VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT on

Artificial Intelligence (23CS5PCAIN)

Submitted by

JAWIN ROYS FERNANDES (1BM23CS122)

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 **JAWIN ROYS FERNANDES (1BM23CS122),** 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.

Sreevidya B S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

9	29-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	29-31
10	12-11-2025	Implement Alpha-Beta Pruning.	32-34

INDEX

5 'C'

Name Jawin Roys Fernandes ~~Std~~ Sec

Roll No CS122 Subject AI Lab School/College

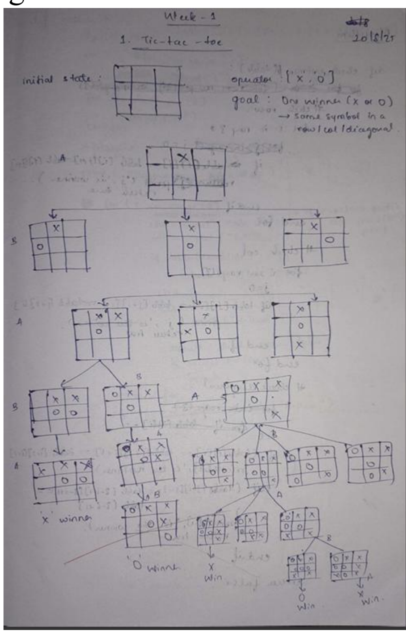
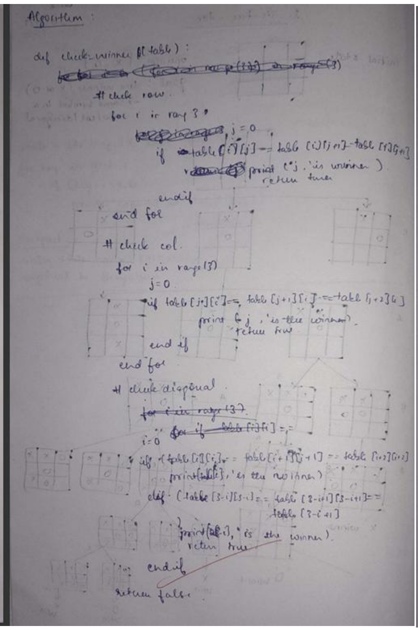
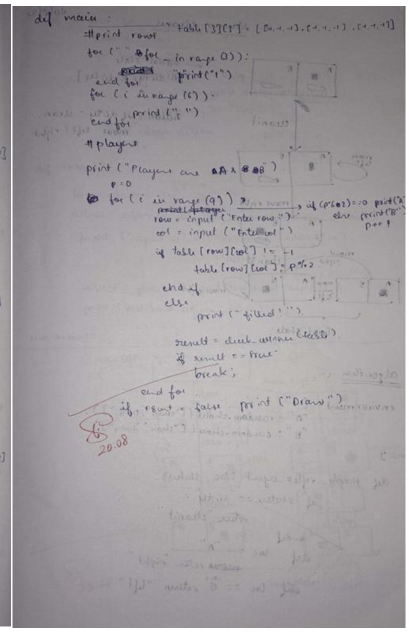
School/College Tel. No. Parents Tel. No.

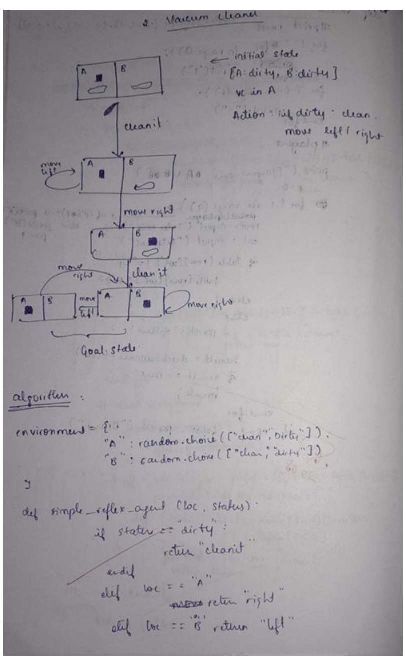
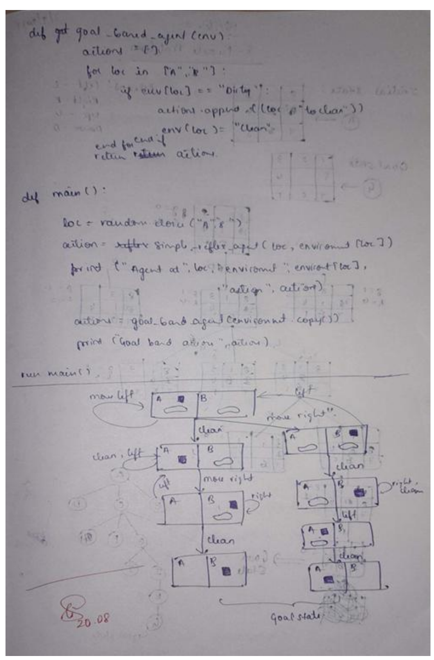
Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1	20/08/25	Tic-Tac-Toe & vacum agen		9
2	03/09/25	8 - Puzzle		10
3	10/09/25	A* on 8-Puzzle		10
4	17/09/25	Hill Climbing to solve N queens		10
5	17/09/25	Simulated Annealing to solve N Queens		10
6	08/10/25	Propositional Logic		10
7	15/10/25	Unification in FOL		6x2
8	15/10/25	propositional FOL / Forward chainin		6x2
9.	29/10/25	Resolution in FOL		10
10	29/10/25	Alpha beta Pruning		10

217

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

Algorithm:

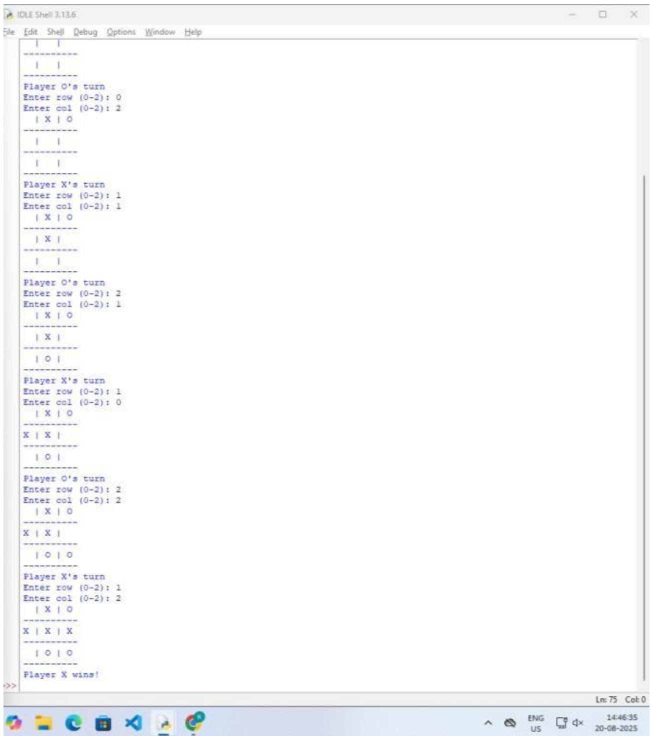




Code: #tic_tac_toe def
print_board(board):   for row in board:
print(" | ".join(row))   print("--" * 5)

```
def check_winner(board, 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)) or \       all(board[i][2 - i] ==
player for i in range(3)):
        return True

    return False

def tic_tac_toe():     board = [[" " for _ in range(3)]
for _ in range(3)]     players = ["X", "O"]
moves = 0

    while          moves          <          9:
print_board(board)             player =
players[moves              %              2]
print(f"Player {player}'s turn")

        row = int(input("Enter row (0-2): "))
col = int(input("Enter col (0-2): "))

        if board[row][col] == " ":
board[row][col] = player             moves += 1
else:              print("Cell already taken, try
again!")             continue

        if check_winner(board, player):
print_board(board)
print(f"Player {player} wins!")
return

    print_board(board)
    print("It's a draw!")

# Run the game  tic_tac_toe()
```

```
#vaccum cleaner  import
random

# Environment: 2 rooms A and B, both start dirty environment =
{
   "A": "Dirty",
   "B": "Dirty"
}

# Simple Reflex Agent def
simple_reflex_agent(location, status):     if
status == "Dirty":
      return "Cleanit please"
elif location == "A":
return "Right"
else:
      return "Left"

# Goal-Based Agent def
goal_based_agent(env):     actions = []     for
location in ["A", "B"]:        if env[location] ==
"Dirty":            actions.append((location,
"toClean"))            env[location] = "Clean"
return actions

# Simulation def run_simulation():
   print("Initial Environment:", environment)

   # Reflex agent      location = random.choice(["A", "B"])      action =
simple_reflex_agent(location, environment[location])      print(f"Reflex Agent at
{location} sees {environment[location]} -> Action: {action}")

   #   Goal-based   agent            actions   =
goal_based_agent(environment.copy())
print("Goal-Based Agent Actions:", actions)
run_simulation()  Output:
```



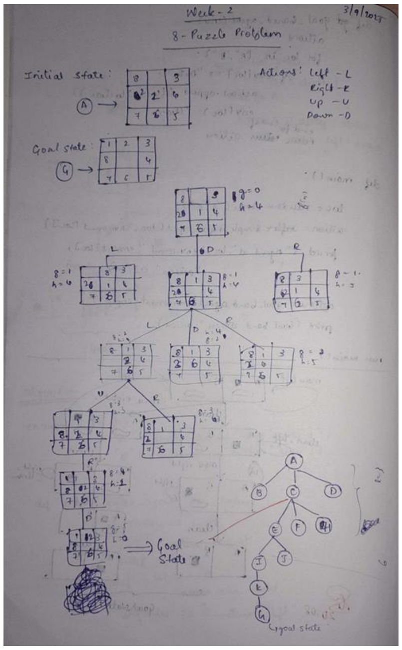
```
IDLE Shell 3.13.6
  |   |
----------
  |   |
----------
Player O's turn
Enter row (0-2): 0
Enter col (0-2): 2
  | X | O
----------
  |   |
----------
  |   |
----------
Player X's turn
Enter row (0-2): 1
Enter col (0-2): 1
  | X | O
----------
  | X |
----------
  |   |
----------
Player O's turn
Enter row (0-2): 2
Enter col (0-2): 1
  | X | O
----------
  | X |
----------
  | O |
----------
Player X's turn
Enter row (0-2): 1
Enter col (0-2): 0
  | X | O
----------
X | X |
----------
  | O |
----------
Player O's turn
Enter row (0-2): 2
Enter col (0-2): 2
  | X | O
----------
X | X |
----------
  | O | O
----------
Player X's turn
Enter row (0-2): 1
Enter col (0-2): 2
  | X | O
----------
X | X | X
----------
  | O | O
----------
Player X wins!
```

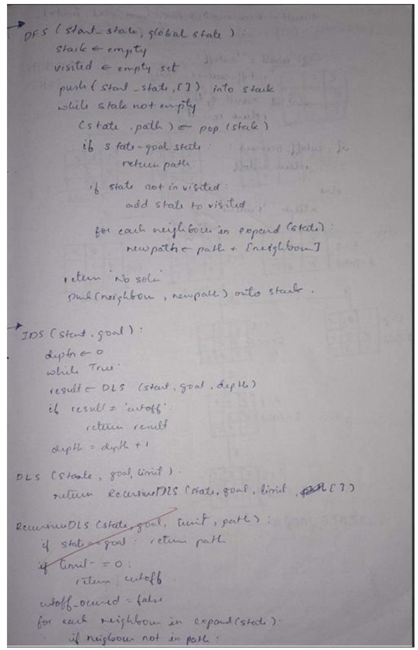
```
================================== RESTART: D:/1BM23CS145/AI/vaccumcleaner.py ============================
Initial Environment: {'A': 'Dirty', 'B': 'Dirty'}

Simple Reflex Agent starts at A
Step 1: Clean A
Step 2: Move Right to B
Step 3: Clean B
Step 4: Move Left to A
Environment after Reflex Agent: {'A': 'Clean', 'B': 'Clean'}

Goal-Based Agent starts at A
Step 1: Clean A
Step 2: Move to B
Step 3: Clean B
Environment after Goal-Based Agent: {'A': 'Clean', 'B': 'Clean'}
```

