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)

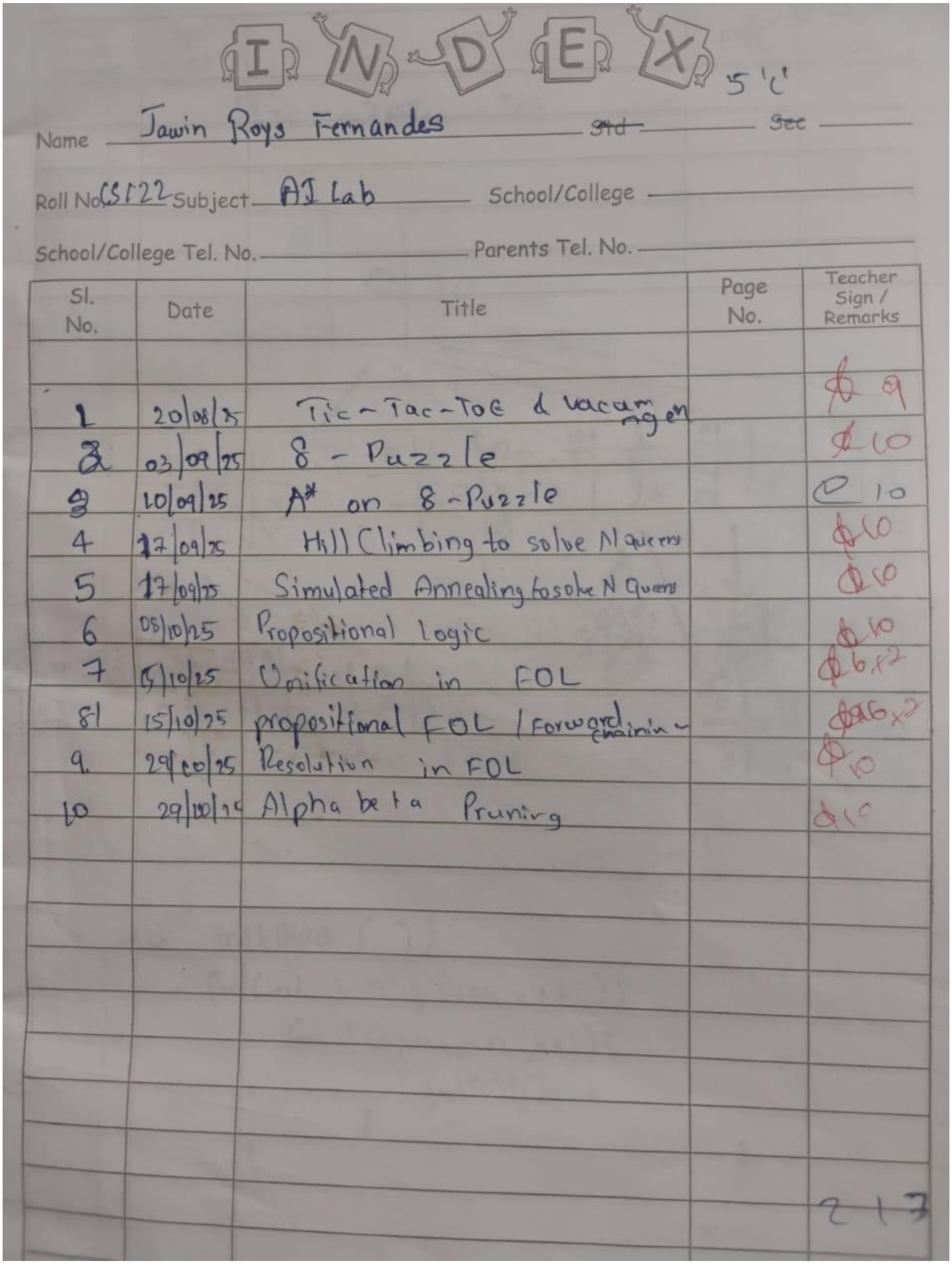
Department of Computer Science and Engineering

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



INDEX

5 'C'

Name Jawin Roys Fernandes ~~Std~~ Sec

Roll No CS122 Subject AI Lab School/College

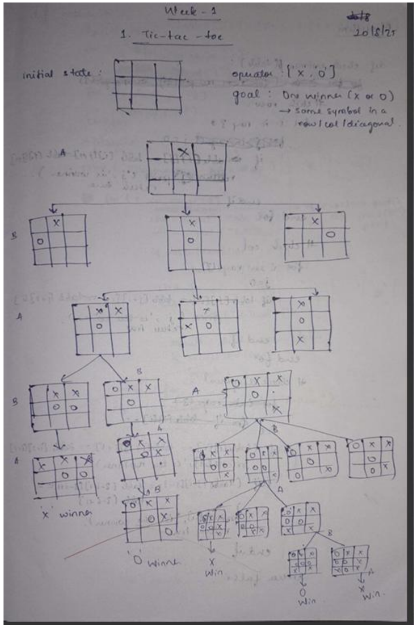
School/College Tel. No. Parents Tel. No.

Github Link:
https://github.com/kavana-ma/AI_Lab

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:



Week - 1

1. Tic-tac-toe

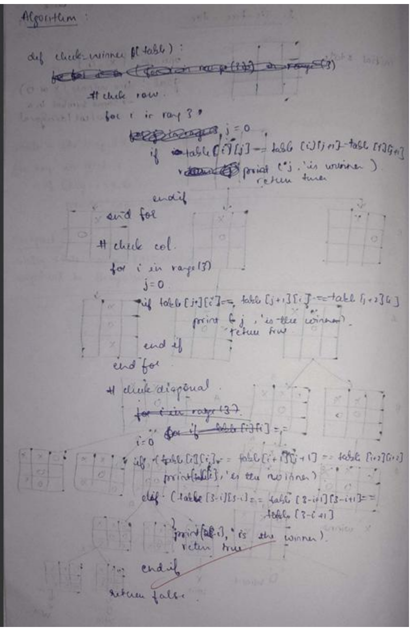
~~date~~ 20/8/25

initial state :

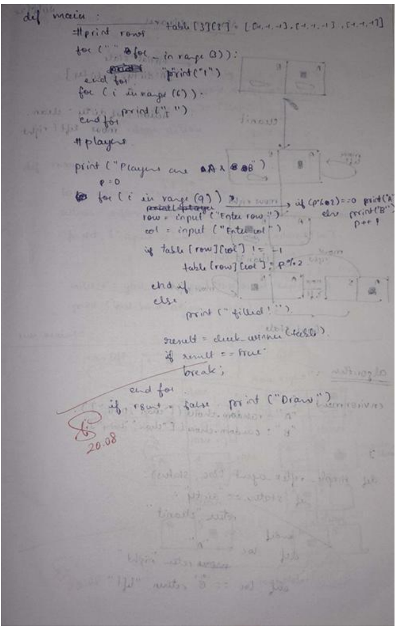
operator ['x','o']

goal : One winner (x or o)
→ same symbol in a row/col/diagonal

A
B
A
B
A
B
A
B
A
B
'x' winner
'O' winner
X win
O win
X win.

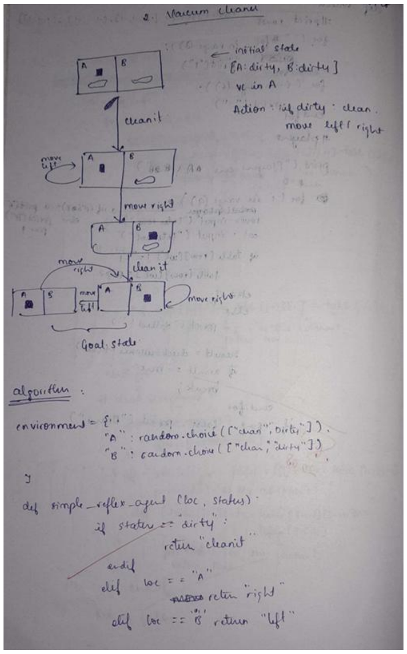


```
Algorithm :

def check_winner ~~[~~(table) :
    ~~for (for i in range(3)) in range(3)~~
    # check row
        for i in rang 3 :
            ~~for j in range~~, j = 0
            if table [i][j] == table [i][j+1] == table [i][j+1]
                print (j, 'is winner )
                return true
            endif
        end for
    # check col.
        for i in range(3)
            j = 0
            if table [j][i] == table [j+1][i] == table [j+2][i]
                print (j, 'is the winner)
                return true
            end if
        end for
    # check diagonal
        ~~for i in range (3)~~
        ~~for if table [i][i] =~~ ,=
        i = 0
        if (table [i][i] == table [i+1][i+1] == table [i+2][i+2]
            print(table[i], 'is the winner)
        elif (table [3-i][3-i] == table [3-i+1][3-i+1] ==
                table [3-i+1]
            print(table[3-i], 'is the winner).
            return true
        endif
    return false
```



```
def main :
    → table [3][3] = [[-1,-1,-1], [-1,-1,-1], [-1,-1,-1]]
    # print rows
    for (" " for _ in range (3)):
        ~~print~~ print("|")
    end for
    for (i in range (6)):
        print ("_")
    end for
    # players
    print ("Players are A & B")
        p = 0
    for (i in range (9)) → if (p%2)==0 print("A")
                           else print("B")
                           p++
        ~~print(player~~
        row = input ("Enter row:")
        col = input ("Enter col")
        if table [row][col] != -1
            table [row][col] = p%2
        end if
        else
            print ("filled!")
        result = check_winner (table)
        if result == true
            break;
    end for
    if result = false print ("Draw")
```

20.08



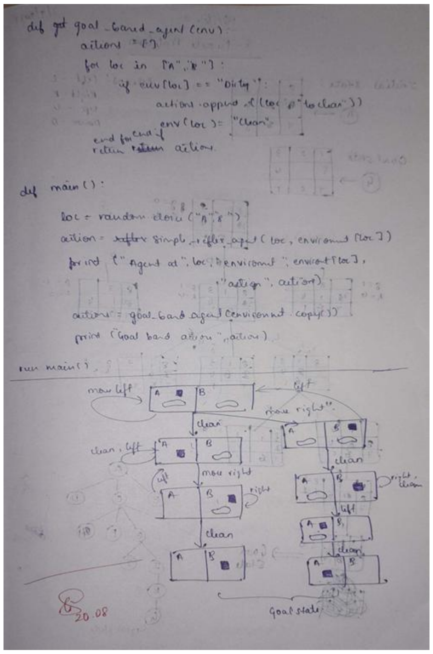
2. Vacuum cleaner

← initial state
[A: dirty, B: dirty]
vc in A

Action : if dirty : clean.
move left / right

A
B
cleanit
move left
move right
clean it
move right
move left
move right
Goal state

```
algorithm :

environment = {
    "A" : random.choice (["clean", "Dirty"]).
    "B" : random.choice (["clean", "dirty"])
}

def simple_reflex_agent (loc, status):
    if status == "dirty" :
        return "cleanit"
    endif
    elif loc == "A"
        ~~move~~ return "right"
    elif loc == "B" return "left"
```



```
def ~~get~~ goal_based_agent (env)
    actions = []
    for loc in ["A", "B"] :
        if env[loc] == "Dirty" :
            actions.append ((loc, "to clean"))
            env (loc) = "clean"
        endif
    end for
    return ~~return~~ actions

def main () :
    loc = random.choice ("A", "B")
    action = ~~reflex~~ simple_reflex_agent ( loc, environment [loc])
    print ("Agent at", loc, "environment", environment [loc],
                "action", action)
    actions = goal_based_agent (environment.copy())
    print ("Goal based actions", actions)

run main()
```

20.08

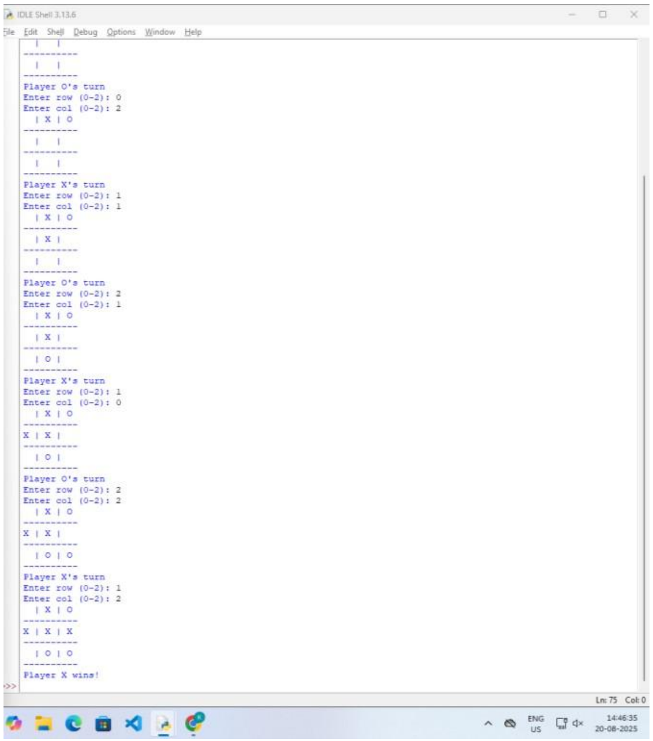
Code: #tic_tac_toe def
print_board(board):

```
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)
```

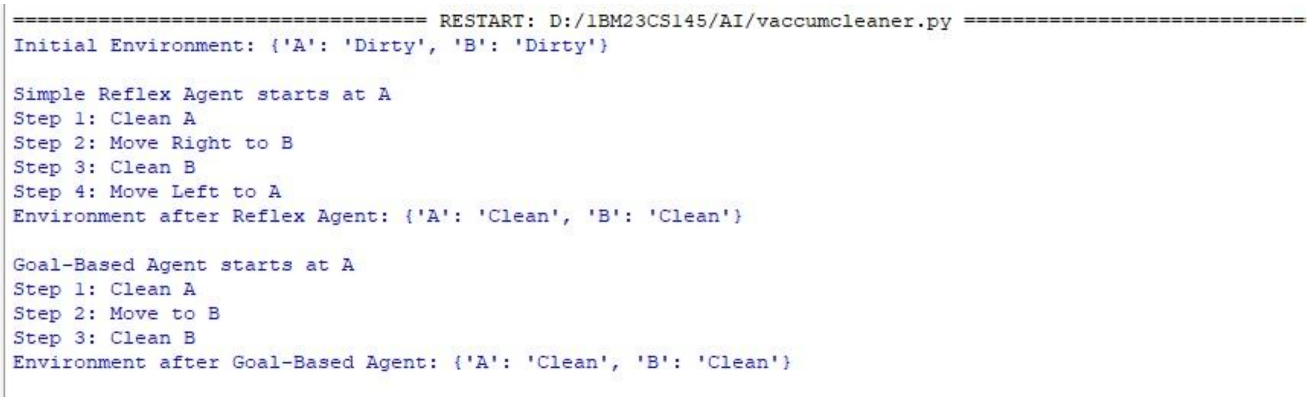
```
    print("It's a draw!")

# 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:



```
  |   |
----------
  |   |
----------
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!
>>>
```

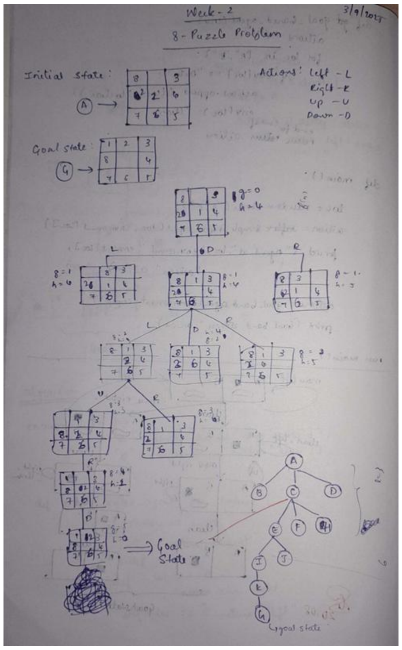


```
================================== RESTART: D:/1BM23CS145/AI/vaccumcleaner.py ============================
Initial Environment: {'A': 'Dirty', 'B': 'Dirty'}

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:



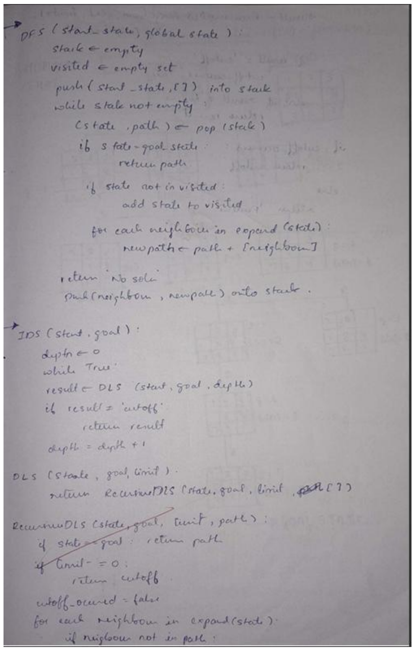
Week-2 3/9/2025

8-Puzzle Problem

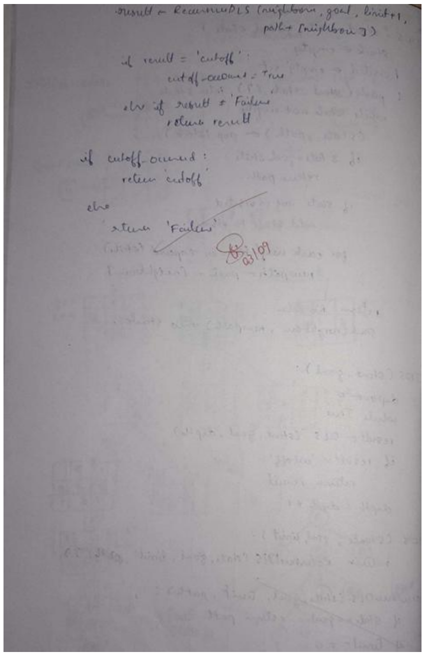
Initial state:

Actions: Left - L
Right - R
Up - U
Down - D

Goal state:



```
DFS (start_state, global state ):
    stack ← empty
    visited ← empty set
    push ( start_state, [ ] ) into stack
    while stack not empty:
        (state , path ) ← pop (stack)
        if state = goal state:
            return path

        if state not in visited:
            add state to visited

        for each neighbour in expand (state):
            newpath ← path + [neighbour]

    return 'No soln'
    push(neighbour , newpath) onto stack.

IDS (start , goal ):
    depth ← 0
    while True:
    result ← DLS (start , goal , depth)
    if result = 'cutoff'
        return result
    depth = depth + 1

DLS (state , goal, limit ):
    return RecursiveDLS (state, goal , limit , [ ])

RecursiveDLS (state, goal, limit , path ):
    if state = goal : return path
    if limit = 0 :
        return cutoff
    cutoff_occured = false
    for each neighbour in expand(state):
        if neighbour not in path :
```



```
            result ← RecursiveDLS (neighbour , goal , limit+1,
                                    path+ [neighbour ] )

            if result = 'cutoff' :
                cutoff_occured = True

            else if result ≠ 'Failure'
                return result

    if cutoff_occured :
        return 'cutoff'

    else
        return 'Failure'
```

03/09

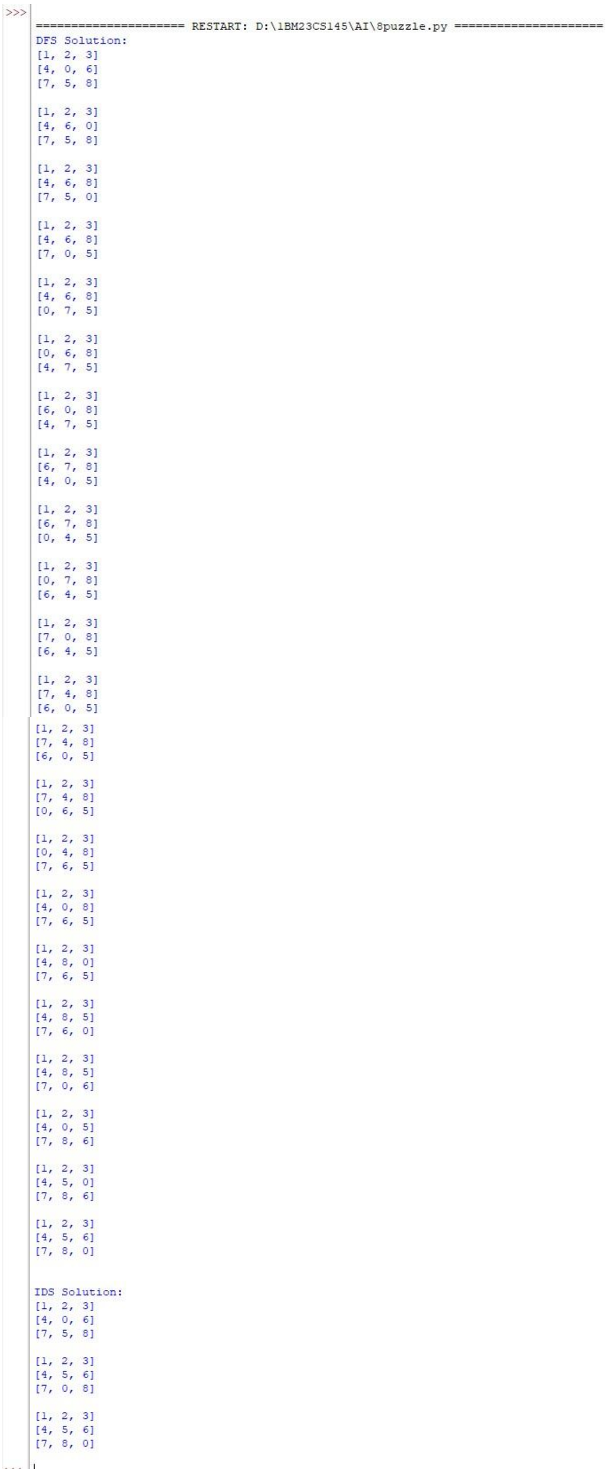
Code: from collections import
deque import copy

```
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:

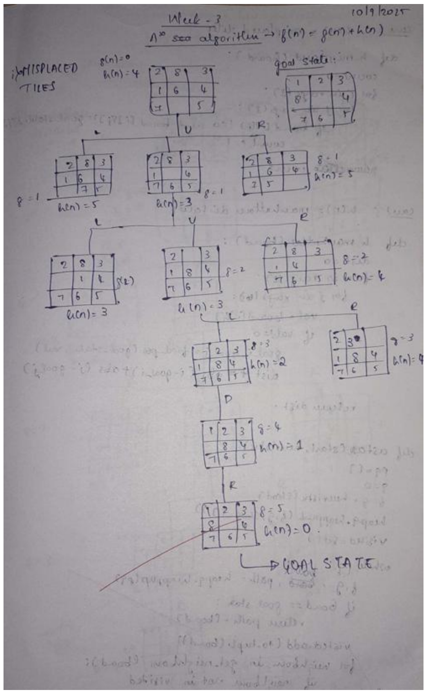


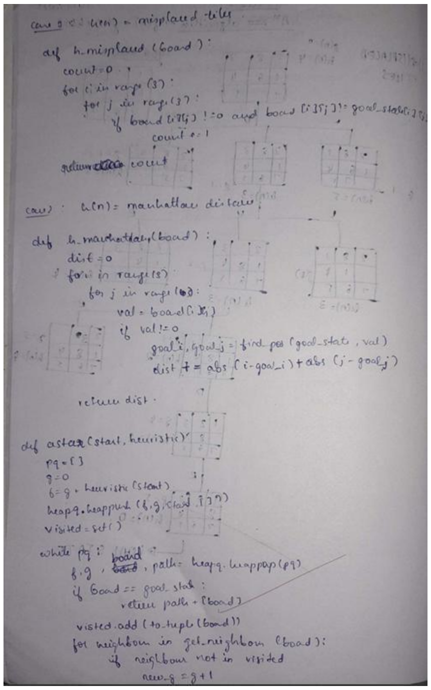
```
>>>
===================== RESTART: D:\1BM23CS145\AI\8puzzle.py =====================
DFS Solution:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

[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]
```

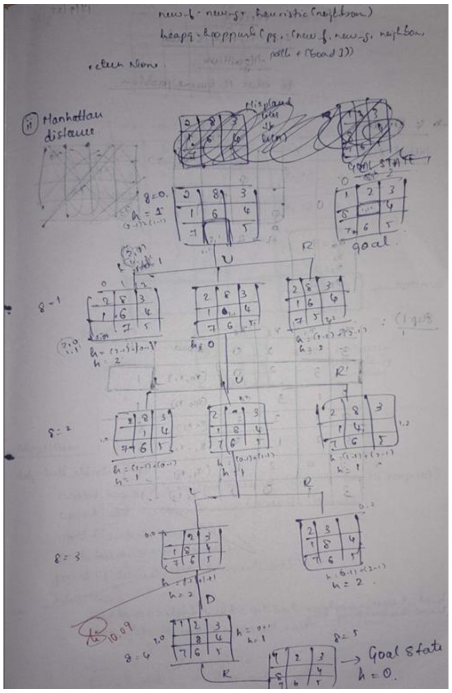
Program 3

Implement A* search algorithm

Algorithm:




```
case 1 :- h(n) = 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~~ return count

case2 :  h(n) = 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 astar (start, heuristic)
    pq = []
    g = 0
    f = g + heuristic (start)
    heapq.heappush (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 neighbour in get_neighbour (board):
            if neighbour not in visited
                new_g = g + 1
```


```
                new_f = new_g + heuristic (neighbour)
                heapq.heappush (pq, (new_f, new_g, neighbour,
                                     path + [board ]))

    return None
```

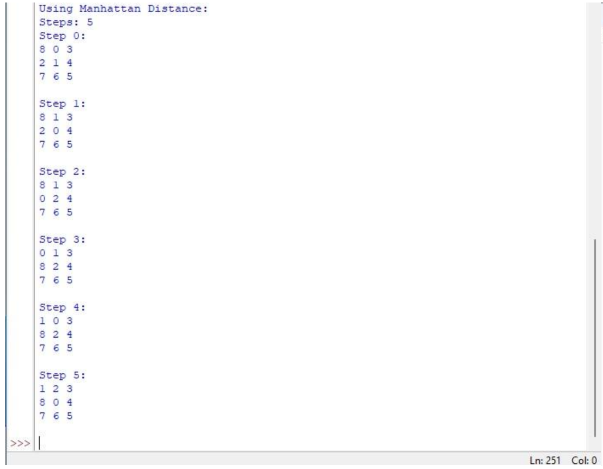
Code:

```
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
```

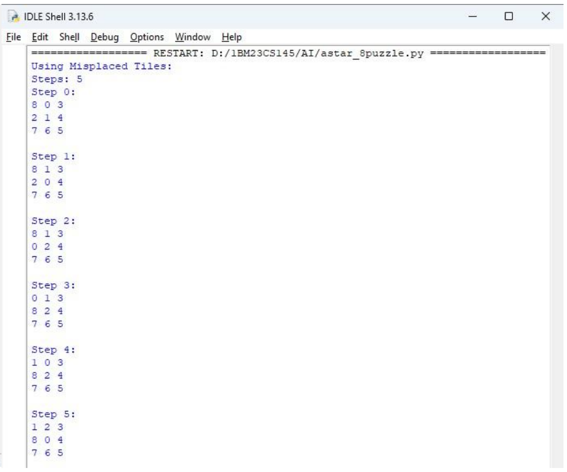
```
# Heuristic 2: manhattan distance def
h_manhattan(board):
   dist = 0     for i in
range(3):        for j in
range(3):            val =
board[i][j]            if
val != 0:
            goal_i, goal_j = find_pos(goal_state, val)
dist += abs(i - goal_i) + abs(j - goal_j)     return dist

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:



```
Using Manhattan Distance:
Steps: 5
Step 0:
8 0 3
2 1 4
7 6 5

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

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

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

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

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

>>> |
```

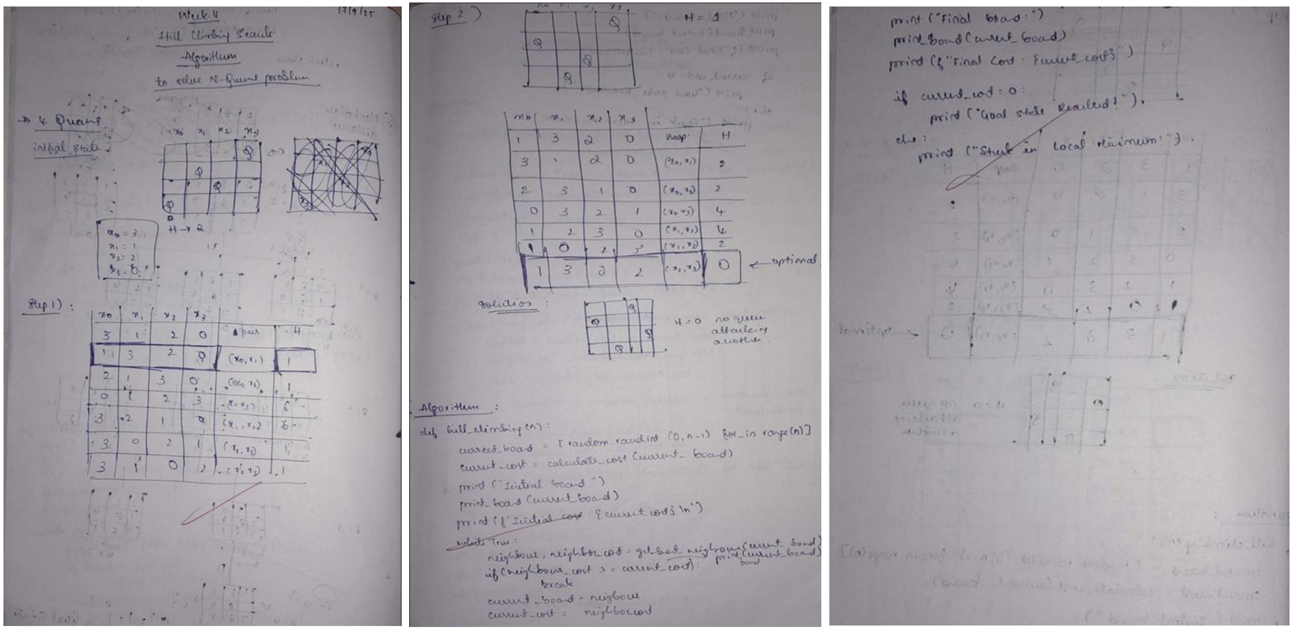


```
================== RESTART: D:/1BM23CS145/AI/astar_8puzzle.py ==================
Using Misplaced Tiles:
Steps: 5
Step 0:
8 0 3
2 1 4
7 6 5

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

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

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

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

Step 5:
1 2 3
8 0 4
7 6 5
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Week 4

17/9/25

Hill Climbing Search

Algorithm

to solve N-Queens problem

→ 4 Queens

initial state :

x0	x1	x2	x3
			Q
	Q		
		Q	
Q			

H → 2

x0 = 3
x1 = 1
x2 = 2
x3 = 0

Step 1) :

x0	x1	x2	x3		
3	1	2	0	pair	4
1	3	2	0	(x0, x1)	1
2	1	3	0	(x0 x2)	1
0	1	2	3	(x0 x3)	6
3	2	1	0	(x1, x2)	6
3	0	2	1	(x1, x3)	1
3	1	0	2	(x2 x3)	1

Step 2)

x0	x1	x2	x3
			Q
Q			
		Q	
	Q		

H = 1

x0	x1	x2	x3		
1	3	2	0	swap	H
3	1	2	0	(x0, x1)	2
2	3	1	0	(x0, x2)	2
0	3	2	1	(x0 x3)	4
1	2	3	0	(x1, x2)	6
1	0	2	3	(x1, x2)	2
1	3	0	2	(x2, x3)	0 ← optimal

Solution :

		Q	
Q			
			Q
	Q		

H = 0 no queen attacking another.

Algorithm :

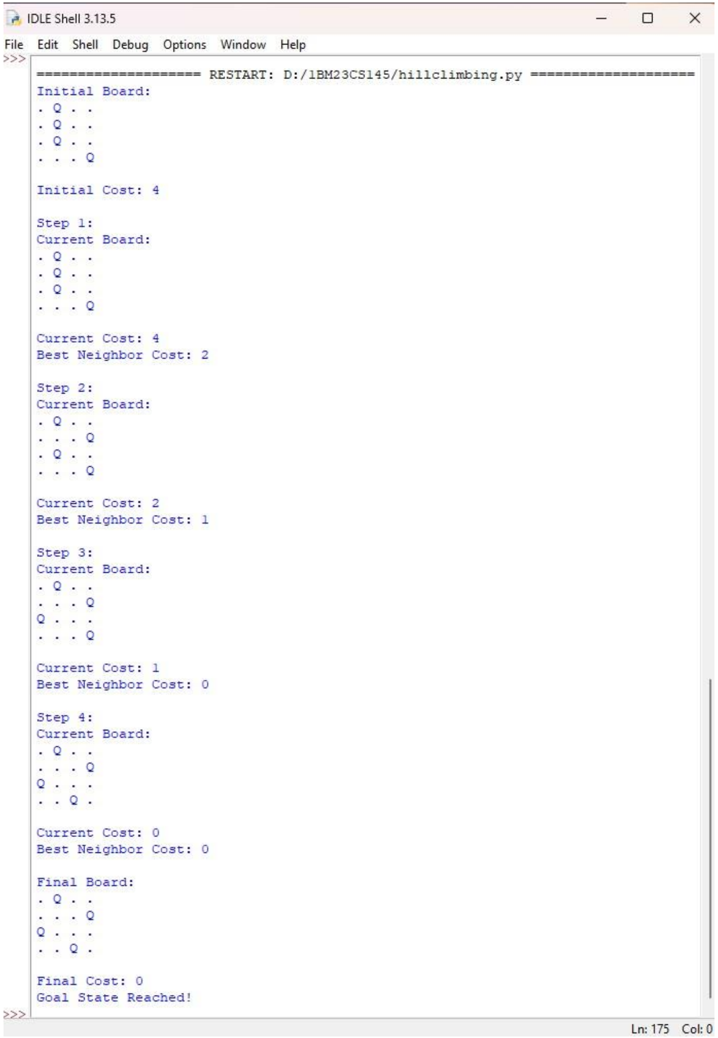
```
def hill_climbing(n):
    current_board = [random.randint (0, n-1) for_in range(n)]
    current_cost = calculate_cost (current_board)
    print ("Initial board ")
    print_board (current_board)
    print (f"Initial cost {current_cost} \n")
    while True :
        neighbour, neighbor_cost = get_best_neighbour(current_board)
        if (neighbour_cost >= current_cost):        print(current_board)
            break
        current_board = neighbour
        current_cost = neighborcost
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!")
```

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:

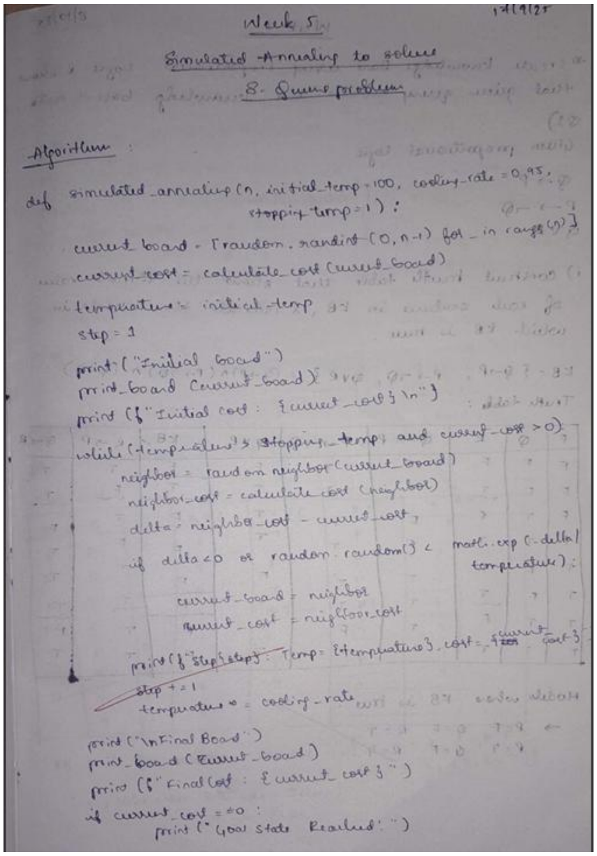


```
>>>
==================== RESTART: D:/1BM23CS145/hillclimbing.py ====================
Initial Board:
. Q . .
. Q . .
. Q . .
. . . Q

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:



17/9/25

Week 5

Simulated Annealing to solve

8- Queens problem

Algorithm :

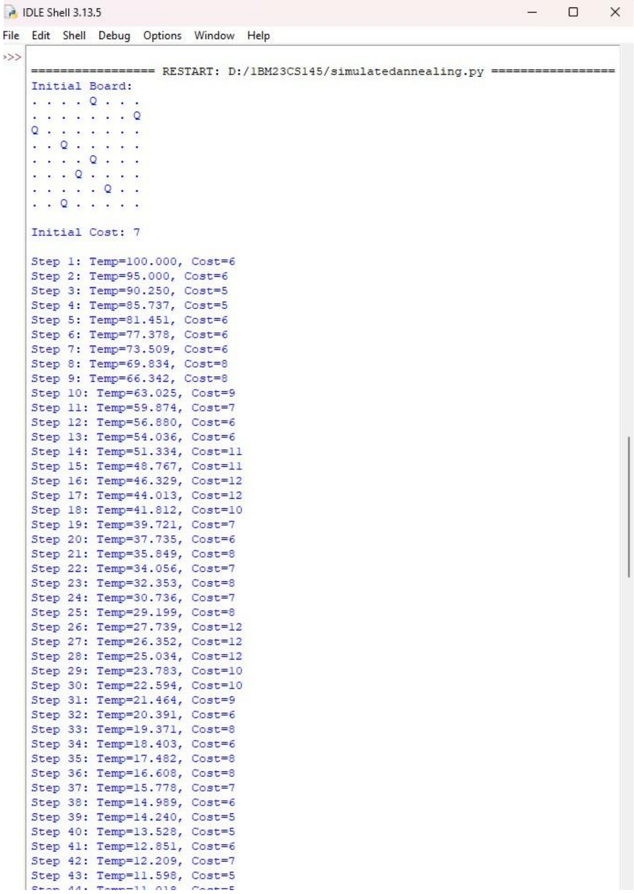
```
def simulated_annealing (n, initial_temp = 100, cooling_rate = 0.95,
                         stopping_temp = 1) :
    current board = [random. randint (0, n-1) for _ in range(n)]
    current_cost = calculate_cost (current_board)
    temperature = initial_temp
    step = 1
    print ("Initial board")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

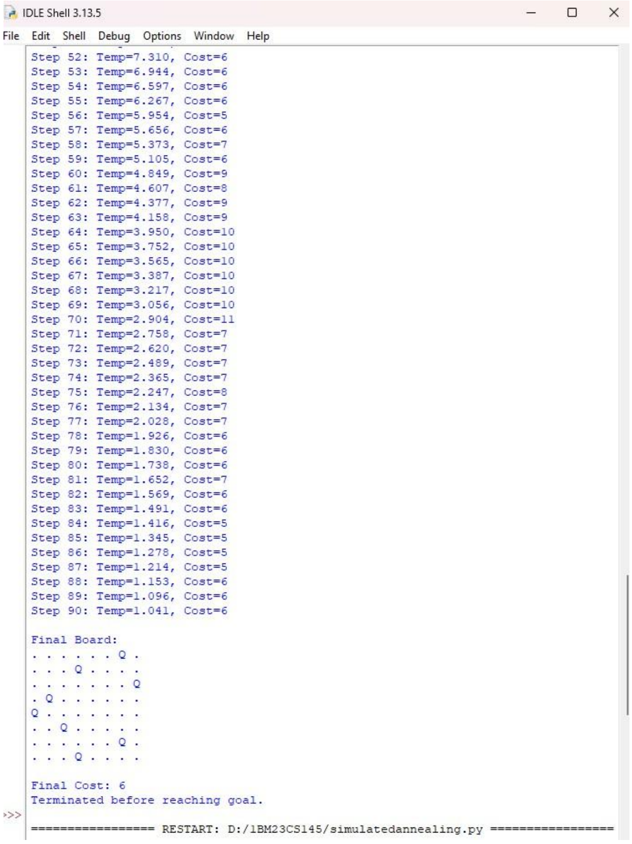
```
import random
import math

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:



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

Initial Cost: 7

Step 1: Temp=100.000, Cost=6
Step 2: Temp=95.000, Cost=6
Step 3: Temp=90.250, Cost=5
Step 4: Temp=85.737, Cost=5
Step 5: Temp=81.451, Cost=6
Step 6: Temp=77.378, Cost=6
Step 7: Temp=73.509, Cost=6
Step 8: Temp=69.834, Cost=8
Step 9: Temp=66.342, Cost=8
Step 10: Temp=63.025, Cost=9
Step 11: Temp=59.874, Cost=7
Step 12: Temp=56.880, Cost=6
Step 13: Temp=54.036, Cost=6
Step 14: Temp=51.334, Cost=11
Step 15: Temp=48.767, Cost=11
Step 16: Temp=46.329, Cost=12
Step 17: Temp=44.013, Cost=12
Step 18: Temp=41.812, Cost=10
Step 19: Temp=39.721, Cost=7
Step 20: Temp=37.735, Cost=6
Step 21: Temp=35.849, Cost=8
Step 22: Temp=34.056, Cost=7
Step 23: Temp=32.353, Cost=8
Step 24: Temp=30.736, Cost=7
Step 25: Temp=29.199, Cost=8
Step 26: Temp=27.739, Cost=12
Step 27: Temp=26.352, Cost=12
Step 28: Temp=25.034, Cost=12
Step 29: Temp=23.783, Cost=10
Step 30: Temp=22.594, Cost=10
Step 31: Temp=21.464, Cost=9
Step 32: Temp=20.391, Cost=6
Step 33: Temp=19.371, Cost=8
Step 34: Temp=18.403, Cost=6
Step 35: Temp=17.482, Cost=8
Step 36: Temp=16.608, Cost=8
Step 37: Temp=15.778, Cost=7
Step 38: Temp=14.989, Cost=6
Step 39: Temp=14.240, Cost=5
Step 40: Temp=13.528, Cost=5
Step 41: Temp=12.851, Cost=6
Step 42: Temp=12.209, Cost=7
Step 43: Temp=11.598, Cost=5
```

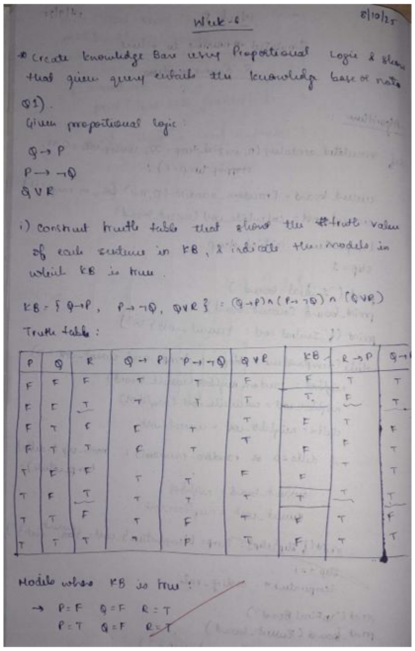


```
Step 52: Temp=7.310, Cost=6
Step 53: Temp=6.944, Cost=6
Step 54: Temp=6.597, Cost=6
Step 55: Temp=6.267, Cost=6
Step 56: Temp=5.954, Cost=5
Step 57: Temp=5.656, Cost=6
Step 58: Temp=5.373, Cost=7
Step 59: Temp=5.105, Cost=6
Step 60: Temp=4.849, Cost=9
Step 61: Temp=4.607, Cost=8
Step 62: Temp=4.377, Cost=9
Step 63: Temp=4.158, Cost=9
Step 64: Temp=3.950, Cost=10
Step 65: Temp=3.752, Cost=10
Step 66: Temp=3.565, Cost=10
Step 67: Temp=3.387, Cost=10
Step 68: Temp=3.217, Cost=10
Step 69: Temp=3.056, Cost=10
Step 70: Temp=2.904, Cost=11
Step 71: Temp=2.758, Cost=7
Step 72: Temp=2.620, Cost=7
Step 73: Temp=2.489, Cost=7
Step 74: Temp=2.365, Cost=7
Step 75: Temp=2.247, Cost=8
Step 76: Temp=2.134, Cost=7
Step 77: Temp=2.028, Cost=7
Step 78: Temp=1.926, Cost=6
Step 79: Temp=1.830, Cost=6
Step 80: Temp=1.738, Cost=6
Step 81: Temp=1.652, Cost=7
Step 82: Temp=1.569, Cost=6
Step 83: Temp=1.491, Cost=6
Step 84: Temp=1.416, Cost=5
Step 85: Temp=1.345, Cost=5
Step 86: Temp=1.278, Cost=5
Step 87: Temp=1.214, Cost=5
Step 88: Temp=1.153, Cost=6
Step 89: Temp=1.096, Cost=6
Step 90: Temp=1.041, Cost=6

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

Final Cost: 6
Terminated before reaching goal.

================= RESTART: D:/1BM23CS145/simulatedannealing.py =================
```

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:

$Q \to P$

$P \to \neg Q$

$Q \vee R$

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

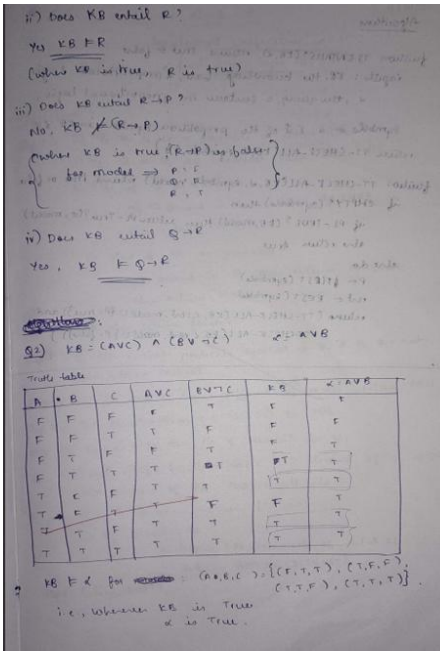
$KB = \{ Q \to P,\ P \to \neg Q,\ Q \vee R \} = (Q \to P) \wedge (P \to \neg Q) \wedge (Q \vee R)$

Truth table:

P	Q	R	$Q \to P$	$P \to \neg Q$	$Q \vee R$	KB	$R \to P$	$Q \to$
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 $KB \vDash R$

(when KB is true, R is true)

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

No, $KB \nvDash (R \to P)$

(when KB is true, $(R \to P)$ is false for model ⇒ P = F, Q = F, R = T)

iv) Does KB entail $Q \to R$

Yes, $KB \vDash Q \to R$

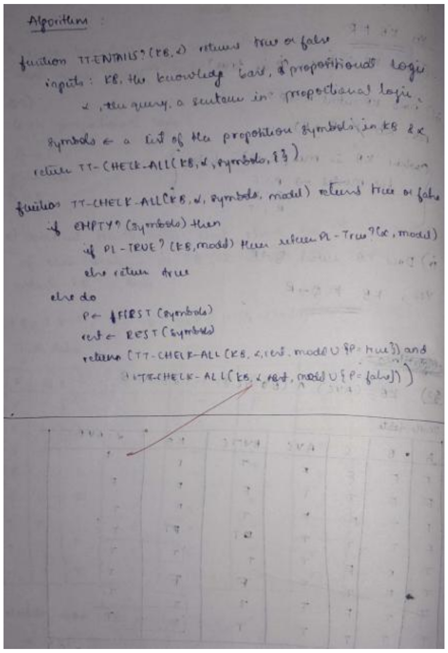
Q2) $KB = (A \vee C) \wedge (B \vee \neg C)$ $\alpha = A \vee B$

Truth table

A	B	C	$A \vee C$	$B \vee \neg C$	KB	$\alpha = A \vee B$
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

$KB \vDash \alpha$ 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}))
```

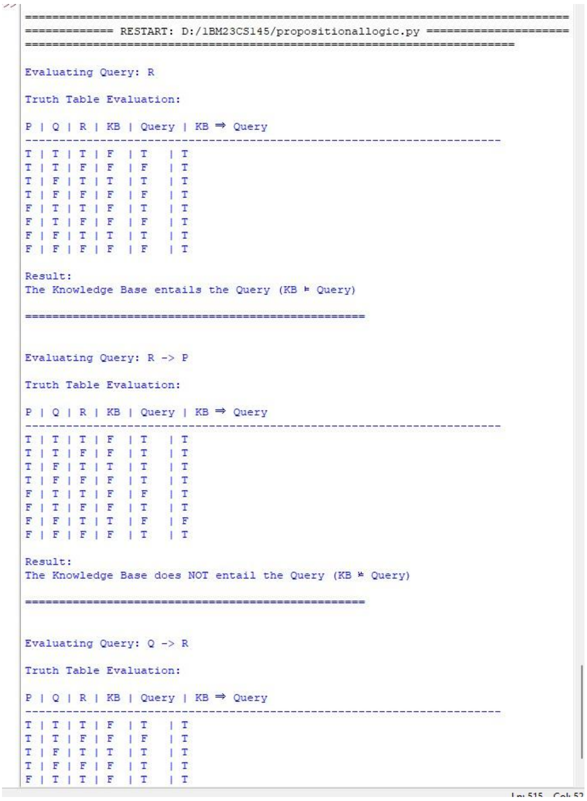
Code:

```
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 ")
```

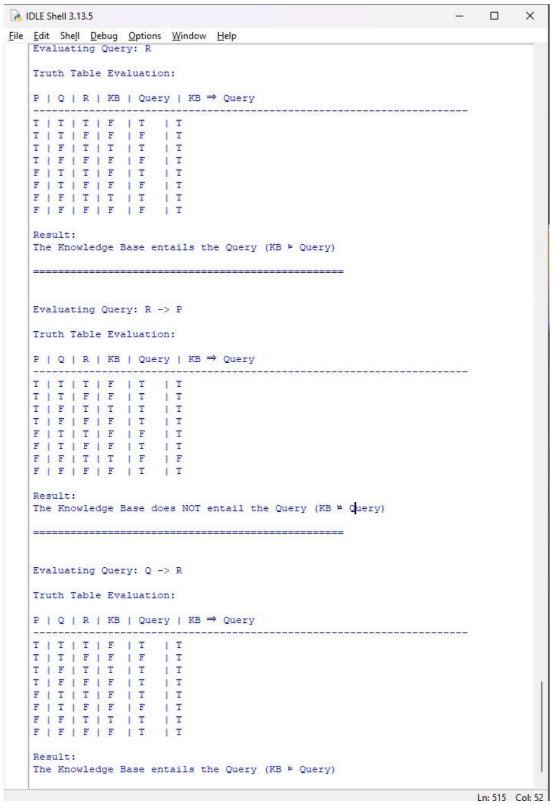
```
    # Replace symbols with their boolean values in the model
for sym, val in model.items():
        expr = re.sub(r'\b' + re.escape(sym) + r'\b', str(val), expr)

    # 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:



```
==========================================================================================
============= RESTART: D:/1BM23CS145/propositionallogic.py ===============================
==========================================================================================

Evaluating Query: R

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
----------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | F     | T
T | F | T | T  | T     | T
T | F | F | F  | F     | T
F | T | T | F  | T     | T
F | T | F | F  | F     | T
F | F | T | T  | T     | T
F | F | F | F  | F     | T

Result:
The Knowledge Base entails the Query (KB ⊨ Query)

==================================================


Evaluating Query: R -> P

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
----------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | T     | T
T | F | T | T  | T     | T
T | F | F | F  | T     | T
F | T | T | F  | F     | T
F | T | F | F  | T     | T
F | F | T | T  | F     | F
F | F | F | F  | T     | T

Result:
The Knowledge Base does NOT entail the Query (KB ⊭ Query)

==================================================


Evaluating Query: Q -> R

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
----------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | F     | T
T | F | T | T  | T     | T
T | F | F | F  | T     | T
F | T | T | F  | T     | T
```

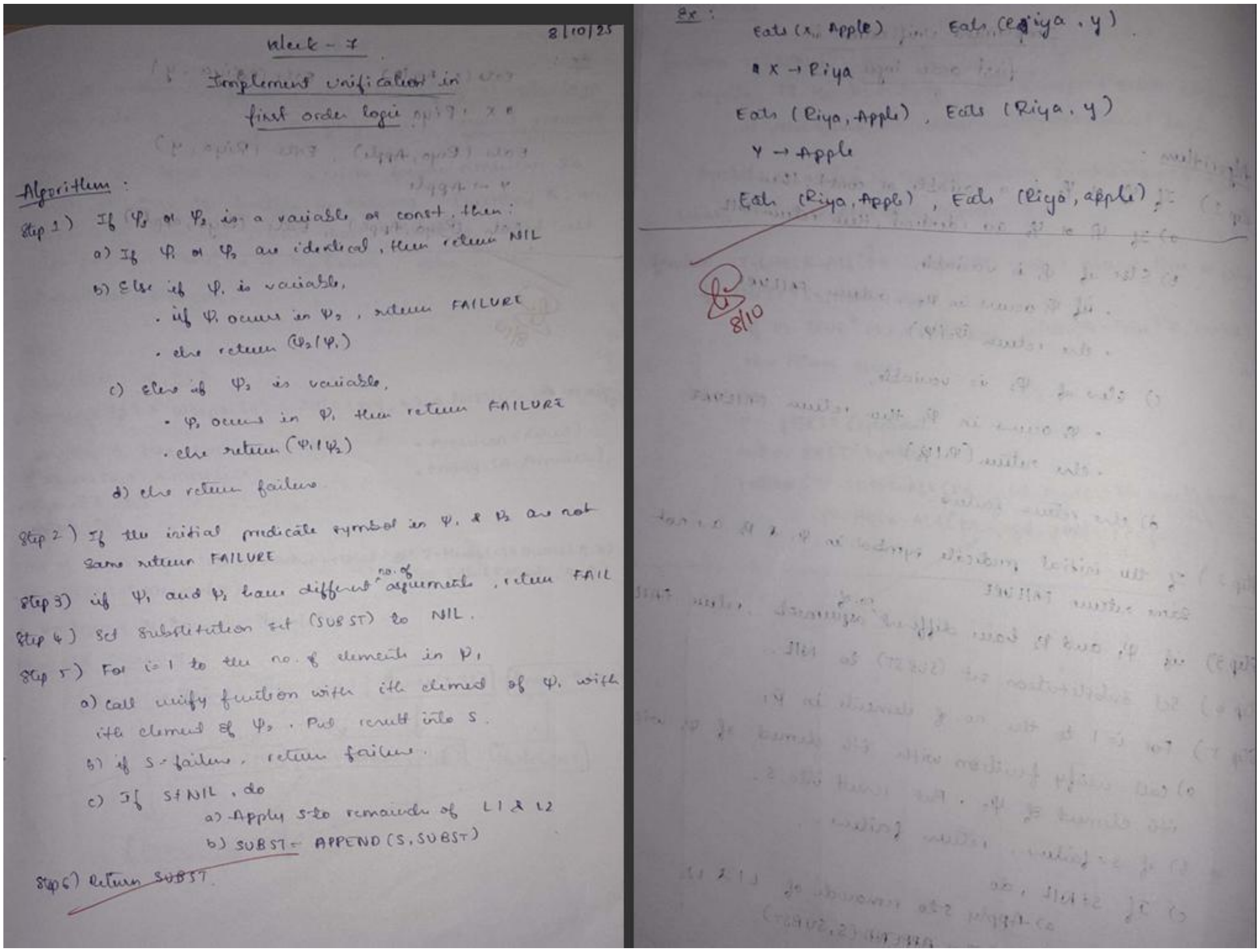


```
Evaluating Query: R

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
----------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | F     | T
T | F | T | T  | T     | T
T | F | F | F  | F     | T
F | T | T | F  | T     | T
F | T | F | F  | F     | T
F | F | T | T  | T     | T
F | F | F | F  | F     | T

Result:
The Knowledge Base entails the Query (KB ⊨ Query)

==================================================


Evaluating Query: R -> P

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
----------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | T     | T
T | F | T | T  | T     | T
T | F | F | F  | T     | T
F | T | T | F  | F     | T
F | T | F | F  | T     | T
F | F | T | T  | F     | F
F | F | F | F  | T     | T

Result:
The Knowledge Base does NOT entail the Query (KB ⊭ Query)

==================================================


Evaluating Query: Q -> R

Truth Table Evaluation:

P | Q | R | KB | Query | KB ⇒ Query
----------------------------------------------------------------------
T | T | T | F  | T     | T
T | T | F | F  | F     | T
T | F | T | T  | T     | T
T | F | F | F  | T     | T
F | T | T | F  | T     | T
F | T | F | F  | F     | T
F | F | T | T  | T     | T
F | F | F | F  | T     | T

Result:
The Knowledge Base entails the Query (KB ⊨ Query)
```

Program 7

Implement unification in first order logic

Algorithm:



8/10/25

Week - 7

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)

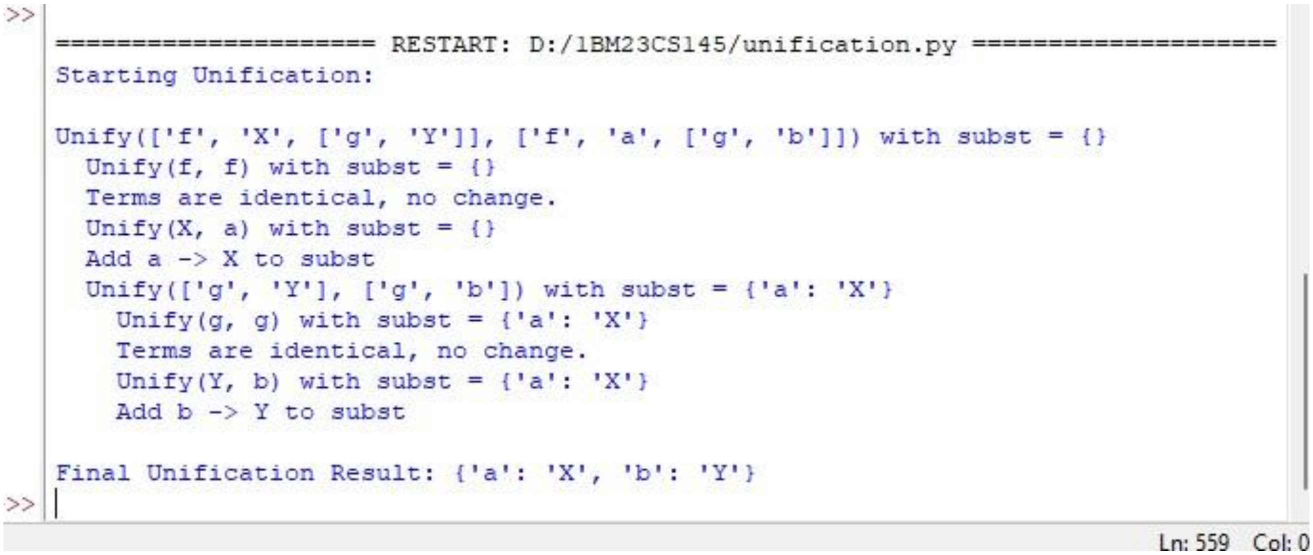
8/10

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:

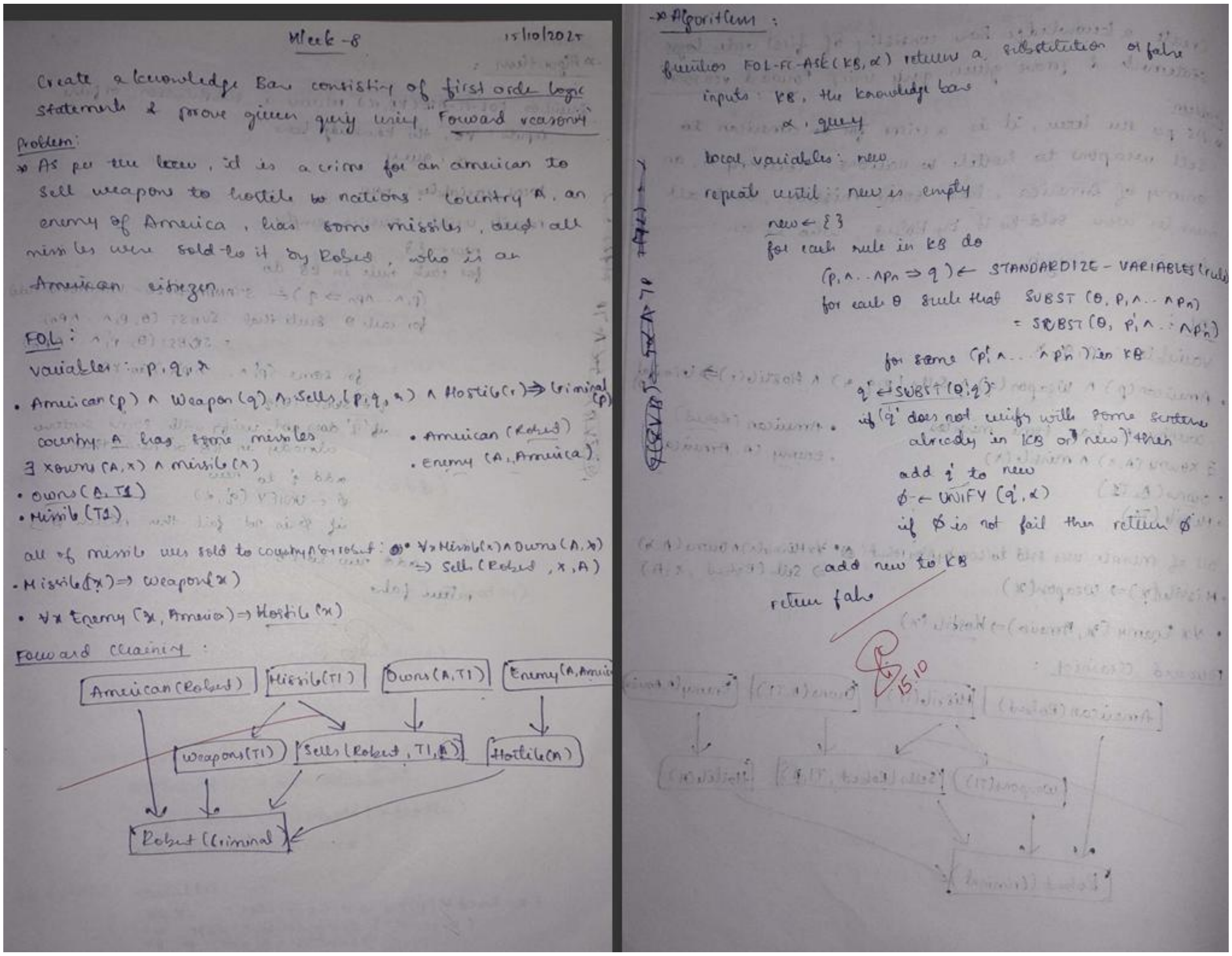


```
>>
    ==================== RESTART: D:/1BM23CS145/unification.py ====================
    Starting Unification:

    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'}
>>
```

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 missile 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 :

-* 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

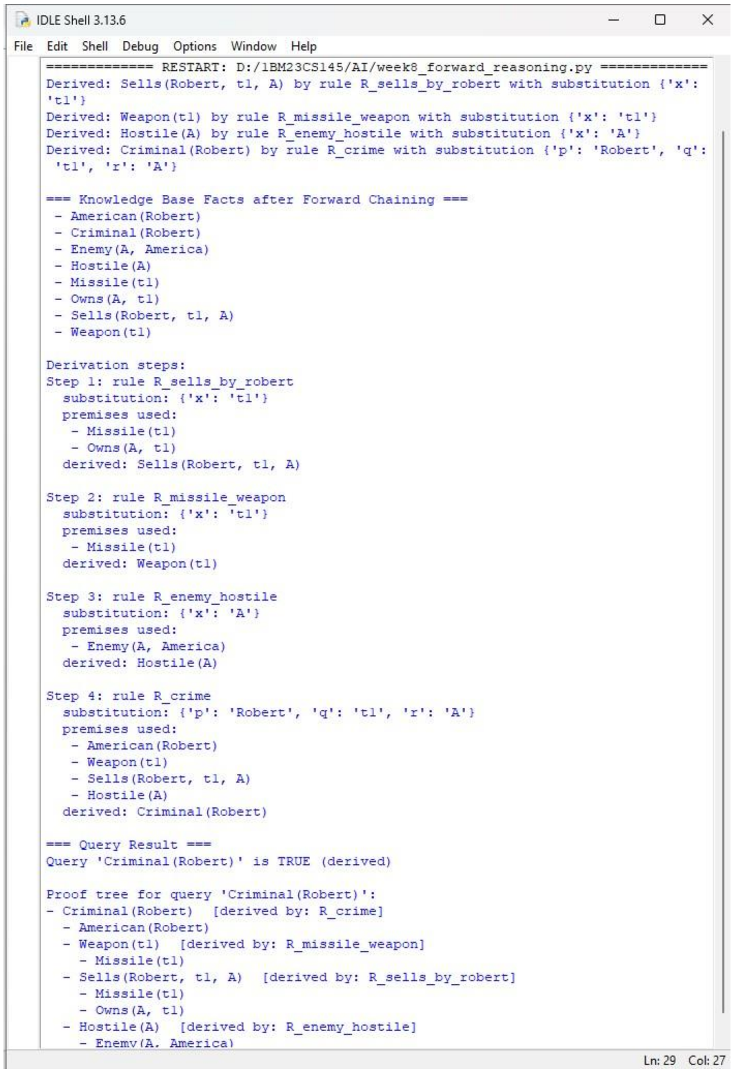
```
class KnowledgeBase:
def     __init__(self):
self.facts = set()
        self.rules = []

    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:

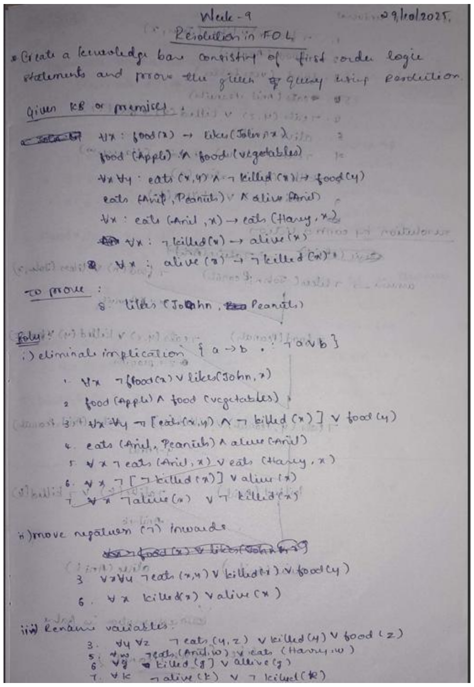


```
============= RESTART: D:/1BM23CS145/AI/week8_forward_reasoning.py =============
Derived: Sells(Robert, t1, A) by rule R_sells_by_robert with substitution {'x':
'tl'}
Derived: Weapon(tl) by rule R_missile_weapon with substitution {'x': 'tl'}
Derived: Hostile(A) by rule R_enemy_hostile with substitution {'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 ~~q~~ query using Resolution.

Given KB or premises :

$\forall x : food(x) \rightarrow likes(John, x)$

$food(Apple) \wedge food(vegetables)$

$\forall x \forall y : eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$

$eats(Anil, Peanuts) \wedge alive(Anil)$

$\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$

$\forall x : \neg killed(x) \rightarrow alive(x)$

$\forall x : alive(x) \rightarrow \neg killed(x)$

To prove :

8. likes (John, Peanuts)

Soln :

i) eliminate implication { a → b : ¬a ∨ b }

1. $\forall x \; \neg food(x) \vee likes(John, x)$
2. $food(Apple) \wedge food(vegetables)$
3. $\forall x \forall y \; \neg [eats(x,y) \wedge \neg killed(x)] \vee food(y)$
4. $eats(Anil, Peanuts) \wedge alive(Anil)$
5. $\forall x \; \neg eats(Anil, x) \vee eats(Harry, x)$
6. $\forall x \; \neg [\neg killed(x)] \vee alive(x)$
7. $\forall x \; \neg alive(x) \vee \neg killed(x)$

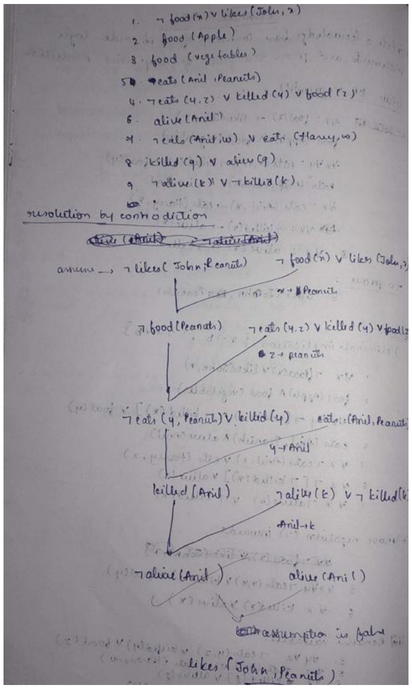
ii) move negation (¬) inwards

~~¬ food (x) ∨ likes(John, x)~~

3. $\forall x \forall y \; \neg eats(x,y) \vee killed(x) \vee food(y)$
6. $\forall x \; killed(x) \vee alive(x)$

iii) Rename variables

3. $\forall y \forall z \; \neg eats(y,z) \vee killed(y) \vee food(z)$
5. $\forall w \; \neg eats(Anil, w) \vee eats(Harry, w)$
6. $\forall g \; killed(g) \vee alive(g)$
7. $\forall k \; \neg alive(k) \vee \neg killed(k)$



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

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

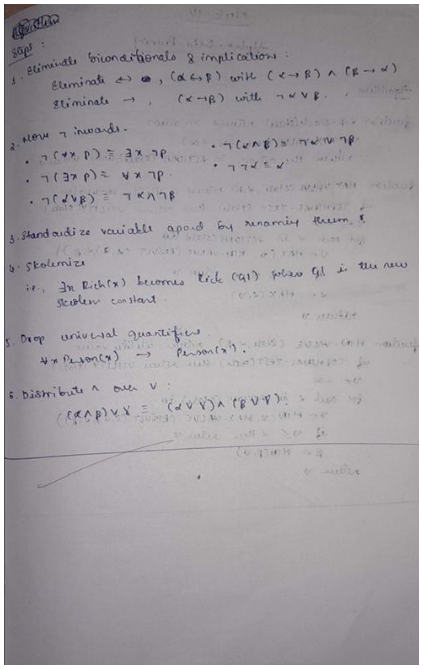
killed (Anil) ¬ alive (k) ∨ ¬ killed(k)

Anil → k

¬ alive (Anil) alive (Anil)

assumption is false

∴ likes (John, Peanuts)



Algorithm

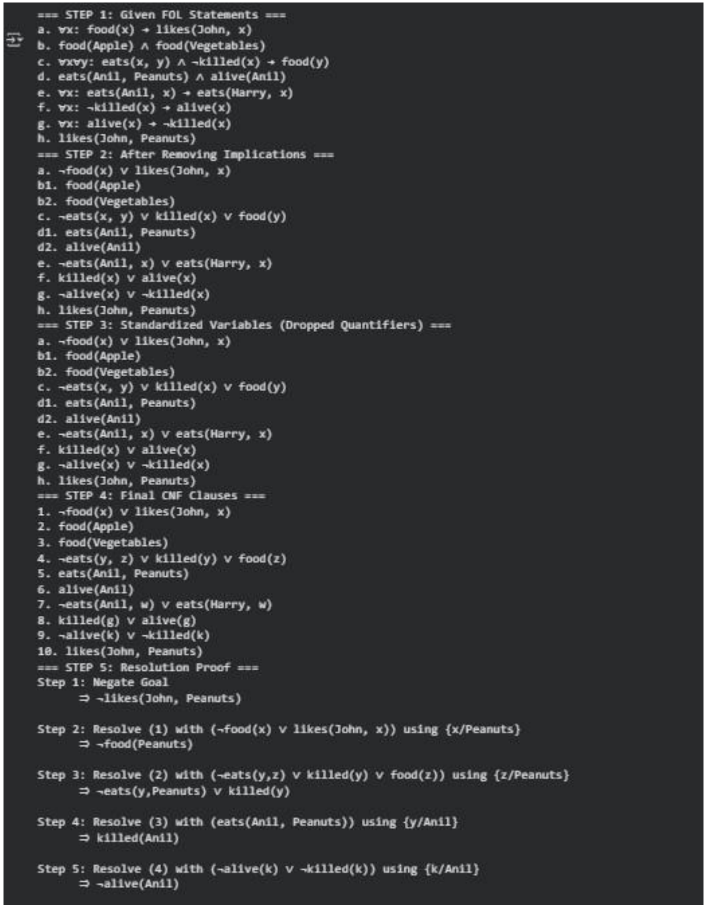
Steps :

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:

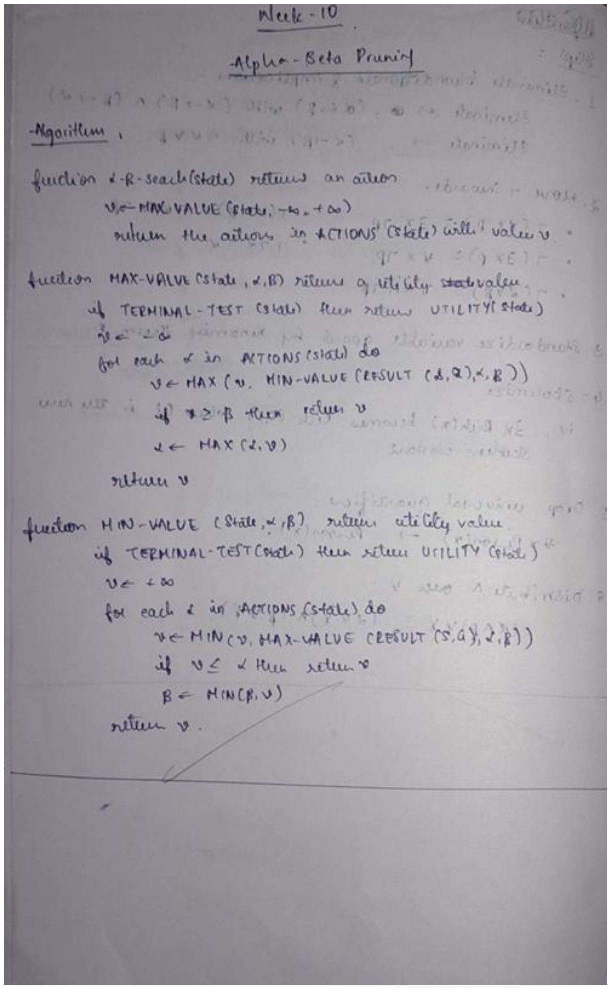


```
=== STEP 1: Given FOL Statements ===
a. ∀x: food(x) → likes(John, x)
b. food(Apple) ∧ food(Vegetables)
c. ∀x∀y: eats(x, y) ∧ ¬killed(x) → food(y)
d. eats(Anil, Peanuts) ∧ alive(Anil)
e. ∀x: eats(Anil, x) → eats(Harry, x)
f. ∀x: ¬killed(x) → alive(x)
g. ∀x: alive(x) → ¬killed(x)
h. likes(John, Peanuts)
=== 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)
```

<u>Program 10</u>

Implement Alpha-Beta Pruning.

Algorithm:

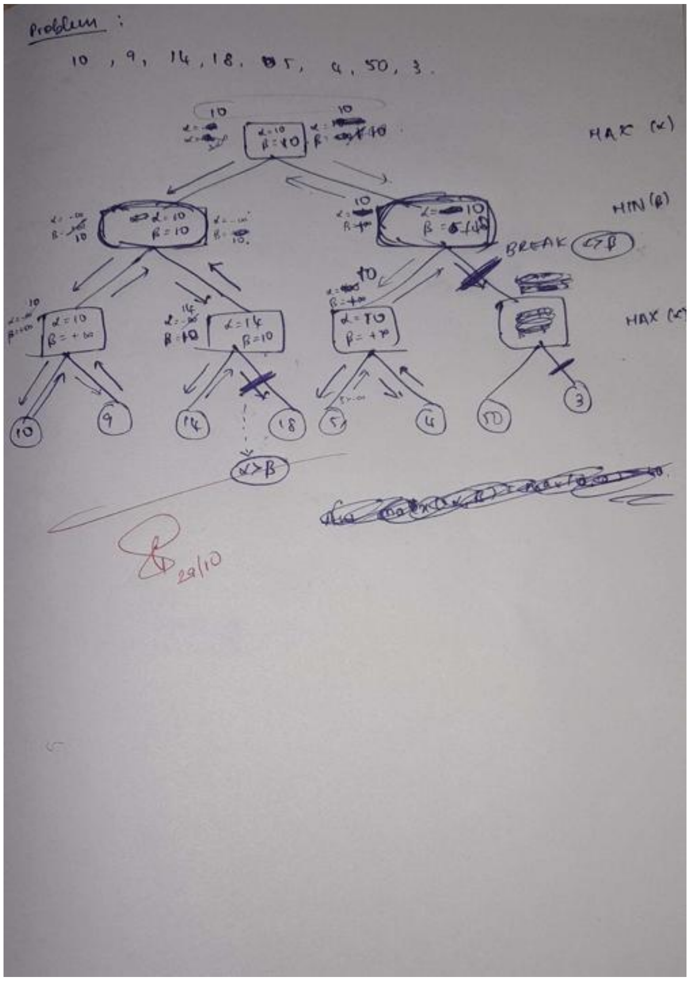


Week - 10

Alpha - Beta Pruning

Algorithm :

```
function α-β-search(state) returns an action
        v ← MAX-VALUE (state, -∞, +∞)
        return the actions in ACTIONS (state) with value v .

function MAX-VALUE (state , α, β) returns a utility ~~state~~ 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 .
```



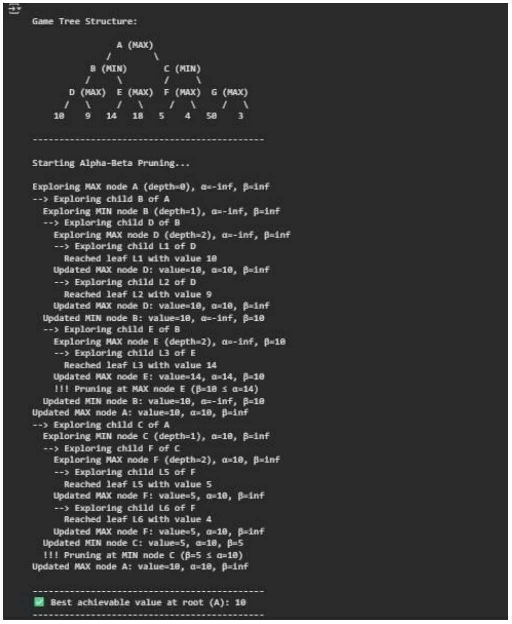
Problem :

10 , 9, 14 , 18, 5, 4 , 50, 3 .

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}Exploring 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
--------------------------------------------
```