**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

**“JnanaSangama”, Belgaum -590014, Karnataka.**

****

**LAB REPORT**

**on**

**Artificial Intelligence (23CS5PCAIN)**

***Submitted by***

**Sarim Ali (1BM23CS304)**

***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**

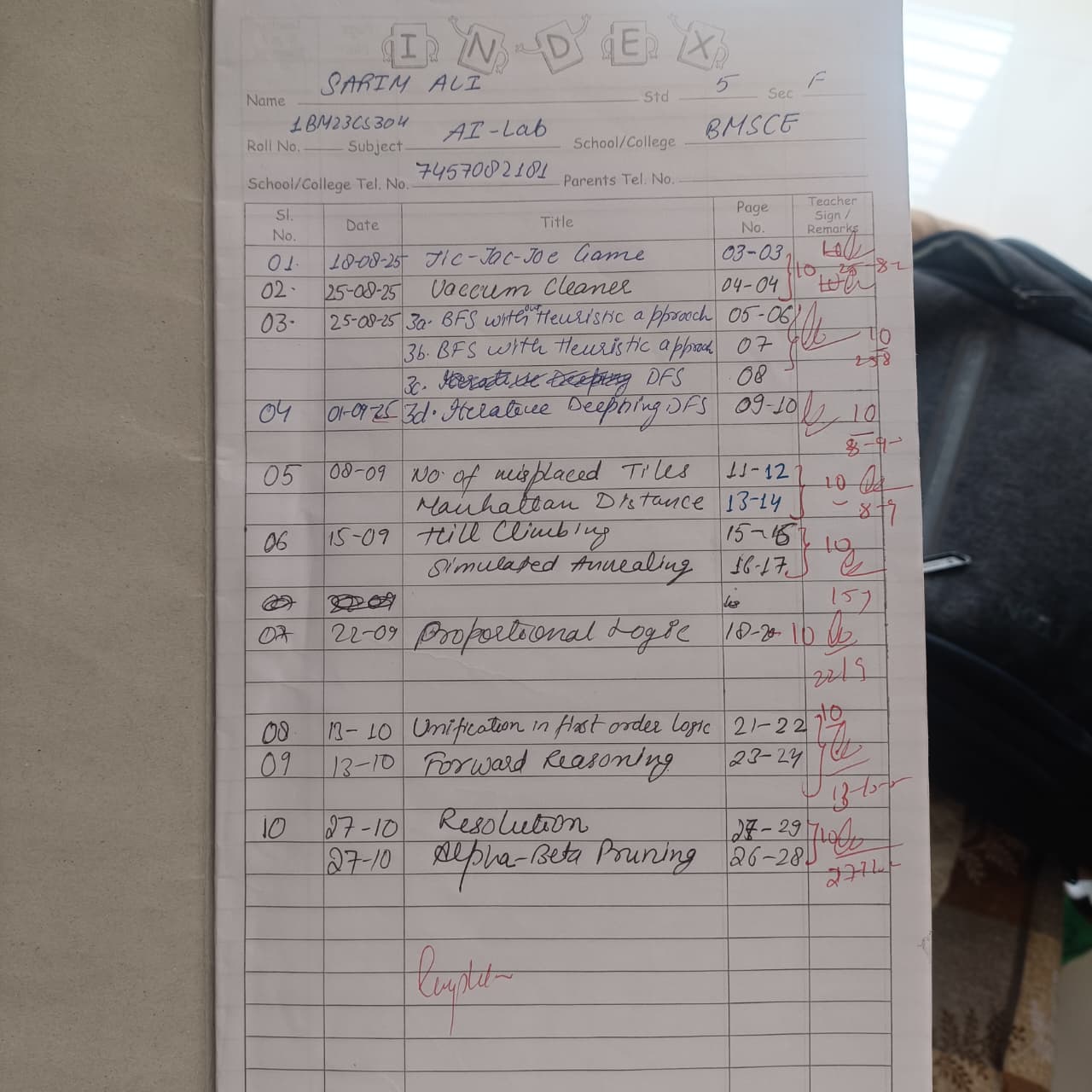
This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Sarim Ali(1BM23CS304),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of 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.

|  |  |
| --- | --- |
| Dr. Seema Patil  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 18-08-2025 | Implement Tic –Tac –Toe Game  Implement vacuum cleaner agent | 4-9 |
| 2 | 25-08-2025 | Implement 8 puzzle problems using Depth First Search (DFS)  Implement Iterative deepening search algorithm | 10-19 |
| 3 | 25-08-2025 | Implement A\* search algorithm | 20-25 |
| 4 | 15-09-2025 | Implement Hill Climbing search algorithm to solve N-Queens problem | 26-29 |
| 5 | 15-09-2025 | Simulated Annealing to Solve 8-Queens problem | 30-32 |
| 6 | 22-09-2025 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 33-37 |
| 7 | 13-10-2025 | Implement unification in first order logic | 38-43 |
| 8 | 13-10-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 44-47 |
| 9 | 27-10-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 48-51 |
| 10 | 16-11-2025 | Implement Alpha-Beta Pruning. | 52-55 |

Github Link: <https://github.com/sarim-aliii/1BM23CS304_AI>

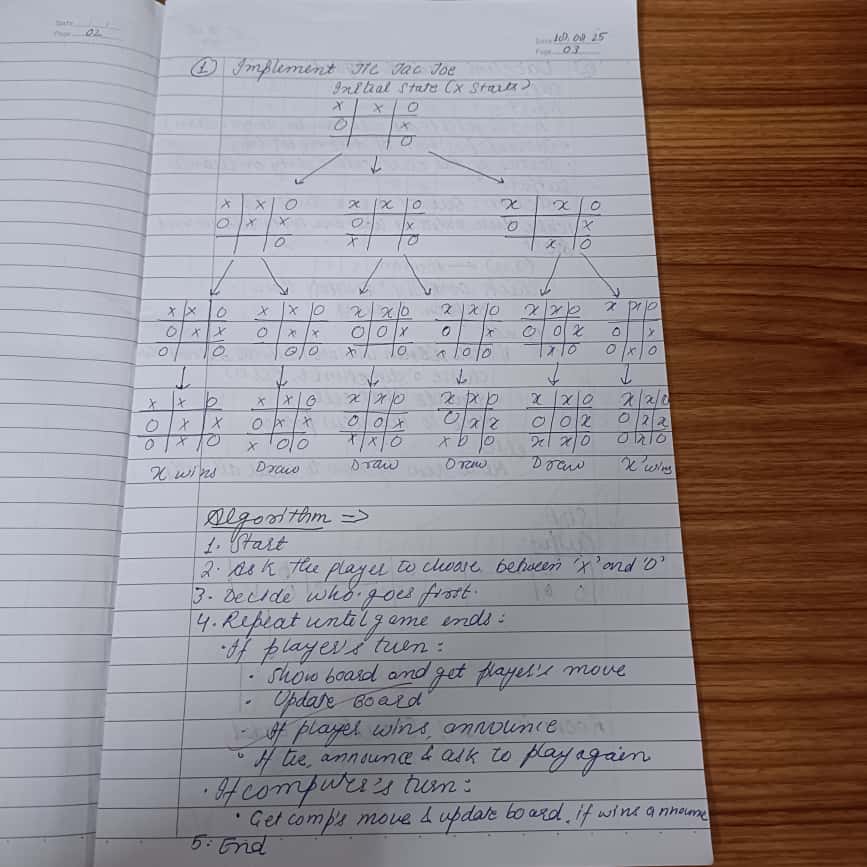


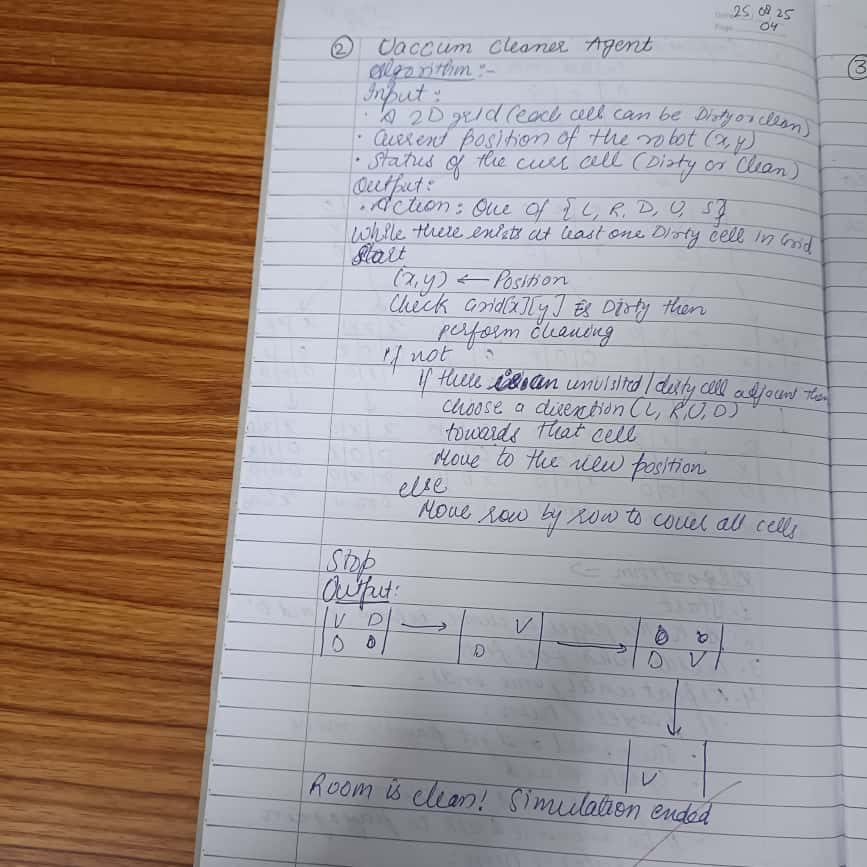
**Program 1**

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:





Code:

Tic tac toe:

def print\_board(board):

for row in board:

print(" ".join(row))

print()

def check\_winner(board, player):

for i in range(3):

if all(board[i][j] == player for j in range(3)):

return True

if all(board[j][i] == player for j in range(3)):

return True

if all(board[i][i] == player for i in range(3)):

return True

if all(board[i][2 - i] == player for i in range(3)):

return True

return False

def is\_draw(board):

return all(board[i][j] != '-' for i in range(3) for j in range(3))

def minimax(board, is\_ai\_turn):

if check\_winner(board, 'O'):

return 1

if check\_winner(board, 'X'):

return -1

if is\_draw(board):

return 0

if is\_ai\_turn:

best\_score = -float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == '-':

board[i][j] = 'O'

score = minimax(board, False)

board[i][j] = '-'

best\_score = max(score, best\_score)

return best\_score

else:

best\_score = float('inf')

for i in range(3):

for j in range(3):

if board[i][j] == '-':

board[i][j] = 'X'

score = minimax(board, True)

board[i][j] = '-'

best\_score = min(score, best\_score)

return best\_score

def manual\_game():

board = [['-' for \_ in range(3)] for \_ in range(3)]

print("Initial Board:")

print\_board(board)

while True:

while True:

try:

x\_row = int(input("Enter X row (1-3): ")) - 1

x\_col = int(input("Enter X col (1-3): ")) - 1

if board[x\_row][x\_col] == '-':

board[x\_row][x\_col] = 'X'

break

else:

print("Cell occupied!")

except:

print("Invalid input!")

print("Board after X move:")

print\_board(board)

if check\_winner(board, 'X'):

print("X wins!")

break

if is\_draw(board):

print("Draw!")

break

while True:

try:

o\_row = int(input("Enter O row (1-3): ")) - 1

o\_col = int(input("Enter O col (1-3): ")) - 1

if board[o\_row][o\_col] == '-':

board[o\_row][o\_col] = 'O'

break

else:

print("Cell occupied!")

except:

print("Invalid input!")

print("Board after O move:")

print\_board(board)

if check\_winner(board, 'O'):

print("O wins!")

break

if is\_draw(board):

print("Draw!")

break

cost = minimax(board, True)

print(f"AI evaluation cost from this position: {cost}")

manual\_game()

print("Name: Sanchit Mehta and USN : 1BM23CS299")

vaccum cleaner:

rooms = int(input("Enter Number of rooms: "))

Rooms = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

cost = 0

Roomval = {}

for i in range(rooms):

print(f"Enter Room {Rooms[i]} state (0 for clean, 1 for dirty): ")

n = int(input())

Roomval[Rooms[i]] = n

loc = input(f"Enter Location of vacuum ({Rooms[:rooms]}): ").upper()

while 1 in Roomval.values():

if Roomval[loc] == 1:

print(f"Room {loc} is dirty. Cleaning...")

Roomval[loc] = 0

cost += 1

else:

print(f"Room {loc} is already clean.")

move = input("Enter L or R to move left or right (or Q to quit): ").upper()

if move == "L":

if loc != Rooms[0]:

loc = Rooms[Rooms.index(loc) - 1]

else:

print("No room to move left.")

elif move == "R":

if loc != Rooms[rooms - 1]:

loc = Rooms[Rooms.index(loc) + 1]

else:

print("No room to move right.")

elif move == "Q":

break

else:

print("Invalid input. Please enter L, R, or Q.")

print("\nAll Rooms Cleaned." if 1 not in Roomval.values() else "Exited before cleaning all rooms.")

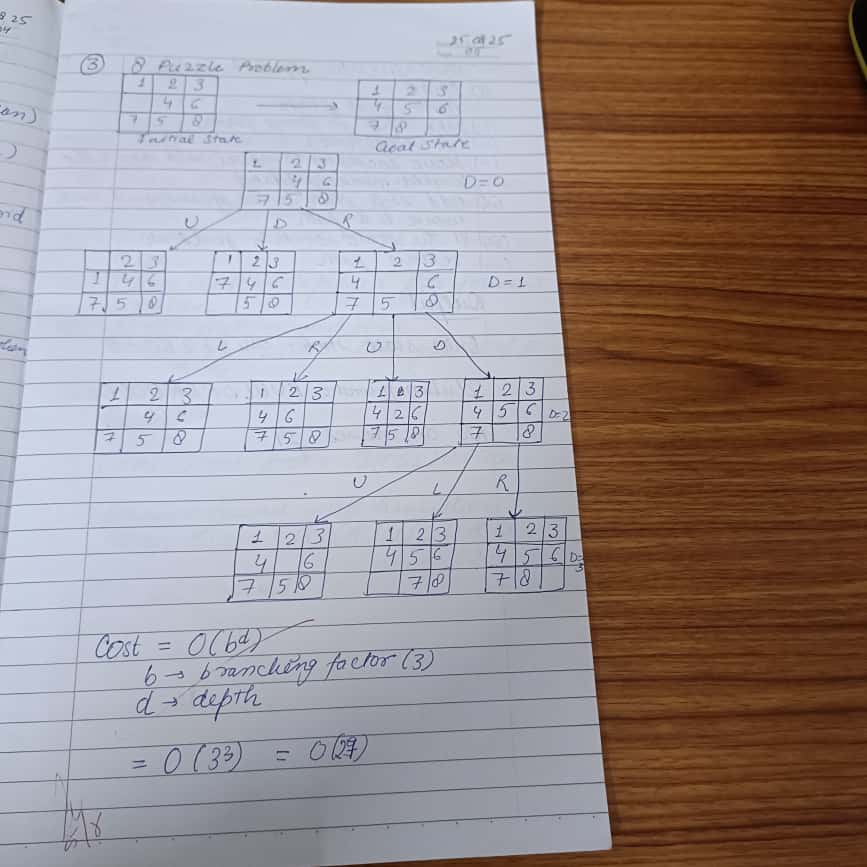
print(f"Total cost: {cost}")

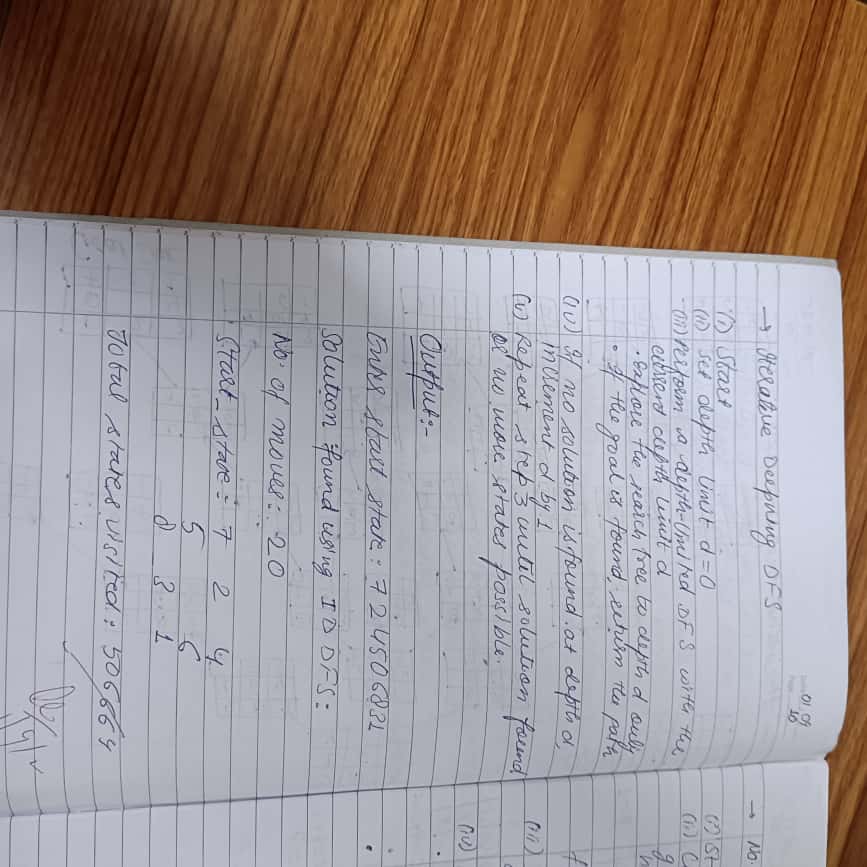
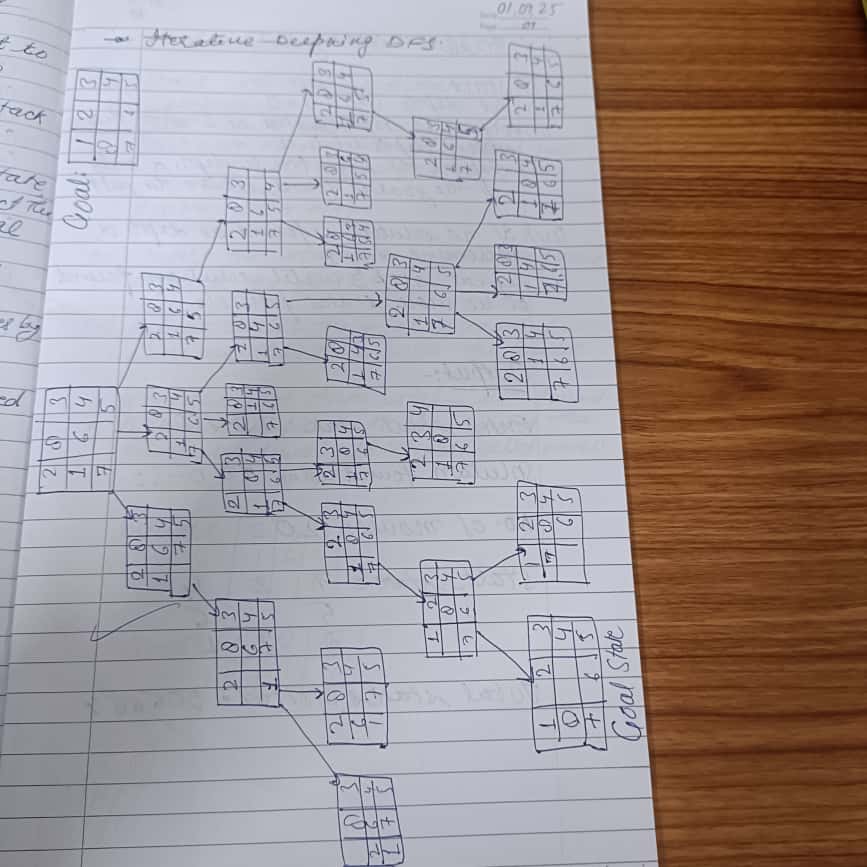
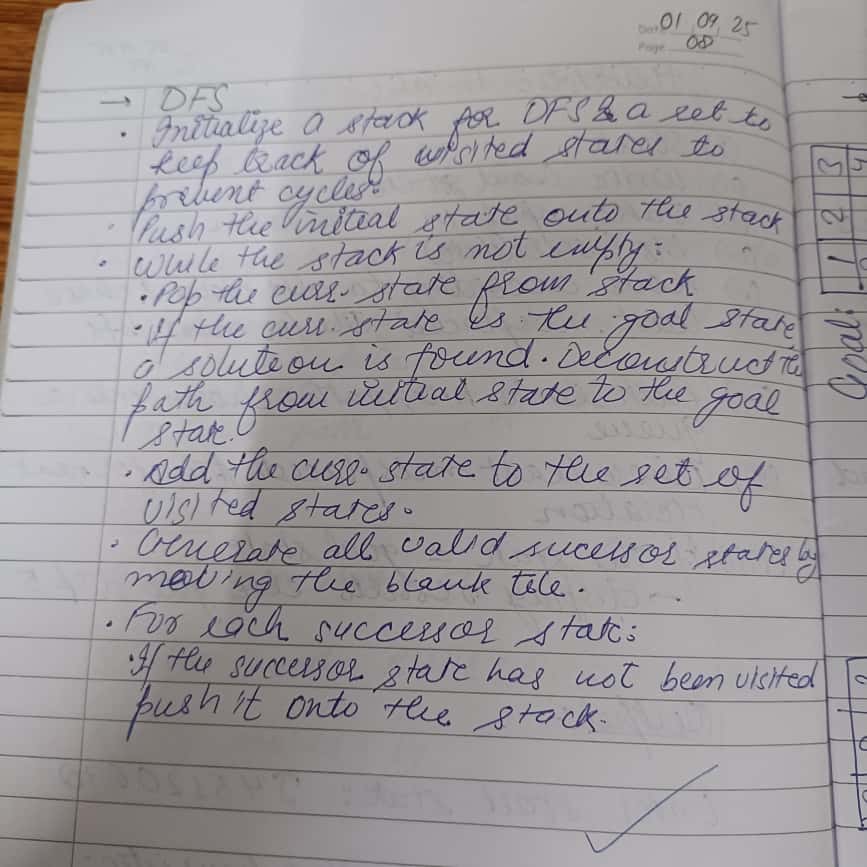
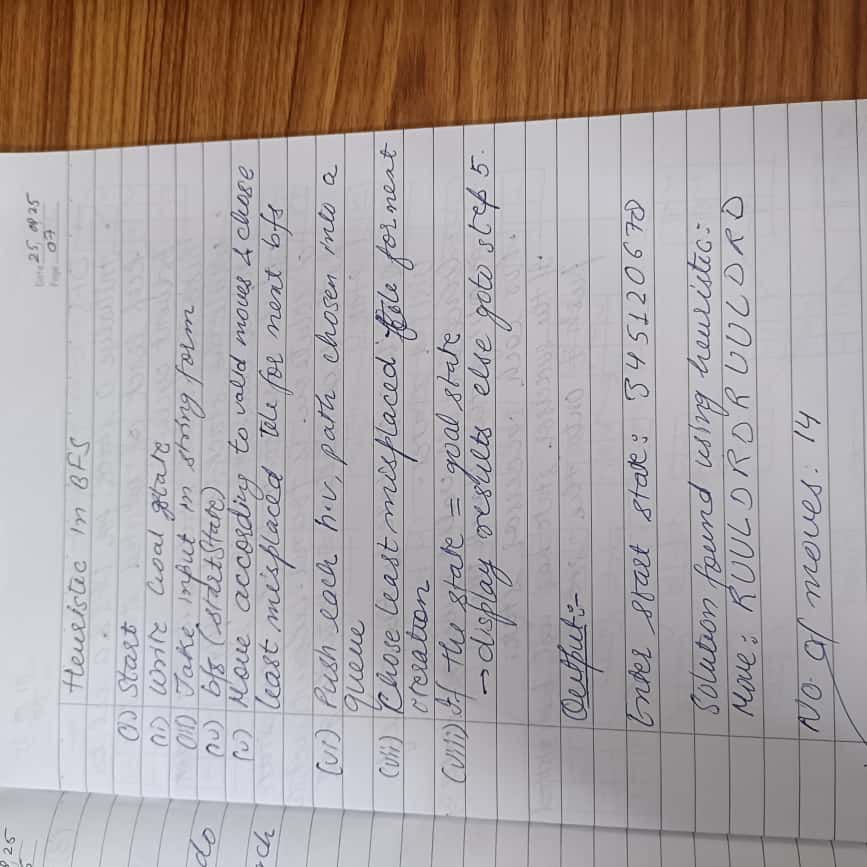
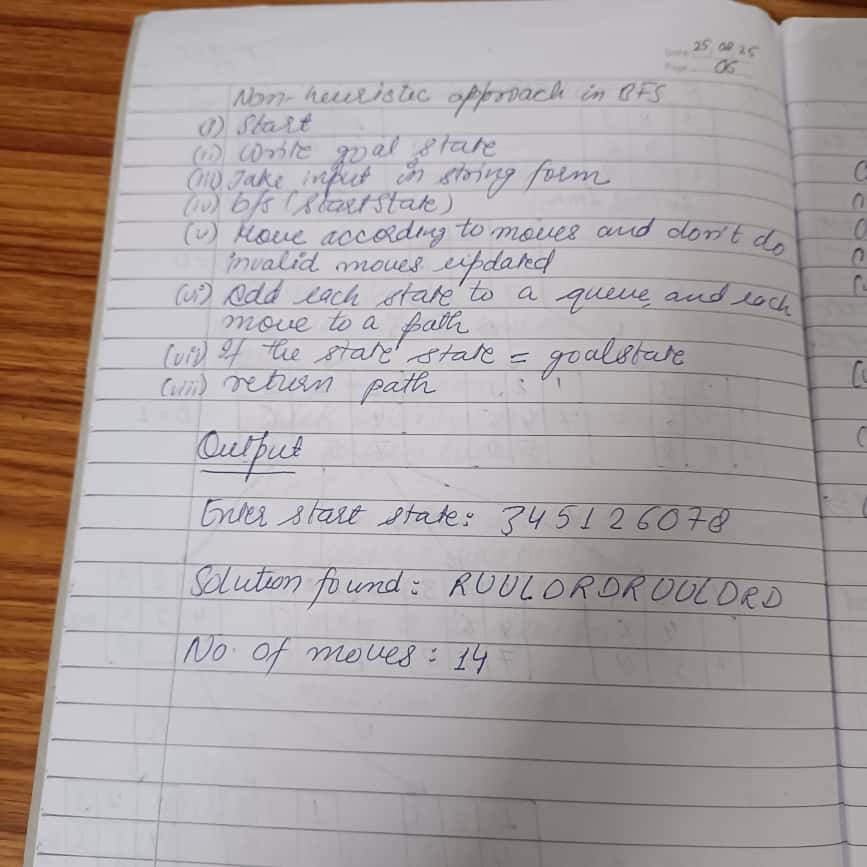
print("1BM23CS304")

print("Sarim Ali")

**Program 2**

Implement 8 puzzle problems using Depth First Search (DFS):





goal\_state = '123804765'

moves = {

'U': -3,

'D': 3,

'L': -1,

'R': 1

}

count = 0

invalid\_moves = {

0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],

3: ['L'], 5: ['R'],

6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']

}

def move\_tile(state, direction):

index = state.index('0')

if direction in invalid\_moves.get(index, []):

return None

new\_index = index + moves[direction]

if new\_index < 0 or new\_index >= 9:

return None

state\_list = list(state)

state\_list[index], state\_list[new\_index] = state\_list[new\_index], state\_list[index]

return ''.join(state\_list)

def print\_state(state):

for i in range(0, 9, 3):

print(' '.join(state[i:i+3]).replace('0', ' '))

print()

def dfs(start\_state, max\_depth=50):

visited = set()

stack = [(start\_state, [])] # Each element: (state, path)

while stack:

current\_state, path = stack.pop()

if current\_state in visited:

continue

# Print every visited state

print("Visited state:")

print\_state(current\_state)

if current\_state == goal\_state:

return path

visited.add(current\_state)

global count

count +=1

if len(path) >= max\_depth:

continue

for direction in moves:

new\_state = move\_tile(current\_state, direction)

if new\_state and new\_state not in visited:

stack.append((new\_state, path + [direction]))

return None

start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):

print("Start state:")

print\_state(start)

result = dfs(start)

if result is not None:

print("Solution found!")

print("Moves:", ' '.join(result))

print("Number of moves:", len(result))

print("Number of visited states",count)

print("1BM23CS299 Sanchit Mehta\n")

current\_state = start

for i, move in enumerate(result, 1):

current\_state = move\_tile(current\_state, move)

print(f"Move {i}: {move}")

print\_state(current\_state)

else:

print("No solution exists for the given start state or max depth reached.")

else:

print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")

Implement Iterative deepening search algorithm:

goal\_state = '123456780'

moves = {

'U': -3,

'D': 3,

'L': -1,

'R': 1

}

invalid\_moves = {

0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],

3: ['L'], 5: ['R'],

6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']

}

def move\_tile(state, direction):

index = state.index('0')

if direction in invalid\_moves.get(index, []):

return None

new\_index = index + moves[direction]

if new\_index < 0 or new\_index >= 9:

return None

state\_list = list(state)

state\_list[index], state\_list[new\_index] = state\_list[new\_index], state\_list[index]

return ''.join(state\_list)

def print\_state(state):

for i in range(0, 9, 3):

print(' '.join(state[i:i+3]).replace('0', ' '))

print()

def dls(state, depth, path, visited, visited\_count):

visited\_count[0] += 1 # Increment visited states count

if state == goal\_state:

return path

if depth == 0:

return None

visited.add(state)

for direction in moves:

new\_state = move\_tile(state, direction)

if new\_state and new\_state not in visited:

result = dls(new\_state, depth - 1, path + [direction], visited, visited\_count)

if result is not None:

return result

visited.remove(state)

return None

def iddfs(start\_state, max\_depth=50):

visited\_count = [0] # Using list to pass by reference

for depth in range(max\_depth + 1):

visited = set()

result = dls(start\_state, depth, [], visited, visited\_count)

if result is not None:

return result, visited\_count[0]

return None, visited\_count[0]

# Main

start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):

print("Start state:")

print\_state(start)

result, visited\_states = iddfs(start,15)

print(f"Total states visited: {visited\_states}")

if result is not None:

print("Solution found!")

print("Moves:", ' '.join(result))

print("Number of moves:", len(result))

print("1BM23CS304 Sarim\n")

current\_state = start

for i, move in enumerate(result, 1):

current\_state = move\_tile(current\_state, move)

print(f"Move {i}: {move}")

print\_state(current\_state)

else:

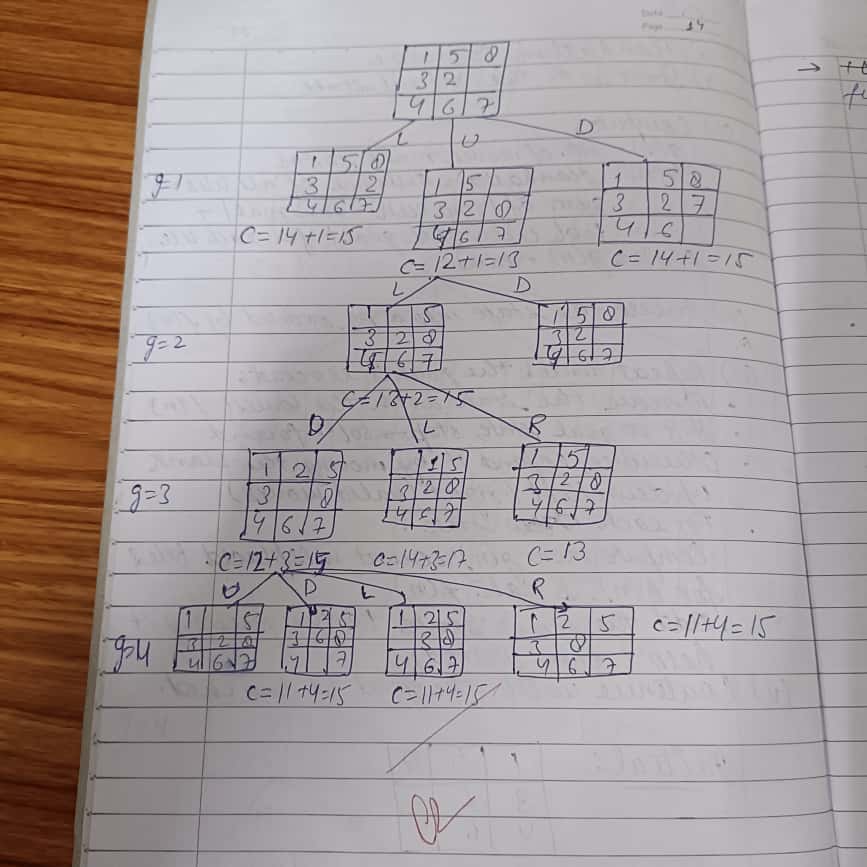
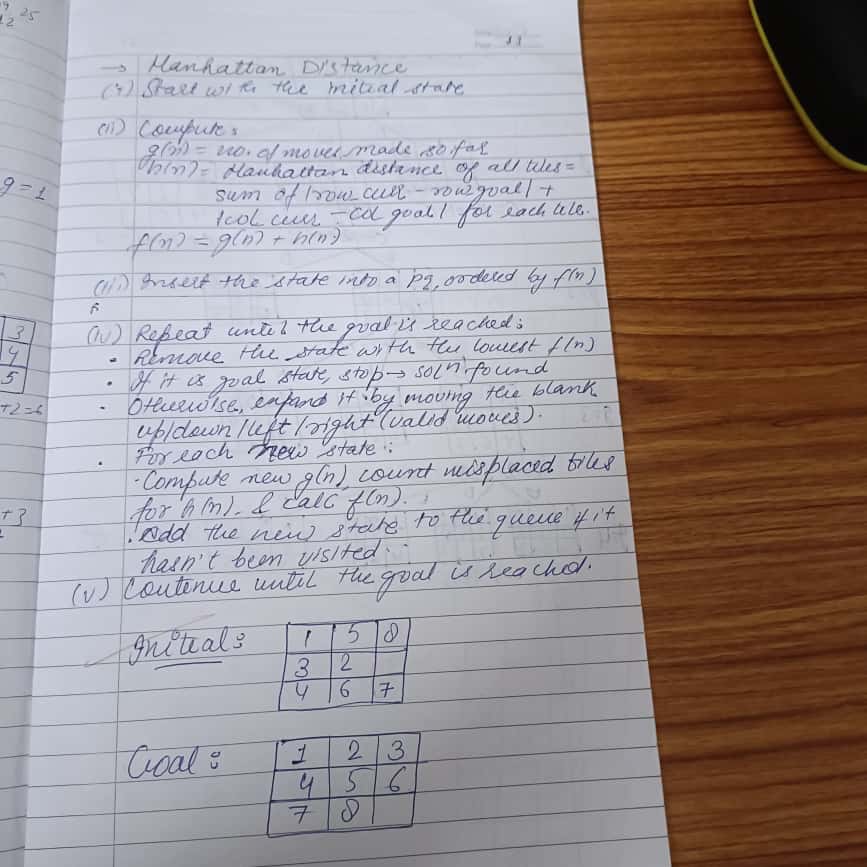
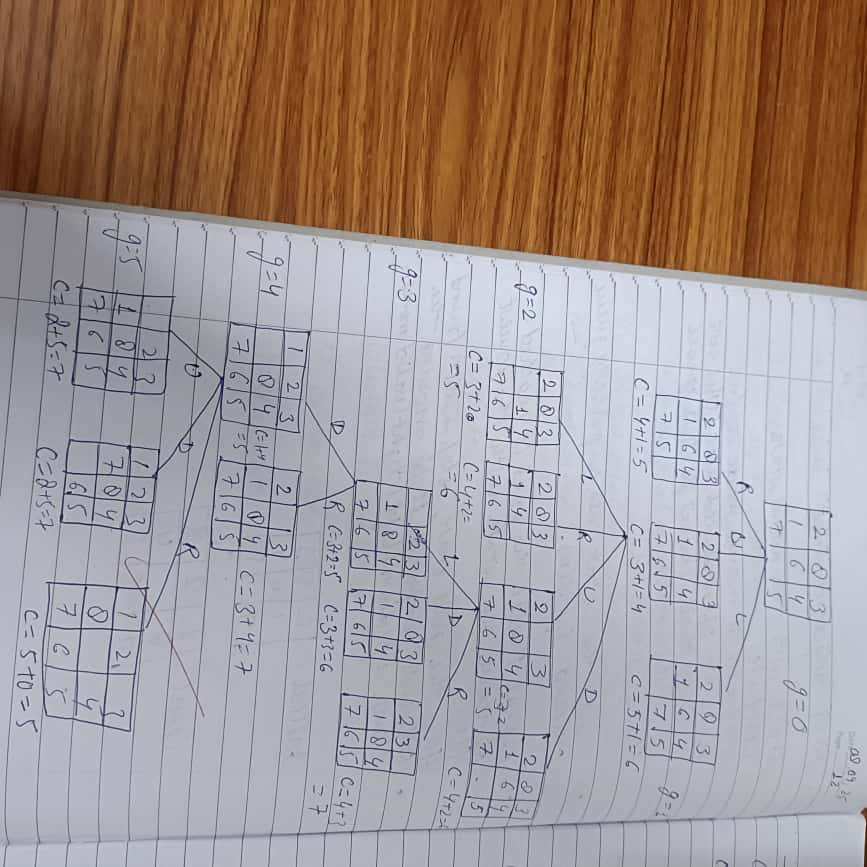
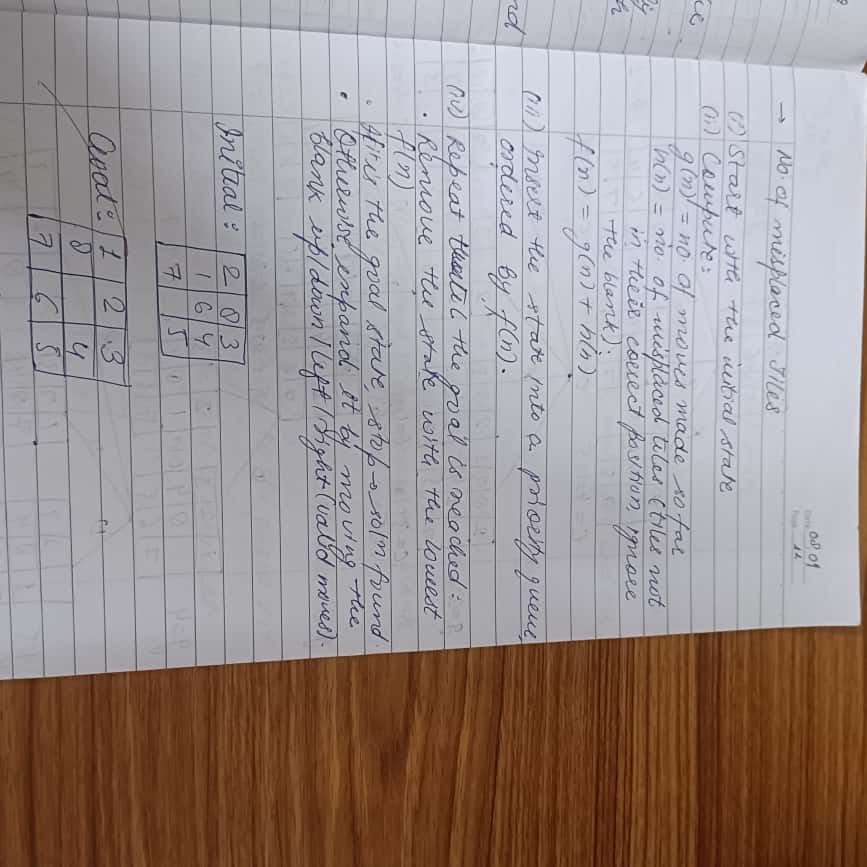
print("No solution exists for the given start state or max depth reached.")

else:

print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")

**Program 3**

Implement A\* search algorithm



import heapq

goal\_state = '123456780'

moves = {

'U': -3,

'D': 3,

'L': -1,

'R': 1

}

invalid\_moves = {

0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],

3: ['L'], 5: ['R'],

6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']

}

def move\_tile(state, direction):

index = state.index('0')

if direction in invalid\_moves.get(index, []):

return None

new\_index = index + moves[direction]

if new\_index < 0 or new\_index >= 9:

return None

state\_list = list(state)

state\_list[index], state\_list[new\_index] = state\_list[new\_index], state\_list[index]

return ''.join(state\_list)

def print\_state(state):

for i in range(0, 9, 3):

print(' '.join(state[i:i+3]).replace('0', ' '))

print()

def manhattan\_distance(state):

distance = 0

for i, val in enumerate(state):

if val == '0':

continue

goal\_pos = int(val) - 1

current\_row, current\_col = divmod(i, 3)

goal\_row, goal\_col = divmod(goal\_pos, 3)

distance += abs(current\_row - goal\_row) + abs(current\_col - goal\_col)

return distance

def a\_star(start\_state):

visited\_count = 0

open\_set = []

heapq.heappush(open\_set, (manhattan\_distance(start\_state), 0, start\_state, []))

visited = set()

while open\_set:

f, g, current\_state, path = heapq.heappop(open\_set)

visited\_count += 1

if current\_state == goal\_state:

return path, visited\_count

if current\_state in visited:

continue

visited.add(current\_state)

for direction in moves:

new\_state = move\_tile(current\_state, direction)

if new\_state and new\_state not in visited:

new\_g = g + 1

new\_f = new\_g + manhattan\_distance(new\_state)

heapq.heappush(open\_set, (new\_f, new\_g, new\_state, path + [direction]))

return None, visited\_count

# Main

start = input("Enter start state (e.g., 724506831): ")

if len(start) == 9 and set(start) == set('012345678'):

print("Start state:")

print\_state(start)

result, visited\_states = a\_star(start)

print(f"Total states visited: {visited\_states}")

if result is not None:

print("Solution found!")

print("Moves:", ' '.join(result))

print("Number of moves:", len(result))

print("1BM23CS304 Sarim\n")

current\_state = start

for i, move in enumerate(result, 1):

current\_state = move\_tile(current\_state, move)

print(f"Move {i}: {move}")

print\_state(current\_state)

else:

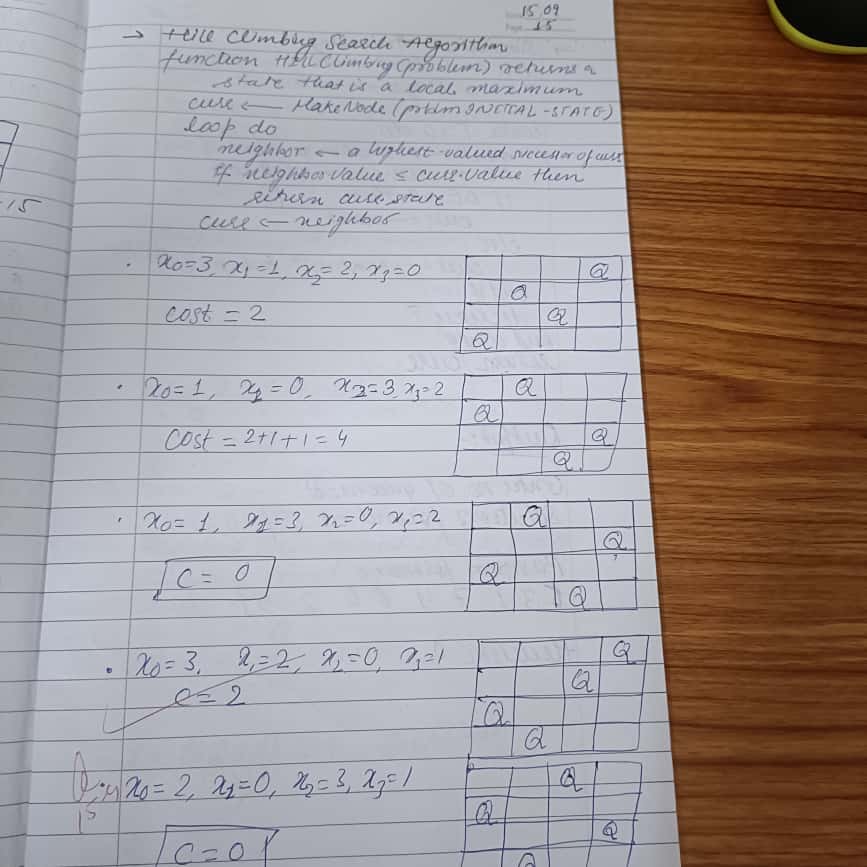
print("No solution exists for the given start state.")

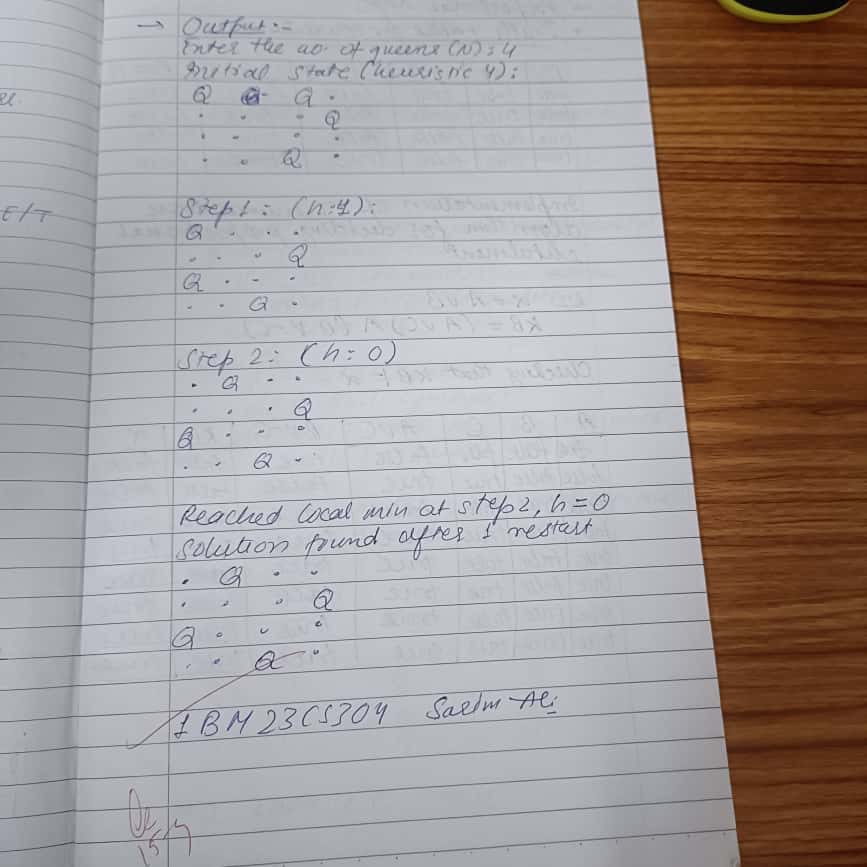
else:

print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")

**Program 4**

Implement Hill Climbing search algorithm to solve N-Queens problem





import random

import time

def print\_board(state):

"""Prints the chessboard for a given state."""

n = len(state)

for row in range(n):

line = ""

for col in range(n):

if state[col] == row:

line += "Q "

else:

line += ". "

print(line)

print()

def compute\_heuristic(state):

"""Computes the number of attacking pairs of queens."""

h = 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):

h += 1

return h

def get\_neighbors(state):

"""Generates all possible neighbors by moving one queen in its column."""

neighbors = []

n = len(state)

for col in range(n):

for row in range(n):

if row != state[col]:

neighbor = list(state)

neighbor[col] = row

neighbors.append(neighbor)

return neighbors

def hill\_climb\_verbose(initial\_state, step\_delay=0.5):

"""Hill climbing algorithm with verbose output at each step."""

current = initial\_state

current\_h = compute\_heuristic(current)

step = 0

print(f"Initial state (heuristic: {current\_h}):")

print\_board(current)

time.sleep(step\_delay)

while True:

neighbors = get\_neighbors(current)

next\_state = None

next\_h = current\_h

for neighbor in neighbors:

h = compute\_heuristic(neighbor)

if h < next\_h:

next\_state = neighbor

next\_h = h

if next\_h >= current\_h:

print(f"Reached local minimum at step {step}, heuristic: {current\_h}")

return current, current\_h

current = next\_state

current\_h = next\_h

step += 1

print(f"Step {step}: (heuristic: {current\_h})")

print\_board(current)

time.sleep(step\_delay)

def solve\_n\_queens\_verbose(n, max\_restarts=1000):

"""Solves N-Queens problem using hill climbing with restarts and verbose output."""

for attempt in range(max\_restarts):

print(f"\n=== Restart {attempt + 1} ===\n")

initial\_state = [random.randint(0, n - 1) for \_ in range(n)]

solution, h = hill\_climb\_verbose(initial\_state)

if h == 0:

print(f" Solution found after {attempt + 1} restart(s):")

print\_board(solution)

return solution

else:

print(f"No solution in this attempt (local minimum).\n")

print("Failed to find a solution after max restarts.")

return None

# --- Run the algorithm ---

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

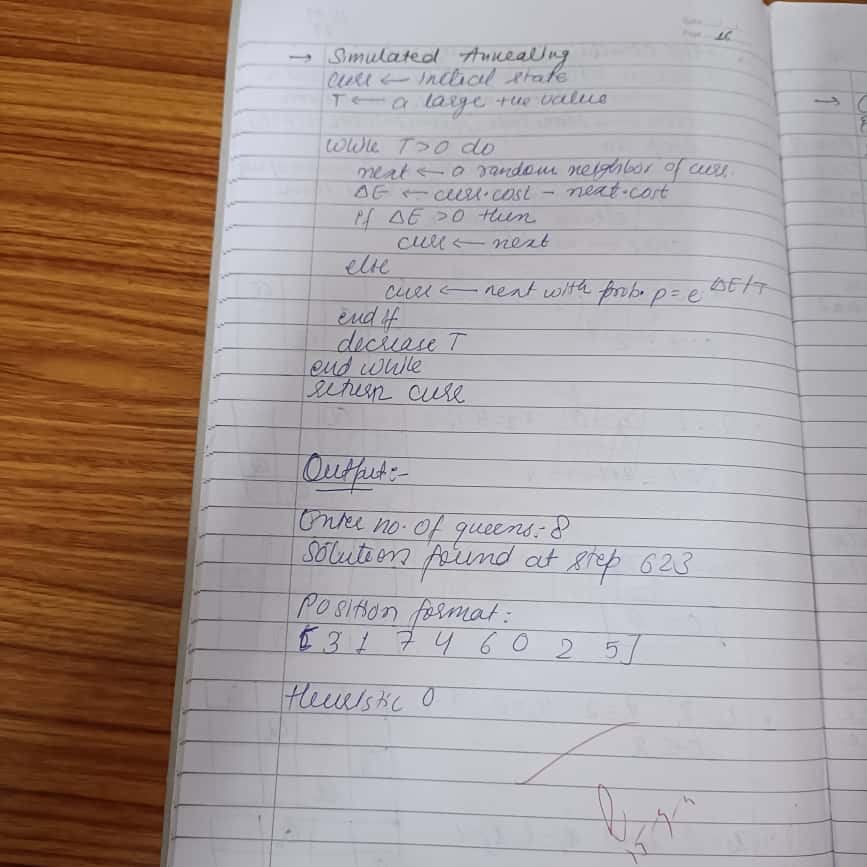
N = int(input("Enter the number of queens (N): "))

solve\_n\_queens\_verbose(N)

print("1BM23CS304 Sarim Ali")

**Program 5**

Simulated Annealing to Solve 8-Queens problem



import random

import math

def compute\_heuristic(state):

"""Number of attacking pairs."""

h = 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):

h += 1

return h

def random\_neighbor(state):

"""Returns a neighbor by randomly changing one queen's row."""

n = len(state)

neighbor = state[:]

col = random.randint(0, n - 1)

old\_row = neighbor[col]

new\_row = random.choice([r for r in range(n) if r != old\_row])

neighbor[col] = new\_row

return neighbor

def dual\_simulated\_annealing(n, max\_iter=10000, initial\_temp=100.0, cooling\_rate=0.99):

"""Simulated Annealing with dual acceptance strategy."""

current = [random.randint(0, n - 1) for \_ in range(n)]

current\_h = compute\_heuristic(current)

temperature = initial\_temp

for step in range(max\_iter):

if current\_h == 0:

print(f"✅ Solution found at step {step}")

return current

neighbor = random\_neighbor(current)

neighbor\_h = compute\_heuristic(neighbor)

delta = neighbor\_h - current\_h

if delta < 0:

current = neighbor

current\_h = neighbor\_h

else:

# Dual acceptance: standard + small chance of higher uphill move

probability = math.exp(-delta / temperature)

if random.random() < probability:

current = neighbor

current\_h = neighbor\_h

temperature \*= cooling\_rate

if temperature < 1e-5: # Restart if stuck

temperature = initial\_temp

current = [random.randint(0, n - 1) for \_ in range(n)]

current\_h = compute\_heuristic(current)

print("❌ Failed to find solution within max iterations.")

return None

# --- Run the algorithm ---

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

N = int(input("Enter number of queens (N): "))

solution = dual\_simulated\_annealing(N)

if solution:

print("Position format:")

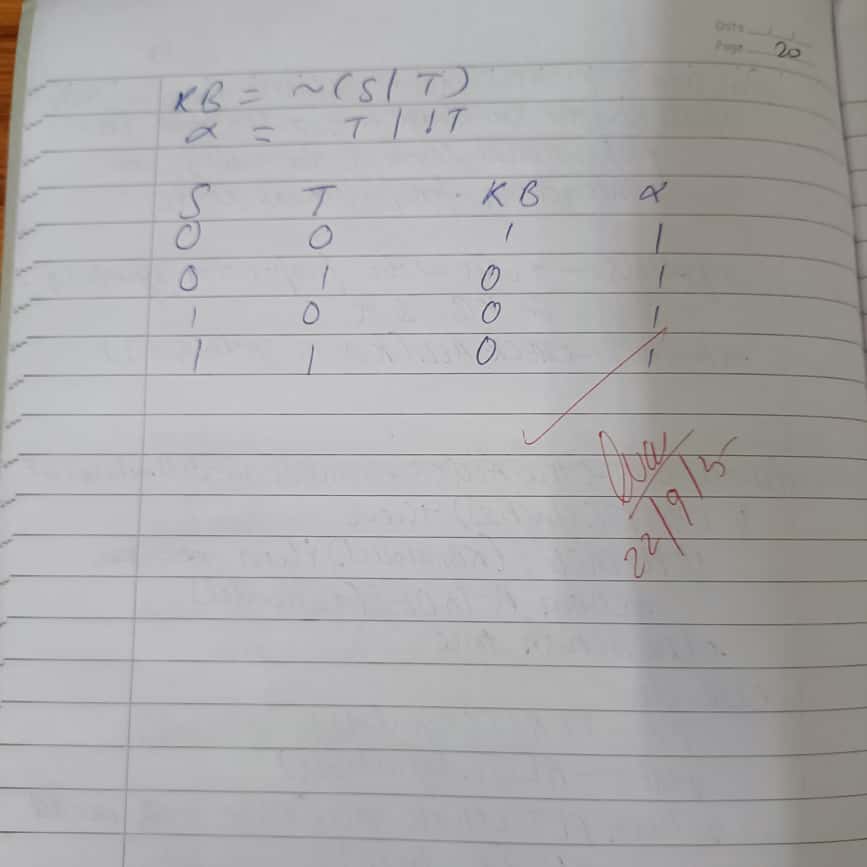
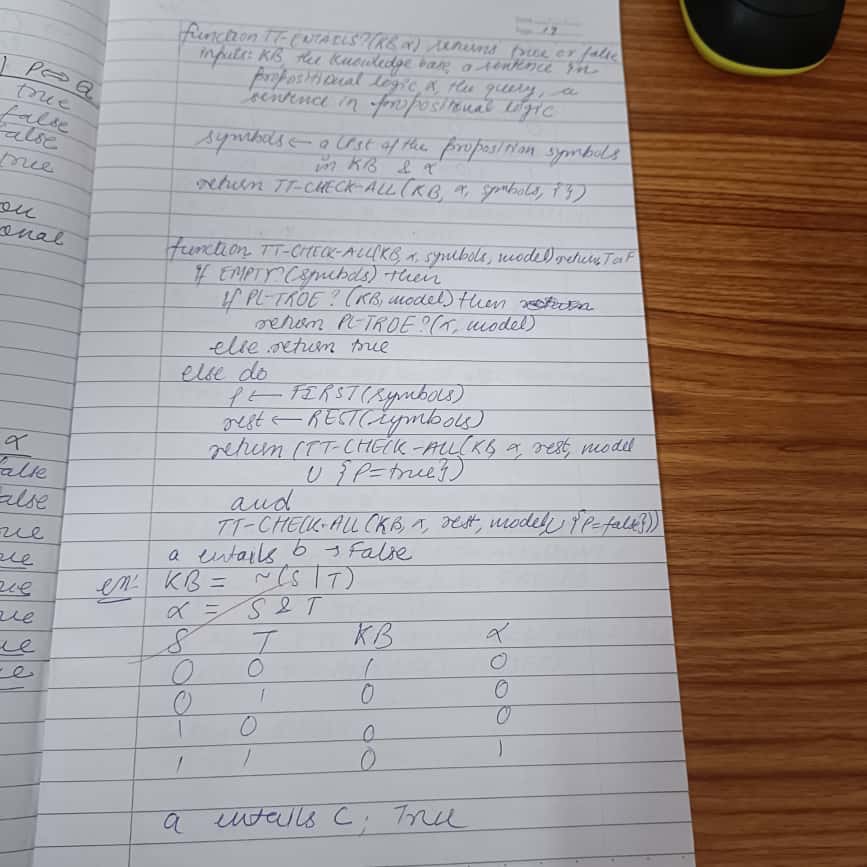
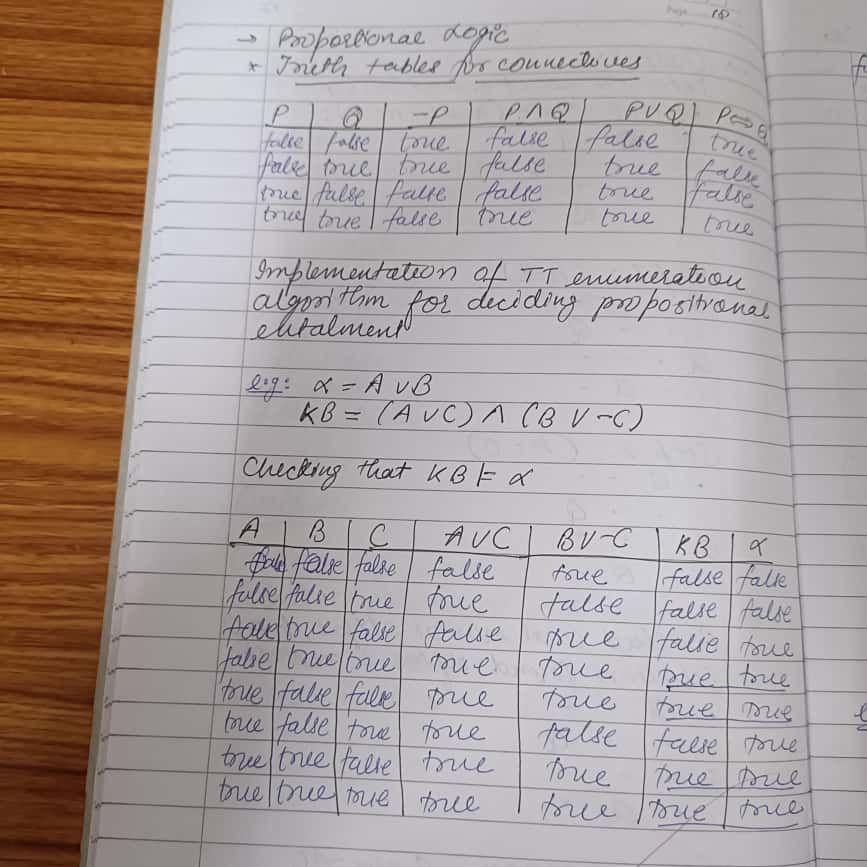
print("[", " ".join(str(x) for x in solution), "]")

print("Heuristic:", compute\_heuristic(solution))

print("1BM23CS304 Sarim")

**Program 6**

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



from typing import List, Tuple, Dict, Set, Union

Predicate = Tuple[str, Tuple[str, ...]]

class Rule:

def \_\_init\_\_(self, head: Predicate, body: List[Predicate]):

self.head = head

self.body = body

def \_\_repr\_\_(self):

body\_str = ', '.join(f"{p[0]}{p[1]}" for p in self.body)

return f"{body\_str} => {self.head[0]}{self.head[1]}"

# Knowledge base

class KnowledgeBase:

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

self.facts: Set[Predicate] = set()

self.rules: List[Rule] = []

def add\_fact(self, fact: Predicate):

self.facts.add(fact)

def add\_rule(self, rule: Rule):

self.rules.append(rule)

def forward\_chain(self, query: Predicate) -> bool:

inferred = set(self.facts)

added = True

while added:

added = False

for rule in self.rules:

if all(self.\_match\_fact(body\_pred, inferred) for body\_pred in rule.body):

if not self.\_match\_fact(rule.head, inferred):

inferred.add(rule.head)

added = True

print(f"Inferred: {rule.head}")

if self.\_match\_fact(query, inferred):

return True

return self.\_match\_fact(query, inferred)

def \_match\_fact(self, pred: Predicate, fact\_set: Set[Predicate]) -> bool:

return pred in fact\_set

# --- Example usage ---

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

kb = KnowledgeBase()

kb.add\_fact(("Parent", ("John", "Mary")))

kb.add\_fact(("Parent", ("Mary", "Sue")))

facts\_list = list(kb.facts)

for f1 in facts\_list:

for f2 in facts\_list:

if f1[0] == "Parent" and f2[0] == "Parent":

if f1[1][1] == f2[1][0]:

head = ("Grandparent", (f1[1][0], f2[1][1]))

body = [f1, f2]

kb.add\_rule(Rule(head, body))

query = ("Grandparent", ("John", "Sue"))

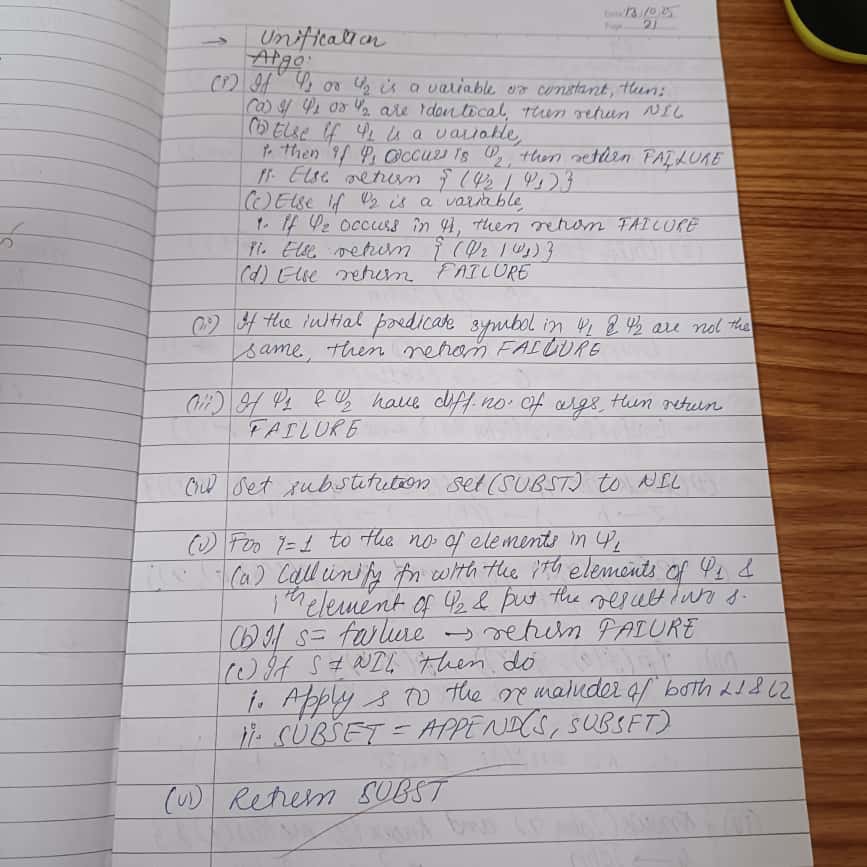
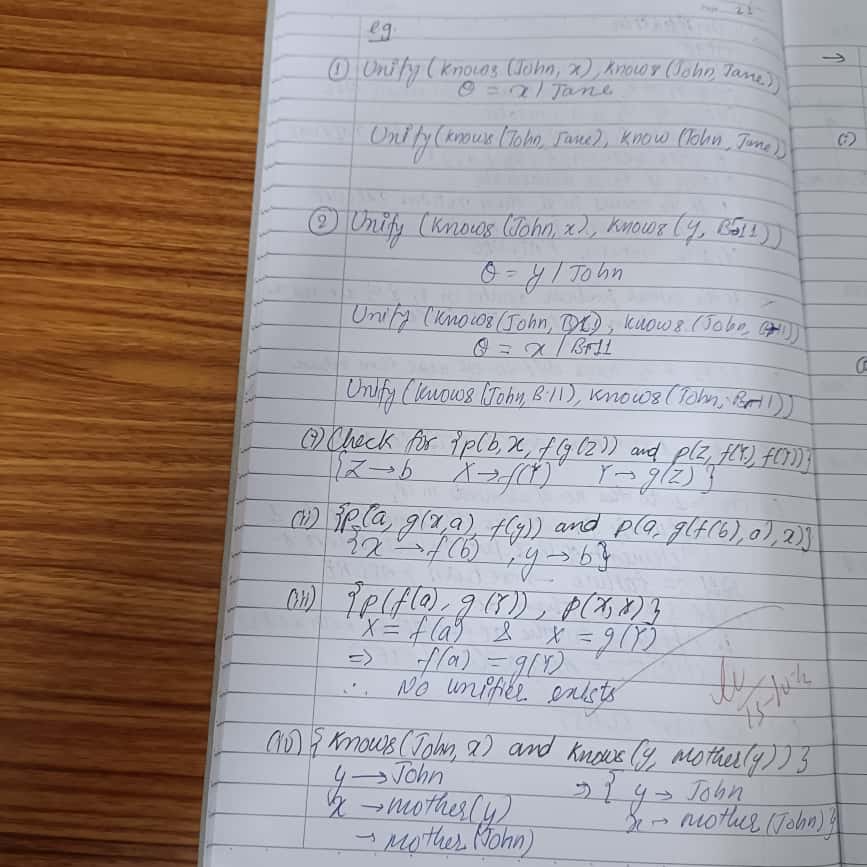
result = kb.forward\_chain(query)

print(f"Query {query} is", "True" if result else "False")

print("1BM23CS304 Sarim")

**Program 7**

Implement unification in first order logic:



import json

# --- Helper Functions for Term Manipulation ---

def is\_variable(term):

"""Checks if a term is a variable (a single capital letter string)."""

return isinstance(term, str) and len(term) == 1 and 'A' <= term[0] <= 'Z'

def occurs\_check(variable, term, sigma):

"""

Checks if 'variable' occurs anywhere in 'term' under the current substitution 'sigma'.

This prevents infinite recursion (e.g., unifying X with f(X)).

"""

term = apply\_substitution(term, sigma) # Check the substituted term

if term == variable:

return True

# If the term is a list (function/predicate), check its arguments recursively

if isinstance(term, list):

for arg in term[1:]:

if occurs\_check(variable, arg, sigma):

return True

return False

def apply\_substitution(term, sigma):

"""

Applies the current substitution 'sigma' to a 'term' recursively.

"""

if is\_variable(term):

# If the variable is bound in sigma, apply the binding

if term in sigma:

# Recursively apply the rest of the substitutions to the binding's value

# This is critical for chains like X/f(Y), Y/a -> X/f(a)

return apply\_substitution(sigma[term], sigma)

return term

if isinstance(term, list):

# Apply substitution to the arguments of the function/predicate

new\_term = [term[0]] # Keep the function/predicate symbol

for arg in term[1:]:

new\_term.append(apply\_substitution(arg, sigma))

return new\_term

# Term is a constant or an unhandled type, return as is

return term

def term\_to\_string(term):

"""

Converts the internal list representation of a term into standard logic notation string.

e.g., ['f', 'Y'] -> "f(Y)"

"""

if isinstance(term, str):

return term

if isinstance(term, list):

# Term is a function or predicate

symbol = term[0]

args = [term\_to\_string(arg) for arg in term[1:]]

return f"{symbol}({', '.join(args)})"

return str(term)

# --- Main Unification Function ---

def unify(term1, term2):

"""

Implements the Unification Algorithm to find the MGU for term1 and term2.

Returns the MGU as a dictionary or None if unification fails.

"""

# Initialize the substitution set (MGU)

sigma = {}

# Initialize the list of pairs to resolve (the difference set)

diff\_set = [[term1, term2]]

print(f"--- Unification Process Started ---")

print(f"Initial Terms:")

print(f"L1: {term\_to\_string(term1)}")

print(f"L2: {term\_to\_string(term2)}")

print("-" \* 35)

while diff\_set:

# Pop the current pair of terms to unify

t1, t2 = diff\_set.pop(0)

# 1. Apply the current MGU to the terms before comparison

t1\_prime = apply\_substitution(t1, sigma)

t2\_prime = apply\_substitution(t2, sigma)

print(f"Attempting to unify: {term\_to\_string(t1\_prime)} vs {term\_to\_string(t2\_prime)}")

# 2. Check if terms are identical

if t1\_prime == t2\_prime:

print(f" -> Identical. Current MGU: {term\_to\_string(sigma)}")

continue

# 3. Handle Variable-Term unification

if is\_variable(t1\_prime):

var, term = t1\_prime, t2\_prime

elif is\_variable(t2\_prime):

var, term = t2\_prime, t1\_prime

else:

var, term = None, None

if var:

# Check if term is a variable, and if so, don't bind V/V

if is\_variable(term):

print(f" -> Both are variables. Skipping {var} / {term}")

# Ensure they are added back if not identical (which is caught by step 2).

# If V1 != V2, we add V1/V2 or V2/V1 to sigma. Since step 2 handles V/V, this means V1 != V2 here.

if var != term:

sigma[var] = term

print(f" -> Variable binding added: {var} / {term\_to\_string(term)}. New MGU: {term\_to\_string(sigma)}")

# Occurs Check: Fail if the variable occurs in the term it's being bound to

elif occurs\_check(var, term, sigma):

print(f" -> OCCURS CHECK FAILURE: Variable {var} occurs in {term\_to\_string(term)}")

return None

# Create a new substitution {var / term}

else:

sigma[var] = term

print(f" -> Variable binding added: {var} / {term\_to\_string(term)}. New MGU: {term\_to\_string(sigma)}")

# 4. Handle Complex Term (Function/Predicate) unification

elif isinstance(t1\_prime, list) and isinstance(t2\_prime, list):

# Check functor/predicate symbol and arity (number of arguments)

if t1\_prime[0] != t2\_prime[0] or len(t1\_prime) != len(t2\_prime):

print(f" -> FUNCTOR/ARITY MISMATCH: {t1\_prime[0]} != {t2\_prime[0]} or arity mismatch.")

return None

# Add corresponding arguments to the difference set

# Start from index 1 (after the symbol)

for arg1, arg2 in zip(t1\_prime[1:], t2\_prime[1:]):

diff\_set.append([arg1, arg2])

print(f" -> Complex terms matched. Adding arguments to difference set.")

# 5. Handle Constant-Constant or other mismatches (Fail)

else:

print(f" -> TYPE/CONSTANT MISMATCH: {term\_to\_string(t1\_prime)} and {term\_to\_string(t2\_prime)} cannot be unified.")

return None

print("-" \* 35)

print("--- Unification Successful ---")

# Final cleanup to ensure all bindings are fully resolved

final\_mgu = {k: apply\_substitution(v, sigma) for k, v in sigma.items()}

return final\_mgu

# --- Define the Input Terms ---

# L1 = Q(a, g(X, a), f(Y))

literal1 = ['Q', 'a', ['g', 'X', 'a'], ['f', 'Y']]

# L2 = Q(a, g(f(b), a), X)

literal2 = ['Q', 'a', ['g', ['f', 'b'], 'a'], 'X']

# --- Run the Unification ---

mgu\_result = unify(literal1, literal2)

if mgu\_result is not None:

print("\n[ Final MGU Result ]")

# Format the final MGU for display using the new helper function

clean\_mgu = {k: term\_to\_string(v) for k, v in mgu\_result.items()}

final\_output = ', '.join([f"{k} / {v}" for k, v in clean\_mgu.items()])

print(f"Final MGU: {{ {final\_output} }}")

# --- Verification ---

print("\n[ Verification ]")

unified\_l1 = apply\_substitution(literal1, mgu\_result)

unified\_l2 = apply\_substitution(literal2, mgu\_result)

print(f"L1 after MGU: {term\_to\_string(unified\_l1)}")

print(f"L2 after MGU: {term\_to\_string(unified\_l2)}")

if unified\_l1 == unified\_l2:

print("-> SUCCESS: L1 and L2 are identical after applying the MGU.")

else:

print("-> ERROR: Unification failed verification.")

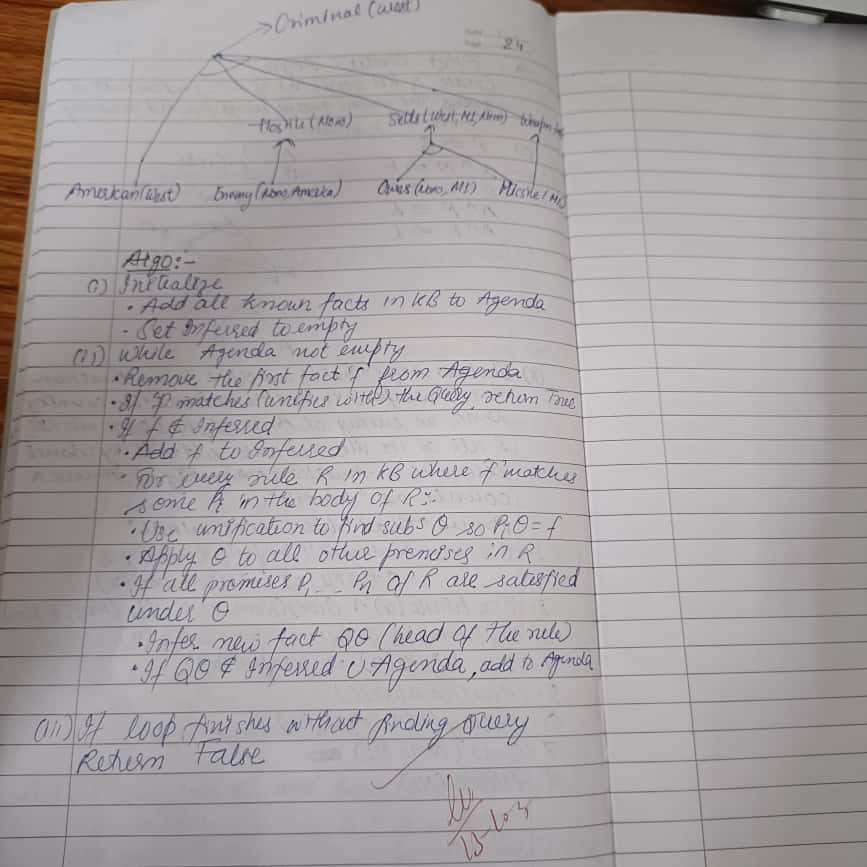
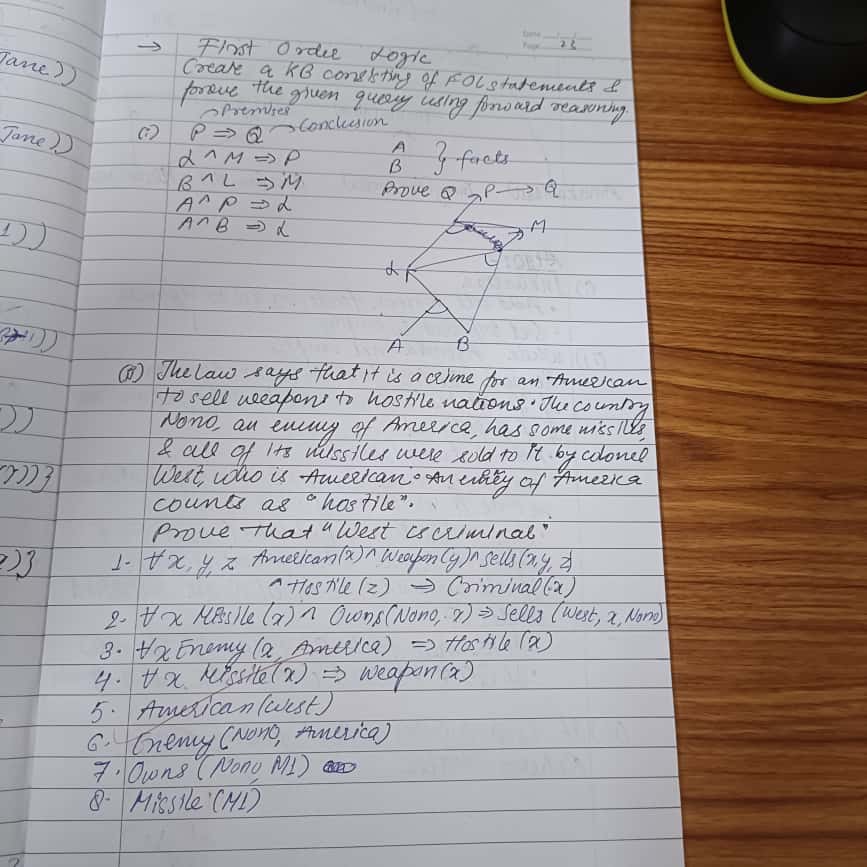
else:

print("\nUnification FAILED.")

print("Sarim 1BM23CS304")

**Program 8**

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



from typing import List, Tuple, Dict, Set, Union

Predicate = Tuple[str, Tuple[str, ...]]

class Rule:

def \_\_init\_\_(self, head: Predicate, body: List[Predicate]):

self.head = head

self.body = body

def \_\_repr\_\_(self):

body\_str = ', '.join(f"{p[0]}{p[1]}" for p in self.body)

return f"{body\_str} => {self.head[0]}{self.head[1]}"

# Knowledge base

class KnowledgeBase:

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

self.facts: Set[Predicate] = set()

self.rules: List[Rule] = []

def add\_fact(self, fact: Predicate):

self.facts.add(fact)

def add\_rule(self, rule: Rule):

self.rules.append(rule)

def forward\_chain(self, query: Predicate) -> bool:

inferred = set(self.facts)

added = True

while added:

added = False

for rule in self.rules:

if all(self.\_match\_fact(body\_pred, inferred) for body\_pred in rule.body):

if not self.\_match\_fact(rule.head, inferred):

inferred.add(rule.head)

added = True

print(f"Inferred: {rule.head}")

if self.\_match\_fact(query, inferred):

return True

return self.\_match\_fact(query, inferred)

def \_match\_fact(self, pred: Predicate, fact\_set: Set[Predicate]) -> bool:

return pred in fact\_set

# --- Example usage ---

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

kb = KnowledgeBase()

kb.add\_fact(("Parent", ("John", "Mary")))

kb.add\_fact(("Parent", ("Mary", "Sue")))

facts\_list = list(kb.facts)

for f1 in facts\_list:

for f2 in facts\_list:

if f1[0] == "Parent" and f2[0] == "Parent":

if f1[1][1] == f2[1][0]:

head = ("Grandparent", (f1[1][0], f2[1][1]))

body = [f1, f2]

kb.add\_rule(Rule(head, body))

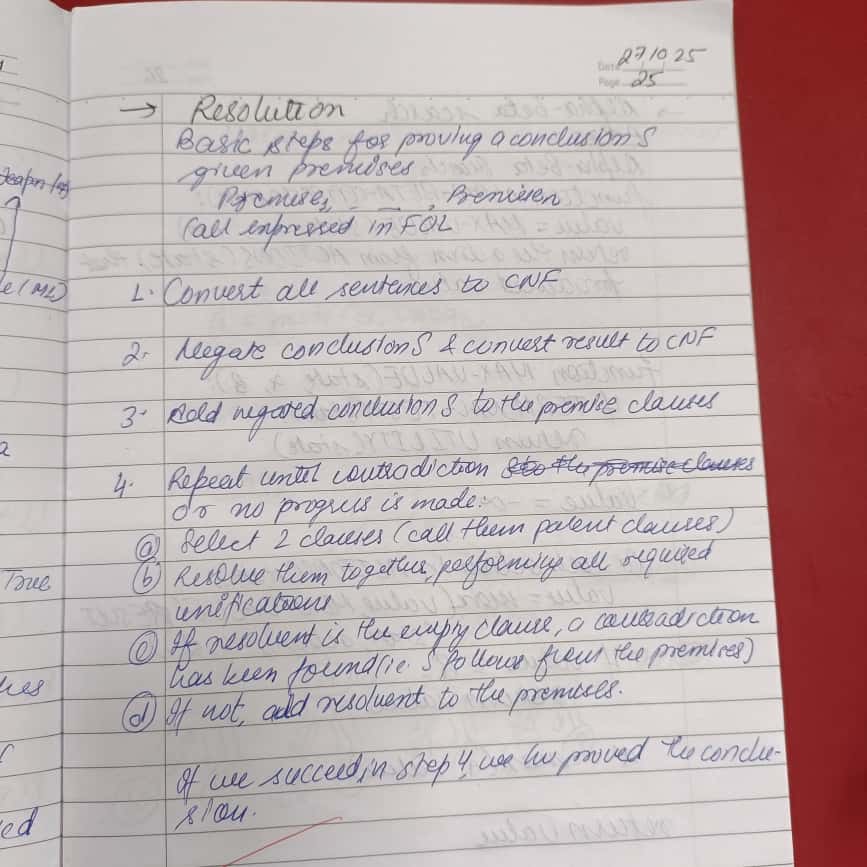
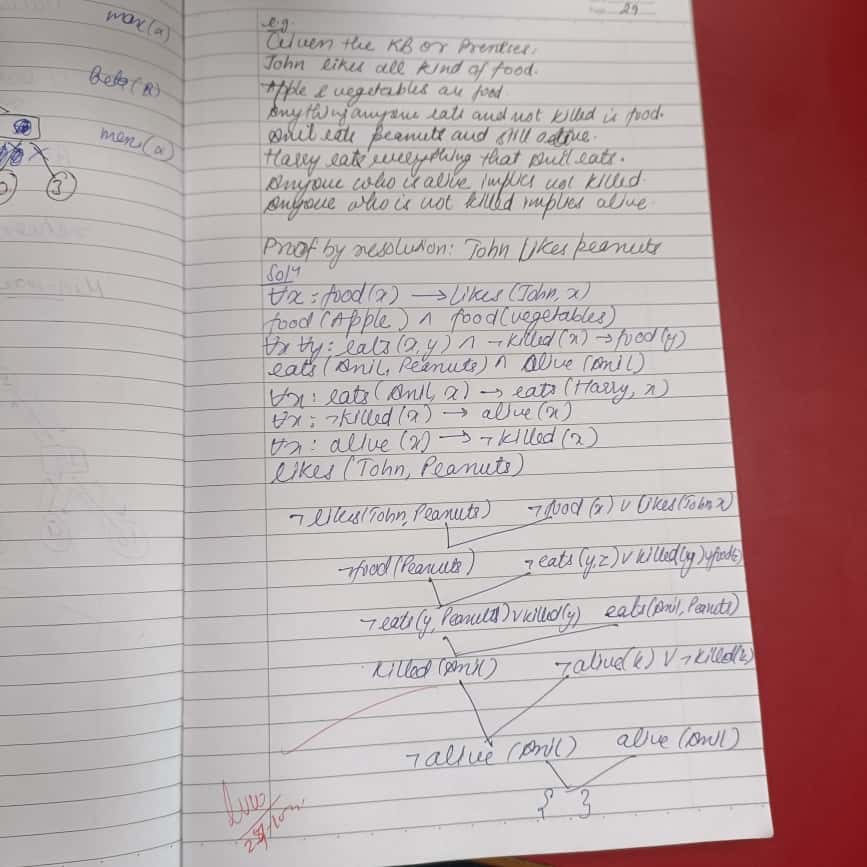
query = ("Grandparent", ("John", "Sue"))

result = kb.forward\_chain(query)

print(f"Query {query} is", "True" if result else "False")

**Program 9**:

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



from itertools import combinations

def pl\_resolution(KB, query):

# Negate the query and add to KB

clauses = KB + [negate(query)]

print("Initial Clauses:")

for c in clauses:

print(c)

print("-" \* 40)

new = set()

while True:

# Generate all possible pairs of clauses

pairs = list(combinations(clauses, 2))

for (ci, cj) in pairs:

resolvents = resolve(ci, cj)

if [] in resolvents:

print(f"Derived empty clause from {ci} and {cj}")

return True

new.update(tuple(sorted(r)) for r in resolvents)

if new.issubset(set(tuple(sorted(c)) for c in clauses)):

# No new clauses added — cannot derive contradiction

return False

for c in new:

if list(c) not in clauses:

clauses.append(list(c))

def resolve(ci, cj):

"""Resolve two clauses and return the resolvents."""

resolvents = []

for di in ci:

for dj in cj:

if di == negate(dj):

new\_clause = list(set(ci + cj))

new\_clause.remove(di)

new\_clause.remove(dj)

resolvents.append(new\_clause)

return resolvents

def negate(literal):

"""Negate a literal."""

if literal.startswith('~'):

return literal[1:]

else:

return '~' + literal

KB = [

['~R', 'W'],

['~W', 'G'],

['R']

]

query = 'G'

# --- Run Resolution ---

entailed = pl\_resolution(KB, query)

print("1BM23CS304 Sarim")

print("\nResult:")

if entailed:

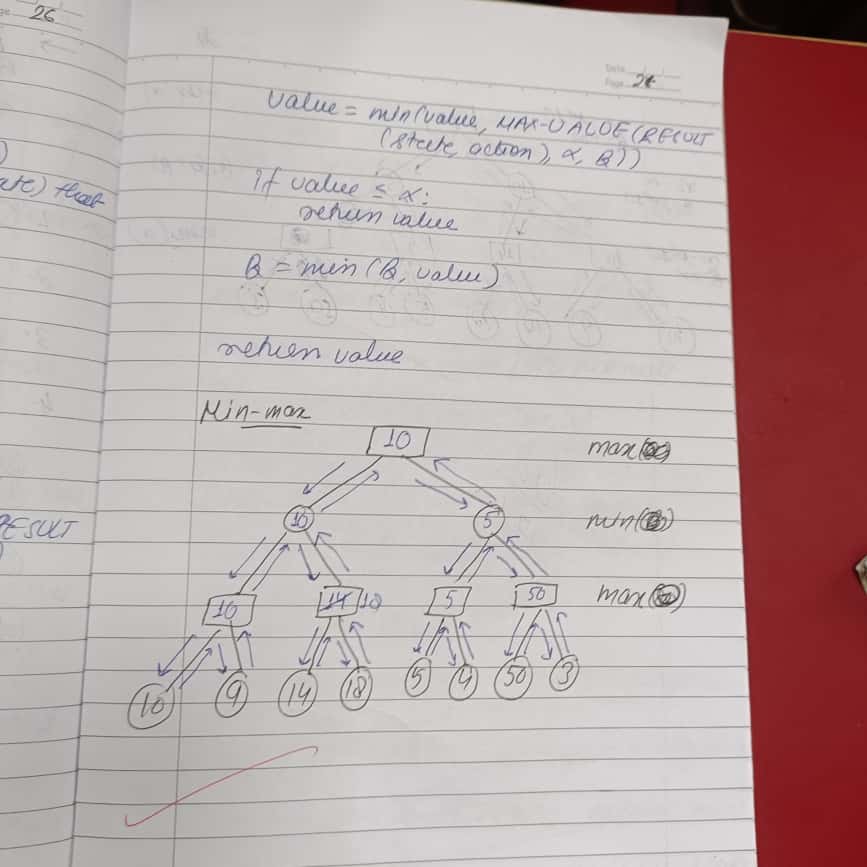
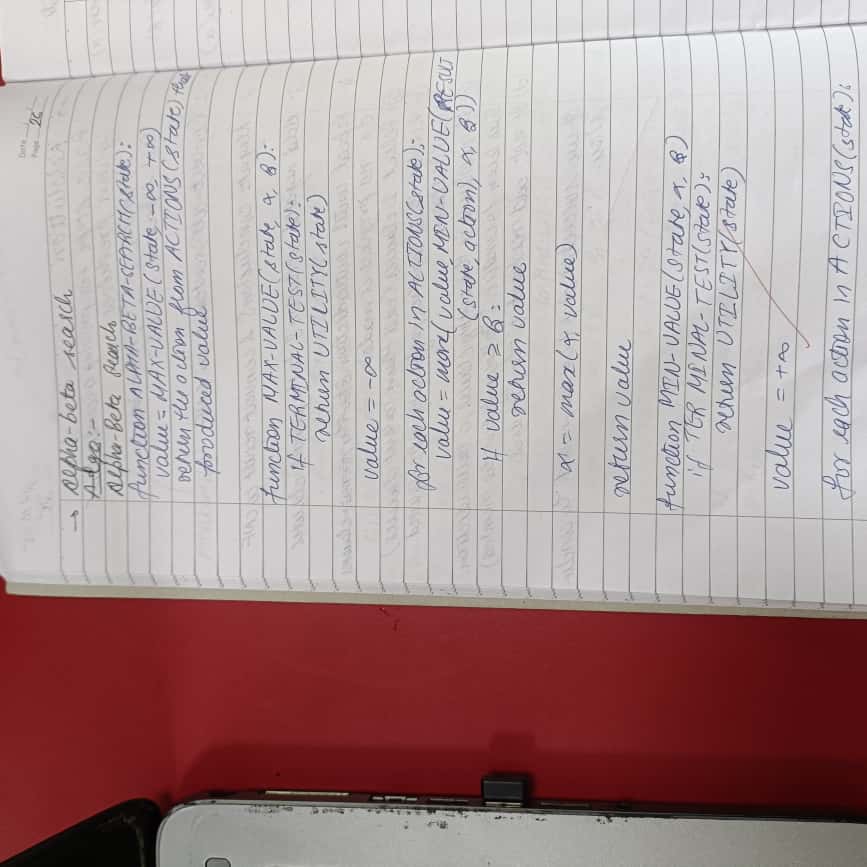
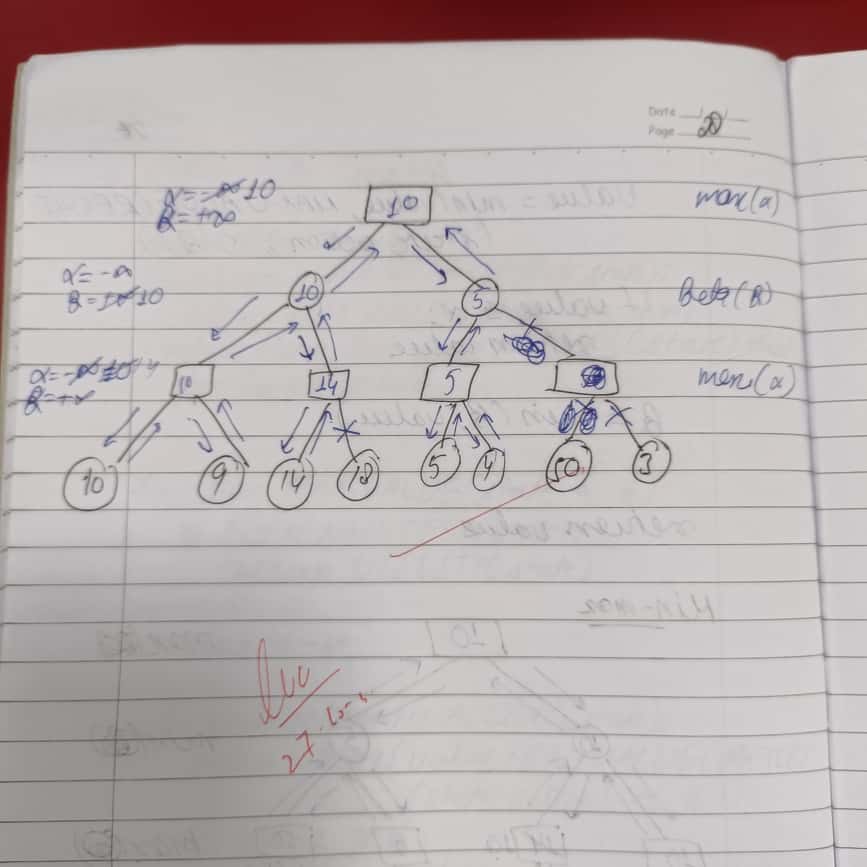
print(f"✅ The knowledge base entails {query}.")

else:

print(f"❌ The knowledge base does NOT entail {query}.")

**Program 10:**

Implement Alpha-Beta Pruning



# --- Alpha-Beta Pruning Implementation ---

import math

# Minimax with Alpha-Beta Pruning

def alphabeta(depth, node\_index, maximizing\_player, values, alpha, beta, max\_depth):

"""

depth: current depth in the game tree

node\_index: index of the current node in the list of values

maximizing\_player: boolean, True if it's the maximizing player's turn

values: list of terminal node values

alpha, beta: alpha-beta values for pruning

max\_depth: maximum depth of the tree

"""

# Base case: if we reach the leaf node

if depth == max\_depth:

return values[node\_index]

if maximizing\_player:

best = -math.inf

# Explore left and right child

for i in range(2):

val = alphabeta(depth + 1, node\_index \* 2 + i, False, values, alpha, beta, max\_depth)

best = max(best, val)

alpha = max(alpha, best)

# Prune

if beta <= alpha:

break

return best

else:

best = math.inf

# Explore left and right child

for i in range(2):

val = alphabeta(depth + 1, node\_index \* 2 + i, True, values, alpha, beta, max\_depth)

best = min(best, val)

beta = min(beta, best)

# Prune

if beta <= alpha:

break

return best

# --- Example Game Tree ---

# Terminal node values (leaf nodes)

values = [3, 5, 6, 9]

# Tree depth = log2(len(values)) = 2

max\_depth = 2

# Run alpha-beta pruning

best\_value = alphabeta(0, 0, True, values, -math.inf, math.inf, max\_depth)

print("1BM23CS304 Sarim")

print(f"The optimal value is: {best\_value}")