**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



**LAB REPORT**

**on**

**Artificial Intelligence (23CS5PCAIN)**

***Submitted by***

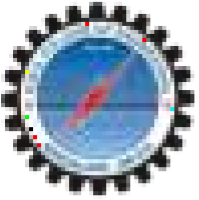
**Dhanush S (1BM23CS089)**

***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 **Dhanush S (1BM23CS089),** 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.Sheetal V L  Assistant Professor  Department of CSE, BMSCE | Dr. Kavitha Sooda  Professor & HOD  Department of CSE, BMSCE |

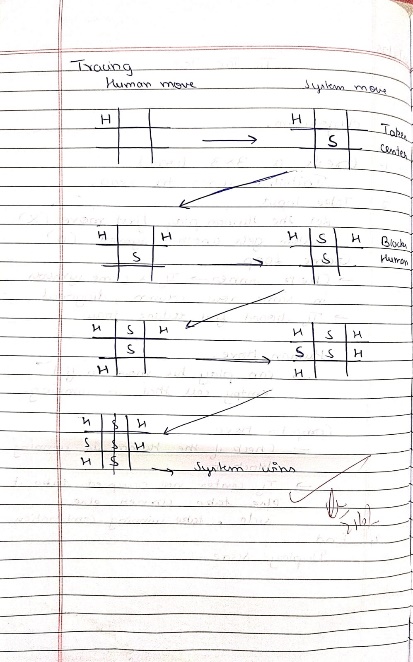
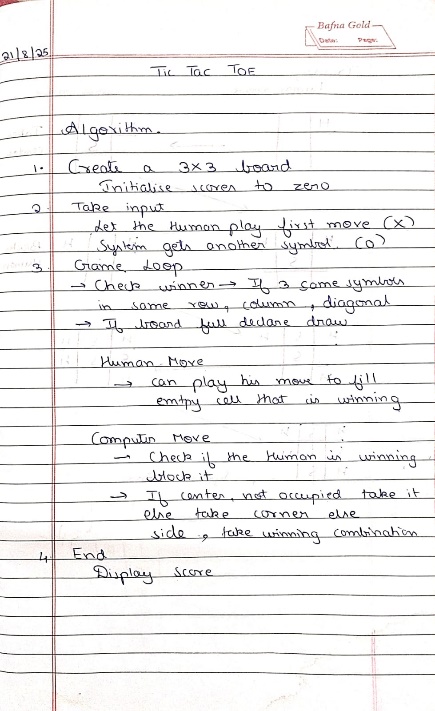
**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sl.**  **No.** | **Date** | **Experiment Title** | **Page No.** |
| 1 | 21-08-2025 | Implement Tic –Tac –Toe Game Implement vacuum cleaner agent | 4 |
| 2 | 28-08-2025 | Implement 8 puzzle problems using Depth First  Search (DFS)  Implement Iterative deepening search algorithm | 13 |
| 3 | 11-09-2025 | Implement A\* search algorithm | 19 |
| 4 | 09-10-2025 | Implement Hill Climbing search algorithm to solve N-Queens problem | 22 |
| 5 | 09-10-2025 | Simulated Annealing to Solve 8-Queens problem | 25 |
| 6 | 16-10-2025 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not | 28 |
| 7 | 30-10-2025 | Implement unification in first order logic | 31 |
| 8 | 06-11-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning | 34 |
| 9 | 06-11-2025 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 36 |
| 10 | 13-11-2025 | Implement Alpha-Beta Pruning | 44 |
| 11 | 13-11-2025 | Conversion to CNF | 46 |

Github Link: <https://github.com/DhanushSGouda/AI-LAB_1BM23CS089>

**Program 1**

Implement Tic –Tac –Toe Game Algorithm:



Code: 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))

cost\_counter = 0

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

global cost\_counter

cost\_counter += 1

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!")

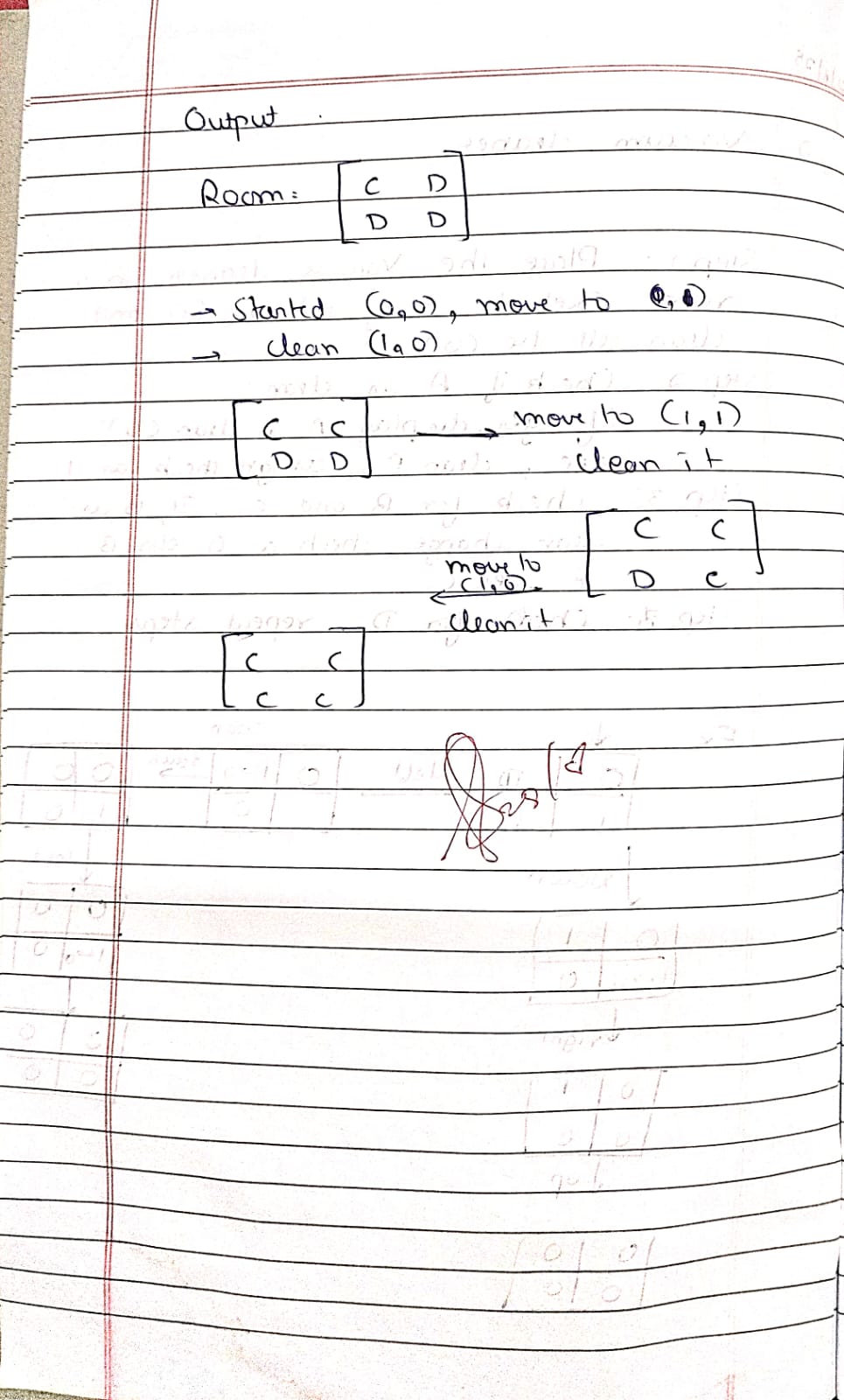
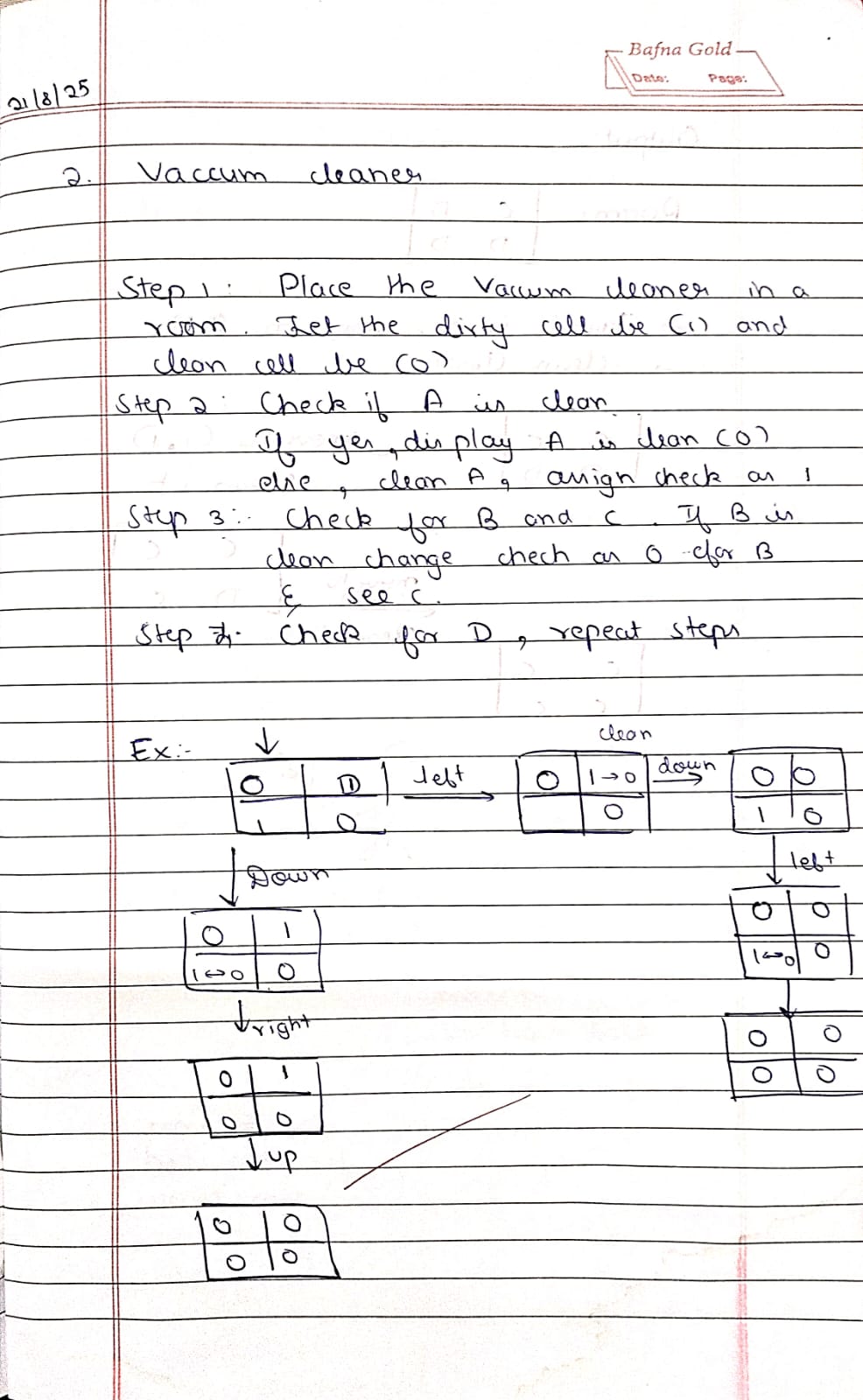
break

