VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

DHRUVDEEP NAYAK(1BM23CS093)

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 **DHRUVDEEP Nayak (1BM23CS093),** who is a 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.Sheetal VA Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

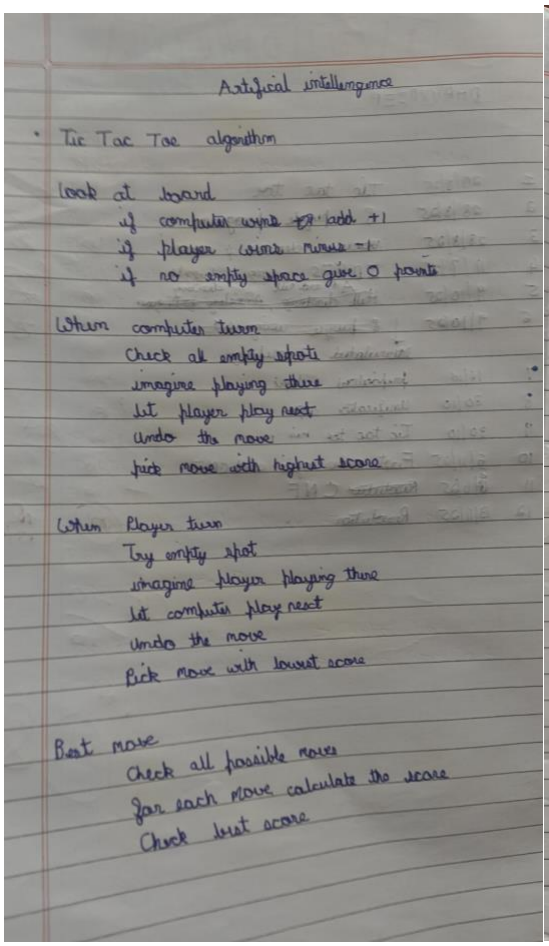
INDEX

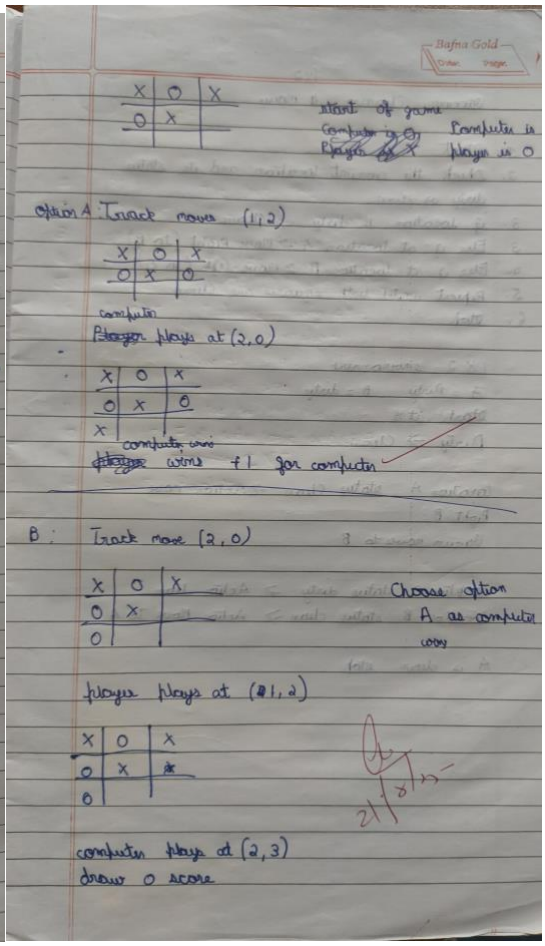
10	14-11-2025	Resolution	48 - 50

GITHUB: https://github.com/DhruvdeepBMSCE/AIlab

Program 1: Implement Tic –Tac –Toe Game & Implement vacuum cleaner agent.

ALGORITHM (Tic-Tac-Toe) :





CODE (Tic-Tac-Toe) :

```
def computer_move():
    best_move = minimax_recurse(game_board, active_player, 0)
    print("The best move is ", best_move)
    make_move(game_board, best_move, active_player)

    print("COMPUTER MOVE DONE")

def minimax_recurse(game_board, player, depth):
    winner = is_winner(game_board)
    if winner == active_player:
        return 1
    elif winner is not None and winner != active_player:
        return -1
    elif len(get_move_list(game_board)) == 0:
        return 0

    if player == player1:
```

```
        other_player = player2
    else:
        other_player = player1

    if player == active_player:
        alpha = -1
    else:
        alpha = 1

    movelist = get_move_list(game_board)
    best_move = None

    for move in movelist:
        board2 = [list(row) for row in game_board] # Create a copy of the board

        make_move(board2, move, player)

        subalpha = minimax_recurse(board2, other_player, depth + 1)

        if player == active_player:
            if depth == 0 and alpha <= subalpha:
                best_move = move
            alpha = max(alpha, subalpha)
            if alpha == 1: # Alpha-beta pruning
                if depth == 0:
                  return best_move
                return alpha

        else:
            alpha = min(alpha, subalpha)
            if alpha == -1: # Alpha-beta pruning
                return alpha

    if depth == 0:
      return best_move
    return alpha


# BOARD FUNCTIONS
game_board = [['1','2','3'],['4','5','6'],['7','8','9']] # Changed to strings to avoid confusion with
available moves
def print_board(board) :

    for row in board :
```

```
        print(row)

def make_move(game_board, player_move, active_player):

    x = 0
    y = 0
    try:
        player_move = int(player_move)
    except ValueError:
        print("Invalid input. Please enter a number.")
        return game_board

    if player_move == 1 :
        x = 0
        y = 0
    elif player_move == 2 :
        x = 0
        y = 1
    elif player_move == 3 :
        x = 0
        y = 2
    elif player_move == 4 :
        x = 1
        y = 0
    elif player_move == 5 :
        x = 1
        y = 1
    elif player_move == 6 :
        x = 1
        y = 2
    elif player_move == 7 :
        x = 2
        y = 0
    elif player_move == 8 :
        x = 2
        y = 1
    elif player_move == 9 :
        x = 2
        y = 2
    else :
        print ("value is out of range")
        return game_board
```

```
    if game_board[x][y] == "O" or game_board[x][y] == "X" :
        print("move not available")
        return game_board

    game_board[x][y] = str(active_player)
    return game_board

def is_winner(board):
    for i in range (0,3) :
        if board[i][0] == player1 and board[i][1] == player1 and board[i][2] == player1 :
            return player1

        if board[i][0] == player2 and board[i][1] == player2 and board[i][2] == player2 :
            return player2

    # checking for columns
    for i in range(0,3):
        if board[0][i] == player1 and board[1][i] == player1 and board[2][i] == player1:
            return player1
        if board[0][i] == player2 and board[1][i] == player2 and board[2][i] == player2:
            return player2

    # checking for diagonals
    if board[0][0] == player1 and board[1][1] == player1 and board[2][2] == player1 :
        return player1
    if board[0][0] == player2 and board[1][1] == player2 and board[2][2] == player2 :
        return player2

    if board[2][0] == player1 and board[1][1] == player1 and board[0][2] == player1 :
        return player1
    if board[2][0] == player2 and board[1][1] == player2 and board[0][2] == player2 :
        return player2

    return None


def get_move_list (game_board) :

    move = []

    for row in game_board :
        for i in row :
            if i.isdigit():
                move.append(int(i))
```

```
    return move

# Main Loop
player1 = "X"
player2 = "O"
print_board(game_board)
while True :
    active_player = player1
    # this is for player move
    print(get_move_list(game_board))
    player_move = input("Please insert your move >>> ")
    game_board = make_move(game_board,player_move,active_player)
    print_board(game_board)

    if is_winner(game_board) == player1 :
        print("Player1 is the winner")
        break
    if is_winner(game_board) == player2 :
        print("Player2 is the winner")
        break
    if len(get_move_list(game_board)) == 0:
        print("It's a tie!")
        break
    print(get_move_list(game_board))
    # computer time
    active_player = player2
    computer_move()
    print_board(game_board)
    if is_winner(game_board) == player1 :
        print("Player1 is the winner")
        break
    if is_winner(game_board) == player2 :
        print("Player2 is the winner")
        break
    if len(get_move_list(game_board)) == 0:
        print("It's a tie!")
        break
```

OUTPUT:

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
Enter your move (row and column: 0 1 2): 2 1
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   | X |   |
-------------
AI played:
-------------
|   | O |   |
-------------
|   |   |   |
-------------
|   | X |   |
-------------
Enter your move (row and column: 0 1 2): 2 2
-------------
|   | O |   |
-------------
|   |   |   |
-------------
|   | X | X |
-------------
AI played:
-------------
|   | O |   |
-------------
|   |   |   |
```

ALGORITHM (Vacuum Cleaner)

28/8/25

Vacuum Cleaner 4 Rooms

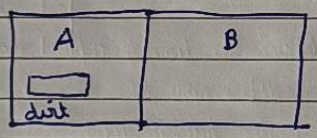
Algorithm

Step 1: Initialize the number of rooms and assign a number to each room

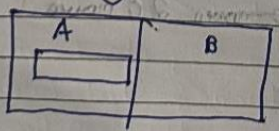
Step 2: Install a for loop and create two functions move and clean

Step 3: vaccum cleaner checks if the current room is clean or not using the sensors

Step 4: if the room is clean the vaccum cleaner performs the move() function if not it clean() function and the move() function to next room



room A is found to be dirty execute clean() function



room A is clean implement move() function

CODE (Vacuum cleaner) :

```
import random

# Define environment
rooms = {
    'A': random.choice(['Clean', 'Dirty']),
    'B': random.choice(['Clean', 'Dirty']),
    'C': random.choice(['Clean', 'Dirty']),
    'D': random.choice(['Clean', 'Dirty'])
}

# Initial position of the agent
agent_position = random.choice(['A', 'B', 'C', 'D'])

# Display current state
def display_state():
    print(f"Agent is in Room {agent_position}")
    for room, status in rooms.items():
        print(f"Room {room}: {status}")
    print()

# Rule-based agent logic with smart movement
def vacuum_agent():
    global agent_position
    steps = 0

    # Keep track of cleaned rooms
    cleaned_rooms = set()

    while len(cleaned_rooms) < 4:
        display_state()

        # Clean current room if dirty
        if rooms[agent_position] == 'Dirty':
            print(f" Cleaning Room {agent_position}")
            rooms[agent_position] = 'Clean'
            cleaned_rooms.add(agent_position)
        else:
            print(f"Room {agent_position} is already clean.")
            cleaned_rooms.add(agent_position)

