Autonomous Systems

Lab #3 – Graph Search A* Algorithm

Delivered by:

Mazen Elgabalawy u1999109
Solomon Chibuzo Nwafor u1999124

Supervisor:

Alaaeddine El Masri El Chaarani

Date of Submission:

21/11/2024

Table of Contents

1. Final Outcome:

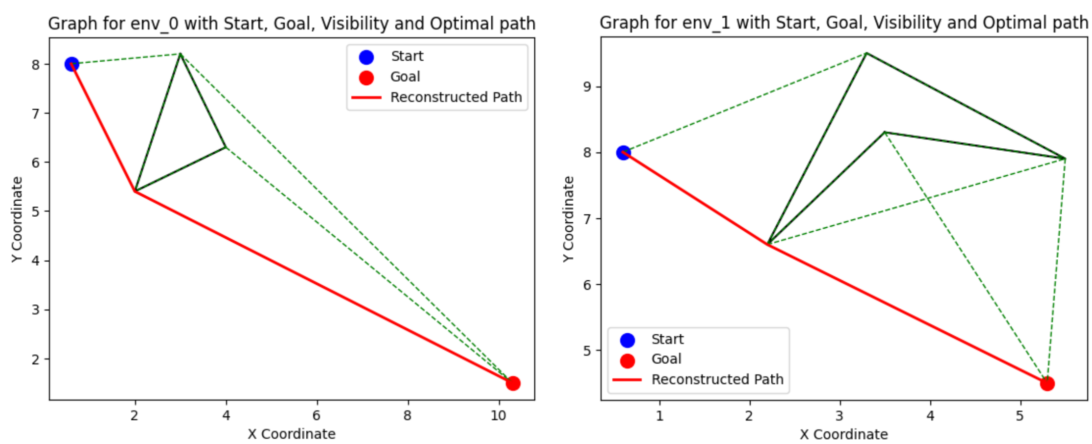
The following section presents the results for the **Graph Search** and the **Grid-map Search** parts of the lab.

1.1. Graph Search:

The results of applying the A* algorithm on the graphs generated using the *csv* files provided for the lab. Figure 1 shows the results for the 4 provided environments, showing the **Start, Goal, Visibility** and the **Optimal Path** generated using A*. Table 1 contains the cost for the path obtained in each map as well as the path itself, both in vertex numbers and vertex coordinates.

Environment	Path Cost	Path Vertices	Path Coordinates
Env0	12.12	[0, 3, 4]	[(0.6, 8.0), (2.0, 5.4), (10.3, 1.5)]
Env1	5.87	[0, 2, 5]	[(0.6, 8.0), (2.2, 6.6), (5.3, 4.5)]
Env2	15.99	[0, 1, 4, 8, 10, 13]	[(0.6, 8.0), (3.0, 8.2), (6.5, 8.3), (8.5, 7.9), (12.2, 6.6), (16.3, 7.0)]
Envmx	39.86	[0, 4, 5, 6, 7, 8, 31]	[(14.0, 10.1), (12.2, 21.5), (13.0, 25.0), (17.1, 25.4), (22.0, 24.4), (25.5, 25.0), (37.5, 23.8)]

Table.1: Path and Path cost for each environment.



