Artificial Intelligence

Lab 07 Tasks

Name: Hajira Imran

Sap ID: 44594

Batch: BSCS-6th semester

Lab Instructor:

Ayesha Akram

Task1.

Solution:

```python
# Graph as an adjacency list
graph = {
    "A": ["B", "C", "H"],
    "B": ["A"],
    "C": ["A", "D"],
    "D": ["C", "E", "F"],
    "E": ["D", "G", "H"],
    "F": ["D", "G"],
    "G": ["E", "F"],
    "H": ["A", "E"]
}

# Function to find the shortest path using BFS
def find_shortest_path(graph, start, goal):
    queue = [[start]]  # Queue stores paths
    visited = set()  # Track visited nodes

    while queue:
        path = queue.pop(0)  # Get first path from queue
        node = path[-1]  # Last node in path

        if node == goal:
            return path  # Return the shortest path
```

```python
        if node == goal:
            return path  # Return the shortest path

        if node not in visited:
            visited.add(node)  # Mark as visited
            for neighbor in graph[node]:
                queue.append(path + [neighbor])  # Add new path

    return None  # No path found

# Example usage
start = "A"
end = "G"
path = find_shortest_path(graph, start, end)

if path:
    print("Shortest path:", " -> ".join(path))
else:
    print("No path found")
```

```
Shortest path: A -> H -> E -> G

Process finished with exit code 0
```

Task2.
Solution:

```python
graph = {
    "A": ["B", "C", "H"],
    "B": ["A"],
    "C": ["A", "D"],
    "D": ["C", "E", "F"],
    "E": ["D", "G", "H"],
    "F": ["D", "G"],
    "G": ["E", "F"],
    "H": ["A", "E"]
}

def depth_first_search(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)  # Mark the current node as visited
    print(start, end=" ") # Print the visited node

    for neighbor in graph[start]:
        if neighbor not in visited:
            depth_first_search(graph, neighbor, visited)  # Recursively call DFS on unvisited neighbors

# Example usage:
print("DFS traversal starting from node 'A':")
depth_first_search(graph, "A")
```

```
DFS traversal starting from node 'A':
A B C D E G F H
Process finished with exit code 0
```

Task3.
Solution:

```python
# Possible moves: Right, Left, Down, Up
directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

# Function to find the empty space (0) in the puzzle
def find_empty(puzzle):  1 usage
    for row in range(3):
        for col in range(3):
            if puzzle[row][col] == 0:
                return row, col  # Return the position of empty space

# Function to swap numbers and return a new puzzle state
def swap(puzzle, x1, y1, x2, y2):  1 usage
    new_puzzle = [row[:] for row in puzzle]  # Copy puzzle
    new_puzzle[x1][y1], new_puzzle[x2][y2] = new_puzzle[x2][y2], new_puzzle[x1][y1]  # S
    return new_puzzle

# BFS function to solve the 8-puzzle
def solve_puzzle(start, goal):  1 usage
    queue = [(start, [])]  # Use a list as a queue (FIFO)
    visited = set()
    visited.add(tuple(map(tuple, start)))  # Convert puzzle to tuple for storage

    while queue:
        current_puzzle, path = queue.pop(0)  # Get the front of the queue
```

```python
        if current_puzzle == goal:
            return path  # Return moves if goal is reached

        # Find empty space position
        empty_x, empty_y = find_empty(current_puzzle)

        # Try all possible moves
        for dx, dy in directions:
            new_x, new_y = empty_x + dx, empty_y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:  # Check if move is valid
                new_puzzle = swap(current_puzzle, empty_x, empty_y, new_x, new_y)

                if tuple(map(tuple, new_puzzle)) not in visited:  # Avoid repeating states
                    queue.append((new_puzzle, path + [(new_x, new_y)]))
                    visited.add(tuple(map(tuple, new_puzzle)))

    return None  # No solution found

# Example start puzzle
start_puzzle = [
    [1, 2, 3],
    [4, 0, 6],
    [7, 5, 8]
```

```python
]

# Goal puzzle
goal_puzzle = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

# Solve the puzzle
solution = solve_puzzle(start_puzzle, goal_puzzle)

# Print the result
if solution:
    print("Solution found in", len(solution), "moves")
    print("Moves:", solution)
else:
    print("No solution found")
```

```
Solution found in 2 moves
Moves: [(2, 1), (2, 2)]
```

Task4.
Solution:

```python
map_data = {
    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Sibiu': {'Arad': 140, 'Fagaras': 99, 'Rimnicu Vilcea': 80},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Bucharest': 101},
    'Bucharest': {} # Destination
}
def find_path(map, start, end, path=[]):  2 usages
    path = path + [start]
    if start == end:
        return path
    for city in map[start]:
        if city not in path:
            new_path = find_path(map, city, end, path)
            if new_path: return new_path
    return None

path = find_path(map_data, start: 'Arad', end: 'Bucharest')

if path:
    print("Path:", path)
else:
    print("No path found.")
```

```
Path: ['Arad', 'Zerind', 'Oradea', 'Sibiu', 'Fagaras', 'Bucharest']

Process finished with exit code 0
```

Task5:
Solution:

```
# Graph representation with nodes and weighted edges
Graph_nodes = {
    'A': [('B', 6), ('F', 3)],
    'B': [('C', 3), ('D', 2)],
    'C': [('D', 1), ('E', 5)],
    'D': [('C', 1), ('E', 8)],
    'E': [('I', 5), ('J', 5)],
    'F': [('G', 1), ('H', 7)],
    'G': [('I', 3)],
    'H': [('I', 2)],
    'I': [('E', 5), ('J', 3)],
}
# Function to get the neighbors (connections) for a nc
def get_neighbors(v):  1 usage
    return Graph_nodes.get(v, None)
# Heuristic function: defines the estimated cost from
def h(n):  2 usages
    H_dist = {
        'A': 10,
        'B': 8,
        'C': 5,
        'D': 7,
        'E': 3,
        'F': 6,
        'G': 5,
        'H': 3,
```

```python
            'H': 3,
            'I': 1,
            'J': 0
        }
        return H_dist.get(n, float('inf'))  # Return a high number for unk
# A* Algorithm to find the shortest path from start_node to stop_node
def aStarAlgo(start_node, stop_node):  1 usage
    # Open set stores the nodes to be evaluated
    open_set = {start_node}
    # Closed set stores the nodes that have already been evaluated
    closed_set = set()
    # g stores the actual cost of reaching a node
    g = {start_node: 0}
    # parents stores the path for reconstructing the shortest path
    parents = {start_node: start_node}

    while open_set:
        # Find the node in open_set with the lowest f = g + h
        n = None
        for v in open_set:
            if n is None or g[v] + h(v) < g[n] + h(n):
                n = v
```

```python
        # If the goal node is reached, reconstruct the path
        if n == stop_node:
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start_node)
            path.reverse()

            print(f"Path found: {path}")
            return path

        # If there is no node left in open_set, path does not exis
        if n is None:
            print("Path does not exist!")
            return None

        # Evaluate the neighbors of the current node
        for (m, weight) in get_neighbors(n):
            if m not in open_set and m not in closed_set:
                open_set.add(m)
                parents[m] = n
                g[m] = g[n] + weight
            else:
```

```
            else:
                # If the new path to m is shorter, update the values
                if g[m] > g[n] + weight:
                    g[m] = g[n] + weight
                    parents[m] = n
                    if m in closed_set:
                        closed_set.remove(m)
                        open_set.add(m)

        # Move the node from open set to closed set
        open_set.remove(n)
        closed_set.add(n)

    print("Path does not exist!")
    return None

# Example usage: Finding the shortest path from 'A' to 'J'
aStarAlgo( start_node: 'A',  stop_node: 'J')
```

```
Path found: ['A', 'F', 'G', 'I', 'J']

Process finished with exit code 0
```

Task6:

Solution:

```python
# 3x3 Tic-Tac-Toe board
board = [[' ' for _ in range(3)] for _ in range(3)]  # Empty board

# Print the board
def print_board():  3 usages
    for row in board:
        print('|'.join(row))
        print('-' * 5)

# Check if the current player has won
def check_winner(player):  5 usages
    # Check rows, columns, and diagonals
    for i in range(3):
        if all([board[i][j] == player for j in range(3)]) or \
           all([board[j][i] == player for j in range(3)]):
            return True
    if board[0][0] == board[1][1] == board[2][2] == player or \
       board[0][2] == board[1][1] == board[2][0] == player:
        return True
    return False

# Check if the board is full (draw)
def is_board_full():  3 usages
    return all([board[i][j] != ' ' for i in range(3) for j in range(3)])
```

```python
# Minimax function
def minimax(is_maximizing):  2 usages
    if check_winner('X'): return 1
    if check_winner('O'): return -1
    if is_board_full(): return 0

    best = -float('inf') if is_maximizing else float('inf')
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X' if is_maximizing else 'O'
                score = minimax(not is_maximizing)
                board[i][j] = ' '  # Undo the move
                if is_maximizing:
                    best = max(score, best)
                else:
                    best = min(score, best)
    return best
```

```python
# Find the best move for 'X'
def best_move():  2 usages
    best = -float('inf')
    move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X'
                score = minimax(False)
                board[i][j] = ' '
                if score > best:
                    best = score
                    move = (i, j)
    return move
```

```python
# Play the game
def play_game():  1 usage
    while True:
        print_board()
        if check_winner('X'):
            print("X wins!")
            break
        if check_winner('O'):
            print("O wins!")
            break
        if is_board_full():
            print("It's a draw!")
            break

        print("X's move:")
        row, col = best_move()
        board[row][col] = 'X'

        if check_winner('X'):
            print_board()
            print("X wins!")
            break
        if is_board_full():
            print_board()
            print("It's a draw!")
```

```
        print("O's move:")
        row, col = best_move()
        board[row][col] = 'O'

# Start the game
play_game()
```

```
-----
X|X|O
-----
X| |
-----
X wins!
```

S