*Day-2 Experiments*

**7 Write the python program to implement BFS.**

**Program:**

```
from collections import deque
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    print("BFS Traversal:", end=" ")
    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E', 'F'],
    'D': [],
    'E': [],
    'F': []
}
bfs(graph, 'A')
```

**Sample output:**

```
All rooms are clean!

==================================================== RI
BFS Traversal: A B C D E F
```

**8 Write the python program to implement DFS.**

**Program:**

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)
graph = {
    'A': ['D', 'E', 'C'],
    'B': [],
    'C': ['B'],
    'D': ['E'],
    'E': []
}
start_node = 'A'
print("Depth-First Search traversal:")
dfs(graph, start_node)
```

**Sample output:**

```
==================================================== RI
BFS Traversal: A B C D E F

==================================================== RI
Depth-First Search traversal:
A D E C B
```

**9 Write the python to implement Travelling Salesman Problem**

**Program:**

```
from itertools import permutations
distance_matrix = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
```

```
    [20, 25, 30, 0]
]
def calculate_distance(route):
    total_distance = 0
    for i in range(len(route) - 1):
        total_distance += distance_matrix[route[i]][route[i + 1]]
    total_distance += distance_matrix[route[-1]][route[0]]
    return total_distance
def find_shortest_route():
    num_cities = len(distance_matrix)
    all_routes = permutations(range(num_cities))
    shortest_distance = float('inf')
    best_route = None
    for route in all_routes:
        current_distance = calculate_distance(route)
        if current_distance < shortest_distance:
            shortest_distance = current_distance
            best_route = route
    return best_route, shortest_distance
route, distance = find_shortest_route()
print("Shortest route:", route)
print("Minimum distance:", distance)
```

**Sample output:**

```
==========================================
Shortest route: (0, 1, 3, 2)
Minimum distance: 80
```

**10 Write the python program to implement A* algorithm**

**Program:**

```
import heapq
graph = {'A': [('B', 1), ('C', 4)], 'B': [('D', 2), ('E', 5)], 'C': [('E', 1)], 'D': [], 'E': []}
heuristics = {'A': 7, 'B': 6, 'C': 2, 'D': 1, 'E': 0}
def a_star(start, goal):
    queue, visited = [(0, start, [start])], set()
    while queue:
        cost, node, path = heapq.heappop(queue)
        if node == goal: return path, cost
        if node in visited: continue
        visited.add(node)
        for neighbor, travel_cost in graph[node]:
            heapq.heappush(queue, (cost + travel_cost + heuristics[neighbor], neighbor, path + [neighbor]))
path, cost = a_star('A', 'E')
print("Path:", " -> ".join(path) if path else "No path found.", "\nCost:", cost)
```

**Sample output:**

```
===================================================== RESTART:
Path: A -> C -> E
Cost: 7
```

**11 Write the python program for Map Coloring to implement CSP.**

**Program:**

```
regions = {
    "A": ["B", "C"],
    "B": ["A", "C", "D"],
    "C": ["A", "B", "D", "E"],
    "D": ["B", "C", "E"],
    "E": ["C", "D"]
}
colors = ["Red", "Green", "Blue"]
def assign_colors(region, color_map):
    if region == len(regions):
        return True
```

```
    region_name = list(regions.keys())[region]
    for color in colors:
        if all(color_map.get(neighbor) != color for neighbor in regions[region_name]):
            color_map[region_name] = color
            if assign_colors(region + 1, color_map):
                return True
            color_map.pop(region_name)
    return False
color_map = {}
assign_colors(0, color_map)
print("Color Assignments:", color_map)
```

**Sample output:**

```
================================================= RESTART: C:/Users/Admin/Desktop/AIF
Color Assignments: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red', 'E': 'Green'}

================================================= RESTART: C:/Users/Admin/Desktop/AIF
Color Assignments: {'A': 'Red', 'B': 'Green', 'C': 'Blue', 'D': 'Red', 'E': 'Green'}
|
```

**12 Write the python program for Tic Tac Toe game**

**Program:**

```
board = [
    ['X', 'O', 'X'],
    ['O', ' ', ' '],
    [' ', 'X ', 'O']
]
def evaluate(board):
    win_positions = [
        [(0,0), (0,1), (0,2)], [(1,0), (1,1), (1,2)], [(2,0), (2,1), (2,2)], # Rows
        [(0,0), (1,0), (2,0)], [(0,1), (1,1), (2,1)], [(0,2), (1,2), (2,2)], # Columns
        [(0,0), (1,1), (2,2)], [(0,2), (1,1), (2,0)]                        # Diagonals
    ]
    for positions in win_positions:
        values = [board[x][y] for x, y in positions]
        if values == ['O'] * 3: return 1
```

```
        if values == ['X'] * 3: return -1
    return 0
def minimax(board, is_max):
    score = evaluate(board)
    if score != 0 or not any(' ' in row for row in board):
        return score
    best = -float('inf') if is_max else float('inf')
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'O' if is_max else 'X'
                value = minimax(board, not is_max)
                board[i][j] = ' '
                best = max(best, value) if is_max else min(best, value)
    return best
best_move = None
best_val = -float('inf')
for i in range(3):
    for j in range(3):
        if board[i][j] == ' ':
            board[i][j] = 'O'
            move_val = minimax(board, False)
            board[i][j] = ' '
            if move_val > best_val:
                best_val, best_move = move_val, (i, j)
print("Best Move for O:", best_move)
print("Utility Value for the Move:", best_val)
```

**Sample output:**

```
===================================================== RESTART: C:
Best Move for O: (1, 1)
Utility Value for the Move: 0
```