global cost\_counter cost\_counter = 0 cost = minimax(board, True) print(f"AI evaluation cost from this position: {cost\_counter} states examined") print(f"AI evaluation score from this position: {cost}")

manual\_game()

Implement vacuum cleaner agent.

Algorithm:



Code: rooms = {

'A': True,

'B': True,

'C': True,

'D': True

}

# The agent's current location current\_room = 'A' def vacuum\_cleaner\_agent(): global current\_room print("---Starting Vacuum Cleaner Agent---") print("Initial state:", rooms) print("Agent starts in room A.") # A set to track visited rooms to avoid loops visited = set()

# While there's any dirty room left while any(rooms.values()):

# Clean the current room if dirty if rooms[current\_room]:

print(f"\nSucking dust in room {current\_room}...") rooms[current\_room] = False

print(f"Room {current\_room} is now clean.") visited.add(current\_room)

# Decide where to go next based on current location and available dirty rooms next\_room = None if current\_room == 'A':

options = [room for room in ['B', 'C'] if rooms[room] and room not in visited]

if options: while True:

user\_choice = input(f"Do you want to go to room {options[0]} or room {options[-1]}?

(Type '{options[0]}' or '{options[-1]}'): ").upper() if user\_choice in options: next\_room = user\_choice

break else:

print("Invalid input. Please choose a valid dirty room.")

else:

# Default to B or C if no input needed for room in ['B', 'C']: if rooms[room] and room not in visited:

next\_room = room break

elif current\_room == 'B': if rooms['D'] and 'D' not in visited:

print("Moving to room D.") next\_room = 'D'

elif rooms['A'] and 'A' not in visited:

next\_room = 'A' elif current\_room == 'C': if rooms['D'] and 'D' not in visited: print("Moving to room D.") next\_room = 'D' elif rooms['A'] and 'A' not in visited:

next\_room = 'A' elif current\_room == 'D': if rooms['C'] and 'C' not in visited: print("Moving to room C.") next\_room = 'C' elif rooms['B'] and 'B' not in visited:

next\_room = 'B'

# Fallback: find any remaining dirty room not visited yet if not next\_room:

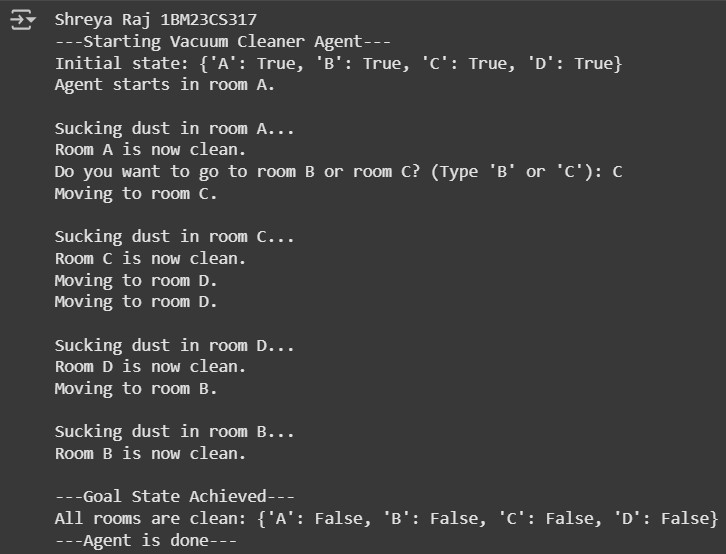
for room in ['A', 'B', 'C', 'D']: if rooms[room] and room not in visited:

next\_room = room break if next\_room:

print(f"Moving to room {next\_room}.") current\_room = next\_room else:

# No dirty unvisited rooms left break print("\n---Goal State Achieved---") print("All rooms are clean:", rooms) print("---Agent is done---") vacuum\_cleaner\_agent()

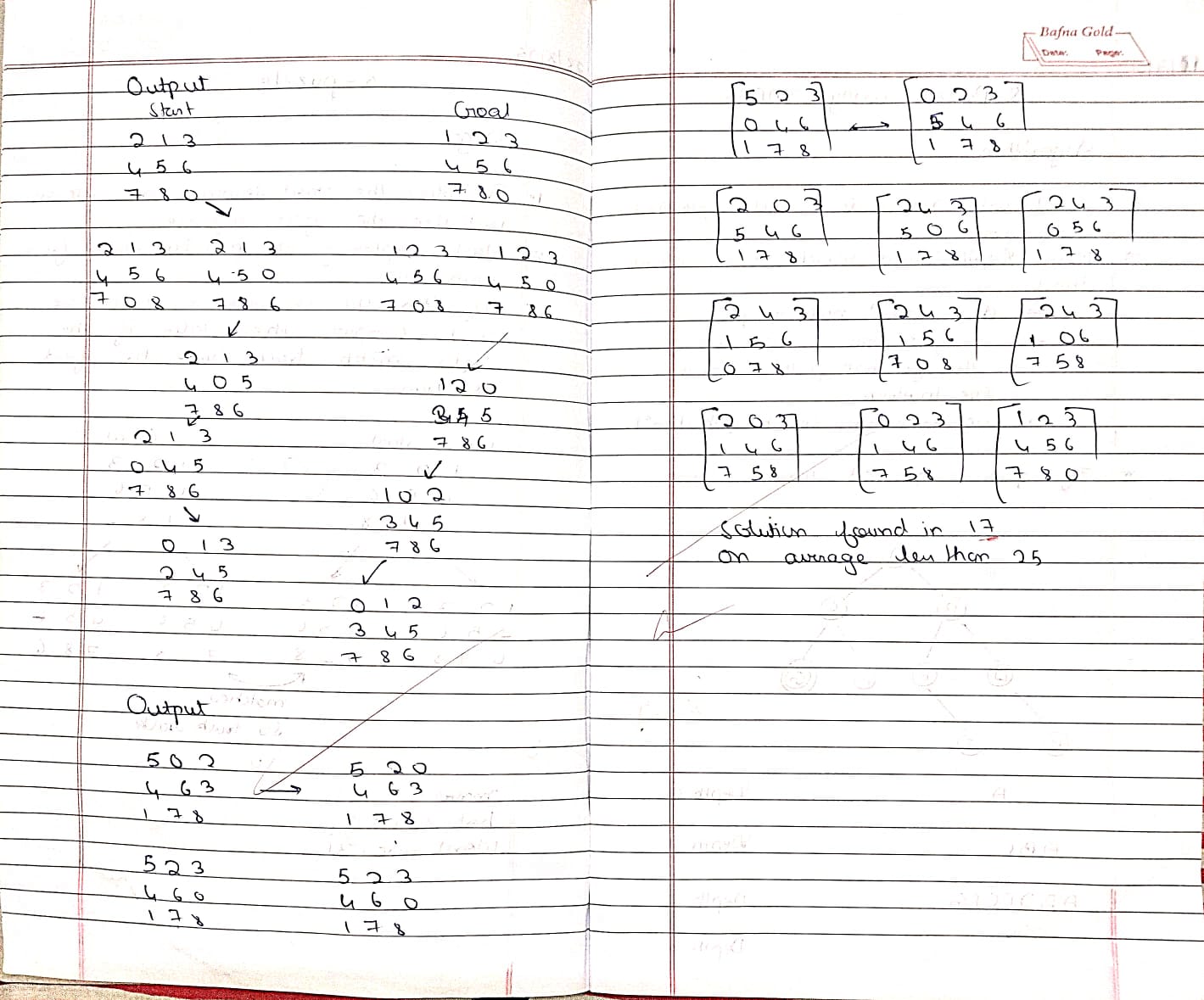
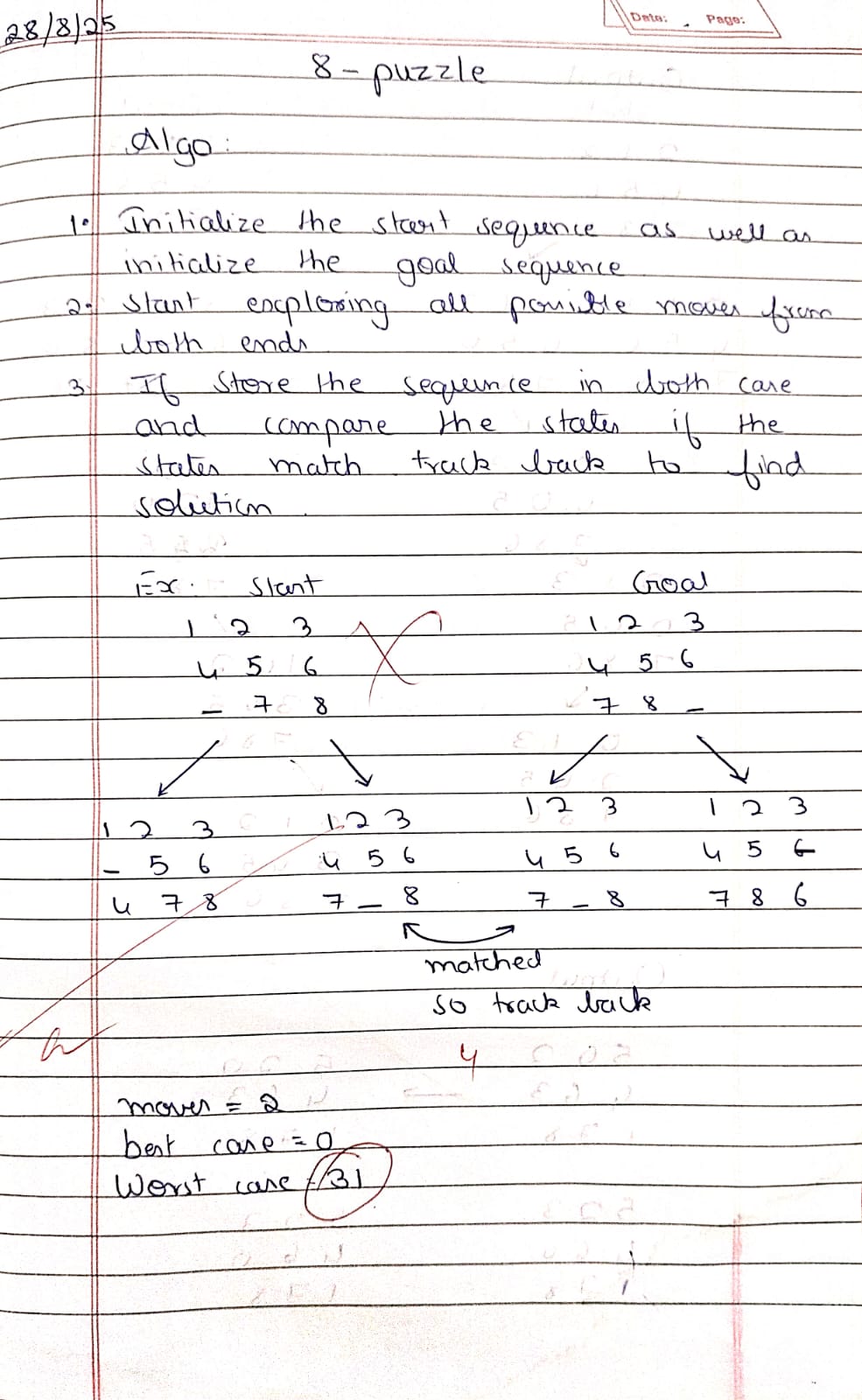
Outputs:



**Program 2**

Using BFS solve 8 puzzle without heuristic approach.

Algorithm:



Code:

from collections import deque

def print\_state(state): for i in range(0, 9, 3): print(state[i:i+3]) print()

def bfs(start, goal): queue = deque([(start, [])]) visited = set([start]) while queue:

state, path = queue.popleft() if state == goal:

return path + [state] zero = state.index(0) moves = [] if zero % 3 > 0:

moves.append(zero - 1) if zero % 3 < 2:

moves.append(zero + 1) if zero // 3 > 0:

moves.append(zero - 3) if zero // 3 < 2:

moves.append(zero + 3) for move in moves:

new\_state = list(state) new\_state[zero], new\_state[move] = new\_state[move], new\_state[zero] new\_state = tuple(new\_state) if new\_state not in visited: visited.add(new\_state) queue.append((new\_state, path + [state])) return None

def input\_state(prompt): s = input(prompt).strip().split()

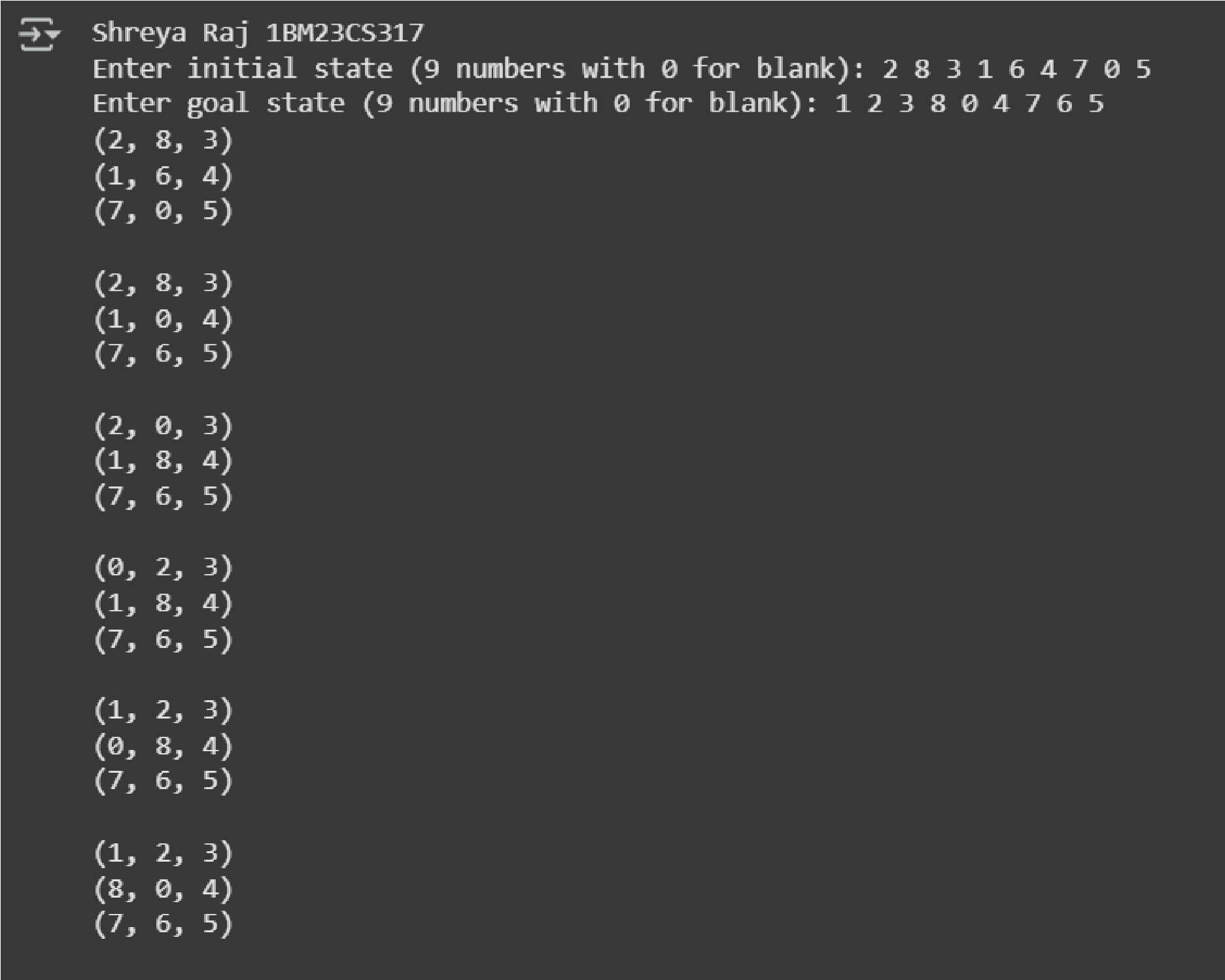
return tuple(map(int, s))

start = input\_state("Enter initial state (9 numbers with 0 for blank): ") goal = input\_state("Enter goal state (9 numbers with 0 for blank): ")

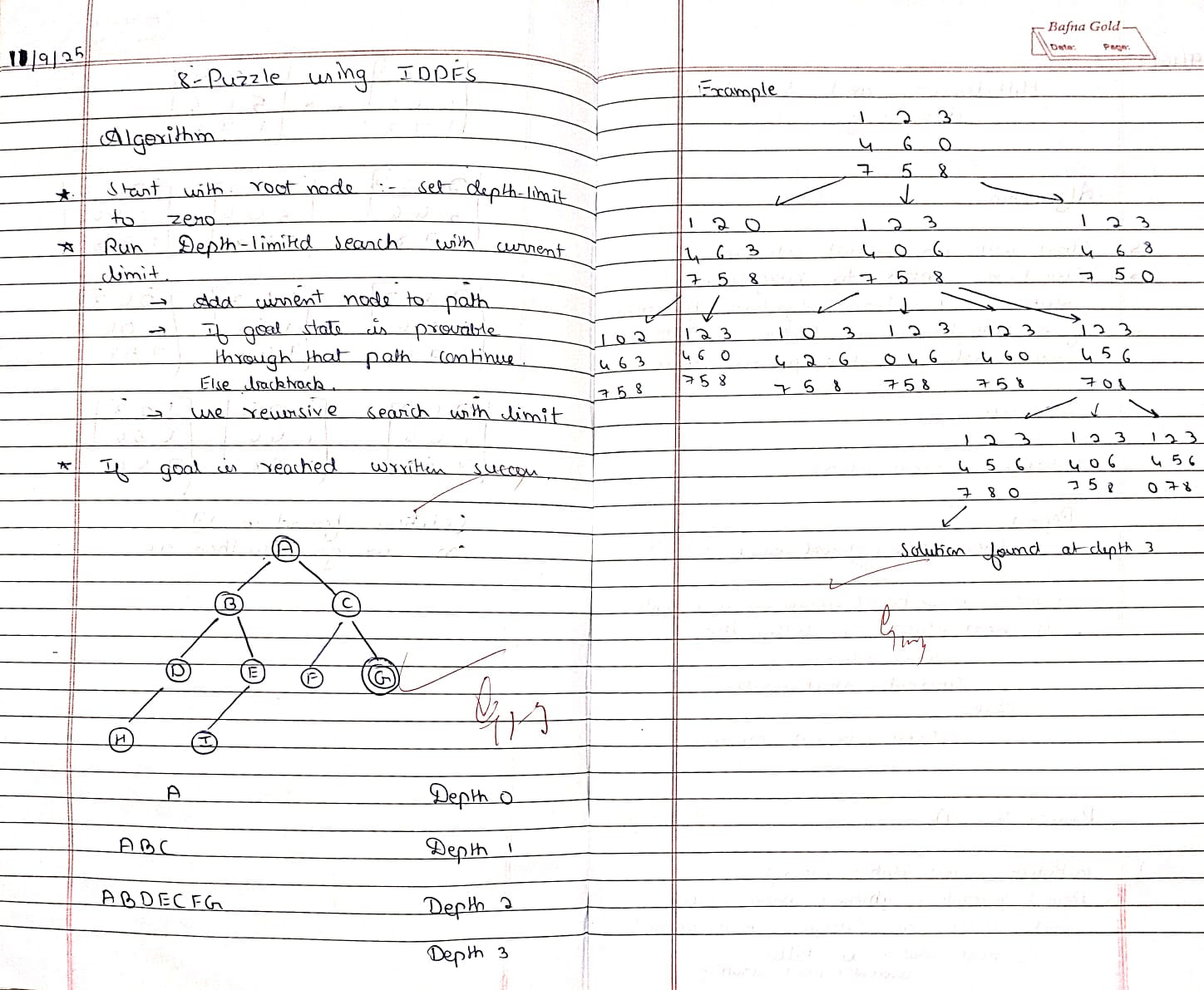
result = bfs(start, goal) if result: for step in result: print\_state(step) else:

print("No solution found")

Output:



Using Iterative Deepening DFS solve 8 puzzle without heuristic approach.



