ARTIFICIAL INTELLIGENCE ASSIGNMENT

- **Name - Himanshu Tiwari**
- **Roll number - 16**
- **Course - MSc Computational Science and Applications**
- **Subject - Artificial Intelligence Practical**

Q1. Write a Program to Implement Breadth First Search using Python.

```
def bfs(visited, graph, node): #function for BFS
  visited.append(node)
  queue.append(node)

  while queue:          # Creating loop to visit each node
    m = queue.pop(0)
    print (m, end = " ")

    for neighbour in graph[m]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
```

```
graph1 = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : []
}

graph2 = {
  "A": ["B", "C"],
  "B": ["D", "E"],
  "C": ["F"],
  "D": [],
  "E": [],
  "F": []
}
```

```
visited = [] # List for visited nodes.
queue = []     #Initialize a queue

print("Breadth-First Search")
bfs(visited, graph1, '5')
print()
bfs(visited, graph2, "A")
```

```
Breadth-First Search
5 3 7 2 4 8
A B C D E F
```

Q2. Write a Program to Implement Depth First Search using Python.

```
def dfs(visited, graph, node):  #function for dfs in recursive manner
    if node not in visited:
        print (node)
        visited.add(node)
        for neighbour in graph[node]:
            dfs(visited, graph, neighbour)
```

```
visited = set()

print("Depth-First Search")
dfs(visited, graph1, '5')
print()
dfs(visited, graph2, 'A')
```

```
Depth-First Search
5
3
```

```
2
4
8
7

A
B
D
E
C
F
```

Q3. Write a Program to Implement Tic-Tac-Toe game using Python.

```
import random


class TicTacToe:

    def __init__(self):
        self.board = []

    def create_board(self):
        for i in range(3):
            row = []
            for j in range(3):
                row.append('-')
            self.board.append(row)

    def get_random_first_player(self):
        return random.randint(0, 1)

    def fix_spot(self, row, col, player):
        self.board[row][col] = player

    def is_player_win(self, player):
        win = None

        n = len(self.board)

        # checking rows
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[i][j] != player:
                    win = False
                    break
            if win:
                return win

        # checking columns
        for i in range(n):
            win = True
            for j in range(n):
                if self.board[j][i] != player:
                    win = False
                    break
            if win:
                return win

        # checking diagonals
        win = True
        for i in range(n):
            if self.board[i][i] != player:
                win = False
                break
        if win:
            return win

        win = True
        for i in range(n):
            if self.board[i][n - 1 - i] != player:
                win = False
                break
        if win:
            return win
        return False

        for row in self.board:
            for item in row:
```

```
                if item == '-':
                    return False
        return True

    def is_board_filled(self):
        for row in self.board:
            for item in row:
                if item == '-':
                    return False
        return True

    def swap_player_turn(self, player):
        return 'X' if player == 'O' else 'O'

    def show_board(self):
        for row in self.board:
            for item in row:
                print(item, end=" ")
            print()

    def start(self):
        self.create_board()

        player = 'X' if self.get_random_first_player() == 1 else 'O'
        while True:
            print(f"Player {player} turn")

            self.show_board()

            # taking user input
            row, col = list(
                map(int, input("Enter row and column numbers to fix spot: ").split()))
            print()

            # fixing the spot
            self.fix_spot(row - 1, col - 1, player)

            # checking whether current player is won or not
            if self.is_player_win(player):
                print(f"Player {player} wins the game!")
                break

            # checking whether the game is draw or not
            if self.is_board_filled():
                print("Match Draw!")
                break

            # swapping the turn
            player = self.swap_player_turn(player)

        # showing the final view of board
        print()
        self.show_board()


# starting the game
tic_tac_toe = TicTacToe()
tic_tac_toe.start()
```

```
Player X turn
- - -
- - -
- - -
Enter row and column numbers to fix spot: 1 1

Player O turn
X - -
- - -
- - -
Enter row and column numbers to fix spot: 1 3

Player X turn
X - O
- - -
- - -
Enter row and column numbers to fix spot: 3 1

Player O turn
X - O
- - -
X - -
Enter row and column numbers to fix spot: 2 1
```

```
Player X turn
X - O
O - -
X - -
Enter row and column numbers to fix spot: 3 3

Player O turn
X - O
O - -
X - X
Enter row and column numbers to fix spot: 3 2

Player X turn
X - O
O - -
X O X
Enter row and column numbers to fix spot: 2 2

Player X wins the game!

X - O
O X -
X O X
```