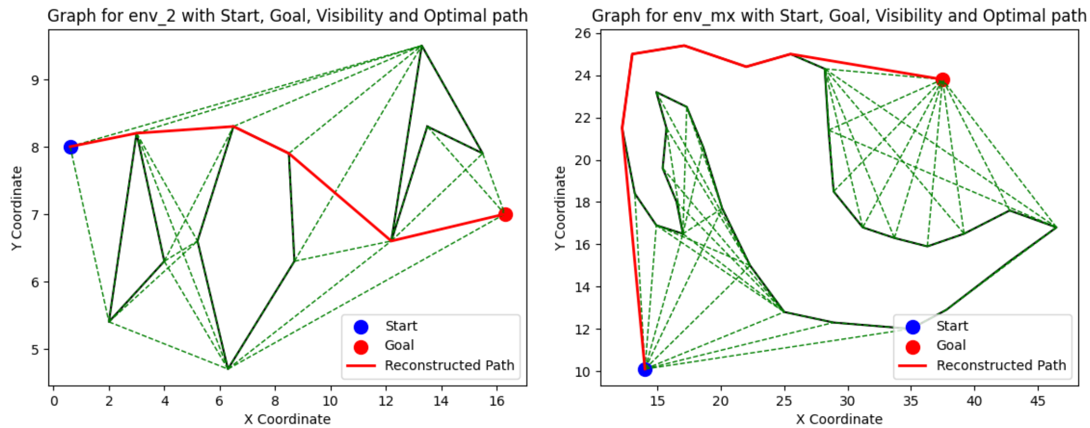


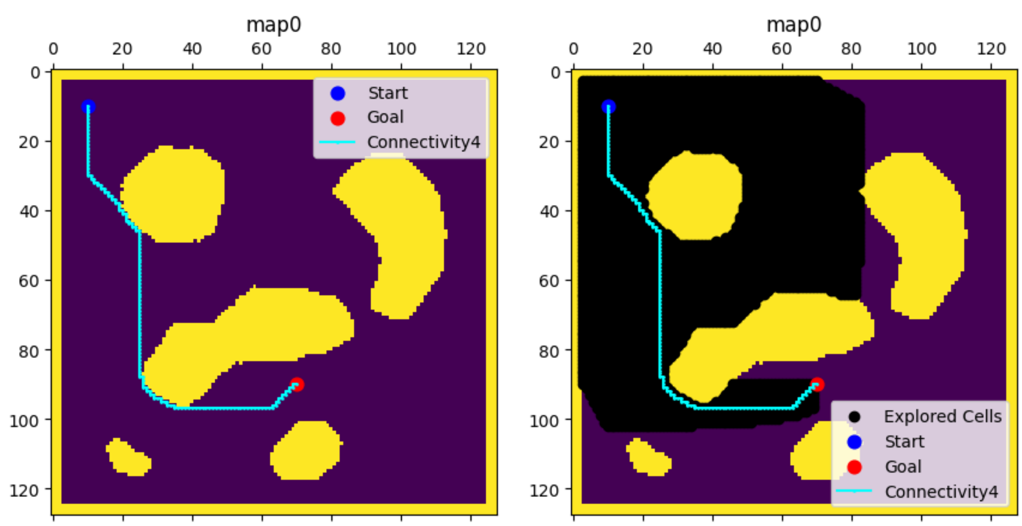
Fig.1: Results for all environments, showing the visibility graph and the optimal path obtained using A*.

1.2. Grid-map Search:

This section shows the results of applying the A* algorithm on all the provided grid-maps. Figure 2 shows the results for the obtained for connectivity 4 along with the map of explored cells, while figure 3 shows the results for connectivity 8. Figure 4 combines paths for both types of connectivity (4 and 8). Table 2 contains the **Start, Goal,** and **Path Cost** for each connectivity type.

Map	Start	Goal	Connectivity 4 Cost	Connectivity 8 Cost
map0	(10,10)	(90,70)	154.0	133.58
map1	(60,60)	(90,60)	240.0	191.96
map2	(8,31)	(139,38)	632.0	555.85
map3	(50,90)	(375,375)	688.0	523.39

Table.2: Start, Goal and Cost for each map.



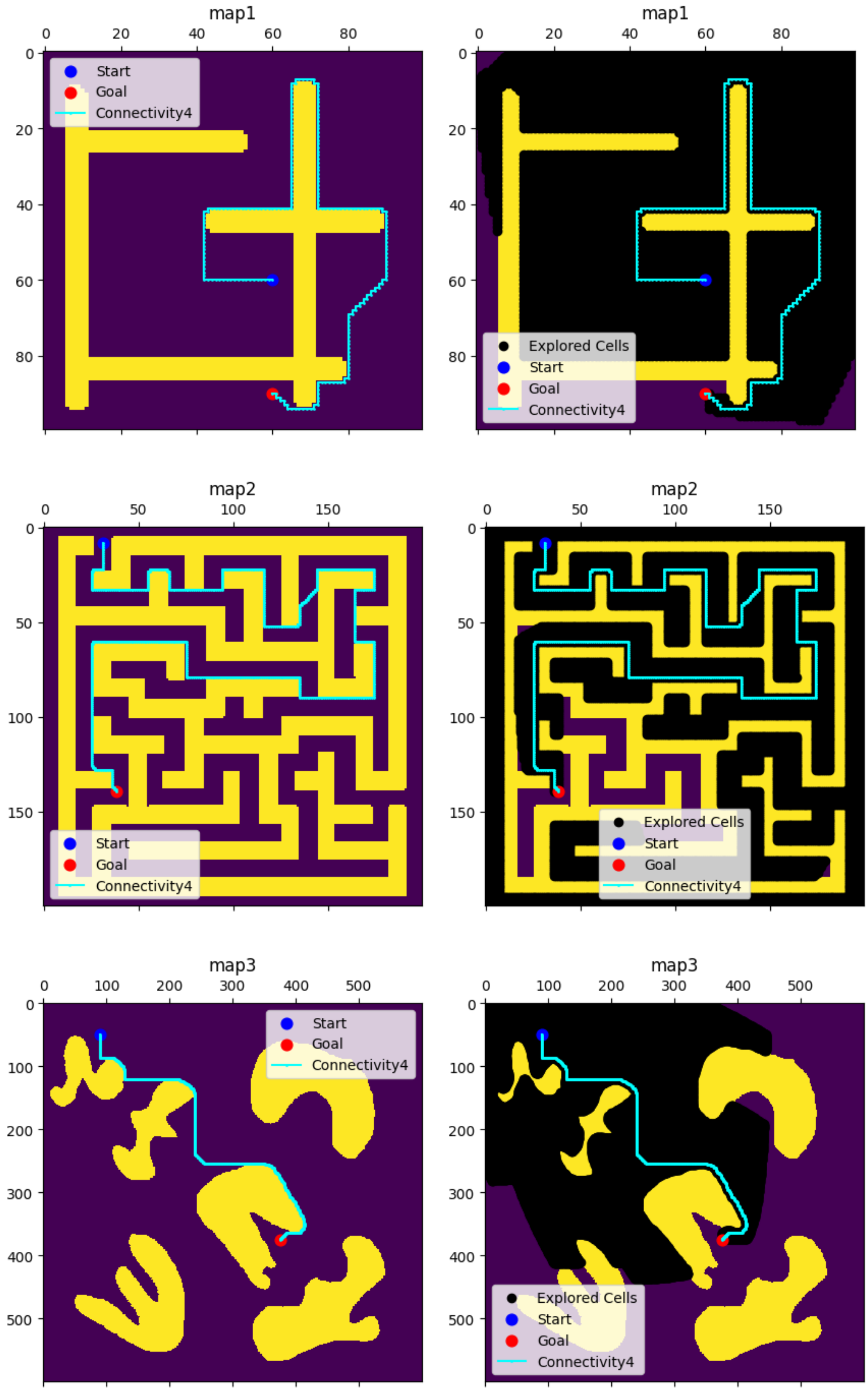
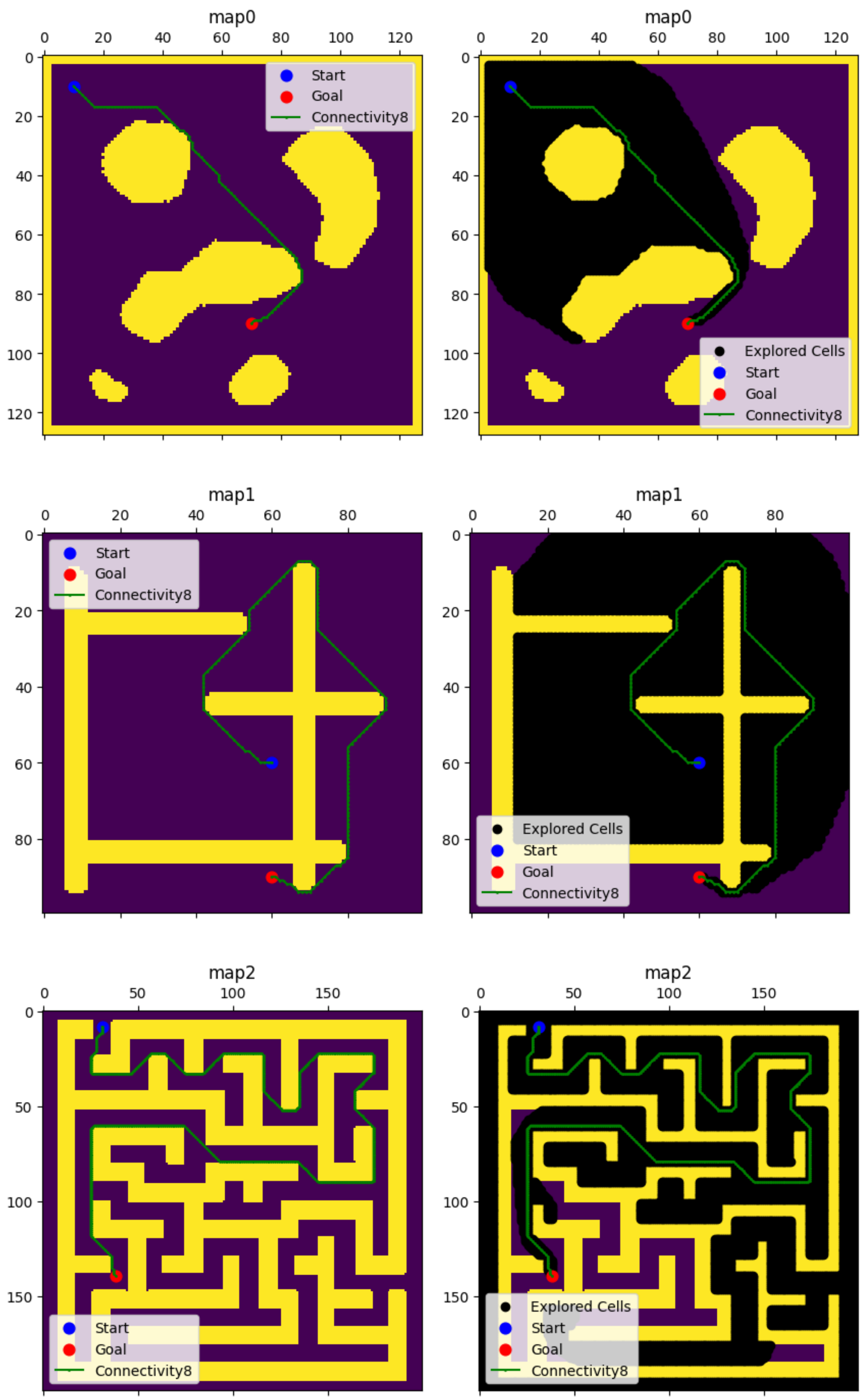


Fig.2: Resulting Path using connectivity 4 with the explored cells.

Lab 3 – Graph Search A* Algorithm – Mazen Elgabalawy u1999109, Solomon Nwafor u1999124



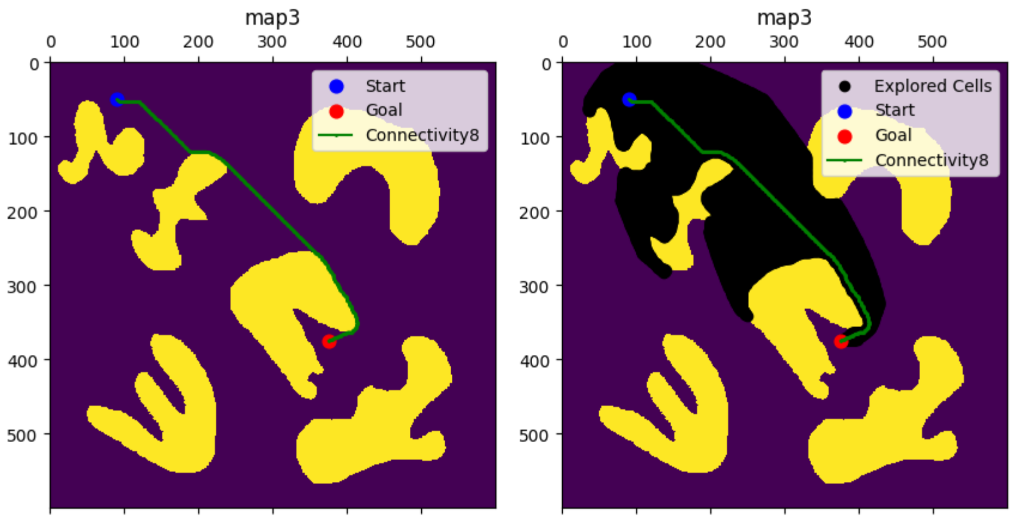


Fig.3: Resulting Path using connectivity 8 with the explored cells.