Code: class PuzzleState: def \_\_init\_\_(self, board, empty\_pos, moves=0, path=None):

self.board = board self.empty\_pos = empty\_pos self.moves = moves

self.path = path or [board]

def is\_goal(self, goal):

return self.board == goal

def get\_neighbors(self): neighbors = [] x, y = self.empty\_pos directions = [(-1,0),(1,0),(0,-1),(0,1)] # Up, Down, Left, Right for dx, dy in directions: nx, ny = x + dx, y + dy if 0 <= nx < 3 and 0 <= ny < 3: new\_board = [list(row) for row in self.board]

# swap empty\_pos with target

new\_board[x][y], new\_board[nx][ny] = new\_board[nx][ny], new\_board[x][y] new\_board = tuple(tuple(row) for row in new\_board) neighbors.append(PuzzleState(new\_board, (nx, ny), self.moves + 1, self.path + [new\_board])) return neighbors

def dls(state, goal, limit, visited): if state.is\_goal(goal): return state.path if limit == 0: return None visited.add(state.board) for neighbor in state.get\_neighbors(): if neighbor.board not in visited:

result = dls(neighbor, goal, limit - 1, visited) if result is not None: return result visited.remove(state.board) return None

def iddfs(start, goal):

depth = 0 while True:

visited = set() result = dls(start, goal, depth, visited) if result is not None:

return result depth += 1

def print\_path(path):

print(f"Solution Found in {len(path)-1} moves") for state in path: for row in state:

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

def get\_user\_board(prompt):

print(prompt) board = [] for i in range(3):

row = list(map(int, input(f"Row {i+1} (space separated, use 0 for empty): ").strip().split())) board.append(tuple(row)) return tuple(board)

start\_board = get\_user\_board("Enter the initial state:") goal\_board = get\_user\_board("Enter the goal state:")

# Locate empty in start state empty\_pos = None for i in range(3): for j in range(3): if start\_board[i][j] == 0:

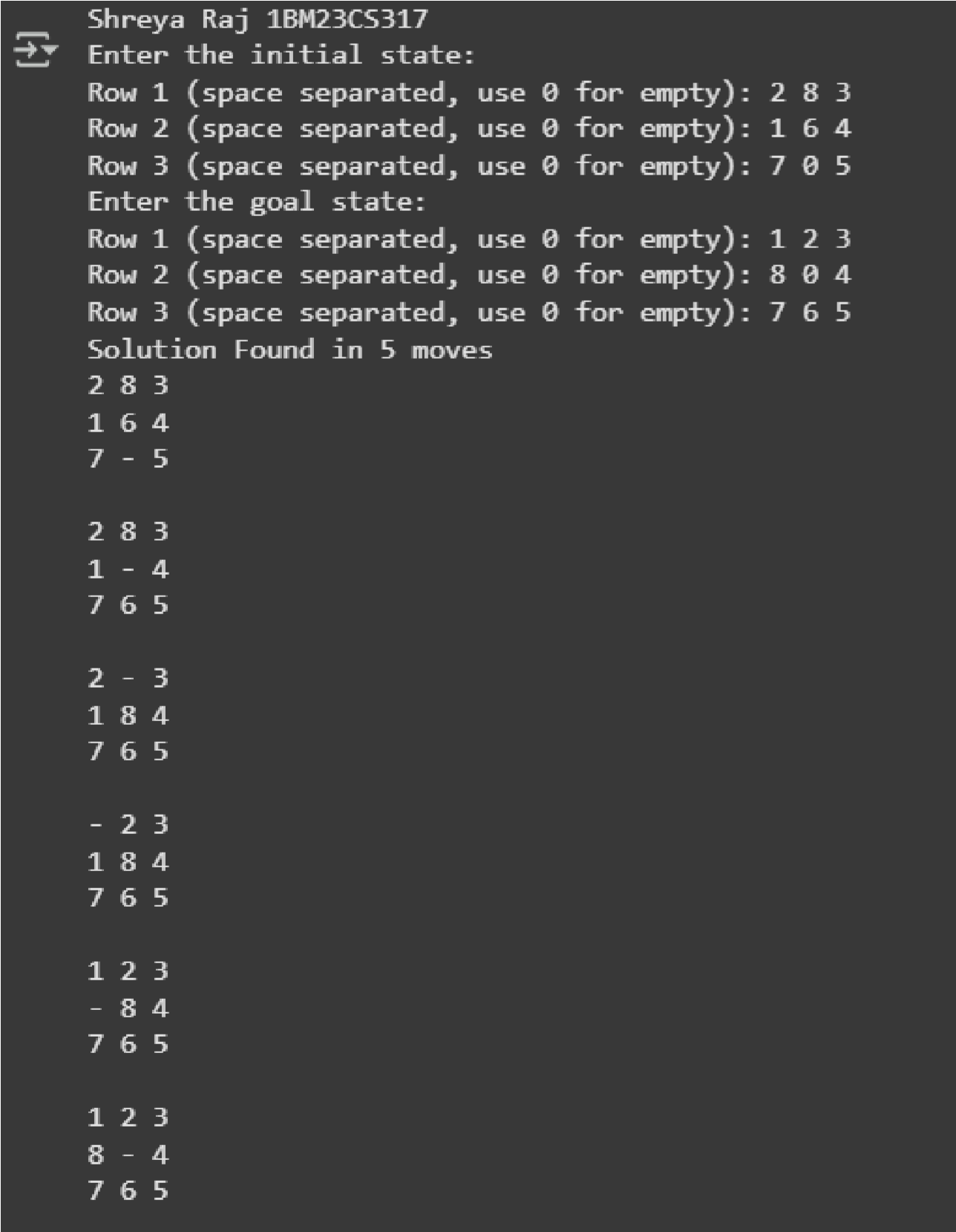
empty\_pos = (i, j) break if empty\_pos is not None: break

start\_state = PuzzleState(start\_board, empty\_pos) path = iddfs(start\_state, goal\_board) if path:

print\_path(path) else:

print("No solution found.")

