Ankushdeep Singh_102003174_COE7
ASSIGNMENT -2

ARTIFICIAL INTELLIGENCE

1Q:

Write a code in python for the 8 puzzle problem by taking the following initial and final states

Initial State

1	2	3
8		4
7	6	5

Goal State

2	8	1
	4	3
7	6	5

CODE:

```
ques_1.py > ...
from collections import deque

# Define the initial and final states
initial_state = [[1, 2, 3], [8, 0, 4], [7, 6, 5]]
final_state = [[2, 8, 1], [0, 4, 3], [7, 6, 5]]

# Function to check if two states are equal
def is_equal(state1, state2):
    for i in range(3):
        for j in range(3):
            if state1[i][j] != state2[i][j]:
                return False
    return True

# Function to generate the possible next states from the current state
def generate_next_states(state):
    next_states = []
    zero_pos = None
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                zero_pos = (i, j)
                break

    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Possible moves: right, left, down, up

    for move in moves:
        new_state = [row[:] for row in state]  # Create a copy of the current state

        new_row = zero_pos[0] + move[0]
        new_col = zero_pos[1] + move[1]

        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state[zero_pos[0]][zero_pos[1]] = new_state[new_row][new_col]
            new_state[new_row][new_col] = 0
            next_states.append(new_state)
```

```python

    return next_states

# Breadth-first search algorithm to find the optimal solution
def bfs(initial_state, final_state):
    visited = set()
    queue = deque([(initial_state, [])])

    while queue:
        current_state, path = queue.popleft()
        visited.add(tuple(map(tuple, current_state)))

        if is_equal(current_state, final_state):
            return path

        next_states = generate_next_states(current_state)
        for next_state in next_states:
            if tuple(map(tuple, next_state)) not in visited:
                queue.append((next_state, path + [next_state]))

    return None

# Solve the 8-puzzle problem
solution = bfs(initial_state, final_state)

# Print the solution
if solution is None:
    print("No solution found.")
else:
    print("Solution:")
    for step, state in enumerate(solution):
        print(f"Step {step + 1}:")
        for row in state:
            print(row)
        print()
```

OUTPUT:

```
PS E:\SUMMER SEM\AI\LABS\ASS_2> python -u "e:\SUMMER SEM\AI\LABS\ASS_2\q
ues_1.py"
Solution:
Step 1:
[1, 0, 3]
[8, 2, 4]
[7, 6, 5]

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

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

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

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

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

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

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

Step 9:
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]
```

2Q:

Given two jugs- a 4 liter and 3 liter capacity. Neither has any measurable markers on it. There is a pump which can be used to fill the jugs with water. Simulate the procedure in Python to get exactly 2 liter of water into 4-liter jug

CODE:

ques_2.py > ...

```python
from collections import deque

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

# Function to check if the desired amount is reached
def is_desired_amount(amount):
    return amount == 2

# Function to generate the possible next states
def generate_next_states(state):
    next_states = []

    # Fill the 4-liter jug
    if state[0] < jug_4_capacity:
        next_states.append((jug_4_capacity, state[1]))

    # Fill the 3-liter jug
    if state[1] < jug_3_capacity:
        next_states.append((state[0], jug_3_capacity))

    # Empty the 4-liter jug
    if state[0] > 0:
        next_states.append((0, state[1]))

    # Empty the 3-liter jug
    if state[1] > 0:
        next_states.append((state[0], 0))

    # Pour water from the 4-liter jug to the 3-liter jug
    if state[0] > 0 and state[1] < jug_3_capacity:
        amount_to_pour = min(state[0], jug_3_capacity - state[1])
        next_states.append((state[0] - amount_to_pour, state[1] + amount_t

    # Pour water from the 3-liter jug to the 4-liter jug
    if state[0] < jug_4_capacity and state[1] > 0:
        amount_to_pour = min(jug_4_capacity - state[0], state[1])
        next_states.append((state[0] + amount_to_pour, state[1] - amount_t

    return next_states
```

```python

# Breadth-first search algorithm to find the solution
def bfs(initial_state):
    visited = set()
    queue = deque([(initial_state, [])])

    while queue:
        current_state, path = queue.popleft()
        visited.add(current_state)

        if is_desired_amount(current_state[0]):
            return path

        next_states = generate_next_states(current_state)
        for next_state in next_states:
            if next_state not in visited:
                queue.append((next_state, path + [next_state]))

    return None

# Solve the jug problem
initial_state = (0, 0)  # Initial state: both jugs are empty
solution = bfs(initial_state)

# Print the solution
if solution is None:
    print("No solution found.")
else:
    print("Solution:")
    for step, state in enumerate(solution):
        print(f"Step {step + 1}: {state[0]}L, {state[1]}L")
```

