VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

ANU SAI SHREE R (1BM23CS045)

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 **ANU SAI SHREE R(1BM23CS045),** 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.

Dr. K. R. Mamatha Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/Anu1BM23CS045/AI-LAB

Certificates of Infosys Springboard course

Infosys
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Anu Sai Shree R

for successfully completing

Artificial Intelligence Foundation Certification

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Anu Sai Shree R

for successfully completing the course

Introduction to Artificial Intelligence

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Anu Sai Shree R

for successfully completing the course

Introduction to Deep Learning

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Anu Sai Shree R

for successfully completing the course

Introduction to Natural Language Processing

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

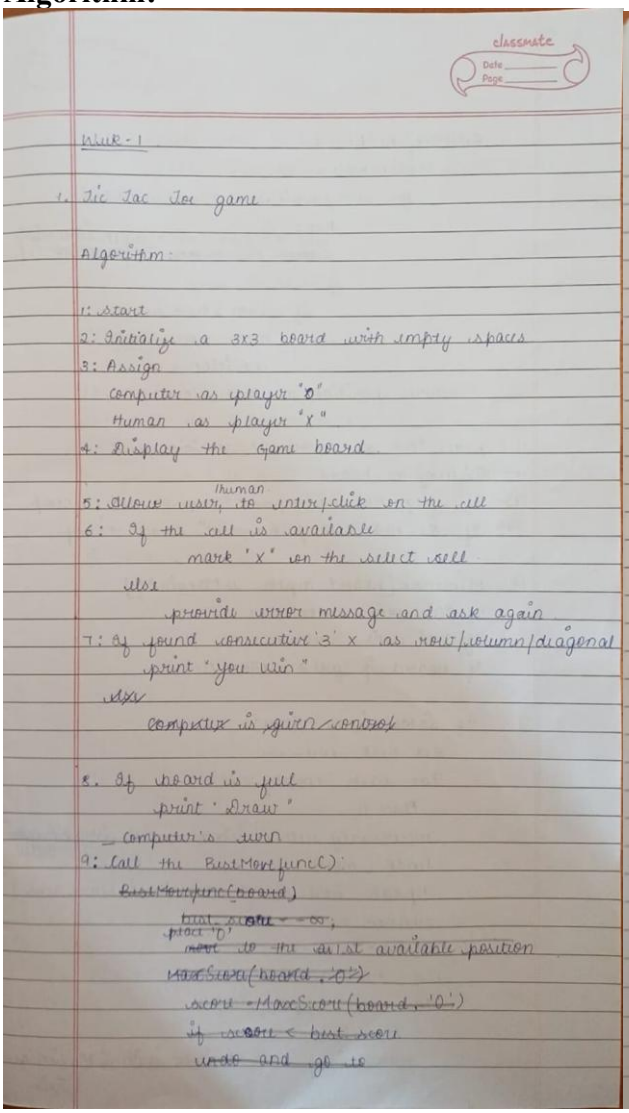
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Program 1

1.1 Implement Tic –Tac –Toe Game

Algorithm:

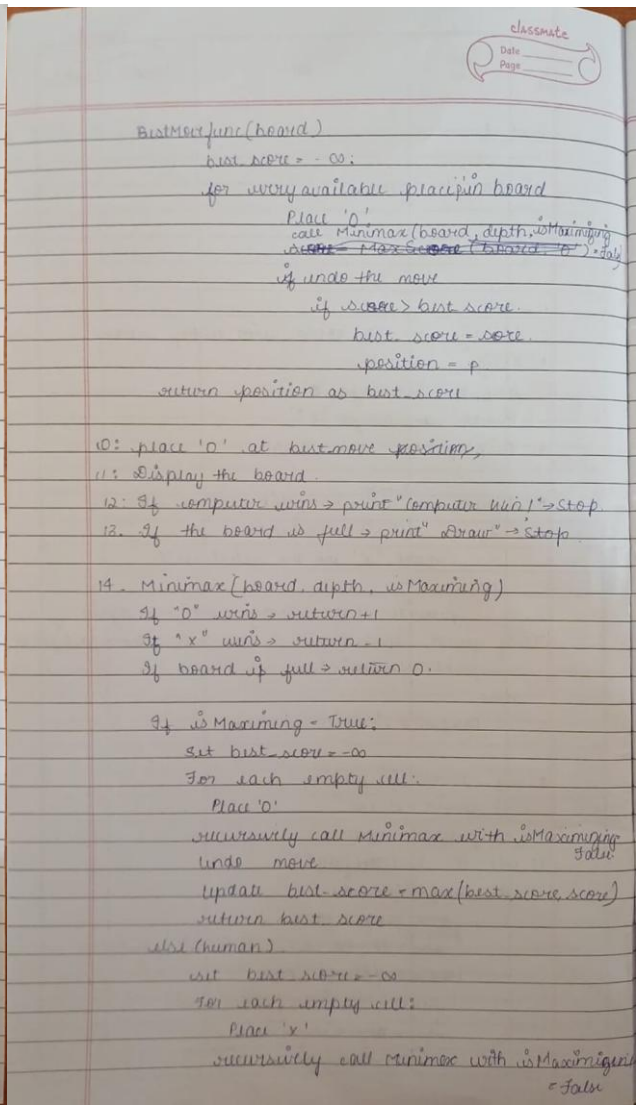


Week - 1

1. Tic Tac Toe game

Algorithm:

1: start
2: Initialize a 3x3 board with empty spaces
3: Assign
computer as player "O"
Human as player "X"
4: Display the game board
5: Allow user (human) to enter/click on the cell
6: If the cell is available
mark 'X' on the select cell
else
provide error message and ask again
7: If found consecutive 3 'x' as row/column/diagonal
print "you win"
else
computer is given/control
8. If board is full
print 'Draw'
computer's turn
9: Call the BestMove func()
BestMove(board)
best_score = -∞,
place 'O'
move to the earliest available position
MaxScore(board, 'O')
score = MaxScore(board, 'O')
if score < best_score
undo and go to

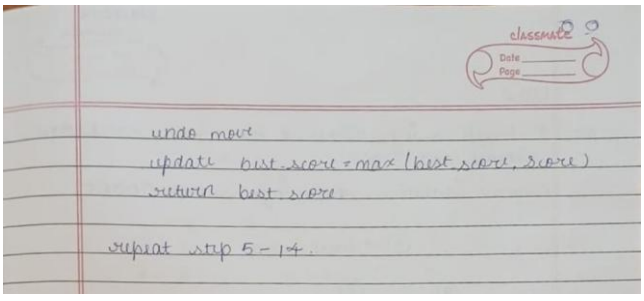


BestMove(board)
best_score = -∞;
for every available place in board
Place 'O'
call Minimax(board, depth, isMaximizing
score = MaxScore(board, 'O')
if undo the move
if score > best_score
best_score = score
position = p
return position as best_score

10: place 'O' at best_move position
11: Display the board
12: If computer wins → print "Computer Win!" → stop
13. If the board is full → print "Draw" → stop

14. Minimax(board, depth, isMaximizing)
If "O" wins → return +1
If "x" wins → return -1
If board is full → return 0.

If isMaximizing = True:
Set best_score = -∞
For each empty cell:
Place 'O'
recursively call Minimax with isMaximizing = False
undo move
update best_score = max(best_score, score)
return best_score
else (human)
set best_score = ∞
for each empty cell:
Place 'x'
recursively call Minimax with isMaximizing = False



undo move
update best_score = max(best_score, score)
return best_score

repeat step 5 - 14.

Code:

```
import math

def print_board(board):
    """Display the Tic Tac Toe board"""
    for row in board:
        print(" | ".join(row))
        print("-" * 5)


def check_winner(board, player):
    """Check all winning conditions for a player"""
    # 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 all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True

    return False


def is_full(board):
    """Check if the board is full"""
    return all(cell != " " for row in board for cell in row)


def minimax(board, depth, is_maximizing):
    """Minimax algorithm for optimal AI moves"""
    if check_winner(board, "O"):  # AI wins
        return 1
    if check_winner(board, "X"):  # Human wins
        return -1
    if is_full(board):  # Draw
```

```
        return 0

    if is_maximizing:  # AI's turn
        best_score = -math.inf
        for r in range(3):
            for c in range(3):
                if board[r][c] == " ":
                    board[r][c] = "O"
                    score = minimax(board, depth + 1, False)
                    board[r][c] = " "
                    best_score = max(score, best_score)
        return best_score
    else:  # Human's turn
        best_score = math.inf
        for r in range(3):
            for c in range(3):
                if board[r][c] == " ":
                    board[r][c] = "X"
                    score = minimax(board, depth + 1, True)
                    board[r][c] = " "
                    best_score = min(score, best_score)
        return best_score


def best_move(board):
    """Find the best move for AI using minimax"""
    best_score = -math.inf
    move = None
    for r in range(3):
        for c in range(3):
            if board[r][c] == " ":
                board[r][c] = "O"
                score = minimax(board, 0, False)
                board[r][c] = " "
                if score > best_score:
                    best_score = score
                    move = (r, c)
    return move


def tic_tac_toe():
    """Main game loop"""
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic Tac Toe!")
    print("You are X. Computer is O.")
```

```
    print_board(board)

    while True:
        # Human turn
        print("Your turn (X).")
        try:
            row = int(input("Enter row (1-3): ")) - 1
            col = int(input("Enter column (1-3): ")) - 1
        except ValueError:
            print("Invalid input! Enter numbers between 1 and 3.")
            continue

        if row not in range(3) or col not in range(3):
            print("Invalid move! Try again.")
            continue
        if board[row][col] != " ":
            print("Cell already taken! Try again.")
            continue

        board[row][col] = "X"
        print_board(board)

        if check_winner(board, "X"):
            print(" 🎉 You win!")
            break
        if is_full(board):
            print("It's a Draw!")
            break

        # Computer turn
        print("Computer's turn (O)...")
        move = best_move(board)
        if move:
            board[move[0]][move[1]] = "O"
        print_board(board)

        if check_winner(board, "O"):
            print(" 💻 Computer wins!")
            break
        if is_full(board):
            print("It's a Draw!")
            break


# Run the game
```

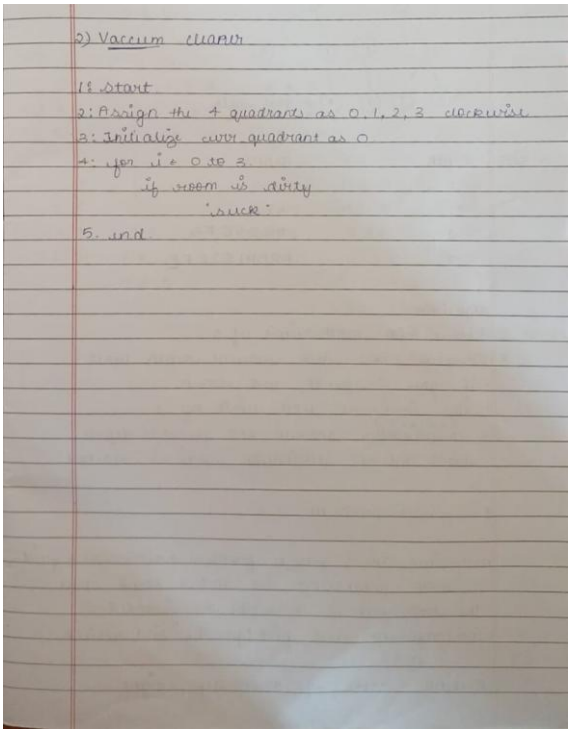
```
if __name__ == "__main__":
    tic_tac_toe()
```

Output :

```
Welcome to Tic Tac Toe!
You are X. Computer is O.
  |   |
-----
  |   |
-----
  |   |
-----
Your turn (X).
Enter row (1-3): 2
Enter column (1-3): 2
  |   |
-----
  | X |
-----
  |   |
-----
Computer's turn (O)...
O |   |
-----
  | X |
-----
  |   |
-----
Your turn (X).
Enter row (1-3): 1
Enter column (1-3): 3
O |   | X
-----
  | X |
-----
  |   |
-----
Computer's turn (O)...
O |   | X
-----
  | X |
-----
O |   |
-----
Your turn (X).
Enter row (1-3): 2
Enter column (1-3): 3
O |   | X
-----
  | X | X
-----
O |   |
-----
Computer's turn (O)...
O |   | X
-----
O | X | X
-----
O |   |
-----
💻 Computer wins!
```

1.2 Implement vacuum cleaner agent

Algorithm:



Code:

```
def vacuum_cleaner()
    A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    cost = 0
    state = {'A': A, 'B': B}

    if location == 'A':
        if state['A'] == 1:  # If A is dirty
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
        else:
            print("A is clean")

        if state['B'] == 1:  # If B is dirty
            print("Moving vacuum right")
```

```
            print("Cleaned B.")
            state['B'] = 0
            cost += 1
            print("Is B clean now? (0 if clean, 1 if dirty):", state['B'])
            print("Is A dirty? (0 if clean, 1 if dirty):", state['A'])
            print("B is clean")
            print("Moving vacuum left")
        else:
            print("Turning vacuum off")

    elif location == 'B':
        if state['B'] == 1:  # If B is dirty
            print("Cleaned B.")
            state['B'] = 0
            cost += 1
        else:
            print("B is clean")

        if state['A'] == 1:  # If A is dirty
            print("Moving vacuum left")
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
            print("Is A clean now? (0 if clean, 1 if dirty):", state['A'])
            print("Is B dirty? (0 if clean, 1 if dirty):", state['B'])
            print("A is clean")
            print("Moving vacuum right")
        else:
            print("Turning vacuum off")

    print("Cost:", cost)
    print(state)


vacuum_cleaner()
```

OUTPUT Case1:

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
Cleaned A.
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 2
{'A': 0, 'B': 0}
```

OUTPUT Case2:

```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A
A is clean
Moving vacuum right
Cleaned B.
Is B clean now? (0 if clean, 1 if dirty): 0
Is A dirty? (0 if clean, 1 if dirty): 0
B is clean
Moving vacuum left
Cost: 1
{'A': 0, 'B': 0}
```

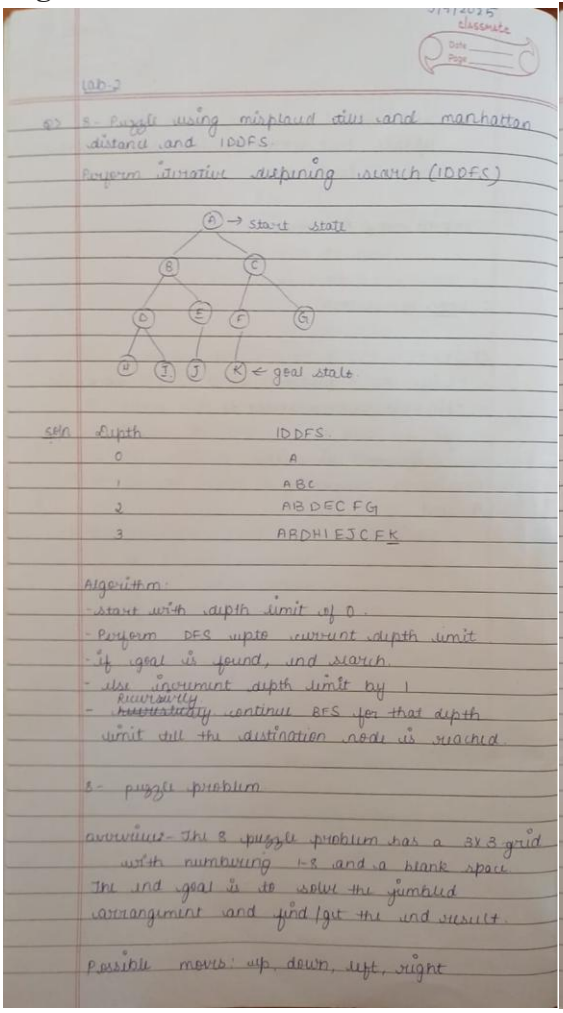
OUTPUT Case3:

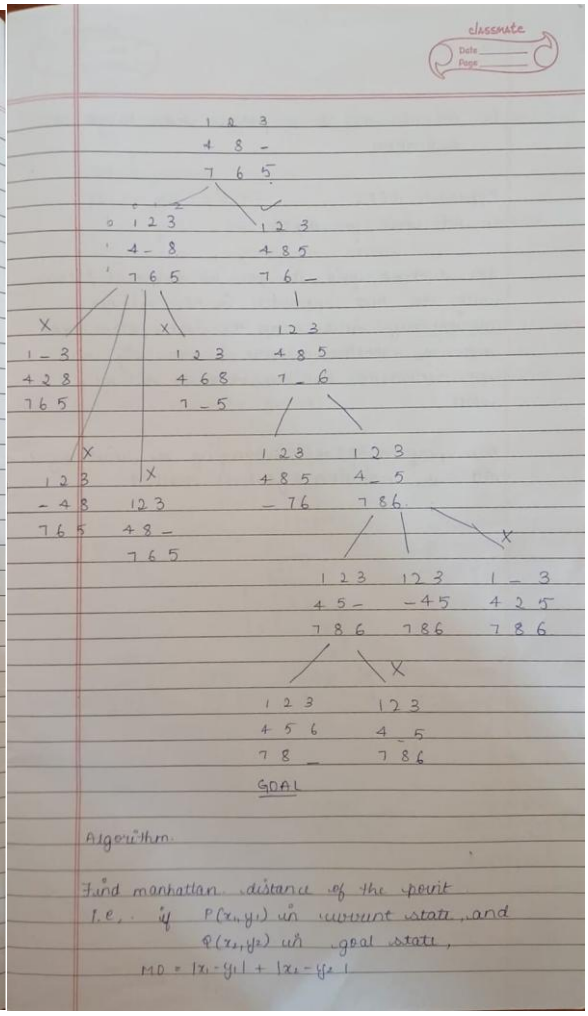
```
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
A is clean
Turning vacuum off
Cost: 0
{'A': 0, 'B': 0}
```

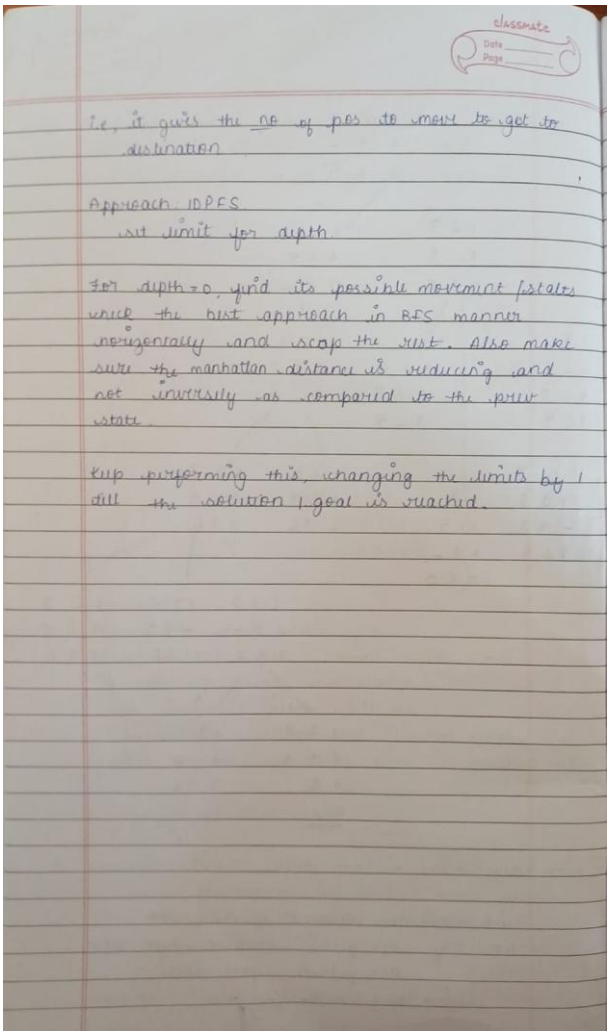
Program2

Implement 8 puzzle problems using Iterative Deeping depth- First Search (BFS)

Algorithm:







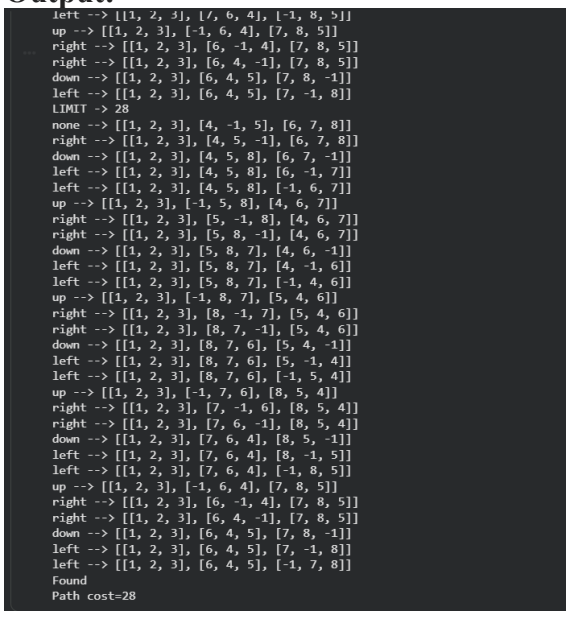
Code:

```
import copy

inp = [[1, 2, 3], [4, -1, 5], [6, 7, 8]]
out = [[1, 2, 3], [6, 4, 5], [-1, 7, 8]]

flag = False

def move(temp, movement):
    if movement == "up":
        for i in range(3):
            for j in range(3):
                if temp[i][j] == -1:
                    if i != 0:
                        temp[i][j] = temp[i - 1][j]
                        temp[i - 1][j] = -1
                    return temp

    if movement == "down":
```

```
        for i in range(3):
            for j in range(3):
                if temp[i][j] == -1:
                    if i != 2:
                        temp[i][j] = temp[i + 1][j]
                        temp[i + 1][j] = -1
                    return temp

    if movement == "left":
        for i in range(3):
            for j in range(3):
                if temp[i][j] == -1:
                    if j != 0:
                        temp[i][j] = temp[i][j - 1]
                        temp[i][j - 1] = -1
                    return temp

    if movement == "right":
        for i in range(3):
            for j in range(3):
                if temp[i][j] == -1:
                    if j != 2:
                        temp[i][j] = temp[i][j + 1]
                        temp[i][j + 1] = -1
                    return temp

def ids():
    global inp, out, flag

    for limit in range(100):
        print('LIMIT -> ' + str(limit))

        stack = []
        inpx = [inp, "none"]
        stack.append(inpx)
        level = 0

        while True:
            if len(stack) == 0:
                break

            puzzle = stack.pop(0)

            if level <= limit:
                print(str(puzzle[1]) + " --> " + str(puzzle[0]))

                if puzzle[0] == out:
                    print("Found")
                    print('Path cost=' + str(level))
                    flag = True
                    return
```

