Week 04 A* Search Algorithm

Manhattan Distance

```
import heapq


class Node:
    def __init__(self, position, parent=None):
        self.position = position
        self.parent = parent
        self.g = 0  # Cost from start to this node
        self.h = 0  # Heuristic cost from this node to target
        self.f = 0  # Total cost

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


def heuristic(a, b):
    # Manhattan distance
    return abs(a[0] - b[0]) + abs(a[1] - b[1])


def astar(start, goal, grid):
    open_list = []
    closed_list = set()

    start_node = Node(start)
    goal_node = Node(goal)

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)
```

```
closed_list.add(current_node.position)

# Goal check
if current_node.position == goal:
    path = []
    while current_node:
        path.append(current_node.position)
        current_node = current_node.parent
    return path[::-1]  # Return reversed path

# Generate neighbors
neighbors = [
    (current_node.position[0] + dx, current_node.position[1] + dy)
    for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]
]

for next_position in neighbors:
    # Check if within bounds and not a wall (assuming 0 is free space)
    if (0 <= next_position[0] < len(grid) and
        0 <= next_position[1] < len(grid[0]) and
        grid[next_position[0]][next_position[1]] == 0):

        if next_position in closed_list:
            continue

        neighbor_node = Node(next_position, current_node)
        neighbor_node.g = current_node.g + 1
        neighbor_node.h = heuristic(next_position, goal)
        neighbor_node.f = neighbor_node.g + neighbor_node.h
```

```
            # Check if this neighbor is already in the open list
            if any(neighbor.position == neighbor_node.position and neighbor.f <=
neighbor_node.f for neighbor in open_list):
                continue

            heapq.heappush(open_list, neighbor_node)

    return []  # Return empty path if no path found

# Example usage
if __name__ == "__main__":
    grid = [
        [0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0],
        [0, 0, 0, 0, 0],
        [0, 1, 1, 0, 0],
        [0, 0, 0, 0, 0]
    ]

    start = (0, 0)
    goal = (4, 4)
    path = astar(start, goal, grid)

    print("Path from start to goal:", path)
    print("Suvina A Shetty")
    print("1BM22CS299")
```

OUTPUT

```
Output                                                    Clear
Path from start to goal: [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2), (2, 3), (3
    , 3), (4, 3), (4, 4)]
Suvina A Shetty
1BM22CS299

=== Code Execution Successful ===
```

Misplaced Tiles

```
import heapq
class PuzzleState:
    def __init__(self, board, g=0):
        self.board = board
        self.g = g  # Cost from start to this state
        self.zero_pos = board.index(0)  # Position of the empty space

    def h(self):
        # Calculate the number of misplaced tiles
        return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i] != i + 1)

    def f(self):
        return self.g + self.h()

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, 3)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_zero_pos = new_x * 3 + new_y
                new_board = self.board[:]
                # Swap zero with the neighboring tile
                new_board[self.zero_pos], new_board[new_zero_pos] =
new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(new_board, self.g + 1))
        return neighbors

def a_star(initial_state, goal_state):
```

```
    open_set = []
    heapq.heappush(open_set, (initial_state.f(), 0, initial_state))  # Add a unique identifier (0 in this case)
    came_from = {}
    g_score = {tuple(initial_state.board): 0}

    while open_set:
        current_f, _, current = heapq.heappop(open_set)
        if current.board == goal_state:
            return reconstruct_path(came_from, current)

        for neighbor in current.get_neighbors():
            neighbor_tuple = tuple(neighbor.board)
            tentative_g_score = g_score[tuple(current.board)] + 1
            if neighbor_tuple not in g_score or tentative_g_score < g_score[neighbor_tuple]:
                came_from[neighbor_tuple] = current
                g_score[neighbor_tuple] = tentative_g_score
                heapq.heappush(open_set, (neighbor.f(), neighbor.g, neighbor))

    return None

def reconstruct_path(came_from, current):
    path = []
    while current is not None:
        path.append(current.board)
        current = came_from.get(tuple(current.board), None)
    return path[::-1]

# Example usage
initial_state = PuzzleState([1, 2, 3, 4, 5, 6, 0, 7, 8])
goal_state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
```

```
solution = a_star(initial_state, goal_state)

if solution:
    for step in solution:
        print(step)
else:
    print("No solution found")

print("SUVINA A SHETTY")
print("1BM22CS299")
```

OUTPUT

```
Output
[1, 2, 3, 4, 5, 6, 0, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 0, 8]
[1, 2, 3, 4, 5, 6, 7, 8, 0]
SUVINA A SHETTY
1BM22CS299

=== Code Execution Successful ===
```