VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sinchana R(1BM22CS278)

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
Sep-2024 to Jan-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 (23CS5PCAIN)" carried out by **Sinchana R(1BM22CS278),** 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. 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.

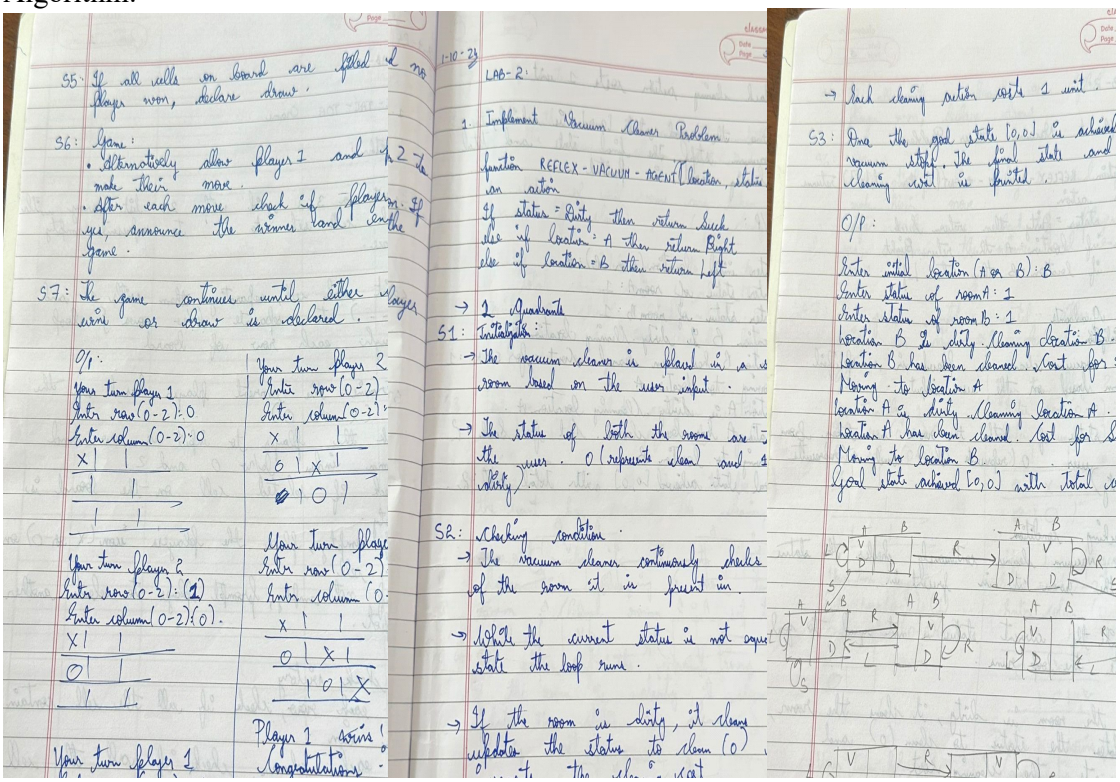
Saritha A N Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/sinchana-08/AI

Program 1
Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



Code:

TicTacToe

```
board = [[' ' for _ in range(3)] for _ in range(3)]
def draw_board():
    for row in board:
        print(' | '.join(row))
        print('---------')
def player_move(icon):
    if icon == 'X':
        number = 1
    elif icon == 'O':
        number = 2

    print("Your turn player {}".format(number))

    row = int(input("Enter row (0-2): "))
```

```
    col = int(input("Enter column (0-2): "))

    if board[row][col] == ' ':
        board[row][col] = icon
    else:
        print("That space is taken!")
def is_victory(icon):
    for row in board:
        if all(cell == icon for cell in row):
            return True
    for col in range(3):
        if all(board[row][col] == icon for row in range(3)):
            return True
    if all(board[i][i] == icon for i in range(3)) or \
       all(board[i][2-i] == icon for i in range(3)):
        return True

    return False
def is_draw():
    return all(cell != ' ' for row in board for cell in row)
def play_game():
    draw_board()
    while True:
        player_move('X')
        draw_board()
        if is_victory('X'):
            print("Player 1 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break
        player_move('O')
        draw_board()
        if is_victory('O'):
            print("Player 2 wins! Congratulations!")
            break
        elif is_draw():
            print("It's a draw!")
            break

play_game()
```

Output:

```
Your turn player 2
Enter row (0-2): 0
Enter column (0-2): 1
X | O | X
---------
  | O |
---------
O |   |
---------
Your turn player 1
Enter row (0-2): 1
Enter column (0-2): 0
X | O | X
---------
X | O |
---------
O |   |
---------
Your turn player 2
Enter row (0-2): 2
Enter column (0-2): 1
X | O | X
```

Vacuum World-2Quad:

```
class VacuumCleaner:
    def __init__(self, location, status):
        self.location = location
        self.status = status
        self.goal_state = [0, 0]
        self.cost = 0

    def clean(self):
        while self.status != self.goal_state:
            if self.location == 'A':
                if self.status[0] == 1:
                    print("Location A is Dirty. Cleaning Location A.")
                    self.status[0] = 0
                    self.cost += 1
                    print(f"Location A has been Cleaned. COST for SUCK: {self.cost}")
                else:
                    print("Location A is already clean.")
                print("Moving to Location B.")
                self.location = 'B'
```

```
        elif self.location == 'B':
            if self.status[1] == 1:
                print("Location B is Dirty. Cleaning Location B.")
                self.status[1] = 0
                self.cost += 1
                print(f"Location B has been Cleaned. COST for SUCK: {self.cost}")
            else:
                print("Location B is already clean.")
            print("Moving to Location A.")
            self.location = 'A'

        print(f"Goal state achieved: {self.status} with total cost: {self.cost}")

def get_user_input():
    location = input("Enter initial location (A or B): ").strip().upper()
    a_status = int(input("Enter status of Room A (0 for clean, 1 for dirty): "))
    b_status = int(input("Enter status of Room B (0 for clean, 1 for dirty): "))
    return location, [a_status, b_status]

def main():
    location, status = get_user_input()
    vacuum = VacuumCleaner(location, status)
    vacuum.clean()

main()
```

Output:

Output1

```
Enter initial location (A or B): A
Enter status of Room A (0 for clean, 1 for dirty): 1
Enter status of Room B (0 for clean, 1 for dirty): 0
Location A is Dirty. Cleaning Location A.
Location A has been Cleaned. COST for SUCK: 1
Moving to Location B.
Goal state achieved: [0, 0] with total cost: 1
```

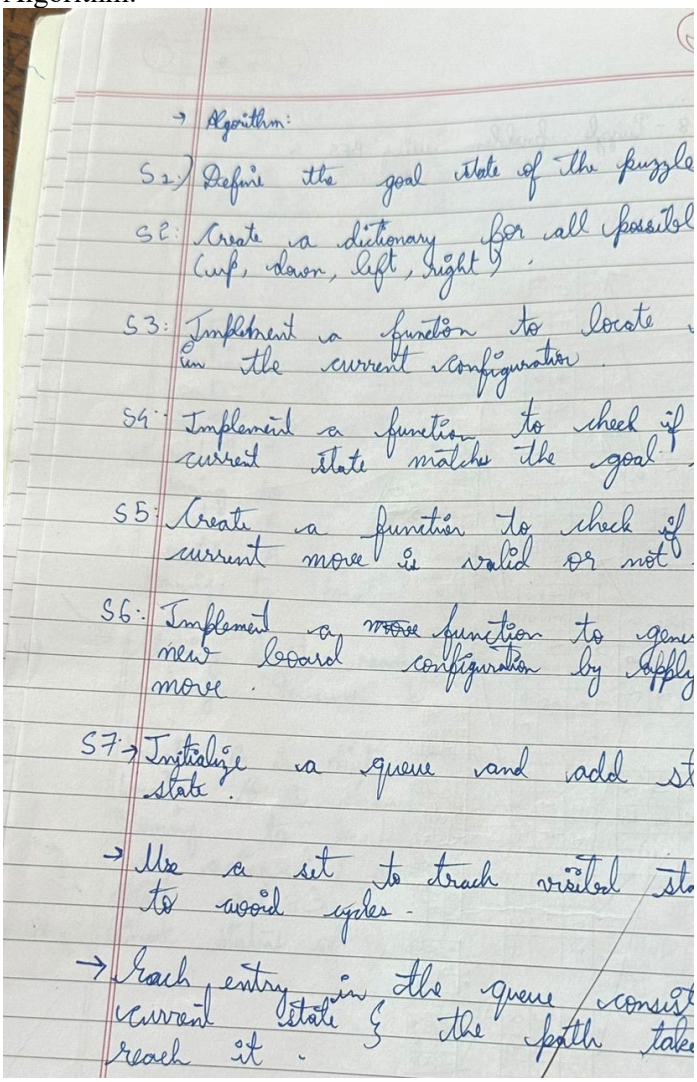
Output2

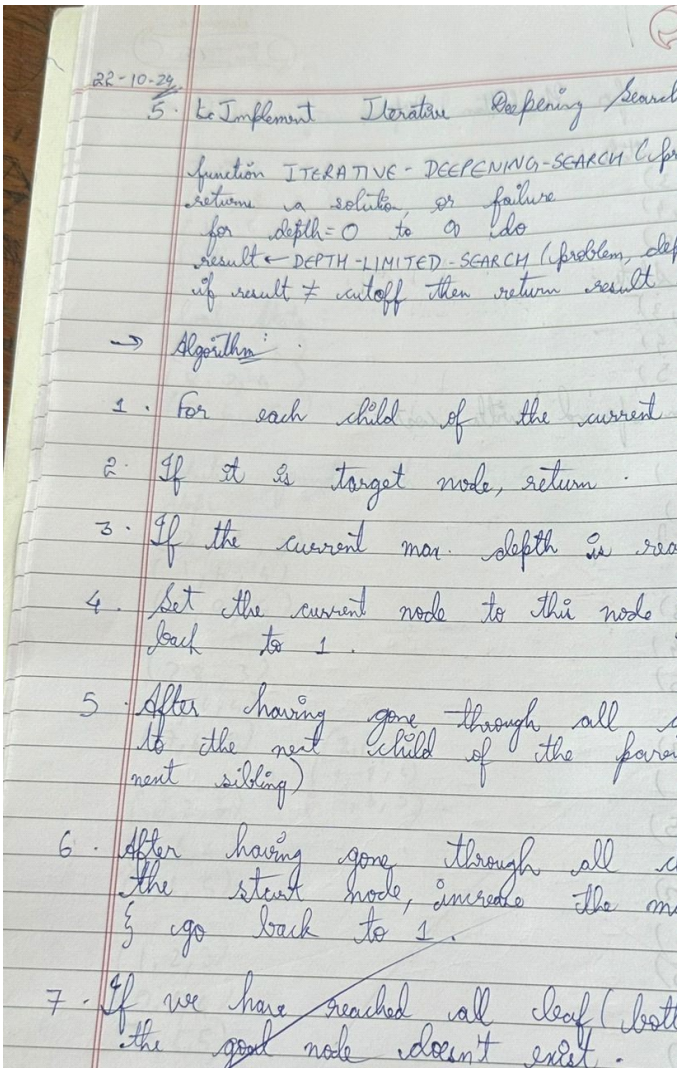
```
Enter initial location (A or B): A
Enter status of Room A (0 for clean, 1 for dirty): 1
Enter status of Room B (0 for clean, 1 for dirty): 1
Location A is Dirty. Cleaning Location A.
Location A has been Cleaned. COST for SUCK: 1
Moving to Location B.
Location B is Dirty. Cleaning Location B.
Location B has been Cleaned. COST for SUCK: 2
Moving to Location A.
Goal state achieved: [0, 0] with total cost: 2
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:





Code:

DFS:

```
from copy import deepcopy

goal_state = [[0, 1, 2],
              [3, 4, 5],
              [6, 7, 8]]
moves = {
   'up': (-1, 0),
```

```
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}

def find_blank(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
    return None

def is_goal(state):
    return state == goal_state

def is_valid_move(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def apply_move(board, move):
    x, y = find_blank(board)
    dx, dy = moves[move]
    new_x, new_y = x + dx, y + dy
    if is_valid_move(new_x, new_y):
        new_board = deepcopy(board)
        new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
        return new_board
    return None

def dfs(start):
    stack = [(start, [], 0)]  # (state, path of moves, number of states explored)
    visited = set()
    explored_count = 0  # Count of unique states explored

    while stack:
        current_state, path, current_count = stack.pop()

        if is_goal(current_state):
            return path, explored_count  # Return path of moves and explored count

        # Create a unique representation of the state to store in visited
        state_tuple = tuple(tuple(row) for row in current_state)
        if state_tuple not in visited:
            visited.add(state_tuple)  # Add to visited states
            explored_count += 1
            for move in moves:
                new_state = apply_move(current_state, move)
                if new_state:  # Only consider valid new states
                    stack.append((new_state, path + [move], explored_count))  # Store the path and
current count
```

```
    return None, explored_count  # Return explored count if no solution is found

def print_board(board):
    for row in board:
        print(row)
    print()

def print_solution(solution, explored_count):
    if solution is not None:
        print(f"Goal achieved with the following moves:")
        current_board = initial_state
        for move in solution:
            print(f"Move: {move}")
            current_board = apply_move(current_board, move)
            print_board(current_board)
        print(f"Total number of  states explored: {explored_count}")
    else:
        print("No solution found")

initial_state = [[1, 2, 5],
                 [3, 4, 8],
                 [6, 7, 0]]
solution, explored_count = dfs(initial_state)
print_solution(solution, explored_count)
```

Iterative deepening Search:

```
from copy import deepcopy
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy
```

```
            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors

    def is_goal(self, goal_state):
        return self.board == goal_state

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
    return None

def iterative_deepening_search(start_state, goal_state):
    depth = 0
    while True:
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
        depth += 1

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    return start_state, goal_state

def main():
```

```
    start_board, goal_board = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state)


    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found.")

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

Output:

```
Goal achieved with the following moves:
Move: left
[1, 2, 5]
[3, 4, 8]
[6, 0, 7]

Move: left
[1, 2, 5]
[3, 4, 8]
[0, 6, 7]

Move: up
[1, 2, 5]
[0, 4, 8]
[3, 6, 7]

Move: right
[1, 2, 5]
[4, 0, 8]
[3, 6, 7]

Move: right
[1, 2, 5]
[4, 8, 0]
```

[3, 6, 7]

Move: down
[1, 2, 5]
[4, 8, 7]
[3, 6, 0]

Move: left
[1, 2, 5]
[4, 8, 7]
[3, 0, 6]

Move: left
[1, 2, 5]
[4, 8, 7]
[0, 3, 6]

Move: up
[1, 2, 5]
[0, 8, 7]
[4, 3, 6]

Move: right
[1, 2, 5]
[8, 0, 7]
[4, 3, 6]

Move: right
[1, 2, 5]
[8, 7, 0]
[4, 3, 6]
Move: up
[1, 2, 5]
[0, 6, 3]
[7, 8, 4]

Move: right
[1, 2, 5]
[6, 0, 3]
[7, 8, 4]

Move: right
[1, 2, 5]
[6, 3, 0]
[7, 8, 4]

Move: down
[1, 2, 5]
[6, 3, 4]
[7, 8, 0]

Move: left
[1, 2, 5]
[6, 3, 4]
[7, 0, 8]

Move: left
[1, 2, 5]
[6, 3, 4]
[0, 7, 8]

Move: up
[1, 2, 5]
[0, 3, 4]
[6, 7, 8]

Move: right
[1, 2, 5]
[3, 0, 4]
[6, 7, 8]

Move: right
[1, 2, 5]
[3, 4, 0]
[6, 7, 8]

Move: up
[1, 2, 0]
[3, 4, 5]
[6, 7, 8]

Move: left
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

Move: left
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]

Total number of unique states explored: 32

Code:
Iterative Deepening Search

```
from copy import deepcopy
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors

    def is_goal(self, goal_state):
        return self.board == goal_state

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
                return found
```

```
    return None

def iterative_deepening_search(start_state, goal_state):
    depth = 0
    while True:
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
        depth += 1

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    return start_state, goal_state

def main():
    start_board, goal_board = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state)


    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found.")

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

Output:

```
Enter the start state (use 0 for the blank):
1 2 3
4 0 5
6 7 8
Enter the goal state (use 0 for the blank):
1 2 0
3 4 5
6 7 8

Searching at depth level: 0

Searching at depth level: 1

Searching at depth level: 2

Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5

Searching at depth level: 6

Searching at depth level: 7

Searching at depth level: 8

Searching at depth level: 9

Searching at depth level: 10

Searching at depth level: 11

Searching at depth level: 12

Goal reached!
1 2 3
4 0 5
6 7 8

1 2 3
0 4 5
6 7 8

0 2 3
1 4 5
6 7 8
```

2 0 3
1 4 5
6 7 8

2 3 0
1 4 5
6 7 8

2 3 5
1 4 0
6 7 8

2 3 5
1 0 4
6 7 8

2 0 5
1 3 4
6 7 8

0 2 5
1 3 4
6 7 8

1 2 5
0 3 4
6 7 8

1 2 5
3 0 4
6 7 8

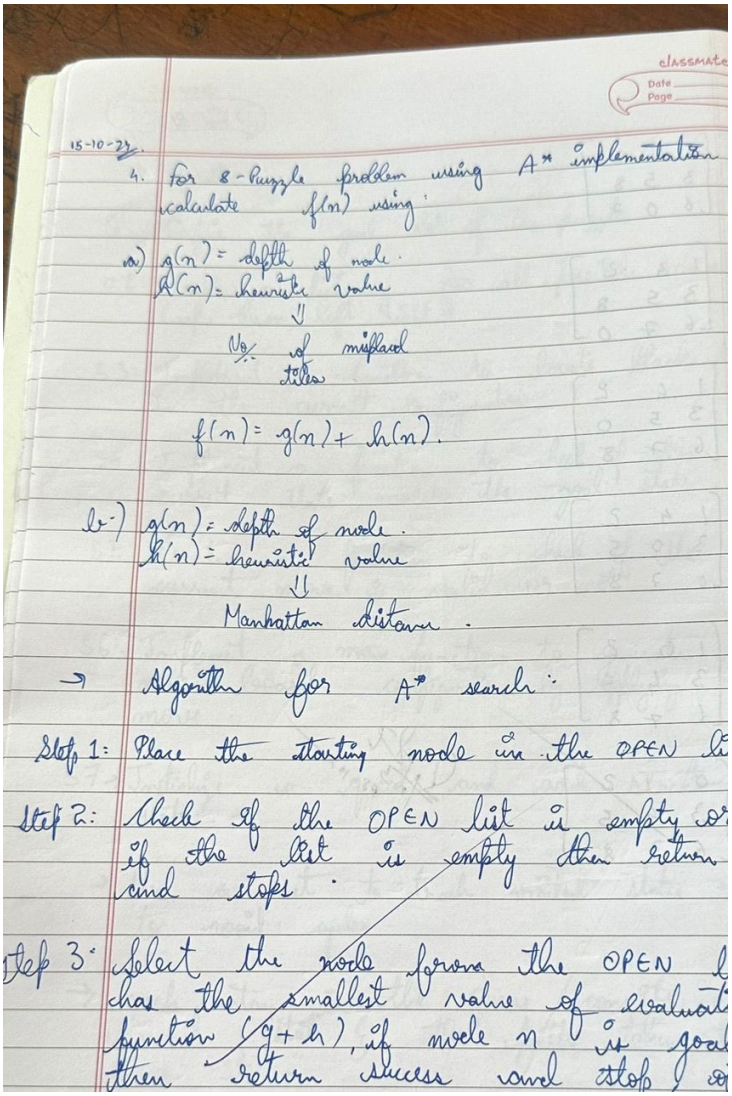
1 2 5
3 4 0
6 7 8

1 2 0
3 4 5
6 7 8

Program 3

Implement A* Algorithm

Algorithm



15-10-22

4. for 8-Puzzle problem using A* implementation calculate f(n) using:

a) g(n) = depth of node.
h(n) = heuristic value
⇓
No. of misplaced tiles

f(n) = g(n) + h(n).

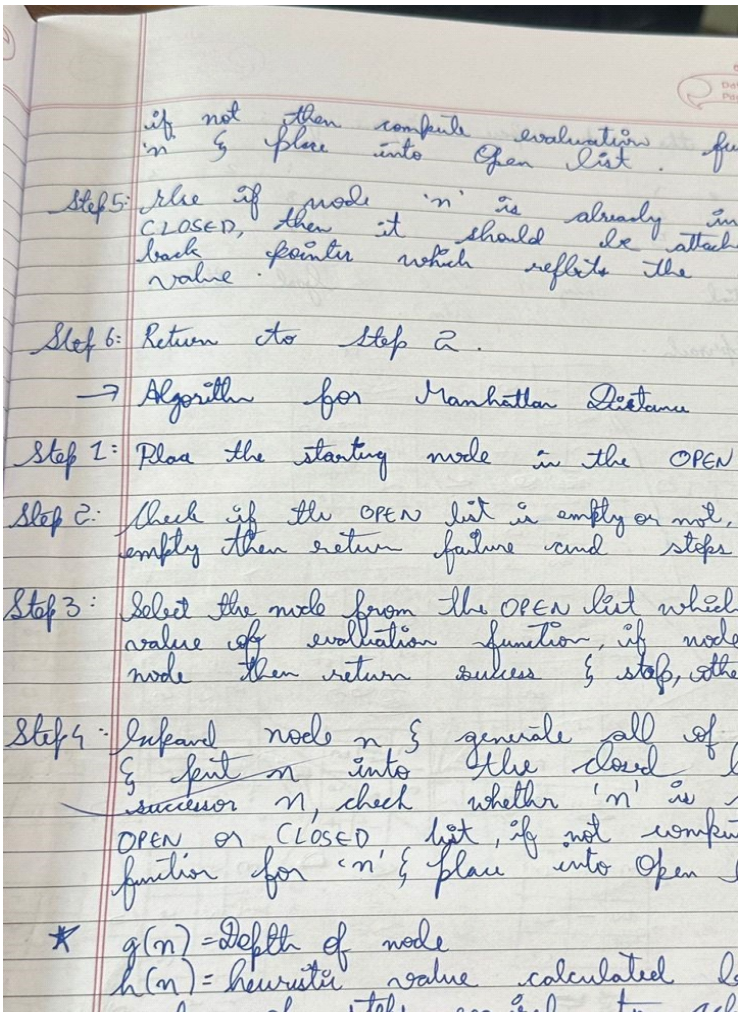
b) g(n) = depth of node.
h(n) = heuristic value
⇓
Manhattan distance.

→ Algorithm for A* search:

Step 1: Place the starting node in the OPEN list

Step 2: Check if the OPEN list is empty or if the list is empty then return and stops.

Step 3: Select the node from the OPEN list has the smallest value of evaluation function (g+h), if node n is goal then return success and stop, o



Code:

```
import heapq
GOAL_STATE = ((1, 2, 3),
              (8, 0, 4),
              (7, 6, 5))
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:

                goal_x, goal_y = divmod(value - 1, 3)
                distance += abs(goal_x - i) + abs(goal_y - j)
    return distance
def find_blank(state):
    for i in range(3):
        for j in range(3):
```

```
            if state[i][j] == 0:
                return i, j
def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def a_star(start):
    open_list = []
    heapq.heappush(open_list, (manhattan_distance(start), 0, start))

    g_score = {start: 0}
    came_from = {}

    visited = set()

    while open_list:
        f, g, current = heapq.heappop(open_list)

        if current == GOAL_STATE:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)
        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue

            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + manhattan_distance(neighbor)
```

```
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None

def print_state(state):
    for row in state:
        print(row)
    print()

if __name__ == "__main__":
    start_state = ((2, 8, 3),
                   (1, 6, 4),
                   (7, 0, 5))

    print("Initial State:")
    print_state(start_state)

    print("Goal State:")
    print_state(GOAL_STATE)

    solution, cost = a_star(start_state)

    if solution:
        print(f"Solution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("No solution found.")Output:

Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Solution found with cost: 5
Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)
```

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

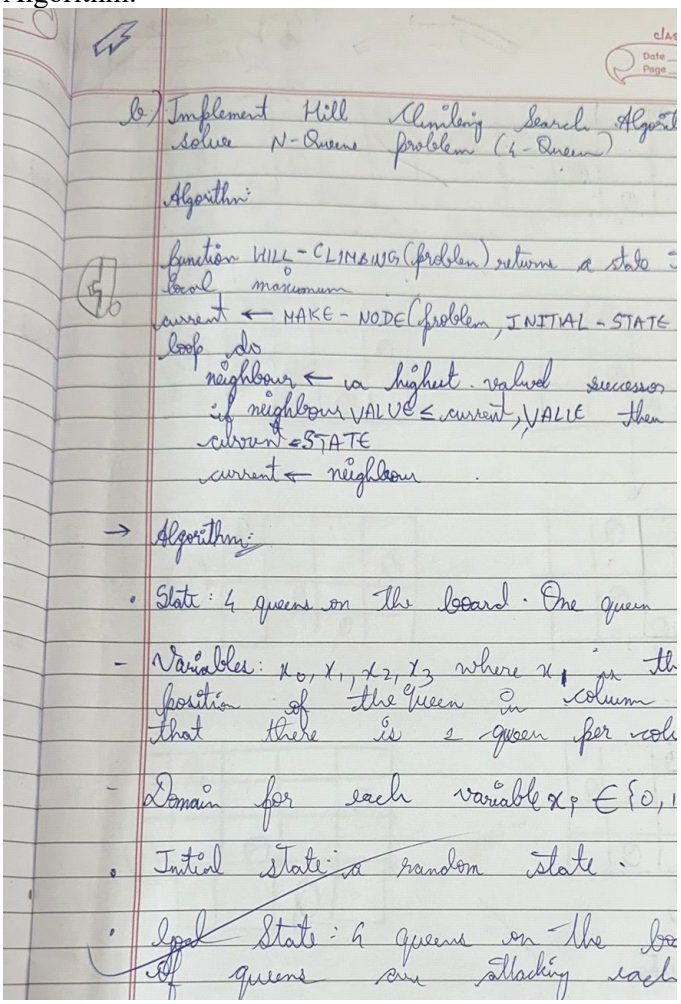
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



b) Implement Hill Climbing Search Algorithm to solve N-Queens problem (4-Queens)

Algorithm:

function HILL-CLIMBING(problem) returns a state that is a local maximum
current ← MAKE-NODE(problem, INITIAL-STATE)
loop do
neighbour ← a highest valued successor of current
if neighbour.VALUE ≤ current.VALUE then return current.STATE
current ← neighbour

→ Algorithm:

- State : 4 queens on the board. One queen in each column.
- Variables: x_0, x_1, x_2, x_3 where x_i is the position of the queen in column i such that there is 1 queen per column.
- Domain for each variable $x_i \in \{0, 1, 2, 3\}$
- Initial state: a random state.
- Goal State: 4 queens on the board such that no queens are attacking each other.

Code:

```
import random

def get_attacking_pairs(state):
    """Calculates the number of attacking pairs of queens."""
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                attacks += 1
            if abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks
```

```
def generate_successors(state):
    """Generates all possible successors by moving each queen to every other column in its
row."""
    n = len(state)
    successors = []
    for row in range(n):
        for col in range(n):
            if col != state[row]:
                new_state = state[:]
                new_state[row] = col
                successors.append(new_state)
    return successors

def hill_climbing(n):
    """Hill climbing algorithm for n-queens problem."""
    current = [random.randint(0, n - 1) for _ in range(n)]
    while True:
        current_attacks = get_attacking_pairs(current)
        successors = generate_successors(current)
        neighbor = min(successors, key=get_attacking_pairs)
        neighbor_attacks = get_attacking_pairs(neighbor)
        if neighbor_attacks >= current_attacks:
            return current, current_attacks
        current = neighbor

def print_board(state):
    """Prints the board with queens placed."""
    n = len(state)
    board = [["." for _ in range(n)] for _ in range(n)]
    for row in range(n):
        board[row][state[row]] = "Q"
    for row in board:
        print(" ".join(row))
    print("\n")

n = 4
solution, attacks = hill_climbing(n)
print("Final State (Solution):", solution)
print("Number of Attacking Pairs:", attacks)
print_board(solution)
```

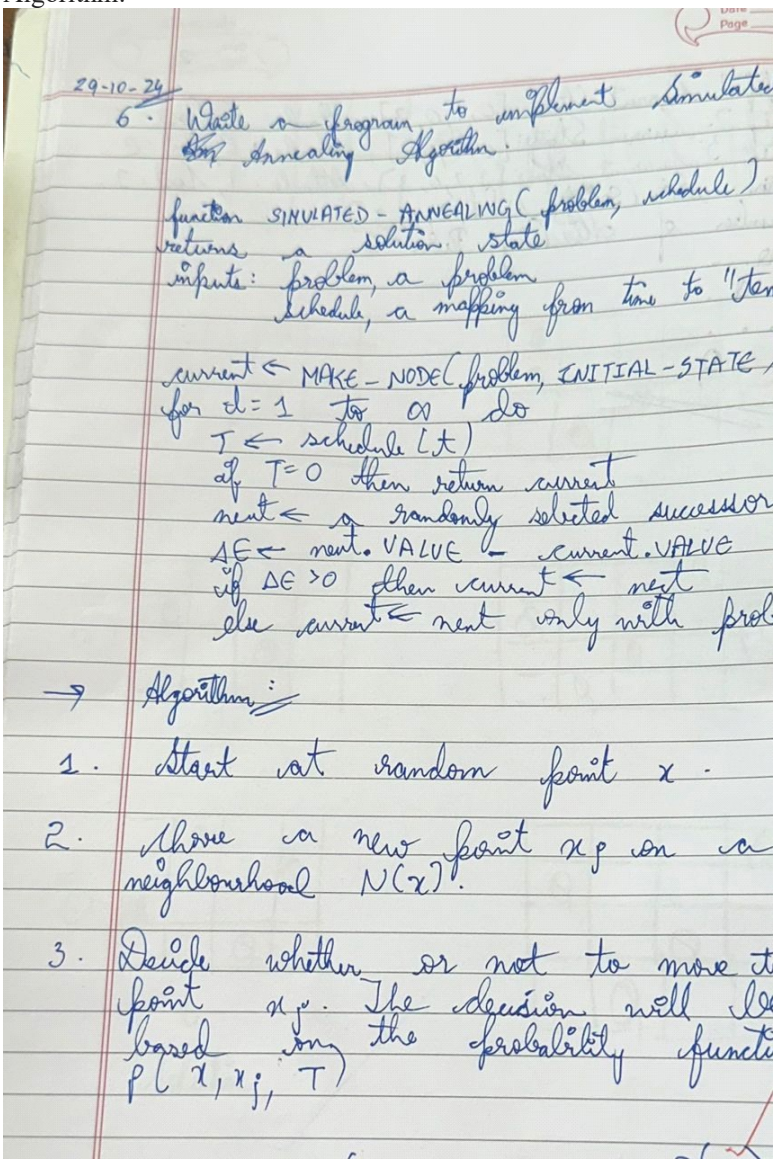
Output:

```
Final State (Solution): [3, 0, 2, 1]
Number of Attacking Pairs: 1
. . . Q
Q . . .
. . Q .
. Q . .
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



29-10-24

6. Write a program to implement Simulated Annealing Algorithm.

function SIMULATED-ANNEALING(problem, schedule)
returns a solution state
inputs: problem, a problem
schedule, a mapping from time to "tem

current ← MAKE-NODE(problem, INITIAL-STATE
for t = 1 to ∞ do
T ← schedule(t)
if T = 0 then return current
next ← a randomly selected successor
ΔE ← next.VALUE − current.VALUE
if ΔE > 0 then current ← next
else current ← next only with prob

→ Algorithm:

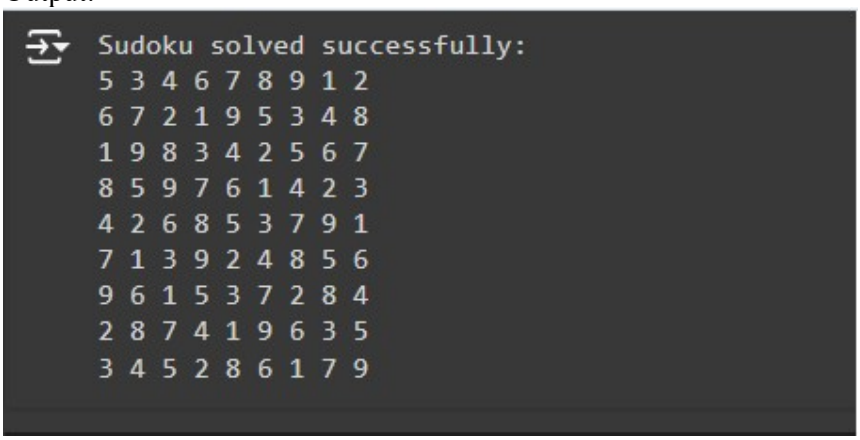
1. Start at random point x.
2. Move a new point x_j on a neighbourhood N(x).
3. Decide whether or not to move t point x_j. The decision will be based on the probability functi $p(x_i, x_j, T)$

Code:

```
def print_board(board):
    for row in board:
        print(" ".join(str(num) for num in row))

def is_valid(board, row, col, num):
    # Check if the number is not in the current row and column
    for i in range(9):
        if board[row][i] == num or board[i][col] == num:
            return False

    # Check the 3x3 grid
    start_row, start_col = 3 * (row // 3), 3 * (col // 3)
    for i in range(3):
        for j in range(3):
            if board[start_row + i][start_col + j] == num:
                return False

    return True

def solve_sudoku(board):
    empty = find_empty_location(board)
    if not empty:
        return True  # Puzzle solved

    row, col = empty

    for num in range(1, 10):
        if is_valid(board, row, col, num):
            board[row][col] = num

            if solve_sudoku(board):
                return True

            # Reset the cell (backtrack)
            board[row][col] = 0

    return False  # Trigger backtracking

def find_empty_location(board):
    for i in range(9):
        for j in range(9):
            if board[i][j] == 0:
                return (i, j)  # Return row, col
    return None

# Example Sudoku puzzle (0 represents empty cells)
sudoku_board = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
```

```
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

if solve_sudoku(sudoku_board):
    print("Sudoku solved successfully:")
    print_board(sudoku_board)
else:
    print("No solution exists.")
```

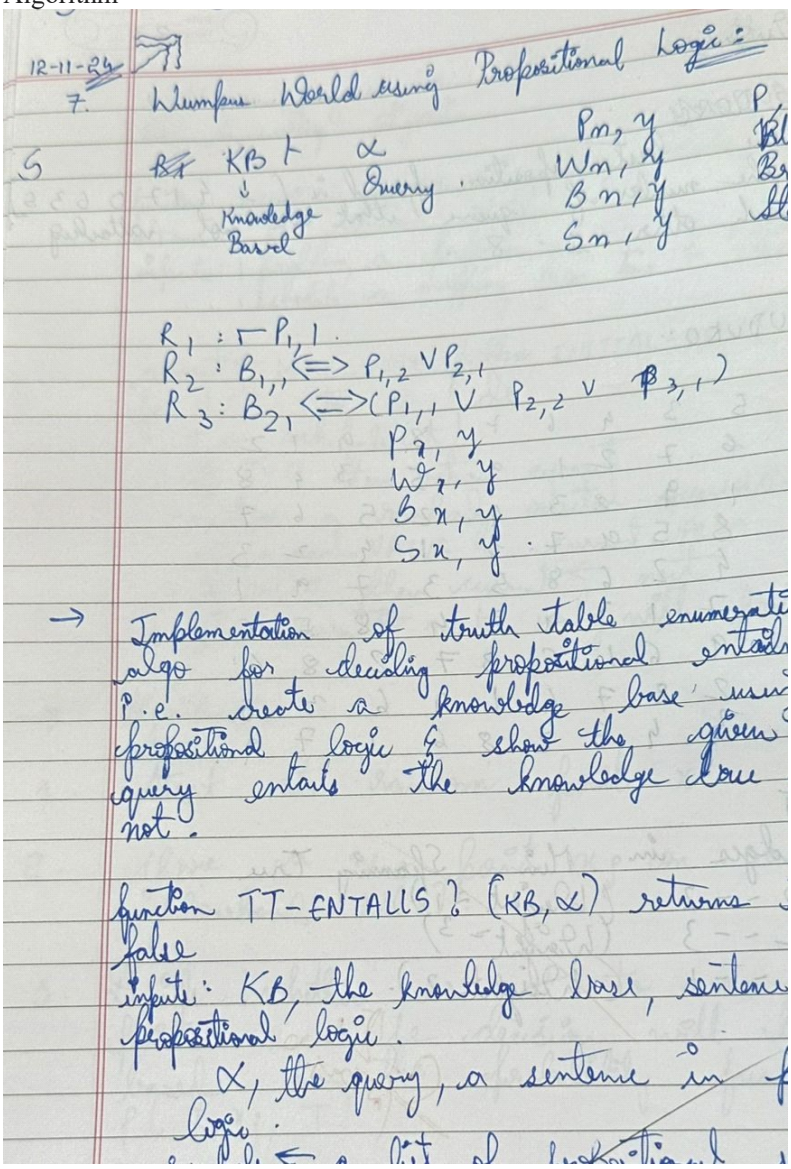
Output:



Program 6

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

Algorithm



function TT-CHECK-ALL(KB, α, symbols, model)
return true or false
if EMPTY?(symbols) then
if PL-true?(KB, model) then return
else return true // when KB is false, return true
else do
P ← FIRST(symbols)
rest ← REST(symbols)
return (TT-CHECK-ALL(KB, α, rest, model
and
TT-CHECK-ALL(KB, α, rest, model ∪

P	Q	¬P	P∧Q	P∨Q	P⇔Q	P⇒Q
true	true	false	true	true	true	true
true	false	false	false	true	false	false
false	true	true	false	true	false	true
false	false	true	false	false	true	true

α = A∨B KB = (A∨C)∧(B∨¬C)

A	B	C	A∨C	¬C	B∨¬C	KB
false	false	false	false	true	true	false
false	false	true	true	false	false	false
false	true	false	false	true	true	false
false	true	true	true	false	true	true
true	false	false	true	true	true	true

Combinations where both KB and α

A	B	C
0	1	1
1	0	0
1	1	0
1	1	1

Code:

```
import itertools

def evaluate_formula(formula, valuation):
    """
    Evaluate the propositional formula under the given truth assignment (valuation).
    The formula is a string of logical operators like 'AND', 'OR', 'NOT', and can contain
variables 'A', 'B', 'C'.
    """
    # Create a local environment (dictionary) for variable assignments
    env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

    # Replace logical operators with Python equivalents
    formula = formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')

    # Replace variables in the formula with their corresponding truth values
    for var in env:
        formula = formula.replace(var, str(env[var]))

    # Evaluate the formula and return the result (True or False)
    try:
        return eval(formula)
    except Exception as e:
        raise ValueError(f"Error in evaluating formula: {e}")

def truth_table(variables):
    """
    Generate all possible truth assignments for the given variables.
    """
    return list(itertools.product([False, True], repeat=len(variables)))

def entails(KB, alpha):
    """
    Decide if KB entails alpha using a truth-table enumeration algorithm.
    KB is a propositional formula (string), and alpha is another propositional formula (string).
    """
    # Generate all possible truth assignments for A, B, and C
    assignments = truth_table(['A', 'B', 'C'])

    print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}")  #
Header for the truth table
    print("-" * 70)  # Separator for readability

    for assignment in assignments:
        # Evaluate KB and alpha under the current assignment
        KB_value = evaluate_formula(KB, assignment)
        alpha_value = evaluate_formula(alpha, assignment)

        # Print the current truth assignment and the results for KB and alpha
```

```
print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB_v
alue):<15}{str(alpha_value):<15}{'Yes' if KB_value and alpha_value else 'No'}")

        # If KB is true and alpha is false, then KB does not entail alpha
        if KB_value and not alpha_value:
            return False

    # If no counterexample was found, then KB entails alpha
    return True

# Define the formulas for KB and alpha
alpha = 'A OR B'
KB = '(A OR C) AND (B OR NOT C)'

# Check if KB entails alpha
result = entails(KB, alpha)

# Print the final result of entailment
print(f"\nDoes KB entail alpha? {result}")
```

Output:

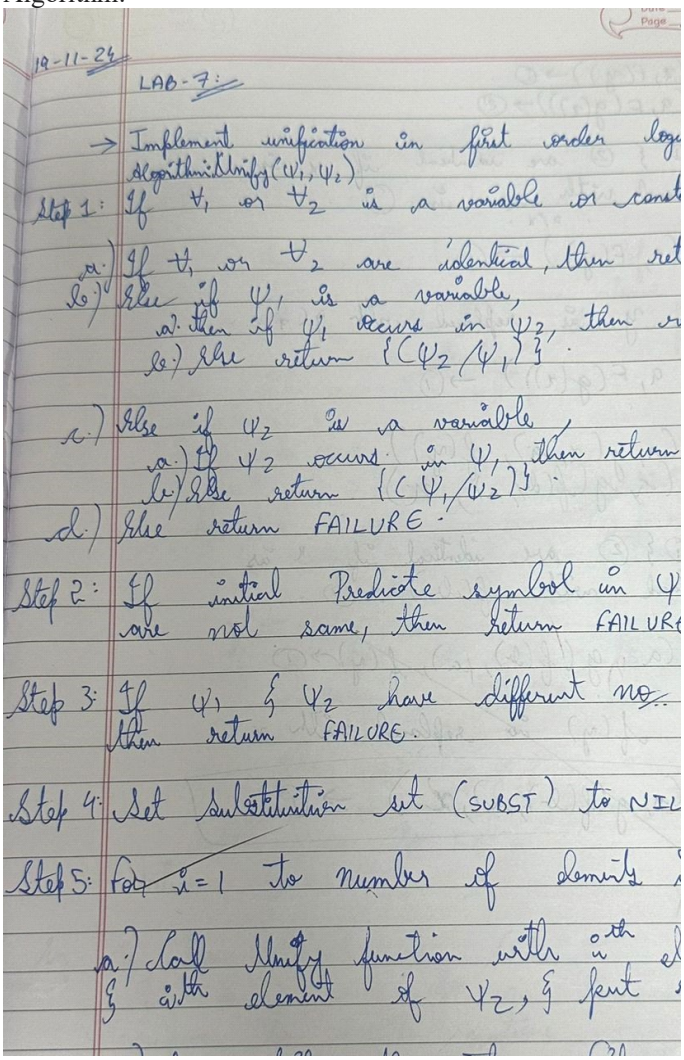
A	B	C	KB	alpha	KB entails alpha?
False	False	False	False	False	No
False	False	True	False	False	No
False	True	False	False	True	No
False	True	True	True	True	Yes
True	False	False	True	True	Yes
True	False	True	False	True	No
True	True	False	True	True	Yes
True	True	True	True	True	Yes

Does KB entail alpha? True

Program 7:

Implement unification in first order logic

Algorithm:


19-11-24

LAB-7:

→ Implement unification in first order log
Algorithm Unify(ψ_1, ψ_2)

Step 1: If ψ_1 or ψ_2 is a variable or const

a.) If ψ_1 or ψ_2 are identical, then ret
b.) Else if ψ_1 is a variable,
a.) then if ψ_1 occurs in ψ_2, then r
b.) Else return $\{(\psi_2 / \psi_1)\}$.

c.) Else if ψ_2 is a variable,
a.) If ψ_2 occurs in ψ_1 then return
b.) Else return $\{(\psi_1 / \psi_2)\}$.
d.) Else return FAILURE.

Step 2: If initial Predicate symbol in ψ
are not same, then return FAIL URE

Step 3: If ψ_1 & ψ_2 have different no.
then return FAILURE.

Step 4: Set Substitution set (SUBST) to NIL

Step 5: For i = 1 to number of elements

a.) Call Unify function with ith el
& ith element of ψ_2, & put

Code:

```
def unify(term1, term2):
    """
    Unifies two terms using the unification algorithm.

    Args:
        term1: The first term.
        term2: The second term.

    Returns:
        A substitution set if the terms are unifiable,
        otherwise None.
    """
```

```
    # Step 1: Handle variables and constants
    if isinstance(term1, str) or isinstance(term2, str):
        if term1 == term2:
            return {}  # NIL
        elif term1.isupper():  # term1 is a variable
            if term1 in term2:
                return None  # FAILURE
            return {term1: term2}
        elif term2.isupper():  # term2 is a variable
            if term2 in term1:
                return None  # FAILURE
            return {term2: term1}
        else:
            return None  # FAILURE

    # Step 2: Check predicate symbols
    if term1[0] != term2[0]:
        return None  # FAILURE

    # Step 3: Check number of arguments
    if len(term1[1:]) != len(term2[1:]):
        return None  # FAILURE

    # Step 4: Initialize substitution set
    substitution_set = {}

    # Step 5: Recursively unify arguments
    for i in range(len(term1[1:])):
        result = unify(term1[1:][i], term2[1:][i])
        if result is None:
            return None  # FAILURE
        else:
            substitution_set = apply_substitution(substitution_set, result)

    # Step 6: Return substitution set
    return substitution_set

def apply_substitution(substitution_set, new_substitution):
    """
    Applies a new substitution to the existing substitution set.

    Args:
        substitution_set: The current substitution set.
        new_substitution: The new substitution to be applied.

    Returns:
        The updated substitution set.
    """
```

```
    for var, value in new_substitution.items():
        # Apply the new substitution to the existing values
        for key, val in substitution_set.items():
            # Replace variables in the existing substitution set
            if key == var:
                substitution_set[key] = value
            elif isinstance(val, str) and val == var:
                substitution_set[key] = value

        # Add the new substitution to the set
        substitution_set[var] = value

    return substitution_set

def parse_term(term_str):
    """
    Parses a term string into a structured format.

    Args:
        term_str: The string representation of the term.

    Returns:
        A structured term (list).
    """
    term_str = term_str.strip()
    if '(' not in term_str:
        return term_str  # Return variable or constant

    # Split the term into the function and its arguments
    func_name, args_str = term_str.split('(', 1)
    args_str = args_str.rstrip(')')  # Remove the closing parenthesis

    # Split arguments by commas, accounting for nested terms
    args = []
    bracket_count = 0
    current_arg = []

    for char in args_str:
        if char == ',' and bracket_count == 0:
            args.append(''.join(current_arg).strip())
            current_arg = []
        else:
            if char == '(':
                bracket_count += 1
            elif char == ')':
                bracket_count -= 1
            current_arg.append(char)

    if current_arg:
        args.append(''.join(current_arg).strip())
```

```
    return [func_name] + [parse_term(arg) for arg in args]

def apply_substitution_to_term(term, substitution_set):
    """
    Applies a substitution set to a term.

    Args:
        term: The term to which the substitution is applied.
        substitution_set: The substitution set.

    Returns:
        The term after applying substitutions.
    """
    if isinstance(term, str):
        return substitution_set.get(term, term)  # Return the substituted value or the term itself

    # Apply substitution recursively to the term's components
    return [term[0]] + [apply_substitution_to_term(arg, substitution_set) for arg in term[1:]]

def main():
    # Take input from the user for the expressions
    term1_str = input("Enter the first term : ")
    term2_str = input("Enter the second term: ")

    # Parse the input terms
    term1 = parse_term(term1_str)
    term2 = parse_term(term2_str)

    # Perform unification
    substitution_set = unify(term1, term2)

    if substitution_set is not None:
        # Apply the substitution to the original terms
        unified_term1 = apply_substitution_to_term(term1, substitution_set)
        unified_term2 = apply_substitution_to_term(term2, substitution_set)

        # Print the final unified expressions
        print("Unified Term 1:", unified_term1)
        print("Unified Term 2:", unified_term2)
    else:
        print("The terms are not unifiable.")

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

Output:

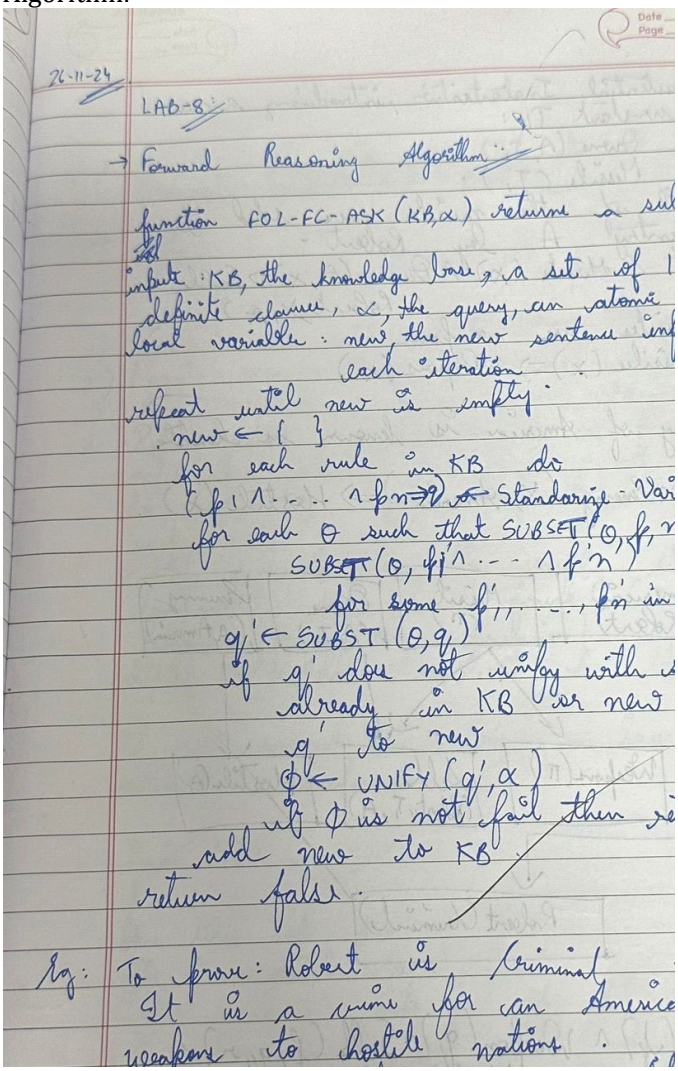
```
Enter the first term : f(X,f(Y))
Enter the second term: f(a,f(g(X))
Unified Term 1: ['f', 'a', ['f', ['g', 'X']]]
Unified Term 2: ['f', 'a', ['f', ['g', 'a']]]
```

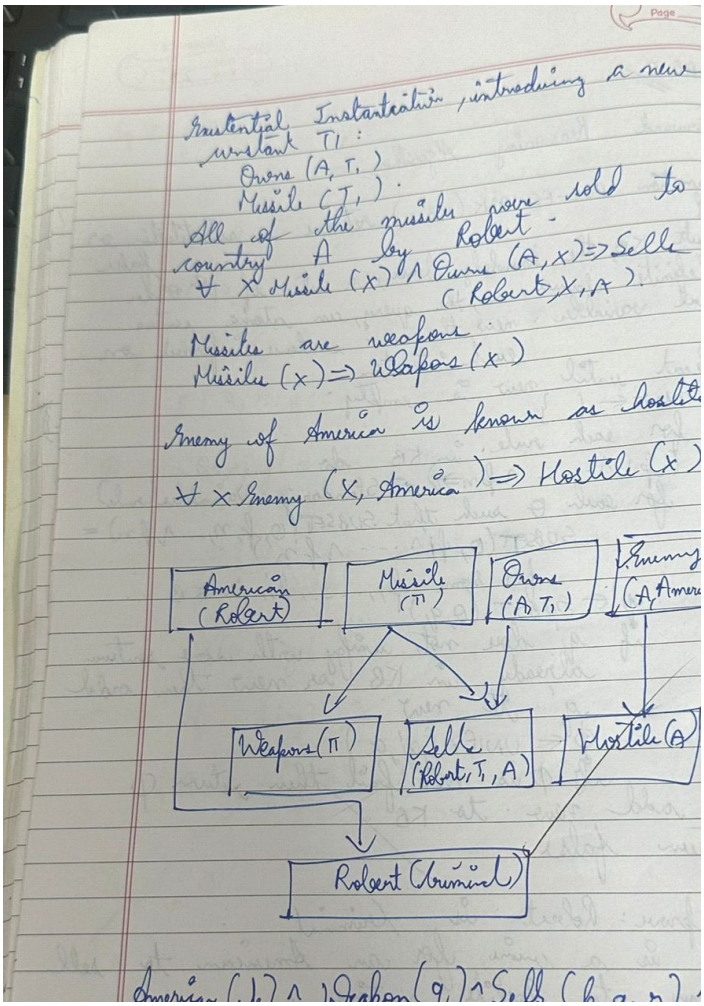
```
Enter the first term : f(f(a,g(Y))
Enter the second term: f(X,X)
The terms are not unifiable.
```

Program 8:

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:





Code:

 # Define the knowledge base as a list of rules and facts

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()   # Set of known facts
        self.rules = []      # List of rules

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
```

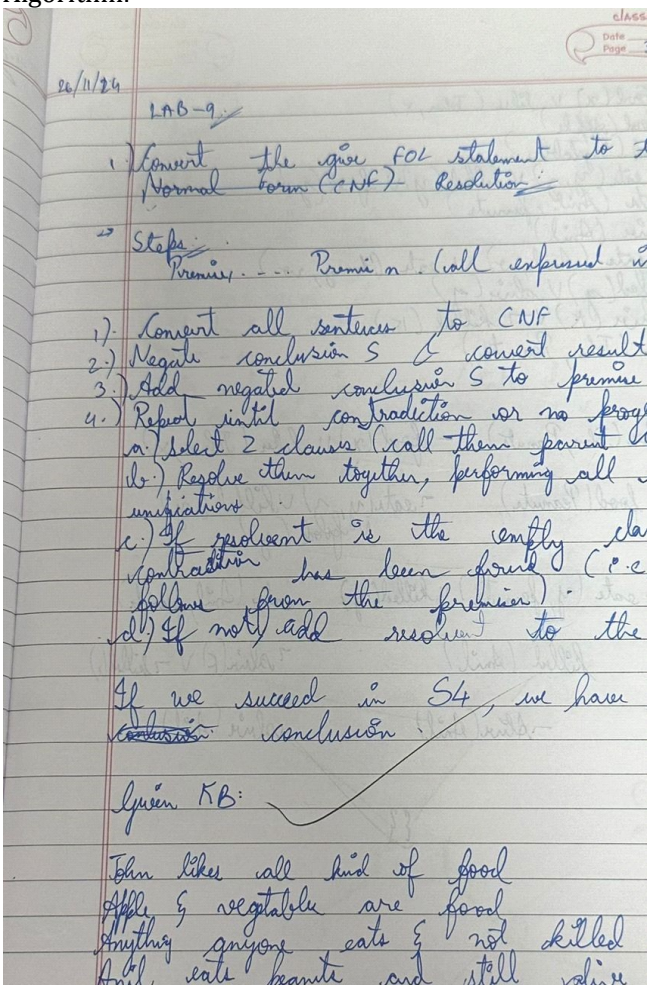
```
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")

# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)",
"Hostile(A)"], "Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")
```

Output:
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Conclusion: Robert is a criminal.

Program 9:

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

Algorithm:



26/11/24

LAB-9

1) Convert the given FOL statement to ... Normal form (CNF) Resolution

=> Steps:

Premise1 ---- Premise n (all expressed in ...

1). Convert all sentences to CNF
2.) Negate conclusion S & convert result ...
3.) Add negated conclusion S to premise ...
4.) Repeat until contradiction or no progress ...
a.) Select 2 clauses (call them parent ...
b.) Resolve them together, performing all ... unifications.
c.) If resolvent is the empty clause, a contradiction has been found (i.e. ... follows from the premises).
d.) If not, add resolvent to the ...

If we succeed in S4, we have ... conclusion.

Given KB:

John likes all kind of food
Apple & vegetables are food
Anything anyone eats & not killed
And eats peanuts and still alive

Code:

```
class CNFReasoner:
    def __init__(self, clauses):
        """
        Initializes the CNF Reasoner.

        Parameters:
            clauses (list of sets): List of clauses in CNF format.
        """
        self.clauses = [set(clause) for clause in clauses]

    def resolve(self, clause1, clause2):
        """
        Resolve two clauses to produce new clauses if possible.
```

```
        Parameters:
            clause1 (set): The first clause (set of literals).
            clause2 (set): The second clause (set of literals).

        Returns:
            list: A list of resolved clauses (sets of literals).
        """
        resolvents = []
        for literal in clause1:
            neg_literal = f"~{literal}" if not literal.startswith("~") else literal[1:]
            if neg_literal in clause2:
                # Create a new clause by removing complementary literals
                new_clause = (clause1 - {literal}) | (clause2 - {neg_literal})
                resolvents.append(new_clause)
        return resolvents

    def infer(self, goal):
        """
        Infer whether the goal is provable using resolution.

        Parameters:
            goal (set): The negation of the goal to be proved.

        Returns:
            bool: True if the goal is provable, False otherwise.
        """
        goal_clause = {f"~{literal}" for literal in goal}
        clauses = self.clauses + [goal_clause]  # Add the negated goal to clauses
        new = set()

        while True:
            pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1,
len(clauses))]
            for (clause1, clause2) in pairs:
                resolvents = self.resolve(clause1, clause2)
                for resolvent in resolvents:
                    if not resolvent:  # Found an empty clause, goal is proved
                        return True
                    new.add(frozenset(resolvent))

            # Check if no new information is being added
            if new.issubset(set(map(frozenset, clauses))):
                return False

            # Add new resolvents to the clauses
            for clause in new:
                if clause not in clauses:
                    clauses.append(set(clause))
```

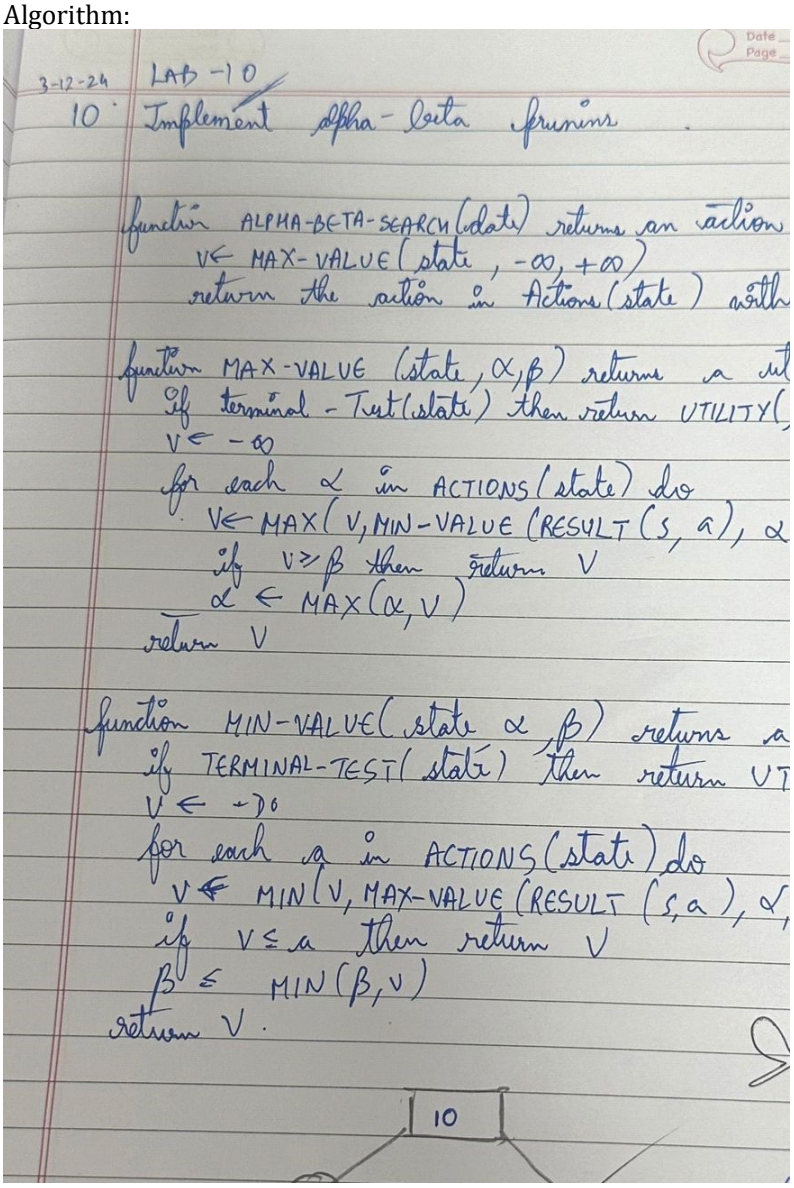
```
# Define the CNF clauses based on the FOL premises
cnf_clauses = [
    {"~Food(X)", "Likes(John, X)"},  # Rule 1: If Food(X), then Likes(John, X)
    {"~Eats(Anil, Peanuts)", "Food(Peanuts)"},  # Rule 2.1: If Anil eats peanuts, peanuts are food
    {"~Alive(Anil)", "Food(Peanuts)"},  # Rule 2.2: If Anil is alive, peanuts are food
    {"~Food(Peanuts)", "Likes(John, Peanuts)"},  # Rule 3: If peanuts are food, John likes peanuts
    {"Eats(Anil, Peanuts)"},  # Fact 1: Anil eats peanuts
    {"Alive(Anil)"},  # Fact 2: Anil is alive
]

# Define the goal: Prove that John likes peanuts
goal = {"Likes(John, Peanuts)"}

# Initialize the reasoner and infer
reasoner = CNFReasoner(cnf_clauses)
if reasoner.infer(goal):
    print("Proved: John likes peanuts.")
else:
    print("Could not prove: John likes peanuts.")
```

Output:
Proved: John likes peanuts.

Program 10:

Implement Alpha-Beta Pruning.

Algorithm:



Code:

```
import math

def minimax(depth, index, maximizing_player, values, alpha, beta):
    # Base case: when we've reached the leaf nodes
    if depth == 0:
        return values[index]

    if maximizing_player:
        max_eval = float('-inf')
        for i in range(2):  # 2 children per node
```

```
            eval = minimax(depth - 1, index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:  # Beta cutoff
                break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):  # 2 children per node
            eval = minimax(depth - 1, index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:  # Alpha cutoff
                break
        return min_eval


# Accept values from the user
leaf_values = list(map(int, input("Enter the leaf node values separated by spaces: ").split()))

# Check if the number of values is a power of 2
if math.log2(len(leaf_values)) % 1 != 0:
    print("Error: The number of leaf nodes must be a power of 2 (e.g., 2, 4, 8, 16).")
else:
    # Calculate depth of the tree
    tree_depth = int(math.log2(len(leaf_values)))

    # Run Minimax with Alpha-Beta Pruning
    optimal_value = minimax(depth=tree_depth, index=0, maximizing_player=True,
values=leaf_values, alpha=float('-inf'), beta=float('inf'))

    print("Optimal value calculated using Minimax:",optimal_value)
```

Output:

```
Enter the leaf node values separated by spaces: -1 8 -3 -1 2 1 -3 4
Optimal value calculated using Minimax: 2
```