1BM22CS241 (LAB 4)

(1) 8-Puzzle : A* Implementation using Misplaced Tiles approach

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost

    def heuristic(self):
        goal_state = [[1,2,3], [8,0,4], [7,6,5]]
        count = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                if self.state[i][j] != 0 and self.state[i][j] !=
goal_state[i][j]:
                    count += 1
        return count

def get_blank_position(state):
    for i in range(len(state)):
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]],
new_state[new_blank_pos[0]][new_blank_pos[1]] = \
        new_state[new_blank_pos[0]][new_blank_pos[1]],
new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
```

```
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)

    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else
"Start"
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (misplaced tiles):
{current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state,
blank_pos, (new_blank_pos[0], new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node,
move=move_direction, cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
```

```
solution_path = a_star_search(initial_state)

if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:

```
Current state:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Start
Heuristic value (misplaced tiles): 4
Cost to reach this node: 0

Current state:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Down
Heuristic value (misplaced tiles): 3
Cost to reach this node: 1

Current state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
Heuristic value (misplaced tiles): 3
Cost to reach this node: 2
```

```
Solution path:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

(2) 8-Puzzle : A* Implementation using Manhattan distance approach

```
class Node:
    def __init__(self, state, parent=None, move=None, cost=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.cost = cost

    def heuristic(self):
        goal_positions = {
            1: (0, 0), 2: (0, 1), 3: (0, 2),
            8: (1, 0), 0: (1, 1), 4: (1, 2),
            7: (2, 0), 6: (2, 1), 5: (2, 2)
        }
        manhattan_distance = 0
        for i in range(len(self.state)):
            for j in range(len(self.state[i])):
                value = self.state[i][j]
                if value != 0:
                    goal_i, goal_j = goal_positions[value]
                    manhattan_distance += abs(i - goal_i) + abs(j -
goal_j)
        return manhattan_distance

def get_blank_position(state):
    for i in range(len(state)):
```

```
        for j in range(len(state[i])):
            if state[i][j] == 0:
                return i, j

def get_possible_moves(position):
    x, y = position
    moves = []
    if x > 0: moves.append((x - 1, y, 'Down'))
    if x < 2: moves.append((x + 1, y, 'Up'))
    if y > 0: moves.append((x, y - 1, 'Right'))
    if y < 2: moves.append((x, y + 1, 'Left'))
    return moves

def generate_new_state(state, blank_pos, new_blank_pos):
    new_state = [row[:] for row in state]
    new_state[blank_pos[0]][blank_pos[1]],
new_state[new_blank_pos[0]][new_blank_pos[1]] = \
        new_state[new_blank_pos[0]][new_blank_pos[1]],
new_state[blank_pos[0]][blank_pos[1]]
    return new_state

def a_star_search(initial_state):
    open_list = []
    closed_list = set()

    initial_node = Node(state=initial_state, cost=0)
    open_list.append(initial_node)

    while open_list:

        open_list.sort(key=lambda node: node.cost + node.heuristic())
        current_node = open_list.pop(0)


        move_description = current_node.move if current_node.move else
"Start"
        print("Current state:")
        for row in current_node.state:
            print(row)
        print(f"Move: {move_description}")
        print(f"Heuristic value (Manhattan distance):
{current_node.heuristic()}")
        print(f"Cost to reach this node: {current_node.cost}\n")

        if current_node.heuristic() == 0:

            path = []
            while current_node:
                path.append(current_node)
```

```
                current_node = current_node.parent
            return path[::-1]
        closed_list.add(tuple(map(tuple, current_node.state)))

        blank_pos = get_blank_position(current_node.state)
        for new_blank_pos in get_possible_moves(blank_pos):
            new_state = generate_new_state(current_node.state,
blank_pos, (new_blank_pos[0], new_blank_pos[1]))

            if tuple(map(tuple, new_state)) in closed_list:
                continue

            cost = current_node.cost + 1
            move_direction = new_blank_pos[2]
            new_node = Node(state=new_state, parent=current_node,
move=move_direction, cost=cost)

            if new_node not in open_list:
                open_list.append(new_node)

    return None

initial_state = [[2,8,3], [1,6,4], [7,0,5]]
solution_path = a_star_search(initial_state)

if solution_path:
    print("Solution path:")
    for step in solution_path:
        for row in step.state:
            print(row)
        print()
else:
    print("No solution found.")
```

Output:

```
Current state:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]
Move: Start
Heuristic value (Manhattan distance): 5
Cost to reach this node: 0

Current state:
[2, 8, 3]
[1, 0, 4]
[7, 6, 5]
Move: Down
Heuristic value (Manhattan distance): 4
Cost to reach this node: 1

Current state:
[2, 0, 3]
[1, 8, 4]
[7, 6, 5]
Move: Down
Heuristic value (Manhattan distance): 3
Cost to reach this node: 2

Current state:
[0, 2, 3]
[1, 8, 4]
[7, 6, 5]
Move: Right
Heuristic value (Manhattan distance): 2
Cost to reach this node: 3
```

```
Solution path:
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```