CERTIFICATE

Class ____________ Year _________________

This is to certify that the work entered in this Journal is the work of

Shri / Kumari ____

of ______________ division _____________ Roll No. _____________

Uni. Exam No. ________________ has satisfactorily completed the required number of practical and worked for the 1st term / 2nd term / both the terms of the year 20_____ 20_____ in the college laboratory as laid down by the university.

Head of the Department	External Examiner	Internal Examiner Subject teacher

Date : / / 20 Department of

INDEX

PRACTICAL NO-1

A. Write a program to implement depth first search algorithm.
B. Write a program to implement breadth first search algorithm

A) AIM: Write a program to implement depth first search algorithm.

```
graph1 = {
    '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, node, visited):
    if node not in visited:
        visited.append(node)
        for n in graph[node]:
            dfs(graph, n, visited)
    return visited

visited = dfs(graph1, 'A', [])
print(visited)
```

OUTPUT:

```
======= RESTART: C:/Users/user/Desktop/practical Question/a:
['A', 'C', 'F', 'E', 'B', 'D']
```

B) AIM: 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'])
}

# Implementing BFS to explore the graph
def bfs(start):
    queue = [start]
    levels = {}  # This keeps track of levels
    levels[start] = 0  # Depth of start node is 0
    visited = set(start)

    while queue:
        node = queue.pop(0)
        neighbours = graph[node]

        for neighbor in neighbours:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                levels[neighbor] = levels[node] + 1

    print(levels)  # Print the level of each node
    return visited

# Printing nodes visited during BFS from node 'A'
print(str(bfs('A')))

# BFS to find all paths from start to goal
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]))
```

```python
# Finding and printing all paths from 'A' to 'F'
result = list(bfs_paths(graph, 'A', 'F'))
print(result)  # [['A', 'C', 'F'], ['A', 'B', 'E', 'F']]

# For finding the shortest path from start to goal
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None

# Finding and printing the shortest path from 'A' to 'F'
result1 = shortest_path(graph, 'A', 'F')
print(result1)  # ['A', 'C', 'F']
```

OUTPUT:

```
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
{'A': 0, 'B': 1, 'C': 1, 'D': 2, 'E': 2, 'F': 2}
{'D', 'F', 'C', 'A', 'E', 'B'}
[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
['A', 'C', 'F']
```

PRACTICAL NO- 2

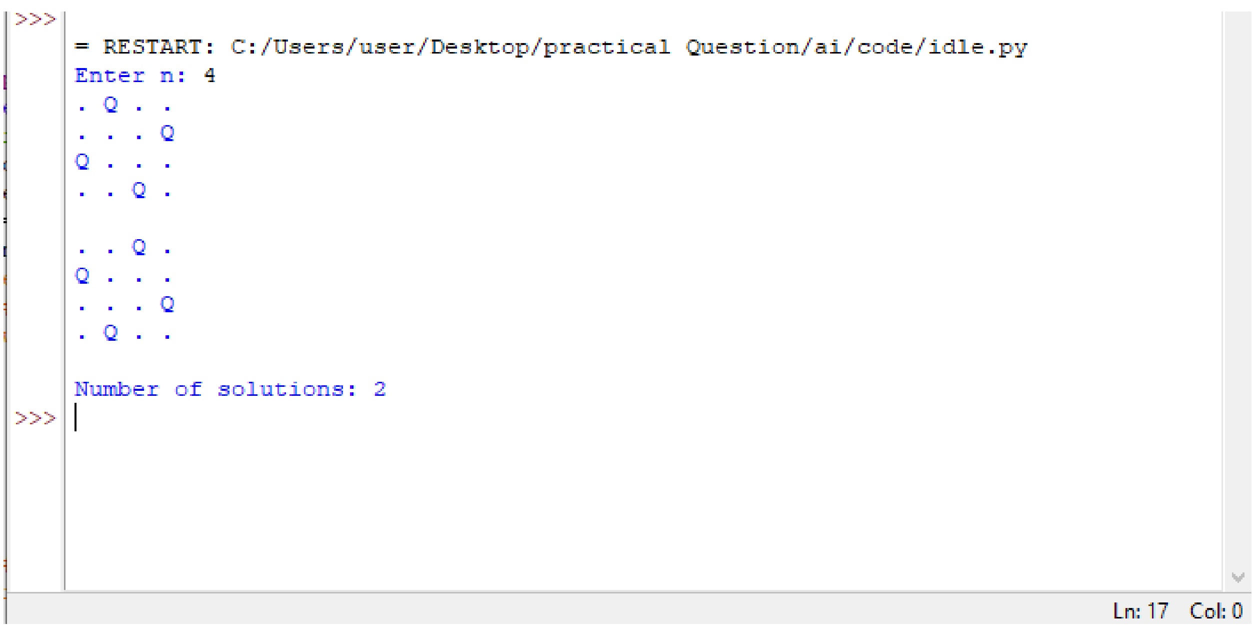
A. Write a program to simulate 4-Queen / N-Queen problem.
B. Write a program to solve tower of Hanoi problem.

A) AIM: Write a program to simulate 4-Queen / N-Queen problem.