```
            else:
                level = level + 1

                if puzzle[1] != "down":
                    temp = copy.deepcopy(puzzle[0])
                    up = move(temp, "up")
                    if up != puzzle[0]:
                        stack.insert(0, [up, "up"])

                if puzzle[1] != "right":
                    temp = copy.deepcopy(puzzle[0])
                    left = move(temp, "left")
                    if left != puzzle[0]:
                        stack.insert(0, [left, "left"])

                if puzzle[1] != "up":
                    temp = copy.deepcopy(puzzle[0])
                    down = move(temp, "down")
                    if down != puzzle[0]:
                        stack.insert(0, [down, "down"])

                if puzzle[1] != "left":
                    temp = copy.deepcopy(puzzle[0])
                    right = move(temp, "right")
                    if right != puzzle[0]:
                        stack.insert(0, [right, "right"])

    print("Not Found")

print('~~~~~~~~~~~~~ IDS ~~~~~~~~~~~~~')
ids()
```

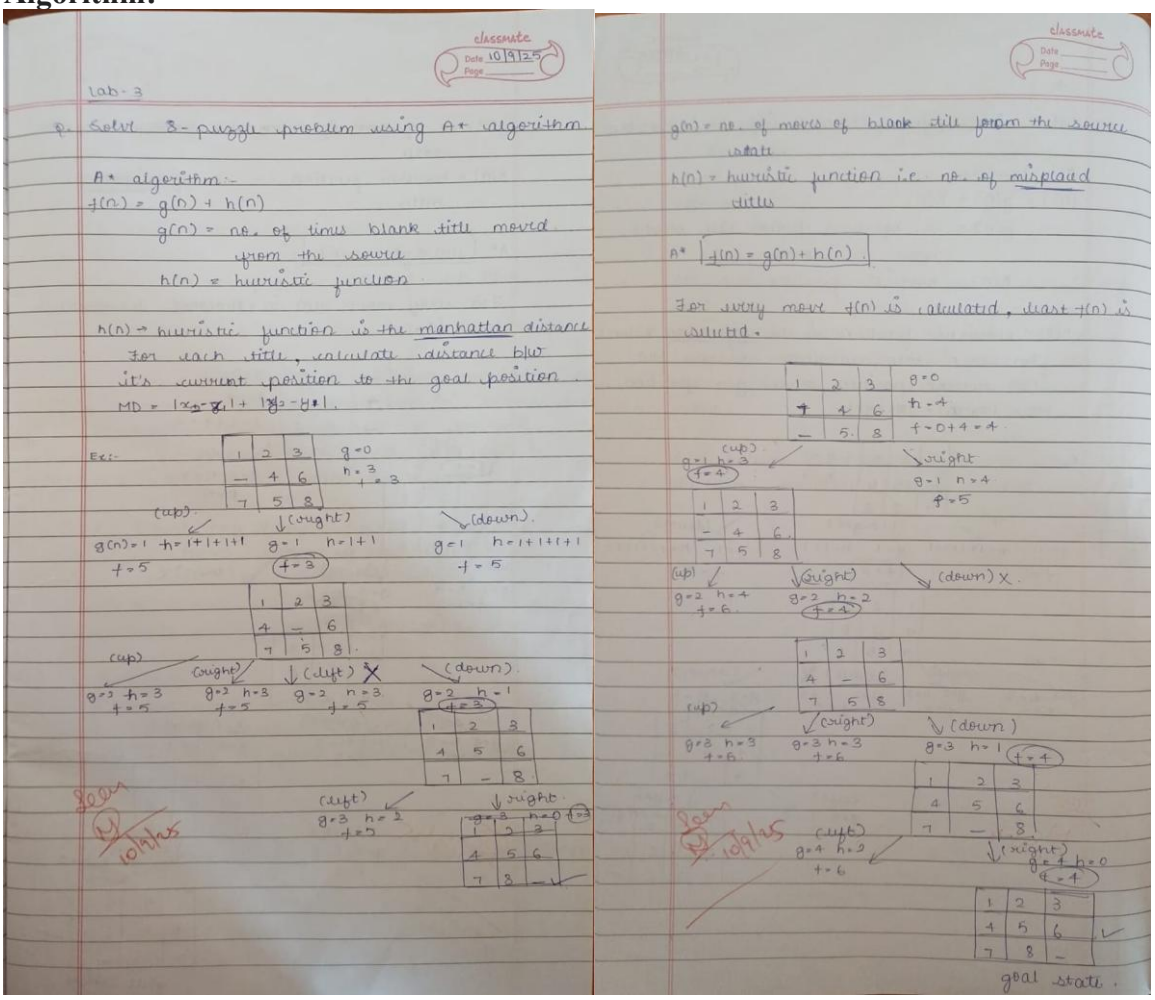
Output:



Program3

Implement A* search algorithm

Algorithm:



Code:

```
#MISPLACED TILE
import heapq
from itertools import count

def misplaced_heuristic(board, goal):
    """h(n): number of tiles not in their goal position (excluding blank 0)."""
    n = len(board)
    misplaced = 0
    for i in range(n):
        for j in range(n):
            if board[i][j] != 0 and board[i][j] != goal[i][j]:
                misplaced += 1
    return misplaced
```

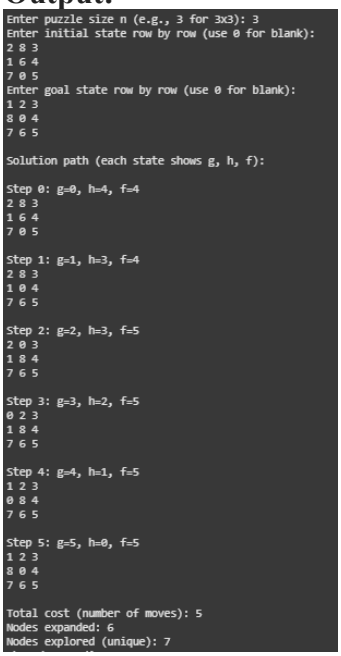
```
def find_blank(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return i, j
    raise ValueError("Board does not contain a blank tile (0)")

def neighbors(board):
    """Generate neighboring boards by sliding one tile into the blank."""
    n = len(board)
    x, y = find_blank(board)
    dirs = [(0,1),(0,-1),(1,0),(-1,0)]
    res = []
    for dx, dy in dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < n:
            b = [list(row) for row in board]
            b[x][y], b[nx][ny] = b[nx][ny], b[x][y]
            res.append(tuple(tuple(row) for row in b))
    return res

def flatten(board):
    return [x for row in board for x in row]

def inversion_count(seq):
    arr = [x for x in seq if x != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv

def blank_row_from_bottom(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return n - i
    raise ValueError("Board does not contain a blank tile (0)")

def is_solvable(start, goal):
    """General n-puzzle solvability test (odd/even width)."""
    n = len(start)
    start_flat = flatten(start)
    goal_flat = flatten(goal)

    pos = {val: idx for idx, val in enumerate(goal_flat)}
    start_perm = [pos[val] for val in start_flat]

    inv = inversion_count(start_perm)
```

```
    if n % 2 == 1:
        # odd grid: inversions parity must be even
        return inv % 2 == 0
    else:
        # even grid: blank row from bottom parity matters
        blank_row = blank_row_from_bottom(start)
        goal_blank_row = blank_row_from_bottom(goal)
        # When using relative permutation to goal, parity of blank rows must match
        return (inv + blank_row) % 2 == (0 + goal_blank_row) % 2

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

def a_star_misplaced(start, goal):
    start = tuple(tuple(row) for row in start)
    goal = tuple(tuple(row) for row in goal)

    if len(start) != len(start[0]) or len(goal) != len(goal[0]) or len(start) != len(goal):
        raise ValueError("Initial and goal must be square boards of the same size.")

    start_vals = sorted(flatten(start))
    goal_vals = sorted(flatten(goal))
    if start_vals != goal_vals:
        raise ValueError("Initial and goal must contain the same set of tiles.")

    if not is_solvable(start, goal):
        return None, None, 0, 0  # unsolvable

    counter = count()  # tie-breaker

    h0 = misplaced_heuristic(start, goal)
    g_score = {start: 0}
    f0 = h0

    open_heap = [(f0, next(counter), start)]
    open_set = {start: f0}
    closed = set()
    came_from = {}

    expansions = 0

    while open_heap:
        _, _, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        closed.add(current)
```

```
        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g_score[current], expansions, len(closed)

        expansions += 1

        for nb in neighbors(current):
            tentative_g = g_score[current] + 1
            if nb in closed:
                continue
            if nb not in g_score or tentative_g < g_score[nb]:
                came_from[nb] = current
                g_score[nb] = tentative_g
                h = misplaced_heuristic(nb, goal)
                f = tentative_g + h
                if nb not in open_set or f < open_set[nb]:
                    heapq.heappush(open_heap, (f, next(counter), nb))
                    open_set[nb] = f

    return None, None, expansions, len(closed)

def read_board(n, prompt):
    print(prompt)
    board = []
    for i in range(n):
        row = list(map(int, input().split()))
        if len(row) != n:
            raise ValueError(f"Row {i+1} must contain exactly {n} integers.")
        board.append(row)
    return board

def print_board(board):
    for row in board:
        print(" ".join(f"{x}" for x in row))

def main():
    try:
        n = int(input("Enter puzzle size n (e.g., 3 for 3x3): ").strip())
        initial = read_board(n, "Enter initial state row by row (use 0 for blank):")
        goal = read_board(n, "Enter goal state row by row (use 0 for blank):")

        result = a_star_misplaced(initial, goal)
        path, cost, expansions, explored = result

        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return

        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx  # each step costs 1
```

```
            h = misplaced_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f"Step {idx}: g={g}, h={h}, f={f}")
            print_board(state)
            print()

        print(f"Total cost (number of moves): {cost}")
        print(f"Nodes expanded: {expansions}")
        print(f"Nodes explored (unique): {explored}")
    except Exception as e:
        print("Error:", e)

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

Output:



Code:

```
#MANHATTAN DISTANCE
import heapq
from itertools import count

def misplaced_heuristic(board, goal):
    misplaced = 0
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] != 0 and board[i][j] != goal[i][j]:
                misplaced += 1
    return misplaced

def manhattan_heuristic(board, goal):
```

```
    n = len(board)
    # Map goal positions for each tile
    goal_pos = {}
    for i in range(n):
        for j in range(n):
            goal_pos[goal[i][j]] = (i, j)

    dist = 0
    for i in range(n):
        for j in range(n):
            val = board[i][j]
            if val != 0:
                gi, gj = goal_pos[val]
                dist += abs(i - gi) + abs(j - gj)
    return dist

def find_blank(board):
    n = len(board)
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return i, j
    raise ValueError("Board does not contain a blank tile (0)")

def neighbors(board):
    n = len(board)
    x, y = find_blank(board)
    dirs = [(0,1),(0,-1),(1,0),(-1,0)]
    res = []
    for dx, dy in dirs:
        nx, ny = x + dx, y + dy
        if 0 <= nx < n and 0 <= ny < n:
            b = [list(row) for row in board]
            b[x][y], b[nx][ny] = b[nx][ny], b[x][y]
            res.append(tuple(tuple(row) for row in b))
    return res

def flatten(board):
    return [x for row in board for x in row]

def inversion_count(seq):
    arr = [x for x in seq if x != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv

def blank_row_from_bottom(board):
    n = len(board)
    for i in range(n):
```

```
        for j in range(n):
            if board[i][j] == 0:
                return n - i
    raise ValueError("Board does not contain a blank tile (0)")

def is_solvable(start, goal):
    n = len(start)
    start_flat = flatten(start)
    goal_flat = flatten(goal)

    pos = {val: idx for idx, val in enumerate(goal_flat)}
    start_perm = [pos[val] for val in start_flat]

    inv = inversion_count(start_perm)

    if n % 2 == 1:
        return inv % 2 == 0
    else:
        blank_row = blank_row_from_bottom(start)
        goal_blank_row = blank_row_from_bottom(goal)
        return (inv + blank_row) % 2 == (0 + goal_blank_row) % 2

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

def a_star_manhattan(start, goal):
    start = tuple(tuple(row) for row in start)
    goal = tuple(tuple(row) for row in goal)

    if len(start) != len(start[0]) or len(goal) != len(goal[0]) or len(start) != len(goal):
        raise ValueError("Initial and goal must be square boards of the same size.")

    start_vals = sorted(flatten(start))
    goal_vals = sorted(flatten(goal))
    if start_vals != goal_vals:
        raise ValueError("Initial and goal must contain the same set of tiles.")

    if not is_solvable(start, goal):
        return None, None, 0, 0

    counter = count()
    h0 = manhattan_heuristic(start, goal)
    g_score = {start: 0}
    f0 = h0

    open_heap = [(f0, next(counter), start)]
    open_set = {start: f0}
```

```
    closed = set()
    came_from = {}

    expansions = 0

    while open_heap:
        _, _, current = heapq.heappop(open_heap)
        if current in closed:
            continue
        closed.add(current)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g_score[current], expansions, len(closed)

        expansions += 1

        for nb in neighbors(current):
            tentative_g = g_score[current] + 1
            if nb in closed:
                continue
            if nb not in g_score or tentative_g < g_score[nb]:
                came_from[nb] = current
                g_score[nb] = tentative_g
                h = manhattan_heuristic(nb, goal)
                f = tentative_g + h
                if nb not in open_set or f < open_set[nb]:
                    heapq.heappush(open_heap, (f, next(counter), nb))
                    open_set[nb] = f

    return None, None, expansions, len(closed)

def read_board(n, prompt):
    print(prompt)
    board = []
    for i in range(n):
        row = list(map(int, input().split()))
        if len(row) != n:
            raise ValueError(f"Row {i+1} must contain exactly {n} integers.")
        board.append(row)
    return board

def print_board(board):
    for row in board:
        print(" ".join(f"{x}" for x in row))

def main():
    try:
        n = int(input("Enter puzzle size n (e.g., 3 for 3x3): ").strip())
        initial = read_board(n, "Enter initial state row by row (use 0 for blank):")
        goal = read_board(n, "Enter goal state row by row (use 0 for blank):")
```

```
        result = a_star_manhattan(initial, goal)
        path, cost, expansions, explored = result

        if path is None:
            print("No solution (unsolvable with given start/goal).")
            return

        print("\nSolution path (each state shows g, h, f):\n")
        for idx, state in enumerate(path):
            g = idx
            h = manhattan_heuristic(state, tuple(tuple(r) for r in goal))
            f = g + h
            print(f"Step {idx}: g={g}, h={h}, f={f}")
            print_board(state)
            print()

        print(f"Total cost (number of moves): {cost}")
        print(f"Nodes expanded: {expansions}")
        print(f"Nodes explored (unique): {explored}")


    except Exception as e:
        print("Error:", e)

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

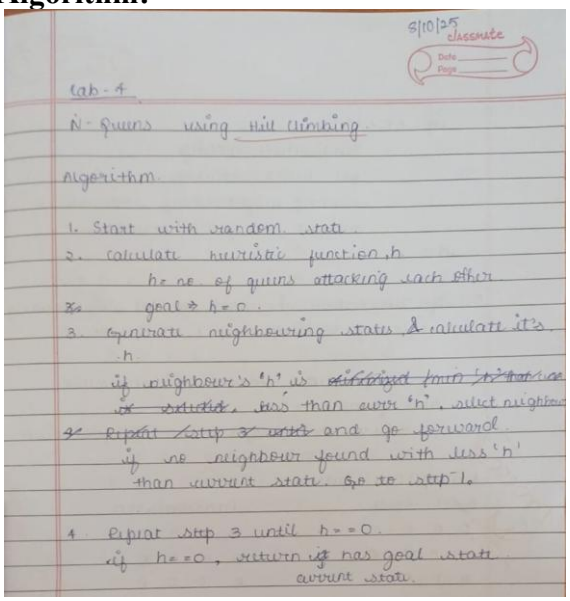
OUTPUT:

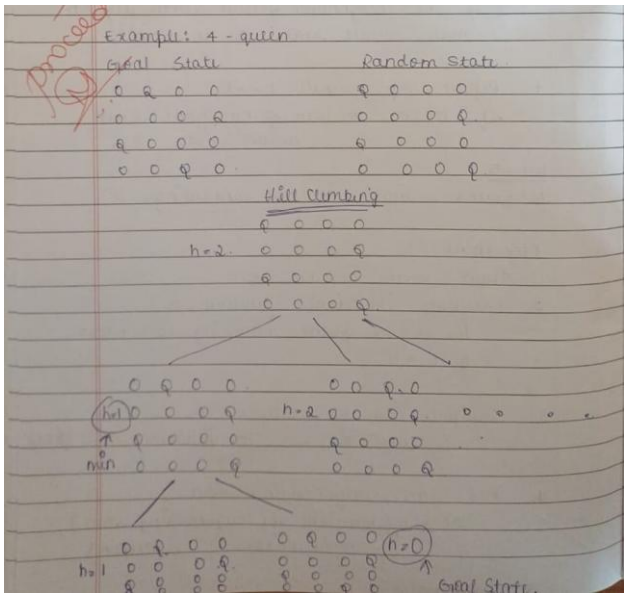
```
Enter puzzle size n (e.g., 3 for 3x3): 3
Enter initial state row by row (use 0 for blank):
2 8 3
1 6 4
7 0 5
Enter goal state row by row (use 0 for blank):
1 2 3
8 0 4
7 6 5

Solution path (each state shows g, h, f):

Step 0: g=0, h=5, f=5
2 8 3
1 6 4
7 0 5

Step 1: g=1, h=4, f=5
2 8 3
1 0 4
7 6 5

Step 2: g=2, h=3, f=5
2 0 3
1 8 4
7 6 5

Step 3: g=3, h=2, f=5
0 2 3
1 8 4
7 6 5

Step 4: g=4, h=1, f=5
1 2 3
0 8 4
7 6 5

Step 5: g=5, h=0, f=5
1 2 3
8 0 4
7 6 5

Total cost (number of moves): 5
Nodes expanded: 5
Nodes explored (unique): 6
```

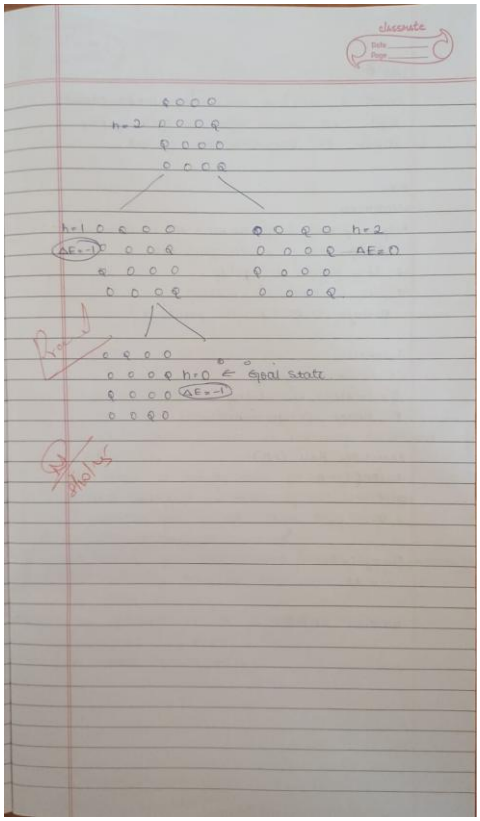
Program4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





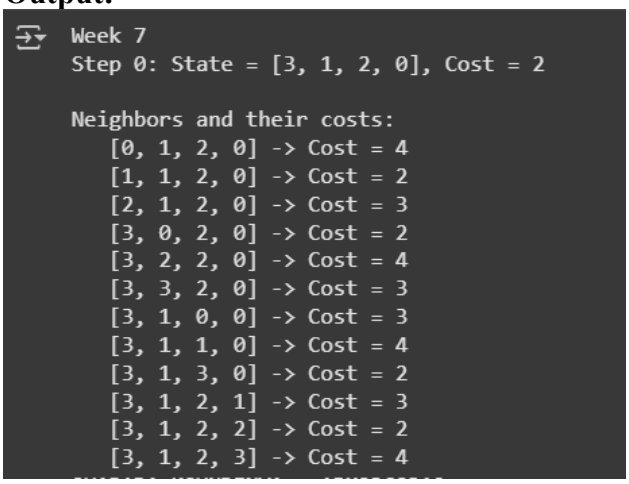


Code:

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

def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:  # move queen
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(initial_state):
    current = initial_state
    current_cost = calculate_cost(current)
    step = 0

    print(f"Step {step}: State = {current}, Cost = {current_cost}")

    while True:
        neighbors = generate_neighbors(current)
        neighbor_costs = [(n, calculate_cost(n)) for n in neighbors]

        # Print state space for this step
        print("\nNeighbors and their costs:")
        for n, c in neighbor_costs:
            print(f"   {n} -> Cost = {c}")

        # Pick the best neighbor (lowest cost)
        best_neighbor, best_cost = min(neighbor_costs, key=lambda x: x[1])

        if best_cost >= current_cost:
            break

        step += 1
        current, current_cost = best_neighbor, best_cost
        print(f"\nStep {step}: Move to {current}, Cost = {current_cost}")

        if current_cost == 0:
            print("\nGoal reached! Solution found.")
            break

initial_state = [3, 1, 2, 0]
```

hill_climbing(initial_state)

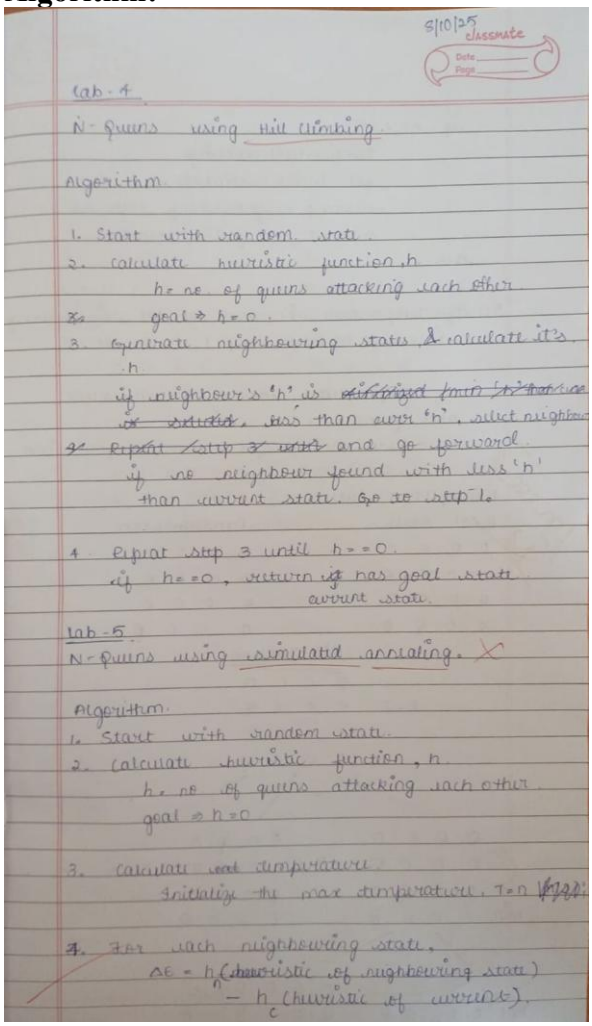
Output:



Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



8/10/25

Lab - 4

N-Queens using Hill climbing

Algorithm

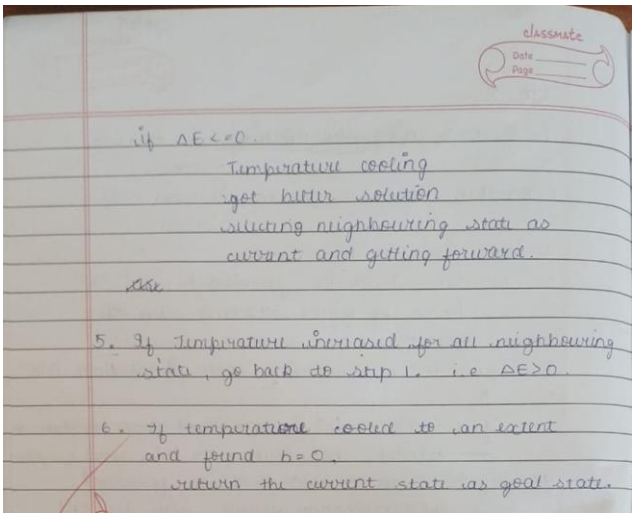
1. Start with random state.
2. Calculate heuristic function, h
   h = no. of queens attacking each other
   goal ⇒ h = 0.
3. Generate neighbouring states & calculate it's h
   if neighbour's 'h' is ~~strictly less than~~ or ~~equal to~~ less than curr 'h', select neighbour
4. Repeat ~~step 3 until~~ and go forward.
   if no neighbour found with less 'h' than current state, go to step 1.
4. Repeat step 3 until h = 0.
   if h = 0, return it has goal state.
   current state.

Lab - 5

N-Queens using simulated annealing.

Algorithm.

1. Start with random state.
2. Calculate heuristic function, h.
   h = no of queens attacking each other.
   goal ⇒ h = 0
3. Calculate ~~cost~~ temperature.
   Initialize the max temperature. T = 0
4. For each neighbouring state.
   ΔE = h_n (heuristic of neighbouring state)
   − h_c (heuristic of current).



Code:

```
import random
import math

def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def get_random_neighbor(state):
    n = len(state)
    new_state = list(state)
    col = random.randint(0, n - 1)
    row = random.randint(0, n - 1)
    new_state[col] = row
    return new_state

def simulated_annealing(n=8, max_iterations=10000, initial_temp=100.0, cooling_rate=0.99):

    current = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current)
    best = current
    best_cost = current_cost
    temperature = initial_temp

    for _ in range(max_iterations):
        if current_cost == 0:
            break

        neighbor = get_random_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
```

```
        delta = neighbor_cost - current_cost

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current, current_cost = neighbor, neighbor_cost

            if current_cost < best_cost:
                best, best_cost = current, current_cost

        temperature *= cooling_rate
        if temperature < 1e-6:
            break

    return best, best_cost

best_state, best_cost = simulated_annealing()

print("The best position found:", best_state)
print("cost =", best_cost)
```

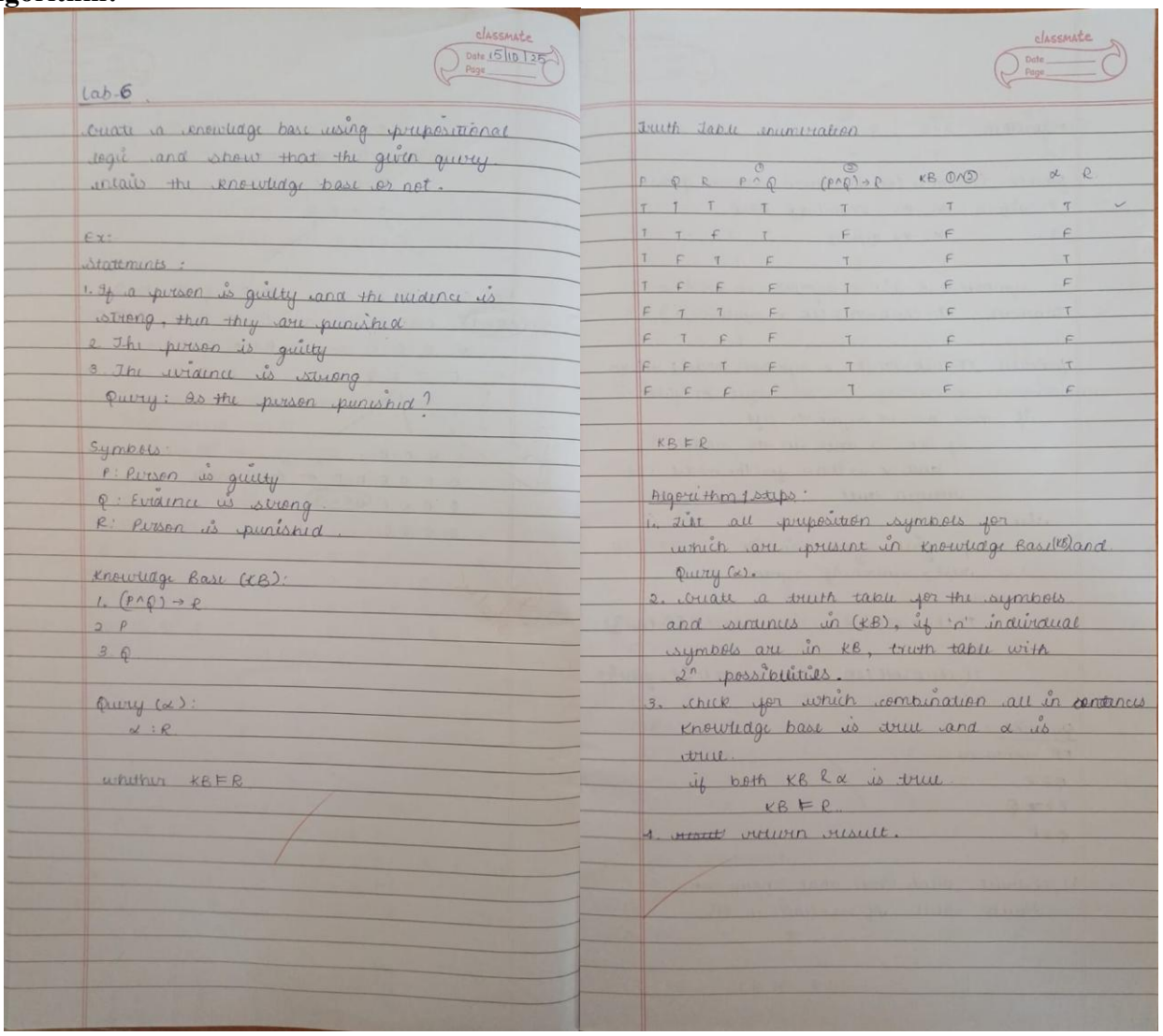
Output:

```
The best position found: [5, 2, 6, 1, 7, 4, 0, 3]
cost = 0
```

Program 6

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

Algorithm:



Lab 6

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

Ex:

Statements:

1. If a person is guilty and the evidence is strong, then they are punished
2. The person is guilty
3. The evidence is strong

Query: Is the person punished?

Symbols:

P: Person is guilty
Q: Evidence is strong
R: Person is punished

Knowledge Base (KB):

1. $(P \wedge Q) \rightarrow R$
2. P
3. Q

Query (α):

α : R

whether KB ⊨ R

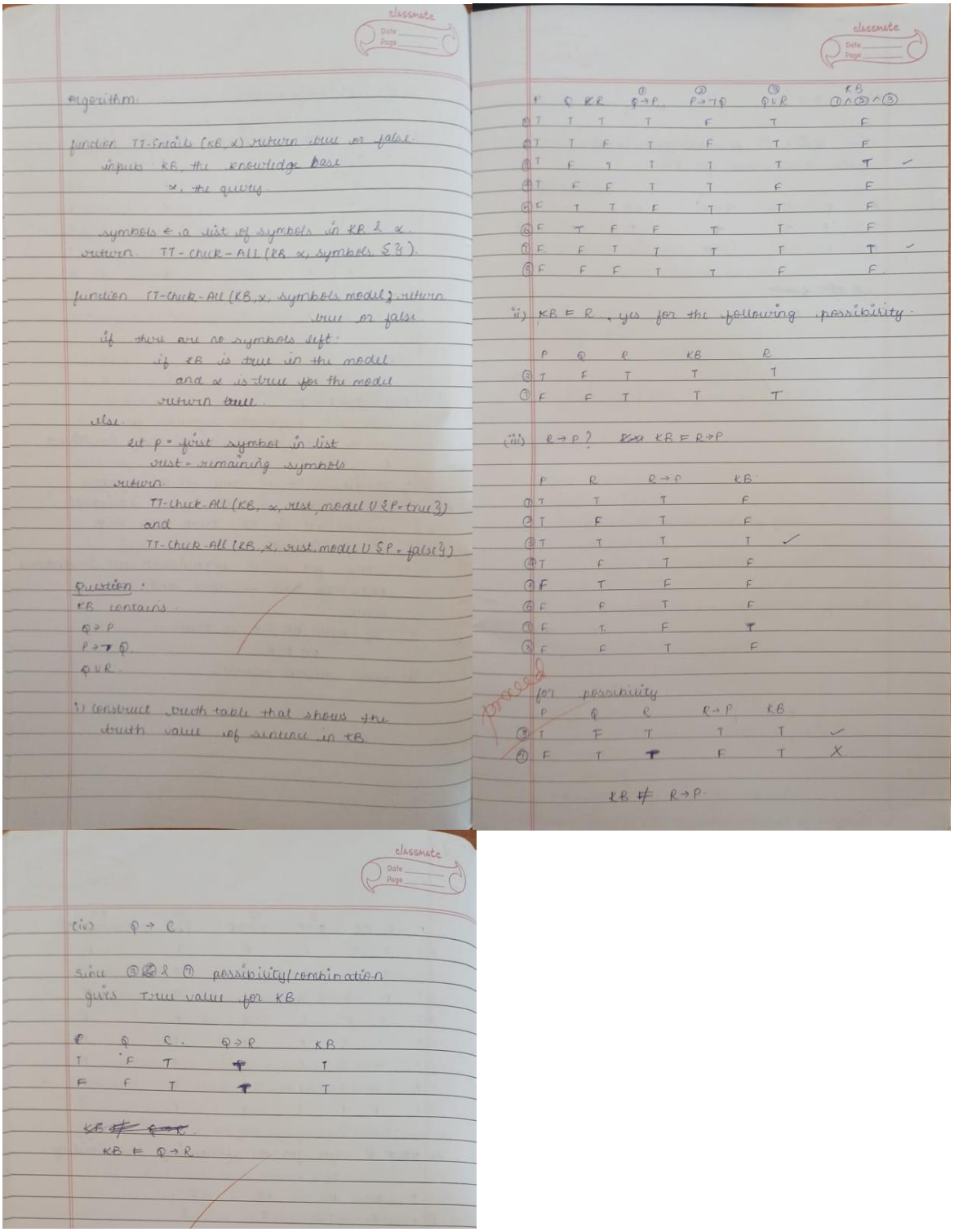
Truth table enumeration

P	Q	R	P ∧ Q ①	(P∧Q) → R ②	KB ①∧②	α R
T	T	T	T	T	T	T ✓
T	T	F	T	F	F	F
T	F	T	F	T	F	T
T	F	F	F	T	F	F
F	T	T	F	T	F	T
F	T	F	F	T	F	F
F	F	T	F	T	F	T
F	F	F	F	T	F	F

KB ⊨ R

Algorithm steps:

1. List all proposition symbols for which are present in knowledge Base(KB) and Query(α).
2. Create a truth table for the symbols and sentences in (KB). if 'n' individual symbols are in KB, truth table with 2^n possibilities.
3. Check for which combination all in sentences Knowledge base is true and α is true.
   if both KB & α is true
   KB ⊨ R.
4. return result.



