Artificial Intelligence for Big Data
Coursework 1

Mihael Zlatev
Ana Petrova

1.Introduction

In this coursework the task is to implement a solution to find a path from a start node to an end node in a maze. We decided to use A* algorithm because it is suitable for finding the shortest path between two points in a grid, considering obstacles and efficiently traversing the grid and it is complete.

2.Choose a suitable representation of the maze nodes using the available Python data structures

For the representation of the maze we implemented a class Maze, where we have information for its width and height. The maze node is represented as a tuple(row and col) withing the grid. The Maze class does not use a separate node class, but rather treats each cell as a coordinate in the grid. This simple representation is effective for grid-based algorithms where each cell only needs to store its position. Each cell's properties—whether it's passable or an obstacle—are stored in the grid attribute, a NumPy array where 1 denotes an obstacle and 0 denotes a passable cell.

3.Define the success criteria for reaching the end node

The success criterion is when the algorithm's current position matches the target end position. Although the success check is not explicitly written in this code, in a full pathfinding implementation, the criterion would be as follows:

```
  if current.position == end:
 # Reached the goal, reconstruct the path
```

This criterion terminates the search once the algorithm reaches the specified end node.

4. Define the test function, which accumulates all possible moves for each node

The function responsible for accumulating possible moves is get_neighbors. This function generates all valid neighboring cells by checking each direction (up, down, left, right) from the current position. If the neighboring cell is within the grid bounds and is not an obstacle, it's added to the neighbors list.

```
DIRECTIONS = [
    (0, 1),
    (1, 0),
    (0, -1),
    (-1, 0)
]
```

```
def get_neighbors(self, node):
        row, col = node
        neighbors = []

        for dx, dy in DIRECTIONS:
            new_row, new_col = row + dx, col + dy
            if self.is_valid_move(new_row, new_col):
                neighbors.append((new_row, new_col))

        return neighbors
```

This function allows the algorithm to explore all potential moves from each cell and forms the basis for pathfinding. The global variable DIRECTIONS is an array of tuples, which represents all the possible moves.

5. Define the Estimation Function to Calculate the Cost for Each Move

In the A* algorithm, the estimation (heuristic) function helps to estimate the cost to reach the goal from a given node. We decided to use the Manhattan distance as the heuristic, which is calculated as the sum of the absolute differences between the coordinates of the current node and the goal node.

The Manhattan distance is suitable for grid-based movement where movement is restricted to horizontal and vertical steps. It effectively guides the algorithm by giving an estimate of the remaining distance to the target.

```
def manhattan_distance(self, node_x1, node_x2) -> float:
        return abs(node_x1[0] - node_x2[0]) + abs(node_x1[1] - node_x2[1])
```

In the a_star function, this heuristic is combined with the cost_from_start to form the total_estimated_cost for each node:

```
class Node:
    total_estimated_cost: float
    cost_from_start: float = field(compare=False)
    position: Tuple[int, int] = field(compare=False)
    parent: Optional['Node'] = field(default=None, compare=False)

    def __eq__(maze, other):
        if isinstance(other, Node):
            return maze.position == other.position
        return False
```

Here:

•cost_from_start represents the actual cost to reach the neighbor node from the start.

•maze.manhattan_distance(neighbor_node, end) estimates the remaining cost to reach the goal.

This combined score (total_estimated_cost) prioritizes the nodes that are closer to the goal, helping the A* algorithm to find the optimal path more efficiently.

6.Define the path function, which accumulates the past nodes into a path.

The path function is accumulated, when the end node is found (current.position == end). The code enters a loop to reconstruct the path by following the parent links back to the start node.

```
if current.position == end:
                path = []
                while current:
                    path.append(current.position)
                    current = current.parent
                # Return the path in reverse order
                return path[::-1]
```

7.Define the cost function, which accumulates the total cost of the path.

The total cost for reaching each node is represented by cost_from_start, which tracks the cumulative cost from the start node to the current node. The variable is incremented by 1 for each move, assuming a constant move cost between neighboring nodes. We are using the cost functions as part of the A* algorithm to determine the optimal path, combining it with an estimated cost to the goal.

```
cost_from_start = current.cost_from_start + 1
```

8.Create an algorithm for search considering the success criteria.

We are using the A* search algorithm, with the following components in it:

- Two sets: to_be_visited_set (a min-heap priority queue for nodes that need to be explored) and visited_set (a set for nodes that have already been visited)
- A node is represented by its total_estimated_cost, calculated as the sum of cost_from_start , and the heuristic distance to the goal
- The algorithm selects the node with the lowest total_estimated_cost (most promising path) and evaluates each of its neighbors. For each neigbour
    - If the neighbor is already in the visited_set, it is skipped
    - The cost_from_start is calculated, and if this path is better than any previous path to the neighbor, the neighbor is added to the to_be_visited_set

9. Run the program and output the optimal path found by the algorithm.

In order to run the program, you need to have python installed on your computer. Also, there are libraries that needs to be installed, which can be found in /requirenments.txt/ file in the zip.

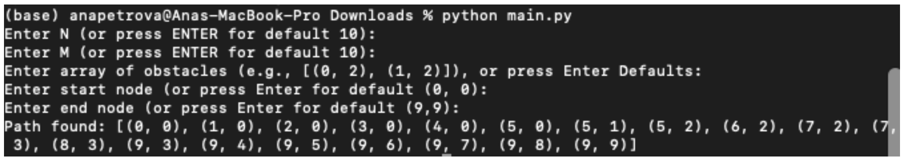
The input for the program is predefined in main.py file, but user input is also possible.

```
if __name__ == "__main__":
    n_input = input("Enter N (or press ENTER for default 10): ")
    m_input = input("Enter M (or press ENTER for default 10): ")
    n = int(n_input) if n_input else 10
    m = int(m_input) if m_input else 10
    obstacles_input = input("Enter array of obstacles (e.g., [(0, 2), (1, 2)]), or
press Enter Defaults: ")
    obstacles = ast.literal_eval(obstacles_input) if obstacles_input else [(0, 2), (1,
2), (2, 2), (2, 3), (3, 1),(4, 1), (4, 2), (4, 3), (5, 3), (6, 3)]
    start_node_input = input("Enter start node (or press Enter for default (0, 0): ")
    start_node = ast.literal_eval(start_node_input) if start_node_input else (0, 0)
    end_node_input = input("Enter end node (or press Enter for default (9,9): ")
    end_node = ast.literal_eval(end_node_input) if end_node_input else (9, 9)

    maze = Maze(n, m, obstacles)

    path = a_star(maze, start_node, end_node)
    if path:
        print(f"Path found: {path}")
    else:
        print("No path exists!")
```

The output is a list of tuples with the coordinates of the points(nodes) that are representing the optimal path found by the algorithm.



```
(base) anapetrova@Anas-MacBook-Pro Downloads % python main.py
Enter N (or press ENTER for default 10):
Enter M (or press ENTER for default 10):
Enter array of obstacles (e.g., [(0, 2), (1, 2)]), or press Enter Defaults:
Enter start node (or press Enter for default (0, 0):
Enter end node (or press Enter for default (9,9):
Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (5, 1), (5, 2), (6, 2), (7, 2), (7,
 3), (8, 3), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8), (9, 9)]
```