```
class QueenChessBoard:
    def __init__(self, size):
        self.size = size
        self.columns = []

    def place_in_next_row(self, column):
        self.columns.append(column)
    def remove_in_current_row(self):
        if self.columns:  # Check if columns list is not empty
            return self.columns.pop()
        return None
    def is_this_column_safe_in_next_row(self, column):
        row = len(self.columns)
        # Check the column and both diagonals
        for queen_row, queen_column in enumerate(self.columns):
            if column == queen_column:
                return False
            if queen_column - queen_row == column - row:
                return False
            if (self.size - queen_column) - queen_row == (self.size - column) - row:
                return False
        return True
    def display(self):
        for row in range(self.size):
            for column in range(self.size):
                if column == self.columns[row]:
                    print('Q', end=' ')
                else:
                    print('.', end=' ')
            print()
        print()
def solve_queen(size):
    """Display chessboard for each valid configuration of placing n queens."""
    board = QueenChessBoard(size)
    number_of_solutions = 0
    row = 0
    column = 0
```

```
    while True:
        # place queen in the next row
        while column < size:
            if board.is_this_column_safe_in_next_row(column):
                board.place_in_next_row(column)
                row += 1
                column = 0
                break
            else:
                column += 1
        # If no valid column found or the board is full
        if column == size or row == size:
            if row == size:
                board.display()  # Found a solution
                number_of_solutions += 1
            # Backtrack after displaying a solution or dead-end
            if not board.columns:
                break  # Exit the loop if there are no queens left to backtrack
            prev_column = board.remove_in_current_row()
            row -= 1
            column = prev_column + 1  # Continue with the next column
    print('Number of solutions:', number_of_solutions)
n = int(input('Enter n: '))
solve_queen(n)
```

OUTPUT:



B) AIM: 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(f"Moving disk from {fp} to {tp}")

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

OUTPUT:

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
Moving disk from A to B
Moving disk from A to C
Moving disk from B to C
Moving disk from A to B
Moving disk from C to A
Moving disk from C to B
Moving disk from A to B
>>>
                                                                  Ln: 26   Col: 0
```

PRACTICAL NO.-3

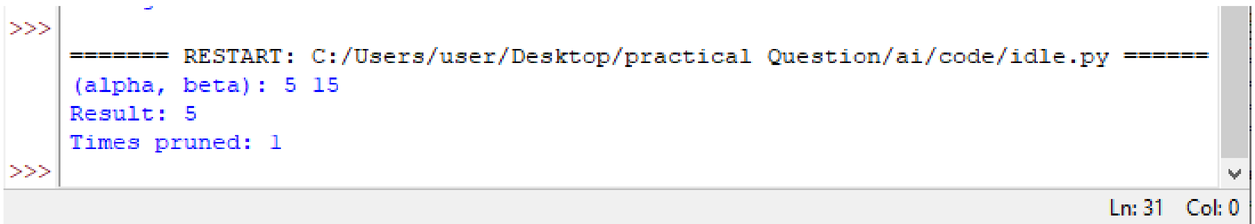
A. Write a program to implement alpha beta search.
B. Write a program for Hill climbing problem.

