VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence

Submitted by

NITHYA LAKSHMI V (1BM22CS186)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
September-2024 to February-2025

B. M. S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated to Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "**Artificial Intelligence**" carried out by **NITHYA LAKSHMI V (1BM22CS186),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum during the academic semester September-2024 to February-2024. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence **(**23CS5PCAIN**)** work prescribed for the said degree.

Prof. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

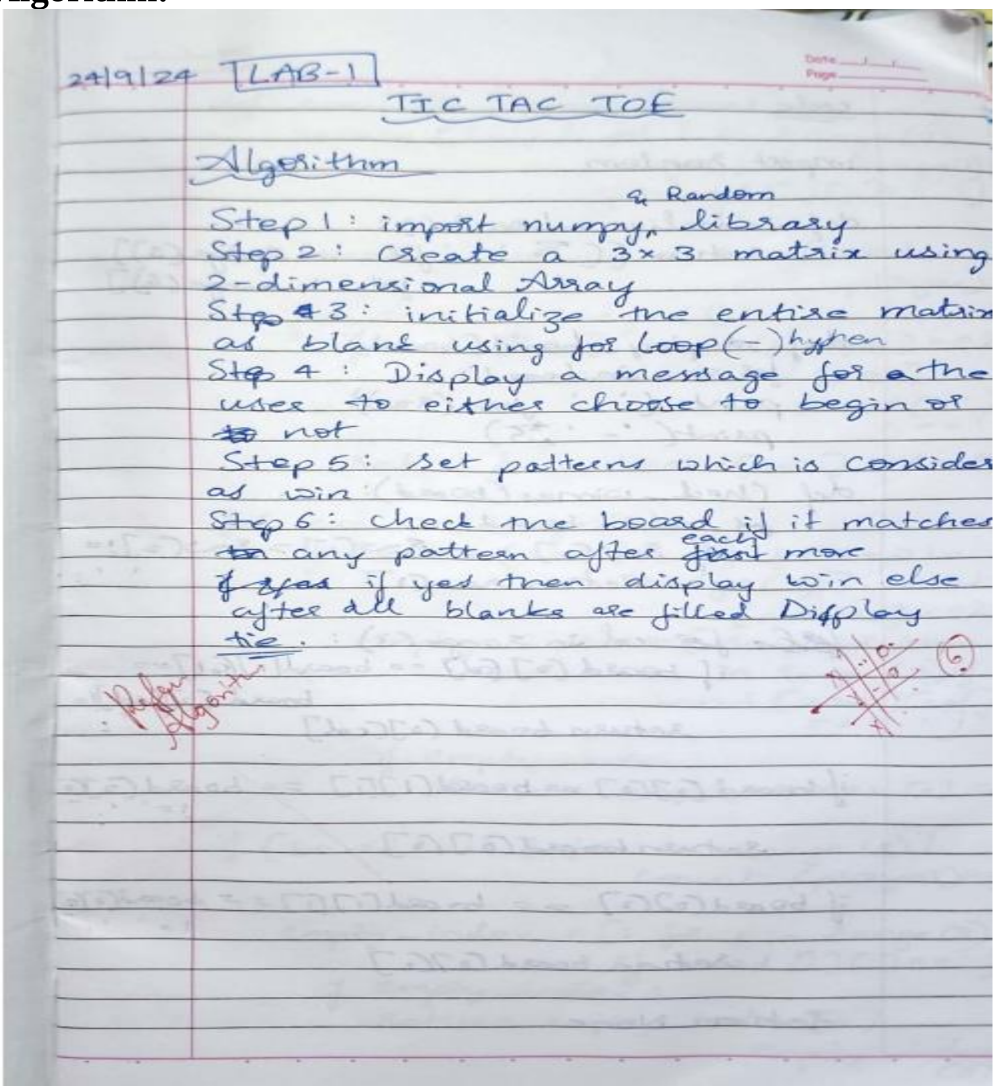
Index

Github Link:
https://github.com/Nithya1909/AI-LAB

WEEK 1 24-09-2024

1. **Implement TIC-TAC-TOE**

Algorithm:



24/9/24 [LAB-1]

TIC TAC TOE

Algorithm

Step 1: import numpy, & Random library

Step 2: Create a 3×3 matrix using 2-dimensional Array

Step ~~4~~3: initialize the entire matrix as blank using for Loop (-) hyphen

Step 4: Display a message for ~~a~~ the user to either choose to begin or ~~to~~ not

Step 5: Set patterns which is considered as win

Step 6: check the board if it matches ~~to~~ any pattern after ~~just~~ each move ~~if yes~~ if yes then display win else after all blanks are filled Display tie.

Refer Algorithm

6

Code:

```
board={1:' ',2:' ',3:' ',
    4:' ',5:' ',6:' ',
    7:' ',8:' ',9:' '
}
```

```
def printBoard(board):
  print(board[1]+'|'+board[2]+'|'+board[3])
  print('-+-+-')
  print(board[4] + '|' + board[5] + '|' + board[6])
  print('-+-+-')
  print(board[7] + '|' + board[8] + '|' + board[9])
  print('\n')

def spaceFree(pos):
  if(board[pos]==' '):
      return True
  else:
      return False
def checkWin():
  if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
      return True
  elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
      return True
  elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
      return True
  elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
      return True
  elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
      return True
  elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
      return True
  elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
      return True
  elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
      return True
  else:
      return False
def checkMoveForWin(move):
  if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
      return True
  elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
      return True
  elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
      return True
  elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
```

```
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True

def insertLetter(letter, position):
    if (spaceFree(position)):
        board[position] = letter
        printBoard(board)
        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return
    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)
        return
player = 'O'
bot ='X'
def playerMove():
    position=int(input('Enter position for O:'))
    insertLetter(player, position)
```

```
    return

def compMove():
    bestScore=-1000
    bestMove=0
    for key in board.keys():
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, False)
            board[key] = ' '
            if (score > bestScore):
                bestScore = score
                bestMove = key
    insertLetter(bot, bestMove)
    return

def minimax(board, isMaximizing):
    if (checkMoveForWin(bot)):
        return 1
    elif (checkMoveForWin(player)):
        return -1
    elif (checkDraw()):
        return 0
    if isMaximizing:
        bestScore = -1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                if (score > bestScore):
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000
        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
```

```
            if (score < bestScore):
                bestScore = score
        return bestScore

while not checkWin():
    compMove()
    playerMove()
```

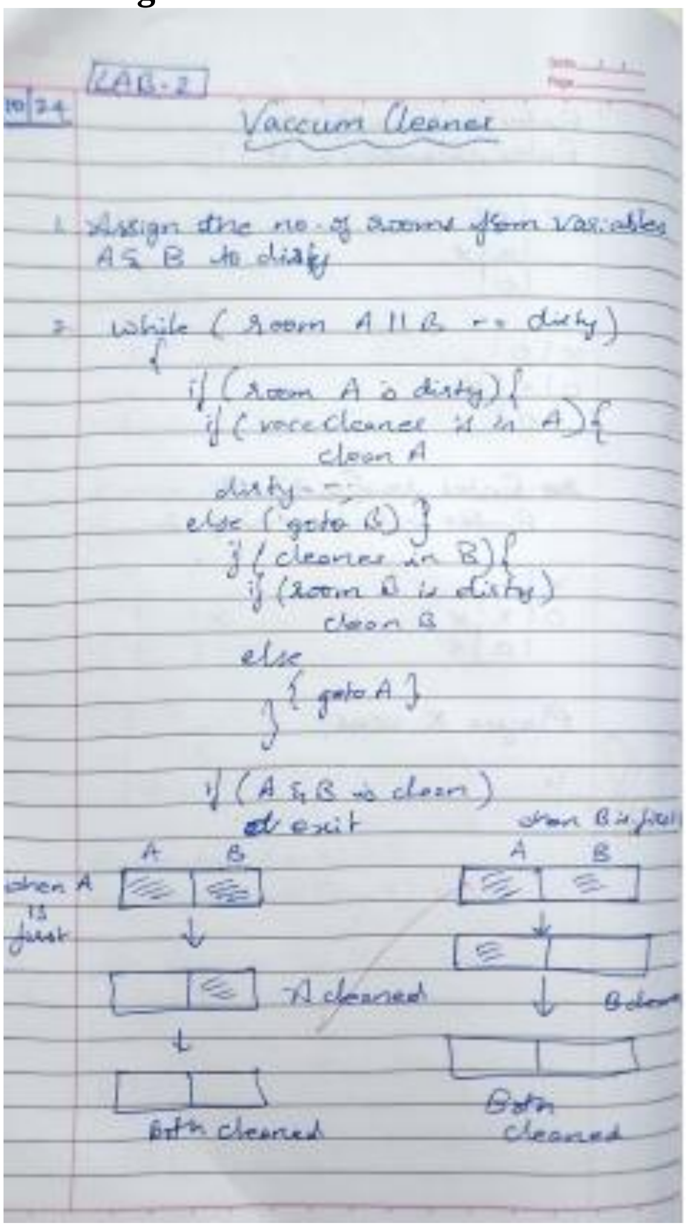
Output:

```
Yashraj Sinha (1BM22CS335)
Welcome to Tic Tac Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X |   |
---------
  |   |
---------
  |   |
---------
Computer chose: (1, 1)
X |   |
---------
  | O |
---------
  |   |
---------
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X |   |
---------
  | O |
---------
  | X |
---------
Computer chose: (1, 0)
X |   |
---------
O | O |
---------
  | X |
---------
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 2
X |   |
---------
O | O | X
---------
  | X |
---------
Computer chose: (0, 2)
X |   | O
---------
O | O | X
---------
  | X |
---------
Enter the row (0, 1, 2): 2
```

```
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 0
X |   | O
---------
O | O | X
---------
X | X |
---------
Computer chose: (2, 2)
X |   | O
---------
O | O | X
---------
X | X | O
---------
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 1
X | X | O
---------
O | O | X
---------
X | X | O
---------
It's a tie!
```

2. **Implement a vacuum cleaner agent.**

Algorithm:



LAB-2

10/24

Vaccum Cleaner

1. Assign the no. of rooms from variables A & B to dirty

2. while (room A || B == dirty)
{
if (room A is dirty) {
if (vacc cleaner is in A) {
clean A
dirty = 0
else { goto B } }
if (cleaner in B) {
if (room B is dirty)
clean B
else
{ goto A }
}

if (A & B is clean)
exit

Code:

```
def vacuum_world():
    goal_state = {'A': '0', 'B': '0'}
```

```
    cost = 0

    location_input = input("Enter Location of Vacuum (A or B): ").strip().upper()
    status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for Dirty): ").strip()

    other_location = 'B' if location_input == 'A' else 'A'
    status_input_complement = input(f"Enter status of {other_location} (0 for Clean, 1 for Dirty):
").strip()

    print("Initial Location Condition:", goal_state)

    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'
            cost += 2
            print("Cost for CLEANING A:", cost)
            print("Location A has been Cleaned.")

        if status_input_complement == '1':
            print("Location B is Dirty.")
            print("Moving right to Location B.")
            cost += 1
            print("COST for moving RIGHT:", cost)
            goal_state['B'] = '0'
            cost += 2
            print("COST for SUCK:", cost)
            print("Location B has been Cleaned.")
        else:
            print("Location B is already clean.")

    elif location_input == 'B':
        print("Vacuum is placed in Location B")
        if status_input == '1':
            print("Location B is Dirty.")
            goal_state['B'] = '0'
            cost += 2
            print("COST for CLEANING B:", cost)
            print("Location B has been Cleaned.")
```

```
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1
            print("COST for moving LEFT:", cost)
            goal_state['A'] = '0'
            cost += 2
            print("COST for SUCK:", cost)
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")

    print("GOAL STATE:", goal_state)
    print("Performance Measurement (Total Cost):", cost)
vacuum_world()
```

Output:

```
Enter Location of Vacuum (A or B): A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of B (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A: 2
Location A has been Cleaned.
Location B is Dirty.
Moving right to Location B.
COST for moving RIGHT: 3
COST for SUCK: 5
Location B has been Cleaned.
GOAL STATE: {'A': '0', 'B': '0'}
Performance Measurement (Total Cost): 5
```

WEEK 2   1-10-2024

1. Solve 8-puzzle problem using DFS

Algorithm:

8/10/24 LAB-3

8 puzzle Game

8 puzzle using DFS and Manhattan distance

1	7	4
5		2
3	8	6

1	2	3
4	5	6
7	8	

Original State

1. ~~Initialize~~ set the initial state of the puzzle.
2. Create stack to store nodes to be explored.
3. Calculate the Manhattan distance for the initial state.
4. add the manhattan distance to the stack

DFS

Start_state = [. . . .]
goal_state = [. . . .]

stack = push(start_state)
visited_set = { }
moves = 0
if (i, j)
{
visited_set = add (current_state)
if (current_state = = goal_state)
return moves

Code:

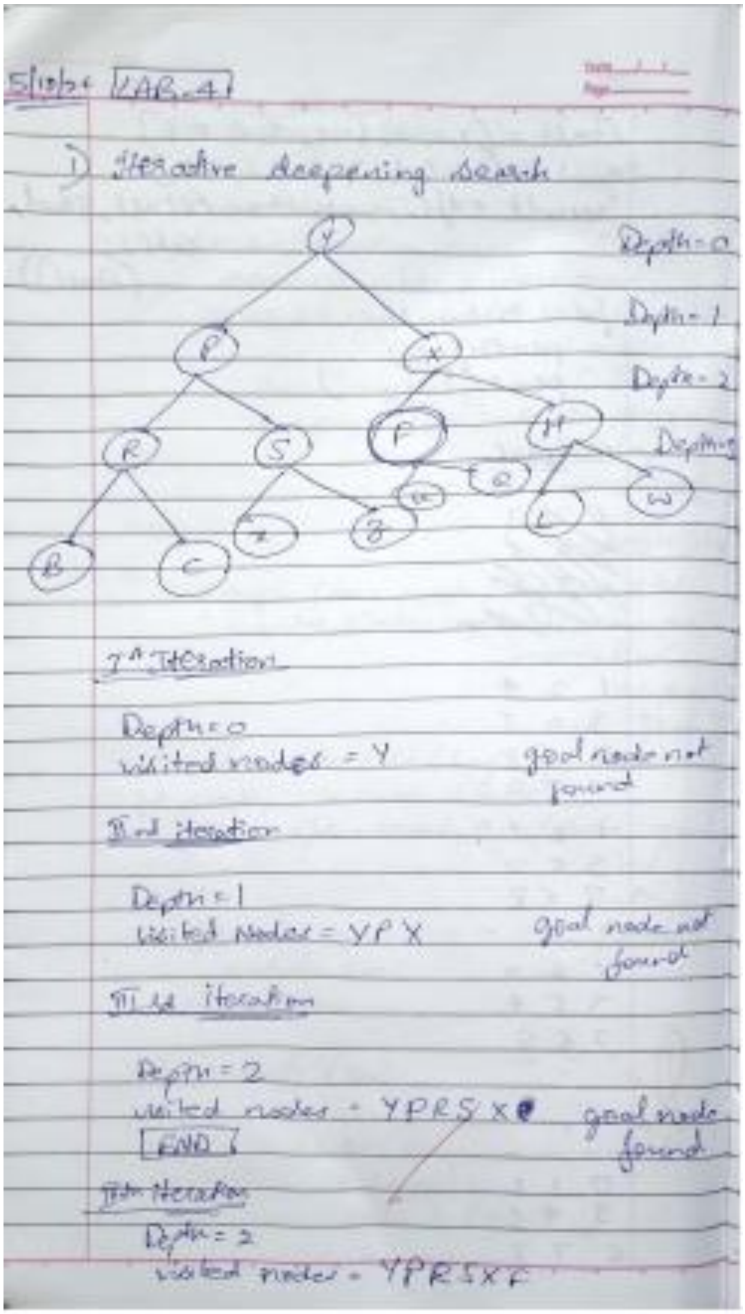
```
from collections import deque

def is_goal(state, goal_state):
    return state == goal_state

def get_neighbors(state):
    neighbors = []
    index = state.index(0)
    row, col = divmod(index, 3)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_row, new_col = row + move[0], col + move[1]
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
            neighbors.append(tuple(new_state))
    return neighbors

def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

def dfs(start, goal):
    visited = set()
    stack = [(start, [])]
    while stack:
        current_state, path = stack.pop()
        if current_state in visited:
            continue
        visited.add(current_state)
        if is_goal(current_state, goal):
            return path + [current_state]
        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                stack.append((neighbor, path + [current_state]))
    return None
```

```
def input_puzzle(prompt):
    print(prompt)
    puzzle = []
    for i in range(3):
        row = input(f"Enter row {i + 1} (3 numbers separated by spaces): ").split()
        puzzle.extend([int(x) for x in row])
    return tuple(puzzle)
def select_goal_state():
    print("Select a goal state:")
    print("1. Goal State:")
    print("   0 1 2")
    print("   3 4 5")
    print("   6 7 8")
    print("2. Goal State:")
    print("   1 2 3")
    print("   4 5 6")
    print("   7 8 0")
    choice = input("Enter 1 or 2: ")
    if choice == '1':
        return (0, 1, 2, 3, 4, 5, 6, 7, 8)
    else:
        return (1, 2, 3, 4, 5, 6, 7, 8, 0)

start_state = input_puzzle("Enter the start state (use 0 for the blank space):")
goal_state = select_goal_state()
print("\nSolving using DFS...")
dfs_solution = dfs(start_state, goal_state)
if dfs_solution:
    print("DFS Solution found! Steps:")
    for i, step in enumerate(dfs_solution):
        print(f"Step {i + 1}:")
        print_state(step)
else:
    print("No solution found using DFS.")
```

Output:

```
Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using DFS
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

Implement iddfs:
Algorithm:



CODE:

```
def depth_limited_search(node, goal, depth, graph):
    """
    Perform Depth Limited Search to find the goal node.
    """
    if node == goal:
```

```
        return True
    if depth <= 0:
        return False
    for child in graph.get(node, []):
        if depth_limited_search(child, goal, depth - 1, graph):
    return True return
    False

def iterative_deepening_dfs(start, goal, max_depth, graph):
    """
    Perform Iterative Deepening Depth-First Search (IDDFS).
    """
    for depth in range(max_depth + 1):
        print(f"Depth: {depth}")
        if depth_limited_search(start, goal, depth, graph):
            return True
    return False

# Input graph from the user
graph = {}
num_edges = int(input("Enter the number of edges in the graph: "))
print("Enter the edges in the format 'node1 node2':")
for _ in range(num_edges):
    node1, node2 = input().split()
    if node1 not in graph:
        graph[node1] = []
    graph[node1].append(node2)

# Input start node, goal node, and maximum depth
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")
max_depth = int(input("Enter the maximum depth: "))

# Perform IDDFS
found = iterative_deepening_dfs(start_node, goal_node, max_depth, graph)
if found:
    print(f"Goal node '{goal_node}' found!")
else:
    print(f"Goal node '{goal_node}' not found within depth {max_depth}.")
```

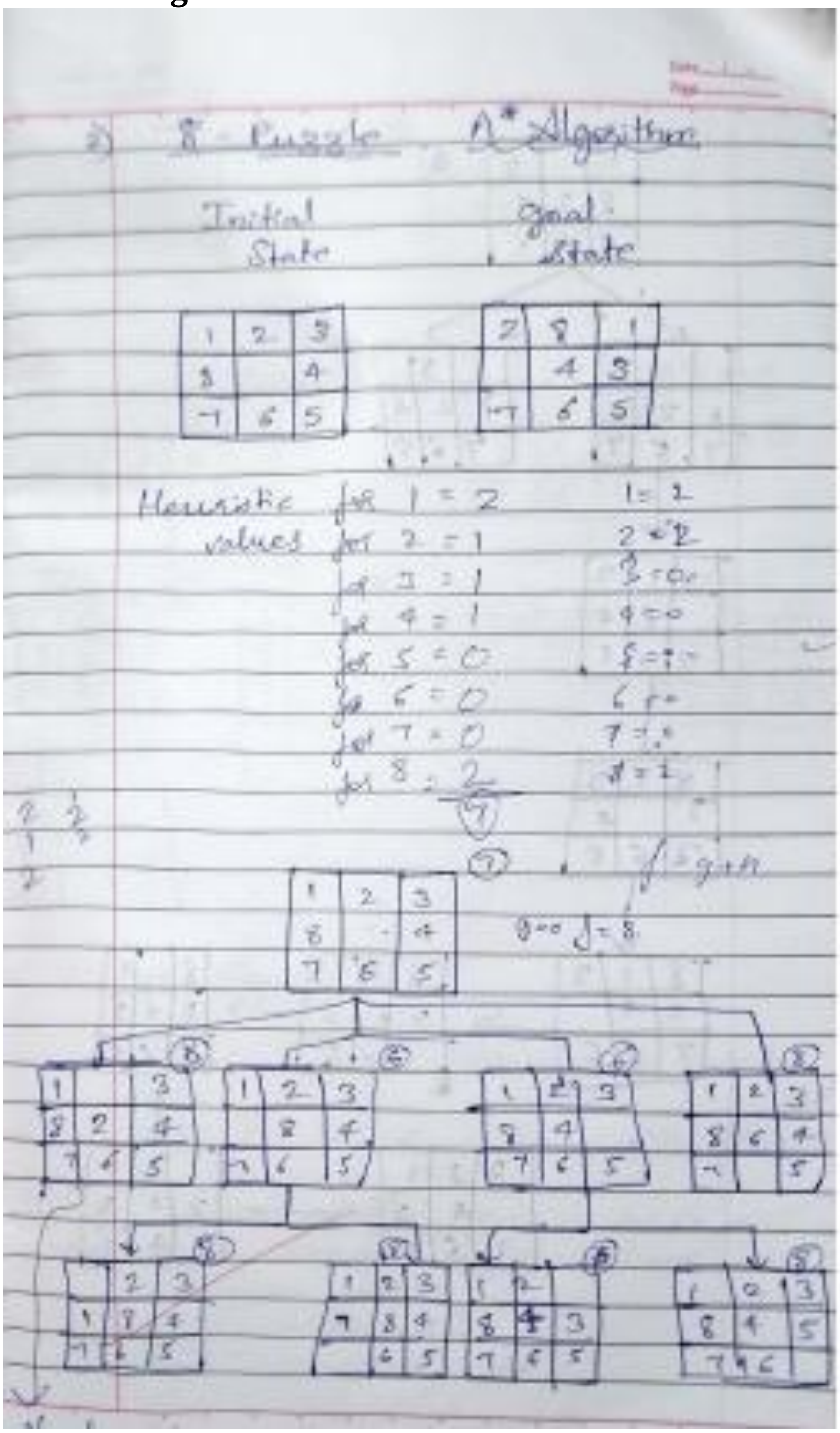
OUTPUT:

```
Yashraj Sinha (1BM22CS335)
8-Puzzle Solver Using Iterative Deepening Search
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

WEEK 3 8-10-2024

For 8-puzzle A* implementation, to calculate, f(n):

1. Algorithm:



Code:

```
import heapq

def is_goal(state, goal_state):
    return state == goal_state

def get_neighbors(state):
    neighbors = []
    index = state.index(0)
    row, col = divmod(index, 3)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_row, new_col = row + move[0], col + move[1]
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
            neighbors.append(tuple(new_state))
    return neighbors

def misplaced_tiles(state, goal):
    return sum(1 for i in range(9) if state[i] != 0 and state[i] != goal[i])

def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

def a_star_level_wise(start, goal, heuristic):
    priority_queue = []
    heapq.heappush(priority_queue, (0, 0, start, []))
    visited = set()
    print("Level-wise output:")

    while priority_queue:
        f_n, g_n, current_state, path = heapq.heappop(priority_queue)
        if current_state in visited:
            continue
        visited.add(current_state)
        print(f"\nLevel {g_n} (g(n) = {g_n}):")
        print(f"f(n) = {f_n}, h(n) = {heuristic(current_state, goal)}")
        print_state(current_state)
```

```
        if is_goal(current_state, goal):
            return path + [current_state]
        neighbors = get_neighbors(current_state)
        for neighbor in neighbors:
            if neighbor not in visited:
                g_new = g_n + 1
                h_new = heuristic(neighbor, goal)
                f_new = g_new + h_new
                print(f"    Adjacent Node (g(n) = {g_new}, h(n) = {h_new}, f(n) = {f_new}):")
                print_state(neighbor)
                heapq.heappush(priority_queue, (f_new, g_new, neighbor, path + [current_state]))
    return None

def input_puzzle(prompt):
    print(prompt)
    puzzle = []
    for i in range(3):
        row = input(f"Enter row {i + 1} (3 numbers separated by spaces): ").split()
        puzzle.extend([int(x) for x in row])
    return tuple(puzzle)
start_state = input_puzzle("Enter the start state (use 0 for the blank space):")
goal_state = input_puzzle("Enter the goal state (use 0 for the blank space):")
heuristic = misplaced_tiles
print("\nSolving using A* Search with level-wise output...")
a_star_solution = a_star_level_wise(start_state, goal_state, heuristic)
if a_star_solution:
    print("A* Solution found! Steps:")
    for i, step in enumerate(a_star_solution):
        print(f"Step {i + 1}:")
```

```
Enter the start state (use 0 for the blank space):
Enter row 1 (3 numbers separated by spaces): 2 8 3
Enter row 2 (3 numbers separated by spaces): 1 6 4
Enter row 3 (3 numbers separated by spaces): 7 0 5
Enter the goal state (use 0 for the blank space):
Enter row 1 (3 numbers separated by spaces): 1 2 3
Enter row 2 (3 numbers separated by spaces): 8 0 4
Enter row 3 (3 numbers separated by spaces): 7 6 5

Solving using A* Search with level-wise output...
Level-wise output:

Level 0 (g(n) = 0):
f(n) = 0, h(n) = 4
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

    Adjacent Node (g(n) = 1, h(n) = 3, f(n) = 4):
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 1, h(n) = 5, f(n) = 6):
(2, 8, 3)
(1, 6, 4)
(0, 7, 5)
```

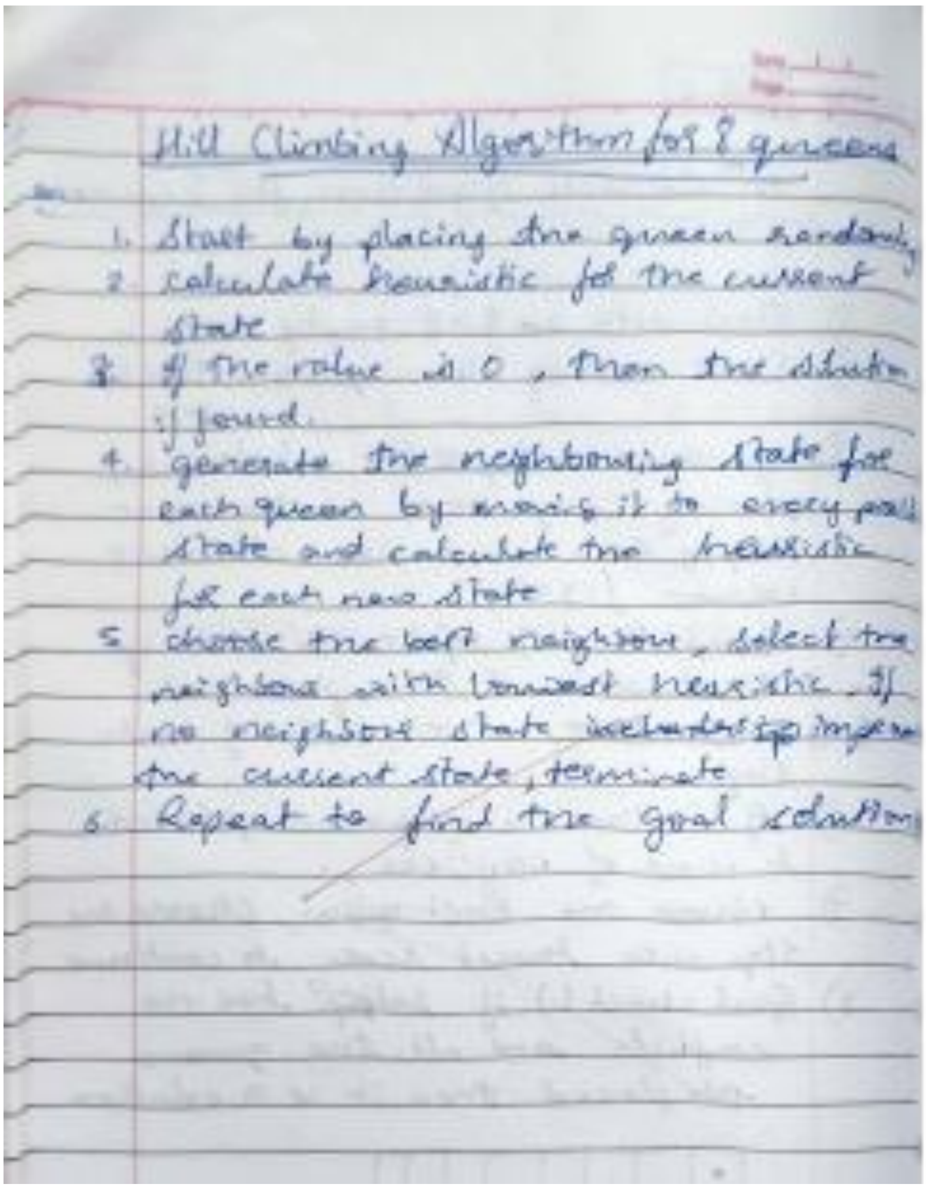
```
Level 1 (g(n) = 1):
f(n) = 4, h(n) = 3
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 2, h(n) = 3, f(n) = 5):
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 2, h(n) = 3, f(n) = 5):
(2, 8, 3)
(0, 1, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 2, h(n) = 4, f(n) = 6):
(2, 8, 3)
(1, 4, 0)
(7, 6, 5)


Level 2 (g(n) = 2):
f(n) = 5, h(n) = 3
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 3, h(n) = 2, f(n) = 5):
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 3, h(n) = 4, f(n) = 7):
(2, 3, 0)
(1, 8, 4)
(7, 6, 5)
```

```
Level 3 (g(n) = 3):
f(n) = 5, h(n) = 2
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 4, h(n) = 1, f(n) = 5):
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)


Level 4 (g(n) = 4):
f(n) = 5, h(n) = 1
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

    Adjacent Node (g(n) = 5, h(n) = 2, f(n) = 7):
(1, 2, 3)
(7, 8, 4)
(0, 6, 5)

    Adjacent Node (g(n) = 5, h(n) = 0, f(n) = 5):
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)


Level 5 (g(n) = 5):
f(n) = 5, h(n) = 0
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

WEEK 4   15-10-2024

Implement Hill Climbing search algorithm to solve N-Queens problem.

Algorithm:



Hill Climbing Algorithm for 8 queens

1. Start by placing the queen randomly
2. calculate heuristic for the current state
3. If the value is 0, then the solution is found.
4. generate the neighbouring state for each queen by moving it to every possible state and calculate the heuristic for each new state
5. choose the best neighbour, select the neighbour with lowest heuristic. If no neighbour state [illegible] to improve the current state, terminate
6. Repeat to find the goal solution

Code:

```
def print_board(state):
    n = len(state)
    board = [['.' for _ in range(n)] for _ in range(n)]
    for col, row in enumerate(state):
        board[row][col] = 'Q'
    for row in board:
        print(' '.join(row))
    print()
def calculate_cost(state):
    cost = 0
    n = len(state)

    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost
def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            new_state = state.copy()
            new_state[i], new_state[j] = new_state[j], new_state[i]  # Swap row positions of two
queens
            neighbors.append(new_state)
    return neighbors
def hill_climbing_4_queens(initial_state):
    n = 4

    current_state = initial_state
    current_cost = calculate_cost(current_state)

    print(f"Initial state: {current_state}, Cost: {current_cost}")
    print_board(current_state)

    steps = 0
    while current_cost != 0:
        steps += 1
```

```
        neighbors = generate_neighbors(current_state)
        neighbor_costs = [calculate_cost(neighbor) for neighbor in neighbors]

        min_cost = min(neighbor_costs)
        best_neighbor = neighbors[neighbor_costs.index(min_cost)]

        print(f"Step {steps}: Best neighbor: {best_neighbor}, Cost: {min_cost}")
        print_board(best_neighbor)

        if min_cost < current_cost:
            current_state = best_neighbor
            current_cost = min_cost
        else:
            print("Stuck at local maximum.")
            break

    if current_cost == 0:
        print("Solution found:")
        print_board(current_state)
    else:
        print("No solution found. Stuck at a local maximum.")
def get_user_input():
    print("Enter the initial positions of the queens on the board (0-based index for each column):")
    initial_state = []
    for col in range(4):
        row = int(input(f"Enter row position for column {col+1}: "))
        initial_state.append(row)
    return initial_state
def main():
    confirm = input("Run Hill Climbing for the 4-Queens problem? (yes/no): ").strip().lower()
    if confirm == 'yes':
        initial_state = get_user_input()
        hill_climbing_4_queens(initial_state)
    else:
        print("Operation cancelled.")
if _name___== "_main_":
    main()
```

Output:

```
Enter the initial positions of the queens on the board (0-based index for each column):
Enter row position for column 1: 0
Enter row position for column 2: 0
Enter row position for column 3: 0
Enter row position for column 4: 0
Initial state: [0, 0, 0, 0], Cost: 6
Q Q Q Q
. . . .
. . . .
. . . .

Step 1: Best neighbor: [0, 3, 0, 0], Cost: 3
Q . Q Q
. . . .
. . . .
. Q . .

Step 2: Best neighbor: [1, 3, 0, 0], Cost: 1
. . Q Q
Q . . .
. . . .
. Q . .

Step 3: Best neighbor: [1, 3, 0, 2], Cost: 0
. . Q .
Q . . .
. . . Q
. Q . .

Solution found:
. . Q .
Q . . .
. . . Q
. Q . .
```

WEEK 5 22-10-2024

Simulated Annealing to Solve 8-Queens problem

Algorithm:

22/10/24 LAB-5

Simulated Annealing

Algorithm

1) Set initial temperature ~~to~~ (T)
2) ~~Set~~ generate an initial solution (S)
3) calculate the energy (E) of the solution
4) Define a cooling schedule that determines how the temp decrease over time
5) While temp > 0 do
   for i to maxiteration do
   newstate = generateneighbor(currentState)
   newenergy = evaluate (new state)
   energyDiff = newenergy - currentEnergy
   if ~~[illegible]~~ energyDiff < 0 then
   Accept the new state
   currentstate = newstate
   currentEnergy = newenergy
   if currentEnergy < Best Energy then
   bestState = currentState
   bestEnergy = currentEnergy
   ~~[illegible]~~ temp = temp * coolingRate
   ~~[illegible]~~
6) Return BestState

22/10/24

Code:

```
import random
import math

def calculate_conflicts(board):
    n = len(board)
    conflicts = 0

    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def simulated_annealing(board, max_steps=1000, initial_temp=100, cooling_rate=0.99):
    current_conflicts = calculate_conflicts(board)
    temperature = initial_temp
    n = len(board)

    for step in range(max_steps):
        if current_conflicts == 0:
            return board

        col = random.randint(0, n - 1)
        new_row = random.randint(0, n - 1)

        new_board = board[:]
        new_board[col] = new_row
        new_conflicts = calculate_conflicts(new_board)

        delta = new_conflicts - current_conflicts
        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / temperature):
            board = new_board
            current_conflicts = new_conflicts

        temperature *= cooling_rate

    return None

def main():
    n = int(input("Enter the number of queens (default is 8): ") or 8)
```

```
    print(f"Enter the positions of the queens as an array of size {n}:")
    print(f"(Example: 0,4,7,5,2,6,1,3 or space-separated values)")
    input_str = input().strip()
    if ',' in input_str:
        board = list(map(int, input_str.split(',')))
    else:
        board = list(map(int, input_str.split()))

    if len(board) != n:
        print("Error: The number of positions must match the number of queens.")
        return

    solution = simulated_annealing(board)

    if solution:
        print("\nSolution found:")
        for row in range(n):
            line = ['.'] * n
            line[solution[row]] = 'Q'
            print(" ".join(line))
    else:
        print("No solution found.")

if _name___== "_main_":
    main()
```

Output:

```
Enter the number of queens (default is 8): 8
Enter the positions of the queens as an array of size 8:
(Example: 0,4,7,5,2,6,1,3 or space-separated values)
0 4 7 5 2 6 1 3

Solution found:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

WEEK 6 29-10-2024

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:

12/11/24 LAB-7

Entailment using literals

KB
Alice is mom of bob
bob is father of charlie
A father is a parent
A mother is a parent
All parents have children
If someone is a parent, their children are siblings.
Alice is married to David

Hypothesis:
Charlie is a sibling of bob

Entailment Reasoning:
Since Alice is mother and she becomes a parent
Since bob is a father he also becomes a parent
Given father and mother are parents & according to "If someone is parents their children are siblings", so finally charlie and bob are siblings.

$A \rightarrow B$ (mother)
$B \rightarrow C$ (father)
$F \rightarrow D$ (parent)
$M \rightarrow D$ (parent)
$P \rightarrow S$ (if parent, siblings)
$A \wedge B \rightarrow Q$ (If Alice is mother of bob & bob is father, charlie [illegible])

check for entailment:

1. If A (Alice is mother of Bob) is true then B (Bob is the father of charlie) must also be true ($A \rightarrow$ [illegible]

2. If B is true then C (Bob is a parent) must be true ($F \rightarrow P$) and M (Alice is a parent) must also be true ($M \rightarrow$ [illegible]

3. If both Alice & charlie are parents (i.e. A & F are true) then S (their children are siblings) must be true ($P \rightarrow S$)

4. Since S is true, the hypothesis Q ("Charlie is sibling of Bob") is true

Conclusion:
Using Propositional logic, we can conclude the hypothesis "Charlie is a sibling of Bob" is entailed by KB.

output

The hypothesis "Charlie is a sibling of Bob" is TRUE

Code:

```
combinations = [
    (True, True, True), (True, True, False),
    (True, False, True), (True, False, False),
    (False, True, True), (False, True, False),
    (False, False, True), (False, False, False)
]
variable = {'p': 0, 'q': 1, 'r': 2}
kb = ''
q = ''
priority = {'~': 3, 'v': 1, '^': 2}
def input_rules():
    global kb, q
    kb = input("Enter rule: ")
    q = input("Enter the Query: ")
def entailment():
    global kb, q
    print('*' * 10 + "Truth Table Reference" + '*' * 10)
    print('p', 'q', 'r', 'kb', 'query')
    print('*' * 10)
    entails = True  # Assumption: The Knowledge Base entails the query
    for comb in combinations:
        s = evaluatePostfix(toPostfix(kb), comb)
        f = evaluatePostfix(toPostfix(q), comb)
        print(comb[0], comb[1], comb[2], s, f)
        print('-' * 10)
        if s and not f:
            entails = False  # Counterexample found
    return entails
def isOperand(c):
    return c.isalpha() and c != 'v'
def isLeftParanthesis(c):
    return c == '('
def isRightParanthesis(c):
    return c == ')'
def isEmpty(stack):
    return len(stack) == 0
def peek(stack):
    return stack[-1]
```

```
def hasLessOrEqualPriority(c1, c2):
    try:
        return priority[c1] <= priority[c2]
    except KeyError:
        return False
def toPostfix(infix):
    stack = []
    postfix = ''
    for c in infix:
        if isOperand(c):
            postfix += c
        else:
            if isLeftParanthesis(c):
                stack.append(c)
            elif isRightParanthesis(c):
                operator = stack.pop()
                while not isLeftParanthesis(operator):
                    postfix += operator
                    operator = stack.pop()
            else:
                while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):
                    postfix += stack.pop()
                stack.append(c)
    while (not isEmpty(stack)):
        postfix += stack.pop()
    return postfix
def evaluatePostfix(exp, comb):
    stack = []
    for i in exp:
        if isOperand(i):
            stack.append(comb[variable[i]])
        elif i == '~':
            val1 = stack.pop()
            stack.append(not val1)
        else:
            val1 = stack.pop()
            val2 = stack.pop()
            stack.append(_eval(i, val2, val1))
    return stack.pop()
def _eval(i, val1, val2):
```

```
    if i == '^':
        return val2 and val1
    return val2 or val1
input_rules()
ans = entailment()
if ans:
    print("The Knowledge Base entails the query.")
else:
    print("The Knowledge Base does not entail the query.")
```

Output:

```
Enter rule: p^q
Enter the Query: r
**********Truth Table Reference**********
kb alpha
**********
True True
----------
True False
----------
The Knowledge Base does not entail query
```

WEEK 7 **12-11-2024**

Implement unification in first order logic.

Algorithm:

12/11/24 LAB-8

First-order logic (FOL)

Scenario:

All dogs are mammals. Fido is a dog. Therefore, Fido is a mammal"

FOL Representation:

1. Define the predicates:

Dog(x) : x is a dog

Mammal(x) : x is a mammal

2. Define the constant:

Fido : a specific dog

3. Represent the statements in FOL:

$\forall x\, (Dog(x) \rightarrow Mammal(x))$

// All dogs are mammals

Dog(Fido) // Fido is a dog

Conclusion:

Mammal(Fido)

output

Fido is a mammal

Code:

```
def is_variable(x):
    """Check if x is a variable."""
    return isinstance(x, str) and x[0].islower()

def unify(x, y, subst):
    """Unify two terms x and y under a given substitution subst."""
    print(f"Comparing: {x} with {y}")
    if subst is None:
        return None
    elif x == y:
        print(f"Both are equal: {x} == {y}")
        return subst
    elif is_variable(x):
        return unify_variable(x, y, subst)
    elif is_variable(y):
        return unify_variable(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        for xi, yi in zip(x, y):
            subst = unify(xi, yi, subst)
            if subst is None:
                return None
        return subst
    else:
        print(f"Cannot unify {x} and {y}")
        return None

def unify_variable(var, x, subst):
    """Handle variable unification."""
    if var in subst:
        print(f"Variable {var} is already in substitution. Resolving with {subst[var]}.")
        return unify(subst[var], x, subst)
    elif occurs_check(var, x, subst):
        print(f"Occurs check failed: {var} occurs in {x}.")
        return None  # Avoid infinite loops in recursive substitutions
    else:
        print(f"Adding substitution: {var} -> {x}")
        new_subst = subst.copy()
        new_subst[var] = x
        return new_subst
def occurs_check(var, x, subst):
    """Check if var occurs in x to avoid infinite substitution."""
```

```
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, subst) for xi in x)
    elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    else:
        return False

def parse_sentence_to_expression(sentence):
    """Convert an English sentence to a logical expression."""
    sentence = sentence.strip().replace("(", " ( ").replace(")", " ) ").replace(",", " , ")
    tokens = sentence.split()
    stack = []
    current = []
    for token in tokens:
        if token == "(":
            stack.append(current)
            current = []
        elif token == ")":
            if stack:
                last = stack.pop()
                last.append(tuple(current))
                current = last
        elif token == ",":
            continue
        else:
            current.append(token)
    return tuple(current) if len(current) == 1 else tuple(current)

def unification_with_explanation(expr1, expr2):
    """Perform unification on two expressions with step-by-step explanation."""
    print("\nStarting Unification Process...\n")
    subst = unify(expr1, expr2, {})
    if subst is not None:
        print("\nUnification Successful!")
        print("Substitution:", subst)
    else:
        print("\nUnification Failed!")
# Input from the user
print("Enter the logical expressions in English-like format.")
print("Example: Eats(x, Apple)")
```

```
sentence1 = input("Enter the first expression: ")
sentence2 = input("Enter the second expression: ")
# Parse sentences
expr1 = parse_sentence_to_expression(sentence1)
expr2 = parse_sentence_to_expression(sentence2)
# Perform unification with explanation
unification_with_explanation(expr1, expr2)
```

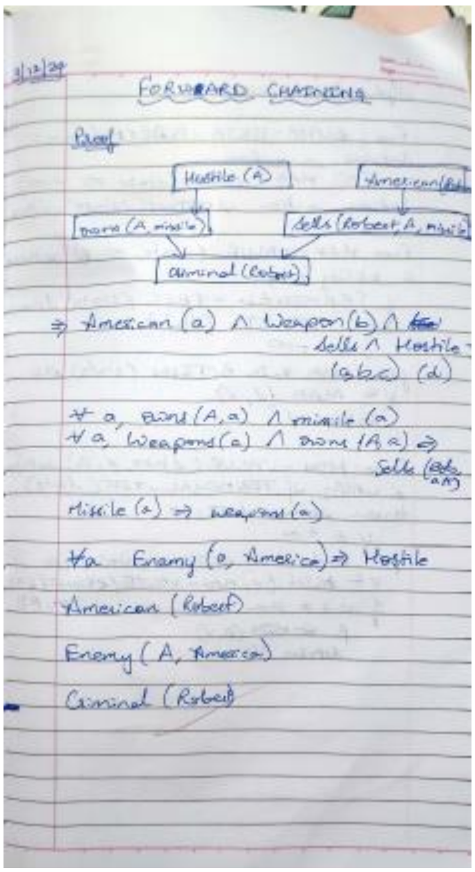
Output:

```
Enter the logical expressions in English-like format.
Example: Eats(x, Apple)
Enter the first expression: Works(x,y)
Enter the second expression: Works(Apple,Banana)

Starting Unification Process...

Comparing: ('Works', ('x', 'y')) with ('Works', ('Apple', 'Banana'))
Comparing: Works with Works
Both are equal: Works == Works
Comparing: ('x', 'y') with ('Apple', 'Banana')
Comparing: x with Apple
Adding substitution: x -> Apple
Comparing: y with Banana
Adding substitution: y -> Banana

Unification Successful!
Substitution: {'x': 'Apple', 'y': 'Banana'}
```

WEEK 8 19-11-2024

Create a KB consisting of first order logic statements and prove the following reasoning.

Algorithm:



Code:

```
class ForwardChainingFOL:
    def _init_(self):
        self.facts = set()  # Set of known facts
        self.rules = []  # List of rules in the form (premises, conclusion)
    def add_fact(self, fact):
        self.facts.add(fact)
    def add_rule(self, premises, conclusion):
```

```
        self.rules.append((premises, conclusion))
    def unify(self, fact1, fact2):
        """
        Unifies two facts if possible. Returns a substitution dictionary or None if unification fails.
        """
        if fact1 == fact2:
            return {}  # No substitution needed
        if "(" in fact1 and "(" in fact2:
            # Split into predicate and arguments
            pred1, args1 = fact1.split("(", 1)
            pred2, args2 = fact2.split("(", 1)
            args1 = args1[:-1].split(",")
            args2 = args2[:-1].split(",")
            if pred1 != pred2 or len(args1) != len(args2):
                return None
            # Unify arguments
            substitution = {}
            for a1, a2 in zip(args1, args2):
                if a1 != a2:
                    if a1.islower():  # a1 is a variable
                        substitution[a1] = a2
                    elif a2.islower():  # a2 is a variable
                        substitution[a2] = a1
                    else:  # Both are constants and different
                        return None
            return substitution
        return None
    def apply_substitution(self, fact, substitution):
        """
        Applies a substitution to a fact and returns the substituted fact.
        """
        if "(" in fact:
            pred, args = fact.split("(", 1)
            args = args[:-1].split(",")
            substituted_args = [substitution.get(arg, arg) for arg in args]
            return f"{pred}({','.join(substituted_args)})"
        return fact
    def forward_chain(self, goal):
        iteration = 1
        while True:
```

```
            new_facts = set()
            print(f"\n=== Iteration {iteration} ===")
            print("Known Facts:")
            for fact in self.facts:
                print(f"  - {fact}")
            print("\nApplying rules...")
            rule_triggered = False
            for premises, conclusion in self.rules:
                substitutions = [{}]
                for premise in premises:
                    new_substitutions = []
                    for fact in self.facts:
                        for sub in substitutions:
                            unified = self.unify(self.apply_substitution(premise, sub), fact)
                            if unified is not None:
                                new_substitutions.append({**sub, **unified})
                    substitutions = new_substitutions
                for sub in substitutions:
                    inferred_fact = self.apply_substitution(conclusion, sub)
                    if inferred_fact not in self.facts:
                        rule_triggered = True
                        print(f"Rule triggered: {premises} → {conclusion}")
                        print(f"  New fact inferred: {inferred_fact}")
                        new_facts.add(inferred_fact)
            if not new_facts:
                if not rule_triggered:
                    print("No rules triggered in this iteration.")
                print("No new facts inferred in this iteration.")
                break
            self.facts.update(new_facts)
            if goal in self.facts:
                print(f"\nGoal {goal} reached!")
                return True
            iteration += 1
        print("\nGoal not reached.")
        return False
# Problem setup
fc = ForwardChainingFOL()
# Facts
fc.add_fact("American(Robert)")
```

```
fc.add_fact("Enemy(A,America)")
fc.add_fact("Owns(A,T1)")
fc.add_fact("Missile(T1)")
# Rules
fc.add_rule(["Missile(T1)"], "Weapon(T1)")
fc.add_rule(["Enemy(A,America)"], "Hostile(A)")
fc.add_rule(["Missile(p)", "Owns(A,p)"], "Sells(Robert,p,A)")
fc.add_rule(["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"], "Criminal(p)")
# Goal
goal = "Criminal(Robert)"
# Perform forward chaining
if fc.forward_chain(goal):
    print(f"\nFinal result: Goal achieved: {goal}")
else:
    print("\nFinal result: Goal not achieved.")
```

Output:

```
=== Iteration 1 ===
Known Facts:
  - American(Robert)
  - Missile(T1)
  - Owns(A,T1)
  - Enemy(A,America)

Applying rules...
Rule triggered: ['Missile(T1)'] → Weapon(T1)
  New fact inferred: Weapon(T1)
Rule triggered: ['Enemy(A,America)'] → Hostile(A)
  New fact inferred: Hostile(A)
Rule triggered: ['Missile(p)', 'Owns(A,p)'] → Sells(Robert,p,A)
  New fact inferred: Sells(Robert,T1,A)

=== Iteration 2 ===
Known Facts:
  - Hostile(A)
  - Sells(Robert,T1,A)
  - American(Robert)
  - Missile(T1)
  - Enemy(A,America)
  - Weapon(T1)
  - Owns(A,T1)

Applying rules...
Rule triggered: ['American(p)', 'Weapon(q)', 'Sells(p,q,r)', 'Hostile(r)'] → Criminal(p)
  New fact inferred: Criminal(Robert)

Goal Criminal(Robert) reached!

Final result: Goal achieved: Criminal(Robert)
```

WEEK 9 03-12-2024

Write a proof tree generated using CNF.

Algorithm:

26/11/24 LAB-9

Unification

Algorithm

1. Check for a base case:
   If both expressions are identical, return an empty substitution.

2. Check for a variable case:
   If one expression is a variable.
   — If the variable is already bound to another term, recursively unify the bound term with the other expression.
   — otherwise, create a new binding between the variable and the other expression.

3. Check for a Constant Case:
   If both expressions are constants they must be identical. If not, fai

4. Check for a Complex term case:
   If the main functions or operations of both expressions are different fail

Code:

```
def negate(literal):
```

```
    """Return the negation of a literal."""
    if isinstance(literal, tuple) and literal[0] == "not":
        return literal[1]
    else:
        return ("not", literal)
def resolve(clause1, clause2):
    """Return the resolvent of two clauses."""
    resolvents = set()
    for literal1 in clause1:
        for literal2 in clause2:
            if literal1 == negate(literal2):
                resolvent = (clause1 - {literal1}) | (clause2 - {literal2})
                print(f"    Resolving literal: {literal1} with {literal2}")
                print(f"    Resulting Resolvent: {resolvent}")
                resolvents.add(frozenset(resolvent))
    return resolvents
def resolution_algorithm(KB, query):
    """Perform the resolution algorithm to check if the query can be proven."""
    print("\n--- Step-by-Step Resolution Process ---")
    negated_query = negate(query)
    KB.append(frozenset([negated_query]))
    print(f"Negated Query Added to KB: {negated_query}")
    clauses = set(KB)
    step = 1
    while True:
        new_clauses = set()
        print(f"\nStep {step}: Resolving Clauses")
        for c1 in clauses:
            for c2 in clauses:
                if c1 != c2:
                    print(f"  Resolving clauses: {c1} and {c2}")
                    resolvent = resolve(c1, c2)
                    for res in resolvent:
                        if frozenset([]) in resolvent:
                            print("\nEmpty clause derived! The query is provable.")
                            return True
                        new_clauses.add(res)
        if new_clauses.issubset(clauses):
            print("\nNo new clauses can be derived. The query is not provable.")
            return False
        clauses.update(new_clauses)
        step += 1
```

```
KB = [
    frozenset([("not", "food(x)"), ("likes", "John", "x")]),  # 1
    frozenset([("food", "Apple")]),                           # 2
    frozenset([("food", "vegetables")]),                      # 3
    frozenset([("not", "eats(y, z)"), ("killed", "y"), ("food", "z")]),  # 4
    frozenset([("eats", "Anil", "Peanuts")]),                 # 5
    frozenset([("alive", "Anil")]),                           # 6
    frozenset([("not", "eats(Anil, w)"), ("eats", "Harry", "w")]),  # 7
    frozenset([("killed", "g"), ("alive", "g")]),             # 8
    frozenset([("not", "alive(k)"), ("not", "killed(k)")]),   # 9
    frozenset([("likes", "John", "Peanuts")])                 # 10
]
query = ("likes", "John", "Peanuts")
result = resolution_algorithm(KB, query)
if result:
    print("\nQuery is provable.")
else:
    print("\nQuery is not provable.")
```

Output:

```
--- Step-by-Step Resolution Process ---
Negated Query Added to KB: ('not', ('likes', 'John', 'Peanuts'))

Step 1: Resolving Clauses
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('alive', 'g'),
 ('killed', 'g')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('eats', 'Anil'
, 'Peanuts')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('food', 'z'),
('killed', 'y'), ('not', 'eats(y, z)')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('alive', 'Anil
')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('food', 'veget
ables')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('food', 'Apple
')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('not', 'killed
(k)'), ('not', 'alive(k)')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('not', ('likes
', 'John', 'Peanuts'))})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('not', 'eats(A
nil, w)'), ('eats', 'Harry', 'w')})
  Resolving clauses: frozenset({('likes', 'John', 'x'), ('not', 'food(x)')}) and frozenset({('likes', 'John
', 'Peanuts')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('likes', 'John', 'x'), ('
not', 'food(x)')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('eats', 'Anil', 'Peanuts'
)})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('food', 'z'), ('killed',
'y'), ('not', 'eats(y, z)')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('alive', 'Anil')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('food', 'vegetables')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('food', 'Apple')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('not', 'killed(k)'), ('no
t', 'alive(k)')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('not', ('likes', 'John',
```

```
'Peanuts'))})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('not', 'eats(Anil, w)'),
('eats', 'Harry', 'w')})
  Resolving clauses: frozenset({('alive', 'g'), ('killed', 'g')}) and frozenset({('likes', 'John', 'Peanuts
')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('likes', 'John', 'x'), ('not'
, 'food(x)')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('alive', 'g'), ('killed', 'g'
)})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('food', 'z'), ('killed', 'y')
, ('not', 'eats(y, z)')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('alive', 'Anil')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('food', 'vegetables')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('food', 'Apple')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', 'killed(k)'), ('not',
'alive(k)')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', ('likes', 'John', 'Pea
nuts'))})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('not', 'eats(Anil, w)'), ('ea
ts', 'Harry', 'w')})
  Resolving clauses: frozenset({('eats', 'Anil', 'Peanuts')}) and frozenset({('likes', 'John', 'Peanuts')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('li
kes', 'John', 'x'), ('not', 'food(x)')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('al
ive', 'g'), ('killed', 'g')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('ea
ts', 'Anil', 'Peanuts')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('al
ive', 'Anil')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('fo
od', 'vegetables')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('fo
od', 'Apple')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('no
t', 'killed(k)'), ('not', 'alive(k)')})
  Resolving clauses: frozenset({('food', 'z'), ('killed', 'y'), ('not', 'eats(y, z)')}) and frozenset({('no
t', ('likes', 'John', 'Peanuts'))})
```

```
  Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('alive', 'Anil'
)})
  Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('food', 'vegeta
bles')})
  Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('food', 'Apple'
)})
  Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('not', ('likes'
, 'John', 'Peanuts'))})
  Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('not', 'eats(An
il, w)'), ('eats', 'Harry', 'w')})
  Resolving clauses: frozenset({('not', 'killed(k)'), ('not', 'alive(k)')}) and frozenset({('likes', 'John'
, 'Peanuts')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('likes', 'John', 'x
'), ('not', 'food(x)')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('alive', 'g'), ('ki
lled', 'g')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('eats', 'Anil', 'Pe
anuts')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('food', 'z'), ('kil
led', 'y'), ('not', 'eats(y, z)')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('alive', 'Anil')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('food', 'vegetables
')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('food', 'Apple')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('not', 'killed(k)')
, ('not', 'alive(k)')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('not', 'eats(Anil,
w)'), ('eats', 'Harry', 'w')})
  Resolving clauses: frozenset({('not', ('likes', 'John', 'Peanuts'))}) and frozenset({('likes', 'John', 'P
eanuts')})
    Resolving literal: ('not', ('likes', 'John', 'Peanuts')) with ('likes', 'John', 'Peanuts')
    Resulting Resolvent: frozenset()

Empty clause derived! The query is provable.

Query is provable.
```

WEEK 10 3-12-2024

Implement Alpha Beta pruning.

Algorithm:

Alpha-beta Search

func ALPHA-BETA-SEARCH (state)
returns an action
  $v \leftarrow$ MAX-VALUE (state, ∞, $+\infty$)
return action ACTIONS (state) with

Func MAX-VALUE (state, α, β) returns
a utility value
 if TERMINAL-TEST (state) then
return UTILITY (state)
  $v \leftarrow -\infty$
for each α in ACTION (state) do
 $v \leftarrow$ MAX (α, v)
  return v

Func MIN-VALUE (state, α, β) returns
a utility if TERMINAL-TEST (state)
then return UTILITY
  $v \leftarrow +\infty$
for each α in ACTIONS (state) do
 $v \leftarrow$ MIN (v, MAX-VALUE (RESULT(s, a), α, β))
 if $v \le \alpha$ then return v
  $\beta \leftarrow$ MIN (β, v)
  return v

Code:

```
class AlphaBetaPruning:
    def _init_(self):
        self.pruned_branches = []
    def alpha_beta(self, node, depth, alpha, beta, maximizing_player):
        if isinstance(node, int):
            return node
        if maximizing_player:
            max_eval = float('-inf')
            for child in node:
                eval = self.alpha_beta(child, depth - 1, alpha, beta, False)
                max_eval = max(max_eval, eval)
                alpha = max(alpha, eval)
                if beta <= alpha:
                    self.pruned_branches.append(child)
                    break
            return max_eval
        else:
            min_eval = float('inf')
            for child in node:
                eval = self.alpha_beta(child, depth - 1, alpha, beta, True)
                min_eval = min(min_eval, eval)
                beta = min(beta, eval)
                if beta <= alpha:
                    self.pruned_branches.append(child)
                    break
            return min_eval

    def run(self, game_tree):
        alpha = float('-inf')
        beta = float('inf')
        max_value = self.alpha_beta(game_tree, float('inf'), alpha, beta, True)
        return max_value, self.pruned_branches

def construct_tree_from_leaves(leaves):
    current_level = leaves
    while len(current_level) > 1:
        next_level = []
        for i in range(0, len(current_level), 2):
            if i + 1 < len(current_level):
```

```
                next_level.append([current_level[i], current_level[i + 1]])
            else:
                next_level.append(current_level[i])
        current_level = next_level
    return current_level[0]
def input_leaf_nodes():
    print("Enter the leaf nodes of the game tree separated by spaces (e.g., 3 5 6 9 1 4 7 10 11):")
    while True:
        try:
            leaves = list(map(int, input("Leaf nodes: ").split()))
            if len(leaves) >= 2:
                return leaves
            else:
                print("Please enter at least two leaf nodes.")
        except ValueError:
            print("Invalid input. Please enter integers only.")
if __name__ == "__main__":
    leaves = input_leaf_nodes()
    game_tree = construct_tree_from_leaves(leaves)
    abp = AlphaBetaPruning()
    final_max_value, pruned_branches = abp.run(game_tree)
    print(f"Final value of MAX node: {final_max_value}")
    print(f"Subtrees pruned: {pruned_branches}")
```

Output:

```
Enter the leaf nodes of the game tree separated by spaces (e.g., 3 5 6 9 1 4 7 10 11):
Leaf nodes: 10 9 14 18 5 4 50 3
Final value of MAX node: 10
Subtrees pruned: [14, [5, 4]]
```