**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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

****

**LAB REPORT**

**on**

**Artificial Intelligence (23CS5PCAIN)**

***Submitted by***

**Soham Hathi (1BM23CS335)**

***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**

****

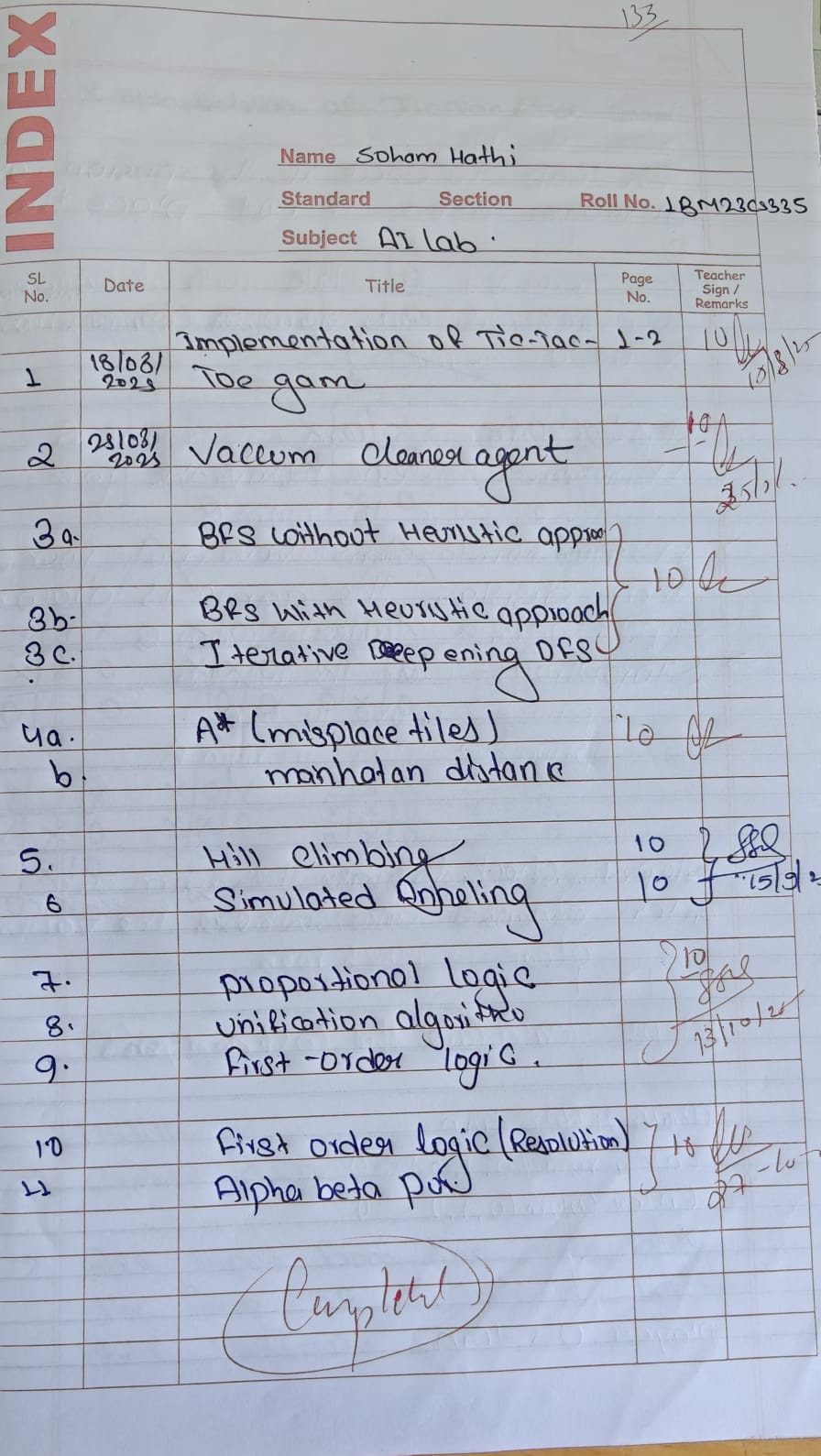
**CERTIFICATE**

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Soham Hathi (1BM23CS335),** 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.

|  |  |
| --- | --- |
| Sonika Sharma  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 30-9-2024 | Implement Tic –Tac –Toe Game  Implement vacuum cleaner agent | 5-11 |
| 2 | 7-10-2024 | Implement 8 puzzle problems using Depth First Search (DFS)  Implement Iterative deepening search algorithm | 12-18 |
| 3 | 14-10-2024 | Implement A\* search algorithm | 19-27 |
| 4 | 21-10-2024 | Implement Hill Climbing search algorithm to solve N-Queens problem | 28-31 |
| 5 | 28-10-2024 | Simulated Annealing to Solve 8-Queens problem | 32-34 |
| 6 | 11-11-2024 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 35-39 |
| 7 | 2-12-2024 | Implement unification in first order logic | 40-44 |
| 8 | 2-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 45-48 |
| 9 | 16-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 49-55 |
| 10 | 16-12-2024 | Implement Alpha-Beta Pruning. | 56-60 |



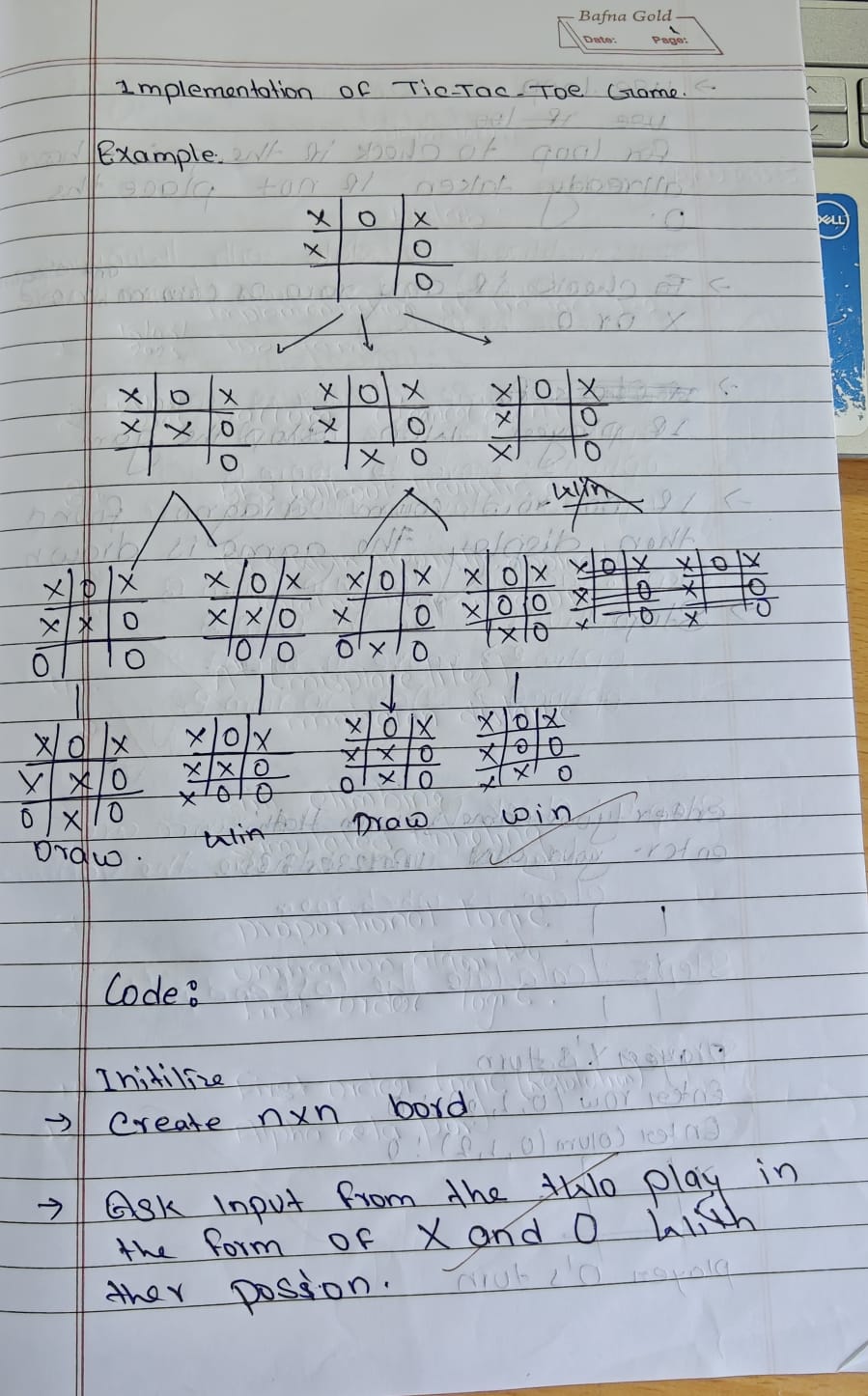
Github Link: <https://github.com/sohamhathi/sohamHathi-1BM23CS335-AI.git>

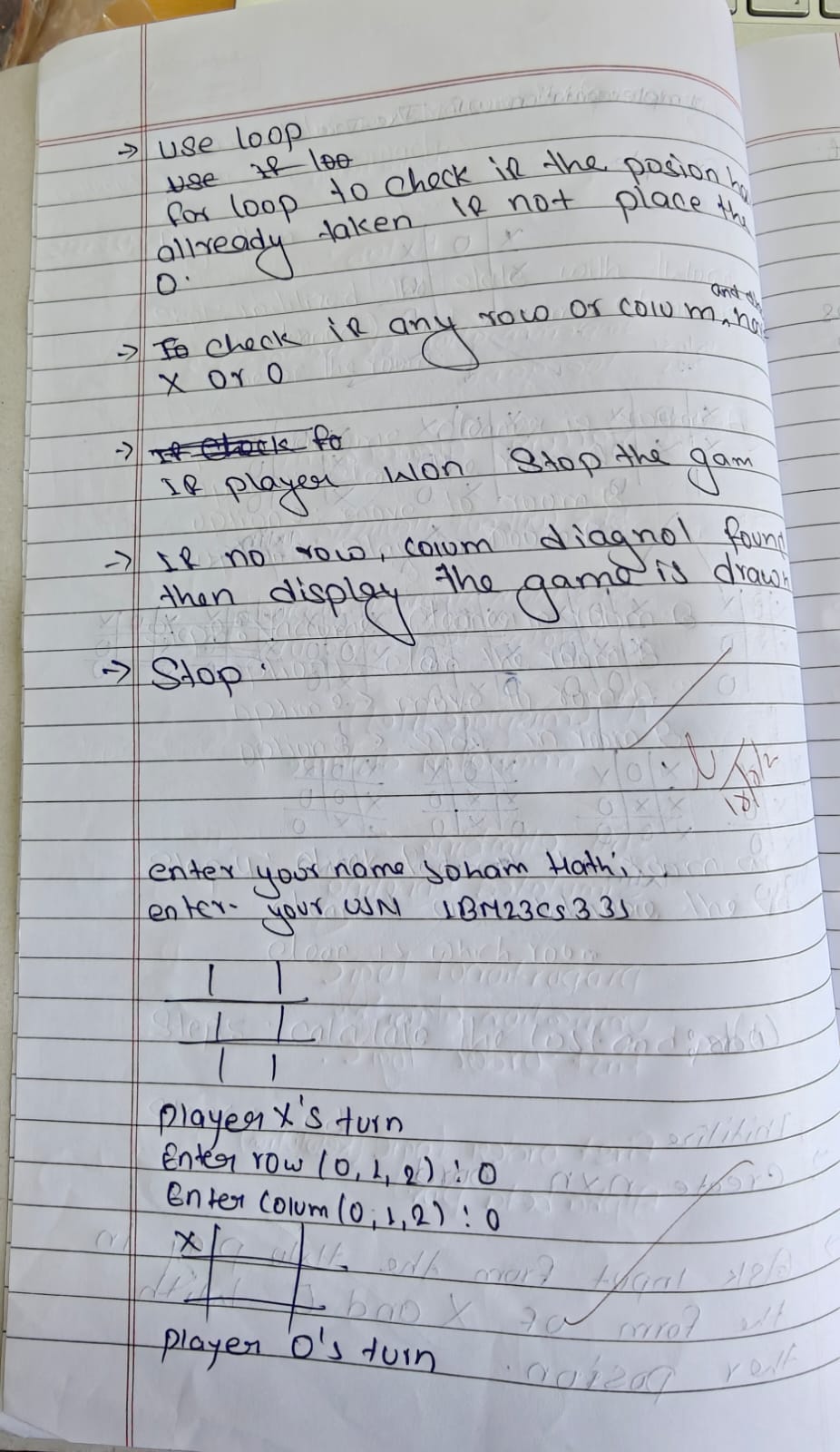
**Program 1**

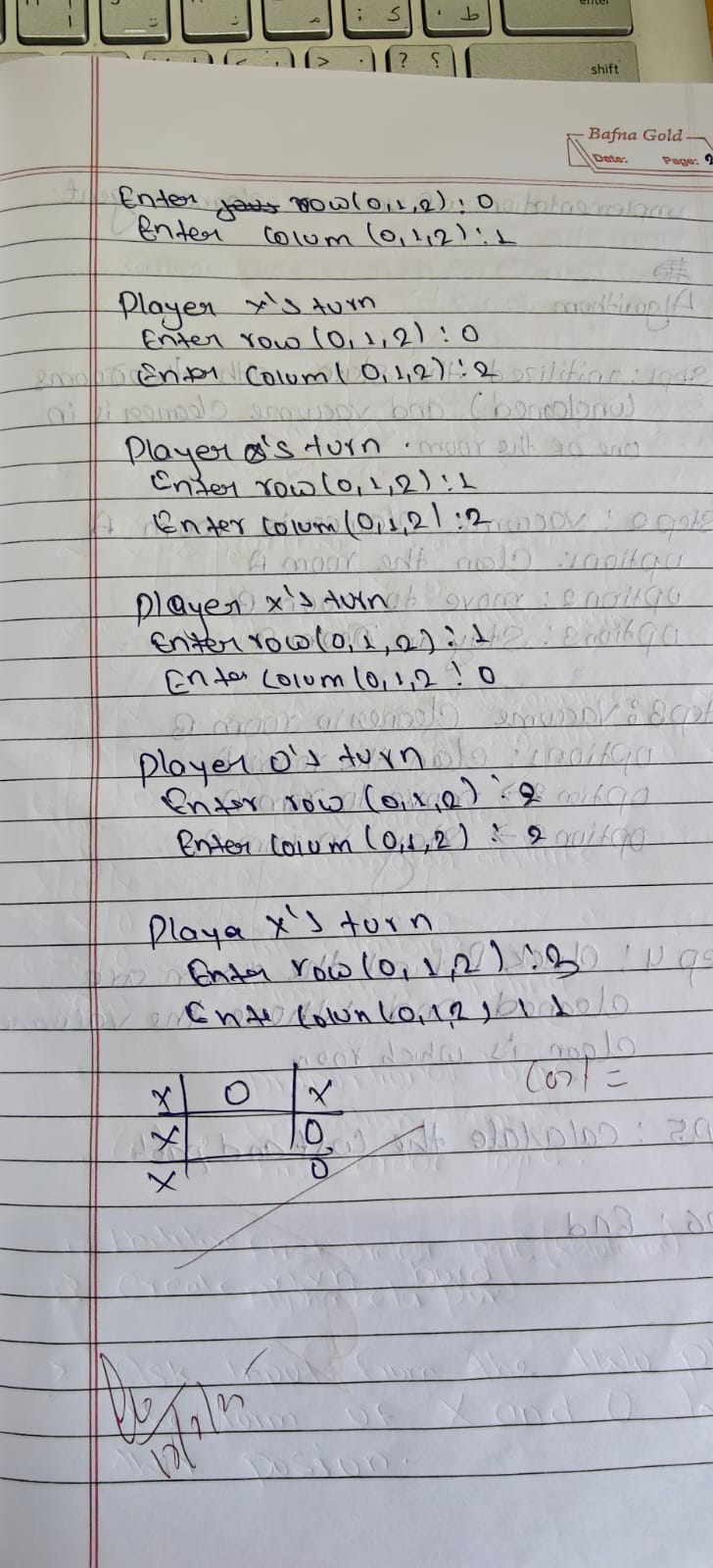
Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:







**Code:**

# tic tac toe game lab 1

name = input("enter your name ");

usn=input ("enter your usn");

def print\_board(board):

for row in board:

print(" | ".join(row))

print("-" \* 5)

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\_full(board):

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

def tic\_tac\_toe():

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

current\_player = "X"

path\_cost = 0

while True:

print\_board(board)

print(f"Player {current\_player}'s turn.")

# Input move

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

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

if board[row][col] == " ":

board[row][col] = current\_player

path\_cost += 1

else:

print("Invalid move! Try again.")

continue

if check\_winner(board, current\_player):

print\_board(board)

print(f"Player {current\_player} wins!")

print(f"Path cost: {path\_cost}")

break

if is\_full(board):

print\_board(board)

print("It's a draw!")

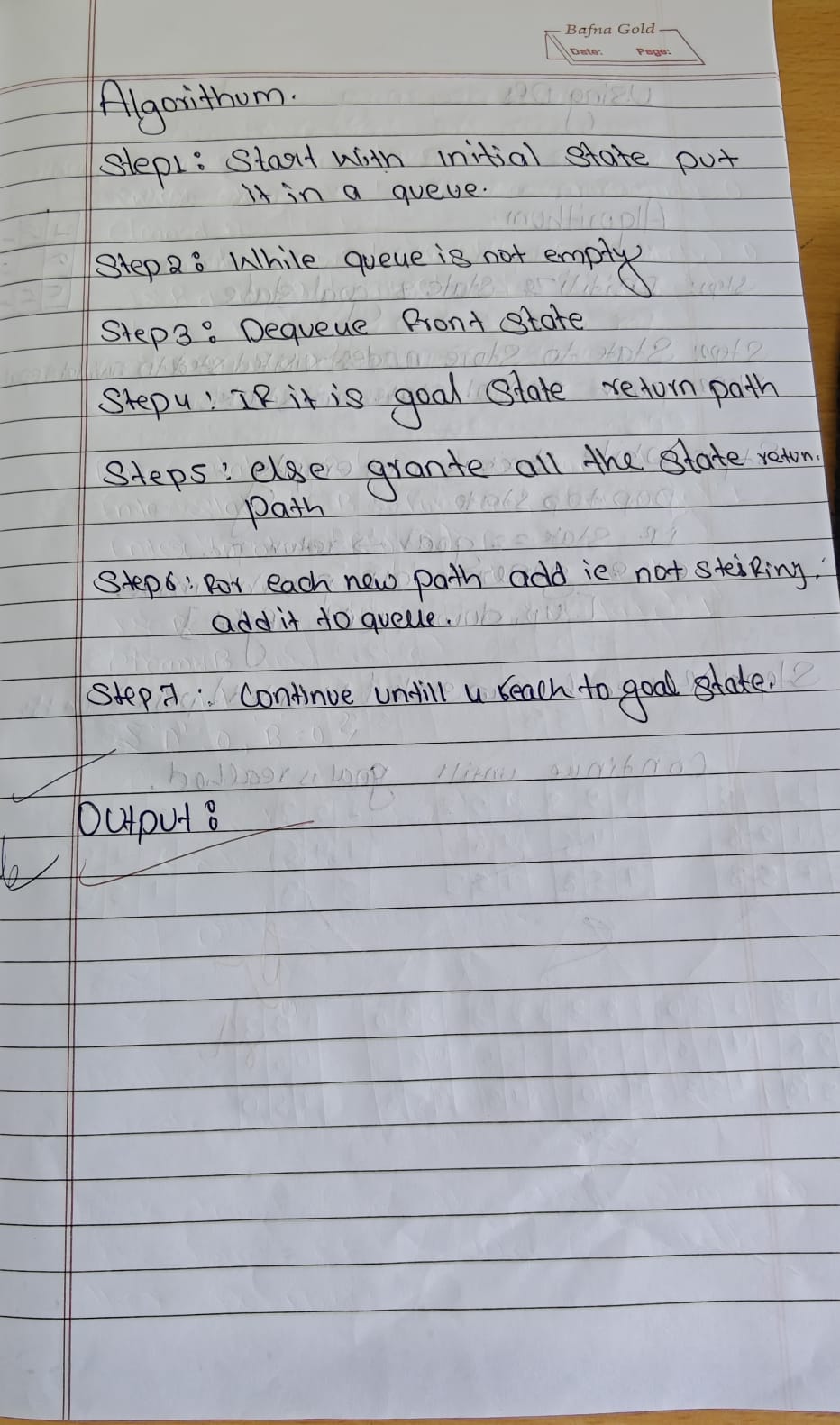
print(f"Path cost: {path\_cost}")

break

current\_player = "O" if current\_player == "X" else "X"

tic\_tac\_toe()

**Implement vacuum cleaner agent**



def vacuum\_cleaner\_4\_rooms\_grid():

try:

state\_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))

state\_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))

state\_C = int(input("Enter state of C (0 for clean, 1 for dirty): "))

state\_D = int(input("Enter state of D (0 for clean, 1 for dirty): "))

location = input("Enter location (A, B, C, D): ").upper()

if (state\_A not in (0, 1) or state\_B not in (0, 1) or

state\_C not in (0, 1) or state\_D not in (0, 1) or

location not in ('A', 'B', 'C', 'D')):

raise ValueError("Invalid input! Enter 0 or 1 for states, and location as A/B/C/D.")

except ValueError as e:

print("Error:", e)

return

environment = {'A': state\_A, 'B': state\_B, 'C': state\_C, 'D': state\_D}

cost = 0

moves = {

'A': {'RIGHT': 'B', 'DOWN': 'C'},

'B': {'LEFT': 'A', 'DOWN': 'D'},

'C': {'UP': 'A', 'RIGHT': 'D'},

'D': {'UP': 'B', 'LEFT': 'C'}

}

while True:

print("\nCurrent Environment:", environment)

print("Vacuum is at Room", location)

if environment[location] == 1:

print(f"{location} is dirty. Cleaning...")

environment[location] = 0

cost += 1

print(f"{location} is now clean.")

else:

print(f"{location} is already clean.")

if all(state == 0 for state in environment.values()):

print("\nAll rooms are clean. Turning vacuum off.")

break

move = input("Enter action (LEFT / RIGHT / UP / DOWN / STAY / EXIT): ").upper()

if move in moves[location]:

new\_location = moves[location][move]

print(f"Moving vacuum {move} to {new\_location}.")

location = new\_location

cost += 1

elif move == "STAY":

print("Staying in the same room. (No cost)")

elif move == "EXIT":

print("Exiting manually.")

break

else:

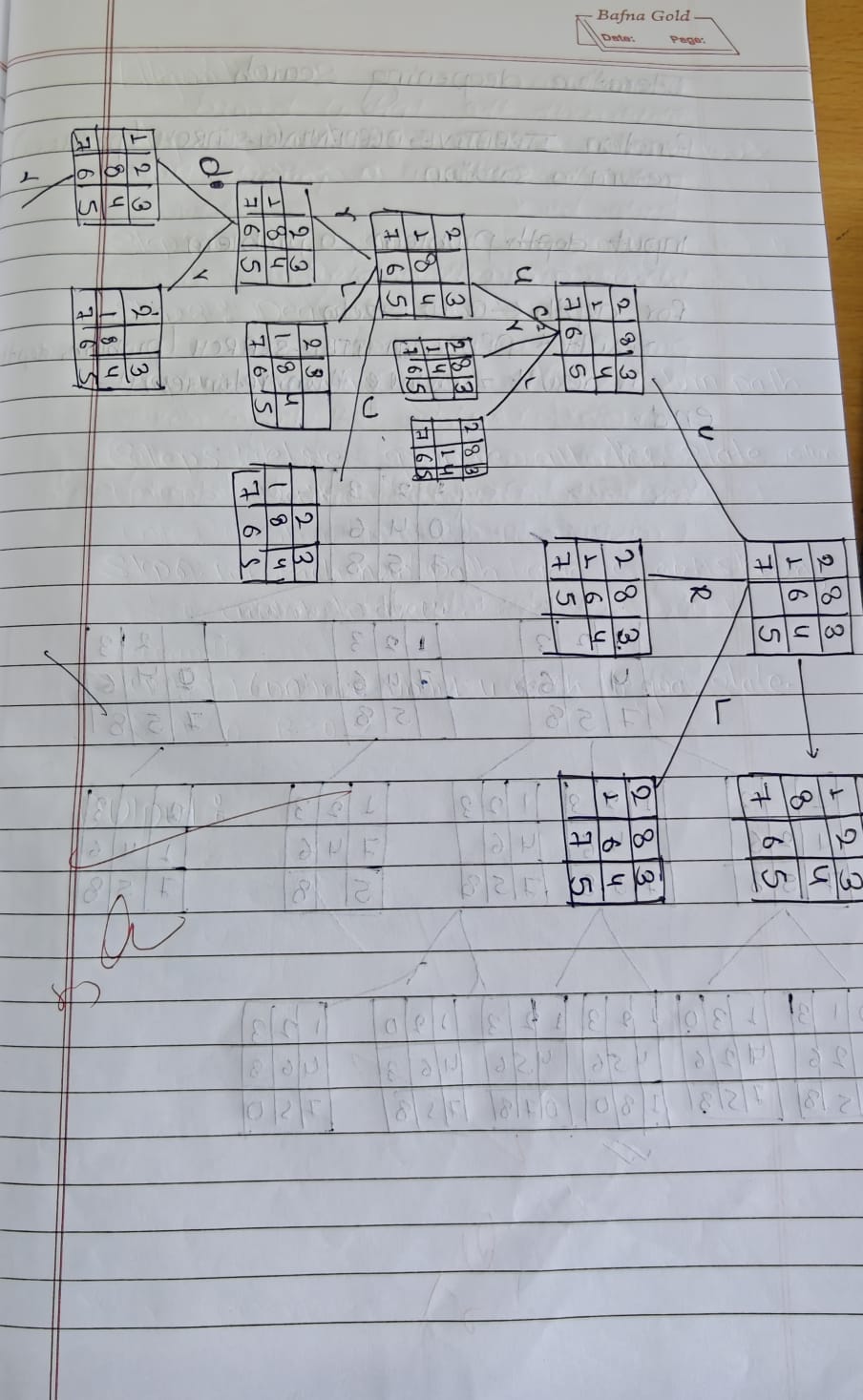
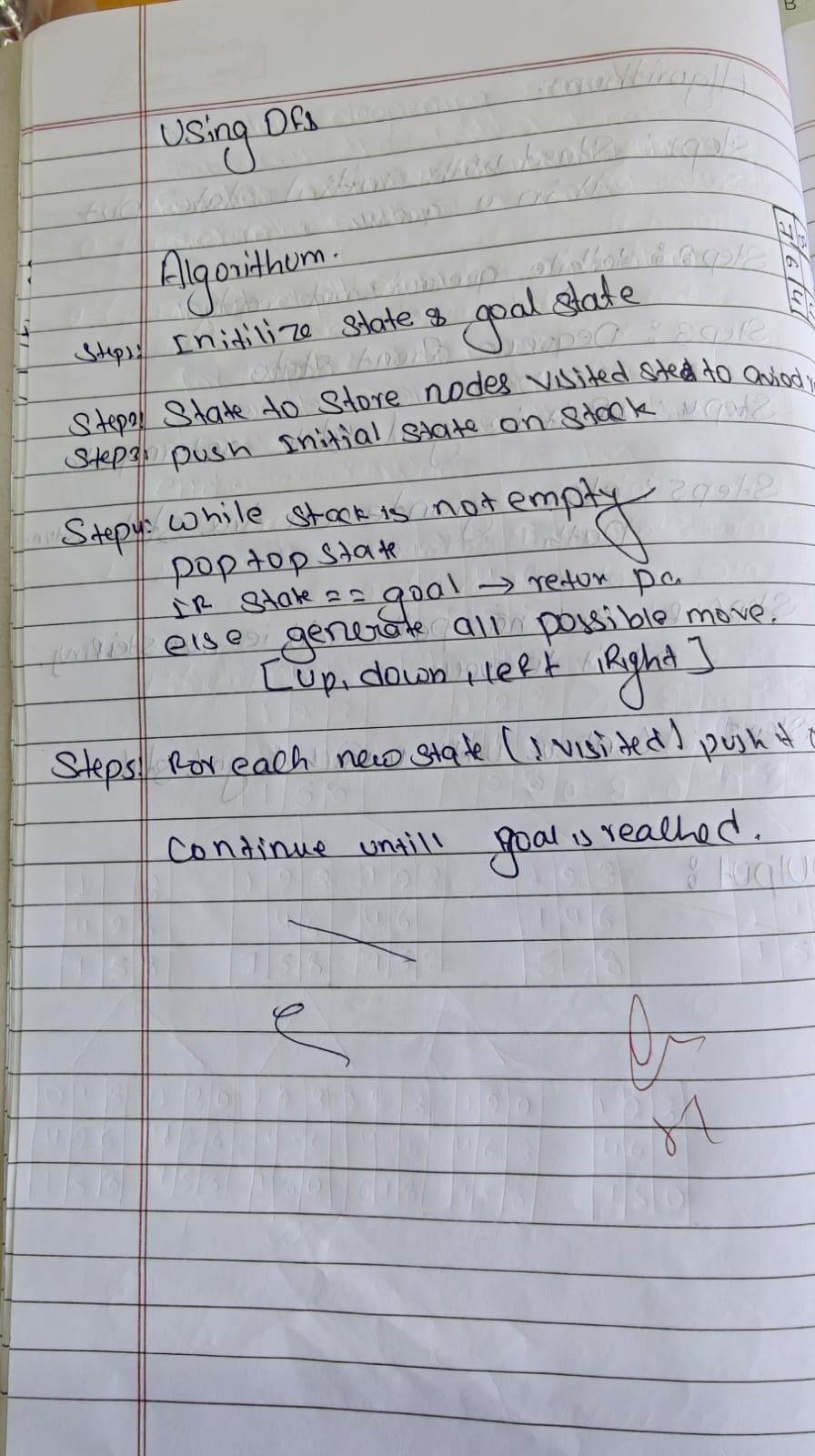
print("Invalid move from this room! Try again.")

print("\nFinal Environment:", environment)

print("Total Cost:", cost)

vacuum\_cleaner\_4\_rooms\_grid()

**PROGRAM-2**



print("Soham Hathi")

print("1BM23CS335")

from copy import deepcopy

GOAL\_STATE = [[1, 2, 3],

[8, 0, 4],

[7, 6, 5]]

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

def is\_valid(x, y):

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

def serialize(state):

return str(state)

def find\_zero(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\_zero(state)

for dx, dy in DIRECTIONS:

nx, ny = x + dx, y + dy

if is\_valid(nx, ny):

new\_state = deepcopy(state)

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(new\_state)

return neighbors

def dfs(start\_state, max\_depth=30):

visited = set()

path = []

solution\_path = []

found = False

def dfs\_recursive(state, depth):

nonlocal found, solution\_path

if depth > max\_depth or found:

return

serial = serialize(state)

if serial in visited:

return

visited.add(serial)

path.append(state)

if state == GOAL\_STATE:

solution\_path = path.copy()

found = True

return

for neighbor in get\_neighbors(state):

dfs\_recursive(neighbor, depth + 1)

path.pop()

visited.remove(serial)

dfs\_recursive(start\_state, 0)

if found:

return solution\_path, len(solution\_path) - 1

else:

return None, -1

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

start = [[2, 8, 3],

[1, 6, 4],

[7, 0, 5]]

path, cost = dfs(start, max\_depth=50)

if path:

print(f"DFS found a path with cost = {cost}:")

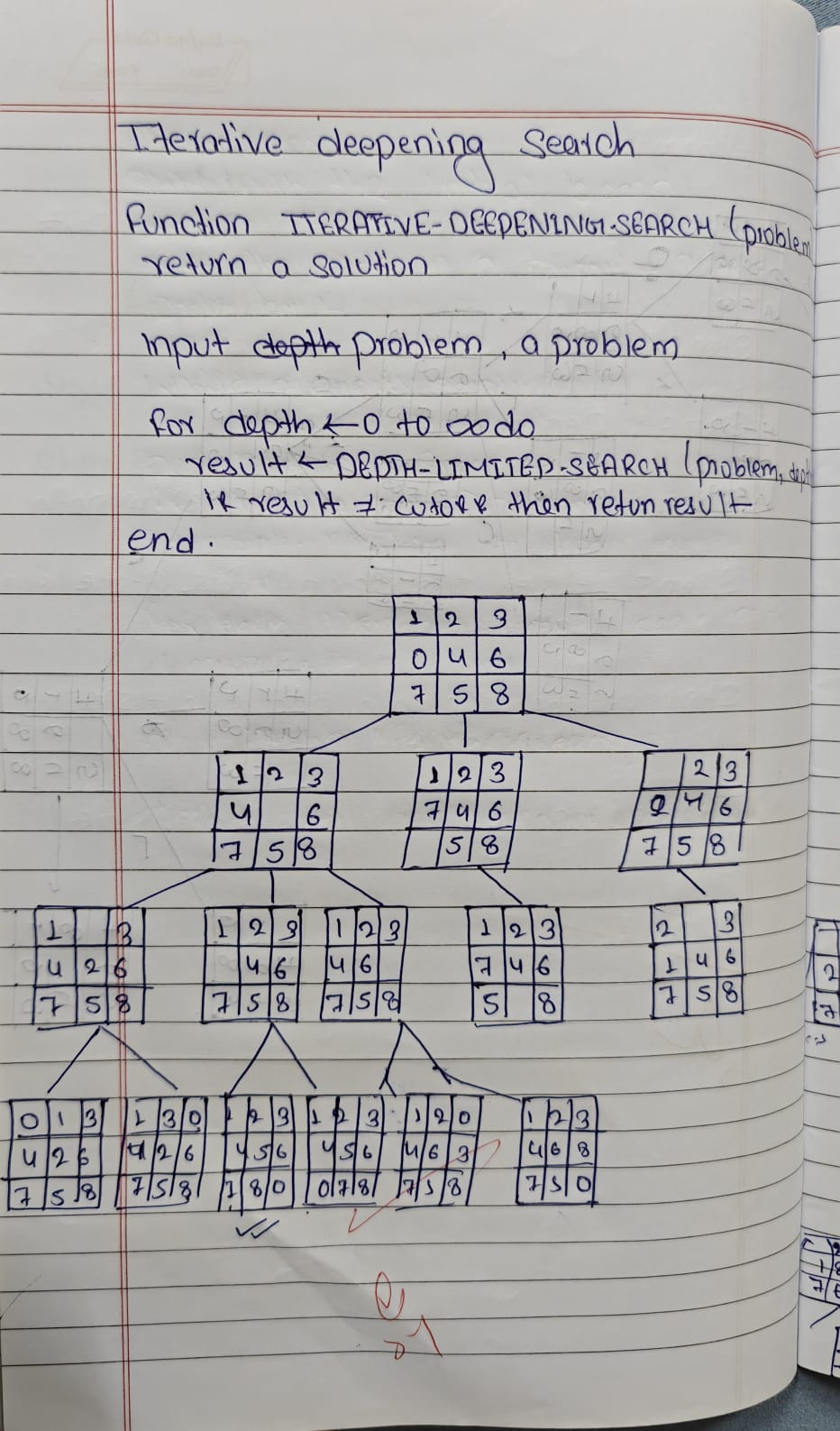
for step in path:

for row in step:

print(row)

print("------")

else:

print("No solution found within depth limit.") 

**ITERATIVE DFS**

from copy import deepcopy

print("soham Hathi")

print("1BM23CS335")

# Goal state

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 find\_blank(state):

for i in range(3):

for j in range(3):

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

return i, j

return None

def is\_goal(state):

return state == GOAL\_STATE

def get\_successors(state):

successors = []

x, y = find\_blank(state)

for move, (dx, dy) in DIRECTIONS.items():

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

if 0 <= new\_x < 3 and 0 <= new\_y < 3:

new\_state = deepcopy(state)

new\_state[x][y], new\_state[new\_x][new\_y] = new\_state[new\_x][new\_y], new\_state[x][y]

successors.append((new\_state, move))

return successors

def depth\_limited\_search(state, depth, path, visited):

if is\_goal(state):

return path

if depth == 0:

return None

visited.append(state)

for successor, move in get\_successors(state):

if successor not in visited:

result = depth\_limited\_search(successor, depth - 1, path + [move], visited)

if result is not None:

return result

visited.pop()

return None

# Iterative Deepening DFS

def iterative\_deepening\_search(start\_state, max\_depth=50):

for depth in range(max\_depth):

print(f"Trying depth limit: {depth}")

visited = []

result = depth\_limited\_search(start\_state, depth, [], visited)

if result is not None:

return result

return None

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

initial\_state = [[2, 8, 3],

[1 ,6, 4],

[7, 0, 5]]

solution = iterative\_deepening\_search(initial\_state)

if solution:

print("\nSolution found!")

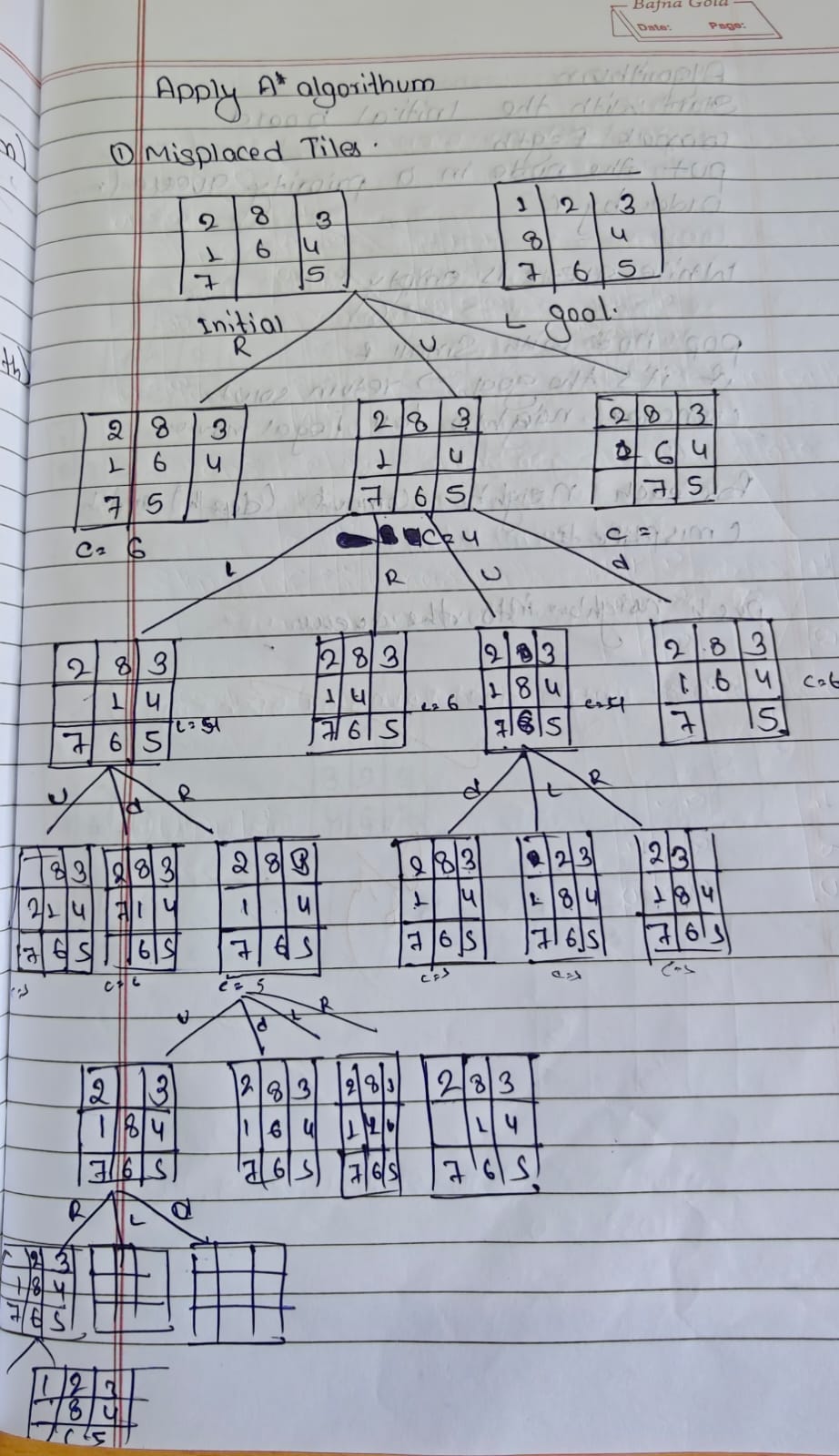
print("Moves to solve:", solution)

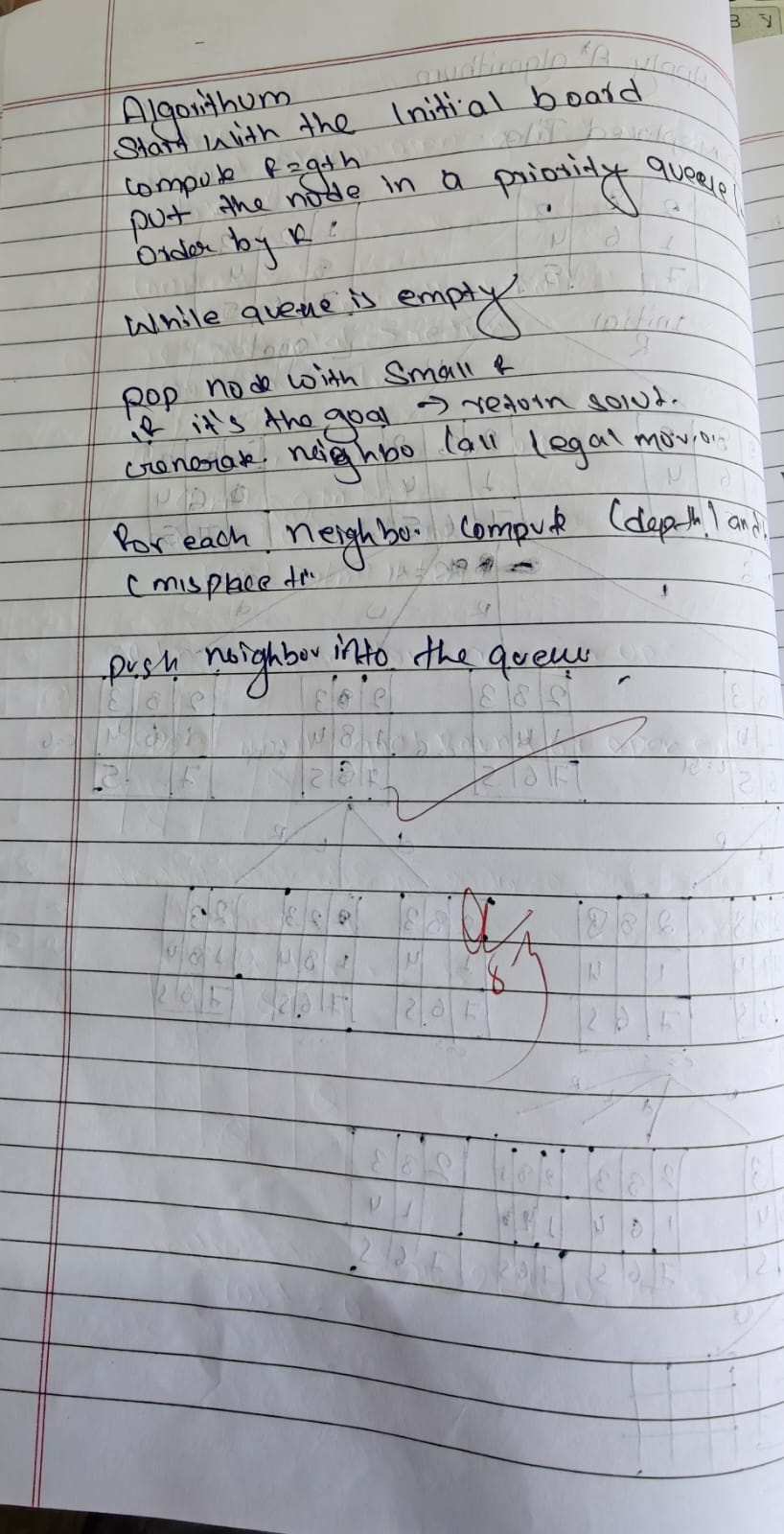
print("Number of moves:", len(solution))

else:

print("No solution found within depth limit.")

**PROGRAM-3**





import heapq

print("soham Hathi")

print("1BM23CS335")

def manhattan\_distance(state, goal):

"""Calculate total Manhattan distance of all tiles from their goal positions."""

goal\_positions = {}

for i in range(3):

for j in range(3):

goal\_positions[goal[i][j]] = (i, j)

dist = 0

for i in range(3):

for j in range(3):

tile = state[i][j]

if tile != 0:

goal\_i, goal\_j = goal\_positions[tile]

dist += abs(i - goal\_i) + abs(j - goal\_j)

return dist

def get\_neighbors(state):

neighbors = []

for i in range(3):

for j in range(3):

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

x, y = i, j

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 = [list(row) for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(tuple(tuple(row) for row in new\_state))

return neighbors

def print\_state(state):

for row in state:

print(' '.join(str(x) if x != 0 else ' ' for x in row))

print()

def get\_user\_state(prompt):

print(prompt)

print("Enter 9 numbers separated by space, use 0 for blank tile (e.g. '2 8 3 1 6 4 7 0 5'):")

while True:

try:

entries = list(map(int, input().strip().split()))

if len(entries) != 9 or set(entries) != set(range(9)):

raise ValueError

break

except ValueError:

print("Invalid input! Enter exactly 9 unique digits from 0 to 8 separated by spaces.")

return tuple(tuple(entries[i\*3:(i+1)\*3]) for i in range(3))

def a\_star\_verbose(start\_state, goal\_state):

open\_set = []

start\_h = manhattan\_distance(start\_state, goal\_state)

heapq.heappush(open\_set, (start\_h, 0, start\_state, [])) # (f, g, state, path)

closed\_set = set()

step\_counter = 0

while open\_set:

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

step\_counter += 1

print(f"Step {step\_counter}:")

print(f"Current state with f = g + h = {g} + {f - g} = {f}")

print\_state(current)

if current == goal\_state:

print("Goal reached!")

return path + [current]

if current in closed\_set:

print("This state has already been visited. Skipping.\n")

continue

closed\_set.add(current)

neighbors = get\_neighbors(current)

print(f"Expanding neighbors ({len(neighbors)}):")

for n in neighbors:

if n not in closed\_set:

h = manhattan\_distance(n, goal\_state)

new\_g = g + 1

new\_f = new\_g + h

print(f"Neighbor state with g={new\_g}, h={h}, f={new\_f}:")

print\_state(n)

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

else:

print("Neighbor already visited, skipping.")

print("-----\n")

return None

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

start = get\_user\_state("Enter the START state:")

goal = get\_user\_state("Enter the GOAL state:")

print("\nStarting A\* search with Manhattan distance heuristic...\n")

solution = a\_star\_verbose(start, goal)

if solution:

print(f"\nSolution found in {len(solution)-1} moves:\n")

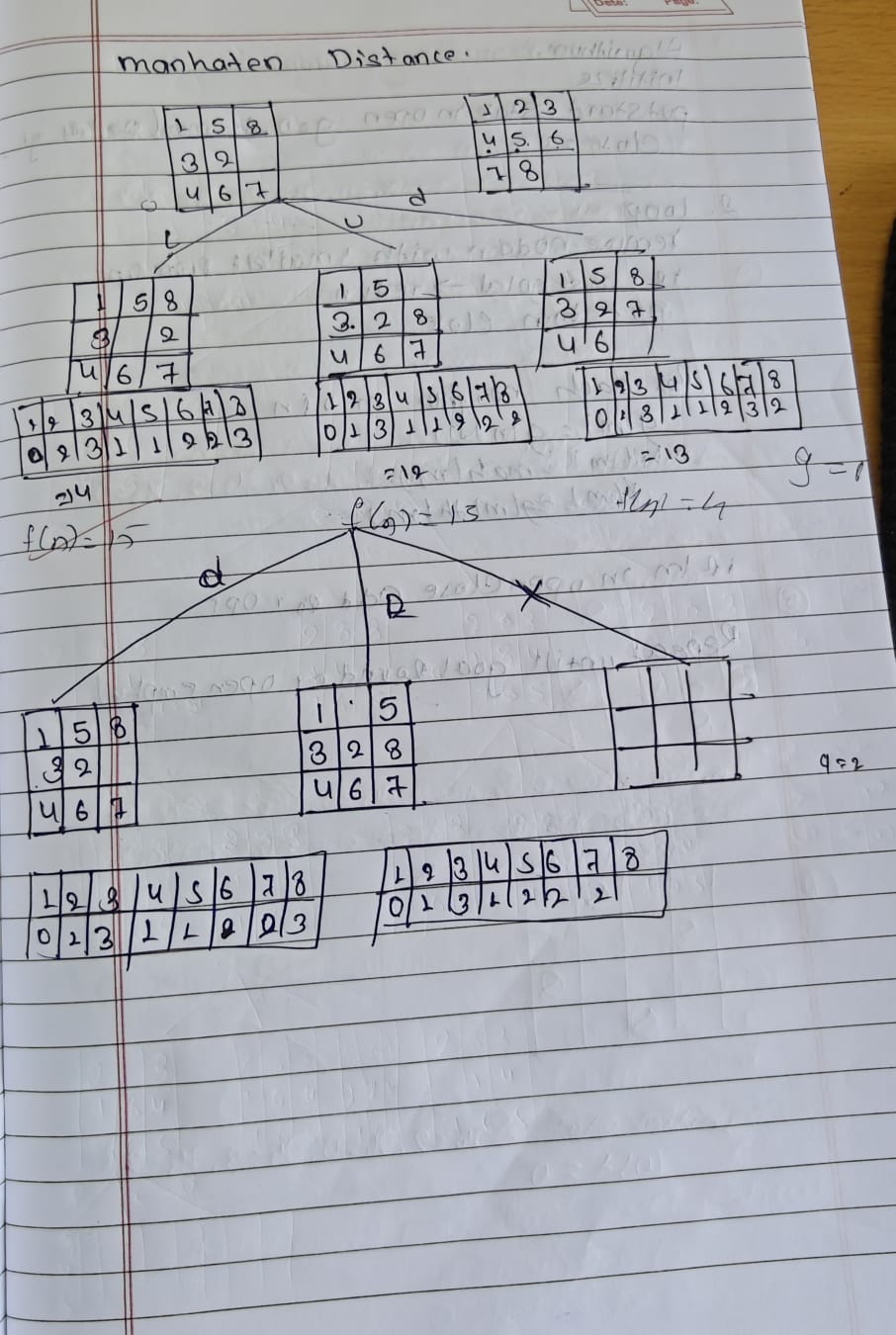
for step\_num, step in enumerate(solution):

print(f"Step {step\_num}:")

print\_state(step)

else:

print("No solution found.")



**MANHATTAN DISTANCE**

import heapq

print("soham Hathi")

print("1BM23CS335")

def manhattan\_distance(state, goal):

"""Calculate total Manhattan distance of all tiles from their goal positions."""

goal\_positions = {}

for i in range(3):

for j in range(3):

goal\_positions[goal[i][j]] = (i, j)

dist = 0

for i in range(3):

for j in range(3):

tile = state[i][j]

if tile != 0:

goal\_i, goal\_j = goal\_positions[tile]

dist += abs(i - goal\_i) + abs(j - goal\_j)

return dist

def get\_neighbors(state):

neighbors = []

for i in range(3):

for j in range(3):

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

x, y = i, j

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 = [list(row) for row in state]

new\_state[x][y], new\_state[nx][ny] = new\_state[nx][ny], new\_state[x][y]

neighbors.append(tuple(tuple(row) for row in new\_state))

return neighbors

def print\_state(state):

for row in state:

print(' '.join(str(x) if x != 0 else ' ' for x in row))

print()

def get\_user\_state(prompt):

print(prompt)

print("Enter 9 numbers separated by space, use 0 for blank tile (e.g. '2 8 3 1 6 4 7 0 5'):")

while True:

try:

entries = list(map(int, input().strip().split()))

if len(entries) != 9 or set(entries) != set(range(9)):

raise ValueError

break

except ValueError:

print("Invalid input! Enter exactly 9 unique digits from 0 to 8 separated by spaces.")

return tuple(tuple(entries[i\*3:(i+1)\*3]) for i in range(3))

def a\_star\_verbose(start\_state, goal\_state):

open\_set = []

start\_h = manhattan\_distance(start\_state, goal\_state)

heapq.heappush(open\_set, (start\_h, 0, start\_state, [])) # (f, g, state, path)

closed\_set = set()

step\_counter = 0

while open\_set:

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

step\_counter += 1

print(f"Step {step\_counter}:")

print(f"Current state with f = g + h = {g} + {f - g} = {f}")

print\_state(current)

if current == goal\_state:

print("Goal reached!")

return path + [current]

if current in closed\_set:

print("This state has already been visited. Skipping.\n")

continue

closed\_set.add(current)

neighbors = get\_neighbors(current)

print(f"Expanding neighbors ({len(neighbors)}):")

for n in neighbors:

if n not in closed\_set:

h = manhattan\_distance(n, goal\_state)

new\_g = g + 1

new\_f = new\_g + h

print(f"Neighbor state with g={new\_g}, h={h}, f={new\_f}:")

print\_state(n)

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

else:

print("Neighbor already visited, skipping.")

print("-----\n")

return None

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

start = get\_user\_state("Enter the START state:")

goal = get\_user\_state("Enter the GOAL state:")

print("\nStarting A\* search with Manhattan distance heuristic...\n")

solution = a\_star\_verbose(start, goal)

if solution:

print(f"\nSolution found in {len(solution)-1} moves:\n")

for step\_num, step in enumerate(solution):

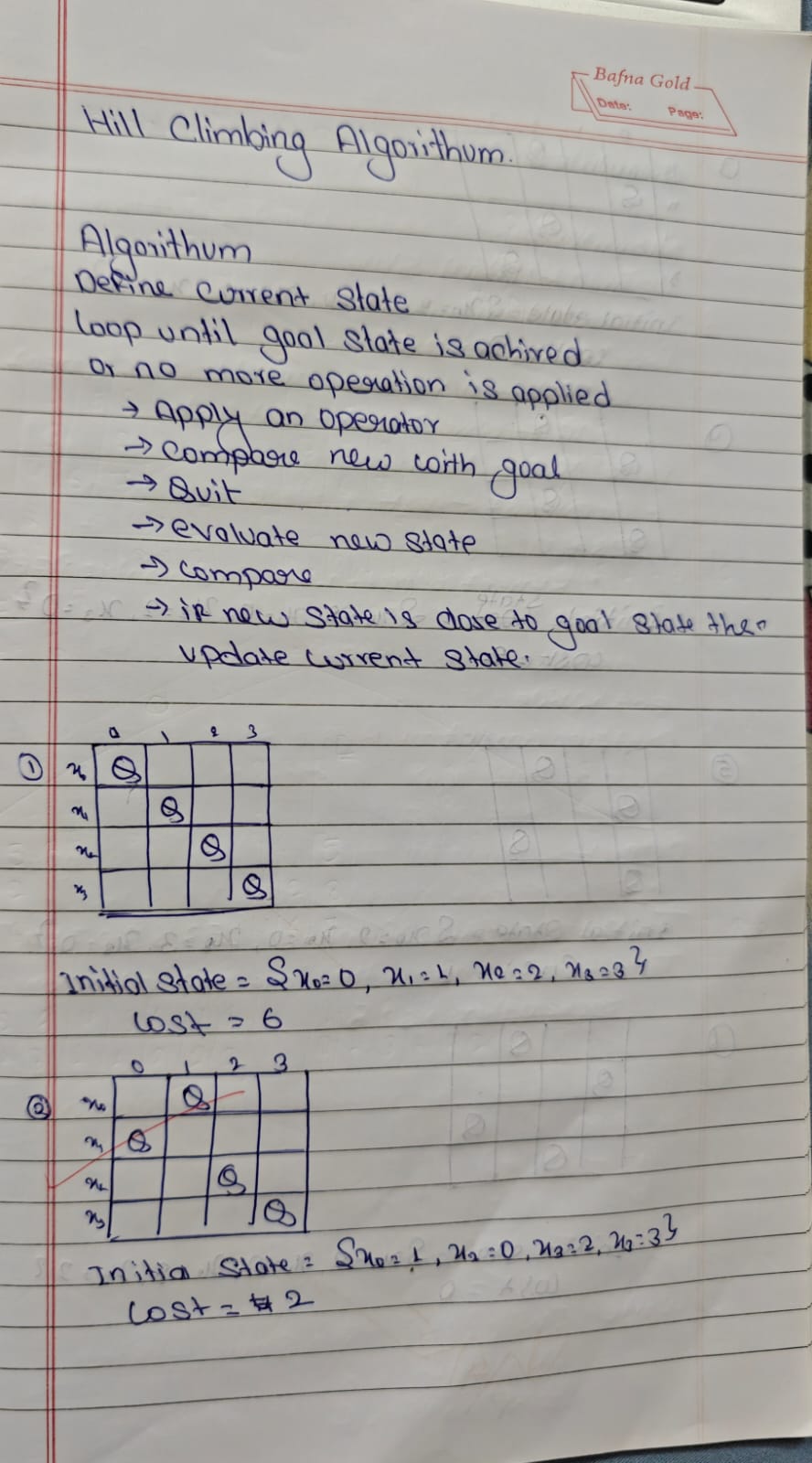
print(f"Step {step\_num}:")

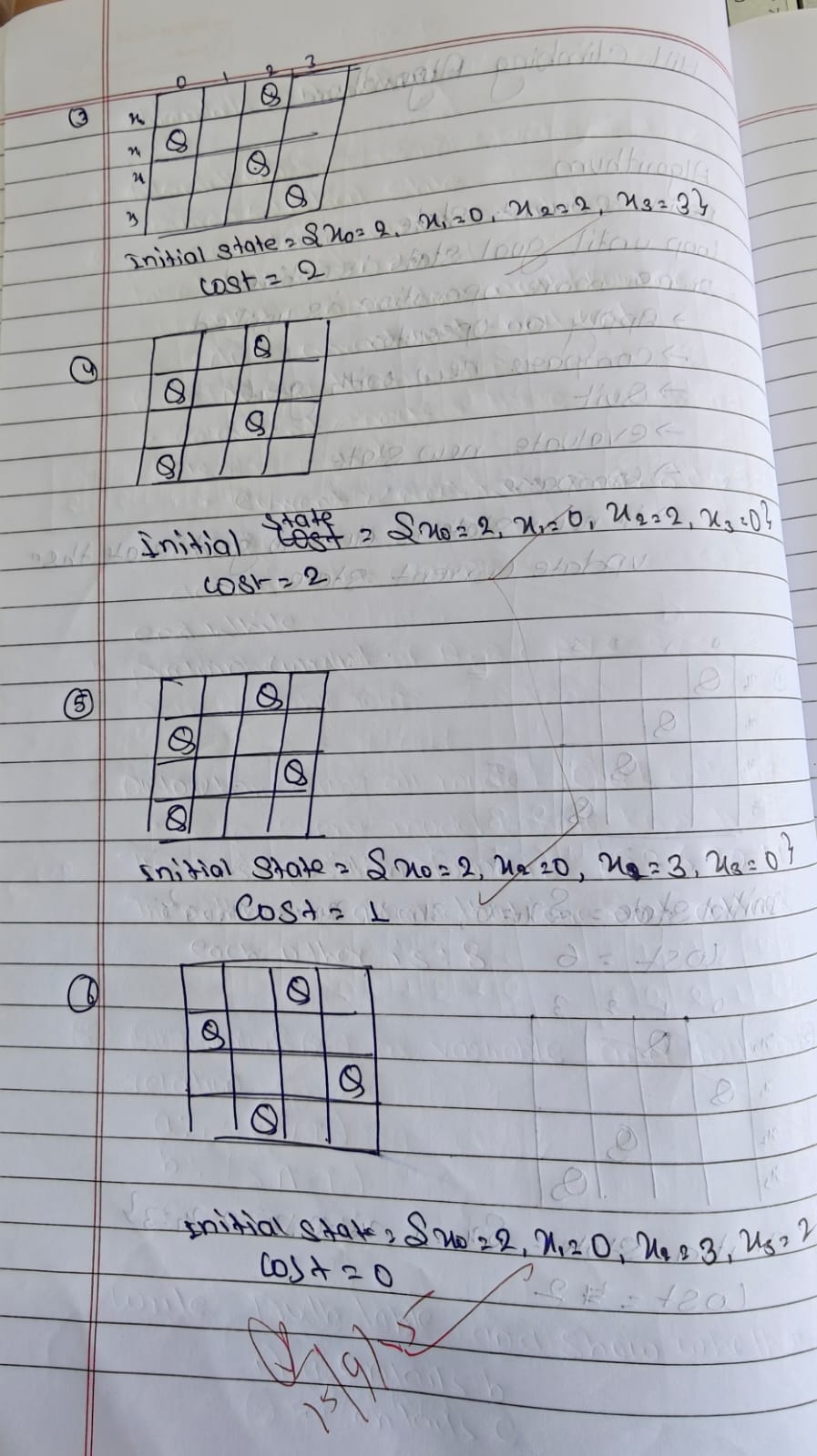
print\_state(step)

else:

print("No solution found.")

**PROGRAM-4**





import random

import copy

print("soham Hathi")

print("1BM23CS335")

N = 4

def print\_board(state):

for row in range(N):

line = ""

for col in range(N):

if state[col] == row:

line += " Q "

else:

line += " . "

print(line)

print()

def heuristic(state):

"""Calculate the number of pairs of queens attacking each other."""

h = 0

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):

"""Generate all neighboring states (each queen moved to another row in its column)."""

neighbors = []

for col in range(N):

for row in range(N):

if state[col] != row:

neighbor = list(state)

neighbor[col] = row

neighbors.append(neighbor)

return neighbors

def hill\_climbing(initial\_state):

current = initial\_state

step = 0

while True:

print(f"\n🔄 Step {step}:")

print(f"Current state: {current}")

print(f"Heuristic: {heuristic(current)}")

print\_board(current)

neighbors = get\_neighbors(current)

neighbor\_heuristics = [(neighbor, heuristic(neighbor)) for neighbor in neighbors]

print("Generated neighbors and their heuristics:")

for i, (neighbor, h) in enumerate(neighbor\_heuristics):

print(f"{i + 1}. {neighbor} -> h = {h}")

best\_neighbor, best\_h = min(neighbor\_heuristics, key=lambda x: x[1])

if best\_h >= heuristic(current):

print("\n Local minimum reached. Stopping.")

break

current = best\_neighbor

step += 1

if best\_h == 0:

print("\n✅ Goal state found!")

print\_board(current)

break

return current

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

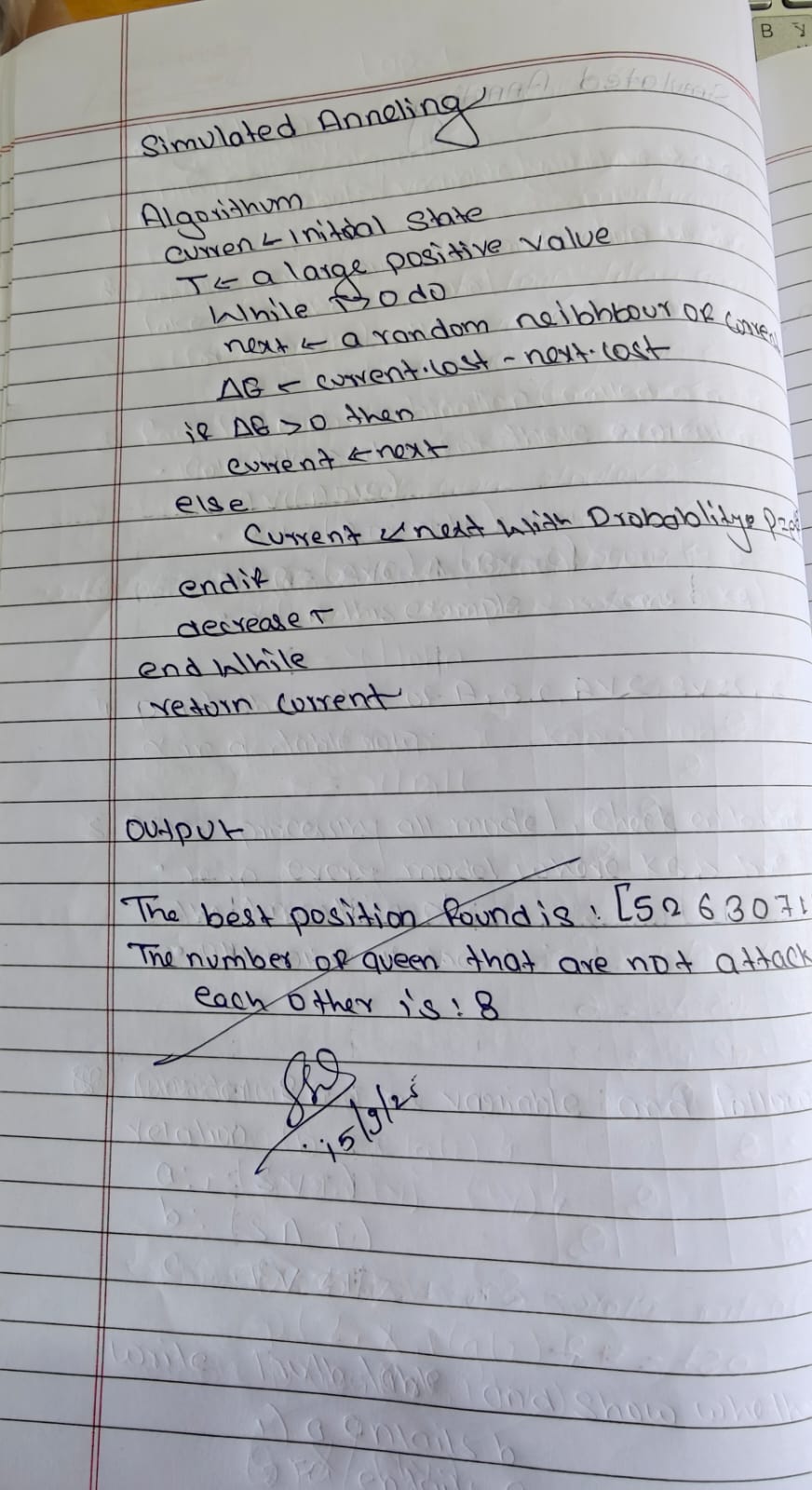
print(" Initial Random State:", random\_initial\_state)

final\_state = hill\_climbing(random\_initial\_state)

print(" Final State:", final\_state)

print("Final Heuristic:", heuristic(final\_state))

**PROGRAM-5**



import numpy as np

print("soham Hathi")

print("1BM23CS335")

def compute\_conflicts(queens):

"""Compute number of pairs of queens attacking each other."""

conflicts = 0

n = len(queens)

for i in range(n):

for j in range(i + 1, n):

if queens[i] == queens[j] or abs(queens[i] - queens[j]) == abs(i - j):

conflicts += 1

return conflicts

def random\_neighbor(queens):

"""Generate a neighbor by swapping two random positions."""

n = len(queens)

new\_queens = queens.copy()

i, j = np.random.choice(n, 2, replace=False)

new\_queens[i], new\_queens[j] = new\_queens[j], new\_queens[i]

return new\_queens

def simulated\_annealing(n, initial\_temp=100, cooling\_rate=0.95, max\_iter=10000):

queens = np.arange(n) # Start with a permutation (0 to n-1)

current\_conflicts = compute\_conflicts(queens)

best\_queens = queens.copy()

best\_conflicts = current\_conflicts

temp = initial\_temp

for i in range(max\_iter):

if temp <= 1e-3 or best\_conflicts == 0:

break

candidate = random\_neighbor(queens)

candidate\_conflicts = compute\_conflicts(candidate)

delta = candidate\_conflicts - current\_conflicts

if delta < 0 or np.random.rand() < np.exp(-delta / temp):

queens = candidate

current\_conflicts = candidate\_conflicts

if current\_conflicts < best\_conflicts:

best\_queens = queens.copy()

best\_conflicts = current\_conflicts

temp \*= cooling\_rate

return best\_queens, n - best\_conflicts

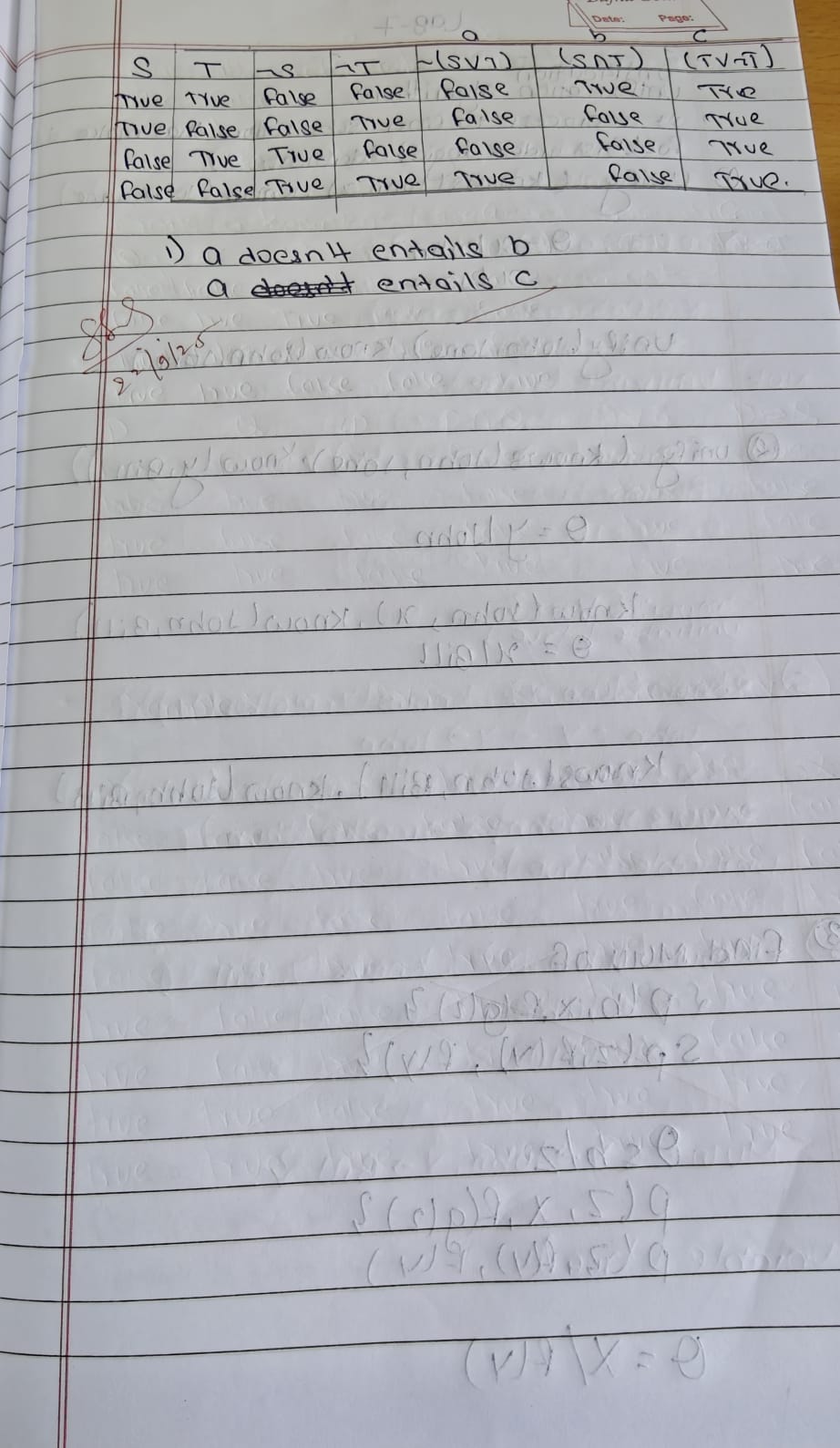
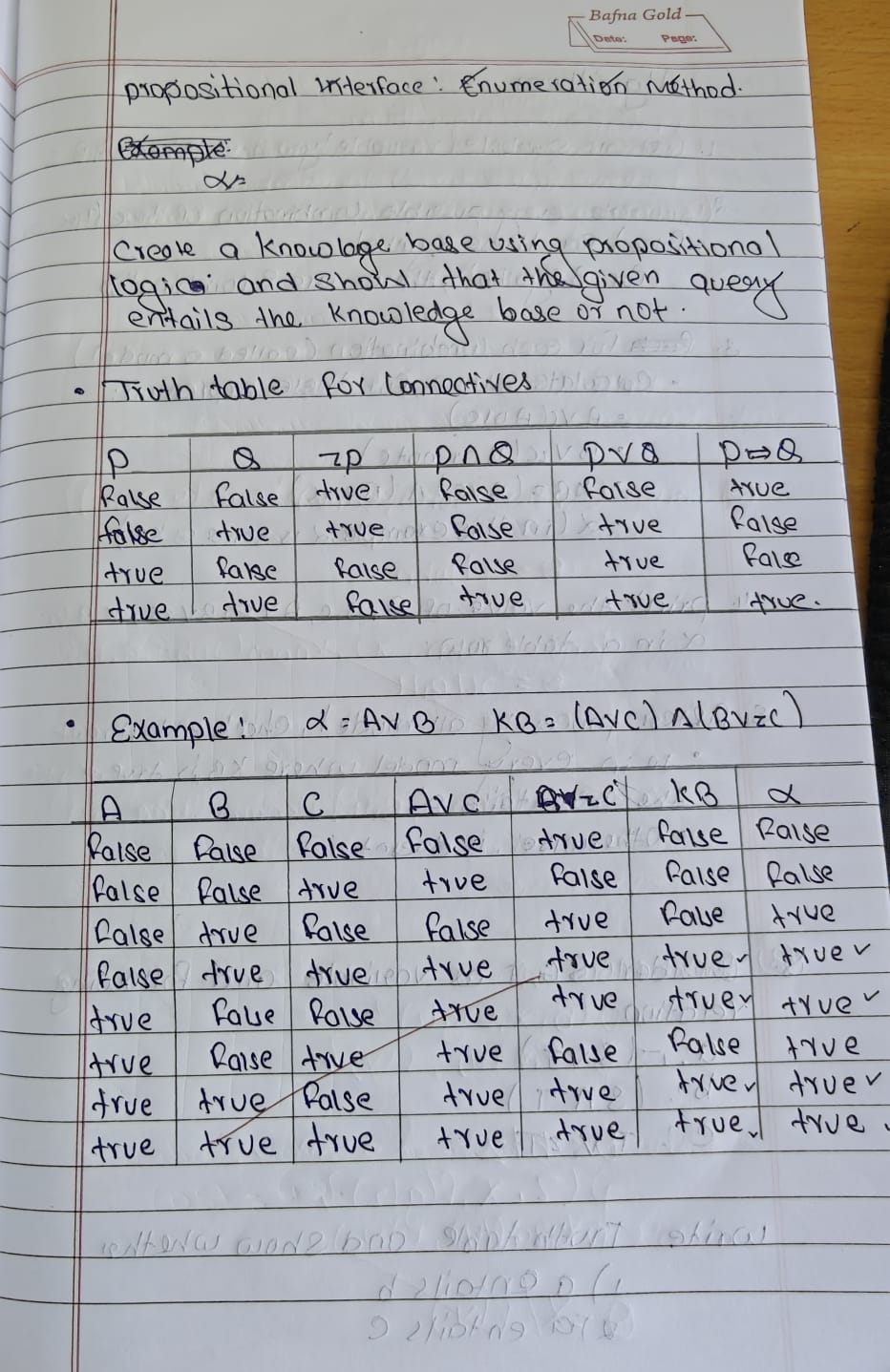
n = 8

solution, non\_attacking\_queens = simulated\_annealing(n)

print(f"The best position found is: {solution}")

print(f"The number of queens that are not attacking each other is: {non\_attacking\_queens}")

**PROGRAM-6**



print("Soham Hathi 1BM23CS335")

from itertools import product

def pl\_true(sentence, model):

if sentence == "A":

return model["A"]

elif sentence == "B":

return model["B"]

elif sentence == "C":

return model["C"]

elif sentence == "A\_or\_C":

return model["A"] or model["C"]

elif sentence == "B\_or\_not\_C":

return model["B"] or (not model["C"])

elif sentence == "KB":

return (model["A"] or model["C"]) and (model["B"] or (not model["C"]))

elif sentence == "alpha":

return model["alpha"]

else:

return False

def print\_truth\_table():

print(f"{'A':<7} {'B':<7} {'C':<7} {'A∨C':<7} {'B∨¬C':<7} {'KB':<7} {'α':<7}")

print("-"\*50)

all\_rows = []

for values in product([False, True], repeat=3):

model = {"A": values[0], "B": values[1], "C": values[2]}

model["A\_or\_C"] = pl\_true("A\_or\_C", model)

model["B\_or\_not\_C"] = pl\_true("B\_or\_not\_C", model)

model["KB"] = pl\_true("KB", model)

model["alpha"] = model["KB"]

all\_rows.append(model)

print(f"{str(model['A']).lower():<7} {str(model['B']).lower():<7} {str(model['C']).lower():<7} "

f"{str(model['A\_or\_C']).lower():<7} {str(model['B\_or\_not\_C']).lower():<7} "

f"{str(model['KB']).lower():<7} {str(model['alpha']).lower():<7}")

return all\_rows

def tt\_entails(all\_rows):

for model in all\_rows:

if model["KB"] and not model["alpha"]:

return False

return True

def main():

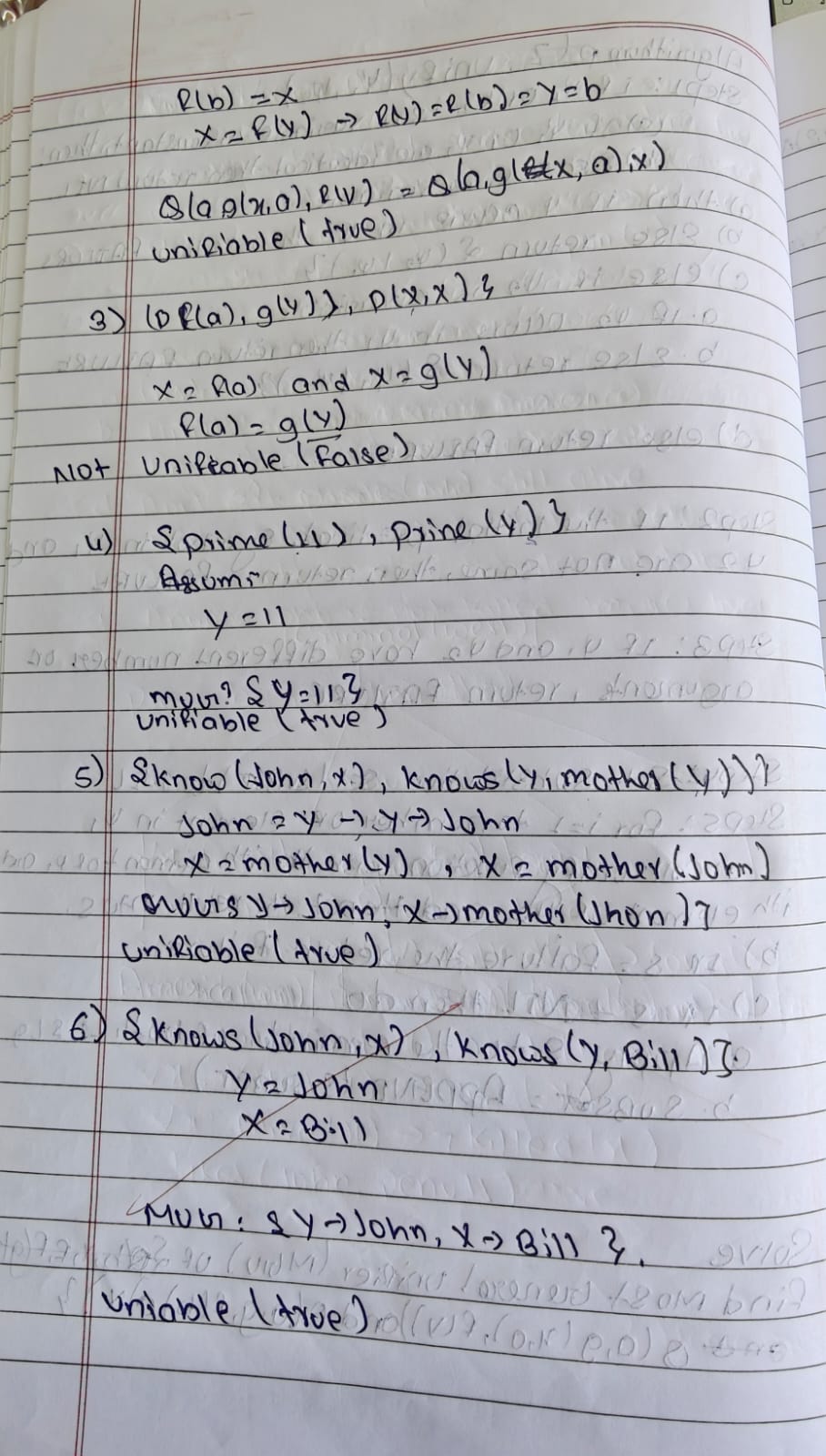
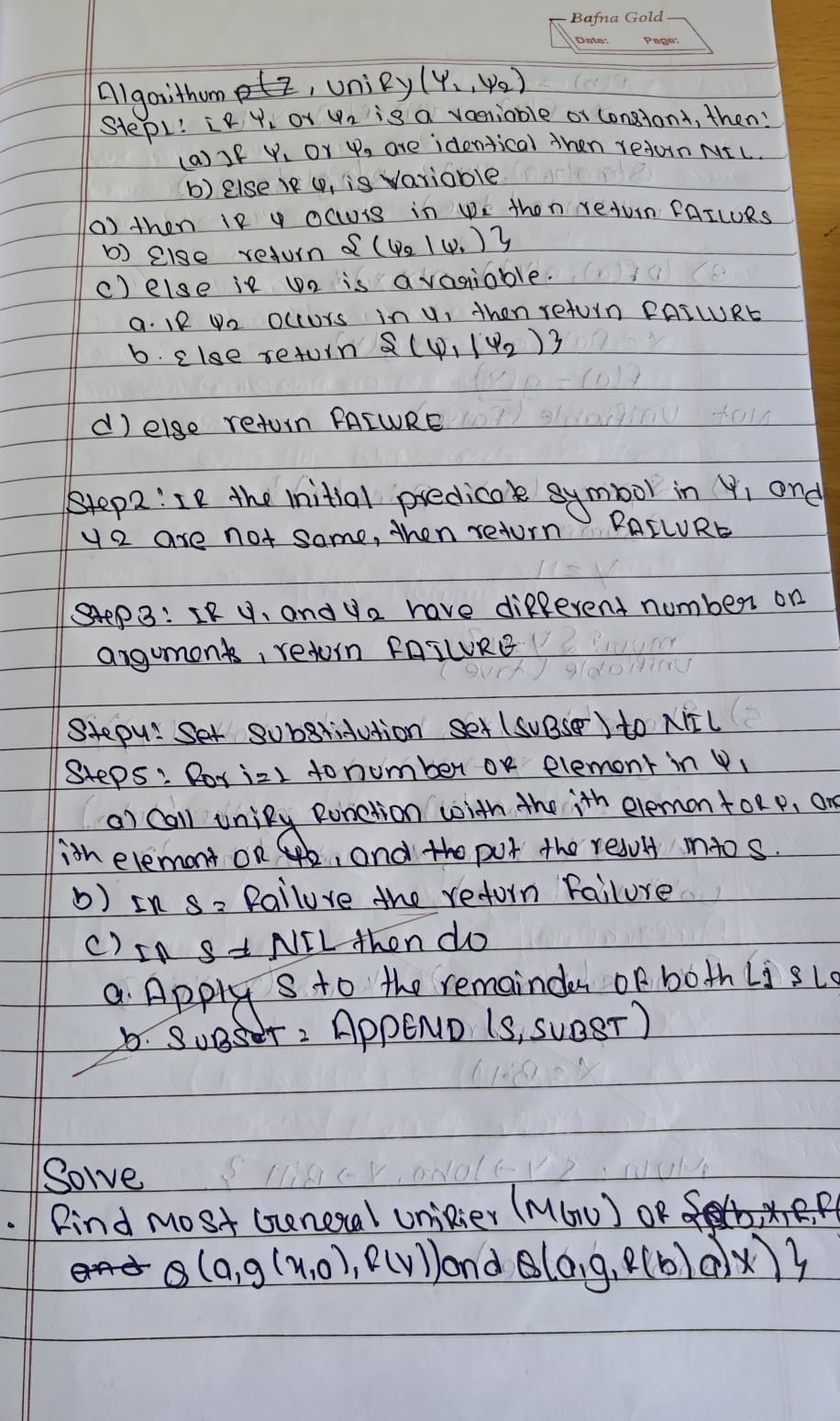
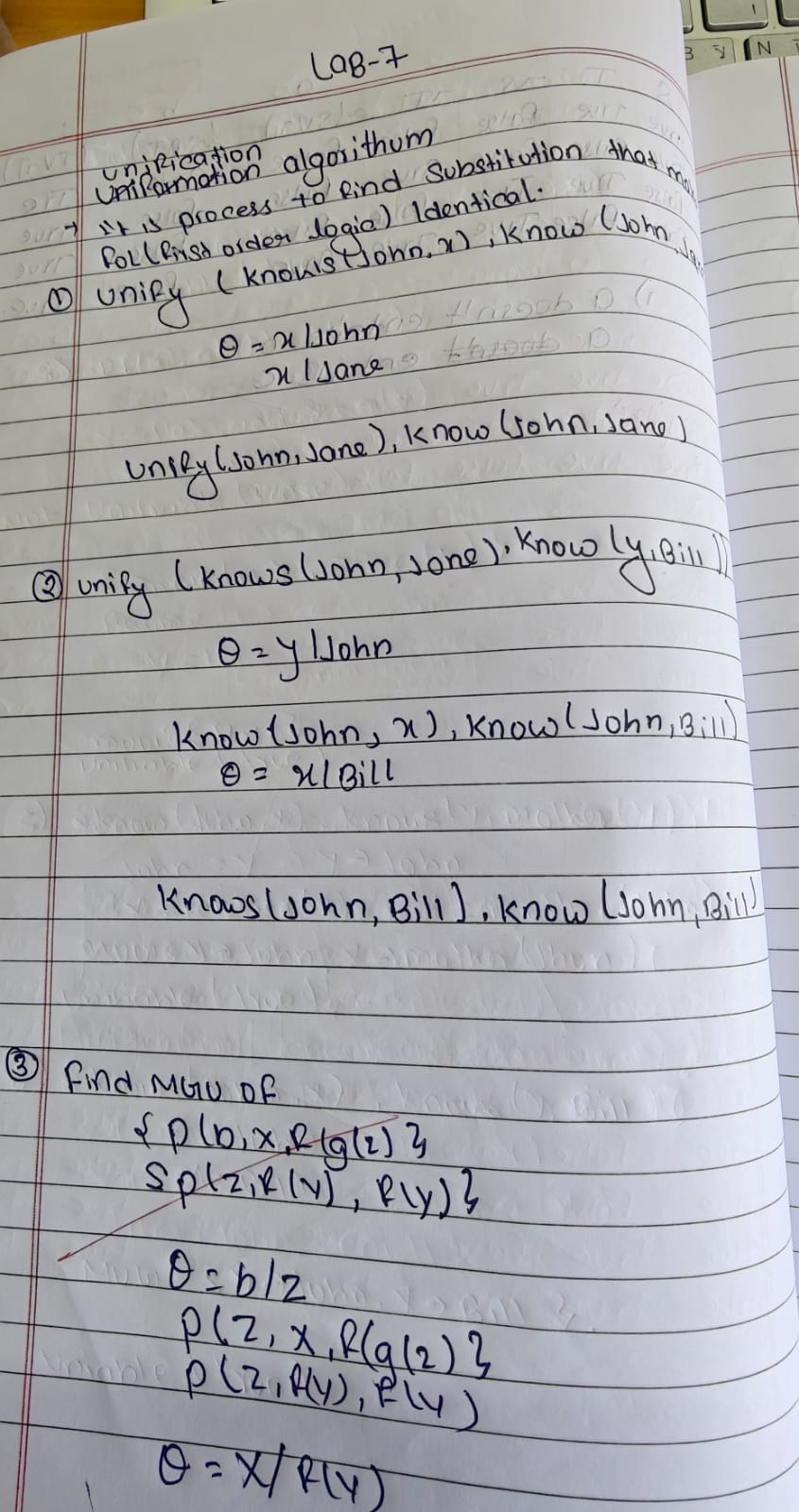
all\_rows = print\_truth\_table()

entails = tt\_entails(all\_rows)

print("\nDoes KB entail α? ", entails)

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

**PROGRAM-7**



# Unification algorithm (fixed parser & example)

import re

def is\_variable(x):

"""Variable if it is a string and starts with an uppercase letter."""

return isinstance(x, str) and len(x) > 0 and x[0].isupper()

def unify(x, y, subs=None):

if subs is None:

subs = {}

if x == y:

return subs

if is\_variable(x):

return unify\_var(x, y, subs)

if is\_variable(y):

return unify\_var(y, x, 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(apply\_subs(a, subs), apply\_subs(b, subs), subs)

if subs is None:

return None

return subs

return None

def unify\_var(var, x, subs):

if var in subs:

return unify(subs[var], x, subs)

elif is\_variable(x) and x in subs:

return unify(var, subs[x], subs)

elif occur\_check(var, x, subs):

return None

else:

subs[var] = x

return subs

def occur\_check(var, x, subs):

if var == x:

return True

elif isinstance(x, tuple):

return any(occur\_check(var, arg, subs) for arg in x[1])

elif isinstance(x, str) and is\_variable(x) and x in subs:

return occur\_check(var, subs[x], subs)

return False

def apply\_subs(term, subs):

if isinstance(term, str) and is\_variable(term) and term in subs:

return apply\_subs(subs[term], subs)

elif isinstance(term, tuple):

return (term[0], [apply\_subs(t, subs) for t in term[1]])

return term

def parse\_term(expr):

expr = expr.strip()

# strip outer braces if present, e.g. "{ ... }"

if expr.startswith('{') and expr.endswith('}'):

expr = expr[1:-1].strip()

if '(' not in expr:

return expr

functor = expr[:expr.index('(')].strip()

args = expr[expr.index('(')+1:-1]

return (functor, parse\_args(args))

def parse\_args(s):

args, level, current = [], 0, ''

for ch in s:

if ch == ',' and level == 0:

args.append(parse\_term(current.strip()))

current = ''

else:

if ch == '(':

level += 1

elif ch == ')':

level -= 1

current += ch

if current.strip():

args.append(parse\_term(current.strip()))

return args

expr1 = "prime(11)"

expr2 = "prime(Y)" # NOTE: Y is uppercase -> variable

term1 = parse\_term(expr1)

term2 = parse\_term(expr2)

result = unify(term1, term2)

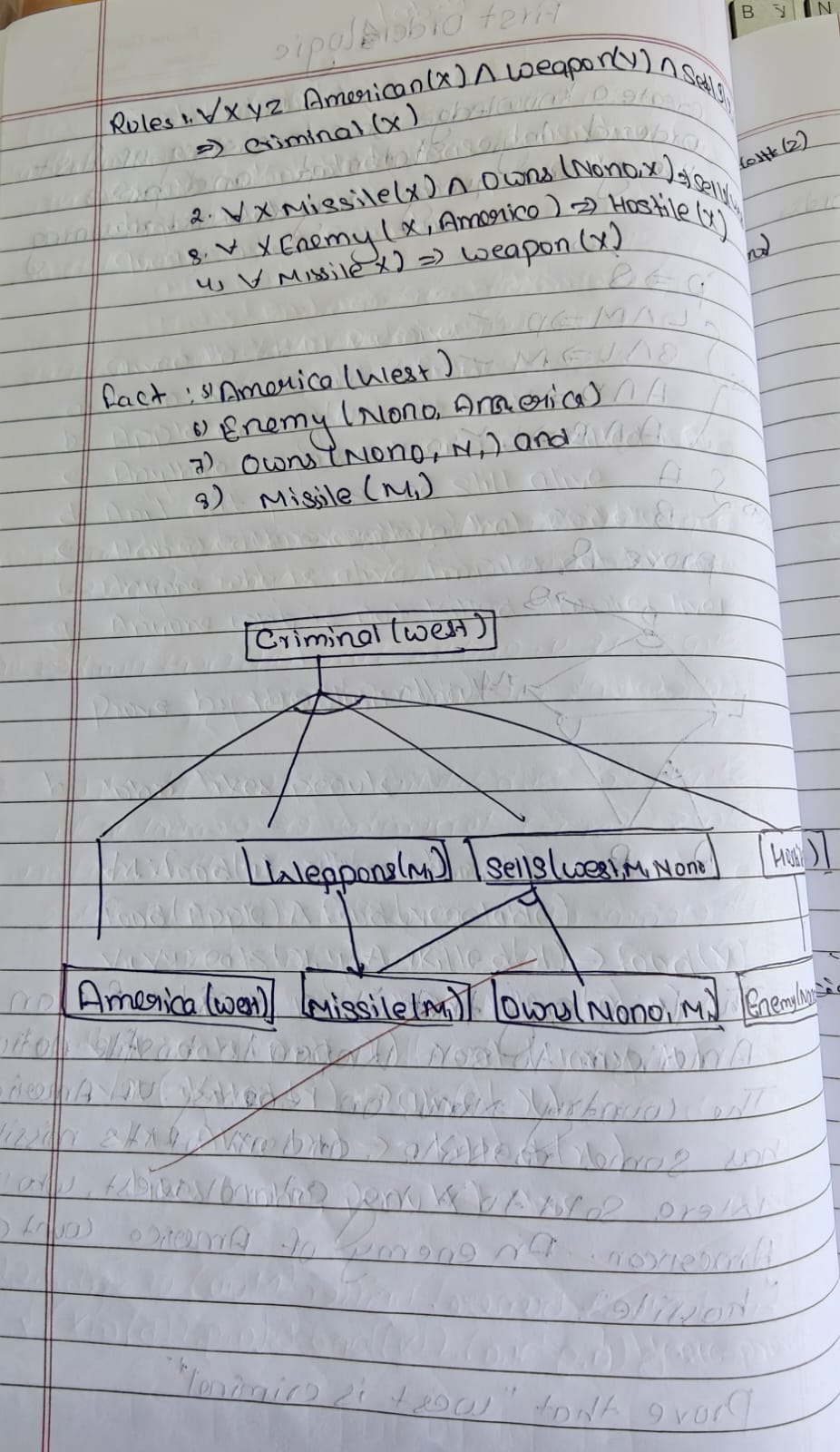
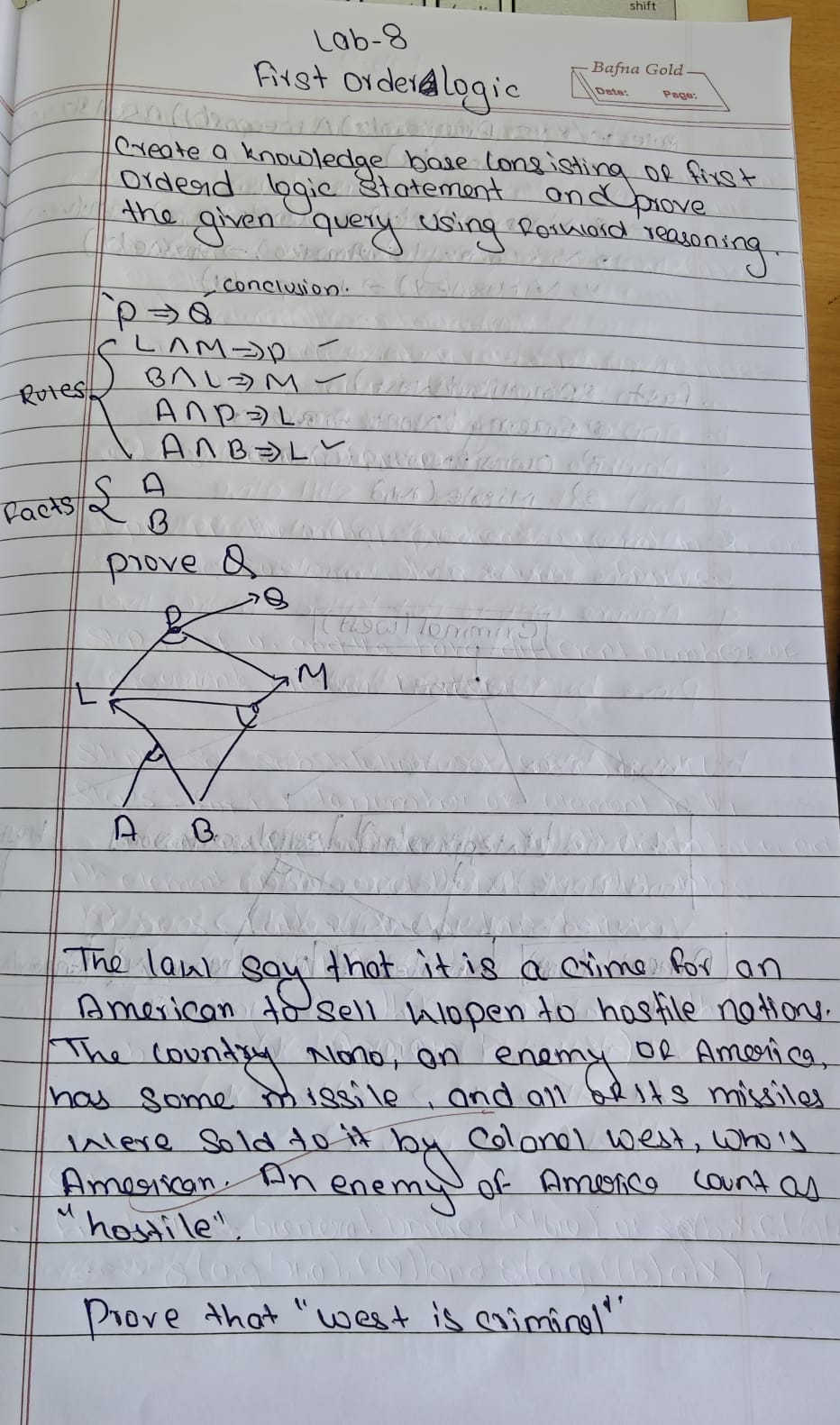
if result:

print("MGU:", result)

else:

print("No unifier found.")

**PROGRAM 8**



import copy

from itertools import product

print("Soham Hathi")

print("1BM23CS335")

def is\_variable(x):

return isinstance(x, str) and x[0].islower()

def substitute(expr, subs):

if isinstance(expr, list):

return [substitute(e, subs) for e in expr]

else:

return subs.get(expr, expr)

def unify(x, y, subs=None):

if subs is None:

subs = {}

if subs is False:

return False

elif x == y:

return subs

elif is\_variable(x):

return unify\_var(x, y, subs)

elif is\_variable(y):

return unify\_var(y, x, subs)

elif isinstance(x, list) and isinstance(y, list):

if len(x) != len(y):

return False

for xi, yi in zip(x, y):

subs = unify(xi, yi, subs)

if subs is False:

return False

return subs

else:

return False

def unify\_var(var, x, subs):

if var in subs:

return unify(subs[var], x, subs)

elif x in subs:

return unify(var, subs[x], subs)

else:

subs[var] = x

return subs

def forward\_chain(KB, query):

facts = [list(f) for f in KB['facts']]

added = True

while added:

added = False

for premises, conclusion in KB['rules']:

all\_matches = []

for premise in premises:

matches = []

for fact in facts:

subs = unify(premise, fact)

if subs:

matches.append(subs)

all\_matches.append(matches)

for combo in product(\*all\_matches):

merged = {}

for subs in combo:

merged.update(subs)

inferred = substitute(conclusion, merged)

if inferred not in facts:

facts.append(inferred)

added = True

if inferred == query:

return True

return query in facts

KB = {

'facts': [

["American", "West"],

["Weapon", "Missile"],

["Sells", "West", "Missile", "Nono"],

["Enemy", "Nono", "America"]

],

'rules': [

# Rule 1: Enemy of America → Hostile

([[ "Enemy", "x", "America" ]], [ "Hostile", "x" ]),

# Rule 2: American sells weapon to hostile nation → Criminal

([[ "American", "x" ], [ "Weapon", "y" ], [ "Sells", "x", "y", "z" ], [ "Hostile", "z" ]],

[ "Criminal", "x" ])

]

}

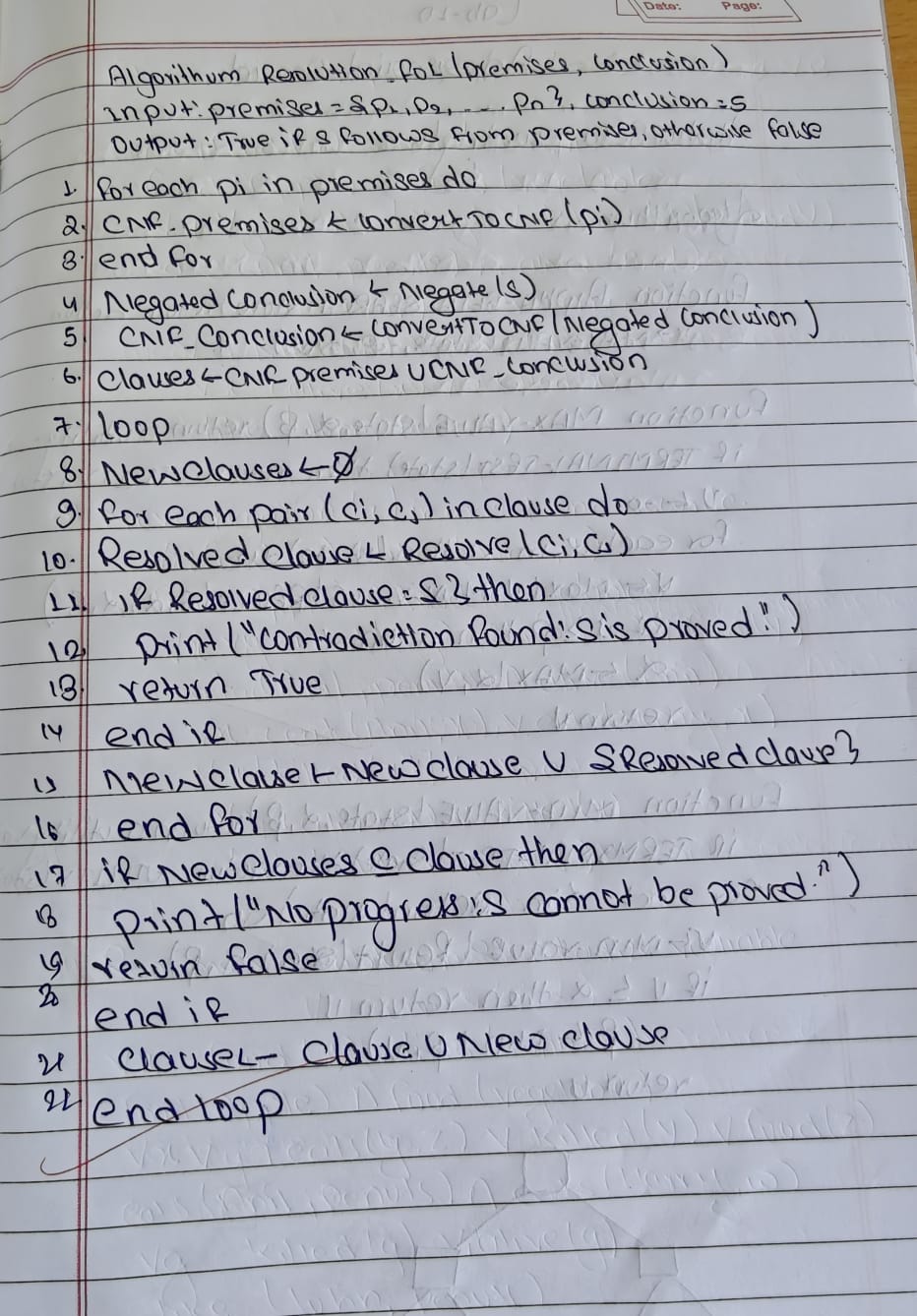
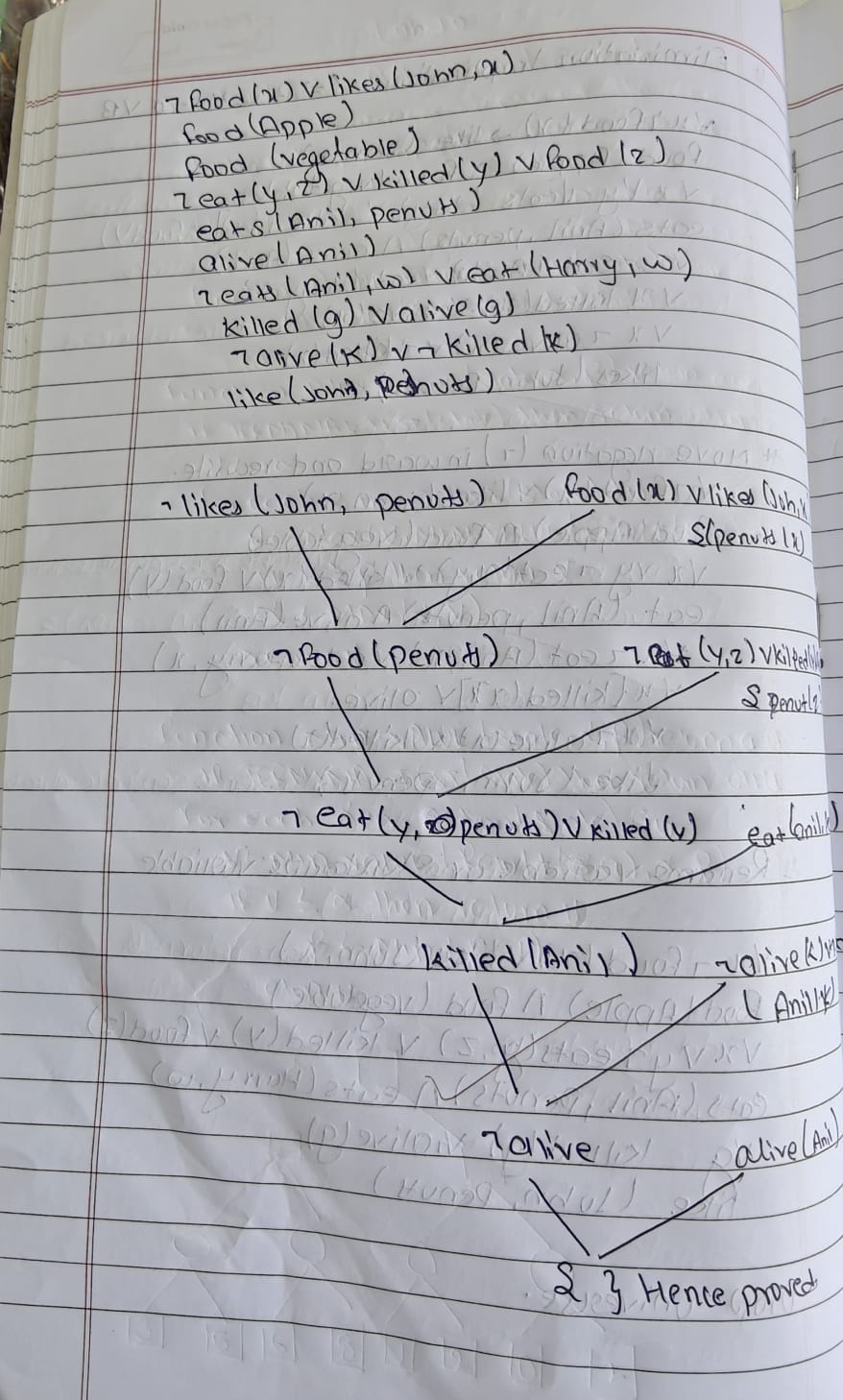
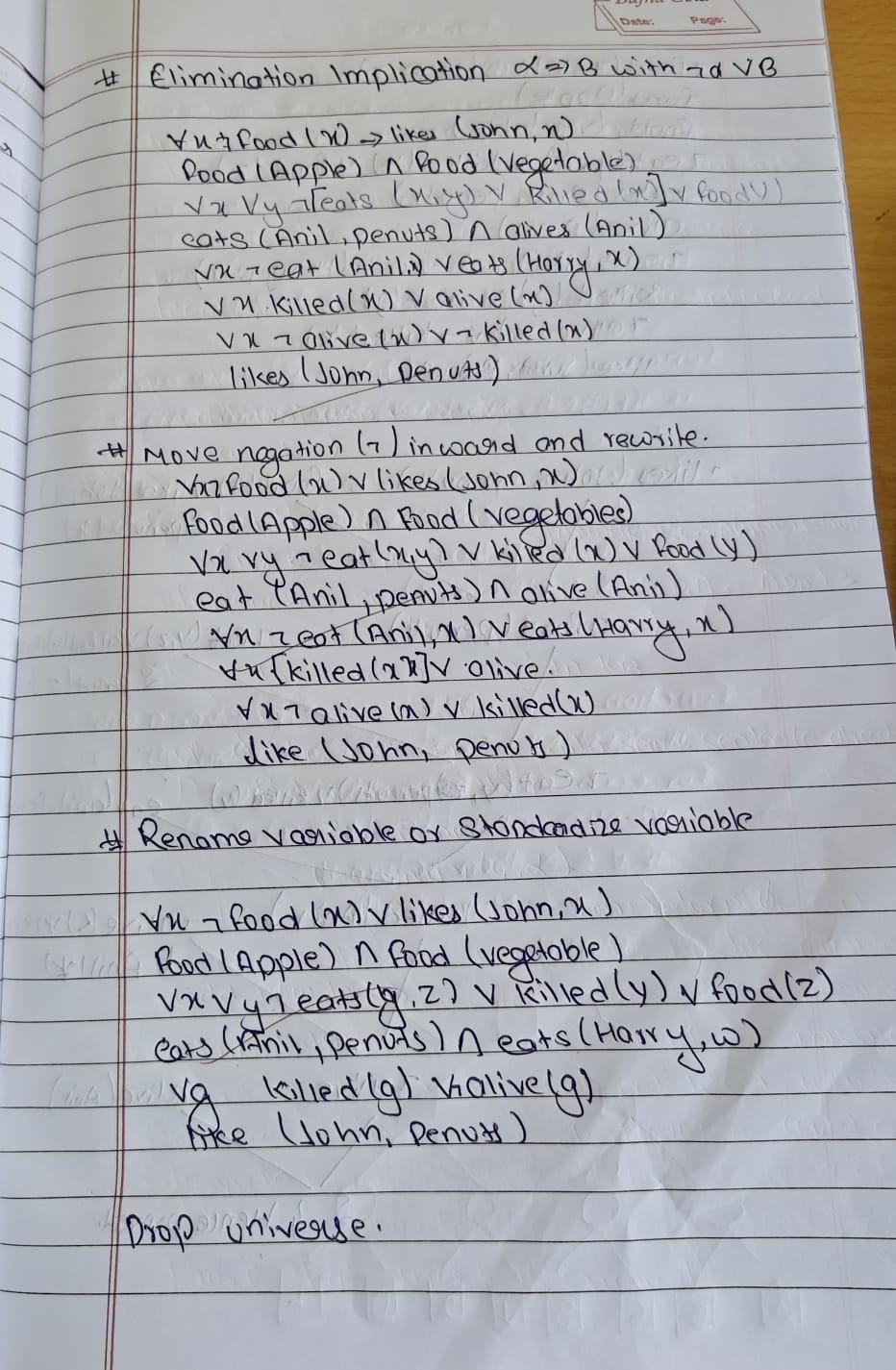
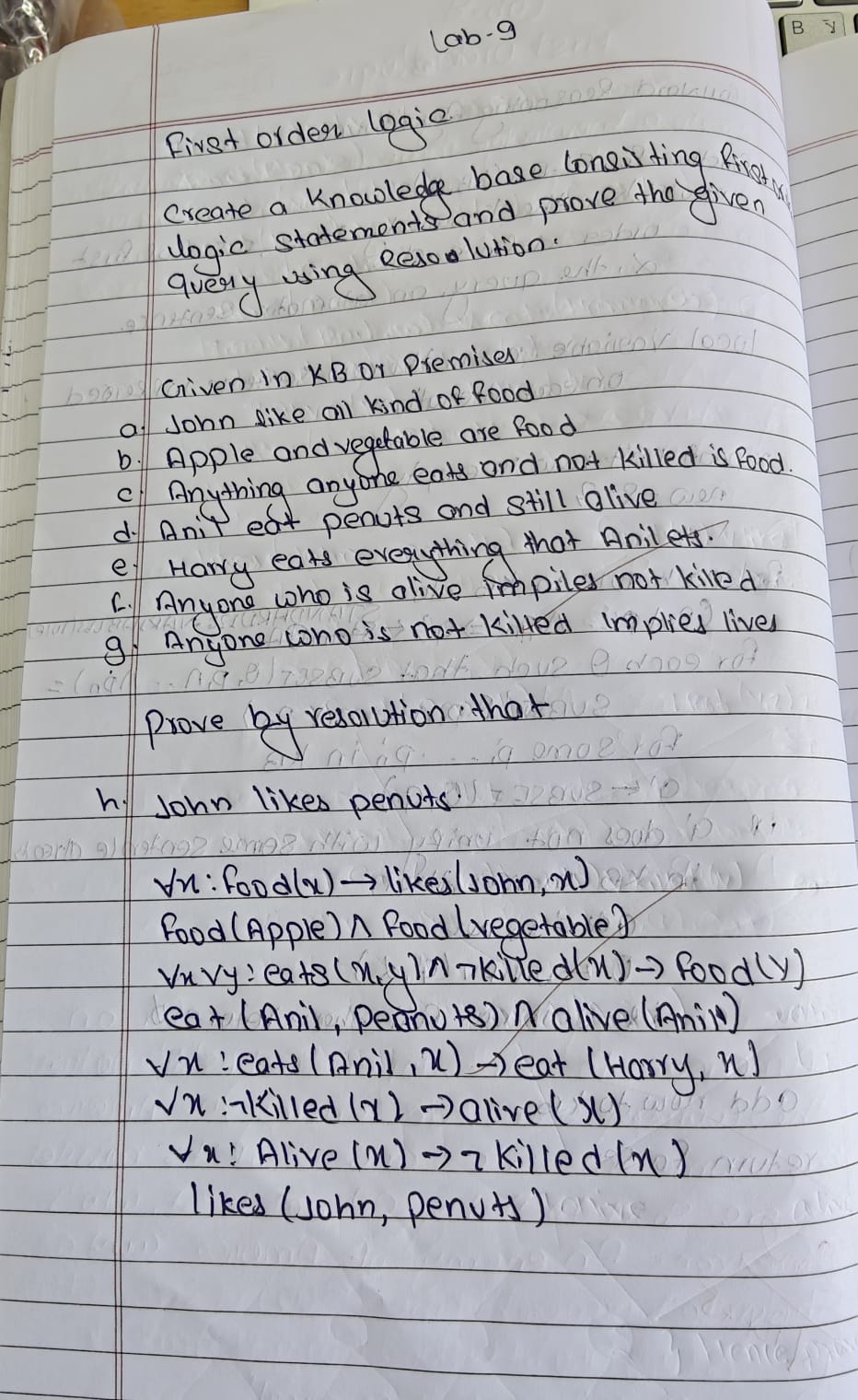
query = ["Criminal", "West"]

result = forward\_chain(copy.deepcopy(KB), query)

print("\nQuery:", query)

print("Result:", " TRUE" if result else "FALSE")

**PROGRAM-9**



print("Soham Hathi")

print("1BM23CS335")

from typing import List, Set

def substitute(clause, var, value):

"""Substitute a variable with a value in a clause."""

return {literal.replace(var, value) for literal in clause}

def unify(literal1: str, literal2: str):

"""Check if two literals can be unified and return substitution if possible."""

if literal1.startswith("~"):

l1\_pred = literal1[1:]

neg1 = True

else:

l1\_pred = literal1

neg1 = False

if literal2.startswith("~"):

l2\_pred = literal2[1:]

neg2 = True

else:

l2\_pred = literal2

neg2 = False

# Must have same predicate name

if l1\_pred.split("(")[0] != l2\_pred.split("(")[0]:

return None

# Must be opposite signs for resolution

if neg1 == neg2:

return None

args1 = l1\_pred[l1\_pred.find("(")+1:-1].split(",")

args2 = l2\_pred[l2\_pred.find("(")+1:-1].split(",")

if len(args1) != len(args2):

return None

substitution = {}

for a1, a2 in zip(args1, args2):

a1, a2 = a1.strip(), a2.strip()

# Variable (lowercase) can unify with constant (uppercase)

if a1[0].islower() and not a2[0].islower():

substitution[a1] = a2

elif a2[0].islower() and not a1[0].islower():

substitution[a2] = a1

elif a1 != a2:

return None

return substitution

def resolve(clause1: Set[str], clause2: Set[str]):

"""Resolve two clauses."""

resolvents = []

for lit1 in clause1:

for lit2 in clause2:

substitution = unify(lit1, lit2)

if substitution:

(var, val) = list(substitution.items())[0]

new\_clause1 = substitute(clause1 - {lit1}, var, val)

new\_clause2 = substitute(clause2 - {lit2}, var, val)

new\_clause = new\_clause1.union(new\_clause2)

resolvents.append((lit1, lit2, substitution, new\_clause))

return resolvents

def resolution(kb: List[Set[str]], query: Set[str]) -> bool:

"""Resolution algorithm with step-by-step printing."""

clauses = kb + [set(f"~{q}" for q in query)] # Negate query

step = 1

print("\n--- Resolution Steps ---")

while True:

n = len(clauses)

pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]

new = set()

for (ci, cj) in pairs:

resolvents = resolve(ci, cj)

for (lit1, lit2, substitution, result\_clause) in resolvents:

print(f"\nStep {step}:")

print(f"Resolving {ci} and {cj}")

print(f" - Unifying {lit1} with {lit2}")

print(f" - Substitution: {substitution}")

print(f" - Resulting clause: {result\_clause if result\_clause else '{}'}")

step += 1

if not result\_clause: # Empty clause => proved

print("\n❗ Contradiction found — query is proven true.")

return True

new.add(frozenset(result\_clause))

if new.issubset(set(map(frozenset, clauses))):

print("\nNo new clauses — query cannot be proven.")

return False

for c in new:

if c not in clauses:

clauses.append(set(c))

def main():

print("=== First Order Logic Resolution Prover ===")

n = int(input("Enter number of clauses in the knowledge base: "))

kb = []

print("\nEnter each clause separated by OR (|). Use '~' for NOT.\n")

for i in range(n):

clause\_str = input(f"Clause {i+1}: ").replace(" ", "")

literals = set(clause\_str.split("|"))

kb.append(literals)

query\_str = input("\nEnter query: ").replace(" ", "")

query = {query\_str}

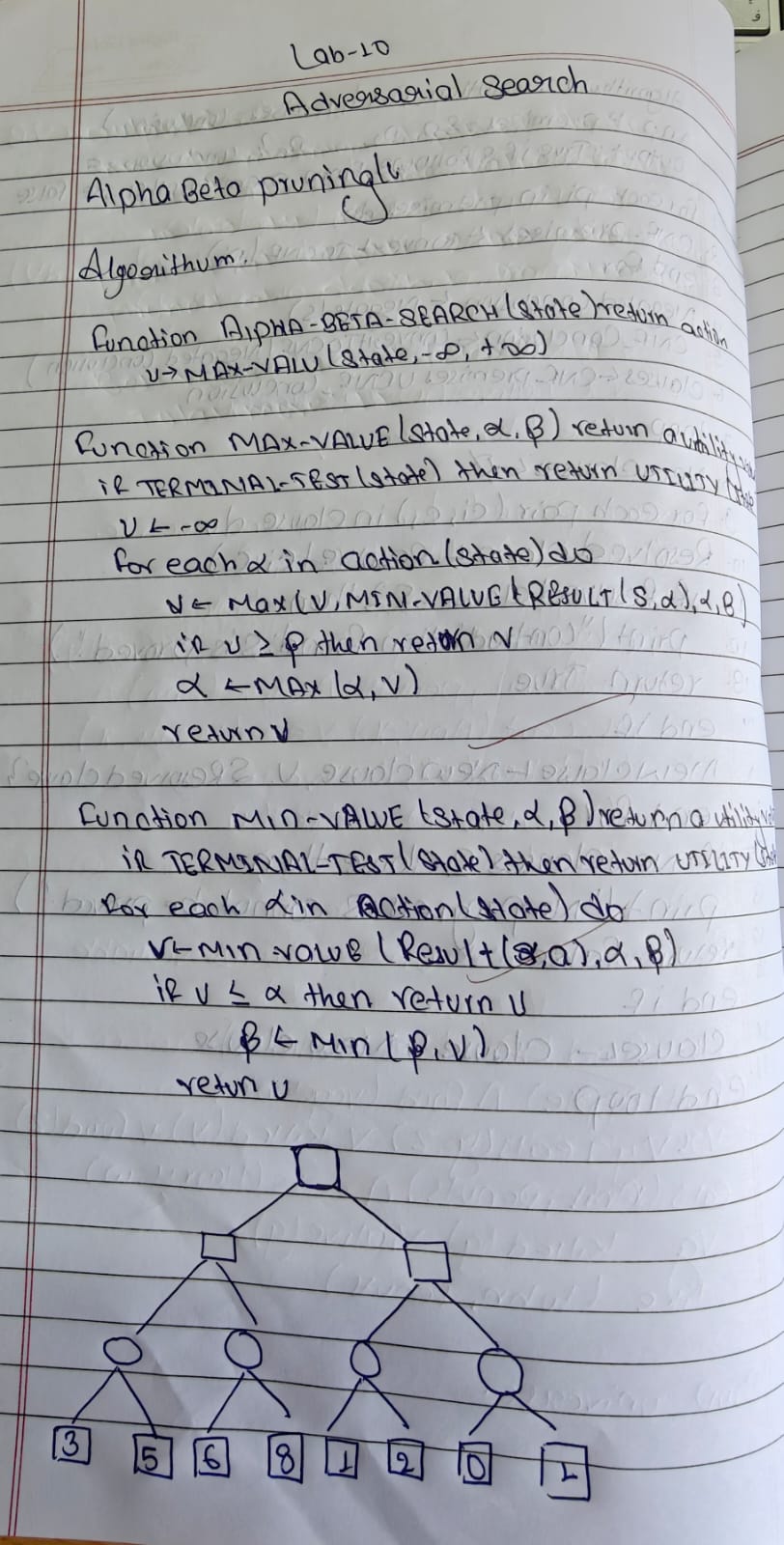
print("\n=== Resolution Process ===")

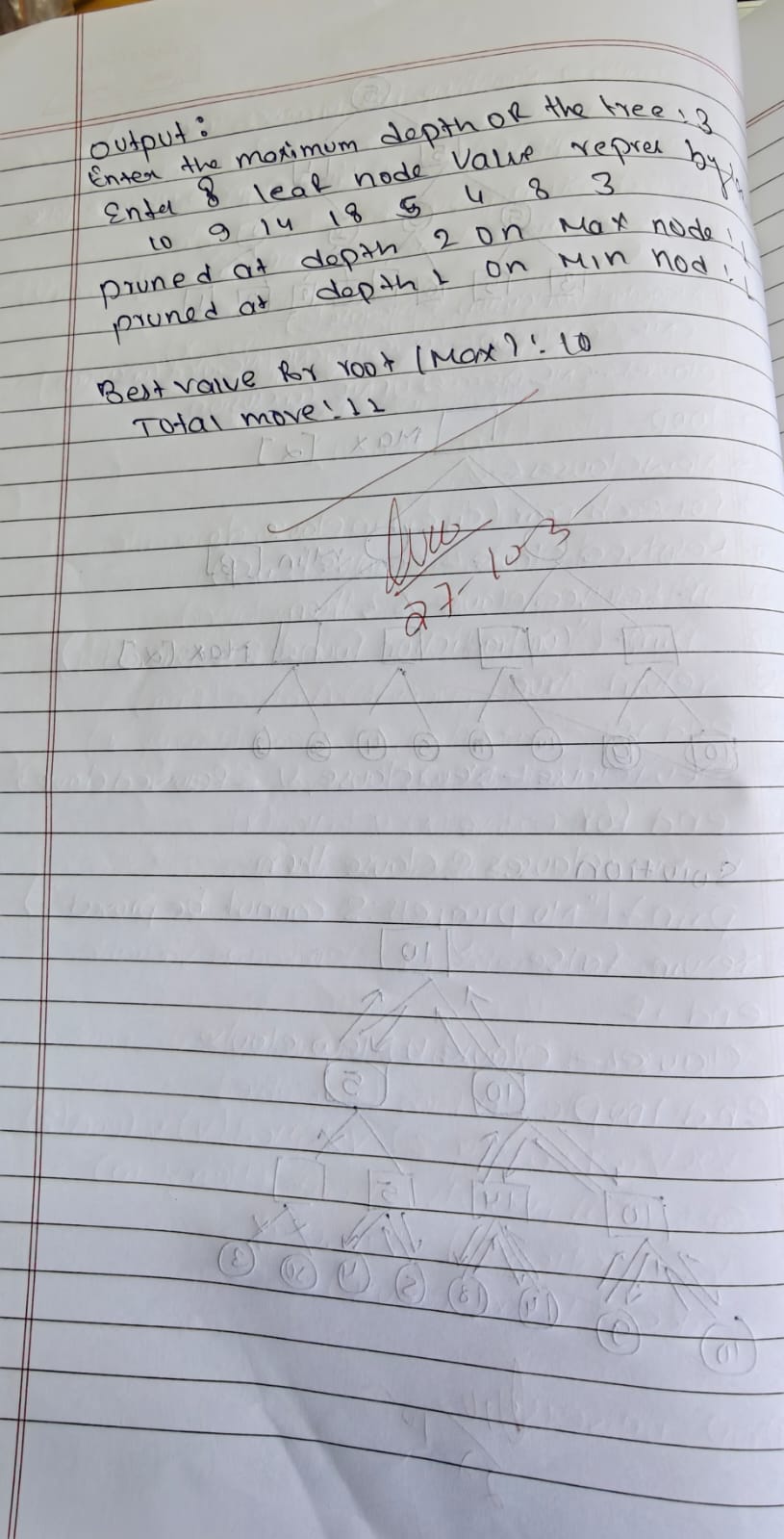
result = resolution(kb, query)

print("\n Final Result:", "Proved " if result else "Not Proved ")

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

**PROGRAM-10**



)

print("Soham Hathi")

print("1BM23CS335")

import math

# --- Alpha-Beta Search ---

def alpha\_beta\_search(values, max\_depth):

"""Alpha-Beta Search using standard pseudocode."""

tree = {} # to store computed node values

pruned = [] # to store pruned branches

def max\_value(depth, node\_index, alpha, beta):

indent = " " \* depth

if depth == max\_depth:

print(f"{indent}Leaf Node[{node\_index}] = {values[node\_index]}")

tree[(depth, node\_index)] = values[node\_index]

return values[node\_index]

print(f"{indent}MAX Node[{node\_index}] (α={alpha}, β={beta})")

v = -math.inf

for i in range(2):

child\_index = node\_index \* 2 + i

val = min\_value(depth + 1, child\_index, alpha, beta)

v = max(v, val)

alpha = max(alpha, v)

print(f"{indent}→ MAX[{node\_index}] updated to {v} (α={alpha}, β={beta})")

if v >= beta:

print(f"{indent}⛔ Beta cutoff at Node[{child\_index}]")

pruned.append((depth, node\_index, "β"))

break

tree[(depth, node\_index)] = v

return v

def min\_value(depth, node\_index, alpha, beta):

indent = " " \* depth

if depth == max\_depth:

print(f"{indent}Leaf Node[{node\_index}] = {values[node\_index]}")

tree[(depth, node\_index)] = values[node\_index]

return values[node\_index]

print(f"{indent}MIN Node[{node\_index}] (α={alpha}, β={beta})")

v = math.inf

for i in range(2):

child\_index = node\_index \* 2 + i

val = max\_value(depth + 1, child\_index, alpha, beta)

v = min(v, val)

beta = min(beta, v)

print(f"{indent}→ MIN[{node\_index}] updated to {v} (α={alpha}, β={beta})")

if v <= alpha:

print(f"{indent}⛔ Alpha cutoff at Node[{child\_index}]")

pruned.append((depth, node\_index, "α"))

break

tree[(depth, node\_index)] = v

return v

print("\n=== Alpha–Beta Search Steps ===")

best\_value = max\_value(0, 0, -math.inf, math.inf)

return best\_value, tree, pruned

# --- Function to Print Tree in ASCII ---

def print\_tree(values, tree, max\_depth):

print("\n=== TREE STRUCTURE ===")

index = 0

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

nodes = 2 \*\* depth

line = ""

for i in range(nodes):

if (depth, i) in tree:

val = tree[(depth, i)]

else:

val = "—"

line += f"[{val:^5}] "

print(" " \* (max\_depth - depth) + line)

print()

# --- Main Program ---

def main():

print("=== Alpha–Beta Search Algorithm (Text Tree Output) ===")

max\_depth = int(input("Enter maximum depth of tree (e.g., 3): "))

num\_leaves = 2 \*\* max\_depth

print(f"\nEnter {num\_leaves} leaf node values (space-separated):")

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

if len(values) != num\_leaves:

print(f"Error: Expected {num\_leaves} values for a complete binary tree.")

return

best\_value, tree, pruned = alpha\_beta\_search(values, max\_depth)

print\_tree(values, tree, max\_depth)

print("=== RESULTS ===")

print(f"Optimal root node value: {best\_value}")

if pruned:

print("Pruned branches:")

for p in pruned:

print(f" Node[{p[1]}] at depth {p[0]} ({p[2]} cutoff)")

else:

print("No pruning occurred.")

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

main()