VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

ARYAN NAVLANI(1BM23CS055)

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 **ARYAN NAVLANI(1BM23CS055),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Lab faculty **Sandhya A Kulkarni** Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Infosys®
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aryan Navlani

for successfully completing the course

OpenAI Generative Pre-trained Transformer 3 (GPT-3) for developers

on November 20, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B.N.

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

Issued on: Thursday, November 20, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Github Link: https://github.com/Shreyas-2607/AI_LAB

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:

TIC TAC TOE GAME

```
import math :

def print_board (board):
    print ()
    for i in range (3):
        print ("|".join (board[i]))
        if i<2
            print ("------");
    print ()

def checks winner (board, player):
if for all in range (3) [board [i] [j] == player for i in
                range (3)) : # Rows.
            return True;

if (all (board [i] [j] == player for i in range (3)) or all
    (board [i] [2-i] == player for i in range (3)):
    return True
    return false.

# if the board is full => draw

def is_full (board):
    return all (board [i] [j] != ' '
    for i in range (3) for j in range (3))

# back tracking algo:

def minimax (board, depth, maxi):

    if check_winner (board, "o"):
        return 1, # compute;
```

```
def best_move (board):
    best score = -math.inf
    move = None
    for i in range (3):
        for j in range (3):
            if board [i][j] = " ":
                board [i][j] = "O"
                score = minimax (board, 0, false)
                board [i] [j] = " "

    if score >best_score :
        best_score = score
            move = (i,j)

return move :
```

```
if check_winner (board: "x"):
        return -1 # user.

    if is_full (board):
        return 0

# In computer's turn maximise the score.

   if maxi:
        best_score = -math.int
        for i in range (3):
            for j in range (3):
                if board [i] [j] "":
                    board [i] [j] = "o"

score = minimax (board, depth+1, false);
      board [i] [j] = "";
      best_score = max (best_score, score);

return best_score

#user's turn [minimising score]

 else:
      best_score = -math.inf
      for i in range (3):
          for j in range (3):
              if board [i] [j] == "":
                  best_score = min (best_score, score)

return best_score.
```

Code:

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

def check_winner(board, player): for i in range(3):
if all(board[i][j] == player for j in range(3)): return True
if all(board[j][i] == player for j in range(3)): return True
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_draw(board):
return all(board[i][j] != '-' for i in range(3) for j in range(3))

def minimax(board, is_ai_turn):
if check_winner(board, 'O'): # AI win return 1
if check_winner(board, 'X'): # Player win return -1
if is_draw(board):
return 0

if is_ai_turn:
best_score = -float('inf') for i in range(3):
for j in range(3):
if board[i][j] == '-':
board[i][j] = 'O'
score = minimax(board, False) board[i][j] = '-'
best_score = max(score, best_score) return best_score
else:
best_score = float('inf') for i in range(3):
for j in range(3):
if board[i][j] == '-':
board[i][j] = 'X'
score = minimax(board, True) board[i][j] = '-'
best_score = min(score, best_score) return best_score

def manual_game():
board = [['-' for _ in range(3)] for _ in range(3)] print("Initial Board:")
print_board(board)

while True:
# Input X move while True:
try:
x_row = int(input("Enter X row (1-3): ")) - 1 x_col = int(input("Enter X col (1-3): ")) - 1
if board[x_row][x_col] == '-': board[x_row][x_col] = 'X' break
else:
print("Cell occupied!") except:
print("Invalid input!")

print("Board after X move:") print_board(board)

if check_winner(board, 'X'):
print("X wins!") break
if is_draw(board):
print("Draw!") break

# Input O move while True:
try:
o_row = int(input("Enter O row (1-3): ")) - 1 o_col = int(input("Enter O col (1-3): ")) - 1
```

```
if board[o_row][o_col] == '-': board[o_row][o_col] = 'O' break
else:
print("Cell occupied!") except:
print("Invalid input!")

print("Board after O move:") print_board(board)

if check_winner(board, 'O'): print("O wins!")

break
if is_draw(board):
print("Draw!") break

# AI evaluates the board (from current position)
cost = minimax(board, True) # AI's turn to move next print(f"AI evaluation cost from this position: {cost}")

manual_game()
```

LAB → 1

VACCUM CLEANER :

```
room = {
        'A' = 'Dirty'          'C' = 'Dirty'
        'B' = 'Dirty'          'D' = 'Dirty'
}
    vac_loc = 'A'

    def suck ():
        print (f'Sucking dirt in room + vac_loc}');

        rooms [vac_voc] = 'clean'

def move ():
    global vac_loc:
     if (vac_loc == A);
        print (Move to B)
        vac_loc = B

    else if (vac_loc == B);
        print (Move to B)
        vac_loc = B

  else if (vac_loc == C):
      print (Move to D)
  else {
     print (Move to A)
        vac_loc = A
    }

while 'Dirty' in room values ():
  if rooms [vac-loc] = 'Dirty':
     suck ()
```

classmate
Date
Page

```
else
    move ()

print ('All rooms are clean');
```

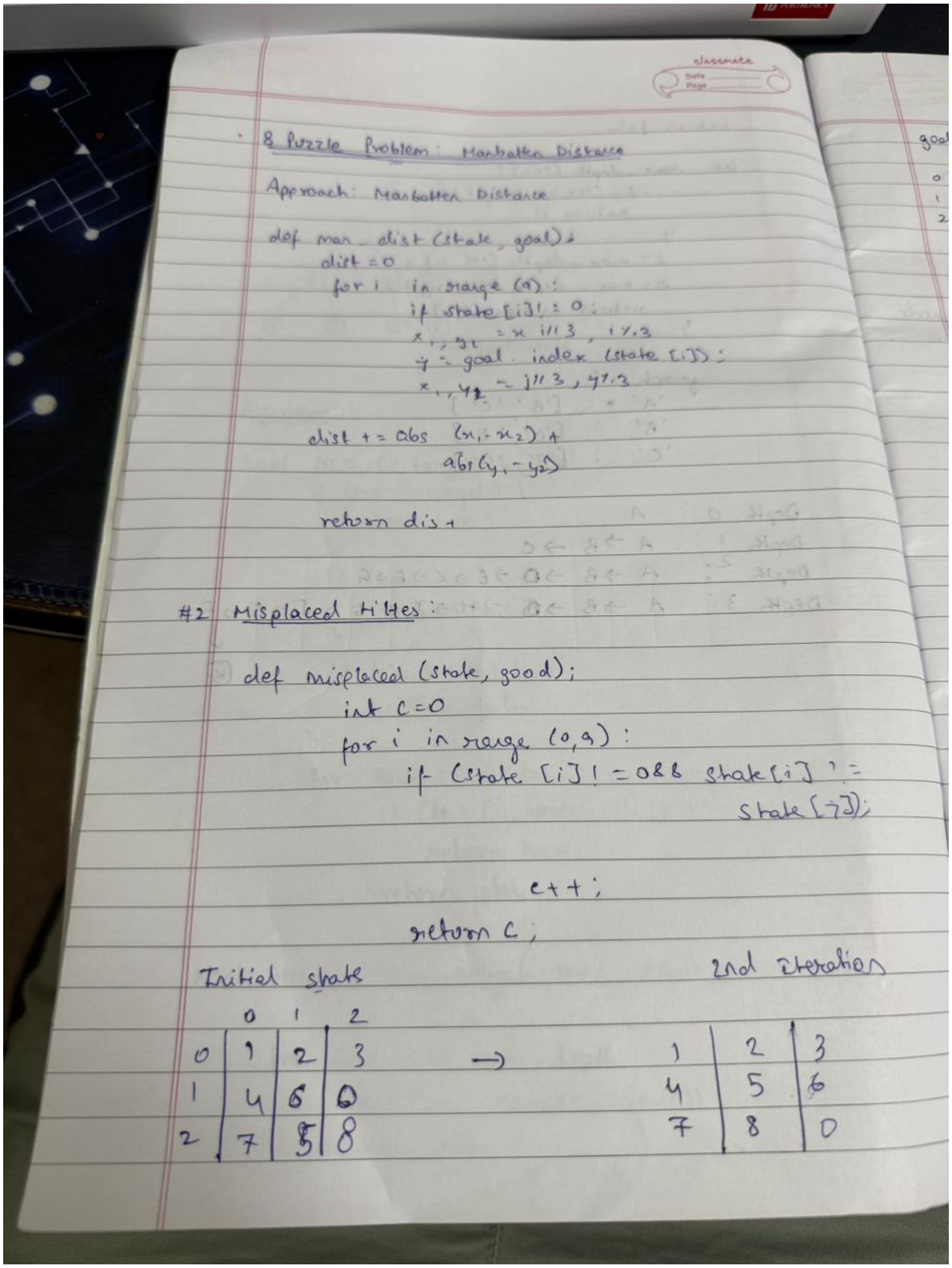
Code:

```
def vacuum_cleaner():
# Taking user input for the state of each room
state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
state_C = int(input("Enter state of C (0 for clean, 1 for dirty): "))
state_D = int(input("Enter state of D (0 for clean, 1 for dirty): "))
location = input("Enter location (A, B, C, or D): ").upper()

cost = 0
rooms = {'A': state_A, 'B': state_B, 'C': state_C, 'D': state_D}

# Function to clean a room and update the cost def clean_room(room):
nonlocal cost
if rooms[room] == 1: print(f"Cleaned {room}.") rooms[room] = 0
cost += 1 else:
print(f"{room} is clean.")

if location == 'A': clean_room('A')
print("Moving vacuum right")
clean_room('B')
print("Moving vacuum down")
clean_room('D')
print("Moving vacuum left")
clean_room('C')
elif location == 'B': clean_room('B')
print("Moving vacuum left")
clean_room('A')
print("Moving vacuum down")
clean_room('D')
print("Moving vacuum right")
clean_room('C')

elif location == 'C': clean_room('C')
print("Moving vacuum right")
clean_room('D')
print("Moving vacuum up")
clean_room('B')
print("Moving vacuum left")
clean_room('A')

elif location == 'D': clean_room('D')
print("Moving vacuum up")
clean_room('B')
print("Moving vacuum right")

clean_room('C')
print("Moving vacuum left")
clean_room('A')

else:
print("Invalid starting location!")

print(f"Cost: {cost}") print("Room states:", rooms)
```

Program 2

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

Algorithm:



8 Puzzle Problem: Manhatten Distance

Approach: Manhatten Distance

```
def man_dist (state, goal):
    dist = 0
    for i in range (9):
        if state [i] != 0:
            x1, y1 = x i// 3 , i%3
            j = goal. index (state [i]):
            x2, y2 = j// 3 , j%3

    dist += abs (x1 - x2) +
            abs (y1 - y2)

    return dist
```

#2 Misplaced tiles:

```
def misplaced (state, good);
    int c=0
    for i in range (0,9):
        if (state [i] != 0 && state[i] != 
                                 state [j]);

            c++;
        return c;
```

Initial state

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

→

2nd iteration

1	2	3
4	5	6
7	8	0

goal state:

0	1	2	3
1	4	5	6
2	7	8	0 ← empty

Misplaced Tiles → 1 + 1 + 1 = 3

(6) (5) (8)

Manhattan:

6: |(1 − 1)| + |(1 − 2)| = 1
5: |(2 − 1)| + |1 − 1| = 1
8: |(2 − 2)| + |2 − 1| = 1

Code:

```
from collections import deque def find_blank(state):
    """Finds the position of the blank tile (0).""" for i in range(3):
        for j in range(3):
            if state[i][j] == 0: return (i, j)def
                get_neighbors(state):
    """Generates all possible next states from the current state.""" neighbors = []
    blank_row, blank_col = find_blank(state)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]  # Right, Left, Down, Up

    for move_row, move_col in moves:
        new_row, new_col = blank_row + move_row, blank_col + move_col

        if 0 <= new_row < 3 and 0_state = ( (1, 2, 3),
    (4, 0, 5),
    (6, 7, 8)
)

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

solution_path = dfs(initial_state, goal_state) if solution_path:
    print("Solution Found!")
    for i, state in enumerate(solution_path): print(f"Step {i+1}:")
        for row in state: print(row)
        print("-" * 10) else:
    print("No solution exists.")
```

Program 3

Implement A* search algorithm

Algorithm:

LAB-3

- 8 Puzzel by A^* Method
- Main function used → $f(n) = g(n) + h(n)$
- $g(n)$ → Number of moves made
- $h(n)$ → Number of misplaced tiles

→ We compare the initial and goal state, if the node matches, we stop. If the nodes don't match, we countinue using the function $[f(n) = g(n) + h(n)]$

When $N=8$
Number of Rows $= \sqrt{8+1} = \sqrt{9} = 3$
Number of Columns $= \sqrt{8+1} = \sqrt{9} = 3$

Initial State

1	2	3
8		4
7	6	5

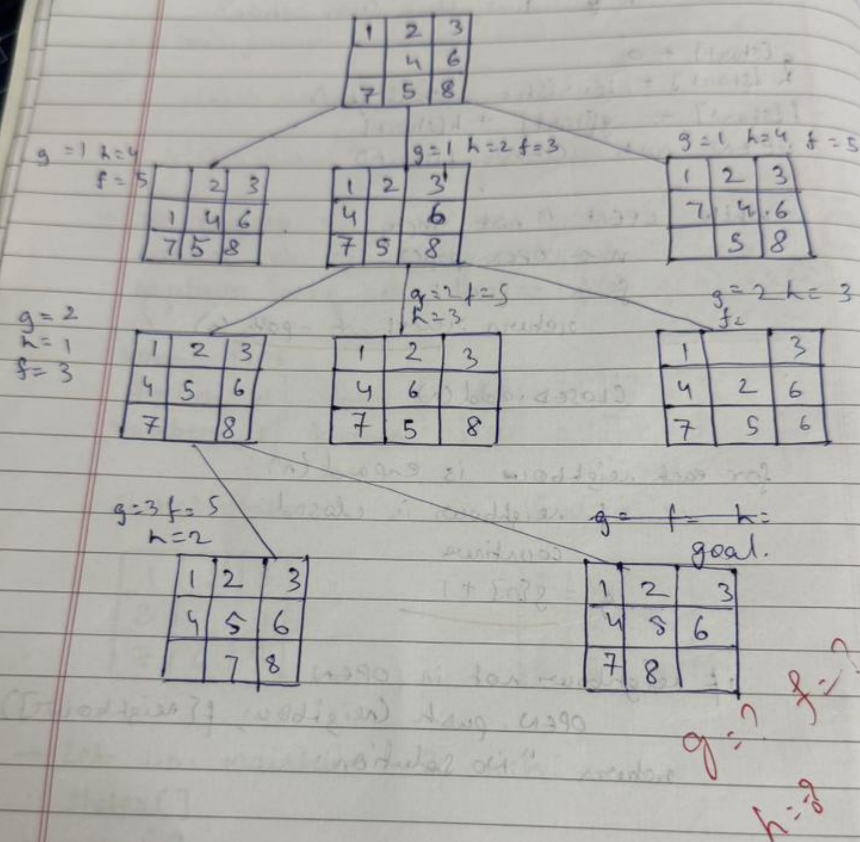
Goal State

2	8	1
	4	3
7	6	5

→ Let us maintain 2 lists:
- States[]
- paths[]

```
def find_blank (curr_state {
        for i in range (3):
            for j in range (3):
                if(curr_state [i][j] == '0'):
                    return i,j:
```

```
def find_possible_state (curr_state)
        x, y = find_blank (curr_state)

g [start] ← 0
h [start] ← heuristic (start, goal)
f [start] ← g[start] + h[start]
OPEN. push (start, f[start]).

while OPEN is not empty
        n ← OPEN. pop()
        f in == goal :
            return reconstruct - path (n) ?

        CLOSED. add (n)

for each neighbour is expand (n) :
        if neighbour is closed :
            continue
        g = g[n] + 1

if neighbour not in OPEN :
        OPEN. push (neighbour, f[neighbour])
    return "No solution"
```


g = 1 h = 4 f = 5
g = 1 h = 2 f = 3
g = 1, h = 4, f = 5
g = 2 h = 1 f = 3
g = 2 f = 5 h = 3
g = 2 h = 3 f =
g = 3 f = 5 h = 2
~~g = f = h =~~
goal.
g = ? f = ?
h = ?

Code:

```
import heapq
def manhattan_distance(state, goal): distance = 0
for i in range(3): for j in range(3):
if state[i][j] != 0: value = state[i][j]
# Find the position of the value in the goal state for gi in range(3):
for gj in range(3):
if goal[gi][gj] == value: goal_pos = (gi, gj) break
else:
continue break
distance += abs(i - goal_pos[0]) + abs(j - goal_pos[1]) return distance

def get_neighbors(state): neighbors = []
for i in range(3): for j in range(3):
if state[i][j] == 0: x, y = i, j break
else:
continue break

moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
for dx, dy in moves:
nx, ny = x + dx, y + dy
if 0 <= nx < 3 and 0 <= ny < 3:

new_state = [list(row) for row in state]
new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y] neighbors.append(tuple(tuple(row) for row in
new_state))
return neighbors

def astar_search_manhattan(initial, goal):
frontier = [(manhattan_distance(initial, goal), 0, initial)] explored = set()
parent = {}
cost = {initial: 0}

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

if current == goal: path = []
while current in parent: path.append(current) current = parent[current]
path.append(initial) return path[::-1]

explored.add(current)

for neighbor in get_neighbors(current): new_cost = cost[current] + 1
if neighbor not in cost or new_cost < cost[neighbor]: cost[neighbor] = new_cost
priority = new_cost + manhattan_distance(neighbor, goal) heapq.heappush(frontier, (priority, new_cost, neighbor))
parent[neighbor] = current
return None

def get_state_input(prompt): print(prompt)
state = []
for _ in range(3):
row = list(map(int, input().split()))

state.append(row)
return tuple(tuple(row) for row in state)
```

initial_state_m = get_state_input("Enter the initial state for Manhattan distance (3 rows of 3 numbers separated by spaces, use 0 for the blank):")
goal_state_m = get_state_input("Enter the goal state for Manhattan distance (3 rows of 3 numbers separated by spaces, use 0 for the blank):")
path_m = astar_search_manhattan(initial_state_m, goal_state_m) if path_m:
print("Solution found using Manhattan distance:")
for step in path_m: for row in step:
print(row) print()
else:
print("No solution found using Manhattan distance.")

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm

→ HILL-CLIMB SEARCH:

function HILL-CLIMBING (problem) returns a state that is a local ~~minimum~~ maximum

current ← MAKE-NODE(problem.INITIAL-STATE)

loop do

neighbour ← a highest-valued successor of current

if neighbour.VALUE ≤ current.VALUE then return current.STATE

current ← neighbour.

OUTPUT:

Initial State : Cost = 2

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

Next State :

Cost = 1

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

Next state:

Cost = 0

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

Next State: Cost = 1

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

Solution found:
Cost = 0

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

8/10/24

Code:

```
import random

def cost(state):

attacking_pairs = 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):
attacking_pairs += 1
return attacking_pairs

def print_board(state):

n = len(state)
board = [['.' for _ in range(n)] for _ in range(n)] for i in range(n):
board[state[i]][i] = 'Q'

for row in board: print(" ".join(row))

def get_neighbors(state):

neighbors = [] n = len(state)
for i in range(n):
for j in range(i + 1, n): neighbor = list(state)
neighbor[i], neighbor[j] = neighbor[j], neighbor[i] neighbors.append(tuple(neighbor))
return neighbors

def hill_climbing(initial_state):

current = initial_state print(f"Initial state:") print_board(current) print(f"Cost: {cost(current)}") print('-' * 20)

while True:
neighbors = get_neighbors(current)

next_state = min(neighbors, key=lambda x: cost(x)) print(f"Next state:")
print_board(next_state)

print(f"Cost: {cost(next_state)}") print('-' * 20)

if cost(next_state) >= cost(current):

print(f"Solution found:") print_board(current) print(f"Cost: {cost(current)}") return current
current = next_state if  name  == " main ":
initial_state = (3, 1, 2, 0)
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

LAB - 4

1) N- Queens using Hill Climbing:

2) Simulated Annealing:

→ SIMULATED ANNEALING:

```
        curr ← Initial state
        T ← Large Positive value
while T>0 do
        next ← random neighbour of curr
        ΔE ← curr cost − next cost
    if ΔE >0 then
        curr ← next
    else
        curr ← event with p= e^(ΔE/T)
    end if
        decrease T
    end while
        return curr
```

Output:

Enter number of queens: 8

Solution found at step 623

Position format:

3 1 7 4 6 0 2 5

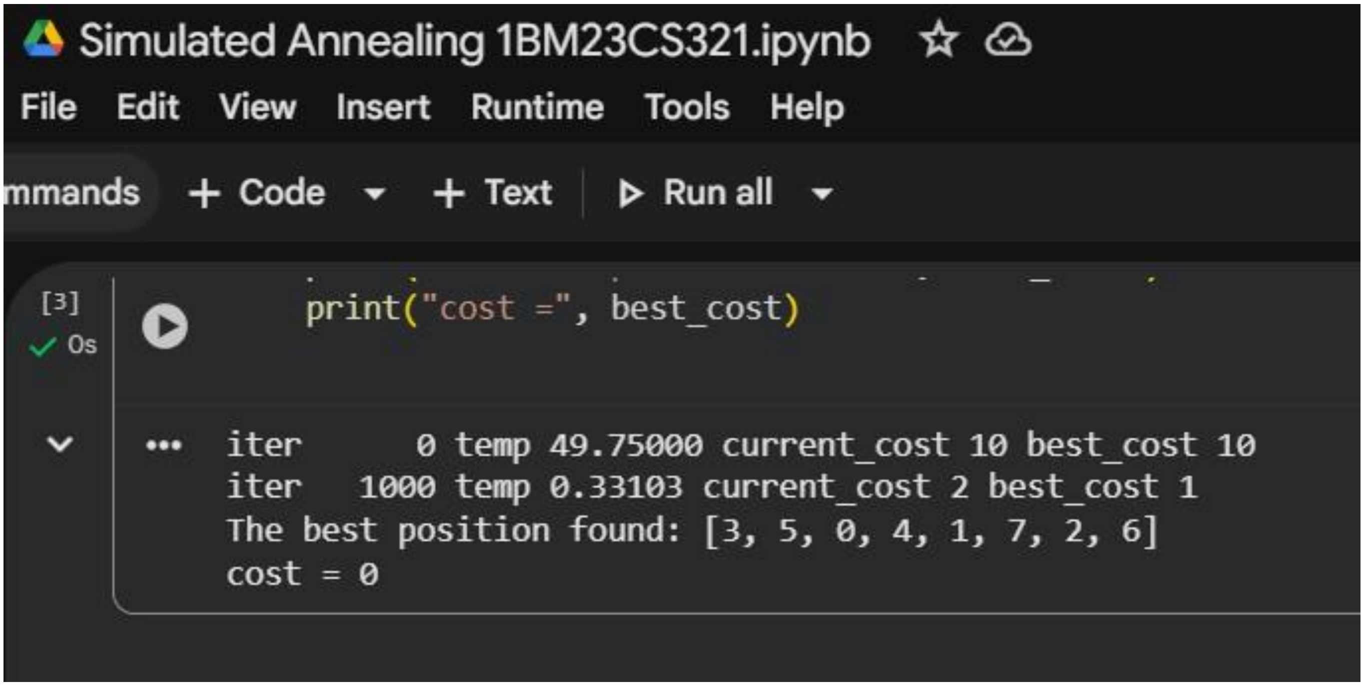
Heuristic 0

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:



Program 6

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

Algorithm:

WEEK → 6

Knowledge base using prepositional logic:

```
def entails (KB, Q):

prepositions = get_unique_prepositions (KB, Q)
truth_table = gen_truth_table (prepositions)

        for row in truth_table:
                if evaluate (KB, row):
                        if (!evaluate (Q, row)):
                                return false

return True
```

Q2

$Q \rightarrow P$

$P \rightarrow Q$

$Q \vee R$

i) Construct Truth Table

ii) Does KB entail R?

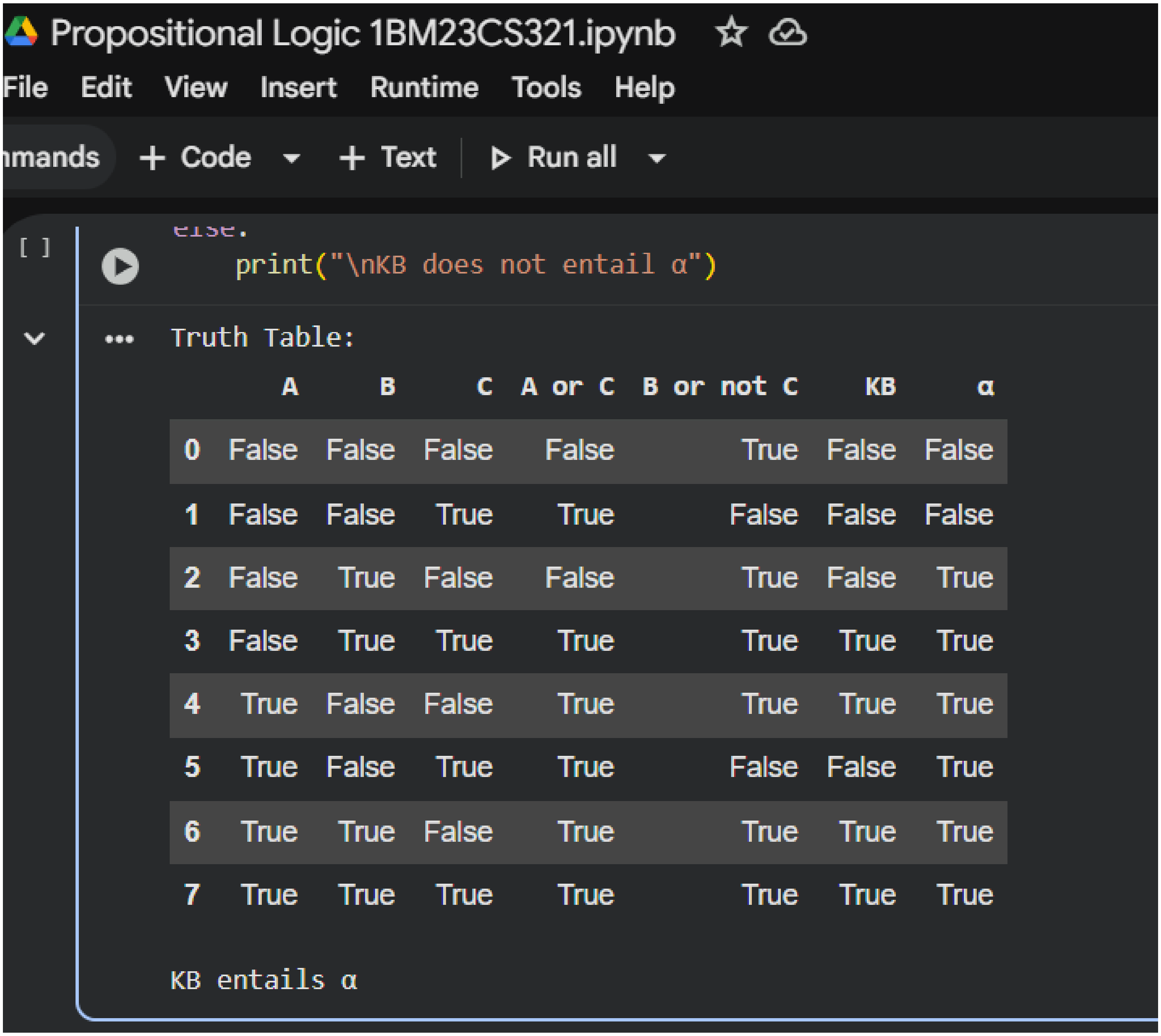
iii) Does KB entail $R \rightarrow P$?

iv) Does KB entail $Q \rightarrow R$?