        # Decide next move only if not all rooms are clean
        if len(cleaned_rooms) < 4:
```

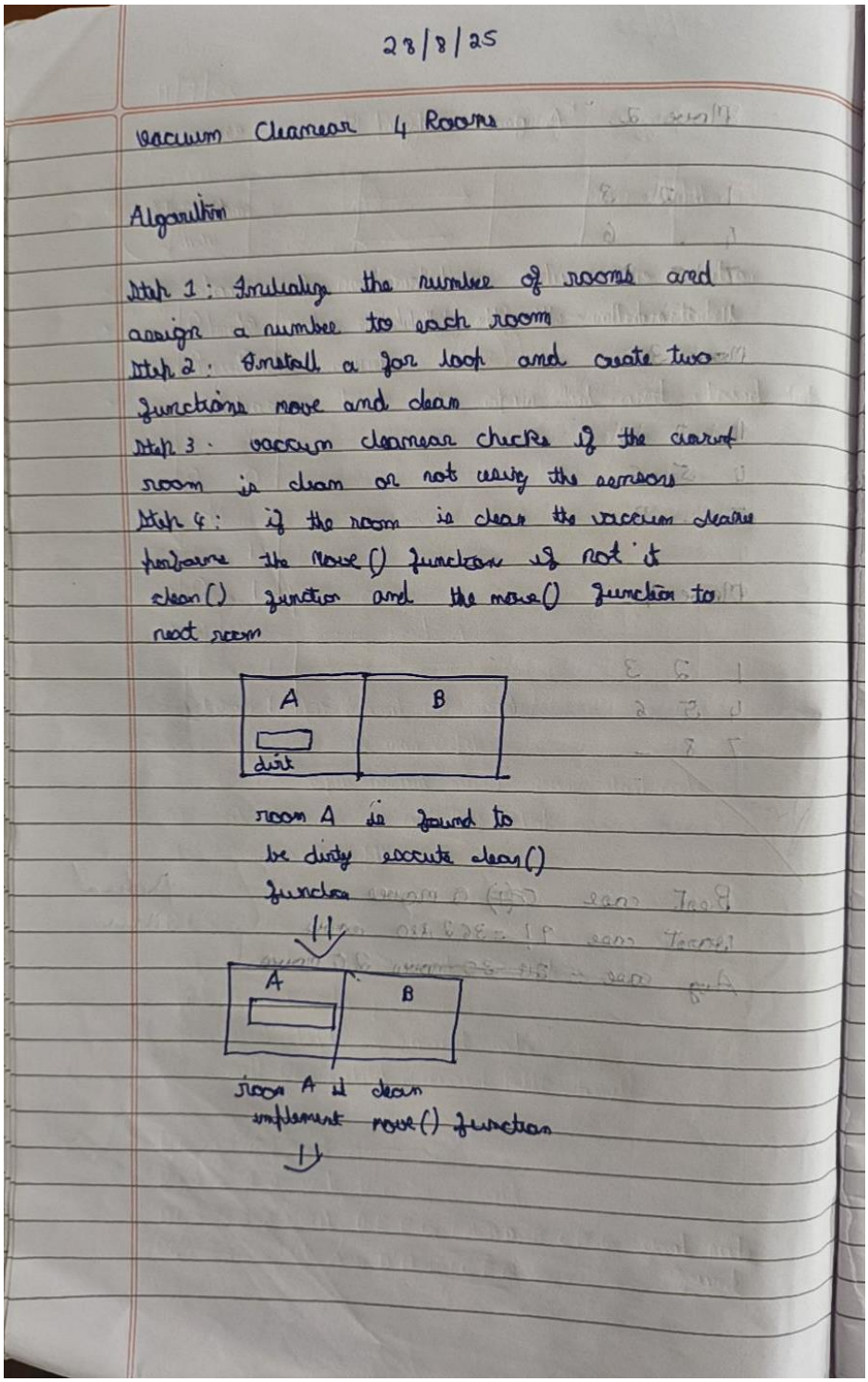
```
            # Move to the next room that is still dirty
            for next_room in ['A', 'B', 'C', 'D']:
                if next_room not in cleaned_rooms:
                    agent_position = next_room
                    break

        steps += 1
        print(f"Step {steps} complete.\n")

    print("All rooms are clean!")
    display_state()

vacuum_agent()
```

OUTPUT :

```
Agent is in Room B
Room A: Clean
Room B: Dirty
Room C: Clean
Room D: Dirty

Cleaning Room B
Step 1 complete.

Agent is in Room A
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Dirty

Room A is already clean.
Step 2 complete.

Agent is in Room C
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Dirty

Room C is already clean.
Step 3 complete.

Agent is in Room D
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Dirty

Cleaning Room D
Step 4 complete.

 All rooms are clean!
Agent is in Room D
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Clean
```

Program 2 : Implement 8-puzzle

ALGORITHM :



23/8/25

Vacuum Cleaner 4 Rooms

Algorithm

Step 1: Initialize the number of rooms and assign a number to each room

Step 2: Install a for loop and create two functions move and clean

Step 3: vacuum cleaner checks if the current room is clean or not using the sensors

Step 4: if the room is clean the vacuum cleaner performs the move() function if not it clean() function and the move() function to next room

room A is found to be dirty execute clean() function

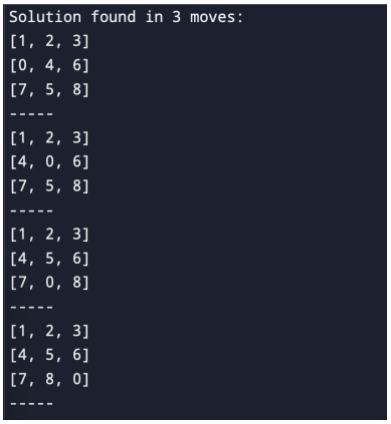
room A is clean implement move() function

CODE :

```
import heapq

goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

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

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

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value == 0:
                continue
            goal_x, goal_y = divmod(value - 1, 3)
            distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def a_star(start_state):
    visited = set()
    priority_queue = []
    # f(n) = g(n) + h(n)
    heapq.heappush(priority_queue, (manhattan_distance(start_state), 0, start_state, []))

    while priority_queue:
        f, g, current_state, path = heapq.heappop(priority_queue)

        if current_state == goal_state:
            return path + [current_state]
```

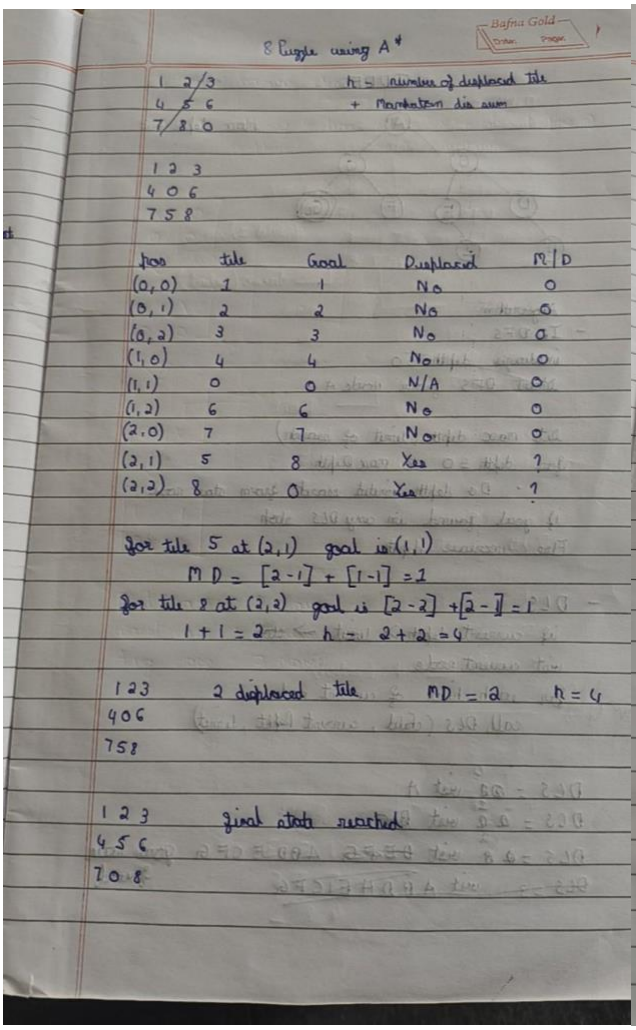
```
        visited.add(state_to_tuple(current_state))
        x, y = find_blank(current_state)

        for dx, dy in moves:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny):
                new_state = [row[:] for row in current_state]
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]

                if state_to_tuple(new_state) not in visited:
                    g_new = g + 1
                    h_new = manhattan_distance(new_state)
                    f_new = g_new + h_new
                    heapq.heappush(priority_queue, (f_new, g_new, new_state, path + [current_state]))

    return None

start_state = [[1, 2, 3],
               [0, 4, 6],
               [7, 5, 8]]

solution_path = a_star(start_state)

if solution_path:
    print("Solution found in", len(solution_path) - 1, "moves:")
    for step in solution_path:
        for row in step:
            print(row)
        print("-----")
else:
    print("No solution found.")
```

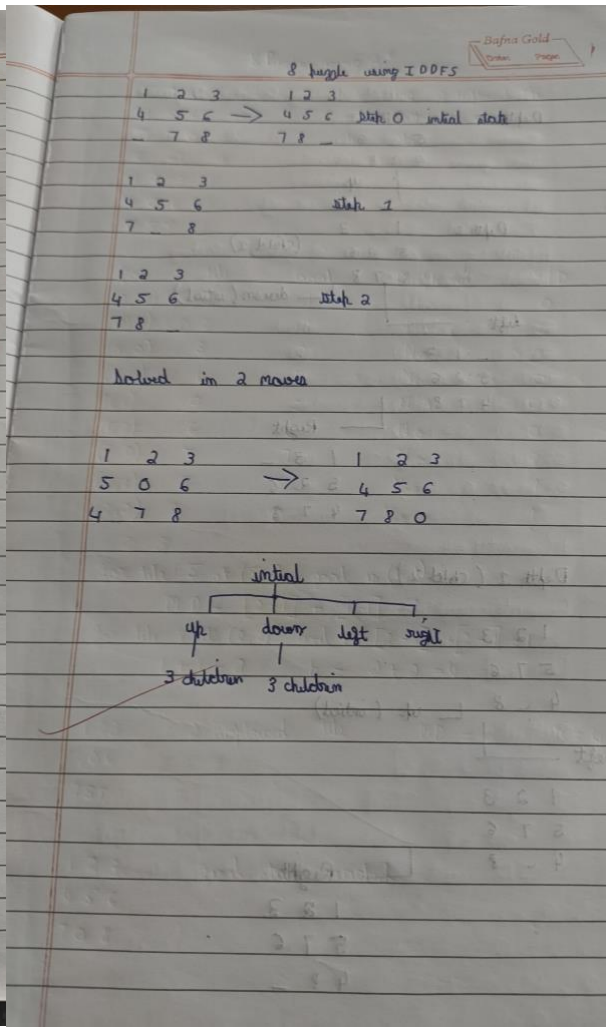
OUTPUT :

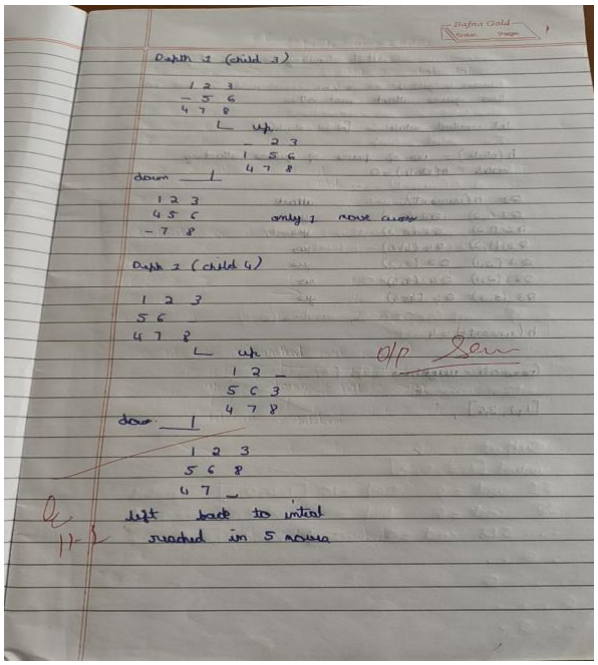


Program 3: Implement IDDFS

ALGORITHM :







CODE :

```
from collections import deque

# Directions for movement
MOVES = {
    'Up': -3,
    'Down': 3,
    'Left': -1,
    'Right': 1
}

# Define the goal state
GOAL_STATE = (1, 2, 3,
              4, 5, 6,
              7, 8, 0)

# Valid indices for moves
def valid_moves(index):
    moves = []
    row, col = divmod(index, 3)

    if row > 0: moves.append('Up')
    if row < 2: moves.append('Down')
    if col > 0: moves.append('Left')
    if col < 2: moves.append('Right')

    return moves
```

```
# Apply move to a state
def apply_move(state, move):
    idx = state.index(0)
    new_idx = idx + MOVES[move]

    # Special case for left/right edge wrapping
    if move == 'Left' and idx % 3 == 0:
        return None
    if move == 'Right' and idx % 3 == 2:
        return None

    state = list(state)
    state[idx], state[new_idx] = state[new_idx], state[idx]
    return tuple(state)

# DFS with depth limit
def dls(state, depth, visited, path):
    if state == GOAL_STATE:
        return path

    if depth == 0:
        return None

    visited.add(state)
    for move in valid_moves(state.index(0)):
        next_state = apply_move(state, move)
        if next_state and next_state not in visited:
            result = dls(next_state, depth - 1, visited.copy(), path + [(move, next_state)])
            if result:
                return result
    return None

# IDDFS main function
def iddfs(start_state, max_depth=50):
    for depth in range(max_depth):
        print(f"\n--- Iteration {depth + 1}: Depth Limit = {depth} ---")
        visited = set()
        path = dls(start_state, depth, visited, [])
        if path is not None:
            return path
    return None

# Function to print the puzzle state in a readable format
```

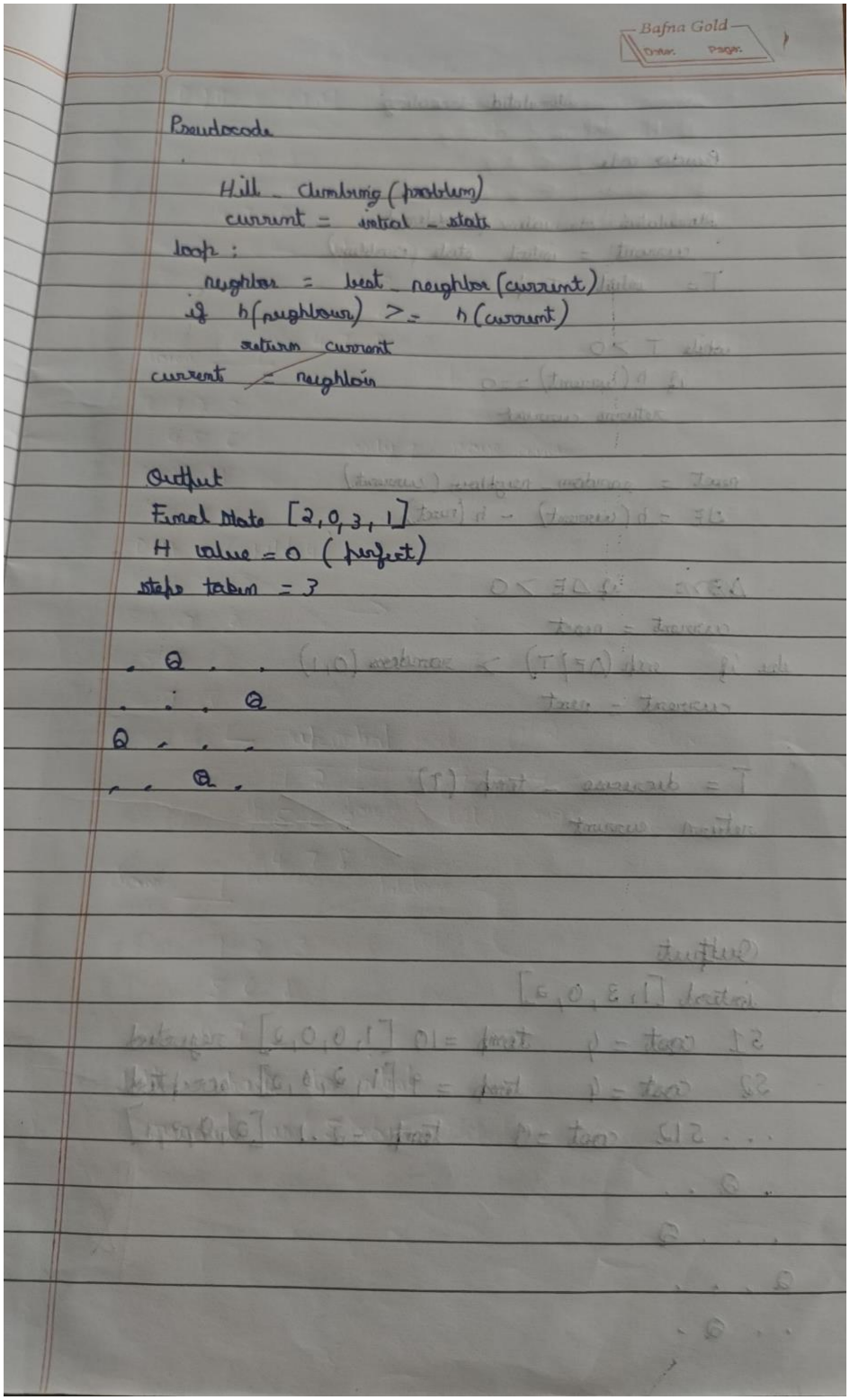
```
def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Example usage
if __name__ == '__main__':
    start = (1, 2, 3,
             4, 5, 6,
             0, 7, 8)

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

    solution = iddfs(start)
    if solution:
        print(f"Solution found in {len(solution)} moves:")
        current_state = start
        for move, state in solution:
            print(f"Move: {move}")
            print_state(state)
            current_state = state
    else:
        print("No solution found.")
```

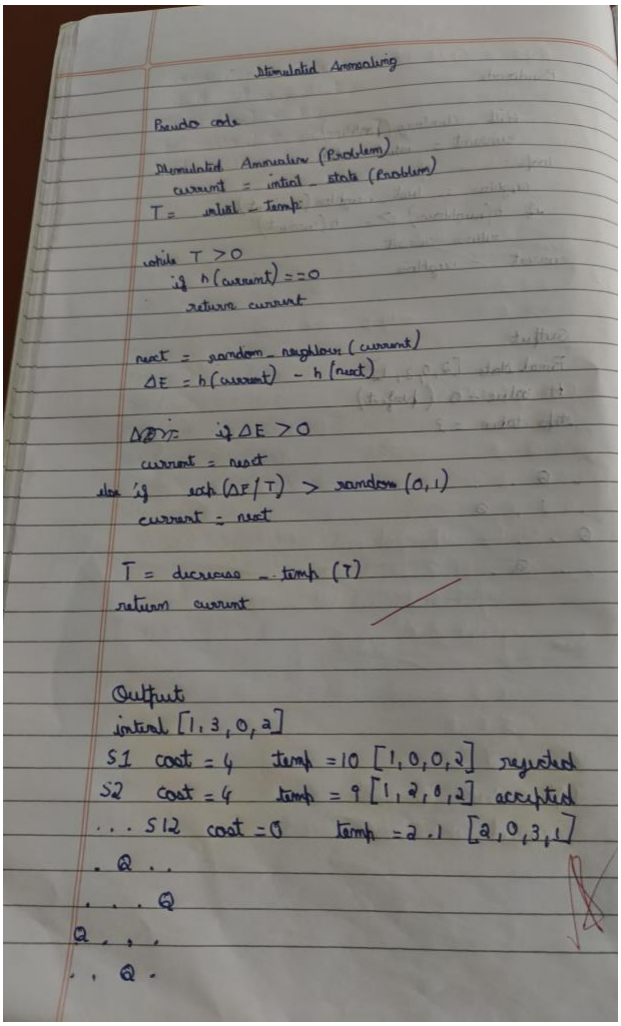
OUTPUT :

```
--- IDDFS iteration with limit = 2 ---
call dfs(node=A, depth=0, limit=2)
  visit A (depth 0) -> current order: A
call dfs(node=B, depth=1, limit=2)
  visit B (depth 1) -> current order: AB
call dfs(node=D, depth=2, limit=2)
  visit D (depth 2) -> current order: ABD
call dfs(node=H, depth=3, limit=2)
  depth 3 > limit 2 - stop, backtrack
backtrack from D (depth 2)
call dfs(node=E, depth=2, limit=2)
  visit E (depth 2) -> current order: ABDE
call dfs(node=I, depth=3, limit=2)
  depth 3 > limit 2 - stop, backtrack
backtrack from E (depth 2)
backtrack from B (depth 1)
call dfs(node=C, depth=1, limit=2)
  visit C (depth 1) -> current order: ABDEC
call dfs(node=F, depth=2, limit=2)
  visit F (depth 2) -> current order: ABDECF
backtrack from F (depth 2)
call dfs(node=G, depth=2, limit=2)
  visit G (depth 2) -> current order: ABDECFG
backtrack from G (depth 2)
backtrack from C (depth 1)
backtrack from A (depth 0)
Visited order at limit 2: ABDECFG

Final output (order at last limit): ABDECFG
```

Program 4: Implement Hill Climbing & Simulated Annealing

ALGORITHM :



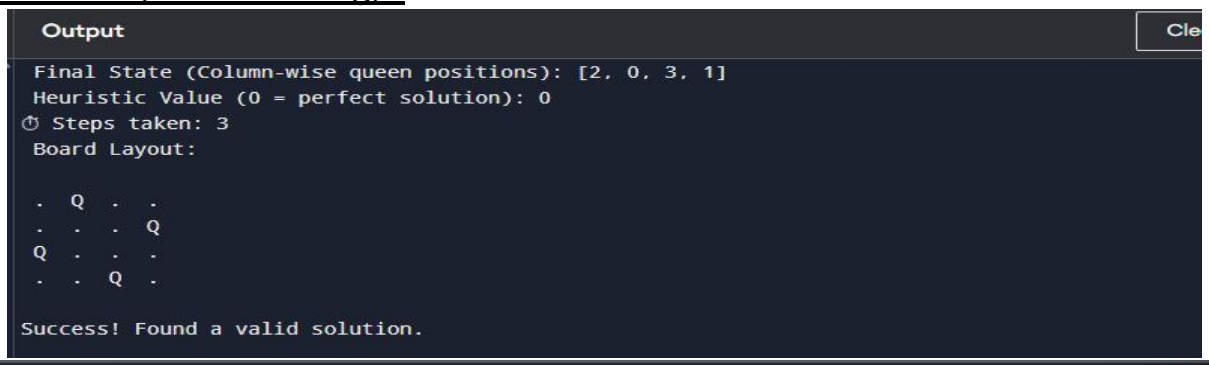


CODE (Hill Climbing) :

```
import random
def heuristic(board):
    h = 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]) == abs(i-j):
                h += 1
    return h

def hill_climbing_restart(initial_board, max_restarts=100):
    N = len(initial_board)
    board = [x-1 for x in initial_board]  # 0-based
    h = heuristic(board)
```

```
    restart_count = 0
    while h != 0 and restart_count < max_restarts:
        steps = 0
        while True:
            best_board = board[:]
            best_h = h
            for col in range(N):
                for row in range(N):
                    if row != board[col]:
                        neighbor = board[:]
                        neighbor[col] = row
                        h_neighbor = heuristic(neighbor)
                        if h_neighbor < best_h:
                            best_board = neighbor
                            best_h = h_neighbor
            steps += 1
            if best_h >= h:  # stuck
                break
            board = best_board
            h = best_h
            if h == 0:
                break
        if h == 0:
            print(f"Solution found after {restart_count} restarts and {steps} steps.")
            break
        # Random restart
        board = [random.randint(0, N-1) for _ in range(N)]
        h = heuristic(board)
        restart_count += 1

    return [x+1 for x in board], h

# User input
N = int(input("Enter number of queens (N): "))
print(f"Enter the initial positions of {N} queens (row numbers 1 to {N}):")
initial_board = list(map(int, input().split()))

solution, h_val = hill_climbing_restart(initial_board)
print("Final board:", solution)
print("Heuristic H =", h_val)
```

OUTPUT (Hill Climbing) :




```
Enter number of queens (N): 4
Enter the initial positions of 4 queens (row numbers 1 to 4):
1 2 1 4
Solution found after 1 restarts and 2 steps.
Final board: [2, 4, 1, 3]
Heuristic H = 0

=== Code Execution Successful ===
```

CODE (Simulated Annealing):

```
from datetime import datetime
import random, time, math
from copy import deepcopy, copy
import decimal

class Board:
    def __init__(self, queen_count=4):
        self.queen_count = queen_count
        self.reset()

    def reset(self):
        self.queens = [-1 for i in range(0, self.queen_count)]

        for i in range(0, self.queen_count):
            self.queens[i] = random.randint(0, self.queen_count - 1)
            # self.queens[row] = column


    def calculateCost(self):
        threat = 0

        for queen in range(0, self.queen_count):
            for next_queen in range(queen+1, self.queen_count):
```

```
            if self.queens[queen] == self.queens[next_queen] or abs(queen - next_queen) ==
abs(self.queens[queen] - self.queens[next_queen]):
                threat += 1

        return threat

    @staticmethod
    def calculateCostWithQueens(queens):
        threat = 0
        queen_count = len(queens)

        for queen in range(0, queen_count):
            for next_queen in range(queen+1, queen_count):
                if queens[queen] == queens[next_queen] or abs(queen - next_queen) ==
abs(queens[queen] - queens[next_queen]):
                    threat += 1

        return threat

    @staticmethod
    def toString(queens):
        board_string = ""

        for row, col in enumerate(queens):
            board_string += "(%s, %s)\n" % (row, col)

        return board_string

    def getLowerCostBoard(self):
        displacement_count = 0
        temp_queens = self.queens
        lowest_cost = self.calculateCost(temp_queens)

        for i in range(0, self.queen_count):
            temp_queens[i] = (temp_queens[i] + 1) % (self.queen_count - 1)

            for j in range(queen+1, self.queen_count):
                temp_queens[j] = (temp_queens[j] + 1) % (self.queen_count - 1)

    def __str__(self):
        board_string = ""

        for row, col in enumerate(self.queens):
            board_string += "(%s, %s)\n" % (row, col)
```

```
        return board_string

class SimulatedAnnealing:
    def __init__(self, board):
        self.elapsedTime = 0;
        self.board = board
        self.temperature = 4000
        self.sch = 0.99
        self.startTime = datetime.now()


    def run(self):
        board = self.board
        board_queens = self.board.queens[:]
        solutionFound = False

        for k in range(0, 170000):
            self.temperature *= self.sch
            board.reset()
            successor_queens = board.queens[:]
            dw = Board.calculateCostWithQueens(successor_queens) -
Board.calculateCostWithQueens(board_queens)
            exp = decimal.Decimal(decimal.Decimal(math.e) ** (decimal.Decimal(-dw) *
decimal.Decimal(self.temperature)))

            if dw > 0 or random.uniform(0, 1) < exp:
                board_queens = successor_queens[:]

            if Board.calculateCostWithQueens(board_queens) == 0:
                print("Solution:")
                print(Board.toString(board_queens))
                self.elapsedTime = self.getElapsedTime()
                print("Success, Elapsed Time: %sms" % (str(self.elapsedTime)))
                solutionFound = True
                break

        if solutionFound == False:
            self.elapsedTime = self.getElapsedTime()
            print("Unsuccessful, Elapsed Time: %sms" % (str(self.elapsedTime)))

        return self.elapsedTime

    def getElapsedTime(self):
```

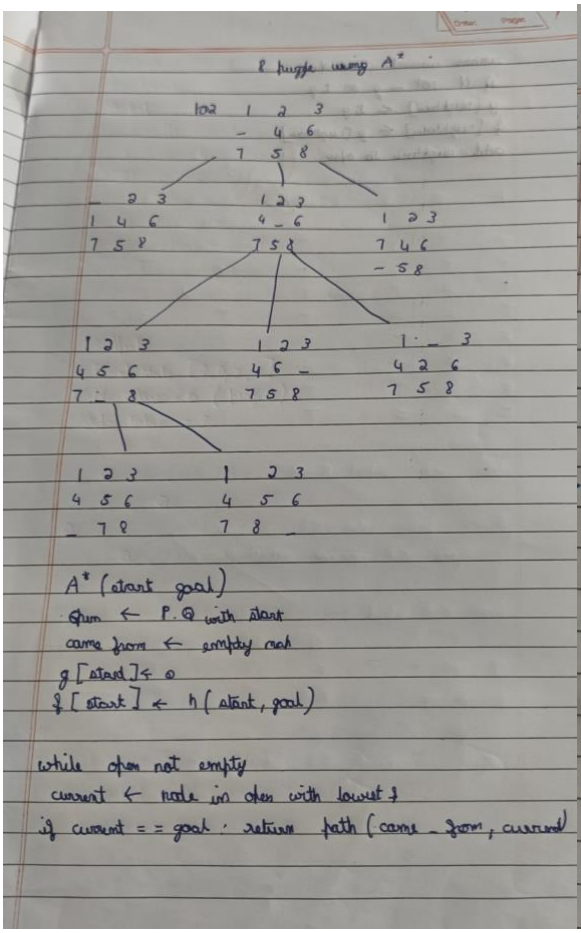
```
        endTime = datetime.now()
        elapsedTime = (endTime - self.startTime).microseconds / 1000
        return elapsedTime


if __name__ == '__main__':
    board = Board()
    print("Board:")
    print(board)
    SimulatedAnnealing(board).run()
```

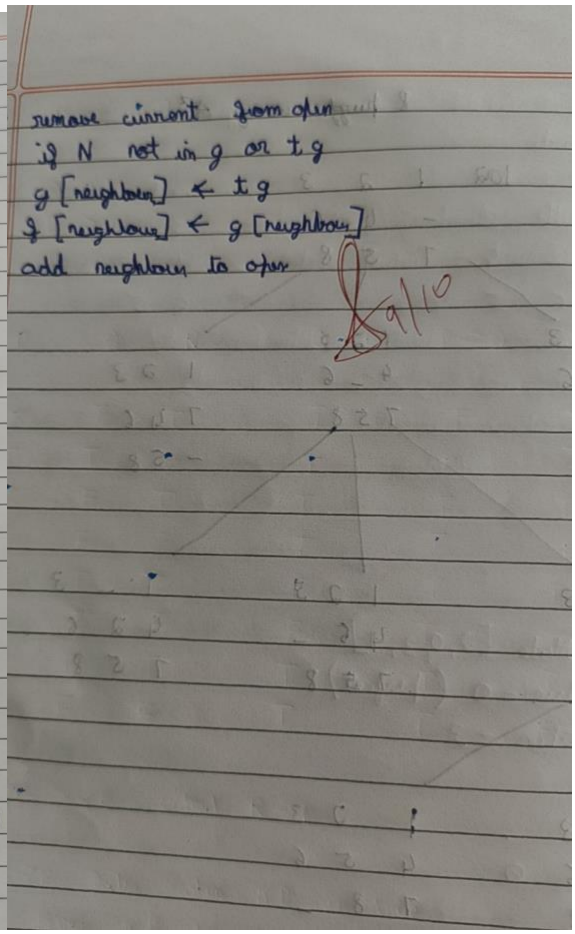
OUTPUT (Simulated Annealing):

```
Board:
(0, 1)
(1, 0)
(2, 1)
(3, 3)

Solution:
(0, 2)
(1, 0)
(2, 3)
(3, 1)

Success, Elapsed Time: 7.9ms
```

Program 5: A* Algorithm

ALGORITHM :





CODE :

```
import heapq

class PuzzleState:
    def __init__(self, board, parent=None, move="", depth=0, cost=0):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.cost = cost

    def __lt__(self, other):
        return self.cost < other.cost

    def blank_pos(self):
```

```
        return self.board.index(0)

    def expand(self):
        b = self.blank_pos()
        row, col = divmod(b, 3)
        dirs = {
            "Up": (row - 1, col),
            "Down": (row + 1, col),
            "Left": (row, col - 1),
            "Right": (row, col + 1)
        }
        nxt = []
        for mv, (r, c) in dirs.items():
            if 0 <= r < 3 and 0 <= c < 3:
                idx = r * 3 + c
                nb = self.board[:]
                nb[b], nb[idx] = nb[idx], nb[b]
                nxt.append(PuzzleState(nb, self, mv, self.depth + 1))
        return nxt

    def build_path(self):
        p, node = [], self
        while node:
            p.append((node.move, node.board, node.depth))
            node = node.parent
        return list(reversed(p))

def misplaced_tiles(state, goal):
    return sum(1 for i in range(9) if state.board[i] not in (0, goal[i]))

def manhattan_distance(state, goal):
    d = 0
    for i, v in enumerate(state.board):
        if v != 0:
            r1, c1 = divmod(i, 3)
            r2, c2 = divmod(goal.index(v), 3)
            d += abs(r1 - r2) + abs(c1 - c2)
    return d

def a_star(start, goal, h):
    opened = []
    closed = set()
    nodes = 0
    s = PuzzleState(start)
```

```
    s.cost = h(s, goal)
    heapq.heappush(opened, s)

    while opened:
        cur = heapq.heappop(opened)
        nodes += 1

        if cur.board == goal:
            return cur.build_path(), nodes

        closed.add(tuple(cur.board))

        for nxt in cur.expand():
            if tuple(nxt.board) in closed:
                continue
            nxt.cost = nxt.depth + h(nxt, goal)
            heapq.heappush(opened, nxt)

    return None, nodes

def print_solution(path, total_nodes):
    print("Steps:\n")
    for mv, st, d in path:
        label = "Start" if mv == "" else f"Move {mv}"
        print(f"{label} | Depth {d}")
        for i in range(0, 9, 3):
            print(" ".join(str(x) if x != 0 else " " for x in st[i:i+3]))
        print()
    print(f"Total Moves: {len(path)-1}")
    print(f"Nodes Expanded: {total_nodes}")

if __name__ == "__main__":
    start = [1, 2, 3,
             4, 0, 6,
             7, 5, 8]

    goal = [1, 2, 3,
            4, 5, 6,
            7, 8, 0]

    print("A* (Misplaced Tiles)\n")
    sol1, n1 = a_star(start, goal, misplaced_tiles)
    if sol1:
        print_solution(sol1, n1)
```

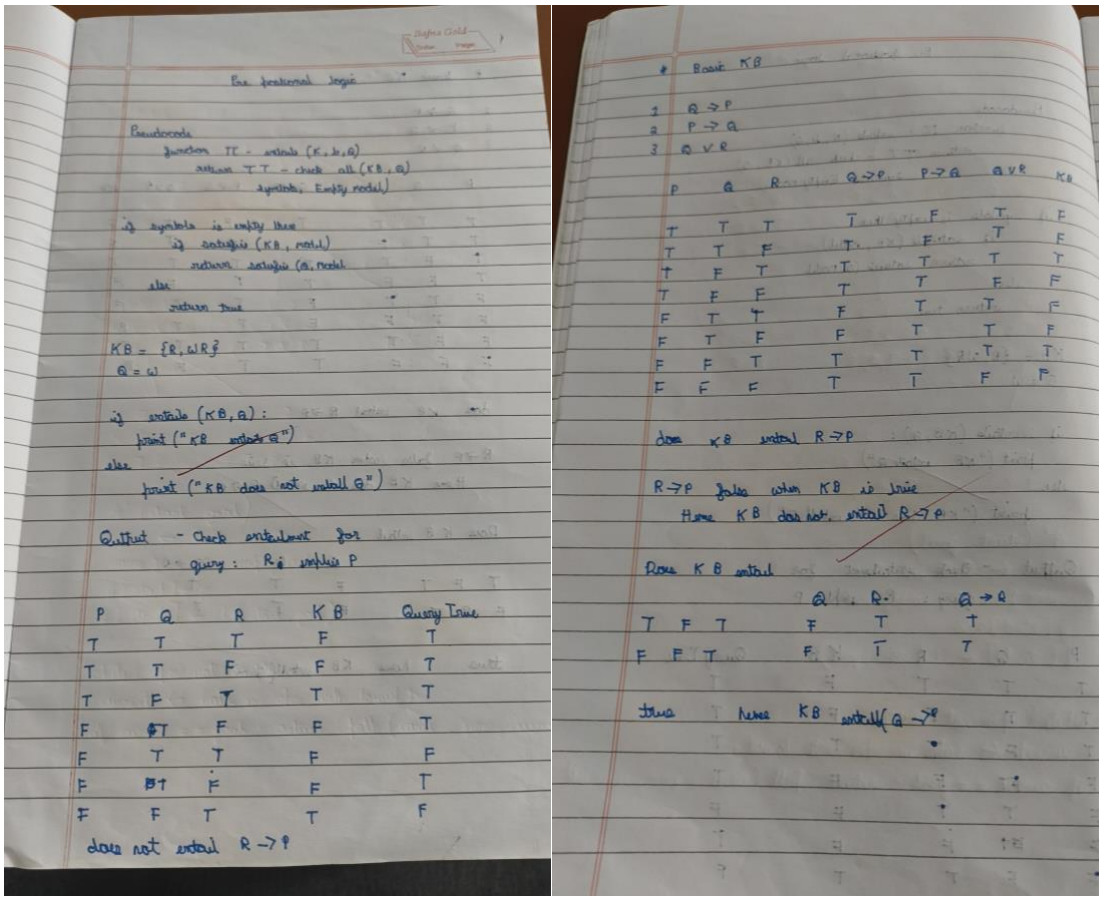
```
else:
    print("No solution.")

print("\nA* (Manhattan Distance)\n")
sol2, n2 = a_star(start, goal, manhattan_distance)
if sol2:
    print_solution(sol2, n2)
else:
    print("No solution.")
```

OUTPUT :

```
A* (Misplaced Tiles)

Steps:

Start | Depth 0
1 2 3
4   6
7 5 8

Move Down | Depth 1
1 2 3
4 5 6
7   8

Move Right | Depth 2
1 2 3
4 5 6
7 8

Total Moves: 2
Nodes Expanded: 3

A* (Manhattan Distance)

Steps:

Start | Depth 0
1 2 3
4   6
7 5 8

Move Down | Depth 1
1 2 3
4 5 6
7   8

Move Right | Depth 2
1 2 3
4 5 6
7 8

Total Moves: 2
Nodes Expanded: 3
```

Program 6: Propositional Logic

ALGORITHM :



CODE :

```
import itertools

# Logical operations
def implies(a, b):
    return not a or b

def or_operator(a, b):
    return a or b

def not_operator(a):
    return not a

# Constructing the truth table
def construct_truth_table():
```

```
    truth_values = [True, False]
    truth_table = []

    # Generate all combinations for Q, P, R
    for values in itertools.product(truth_values, repeat=3):
        Q, P, R = values

        # Evaluate KB sentences
        q_implies_p = implies(Q, P)
        p_implies_not_q = implies(P, not_operator(Q))
        q_or_r = or_operator(Q, R)

        # KB = (Q → P) ∧ (P → ¬Q) ∧ (Q ∨ R)
        kb_is_true = q_implies_p and p_implies_not_q and q_or_r

        # Entailment expressions
        entail_r = R
        entail_r_implies_p = implies(R, P)
        entail_q_implies_r = implies(Q, R)

        # Add row to truth table
        truth_table.append((
            Q, P, R,
            q_implies_p, p_implies_not_q, q_or_r,
            kb_is_true,
            entail_r, entail_r_implies_p, entail_q_implies_r
        ))
    return truth_table

# Print the truth table nicely
def print_truth_table(truth_table):
    header = [
        "Q", "P", "R",
        "Q -> P", "P -> ~Q", "Q v R",
        "KB (all true)",
        "R", "R -> P", "Q -> R"
    ]

    # Calculate the width for each column based on header length + padding
    # We ensure a minimum width of 3 for readability
    col_widths = [max(len(h) + 2, 3) for h in header]

    # Format and print the header row
    # {h:^{w}} centers the text 'h' within width 'w'
```

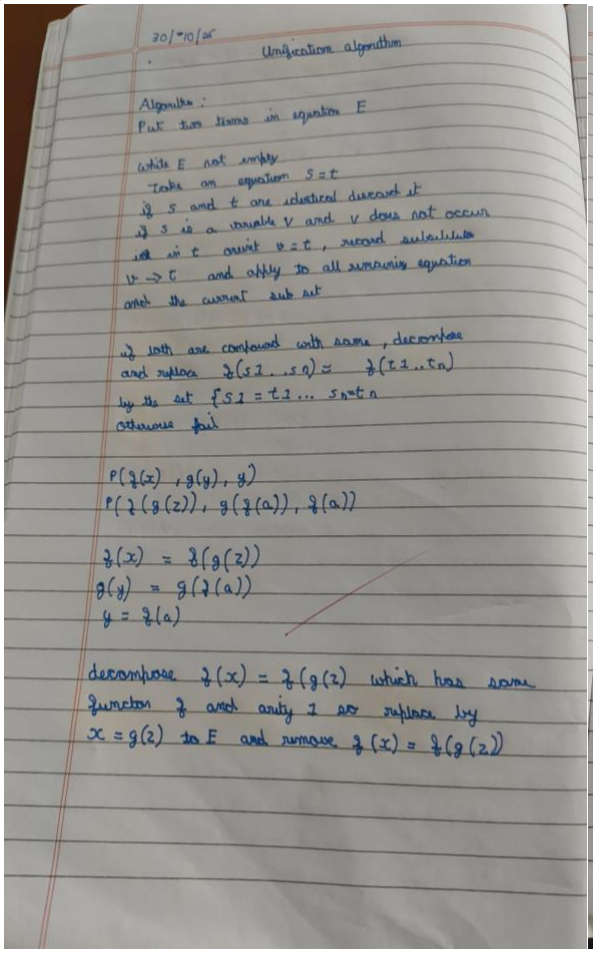
```
    header_row = " | ".join(f"{h:^{w}}" for h, w in zip(header, col_widths))
    print(header_row)
    print("-" * len(header_row))

    for row in truth_table:
        # Format True/False as T/F
        vals = ["T" if val else "F" for val in row]

        # Print the row using the same column widths calculated above
        formatted_row = " | ".join(f"{v:^{w}}" for v, w in zip(vals, col_widths))
        print(formatted_row)

# Generate and print truth table
truth_table = construct_truth_table()
print_truth_table(truth_table)

# Additionally, check entailment by verifying if for all models where KB is true, entailment is
true
def check_entailment(truth_table, entailment_index):
    for row in truth_table:
        kb_true = row[6]
        entailment_val = row[entailment_index]
        if kb_true and not entailment_val:
            return False
    return True

print("\nEntailment Results:")
print(f"Does KB entail R?      {check_entailment(truth_table, 7)}")
print(f"Does KB entail R -> P? {check_entailment(truth_table, 8)}")
print(f"Does KB entail Q -> R? {check_entailment(truth_table, 9)}")
```

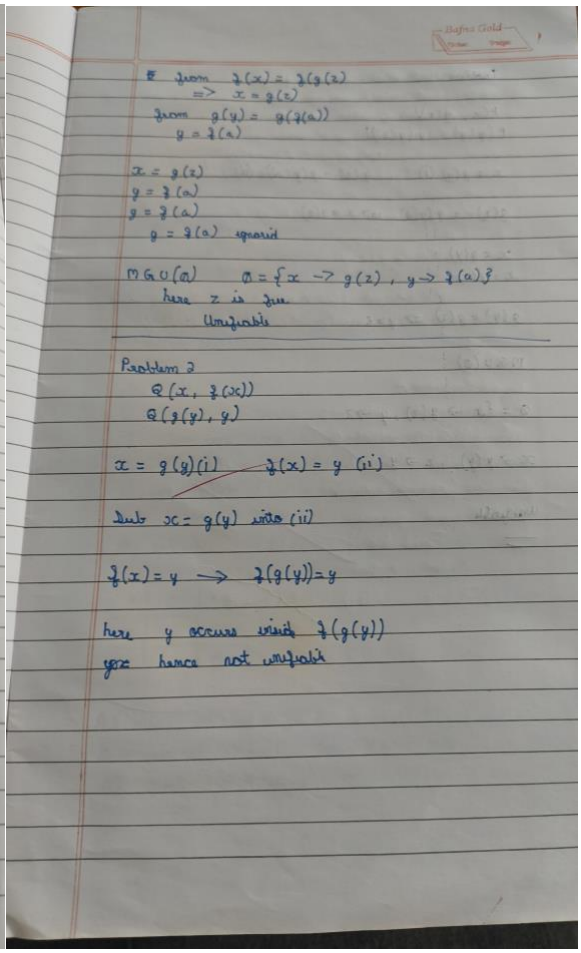
OUTPUT :

```
  Q   |   P   |   R   |  Q -> P  |  P -> ~Q  |  Q v R  |  KB (all true)  |   R   |  R -> P  |  Q -> R
-------------------------------------------------------------------------------------------------------
  T   |   T   |   T   |    T     |     F     |    T    |        F        |   T   |    T     |    T
  T   |   T   |   F   |    T     |     F     |    T    |        F        |   F   |    T     |    F
  T   |   F   |   T   |    F     |     T     |    T    |        F        |   T   |    F     |    T
  T   |   F   |   F   |    F     |     T     |    T    |        F        |   F   |    T     |    F
  F   |   T   |   T   |    T     |     T     |    T    |        T        |   T   |    T     |    T
  F   |   T   |   F   |    T     |     T     |    F    |        F        |   F   |    T     |    T
  F   |   F   |   T   |    T     |     T     |    T    |        T        |   T   |    F     |    T
  F   |   F   |   F   |    T     |     T     |    F    |        F        |   F   |    T     |    T

Entailment Results:
Does KB entail R?      True
Does KB entail R -> P? False
Does KB entail Q -> R? True
```

Program 7: Unification

Algorithm:





CODE :

```
class Variable:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return isinstance(other, Variable) and self.name == other.name

    def __hash__(self):
        return hash(self.name)

    def __repr__(self):
        return self.name

class Constant:
```

```
    def __init__(self, value):
        self.value = value

    def __eq__(self, other):
        return isinstance(other, Constant) and self.value == other.value

    def __hash__(self):
        return hash(self.value)

    def __repr__(self):
        return str(self.value)

class Function:
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __eq__(self, other):
        return (isinstance(other, Function) and
                self.name == other.name and
                len(self.args) == len(other.args) and
                all(a == b for a, b in zip(self.args, other.args)))

    def __hash__(self):
        return hash((self.name, tuple(self.args)))

    def __repr__(self):
        return f"{self.name}({', '.join(map(str, self.args))})"

def unify(term1, term2, substitution=None):
    """
    Unifies two first-order logic terms and returns the MGU (substitution)
    or None if unification is not possible.
    """
    if substitution is None:
        substitution = {}

    # Apply existing substitutions
    term1 = substitute(term1, substitution)
    term2 = substitute(term2, substitution)

    if term1 == term2:
        return substitution
    elif isinstance(term1, Variable):
```

```
        return unify_var(term1, term2, substitution)
    elif isinstance(term2, Variable):
        return unify_var(term2, term1, substitution)
    elif isinstance(term1, Function) and isinstance(term2, Function):
        if term1.name != term2.name or len(term1.args) != len(term2.args):
            return None  # Function symbols or arity don't match
        for arg1, arg2 in zip(term1.args, term2.args):
            substitution = unify(arg1, arg2, substitution)
            if substitution is None:
                return None  # Sub-unification failed
        return substitution
    else:
        return None  # Cannot unify different types (e.g., Constant and Function)

def unify_var(var, x, substitution):
    """Handles unification when one of the terms is a variable."""
    if var in substitution:
        return unify(substitution[var], x, substitution)
    elif x in substitution:
        return unify(var, substitution[x], substitution)
    elif occurs_check(var, x, substitution):
        return None  # Occurs check fails
    else:
        substitution[var] = x
        return substitution

def occurs_check(var, term, substitution):
    """Checks if a variable occurs within a term, preventing infinite substitutions."""
    term = substitute(term, substitution)  # Apply current substitutions
    if var == term:
        return True
    elif isinstance(term, Function):
        return any(occurs_check(var, arg, substitution) for arg in term.args)
    return False

def substitute(term, substitution):
    """Applies a given substitution to a term."""
    if isinstance(term, Variable):
        return substitution.get(term, term)
    elif isinstance(term, Function):
        return Function(term.name, [substitute(arg, substitution) for arg in term.args])
    return term

if __name__ == "__main__":
```

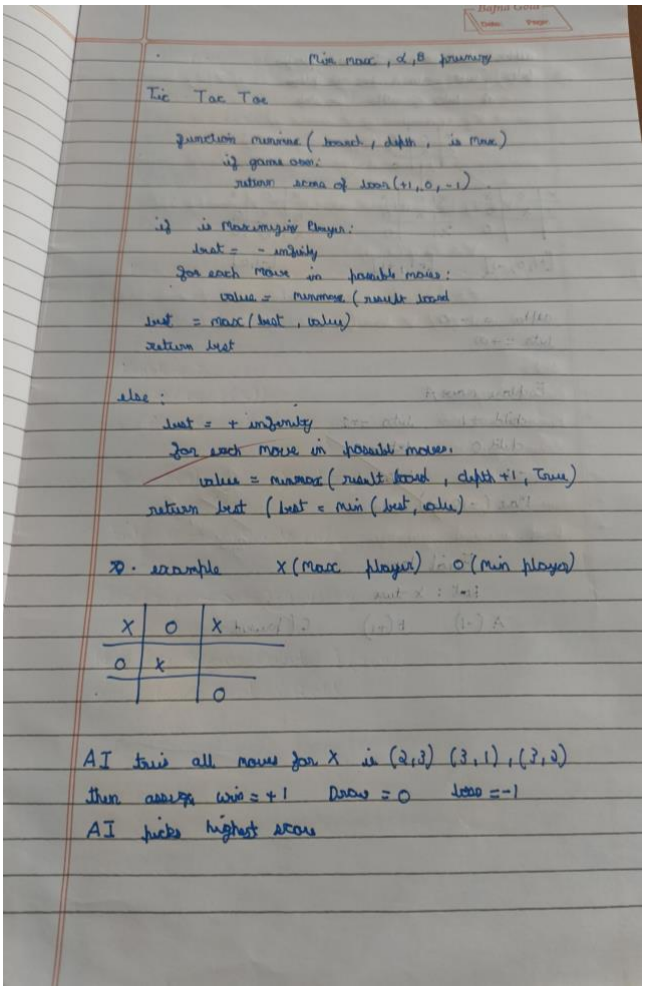
```
    # Define terms
    x, y, z = Variable('x'), Variable('y'), Variable('z')
    a, b = Constant('a'), Constant('b')
    f = Function('f', [x, Constant('b')])
    g = Function('g', [Constant('a'), y])
    h = Function('h', [z])

    print(f"Unify(f(x, b), f(a, y)): {unify(Function('f', [x, b]), Function('f', [a, y]))}")
    print(f"Unify(g(a, y), g(a, b)): {unify(Function('g', [a, y]), Function('g', [a, b]))}")
    print(f"Unify(x, f(x, b)): {unify(x, Function('f', [x, b]))}") # Occurs check failure
    print(f"Unify(f(x, y), f(a, g(z))): {unify(Function('f', [x, y]), Function('f', [a, Function('g',
[z])]))}")
    print(f"Unify(P(x, A), P(B, y)): {unify(Function('P', [x, Constant('A')]), Function('P',
[Constant('B'), y]))}")
    print("\n--- Requested Tests ---")

    # 1. p(f(x), g(y), y) and p(f(g(z)), g(f(a)), f(a))
    term1_1 = Function('p', [Function('f', [x]), Function('g', [y]), y])
    term1_2 = Function('p', [Function('f', [Function('g', [z])]), Function('g', [Function('f', [a])]),
Function('f', [a])])
    print(f"Unify(p(f(x), g(y), y), p(f(g(z)), g(f(a)), f(a))): {unify(term1_1, term1_2)}")

    # 2. q(x, f(x)) and q(f(y), y)
    term2_1 = Function('q', [x, Function('f', [x])])
    term2_2 = Function('q', [Function('f', [y]), y])
    print(f"Unify(q(x, f(x)), q(f(y), y)): {unify(term2_1, term2_2)}")

    # 3. p(x, g(x)) and p(g(y), g(g(z)))
    term3_1 = Function('p', [x, Function('g', [x])])
    term3_2 = Function('p', [Function('g', [y]), Function('g', [Function('g', [z])])])
    print(f"Unify(p(x, g(x)), p(g(y), g(g(z)))): {unify(term3_1, term3_2)}")
```

OUTPUT :

```
Unify(f(x, b), f(a, y)): {x: a, y: b}
Unify(g(a, y), g(a, b)): {y: b}
Unify(x, f(x, b)): None
Unify(f(x, y), f(a, g(z))): {x: a, y: g(z)}
Unify(P(x, A), P(B, y)): {x: B, y: A}

--- Requested Tests ---
Unify(p(f(x), g(y), y), p(f(g(z)), g(f(a)), f(a))): {x: g(z), y: f(a)}
Unify(q(x, f(x)), q(f(y), y)): None
Unify(p(x, g(x)), p(g(y), g(g(z)))): {x: g(y), y: z}
```

Program 8: MinMax & AlphaBeta

ALGORITHM :



CODE :

```
def alpha_beta(node, depth, alpha, beta, maximizing_player, path):
    # Base case: if node is a leaf (integer), return its value and path
    if isinstance(node, int):
        return node, path

    if maximizing_player:
        value = float('-inf')
        best_path = None
```

```
        for i, child in enumerate(node):
            child_value, child_path = alpha_beta(
                child, depth + 1, alpha, beta, False, path + [i]
            )

            if child_value > value:
                value = child_value
                best_path = child_path

            # Artifact '62' removed here
            alpha = max(alpha, value)

            if alpha >= beta:
                print(f"  [PRUNE] MAX (Depth {depth}): Alpha ({alpha}) >= Beta ({beta})")
                break

        return value, best_path

    else:
        value = float('inf')
        best_path = None

        for i, child in enumerate(node):
            child_value, child_path = alpha_beta(
                child, depth + 1, alpha, beta, True, path + [i]
            )

            if child_value < value:
                value = child_value
                best_path = child_path

            beta = min(beta, value)

            if beta <= alpha:
                print(f"  [PRUNE] MIN (Depth {depth}): Beta ({beta}) <= Alpha ({alpha})")
                break

        return value, best_path

if __name__ == "__main__":
    # Tree structure with artifact '63' removed
    tree = [
        [
            [
```

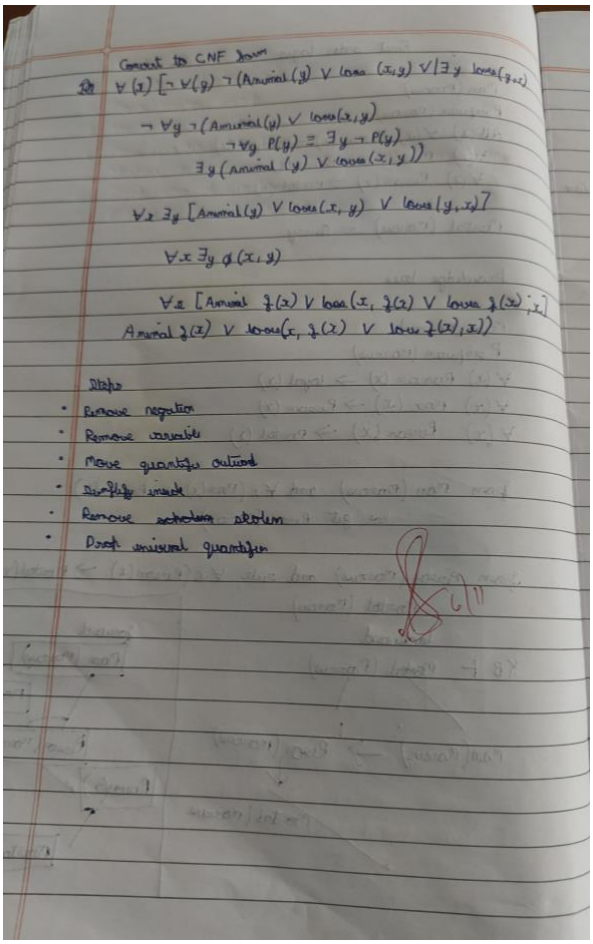
```
            [10, 11],
            [9, 12]
        ],
        [
            [14, 15],
            [13, 14]
        ],
    ],
    [
        [
            [5, 2],
            [4, 1]
        ],
        [
            [3, 22],
            [20, 21]
        ],
    ]
]

print("Starting Alpha-Beta Pruning...\n" + "-"*30)

value, best_path = alpha_beta(tree, 0, float('-inf'), float('inf'), True, [])

print("-" * 30)
print(f"FINAL MINIMAX VALUE AT ROOT: {value}")
print(f"BEST PATH INDICES: {best_path}")
```

OUTPUT :

```
Starting Alpha-Beta Pruning...
------------------------------
  [PRUNE] MIN (Depth 3): Beta (9) <= Alpha (10)
  [PRUNE] MAX (Depth 2): Alpha (14) >= Beta (10)
  [PRUNE] MIN (Depth 3): Beta (5) <= Alpha (10)
  [PRUNE] MIN (Depth 3): Beta (4) <= Alpha (10)
  [PRUNE] MIN (Depth 1): Beta (5) <= Alpha (10)
------------------------------
FINAL MINIMAX VALUE AT ROOT: 10
BEST PATH INDICES: [0, 0, 0, 0]
```

Program 9: Forward Chaining & Conversion to CNF

ALGORITHM :



CODE (Forward Chaining) :

```
import re

def isVariable(x):
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    expr = r'\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    expr = r'([a-zA-Z~]+)\([^&|]+\)'
    return re.findall(expr, string)
```

```
class Fact:
    def __init__(self, expression):
        self.expression = expression.strip()
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params

    def splitExpression(self, expression):
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, [p.strip() for p in params]]

    def substitute(self, var_map):
        params = [var_map.get(p, p) for p in self.params]
        return Fact(f"{self.predicate}({','.join(params)})")

    def __repr__(self):
        return self.expression

class Implication:
    def __init__(self, expression):
        self.expression = expression.strip()
        lhs, rhs = expression.split('=>')
        self.lhs = [Fact(f.strip()) for f in lhs.split('&')]
        self.rhs = Fact(rhs.strip())

    def infer(self, known_facts):
        substitutions = {}

        for fact in self.lhs:
            matched = False
            for known in known_facts:
                if known.predicate == fact.predicate:
                    mapping = {}
                    for i, param in enumerate(fact.params):
                        if isVariable(param):
                            mapping[param] = known.params[i]
                        elif param != known.params[i]:
                            break
                    else:
                        substitutions.update(mapping)
                        matched = True
                        break
```

```
            if not matched:
                return None

        return self.rhs.substitute(substitutions)

class KB:
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, expr):
        if '=>' in expr:
            self.implications.add(Implication(expr))
        else:
            self.facts.add(Fact(expr))

    def infer_all(self):
        added = True
        while added:
            added = False
            for rule in self.implications:
                new_fact = rule.infer(self.facts)
                if new_fact and new_fact.expression not in [f.expression for f in self.facts]:
                    print(f"Derived: {new_fact.expression}")
                    self.facts.add(new_fact)
                    added = True

    def ask(self, query):
        print(f"\nQuerying {query}:")
        self.infer_all()
        facts = [f.expression for f in self.facts]
        if query in facts:
            print(f"Yes, {query.split('(')[1].strip(')')} is {query.split('(')[0]}.")
        else:
            print(f"No, cannot infer {query}.")

    def display(self):
        print("\nAll facts in Knowledge Base:")
        for i, f in enumerate(sorted([f.expression for f in self.facts])):
            print(f"\t{i+1}. {f}")

def main():
    kb = KB()
    n = int(input("Enter number of FOL expressions: "))
```

```
    print("Enter expressions:")
    for _ in range(n):
        kb.tell(input().strip())

    query = input("Enter query: ").strip()
    kb.ask(query)
    kb.display()

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

OUTPUT (Forward Chaining) :

```
Enter number of FOL expressions: 6
Enter expressions:
Man(Marcus)
Pompeian(Marcus)
Pompeian(x) => Roman(x)
Roman(x) => Loyal(x)
Man(x) => Person(x)
Person(x) => Mortal(x)
Enter query: Mortal(Marcus)

Querying Mortal(Marcus):
Derived: Person(Marcus)
Derived: Roman(Marcus)
Derived: Mortal(Marcus)
Derived: Loyal(Marcus)
Yes, Marcus is Mortal.

All facts in Knowledge Base:
    1. Loyal(Marcus)
    2. Man(Marcus)
    3. Mortal(Marcus)
    4. Person(Marcus)
    5. Pompeian(Marcus)
    6. Roman(Marcus)
```

CODE (CNF) :

```
import re
def getAttributes(string):
    expr = r'\([^)]+\)'
    matches = re.findall(expr, string)
    return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):
    expr = r'[A-Za-z~]+\([A-Za-z,]+\)'
    return re.findall(expr, string)

def DeMorgan(sentence):
    string = ''.join(list(sentence).copy())
    string = string.replace('~~', '')
    flag = '[' in string
    string = string.replace('~[', '')
    string = string.strip(']')

    for predicate in getPredicates(string):
        string = string.replace(predicate, f'~{predicate}')

    s = list(string)
    for i, c in enumerate(s):
        if c == 'V':
            s[i] = '^'
        elif c == '^':
            s[i] = 'V'

    string = ''.join(s)
    string = string.replace('~~', '')
    return f'[{string}]' if flag else string
```

```
def Skolemization(sentence):
    SKOLEM_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]
    statement = ''.join(list(sentence).copy())
     matches = re.findall('[∀∃].', statement)

    for match in matches[::-1]:
        statement = statement.replace(match, '')

    statements = re.findall(r'\[\[^]]+\]\]', statement)
    for s in statements:
        statement = statement.replace(s, s[1:-1])

    for predicate in getPredicates(statement):
        attributes = getAttributes(predicate)
        if ''.join(attributes).islower():
            statement = statement.replace(predicate, predicate)
        else:
            aL = [a for a in attributes if a.islower()]
            aU = [a for a in attributes if not a.islower()][0] if attributes else ''
            if aU:
                statement = statement.replace(aU, f'{SKOLEM_CONSTANTS.pop(0)}({aL[0] if
len(aL) else match[1]})')
    return statement

def clean_output(expr):
    # Remove multiple brackets and redundant negations
    expr = expr.replace('~~', '')
    while '[[' in expr or ']]' in expr:
        expr = expr.replace('[[', '[').replace(']]', ']')
```

```
    expr = expr.strip('[] ')

    # Remove redundant outer brackets like [(p | q)] -> p | q
    if expr.startswith('(') and expr.endswith(')'):
        expr = expr[1:-1]

    # Replace internal redundant patterns
    expr = re.sub(r'\s+', ' ', expr)
    return expr

def fol_to_cnf(fol):
    statement = fol.replace("<=>", "_")
    while '_' in statement:
        i = statement.index('_')
        new_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^[' + statement[i+1:] + '=>' +
statement[:i] + ']'
        statement = new_statement

    statement = statement.replace("=>", "-")

    expr = r'\[([^]]+)\]'
    statements = re.findall(expr, statement)
    for i, s in enumerate(statements):
        if '[' in s and ']' not in s:
            statements[i] += ']'
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))

    while '-' in statement:
        i = statement.index('-')
        br = statement.index('[') if '[' in statement else 0
        new_statement = '~' + statement[br:i] + 'V' + statement[i+1:]
```

```
        statement = statement[:br] + new_statement if br > 0 else new_statement

    while '~∀' in statement:
        i = statement.index('~∀')
        statement = list(statement)
        statement[i], statement[i+1], statement[i+2] = '∃',
statement[i+2], '~'
        statement = ''.join(statement)

    while '~∃' in statement:
        i = statement.index('~∃')
        s = list(statement)
        s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'
        statement = ''.join(s)

    statement = statement.replace('~[∀', '[~∀')
    statement = statement.replace('~[∃', '[~∃')

    expr = r'(~[∀V∃].)'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, fol_to_cnf(s))

    expr = r'~\[[^]]+\]'
    statements = re.findall(expr, statement)
    for s in statements:
        statement = statement.replace(s, DeMorgan(s))

    return statement
```

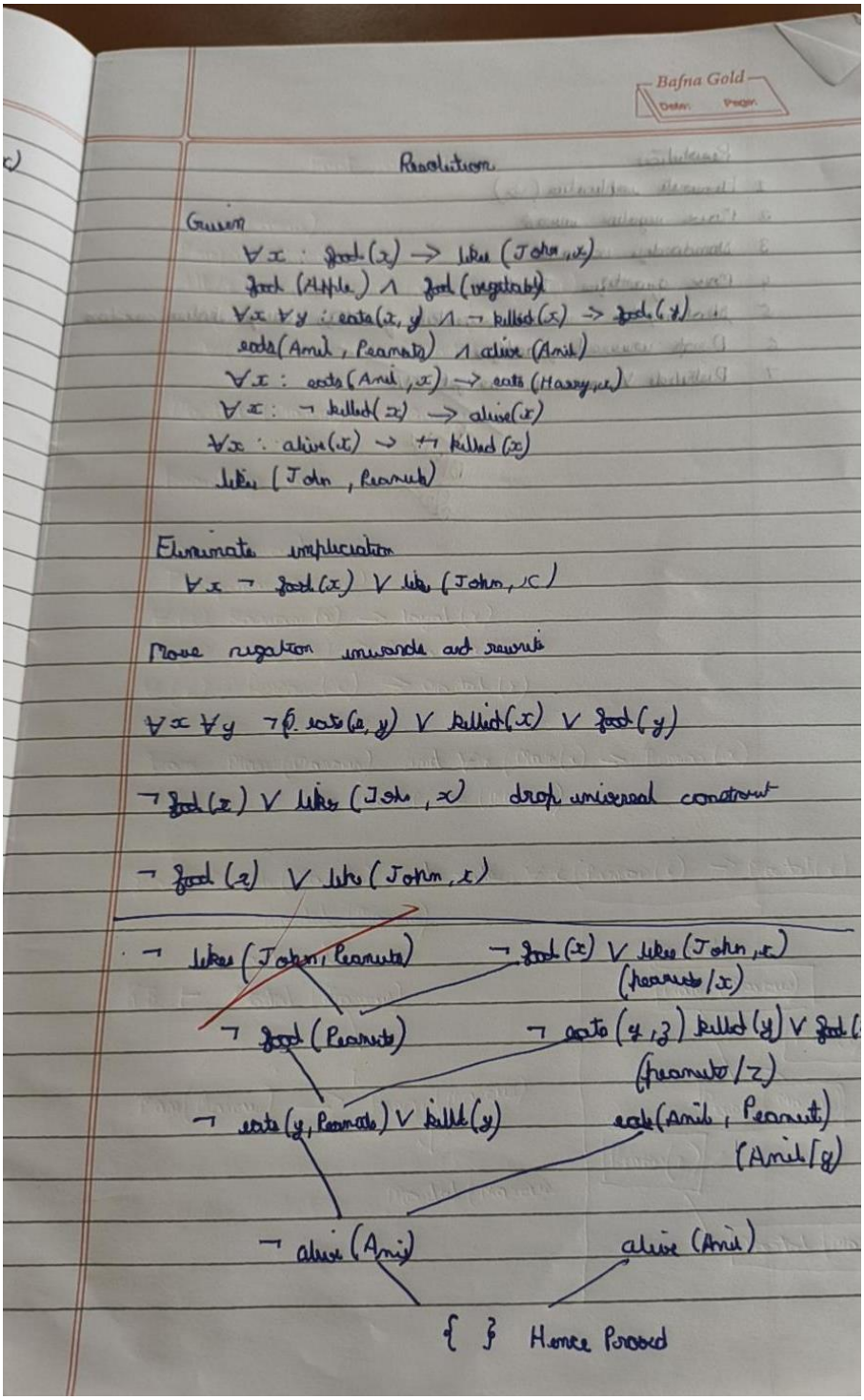
```
def main():
    print("\n" + "="*50)
    print("    FOL to CNF Converter (Simplified Output)")
    print("="*50)
     print("Supports: ∀, ∃, ~, &, |, >>, <=>, brackets [] () {}")
    print("NOTE: Use 'V' for OR inside the formula if needed.")
    print("-" * 50)
    fol = input("Enter FOL formula: ").strip()
        print("-" * 50)
    try:
        raw_cnf = fol_to_cnf(fol)
        result = Skolemization(raw_cnf)
        cleaned = clean_output(result)
        print(f"Original:    {fol}")
        print(f"CNF Form:    {cleaned}")
    except Exception as e:
        print("\nError: Could not parse the formula.")
        print("Details:", e)
    print("="*50 + "\n")
if __name__ == "__main__":
    main()
```

OUTPUT (CNF);

```
==================================================
   FOL to CNF Converter (Simplified Output)
==================================================
Supports: ∀, ∃, ~, &, |, >>, <=>, brackets [] () {}
NOTE: Use 'V' for OR inside the formula if needed.
--------------------------------------------------
Enter FOL formula: ∀x[~∀y~(Animal(y)VLoves(x,y))]V[∃y Loves(y,x)]
--------------------------------------------------
Original:   ∀x[~∀y~(Animal(y)VLoves(x,y))]V[∃y Loves(y,x)]
CNF Form:   Animal(y)VLoves(x,y))]V[ Loves(y,x
==================================================
```

Program 10: Resolution

ALGORITHM :



CODE :

```
from sympy import symbols
from sympy.logic.boolalg import Implies, And, Or, Not, to_cnf
from sympy.logic.inference import satisfiable

# Define symbols
Food = symbols('Food')
Apple = symbols('Apple')
Vegetables = symbols('Vegetables')
Peanuts = symbols('Peanuts')
John_likes_x = symbols('John_likes_x')
Anil_eats_x = symbols('Anil_eats_x')
Harry_eats_x = symbols('Harry_eats_x')
Alive_x = symbols('Alive_x')
Killed_x = symbols('Killed_x')

# Knowledge Base in propositional logic
# a. John likes all kind of food -> For each food x, John_likes_x if Food(x)
kb = And(
    Implies(Food, John_likes_x),  # a
    Implies(Or(Apple, Vegetables), Food),  # b
    Implies(And(Anil_eats_x, Not(Killed_x)), Food),  # c
    And(Anil_eats_x, Alive_x),  # d
    Implies(Anil_eats_x, Harry_eats_x),  # e
    Implies(Alive_x, Not(Killed_x)),  # f
    Implies(Not(Killed_x), Alive_x)  # g
)

# We want to prove: John likes peanuts -> John_likes_x for Peanuts
# Assume the negation of the goal for resolution
goal_negation = Not(John_likes_x)

# Combine KB with negated goal
combined = And(kb, goal_negation)

# Check satisfiability
sat_result = satisfiable(combined)

if sat_result:
    print("The combined knowledge base and the negation of the goal is satisfiable.")
    print("This means that there is a scenario where the KB is true and John does NOT like
peanuts.")
```

```
    print("Therefore, we cannot prove that John likes peanuts from the given knowledge base using resolution.")
else:
    print("The combined knowledge base and the negation of the goal is unsatisfiable.")
    print("This means that there is no scenario where the KB is true and John does NOT like peanuts simultaneously.")
    print("By the principle of resolution (reductio ad absurdum), this implies that John likes peanuts MUST be true given the knowledge base.")
    print("Therefore, John likes peanuts is proven!")
```

OUTPUT :

```
The combined knowledge base and the negation of the goal is unsatisfiable.
This means that there is no scenario where the KB is true and John does NOT like peanuts simultaneously.
By the principle of resolution (reductio ad absurdum), this implies that John likes peanuts MUST be true given the knowledge base.
Therefore, John likes peanuts is proven!
```