Name: - Tanishq Parab

Roll No: -A033

TYBSC IT

Artificial Intelligence Practical

Serial No	Details	Date	Sign
1	1. a. Write a program to implement depth first search algorithm.		
2	1.b. Write a program to implement breadth first search algorithm.		
3	2.a. Write a program to simulate 4-Queen / N-Queen problem.		
4	2.b. Write a program to solve tower of Hanoi problem.		
5	5. a. Write a program to solve water jug problem.		
6	5.b. Design the simulation of tic – tac – toe game using min-max algorithm		

7	6. a. Write a program to solve Missionaries and Cannibals problem		
8	6.b Design an application to simulate number puzzle problem.		
9	7. a. Write a program to shuffle Deck of cards.		
10	8. Solve constraint satisfaction problem d. Magic Squares		

Practical 1

1) a. Write a program to implement depth first search algorithm.

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited
print(dfs(graph, 'A')) # {'E', 'D', 'F', 'A', 'C', 'B'}

def dfs_paths(graph, start, goal):
    stack = [(start, [start])]
    while stack:
        (vertex, path) = stack.pop()
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                stack.append((next, path + [next]))
l1 = list(dfs_paths(graph, 'A', 'E')) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
print(l1)
```

Output: -

```
{'B', 'A', 'E', 'D', 'C', 'F'}
[['A', 'C', 'F', 'E'], ['A', 'B', 'E']]
```

1) b. Write a program to implement breadth first search algorithm.

```
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])}

def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited

print(bfs(graph, 'C'))


def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))

print(list(bfs_paths(graph, 'A', 'F'))) # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]


def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

print(shortest_path(graph, 'A', 'F')) # ['A', 'C', 'F']
```

Output: -

```
{'E', 'A', 'C', 'F', 'B', 'D'}
[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
['A', 'C', 'F']
```

Practical 2

2) a. Write a program to simulate 4-Queen / N-Queen problem.

```
N = 4
def print_solution(board):
    for row in board:
        print(" ".join(str(x) for x in row))
    print()

def is_safe(board, row, col):
        # Check this row on left side
        for i in range(col):
            if board[row][i] == 1:
                return False
  for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 1:
                return False
 for i, j in zip(range(row, N), range(col, -1, -1)):
            if board[i][j] == 1:
                return False
     return True

def solve_nq_util(board, col):
    if col >= N:
        print_solution(board)
```

```
            return True

        res = False
        for i in range(N):
            if is_safe(board, i, col):
                board[i][col] = 1
                res = solve_nq_util(board, col + 1) or res
                board[i][col] = 0  # Backtrack
return res
def solve_nq():
        board = [[0] * N for _ in range(N)]
        if not solve_nq_util(board, 0):
            print("Solution does not exist")
solve_nq()
```

Output: -

```
>>>
============================ RESTART: D:/ai/bfs.py ============================
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0

0 1 0 0
0 0 0 1
1 0 0 0
0 0 1 0

>>> |
```

2) b. Write a program to solve tower of Hanoi problem.

```
def moveTower(height, fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height - 1, fromPole, withPole, toPole)
        moveDisk(fromPole, toPole)
        moveTower(height - 1, withPole, toPole, fromPole)

def moveDisk(fp, tp):
    print("moving disk from", fp, "to", tp)

moveTower(3, "A", "C", "B")
```

Output: -

```
moving disk from A to C
moving disk from A to B
moving disk from C to B
moving disk from A to C
moving disk from B to A
moving disk from B to C
moving disk from A to C
```

Practical 5

5. a. Write a program to solve water jug problem.

```
# 3 water jugs capacity -> (x,y,z) where x>y>z

# initial state (12,0,0)

# final state (6,5,1)

capacity = (12,8,5)

# Maximum capacities of 3 jugs -> x,y,z

x = capacity[0]

y = capacity[1]

z = capacity[2]

# to mark visited states memory is a dictionarycontaining key value pair.

memory = {}

# store solution path

ans = []

def get_all_states(state):

    # Let the 3 jugs be called a,b,c

    a = state[0]

    b = state[1]
```

```
c = state[2]
if(a==6 and b==6):
    ans.append(state)
    return True

# if current state is already visited earlier
if((a,b,c) in memory):
    return False
memory[(a,b,c)] = 1
#empty jug a
if(a>0):
    #empty a into b
    if(a+b<=y):
        if( get_all_states((0,a+b,c)) ):
            ans.append(state)
            return True
    else:
        if( get_all_states((a-(y-b), y, c)) ):
            ans.append(state)
            return True
    #empty a into c
    if(a+c<=z):
        if( get_all_states((0,b,a+c)) ):
            ans.append(state)
            return True
```

