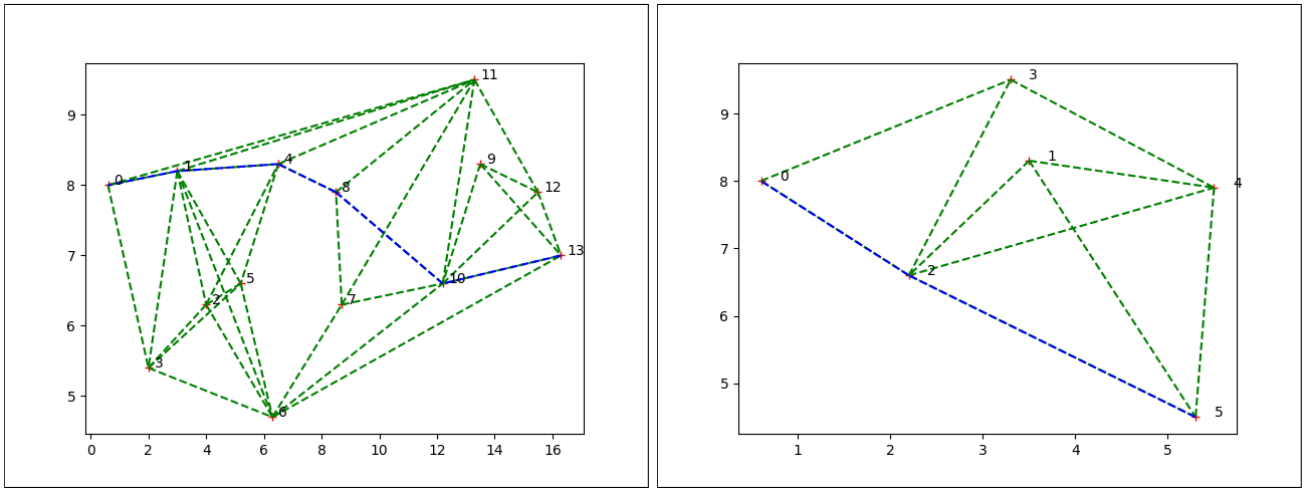
Algorithm 1: A* Search Algorithm

Input: Starting node, Goal node
Output: Optimal path from start to goal

Function `Astar(`*start, goal*`)`**:**
- **Initialize** open list
- **Initialize** closed list
- Put the starting node on the open list (you can leave its f at zero)
- **while** *open list is not empty* **do**
  - Find the node with the least f on the open list, call it q
  - Pop q off the open list
  - Generate q's successors and set their parents to q
  - **for** *each successor* **do**
    - **if** *successor is the goal* **then**
      - Stop search and reconstract the path
    - **else**
      - Compute both g and h for successor
      - $successor.g = q.g+$ distance between *successor* and q
      - $successor.h =$ distance from *goal* to *successor*
      - $successor.f = successor.g + successor.h$
      - **if** *a node with the same position as successor is in the OPEN list and has a lower f than successor* **then**
        - Skip this successor
      - **if** *a node with the same position as successor is in the CLOSED list and has a lower f than successor* **then**
        - Skip this successor
      - Otherwise, add the node to the open list
  - Push q on the closed list
- **return** Failure (no path found)

Result

1 Part 1: Graph environment



(a) Map 2 (b) Map 1

Figure 1: The results on the graph environment

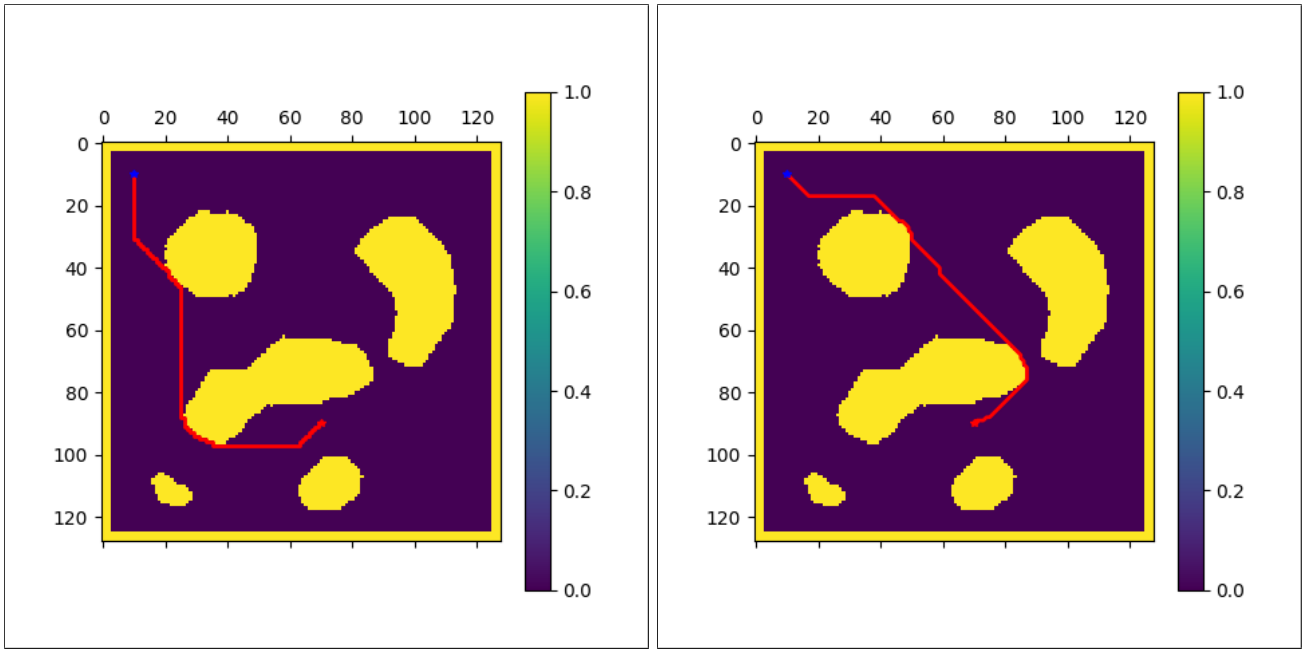
Map 2

The list of vertices representing the optimal path:
Path: [5, 2, 0]
Position: [(5.3, 4.5), (2.2, 6.6), (0.6, 8.0)]
The score of the optimal path: 5.87

Map 1

The list of vertices representing the optimal path:
Path: [13, 10, 8, 4, 1, 0]
Position: [(16.3, 7.0), (12.2, 6.6), (8.5, 7.9), (6.5, 8.3), (3.0, 8.2), (0.6, 8.0)]
The score of the optimal path: 14.97

2 Part 2: Grid environment



(a) 4-Connectivity (b) 8-Connectivity

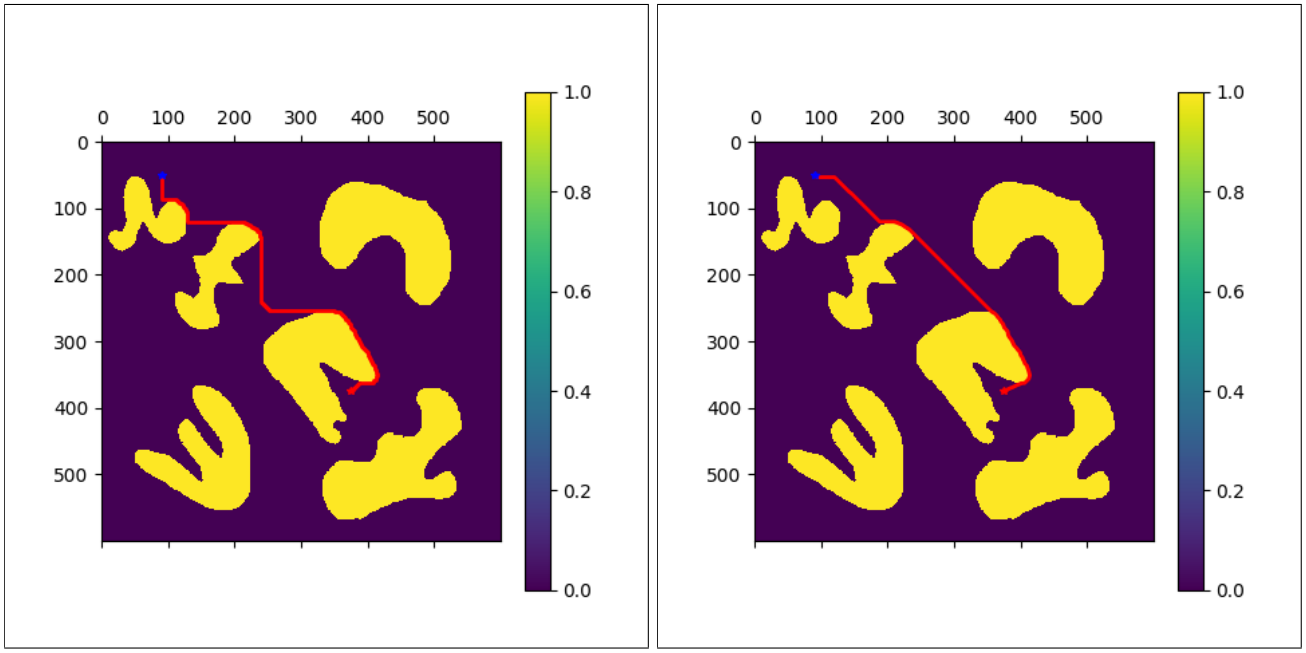
Figure 2: The results on the grid environment with map 0

Result of the 4-connectivity algorithm

The list of vertices representing the optimal path: Path: [(90, 70), (90, 69), (91, 69), (91, 68), (92, 68), (92, 67), (93, 67), (93, 66), (94, 66), (94, 65), (95, 65), (95, 64), (96, 64), (96, 63), (97, 63), (97, 62), (97, 61), (97, 60), (97, 59), (97, 58), (97, 57), (97, 56), (97, 55), (97, 54), (97, 53), (97, 52), (97, 51), (97, 50), (97, 49), (97, 48), (97, 47), (97, 46), (97, 45), (97, 44), (97, 43), (97, 42), (97, 41), (97, 40), (97, 39), (97, 38), (97, 37), (97, 36), (97, 35), (96, 35), (96, 34), (96, 33), (95, 33), (95, 32), (95, 31), (94, 31), (94, 30), (94, 29), (93, 29), (93, 28), (92, 28), (92, 27), (91, 27), (91, 26), (90, 26), (89, 26), (88, 26), (88, 25), (87, 25), (86, 25), (85, 25), (84, 25), (83, 25), (82, 25), (81, 25), (80, 25), (79, 25), (78, 25), (77, 25), (76, 25), (75, 25), (74, 25), (73, 25), (72, 25), (71, 25), (70, 25), (69, 25), (68, 25), (67, 25), (66, 25), (65, 25), (64, 25), (63, 25), (62, 25), (61, 25), (60, 25), (59, 25), (58, 25), (57, 25), (56, 25), (55, 25), (54, 25), (53, 25), (52, 25), (51, 25), (50, 25), (49, 25), (48, 25), (47, 25), (46, 25), (46, 24), (45, 24), (45, 23), (44, 23), (44, 22), (43, 22), (43, 21), (42, 21), (41, 21), (41, 20), (40, 20), (40, 19), (39, 19), (39, 18), (38, 18), (38, 17), (37, 17), (37, 16), (36, 16), (36, 15), (35, 15), (35, 14), (34, 14), (34, 13), (33, 13), (33, 12), (32, 12), (32, 11), (31, 11), (31, 10), (30, 10), (29, 10), (28, 10), (27, 10), (26, 10), (25, 10), (24, 10), (23, 10), (22, 10), (21, 10), (20, 10), (19, 10), (18, 10), (17, 10), (16, 10), (15, 10), (14, 10), (13, 10), (12, 10), (11, 10), (10, 10)]
The score of the optimal path: 154