Code:

```
import itertools

def eval_expr(expr, model):
    try:
        return eval(expr, {}, model)
```

```
    except:
        return False

def tt_entails(KB, query):
    symbols = sorted(set([ch for ch in KB + query if ch.isalpha()]))

    print("\nTruth Table:")
    print(" | ".join(symbols) + " | KB | Query")
    print("-" * (6 * len(symbols) + 20))

    entails = True
    for values in itertools.product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        kb_val = eval_expr(KB, model)
        query_val = eval_expr(query, model)

        row = " | ".join(["T" if model[s] else "F" for s in symbols])
        print(f"{row} |  {kb_val}  |  {query_val}")

        if kb_val and not query_val:
            entails = False

    return entails

KB = input("Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): ")
query = input("Enter Query: ")

result = tt_entails(KB, query)

print("\nResult:")
if result:
    print("KB entails Query (True in all cases).")
else:
    print("KB does NOT entail Query.")
```

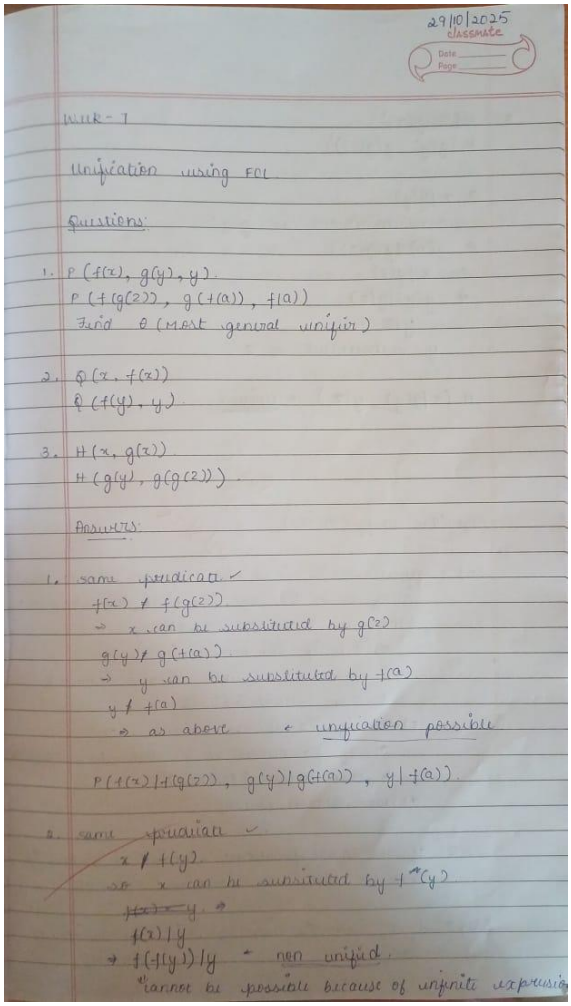
Output:

```
Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): (A|C)&(B|~C)
Enter Query:  A|B

Truth Table:
A | B | C | KB | Query
-------------------------------------
F | F | F |  0  |  False
F | F | T |  0  |  False
F | T | F |  0  |  True
F | T | T |  1  |  True
T | F | F |  1  |  True
T | F | T |  0  |  True
T | T | F |  1  |  True
T | T | T |  1  |  True

Result:
KB entails Query (True in all cases).
```

Program 7

Implement unification in first order logic

Algorithm:



WEEK - 7

Unification using FOL

Questions:

1. P(f(x), g(y), y)
   P(f(g(z)), g(f(a)), f(a))
   Find θ (most general unifier)

2. Q(x, f(x))
   Q(f(y), y)

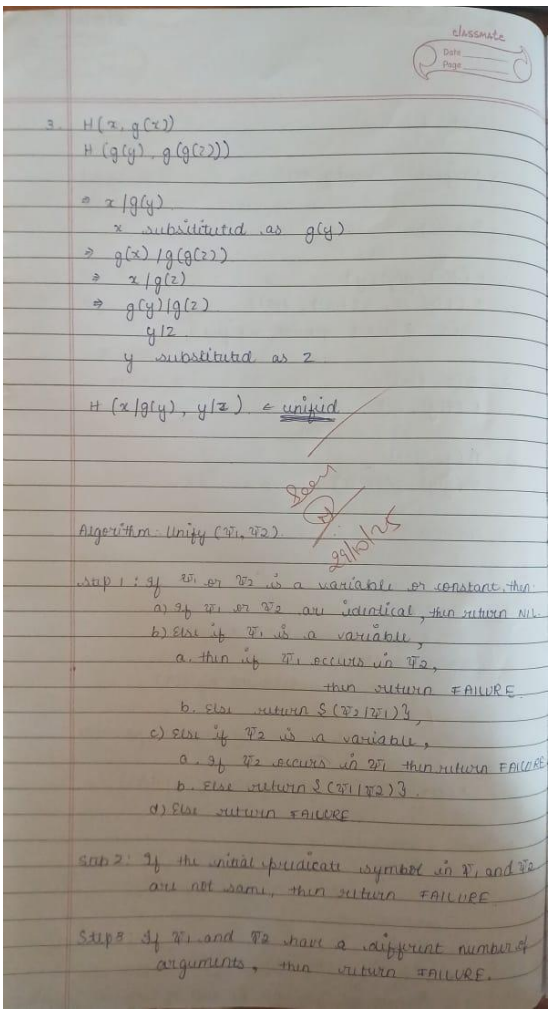
3. H(x, g(z))
   H(g(y), g(g(z)))

Answers:

1. same predicate ✓
   f(x) ≠ f(g(z))
   ⇒ x can be substituted by g(z)
   g(y) ≠ g(f(a))
   ⇒ y can be substituted by f(a)
   y ≠ f(a)
   ⇒ as above ∴ unification possible

   P(f(x)/f(g(z)), g(y)/g(f(a)), y/f(a))

2. same predicate ✓
   x ≠ f(y)
   so x can be substituted by f(y)
   ~~f(x) = y~~ ⇒
   f(x)/y
   ⇒ f(f(y))/y ∴ not unified.
   cannot be possible because of infinite expressio



3. H(x, g(z))
   H(g(y), g(g(z)))

   ⇒ x/g(y)
   x substituted as g(y)
   ⇒ g(x)/g(g(z))
   ⇒ x/g(z)
   ⇒ g(y)/g(z)
   y/z
   y substituted as z

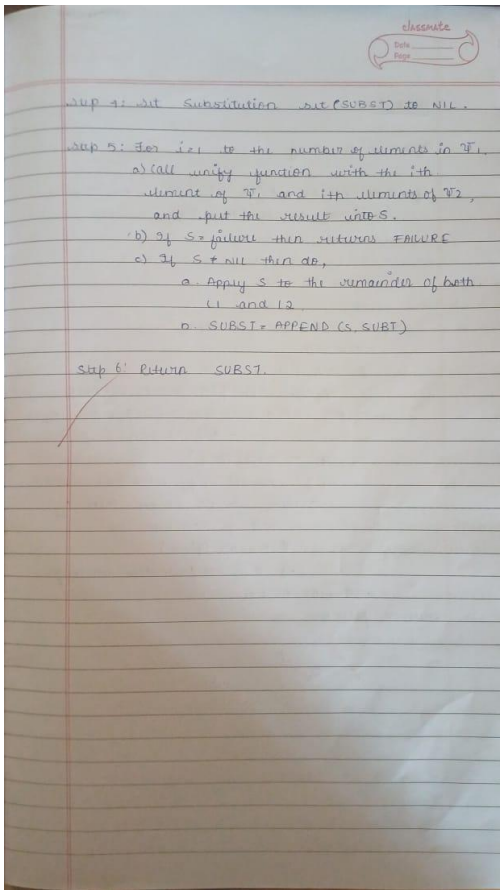
   H(x/g(y), y/z) ∴ unified

Algorithm Unify($\vec{\psi}_1, \vec{\psi}_2$)

Step 1: If $\vec{\psi}_1$ or $\vec{\psi}_2$ is a variable or constant, then
a) If $\vec{\psi}_1$ or $\vec{\psi}_2$ are identical, then return NIL
b) Else if $\vec{\psi}_1$ is a variable,
   a. then if $\vec{\psi}_1$ occurs in $\vec{\psi}_2$,
      then return FAILURE
   b. Else return $\{(\vec{\psi}_2/\vec{\psi}_1)\}$
c) Else if $\vec{\psi}_2$ is a variable,
   a. If $\vec{\psi}_2$ occurs in $\vec{\psi}_1$ then return FAILURE
   b. Else return $\{(\vec{\psi}_1/\vec{\psi}_2)\}$.
d) Else return FAILURE

Step 2: If the initial predicate symbol in $\vec{\psi}_1$ and $\vec{\psi}_2$ are not same, then return FAILURE

Step 3: If $\vec{\psi}_1$ and $\vec{\psi}_2$ have a different number of arguments, then return FAILURE.



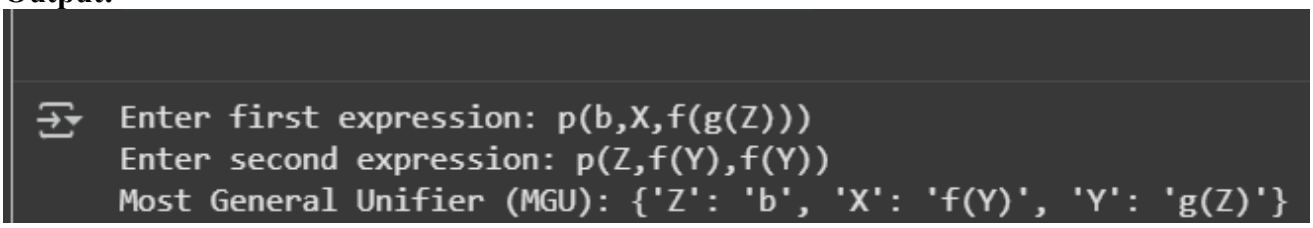
Code:

```
def occurs_check(var, term, subst):
    if var == term:
        return True
    elif isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    elif term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x.isupper():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.isupper():
        return unify_var(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x) != len(y):
```

```
            return None
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, 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 x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst

def parse_expr(s):
    s = s.replace(" ", "")
    if '(' not in s:
        return s
    name_end = s.index('(')
    name = s[:name_end]
    args = []
    depth = 0
    current = ""
    for c in s[name_end+1:-1]:
        if c == ',' and depth == 0:
            args.append(parse_expr(current))
            current = ""
        else:
            if c == '(':
                depth += 1
            elif c == ')':
                depth -= 1
            current += c
    if current:
        args.append(parse_expr(current))
    return tuple([name] + args)

def expr_to_str(expr):
    if isinstance(expr, tuple):
        return expr[0] + "(" + ",".join(expr_to_str(e) for e in expr[1:]) + ")"
    else:
        return expr

expr1_input = input("Enter first expression: ")
expr2_input = input("Enter second expression: ")
```

```
expr1 = parse_expr(expr1_input)
expr2 = parse_expr(expr2_input)

subst = unify(expr1, expr2, {})

if subst:
    formatted_subst = {var: expr_to_str(val) for var, val in subst.items()}
else:
    formatted_subst = None

print("Most General Unifier (MGU):", formatted_subst)
```

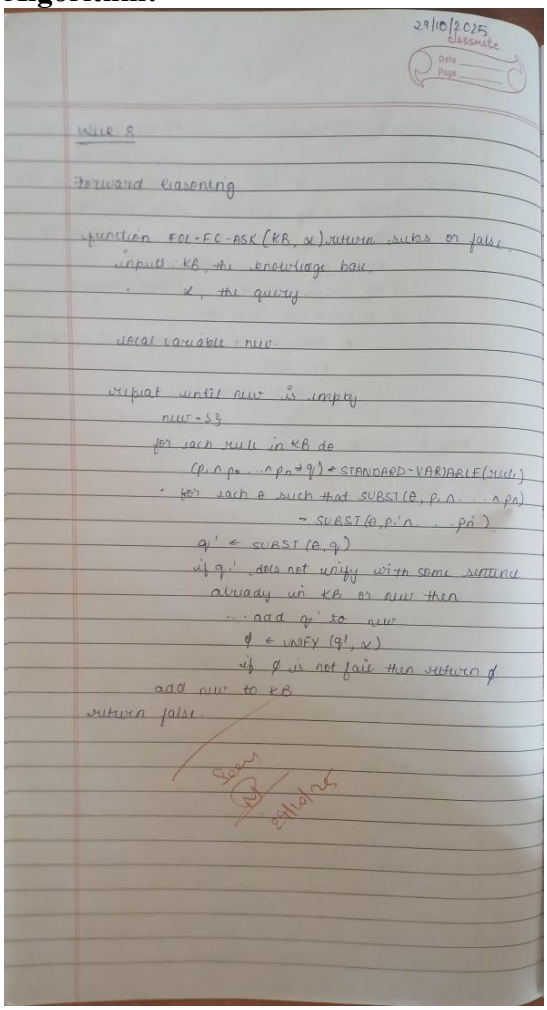
Output:



Program 8

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

Algorithm:



Code:

```
import re

def isVariable(x):
    """
    Heuristically checks if a string is a variable (single, lowercase, alphabetic).
    """
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    """
    Extracts the parenthesized attribute strings. (e.g., 'P(a,b)' -> ['(a,b)'])
```

