VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

AAMAL MOHSIN MAGDUM (1BM23CS002)

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-2025 to Jan-2026

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 **Aamal Mohsin Magdum (1BM23CS002),** 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.

Prof. Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Joythi S Nayak Professor & HOD Department of CSE, BMSCE

Index

Course Certificate:


Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aamal Magdum

for successfully completing the course

Introduction to Artificial Intelligence

on November 18, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com


Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aamal Magdum

for successfully completing the course

Introduction to Natural Language Processing

on November 18, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com


Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Aamal Magdum

for successfully completing the course

Introduction to Deep Learning

on November 18, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com


Infosys
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Aamal Magdum

for successfully completing

Artificial Intelligence Foundation Certification

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B. N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

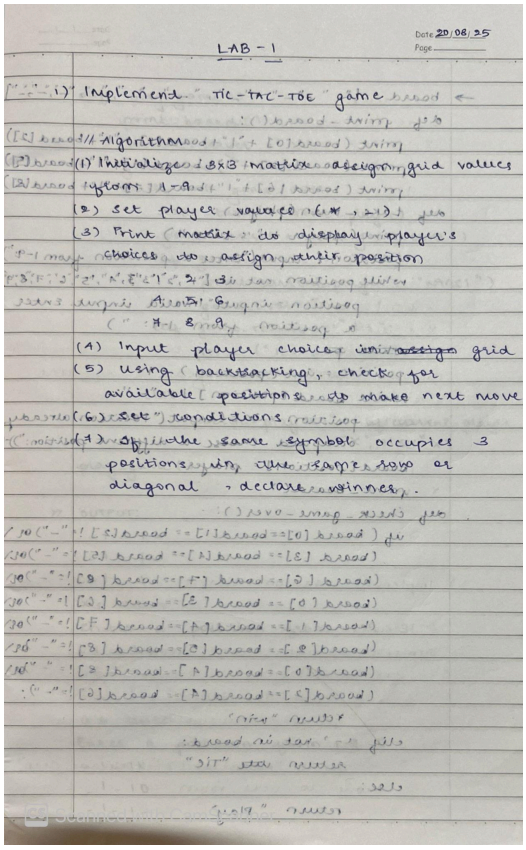
Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

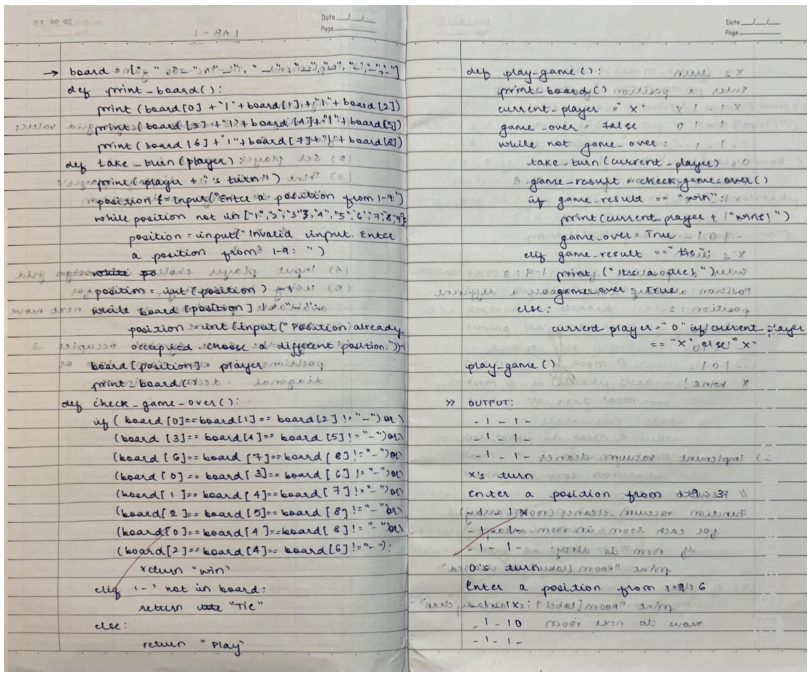
Github Link:
https://github.com/amalmagdum7/AI

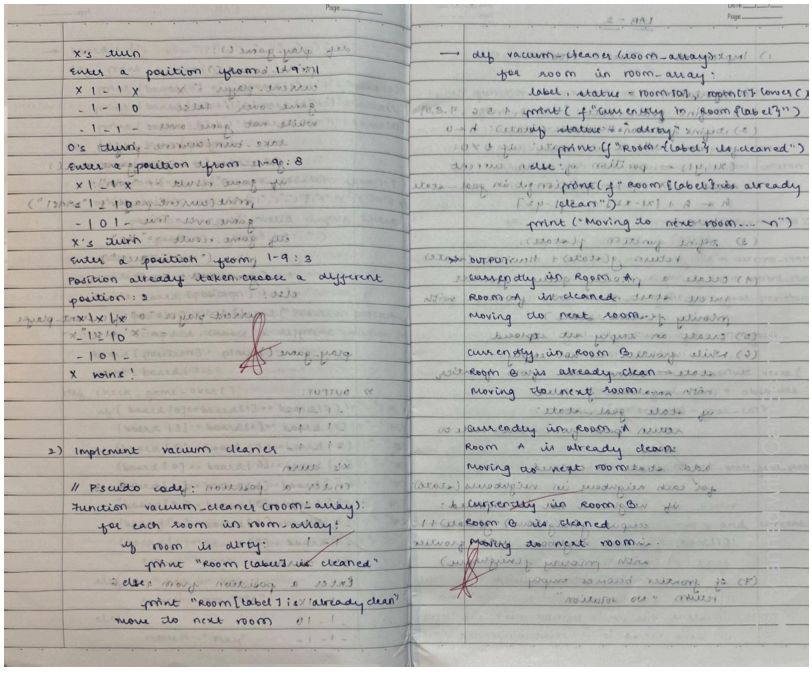
Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:







Code:

```
//tic-tac-toe
board = ["-", "-", "-",
        "-", "-", "-",
        "-", "-", "-"]

def print_board():
    print(board[0] + "|" + board[1] + "|" + board[2])
    print(board[3] + "|" + board[4] + "|" + board[5])
    print(board[6] + "|" + board[7] + "|" + board[8])
```

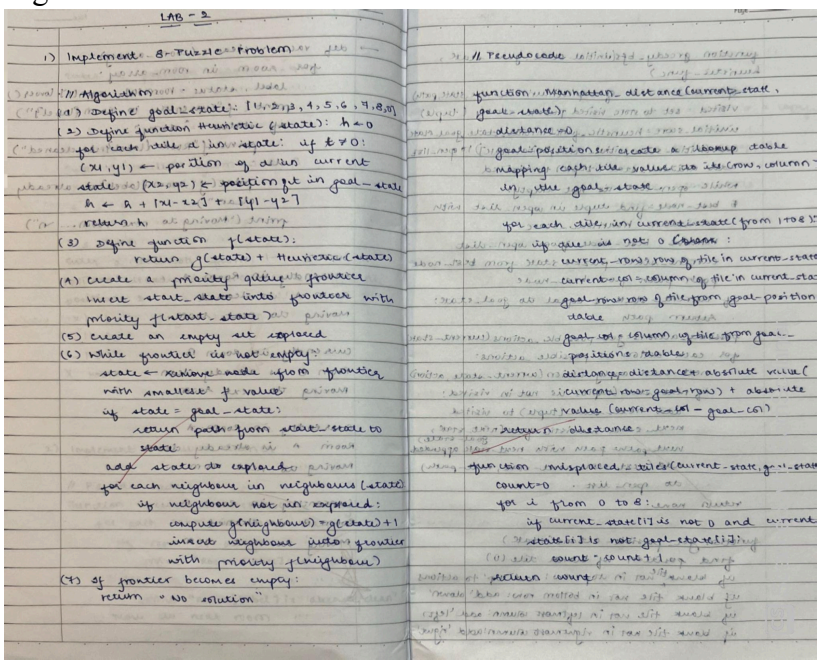
```
def take_turn(player):
    print(player + "'s turn")
    position = input("Enter a position from 1-9: ")
    while position not in ["1", "2", "3", "4", "5", "6", "7", "8", "9"]:
        position = input("Invalid input. Choose a position from 1-9: ")
    position = int(position) - 1
    while board[position] != "-":
        position = int(input("Position already occupied. Choose a different position: ")) - 1
    board[position] = player
    print_board()

def check_game_over():
    if (board[0] == board[1] == board[2] != "-") or \
       (board[3] == board[4] == board[5] != "-") or \
       (board[6] == board[7] == board[8] != "-") or \
       (board[0] == board[3] == board[6] != "-") or \
       (board[1] == board[4] == board[7] != "-") or \
       (board[2] == board[5] == board[8] != "-") or \
       (board[0] == board[4] == board[8] != "-") or \
       (board[2] == board[4] == board[6] != "-"):
        return "win"

    elif '-' not in board:
        return "Tie"

    else:
        return "Play"

def play_game():
    print_board()
    current_player = "X"
    game_over = False
    while not game_over:
        take_turn(current_player)
        game_result = check_game_over()
        if game_result == "win":
            print(current_player + " wins!")
            game_over = True
        elif game_result == "tie":
            print("It's a tie!")
            game_over = True
        else:
            current_player = "O" if current_player == "X" else "X"

play_game()
```

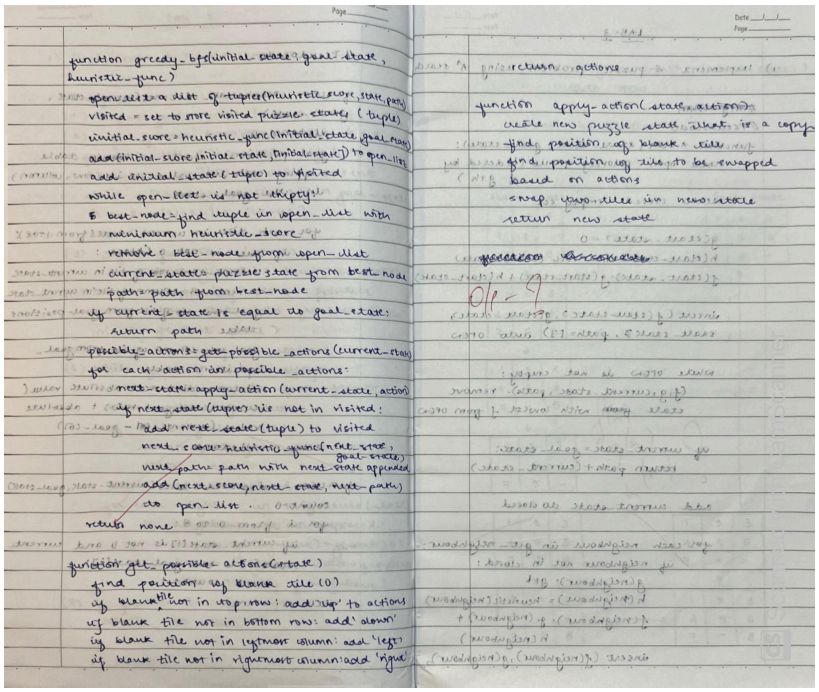
```
//vacuum cleaner
def vacuum_cleaner(room_array):
    for room in room_array:
        label, status = room[0], room[1].lower()
        print(f"Currently in Room {label}")
        if status == "dirty":
            print(f"Room {label} is cleaned")
        else:
            print(f"Room {label} is already clean")
        print("Moving to next room...\n")


rooms = [('A', 'dirty'), ('B', 'clean'), ('A', 'clean'), ('B', 'dirty')]
vacuum_cleaner(rooms)
```

Program 2

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

Algorithm:





Code:

```
//8-puzzle(dfs)
def find_possible_moves(state):
    index = state.index('_')
    moves = {
```

```
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [6, 8, 4],
        8: [5, 7],
    }
    return moves.get(index, [])

def dfs(initial_state, goal_state, max_depth=50):
    stack = [(initial_state, [], 0)]
    visited = {tuple(initial_state)}
    states_explored = 0
    printed_depths = set()

    while stack:
        current_state, path, depth = stack.pop()

        if depth > max_depth:
            continue

        if depth not in printed_depths:
            print(f"\n--- Depth {depth} ---")
            printed_depths.add(depth)

        states_explored += 1
        print(f"State #{states_explored}: {current_state}")

        if current_state == goal_state:
            print(f"\n Goal reached at depth {depth} after exploring {states_explored} states.\n")
            return path

        possible_moves_indices = find_possible_moves(current_state)

        for move_index in reversed(possible_moves_indices):  # Reverse for DFS order
            next_state = list(current_state)
            blank_index = next_state.index('_')
            next_state[blank_index], next_state[move_index] = next_state[move_index],
next_state[blank_index]

            if tuple(next_state) not in visited:
                visited.add(tuple(next_state))
                stack.append((next_state, path + [next_state], depth + 1))
```

```
    print(f"\n Goal state not reachable within depth {max_depth}. Explored {states_explored}
states.\n")
    return None

initial_state = [1, 2, 3, 4, 8, '_', 7, 6, 5]
goal_state    = [1, 2, 3, 4, 5, 6, 7, 8, '_']


solution_path = dfs(initial_state, goal_state, max_depth=50)


if solution_path is None:
    print("No solution found.")
else:
    print("Solution path:")
    for step, state in enumerate(solution_path, start=1):
        print(f"Step {step}: {state}")

//iterative deepening search
import time

# ----------------- MOVE GENERATOR -----------------
def find_possible_moves(state):
    index = state.index('_')

    if index == 0:
        return [1, 3]
    elif index == 1:
        return [0, 2, 4]
    elif index == 2:
        return [1, 5]
    elif index == 3:
        return [0, 4, 6]
    elif index == 4:
        return [1, 3, 5, 7]
    elif index == 5:
        return [2, 4, 8]
    elif index == 6:
        return [3, 7]
    elif index == 7:
        return [4, 6, 8]
    elif index == 8:
        return [5, 7]
    return []

# ----------------- DEPTH LIMITED SEARCH -----------------
def depth_limited_dfs(state, goal_state, limit, path, visited):
```

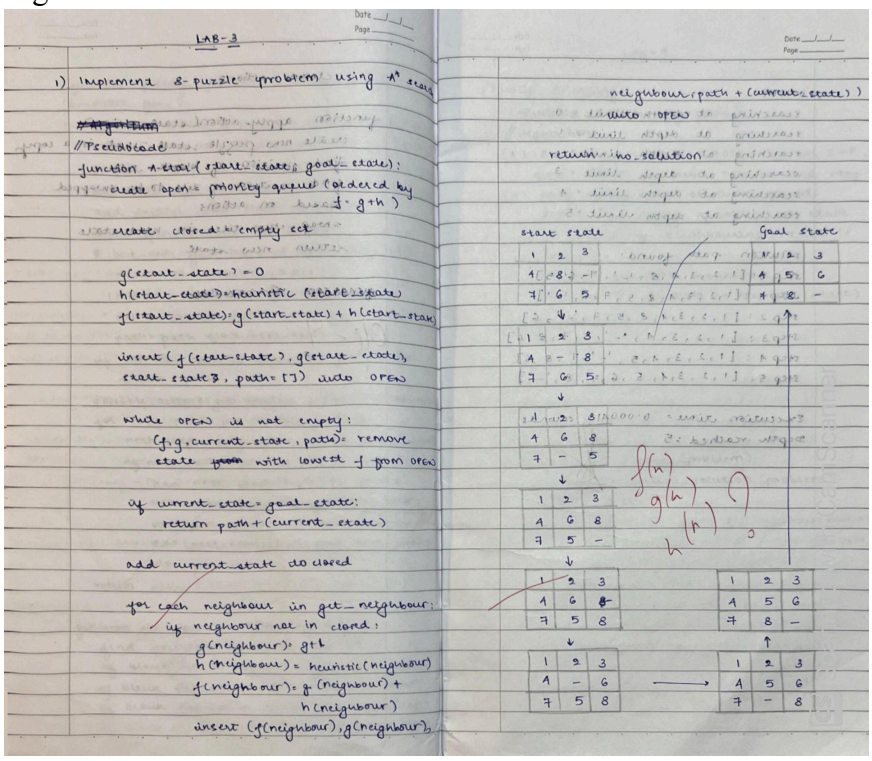
```
    if state == goal_state:
        return path

    if limit <= 0:
        return None

    visited.add(tuple(state))

    for move_index in find_possible_moves(state):
        next_state = list(state)
        blank_index = next_state.index('_')
        next_state[blank_index], next_state[move_index] = next_state[move_index],
next_state[blank_index]

        if tuple(next_state) not in visited:
            result = depth_limited_dfs(next_state, goal_state, limit - 1, path + [next_state], visited)
            if result is not None:
                return result
    return None

# ----------------- ITERATIVE DEEPENING DFS -----------------
def iddfs(initial_state, goal_state, max_depth=30):
    for depth in range(max_depth):
        print(f"Searching at depth limit = {depth}")
        visited = set()
        result = depth_limited_dfs(initial_state, goal_state, depth, [initial_state], visited)
        if result is not None:
            return result, depth
    return None, max_depth

# ----------------- TEST -----------------
initial_state = [1, 2, 3,
                 4, 8, '_',
                 7, 6, 5]

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

# Measure execution time
start_time = time.time()
solution_path, depth_reached = iddfs(initial_state, goal_state, max_depth=30)
end_time = time.time()

if solution_path is None:
    print("Goal state is not reachable within given depth limit.")
else:
```

```
    print("\n\nSolution path found:")
    for step, state in enumerate(solution_path, start=0):
        print(f"Step {step}: {state}")

print("\nExecution time: {:.6f} seconds".format(end_time - start_time))
print("Depth reached:", depth_reached)
```

Program 3

Implement A* search algorithm

Algorithm:



LAB-3

1) Implement 8-puzzle problem using A* search

// Pseudocode

function A*start(start_state, goal_state):

create open = priority queue (ordered by f = g+h)

create closed = empty set

g(start_state) = 0

h(start_state) = heuristic(start_state)

f(start_state) = g(start_state) + h(start_state)

insert(f(start_state), g(start_state), start_state, path=[]) into OPEN

while OPEN is not empty:

(f, g, current_state, path) = remove state from with lowest f from open

if current_state = goal_state:

return path + (current_state)

add current_state to closed

for each neighbour in get_neighbour:

if neighbour not in closed:

g(neighbour) = g+1

h(neighbour) = heuristic(neighbour)

f(neighbour) = g(neighbour) + h(neighbour)

insert(f(neighbour), g(neighbour),

neighbour, path + (current state))

return no solution

Start state

1	2	3
4	8	-
7	6	5

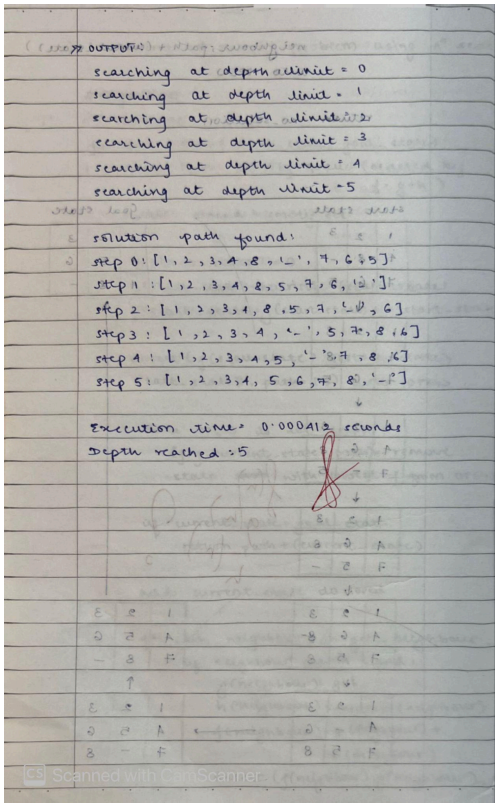
Goal state

1	2	3
4	5	6
7	8	-

f(n)

g(n)

h(n)



OUTPUT:

searching at depth limit = 0

searching at depth limit = 1

searching at depth limit = 2

searching at depth limit = 3

searching at depth limit = 4

searching at depth limit = 5

solution path found:

step 0: [1, 2, 3, 4, 8, '_', 7, 6, 5]

step 1: [1, 2, 3, 4, 8, 5, 7, 6, '_']

step 2: [1, 2, 3, 4, 8, 5, 7, '_', 6]

step 3: [1, 2, 3, 4, '_', 5, 7, 8, 6]

step 4: [1, 2, 3, 4, 5, '_', 7, 8, 6]

step 5: [1, 2, 3, 4, 5, 6, 7, 8, '_']

Execution time: 0.000413 seconds

Depth reached: 5

Code:

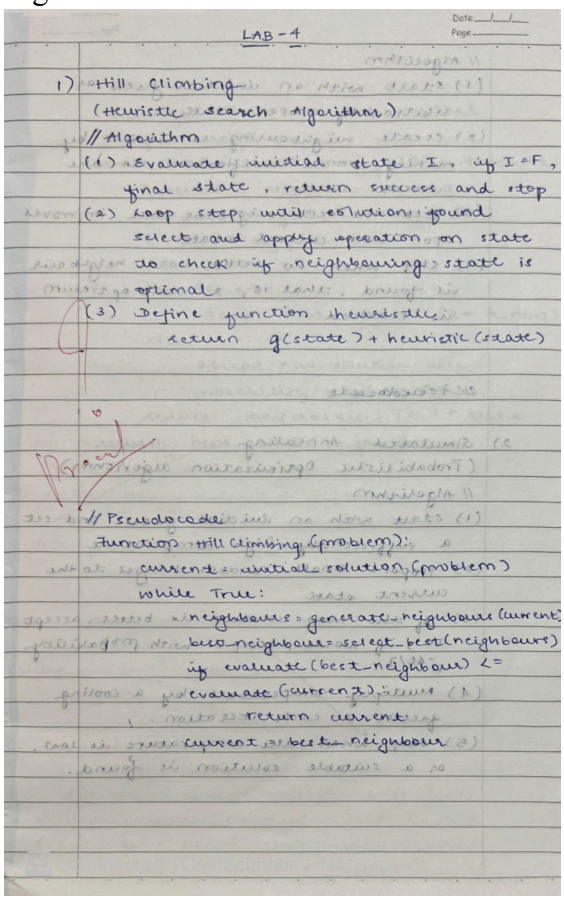
```
import heapq
import time

# Heuristic: Manhattan Distance
def heuristic(state, goal):
    distance = 0
    for i in range(1, 9):  # tile numbers 1 to 8
        x1, y1 = divmod(state.index(i), 3)
        x2, y2 = divmod(goal.index(i), 3)
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance

# Get neighbors by sliding blank (0) up/down/left/right
def get_neighbors(state):
    neighbors = []
    i = state.index(0)  # position of blank
    x, y = divmod(i, 3)
    moves = [(-1,0), (1,0), (0,-1), (0,1)]

    for dx, dy in moves:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            j = new_x * 3 + new_y
            new_state = list(state)
            new_state[i], new_state[j] = new_state[j], new_state[i]
            neighbors.append(tuple(new_state))
    return neighbors

# A* Search for 8-puzzle
def astar(start, goal):
    open_set = []
    heapq.heappush(open_set, (heuristic(start, goal), 0, start))

    came_from = {}
    g_score = {start: 0}

    while open_set:
        _, cost, current = heapq.heappop(open_set)

        if current == goal:
            # Reconstruct path
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
```

```
            path.append(start)
            return path[::-1]

        for neighbor in get_neighbors(current):
            tentative_g = g_score[current] + 1
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + heuristic(neighbor, goal)
                heapq.heappush(open_set, (f_score, tentative_g, neighbor))

    return None  # no solution

# ----------------- TEST -----------------
start = (1, 2, 3,
         4, 8, 0,
         7, 6, 5)

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

# Measure execution time
start_time = time.time()
path = astar(start, goal)
end_time = time.time()

if path:
    print("Steps to solve ({} moves):".format(len(path)-1))
    for state in path:
        for i in range(0, 9, 3):
            print(state[i:i+3])
        print()
else:
    print("No solution found")

print("Execution time: {:.6f} seconds".format(end_time - start_time))
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

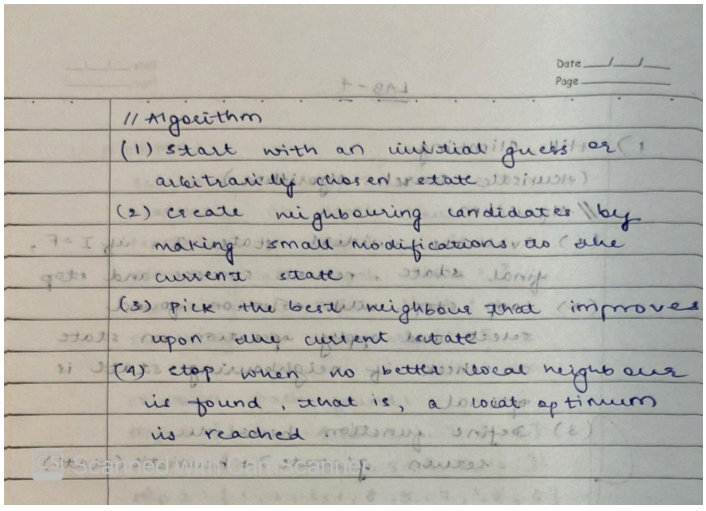


LAB - 4

1) Hill Climbing
(Heuristic search Algorithm)

// Algorithm
(1) Evaluate initial state I, if I = F, final state, return success and stop
(2) Loop step until solution found
select and apply operation on state to check if neighbouring state is optimal
(3) Define function heuristic;
return g(state) + heuristic(state)

// Pseudocode
function Hill Climbing (problem):
current = initial_solution(problem)
while True:
neighbours = generate_neighbours(current)
best_neighbour = select_best(neighbours)
if evaluate(best_neighbour) <=
evaluate(current):
return current
current = best_neighbour



// Algorithm
(1) start with an initial guess or arbitrarily chosen state
(2) create neighbouring candidates by making small modifications to the current state.
(3) pick the best neighbour that improves upon the current state
(4) stop when no better local neighbour is found, that is, a local optimum is reached

Code:

```
import random
import math
```

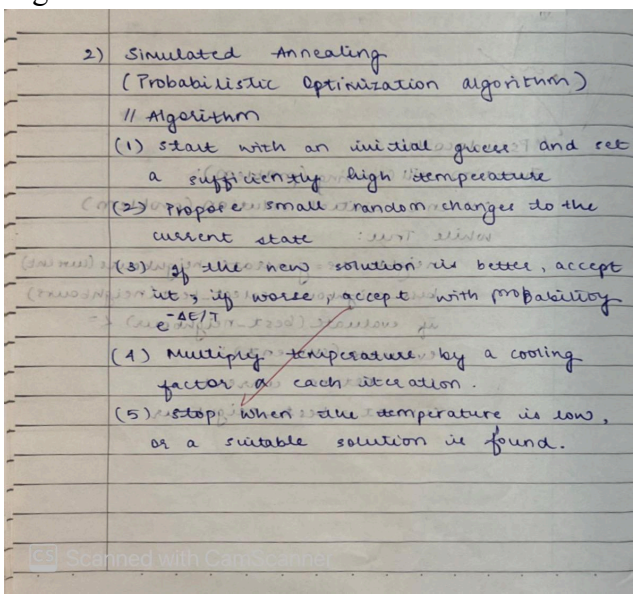
```
def compute_cost(state):
    """Count diagonal conflicts for a permutation-state (one queen per row & column)."""
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def random_permutation(n):
    arr = list(range(n))
    random.shuffle(arr)
    return arr

def neighbors_by_swaps(state):
    """All neighbors obtained by swapping two columns (keeps permutation property)."""
    n = len(state)
    for i in range(n - 1):
        for j in range(i + 1, n):
            nb = state.copy()
            nb[i], nb[j] = nb[j], nb[i]
            yield nb

def hill_climb_with_restarts(n, max_restarts=None):
    """Hill climbing on permutations with random restart on plateau (no revisits)."""
    visited = set()
    total_states = math.factorial(n)
    restarts = 0

    while True:
        # pick a random unvisited start permutation
        if len(visited) >= total_states:
            raise RuntimeError("All states visited — giving up (no solution found).")

        state = random_permutation(n)
        while tuple(state) in visited:
            state = random_permutation(n)
        visited.add(tuple(state))

        # climb from this start
        while True:
            cost = compute_cost(state)
            if cost == 0:
                return state, restarts

            # find best neighbor (swap-based neighbors)
```

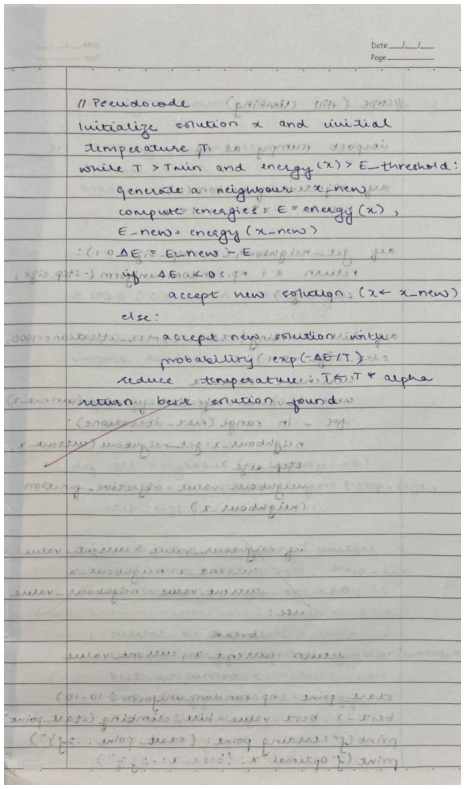
```
            best_neighbor = None
            best_cost = float("inf")
            for nb in neighbors_by_swaps(state):
                c = compute_cost(nb)
                if c < best_cost:
                    best_cost = c
                    best_neighbor = nb

            # if strictly better, move; otherwise it's a plateau/local optimum -> restart
            if best_cost < cost:
                state = best_neighbor
                visited.add(tuple(state))
            else:
                # plateau or local optimum -> restart
                restarts += 1
                if max_restarts is not None and restarts >= max_restarts:
                    raise RuntimeError(f"Stopped after {restarts} restarts (no solution found).")
                break  # go pick a new unvisited start

def format_board(state):
    n = len(state)
    lines = []
    for r in range(n):
        lines.append(" ".join("Q" if state[c] == r else "-" for c in range(n)))
    return "\n".join(lines)

if __name__ == "__main__":
    n = 4
    solution, restarts = hill_climb_with_restarts(n)
    print("Found solution:", solution)
    print(format_board(solution))
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
import random
import math

def cost(state):
```

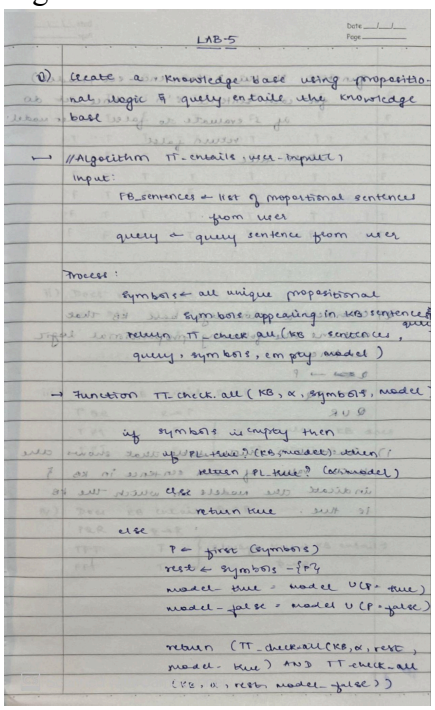
```
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

def get_neighbor(state):

    neighbor = state[:]
    i, j = random.sample(range(len(state)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

def simulated_annealing(n=8, max_iter=10000, temp=100.0, cooling=0.95):

    current = list(range(n))
    random.shuffle(current)
    current_cost = cost(current)

    temperature = temp
    cooling_rate = cooling

    best = current[:]
    best_cost = current_cost

    for _ in range(max_iter):
        if temperature <= 0 or best_cost == 0:
            break

        neighbor = get_neighbor(current)
        neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost

        if delta > 0 or random.random() < math.exp(delta / temperature):
            current, current_cost = neighbor, neighbor_cost
            if neighbor_cost < best_cost:
                best, best_cost = neighbor[:], neighbor_cost

        temperature *= cooling_rate

    return best, best_cost

def print_board(state):
```

```
    n = len(state)
    for row in range(n):
        line = " ".join("Q" if state[col] == row else "." for col in range(n))
        print(line)
    print()


n = 8
solution, cost_val = simulated_annealing(n, max_iter=20000)
print("Best position found:", solution)
print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
print("\nBoard:")
print_board(solution)
```

Program 6

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

Algorithm:


LAB-5

Date / /
Page

(i) Create a knowledge base using propositional logic & query entails the knowledge base

// Algorithm TT-entails (user-input)

input:

KB_sentences ← list of propositional sentences from user

query ← query sentence from user

Process:

symbols ← all unique propositional symbols appearing in KB sentences & query

return TT_check_all (KB, sentences, query, symbols, empty model)

function TT_check_all (KB, α, symbols, model)

if symbols is empty then

if PL_true? (KB, model) then return PL_true? (α, model)

else return true

else

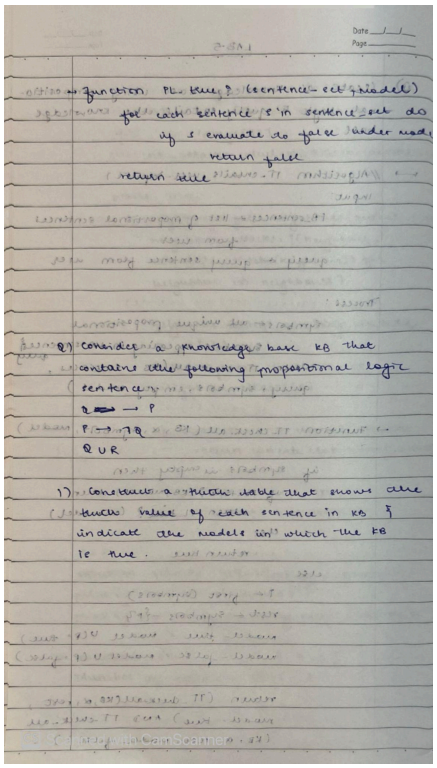
P ← first (symbols)

rest ← symbols − P

model_true = model ∪ (P = true)

model_false = model ∪ (P = false)

return (TT_check_all (KB, α, rest, model_true) AND TT_check_all (KB, α, rest, model_false))


Date / /
Page

function PL_true? (sentence_set, model)

for each sentence 's' in sentence_set do

if s evaluate to false under model

return false

return true

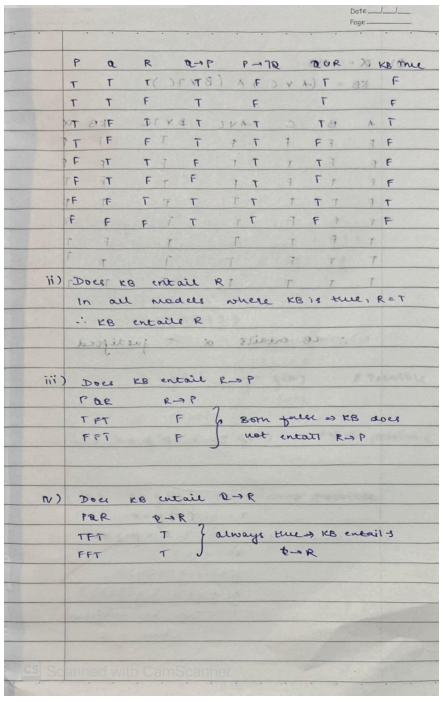
(i) Consider a knowledge base KB that contains the following propositional logic sentences

P ⟹ → P

P → ¬Q

Q ∪ R

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



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

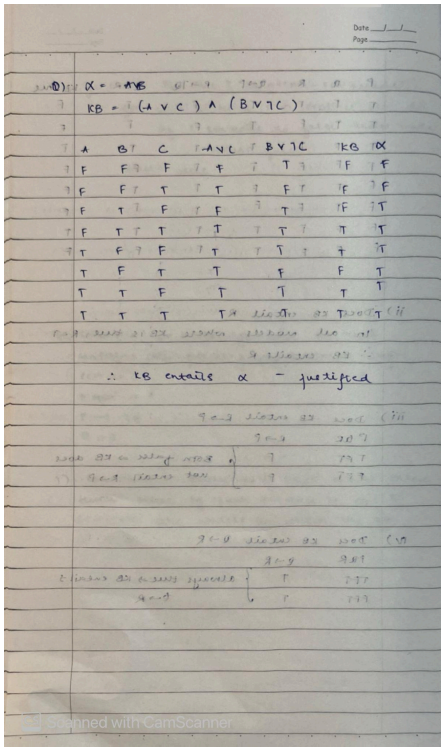
ii) Does KB entail R
In all models where KB is true, R=T
∴ KB entails R

iii) Does KB entail R→P

P	Q	R	R→P	
T	F	T	F	Some places ⇒ KB does not entail R→P
F	F	T	F	

iv) Does KB entail Q→R

P	Q	R	Q→R	
T	F	T	T	always true ⇒ KB entails Q→R
F	F	T	T	



i) α = A∨B
KB = (A∨C) ∧ (B∨¬C)

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

∴ KB entails α - justified

Code:

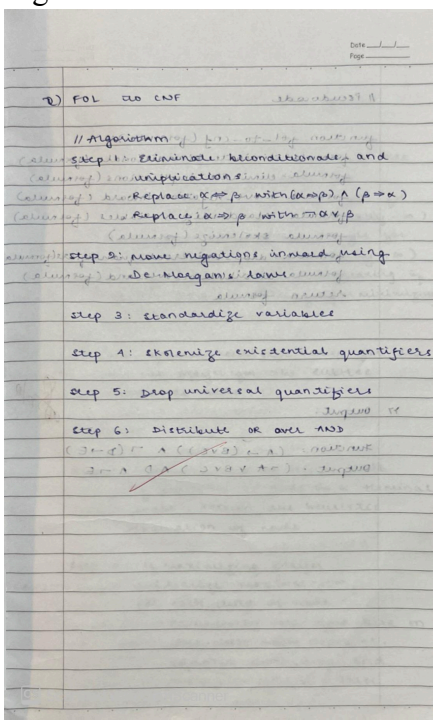
```
import itertools
def evaluate_formula(formula, truth_assignment):
    eval_formula = formula
    for symbol, value in truth_assignment.items():
        eval_formula = eval_formula.replace(symbol, str(value))
    return eval(eval_formula)
```

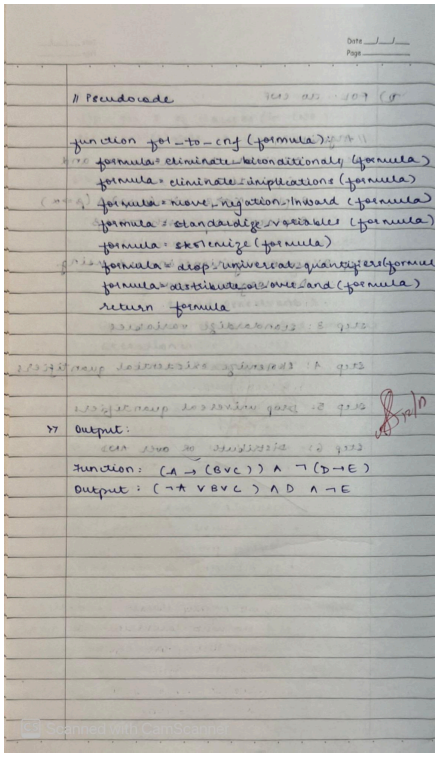
```
def generate_truth_table(variables):
    return list(itertools.product([False, True], repeat=len(variables)))

def is_entailed(KB_formula, alpha_formula, variables):
    truth_combinations = generate_truth_table(variables)
    print(f"{' '.join(variables)} | KB Result | Alpha Result")
    print("-" * (len(variables) * 2 + 15))
    for combination in truth_combinations:
        truth_assignment = dict(zip(variables, combination))
        KB_value = evaluate_formula(KB_formula, truth_assignment)
        alpha_value = evaluate_formula(alpha_formula, truth_assignment)
        result_str = " ".join(["T" if value else "F" for value in combination])
        print(f"{result_str} | {'T' if KB_value else 'F'}         | {'T' if alpha_value else 'F'}")
        if KB_value and not alpha_value:
            return False
    return True

KB = "(A or C) and (B or not C)"
alpha = "A or B"
variables = ['A', 'B', 'C']

if is_entailed(KB, alpha, variables):
    print("\nThe knowledge base entails alpha.")
else:
    print("\nThe knowledge base does not entail alpha.")
```

Program 7

Implement unification in first order logic

Algorithm:





Code:

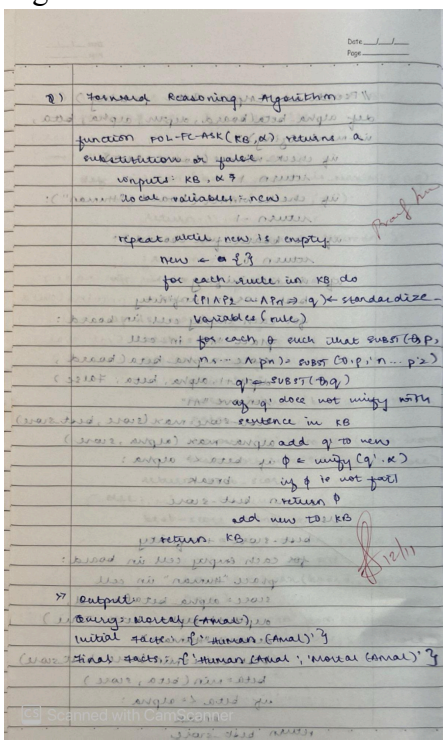
```
import re

def is_variable(x):
    return x[0].islower() and x.isalpha()

def parse(term):
    term = term.strip()
    if '(' not in term:
        return term
    name, args = term.split('(', 1)
    args = args[:-1]  # remove closing parenthesis
    return name.strip(), [parse(a.strip()) for a in args.split(',')]

def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        _, args = expr
        return any(occurs_check(var, a) for a in args)
    return False

def substitute(subs, expr):
    if isinstance(expr, str):
        if expr in subs:
            return substitute(subs, subs[expr])
        return expr
    else:
        func, args = expr
        return (func, [substitute(subs, a) for a in args])

def unify(x, y, subs=None):
    if subs is None:
        subs = {}

    x = substitute(subs, x)
    y = substitute(subs, y)

    if x == y:
        return subs

    if isinstance(x, str) and is_variable(x):
        if occurs_check(x, y):
            return None
        subs[x] = y
        return subs
```

```
    if isinstance(y, str) and is_variable(y):
        if occurs_check(y, x):
            return None
        subs[y] = x
        return subs

    if isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
        for a, b in zip(x[1], y[1]):
            subs = unify(a, b, subs)
            if subs is None:
                return None
        return subs

    return None


def term_to_str(t):
    if isinstance(t, str):
        return t
    func, args = t
    return f"{func}({', '.join(term_to_str(a) for a in args)})"

def pretty_print(subs):
    return ', '.join(f"{v} : {term_to_str(t)}" for v, t in subs.items())


pairs = [
    ("P(f(x),g(y),y)", "P(f(g(z)),g(f(a)),f(a))"),
    ("Q(x,f(x))", "Q(f(y),y)"),
    ("H(x,g(x))", "H(g(y),g(g(z)))")
]

for s1, s2 in pairs:
    print(f"\nUnifying: {s1}  and  {s2}")
    result = unify(parse(s1), parse(s2))
    if result:
        print("=> Substitution:", pretty_print(result))
    else:
        print("=> Not unifiable.")
```

Program 8

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

Algorithm:



Code:

```
import re

def match_pattern(pattern, fact):
    """
    Checks if a fact matches a rule pattern using regex-style variable substitution.
    Variables are lowercase words like p, q, x, r etc.
    Returns a dict of substitutions or None if not matched.
    """
    # Extract predicate name and arguments
    pattern_pred, pattern_args = re.match(r'(\w+)
', pattern).groups()
    fact_pred, fact_args = re.match(r'(\w+)
', fact).groups()

    if pattern_pred != fact_pred:
        return None  # predicate mismatch

    pattern_args = [a.strip() for a in pattern_args.split(",")]
    fact_args = [a.strip() for a in fact_args.split(",")]
```

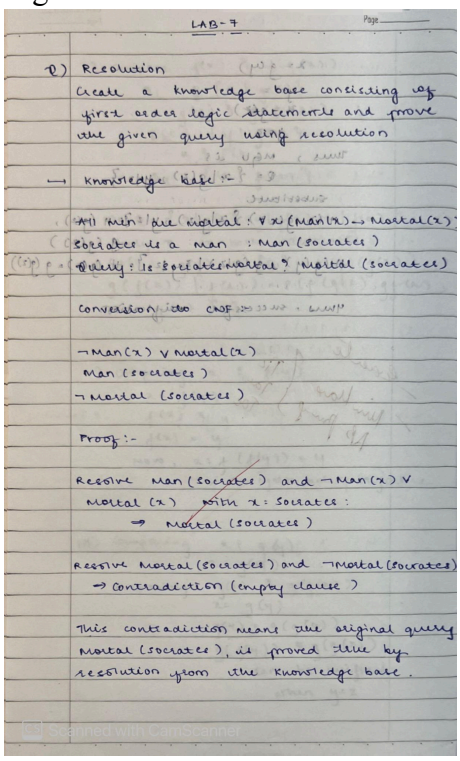
```
    if len(pattern_args) != len(fact_args):
        return None

    subst = {}
    for p_arg, f_arg in zip(pattern_args, fact_args):
        if re.fullmatch(r'[a-z]\w*', p_arg):  # variable
            subst[p_arg] = f_arg
        elif p_arg != f_arg:  # constants mismatch
            return None
    return subst


def apply_substitution(expr, subst):
    """Replaces all variable names in expr using the given substitution dict."""
    for var, val in subst.items():
        expr = re.sub(rf'\b{var}\b', val, expr)
    return expr


# ---------- Knowledge Base ----------

rules = [
    (["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"], "Criminal(p)"),
    (["Missile(x)"], "Weapon(x)"),
    (["Enemy(x, America)"], "Hostile(x)"),
    (["Missile(x)", "Owns(A, x)"], "Sells(Robert, x, A)")
]

facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Owns(A, T1)",
    "Missile(T1)"
}

goal = "Criminal(Robert)"

def forward_chain(rules, facts, goal):
    added = True
    while added:
        added = False
        for premises, conclusion in rules:

            possible_substs = []
            for p in premises:
                for f in facts:
                    subst = match_pattern(p, f)
```

```
                if subst:
                    possible_substs.append(subst)
                    break
            else:

                break
        else:

            combined = {}
            for s in possible_substs:
                combined.update(s)

            new_fact = apply_substitution(conclusion, combined)

            if new_fact not in facts:
                facts.add(new_fact)
                print(f"Inferred: {new_fact}")
                added = True
                if new_fact == goal:
                    return True
    return goal in facts


print("Goal achieved:", forward_chain(rules, facts, goal))
```

Program 9

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

Algorithm:



LAB-7

Page

2) Resolution

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

→ Knowledge Base :-

All men are mortal : ∀x (Man(x) → Mortal(x))

Socrates is a man : Man (Socrates)

Query : Is Socrates mortal ? Mortal (Socrates)

Conversion to CNF :-

¬Man(x) ∨ Mortal(x)

Man (Socrates)

¬Mortal (Socrates)

Proof :-

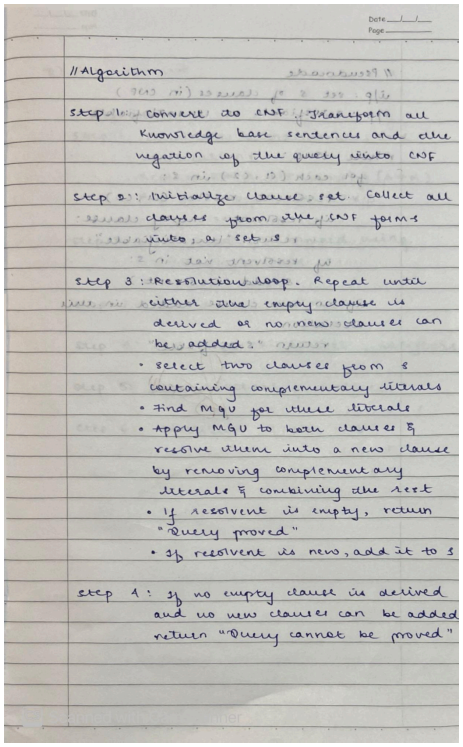
Resolve Man (Socrates) and ¬Man(x) ∨ Mortal (x) with x : Socrates :

⇒ Mortal (Socrates)

Resolve Mortal (Socrates) and ¬Mortal (Socrates)

→ Contradiction (empty clause)

This contradiction means the original query Mortal (Socrates), is proved true by resolution from the knowledge base.



Date

Page

// Algorithm

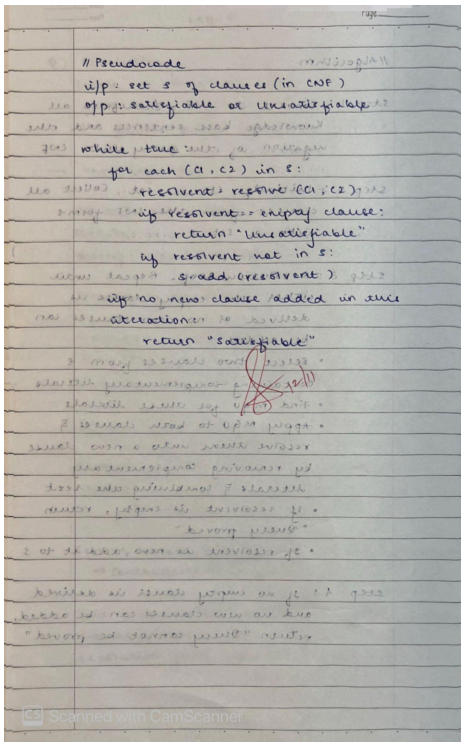
Step 1 : Convert to CNF : Transform all knowledge base sentences and the negation of the query into CNF

Step 2 : Initialize clause set. Collect all clauses from the CNF forms into a set S

Step 3 : Resolution loop. Repeat until either the empty clause is derived or no new clauses can be added :

- Select two clauses from S containing complementary literals
- Find MGU for these literals
- Apply MGU to both clauses & resolve them into a new clause by removing complementary literals & combining the rest
- If resolvent is empty, return "Query proved"
- If resolvent is new, add it to S

Step 4 : If no empty clause is derived and no new clauses can be added return "Query cannot be proved"



Code:

```
from copy import deepcopy

def print_step(title, content):
    print(f"\n{'='*45}\n{title}\n{'='*45}")
    if isinstance(content, list):
        for i, c in enumerate(content, 1):
            print(f"{i}. {c}")
    else:
        print(content)

KB = [
    ["¬Food(x)", "Likes(John,x)"],
    ["Food(Apple)"],
    ["Food(Vegetable)"],
    ["¬Eats(x,y)", "Killed(x)", "Food(y)"],
    ["Eats(Anil,Peanuts)"],
    ["Alive(Anil)"],
    ["¬Alive(x)", "¬Killed(x)"],
    ["Killed(x)", "Alive(x)"]
]

QUERY = ["Likes(John,Peanuts)"]

def negate(literal):
    if literal.startswith("¬"):
```

```
        return literal[1:]
    return "¬" + literal

def substitute(clause, subs):
    new_clause = []
    for lit in clause:
        for var, val in subs.items():
            lit = lit.replace(var, val)
        new_clause.append(lit)
    return new_clause

def unify(lit1, lit2):
    """Small unifier for patterns like Food(x) and Food(Apple)."""
    if "(" not in lit1 or "(" not in lit2:
        return None
    pred1, args1 = lit1.split("(")
    pred2, args2 = lit2.split("(")
    args1 = args1[:-1].split(",")
    args2 = args2[:-1].split(",")
    if pred1 != pred2 or len(args1) != len(args2):
        return None
    subs = {}
    for a, b in zip(args1, args2):
        if a == b:
            continue
        if a.islower():
            subs[a] = b
        elif b.islower():
            subs[b] = a
        else:
            return None
    return subs

def resolve(ci, cj):
    """Return list of (resolvent, substitution, pair)."""
    resolvents = []
    for li in ci:
        for lj in cj:
            if li == negate(lj):
                new_clause = [x for x in ci if x != li] + [x for x in cj if x != lj]
                resolvents.append((list(set(new_clause)), {}, (li, lj)))
            else:
                # same predicate, opposite sign
                if li.startswith("¬") and not lj.startswith("¬") and li[1:].split("(")[0] == lj.split("(")[0]:
                    subs = unify(li[1:], lj)
                    if subs:
                        new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)
```

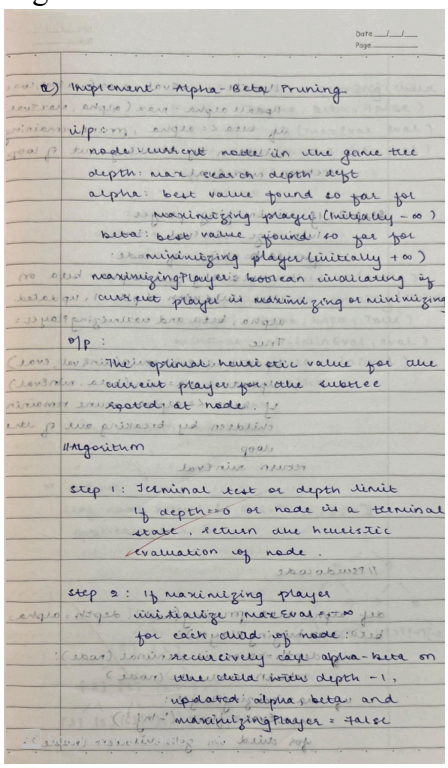
```
                resolvents.append((list(set(new_clause)), subs, (li, lj)))
            elif lj.startswith("¬") and not li.startswith("¬") and lj[1:].split("(")[0] == li.split("(")[0]:
                subs = unify(lj[1:], li)
                if subs:
                    new_clause = substitute([x for x in ci if x != li] + [x for x in cj if x != lj], subs)
                    resolvents.append((list(set(new_clause)), subs, (li, lj)))
    return resolvents

def resolution(kb, query):
    clauses = deepcopy(kb)
    negated_query = [negate(q) for q in query]
    clauses.append(negated_query)
    print_step("Initial Clauses", clauses)

    steps = []
    new = []
    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:
            for r, subs, pair in resolve(ci, cj):
                if not r:
                    steps.append({
                        "parents": (ci, cj),
                        "resolvent": r,
                        "subs": subs
                    })
                    print_tree(steps)
                    print("\n Empty clause derived — query proven.")
                    return True
                if r not in clauses and r not in new:
                    new.append(r)
                    steps.append({
                        "parents": (ci, cj),
                        "resolvent": r,
                        "subs": subs
                    })
        if all(r in clauses for r in new):
            print_step("No New Clauses", "Query cannot be proven ")
            print_tree(steps)
            return False
        clauses.extend(new)

def print_tree(steps):
    print("\n" + "="*45)
    print("Resolution Proof Trace")
    print("="*45)
```

```
    for i, s in enumerate(steps, 1):
        p1, p2 = s["parents"]
        r = s["resolvent"]
        subs = s["subs"]
        subs_text = f"  Substitution: {subs}" if subs else ""

        print(f"  Resolve {p1} and {p2}")
        if subs_text:
            print(subs_text)
        if r:
            print(f"  ⇒ {r}")
        else:
            print("  ⇒ {} (empty clause)")
        print("-"*45)

def main():
    print_step("Knowledge Base in CNF", KB)
    print_step("Negated Query", [negate(q) for q in QUERY])
    proven = resolution(KB, QUERY)
    if proven:
        print("\n Query Proven by Resolution: John likes peanuts.")
    else:
        print("\n Query cannot be proven from KB.")

if __name__ == "__main__":
    main()
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Implement Alpha-Beta Pruning

i/p:

node: current node in the game tree

depth: max / search depth left

alpha: best value found so far for maximizing player (initially $-\infty$)

beta: best value found so far for minimizing player (initially $+\infty$)

maximizingPlayer: boolean indicating if current player is maximizing or minimizing

o/p:

The optimal heuristic value for the current player for the subtree rooted at node.

Algorithm

Step 1: Terminal test or depth limit

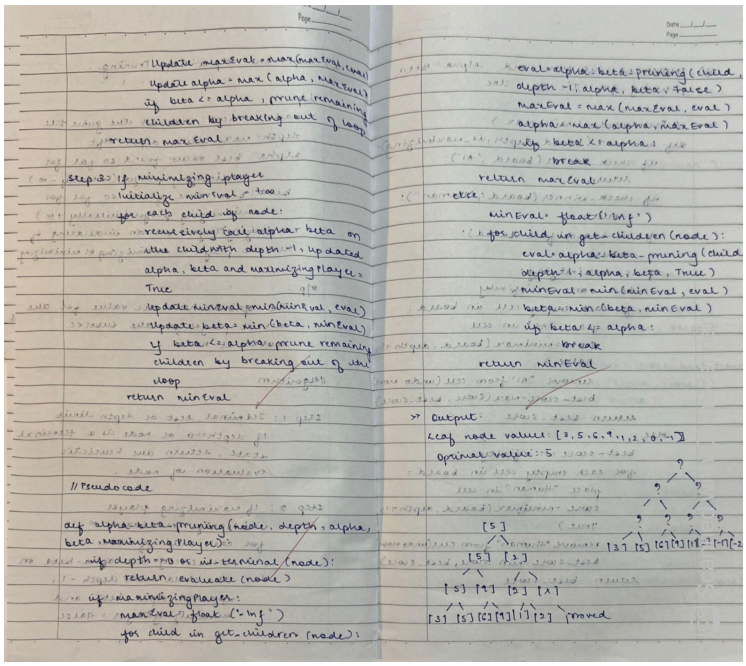
If depth==0 or node is a terminal state, return the heuristic evaluation of node.

Step 2: If maximizing player

initialize maxEval = $-\infty$

for each child of node:

recursively call alpha-beta on the child with depth - 1, updated alpha, beta, and maximizingPlayer = false



Update maxEval = max(maxEval, eval)

Update alpha = max(alpha, maxEval)

if beta <= alpha, prune remaining children by breaking out of loop

return maxEval

Step 3: If minimizing player

initialize minEval = $+\infty$

for each child of node:

recursively call alpha-beta on the child with depth - 1, updated alpha, beta and maximizingPlayer = True

Update minEval = min(minEval, eval)

Update beta = min(beta, minEval)

If beta <= alpha, prune remaining children by breaking out of loop

return minEval

// Pseudo code

def alpha_beta_pruning(node, depth, alpha, beta, maximizingPlayer):

if depth == 0 or is_terminal(node):

return evaluate(node)

if maximizingPlayer:

maxEval = float('-inf')

for child in get_children(node):

eval = alpha_beta_pruning(child, depth - 1, alpha, beta, False)

maxEval = max(maxEval, eval)

alpha = max(alpha, maxEval)

if beta <= alpha:

break

return maxEval

else:

minEval = float('inf')

for child in get_children(node):

eval = alpha_beta_pruning(child, depth - 1, alpha, beta, True)

minEval = min(minEval, eval)

beta = min(beta, minEval)

if beta <= alpha:

break

return minEval

>> Output:

Leaf node values: [3, 5, 6, 9, 1, 2, 0, -1]

Optimal value: 5

Code:

```
def unify(a, b):
    """Very simple unification for small terms like ('line', [X,O,X])"""
    if a == b:
        return {}
    if isinstance(a, str) and a.islower():  # variable
        return {a: b}
    if isinstance(b, str) and b.islower():
        return {b: a}
    if isinstance(a, tuple) and isinstance(b, tuple):
        if a[0] != b[0] or len(a[1]) != len(b[1]):
            return None
        subs = {}
        for x, y in zip(a[1], b[1]):
            s = unify(x, y)
            if s is None:
                return None
            subs.update(s)
        return subs
    return None


# Winning triples (rows, cols, diagonals)
WIN_TRIPLES = [(0,1,2),(3,4,5),(6,7,8),(0,3,6),(1,4,7),(2,5,8),(0,4,8),(2,4,6)]

def winner(board):
    pattern = ('line', ['X','X','X'])
    for i,j,k in WIN_TRIPLES:
        term = ('line', [board[i], board[j], board[k]])
        if unify(term, pattern):
            return 'X'
        if unify(term, ('line',['O','O','O'])):
            return 'O'
    return None

def is_full(board): return all(c != '_' for c in board)

def evaluate(board):
    w = winner(board)
    if w == 'X': return 1
    if w == 'O': return -1
    if is_full(board): return 0
    return None

def alpha_beta(board, player, alpha=-float('inf'), beta=float('inf')):
    val = evaluate(board)
    if val is not None:
```

```
        return val, None

    moves = [i for i,c in enumerate(board) if c == '_']
    best_move = None
    if player == 'X':
        max_eval = -float('inf')
        for m in moves:
            new_board = board[:]
            new_board[m] = 'X'
            eval_, _ = alpha_beta(new_board, 'O', alpha, beta)
            if eval_ > max_eval:
                max_eval, best_move = eval_, m
            alpha = max(alpha, eval_)
            if beta <= alpha: break
        return max_eval, best_move
    else:
        min_eval = float('inf')
        for m in moves:
            new_board = board[:]
            new_board[m] = 'O'
            eval_, _ = alpha_beta(new_board, 'X', alpha, beta)
            if eval_ < min_eval:
                min_eval, best_move = eval_, m
            beta = min(beta, eval_)
            if beta <= alpha: break
        return min_eval, best_move

def print_board(b):
    for i in range(0,9,3):
        print(' '.join(b[i:i+3]))
    print()

# --- Example usage ---
board = ['_']*9
score, move = alpha_beta(board, 'X')
print("Best first move for X:", move)
board[move] = 'X'
print_board(board)
```