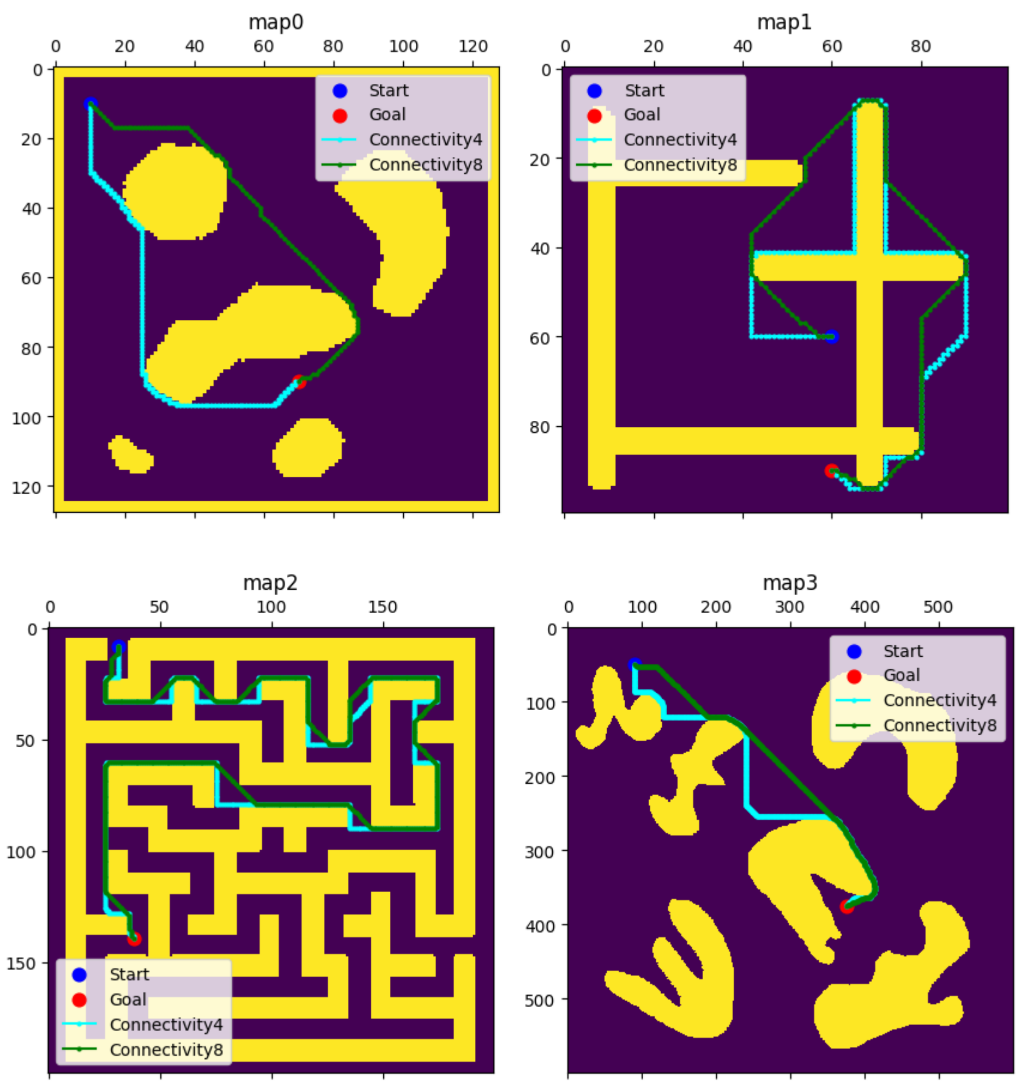


Fig.4: Resulting Path using connectivity 4 and connectivity 8 for all provided maps.

2. Implementations:

This section includes the implementation methods for some of the functions used in this lab.

2.1. Helper Functions:

Here we provide the pseudocode for the functions used to ease the implementation of the A* algorithm.

2.1.1. Euclidean Distance:

This function is used to calculate the Euclidean distance between two points. This is used for calculating the heuristic cost and the cost between the current cell and its neighbor.

```
FUNCTION euclidean_distance(q1, q2):
    # Calculate the squared difference in x-coordinates
    dx_squared ← (q1[0] - q2[0])^2

    # Calculate the squared difference in y-coordinates
    dy_squared ← (q1[1] - q2[1])^2

    # Compute the square root of the sum of squared differences
    distance ← SQUARE_ROOT(dx_squared + dy_squared)

    # Return the calculated distance
    RETURN distance
```

2.1.2. Contains Cell:

This function checks if a certain node is already in the **open_set** or not. This is used to make sure that we don't add nodes that are already in the list twice.

```
FUNCTION contains_cell(lst, node):
    # Iterate through each item in the list
    FOR each pair (_, cell_value) IN lst:
        # Check if the current cell_value matches the node
        IF cell_value EQUALS node:
            RETURN True

    # If no match is found, return False
    RETURN False
```

2.1.3. Reconstruct Path:

After reaching the goal, this method is used to return the whole path from the start configuration to the goal.

```
FUNCTION reconstruct_path(parent, current):
   # Initialize the path list with the current node
   path ← [current]

   # Loop until the current node is no longer in the parent keys
   WHILE current EXISTS IN parent.keys():
      # Update the current node to its parent
      current ← parent[current]
      # Prepend the current node to the path
      path ← [current] + path

   # Return the path excluding the start node
   RETURN path[1:]
```

2.2. Data Load:

For plotting the graph, first it was necessary to load the node and visibility edges data from the provided files. For this, we implemented the function **load_data**, that extracts the required data from the provided **csv** files and returns them. The pseudocode for the function is written below.

```
FUNCTION load_data (env, visibility):
   # Initialize empty lists for storing data and edges
   data ← []
   edges ← []

   # Load point data from the `env` file
   OPEN the file `env` in read mode AS csvfile:
      INITIALIZE reader to read csvfile
      SKIP the first row (header row)
      FOR each row in reader:
         polygon_id ← CONVERT row[0] to integer
         x ← CONVERT row[1] to float
         y ← CONVERT row[2] to float
         APPEND (polygon_id, x, y) to data

   # Load edge data from the `visibility` file
   OPEN the file `visibility` in read mode AS csvfile:
      INITIALIZE reader to read csvfile
      SKIP the first row (header row)
      FOR each row in reader:
         start_vertex ← CONVERT row[0] to integer
         end_vertex ← CONVERT row[1] to integer
         APPEND (start_vertex, end_vertex) to edges

   # Return the loaded data and edges
   RETURN data, edges
```

2.3. A* algorithm:

Here we provide the pseudocode for implementing the A* algorithm. This pseudocode is the general framework from which we implemented two different functions, **a_star_graph** and **a_star_grid**, which are used to search a graph and grid-map respectively. Both functions are very similar but with minor changes to accommodate the different environments which they are implemented on. The pseudocode for the A* algorithm is given below.

```
FUNCTION a_star_algorithm(points, neighbors):
  # Initialize the open set (priority queue)
  open_set ← EMPTY PRIORITY QUEUE

  # Get indices for all vertices
  vertices ← LIST OF INDICES FROM 0 TO LENGTH(points) - 1

  # Define start and goal nodes
  start ← vertices[0]
  goal ← vertices[-1]

  # Create a dictionary to track the parent of each node
  parent ← EMPTY DICTIONARY
  parent[start] ← None

  # Initialize the cost from start (gScore) for all nodes
  gScore ← ARRAY OF SIZE LENGTH(points) FILLED WITH INFINITY
  gScore[start] ← 0

  # Initialize the total estimated cost (fScore) for all nodes
  fScore ← ARRAY OF SIZE LENGTH(points) FILLED WITH INFINITY
  fScore[start] ← gScore[start] + HEURISTIC_DISTANCE(points[start], points[goal])

  # Add the start node to the open set
  ADD (fScore[start], start) TO open_set

  # Main loop: Process nodes from the open set until the goal is reached or open_set is empty
  WHILE open_set IS NOT EMPTY:
    # Get the node with the lowest fScore from the open set
    _, current ← REMOVE NODE WITH LOWEST fScore FROM open_set

    # If the current node is the goal, reconstruct and return the path and its cost
    IF current IS goal:
      path ← RECONSTRUCT_PATH(parent, current)
      cost ← gScore[path[-1]]
      RETURN path, cost

    # Check neighbors of the current node
    FOR each neighbor IN neighbors[current]:
      # Calculate tentative gScore for the neighbor
      tentative_gScore ← gScore[current] + DISTANCE(points[current], points[neighbor])

      # If the tentative gScore is better than the current gScore, update the values
      IF tentative_gScore < gScore[neighbor]:
```

```
            parent[neighbor] ← current
            gScore[neighbor] ← tentative_gScore
            fScore[neighbor] ← tentative_gScore + HEURISTIC_DISTANCE(points[neighbor],
points[goal])

            # If the neighbor is not already in the open set, add it
            IF neighbor IS NOT IN open_set:
               ADD (fScore[neighbor], neighbor) TO open_set

    # If the open set is empty and no path is found, return failure
    RETURN "No path found"
```

3. Challenges:

3.1. Connecting Vertices of Polygons:

One of the challenges during the graph construction process was identifying which vertices belonged to each polygon and ensuring their visibility to establish correct connections between them.

To address this, the vertices of each polygon were extracted sequentially in their defined order. Each vertex was then connected to the next one in the sequence to form edges, and the final vertex was connected back to the first vertex, ensuring the polygon was closed. This approach allowed for a clear and accurate representation of the polygons within the graph.

3.2. Checking for Node in Open-List:

A challenge encountered during the implementation of the A* algorithm was determining whether a neighboring node had already been added to the open list. This difficulty arose because the open list, implemented as a priority queue, stores both priority values and node coordinates, complicating direct checks for a node's presence.

To address this, we developed the contains_cell function, which unpacks the open list into priority values and node coordinates. Using the extracted coordinates, the function efficiently verifies whether the node of interest exists within the list.

4. Conclusion:

In this lab, the A* algorithm was implemented to determine the optimal path in different environments. The algorithm was applied to two distinct map types: a graph representation consisting of nodes and visibility edges, and a grid map containing various obstacles. In both scenarios, the implementation of the A* algorithm was successful, yielding the shortest path across all provided environments. This demonstrates the algorithm's efficacy in navigating diverse spatial configurations and optimizing pathfinding tasks.