Result of the 8-connectivity algorithm

The list of vertices representing the optimal path: Path: [(90, 70), (89, 71), (89, 72), (89, 73), (88, 74), (88, 75), (87, 76), (86, 77), (85, 78), (84, 79), (83, 80), (82, 81), (81, 82), (80, 83), (79, 84), (78, 85), (77, 86), (76, 87), (75, 87), (74, 87), (73, 87), (72, 87), (71, 86), (70, 86), (69, 85), (68, 85), (67, 84), (66, 83), (65, 82), (64, 81), (63, 80), (62, 79), (61, 78), (60, 77), (59, 76), (58, 75), (57, 74), (56, 73), (55, 72), (54, 71), (53, 70), (52, 69), (51, 68), (50, 67), (49, 66), (48, 65), (47, 64), (46, 63), (45, 62), (44, 61), (43, 60), (42, 59), (41, 59), (40, 59), (39, 58), (38, 57), (37, 56), (36, 55), (35, 54), (34, 53), (33, 52), (32, 51), (31, 50), (30, 50), (29, 50), (28, 49), (27, 49), (26, 48), (25, 47), (25, 46), (24, 45), (23, 44), (22, 43), (21, 42), (20, 41), (19, 40), (18, 39), (17, 38), (17, 37), (17, 36), (17, 35), (17, 34), (17, 33), (17, 32), (17, 31), (17, 30), (17, 29), (17, 28), (17, 27), (17, 26), (17, 25), (17, 24), (17, 23), (17, 22), (17, 21), (17, 20), (17, 19), (17, 18), (17, 17), (16, 16), (15, 15), (14, 14), (13, 13), (12, 12), (11, 11), (10, 10)]
The score of the optimal path: 133.99



(a) 4-Connectivity (b) 8-Connectivity

Figure 3: The results on the grid environment with map 3

Result of the 4-connectivity algorithm

The list of vertices representing the optimal path: [(375, 375), (375, 376), (374, 376), (374, 377), (373, 377), (373, 378), (372, 378), (372, 379), (371, 379), (371, 380), (370, 380), (370, 381), (369, 381), (369, 382), (368, 382), (368, 383), (367, 383), (367, 384), ... , (56, 90), (55, 90), (54, 90), (53, 90), (52, 90), (51, 90), (50, 90)]
The score of the optimal path: 688

Result of the 8-connectivity algorithm

The list of vertices representing the optimal path: Path: [(375, 375), (374, 376), (374, 377), (374, 378), (373, 379), (373, 380), (372, 381), (372, 382), (371, 383), (371, 384), (370, 385), (370, 386), (370, 387), (369, 388), (369, 389), ... , (51, 91), (50, 90)]
The score of the optimal path: 523.81