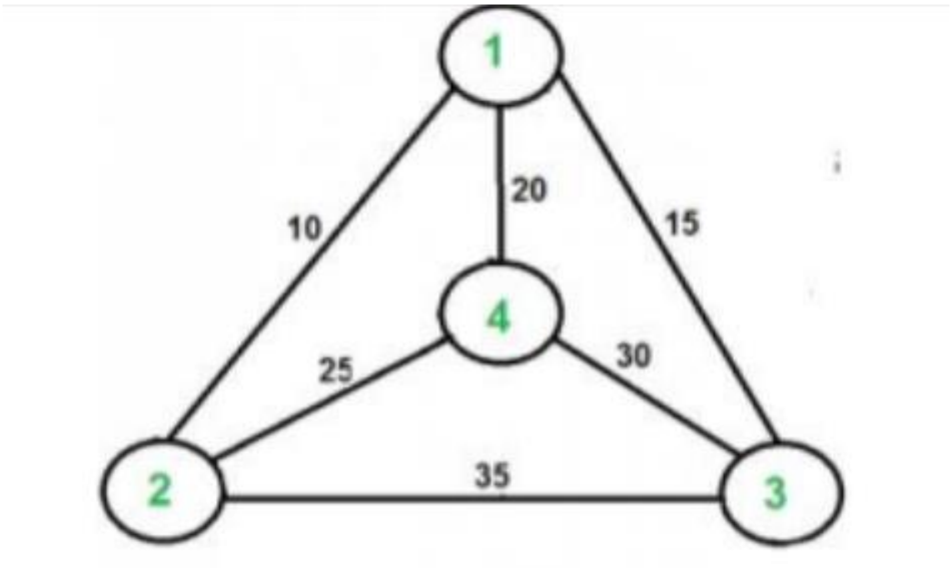
OUTPUT:

```
PS E:\SUMMER SEM\AI\LABS\ASS_2> python -u "e:\SUMMER SEM\AI\LABS\ASS_2\ques_2.py"

Solution:
Step 1: 4L, 0L
Step 2: 1L, 3L
Step 3: 1L, 0L
Step 4: 0L, 1L
Step 5: 4L, 1L
Step 6: 2L, 3L
```

3Q:

Write a Python program to implement Travelling Salesman Problem (TSP). Take the starting node from the user at run time.



CODE:

ques_3.py > ...

```python
import copy

class TSP():
    def __init__(self, map, startCity):
        TSP.map = map
        self.startCity = startCity
        self.currentCity = startCity
        self.visitedList = []
        self.visitedList.append(self.currentCity)
        self.cost = 0
        self.prevState = None

    def isGoalReached(self):
        if len(self.visitedList) == len(TSP.map[0]) + 1:
            return True
        else:
            return False

    def __gt__(self, other):
        return self.cost > other.cost

    def move(self, city):
        if TSP.map[self.currentCity][city] != 0 and city not in self.visitedList:
            self.prevState = copy.deepcopy(self)
            self.cost = self.cost + TSP.map[self.currentCity][city]
            self.currentCity = city
            self.visitedList.append(self.currentCity)
            return True
        elif len(self.visitedList) == len(TSP.map[0]) and TSP.map[self.currentCity][city] != 0:
            self.prevState = copy.deepcopy(self)
            self.cost = self.cost + TSP.map[self.currentCity][city]
            self.currentCity = city
            self.visitedList.append(self.currentCity)
            return True
        else:
            return False
```

```python

    def possibleNextStates(self):
        stateList = []

        for i in range(0, len(TSP.map[0])):
            stateCopy = copy.deepcopy(self)
            if stateCopy.move(i):
                stateList.append(stateCopy)

        return stateList

def constructGoalPath(goalState):
    path = []
    while goalState:
        path.append(goalState.currentCity)
        goalState = goalState.prevState
    path.reverse()
    return path

open = []
closed = []

def UCS(startState):
    open.append(startState)

    while open:
        thisState = open.pop(0)

        closed.append(thisState)

        if thisState.isGoalReached():
            return constructGoalPath(thisState)
```

```python

        nextStates = thisState.possibleNextStates()
        for eachState in nextStates:
            if eachState not in open and eachState not in closed:
                open.append(eachState)
                open.sort()
            elif eachState in open:
                index = open.index(eachState)
                if eachState < open[index]:
                    open.append(eachState)
                    open.sort()
            elif eachState in closed:
                index = closed.index(eachState)
                if eachState < closed[index]:
                    closed.append(eachState)
                    closed.sort()

# Get the number of cities from the user
num_cities = int(input("Enter the number of cities: "))

# Get the distances between cities from the user
map = []
for i in range(num_cities):
    row = []
    for j in range(num_cities):
        if i == j:
            row.append(0)
        else:
            dist = float(input(f"Enter the distance between city {i+1} and city {j+1}: "))
            row.append(dist)
    map.append(row)
```

```python

startCity = int(input("Enter the starting city (1 to N): ")) - 1
problem = TSP(map, startCity)
solution = UCS(problem)

# Print the optimal path
print("Optimal Path:")
for city in solution:
    print(f"City {city+1}")
```

OUTPUT:

```
PS E:\SUMMER SEM\AI\LABS\ASS_2> python -u "e:\SUMMER SEM\AI\LABS\ASS_2\ques_3.py"

Enter the number of cities: 4
Enter the distance between city 1 and city 2: 10
Enter the distance between city 1 and city 3: 15
Enter the distance between city 1 and city 4: 20
Enter the distance between city 2 and city 1: 10
Enter the distance between city 2 and city 3: 35
Enter the distance between city 2 and city 4: 25
Enter the distance between city 3 and city 1: 15
Enter the distance between city 3 and city 2: 35
Enter the distance between city 3 and city 4: 30
Enter the distance between city 4 and city 1: 20
Enter the distance between city 4 and city 2: 25
Enter the distance between city 4 and city 3: 30
Enter the starting city (1 to N): 1
Optimal Path:
City 1
City 2
City 4
City 3
City 1
```