```
    else:
        if( get_all_states((a-(z-c), b, z)) ):
            ans.append(state)
            return True
    #empty jug b
if(b>0):
    #empty b into a
        if(a+b<=x):
            if( get_all_states((a+b, 0, c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((x, b-(x-a), c)) ):
                ans.append(state)
                return True
        #empty b into c
        if(b+c<=z):
            if( get_all_states((a, 0, b+c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a, b-(z-c), z)) ):
                ans.append(state)
                return True
    #empty jug c
```

```
    if(c>0):
        #empty c into a
            if(a+c<=x):
                if( get_all_states((a+c, b, 0)) ):
                    ans.append(state)
                    return True
            else:
                if( get_all_states((x, b, c-(x-a))) ):
                    ans.append(state)
                    return True
            #empty c into b
            if(b+c<=y):
                if( get_all_states((a, b+c, 0)) ):
                    ans.append(state)
                    return True
            else:
                if( get_all_states((a, y, c-(y-b))) ):
                    ans.append(state)
                    return True
    return False
initial_state = (12,0,0)
print("Starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
```

```
    print(i)
```

Output: -

```
Starting work...

(12, 0, 0)
(4, 8, 0)
(0, 8, 4)
(8, 0, 4)
(8, 4, 0)
(3, 4, 5)
(3, 8, 1)
(11, 0, 1)
(11, 1, 0)
(6, 1, 5)
(6, 6, 0)
```

5) b. Design the simulation of tic – tac – toe game using min-max algorithm.

```
import os

import time


board = [' ' for _ in range(10)]

player = 1

win = 1

draw = -1

running = 0

stop = 1

game = running
```

```
def drawboard():
    print(f"{board[1]}  |{board[2]}  |{board[3]}")
    print("---+---+---")
    print(f"{board[4]}  |{board[5]}  |{board[6]}")
    print("---+---+---")
    print(f"{board[7]}  |{board[8]}  |{board[9]}")
    print("---+---+---")

def checkposition(x):
    return board[x] == ' '

def checkwin():
    global game
    if (board[1] == board[2] == board[3] != ' ' or
        board[4] == board[5] == board[6] != ' ' or
        board[7] == board[8] == board[9] != ' ' or
        board[1] == board[4] == board[7] != ' ' or
        board[2] == board[5] == board[8] != ' ' or
        board[3] == board[6] == board[9] != ' ' or
        board[1] == board[5] == board[9] != ' ' or
        board[3] == board[5] == board[7] != ' '):
        game = win
    elif all(space != ' ' for space in board[1:]):
        game = draw
```

```
print("Tic-tac-toe game")
print("Please wait...")
time.sleep(1)

while game == running:
    os.system('cls' if os.name == 'nt' else 'clear')
    drawboard()

    if player % 2 != 0:
        print("Player 1's turn")
        mark = 'X'
    else:
        print("Player 2's turn")
        mark = 'O'

    try:
        choice = int(input("Enter position [1-9]: "))
        if choice < 1 or choice > 9:
            print("Invalid position, try again.")
            time.sleep(1)
            continue

        if checkposition(choice):
            board[choice] = mark
```

```
            player += 1
            checkwin()
        else:
            print("Position already taken!")
            time.sleep(1)

    except ValueError:   # fixed typo
        print("Please enter a valid position.")
        time.sleep(1)

os.system('cls' if os.name == 'nt' else 'clear')
drawboard()

if game == draw:
    print("Game Draw!")
elif game == win:
    winner = "Player 1 won!" if player % 2 == 0 else "Player 2 won!"
    print(winner)
```

Output: -

```
Tic-tac-toe game
Please wait...
   |   |
---+---+---
   |   |
---+---+---
   |   |
---+---+---
Player 1's turn
Enter position [1-9]: 1
X  |   |
---+---+---
   |   |
---+---+---
   |   |
---+---+---
Player 2's turn
Enter position [1-9]: 3
X  |   |O
---+---+---
   |   |
---+---+---
   |   |
---+---+---
Player 1's turn
Enter position [1-9]: 5
X  |   |O
---+---+---
   |X  |
---+---+---
   |   |
---+---+---
Player 2's turn
Enter position [1-9]: 6
X  |   |O
---+---+---
   |X  |O
---+---+---
   |   |
---+---+---
Player 1's turn

Enter position [1-9]: 9
X  |   |O
---+---+---
   |X  |O
---+---+---
   |   |X
---+---+---
Player 1 won!
```

Practical 6

6) a. Write a program to solve Missionaries and Cannibals problem.

```
import math


# Missionaries and Cannibals Problem
class State:
    def __init__(self, cannibal_left, missionary_left, boat, cannibal_right,
missionary_right):
        self.cannibal_left = cannibal_left
        self.missionary_left = missionary_left
        self.boat = boat
        self.cannibal_right = cannibal_right
        self.missionary_right = missionary_right
        self.parent = None

    def is_goal(self):
        if self.cannibal_left == 0 and self.missionary_left == 0:
            return True
        else:
            return False

    def is_valid(self):
        # All counts must be non-negative
        if (self.missionary_left < 0 or self.missionary_right < 0 or
                self.cannibal_left < 0 or self.cannibal_right < 0):
```

```
            return False

        # Cannibals cannot outnumber missionaries on either bank
        if (self.missionary_left > 0 and self.missionary_left <
self.cannibal_left):
            return False
        if (self.missionary_right > 0 and self.missionary_right <
self.cannibal_right):
            return False

        return True

    def __eq__(self, other):
        return (self.cannibal_left == other.cannibal_left and
                self.missionary_left == other.missionary_left and
                self.boat == other.boat and
                self.cannibal_right == other.cannibal_right and
                self.missionary_right == other.missionary_right)

    def __hash__(self):
        return hash((self.cannibal_left, self.missionary_left, self.boat,
                     self.cannibal_right, self.missionary_right))

    def successors(self):
        children = []
```

```
# Boat is on the left side
if self.boat == 'left':
    # Two missionaries cross left to right
    new_state = State(self.cannibal_left, self.missionary_left - 2,
                      'right', self.cannibal_right, self.missionary_right + 2)
    if new_state.is_valid():
        new_state.parent = self
        children.append(new_state)

    # Two cannibals cross left to right
    new_state = State(self.cannibal_left - 2, self.missionary_left,
                      'right', self.cannibal_right + 2, self.missionary_right)
    if new_state.is_valid():
        new_state.parent = self
        children.append(new_state)

    # One missionary and one cannibal cross left to right
    new_state = State(self.cannibal_left - 1, self.missionary_left - 1,
                      'right', self.cannibal_right + 1, self.missionary_right + 1)
    if new_state.is_valid():
        new_state.parent = self
        children.append(new_state)

    # One missionary crosses left to right
    new_state = State(self.cannibal_left, self.missionary_left - 1,
```

```
                 'right', self.cannibal_right, self.missionary_right + 1)
   if new_state.is_valid():
      new_state.parent = self
      children.append(new_state)

   # One cannibal crosses left to right
   new_state = State(self.cannibal_left - 1, self.missionary_left,
                 'right', self.cannibal_right + 1, self.missionary_right)
   if new_state.is_valid():
      new_state.parent = self
      children.append(new_state)

# Boat is on the right side
else:
   # Two missionaries cross right to left
   new_state = State(self.cannibal_left, self.missionary_left + 2,
                 'left', self.cannibal_right, self.missionary_right - 2)
   if new_state.is_valid():
      new_state.parent = self
      children.append(new_state)

   # Two cannibals cross right to left
   new_state = State(self.cannibal_left + 2, self.missionary_left,
                 'left', self.cannibal_right - 2, self.missionary_right)
   if new_state.is_valid():
```

```
            new_state.parent = self
            children.append(new_state)

        # One missionary and one cannibal cross right to left
        new_state = State(self.cannibal_left + 1, self.missionary_left + 1,
                          'left', self.cannibal_right - 1, self.missionary_right - 1)
        if new_state.is_valid():
            new_state.parent = self
            children.append(new_state)

        # One missionary crosses right to left
        new_state = State(self.cannibal_left, self.missionary_left + 1,
                          'left', self.cannibal_right, self.missionary_right - 1)
        if new_state.is_valid():
            new_state.parent = self
            children.append(new_state)

        # One cannibal crosses right to left
        new_state = State(self.cannibal_left + 1, self.missionary_left,
                          'left', self.cannibal_right - 1, self.missionary_right)
        if new_state.is_valid():
            new_state.parent = self
            children.append(new_state)

    return children
```

```
def breadth_first_search():
    initial_state = State(3, 3, 'left', 0, 0)

    if initial_state.is_goal():
        return initial_state

    frontier = [initial_state]
    explored = {initial_state}

    while frontier:
        current_state = frontier.pop(0)

        if current_state.is_goal():
            return current_state

        children = current_state.successors()
        for child in children:
            if child not in explored:
                explored.add(child)
                frontier.append(child)

    return None
```

```
def print_solution(solution):
    path = []
    parent = solution
    while parent:
        path.append(parent)
        parent = parent.parent

    # Print the path from start to finish
    for i in range(len(path) - 1, -1, -1):
        state = path[i]
        print(f"CannibalLeft: {state.cannibal_left}, MissionaryLeft: 
{state.missionary_left}, "
              f"Boat: {state.boat}, CannibalRight: {state.cannibal_right}, "
              f"MissionaryRight: {state.missionary_right}")


def main():
    solution = breadth_first_search()
    if solution:
        print("Missionaries and Cannibal Solution Found!")
        print_solution(solution)
    else:
        print("No solution found.")


if __name__ == "__main__":
    main()
```

Output: -

```
Missionaries and Cannibal Solution Found!
CannibalLeft: 3, MissionaryLeft: 3, Boat: left, CannibalRight: 0, MissionaryRight: 0
CannibalLeft: 1, MissionaryLeft: 3, Boat: right, CannibalRight: 2, MissionaryRight: 0
CannibalLeft: 2, MissionaryLeft: 3, Boat: left, CannibalRight: 1, MissionaryRight: 0
CannibalLeft: 0, MissionaryLeft: 3, Boat: right, CannibalRight: 3, MissionaryRight: 0
CannibalLeft: 1, MissionaryLeft: 3, Boat: left, CannibalRight: 2, MissionaryRight: 0
CannibalLeft: 1, MissionaryLeft: 1, Boat: right, CannibalRight: 2, MissionaryRight: 2
CannibalLeft: 2, MissionaryLeft: 2, Boat: left, CannibalRight: 1, MissionaryRight: 1
CannibalLeft: 2, MissionaryLeft: 0, Boat: right, CannibalRight: 1, MissionaryRight: 3
CannibalLeft: 3, MissionaryLeft: 0, Boat: left, CannibalRight: 0, MissionaryRight: 3
CannibalLeft: 1, MissionaryLeft: 0, Boat: right, CannibalRight: 2, MissionaryRight: 3
CannibalLeft: 1, MissionaryLeft: 1, Boat: left, CannibalRight: 2, MissionaryRight: 2
CannibalLeft: 0, MissionaryLeft: 0, Boat: right, CannibalRight: 3, MissionaryRight: 3
```

6) b. Design an application to simulate number puzzle problem.

```
from __future__ import print_function

from simpleai.search import astar, SearchProblem


GOAL = '''1-2-3

4-5-6

7-8-e'''


INITIAL = '''4-1-2

7-3-e

8-5-6'''

def list_to_string(list_):

    return '\n'.join(['-'.join(row) for row in list_])


def string_to_list(string_):

    return [row.split('-') for row in string_.split('\n')]


def find_location(rows, element_to_find):
```

```
    """Find the location (row, col) of a piece in the puzzle."""
    for ir, row in enumerate(rows):
        for ic, element in enumerate(row):
            if element == element_to_find:
                return ir, ic

# Precompute goal positions
goal_positions = {}
rows_goal = string_to_list(GOAL)
for number in '12345678e':
    goal_positions[number] = find_location(rows_goal, number)

#  Problem Definition
class EightPuzzleProblem(SearchProblem):
    def actions(self, state):
        rows = string_to_list(state)
        row_e, col_e = find_location(rows, 'e')
        actions = []
        if row_e > 0:
            actions.append(rows[row_e - 1][col_e])
        if row_e < 2:
            actions.append(rows[row_e + 1][col_e])
        if col_e > 0:
            actions.append(rows[row_e][col_e - 1])
        if col_e < 2:
```

```
            actions.append(rows[row_e][col_e + 1])
        return actions


    def result(self, state, action):
        rows = string_to_list(state)
        row_e, col_e = find_location(rows, 'e')
        row_n, col_n = find_location(rows, action)
        rows[row_e][col_e], rows[row_n][col_n] = rows[row_n][col_n],
rows[row_e][col_e]
        return list_to_string(rows)


    def is_goal(self, state):
        return state == GOAL


    def cost(self, state1, action, state2):
        return 1  # Each move has same cost
def heuristic(self, state):
        """ Manhattan distance heuristic """
        rows = string_to_list(state)
        distance = 0
        for number in '12345678e':
            row_n, col_n = find_location(rows, number)
            row_g, col_g = goal_positions[number]
            distance += abs(row_n - row_g) + abs(col_n - col_g)
        return distance
```