A) AIM: Write a program to implement alpha beta search.

```
tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]
root = 0
pruned = 0

def children(branch, depth, alpha, beta):
    global tree
    global root
    global pruned
    i = 0
    for child in branch:
        if isinstance(child, list):
            (nalpha, nbeta) = children(child, depth + 1, alpha, beta)
            if depth % 2 == 1:
                beta = min(nalpha, beta)
            else:
                alpha = max(nbeta, alpha)
            branch[i] = alpha if depth % 2 == 0 else beta
            i += 1
        else:
            if depth % 2 == 0 and alpha < child:
                alpha = child
            if depth % 2 == 1 and beta > child:
                beta = child
            if alpha >= beta:
                pruned += 1
                break
    if depth == root:
        tree = alpha if root == 0 else beta
    return (alpha, beta)

def alphabeta(in_tree=tree, start=root, upper=-15, lower=15):
    global tree
    global pruned
    (alpha, beta) = children(tree, start, upper, lower)
```

```
    print("(alpha, beta):", alpha, beta)
    print("Result:", tree)
    print("Times pruned:", pruned)
    return (alpha, beta, tree, pruned)

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

OUTPUT:



B) AIM: Write a program for Hill climbing problem.

```
import math

increment = 0.1
startingPoint = [1, 1]
point1 = [1, 5]
point2 = [6, 4]
point3 = [5, 2]
point4 = [2, 1]

def distance(x1, y1, x2, y2):
    return math.pow(x2 - x1, 2) + math.pow(y2 - y1, 2)

def sumOfDistances(x1, y1, px1, py1, px2, py2, px3, py3, px4, py4):
    return (distance(x1, y1, px1, py1) +
            distance(x1, y1, px2, py2) +
            distance(x1, y1, px3, py3) +
            distance(x1, y1, px4, py4))

def newDistance(x1, y1, point1, point2, point3, point4):
    d1 = [x1, y1]
    d1temp = sumOfDistances(x1, y1,
                            point1[0], point1[1],
                            point2[0], point2[1],
                            point3[0], point3[1],
                            point4[0], point4[1])
    d1.append(d1temp)
    return d1

minDistance = sumOfDistances(startingPoint[0], startingPoint[1],
                             point1[0], point1[1],
                             point2[0], point2[1],
                             point3[0], point3[1],
                             point4[0], point4[1])

flag = True

def newPoints(minimum, d1, d2, d3, d4):
```

```
    if d1[2] == minimum:
        return [d1[0], d1[1]]
    elif d2[2] == minimum:
        return [d2[0], d2[1]]
    elif d3[2] == minimum:
        return [d3[0], d3[1]]
    elif d4[2] == minimum:
        return [d4[0], d4[1]]

i = 1
while flag:
    d1 = newDistance(startingPoint[0] + increment, startingPoint[1], point1, point2, point3, point4)
    d2 = newDistance(startingPoint[0] - increment, startingPoint[1], point1, point2, point3, point4)
    d3 = newDistance(startingPoint[0], startingPoint[1] + increment, point1, point2, point3, point4)
    d4 = newDistance(startingPoint[0], startingPoint[1] - increment, point1, point2, point3, point4)

    print(i, round(startingPoint[0], 2), round(startingPoint[1], 2))

    minimum = min(d1[2], d2[2], d3[2], d4[2])
    if minimum < minDistance:
        startingPoint = newPoints(minimum, d1, d2, d3, d4)
        minDistance = minimum
        i += 1
    else:
        flag = False
```

OUTPUT:

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
1 1 1
2 1.1 1
3 1.2 1
4 1.3 1
5 1.4 1
6 1.5 1
7 1.6 1
8 1.6 1.1
9 1.7 1.1
10 1.7 1.2
11 1.7 1.3
12 1.8 1.3
13 1.8 1.4
14 1.9 1.4
15 2.0 1.4
16 2.0 1.5
17 2.1 1.5
18 2.1 1.6
19 2.2 1.6
20 2.2 1.7
21 2.3 1.7
22 2.3 1.8
23 2.3 1.9
24 2.4 1.9
25 2.5 1.9
26 2.5 2.0
27 2.6 2.0
28 2.6 2.1
29 2.7 2.1
30 2.7 2.2
31 2.8 2.2
32 2.8 2.3
33 2.9 2.3
34 2.9 2.4
35 3.0 2.4
36 3.0 2.5
37 3.1 2.5
```

```
37 3.1 2.5
38 3.1 2.6
39 3.2 2.6
40 3.2 2.7
41 3.2 2.8
42 3.3 2.8
43 3.4 2.8
44 3.4 2.9
45 3.5 2.9
46 3.5 3.0
```

PRACTICAL NO-4

A. Write a program to implement A* algorithm.
B. Write a program to solve water jug problem.

A) AIM: Write a program to implement A* algorithm.

```
from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'

class HelloProblem(SearchProblem):
    def actions(self, state):
        if len(state) < len(GOAL):
            return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        else:
            return []

    def result(self, state, action):
        return state + action

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

    def heuristic(self, state):
        # How far are we from the goal?
        wrong = sum([1 if state[i] != GOAL[i] else 0 for i in range(len(state))])
        missing = len(GOAL) - len(state)
        return wrong + missing

problem = HelloProblem(initial_state='')
result = astar(problem)

print(result.state)
print(result.path())
```

OUTPUT:

```
PS C:\Users\user\Desktop\practical Question\ai\code> python -u "c:\Users\user\Desktop\practical Question\ai\code\test.py"
HELLO WORLD
[(None, ''), ('H', 'H'), ('E', 'HE'), ('L', 'HEL'), ('L', 'HELL'), ('O', 'HELLO'), (' ', 'HELLO '), ('W', 'HELLO W'), ('O', 'HELLO WO'),
('R', 'HELLO WOR'), ('L', 'HELLO WORL'), ('D', 'HELLO WORLD')]
PS C:\Users\user\Desktop\practical Question\ai\code>
```

B) AIM: Write a program to solve water jug problem.

```
capacity = (12, 8, 5)  # Maximum capacities of the three jugs: x, y, z
x = capacity[0]
y = capacity[1]
z = capacity[2]

# Memory to keep track of visited states
memory = {}

# To store the solution path
ans = []

# Function to get all possible states
def get_all_states(state):
    # Let the three jugs be called a, b, and c
    a = state[0]
    b = state[1]
    c = state[2]

    # Check if the goal state is reached
    if (a == 6 and b == 6):
        ans.append(state)
        return True

    # If the current state has already been visited, skip it
    if (a, b, c) in memory:
        return False

    # Mark the current state as visited
    memory[(a, b, c)] = 1

    # Try all possible moves (pour from one jug to another)

    # Empty jug a into jug b
    if a > 0:
        # If a can completely fit into b
        if a + b <= y:
            if get_all_states((0, a + b, c)):
                ans.append(state)
                return True
        else:
            # Otherwise pour as much as b can hold
            if get_all_states((a - (y - b), y, c)):
                ans.append(state)
                return True
```

```
    # Empty jug a into jug 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 into jug a
if b > 0:
    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 jug b into jug 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 into jug a
if c > 0:
    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 jug c into jug 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 if no solution is found in this path
    return False


# Initial state (12, 0, 0)
initial_state = (12, 0, 0)

print("Starting search...\n")
get_all_states(initial_state)

# Reverse the solution path to show from initial state to goal state
ans.reverse()
# Print the solution path
for i in ans:
    print(i)
```

OUTPUT:

```
>>>
    ================================================== RESTART: C:/Users/use
    Starting search...

    (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)
>>>
```

PRACTICAL NO-5

A. Design the simulation of tic – tac – toe game using min-max algorithm.
B. Write a program to shuffle Deck of cards.

A) AIM: Design the simulation of tic – tac – toe game using min-max algorithm.

```
import os
import time

board = [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
player = 1
# Win/Draw/Running flags
Win = 1
Draw = -1
Running = 0
Game = Running
# This function draws the game board
def DrawBoard():
    print(" %c | %c | %c " % (board[1], board[2], board[3]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[4], board[5], board[6]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[7], board[8], board[9]))
    print("   |   |   ")
# This function checks if a position is empty
def CheckPosition(x):
    return board[x] == ' '
# This function checks if the player has won or not
def CheckWin():
    global Game
    # Horizontal winning condition
    if board[1] == board[2] == board[3] and board[1] != ' ':
        Game = Win
    elif board[4] == board[5] == board[6] and board[4] != ' ':
        Game = Win
    elif board[7] == board[8] == board[9] and board[7] != ' ':
        Game = Win
    # Vertical winning condition
    elif board[1] == board[4] == board[7] and board[1] != ' ':
        Game = Win
    elif board[2] == board[5] == board[8] and board[2] != ' ':
        Game = Win
    elif board[3] == board[6] == board[9] and board[3] != ' ':
```

```
        Game = Win
    # Diagonal winning condition
    elif board[1] == board[5] == board[9] and board[1] != ' ':
        Game = Win
    elif board[3] == board[5] == board[7] and board[3] != ' ':
        Game = Win
    # Match draw condition
    elif all(board[i] != ' ' for i in range(1, 10)):
        Game = Draw
    else:
        Game = Running
# Main game loop
print("Tic-Tac-Toe Game")
print("Player 1 [X] --- Player 2 [O]\n")
time.sleep(1)
while Game == Running:
    os.system('cls' if os.name == 'nt' else 'clear')  # Clear console
    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 the position [1-9]: "))
        if choice < 1 or choice > 9:
            print("Invalid position! Choose between 1-9.")
            continue
    except ValueError:
        print("Invalid input! Please enter a number.")
        continue

    if CheckPosition(choice):
        board[choice] = Mark
        player += 1
        CheckWin()
    else:
        print("Position already taken! Try another.")

os.system('cls' if os.name == 'nt' else 'clear')
DrawBoard()
if Game == Draw:
    print("It's a draw!")
elif Game == Win:
```

```
    winner = "Player 1" if player % 2 == 0 else "Player 2"
    print(f"{winner} wins!")
```

OUTPUT:

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
Tic-Tac-Toe Game
Player 1 [X] --- Player 2 [O]

   |   |
___|___|___
   |   |
___|___|___
   |   |
   |   |
Player 1's turn
Enter the position [1-9]: 1
 X |   |
___|___|___
   |   |
___|___|___
   |   |
   |   |
Player 2's turn
Enter the position [1-9]: 5
 X |   |
___|___|___
   | O |
___|___|___
   |   |
   |   |
Player 1's turn
Enter the position [1-9]: 2
 X | X |
___|___|___
   | O |
___|___|___
   |   |
   |   |
Player 2's turn
Enter the position [1-9]: 7
 X | X |
___|___|___
   | O |
___|___|___
 O |   |
   |   |
Player 1's turn
Enter the position [1-9]: 3
 X | X | X
___|___|___
   | O |
___|___|___
 O |   |
   |   |
Player 1 wins!
```

B) Write a program to shuffle Deck of cards.

```
import random
# List holders for card faces, suits, royals, and deck
cardfaces = []
suits = ["Hearts", "Diamonds", "Clubs", "Spades"]
royals = ["J", "Q", "K", "A"]
deck = []
# Add number cards 2-10 to cardfaces
for i in range(2, 11):
    cardfaces.append(str(i))  # Convert number to string and add to cardfaces
# Add royal faces (J, Q, K, A) to cardfaces
for j in range(4):
    cardfaces.append(royals[j])
# Create the full deck of cards by combining faces and suits
for k in range(4):  # For each suit
    for l in range(13):  # For each face
        card = (cardfaces[l] + " of " + suits[k])
        deck.append(card)
# Shuffle the deck
random.shuffle(deck)
# Display all cards in the shuffled deck
for m in range(52):
    print(deck[m])
```

```
>>>
================================================ RESTART: C:/Users/user/Desktop/practical Question/ai/code/:
Q of Spades
2 of Clubs
3 of Hearts
9 of Hearts
8 of Hearts
9 of Clubs
10 of Spades
6 of Spades
3 of Diamonds
5 of Diamonds
7 of Diamonds
4 of Clubs
J of Spades
A of Diamonds
5 of Spades
10 of Diamonds
J of Clubs
K of Diamonds
J of Diamonds
5 of Hearts
Q of Diamonds
A of Hearts
Q of Clubs
8 of Diamonds
8 of Spades
J of Hearts
2 of Hearts
6 of Diamonds
7 of Hearts
A of Clubs
A of Spades
K of Spades
K of Hearts
6 of Hearts
6 of Clubs
4 of Hearts
3 of Spades
2 of Spades
```

```
2 of Diamonds
9 of Diamonds
4 of Spades
Q of Hearts
5 of Clubs
10 of Clubs
7 of Spades
K of Clubs
9 of Spades
8 of Clubs
10 of Hearts
3 of Clubs
7 of Clubs
4 of Diamonds
>>>
```

PRACTICAL NO.-6

A. 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-e-3
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 of a piece in the puzzle.
    Returns a tuple: row, column'''
    for ir, row in enumerate(rows):
        for ic, element in enumerate(row):
            if element == element_to_find:
                return ir, ic
# Cache for the goal position of each piece to avoid recalculating
goal_positions = {}
rows_goal = string_to_list(GOAL)
for number in '12345678e':
    goal_positions[number] = find_location(rows_goal, number)

class EigthPuzzleProblem(SearchProblem):
    def actions(self, state):
        '''Returns a list of the pieces we can move to the empty space.'''
        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
```

```python
    def result(self, state, action):
        '''Return the resulting state after moving a piece to the empty space.'''
        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):
        '''Returns true if a state is the goal state.'''
        return state == GOAL

    def cost(self, state1, action, state2):
        '''Returns the cost of performing an action.'''
        return 1
    def heuristic(self, state):
        '''Returns an estimation of the distance from a state to the goal using Manhattan distance.'''
        rows = string_to_list(state)
        distance = 0
        for number in '12345678e':
            row_n, col_n = find_location(rows, number)
            row_n_goal, col_n_goal = goal_positions[number]
            distance += abs(row_n - row_n_goal) + abs(col_n - col_n_goal)
        return distance

result = astar(EigthPuzzleProblem(INITIAL))
for action, state in result.path():
    print('Move number:', action)
    print(state)
```

OUTPUT:

```
PS C:\Users\user\Desktop\practical Question\ai> python
Move number: None
4-1-2
7-e-3
8-5-6
Move number: 5
4-1-2
7-5-3
8-e-6
Move number: 8
4-1-2
7-5-3
e-8-6
Move number: 7
4-1-2
e-5-3
7-8-6
Move number: 4
e-1-2
4-5-3
7-8-6
Move number: 1
1-e-2
4-5-3
7-8-6
Move number: 2
1-2-e
4-5-3
7-8-6
Move number: 3
1-2-3
4-5-e
7-8-6
Move number: 6
1-2-3
4-5-6
7-8-e
```

PRACTICAL No.-7

AIM: Solve constraint satisfaction problem

```
from __future__ import print_function
from simpleai.search import CspProblem, backtrack, min_conflicts,
MOST_CONSTRAINED_VARIABLE, HIGHEST_DEGREE_VARIABLE,
LEAST_CONSTRAINING_VALUE
variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')
domains = dict((v, ['red', 'green', 'blue']) for v in variables)

def const_different(variables, values):
    return values[0] != values[1]  # expect the value of the neighbors to be different

constraints = [
    (('WA', 'NT'), const_different),
    (('WA', 'SA'), const_different),
    (('SA', 'NT'), const_different),
    (('SA', 'Q'), const_different),
    (('NT', 'Q'), const_different),
    (('SA', 'NSW'), const_different),
    (('Q', 'NSW'), const_different),
    (('SA', 'V'), const_different),
    (('NSW', 'V'), const_different),
]

my_problem = CspProblem(variables, domains, constraints)

print(backtrack(my_problem))
print(backtrack(my_problem, variable_heuristic=MOST_CONSTRAINED_VARIABLE))
print(backtrack(my_problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE))
print(backtrack(my_problem, value_heuristic=LEAST_CONSTRAINING_VALUE))
print(backtrack(my_problem, variable_heuristic=MOST_CONSTRAINED_VARIABLE,
value_heuristic=LEAST_CONSTRAINING_VALUE))
print(backtrack(my_problem, variable_heuristic=HIGHEST_DEGREE_VARIABLE,
value_heuristic=LEAST_CONSTRAINING_VALUE))
print(min_conflicts(my_problem))
```

OUTPUT:

```
PS C:\Users\user\Desktop\practical Question\ai\code> python -u "c:\Users\user\Desktop\practical Question\ai\
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'SA': 'red', 'NT': 'green', 'Q': 'blue', 'NSW': 'green', 'WA': 'blue', 'V': 'blue', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'WA': 'red', 'NT': 'green', 'SA': 'blue', 'Q': 'red', 'NSW': 'green', 'V': 'red', 'T': 'red'}
{'SA': 'red', 'NT': 'green', 'Q': 'blue', 'NSW': 'green', 'WA': 'blue', 'V': 'blue', 'T': 'red'}
{'WA': 'blue', 'NT': 'red', 'SA': 'green', 'Q': 'blue', 'NSW': 'red', 'V': 'blue', 'T': 'red'}
PS C:\Users\user\Desktop\practical Question\ai\code>
```

PRACTICAL No.-8

A) AIM: Derive the expressions based on associative law

```
def demonstrate_associative_law(a, b, c):
    # Associative Law of Addition
    sum1 = (a + b) + c
    sum2 = a + (b + c)
    # Associative Law of Multiplication
    product1 = (a * b) * c
    product2 = a * (b * c)
    print(f"Associative Law of Addition:")
    print(f"(a + b) + c = {sum1} and a + (b + c) = {sum2}")
    print(f"Result: {'Equal' if sum1 == sum2 else 'Not Equal'}\n")
    print(f"Associative Law of Multiplication:")
    print(f"(a * b) * c = {product1} and a * (b * c) = {product2}")
    print(f"Result: {'Equal' if product1 == product2 else 'Not Equal'}")# Example usage
a = 2
b = 3
c = 4
demonstrate_associative_law(a, b, c)
```

OUTPUT:

```
practical Question/ai/code/idle.py ============================================
===
Associative Law of Addition:
(a + b) + c = 9 and a + (b + c) = 9
Result: Equal

Associative Law of Multiplication:
(a * b) * c = 24 and a * (b * c) = 24
Result: Equal
>>>
                                                              Ln: 254   Col: 0
```

B) AIM: Derive the expressions based on distributive law

```
def distributive_law(a, b, c):
    # Calculate both sides of the distributive law
    left_side = a * (b + c)           # Left side: a * (b + c)
    right_side = (a * b) + (a * c)    # Right side: (a * b) + (a * c)
    return left_side, right_side

# Test values
a = 3
b = 4
c = 5

# Check Distributive Law
left_result, right_result = distributive_law(a, b, c)
print(f"Distributive Law: a * (b + c) = {left_result}, (a * b) + (a * c) = {right_result}")

# Verify if both sides are equal
if left_result == right_result:
    print("The Distributive Law holds true.")
else:
    print("The Distributive Law does not hold true.")
```

OUTPUT:

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
Distributive Law: a * (b + c) = 27, (a * b) + (a * c) = 27
The Distributive Law holds true.
>>>
```

Ln: 258   Col: 0

PRACTICAL No.-9

AIM: derive the predicate .(for eg: sachin is batman , batman is cricketer)-> schin is cricketer

```
# Define facts as a dictionary where key is the subject and value is the predicate
facts = {
    'Sachin': 'batsman',
    'batsman': 'cricketer'
}
# Function to derive the final fact from the initial fact
def derive_fact(starting_subject, target_predicate):
    current_subject = starting_subject
    while current_subject in facts:
        current_subject = facts[current_subject]
        if current_subject == target_predicate:
            return f"{starting_subject} is {target_predicate}"
    return "Cannot derive the fact"
# Define the starting subject and the target predicate
starting_subject = 'Sachin'
target_predicate = 'cricketer'
# Derive and print the fact
result = derive_fact(starting_subject, target_predicate)
print(result)
```

OUTPUT:

```
>>>
======= RESTART: C:/Users/user/Desktop/practical Question/ai/code/idle.py ======
Sachin is cricketer
>>> |
                                                                  Ln: 261   Col: 0
```

PRACTICAL No.-10

AIM: Write a program which contains three predicates: male, female, parent. Make rule for following family relations: Father, Mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew, and niece, cousin.
Question:
1. Draw family tree
2. Define: Clauses,facts,Predicates and Rules with conjunction and disjunction

```
# Facts about individuals
male = {'John', 'Robert', 'Michael', 'Kevin'}
female = {'Mary', 'Patricia', 'Jennifer', 'Linda'}
parents = {
    'John': ['Michael', 'Sarah'],   # John is the father of Michael and Sarah
    'Mary': ['Michael', 'Sarah'],    # Mary is the mother of Michael and Sarah
    'Robert': ['Kevin'],             # Robert is the father of Kevin
    'Patricia': ['Kevin']            # Patricia is the mother of Kevin
}
def is_father(father, child):
    return father in male and child in parents.get(father, [])
def is_mother(mother, child):
    return mother in female and child in parents.get(mother, [])
def is_grandfather(grandfather, grandchild):
    return grandfather in male and any(is_father(parent, grandchild) or is_mother(parent,
grandchild) for parent in parents.get(grandfather, []))
def is_grandmother(grandmother, grandchild):
    return grandmother in female and any(is_father(parent, grandchild) or is_mother(parent,
grandchild) for parent in parents.get(grandmother, []))
def is_brother(sibling1, sibling2):
    return sibling1 in male and sibling2 in parents.get(sibling1, parents[sibling2])
def is_sister(sibling1, sibling2):
    return sibling1 in female and sibling2 in parents.get(sibling1, parents[sibling2])
def is_uncle(uncle, nephew):
    return uncle in male and (any(is_brother(uncle, parent) for parent in parents.get(nephew, []))
or any(is_sister(uncle, parent) for parent in parents.get(nephew, [])))
def is_aunt(aunt, niece):
    return aunt in female and (any(is_brother(aunt, parent) for parent in parents.get(niece, [])) or
any(is_sister(aunt, parent) for parent in parents.get(niece, [])))
def is_cousin(cousin1, cousin2):
    return any(parent1 != parent2 and (parent1 in parents.get(cousin2, []) or parent2 in
parents.get(cousin1, []))
            for parent1 in parents.get(cousin1, [])
            for parent2 in parents.get(cousin2, []))
# Example Queries
```

```
print("Is John the father of Michael?", is_father('John', 'Michael'))
print("Is Mary the mother of Sarah?", is_mother('Mary', 'Sarah'))
print("Is Robert a grandfather of Michael?", is_grandfather('Robert', 'Michael'))
print("Is Patricia an aunt of Kevin?", is_aunt('Patricia', 'Kevin'))
```

OUTPUT: