VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

SHREYAS SINHA (1BM23CS321)

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 **SHREYAS SINHA (1BM23CS321),** 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 Incharge Name Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

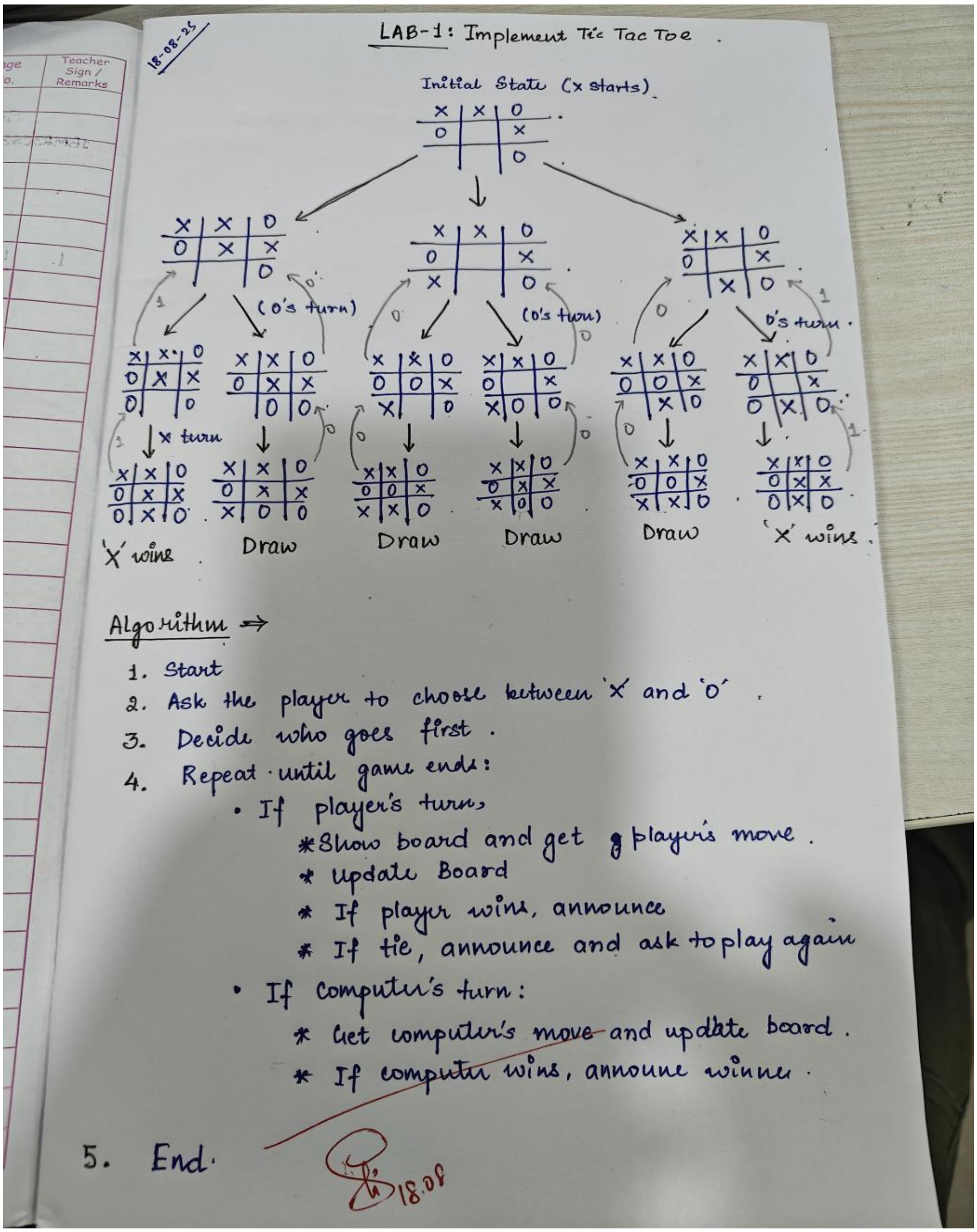
Github Link: https://github.com/Shreyas-2607/AI_LAB

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



Algorithm ⇒

1. Start
2. Ask the player to choose between 'X' and 'O'.
3. Decide who goes first.
4. Repeat until game ends:
   - If player's turn,
     * Show board and get player's move.
     * Update Board
     * If player wins, announce
     * If tie, announce and ask to play again
   - If Computer's turn:
     * Get computer's move and update board.
     * If computer wins, announce winner.
5. End.

18.08

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–II : VACCUM CLEANER AGENT

(#) Algorithm ⇒

① Two Room setup:

(i) Start .

(ii) Implement initial state with dust and vaccum cleaner with 2 room setup.

(iii) If vaccum cleaner is in room 'A', and dust is present suck it.

(iv) After cleaning A, ask user to move to room 'B' and clean the dust in B.

(v) Then move the cleaner back to A.

(vi) End.

~~② 4 room setup~~:

② 4 – room setup:

(i) Initialize

(ii) Start at R1 and move through rooms in a specific path.

(iii) If R1 is not clean, clean the dust and then ask user for the next room.

(iv) Move to either R2 or R3 and then clean the dust in that room.

(v) Then repeat the process so that all the rooms are clean and the objective is achieved.

(vi) End the process.

25/8/25

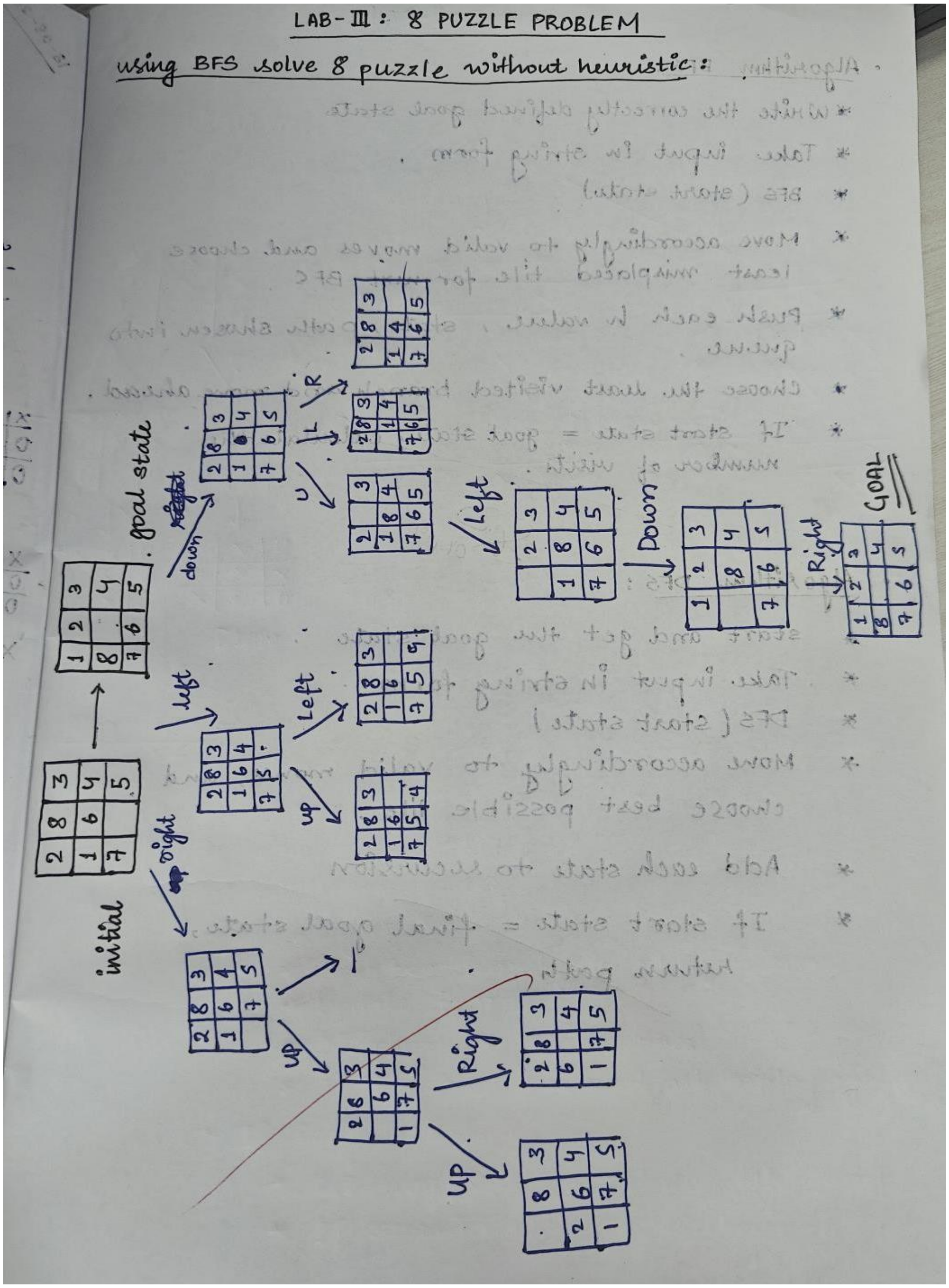
Code:

```
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)
```

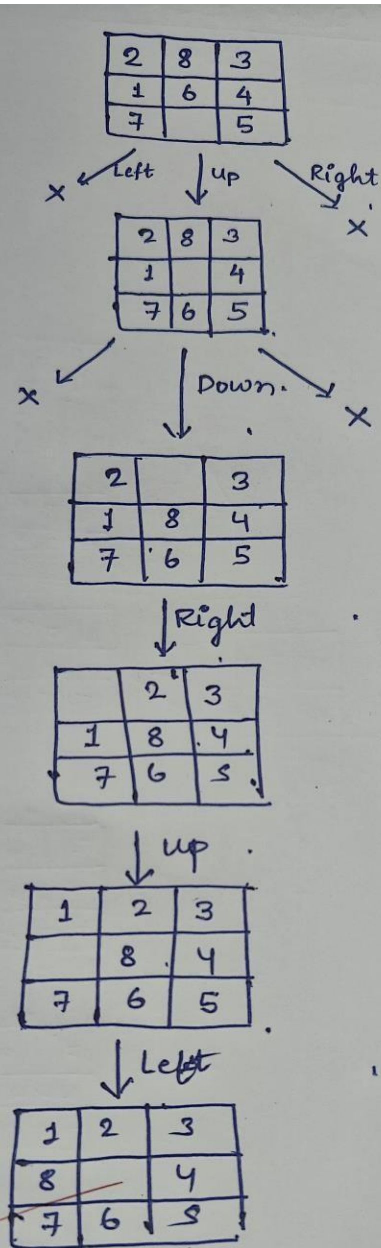
Program 2

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

Algorithm:



DFS solution :



01.09

- **Algorithm BFS :**

  - Write the correctly defined goal state
  - Take input in string form .
  - BFS (start state)
  - Move accordingly to valid moves and choose least misplaced tile for next BFC .
  - Push each h value , state , path chosen into queue .
  - Choose the least visited branch and move ahead .
  - If start state = goal state, calculate the number of visits .

01.09

- **Algorithm DFS :**

  - start and get the goal state .
  - Take input in string form .
  - DFS (start state)
  - Move accordingly to valid moves, and choose best possible tile .
  - Add each state to recursion
  - If start state = final goal state, return path .

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:

8-9-25

LAB IV - A* ALGORITHM

1

Misplaced Tiles

Manhattan Distance.

2	8	3
1	6	4
7		5

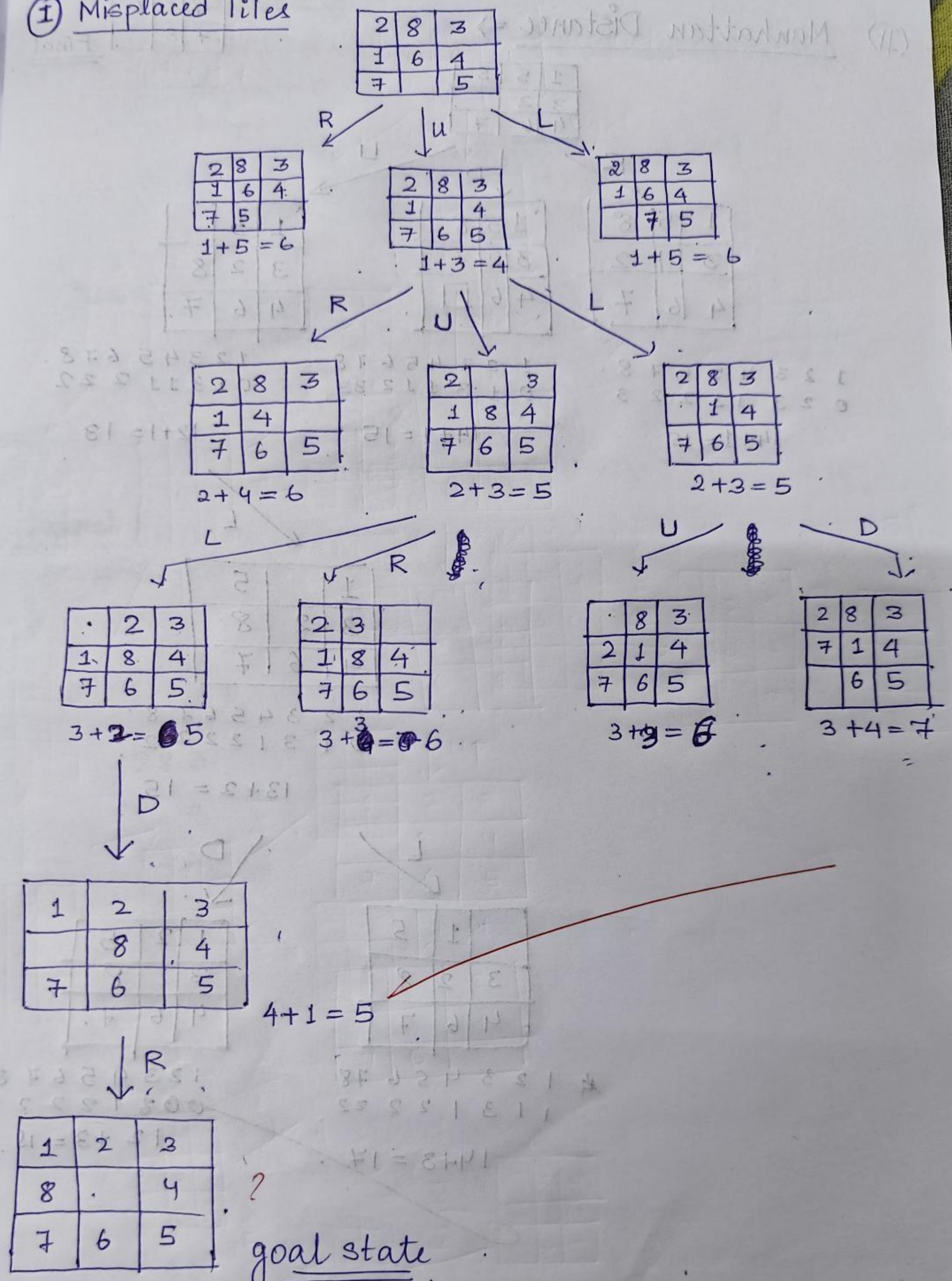
I

1	2	3
8		4
7	6	5

F

$$\boxed{f(n) = g(n) + h(n)}$$

① Misplaced Tiles



(II) Manhattan Distance ⇒

1	2	3
4	5	6
7	8	

Final

1	5	8
3	2	
4	6	7

R

1	5	8
3		2
4	6	7

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

14+1 = 15

D

1	5	8
3	2	7
4	6	

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

14+1 = 15

U

1	5	
3	2	8
4	6	7.

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

12+1 = 13

L.

1		5
3	2	8.
4	6	7

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

13+2 = 15

L

	1	5
3	2	8
4	6	7

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

14+3 = 17.

D.

1	2	5
3		8
4	6	7.

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

12 +3 = 15

Code:

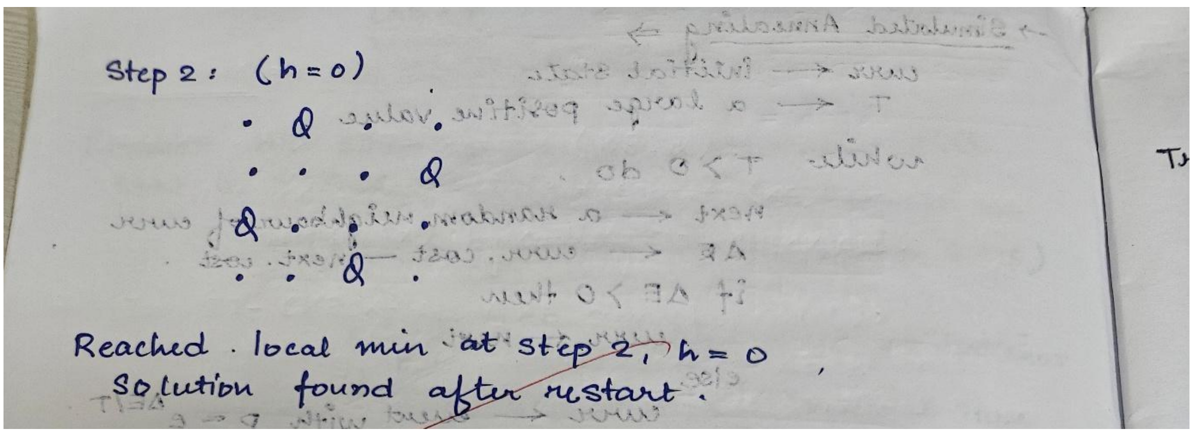
```
import heapq
def manhattan_distance(state, goal): distance = 0
for i in range(3): for j in range(3):
if state[i][j] != 0: value = state[i][j]
# Find the position of the value in the goal state for gi in range(3):
for gj in range(3):
if goal[gi][gj] == value: goal_pos = (gi, gj) break
else:
continue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



Step 2: (h=0)

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

Reached local min at step 2, h = 0
Solution found after restart.

LAB - V

Hill climbing Search Algorithm.

Function Hill climbing (problem) returns a state that is a local maximum

curr ← MakeNode (problem INITIAL-STATE)

loop do

neighbours ← a highest valued successor

if neighbour value ≤ curr. value then

return curr. state.

curr ← neighbour

- $x_0 = 3, \; x_1 = 1, \; x_2 = 2, \; x_3 = 0$

cost = 2.

			Q
	Q		
		Q	
Q			

- $x_0 = 1, \; x_1 = 0, \; x_2 = 3, \; x_3 = 2$.

cost = 2+1+1 = 4

	Q		
Q			
			Q
		Q	

- $x_0 = 1, \; x_1 = 3, \; x_2 = 0, \; x_3 = 2$.

cost = 0.

	Q		
			Q
Q			
		Q	

- $x_0 = 3, \; x_1 = 2, \; x_2 = 0, \; x_3 = 1$

cost = 2.

		Q	
Q			
			Q
	Q		

Output ⇒

Enter no. of queens : 8 .

Solution found at step 623 .

Position format :

[illegible] 3 1 7 4 6 0 2 5

Hueristic 0 .

Output :-

Enter the no queens (N) : 4

Initial state (hueristic 4) :

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

Step 1 : (h = 1)

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

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:

→ Simulated Annealing ⇒

curr ← Initial state
T ← a large positive value

while T > 0 do .
  next ← a random neighbour of curr
  ΔE ← curr. cost — next. cost
  if ΔE > 0 then
    curr ← next
  else
    curr ← next with $p = e^{\Delta E/T}$
  end if .
  decrease T.
end while
return curr.

Output ⇒

Enter no. of queens : 8
Solution found at step 623.
Position format :
3 1 7 4 6 0 2 5

Hueristic 0 .

Output :=

Enter the no queens (N): 4
Initial state (hueristic 4) :

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

Step 1 : (h = 1)

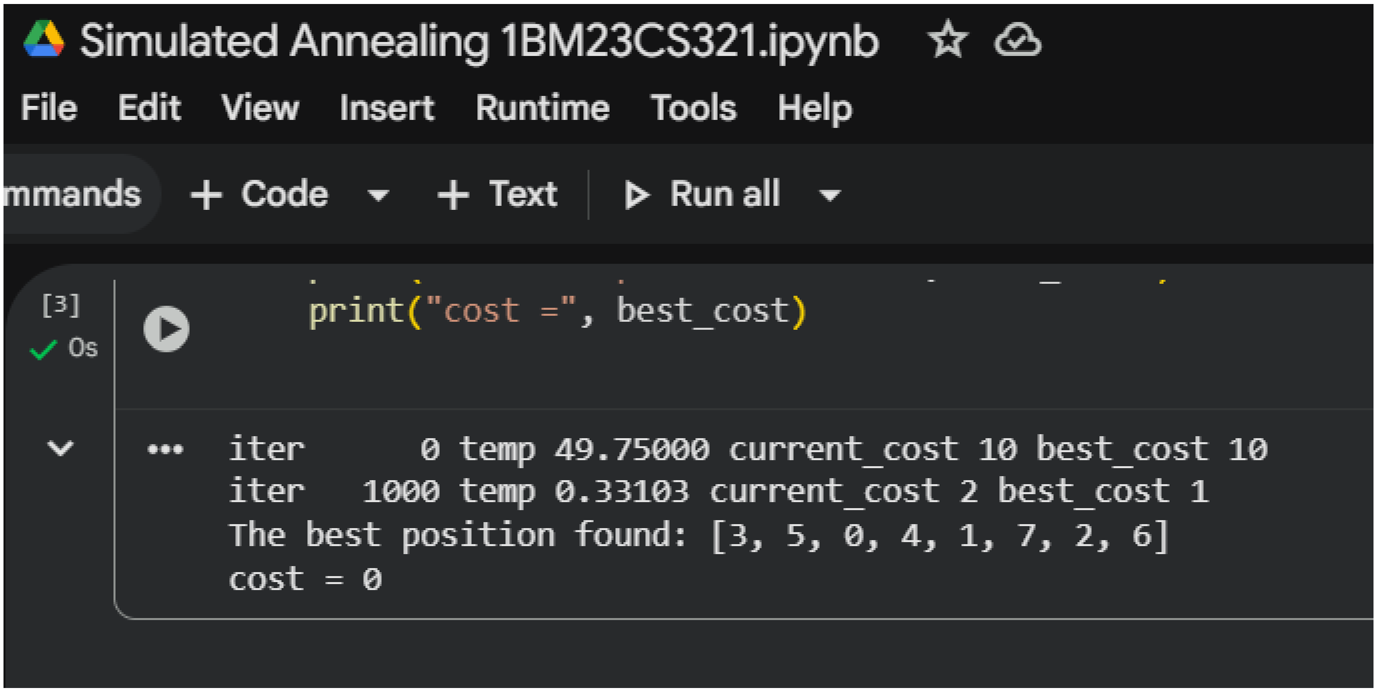
Q . . .
. . . Q
Q . . .
. . Q .

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:

LAB – VI

Propositional Logic.

Truth table for connectives.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Leftrightarrow Q$
False	False	True	False	false	true
False	True	True	false	true	false
True	False	false	false	true	false
True	True	false	true	true	true

Proposition al Inference : Enumeration Method ⇒

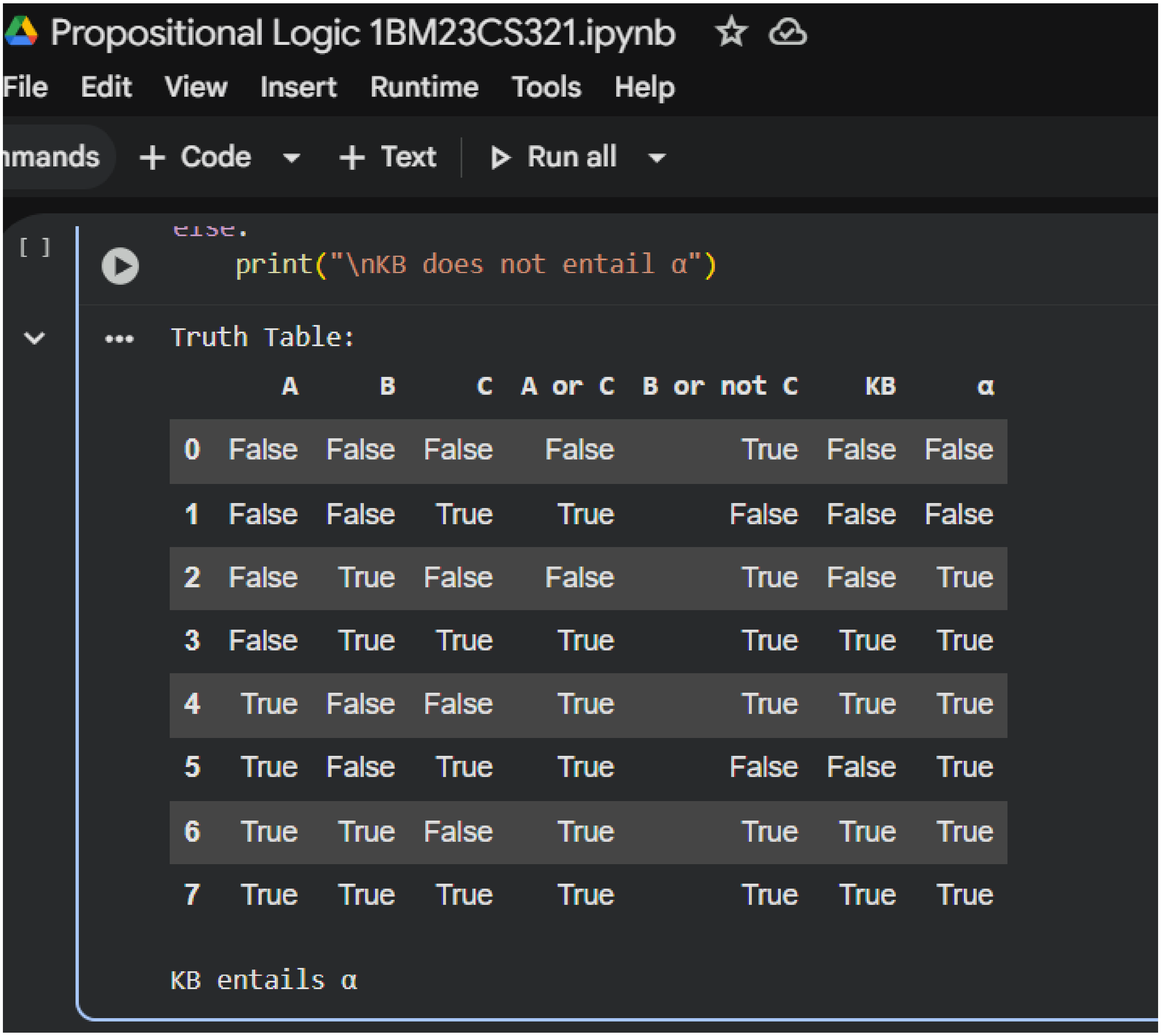
$\alpha = A \vee B$. $KB = (A \vee C) \wedge (B \vee \neg C)$.

A	B	C	$A \vee C$	$B \vee \neg C$	KB	α
False	False	False	False	True	False	False
False	False	True	True	false	false	false
False	True	False	False	True	false	True
False	True	True	True	True	True	True
True	False	False	True	True	True	True
True	False	True	True	false	False	True
True	True	False	True	True	True	True
True.	True	True,	True	True	True	True

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:

LAB - 7 : Unification

Algorithm ⇒

(i) If Ψ_1 and Ψ_2 is a variable constant, then :

(a) If Ψ_1 or Ψ_2 are identical, then return NIL .

(b) Else if Ψ_1 is a variable,

* then if Ψ_1 occurs in Ψ_2 then return FAILURE .
* Else return $\{(\Psi_2/\Psi_1)\}$.

(c) Else if Ψ_2 is a variable,

* If Ψ_2 occurs in Ψ_1, then return FAILURE .
* Else return $\{(\Psi_2/\Psi_1)\}$

(d) Else return FAILURE

(ii) If the initial predicate symbol in Ψ_1 & Ψ_2 are not the same, then return Failure.

(iii) If Ψ_1 and Ψ_2 have different number of args, then return FAILURE .

(iv) Set substitution set (SUBST) to NIL .

(v). For $i = 1$ to the number of elements in Ψ_1

(a) Call unify function with the i^{th} element of Ψ_1 and i^{th} element of Ψ_2 & put the result into s.

(b) If s = failure → return FAILURE

(c). If $s \neq$ NIL , then do

* Apply 's' to the remainder of both L1 & L2 .
* SUBSET = APPEND (s, SUBSET) .

(vi) Return SUBST .

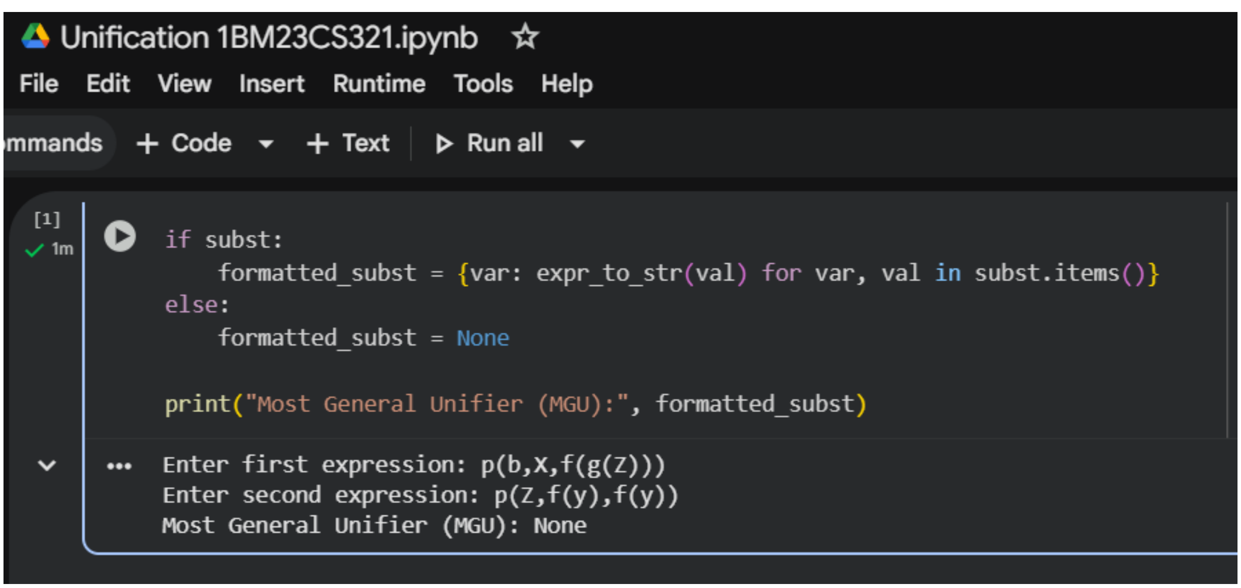
Code:

```
def occurs_check(var, term, subst):
    if var == term:
```

```python
        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)
```

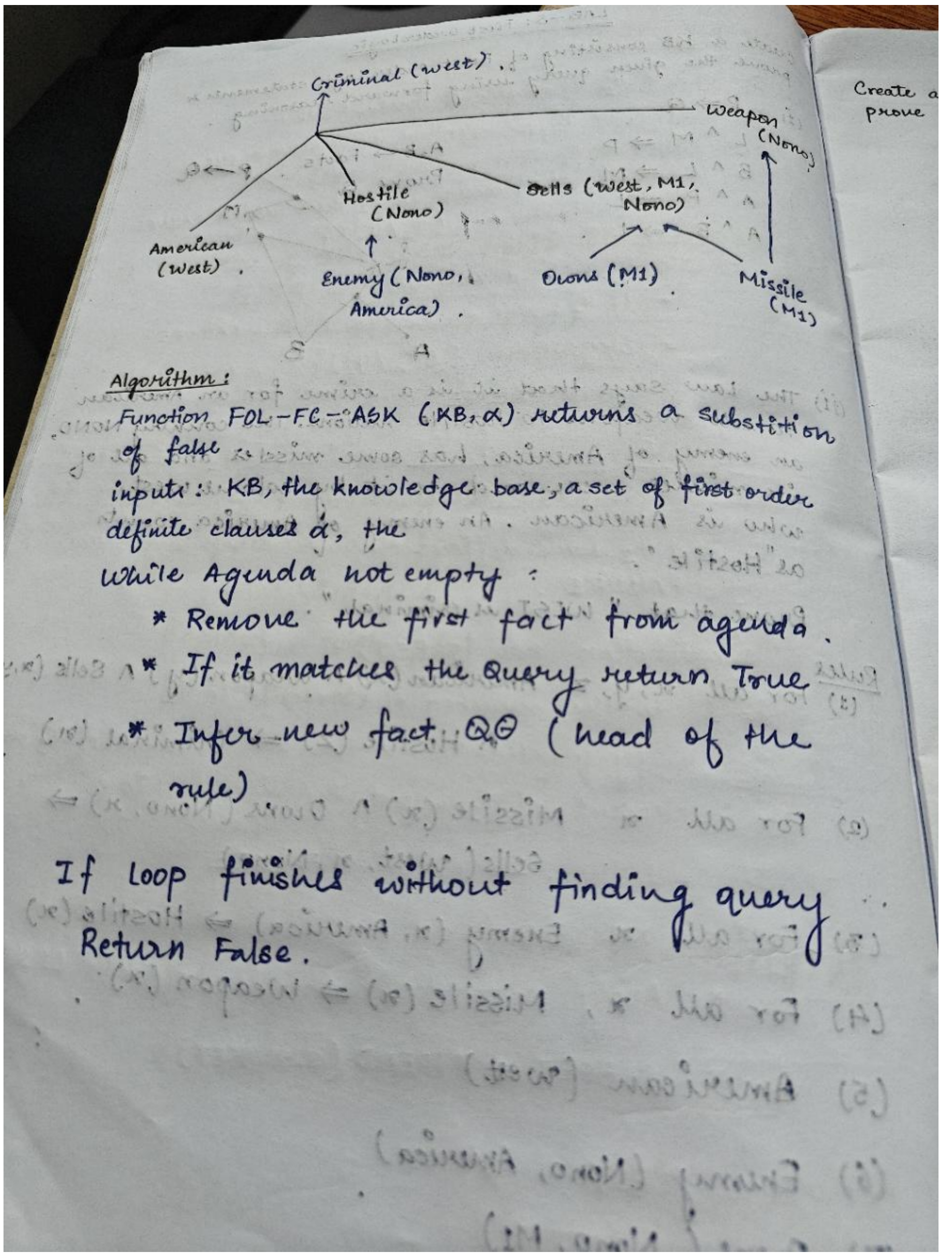
Output:



Program 8

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

Algorithm:



Code:

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:

LAB-9:

Create a KB consisting of first order logic statements & prove the given query using Resolution.

Basic steps for proving a conclusion S given premises:

Premise$_1$, ---, Premise$_n$

all expressed in FOL.

Algorithm ⇒

(i) Convert all sentences to CNF

(ii) Negate conclusion S and convert result to CNF.

(iii) Add negated contradiction S to the premise clauses.

(iv) Repeat until contradiction or no progress is made:

(a) Select 2 clauses

(b) Resolve them together, performing all required unifications.

(c) If the resolvent is the empty clause, a contradiction has been found

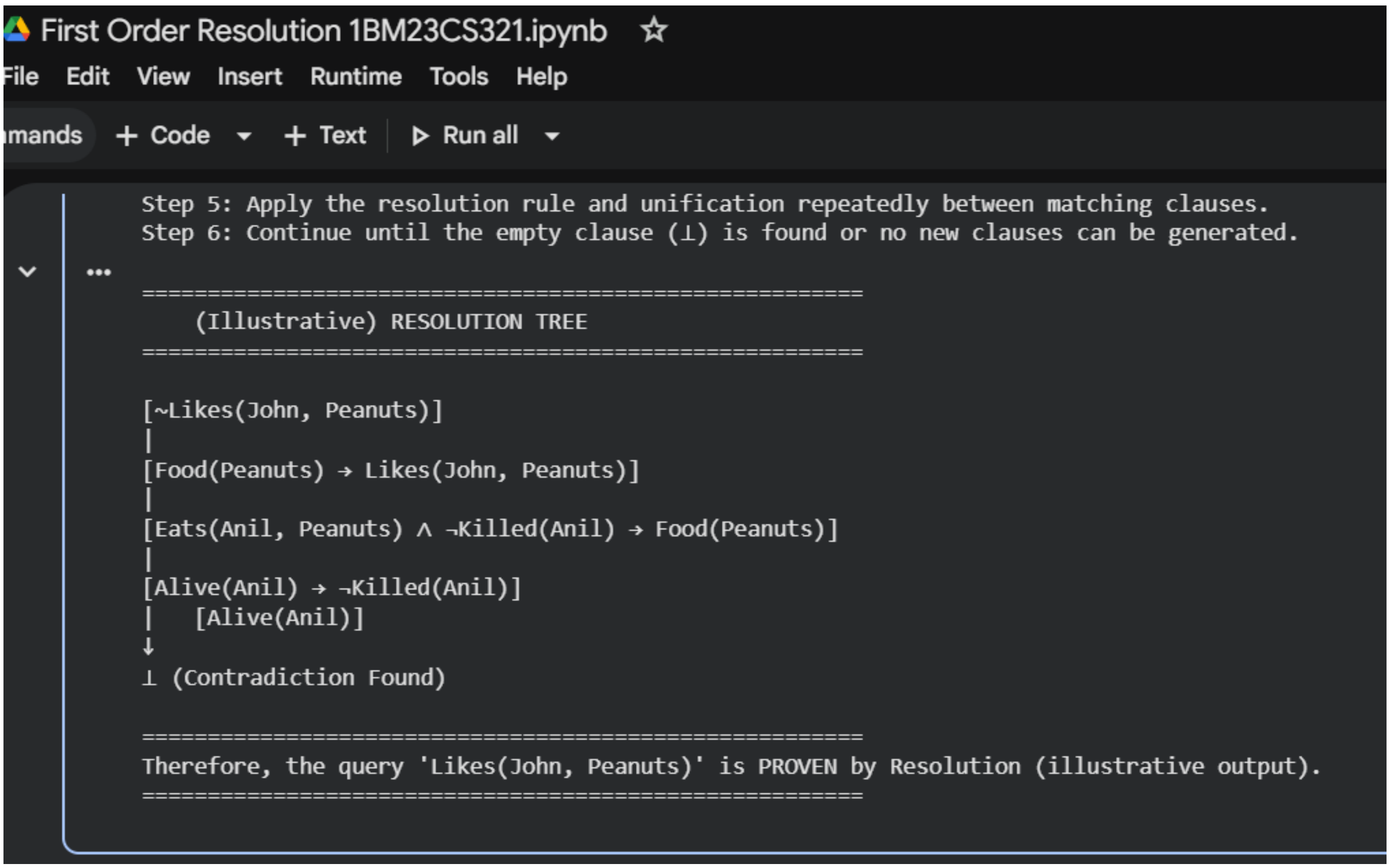
(d) If not, add resolvent to the premises.

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:



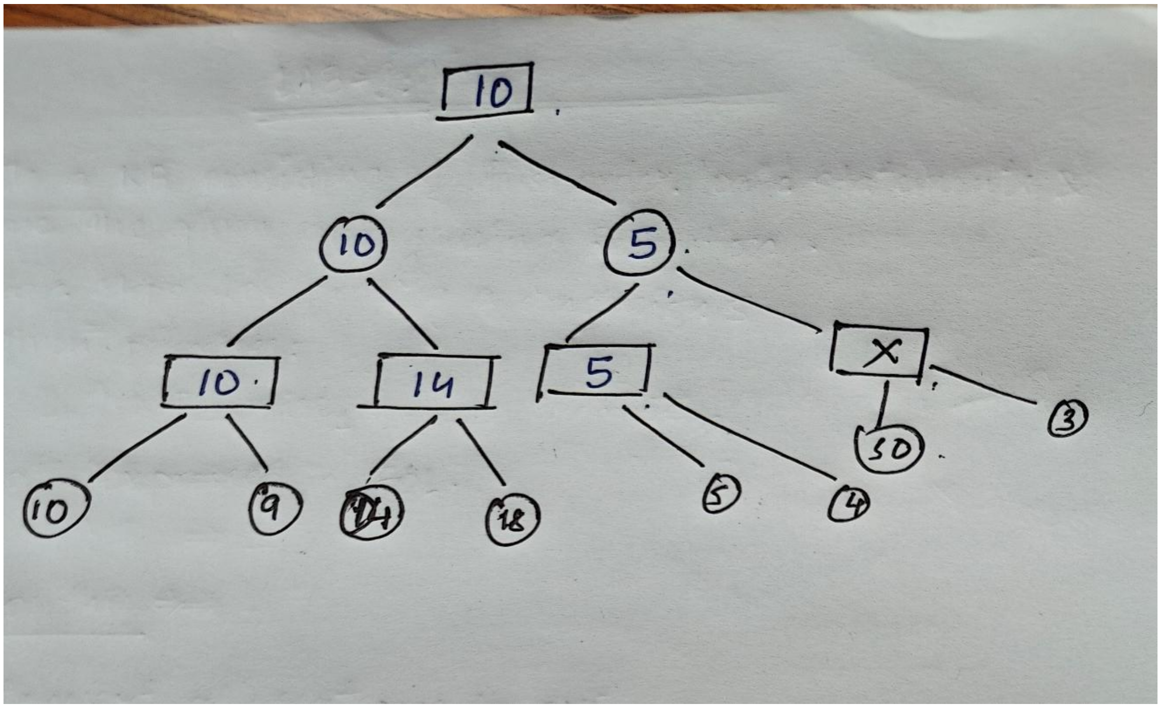
Program 10

Implement Alpha-Beta Pruning.

Algorithm:

LAB - 10: Alpha Beta Pruning ,

```
function· ALPHA- BETA (state):
        value = MAX- value (state, -∞, ∞).
        return the action from ACTIONS (state) that
        produced value.

function MAX- VALUE (state, α, β):
        if TERMINAL-TEST (state):
                return UTILITY (state)

        value (-∞).

        for each action in ACTIONS( state):
        value = max (value, MIN-VALUE, α, β))

        if value >= β:
                return value.

        α = max (α, value)

return value.

function MIN_ VALUE (state, α, β):
        if TERMINAL_TEST (state):
                return UTILITY (state).

value = + ∞ .

for each action in ACTIONS (state):
        value = minValue , MAX_VALUE).
        if value <= α:
                return value.

        β = min (β, value).

return value,
```



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: