VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sumanth S Shetty (1BM23CS348)

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
Aug 2025 to Dec 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 **Sumanth S Shetty (1BM23CS348),** 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.

Prof. Sonika Sharma Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

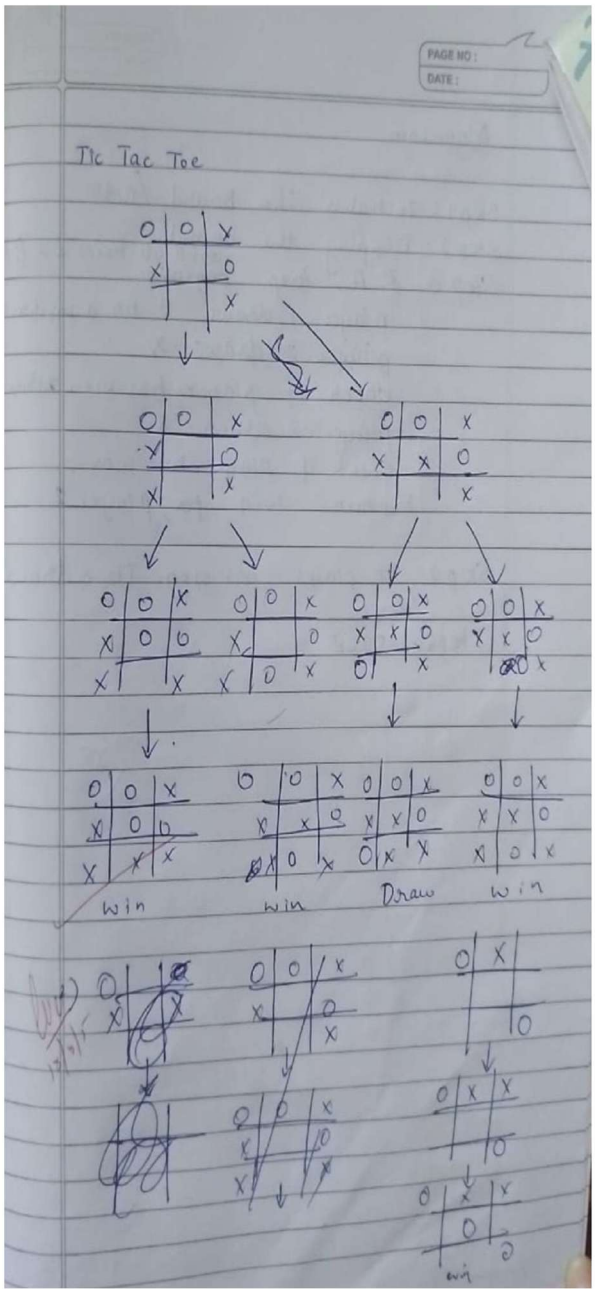
Index

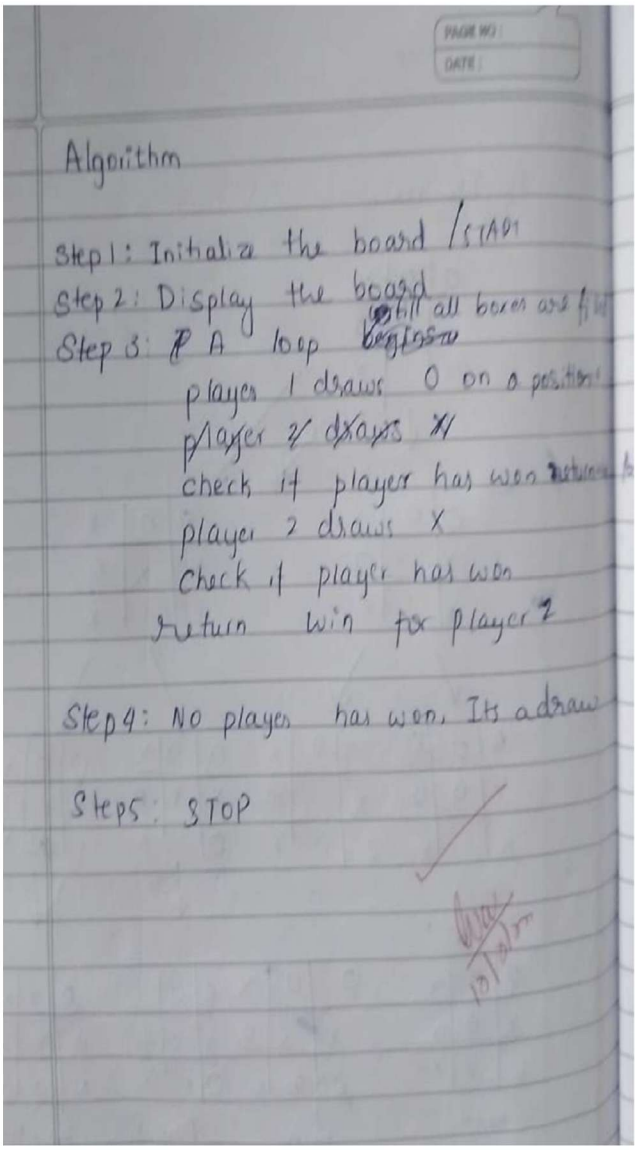
Sl. No.	Date	Experiment Title	Page No.
1	18-8-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	4-11
2	25-8-2025	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	12-17
3	8-9-2025	Implement A* search algorithm	18-22
4	15-9-2025	Implement Hill Climbing search algorithm to solve N-Queens problem	23-26
5	15-9-2025	Simulated Annealing to Solve 8-Queens problem	27-29
6	22-9-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	30-34
7	13-10-2025	Implement unification in first order logic	35-37
8	13-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	38-41
9	27-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	42-46
10	27-10-2025	Implement Alpha-Beta Pruning.	47-50

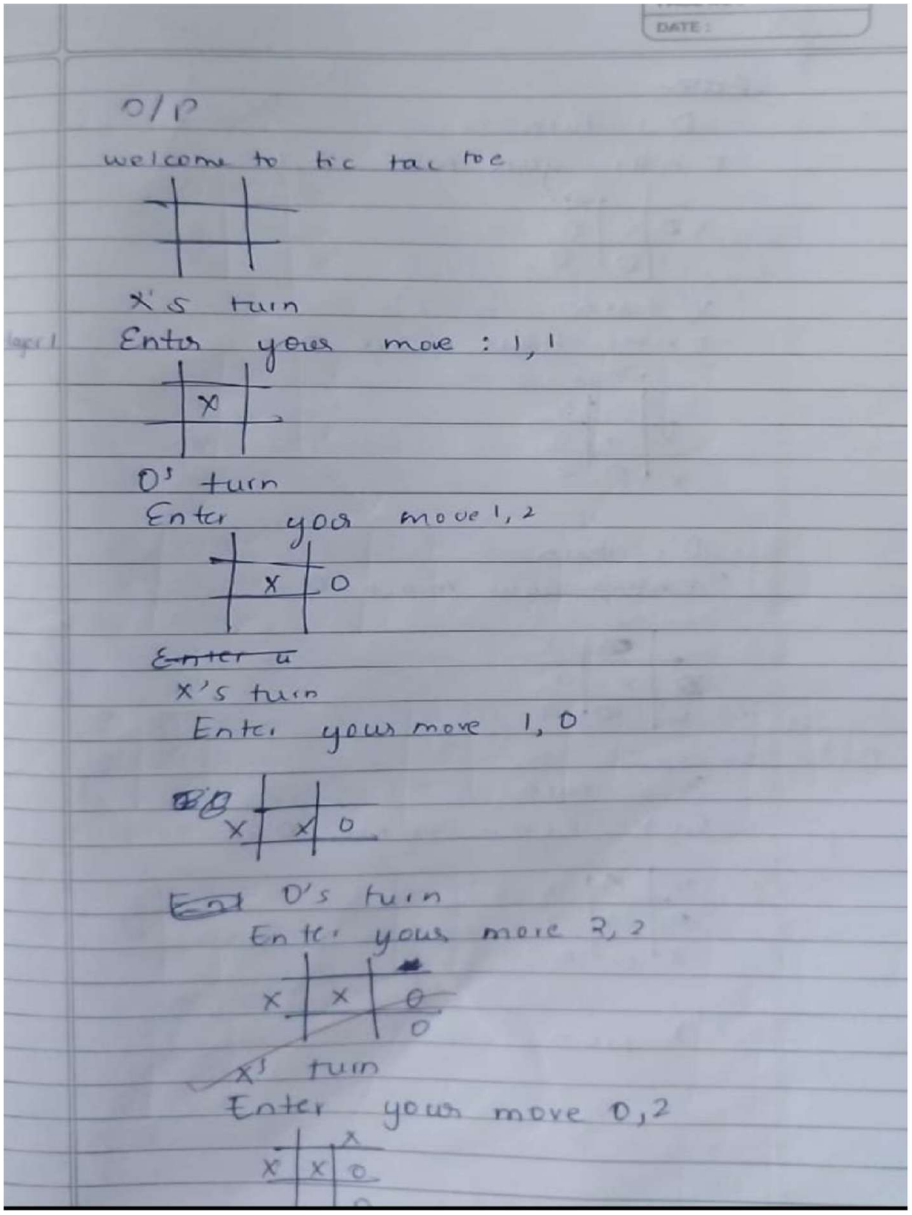
Github Link:
https://github.com/SumanthS117/1BM23CS348-AiLab

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

Algorithm:







Code:
TIC TAC TOE Game

```
print("Sumanth S Shetty\n 1BM23CS348")

def print_board(board):
    """Prints the Tic-Tac-Toe board."""
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def get_move(player):
    """Gets valid move input from the player."""
    while True:
        try:
```

```
            move = input(f"Player {player}, enter your move (row,col): ")
            row, col = map(int, move.split(','))
            if 0 <= row < 3 and 0 <= col < 3:
                return row, col
            else:
                print("Invalid input. Row and column must be between 0 and 2.")
        except ValueError:
            print("Invalid input. Please enter in the format row,col (e.g., 0,1).")

def is_valid_move(board, row, col):
    """Checks if the chosen cell is empty."""
    return board[row][col] == " "

def check_win(board, player):
    """Checks if the current player has won."""
    # Check rows
    for row in board:
        if all([cell == player for cell in row]):
            return True
    # Check columns
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] == player:
        return True
    if board[0][2] == board[1][1] == board[2][0] == player:
        return True
    return False

def play_game():
    """Runs the Tic-Tac-Toe game."""
    board = [[" " for _ in range(3)] for _ in range(3)]
    players = ["X", "O"]
    turn = 0

    print("Welcome to Tic-Tac-Toe!")
    print_board(board)

    while True:
        player = players[turn % 2]
        row, col = get_move(player)

        if is_valid_move(board, row, col):
            board[row][col] = player
            print_board(board)

            if check_win(board, player):
                print(f"Player {player} wins!")
                break
            elif all([cell != " " for row in board for cell in row]):
                print("It's a tie!")
                break

            turn += 1
        else:
            print("That cell is already occupied. Try again.")
```

```
# Start the game
while(True):
  play_again = input("Do you want to play again? (yes/no): ")
  if play_again.lower()== "yes":
    play_game()
  else: break
```

Sumanth S Shetty
1BM23CS348

```
Do you want to play again? (yes/no): yes
Welcome to Tic-Tac-Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row,col): 0,0
X |   |
---------
  |   |
---------
  |   |
---------
Player O, enter your move (row,col): 0,1
X | O |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row,col): 1,0
X | O |
---------
X |   |
---------
  |   |
---------
Player O, enter your move (row,col): 0,2
X | O | O
---------
X |   |
---------
  |   |
---------
Player X, enter your move (row,col): 2,0
X | O | O
---------
X |   |
---------
X |   |
---------
Player X wins!
Do you want to play again? (yes/no): yes
Welcome to Tic-Tac-Toe!
  |   |
---------
  |   |
```

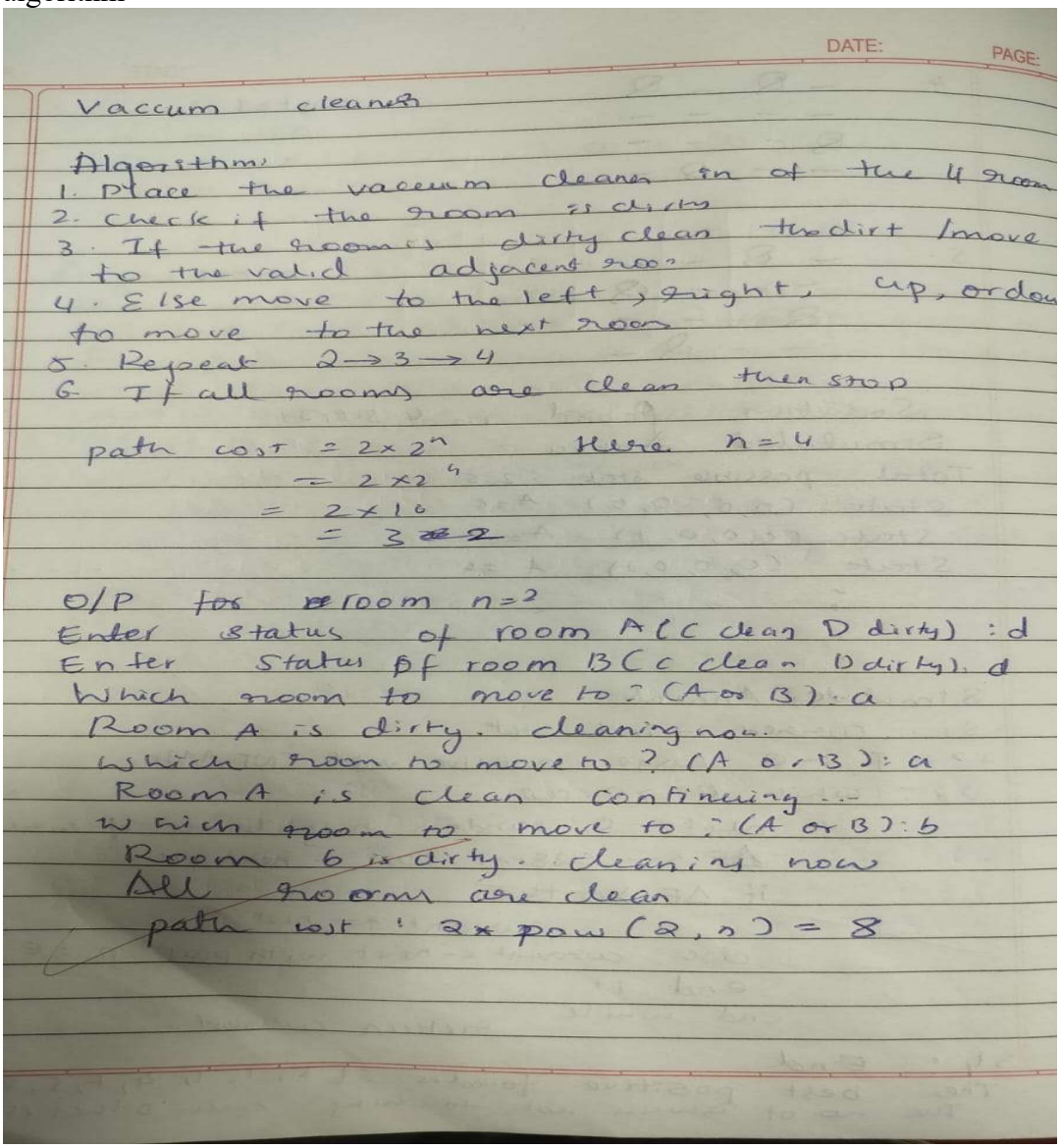
```
---------
  |  |
---------
Player X, enter your move (row,col): 0,2
  |  | X
---------
  |  |
---------
  |  |
---------
Player O, enter your move (row,col): 0,0
O |  | X
---------
  |  |
---------
  |  |
---------
Player X, enter your move (row,col): 1,2
O |  | X
---------
  |  | X
---------
  |  |
---------
Player O, enter your move (row,col): 1,0
O |  | X
---------
O |  | X
---------
  |  |
---------
Player X, enter your move (row,col): 1,1
O |  | X
---------
O | X | X
---------
  |  |
---------
Player O, enter your move (row,col): 2,0
O |  | X
---------
O | X | X
---------
O |  |
---------
Player O wins!
Do you want to play again? (yes/no): yes
Welcome to Tic-Tac-Toe!
  |  |
---------
  |  |
---------
  |  |
---------
Player X, enter your move (row,col): 0,0
X |  |
---------
  |  |
```

```
---------
 |  |
---------
Player O, enter your move (row,col): 1,0
X |  |
---------
O |  |
---------
 |  |
---------
Player X, enter your move (row,col): 1,2
X |  |
---------
O |  | X
---------
 |  |
---------
Player O, enter your move (row,col): 2,0
X |  |
---------
O |  | X
---------
O |  |
---------
Player X, enter your move (row,col): 2,1
X |  |
---------
O |  | X
---------
O | X |
---------
Player O, enter your move (row,col): 0,1
X | O |
---------
O |  | X
---------
O | X |
---------
Player X, enter your move (row,col): 0,2
X | O | X
---------
O |  | X
---------
O | X |
---------
Player O, enter your move (row,col): 2,2
X | O | X
---------
O |  | X
---------
O | X | O
---------
Player X, enter your move (row,col): 1,1
X | O | X
---------
O | X | X
---------
O | X | O
```

It's a tie!
Do you want to play again? (yes/no): no

Vacuum Cleaner Agent

algorithm



```
def vacuum_simulator():
    # Get initial status of rooms
    print("sumanths s shetty 1bm23cs348")
    rooms = {}
    for room in ['A', 'B']:
        while True:
            status = input(f"Enter status of room {room} (C for clean, D for dirty): ").strip().upper()
            if status in ['C', 'D']:
```

```
            rooms[room] = status
            break
        else:
            print("Invalid input. Please enter 'C' or 'D'.")

    total_cost = 0

    while True:
        # Check if all rooms are clean
        if all(status == 'C' for status in rooms.values()):
            print("All rooms are clean. Exiting.")
            break

        move = input("Which room to move to? (A or B): ").strip().upper()
        if move not in rooms:
            print("Invalid room. Please enter 'A' or 'B'.")
            continue

        total_cost += 1

        if rooms[move] == 'C':
            print(f"Room {move} is clean. Continuing...")
        else:
            print(f"Room {move} is dirty. Cleaning now.")
            rooms[move] = 'C'

    print(f"Total cost of moves: {total_cost}")

vacuum_simulator()
```

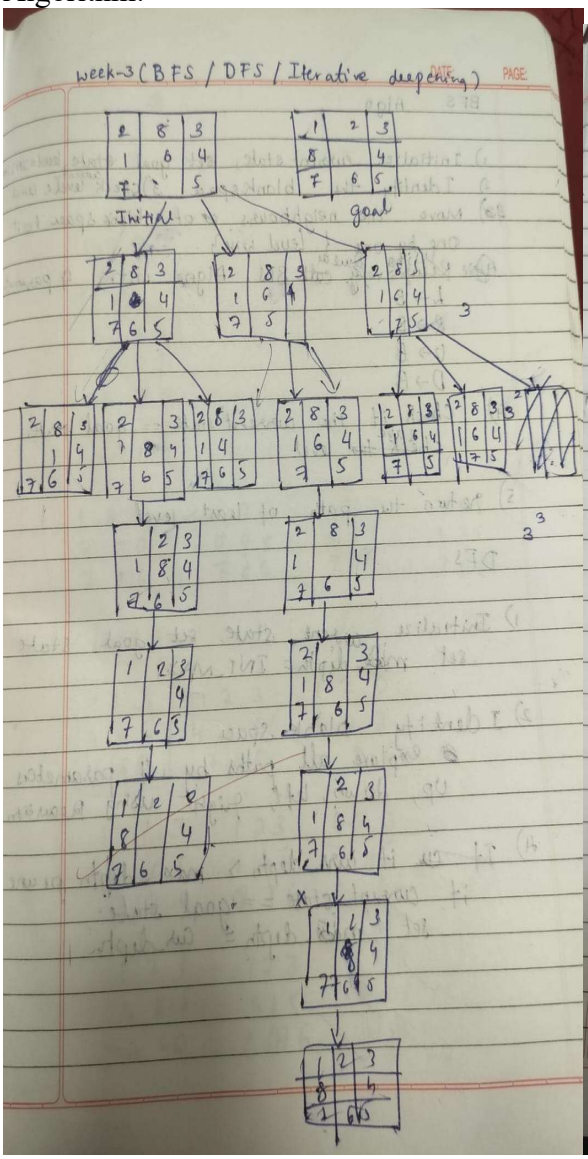
```
sumanths s shetty 1bm23cs348
Enter status of room A (C for clean, D for dirty): d
Enter status of room B (C for clean, D for dirty): d
Which room to move to? (A or B): a
Room A is dirty. Cleaning now.
Which room to move to? (A or B): a
Room A is clean. Continuing...
Which room to move to? (A or B): b
Room B is dirty. Cleaning now.
All rooms are clean. Exiting.
Total cost of moves: 3
```

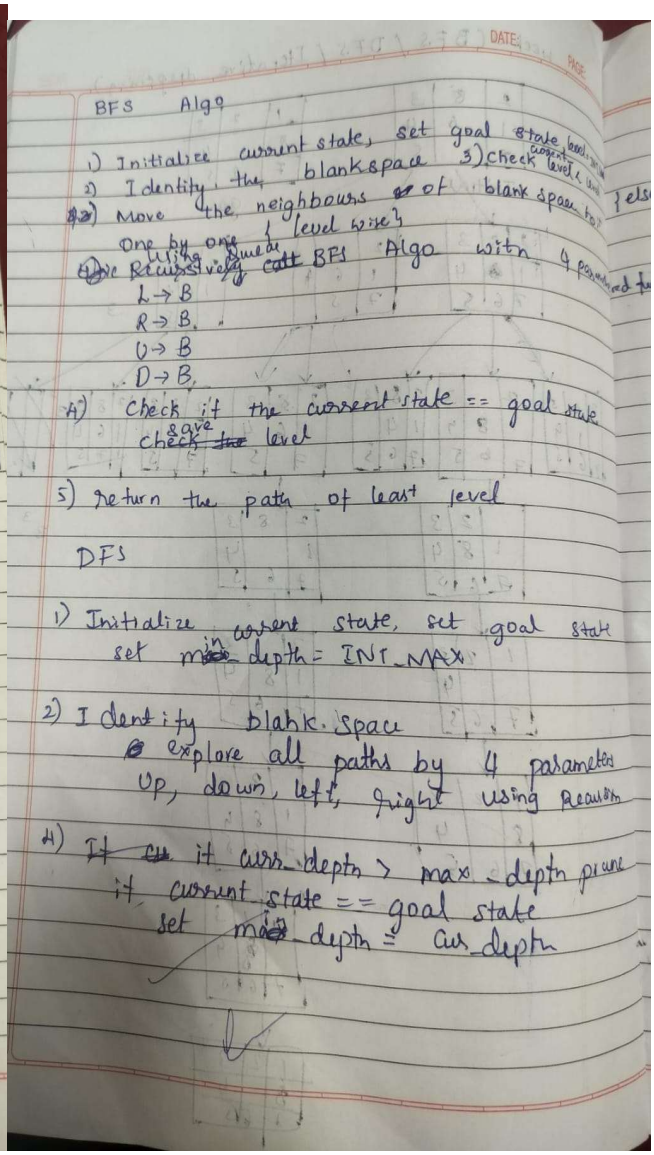
Program 2

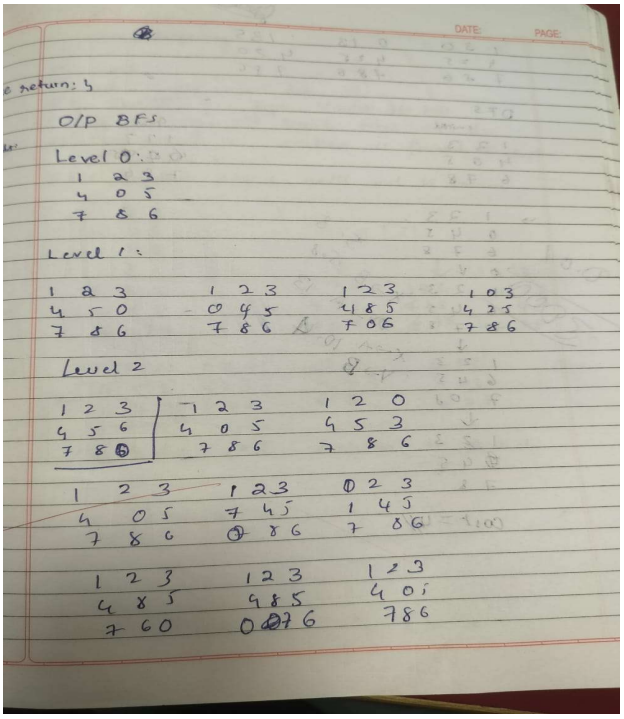
Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

8 Puzzle using DFS

Algorithm:







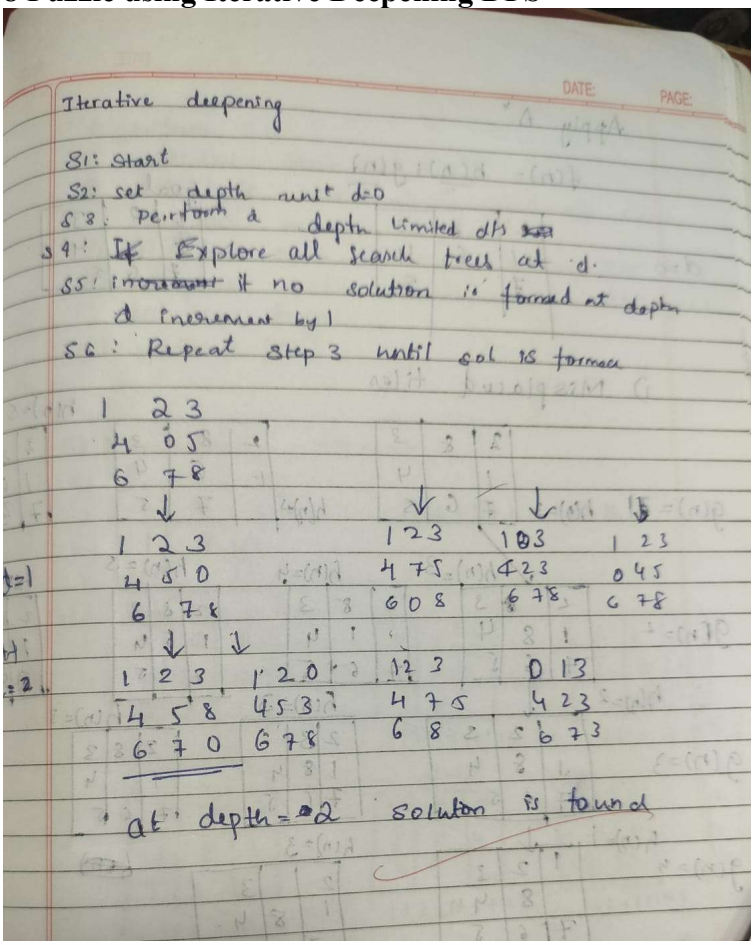
Code:

```
import collections
from typing import Tuple, List, Optional

State = Tuple[Tuple[int, ...], ...]

def find_empty_tile(state: State) -> Tuple[int, int]:
    for r, row in enumerate(state):
        for c, val in enumerate(row):
            if val == 0:
                return r, c
    raise ValueError

def get_next_states(state: State) -> List[State]:
    er, ec = find_empty_tile(state)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    out = []
    for dr, dc in moves:
        nr, nc = er + dr, ec + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            nb = [list(r) for r in state]
            nb[er][ec], nb[nr][nc] = nb[nr][nc], nb[er][ec]
            out.append(tuple(tuple(r) for r in nb))
    return out

def pretty_print(state: State) -> None:
    for row in state:
        print(" ".join(str(v) for v in row))
    print()

def solve_puzzle_dfs(initial_state: State, goal_state: State, max_depth: Optional[int] = None) -> Optional[List[State]]:
    stack = [(initial_state, [initial_state], 0)]
    visited = {initial_state}
    while stack:
```

```
        cur, path, depth = stack.pop()
        if cur == goal_state:
            return path
        if max_depth is not None and depth >= max_depth:
            continue
        for nxt in get_next_states(cur):
            if nxt not in visited:
                visited.add(nxt)
                stack.append((nxt, path + [nxt], depth + 1))
    return None

initial_state_3x3: State = (
    (2, 8, 3),
    (1, 6, 4),
    (7, 0, 5),
)

goal_state_3x3: State = (
    (2, 8, 3),
    (0, 1, 4),
    (7, 6, 5),
)

print("Attempting to solve the 3×3 puzzle with DFS...\n")
solution_path = solve_puzzle_dfs(initial_state_3x3, goal_state_3x3)

if solution_path:
    print(f"Solution found in {len(solution_path)-1} moves:\n")
    for i, st in enumerate(solution_path):
        print(f"Step {i}:")
        pretty_print(st)
else:
    print("No solution found.")

print("By \n  Sumanth S Shetty \n  1BM23CS348")
```

```
Attempting to solve the 3×3 puzzle with DFS...

Solution found in 2 moves:

Step 0:
2 8 3
1 6 4
7 0 5

Step 1:
2 8 3
1 0 4
7 6 5

Step 2:
2 8 3
0 1 4
7 6 5

By
  Sumanth S Shetty
  1BM23CS348
```

8 Puzzle using Iterative Deepening DFS



Code:

```
from collections import deque

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

MOVES = {
    'UP': (-1, 0),
    'DOWN': (1, 0),
    'LEFT': (0, -1),
    'RIGHT': (0, 1)
}

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
```

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

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    x1, y1 = pos1
    x2, y2 = pos2
    state_list[x1][y1], state_list[x2][y2] = state_list[x2][y2], state_list[x1][y1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    x, y = find_zero(state)
    for move in MOVES.values():
        new_x, new_y = x + move[0], y + move[1]
        if is_valid_pos(new_x, new_y):
            neighbors.append(swap_positions(state, (x, y), (new_x, new_y)))
    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 dls(state, depth_limit, came_from, visited):
    """Depth Limited Search"""
    if state == GOAL_STATE:
        return True
    if depth_limit <= 0:
        return False

    for neighbor in get_neighbors(state):
        if neighbor not in visited:
            visited.add(neighbor)
            came_from[neighbor] = state
            if dls(neighbor, depth_limit - 1, came_from, visited):
                return True
    return False

def iddfs(start_state, max_depth=50):
    """Iterative Deepening DFS"""
    for depth in range(max_depth):
        came_from = {}
        visited = {start_state}
        if dls(start_state, depth, came_from, visited):
            return reconstruct_path(came_from, GOAL_STATE)
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()

if __name__ == "__main__":
```

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

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

    print("Solving with Iterative Deepening DFS...")
    iddfs_path = iddfs(start_state)
    if iddfs_path:
        print(f"Solution found in {len(iddfs_path) - 1} moves!")
        for state in iddfs_path:
            print_state(state)
    else:
        print("No solution found with IDDFS.")

    print("By:\n   Sumanth S Sheety\n   1BM24CS348\n")
```

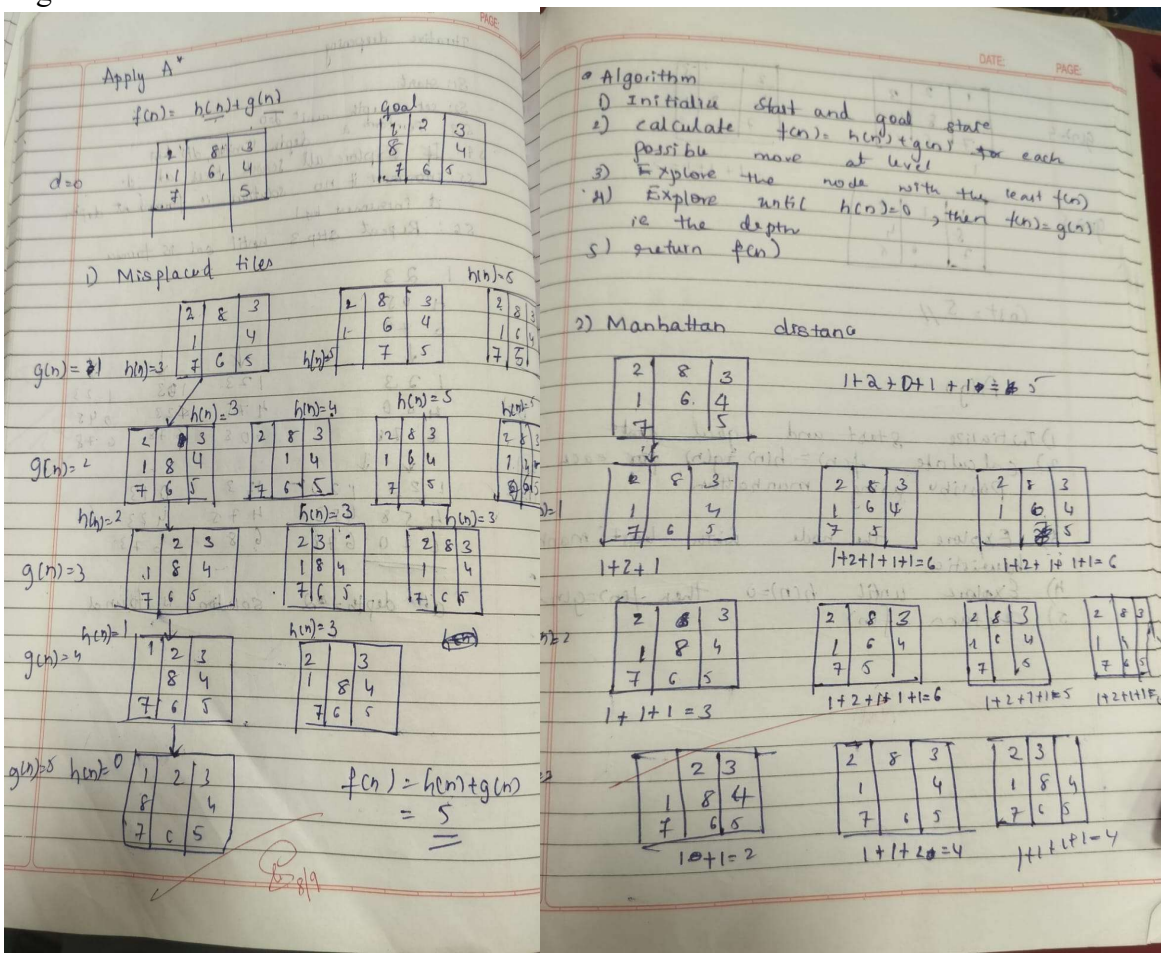
```
Initial State:
2 8 3
1 6 4
7 _ 5

Solving with Iterative Deepening DFS...
Solution found in 5 moves!
2 8 3
1 6 4
7 _ 5

2 8 3
1 _ 4
7 6 5

2 _ 3
1 8 4
7 6 5

_ 2 3
1 8 4
7 6 5

1 2 3
_ 8 4
7 6 5

1 2 3
8 _ 4
7 6 5

By:
  Sumanth S Sheety
  1BM24CS348
```

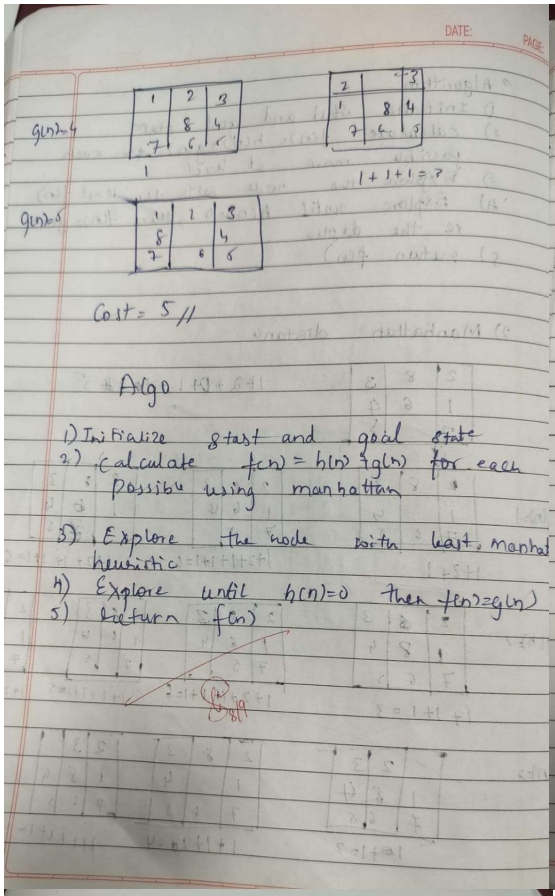
Program 3

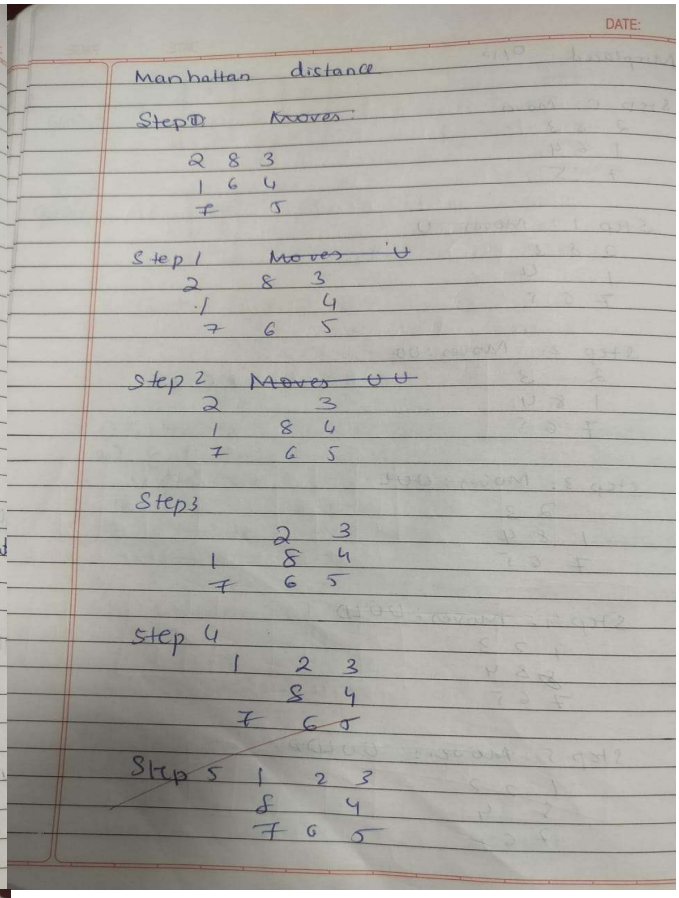
Implement A* search algorithm

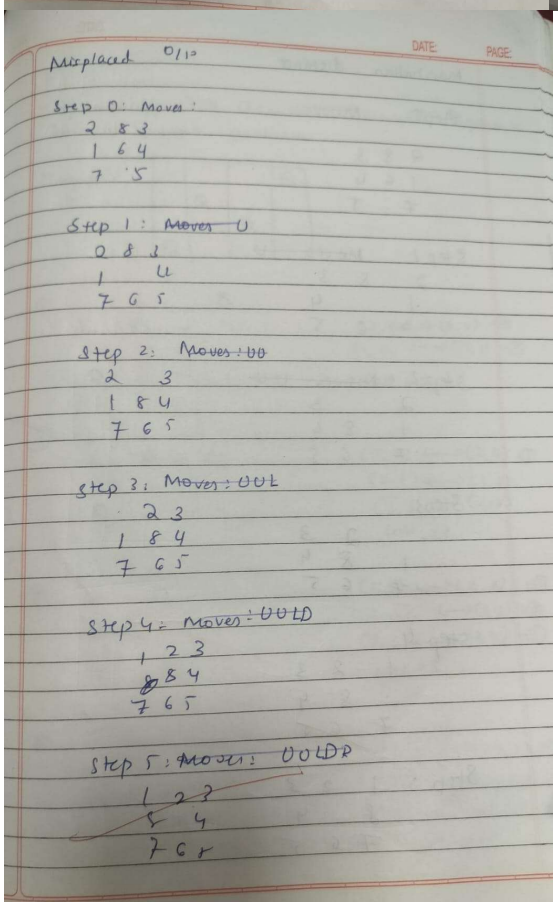
A* using misplaced tiles for 8 puzzle

Algorithm:




g(n)=4
1
1+1+1=?
g(n)=5
Cost = 5 //
Algo
1) Initialize start and goal state
2) Calculate f(n) = h(n) + g(n) for each possible using manhattan
3) Explore the node with least manhat heuristic
4) Explore until h(n)=0 then f(n)=g(n)
5) Return f(n)


Manhattan distance
Step 0 Moves:
2 8 3
1 6 4
7 5
Step 1 Moves U
2 8 3
1 4
7 6 5
Step 2 Moves UU
2 3
1 8 4
7 6 5
Step 3
2 3
1 8 4
7 6 5
Step 4
1 2 3
8 4
7 6 5
Step 5 1 2 3
8 4
7 6 5


Misplaced 0/1>
Step 0: Moves:
2 8 3
1 6 4
7 5
Step 1: Moves U
0 8 3
1 4
7 6 5
Step 2: Moves: UU
2 3
1 8 4
7 6 5
Step 3: Moves: UUL
2 3
1 8 4
7 6 5
Step 4: Moves: UULD
1 2 3
8 4
7 6 5
Step 5: Moves: UULDR
1 2 3
5 4
7 6

Code:

Mismatched

```
import heapq
import time

class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def heuristic(self):

        misplaced = 0
        for i, tile in enumerate(self.board):
            if tile != 0 and tile != self.goal[i]:
                misplaced += 1
        return misplaced

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y - 1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)
                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos], new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board), self.goal, self.path + move, self.cost + 1))
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map, current_state.board)

        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
```

```
            if neighbor.board not in explored and neighbor.board not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

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

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

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)
    else:
        print("No solution found.")


    print("By \n  Sumanth S Shetty\n  1BM23CS348")
```

Step-by-step solution:

Step 0: Moves:
2 8 3
1 6 4
7   5

Step 1: Moves: U
2 8 3
1 4
7 6 5

Step 2: Moves: UU
2 3
1 8 4
7 6 5

Step 3: Moves: UUL
 2 3
1 8 4
7 6 5

Step 4: Moves: UULD
1 2 3
 8 4
7 6 5

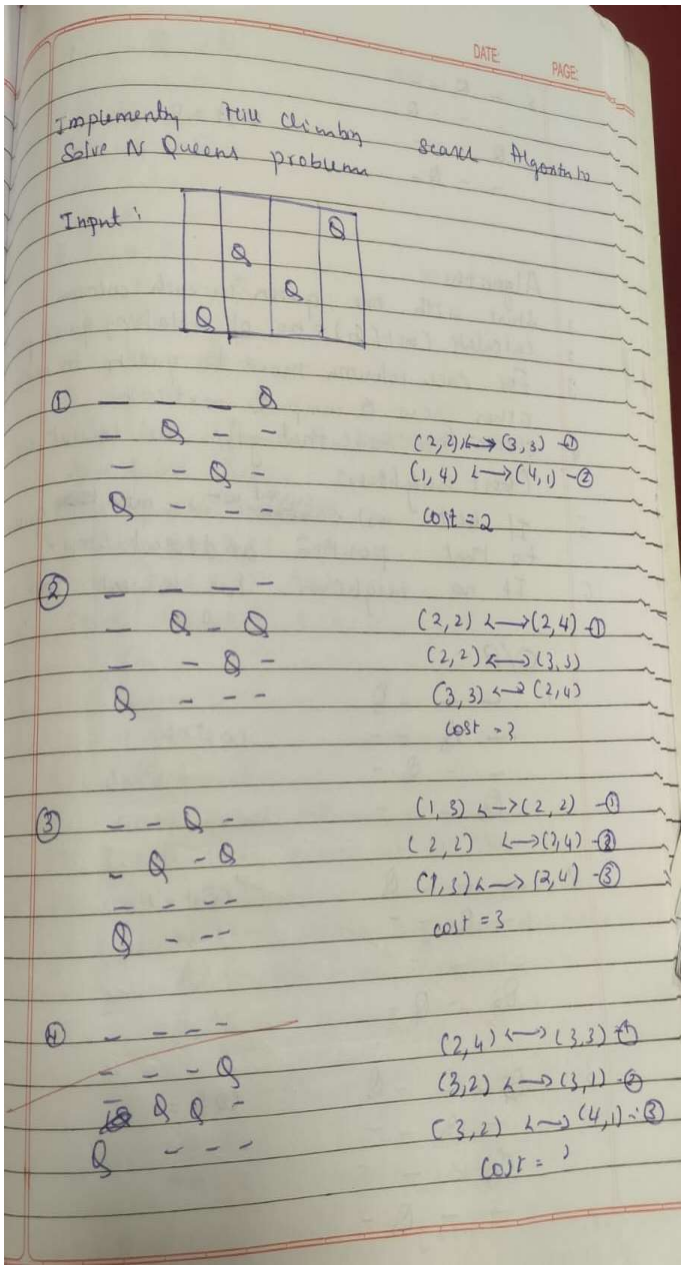
Step 5: Moves: UULDR
1 2 3
8 4
7 6 5

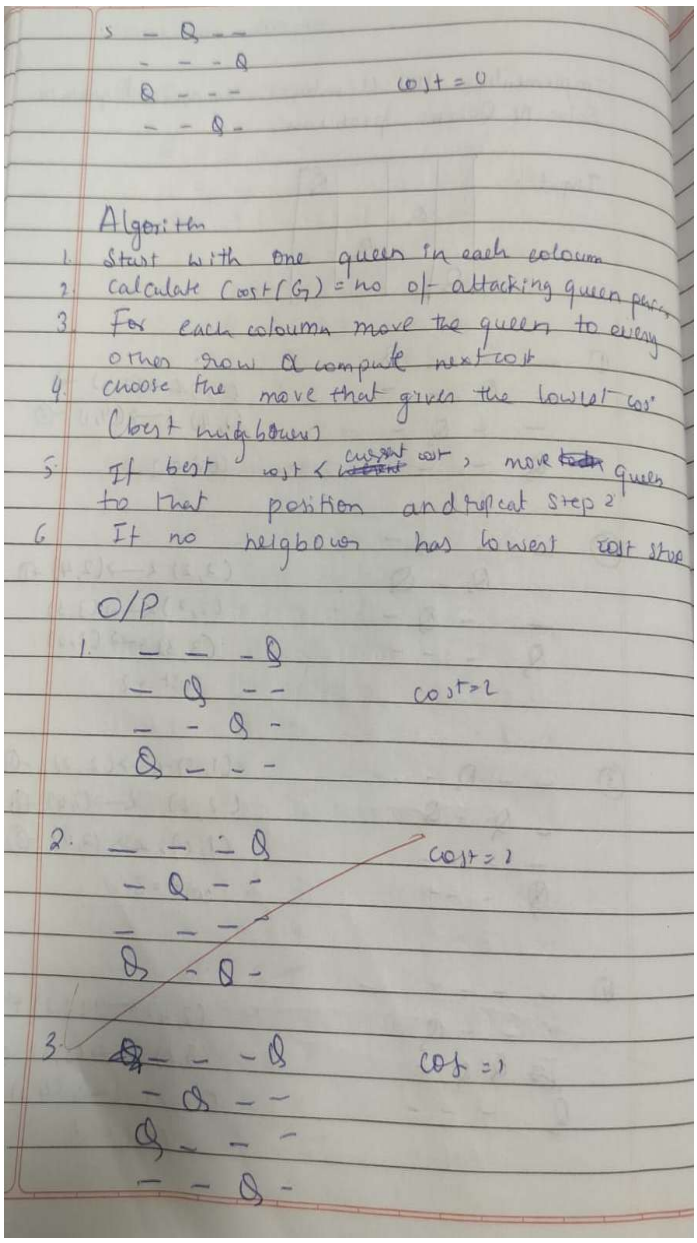
By
Sumanth S Shetty
1BM23CS348

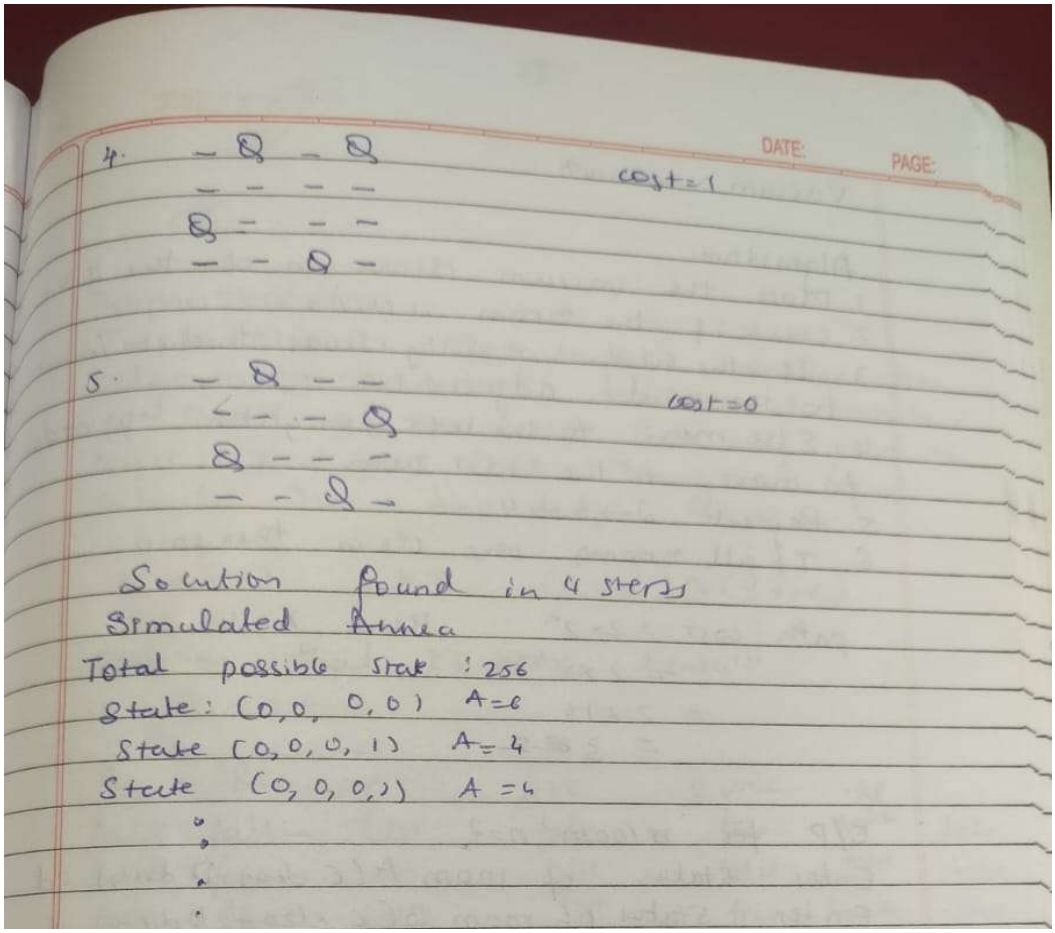
Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:







Code:

```
def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(line))
    print()

def hill_climbing_step_by_step(board):
    n = len(board)
    current_state = board[:]
    current_conflicts = calculate_conflicts(current_state)

    step = 0
    print(f"Initial board with conflicts = {current_conflicts}:")
```

```
    print_board(current_state)

    while current_conflicts > 0:
        step += 1
        print(f"Step {step}:")
        best_state = current_state[:]
        best_conflicts = current_conflicts

        for row in range(n):
            original_col = current_state[row]
            for col in range(n):
                if col != original_col:
                    current_state[row] = col
                    conflicts = calculate_conflicts(current_state)

                    if conflicts < best_conflicts:
                        best_conflicts = conflicts
                        best_state = current_state[:]

            current_state[row] = original_col

        if best_conflicts == current_conflicts:
            print("No better neighbor found, stuck at local optimum.")
            break

        current_state = best_state
        current_conflicts = best_conflicts

        print(f"Board with conflicts = {current_conflicts}:")
        print_board(current_state)

    if current_conflicts == 0:
        print("Solution found!")
    else:
        print("No solution found.")
    return current_state

initial_board = [3, 0, 1, 2]
solution = hill_climbing_step_by_step(initial_board)

print("By:\n   Sumanth S Shetty\n   1BM23CS348")


Initial board with conflicts = 4:
. . . Q
Q . . .
. Q . .
. . Q .

Step 1:
Board with conflicts = 2:
. . . Q
Q . . .
Q . . .
. . Q .

Step 2:
Board with conflicts = 1:
```

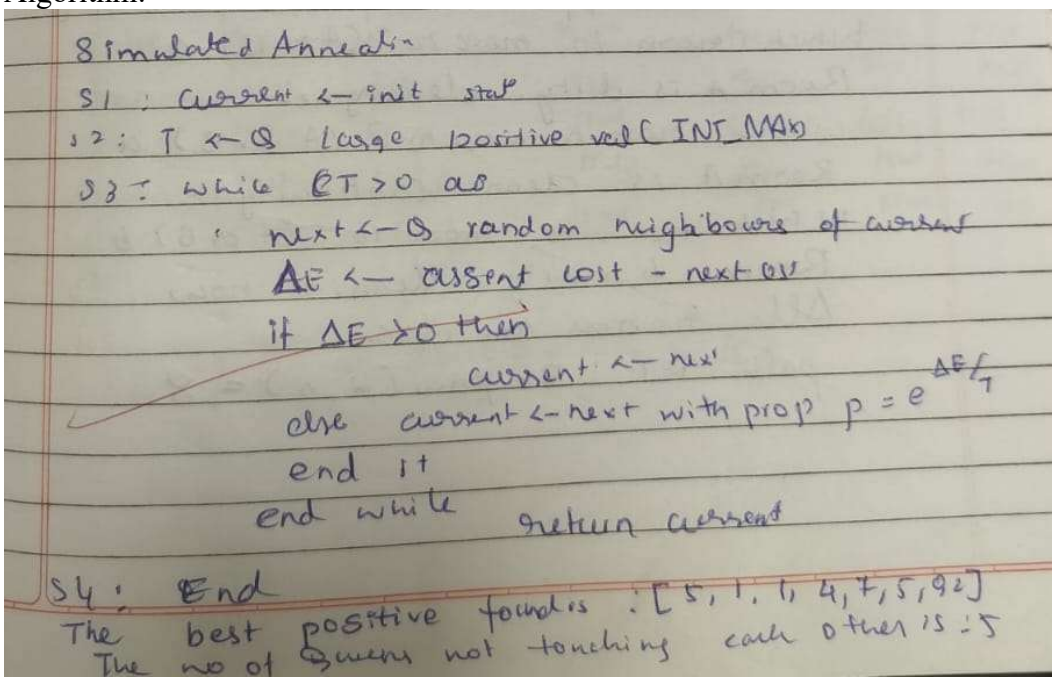
. . . Q
. Q . .
Q . . .
. . Q .

Step 3:
No better neighbor found, stuck at local optimum.
No solution found.
By:
Sumanth S Shetty
1BM23CS348

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

```
import random
import math

def get_user_input(n):
    board = []
    print(f"Enter the row positions (0 to {n-1}) of the queens for each column:")
    for col in range(n):
        while True:
            try:
                row = int(input(f"Column {col}: "))
                if 0 <= row < n:
                    board.append(row)
                    break
                else:
                    print(f"Invalid input. Please enter a number between 0 and {n-1}.")
            except ValueError:
                print("Invalid input. Please enter an integer.")
    return board

def print_board(board):
    n = len(board)
    for row in range(n):
        line = ""
        for col in range(n):
            line += " Q " if board[col] == row else " . "
        print(line)
    print()
```

```
def heuristic(board):
    n = len(board)
    attacks = 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):
                attacks += 1
    return attacks

def random_neighbor(board):
    n = len(board)
    neighbor = list(board)
    col = random.randint(0, n-1)
    row = random.randint(0, n-1)
    while row == neighbor[col]:
        row = random.randint(0, n-1)
    neighbor[col] = row
    return neighbor

def simulated_annealing(n=8, max_iter=100000, initial_temp=100, cooling_rate=0.995):
    current_board = get_user_input(n)
    current_heuristic = heuristic(current_board)
    temperature = initial_temp
    iteration = 0

    print(f"Initial heuristic: {current_heuristic}")
    print_board(current_board)

    while temperature > 0.1 and current_heuristic > 0 and iteration < max_iter:
        neighbor = random_neighbor(current_board)
        neighbor_heuristic = heuristic(neighbor)
        delta_e = current_heuristic - neighbor_heuristic

        if delta_e > 0:
            current_board = neighbor
            current_heuristic = neighbor_heuristic
        else:
            probability = math.exp(delta_e / temperature)
            if random.random() < probability:
                current_board = neighbor
                current_heuristic = neighbor_heuristic

        temperature *= cooling_rate
        iteration += 1

        if iteration % 1000 == 0:
            print(f"Iteration {iteration}, Temperature: {temperature:.2f}, Heuristic: {current_heuristic}")
            print_board(current_board)

    if current_heuristic == 0:
        print("Solution found!")
    else:
        print("Stopped without full solution. Best board found:")
    print(f"Final heuristic: {current_heuristic}")
    print_board(current_board)

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

Enter the row positions (0 to 7) of the queens for each column:
Column 0: 4
Column 1: 6
Column 2: 1
Column 3: 5
Column 4: 2
Column 5: 0
Column 6: 3
Column 7: 7

Initial heuristic: 0

```
. . . . . Q . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
```

Solution found!
Final heuristic: 0

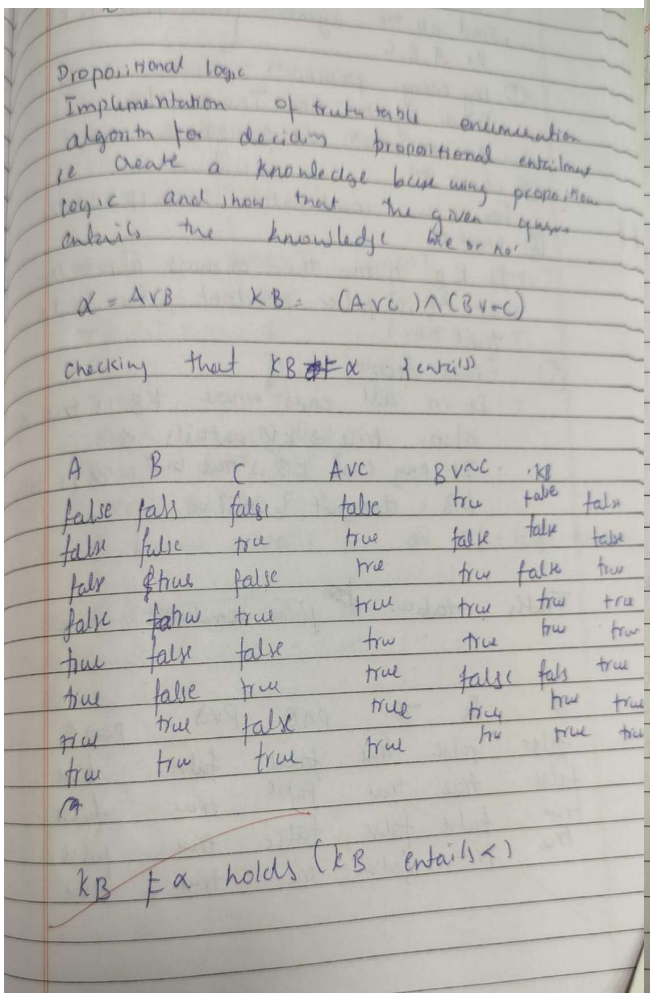
```
. . . . . Q . .
. . Q . . . . .
. . . . Q . . .
. . . . . . Q .
Q . . . . . . .
. . . Q . . . .
. Q . . . . . .
. . . . . . . Q
```

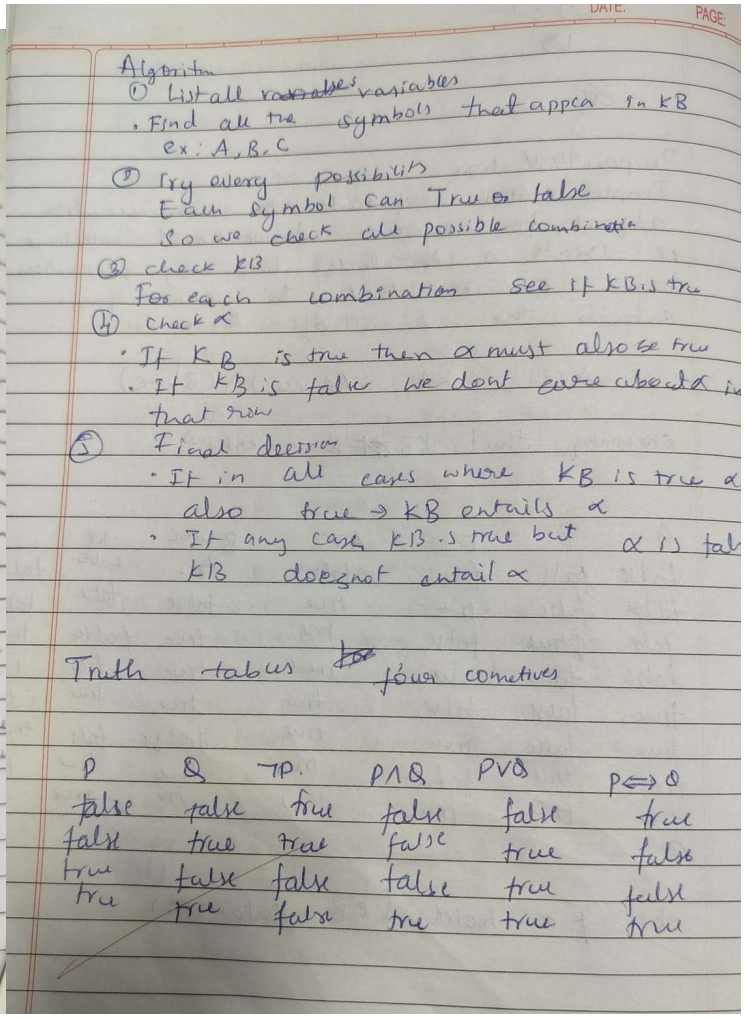
\

Program 6

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

Algorithm:





DATE: PAGE:

Consider S and T as variables and following relation

a: ⌐(S ∨ T)

b: (S ∧ T)

c: T ∨ ⌐T

Write truth table and show whether

i) a entails b

ii) a entails c

S	T	a = ⌐(S ∨ T)	b = S ∧ T	C = T ∨ T
T	T	F	T	T
T	F	F	F	T
F	T	F	F	T
F	F	T	F	T

Row 1: a = T, b = T ✓

Row 2: a = T, b = F ✗

Row 3: a = T, b = F ✗

a doesnot entail b as b is false for this

~~Row 2: a = T, C = T~~ ✓

Row 4: a = T, C = T ✓

Row 3: a = T

22/9/25

Code:

```
import itertools
import pandas as pd
import re
print("SUMANTH S SHETTY 1BM23CS348")
def replace_implications(expr):
    """
    Replace every X => Y with (not X or Y).
    This uses regex with a callback to avoid partial string overwrites.
    """
    pattern = r'([^=><]+?)\s*=>\s*([^=><]+?)(?=\s|$|[&|)])'
    while re.search(pattern, expr):
        expr = re.sub(pattern,
                      lambda m: f"(not {m.group(1).strip()} or {m.group(2).strip()})",
                      expr,
                      count=1)
    return expr

def pl_true(sentence, model):
    expr = sentence.strip()
    expr = expr.replace("<=>", "==")
    expr = replace_implications(expr)

    # Replace propositional symbols with their truth values safely
    for sym, val in model.items():
        expr = re.sub(rf'\b{sym}\b', str(val), expr)

    # Clean up spacing and add proper spacing for boolean operators
    expr = re.sub(r'\s+', ' ', expr)  # Remove extra spaces
    expr = expr.replace(" and ", " and ").replace(" or ", " or ").replace(" not ", " not ")

    return eval(expr)

def get_symbols(KB, alpha):
    symbols = set()
    for sentence in KB + [alpha]:
        for token in re.findall(r'\b[A-Za-z]+\b', sentence):
            if token not in ['and', 'or', 'not']:  # Exclude boolean operators
                symbols.add(token)
    return sorted(list(symbols))

def tt_entails(KB, alpha):
    symbols = get_symbols(KB, alpha)
    rows = []
    entails = True

    for values in itertools.product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        try:
```

```
            kb_val = all(pl_true(sentence, model) for sentence in KB)
            alpha_val = pl_true(alpha, model)

            rows.append({**model, "KB": kb_val, "alpha": alpha_val})

            if kb_val and not alpha_val:
                entails = False
        except Exception as e:
            print(f"Error evaluating with model {model}: {e}")
            return False

    df = pd.DataFrame(rows)

    # Print truth table nicely
    print("\n" + "="*50)
    print("                TRUTH TABLE")
    print("="*50)

    col_widths = {}
    for col in df.columns:
        col_widths[col] = max(len(str(col)), df[col].astype(str).str.len().max())

    table_width = sum(col_widths.values()) + len(df.columns) * 3 - 1

    print(" ┌" + "─" * table_width + "┐ ")

    header = " │ "
    for col in df.columns:
        header += f" {col:^{col_widths[col]}} │ "
    print(header)

    separator = " ├"
    for col in df.columns:
        separator += "─" * (col_widths[col] + 2) + "┼"
    separator = separator[:-1] + "┤ "
    print(separator)

    for _, row in df.iterrows():
        row_str = " │ "
        for col in df.columns:
            value = str(row[col])
            row_str += f" {value:^{col_widths[col]}} │ "
        print(row_str)

    print(" └" + "─" * table_width + "┘ ")

    print("\n" + "="*50)
    result_text = f"KB ENTAILS ALPHA: {'✓ YES' if entails else '✗ NO'}"
    print(f"{result_text:^50}")
    print("="*50)
    return entails

# --- Interactive input ---
print("Enter Knowledge Base (KB) sentences, separated by commas.")
print("Use symbols like A, B, C and operators: and, or, not, =>, <=>")
```

```
kb_input = input("KB: ").strip()
KB = [x.strip() for x in kb_input.split(",")]
alpha = input("Enter query (alpha): ").strip()

result = tt_entails(KB, alpha)
print(f"Result: {result}")
```

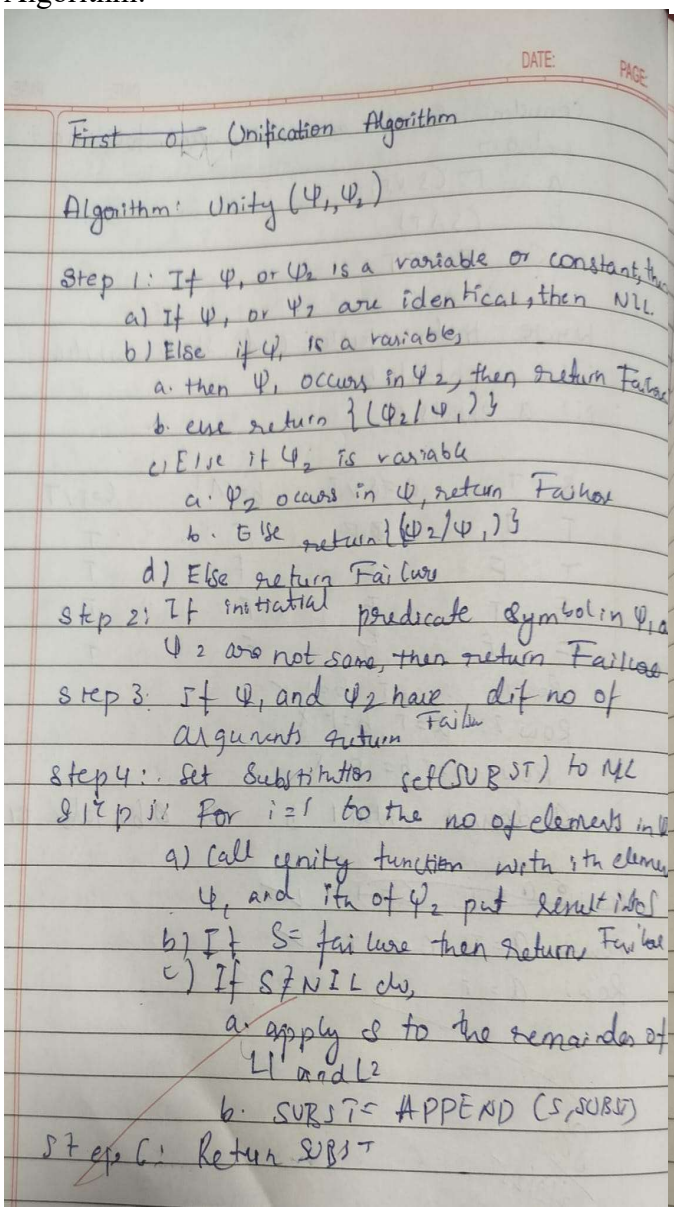
SUMANTH S SHETTY 1BM23CS348

```
Enter Knowledge Base (KB) sentences, separated by commas.
Use symbols like A, B, C and operators: and, or, not, =>, <=>
KB: not (S or T)
Enter query (alpha): T or (not T)

================================================
            TRUTH TABLE
================================================
┌──────────────────────────────────────┐
│  S   │  T   │  KB   │ alpha │
├──────┼──────┼───────┼───────┤
│ True  │ True  │ False │ True  │
│ True  │ False │ False │ True  │
│ False │ True  │ False │ True  │
│ False │ False │ True  │ True  │
└──────────────────────────────────────┘

================================================
        KB ENTAILS ALPHA: ✓ YES
================================================
Result: True
```

Program 7

Implement unification in first order logic

Algorithm:



First of Unification Algorithm

Algorithm: Unify(Ψ_1, Ψ_2)

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then
a) If Ψ_1 or Ψ_2 are identical, then NIL.
b) Else if Ψ_1 is a variable,
a. then Ψ_1 occurs in Ψ_2, then return Failure
b. else return $\{(\Psi_2/\Psi_1)\}$
c) Else if Ψ_2 is variable
a. Ψ_2 occurs in Ψ_1, return Failure
b. Else return $\{(\Psi_2/\Psi_1)\}$
d) Else return Failure

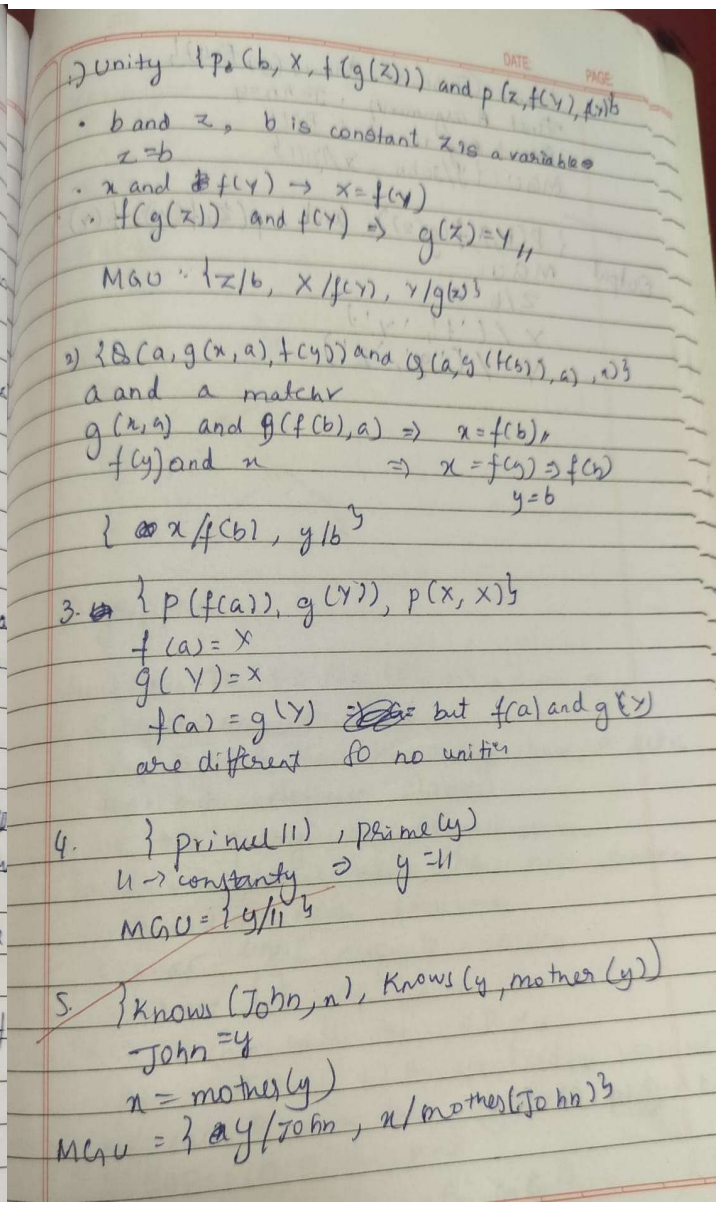
Step 2: If initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return Failure

Step 3: If Ψ_1 and Ψ_2 have a diff no of arguments return Failure

Step 4: Set Substitution set(SUBST) to NIL

Step 5: For i=1 to the no of elements in Ψ_1
a) Call unify function with ith element of Ψ_1 and ith of Ψ_2 put result into S
b) If S= failure then return Failure
c) If S≠NIL do,
a. apply S to the remainder of $L1$ and $L2$
b. SUBST= APPEND (S, SUBST)

Step 6: Return SUBST



1) Unify $\{P(b, X, f(g(z)))$ and $P(z, f(Y), f(Y))\}$
- b and z, b is constant z is a variable
z=b
- x and f(Y) → x=f(Y)
- f(g(z)) and f(Y) ⇒ g(z)=Y

MGU = $\{z/b, X/f(Y), Y/g(z)\}$

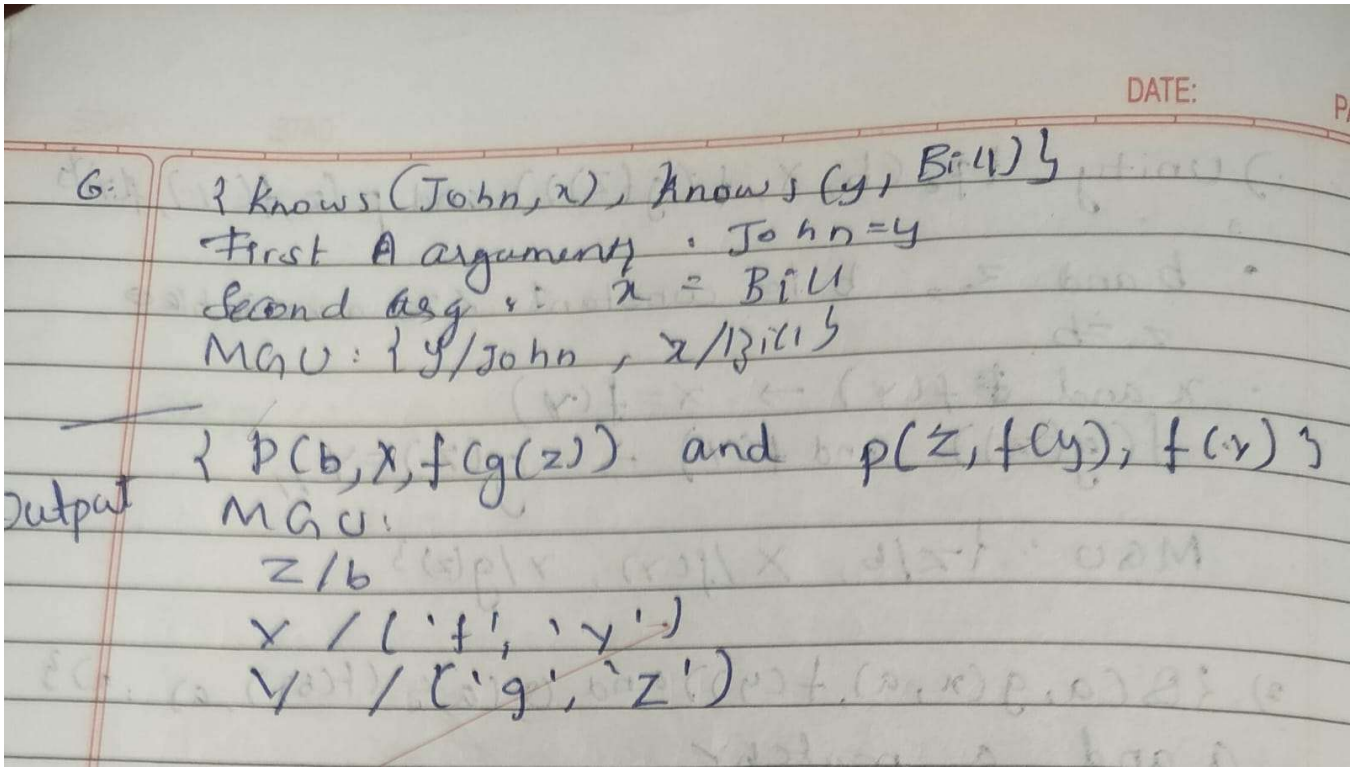
2) $\{P(a, g(x, a), f(y))$ and $P(a, g(f(b), a), x)\}$
a and a match
g(x,a) and g(f(b),a) ⇒ x=f(b)
f(y) and x ⇒ x=f(y) ⇒ f(b)
y=b

$\{x/f(b), y/b\}$

3. $\{P(f(a)), g(Y)), P(X, X)\}$
f(a)=X
g(Y)=X
f(a)=g(Y) but f(a) and g(Y) are different so no unifier

4. $\{Prime(11), Prime(y)\}$
u → constant ⇒ y=11
MGU = $\{y/11\}$

5. $\{Knows(John, x), Knows(y, mother(y))\}$
John=y
x=mother(y)
MGU = $\{y/John, x/mother(John)\}$



Code:

```
print("Sumanth S Shetty 1BM23CS348")
def is_variable(x):
    # Variables are single uppercase letters only, like 'X', 'Y', 'Z'
    return isinstance(x, str) and len(x) == 1 and x.isupper()

def is_ (x):
    return isinstance(x, tuple) and len(x) > 0

def occurs_check(var, term, subst):
    if var == term:
        return True
    elif is_variable(term) and term in subst:
        return occurs_check(var, subst[term], subst)
    elif is_compound(term):
        return any(occurs_check(var, t, subst) for t in term[1:])
    return False

def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    if x == y:
        return subst
    if is_variable(x):
        return unify_var(x, y, subst)
    elif is_variable(y):
        return unify_var(y, x, subst)
    elif is_compound(x) and is_compound(y):
        if x[0] != y[0] or len(x) != len(y):
            return None
        for x_arg, y_arg in zip(x[1:], y[1:]):
```

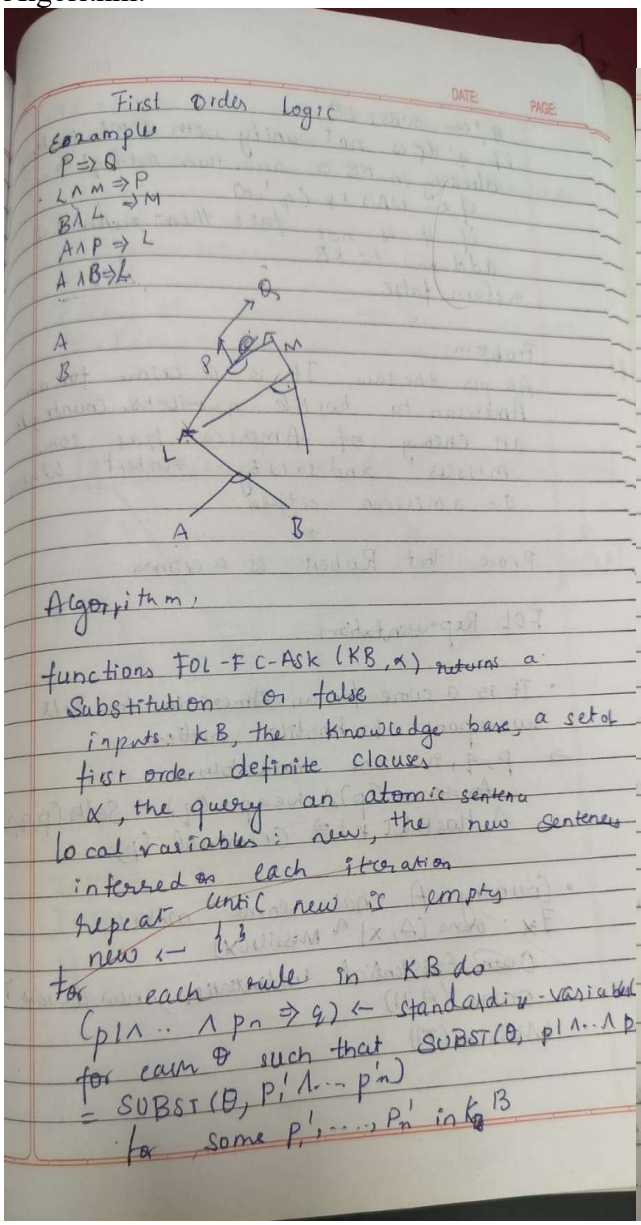
```
            subst = unify(x_arg, y_arg, subst)
            if subst is None:
                return None
        return subst
    else:
        return None

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif is_variable(x) and x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst

def print_substitution(subst):
    if subst is None:
        print("No unifier exists")
    else:
        print("MGU:")
        for var, val in subst.items():
            print(f"{var} / {val}")

# Terms:
expr1 = ('p', 'b', 'X', ('f', ('g', 'Z')))
expr2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

result = unify(expr1, expr2)
print_substitution(result)
```

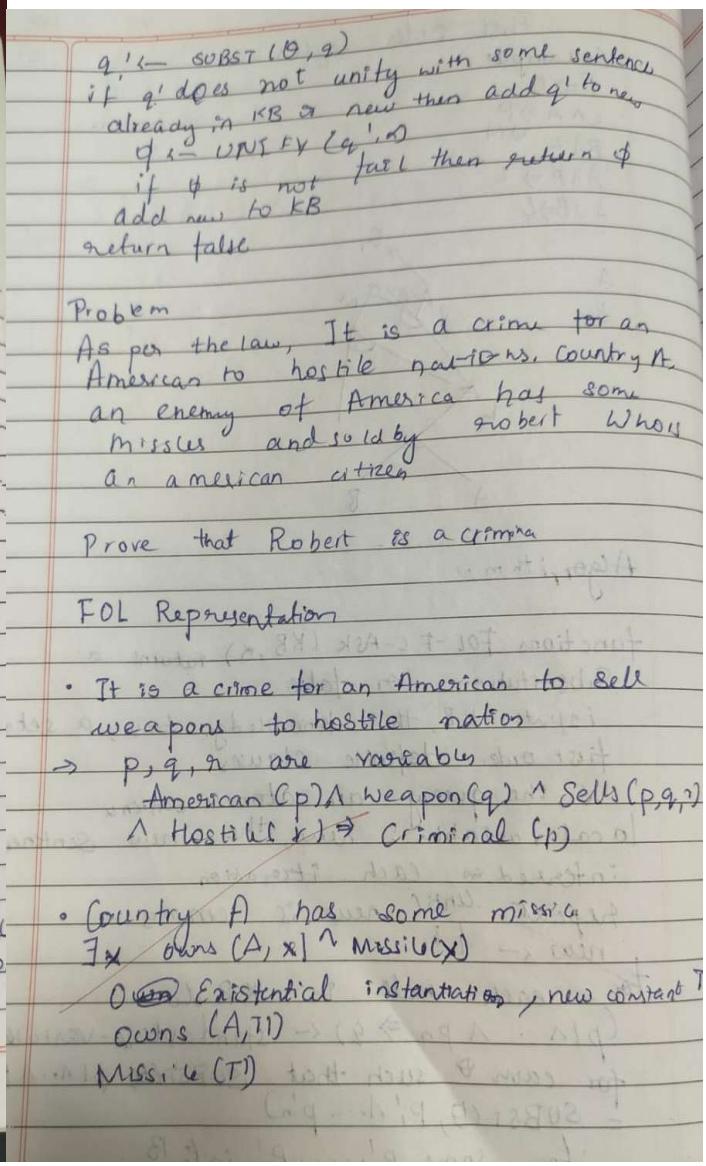
Sumanth S Shetty 1BM23CS348
MGU:
Z / b
X / ('f', 'Y')
Y / ('g', 'Z')

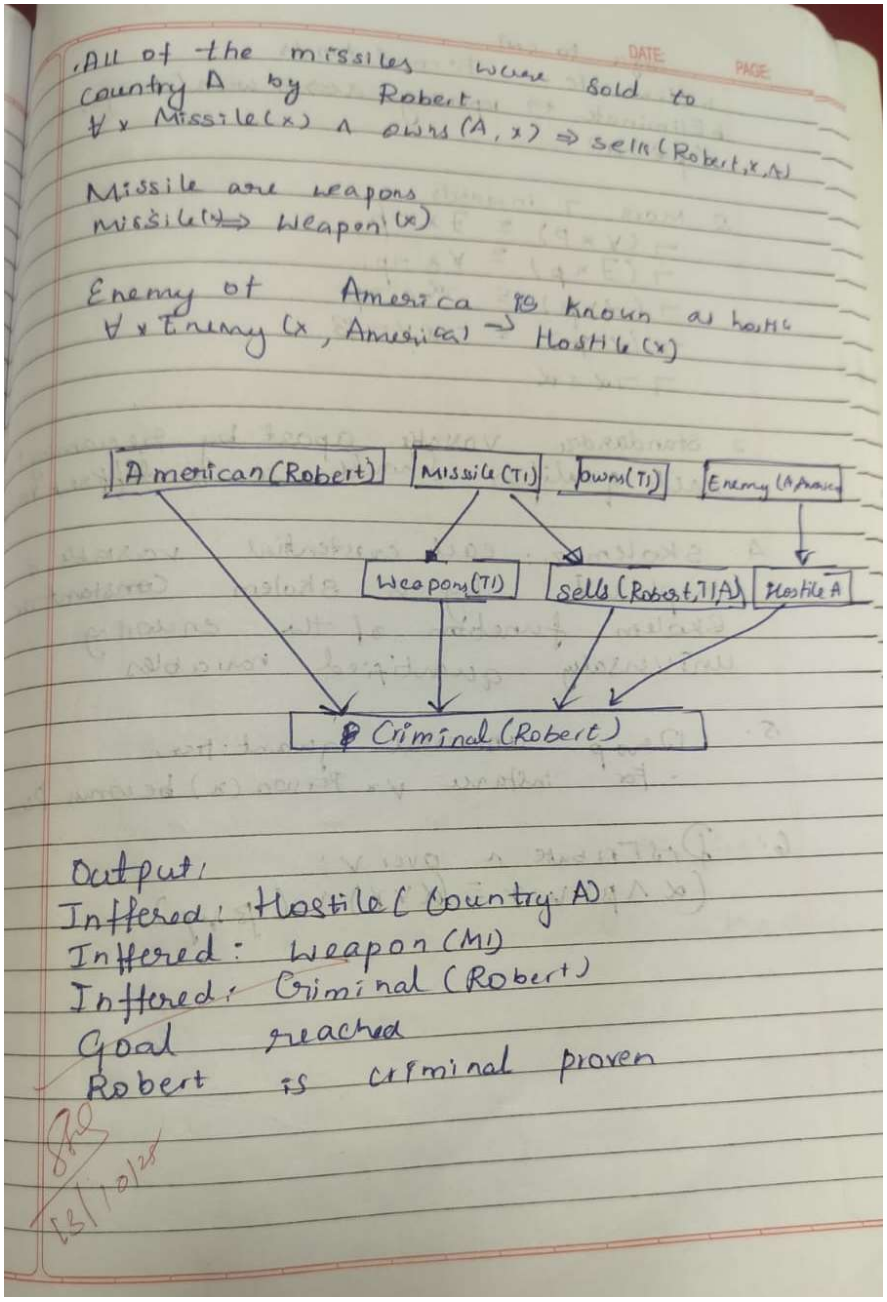
Program 8

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

Algorithm:







Code:

```
# Define rules and facts in a simple form
print("sumanth S Shetty 1bm23cs348")
# Facts (initial knowledge base)
facts = {
    "American(Robert)",
    "Enemy(CountryA, America)",
    "Missile(M1)",
    "Has(CountryA, M1)",
    "Sold(Robert, CountryA, M1)"
}

# Rules (implications)
rules = [
    # If a country is enemy to America, then it is hostile
```

```
    ("Enemy(X, America)", "Hostile(X)"),

    # If Robert sold weapons to a hostile nation and Robert is American, Robert is a criminal
    ("American(Robert) & Sold(Robert, Y, Z) & Hostile(Y) & Weapon(Z)", "Criminal(Robert)"),

    # Define what is a weapon (missile is a weapon)
    ("Missile(Z)", "Weapon(Z)")
]

# Parse facts and rules simply using sets and strings
def forward_chaining(facts, rules, goal):
    new_facts = set(facts)
    while True:
        added = False
        for premise, conclusion in rules:
            # Try to match premise to facts (handle & conjunctions)
            premises = premise.split(" & ")
            substitutions = [{}]  # List of possible substitutions

            for p in premises:
                new_substitutions = []
                for sub in substitutions:
                    # Substitute variables in p using sub
                    p_inst = substitute(p, sub)
                    matches = match_fact(p_inst, new_facts)
                    for m in matches:
                        # Merge substitutions
                        merged = merge_substitutions(sub, m)
                        if merged is not None:
                            new_substitutions.append(merged)
                substitutions = new_substitutions

            # Apply substitutions to conclusion and add to facts
            for sub in substitutions:
                conclusion_inst = substitute(conclusion, sub)
                if conclusion_inst not in new_facts:
                    new_facts.add(conclusion_inst)
                    print(f"Inferred: {conclusion_inst}")
                    added = True
                    if conclusion_inst == goal:
                        print("Goal reached!")
                        return True
        if not added:
            break
    return goal in new_facts

# Substitute variables with their bindings
def substitute(expr, sub):
    for var, val in sub.items():
        expr = expr.replace(var, val)
    return expr

# Match premise to facts with variable bindings
def match_fact(premise, facts):
    # premise like Predicate(arg1,arg2,...)
    import re
    pattern = re.compile(r"(\w+)")
    m = pattern.match(premise)
```

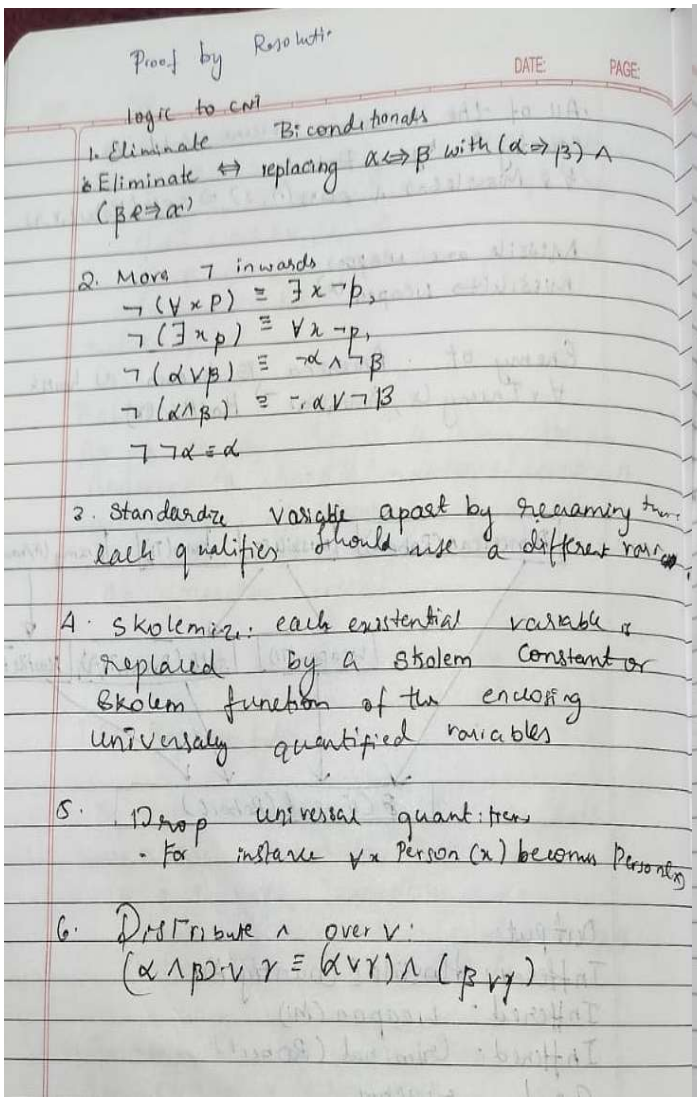
```
    if not m:
        return []
    pred = m.group(1)
    args = m.group(2).split(",")
    args = [a.strip() for a in args]
    matches = []
    for f in facts:
        fm = pattern.match(f)
        if not fm:
            continue
        fpred = fm.group(1)
        fargs = fm.group(2).split(",")
        fargs = [a.strip() for a in fargs]
        if fpred != pred or len(fargs) != len(args):
            continue
        sub = {}
        failed = False
        for a, fa in zip(args, fargs):
            if is_variable(a):
                if a in sub and sub[a] != fa:
                    failed = True
                    break
                sub[a] = fa
            else:
                if a != fa:
                    failed = True
                    break
        if not failed:
            matches.append(sub)
    return matches

# Merge two substitutions if compatible
def merge_substitutions(s1, s2):
    s = s1.copy()
    for k,v in s2.items():
        if k in s and s[k] != v:
            return None
        s[k] = v
    return s

# Check if a term is a variable (single uppercase letter)
def is_variable(x):
    return len(x) == 1 and x.isupper()

# Run the forward chaining to prove Criminal(Robert)
goal = "Criminal(Robert)"
if forward_chaining(facts, rules, goal):
    print("Robert is criminal: Proven")
else:
    print("Could not prove Robert is criminal")
```

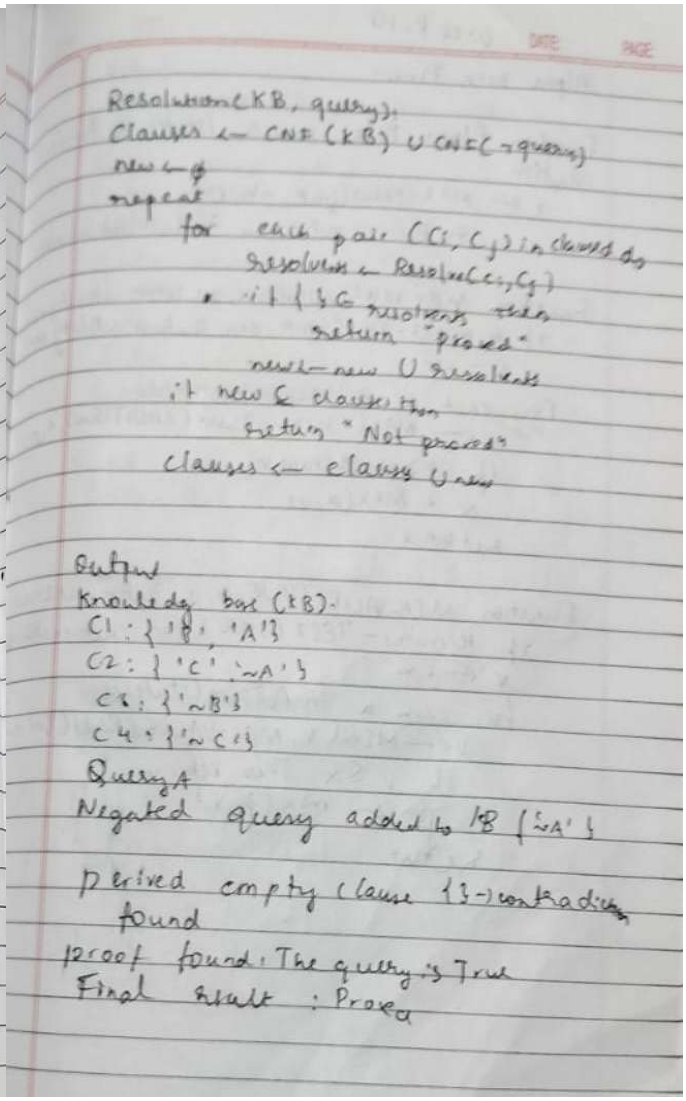
```
sumanth S Shetty 1bm23cs348
Inferred: Hostile(CountryA)
Inferred: Weapon(M1)
Inferred: Criminal(Robert)
Goal reached!
Robert is criminal: Proven
```

Program 9

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

Algorithm





DATE: PAGE:

week-9 continue

Proof by Resln that
¬John likes peanuts

Representation in FOL

(a) $\forall x$: food(x) → likes (John, x)

(b) food (Apple) ∧ food (vegetables)

Proof By Resolution

John likes all kind of food

Apple and vegetables are good

Anything anyone eats and not killed is food

Anil eats peanuts and still alive

Harry eats everything that Anil eats

Anyone who is alive implies not killed

Anyone who is not killed implies alive

Prove by resolution

~~Given~~

John likes peanuts

Representation in FOL

a. $\forall x$ food(x) → likes (John, x)

food (Apple) ∧ food (Vegetables)

$\forall x \forall y$: eats (x, y) ∧ ¬ killed (x) → food (y)

eats (Anil, peanuts) ∧ Alive (Anil)

$\forall x$: eats (Anil, x) → eats (Harry, x)

$\forall x$: ¬ killed (x) → alive (x)

$\forall x$ alive (x) → ¬ killed (x)

likes (John, peanuts)

a $\forall x$ ¬ food(x) ∨ likes (John, x)

b food(Apple) ∧ food vegetables

c $\forall x \forall y$ ¬ [eats (x, y) ∧ ¬ killed (y)] ∨ food(y)

d eats (Anil, peanuts) ∧ alive (Anil)

e $\forall x$ ¬ eats (Anil, x) ∨ eats (Harry, x)

f $\forall x$ ¬ [¬ killed (x)] ∨ alive(x)

g $\forall x$ ¬ alive(x) ∨ ¬ killed(x)

h

Rename variables

$\forall x$ ¬ food (x) ∨ likes (John, x)

food (Apple) ∧ food vegetables

$\forall y \forall z$ ¬ eats (y, z) ∨ killed (y) ∨ food(z)

eats (Anil, peanuts) ∧ alive (Anil)

$\forall w$ ¬ eats (Anil, w) ∨ eats (Harry, w)

$\forall g$ killed (g)] ∨ alive(g)

$\forall k$ ¬ alive(k) ∨ ¬ killed (k)

~~likes (John~~

drop universal

¬ food (x) ∨ likes (John, x)

food (Apple)

food (vegetables)

¬ eats (y, z) ∨ killed (y) ∨ food (z)

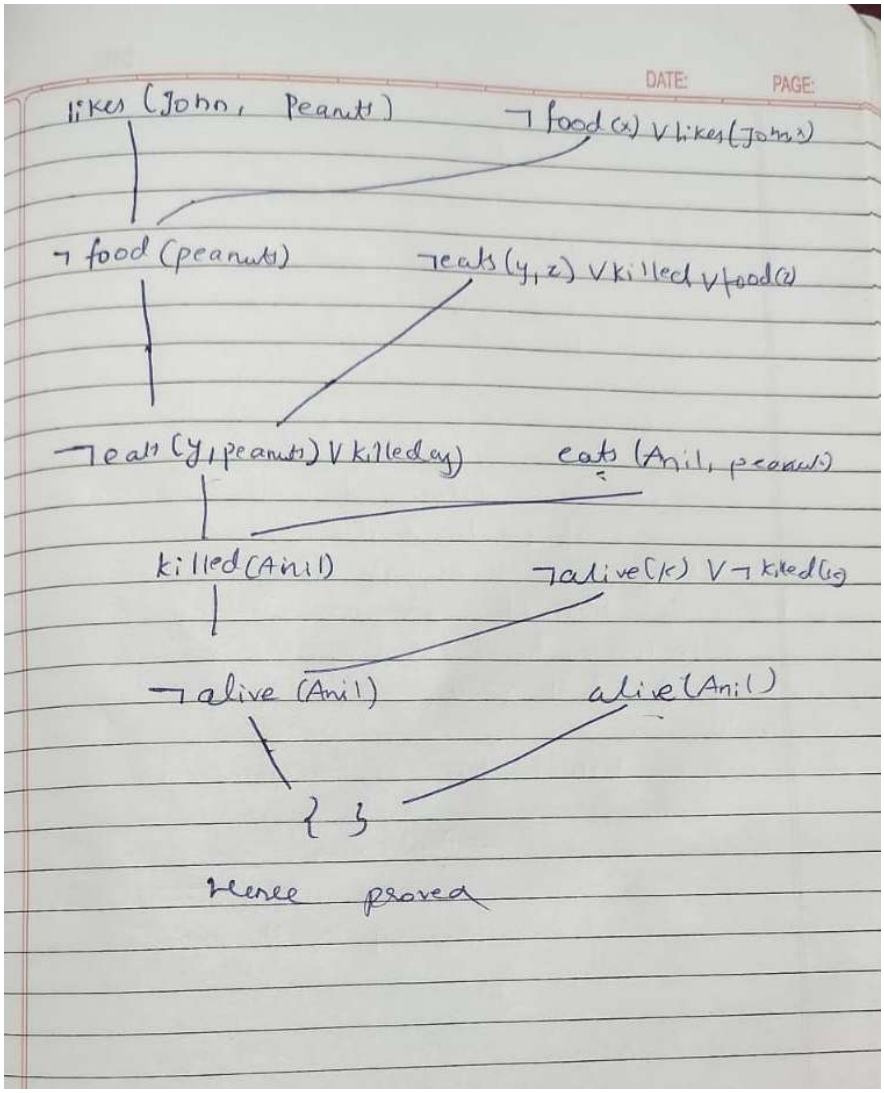
eats (Anil, peanuts)

alive (Anil)

¬ eats (Anil, w) ∨ eats (Harry, w)

killed (g) ∨ alive (g)

¬ alive (k) ∨ ¬ killed (k)



Code:

```
def resolve(ci, cj):
    """
    Try to resolve two clauses ci and cj.
    Returns a set of resolvents (new clauses).
    """
    resolvents = set()
    for di in ci:
        for dj in cj:
            # Check for complementary literals
            if di == ('~' + dj) or dj == ('~' + di):
                new_clause = (ci - {di}) | (cj - {dj})
                resolvents.add(frozenset(new_clause))
    return resolvents


def resolution(kb, query):
    """
    Proof by resolution.
    kb: list of clauses (each a set of literals)
    query: literal to prove
    """
    # Negate query and add it to the KB
```

```
    negated_query = {('~' + query) if not query.startswith('~') else query[1:]}
    clauses = set(map(frozenset, kb))
    clauses.add(frozenset(negated_query))

    print("Knowledge Base (KB):")
    for i, clause in enumerate(kb, start=1):
        print(f"  C{i}: {clause}")
    print(f"\nQuery: {query}")
    print(f"Negated Query Added to KB: {negated_query}\n")
    print("-" * 40)

    new = set()
    step = 1

    while True:
        pairs = [(ci, cj) for i, ci in enumerate(clauses)
                          for j, cj in enumerate(clauses) if i < j]
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if resolvents:
                #  FIXED PRINT LINE (avoid unhashable type error)
                print(f"Step {step}: Resolving {set(ci)} and {set(cj)} → {[set(r) for r in resolvents]}")
                step += 1

            if frozenset() in resolvents:
                print("\nDerived empty clause {} → Contradiction found!")
                print("proof Found: The query is TRUE.")
                return True
            new = new.union(resolvents)

        if new.issubset(clauses):
            print("\nNo new clauses can be generated.")
            print("Proof Not Possible: The query cannot be proved.")
            return False

        clauses = clauses.union(new)


# Example usage:
if __name__ == "__main__":
    # Example KB: (A ∨ B), (¬A ∨ C), (¬B), (¬C)
    kb = [
        {'A', 'B'},
        {'~A', 'C'},
        {'~B'},
        {'~C'}
    ]
    query = 'A'  # Try to prove A

    result = resolution(kb, query)
    print("\nFinal Result:", "Proved " if result else "Not Proved ")
 output
Knowledge Base (KB):
  C1: {'A', 'B'}
  C2: {'~A', 'C'}
  C3: {'~B'}
  C4: {'~C'}
```

Query: A
Negated Query Added to KB: {'~A'}

Step 1: Resolving {'A', 'B'} and {'~A', 'C'} → [{'B', 'C'}]
Step 2: Resolving {'A', 'B'} and {'~B'} → [{'A'}]
Step 3: Resolving {'~A', 'C'} and {'~C'} → [{'~A'}]
Step 4: Resolving {'A'} and {'~A'} → [set()]

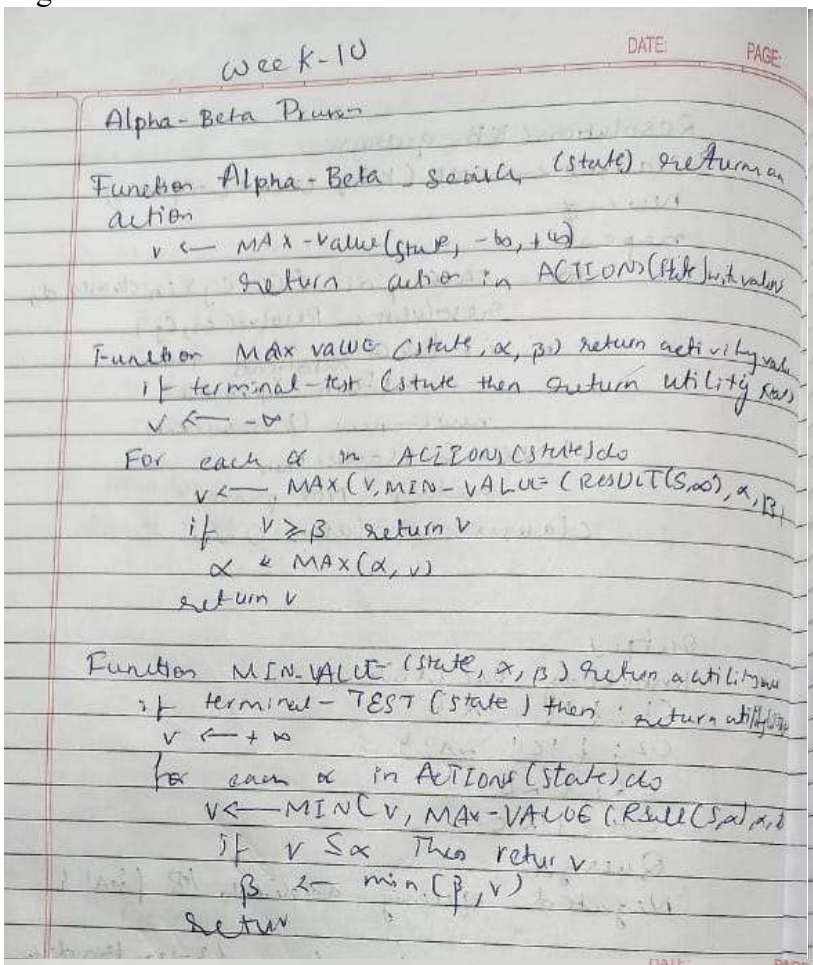
Derived empty clause {} → Contradiction found!
proof Found: The query is TRUE.

Final Result: Proved

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Week-10

Alpha-Beta Pruning

Function Alpha-Beta_search (state) returns an action

v ← MAX-value(state, -∞, +∞)

return action in ACTIONS(state) with value v

Function MAX value (state, α, β) return utility value

if terminal-test (state) then return utility value

v ← -∞

For each α in ACTIONS(state) do

v ← MAX(v, MIN-VALUE(RESULT(s,a)), α, β)

if v ≥ β return v

α ← MAX(α, v)

return v

Function MIN-VALUE (state, α, β) return utility value

if terminal-TEST (state) then return utility value

v ← +∞

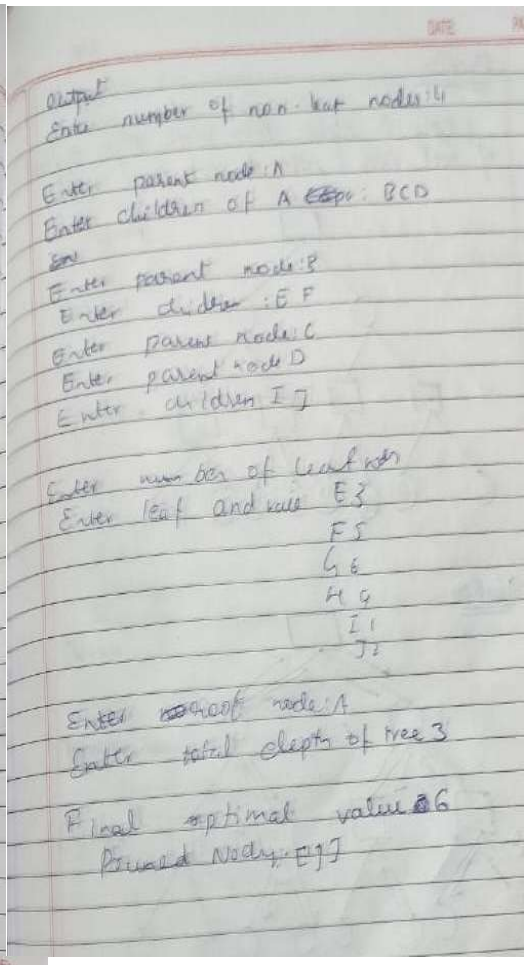
for each α in ACTIONS(state) do

v ← MIN(v, MAX-VALUE(RESULT(s,a)), α, β)

if v ≤ α then return v

β ← min(β, v)

return v



Output

Enter number of non-leaf nodes: 4

Enter parent node: A

Enter children of A: B C D

Enter parent node: B

Enter children: E F

Enter parent node: C

Enter parent node: D

Enter children: I J

Enter number of leaf nodes: 8

Enter leaf and value: E 3

F 5

G 6

H 4

I 1

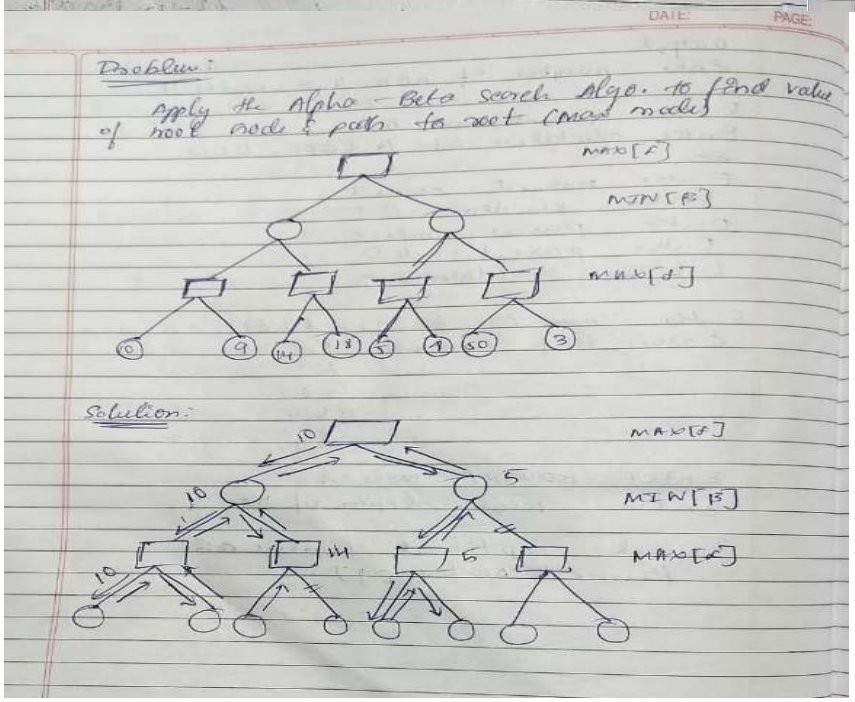
J 2

Enter root node: A

Enter total depth of tree: 3

Final optimal value: 6

Pruned Nodes: E J



Problem:

Apply the Alpha-Beta search Algo. to find value of root node & path to root (max nodes)

MAX[A]

MIN[B]

MAX[D]

(10) (9) (14) (18) (5) (4) (50) (3)

Solution:

MAX[A]

MIN[B]

MAX[D]

10 10 10 14 5 5

Code:

```
import networkx as nx
import matplotlib.pyplot as plt
import math

def alpha_beta(node, depth, alpha, beta, is_max, tree, values, pruned, path):
    if depth == 0 or node not in tree:
        return values.get(node, None)

    if is_max:
        best, chosen = -math.inf, None
        for child in tree[node]:
            val = alpha_beta(child, depth - 1, alpha, beta, False, tree, values, pruned, path)
            if val is None:
                continue
            if val > best:
                best, chosen = val, child
            alpha = max(alpha, best)
            if beta <= alpha:
                i = tree[node].index(child) + 1
                for c in tree[node][i:]:
                    pruned.append(c)
                break
        values[node] = best
        if chosen:
            path.append((node, chosen))
        return best
    else:
        best, chosen = math.inf, None
        for child in tree[node]:
            val = alpha_beta(child, depth - 1, alpha, beta, True, tree, values, pruned, path)
            if val is None:
                continue
            if val < best:
                best, chosen = val, child
            beta = min(beta, best)
            if beta <= alpha:
                i = tree[node].index(child) + 1
                for c in tree[node][i:]:
                    pruned.append(c)
                break
        values[node] = best
        if chosen:
            path.append((node, chosen))
        return best

def draw_tree(G, path_edges, pruned):
    pos = nx.nx_agraph.graphviz_layout(G, prog="dot")
    plt.figure(figsize=(9, 6))
    edge_colors = []
    for u, v in G.edges():
        if (u, v) in path_edges:
            edge_colors.append('green')
        elif v in pruned:
            edge_colors.append('red')
        else:
            edge_colors.append('black')
    node_colors = []
```

```
        for node in G.nodes():
            if any(node in e for e in path_edges):
                node_colors.append('green')
            elif node in pruned:
                node_colors.append('red')
            else:
                node_colors.append('skyblue')
        nx.draw(G, pos, with_labels=True, node_color=node_colors, edge_color=edge_colors,
                node_size=1200, font_size=10)
        plt.title("Alpha-Beta Pruning Tree\nGreen = Optimal Path | Red = Pruned Nodes")
        plt.show()

    def main():
        tree, G = {}, nx.DiGraph()
        n = int(input("Enter number of non-leaf nodes: "))
        for _ in range(n):
            parent = input("\nEnter parent node: ").strip()
            children = input(f"Enter children of {parent}: ").split()
            tree[parent] = children
            for c in children:
                G.add_edge(parent, c)

        leaf_count = int(input("\nEnter number of leaf nodes: "))
        values = {}
        for _ in range(leaf_count):
            leaf, val = input("Enter leaf node and its value: ").split()
            values[leaf] = int(val)

        root = input("\nEnter root node: ").strip()
        depth = int(input("Enter total depth of tree: "))

        pruned, path = [], []
        print("\n---------------------------------")
        result = alpha_beta(root, depth, -math.inf, math.inf, True, tree, values, pruned, path)
        print(f"Final Optimal Value: {result}")
        print(f"Pruned Nodes: {pruned}")
        print("---------------------------------")

        draw_tree(G, path_edges=path, pruned=pruned)

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

output

```
Enter number of non-leaf nodes: 4

Enter parent node: A
Enter children of A: B C D

Enter parent node: B
Enter children of B: E F

Enter parent node: C
Enter children of C: G H

Enter parent node: D
Enter children of D: I J
```

Enter number of leaf nodes: 6
Enter leaf node and its value: E 3
Enter leaf node and its value: F 5
Enter leaf node and its value: G 6
Enter leaf node and its value: H 9
Enter leaf node and its value: I 1
Enter leaf node and its value: J 2

Enter root node: A
Enter total depth of tree: 3
Final Optimal Value: 6
Pruned Nodes: ['J']