Assignment – 2

Submission by : Mrituanjay Jha – 102103807 – 2CO18

Q.1)

```
from queue import PriorityQueue
initial_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
final_state = [[2, 8, 1], [0, 4, 3], [7, 6, 5]]

rows = len(initial_state)
cols = len(initial_state[0])

# Define the Manhattan distance heuristic function
def manhattan_distance(state):
    distance = 0
    for i in range(rows):
        for j in range(cols):
            value = state[i][j]
            if value != 0:
                target_row = (value - 1) // rows
                target_col = (value - 1) % cols
                distance += abs(i - target_row) + abs(j - target_col)
    return distance

# Define the Node class for the A* algorithm
class Node:
    def __init__(self, state, g, h, parent=None):
        self.state = state
        self.g = g
        self.h = h
        self.parent = parent

    def __lt__(self, other):
        return (self.g + self.h) < (other.g + other.h)

# Define the A* algorithm function
def a_star(initial_state, final_state):
    start_node = Node(initial_state, 0, manhattan_distance(initial_state))
    visited = set()
    queue = PriorityQueue()
    queue.put(start_node)
```

```
    while not queue.empty():
        current_node = queue.get()
        current_state = current_node.state

        if current_state == final_state:
            path = []
            while current_node.parent:
                path.append(current_node.state)
                current_node = current_node.parent
            path.append(initial_state)
            return path[::-1]

        visited.add(str(current_state))

        for move, (i, j) in [('UP', (-1, 0)), ('DOWN', (1, 0)), ('LEFT', (0, -1)), ('RIGHT', (0, 1))]:
            new_state = [row[:] for row in current_state]
            row, col = empty_square = next((r, c) for r in range(rows) for c in range(cols) if new_state[r][c] == 0)
            new_row, new_col = row + i, col + j

            if 0 <= new_row < rows and 0 <= new_col < cols:
                new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col], new_state[row][col]

                if str(new_state) not in visited:
                    g = current_node.g + 1
                    h = manhattan_distance(new_state)
                    new_node = Node(new_state, g, h, current_node)
                    queue.put(new_node)

    return None

# Test the A* algorithm with the initial and final states
solution = a_star(initial_state, final_state)
if solution:
    print("Solution found with", len(solution) - 1, "moves:")
    for state in solution:
        print(state)
else:
    print("No solution found.")
```

```
Solution found with 9 moves:
[[1, 2, 3], [8, 0, 4], [7, 6, 5]]
[[1, 0, 3], [8, 2, 4], [7, 6, 5]]
[[0, 1, 3], [8, 2, 4], [7, 6, 5]]
[[8, 1, 3], [0, 2, 4], [7, 6, 5]]
[[8, 1, 3], [2, 0, 4], [7, 6, 5]]
[[8, 1, 3], [2, 4, 0], [7, 6, 5]]
[[8, 1, 0], [2, 4, 3], [7, 6, 5]]
[[8, 0, 1], [2, 4, 3], [7, 6, 5]]
[[0, 8, 1], [2, 4, 3], [7, 6, 5]]
[[2, 8, 1], [0, 4, 3], [7, 6, 5]]
```

Q.2)

```python
# Define the capacities of the jugs
jug_3_capacity = 3
jug_4_capacity = 4

# Define a function to print the state of the jugs
def print_jugs(jug_3, jug_4):
    print("Jug 3:", jug_3, "liters")
    print("Jug 4:", jug_4, "liters")
    print()

# Define a function to simulate pouring water from one jug to another
def pour(from_jug, to_jug, to_jug_capacity):
    if from_jug == 0 or to_jug == to_jug_capacity:
        return from_jug, to_jug

    space_available = to_jug_capacity - to_jug
    amount_to_pour = min(from_jug, space_available)

    from_jug -= amount_to_pour
    to_jug += amount_to_pour

    return from_jug, to_jug

# Define a function to check if the goal state has been reached
def is_goal(jug_3, jug_4):
    return jug_4 == 2

# Define a function to perform a depth-first search of all possible actions
def dfs(jug_3, jug_4, visited):
    if is_goal(jug_3, jug_4):
        return True

    visited.add((jug_3, jug_4))
```

```python
 for action in [('pour_3_into_4', jug_3, jug_4), ('pour_4_into_3', jug_4, jug_3),
('empty_3', 0, jug_4), ('empty_4', jug_3, 0), ('fill_3', jug_3_capacity, jug_4),
('fill_4', jug_3, jug_4_capacity)]:
```

```python
        action_name, new_jug_3, new_jug_4 = action
        if (new_jug_3, new_jug_4) not in visited:
            print("Action:", action_name)
            print_jugs(new_jug_3, new_jug_4)
            if dfs(new_jug_3, new_jug_4, visited):
                return True

    return False

# Perform the search to get 2 liters in the 4-liter jug
dfs(jug_3_capacity, 0, set())
```

Q.3)

```python
import itertools

# Define a function to calculate the distance between two cities
def distance(city1, city2):
    x1, y1 = city1
    x2, y2 = city2
    return ((x1 - x2) ** 2 + (y1 - y2) ** 2) ** 0.5

# Define a function to calculate the total distance of a tour
def tour_distance(tour, cities):
    return sum(distance(cities[tour[i]], cities[tour[i + 1]]) for i in range(len(tour) - 1)) + distance(cities[tour[-1]], cities[tour[0]])

# Get the number of cities and their coordinates from the user
n = int(input("Enter the number of cities: "))
cities = []
for i in range(n):
    x, y = map(int, input("Enter the coordinates of city {}: ".format(i + 1)).split())
    cities.append((x, y))

# Get the starting city from the user
start = int(input("Enter the starting city (1-{}): ".format(n)))

# Generate all possible tours and calculate their distances
tours = itertools.permutations(range(n))
shortest_tour = None
shortest_distance = float('inf')
for tour in tours:
    if tour[0] != start - 1:
        continue
    tour_distance = tour_distance(tour, cities)
    if tour_distance < shortest_distance:
        shortest_tour = tour
        shortest_distance = tour_distance

# Print the shortest tour and its distance
print("Shortest tour:", " -> ".join(str(city + 1) for city in shortest_tour))
print("Shortest distance:", shortest_distance)
```