3 Part 3: Maze environment

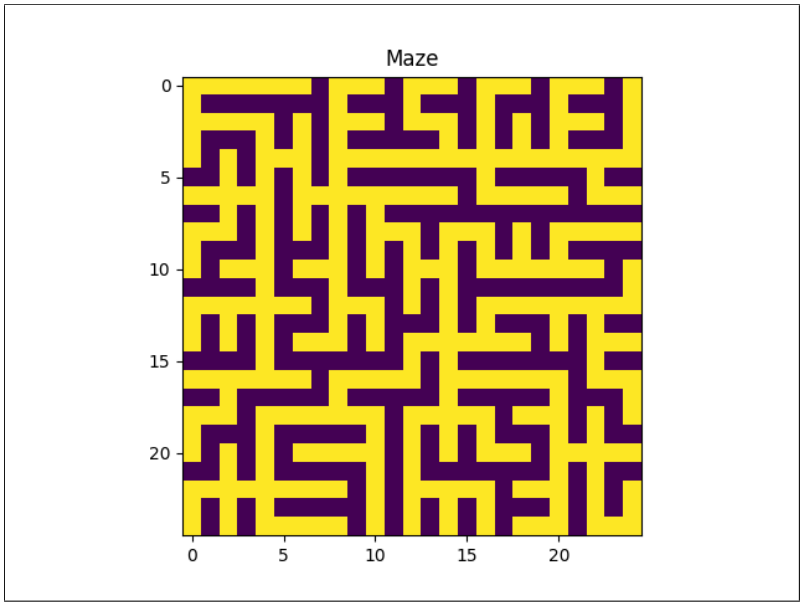


Figure 4: The maze environment

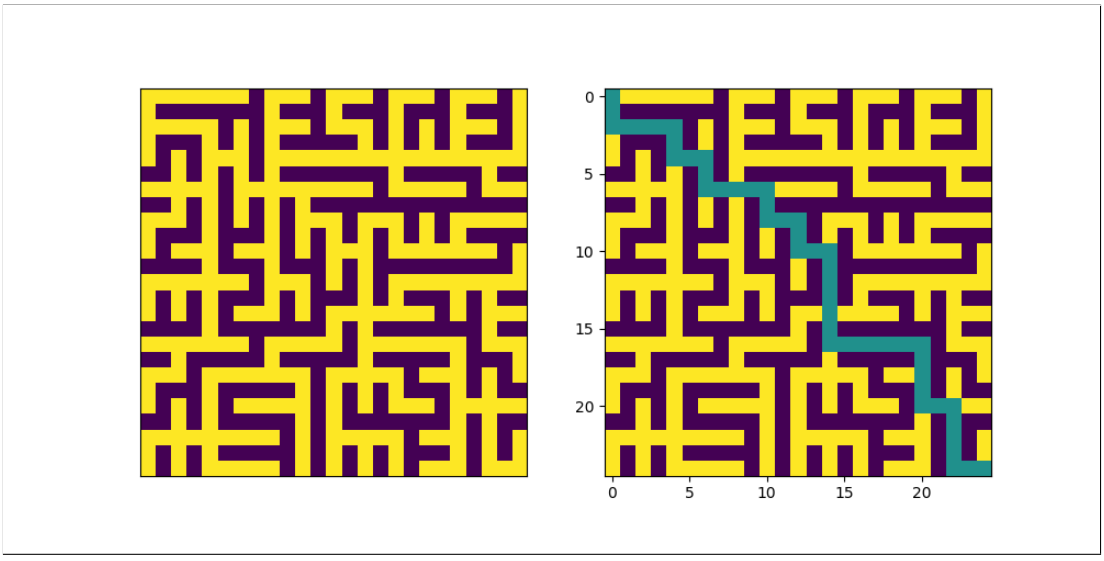


Figure 5: The results on the maze environment

The list of vertices representing the optimal path: Path: [(24, 24), (24, 23), (24, 22), (23, 22), (22, 22), (21, 22), (20, 22), (20, 21), (20, 20), (19, 20), (18, 20), (17, 20), (16, 20), (16, 19), (16, 18), (16, 17), (16, 16), (16, 15), (16, 14), (15, 14), (14, 14), (13, 14), (12, 14), (11, 14), (10, 14), (10, 13), (10, 12), (9, 12), (8, 12), (8, 11), (8, 10), (7, 10), (6, 10), (6, 9), (6, 8), (6, 7), (6, 6), (5, 6), (4, 6), (4, 5), (4, 4), (3, 4), (2, 4), (2, 3), (2, 2), (2, 1), (2, 0), (1, 0), (0, 0)]
The score of the optimal path: 48

Issues

Dictionary Data Type

To facilitate the A* algorithm implementation process, in this lab, I use the dictionary data type for vertices and visibility graph. With each vertex, we didn't number it like the previous lab to differentiate to another vertex, we use the coordinate or position of the vertex as the key of a dictionary and the position of other visibility vertices from this vertex as a values of the dictionary. By this way, we do not have to check what is the number of vertices and map this number to list of visibility graph. By this way, getting data to compute the A* algorithm will be simpler and save time to process.

Maze environment

To check the correctness of the program and the A* algorithm, we create a new environment, which is maze environment. There is a maze generator, which we clone from an open source on the Github.com, we use this one to create a random maze and test my A* program. Finally, we have results in the figure 4, 5 above.

Conclusion

In conclusion, the A-star (A*) is a mighty algorithm in Artificial Intelligence with a wide range of usage. However, it is only as good as its heuristic function (which can be highly variable considering the nature of a problem).