جامـــــعـة المـــســتـقـبـل
AL MUSTAQBAL UNIVERSITY

كلية العلوم
قـســـــــــم الانــظـــمـــة الـــطـبـيـة الــذكـــيــة

Lab: (5)

A* (A-Star) algorithm in Python,

Subject: Artificial Intelligence

Class: Third

Lecturer: Dr. Maytham N. Meqdad

A* (A-Star) algorithm in Python, which is commonly used for path finding and graph traversal. In this example, the algorithm searches for the shortest path on a grid from a starting point to a goal point. The algorithm uses a heuristic to guide its search (typically the Manhattan distance or Euclidean distance in a grid).

```
import heapq
class Node:
    def __init__(self, position, parent=None):
        self.position = position  # (x, y) tuple
        self.parent = parent
        self.g = 0  # Cost from start to the node
        self.h = 0  # Heuristic (estimated cost to the goal)
        self.f = 0  # Total cost (g + h)

    def __lt__(self, other):
        return self.f < other.f

def heuristic(current, goal):
    # Manhattan distance as heuristic
    return abs(current[0] - goal[0]) + abs(current[1] - goal[1])

def a_star(grid, start, end):
    open_list = []
    closed_set = set()

    start_node = Node(start)
    end_node = Node(end)

    heapq.heappush(open_list, start_node)

    while open_list:
        # Get the node with the lowest f score
        current_node = heapq.heappop(open_list)
        closed_set.add(current_node.position)

        # Check if we have reached the goal
        if current_node.position == end_node.position:
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return reversed path

        # Generate neighbors (up, down, left, right)
        neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        for offset in neighbors:
            neighbor_position = (current_node.position[0] + offset[0],
                                 current_node.position[1] + offset[1])

            # Check if neighbor is within grid bounds and walkable
            if (0 <= neighbor_position[0] < len(grid) and
```

```
            0 <= neighbor_position[1] < len(grid[0]) and
            grid[neighbor_position[0]][neighbor_position[1]] == 0 and
            neighbor_position not in closed_set):

            neighbor_node = Node(neighbor_position, current_node)
            neighbor_node.g = current_node.g + 1
            neighbor_node.h = heuristic(neighbor_position, end_node.position)
            neighbor_node.f = neighbor_node.g + neighbor_node.h

            # Check if this path to neighbor is better
            if not any(open_node for open_node in open_list if open_node.position == neighbor_position and
open_node.g <= neighbor_node.g):
                heapq.heappush(open_list, neighbor_node)

    return None  # No path found

# Example grid (0 = walkable, 1 = obstacle)
grid = [
    [0, 1, 0, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 1, 0, 0],
    [0, 0, 0, 0, 0]
]

# Start and end points
start = (0, 0)
end = (4, 4)

# Find path
path = a_star(grid, start, end)
print("Path:", path)
```

Explanation of the Code:

- **Node Class**: Represents a node in the search with position, parent (for tracing the path), and scores (`g`, `h`, and `f`).
- **Heuristic Function**: Here, we use the Manhattan distance to estimate the cost from a node to the goal.
- *A Algorithm**:
    - `open_list` holds the nodes that need to be evaluated.
    - `closed_set` keeps track of evaluated nodes.
    - Nodes are generated for each neighbor (up, down, left, right).
    - If a better path to a neighbor is found, it's added to `open_list`.

Example Output

Given the grid, it will return the path from the start to the end if a path exists:

Path: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (3, 3), (4, 4)]

https://www.programiz.com/online-compiler/2apM9OVBXejE0