VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by Srikar R Olety(1BM22CS289)

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

Sep-2024 to Jan-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 Srikar R Olety(1BM22CS289), 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.

|  |  |
| --- | --- |
| Rashmi H  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

Index

|  |  |  |  |
| --- | --- | --- | --- |
| Sl. No. | Date | Experiment Title | Page No. |
| 1 | 30-9-2024 | Implement Tic –Tac –Toe Game Implement vacuum cleaner agent | 1 |
| 2 | 7-10-2024 | Implement 8 puzzle problems using Depth First  Search (DFS)  Implement Iterative deepening search algorithm | 5 |
| 3 | 14-10-2024 | Implement A\* search algorithm | 17 |
| 4 | 21-10-2024 | Implement Hill Climbing search algorithm to solve N-Queens problem | 22 |
| 5 | 28-10-  2024 | Simulated Annealing to Solve 8Queens problem | 24 |
| 6 | 11-11-2024 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 27 |
| 7 | 2-12-2024 | Implement unification in first order logic | 31 |
| 8 | 2-12-2024 | Create a knowledge base consisting of irst order logic statements and prove the given query using forward reasoning. | 36 |
| 9 | 16-12-  2024 | Create a knowledge base consisting of irst order logic statements and prove the given query using Resolution | 40 |
| 10 | 16-12-2024 | Implement Alpha-Beta Pruning. | 43 |

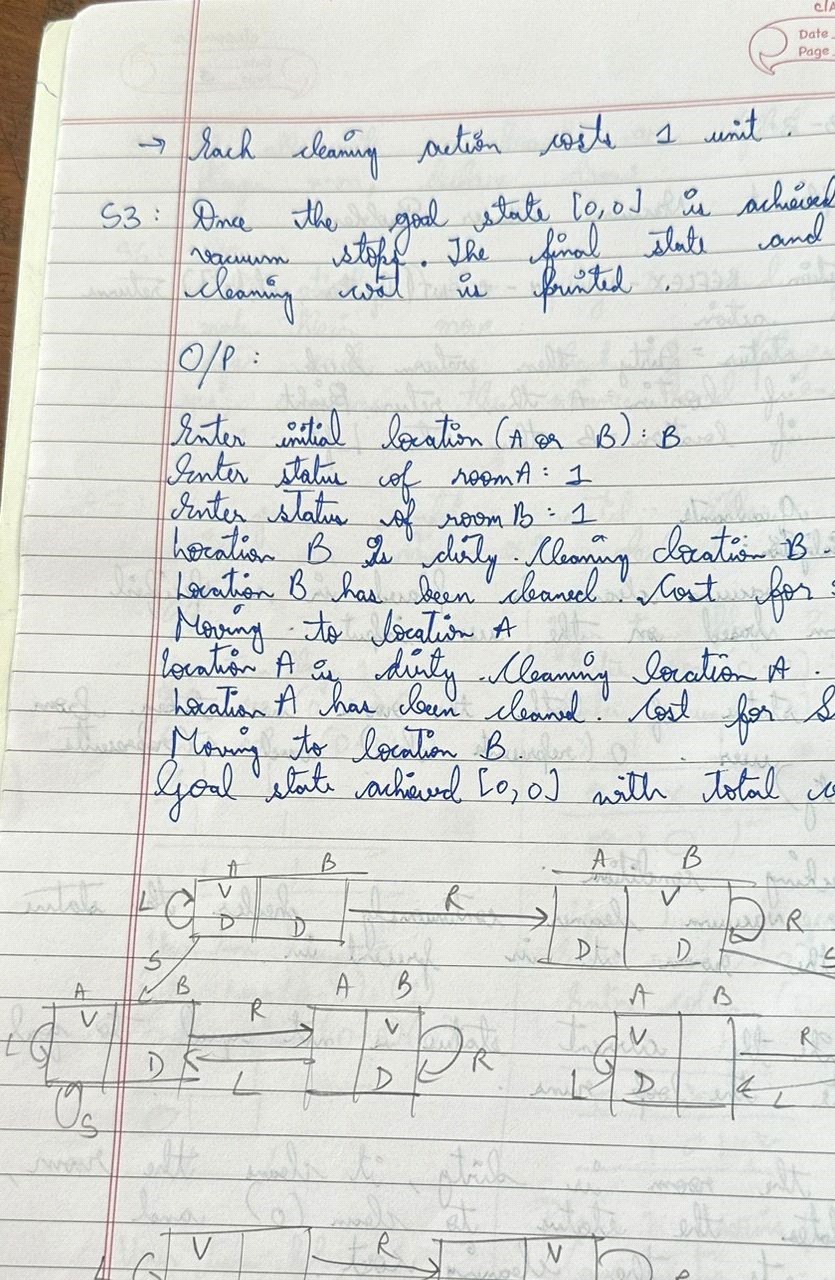
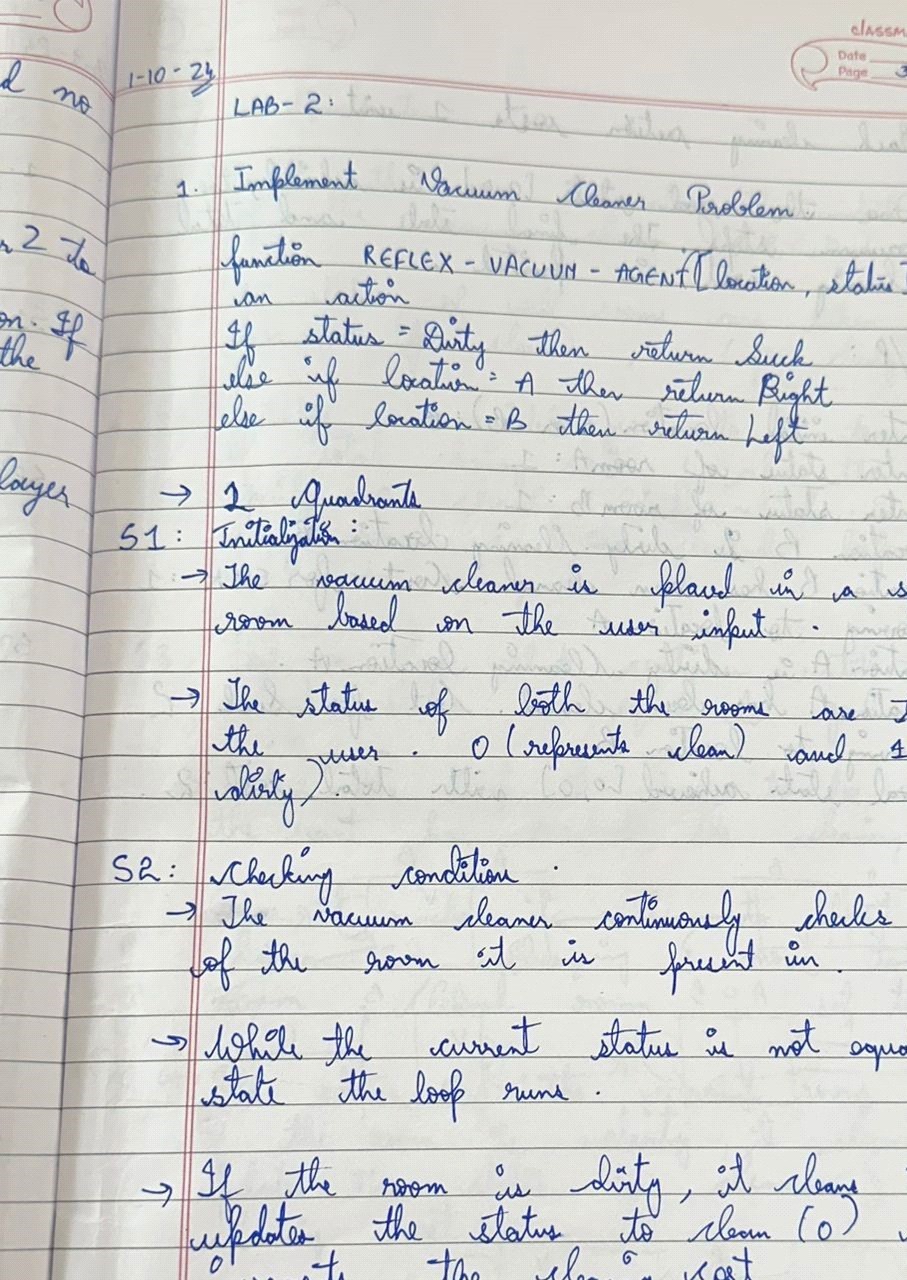
Github Link: https://github.com/Srikar2507/AI\_LAB

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



Code:

TicTacToe

board = [[' ' for \_ in range(3)] for \_ in range(3)] def draw\_board(): for row in board:

print(' | '.join(row)) print('---------') def player\_move(icon): if icon == 'X': number = 1 elif icon == 'O': number = 2

print("Your turn player {}".format(number))

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

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

if board[row][col] == ' ': board[row][col] = icon else:

print("That space is taken!") def is\_victory(icon): for row in board: if all(cell == icon for cell in row):

return True for col in range(3): if all(board[row][col] == icon for row in range(3)):

return True if all(board[i][i] == icon for i in range(3)) or \ all(board[i][2-i] == icon for i in range(3)):

return True

return False def is\_draw():

return all(cell != ' ' for row in board for cell in row) def play\_game(): draw\_board() while True: player\_move('X') draw\_board() if is\_victory('X'):

print("Player 1 wins! Congratulations!")

break elif is\_draw():

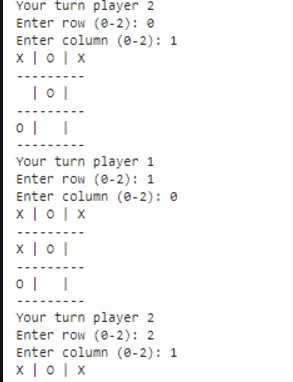
print("It's a draw!") break player\_move('O') draw\_board() if is\_victory('O'):

print("Player 2 wins! Congratulations!")

break elif is\_draw():

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

play\_game() Output:



Vacuum World-2Quad:

class VacuumCleaner: def \_\_init\_\_(self, location, status):

self.location = location self.status = status self.goal\_state = [0, 0] self.cost = 0

def clean(self): while self.status != self.goal\_state: if self.location == 'A': if self.status[0] == 1: print("Location A is Dirty. Cleaning Location A.") self.status[0] = 0 self.cost += 1 print(f"Location A has been Cleaned. COST for SUCK: {self.cost}") else:

print("Location A is already clean.") print("Moving to Location B.") self.location = 'B'

elif self.location == 'B': if self.status[1] == 1: print("Location B is Dirty. Cleaning Location B.") self.status[1] = 0 self.cost += 1 print(f"Location B has been Cleaned. COST for SUCK: {self.cost}") else:

print("Location B is already clean.") print("Moving to Location A.")

self.location = 'A'

print(f"Goal state achieved: {self.status} with total cost: {self.cost}")

def get\_user\_input():

location = input("Enter initial location (A or B): ").strip().upper() a\_status = int(input("Enter status of Room A (0 for clean, 1 for dirty): ")) b\_status = int(input("Enter status of Room B (0 for clean, 1 for dirty): ")) return location, [a\_status, b\_status]

def main():

location, status = get\_user\_input() vacuum = VacuumCleaner(location, status) vacuum.clean()

main()

Output:

Output1

Enter initial location (A or B): A

Enter status of Room A (0 for clean, 1 for dirty): 1 Enter status of Room B (0 for clean, 1 for dirty): 0 Location A is Dirty. Cleaning Location A. Location A has been Cleaned. COST for SUCK: 1 Moving to Location B.

Goal state achieved: [0, 0] with total cost: 1

Output2

Enter initial location (A or B): A

Enter status of Room A (0 for clean, 1 for dirty): 1 Enter status of Room B (0 for clean, 1 for dirty): 1 Location A is Dirty. Cleaning Location A. Location A has been Cleaned. COST for SUCK: 1 Moving to Location B.

Location B is Dirty. Cleaning Location B. Location B has been Cleaned. COST for SUCK: 2 Moving to Location A.

Goal state achieved: [0, 0] with total cost: 2

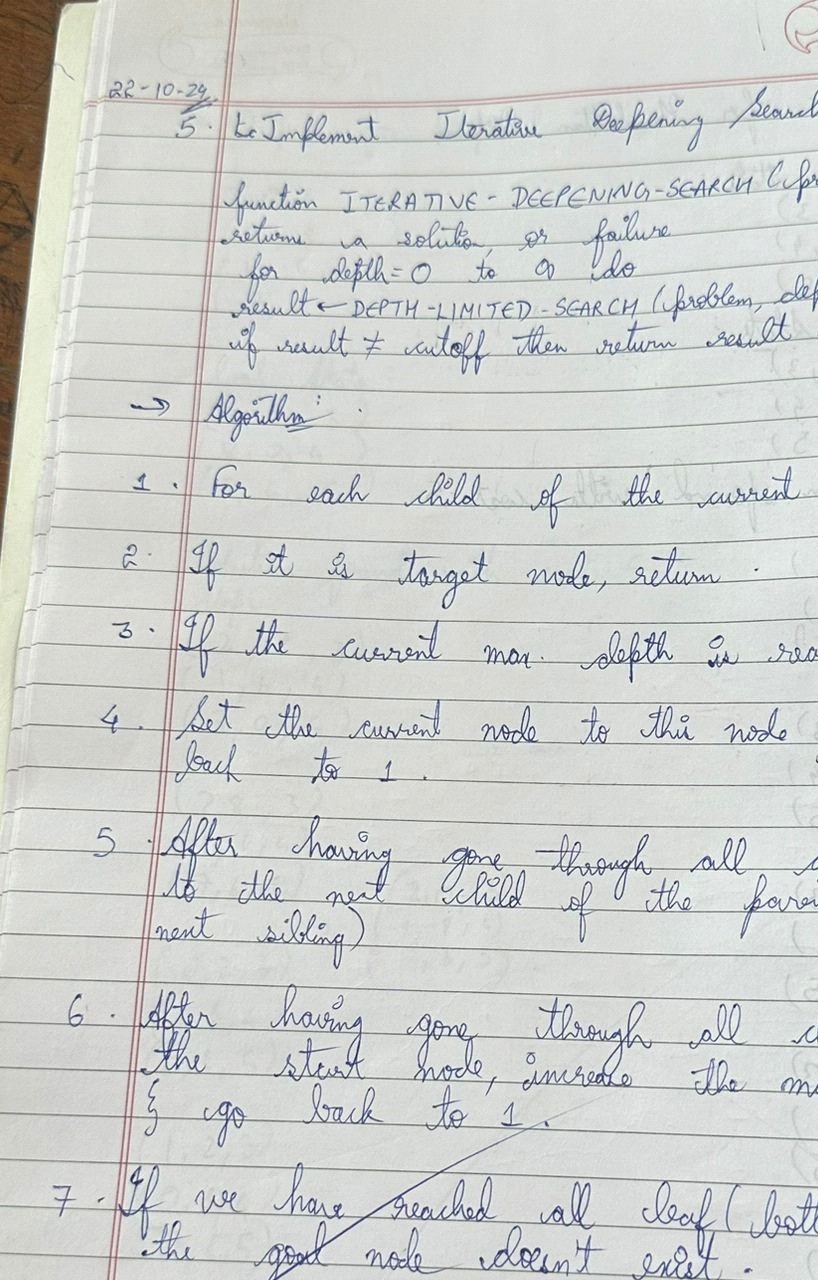
Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Implement Iterative deepening search algorithm

Algorithm:





Code:

DFS: from copy import deepcopy

goal\_state = [[0, 1, 2],

[3, 4, 5], [6, 7, 8]]

moves = {

'up': (-1, 0),

'down': (1, 0),

'left': (0, -1),

'right': (0, 1)

}

def find\_blank(board): for i in range(3): for j in range(3): if board[i][j] == 0:

return i, j return None

def is\_goal(state):

return state == goal\_state

def is\_valid\_move(x, y):

return 0 <= x < 3 and 0 <= y < 3

def apply\_move(board, move): x, y = find\_blank(board) dx, dy = moves[move] new\_x, new\_y = x + dx, y + dy if is\_valid\_move(new\_x, new\_y): new\_board = deepcopy(board) new\_board[x][y], new\_board[new\_x][new\_y] = new\_board[new\_x][new\_y], new\_board[x][y] return new\_board

return None

def dfs(start): stack = [(start, [], 0)] # (state, path of moves, number of states explored) visited = set() explored\_count = 0 # Count of unique states explored

while stack:

current\_state, path, current\_count = stack.pop()

if is\_goal(current\_state):

return path, explored\_count # Return path of moves and explored count

# Create a unique representation of the state to store in visited state\_tuple = tuple(tuple(row) for row in current\_state) if state\_tuple not in visited: visited.add(state\_tuple) # Add to visited states explored\_count += 1 for move in moves:

new\_state = apply\_move(current\_state, move) if new\_state: # Only consider valid new states

stack.append((new\_state, path + [move], explored\_count)) # Store the path and current count

return None, explored\_count # Return explored count if no solution is found

def print\_board(board): for row in board: print(row)

print()

def print\_solution(solution, explored\_count): if solution is not None:

print(f"Goal achieved with the following moves:")

current\_board = initial\_state for move in solution:

print(f"Move: {move}") current\_board = apply\_move(current\_board, move)

print\_board(current\_board) print(f"Total number of states explored: {explored\_count}") else:

print("No solution found")

initial\_state = [[1, 2, 5],

[3, 4, 8], [6, 7, 0]]

solution, explored\_count = dfs(initial\_state)

print\_solution(solution, explored\_count)

Iterative deepening Search: from copy import deepcopy

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

class PuzzleState: def \_\_init\_\_(self, board, parent=None, move=""):

self.board = board self.parent = parent

self.move = move

def get\_blank\_position(self):

for i in range(3): for j in range(3): if self.board[i][j] == 0:

return i, j

def generate\_successors(self): successors = [] x, y = self.get\_blank\_position()

for dx, dy in DIRECTIONS:

new\_x, new\_y = x + dx, y + dy

if 0 <= new\_x < 3 and 0 <= new\_y < 3: new\_board = deepcopy(self.board) new\_board[x][y], new\_board[new\_x][new\_y] = new\_board[new\_x][new\_y], new\_board[x][y] successors.append(PuzzleState(new\_board, parent=self))

return successors

def is\_goal(self, goal\_state):

return self.board == goal\_state

def \_\_str\_\_(self):

return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth\_limited\_search(current\_state, goal\_state, depth):

if depth == 0 and current\_state.is\_goal(goal\_state):

return current\_state

if depth > 0: for successor in current\_state.generate\_successors():

found = depth\_limited\_search(successor, goal\_state, depth - 1) if found: return found

return None

def iterative\_deepening\_search(start\_state, goal\_state):

depth = 0 while True:

print(f"\nSearching at depth level: {depth}") result = depth\_limited\_search(start\_state, goal\_state, depth) if result: return result

depth += 1

def get\_user\_input(): print("Enter the start state (use 0 for the blank):") start\_state = [] for \_ in range(3):

row = list(map(int, input().split())) start\_state.append(row)

print("Enter the goal state (use 0 for the blank):") goal\_state = [] for \_ in range(3):

row = list(map(int, input().split())) goal\_state.append(row)

return start\_state, goal\_state

def main():

start\_board, goal\_board = get\_user\_input() start\_state = PuzzleState(start\_board)

goal\_state = goal\_board

result = iterative\_deepening\_search(start\_state, goal\_state)

if result:

print("\nGoal reached!")

path = [] while result:

path.append(result) result = result.parent path.reverse() for state in path: print(state, "\n") else:

print("Goal state not found.")

if \_\_name\_\_ == "\_\_main\_\_": main()

Output:

Goal achieved with the following moves:

Move: left

[1, 2, 5]

[3, 4, 8]

[6, 0, 7]

Move: left

[1, 2, 5]

[3, 4, 8]

[0, 6, 7]

Move: up

[1, 2, 5]

[0, 4, 8]

[3, 6, 7]

Move: right

[1, 2, 5]

[4, 0, 8]

[3, 6, 7]

Move: right

[1, 2, 5]

[4, 8, 0]

[3, 6, 7]

Move: down

[1, 2, 5]

[4, 8, 7]

[3, 6, 0]

Move: left

[1, 2, 5]

[4, 8, 7]

[3, 0, 6]

Move: left

[1, 2, 5]

[4, 8, 7]

[0, 3, 6]

Move: up

[1, 2, 5]

[0, 8, 7]

[4, 3, 6]

Move: right

[1, 2, 5]

[8, 0, 7]

[4, 3, 6]

Move: right

[1, 2, 5]

[8, 7, 0]

[4, 3, 6]

Move: up

[1, 2, 5]

[0, 6, 3]

[7, 8, 4]

Move: right

[1, 2, 5]

[6, 0, 3]

[7, 8, 4]

Move: right

[1, 2, 5]

[6, 3, 0]

[7, 8, 4]

Move: down

[1, 2, 5]

[6, 3, 4]

[7, 8, 0] Move: left

[1, 2, 5]

[6, 3, 4]

[7, 0, 8]

Move: left

[1, 2, 5]

[6, 3, 4]

[0, 7, 8]

Move: up

[1, 2, 5]

[0, 3, 4]

[6, 7, 8]

Move: right

[1, 2, 5]

[3, 0, 4]

[6, 7, 8]

Move: right

[1, 2, 5]

[3, 4, 0]

[6, 7, 8]

Move: up

[1, 2, 0]

[3, 4, 5]

[6, 7, 8]

Move: left

[1, 0, 2]

[3, 4, 5]

[6, 7, 8]

Move: left

[0, 1, 2]

[3, 4, 5]

[6, 7, 8]

Total number of unique states explored: 32 Code:

Iterative Deepening Search

from copy import deepcopy

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

class PuzzleState: def \_\_init\_\_(self, board, parent=None, move=""):

self.board = board self.parent = parent

self.move = move

def get\_blank\_position(self):

for i in range(3): for j in range(3): if self.board[i][j] == 0:

return i, j

def generate\_successors(self):

successors = [] x, y = self.get\_blank\_position()

for dx, dy in DIRECTIONS:

new\_x, new\_y = x + dx, y + dy

if 0 <= new\_x < 3 and 0 <= new\_y < 3: new\_board = deepcopy(self.board) new\_board[x][y], new\_board[new\_x][new\_y] = new\_board[new\_x][new\_y], new\_board[x][y]

successors.append(PuzzleState(new\_board, parent=self))

return successors

def is\_goal(self, goal\_state):

return self.board == goal\_state

def \_\_str\_\_(self):

return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth\_limited\_search(current\_state, goal\_state, depth): if depth == 0 and current\_state.is\_goal(goal\_state):

return current\_state

if depth > 0:

for successor in current\_state.generate\_successors():

found = depth\_limited\_search(successor, goal\_state, depth - 1) if found:

return found

return None

def iterative\_deepening\_search(start\_state, goal\_state):

depth = 0 while True:

print(f"\nSearching at depth level: {depth}") result = depth\_limited\_search(start\_state, goal\_state, depth) if result: return result

depth += 1

def get\_user\_input(): print("Enter the start state (use 0 for the blank):")

start\_state = [] for \_ in range(3):

row = list(map(int, input().split()))

start\_state.append(row)

print("Enter the goal state (use 0 for the blank):") goal\_state = [] for \_ in range(3):

row = list(map(int, input().split()))

goal\_state.append(row)

return start\_state, goal\_state

def main():

start\_board, goal\_board = get\_user\_input() start\_state = PuzzleState(start\_board) goal\_state = goal\_board

result = iterative\_deepening\_search(start\_state, goal\_state)

if result:

print("\nGoal reached!")

path = [] while result:

path.append(result) result = result.parent path.reverse() for state in path: print(state, "\n") else:

print("Goal state not found.")

if \_\_name\_\_ == "\_\_main\_\_": main()

Output:

Enter the start state (use 0 for the blank):

1 2 3

4 0 5

6 7 8

Enter the goal state (use 0 for the blank):

1 2 0

3 4 5

6 7 8

Searching at depth level: 0

Searching at depth level: 1

Searching at depth level: 2

Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5

Searching at depth level: 6

Searching at depth level: 7

Searching at depth level: 8

Searching at depth level: 9

Searching at depth level: 10

Searching at depth level: 11

Searching at depth level: 12

Goal reached! 1 2 3

4 0 5

6 7 8

1 2 3

0 4 5

6 7 8

1. 2 3
2. 4 5

6 7 8

2 0 3

1 4 5

6 7 8

2 3 0

1 4 5

6 7 8

2 3 5

1 4 0

6 7 8

2 3 5

1 0 4

6 7 8

2 0 5

1 3 4

6 7 8

1. 2 5
2. 3 4

6 7 8

1 2 5

0 3 4

6 7 8

1 2 5

3 0 4

6 7 8

1 2 5

3 4 0

6 7 8

1 2 0

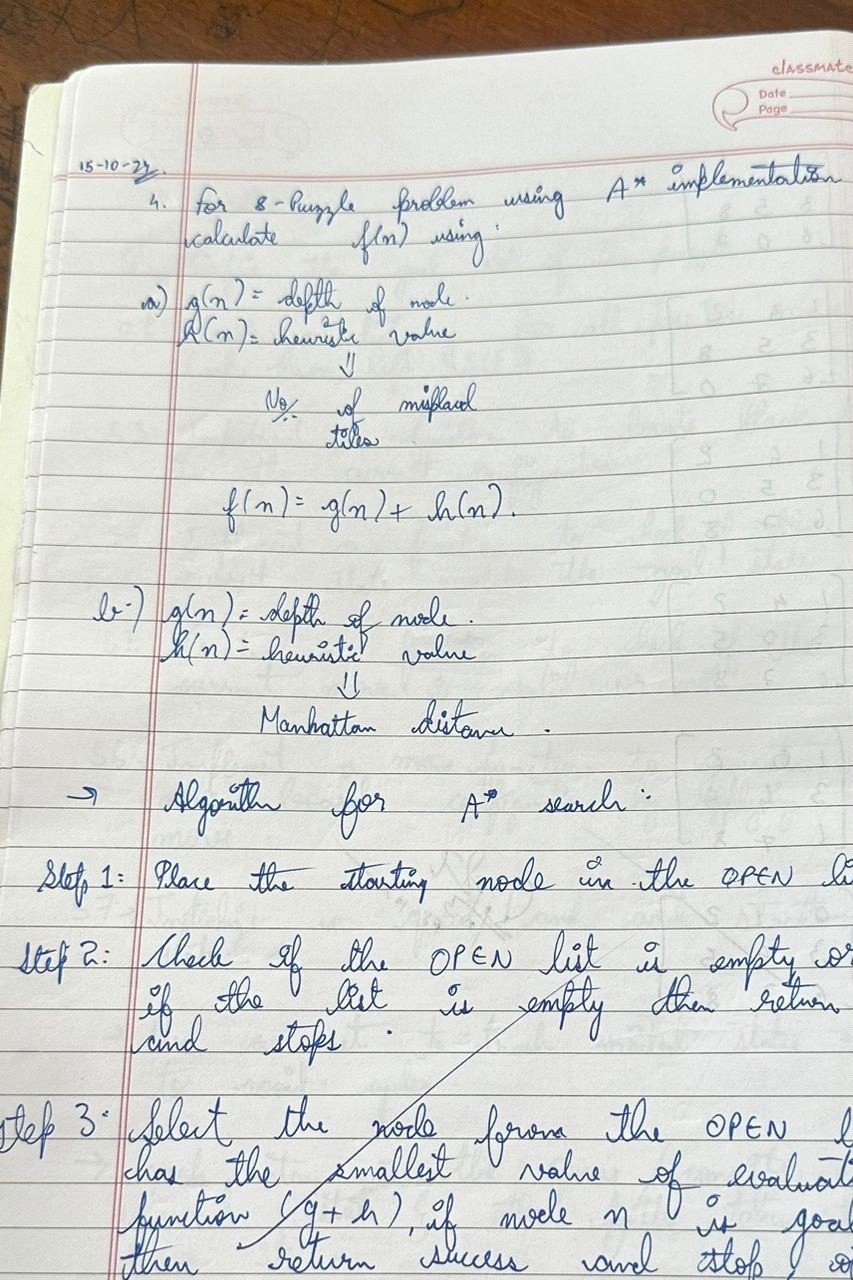
3 4 5

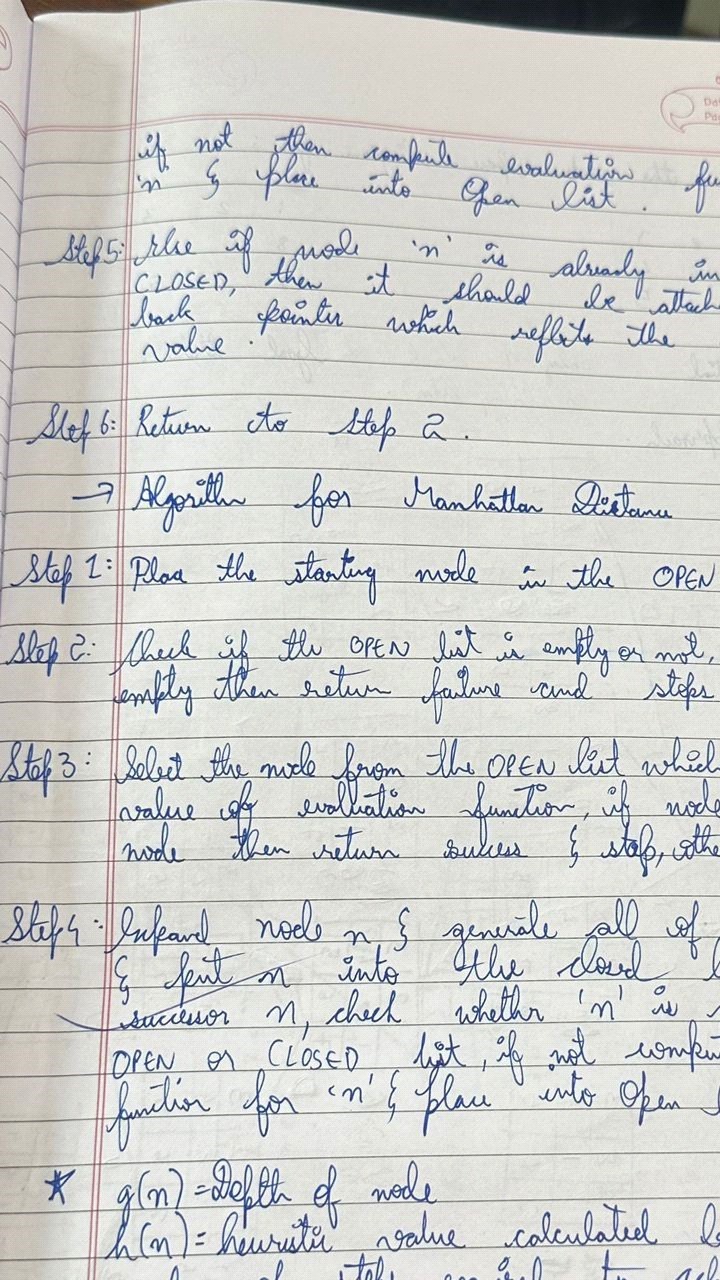
6 7 8

Program 3

Implement A\* Algorithm

Algorithm





Code: import heapq

GOAL\_STATE = ((1, 2, 3),

(8, 0, 4), (7, 6, 5))

def manhattan\_distance(state):

distance = 0 for i in range(3): for j in range(3): value = state[i][j] if value != 0:

goal\_x, goal\_y = divmod(value - 1, 3) distance += abs(goal\_x - i) + abs(goal\_y - j) return distance def find\_blank(state): for i in range(3): for j in range(3):

if state[i][j] == 0:

return i, j def generate\_neighbors(state): neighbors = [] x, y = find\_blank(state) directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

for dx, dy in directions: nx, ny = x + dx, y + dy if 0 <= nx < 3 and 0 <= ny < 3:

new\_state = [list(row) for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y] neighbors.append(tuple(tuple(row) for row in new\_state)) return neighbors

def reconstruct\_path(came\_from, current):

path = [current] while current in came\_from: current = came\_from[current] path.append(current) path.reverse()

return path

def a\_star(start): open\_list = [] heapq.heappush(open\_list, (manhattan\_distance(start), 0, start))

g\_score = {start: 0}

came\_from = {}

visited = set()

while open\_list:

f, g, current = heapq.heappop(open\_list)

if current == GOAL\_STATE:

path = reconstruct\_path(came\_from, current) return path, g

visited.add(current) for neighbor in generate\_neighbors(current):

if neighbor in visited:

continue

tentative\_g = g\_score[current] + 1

if tentative\_g < g\_score.get(neighbor, float('inf')):

came\_from[neighbor] = current g\_score[neighbor] = tentative\_g

f\_score = tentative\_g + manhattan\_distance(neighbor) heapq.heappush(open\_list, (f\_score, tentative\_g, neighbor))

return None, None

def print\_state(state): for row in state: print(row)

print()

if \_\_name\_\_ == "\_\_main\_\_":

start\_state = ((2, 8, 3),

(1, 6, 4),

(7, 0, 5))

print("Initial State:")

print\_state(start\_state)

print("Goal State:")

print\_state(GOAL\_STATE)

solution, cost = a\_star(start\_state)

if solution:

print(f"Solution found with cost: {cost}")

print("Steps:") for step in solution: print\_state(step) else: print("No solution found.")Output:

Initial State: (2, 8, 3)

(1, 6, 4)

(7, 0, 5)

Goal State:

(1, 2, 3)

(8, 0, 4)

(7, 6, 5)

Solution found with cost: 5 Steps:

(2, 8, 3)

(1, 6, 4)

(7, 0, 5)

(2, 8, 3)

(1, 0, 4)

(7, 6, 5)

(2, 0, 3)

(1, 8, 4)

(7, 6, 5)

(0, 2, 3)

(1, 8, 4)

(7, 6, 5)

(1, 2, 3)

(0, 8, 4)

(7, 6, 5)

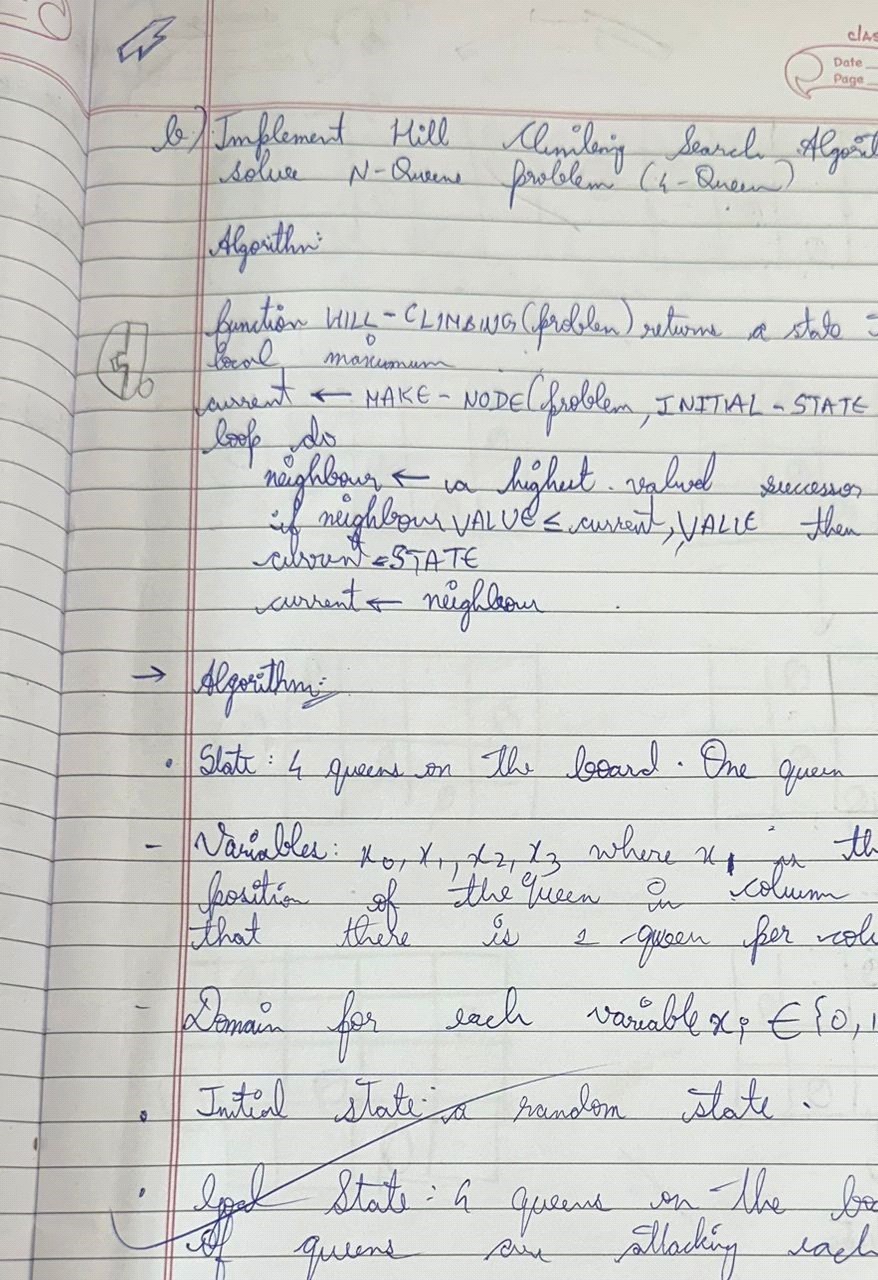
(1, 2, 3)

(8, 0, 4)

(7, 6, 5) Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Code:

import random

def get\_attacking\_pairs(state):

"""Calculates the number of attacking pairs of queens.""" attacks = 0 n = len(state) for i in range(n):

for j in range(i + 1, n): if state[i] == state[j]:

attacks += 1 if abs(state[i] - state[j]) == abs(i - j):

attacks += 1 return attacks

def generate\_successors(state):

"""Generates all possible successors by moving each queen to every other column in its row.""" n = len(state) successors = [] for row in range(n): for col in range(n): if col != state[row]: new\_state = state[:] new\_state[row] = col successors.append(new\_state) return successors

def hill\_climbing(n):

"""Hill climbing algorithm for n-queens problem.""" current = [random.randint(0, n - 1) for \_ in range(n)] while True:

current\_attacks = get\_attacking\_pairs(current) successors = generate\_successors(current) neighbor = min(successors, key=get\_attacking\_pairs) neighbor\_attacks = get\_attacking\_pairs(neighbor) if neighbor\_attacks >= current\_attacks:

return current, current\_attacks current = neighbor

def print\_board(state):

"""Prints the board with queens placed."""

n = len(state) board = [["." for \_ in range(n)] for \_ in range(n)] for row in range(n):

board[row][state[row]] = "Q" for row in board:

print(" ".join(row)) print("\n")

n = 4 solution, attacks = hill\_climbing(n) print("Final State (Solution):", solution) print("Number of Attacking Pairs:", attacks) print\_board(solution)

Output:

Final State (Solution): [3, 0, 2, 1]

Number of Attacking Pairs: 1

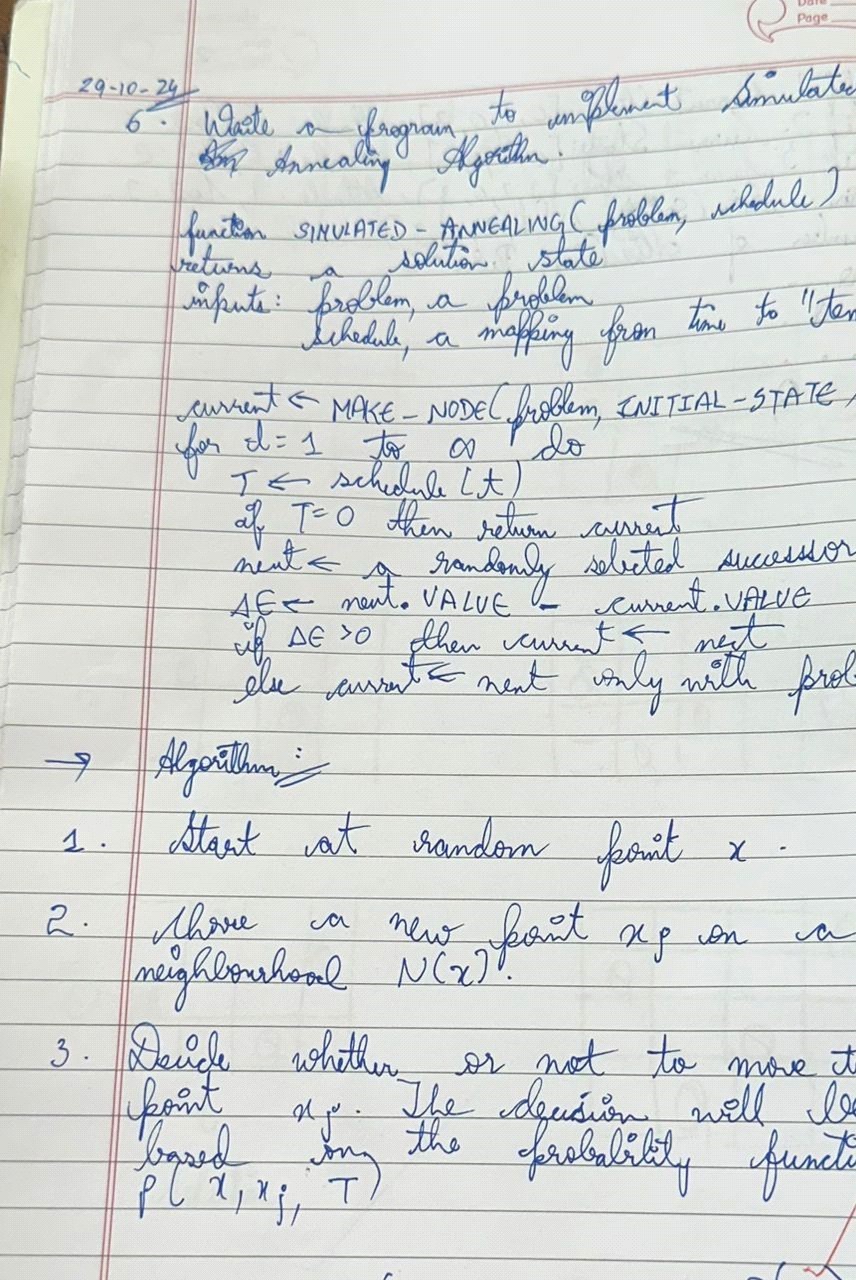
. . . Q Q . . . . . Q .

. Q . .

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

def print\_board(board): for row in board:

print(" ".join(str(num) for num in row))

def is\_valid(board, row, col, num):

# Check if the number is not in the current row and column for i in range(9): if board[row][i] == num or board[i][col] == num: return False

# Check the 3x3 grid start\_row, start\_col = 3 \* (row // 3), 3 \* (col // 3) for i in range(3): for j in range(3): if board[start\_row + i][start\_col + j] == num:

return False

return True

def solve\_sudoku(board):

empty = find\_empty\_location(board) if not empty:

return True # Puzzle solved

row, col = empty

for num in range(1, 10): if is\_valid(board, row, col, num):

board[row][col] = num

if solve\_sudoku(board):

return True

# Reset the cell (backtrack)

board[row][col] = 0

return False # Trigger backtracking

def find\_empty\_location(board): for i in range(9): for j in range(9): if board[i][j] == 0:

return (i, j) # Return row, col

return None

# Example Sudoku puzzle (0 represents empty cells) sudoku\_board = [

[5, 3, 0, 0, 7, 0, 0, 0, 0],

[6, 0, 0, 1, 9, 5, 0, 0, 0],

[0, 9, 8, 0, 0, 0, 0, 6, 0],

[8, 0, 0, 0, 6, 0, 0, 0, 3],

[4, 0, 0, 8, 0, 3, 0, 0, 1],

[7, 0, 0, 0, 2, 0, 0, 0, 6],

[0, 6, 0, 0, 0, 0, 2, 8, 0],

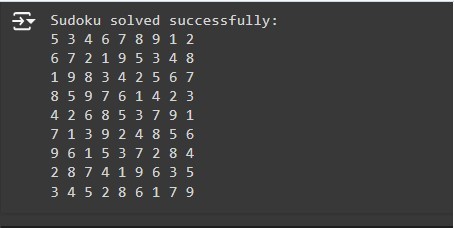
[0, 0, 0, 4, 1, 9, 0, 0, 5],

[0, 0, 0, 0, 8, 0, 0, 7, 9]

]

if solve\_sudoku(sudoku\_board): print("Sudoku solved successfully:") print\_board(sudoku\_board) else:

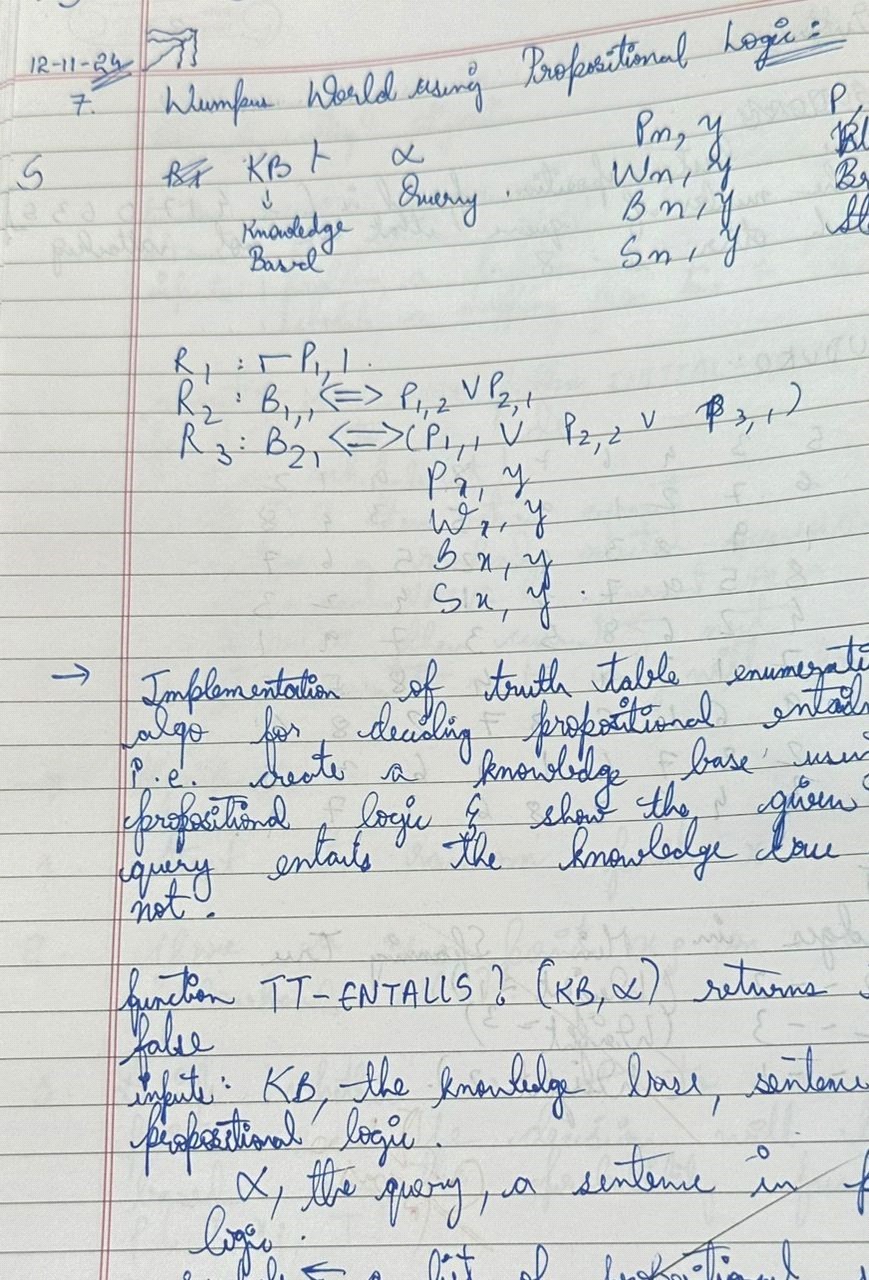
print("No solution exists.") Output:

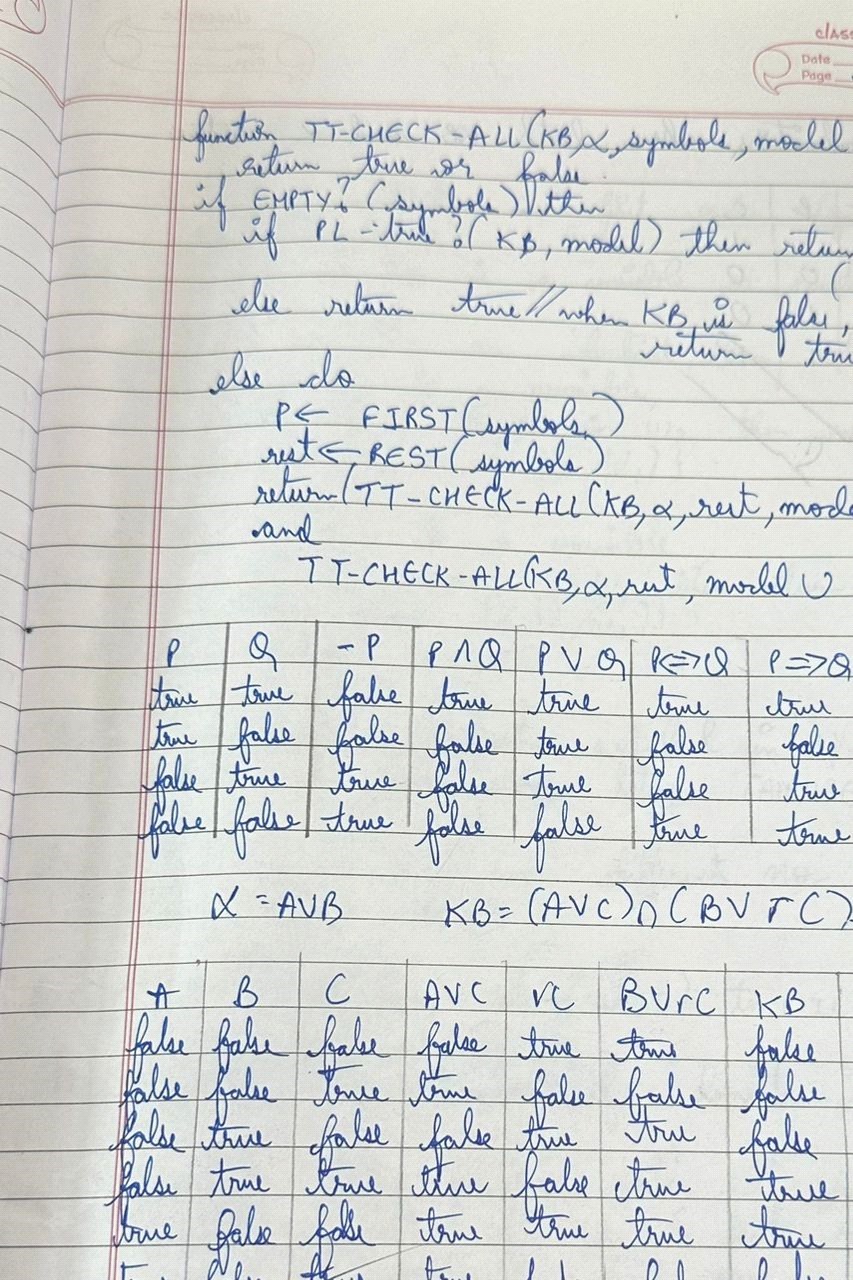


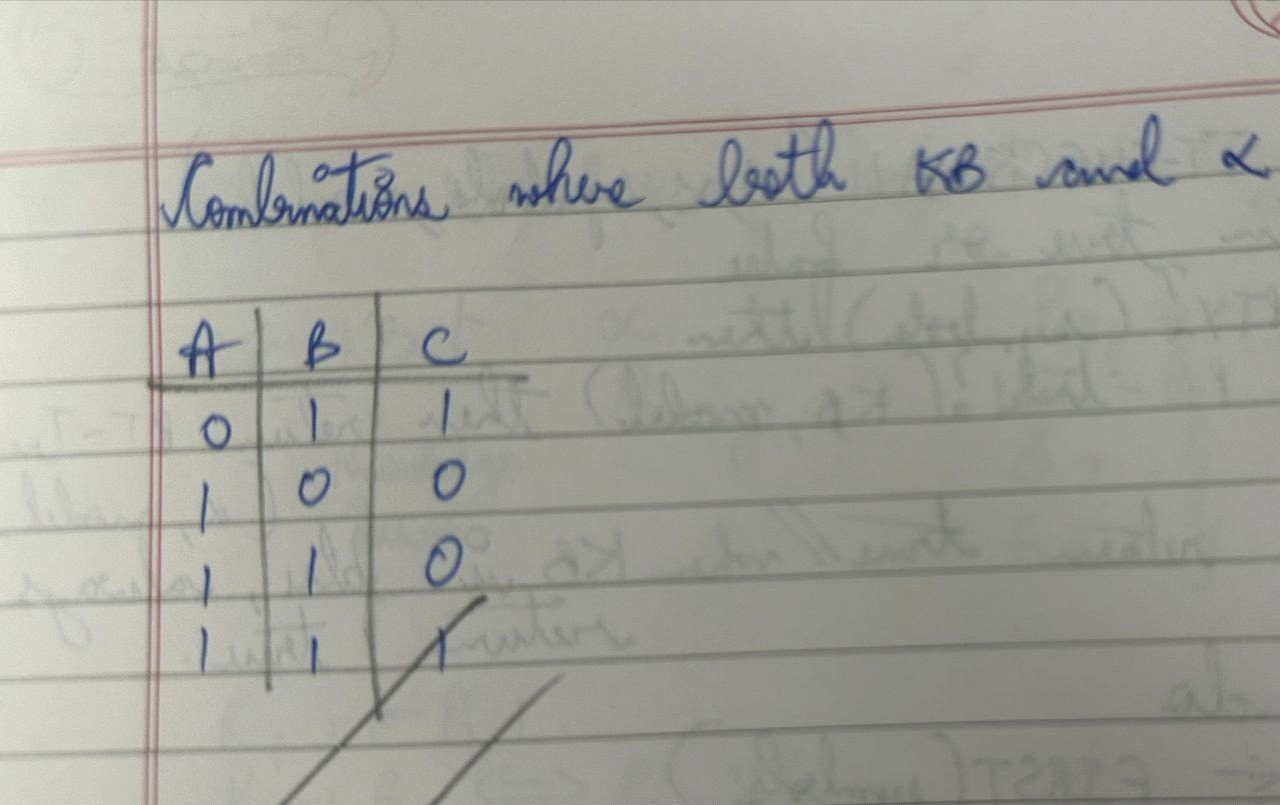
Program 6

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

Algorithm







Code:

import itertools

def evaluate\_formula(formula, valuation):

"""

Evaluate the propositional formula under the given truth assignment (valuation). The formula is a string of logical operators like 'AND', 'OR', 'NOT', and can contain variables 'A', 'B', 'C'.

"""

# Create a local environment (dictionary) for variable assignments env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

# Replace logical operators with Python equivalents

formula = formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')

# Replace variables in the formula with their corresponding truth values for var in env:

formula = formula.replace(var, str(env[var]))

# Evaluate the formula and return the result (True or False) try:

return eval(formula) except Exception as e:

raise ValueError(f"Error in evaluating formula: {e}")

def truth\_table(variables):

"""

Generate all possible truth assignments for the given variables.

"""

return list(itertools.product([False, True], repeat=len(variables)))

def entails(KB, alpha):

"""

Decide if KB entails alpha using a truth-table enumeration algorithm.

KB is a propositional formula (string), and alpha is another propositional formula (string).

"""

# Generate all possible truth assignments for A, B, and C assignments = truth\_table(['A', 'B', 'C'])

print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}") # Header for the truth table print("-" \* 70) # Separator for readability

for assignment in assignments:

# Evaluate KB and alpha under the current assignment KB\_value = evaluate\_formula(KB, assignment) alpha\_value = evaluate\_formula(alpha, assignment)

# Print the current truth assignment and the results for KB and alpha print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB\_v alue):<15}{str(alpha\_value):<15}{'Yes' if KB\_value and alpha\_value else 'No'}")

# If KB is true and alpha is false, then KB does not entail alpha if KB\_value and not alpha\_value:

return False

# If no counterexample was found, then KB entails alpha return True

# Define the formulas for KB and alpha alpha = 'A OR B'

KB = '(A OR C) AND (B OR NOT C)'

# Check if KB entails alpha

result = entails(KB, alpha)

# Print the final result of entailment

print(f"\nDoes KB entail alpha? {result}")

Output:

A B C KB alpha KB entails alpha?

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

False False False False False No False False True False False No

False True False False True No

False True True True True Yes

True False False True True Yes

True False True False True No

True True False True True Yes

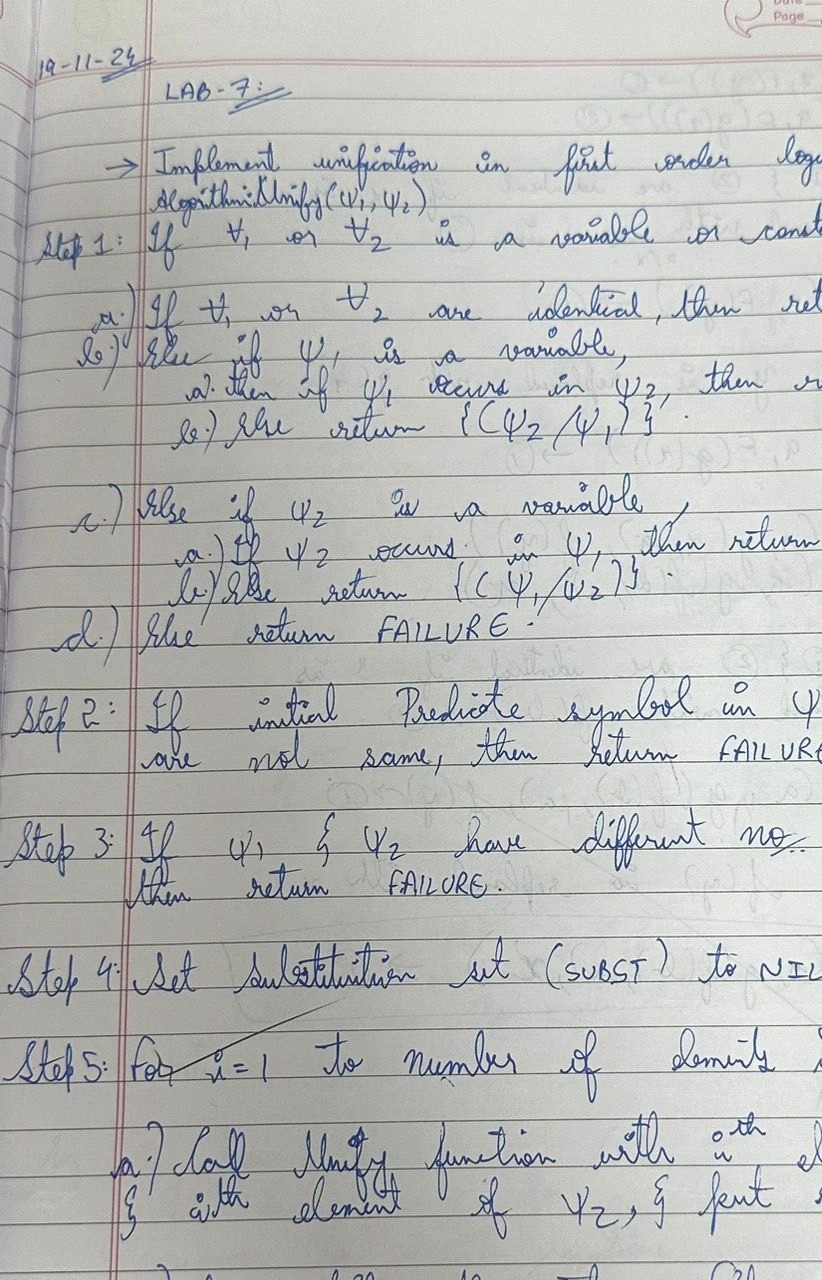
True True True True True Yes

Does KB entail alpha? True

Program 7:

Implement unification in first order logic

Algorithm:



Code:

def unify(term1, term2):

"""

Unifies two terms using the unification algorithm.

Args:

term1: The first term. term2: The second term.

Returns:

A substitution set if the terms are unifiable, otherwise None. """

# Step 1: Handle variables and constants if isinstance(term1, str) or isinstance(term2, str): if term1 == term2: return {} # NIL elif term1.isupper(): # term1 is a variable if term1 in term2: return None # FAILURE return {term1: term2} elif term2.isupper(): # term2 is a variable if term2 in term1: return None # FAILURE return {term2: term1} else:

return None # FAILURE

# Step 2: Check predicate symbols if term1[0] != term2[0]: return None # FAILURE

# Step 3: Check number of arguments if len(term1[1:]) != len(term2[1:]):

return None # FAILURE

# Step 4: Initialize substitution set substitution\_set = {}

# Step 5: Recursively unify arguments for i in range(len(term1[1:])):

result = unify(term1[1:][i], term2[1:][i]) if result is None:

return None # FAILURE else:

substitution\_set = apply\_substitution(substitution\_set, result)

# Step 6: Return substitution set

return substitution\_set

def apply\_substitution(substitution\_set, new\_substitution):

"""

Applies a new substitution to the existing substitution set.

Args:

substitution\_set: The current substitution set.

new\_substitution: The new substitution to be applied.

Returns:

The updated substitution set. """

for var, value in new\_substitution.items(): # Apply the new substitution to the existing values for key, val in substitution\_set.items():

# Replace variables in the existing substitution set if key == var:

substitution\_set[key] = value elif isinstance(val, str) and val == var:

substitution\_set[key] = value

# Add the new substitution to the set

substitution\_set[var] = value

return substitution\_set

def parse\_term(term\_str):

"""

Parses a term string into a structured format.

Args:

term\_str: The string representation of the term.

Returns:

A structured term (list).

""" term\_str = term\_str.strip() if '(' not in term\_str:

return term\_str # Return variable or constant

# Split the term into the function and its arguments func\_name, args\_str = term\_str.split('(', 1)

args\_str = args\_str.rstrip(')') # Remove the closing parenthesis

# Split arguments by commas, accounting for nested terms args = [] bracket\_count = 0

current\_arg = []

for char in args\_str:

if char == ',' and bracket\_count == 0: args.append(''.join(current\_arg).strip()) current\_arg = [] else: if char == '(':

bracket\_count += 1 elif char == ')':

bracket\_count -= 1

current\_arg.append(char)

if current\_arg:

args.append(''.join(current\_arg).strip())

return [func\_name] + [parse\_term(arg) for arg in args]

def apply\_substitution\_to\_term(term, substitution\_set):

"""

Applies a substitution set to a term.

Args:

term: The term to which the substitution is applied. substitution\_set: The substitution set.

Returns:

The term after applying substitutions.

""" if isinstance(term, str):

return substitution\_set.get(term, term) # Return the substituted value or the term itself

# Apply substitution recursively to the term's components return [term[0]] + [apply\_substitution\_to\_term(arg, substitution\_set) for arg in term[1:]]

def main():

# Take input from the user for the expressions term1\_str = input("Enter the first term : ")

term2\_str = input("Enter the second term: ")

# Parse the input terms term1 = parse\_term(term1\_str) term2 = parse\_term(term2\_str)

# Perform unification

substitution\_set = unify(term1, term2)

if substitution\_set is not None:

# Apply the substitution to the original terms unified\_term1 = apply\_substitution\_to\_term(term1, substitution\_set) unified\_term2 = apply\_substitution\_to\_term(term2, substitution\_set)

# Print the final unified expressions print("Unified Term 1:", unified\_term1) print("Unified Term 2:", unified\_term2) else:

print("The terms are not unifiable.")

if \_\_name\_\_ == "\_\_main\_\_": main()

Output:

Enter the first term : f(X,f(Y))

Enter the second term: f(a,f(g(X))

Unified Term 1: ['f', 'a', ['f', ['g', 'X']]]

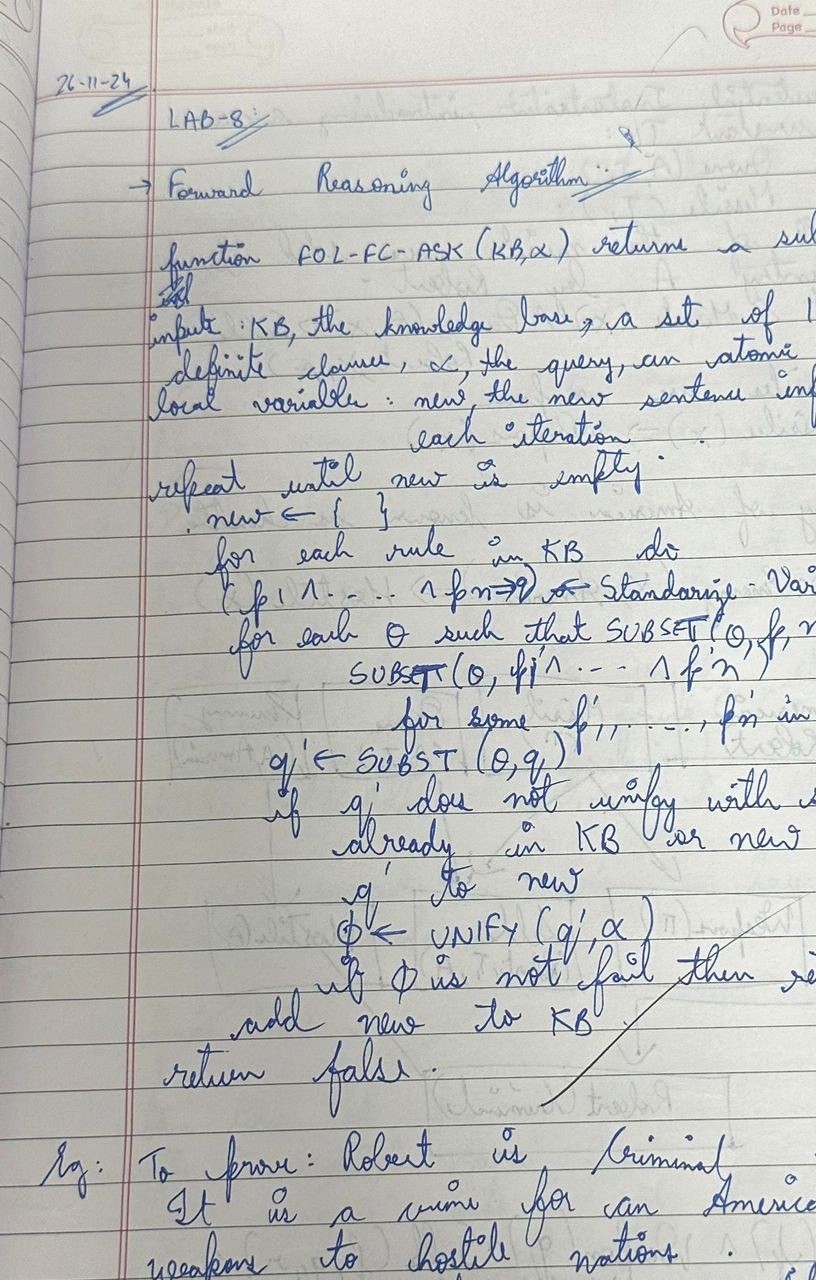
Unified Term 2: ['f', 'a', ['f', ['g', 'a']]]

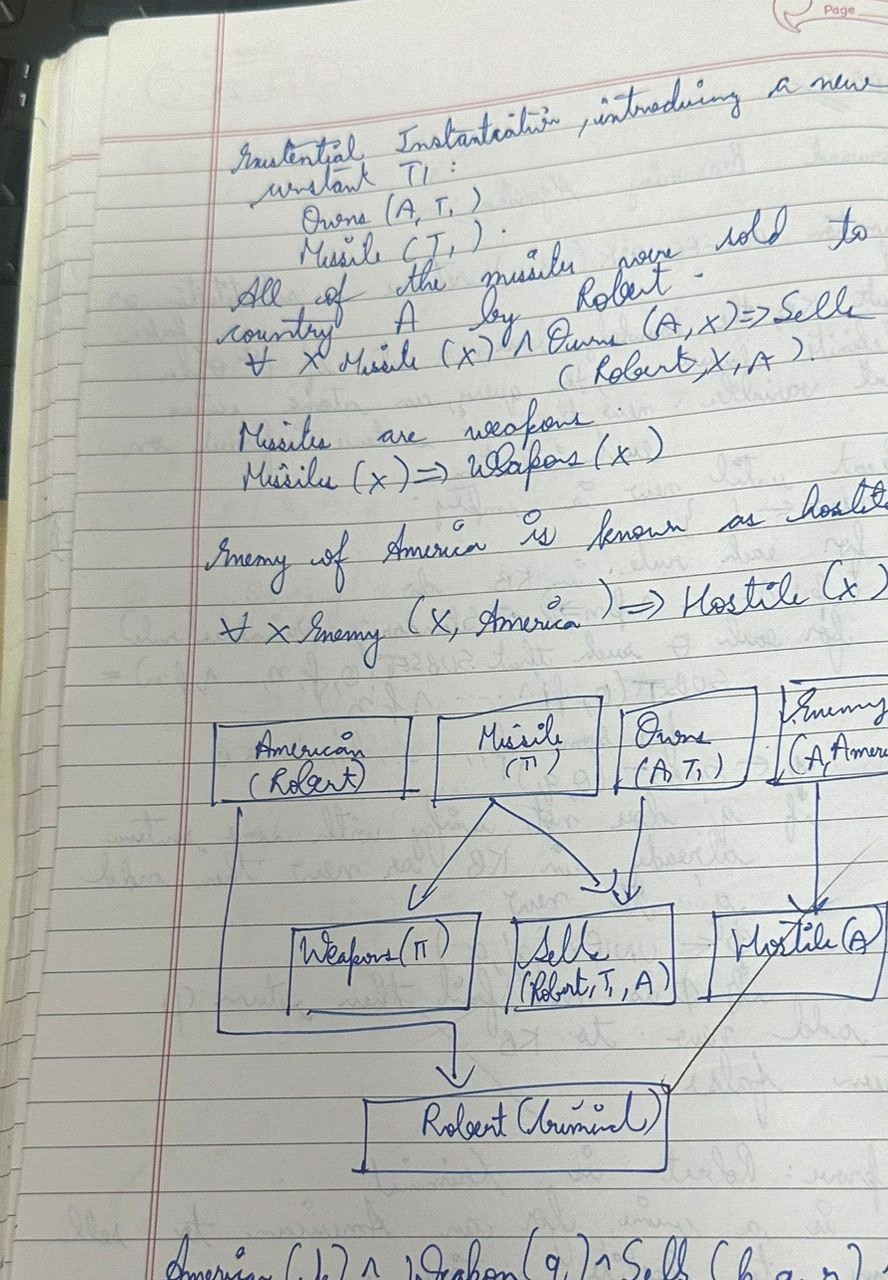
Enter the first term : f(f(a,g(Y)) Enter the second term: f(X,X) The terms are not unifiable.

Program 8:

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

Algorithm:





Code:

# Define the knowledge base as a list of rules and facts

class KnowledgeBase: def \_\_init\_\_(self):

self.facts = set() # Set of known facts

self.rules = [] # List of rules

def add\_fact(self, fact): self.facts.add(fact)

def add\_rule(self, rule): self.rules.append(rule)

def infer(self): inferred = True while inferred: inferred = False for rule in self.rules: if rule.apply(self.facts):

inferred = True

# Define the Rule class class Rule: def \_\_init\_\_(self, premises, conclusion):

self.premises = premises # List of conditions self.conclusion = conclusion # Conclusion to add if premises are met

def apply(self, facts): if all(premise in facts for premise in self.premises): if self.conclusion not in facts: facts.add(self.conclusion) print(f"Inferred: {self.conclusion}")

return True

return False

# Initialize the knowledge base

kb = KnowledgeBase()

# Facts in the problem kb.add\_fact("American(Robert)") kb.add\_fact("Missile(T1)") kb.add\_fact("Owns(A, T1)")

kb.add\_fact("Enemy(A, America)")

# Rules based on the problem # 1. Missile(x) implies Weapon(x)

kb.add\_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)

kb.add\_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A) kb.add\_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p) kb.add\_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)",

"Hostile(A)"], "Criminal(Robert)"))

# Infer new facts based on the rules

kb.infer()

# Check if Robert is a criminal if "Criminal(Robert)" in kb.facts:

print("Conclusion: Robert is a criminal.") else:

print("Conclusion: Unable to prove Robert is a criminal.")

Output:

Inferred: Weapon(T1)

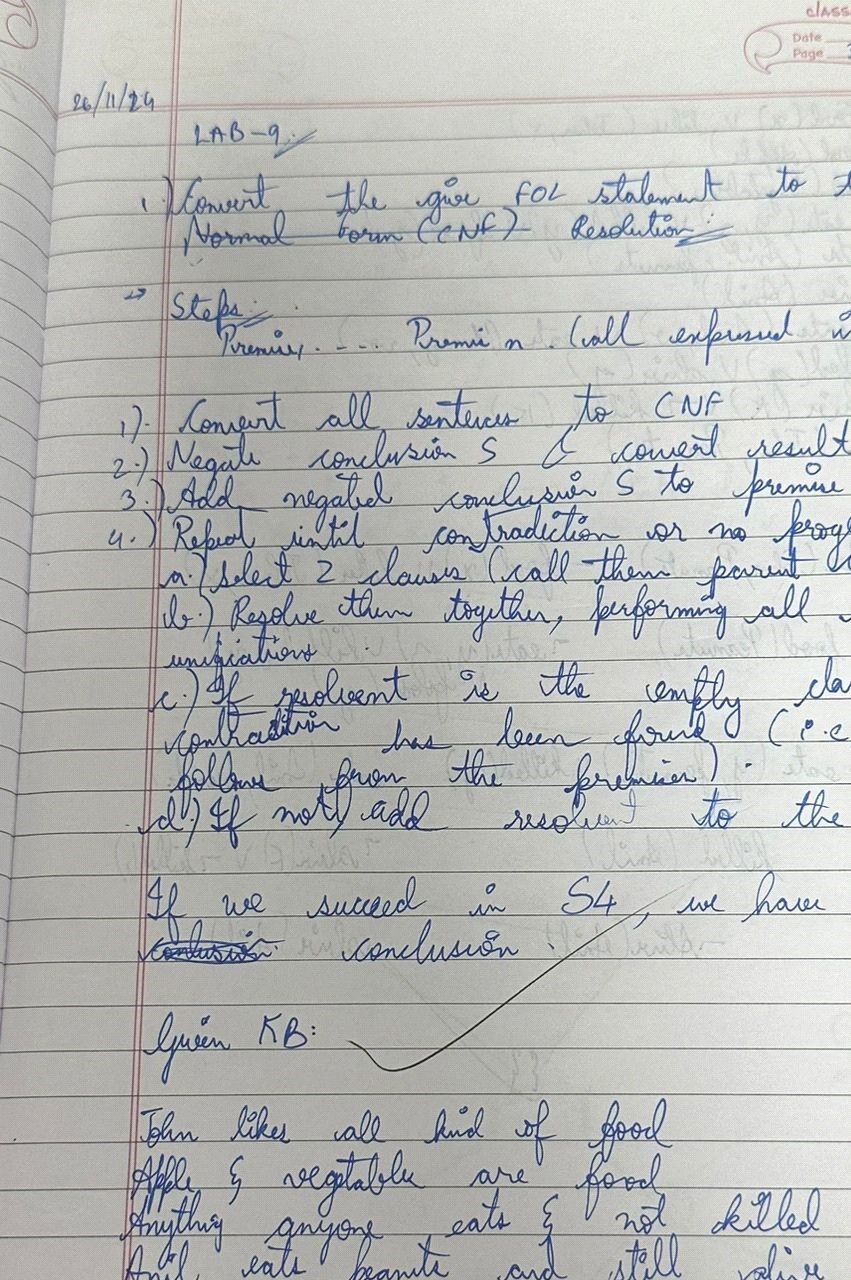
Inferred: Hostile(A)

Inferred: Sells(Robert, T1, A) Inferred: Criminal(Robert) Conclusion: Robert is a criminal.

Program 9:

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

Algorithm:



Code:

class CNFReasoner: def \_\_init\_\_(self, clauses):

"""

Initializes the CNF Reasoner.

Parameters:

clauses (list of sets): List of clauses in CNF format.

"""

self.clauses = [set(clause) for clause in clauses]

def resolve(self, clause1, clause2):

"""

Resolve two clauses to produce new clauses if possible.

Parameters:

clause1 (set): The first clause (set of literals). clause2 (set): The second clause (set of literals).

Returns:

list: A list of resolved clauses (sets of literals).

""" resolvents = [] for literal in clause1:

neg\_literal = f"~{literal}" if not literal.startswith("~") else literal[1:] if neg\_literal in clause2:

# Create a new clause by removing complementary literals new\_clause = (clause1 - {literal}) | (clause2 - {neg\_literal}) resolvents.append(new\_clause) return resolvents

def infer(self, goal):

"""

Infer whether the goal is provable using resolution.

Parameters:

goal (set): The negation of the goal to be proved.

Returns:

bool: True if the goal is provable, False otherwise.

"""

goal\_clause = {f"~{literal}" for literal in goal} clauses = self.clauses + [goal\_clause] # Add the negated goal to clauses new = set()

while True: pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j in range(i + 1, len(clauses))] for (clause1, clause2) in pairs:

resolvents = self.resolve(clause1, clause2) for resolvent in resolvents:

if not resolvent: # Found an empty clause, goal is proved return True new.add(frozenset(resolvent))

# Check if no new information is being added if new.issubset(set(map(frozenset, clauses))):

return False

# Add new resolvents to the clauses for clause in new:

if clause not in clauses:

clauses.append(set(clause))

# Define the CNF clauses based on the FOL premises cnf\_clauses = [

{"~Food(X)", "Likes(John, X)"}, # Rule 1: If Food(X), then Likes(John, X)

{"~Eats(Anil, Peanuts)", "Food(Peanuts)"}, # Rule 2.1: If Anil eats peanuts, peanuts are food

{"~Alive(Anil)", "Food(Peanuts)"}, # Rule 2.2: If Anil is alive, peanuts are food {"~Food(Peanuts)", "Likes(John, Peanuts)"}, # Rule 3: If peanuts are food, John likes peanuts

{"Eats(Anil, Peanuts)"}, # Fact 1: Anil eats peanuts

{"Alive(Anil)"}, # Fact 2: Anil is alive

]

# Define the goal: Prove that John likes peanuts goal = {"Likes(John, Peanuts)"}

# Initialize the reasoner and infer reasoner = CNFReasoner(cnf\_clauses) if reasoner.infer(goal):

print("Proved: John likes peanuts.") else:

print("Could not prove: John likes peanuts.")

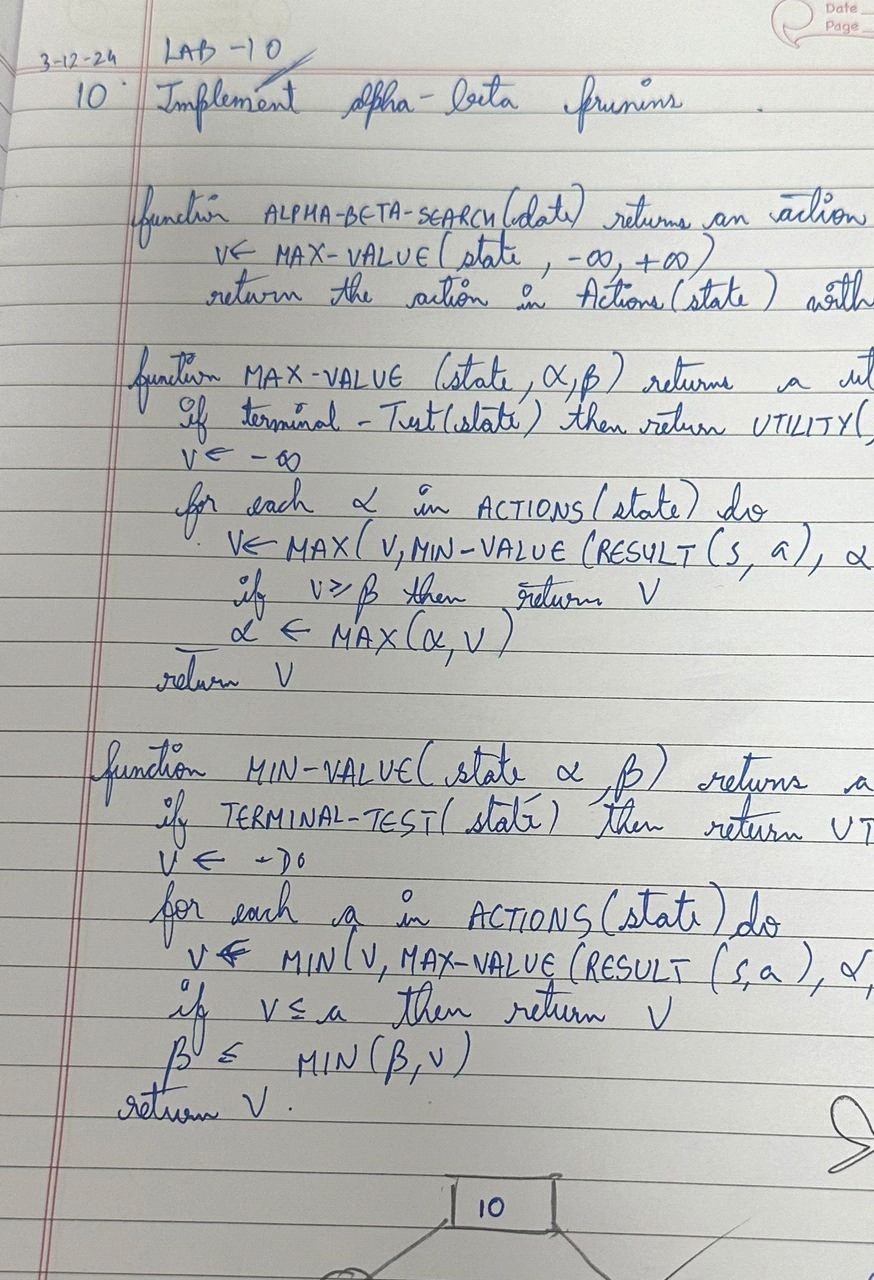
Output:

Proved: John likes peanuts.

Program 10:

Implement Alpha-Beta Pruning.

Algorithm:



Code: import math

def minimax(depth, index, maximizing\_player, values, alpha, beta):

# Base case: when we've reached the leaf nodes if depth == 0:

return values[index]

if maximizing\_player: max\_eval = float('-inf') for i in range(2): # 2 children per node eval = minimax(depth - 1, index \* 2 + i, False, values, alpha, beta) max\_eval = max(max\_eval, eval) alpha = max(alpha, eval) if beta <= alpha: # Beta cutoff

break return max\_eval else:

min\_eval = float('inf')

for i in range(2): # 2 children per node eval = minimax(depth - 1, index \* 2 + i, True, values, alpha, beta) min\_eval = min(min\_eval, eval) beta = min(beta, eval) if beta <= alpha: # Alpha cutoff

break

return min\_eval

# Accept values from the user leaf\_values = list(map(int, input("Enter the leaf node values separated by spaces: ").split()))

# Check if the number of values is a power of 2 if math.log2(len(leaf\_values)) % 1 != 0:

print("Error: The number of leaf nodes must be a power of 2 (e.g., 2, 4, 8, 16).") else:

# Calculate depth of the tree

tree\_depth = int(math.log2(len(leaf\_values)))

# Run Minimax with Alpha-Beta Pruning optimal\_value = minimax(depth=tree\_depth, index=0, maximizing\_player=True, values=leaf\_values, alpha=float('-inf'), beta=float('inf'))

print("Optimal value calculated using Minimax:",optimal\_value)

Output:

Enter the leaf node values separated by spaces: -1 8 -3 -1 2 1 -3 4

Optimal value calculated using Minimax: 2