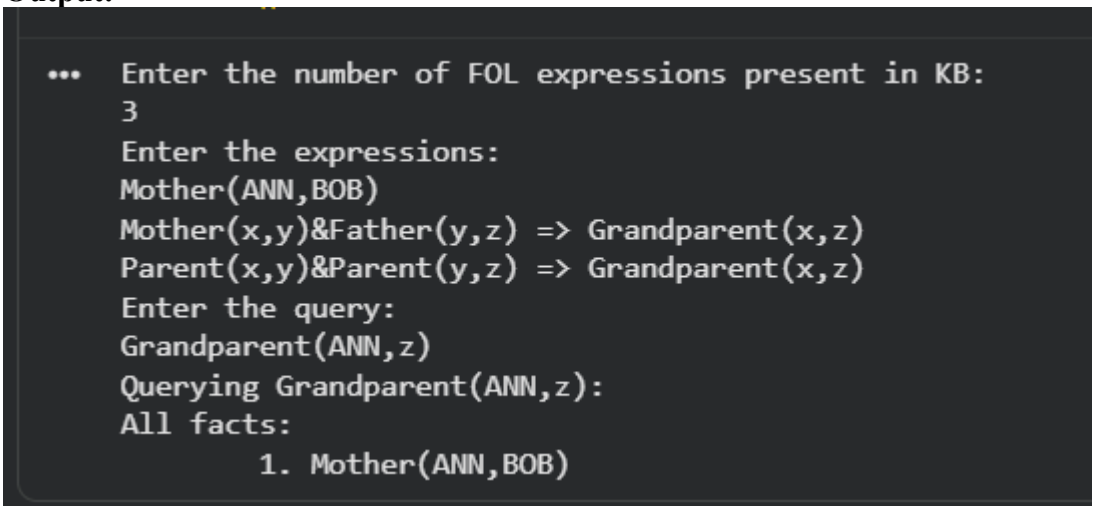
```
    Note: This uses simple regex and will struggle with nested functions.
    """
    expr = r'\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    """
    Extracts the predicate symbol. (e.g., 'P(a,b)' -> ['P'])
    """
    expr = r'([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    """
    Represents a single fact in the KB (e.g., 'Mother(a,b)').
    """
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        # result is true if the fact contains at least one constant
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        """
        Splits the expression into predicate and parameters.
        """
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        """
        Returns True if the fact has at least one constant.
        (Usage seems specific to this simplified inference process).
        """
        return self.result

    def getConstants(self):
        """
        Returns a list of constants (or None for variables).
        """
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        """
        Returns a list of variables (or None for constants).
        """
        return [v if isVariable(v) else None for v in self.params]
```

```
    def substitute(self, constants):
        """
        Applies a substitution (constants) to create a new grounded Fact.
        (Seems unused in the evaluate method, but provided here.)
        """
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in self.params])})"
        return Fact(f)

class Implication:
    """
    Represents an implication rule (e.g., 'Mother(x,y)&Father(y,z) => Grandparent(x,z)').
    """
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        # LHS is a list of Facts connected by '&'
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        """
        Attempts to apply the rule (forward chaining) using existing facts.
        This simplified version finds constants by position matching.
        """
        constants = {}
        new_lhs = [] # Facts that successfully matched LHS predicates

        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    # Collect constants based on variable positions
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)

        # Check if we matched all LHS facts and if all matched facts are "true" (have constants)
        if len(new_lhs) == len(self.lhs) and all([f.getResult() for f in new_lhs]):
            # Construct the RHS fact by applying the collected constants
            predicate, attributes = getPredicates(self.rhs.expression)[0], str(getAttributes(self.rhs.expression)[0])
            for key in constants:
                if constants[key]:
                    # Simple string replacement for substitution
                    attributes = attributes.replace(key, constants[key])

            expr = f'{predicate}{attributes}'
            return Fact(expr)

        return None

class KB:
```

```
    """
    The Knowledge Base containing facts and implications.
    """
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        """
        Adds a new fact or implication to the KB and performs inference.
        """
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))

        # Run forward chaining (simple form: only one pass)
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def ask(self, e):
        """
        Searches the current facts for facts matching the query's predicate.
        """
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        """
        Prints all unique facts currently in the KB.
        """
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')

def main():
    """
    Main execution function to run the KB system.
    """
    kb = KB()
    print("Enter the number of FOL expressions present in KB:")
    # Example input: 2
    n = int(input())

    print("Enter the expressions:")
```

```
    # Example inputs:
    # Mother(ANN,BOB)
    # Mother(x,y)&Father(y,z) => Grandparent(x,z)
    for i in range(n):
        fact = input()
        kb.tell(fact)

    print("Enter the query:")
    # Example input: Grandparent(ANN,z)
    query = input()

    kb.ask(query)
    kb.display()

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

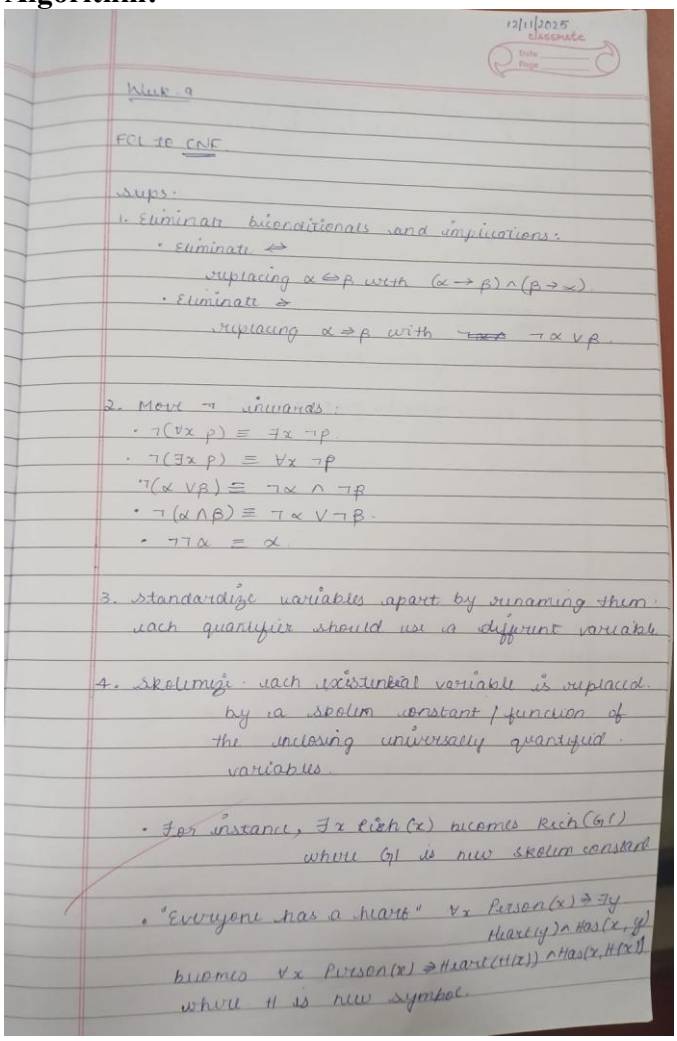
Output:

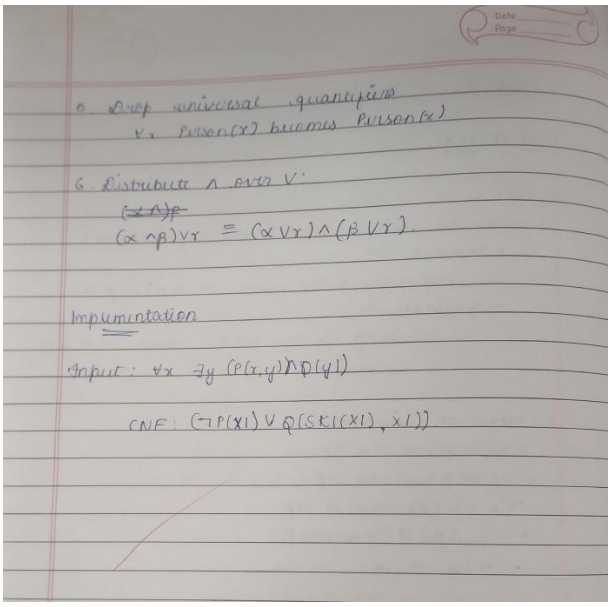


Program 9

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

Algorithm:





Code:

```
import itertools

# ---------------------------------------------------------
# Use ASCII strings as operator names
# ---------------------------------------------------------
FORALL = 'FORALL'       # ∀
EXISTS = 'EXISTS'       # ∃
NOT = 'NOT'             # ¬
AND = 'AND'             # ∧
OR = 'OR'               # ∨
IMPLIES = 'IMPLIES'     # →
IFF = 'IFF'             # ↔
ATOM = 'ATOM'

# ---------------------------------------------------------
# Unicode symbols for pretty printing only
# ---------------------------------------------------------
unicode_symbols = {
    FORALL: '∀',
    EXISTS: '∃',
    NOT: '¬',
    AND: '∧',
    OR: '∨',
    IMPLIES: '→',
    IFF: '↔'
}

# ---------------------------------------------------------
# Utility counters for skolem and variables
# ---------------------------------------------------------
_skolem_counter = itertools.count(1)
_var_counter = itertools.count(1)
```

```
def new_skolem_name():
    return f"SK{next(_skolem_counter)}"

def new_var_name():
    return f"X{next(_var_counter)}"

# ------------------------------------------------------------
# STEP 1: Eliminate ↔ and →
# ------------------------------------------------------------
def eliminate_implications(F):
    if F is None:
        return None

    op = F['op']

    if op == ATOM:
        return F
    elif op == IMPLIES:
        # (α → β) ≡ (¬α ∨ β)
        return {'op': OR,
                'left': {'op': NOT, 'body': eliminate_implications(F['left'])},
                'right': eliminate_implications(F['right'])}
    elif op == IFF:
        # (α ↔ β) ≡ (α → β) ∧ (β → α)
        p = eliminate_implications(F['left'])
        q = eliminate_implications(F['right'])
        return {'op': AND,
                'left': {'op': OR, 'left': {'op': NOT, 'body': p}, 'right': q},
                'right': {'op': OR, 'left': {'op': NOT, 'body': q}, 'right': p}}
    elif op == NOT:
        return {'op': NOT, 'body': eliminate_implications(F['body'])}
    elif op in (AND, OR):
        return {'op': op,
                'left': eliminate_implications(F['left']),
                'right': eliminate_implications(F['right'])}
    elif op in (FORALL, EXISTS):
        return {'op': op, 'var': F['var'], 'body': eliminate_implications(F['body'])}
    return F

# ------------------------------------------------------------
# STEP 2: Move negations inward
# ------------------------------------------------------------
def move_negations_inward(F):
    if F['op'] == ATOM:
        return F
    elif F['op'] == NOT:
        sub = F['body']
        if sub['op'] == ATOM:
            return F
        elif sub['op'] == NOT:
            return move_negations_inward(sub['body'])
        elif sub['op'] == AND:
```

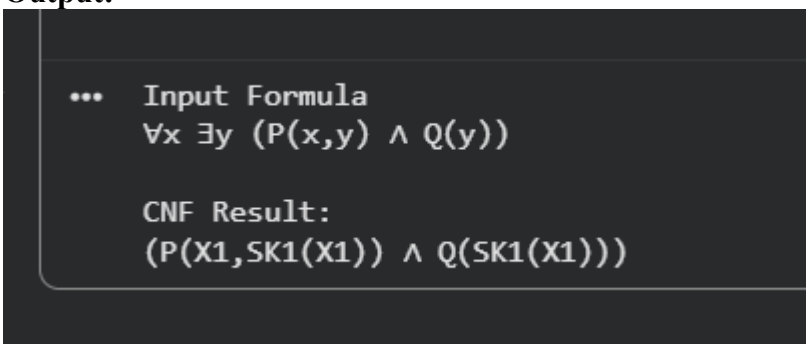
```
            return {'op': OR,
                    'left': move_negations_inward({'op': NOT, 'body': sub['left']}),
                    'right': move_negations_inward({'op': NOT, 'body': sub['right']})}
        elif sub['op'] == OR:
            return {'op': AND,
                    'left': move_negations_inward({'op': NOT, 'body': sub['left']}),
                    'right': move_negations_inward({'op': NOT, 'body': sub['right']})}
        elif sub['op'] == FORALL:
            return {'op': EXISTS, 'var': sub['var'],
                    'body': move_negations_inward({'op': NOT, 'body': sub['body']})}
        elif sub['op'] == EXISTS:
            return {'op': FORALL, 'var': sub['var'],
                    'body': move_negations_inward({'op': NOT, 'body': sub['body']})}
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': move_negations_inward(F['left']),
                'right': move_negations_inward(F['right'])}
    elif F['op'] in (FORALL, EXISTS):
        return {**F, 'body': move_negations_inward(F['body'])}
    return F

# ---------------------------------------------------------
# STEP 3: Standardize variables
# ---------------------------------------------------------
def standardize_variables(F, mapping=None):
    if mapping is None:
        mapping = {}
    if F['op'] == ATOM:
        args = [mapping.get(a, a) for a in F['args']]
        return {'op': ATOM, 'pred': F['pred'], 'args': args}
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': standardize_variables(F['left'], mapping),
                'right': standardize_variables(F['right'], mapping)}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': standardize_variables(F['body'], mapping)}
    elif F['op'] in (FORALL, EXISTS):
        new_var = new_var_name()
        mapping2 = mapping.copy()
        mapping2[F['var']] = new_var
        return {'op': F['op'], 'var': new_var, 'body': standardize_variables(F['body'], mapping2)}
    return F

# ---------------------------------------------------------
# STEP 4: Skolemize (remove ∃)
# ---------------------------------------------------------
def skolemize(F, scope_vars=None):
    if scope_vars is None:
        scope_vars = []
    if F['op'] == ATOM:
        return F
    elif F['op'] == FORALL:
```

```
        return {'op': FORALL, 'var': F['var'], 'body': skolemize(F['body'], scope_vars + [F['var']])}
    elif F['op'] == EXISTS:
        skolem_name = new_skolem_name()
        skolem_term = skolem_name if not scope_vars else f"{skolem_name}({','.join(scope_vars)})"
        return skolemize(substitute(F['body'], F['var'], skolem_term), scope_vars)
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': skolemize(F['left'], scope_vars),
                'right': skolemize(F['right'], scope_vars)}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': skolemize(F['body'], scope_vars)}
    return F

def substitute(F, var, term):
    if F['op'] == ATOM:
        new_args = [term if a == var else a for a in F['args']]
        return {'op': ATOM, 'pred': F['pred'], 'args': new_args}
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': substitute(F['left'], var, term),
                'right': substitute(F['right'], var, term)}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': substitute(F['body'], var, term)}
    elif F['op'] in (FORALL, EXISTS):
        if F['var'] == var:
            return F
        return {**F, 'body': substitute(F['body'], var, term)}
    return F

# ---------------------------------------------------------
# STEP 5: Drop ∀ quantifiers
# ---------------------------------------------------------
def drop_universal_quantifiers(F):
    if F['op'] == FORALL:
        return drop_universal_quantifiers(F['body'])
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': drop_universal_quantifiers(F['left']),
                'right': drop_universal_quantifiers(F['right'])}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': drop_universal_quantifiers(F['body'])}
    return F

# ---------------------------------------------------------
# STEP 6: Distribute ∨ over ∧
# ---------------------------------------------------------
def distribute_or_over_and(F):
    if F['op'] == OR:
        A = distribute_or_over_and(F['left'])
        B = distribute_or_over_and(F['right'])
        if A['op'] == AND:
            return {'op': AND,
```

```
                'left': distribute_or_over_and({'op': OR, 'left': A['left'], 'right': B}),
                'right': distribute_or_over_and({'op': OR, 'left': A['right'], 'right': B})}
        elif B['op'] == AND:
            return {'op': AND,
                'left': distribute_or_over_and({'op': OR, 'left': A, 'right': B['left']}),
                'right': distribute_or_over_and({'op': OR, 'left': A, 'right': B['right']})}
        else:
            return {'op': OR, 'left': A, 'right': B}
    elif F['op'] == AND:
        return {'op': AND,
              'left': distribute_or_over_and(F['left']),
              'right': distribute_or_over_and(F['right'])}
    else:
        return F

# ------------------------------------------------------------
# PRETTY PRINTING (SYMBOLIC OUTPUT)
# ------------------------------------------------------------
def pretty(F):
    """Convert CNF structure to readable symbolic formula."""
    op = F['op']
    if op == ATOM:
        args = ",".join(F['args'])
        return f"{F['pred']}({args})"
    elif op == NOT:
        return f"¬{pretty(F['body'])}"
    elif op in (AND, OR):
        return f"({pretty(F['left'])} {unicode_symbols[op]} {pretty(F['right'])})"
    return str(F)

def pretty_full(F):
    op = F['op']
    if op == ATOM:
        args = ",".join(F['args'])
        return f"{F['pred']}({args})"
    elif op == NOT:
        return f"¬{pretty_full(F['body'])}"
    elif op in (AND, OR):
        return f"({pretty_full(F['left'])} {unicode_symbols[op]} {pretty_full(F['right'])})"
    elif op in (IMPLIES, IFF):
        return f"({pretty_full(F['left'])} {unicode_symbols[op]} {pretty_full(F['right'])})"
    elif op == FORALL:
        return f"∀{F['var']} {pretty_full(F['body'])}"
    elif op == EXISTS:
        return f"∃{F['var']} {pretty_full(F['body'])}"
    else:
        return str(F)

# ------------------------------------------------------------
# MAIN WRAPPER
# ------------------------------------------------------------
def to_CNF(F):
```

```
    F1 = eliminate_implications(F)
    F2 = move_negations_inward(F1)
    F3 = standardize_variables(F2)
    F4 = skolemize(F3)
    F5 = drop_universal_quantifiers(F4)
    F6 = distribute_or_over_and(F5)
    return F6

# ------------------------------------------------------------
# EXAMPLE
# ------------------------------------------------------------
if __name__ == "__main__":
    # ∀x∃y(P(x,y)∧Q(y))
    formula = {
    'op': FORALL, 'var': 'x',
    'body': {
        'op': EXISTS, 'var': 'y',
        'body': {
            'op': AND,
            'left': {'op': ATOM, 'pred': 'P', 'args': ['x', 'y']},
            'right': {'op': ATOM, 'pred': 'Q', 'args': ['y']}
              }
             }
            }


    print("Input Formula ")
    print(pretty_full(formula))

    cnf = to_CNF(formula)

    print("\nCNF Result:")
    print(pretty(cnf))
```

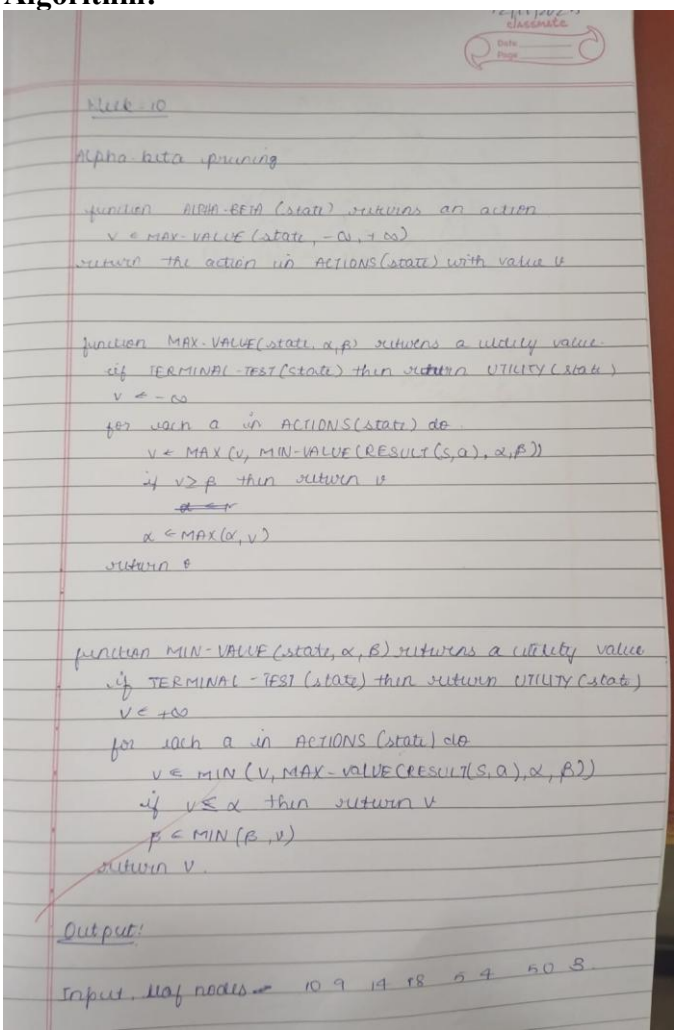
Output:



Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Week - 10

Alpha beta pruning

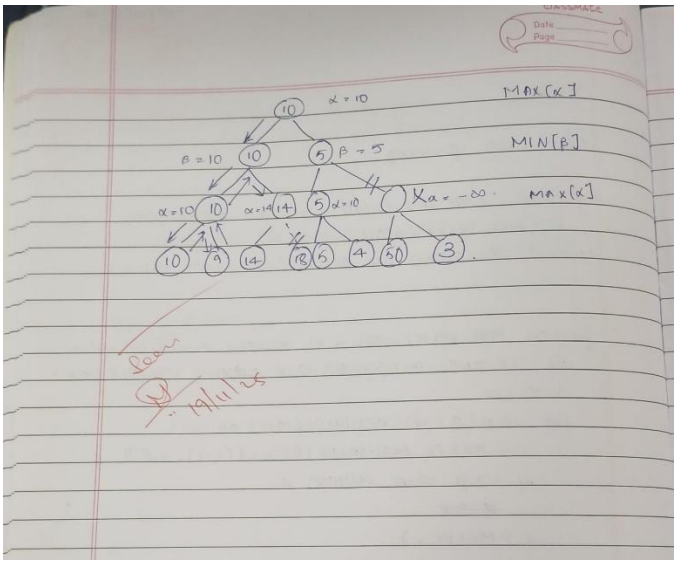
function ALPHA-BETA (state) returns an action
v ← MAX-VALUE (state, -∞, +∞)
return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← -∞
for each a in ACTIONS(state) do
v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
if v ≥ β then return v
α ← MAX(α, v)
return v

function MIN-VALUE(state, α, β) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← +∞
for each a 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:

Input. Nof nodes = 10 9 14 18 5 4 50 8.



Code:

```
import math

# Alpha-Beta Pruning Algorithm
def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta, max_depth, path, pruned):
    # Base case: leaf node
    if depth == max_depth:
        return values[node_index], [node_index]

    if maximizing_player:
        best = -math.inf
        best_path = []
        for i in range(2):  # two children per node
            val, child_path = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth, path, pruned)
            if val > best:
                best = val
                best_path = [node_index] + child_path
            alpha = max(alpha, best)
            if beta <= alpha:
                pruned.append((node_index, "Right" if i == 0 else "Left"))
                break
        return best, best_path
    else:
        best = math.inf
        best_path = []
        for i in range(2):
            val, child_path = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth, path, pruned)
            if val < best:
                best = val
                best_path = [node_index] + child_path
            beta = min(beta, best)
            if beta <= alpha:
                pruned.append((node_index, "Right" if i == 0 else "Left"))
                break
        return best, best_path


# Example usage
if __name__ == "__main__":
```

```
# Example game tree (leaf node values)
values = [3, 5, 6, 9, 1, 2, 0, -1]

print("Leaf Node Values:", values)
path = []
pruned = []

max_depth = 3
result, best_path = alpha_beta(0, 0, True, values, -math.inf, math.inf, max_depth, path, pruned)

print("\nOptimal Value at Root Node:", result)
print("Best Path (Node Indices):", best_path)
print("Pruned Nodes:", pruned)
```

Output:

```
Leaf Node Values: [3, 5, 6, 9, 1, 2, 0, -1]

Optimal Value at Root Node: 5
Best Path (Node Indices): [0, 0, 0, 1]
Pruned Nodes: [(1, 'Right'), (1, 'Right')]
```