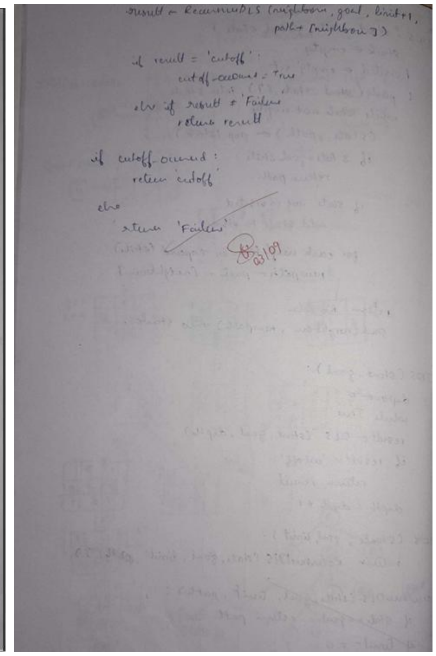
Program 2

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

Algorithm:







Code: from collections import deque
```
import copy

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

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

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

def get_neighbors(state):
neighbors = []      x, y =
find_blank(state)      for dx,
dy in moves:
        nx, ny = x + dx, y + dy          if 0
<= nx < 3 and 0 <= ny < 3:
            new_state = copy.deepcopy(state)             new_state[x][y],
new_state[nx][ny] = new_state[nx][ny], new_state[x][y]             neighbors.append(new_state)
    return neighbors

def is_goal(state):
    return state == goal_state
```

```
def print_state(state):     for
row in state:
print(row)
print()

def dfs(start_state, limit=50):
stack = [(start_state, [])]
    visited = set()

    while stack:
        state, path = stack.pop()         state_tuple =
tuple(tuple(row) for row in state)
         if     state_tuple     in     visited:
continue         visited.add(state_tuple)

        if is_goal(state):
            return path + [state]

        if len(path) >= limit:
            continue

        for neighbor in get_neighbors(state):
stack.append((neighbor, path + [state]))     return
None

def dls(state, depth, path, visited):     if
is_goal(state):
        return path + [state]
if depth == 0:
return None

    state_tuple = tuple(tuple(row) for row in state)
visited.add(state_tuple)

    for neighbor in get_neighbors(state):
        neighbor_tuple = tuple(tuple(row) for row in neighbor)
if neighbor_tuple not in visited:
            result = dls(neighbor, depth - 1, path + [state], visited)
if result:
                return result
return None
```

```
def ids(start_state, max_depth=50):      for
depth in range(max_depth):         visited = set()
result = dls(start_state, depth, [], visited)
if result:             return result      return None


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

    print("DFS Solution:")     sol_dfs =
dfs(start_state, limit=20)     if
sol_dfs:         for step in sol_dfs:
print_state(step)     else:
        print("No solution found with DFS")

    print("\nIDS Solution:")     sol_ids =
ids(start_state,  max_depth=20)         if
sol_ids:                 for  step  in  sol_ids:
print_state(step)     else:
        print("No solution found with IDS")
```

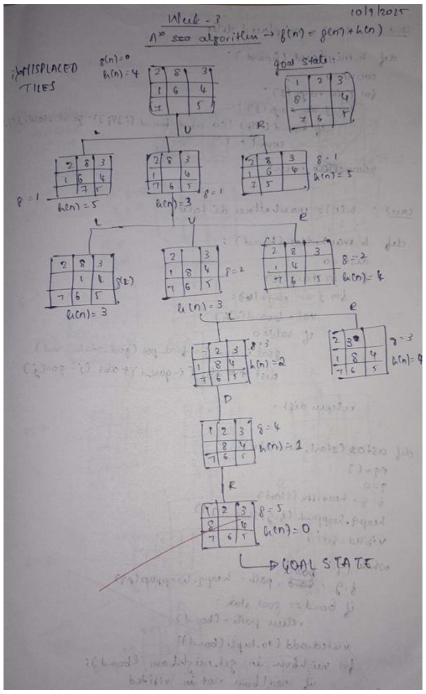
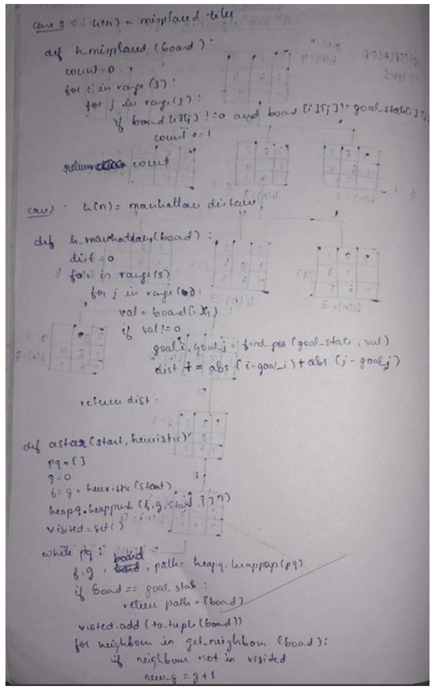
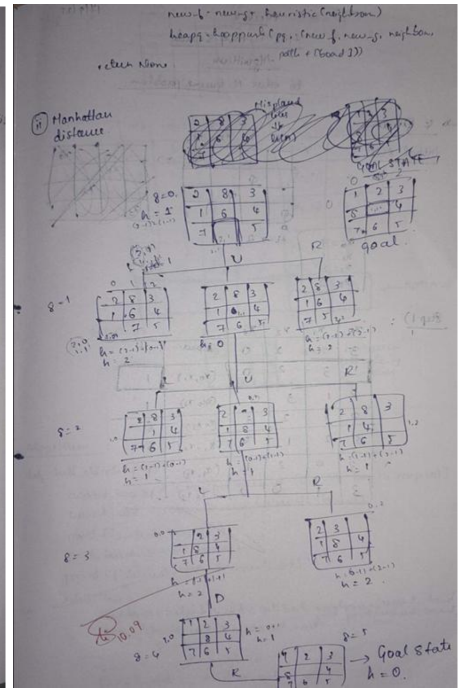
Output:

```
>>>
==================== RESTART: D:\1BM23CS145\AI\8puzzle.py ====================
DFS Solution:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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


IDS Solution:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

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

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

Program 3

Implement A* search algorithm

Algorithm:

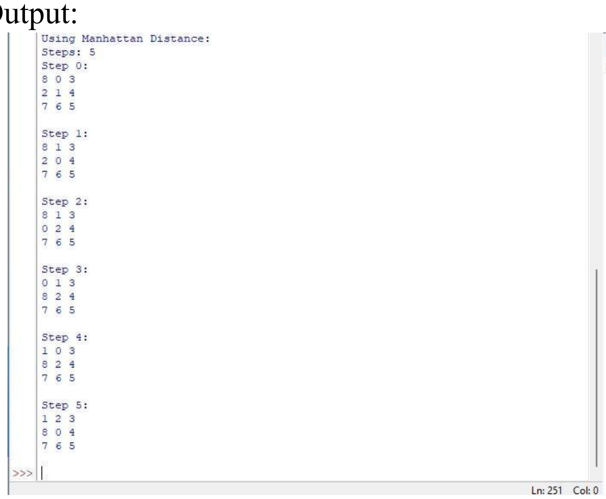
Code:

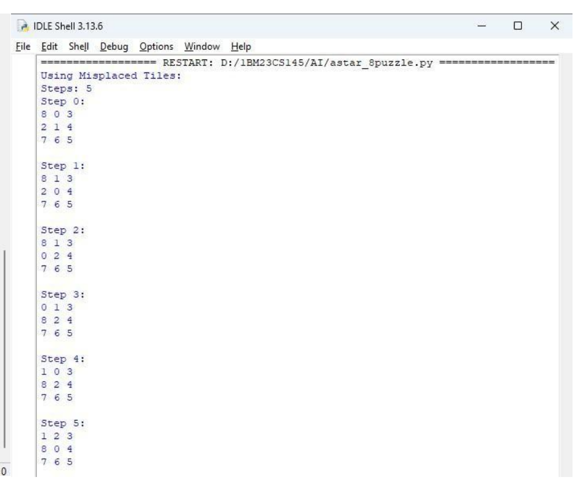
```
import heapq

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

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

def to_tuple(board):
    return tuple(tuple(row) for row in board)

def find_pos(board, value):      for
i in range(3):         for j in
range(3):             if board[i][j] ==
value:
             return (i, j)
# Heuristic 1: misplaced tiles def
h_misplaced(board):      count = 0     for i in range(3):
for j in range(3):            if board[i][j] != 0 and
board[i][j] != goal_state[i][j]:
             count += 1
return count

# Heuristic 2: manhattan distance def
h_manhattan(board):
```

```
    dist = 0     for i in
range(3):        for j in
range(3):            val =
board[i][j]            if val
!= 0:
                goal_i, goal_j = find_pos(goal_state, val)
dist += abs(i - goal_i) + abs(j - goal_j)     return dist

def get_neighbors(board):
neighbors = []     x, y =
find_pos(board, 0)     for dx,
dy in moves:
        nx, ny = x + dx, y + dy         if 0
<= nx < 3 and 0 <= ny < 3:
            new_board = [list(row) for row in board]            new_board[x][y],
new_board[nx][ny] = new_board[nx][ny], new_board[x][y]
neighbors.append(new_board)     return neighbors

def print_board(board):     for
row in board:
        print(' '.join(str(x) for x in row))
print()

def astar(start, heuristic):
    pq = []
g = 0
    f = g + heuristic(start)
heapq.heappush(pq, (f, g, start, []))
    visited = set()

    while pq:
        f, g, board, path = heapq.heappop(pq)
if board == goal_state:            return path
+ [board]

        visited.add(to_tuple(board))

        for neighbor in get_neighbors(board):            if
to_tuple(neighbor) not in visited:
                new_g = g + 1                 new_f = new_g +
heuristic(neighbor)              heapq.heappush(pq, (new_f, new_g,
neighbor, path + [board]))
return None
```

```
start_state1 = [[1,2,3],
             [4,0,6],
             [7,5,8]]

start_state2 = [
   [2, 8, 3],
   [1, 6, 4],
   [7, 0, 5]
]

start_state3 = [
   [8, 0, 3],
   [2, 1, 4],
   [7, 6, 5]
]

print("Using Misplaced Tiles:") solution =
astar(start_state3, h_misplaced)
print("Steps:", len(solution)-1) for step, board
in enumerate(solution):
   print(f"Step {step}:")
print_board(board)

print("Using Manhattan Distance:")
solution = astar(start_state3, h_manhattan)
print("Steps:", len(solution)-1) for step, board
in enumerate(solution):
   print(f"Step {step}:")
print_board(board)
```

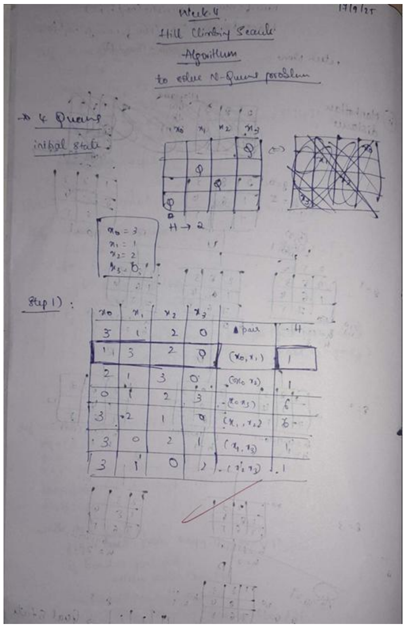
Output:

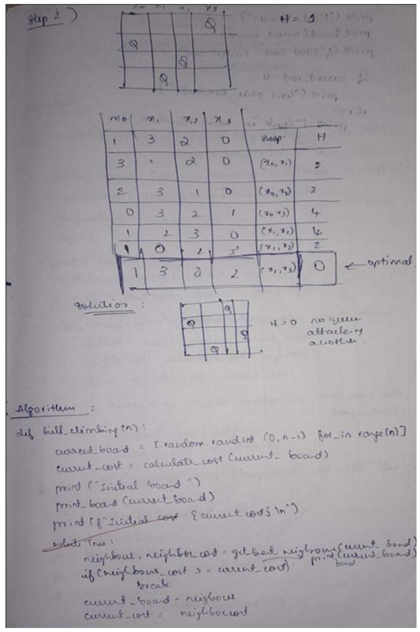


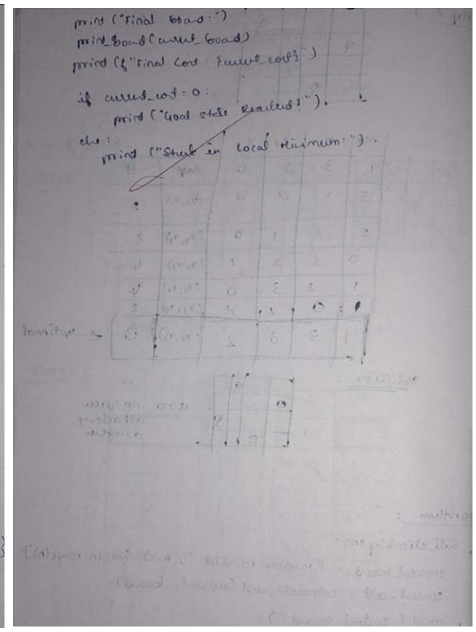


Program 4
Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



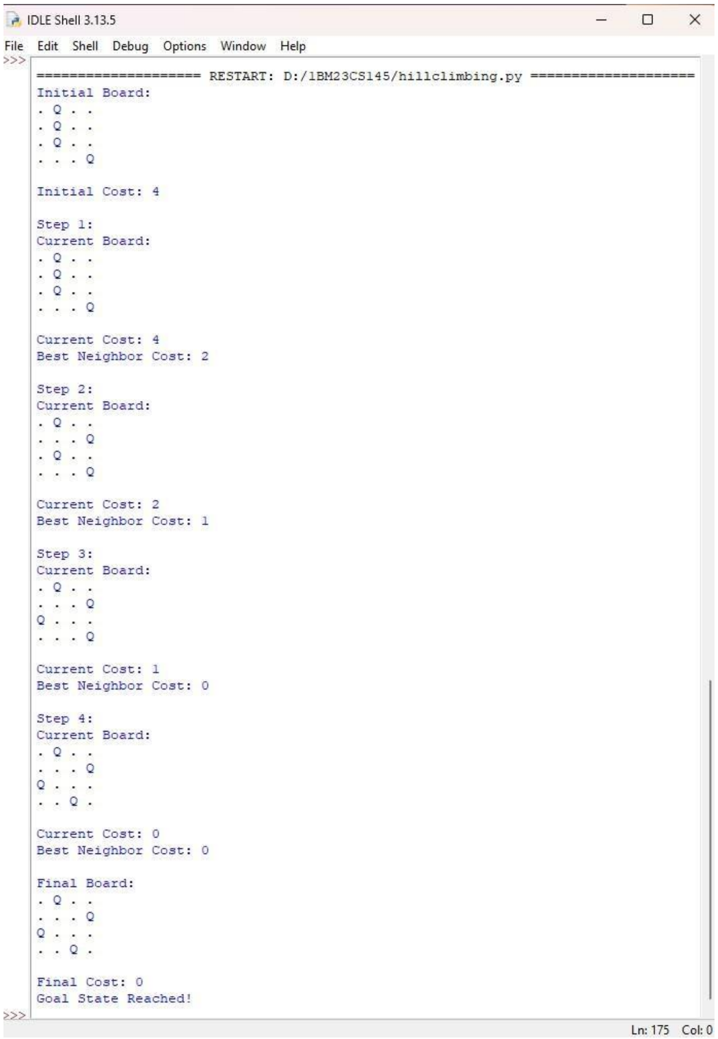




Code:

```
import random

def print_board(board):
    n = len(board)
for i in range(n):          row = ["Q" if board[i] == j else
"." for j in range(n)]
print(" ".join(row))
    print()

def calculate_cost(board):
    """Heuristic: number of pairs of queens attacking each other"""
n = len(board)     cost = 0     for i in range(n):         for j in range(i + 1,
n):             if board[i] == board[j] or abs(board[i] - board[j]) == abs(i
- j):
                cost += 1
return cost

def get_best_neighbor(board):
    n = len(board)
best_board = list(board)
    best_cost = calculate_cost(board)

    for row in range(n):
```

```
        for col in range(n):              if
board[row] != col:                neighbor =
list(board)                 neighbor[row] = col
cost = calculate_cost(neighbor)                if
cost < best_cost:                    best_cost
= cost                    best_board = neighbor
    return best_board, best_cost

def hill_climbing(n):
    current_board = [random.randint(0, n - 1) for _ in range(n)]     current_cost
= calculate_cost(current_board)

    print("Initial Board:")     print_board(current_board)
    print(f"Initial Cost: {current_cost}\n")

    step = 1
while True:
        neighbor, neighbor_cost = get_best_neighbor(current_board)         print(f"Step
{step}:")         print("Current Board:")         print_board(current_board)
print(f"Current Cost:
{current_cost}")         print(f"Best Neighbor Cost:
{neighbor_cost}\n")

        if neighbor_cost >= current_cost:
break

        current_board = neighbor
current_cost = neighbor_cost
step += 1

    print("Final Board:")     print_board(current_board)
    print(f"Final Cost: {current_cost}")

    if current_cost == 0:
        print("Goal State Reached!")
else:
        print("Stuck in Local Minimum!")

# Run for 4-Queens  hill_climbing(4)
```
Output:

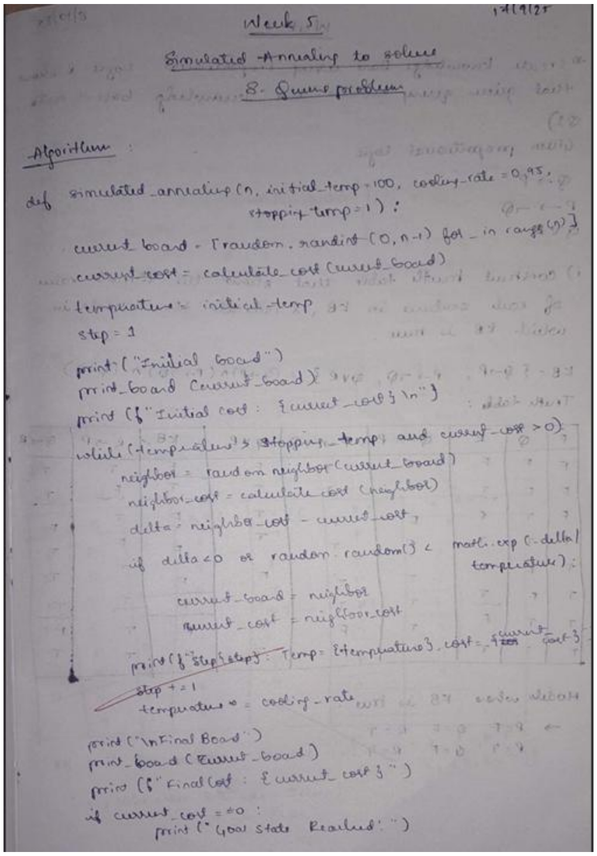



```
>>>
==================== RESTART: D:/1BM23CS145/hillclimbing.py ====================
Initial Board:
. Q . .
. Q . .
. Q . .
. . . Q

Initial Cost: 4

Step 1:
Current Board:
. Q . .
. Q . .
. Q . .
. . . Q

Current Cost: 4
Best Neighbor Cost: 2

Step 2:
Current Board:
. Q . .
. . . Q
. Q . .
. . . Q

Current Cost: 2
Best Neighbor Cost: 1

Step 3:
Current Board:
. Q . .
. . . Q
Q . . .
. . . Q

Current Cost: 1
Best Neighbor Cost: 0

Step 4:
Current Board:
. Q . .
. . . Q
Q . . .
. . Q .

Current Cost: 0
Best Neighbor Cost: 0

Final Board:
. Q . .
. . . Q
Q . . .
. . Q .

Final Cost: 0
Goal State Reached!
>>>
```


Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Week 5

Simulated Annealing to solve 8-Queens problem

17/9/25

Algorithm:

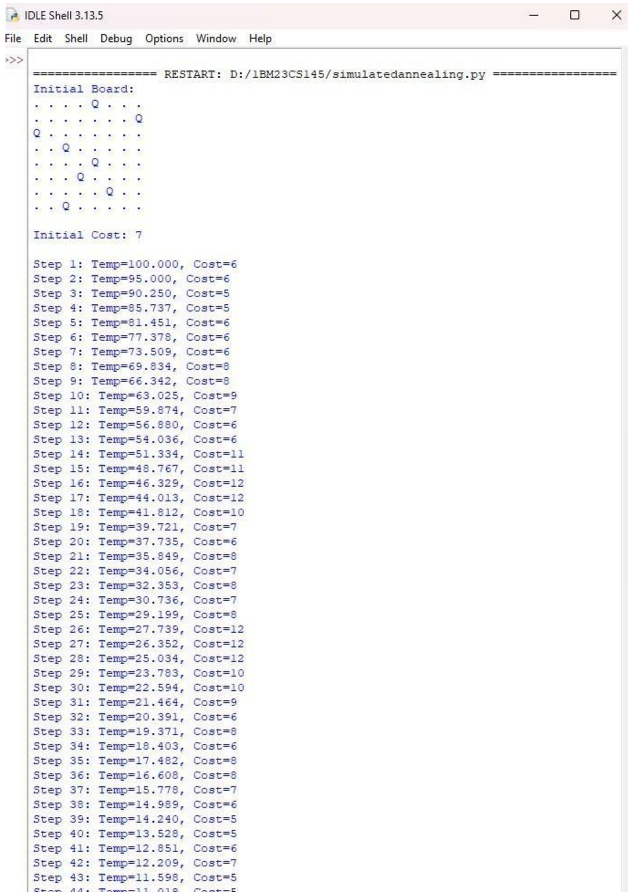
```
def simulated_annealing(n, initial_temp=100, cooling_rate=0.95,
                        stopping_temp=1):
    current_board = [random.randint(0, n-1) for _ in range(n)]
    current_cost = calculate_cost(current_board)
    temperature = initial_temp
    step = 1
    print("Initial board")
    print_board(current_board)
    print(f"Initial cost: {current_cost}\n")
    while (temperature > stopping_temp and current_cost > 0):
        neighbor = random_neighbor(current_board)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost
        if delta < 0 or random.random() < math.exp(-delta /
                                               temperature):
            current_board = neighbor
            current_cost = neighbor_cost
        print(f"Step {step}: Temp = {temperature}, cost = {current_cost}")
        step += 1
        temperature *= cooling_rate
    print("\nFinal Board")
    print_board(current_board)
    print(f"Final cost: {current_cost}")
    if current_cost == 0:
        print("Goal State Reached")
```

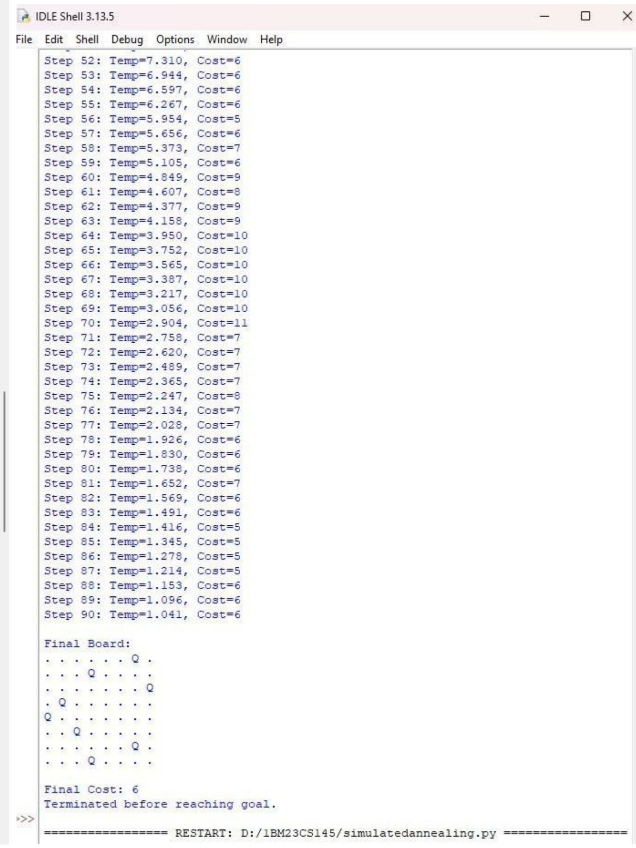
Code:

```
import random  import
math

def print_board(board):
    n = len(board)
for i in range(n):          row = ["Q" if board[i] == j else
"." for j in range(n)]
print(" ".join(row))
    print()

def calculate_cost(board):
    """Heuristic: number of pairs of queens attacking each other"""
n = len(board)     cost = 0     for i in range(n):         for j in range(i +
1, n):             if board[i] == board[j] or abs(board[i] - board[j]) ==
abs(i - j):                 cost += 1
return cost

def random_neighbor(board):
    """Generate a random neighboring board by moving one queen"""
n = len(board)     neighbor = list(board)     row = random.randint(0, n
- 1)     col = random.randint(0, n - 1)     neighbor[row] = col
```

```
    return neighbor

def simulated_annealing(n, initial_temp=100, cooling_rate=0.95, stopping_temp=1):
    current_board = [random.randint(0, n - 1) for _ in range(n)]
current_cost = calculate_cost(current_board)     temperature = initial_temp
    step = 1

    print("Initial Board:")
print_board(current_board)
    print(f"Initial Cost: {current_cost}\n")

    while temperature > stopping_temp and current_cost > 0:
        neighbor = random_neighbor(current_board)
neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        # Acceptance probability         if delta < 0 or random.random() <
math.exp(-delta / temperature):             current_board = neighbor
current_cost = neighbor_cost

        print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
step += 1         temperature *= cooling_rate

    print("\nFinal Board:")
print_board(current_board)
    print(f"Final Cost: {current_cost}")

    if current_cost == 0:
        print("Goal State Reached!")
else:
        print("Terminated before reaching goal.")

# Run for 8-Queens  simulated_annealing(8)
```

Output:

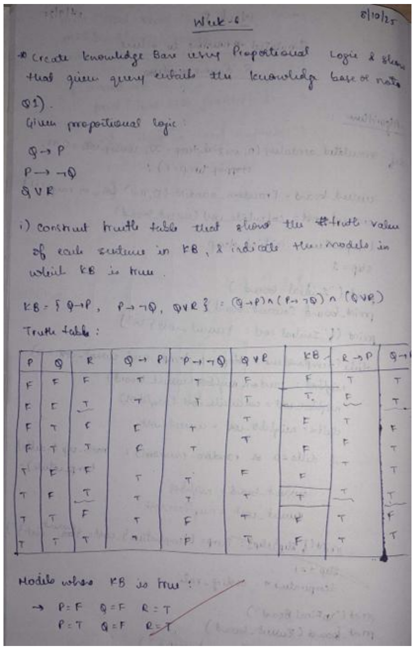




Program 6

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

Algorithm:



Week-6 8/10/25

* Create knowledge Base using Propositional Logic & show that given query entails the knowledge base or not.

Q1).
Given propositional logic:

Q → P
P → ¬Q
Q ∨ R

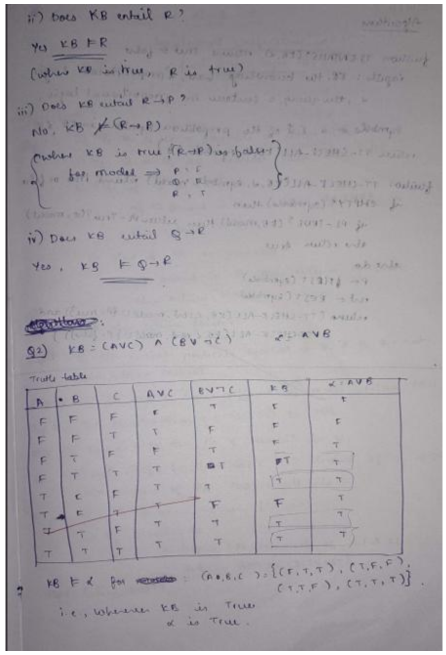
i) Construct truth table that shows the truth value of each sentence in KB, & indicate the models in which KB is true.

KB = {Q → P, P → ¬Q, Q ∨ R} : (Q → P) ∧ (P → ¬Q) ∧ (Q ∨ R)

Truth table:

P	Q	R	Q → P	P → ¬Q	Q ∨ R	KB	R → P	Q → R
F	F	F	T	T	F	F	T	T
F	F	T	T	T	T	T	F	T
F	T	F	F	T	T	F	T	F
F	T	T	F	T	T	F	F	T
T	F	F	T	T	F	F	T	T
T	F	T	T	T	T	T	T	T
T	T	F	T	F	T	F	T	F
T	T	T	T	F	T	F	T	T

Models where KB is true:
→ P = F Q = F R = T
P = T Q = F R = T



ii) Does KB entail R?
Yes KB ⊨ R
(when KB is true, R is true)

iii) Does KB entail R → P?
No, KB ⊭ (R → P)
(when KB is true, (R → P) is false for model ⇒ P: F, Q: F, R: T)

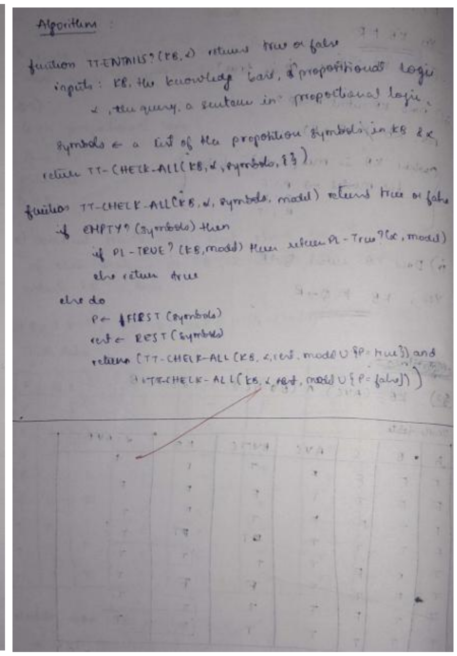
iv) Does KB entail Q → R
Yes, KB ⊨ Q → R

Q2) KB = (A ∨ C) ∧ (B ∨ ¬C) α = A ∨ B

Truth table

A	B	C	A ∨ C	B ∨ ¬C	KB	α : A ∨ B
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

KB ⊨ α for (A, B, C) = {(F, T, T), (T, F, F), (T, T, F), (T, T, T)}.
i.e., wherever KB is True α is True.



Algorithm:

function TT-ENTAILS?(KB, α) returns true or false
inputs: KB, the knowledge base, a propositional logic
α, the query, a sentence in propositional logic

symbols ← a list of the proposition symbols in KB & α
return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
if EMPTY?(symbols) then
if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
else return true
else do
P ← FIRST(symbols)
rest ← REST(symbols)
return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true}) and
TT-CHECK-ALL(KB, α, rest, model ∪ {P = false}))

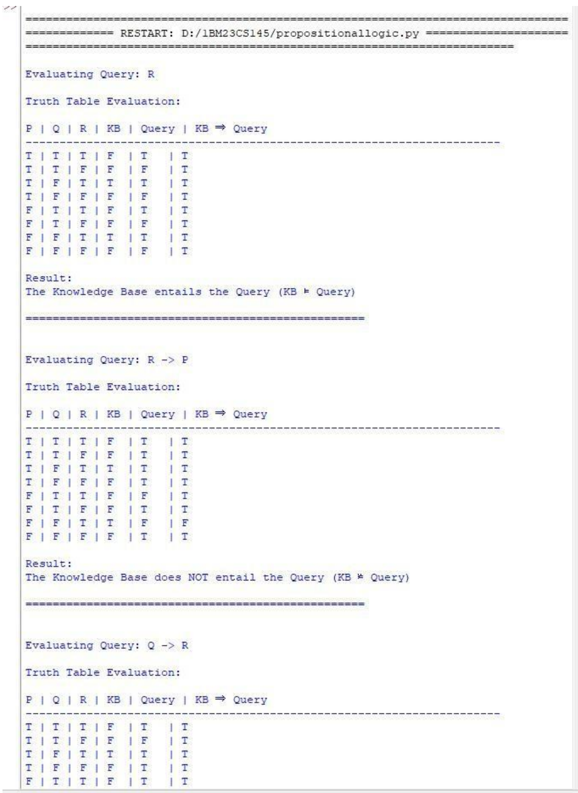
Code: import itertools
import re

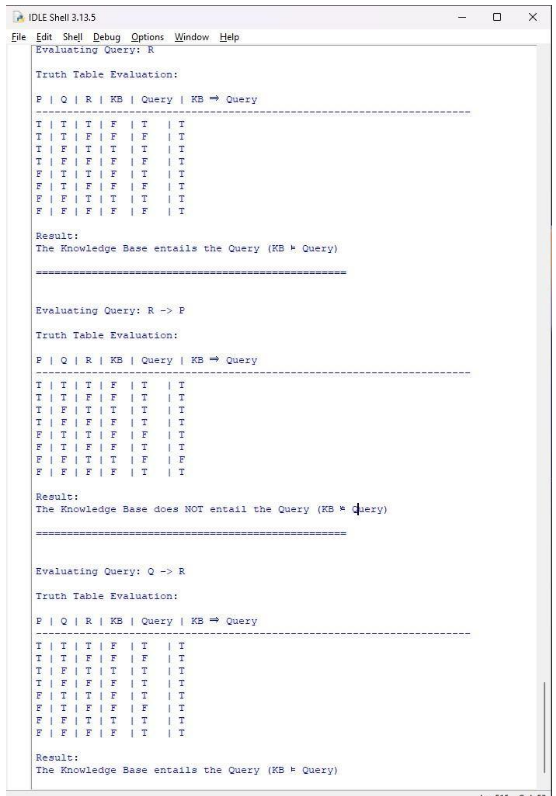
```
def evaluate(expr, model):
    """
    Evaluate a propositional logic expression under a given model (assignment).     Supported operators:
      ~ : NOT
^ : AND       v
: OR
      ->: IMPLIES
      <->: BICONDITIONAL
    """

    # Replace biconditional and implication first     expr
= expr.replace("<->", " == ")
    expr = expr.replace("->", " <= ")

    # Replace negation ~ with explicit parentheses (not X)
    expr = re.sub(r'~(\w+)', r'(not \1)', expr)
expr = re.sub(r'~\(([^)]+)\)', r'(not (\1))', expr)

    # Replace AND and OR     expr
= expr.replace("^", " and ")     expr
= expr.replace("v", " or ")
```

```
    # Replace symbols with their boolean values in the model     for
sym, val in model.items():
        expr = re.sub(r'\b' + re.escape(sym) + r'\b', str(val), expr)

    # Evaluate the final Python boolean expression     return
eval(expr)


def tt_entails(kb, query, symbols):
    """
    Truth-table enumeration to check if KB entails Query.
    Prints the truth table and returns True if entails, else False.     """

    entails = True     models = list(itertools.product([True, False],
repeat=len(symbols)))

    print("Truth Table Evaluation:\n")      header = " |
".join(symbols) + " | KB | Query | KB ⇒ Query"
print(header)     print("-" * len(header) * 2)

    for values in models:
        model = dict(zip(symbols, values))
kb_val = evaluate(kb, model)         query_val
= evaluate(query, model)          implication
= (not kb_val) or query_val

        if kb_val and not query_val:
            entails = False

        row = " | ".join(['T' if v else 'F' for v in values])         row += f" | {'T' if kb_val else 'F'}  |
{'T' if query_val else 'F'}   | {'T' if implication else 'F'}"         print(row)

    print("\nResult:")
if entails:
        print("The Knowledge Base entails the Query (KB ⊨ Query)")
else:
        print("The Knowledge Base does NOT entail the Query (KB ⊭ Query)")


# Example usage:

kb = "(Q -> P) ^ (P -> ~Q) ^ (Q v R)"  symbols
= ["P", "Q", "R"]
```

```
queries = ["R", "R -> P", "Q -> R"]

for query in queries:
   print(f"\nEvaluating Query: {query}\n")
tt_entails(kb, query, symbols)
   print("\n" + "="*50 + "\n")
```

Output:

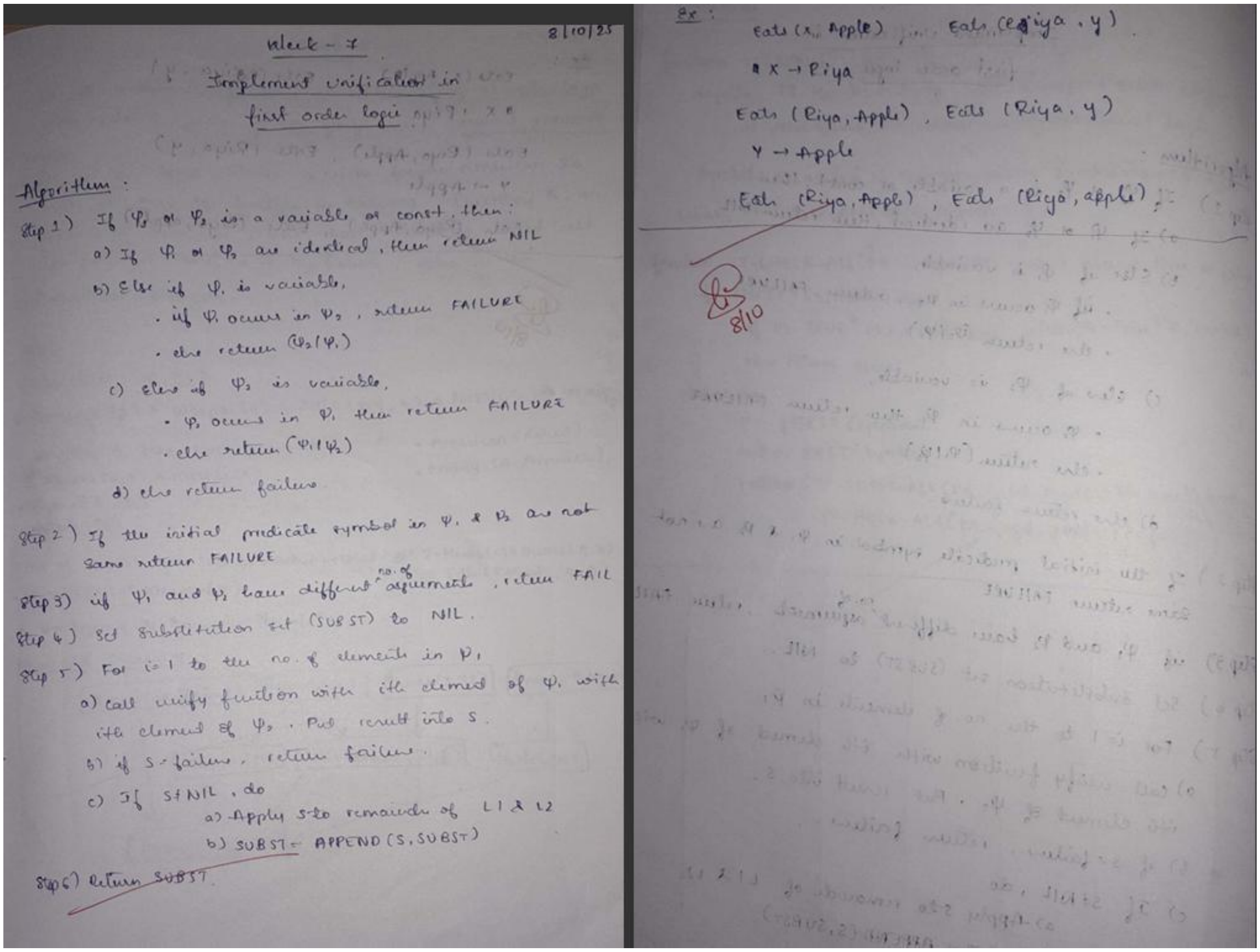




Program 7

Implement unification in first order logic

Algorithm:



Week - 7 8/10/25

Implement unification in first order logic

Algorithm:

Step 1) If Ψ_1 or Ψ_2 is a variable or const, then:

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

b) Else if Ψ_1 is variable,

- if Ψ_1 occurs in Ψ_2, return FAILURE
- else return (Ψ_2/Ψ_1)

c) Else if Ψ_2 is variable,

- Ψ_2 occurs in Ψ_1 then return FAILURE
- else return (Ψ_1/Ψ_2)

d) else return failure

Step 2) If the initial predicate symbol in Ψ_1 & Ψ_2 are not same return FAILURE

Step 3) if Ψ_1 and Ψ_2 have different no. of arguments, return FAIL

Step 4) Set substitution set (SUBST) to NIL.

Step 5) For i=1 to the no. of elements in Ψ_1

a) call unify function with ith element of Ψ_1 with ith element of Ψ_2. Put result into S.

b) if S = failure, return failure.

c) If S ≠ NIL, do

a) Apply S to remainder of L1 & L2

b) SUBST = APPEND(S, SUBST)

Step 6) Return SUBST

Ex: Eats(x, Apple), Eats(Riya, y)

x → Riya

Eats(Riya, Apple), Eats(Riya, y)

y → Apple

Eats(Riya, Apple), Eats(Riya, apple)

8/10

Code:

```
def is_variable(x):
    return isinstance(x, str) and x.islower()

def is_constant(x):
    return isinstance(x, str) and x[0].isupper()

def occurs_check(var, expr, subst):
    """Check if var occurs in expr after applying current substitution"""
if var == expr:         return True     elif isinstance(expr, list):
return any(occurs_check(var, e, subst) for e in expr)
elif expr in subst:
        return occurs_check(var, subst[expr], subst)
return False

def unify(x, y, subst=None, depth=0):
    """Main unification function with debug prints"""
    indent = "  " * depth     if subst is None:
```

print(indent + f"Substitution failed.")   return None   print(indent
+ f"Unify({x}, {y}) with subst = {subst}")

    if x == y:
        print(indent + "Terms are identical, no change.")
return subst   elif is_variable(x):
        return unify_var(x, y, subst, depth)
elif is_variable(y):
        return unify_var(y, x, subst, depth)   elif
isinstance(x, list) and isinstance(y, list):   if len(x) !=
len(y):   print(indent + "Lists have different lengths.
Fail.")   return None   for xi, yi in zip(x, y):
            subst = unify(xi, yi, subst, depth + 1)   if
subst is None: print(indent + "Failed to unify list
elements.")   return None   return subst
else:
        print(indent + "Cannot unify different constants or structures. Fail.")
return None

def unify_var(var, x, subst, depth):
    indent = " " * depth
if var in subst:
        print(indent + f"{var} is in subst, unify({subst[var]}, {x})")
return unify(subst[var], x, subst, depth + 1)   elif is_variable(x) and
x in subst:
        print(indent + f"{x} is in subst, unify({var}, {subst[x]})")
return unify(var, subst[x], subst, depth + 1)   elif occurs_check(var,
x, subst):
        print(indent + f"Occurs check failed: {var} occurs in {x}")
return None   else: print(indent + f"Add
{var} -> {x} to subst")   subst[var] = x
return subst

# Example expressions
expr1 = ['f', 'X', ['g', 'Y']] expr2
= ['f', 'a', ['g', 'b']]

print("Starting Unification:\n") result =
unify(expr1, expr2, subst={})
print("\nFinal Unification Result:", result)

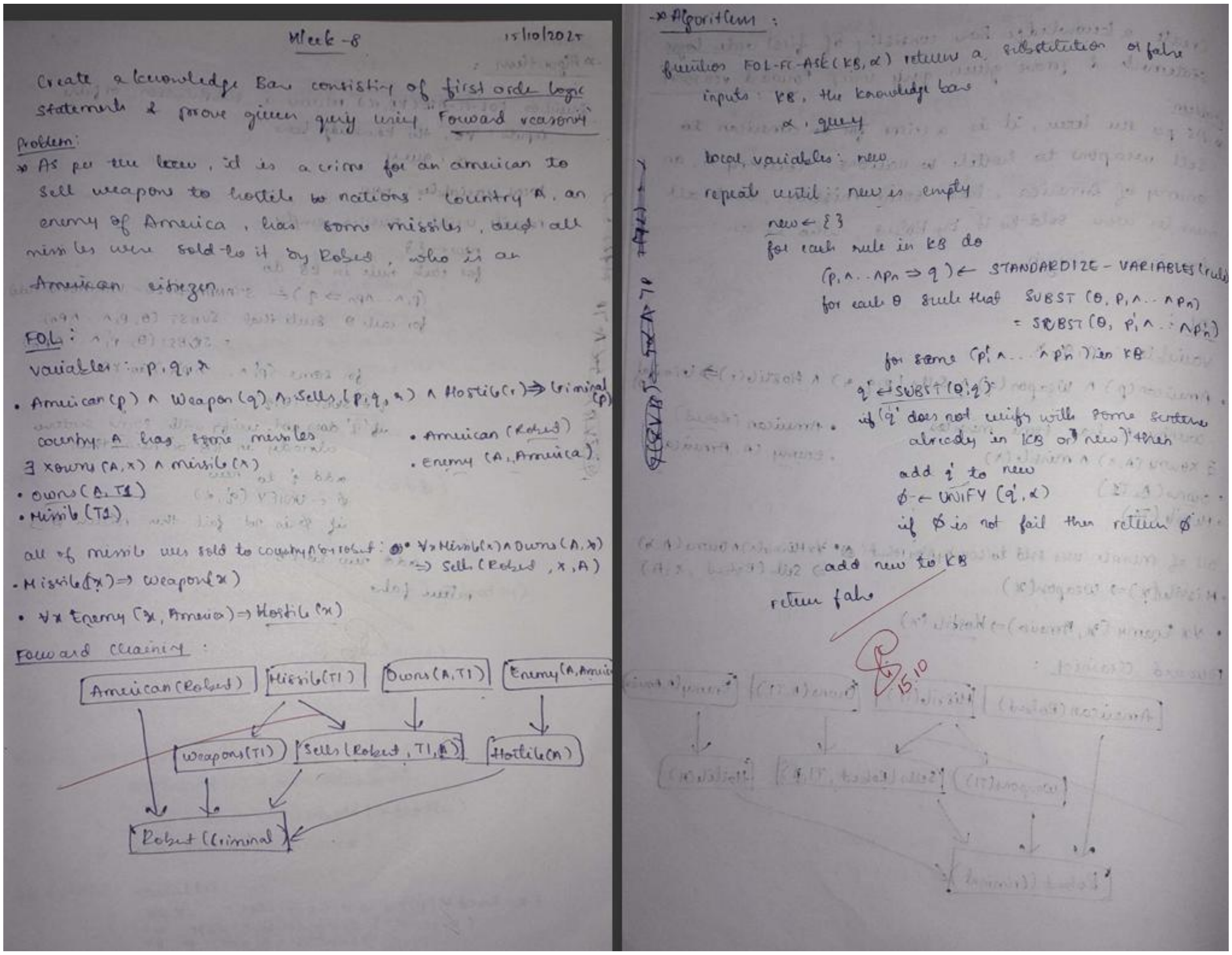
Output:

```
>>
==================== RESTART: D:/1BM23CS145/unification.py ====================
Starting Unification:

Unify(['f', 'X', ['g', 'Y']], ['f', 'a', ['g', 'b']]) with subst = {}
  Unify(f, f) with subst = {}
  Terms are identical, no change.
  Unify(X, a) with subst = {}
  Add a -> X to subst
  Unify(['g', 'Y'], ['g', 'b']) with subst = {'a': 'X'}
    Unify(g, g) with subst = {'a': 'X'}
    Terms are identical, no change.
    Unify(Y, b) with subst = {'a': 'X'}
    Add b -> Y to subst

Final Unification Result: {'a': 'X', 'b': 'Y'}
>>
                                                                  Ln: 559   Col: 0
```

Program 8

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

Algorithm:



Week -8 15/10/2025

Create a knowledge Base consisting of first order logic statements & prove given query using "Forward reasoning"

Problem:

* As per the law, it is a crime for an american to sell weapons to hostile nations. Country A, an enemy of America, has some missiles, and all missiles were sold to it by Robert, who is an American citizen.

FOL:

variables: p, q, r

- American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)

country A has some missiles
∃ x Owns(A, x) ∧ Missile(x)

- Owns(A, T1)
- Missile(T1)

- American(Robert)
- Enemy(A, America)

all of missiles were sold to country A by robert: ∀x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)

- Missile(x) ⇒ Weapon(x)
- ∀x Enemy(x, America) ⇒ Hostile(x)

Forward Chaining:

American(Robert) | Missile(T1) | Owns(A, T1) | Enemy(A, America)

Weapons(T1) | Sells(Robert, T1, A) | Hostile(A)

Robert Criminal

* Algorithm:

function FOL-FC-ASK(KB, α) returns a substitution or false

inputs: KB, the knowledge base

α, query

local variables: new

repeat until new is empty

new ← {}

for each rule in KB do

(p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)

for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)

for some (p'1 ∧ ... ∧ p'n) in KB

q' ← SUBST(θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

φ ← UNIFY(q', α)

if φ is not fail then return φ

add new to KB

return false

15/10

Code: import re

```
class KnowledgeBase:     def
__init__(self):
self.facts = set()
        self.rules = []

    def add_fact(self, fact):
self.facts.add(fact)

    def add_rule(self, head, body, label=None):
        self.rules.append({"head": head, "body": body, "label": label})

def substitute(expr, subs):
for var, val in subs.items():
        expr = re.sub(r'\b' + var + r'\b', val, expr)
return expr
def extract_predicate(expr):
```

```
    m = re.match(r'(\w+)\(([^()]*)\)', expr)
if not m:
        return None, []     pred, args = m.groups()
args = [a.strip() for a in args.split(',') if a.strip()]
    return pred, args

def unify(pattern, fact):
    p_pred, p_args = extract_predicate(pattern)
f_pred, f_args = extract_predicate(fact)     if p_pred !=
f_pred or len(p_args) != len(f_args):
        return None     subs = {}     for
pa, fa in zip(p_args, f_args):
if pa[0].islower():            if pa in
subs:                if subs[pa] != fa:
return None             else:
subs[pa] = fa
elif pa != fa:             return
None
    return subs

def forward_chain(kb, query):
derived = True     steps = []
while derived:        derived =
False        for rule in kb.rules:
body = rule["body"]
head = rule["head"]
            label = rule["label"]
              matches = [{}]
for cond in body:
new_matches = []                for
m in matches:                    for
fact in kb.facts:
                        subs = unify(cond, substitute(fact, m))
if subs is not None:
                            combined = {**m, **subs}                            consistent = True                            for
k in combined:                                if k in m and m[k] != combined[k]:
                                    consistent = False
break                               if consistent:
                                new_matches.append(combined)                matches
= new_matches
            for subs in matches:
                new_fact = substitute(head, subs)
if new_fact not in kb.facts:
kb.facts.add(new_fact)
```

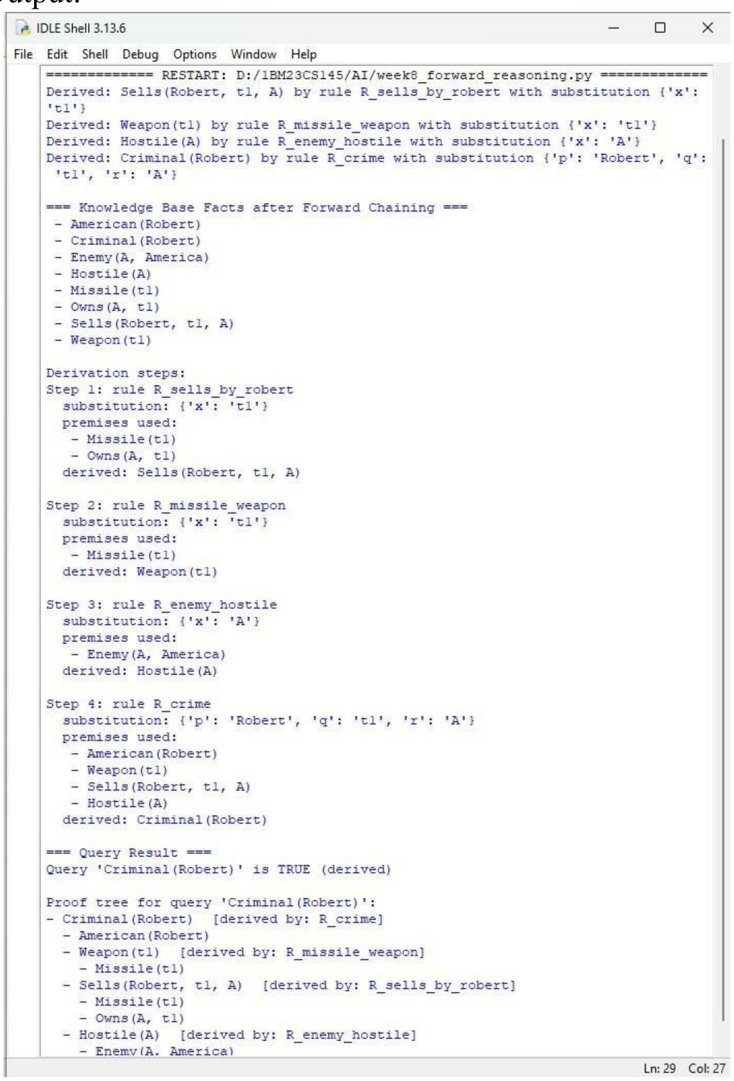
```
derived = True                    steps.append({                         "rule":
label,
                    "substitution": subs,
                    "premises": [substitute(c, subs) for c in body],
                    "derived": new_fact
                })
                print(f"Derived: {new_fact} by rule {label} with substitution {subs}")
return steps

def print_proof(query, steps):
   derived_by = {step["derived"]: step for step in steps}

   def print_tree(goal, indent=""):
if goal not in derived_by:
print(f"{indent}- {goal}")        else:
          step = derived_by[goal]           print(f"{indent}- {goal}
[derived by: {step['rule']}]")           for p in step["premises"]:
print_tree(p, indent + "  ")

   print(f"\nProof tree for query '{query}':")
print_tree(query)

# ------------------------- #
Create knowledge base
# ------------------------- kb
= KnowledgeBase()


kb.add_fact("Owns(A, t1)") kb.add_fact("Missile(t1)")
kb.add_fact("American(Robert)") kb.add_fact("Enemy(A,
America)")

kb.add_rule("Criminal(p)", ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"],
label="R_crime") kb.add_rule("Sells(Robert, x, A)", ["Missile(x)", "Owns(A, x)"],
label="R_sells_by_robert") kb.add_rule("Weapon(x)", ["Missile(x)"],
label="R_missile_weapon") kb.add_rule("Hostile(x)", ["Enemy(x, America)"],
label="R_enemy_hostile")

query = "Criminal(Robert)"
steps = forward_chain(kb, query)

print("\n=== Knowledge Base Facts after Forward Chaining ===") for
f in sorted(kb.facts):
   print(" -", f)
```

print("\nDerivation steps:") for i, step in enumerate(steps, 1):
print(f"Step {i}: rule {step['rule']}")
print(" substitution:", step["substitution"])
print(" premises used:") for p in step["premises"]:
print(" -", p) print(" derived:", step["derived"], "\n")

print("=== Query Result ===") if query in kb.facts:
print(f"Query '{query}' is TRUE (derived)") else:
print(f"Query '{query}' could NOT be derived")

print_proof(query, steps)

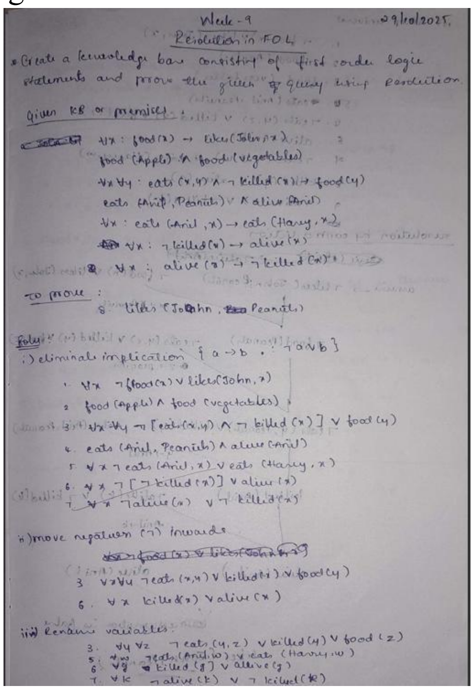
Output:



```
============= RESTART: D:/1BM23CS145/AI/week8_forward_reasoning.py =============
Derived: Sells(Robert, tl, A) by rule R_sells_by_robert with substitution {'x':
'tl'}
Derived: Weapon(tl) by rule R_missile_weapon with substitution {'x': 'tl'}
Derived: Hostile(A) by rule R_enemy_hostile with substitution {'x': 'A'}
Derived: Criminal(Robert) by rule R_crime with substitution {'p': 'Robert', 'q':
 'tl', 'r': 'A'}

=== Knowledge Base Facts after Forward Chaining ===
 - American(Robert)
 - Criminal(Robert)
 - Enemy(A, America)
 - Hostile(A)
 - Missile(tl)
 - Owns(A, tl)
 - Sells(Robert, tl, A)
 - Weapon(tl)

Derivation steps:
Step 1: rule R_sells_by_robert
  substitution: {'x': 'tl'}
  premises used:
   - Missile(tl)
   - Owns(A, tl)
  derived: Sells(Robert, tl, A)

Step 2: rule R_missile_weapon
  substitution: {'x': 'tl'}
  premises used:
   - Missile(tl)
  derived: Weapon(tl)

Step 3: rule R_enemy_hostile
  substitution: {'x': 'A'}
  premises used:
   - Enemy(A, America)
  derived: Hostile(A)

Step 4: rule R_crime
  substitution: {'p': 'Robert', 'q': 'tl', 'r': 'A'}
  premises used:
   - American(Robert)
   - Weapon(tl)
   - Sells(Robert, tl, A)
   - Hostile(A)
  derived: Criminal(Robert)

=== Query Result ===
Query 'Criminal(Robert)' is TRUE (derived)

Proof tree for query 'Criminal(Robert)':
- Criminal(Robert)  [derived by: R_crime]
  - American(Robert)
  - Weapon(tl)  [derived by: R_missile_weapon]
    - Missile(tl)
  - Sells(Robert, tl, A)  [derived by: R_sells_by_robert]
    - Missile(tl)
    - Owns(A, tl)
  - Hostile(A)  [derived by: R_enemy_hostile]
    - Enemy(A, America)
```

Program 9
Create a knowledge base consisting of first order logic statements and prove the given query using Resolution

Algorithm:


Week - 9 29/10/2025.
Resolution in FOL
• Create a knowledge base consisting of first order logic statements and prove the given query using Resolution.

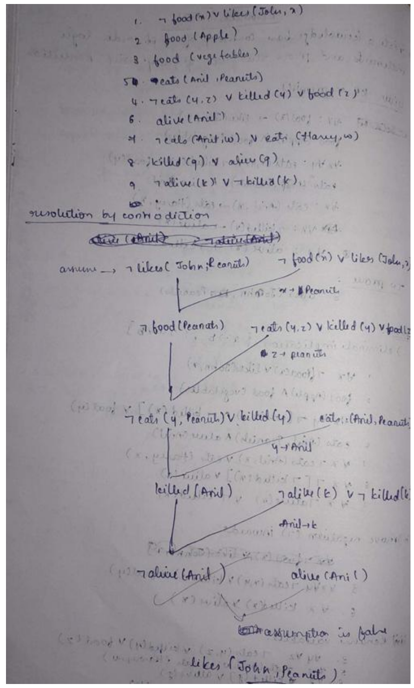
Given KB or premises:
∀x : food(x) → likes(John, x)
food (Apple) ∧ food (vegetables)
∀x ∀y : eats (x, y) ∧ ¬ killed (x) → food (y)
eats (Anil, Peanuts) ∧ alive (Anil)
∀x : eats (Anil, x) → eats (Harry, x)
∀x : ¬ killed (x) → alive (x)
∀x : alive (x) → ¬ killed (x)

To prove:
8. likes (John, Peanuts)

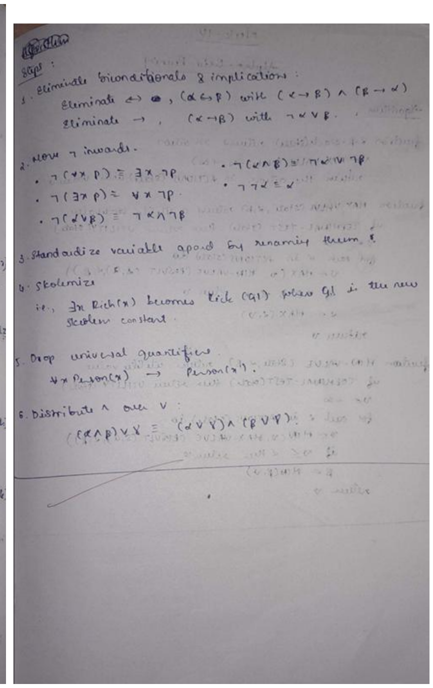
Proof:
i) eliminate implication { a → b : ¬a ∨ b }
1. ∀x ¬food(x) ∨ likes(John, x)
2. food (Apple) ∧ food (vegetables)
3. ∀x ∀y ¬ [eats (x, y) ∧ ¬ killed (x)] ∨ food (y)
4. eats (Anil, Peanuts) ∧ alive (Anil)
5. ∀x ¬ eats (Anil, x) ∨ eats (Harry, x)
6. ∀x ¬ [¬ killed (x)] ∨ alive (x)
7. ∀x ¬alive (x) ∨ ¬ killed (x)

ii) move negation (¬) inwards
3. ∀x∀y ¬eats (x, y) ∨ killed (x) ∨ food (y)
6. ∀x killed (x) ∨ alive (x)

iii) Rename variables
3. ∀y ∀z ¬ eats (y, z) ∨ killed (y) ∨ food (z)
5. ∀w ¬eats (Anil, w) ∨ eats (Harry, w)
6. ∀g killed (g) ∨ alive (g)
7. ∀k ¬alive (k) ∨ ¬ killed (k)


1. ¬ food (x) ∨ likes (John, x)
2. food (Apple)
3. food (vegetables)
5. eats (Anil, Peanuts)
4. ¬eats (y, z) ∨ killed (y) ∨ food (z)
6. alive (Anil)
7. ¬eats (Anil, w) ∨ eats (Harry, w)
8. killed (g) ∨ alive (g)
9. ¬alive (k) ∨ ¬ killed (k)

resolution by contradiction
assume → ¬ likes (John, Peanuts)
¬ food (x) ∨ likes (John, x)
x → Peanuts
¬ food (Peanuts)
¬eats (y, z) ∨ killed (y) ∨ food (z)
z → peanuts
¬eats (y, Peanuts) ∨ killed (y)
eats (Anil, Peanuts)
y → Anil
killed (Anil)
¬alive (k) ∨ ¬ killed (k)
Anil → k
¬alive (Anil)
alive (Anil)
assumption is false
likes (John, Peanuts)


Algorithm
Steps:
1. Eliminate biconditionals & implications:
Eliminate ↔, (α ↔ β) with (α → β) ∧ (β → α)
Eliminate →, (α → β) with ¬α ∨ β.
2. Move ¬ inwards.
• ¬(∀x p) ≡ ∃x ¬p
• ¬(∃x p) ≡ ∀x ¬p
• ¬(α ∨ β) ≡ ¬α ∧ ¬β
• ¬(α ∧ β) ≡ ¬α ∨ ¬β
• ¬¬α ≡ α
3. Standardize variable apart by renaming them.
4. Skolemize
i.e., ∃x Rich(x) becomes Rich (G1) where G1 is the new Skolem constant.
5. Drop universal quantifiers.
∀x Person(x) → Person(x).
6. Distribute ∧ over ∨:
(α ∧ β) ∨ γ ≡ (α ∨ γ) ∧ (β ∨ γ).

Code:
STEP 1. Input FOL Statements

```
FOL_statements = {
    'a': "∀x: food(x) → likes(John, x)",
    'b': "food(Apple) ∧ food(Vegetables)",
    'c': "∀x∀y: eats(x, y) ∧ ¬killed(x) → food(y)",
    'd': "eats(Anil, Peanuts) ∧ alive(Anil)",
    'e': "∀x: eats(Anil, x) → eats(Harry, x)",
    'f': "∀x: ¬killed(x) → alive(x)",
    'g': "∀x: alive(x) → ¬killed(x)",
    'h': "likes(John, Peanuts)"
}

print("=== STEP 1: Given FOL Statements ===") for
key, val in FOL_statements.items():
    print(f"{key}. {val}")
```

STEP 2. Eliminate Implications

```
print("=== STEP 2: After Removing Implications ===")

CNF_imp_removed = {
    'a': "¬food(x) ∨ likes(John, x)",
    'b1': "food(Apple)",
    'b2': "food(Vegetables)",
    'c': "¬eats(x, y) ∨ killed(x) ∨ food(y)",
    'd1': "eats(Anil, Peanuts)",
    'd2': "alive(Anil)",
    'e': "¬eats(Anil, x) ∨ eats(Harry, x)",
    'f': "killed(x) ∨ alive(x)",
    'g': "¬alive(x) ∨ ¬killed(x)",
    'h': "likes(John, Peanuts)"
}

for key, val in CNF_imp_removed.items():
    print(f"{key}. {val}")

# STEP 3. Standardize Variables and Drop Quantifiers

print("=== STEP 3: Standardized Variables (Dropped Quantifiers) ===") for
key, val in CNF_imp_removed.items():
    print(f"{key}. {val}")

# STEP 4. Final CNF Knowledge Base

print("=== STEP 4: Final CNF Clauses ===")

CNF_clauses = [
    "¬food(x) ∨ likes(John, x)",
    "food(Apple)",
    "food(Vegetables)",
    "¬eats(y, z) ∨ killed(y) ∨ food(z)",
    "eats(Anil, Peanuts)",
    "alive(Anil)",
    "¬eats(Anil, w) ∨ eats(Harry, w)",
    "killed(g) ∨ alive(g)",
    "¬alive(k) ∨ ¬killed(k)",
    "likes(John, Peanuts)"
]

for i, clause in enumerate(CNF_clauses, start=1):
    print(f"{i}. {clause}")
```

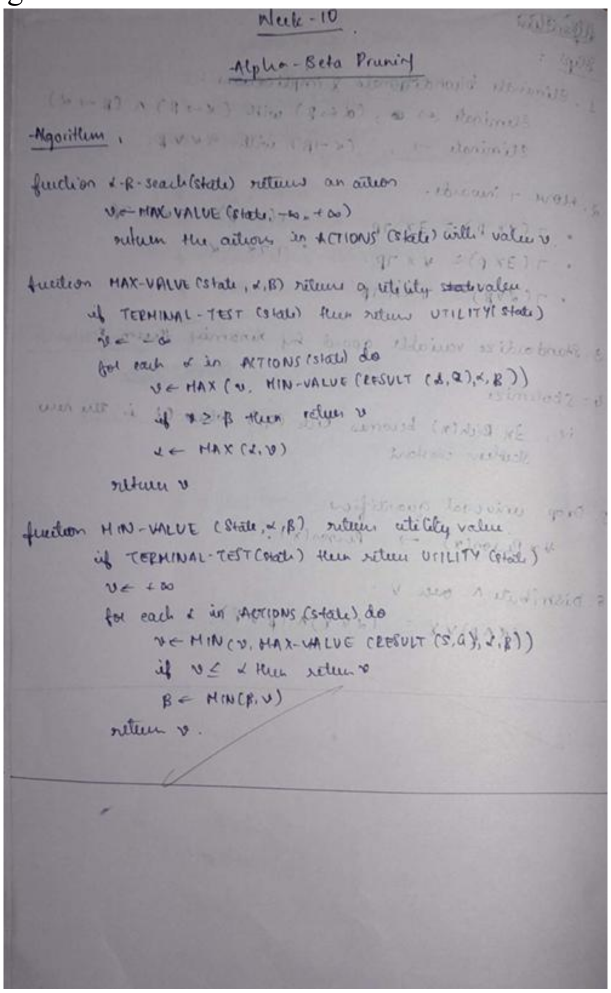
```
# STEP 5. Resolution Proof (Text-Based)

print("=== STEP 5: Resolution Proof ===")

steps = [
   ("1", "Negate Goal", "¬likes(John, Peanuts)"),
   ("2", "Resolve (1) with (¬food(x) ∨ likes(John, x)) using {x/Peanuts}", "¬food(Peanuts)"),     ("3",
"Resolve (2) with (¬eats(y,z) ∨ killed(y) ∨ food(z)) using {z/Peanuts}", "¬eats(y,Peanuts) ∨ killed(y)"),
   ("4", "Resolve (3) with (eats(Anil, Peanuts)) using {y/Anil}", "killed(Anil)"),
   ("5", "Resolve (4) with (¬alive(k) ∨ ¬killed(k)) using {k/Anil}", "¬alive(Anil)"),     ("6",
"Resolve (5) with (alive(Anil))", "⊥ (Contradiction)")
]

for num, action, result in steps:     print(f"Step
{num}: {action}")
   print(f"      ⇒ {result}\n")

print("Contradiction reached ⇒ Therefore, John likes Peanuts is TRUE.\n")
```

Output:

```
=== STEP 1: Given FOL Statements ===
a. ∀x: food(x) → likes(John, x)
b. food(Apple) ∧ food(Vegetables)
c. ∀x∀y: eats(x, y) ∧ ¬killed(x) → food(y)
d. eats(Anil, Peanuts) ∧ alive(Anil)
e. ∀x: eats(Anil, x) → eats(Harry, x)
f. ∀x: ¬killed(x) → alive(x)
g. ∀x: alive(x) → ¬killed(x)
h. likes(John, Peanuts)
=== STEP 2: After Removing Implications ===
a. ¬food(x) v likes(John, x)
b1. food(Apple)
b2. food(Vegetables)
c. ¬eats(x, y) v killed(x) v food(y)
d1. eats(Anil, Peanuts)
d2. alive(Anil)
e. ¬eats(Anil, x) v eats(Harry, x)
f. killed(x) v alive(x)
g. ¬alive(x) v ¬killed(x)
h. likes(John, Peanuts)
=== STEP 3: Standardized Variables (Dropped Quantifiers) ===
a. ¬food(x) v likes(John, x)
b1. food(Apple)
b2. food(Vegetables)
c. ¬eats(x, y) v killed(x) v food(y)
d1. eats(Anil, Peanuts)
d2. alive(Anil)
e. ¬eats(Anil, x) v eats(Harry, x)
f. killed(x) v alive(x)
g. ¬alive(x) v ¬killed(x)
h. likes(John, Peanuts)
=== STEP 4: Final CNF Clauses ===
1. ¬food(x) v likes(John, x)
2. food(Apple)
3. food(Vegetables)
4. ¬eats(y, z) v killed(y) v food(z)
5. eats(Anil, Peanuts)
6. alive(Anil)
7. ¬eats(Anil, w) v eats(Harry, w)
8. killed(g) v alive(g)
9. ¬alive(k) v ¬killed(k)
10. likes(John, Peanuts)
=== STEP 5: Resolution Proof ===
Step 1: Negate Goal
      ⇒ ¬likes(John, Peanuts)

Step 2: Resolve (1) with (¬food(x) v likes(John, x)) using {x/Peanuts}
      ⇒ ¬food(Peanuts)

Step 3: Resolve (2) with (¬eats(y,z) v killed(y) v food(z)) using {z/Peanuts}
      ⇒ ¬eats(y,Peanuts) v killed(y)

Step 4: Resolve (3) with (eats(Anil, Peanuts)) using {y/Anil}
      ⇒ killed(Anil)

Step 5: Resolve (4) with (¬alive(k) v ¬killed(k)) using {k/Anil}
      ⇒ ¬alive(Anil)
```

Program 10
Implement Alpha-Beta Pruning.

Algorithm:


Week - 10

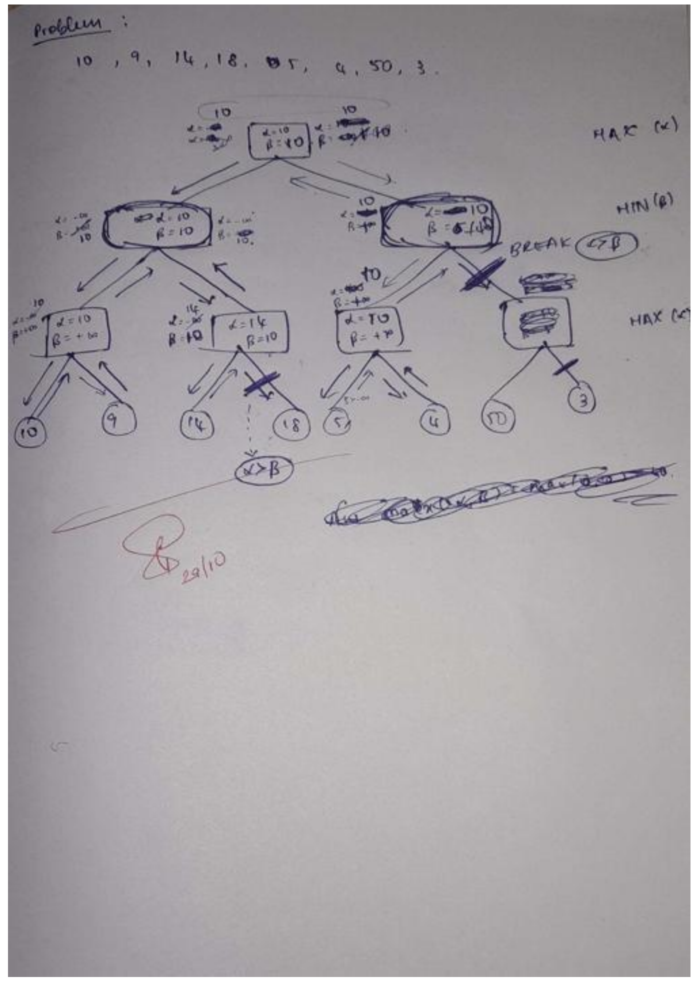
Alpha-Beta Pruning

Algorithm :

function α-β-search(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 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.



Code:

```
import math

# -------------------------------------------  #
Define the game tree structure
# ------------------------------------------- game_tree =
{
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['L1', 'L2'],
    'E': ['L3', 'L4'],
    'F': ['L5', 'L6'],
    'G': ['L7', 'L8'],
```

```
    'L1': 10,
'L2': 9,
    'L3': 14,
    'L4': 18,
    'L5': 5,
    'L6': 4,
    'L7': 50,
    'L8': 3
}

# ------------------------------------------ # Pretty
print the game tree as ASCII art # ---------------
---------------------------- def print_tree():
    print("\nGame Tree Structure:\n")
print("                A (MAX)")
print("             /       \\")     print("
B (MIN)       C (MIN)")
    print("         /    \\       /     \\")
    print("       D (MAX)  E (MAX)  F (MAX)  G (MAX)")
    print("     /  \\     /   \\     /   \\     /   \\")     print("
10    9   14   18   5    4   50    3")     print("\n---------
----------------------------------\n")


# ------------------------------------------
# Alpha-Beta Pruning Implementation (with detailed trace)  #
------------------------------------------ def alphabeta(node,
depth, alpha, beta, maximizing_player):     indent = "  " *
depth  # indentation for better readability

    # If leaf node     if isinstance(game_tree[node],
int):
        print(f"{indent}Reached leaf {node} with value {game_tree[node]}")
return game_tree[node]

    # MAX node     if maximizing_player:         print(f"{indent}Exploring MAX node {node}
(depth={depth}), α={alpha}, β={beta}")         max_eval = -math.inf         for child in
game_tree[node]:
            print(f"{indent}--> Exploring child {child} of {node}")
eval = alphabeta(child, depth + 1, alpha, beta, False)
            max_eval = max(max_eval, eval)             alpha = max(alpha, eval)
print(f"{indent}Updated MAX node {node}: value={max_eval}, α={alpha}, β={beta}")
if beta <= alpha:
                print(f"{indent}!!! Pruning at MAX node {node} (β={beta} ≤ α={alpha})")
```

```
break         return max_eval

    # MIN node     else:
print(f"{indent}Explo
ring MIN node
{node}
(depth={depth}),
α={alpha},
β={beta}")
min_eval = math.inf
for child in
game_tree[node]:
            print(f"{indent}--> Exploring child {child} of {node}")            eval = alphabeta(child,
depth + 1, alpha, beta, True)             min_eval = min(min_eval, eval)             beta = min(beta,
eval)            print(f"{indent}Updated MIN node {node}: value={min_eval}, α={alpha},
β={beta}")            if beta <= alpha:
                print(f"{indent}!!! Pruning at MIN node {node} (β={beta} ≤ α={alpha})")
break
        return min_eval


# -------------------------------------------
# Run the algorithm
# ------------------------------------------- print_tree()
print("Starting Alpha-Beta
Pruning...\n")

best_value = alphabeta('A', 0, -math.inf, math.inf, True)

print("\n-------------------------------------------") print(f"
Best achievable value at root (A): {best_value}") print("-------------------------------------------")
```

Output:

```
Game Tree Structure:

               A (MAX)
             /         \
          B (MIN)       C (MIN)
         /     \       /     \
      D (MAX)  E (MAX)  F (MAX)  G (MAX)
     /   \    /   \    /   \    /   \
    10    9   14   18   5    4   50    3

--------------------------------------------

Starting Alpha-Beta Pruning...

Exploring MAX node A (depth=0), α=-inf, β=inf
--> Exploring child B of A
  Exploring MIN node B (depth=1), α=-inf, β=inf
  --> Exploring child D of B
    Exploring MAX node D (depth=2), α=-inf, β=inf
    --> Exploring child L1 of D
      Reached leaf L1 with value 10
    Updated MAX node D: value=10, α=10, β=inf
    --> Exploring child L2 of D
      Reached leaf L2 with value 9
    Updated MAX node D: value=10, α=10, β=inf
  Updated MIN node B: value=10, α=-inf, β=10
  --> Exploring child E of B
    Exploring MAX node E (depth=2), α=-inf, β=10
    --> Exploring child L3 of E
      Reached leaf L3 with value 14
    Updated MAX node E: value=14, α=14, β=10
    !!! Pruning at MAX node E (β=10 ≤ α=14)
  Updated MIN node B: value=10, α=-inf, β=10
Updated MAX node A: value=10, α=10, β=inf
--> Exploring child C of A
  Exploring MIN node C (depth=1), α=10, β=inf
  --> Exploring child F of C
    Exploring MAX node F (depth=2), α=10, β=inf
    --> Exploring child L5 of F
      Reached leaf L5 with value 5
    Updated MAX node F: value=5, α=10, β=inf
    --> Exploring child L6 of F
      Reached leaf L6 with value 4
    Updated MAX node F: value=5, α=10, β=inf
  Updated MIN node C: value=5, α=10, β=5
  !!! Pruning at MIN node C (β=5 ≤ α=10)
Updated MAX node A: value=10, α=10, β=inf

--------------------------------------------
✅ Best achievable value at root (A): 10
--------------------------------------------
```