Q4. Write a Program to Implement 8-Puzzle problem using Python.

```
from __future__ import print_function
import sys

def misplacedTiles(currentState):
    numberOfMisplacedTiles = 0
    goal = [1, 2, 3,
            4, 5, 6,
            7, 8, 0]

    for i in range(len(goal)):
        if currentState[i] != 0 and currentState[i] != goal[i]:
            numberOfMisplacedTiles += 1

    return numberOfMisplacedTiles

def printAsMatrix(lst):
    for i in range(len(lst)):
        if i % 3 == 0 and i > 0:
            print("")
        print(str(lst[i]) + " ", end="")
    print("")

def isSolvable(state):
    invCount = 0
    for i in range(0, 8):
        for j in range(i + 1, 9):
            if state[i] != 0 and state[j] != 0 and state[i] > state[j]:
                invCount += 1

    return invCount % 2 == 0

def move(indexes, position, state, path):
    minMisplacedTilesNumber = sys.maxsize
    storedState = list(state)

    for i in range(len(indexes)):
        duplicatedState = list(state)

        duplicatedState[position], duplicatedState[indexes[i]] = duplicatedState[indexes[i]], duplicatedState[position]

        misplacedTilesNumber = misplacedTiles(duplicatedState)

        if misplacedTilesNumber < minMisplacedTilesNumber and duplicatedState not in path:
            minMisplacedTilesNumber = misplacedTilesNumber
            storedState = list(duplicatedState)

    return storedState, minMisplacedTilesNumber

if __name__ == '__main__':
    state = []

    while len(state) != 9:
        x = int(input("Enter " + str(len(state)) + " element : "))
        state.append(x)

    path = [state]
```

```
    path = [state]
    heuristicValue = misplacedTiles(state)
    level = 1

    print("\n--- Level --- " + str(level))
    printAsMatrix(state)
    print("Heuristic Value : " + str(heuristicValue))

    if isSolvable(state):
        while heuristicValue > 0:
            position = state.index(0)
            level += 1
            possibleIndexesOfEmptyTile = []

            if position == 0:
                possibleIndexesOfEmptyTile = [1, 3]
            elif position == 1:
                possibleIndexesOfEmptyTile = [0, 2, 4]
            elif position == 2:
                possibleIndexesOfEmptyTile = [1, 5]
            elif position == 3:
                possibleIndexesOfEmptyTile = [0, 4, 6]
            elif position == 4:
                possibleIndexesOfEmptyTile = [1, 3, 5, 7]
            elif position == 5:
                possibleIndexesOfEmptyTile = [2, 4, 8]
            elif position == 6:
                possibleIndexesOfEmptyTile = [3, 7]
            elif position == 7:
                possibleIndexesOfEmptyTile = [4, 6, 8]
            elif position == 8:
                possibleIndexesOfEmptyTile = [5, 7]

            state, heuristicValue = move(possibleIndexesOfEmptyTile, position, state, path)

            path.append(state)

            print("\n-- Level -- " + str(level))
            printAsMatrix(state)
            print("\nHeuristic Value : " + str(heuristicValue))
    else:
        print("This puzzle is unsolvable")
```

```
Enter 0 element : 1
Enter 1 element : 2
Enter 2 element : 3
Enter 3 element : 0
Enter 4 element : 4
Enter 5 element : 6
Enter 6 element : 7
Enter 7 element : 5
Enter 8 element : 8

--- Level --- 1
1 2 3
0 4 6
7 5 8
Heuristic Value : 3

-- Level -- 2
1 2 3
4 0 6
7 5 8

Heuristic Value : 2

-- Level -- 3
1 2 3
4 5 6
7 0 8

Heuristic Value : 1

-- Level -- 4
1 2 3
4 5 6
7 8 0

Heuristic Value : 0
```

Q5. Write a Program to Implement Water-Jug problem using Python.

```
class WaterJug:
```

```
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z

    def DFS(self):

        frontier = [(0,0)]
        visited = set()

        while frontier:

            state = frontier.pop()
            x, y = state

            if (x,y) in visited:
                continue

            visited.add((x,y))

            if x == self.z or y == self.z or x+y == self.z:
                print("Goal reached: ", state)
                return True

            # Generate next states
            frontier.append((x, 0))  # Empty jug 1
            frontier.append((0, y))  # Empty jug 2
            frontier.append((self.x, y))  # Fill jug 1
            frontier.append((x, self.y))  # Fill jug 2
            frontier.append((max(0, x - (self.y - y)), min(self.y, y + x)))  # Pour jug 1 into jug 2
            frontier.append((min(self.x, x + y), max(0, y - (self.x - x))))  # Pour jug 2 into jug 1

        print("No solution found")
        return False

if __name__ == "__main__":

    x = 5
    y = 3
    z = 4

    problem = WaterJug(x, y, z)
    problem.DFS()
```

```
Goal reached:  (4, 3)
```

0s completed at 7:15 PM

Could not connect to the reCAPTCHA service. Please check your internet connection and reload to get a reCAPTCHA challenge.