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:



Propositional Logic 1BM23CS321.ipynb

File Edit View Insert Runtime Tools Help

mands + Code + Text ▷ Run all

```
else:
    print("\nKB does not entail α")
```

Truth Table:

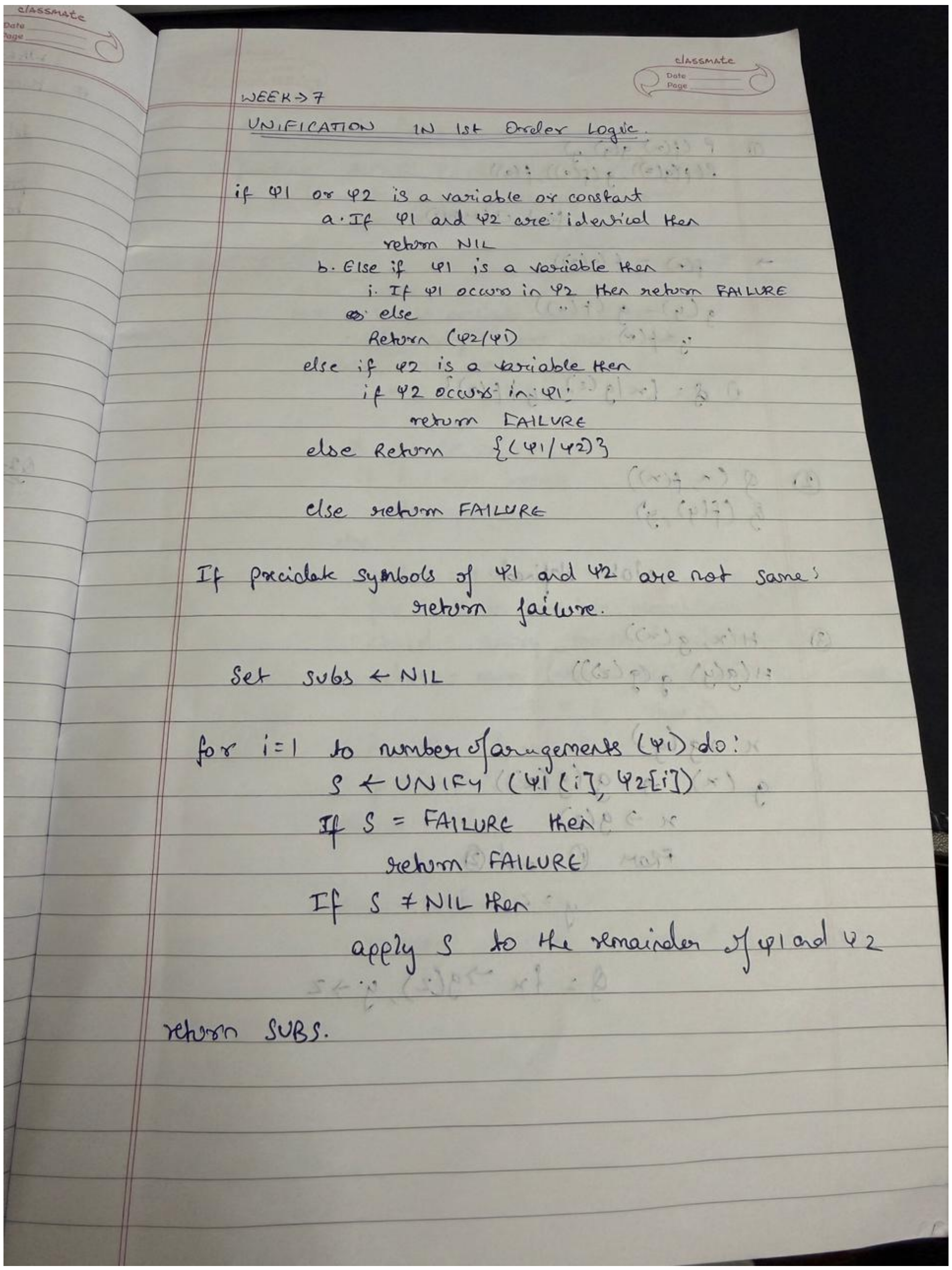
	A	B	C	A or C	B or not C	KB	α
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

KB entails α

Program 7

Implement unification in first order logic

Algorithm:



WEEK->7

UNIFICATION IN 1st Order Logic.

if Ψ1 or Ψ2 is a variable or constant
a. If Ψ1 and Ψ2 are identical then
return NIL
b. Else if Ψ1 is a variable then
i. If Ψ1 occurs in Ψ2 then return FAILURE
else
Return (Ψ2/Ψ1)
else if Ψ2 is a variable then
if Ψ2 occurs in Ψ1
return FAILURE
else Return {(Ψ1/Ψ2)}

else return FAILURE

If predicate symbols of Ψ1 and Ψ2 are not same
return failure.

Set subs ← NIL

for i=1 to number of arugements (Ψi) do:
S ← UNIFY (Ψi[i], Ψ2[i])
If S = FAILURE then
return FAILURE
If S ≠ NIL then
apply S to the remainder of Ψ1 and Ψ2

return SUBS.

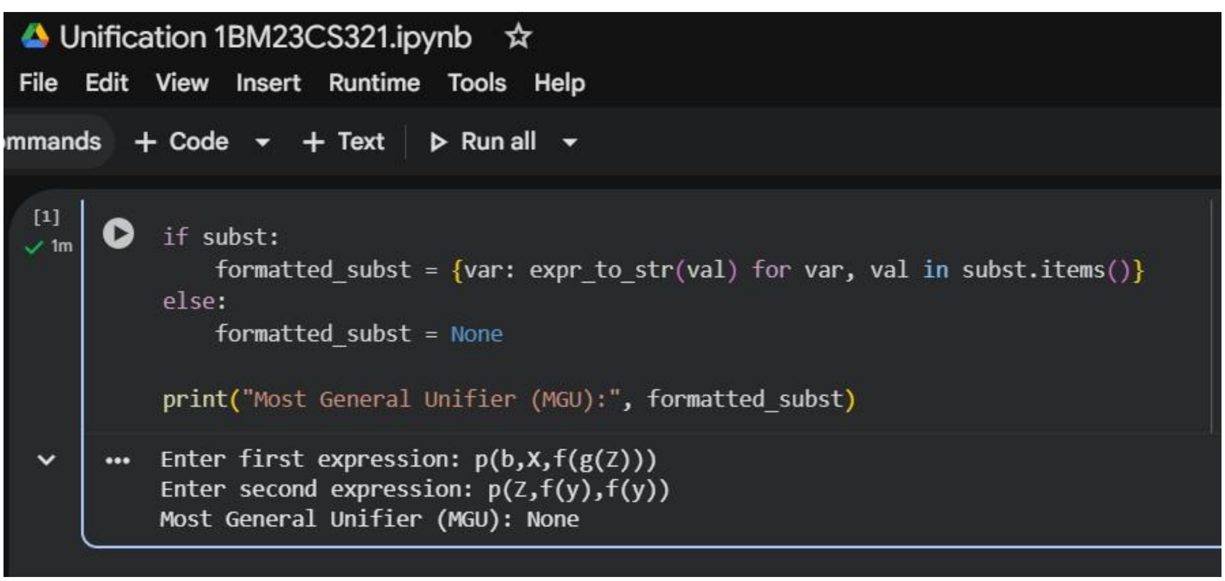
Code:

```
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)
```

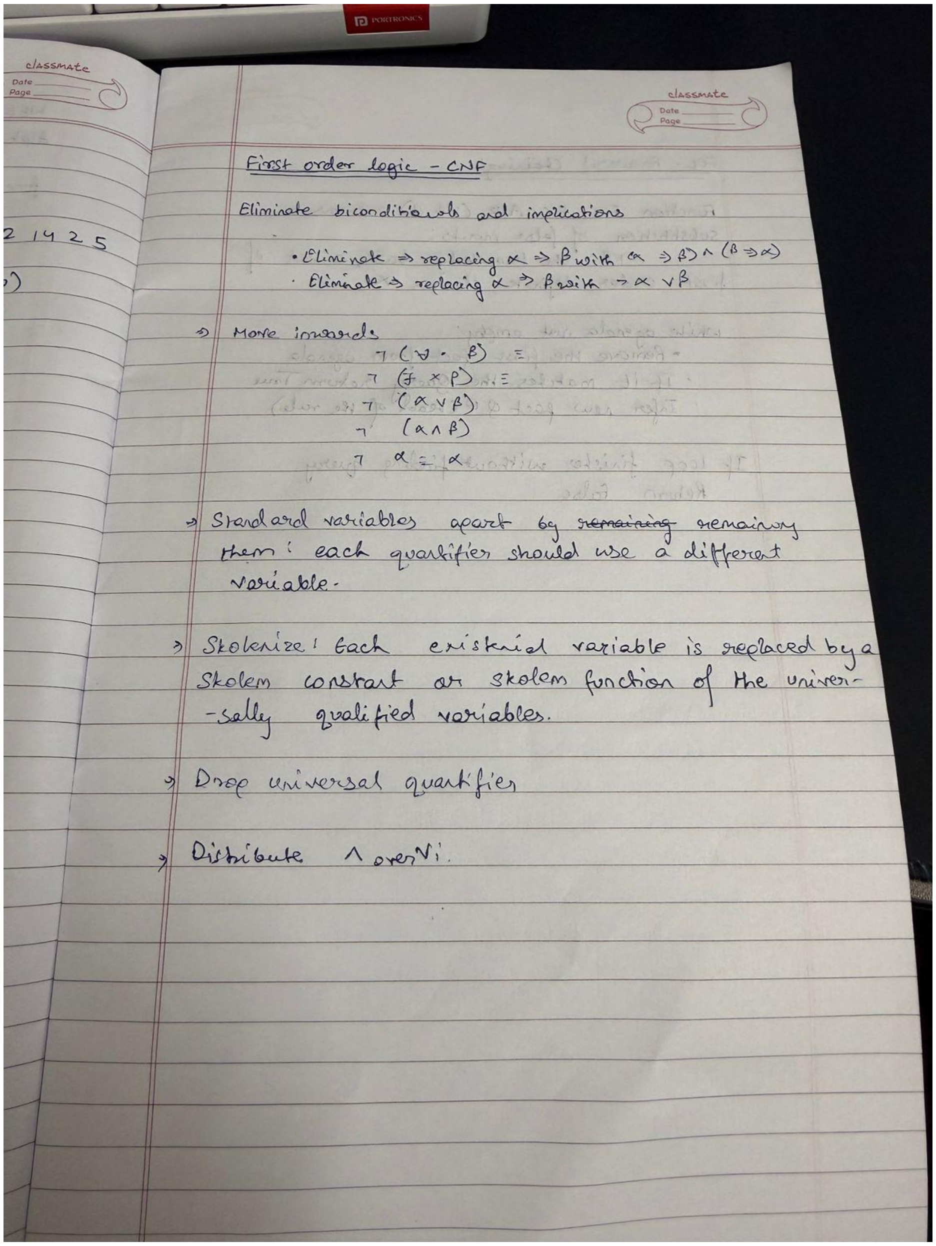
Output:



Program 8

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

Algorithm:



First order logic - CNF

Eliminate biconditionals and implications

- Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
- Eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$

⇒ Move inwards

$\neg(\forall x \, \beta) \equiv$

$\neg(\exists x \, \beta) \equiv$

$\neg(\alpha \vee \beta)$

$\neg(\alpha \wedge \beta)$

$\neg\neg\alpha \equiv \alpha$

⇒ Standard variables apart by ~~remaining~~ remaining them: each quantifier should use a different variable.

⇒ Skolemize: Each existential variable is replaced by a Skolem constant or skolem function of the universally qualified variables.

⇒ Drop universal quantifier

⇒ Distribute $\wedge$ over $\vee$.

Code:

```
facts = {
```

```
    'American(Robert)': True,
    'Hostile(A)': True,
    'Sells_Weapons(Robert, A)': True
}

If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
def forward_reasoning(facts):

 If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
    if facts.get('American(Robert)', False) and facts.get('Hostile(A)', False) and facts.get('Sells_Weapons(Robert,
A)', False):
        facts['Crime(Robert)'] = True

forward_reasoning(facts)

if facts.get('Crime(Robert)', False):
    print("Robert is a criminal.")
else:
    print("Robert is not a criminal.")
```

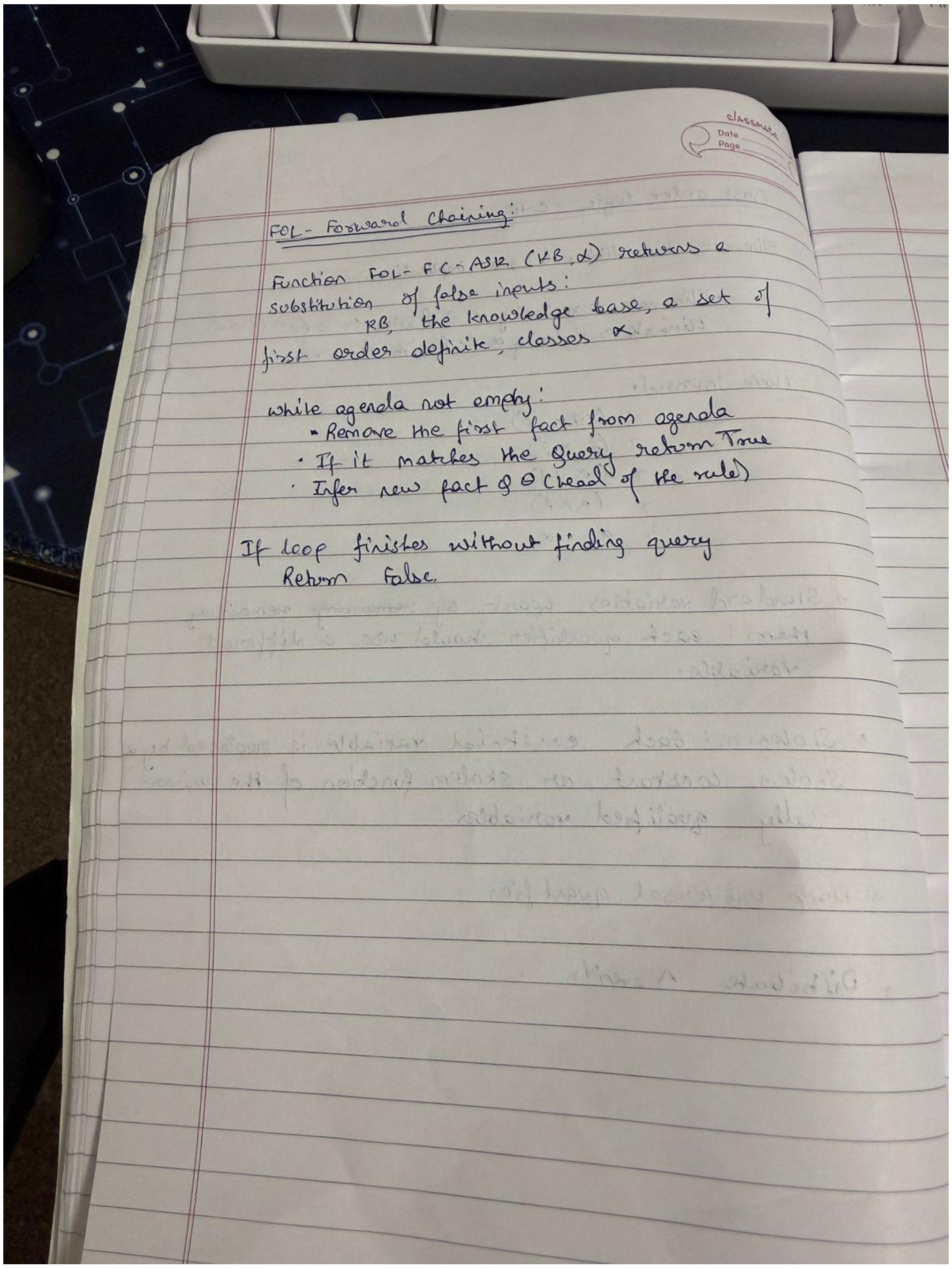
Output:

Robert is a criminal.

Program 9

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

Algorithm:



FOL- Forward Chaining:

Function FOL-FC-ASK (KB, α) returns a substitution of false inputs:
KB, the knowledge base, a set of first order definite classes α

while agenda not empty:
- Remove the first fact from agenda
- If it matches the Query return True
- Infer new fact $q\theta$ (head of the rule)

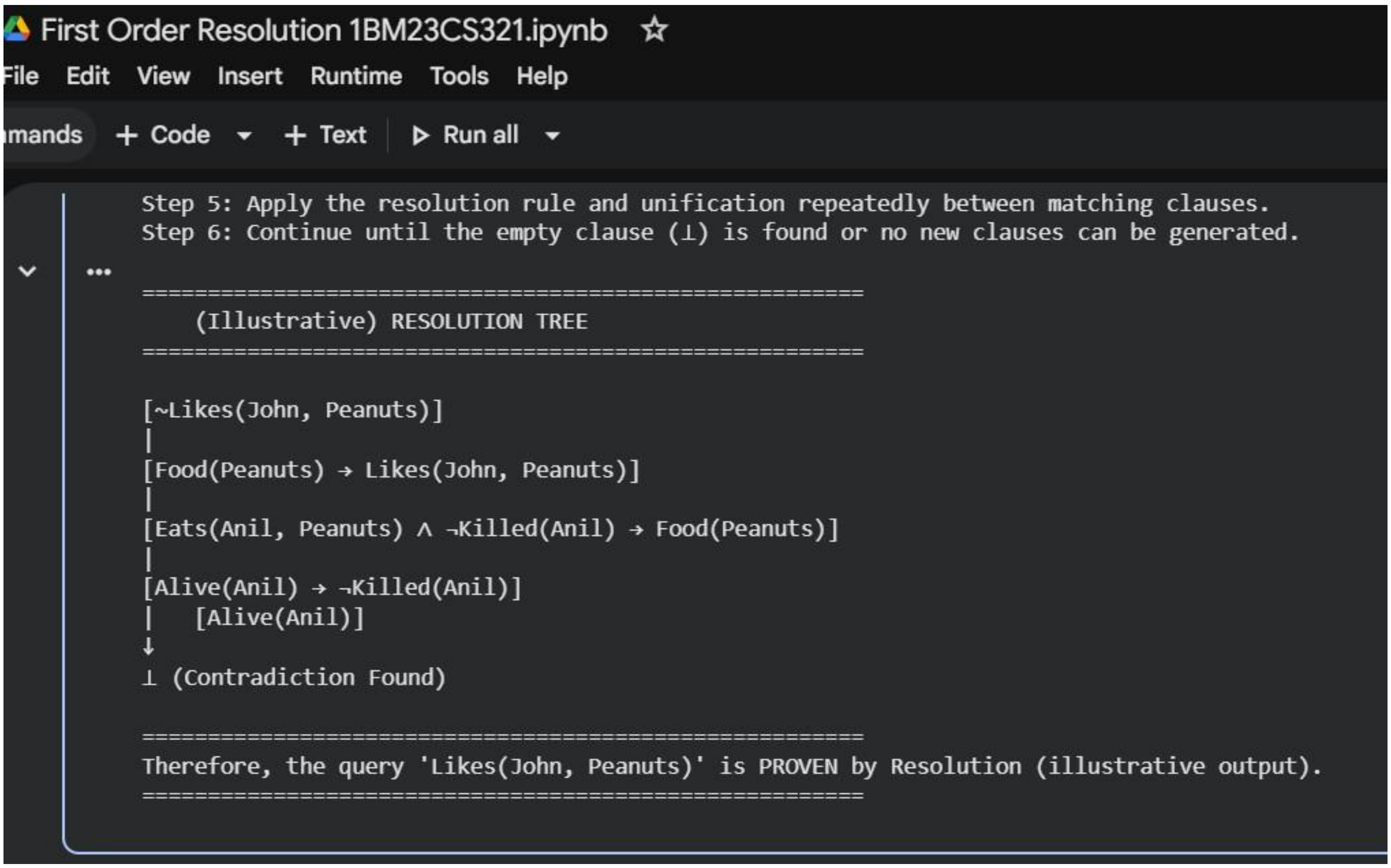
If loop finishes without finding query
Return False.

Code:

```
def fol_resolution(kb, query):
    print("\n" + "="*55)
    print("                KNOWLEDGE BASE")
    print("="*55)
    for i, clause in enumerate(kb, start=1):
        print(f"  {i}. {clause}")

    print("\n" + "="*55)
    print("                      QUERY")
    print("="*55)
    print(f"  Prove: {query}")
    print(f"  Negated Query: ~{query}\n")

    print("="*55)
    print("                RESOLUTION PROCESS")
    print("="*55)
    print("Step 1: Convert all implications (→) to CNF (Conjunctive Normal Form).")
    print("Step 2: Eliminate all universal quantifiers (∀).")
    print("Step 3: Add negated query (~Query) to the KB.")
    print("Step 4: Apply resolution rule between matching clauses.")
    print("Step 5: Continue until the empty clause (⊥) is found.\n")
    print("="*55)
    print("                RESOLUTION TREE")
    print("="*55)
    print("""
                [~Likes(John, Peanuts)]
                          |
                [Food(Peanuts) → Likes(John, Peanuts)]
                          |
                [Eats(Anil, Peanuts) ∧ ¬Killed(Anil) → Food(Peanuts)]
                          |
                [Alive(Anil) → ¬Killed(Anil)]
                          |
                [Alive(Anil)]
                          ↓
                  ⊥ (Contradiction Found)
    """)

    print("="*55)
    print(f" Therefore, the query '{query}' is PROVEN by Resolution.")
    print("="*55 + "\n")

print("\n FIRST ORDER LOGIC - RESOLUTION METHOD")

n = int(input("Enter the number of statements in the Knowledge Base: "))

kb = []
print("\nEnter each statement (e.g., '∀x: Food(x) → Likes(John, x)'):")
for i in range(n):
    stmt = input(f"KB[{i+1}]: ")
```

```
    kb.append(stmt)

query = input("\nEnter the query to prove: ")

fol_resolution(kb, query)
```

Output:

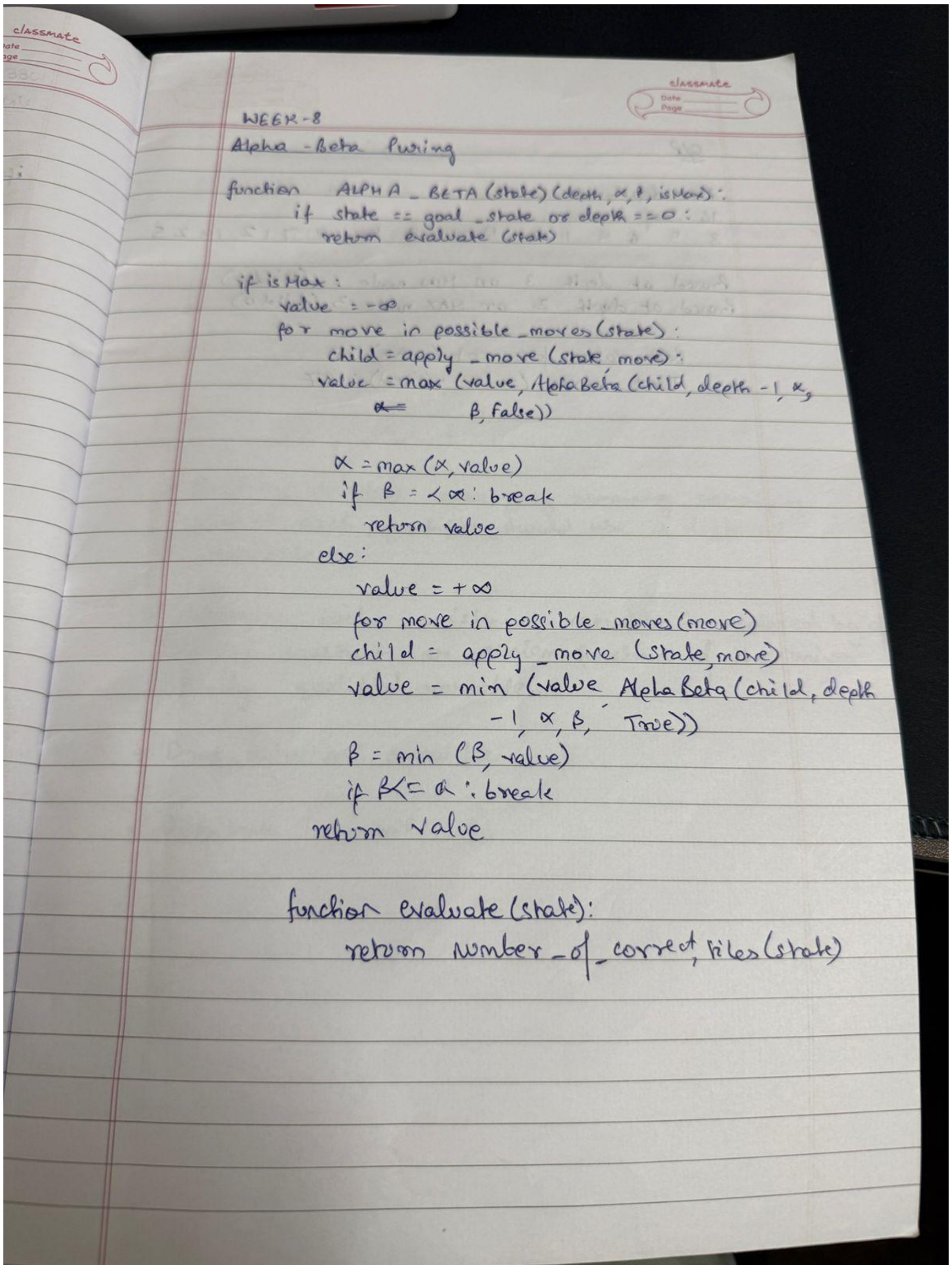



```
Step 5: Apply the resolution rule and unification repeatedly between matching clauses.
Step 6: Continue until the empty clause (⊥) is found or no new clauses can be generated.

=======================================================
    (Illustrative) RESOLUTION TREE
=======================================================

[~Likes(John, Peanuts)]
|
[Food(Peanuts) → Likes(John, Peanuts)]
|
[Eats(Anil, Peanuts) ∧ ¬Killed(Anil) → Food(Peanuts)]
|
[Alive(Anil) → ¬Killed(Anil)]
|   [Alive(Anil)]
↓
⊥ (Contradiction Found)

=======================================================
Therefore, the query 'Likes(John, Peanuts)' is PROVEN by Resolution (illustrative output).
=======================================================
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



WEEK-8

Alpha-Beta Puring

```
function ALPHA_BETA (state) (depth, α, β, isMax):
    if state == goal_state or depth == 0:
        return evaluate (state)

if is Max:
    value = -∞
    for move in possible_moves (state):
        child = apply_move (state move):
        value = max (value, AlphaBeta (child, depth -1, α,
                β, False))

        α = max (α, value)
        if β = < α: break
        return value
else:
    value = +∞
    for move in possible_moves (move)
    child = apply_move (state, move)
    value = min (value AlphaBeta (child, depth
                -1, α, β, True))
    β = min (β, value)
    if β <= α: break
return value

function evaluate (state):
    return number_of_correct_tiles (state)
```

Code:

```
move_count = 0

def alpha_beta(depth, node_index, is_maximizing, values, alpha, beta,
  max_depth): global move_count
  move_count += 1

  if depth ==
    max_depth: return
    values[node_index
    ]

  if
    is_maxim
    izing:
    best =
    float('-
    inf')
    for i in range(2):  # binary tree
      val = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta,
      max_depth) best = max(best, val)
      alpha =
      max(alpha, best)
      if beta <= alpha:
        print(f" Pruned at depth {depth} on MAX node
        {node_index}") break
    return best
  else:
    best =
    float('inf')
    for i in
    range(2):
```

```
        val = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta,
        max_depth) best = min(best, val)
        beta =
        min(beta, best)
        if beta <=
        alpha:
            print(f" Pruned at depth {depth} on MIN node
            {node_index}") break
    return best

max_depth = int(input("Enter the maximum depth of the tree:

")) num_leaves = 2 ** max_depth
print(f"Enter {num_leaves} leaf node values separated by spaces:")
values = list(map(int, input().split()))

if len(values) != num_leaves:
    print(" Error: Number of values does not match
2^depth.") else:
    move_count = 0
    best_value = alpha_beta(0, 0, True, values, float('-inf'), float('inf'), max_depth)
    print("\n Best value for root (MAX):", best_value)
    print(f" Total moves (nodes visited): {move_count}")
```

Output: