K.L.VARSHITA
AP21110010178

Implement A* algorithm to solve 8-Puzzle game (You can use any heuristic and g(n) is the level number of the state). You can start any random state and the final state is fixed whis was discussed in the previous class.

CODE:

```
import heapq

class PuzzleNode:
    def __init__(self, state, parent=None, move=None, level=0):
        self.state = state
        self.parent = parent
        self.move = move
        self.level = level
        self.cost = self.compute_cost()

    def compute_cost(self):
        cost = self.level
        for i in range(3):
            for j in range(3):
                if self.state[i][j] != 0:
                    x, y = divmod(self.state[i][j] - 1, 3)
                    cost += abs(i - x) + abs(j - y)
        return cost

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

# Rest of your code...


def is_valid_position(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def generate_neighbors(node):
    x, y = None, None
    for i in range(3):
        for j in range(3):
```

```
            if node.state[i][j] == 0:
                x, y = i, j
                break

    neighbors = []
    moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if is_valid_position(new_x, new_y):
            new_state = [list(row) for row in node.state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y], new_state[x][y]
            neighbors.append(PuzzleNode(new_state, node, (dx, dy), node.level + 1))
    return neighbors

def solve_puzzle_astar(initial_state, goal_state):
    open_list = []
    closed_set = set()

    initial_node = PuzzleNode(initial_state)
    goal_node = PuzzleNode(goal_state)

    heapq.heappush(open_list, initial_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node.state == goal_node.state:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return list(reversed(path))

        closed_set.add(tuple(map(tuple, current_node.state)))

        for neighbor in generate_neighbors(current_node):
            if tuple(map(tuple, neighbor.state)) not in closed_set:
                heapq.heappush(open_list, neighbor)

    return None

if __name__ == "__main__":
    initial_state = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
```

```
solution_path = solve_puzzle_astar(initial_state, goal_state)
if solution_path:
    for state in solution_path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found.")
```

OUTPUT :

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

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

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

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

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

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

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

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

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

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

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

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

```
[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]
```