```
result = astar(EightPuzzleProblem(INITIAL))

# Print solution path

for action, state in result.path():

    print("Move:", action)

    print(state)

    print()
```

Output: -

```
================================================================================ RESTART: D:/ai/bfs.py ====================
Move: None
4-1-2
7-3-e
8-5-6

Move: 3
4-1-2
7-e-3
8-5-6

Move: 5
4-1-2
7-5-3
8-e-6

Move: 8
4-1-2
7-5-3
e-8-6

Move: 7
4-1-2
e-5-3
7-8-6

Move: 4
e-1-2
4-5-3
7-8-6

Move: 1
1-e-2
4-5-3
7-8-6

Move: 2
1-2-e
4-5-3
7-8-6

Move: 3
1-2-3
4-5-e
7-8-6

Move: 6
```

```
Move: 6
1-2-3
4-5-6
7-8-e

>>>
```

Practical 7

7) a. Write a program to shuffle Deck of cards.

```
#first let's import random procedures since we will be shuffling
import random
#next, let's start building list holders so we can place our cards in there:
cardfaces = []
suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
royals = ["J", "Q", "K", "A"]
deck = []
#now, let's start using loops to add our content:
for i in range(2,11):
    cardfaces.append(str(i)) #this adds numbers 2-10 and converts them to string data
for j in range(4):
    cardfaces.append(royals[j]) #this will add the royal faces to the cardbase
for k in range(4):
    for l in range(13):
        card = (cardfaces[l] + " of " + suits[k])
#this makes each card, cycling through suits, but first through faces
        deck.append(card)
#this adds the information to the "full deck" we want to make
#now let's shuffle our deck!
random.shuffle(deck)
#now let's see the cards!
for m in range(52):
    print(deck[m])
```

Output: -

```
-----------------------
8 of Diamonds
9 of Spades
A of Hearts
4 of Spades
K of Hearts
7 of Spades
8 of Hearts
J of Diamonds
5 of Clubs
J of Spades
9 of Hearts
9 of Clubs
Q of Spades
10 of Clubs
7 of Clubs
8 of Spades
4 of Hearts
K of Spades
Q of Hearts
3 of Diamonds
7 of Hearts
A of Spades
5 of Spades
A of Diamonds
9 of Diamonds
K of Clubs
10 of Hearts
4 of Diamonds
2 of Hearts
10 of Diamonds
10 of Spades
Q of Diamonds
3 of Clubs
2 of Spades
3 of Hearts
A of Clubs
6 of Diamonds
2 of Clubs
6 of Spades
J of Hearts
6 of Clubs
5 of Diamonds
4 of Clubs
```

```
Q of Clubs
J of Clubs
8 of Clubs
K of Diamonds
7 of Diamonds
5 of Hearts
6 of Hearts
2 of Diamonds
3 of Spades
```

Practical 8

8. Solve constraint satisfaction problem

a. Sudoku Solving using CSP

b. Map Coloring

c. Zebra Puzzle

d. Magic Squares

```
# A function to generate odd sized magic squares


def generate_square(n):
    # initialize magic square
    mat = [[0] * n for _ in range(n)]


    # Initialize position for 1
    i = n // 2
    j = n - 1


    # One by one put all values in magic square
    for num in range(1, n * n + 1):
        # if row is -1 and column becomes n,
        # set row = 0, col = n -2
        if i == -1 and j == n:
            j = n - 2
            i = 0
        else:
            # If next number goes to out of
```

```
        # square's right side
        if j == n:
            j = 0

        # If next number goes to out of
        # square's upper side
        if i < 0:
            i = n - 1

    # If number is already present decrement
    # column by 2, and increment row by 1
    if mat[i][j]:
        j -= 2
        i += 1
        continue
    else:
        # set number
        mat[i][j] = num

    # increment and decrement
    # column and row by 1 respectively
    j += 1
    i -= 1

return mat
```

```
n = 5
magic_square = generate_square(n)
for row in magic_square:
    print(" ".join(map(str, row)))
```

Output: -

```
10 3 25 18 17
2 24 22 16 9
0 21 15 8 1
20 14 7 5 0
13 11 4 0 19
```