**Exp: 7: Solve and Implement 8-puzzle problem using A* Algorithm**

**Program:**

```
import heapq
import itertools

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

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

    def __eq__(self, other):
        return self.state == other.state

    def __hash__(self):
        return hash(str(self.state))

    def get_blank_position(self):
        for i, row in enumerate(self.state):
            for j, cell in enumerate(row):
                if cell == 0:
                    return (i, j)

    def generate_children(self):
        blank_i, blank_j = self.get_blank_position()
        children = []
        for di, dj in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            new_i, new_j = blank_i + di, blank_j + dj
            if 0 <= new_i < len(self.state) and 0 <= new_j <
len(self.state[0]):
                new_state = [row[:] for row in self.state]
                new_state[blank_i][blank_j], new_state[new_i][new_j] =
new_state[new_i][new_j], new_state[blank_i][blank_j]
                children.append(PuzzleNode(new_state, parent=self,
move=(di, dj), depth=self.depth + 1))
        return children

    def manhattan_distance(self, goal_state):
        distance = 0
        for i in range(len(self.state)):
```

```
            for j in range(len(self.state[0])):
                if self.state[i][j] != goal_state[i][j] and
self.state[i][j] != 0:
                    x, y = divmod(goal_state[i][j], len(self.state[0]))
                    distance += abs(x - i) + abs(y - j)
        return distance

def solve_puzzle(initial_state, goal_state):
    open_set = []
    heapq.heappush(open_set, initial_state)
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)
        closed_set.add(current_node)

        if current_node.state == goal_state:
            path = []
            while current_node.parent:
                path.append(current_node.move)
                current_node = current_node.parent
            return path[::-1]

        for child in current_node.generate_children():
            if child not in closed_set:
                child.cost = child.depth
                child.heuristic = child.manhattan_distance(goal_state)
                heapq.heappush(open_set, child)

    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]]

    path = solve_puzzle(PuzzleNode(initial_state), goal_state)

    if path:
        print("Moves to reach the goal state:")
        for move in path:
            print(move)
    else:
        print("No solution exists.")
```

**Exp: 8: Implementation of Traveling Salesman Problem using heuristic search**

**Program:**

```
import math

def distance(city1, city2):
    """
    Calculate the Euclidean distance between two cities.
    """
    x1, y1 = city1
    x2, y2 = city2
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)

def nearest_neighbor(start, unvisited_cities):
    """
    Find the nearest unvisited city to the given city.
    """
    min_distance = float('inf')
    nearest_city = None
    for city in unvisited_cities:
        dist = distance(start, city)
        if dist < min_distance:
            min_distance = dist
            nearest_city = city
    return nearest_city, min_distance

def tsp_nearest_neighbor(cities):
    """
    Solve the Traveling Salesman Problem using the Nearest Neighbor
Algorithm.
    """
    current_city = cities[0]
    unvisited_cities = set(cities[1:])
    tour = [current_city]

    while unvisited_cities:
        next_city, distance_to_next = nearest_neighbor(current_city,
unvisited_cities)
        tour.append(next_city)
        unvisited_cities.remove(next_city)
        current_city = next_city

    tour.append(tour[0])  # Return to the starting city to complete the
tour
    total_distance = sum(distance(tour[i], tour[i+1]) for i in
range(len(tour)-1))
    return tour, total_distance
```

```
if __name__ == "__main__":
    # Example cities represented as (x, y) coordinates
    cities = [(0, 0), (1, 2), (3, 1), (5, 3), (4, 0)]

    tour, total_distance = tsp_nearest_neighbor(cities)

    print("Optimal Tour:", tour)
    print("Total Distance:", total_distance)
```