VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shreyas T S (1BM23CS322)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug 2025 to Dec 2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Shreyas T S (1BM23CS322),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Lab faculty Incharge Name Assistant Professor Department of CSE, BMSCE	Dr.Seema patil Professor & HOD Department of CSE, BMSCE

Index

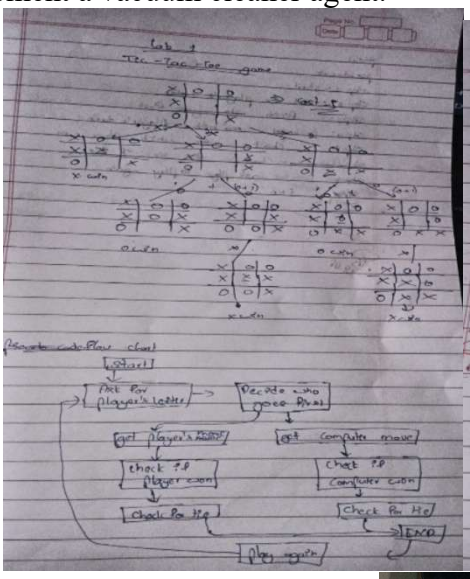
Sl. No.	Date	Experiment Title	Page No.
1	30-9-2024	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent	5-7
2	7-10-2024	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm	8-11
3	14-10-2024	Implement A* search algorithm	12-15
4	21-10-2024	Implement Hill Climbing search algorithm to solve N-Queens problem	16-17
5	28-10-2024	Simulated Annealing to Solve 8-Queens problem	18-19
6	11-11-2024	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.	20-21
7	2-12-2024	Implement unification in first order logic	22-24
8	2-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.	25-26
9	16-12-2024	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	27-28
10	16-12-2024	Implement Alpha-Beta Pruning.	29-30

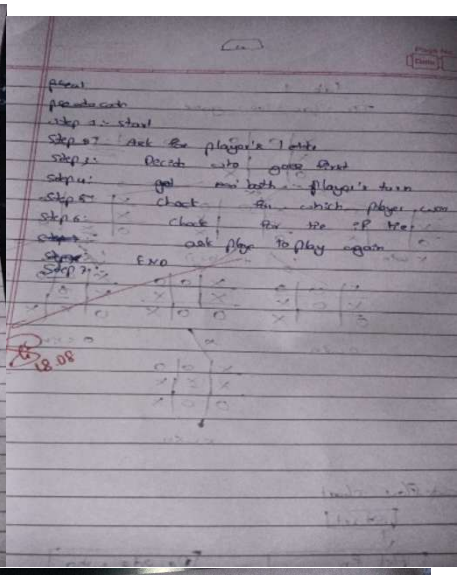
Github Link: https://github.com/Shreyes45/AI-lab

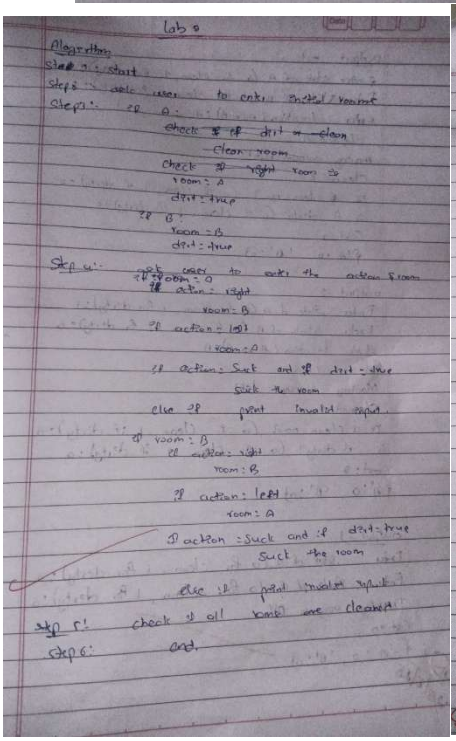
Program 1

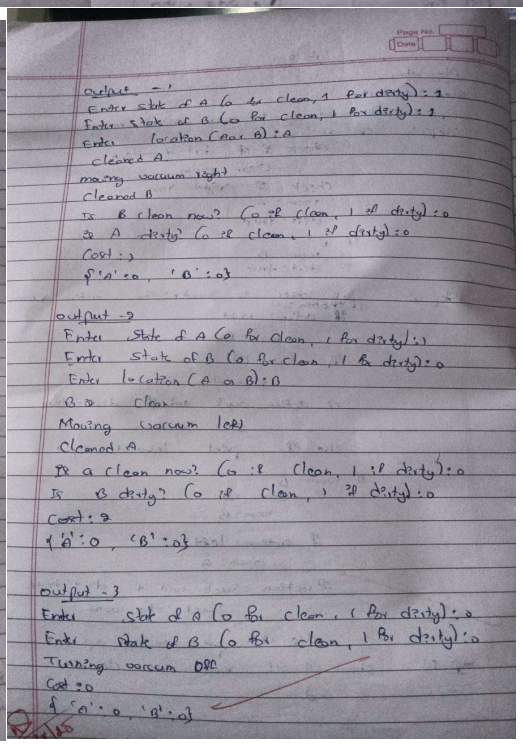
Implement Tic –Tac –Toe Game.

Implement a vacuum cleaner agent.









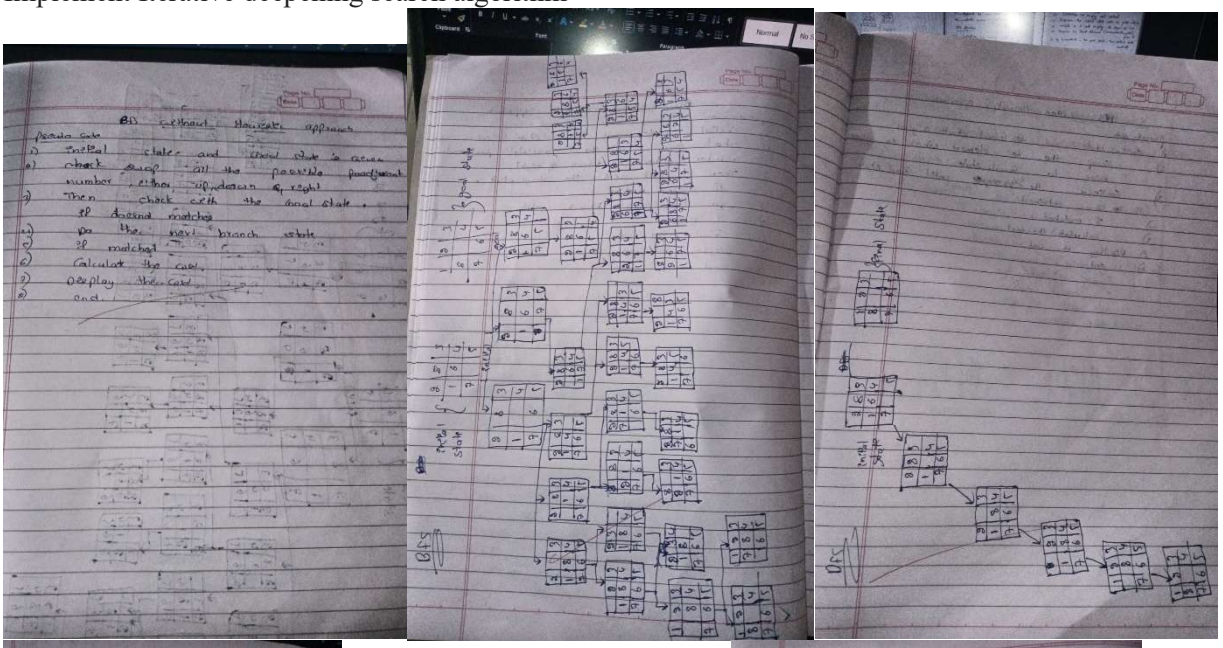
```
print("Shreyas t s 1BM23CS322")

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

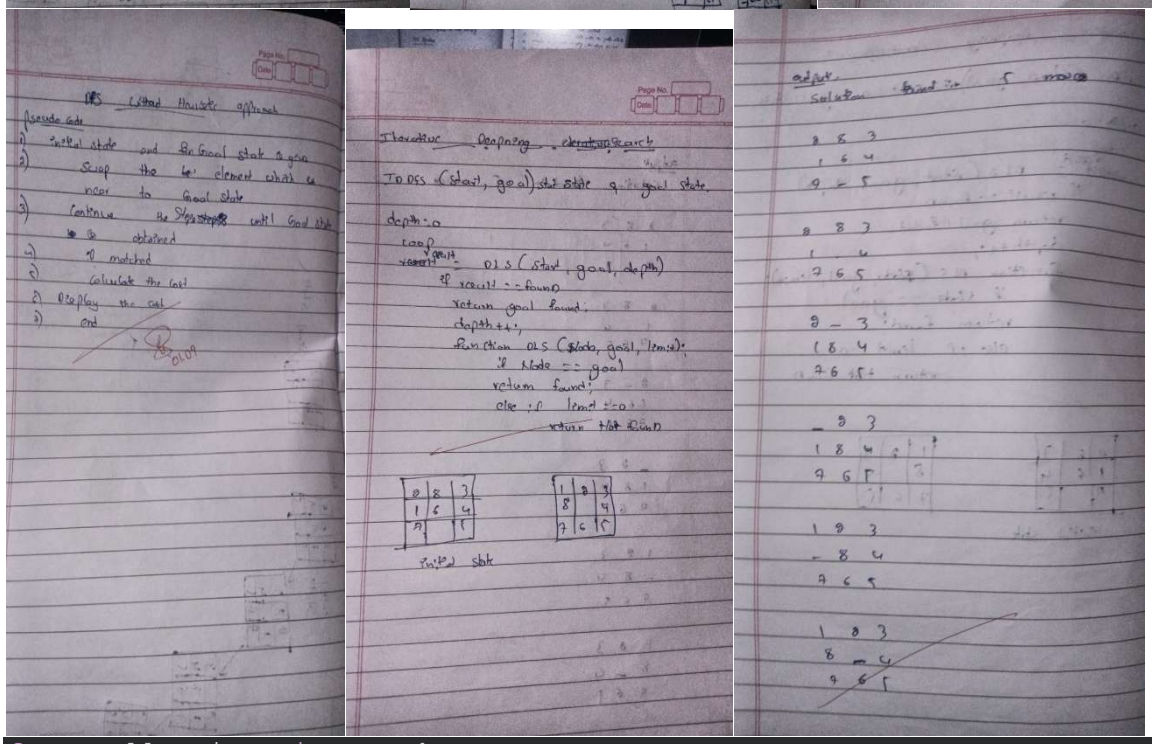
```
def check_win(board, player):
    for row in board:
        if all([cell == player for cell in row]):
            return True
    for col in range(3):
        if all([board[row][col] == player for row in range(3)]):
            return True
    if board[0][0] == board[1][1] == board[2][2] == player:
        return True
    if board[0][2] == board[1][1] == board[2][0] == player:
        return True
    return False

def check_draw(board):
    for row in board:
        if " " in row:
            return False
    return True

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

    while True:
        print_board(board)
        player = players[current_player]
        try:
            row = int(input(f"Player {player}, enter row (0-2): "))
            col = int(input(f"Player {player}, enter column (0-2): "))
            if not (0 <= row <= 2 and 0 <= col <= 2):
                print("Invalid input. Row and column must be between 0 and 2.")
                continue
        except ValueError:
            print("Invalid input. Please enter a number.")
            continue


        if board[row][col] == " ":
            board[row][col] = player
            if check_win(board, player):
                print_board(board)
                print(f"Player {player} wins!")
                break
            elif check_draw(board):
                print_board(board)
                print("It's a draw!")
```

```
            break
        else:
            current_player = (current_player + 1) % 2
    else:
        print("That spot is already taken! Try again.")

play_game()
print("Shreyas t s 1BM23CS322")

rooms = {'A': True, 'B': True}

current_room = input("Enter starting room (A/B): ").upper()
while current_room not in rooms:
    current_room = input("Invalid input. Enter starting room (A/B): ").upper()

def suck(room):
    if rooms[room]:
        print(f"Sucking dirt in room {room}.")
        rooms[room] = False
    else:
        print(f"Room {room} is already clean.")

def move(direction):
    global current_room
    if direction == "left" and current_room == "B":
        current_room = "A"
        print("Moved left to room A.")
    elif direction == "right" and current_room == "A":
        current_room = "B"
        print("Moved right to room B.")
    else:
        print("Cannot move in that direction from current room.")

while rooms['A'] or rooms['B']:
    print(f"\nCurrent room: {current_room}")
    print(f"Room A dirty? {rooms['A']}")
    print(f"Room B dirty? {rooms['B']}")
    action = input("Enter action (left, right, suck): ").lower()
    if action == "suck":
        suck(current_room)
    elif action in ["left", "right"]:
        move(action)
    else:
        print("Invalid action. Try again.")

print("Both rooms are clean.")
```

Program 2

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





```
from collections import deque

print("Shreyas ts , 1BM23CS322\n")

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

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

directions = {'Up': (-1, 0), 'Down': (1, 0), 'Left': (0, -1), 'Right': (0,
1)}

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

def swap_positions(state, pos1, pos2):
    state_list = [list(row) for row in state]
    r1, c1 = pos1
    r2, c2 = pos2
    state_list[r1][c1], state_list[r2][c2] = state_list[r2][c2],
state_list[r1][c1]
    return tuple(tuple(row) for row in state_list)

def get_neighbors(state):
    neighbors = []
    r, c = get_blank_pos(state)
    for move, (dr, dc) in directions.items():
        nr, nc = r + dr, c + dc
        if 0 <= nr < 3 and 0 <= nc < 3:
            new_state = swap_positions(state, (r, c), (nr, nc))
            neighbors.append((new_state, move))
    return neighbors

def dfs_8_puzzle(start, goal):
    stack = []
    visited = set()
    visited.add(start)
    stack.append((start, []))

    levels = []

    while stack:
```

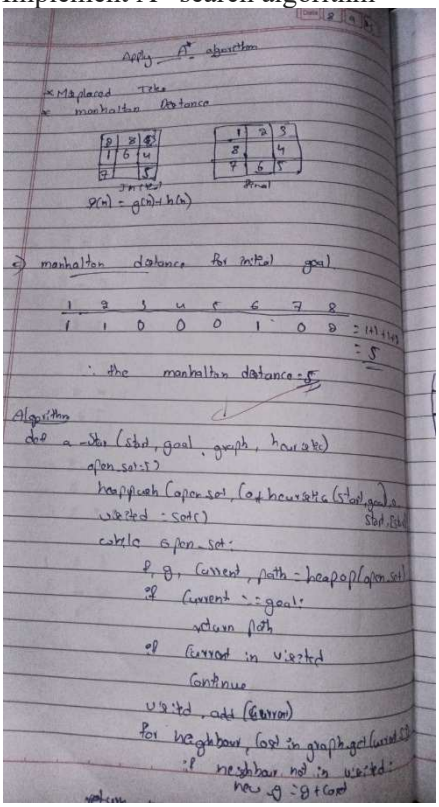
```
        state, path = stack.pop()
        levels.append(state)

        if state == goal:
            return path, levels, len(visited)

        for neighbor, move in get_neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [move]))

    return None, levels, len(visited)

solution_path, level_states, total_visited = dfs_8_puzzle(initial_state,
goal_state)

print(f"Solution length: {len(solution_path)} moves")
print("Solution moves:", solution_path)
print(f"Total states visited: {total_visited}\n")

print("States traversed:")
for i, state in enumerate(level_states):
    print(f"\nState {i+1}:")
    for row in state:
        print(row)
    print("---")
```

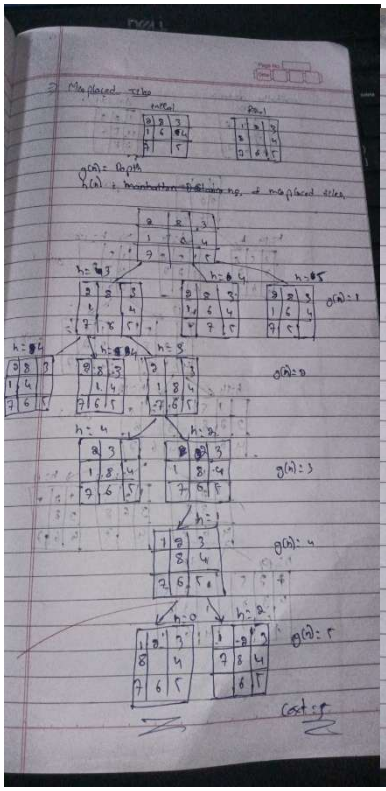
```
print("Shreyas ts  1BM23CS322")

def goal_state_reached(state, goal_state):
    return state == goal_state

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

def get_neighbors(state):
    neighbors = []
```

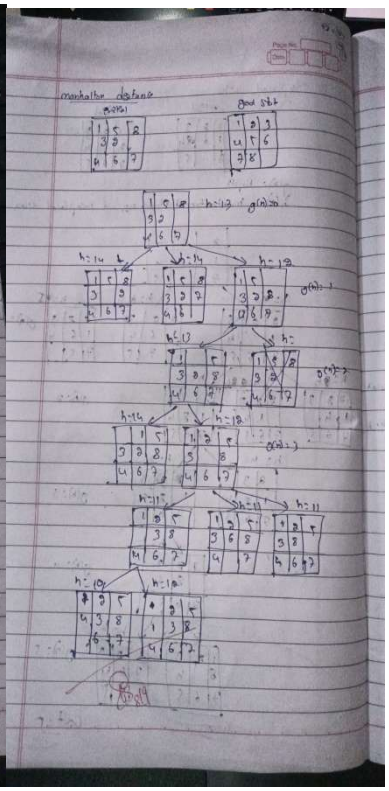
```
    x, y = find_empty(state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny],
new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def depth_limited_dfs(state, goal_state, depth):
    if depth == 0:
        return [state] if goal_state_reached(state, goal_state) else None
    elif depth > 0:
        for neighbor in get_neighbors(state):
            path = depth_limited_dfs(neighbor, goal_state, depth - 1)
            if path:
                return [state] + path
    return None

def iterative_deepening_dfs(start_state, goal_state):
    depth = 0
    while True:
        path = depth_limited_dfs(start_state, goal_state, depth)
        if path:
            return path
        depth += 1

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

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

solution = iterative_deepening_dfs(start_state, goal_state)
for step in solution:
    print(step)
```

Program 3

Implement A* search algorithm







```
import heapq

print("Shreyas t s 1BM23CS322")
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

def manhattan_distance(state, goal):
    distance = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                goal_x, goal_y = divmod(goal_state[i][j] - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
```

```
    return distance

def find_empty(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_empty(state)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny],
new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star(start_state, goal_state):
    open_list = []
    heapq.heappush(open_list, (manhattan_distance(start_state, goal_state), 0,
start_state, []))
    closed_list = set()

    while open_list:
        _, g, current_state, path = heapq.heappop(open_list)
        if current_state == goal_state:
            return path + [current_state]

        closed_list.add(tuple(map(tuple, current_state)))

        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in closed_list:
                f = g + 1 + manhattan_distance(neighbor, goal_state)
                heapq.heappush(open_list, (f, g + 1, neighbor, path +
[current_state]))

    return None
```

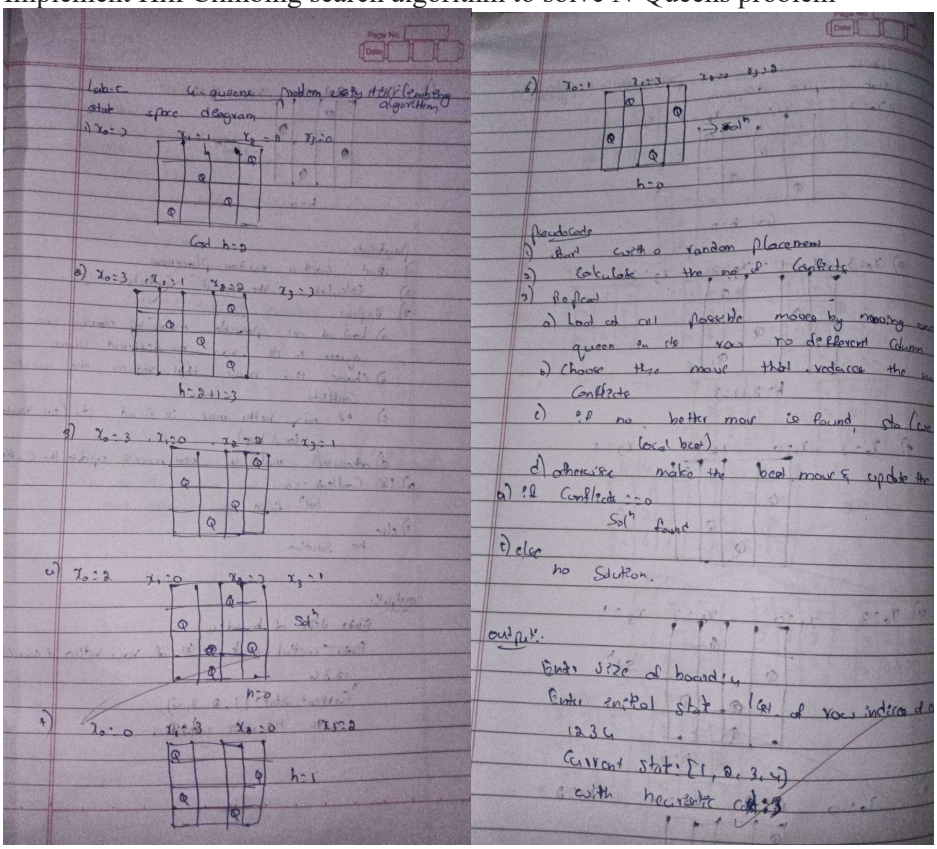
```
start_state = [[1, 2, 3],
               [4, 5, 6],
               [7, 8, 0]]

solution = a_star(start_state, goal_state)
for step in solution:
   print(step)
```

```
import heapq

print("Shreyas ts  1BM23CS322")
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]

def misplaced_tiles(state, goal):
   count = 0
   for i in range(3):
       for j in range(3):
           if state[i][j] != goal[i][j] and state[i][j] != 0:
               count += 1
   return count

def find_empty(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_empty(state)
   directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
   for dx, dy in directions:
       nx, ny = x + dx, y + dy
       if 0 <= nx < 3 and 0 <= ny < 3:
           new_state = [row[:] for row in state]
           new_state[x][y], new_state[nx][ny] = new_state[nx][ny],
```

```
new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star(start_state, goal_state):
    open_list = []
    heapq.heappush(open_list, (misplaced_tiles(start_state, goal_state), 0,
start_state, []))
    closed_list = set()

    while open_list:
        _, g, current_state, path = heapq.heappop(open_list)
        if current_state == goal_state:
            return path + [current_state]

        closed_list.add(tuple(map(tuple, current_state)))

        for neighbor in get_neighbors(current_state):
            if tuple(map(tuple, neighbor)) not in closed_list:
                f = g + 1 + misplaced_tiles(neighbor, goal_state)
                heapq.heappush(open_list, (f, g + 1, neighbor, path +
[current_state]))

    return None

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

solution = a_star(start_state, goal_state)
for step in solution:
    print(step)
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

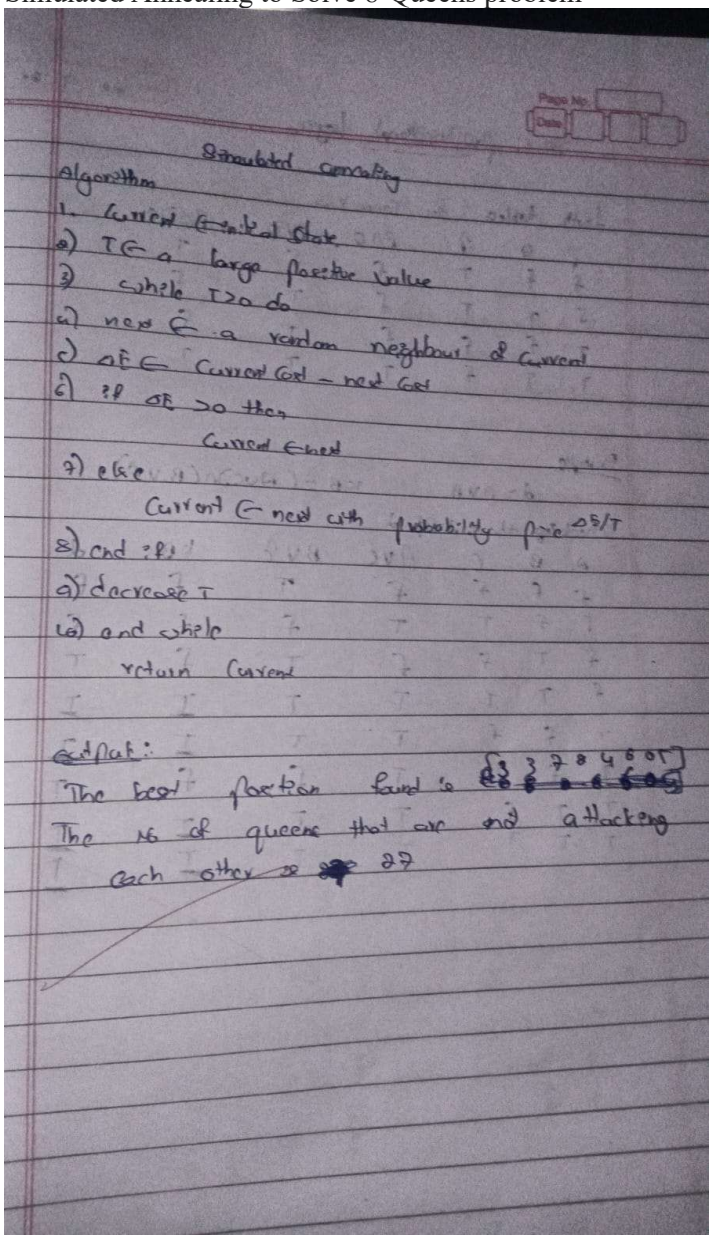


```
import random

def create_board():
    return [random.randint(0, 3) for _ in range(4)]

def calculate_conflicts(board):
    conflicts = 0
    for i in range(4):
        for j in range(i + 1, 4):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

def hill_climbing():
    board = create_board()
    print(f"Initial board: {board} with conflicts: {calculate_conflicts(board)}")

    while True:
        current_conflicts = calculate_conflicts(board)
        if current_conflicts == 0:
            return board
        next_board = None
```

```
        next_conflicts = float('inf')
        for i in range(4):
            temp_board = board[:]
            for j in range(4):
                if temp_board[i] != j:
                    temp_board[i] = j
                    temp_conflicts = calculate_conflicts(temp_board)
                    print(f"Board: {temp_board} with conflicts: {temp_conflicts}")
                    if temp_conflicts < next_conflicts:
                        next_conflicts = temp_conflicts
                        next_board = temp_board[:]
        if next_conflicts >= current_conflicts:
            return board
        board = next_board

solution = hill_climbing()
print(f"Final solution: {solution}")
```

Program 5

Simulated Annealing to Solve 8-Queens problem

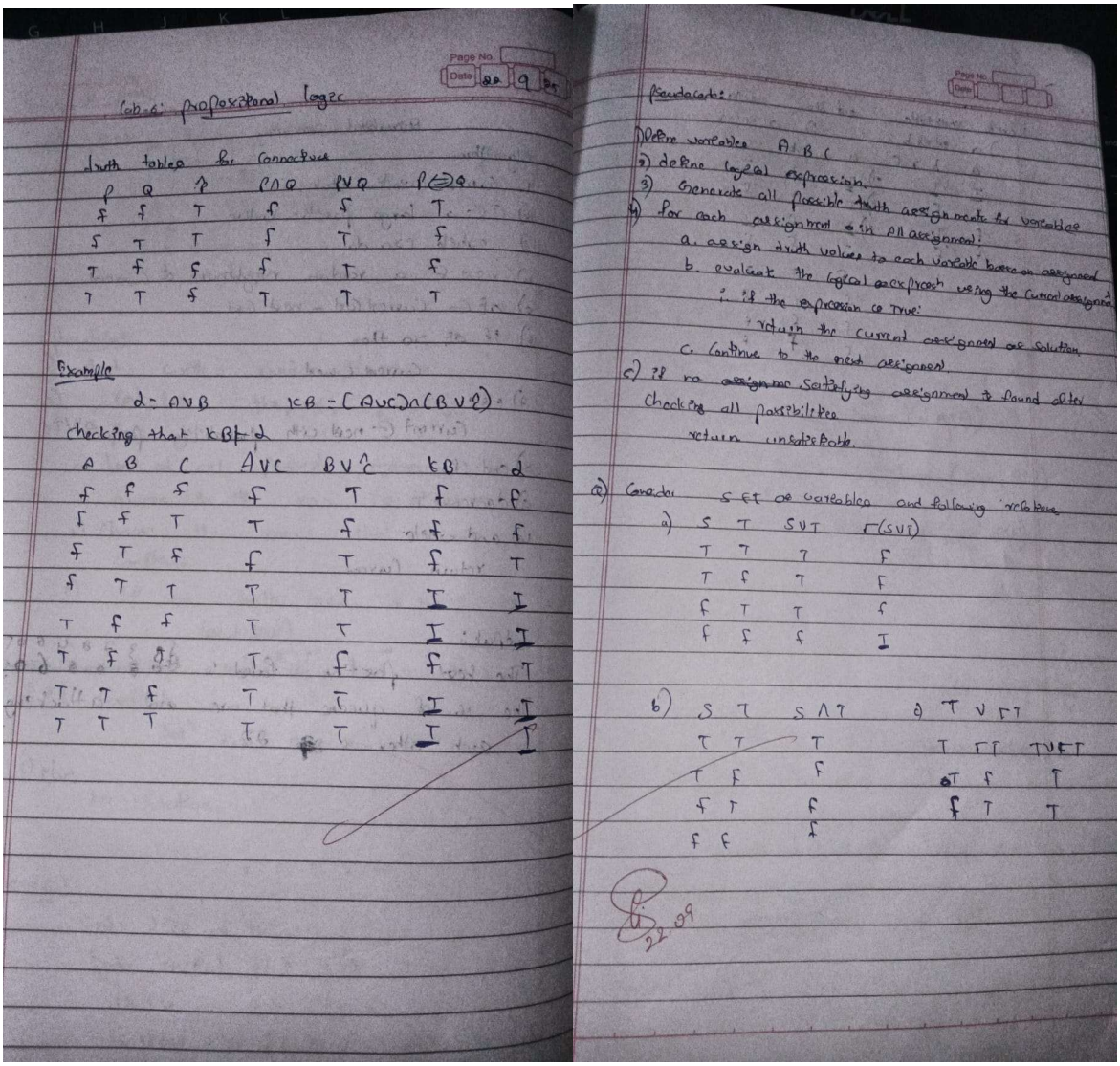



```
import random
import math

print("Shreyas ts  1BM23CS322")
def objective_function(state):
    return sum(state)

def generate_neighbor(state):
    neighbor = state[:]
    index = random.randint(0, len(state) - 1)
```

```
    neighbor[index] = random.randint(1, 100)
    return neighbor

def simulated_annealing(initial_state, initial_temp, cooling_rate,
iterations):
    current_state = initial_state
    current_temp = initial_temp
    best_state = current_state
    best_score = objective_function(current_state)

    for i in range(iterations):
        neighbor = generate_neighbor(current_state)
        current_score = objective_function(current_state)
        neighbor_score = objective_function(neighbor)
        if neighbor_score < best_score:
            best_state = neighbor
            best_score = neighbor_score

        if neighbor_score < current_score or random.random() <
math.exp((current_score - neighbor_score) / current_temp):
            current_state = neighbor

        current_temp *= cooling_rate

    return best_state, best_score

initial_state = [random.randint(1, 100) for _ in range(10)]
initial_temp = 1000
cooling_rate = 0.99
iterations = 1000

best_state, best_score = simulated_annealing(initial_state, initial_temp,
cooling_rate, iterations)
print("Best State:", best_state)
print("Best Score:", best_score)
```

Program 6

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

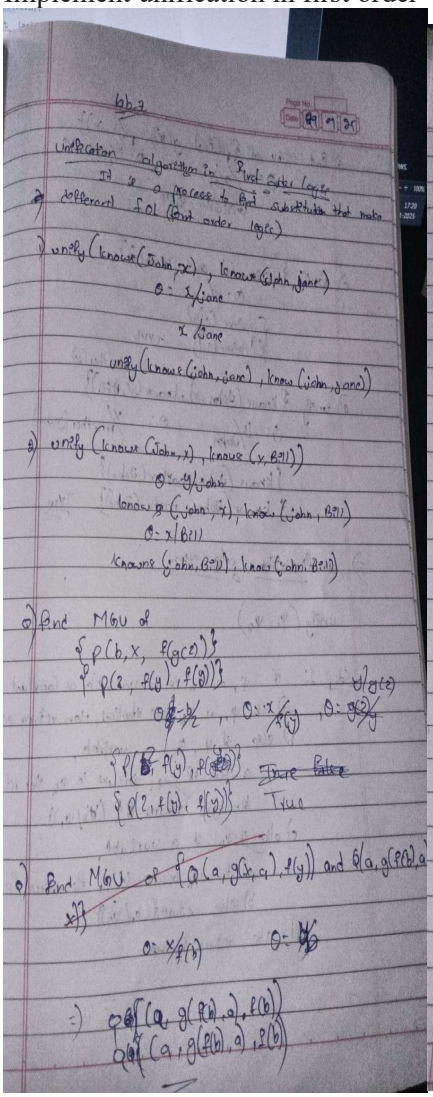


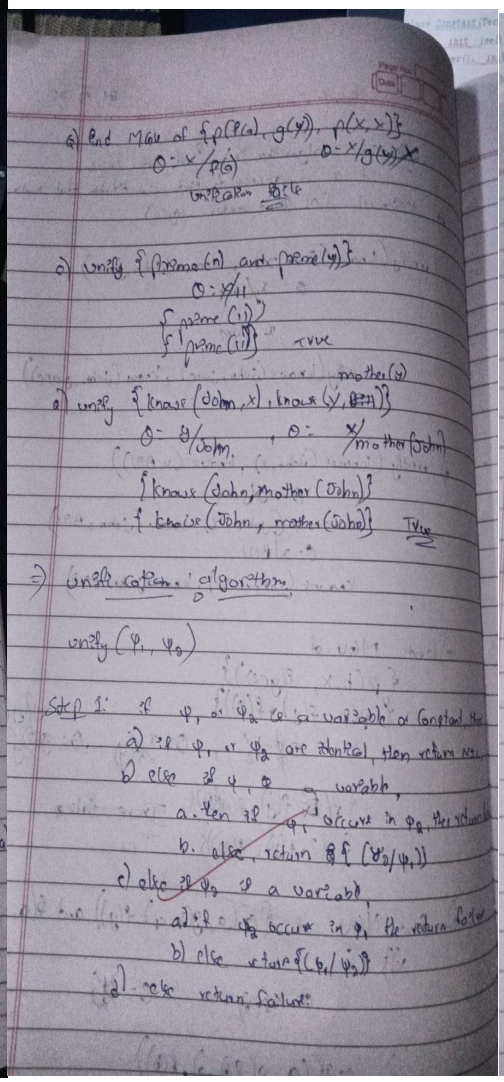
```
import itertools

def evaluate_expression(expression, assignment):
    expression = expression.replace('A', str(assignment[0]))
    expression = expression.replace('B', str(assignment[1]))
    expression = expression.replace('C', str(assignment[2]))
    expression = expression.replace('and', 'and').replace('or', 'or').replace('not', 'not')
    return eval(expression)

def check_entailment(alpha, kb):
    variables = ['A', 'B', 'C']
    all_assignments = list(itertools.product([False, True], repeat=len(variables)))
```

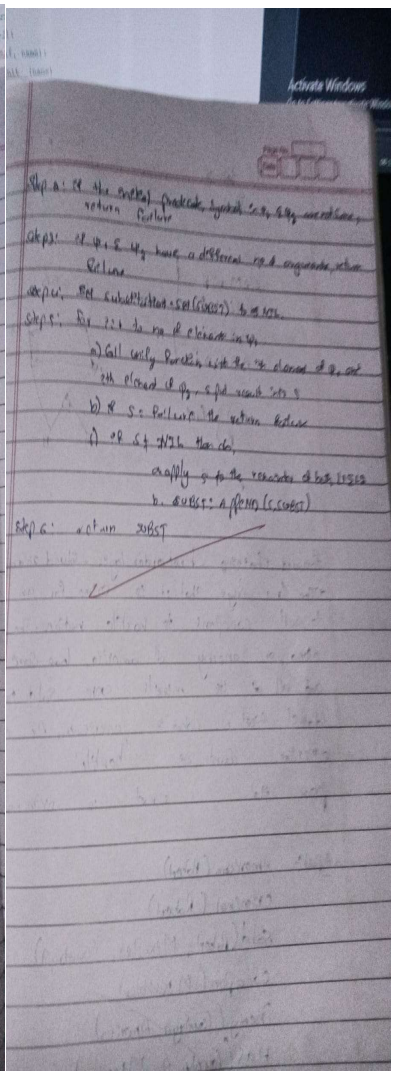
```
    print("Truth Table:")
    print("A\tB\tC\tKB\tα")
    for assignment in all_assignments:
        kb_result = evaluate_expression(kb, assignment)
        alpha_result = evaluate_expression(alpha, assignment)

        print(f"{assignment[0]}\t{assignment[1]}\t{assignment[2]}\t{kb_result}\t{alpha_result}")

        if kb_result and not alpha_result:
            return False

    return True

alpha = "A or B"
kb = "(A or C) and (B and not C)"

if check_entailment(alpha, kb):
    print("KB entails α")
else:
    print("KB does not entail α")
```

Program 7

Implement unification in first order logic.







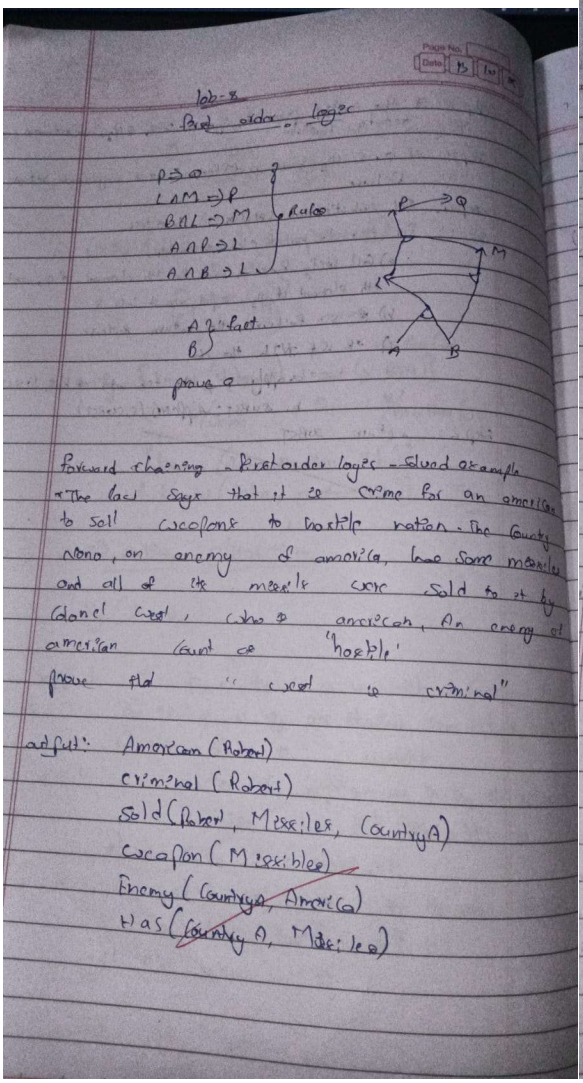
```
print("Shreyas ts  1BM23CS322")
class Term:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return self.name

class Variable(Term):
    def __init__(self, name):
        super().__init__(name)

class Constant(Term):
```

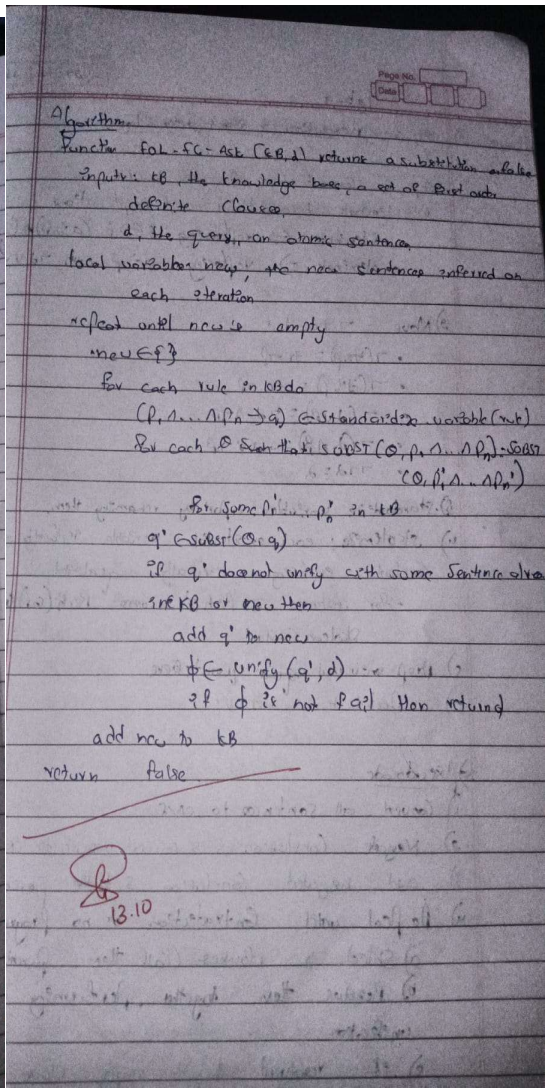
```
    def __init__(self, name):
        super().__init__(name)

class Predicate:
    def __init__(self, symbol, terms):
        self.symbol = symbol
        self.terms = terms

    def __repr__(self):
        return f"{self.symbol}({', '.join(map(str, self.terms))})"

def unify(x, y, subst):
    if subst is None:
        return None

    if x == y:
        return subst

    elif isinstance(x, Variable) and not isinstance(y, Variable):
        return unify_var(x, y, subst)

    elif isinstance(y, Variable) and not isinstance(x, Variable):
        return unify_var(y, x, subst)

    elif isinstance(x, Predicate) and isinstance(y, Predicate):
        if x.symbol != y.symbol:
            raise Exception(f"Cannot unify predicates with different symbols:
{x} and {y}")
        else:
            return unify_lists(x.terms, y.terms, subst)

    else:
        raise Exception(f"Unification failed: {x} and {y} are not unifiable")

def unify_var(var, term, subst):
    if var in subst:
        return unify(subst[var], term, subst)

    if occurs_check(var, term):
        raise Exception(f"Occurs check failed: {var} cannot be unified with
{term}")
```

```
    subst[var] = term
    return subst

def occurs_check(var, term):
    if isinstance(term, Variable):
        return term == var
    elif isinstance(term, Predicate):
        return any(occurs_check(var, t) for t in term.terms)
    return False

def unify_lists(list_x, list_y, subst):
    if len(list_x) != len(list_y):
        raise Exception(f"Cannot unify lists of different lengths: {list_x}
and {list_y}")

    for term_x, term_y in zip(list_x, list_y):
        subst = unify(term_x, term_y, subst)
        if subst is None:
            return None
    return subst

if __name__ == "__main__":
    x = Variable('X')
    y = Variable('Y')
    c = Constant('a')
    p1 = Predicate('father', [x, c])
    p2 = Predicate('father', [y, c])

    substitution = unify(p1, p2, {})
    if substitution is not None:
        print("Unification succeeded:", substitution)
    else:
        print("Unification failed.")
```

Program 8

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



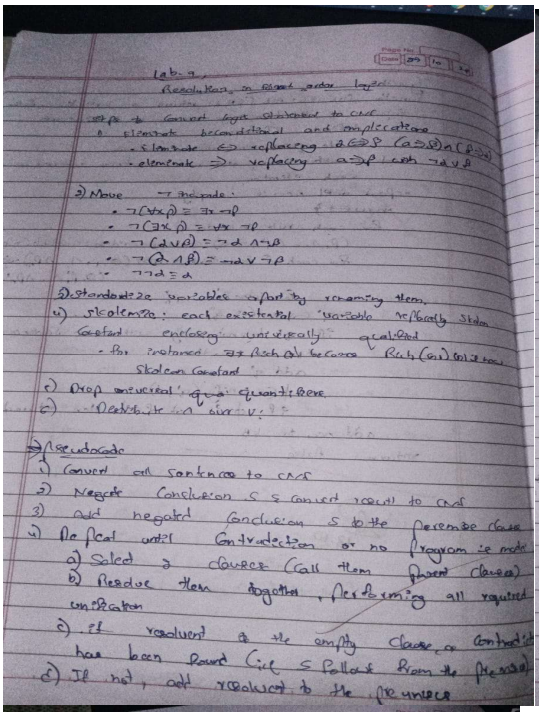


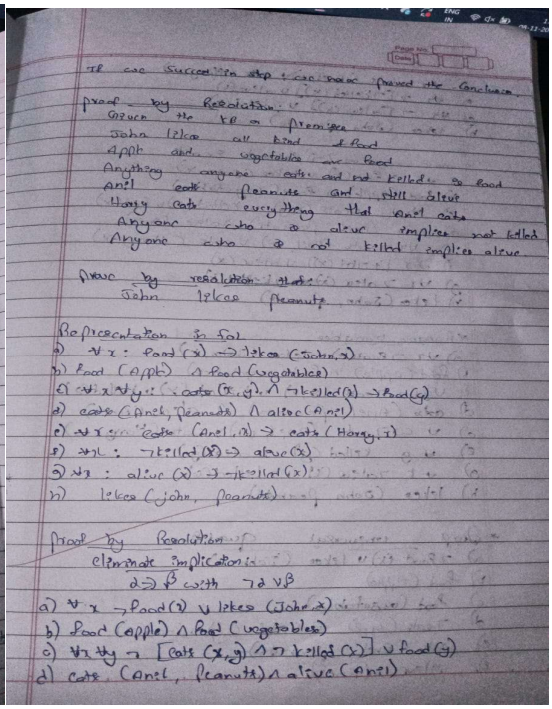

```
def parse_clause(clause):
    if "=>" in clause:
        premises, conclusion = clause.split("=>")
        premises = [p.strip() for p in premises.split("&")]
        conclusion = conclusion.strip()
        return premises, conclusion
    else:
        return [], clause.strip()

def forward_reasoning(kb, query):
    known = set()
    rules = []
    for clause in kb:
        premises, conclusion = parse_clause(clause)
```

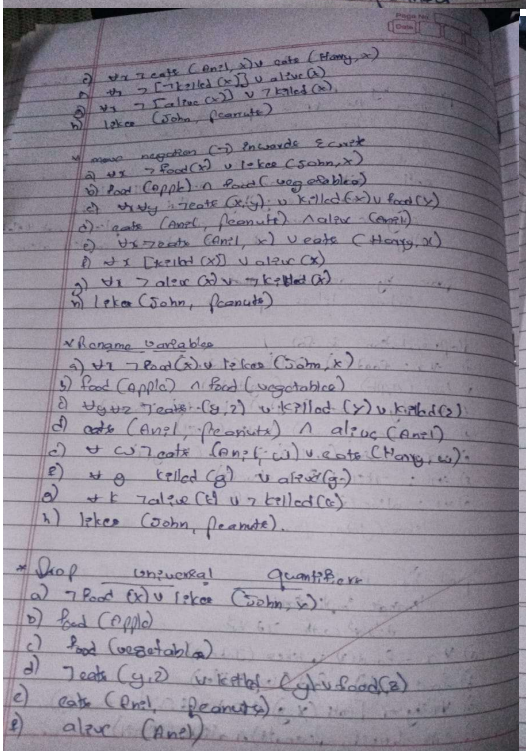
```
        if not premises:
            known.add(conclusion)
        else:
            rules.append((premises, conclusion))

    added = True
    while added:
        added = False
        for premises, conclusion in rules:
            if all(p in known for p in premises) and conclusion not in known:
                known.add(conclusion)
                added = True
            if query in known:
                return True
    return query in known

n = int(input("Enter number of statements in KB: "))
kb = [input(f"Statement {i+1}: ") for i in range(n)]
query = input("Enter query: ")

if forward_reasoning(kb, query):
    print("Query is proved by forward reasoning")
else:
    print("Query cannot be proved")
```

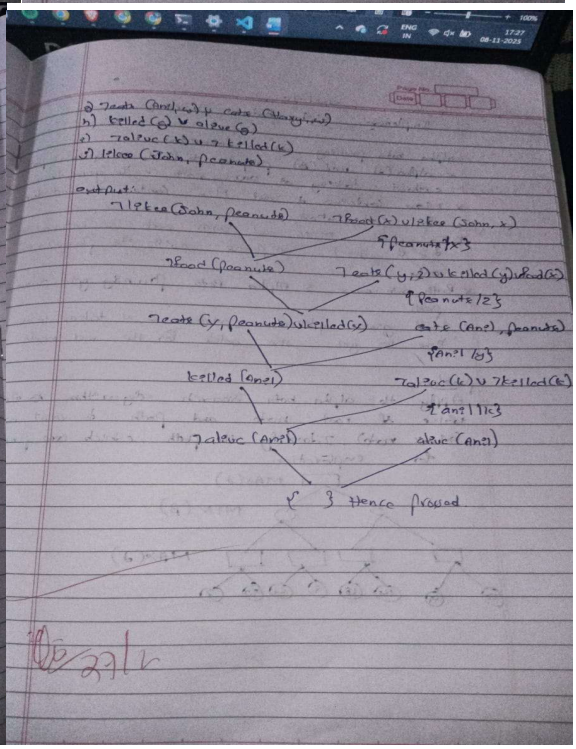
Program 9

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







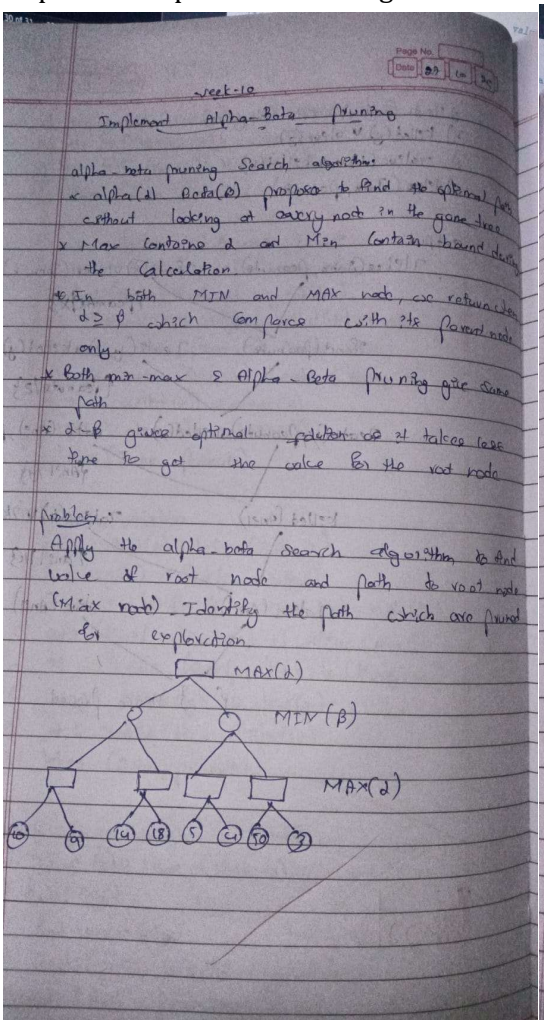


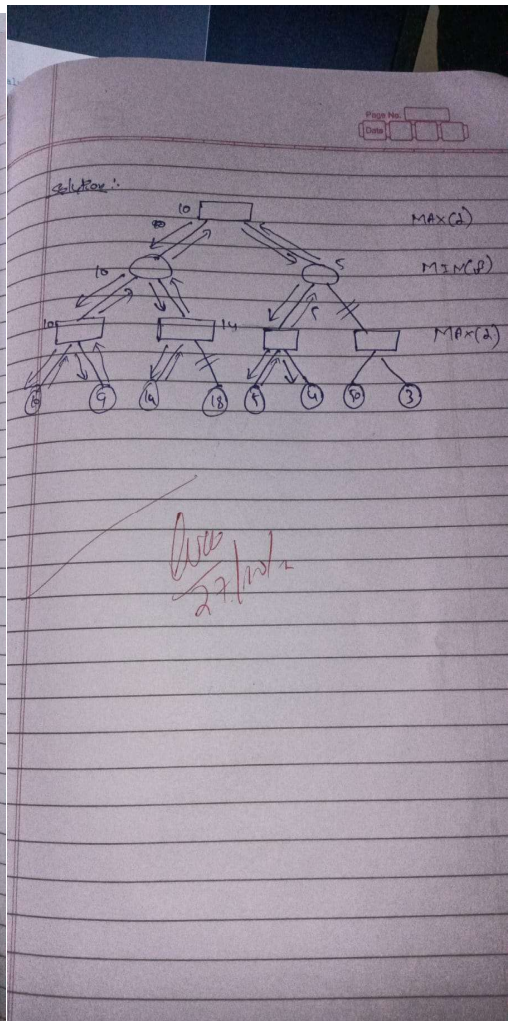
```
from sympy import symbols, Or, Not


P, Q, R = symbols('P Q R')
kb = [
```

```
    (Not(P), Q),
    (Not(Q), R),
    (P,)
]
query = R

def resolve(ci, cj):
    resolvents = set()
    for di in ci:
        for dj in cj:
            if di == Not(dj) or Not(di) == dj:
                new_clause = tuple(sorted(set(ci + cj) - {di, dj}, key=str))
                if not new_clause:
                    resolvents.add(())
                else:
                    resolvents.add(new_clause)
    return resolvents

def pl_resolution(kb, query):
    clauses = kb + [(Not(query),)]
    new = set()
    while True:
        pairs = [(clauses[i], clauses[j]) for i in range(len(clauses)) for j
in range(i+1, len(clauses))]
        for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)
            if () in resolvents:
                return True
            new.update(resolvents)
        if new.issubset(set(clauses)):
            return False
        for c in new:
            if c not in clauses:
                clauses.append(c)

print(pl_resolution(kb, query))
```

Program 10

Implement Alpha-Beta Pruning.






```
import math

def alpha_beta(depth, node_index, is_maximizing, values, alpha, beta,
max_depth):
    indent = "    " * depth
    if depth == max_depth:
        print(f"{indent}Leaf node[{node_index}] = {values[node_index]}")
        return values[node_index]

    if is_maximizing:
        best = -math.inf
        print(f"{indent}MAX node (α={alpha}, β={beta})")
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, False, values,
```

```
alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            print(f"{indent}   MAX updating α={alpha}")
            if beta <= alpha:
                print(f"{indent}   β cut-off (β={beta}, α={alpha}) ")
                break
        return best
    else:
        best = math.inf
        print(f"{indent}MIN node (α={alpha}, β={beta})")
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, True, values,
alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
            print(f"{indent}   MIN updating β={beta}")
            if beta <= alpha:
                print(f"{indent}   α cut-off (β={beta}, α={alpha}) ")
                break
        return best
values = [10, 9, 14, 18, 5, 4, 50, 3]
max_depth = int(math.log2(len(values)))

print("Leaf Nodes:", values)
result = alpha_beta(0, 0, True, values, -math.inf, math.inf, max_depth)
print("\nOptimal value:", result)
```