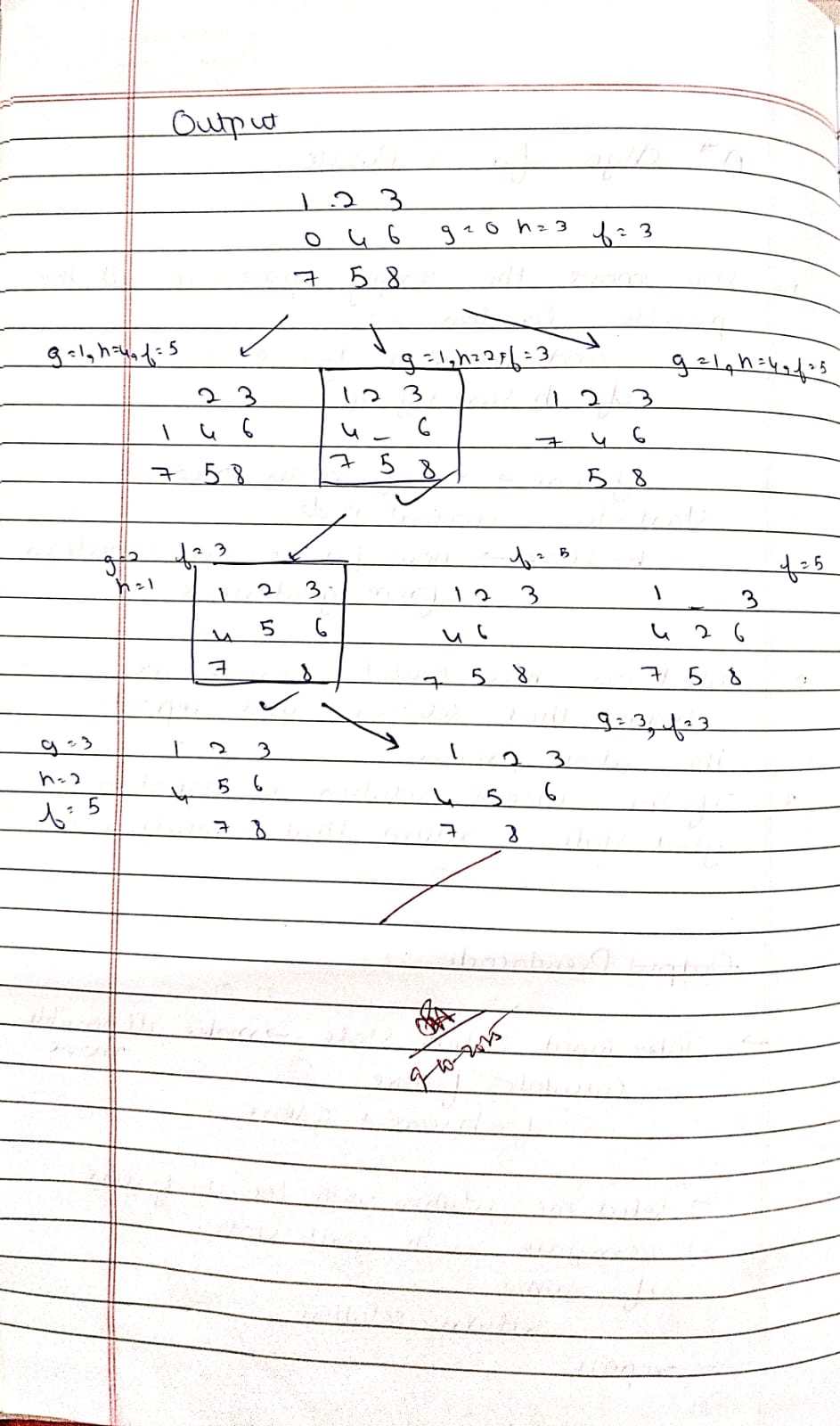
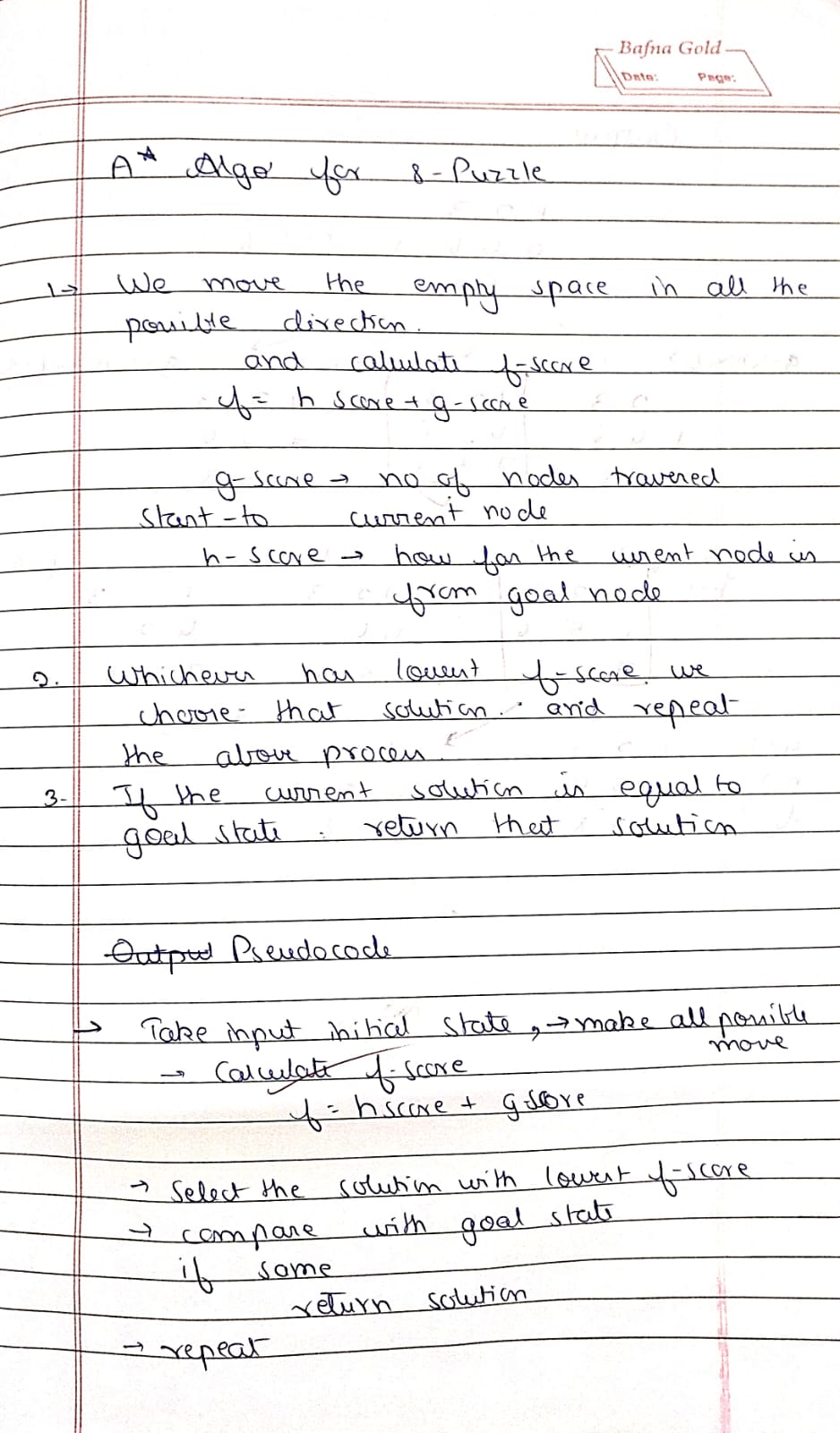
Output:



**Program 3**

Apply A\* algorithm for misplaced tiles.

Algorithm:



Code: import heapq

class PuzzleState: def \_\_init\_\_(self, board, goal, parent=None, g=0):

self.board = board self.goal = goal self.parent = parent

self.g = g self.h = self.misplaced\_tiles() self.f = self.g + self.h

def misplaced\_tiles(self):

"""Count misplaced tiles (excluding 0)."""

return sum(1 for i in range(9) if self.board[i] != 0 and self.board[i] != self.goal[i])

def get\_neighbors(self):

"""Generate possible moves by sliding the blank (0).""" neighbors = [] idx = self.board.index(0) x, y = divmod(idx, 3) # row, col moves = [(-1,0),(1,0),(0,-1),(0,1)] # up, down, left, right

for dx, dy in moves: nx, ny = x+dx, y+dy if 0 <= nx < 3 and 0 <= ny < 3: new\_idx = nx\*3 + ny new\_board = self.board[:] new\_board[idx], new\_board[new\_idx] = new\_board[new\_idx], new\_board[idx] neighbors.append(PuzzleState(new\_board, self.goal, self, self.g+1)) return neighbors

def \_\_lt\_\_(self, other):

return self.f < other.f # priority queue uses f value

def reconstruct\_path(state):

path = [] while state:

path.append(state.board) state = state.parent

return path[::-1]

def astar(start, goal):

start\_state = PuzzleState(start, goal) open\_list = [] heapq.heappush(open\_list, start\_state) closed\_set = set()

while open\_list:

current = heapq.heappop(open\_list)

if current.board == goal:

return reconstruct\_path(current)

closed\_set.add(tuple(current.board))

for neighbor in current.get\_neighbors(): if tuple(neighbor.board) in closed\_set: continue heapq.heappush(open\_list, neighbor) return None

print("Enter the 8-puzzle START state (use 0 for blank).") start\_input = list(map(int, input("Enter 9 numbers separated by spaces: ").split()))

print("\nEnter the GOAL state (use 0 for blank).") goal\_input = list(map(int, input("Enter 9 numbers separated by spaces: ").split()))

if len(start\_input) != 9 or len(goal\_input) != 9:

print("Invalid input! Please enter exactly 9 numbers for each state.") else:

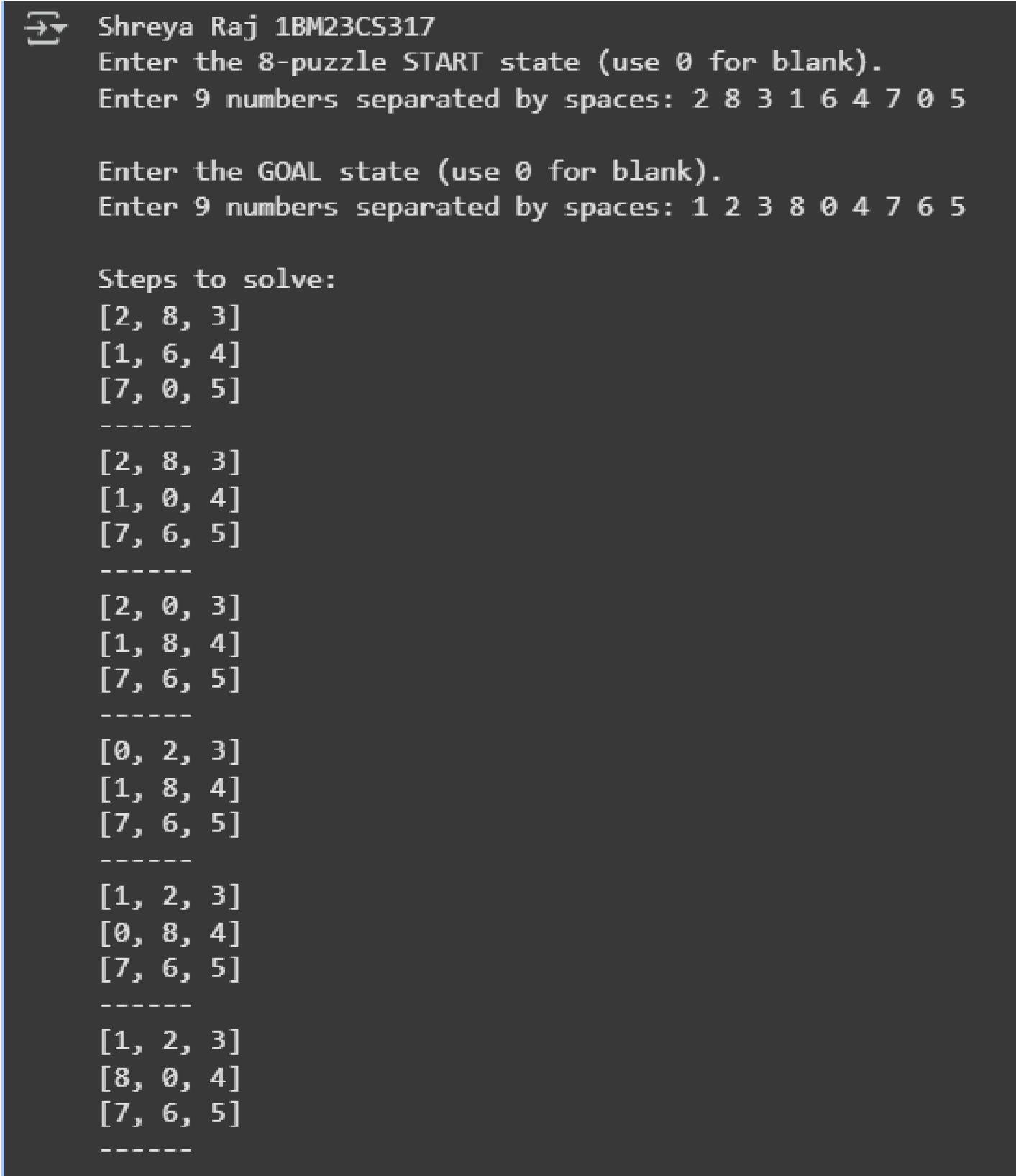
solution = astar(start\_input, goal\_input)

if solution:

print("\n Steps to solve:") for step in solution: for i in range(0,9,3): print(step[i:i+3]) print("------") else:

print(" No solution found!")

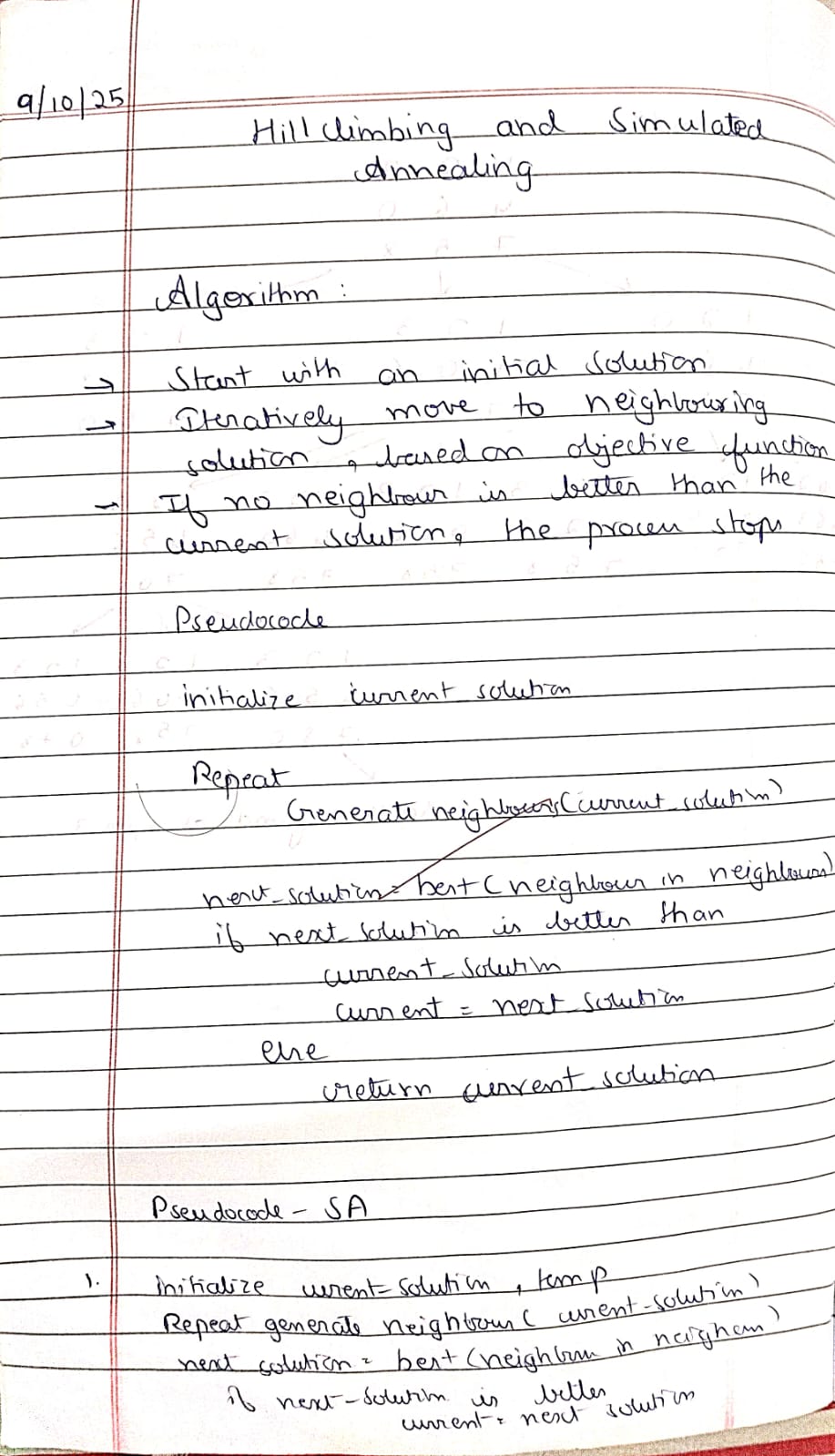
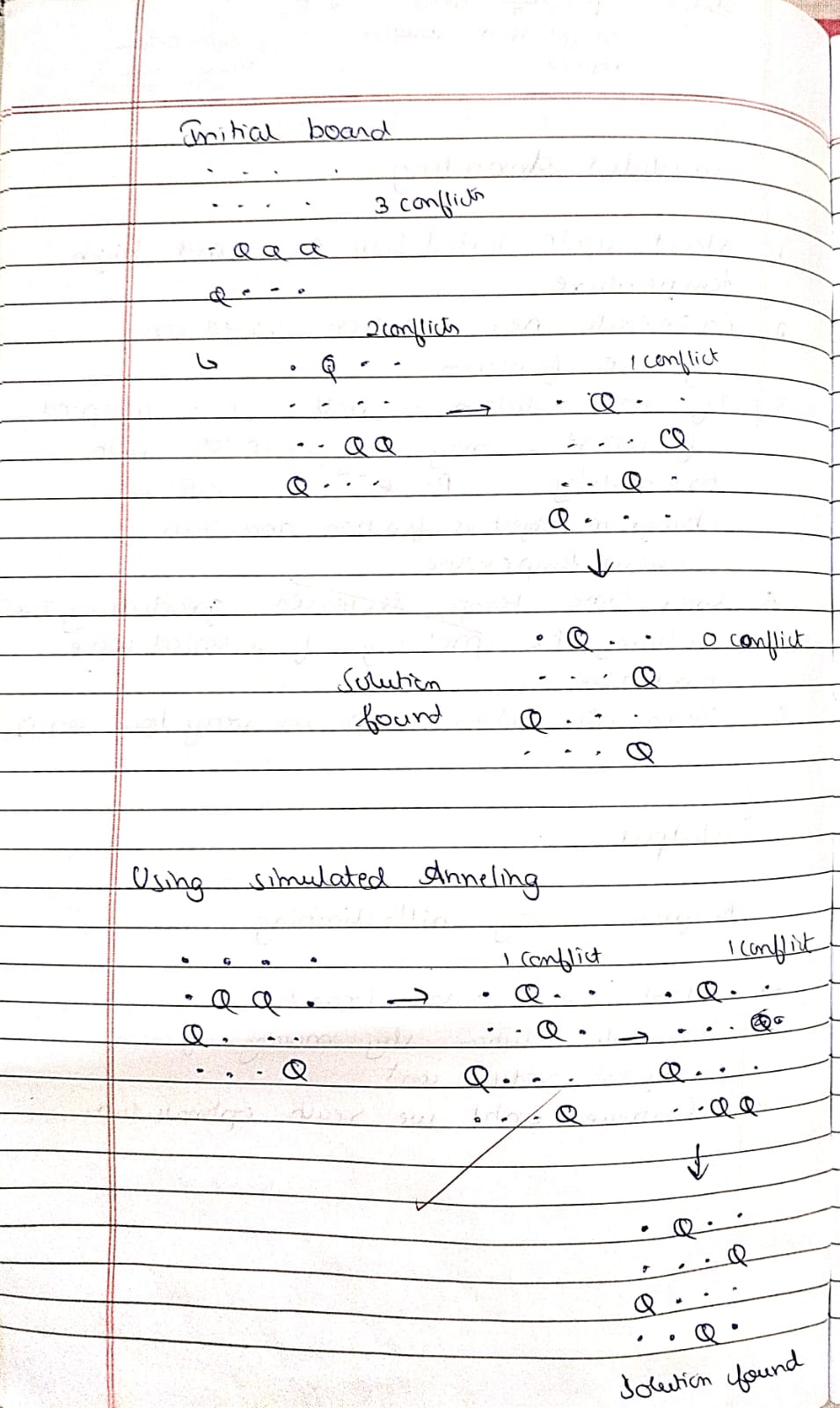
Output:



**Program 4**

Implement hill climbing search algorithm to solve N-Queens problem.

Algorithm:

Code:

def print\_board(state):

n = len(state) for row in range(n): line = "" for col in range(n):

line += "Q " if state[col] == row else ". "

print(line)

print()

def heuristic(state): attacks = 0 n = len(state) for i in range(n): for j in range(i + 1, n):

if state[i] == state[j]: # same row

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

return attacks

def get\_neighbors(state):

neighbors = [] n = len(state) for col in range(n): for row in range(n): if state[col] != row: new\_state = list(state) new\_state[col] = row neighbors.append(new\_state) return neighbors

def hill\_climbing(start\_state):

current = copy.deepcopy(start\_state) while True:

current\_h = heuristic(current) if current\_h == 0: return current, 0

neighbors = get\_neighbors(current)

neighbor\_h = [heuristic(neigh) for neigh in neighbors]

min\_h = min(neighbor\_h) if min\_h >= current\_h: # No improvement possible return current, current\_h

current = neighbors[neighbor\_h.index(min\_h)]

def generate\_all\_states(n):

states = [] def backtrack(col=0, state=[]): if col == n: states.append(state.copy())

return for row in range(n): state.append(row) backtrack(col+1, state)

state.pop() backtrack()

return states

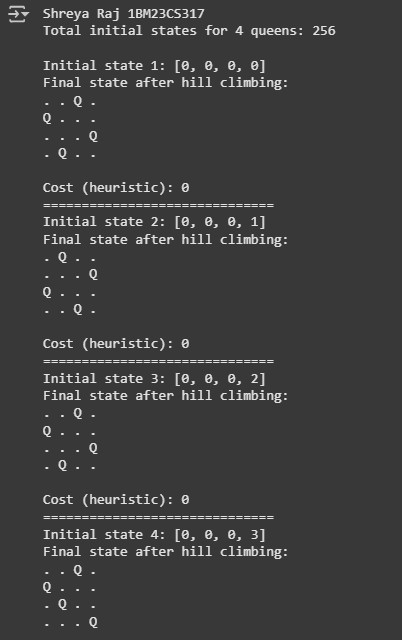
if \_\_name\_\_ == "\_\_main\_\_":

n = 4 all\_states = generate\_all\_states(n)

print(f"Total initial states for {n} queens: {len(all\_states)}\n")

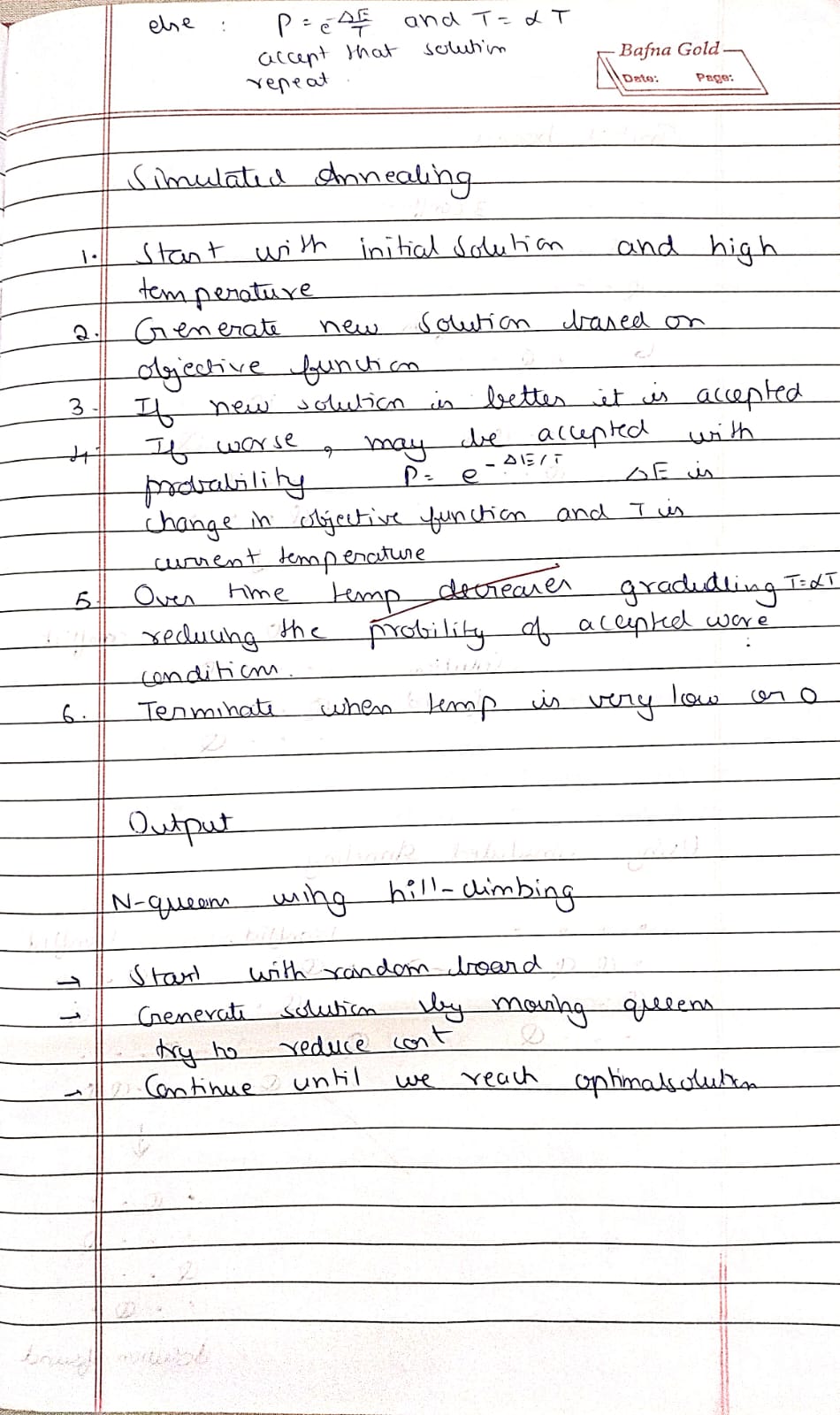
for i, start in enumerate(all\_states, start=1): final\_state, cost = hill\_climbing(start) print(f"Initial state {i}: {start}") print(f"Final state after hill climbing:") print\_board(final\_state) print(f"Cost (heuristic): {cost}") print("="\*30)

Output:



**Program 5**

8 Queens Problem using Simulated Annealing 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):

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

current\_cost = cost(current)

temperature = 100.0

cooling\_rate = 0.95

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:

current, current\_cost = neighbor, neighbor\_cost if neighbor\_cost < best\_cost:

best, best\_cost = neighbor, neighbor\_cost else:

probability = math.exp(delta / temperature) if random.random() < probability:

current, current\_cost = neighbor, neighbor\_cost

temperature \*= cooling\_rate

return best, best\_cost

def print\_board(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()

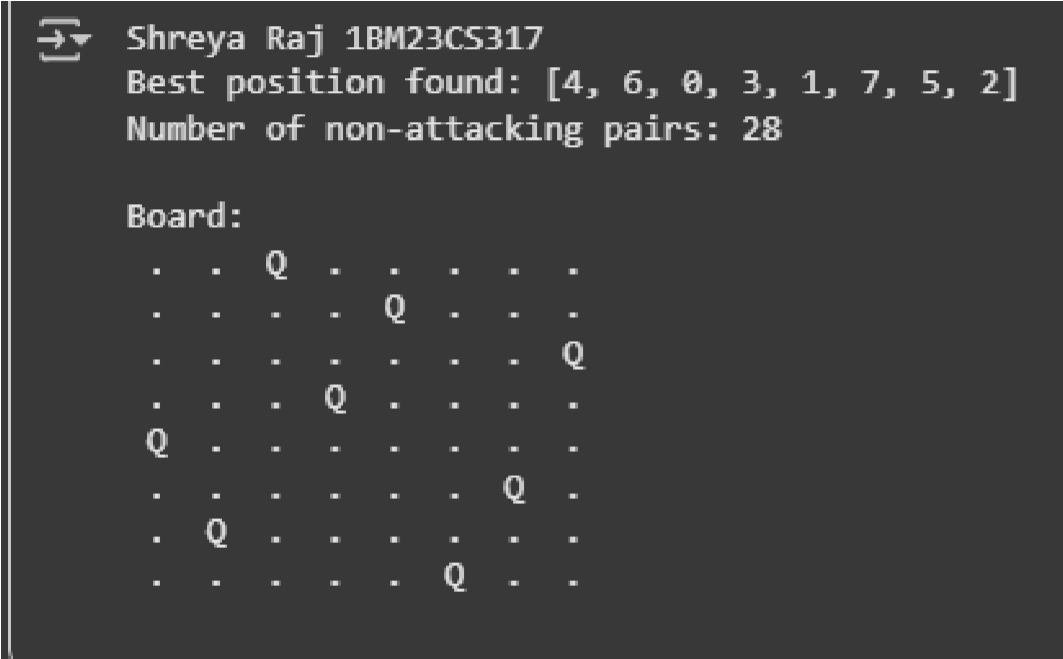
if \_\_name\_\_ == "\_\_main\_\_":

n = 8 solution, cost\_val = simulated\_annealing(n)

print("Best position found:", solution) print(f"Number of non-attacking pairs: {n\*(n-1)//2 - cost\_val}") print("\nBoard:")

print\_board(solution)

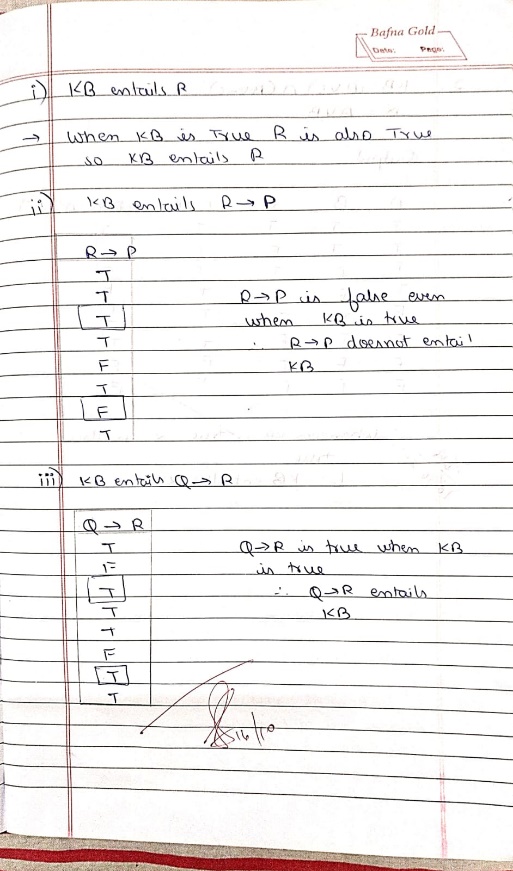
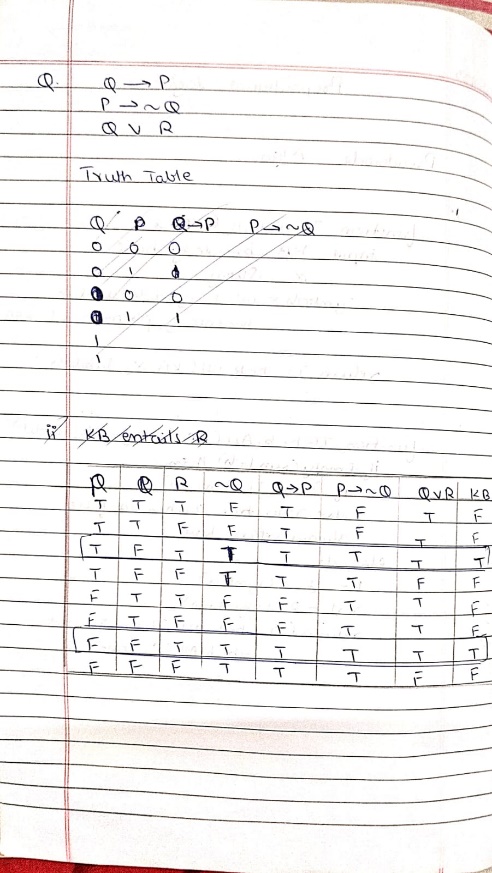
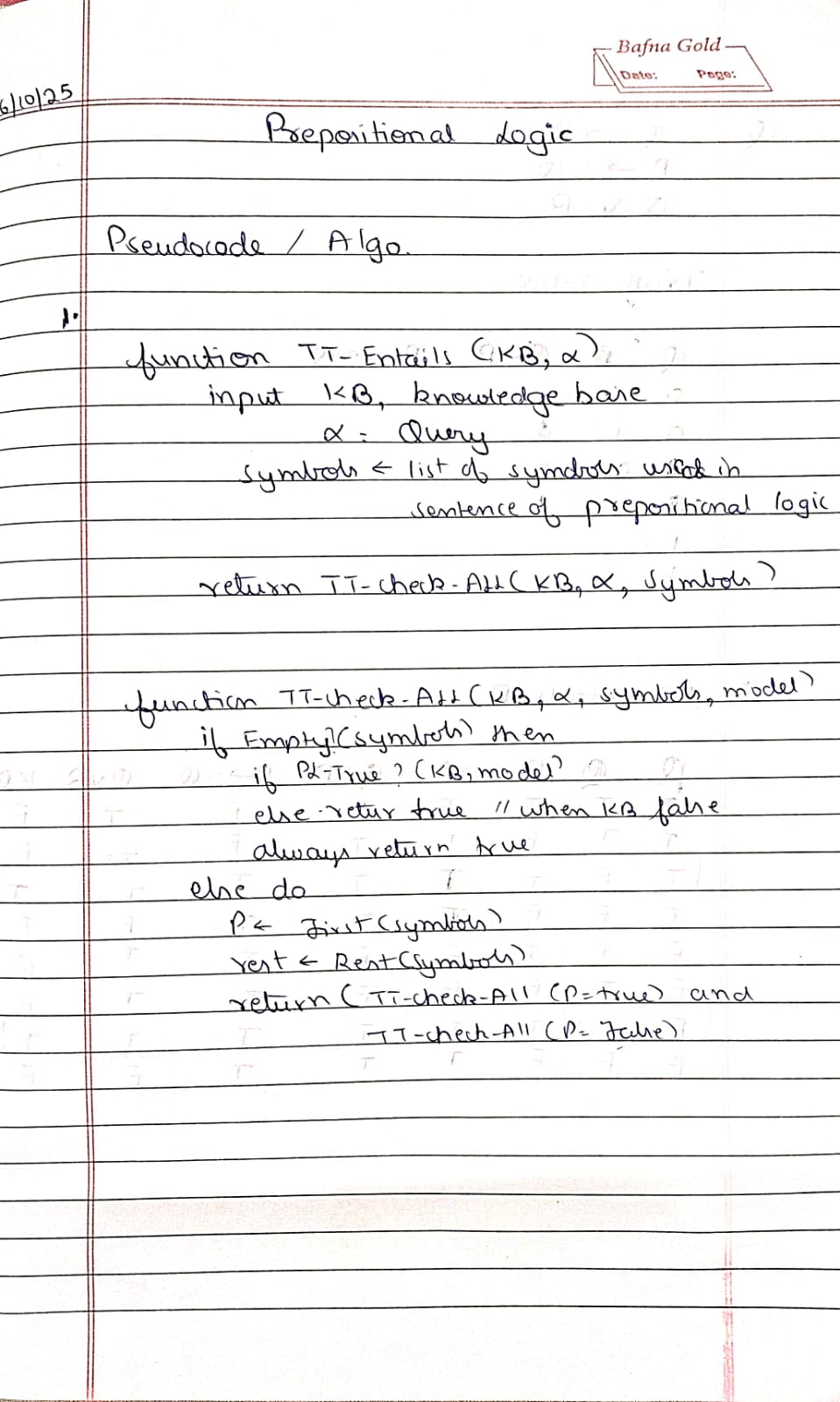
Output:



**Program 6**

Implement truth table enumeration algorithm for deciding propositional entailment.

Algorithm:



Code:

import pandas as pd from itertools import product import re

def tokenize(sentence):

# Now also tokenize symbols like  (logical OR) # Added ∨, ∧, ¬ in the regex to separate them as tokens token\_pattern = r'\w+|[()∨∧¬]' return re.findall(token\_pattern, sentence)

def pl\_true(sentence, model): tokens = tokenize(sentence) logical\_ops = {'and', 'or', 'not', '∨', '∧', '¬'}

evaluated\_tokens = [] for token in tokens: if token == '':

evaluated\_tokens.append('or') # replace symbol with python 'or' elif token == '':

evaluated\_tokens.append('and') # replace symbol with python 'and' elif token == '¬':

evaluated\_tokens.append('not') # replace symbol with python 'not' elif token.lower() in logical\_ops:

evaluated\_tokens.append(token.lower()) elif token in model:

evaluated\_tokens.append(str(model[token])) else:

evaluated\_tokens.append(token) eval\_sentence = ' '.join(evaluated\_tokens) try:

return eval(eval\_sentence) except Exception as e:

print(f"Error evaluating sentence: {eval\_sentence}") raise e

def tt\_entails(kb, alpha, symbols):

truth\_table = [] for model in product([False, True], repeat=len(symbols)):

model\_dict = dict(zip(symbols, model)) kb\_value = pl\_true(kb, model\_dict) alpha\_value = pl\_true(alpha, model\_dict) row = {

'A': model\_dict.get('A', False),

'B': model\_dict.get('B', False),

'C': model\_dict.get('C', False),

'A  C': model\_dict.get('A', False) or model\_dict.get('C', False),

'B  ¬C': model\_dict.get('B', False) or not model\_dict.get('C', False), 'KB': kb\_value,

'α': alpha\_value

}

truth\_table.append(row) if kb\_value and not alpha\_value:

return False, pd.DataFrame(truth\_table)

return True, pd.DataFrame(truth\_table)

def get\_symbols(kb, alpha):

return sorted(set(re.findall(r'[A-Z]', kb + alpha)))

alpha = "A  B"

symbols = get\_symbols(kb, alpha) result, truth\_table = tt\_entails(kb, alpha, symbols)

def highlight\_kb\_alpha(row): if row['KB'] and row['α']:

return ['background-color: lightgreen' if col in ['KB', 'α'] else '' for col in row.index] else:

return ['' for \_ in row.index]

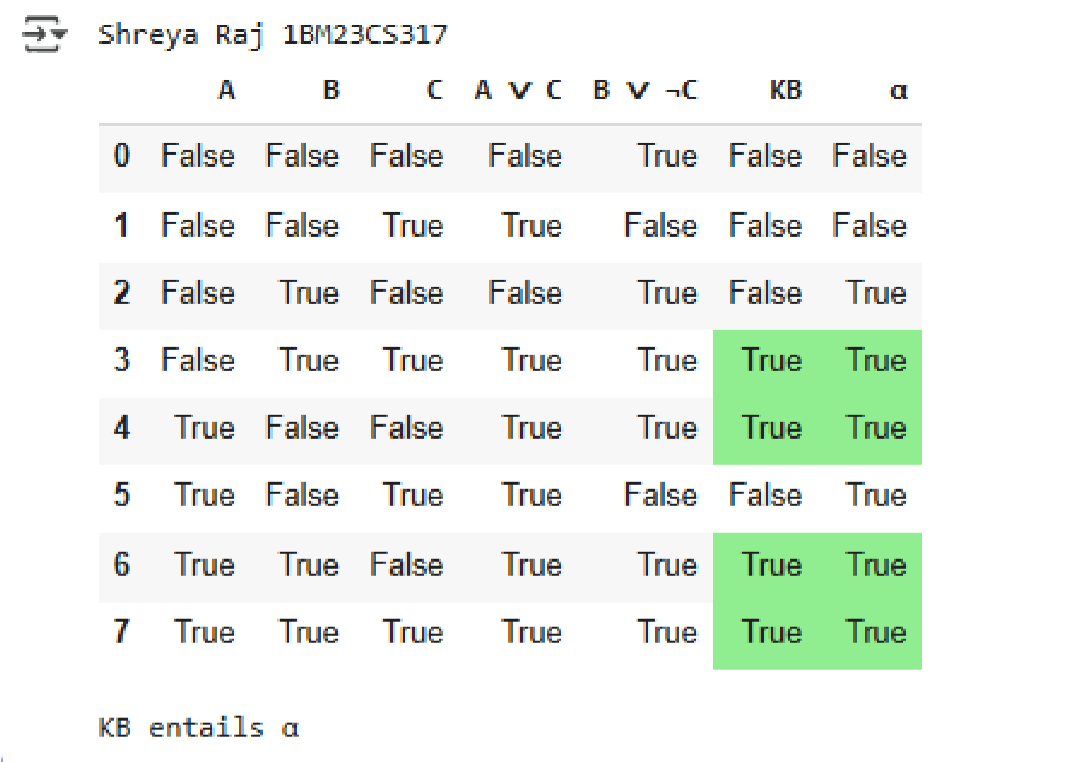
print("Shreya Raj 1BM23CS317") styled\_table = truth\_table.style.apply(highlight\_kb\_alpha, axis=1) display(styled\_table)

if result:

print("\nKB entails α") else:

print("\nKB does not entail α")

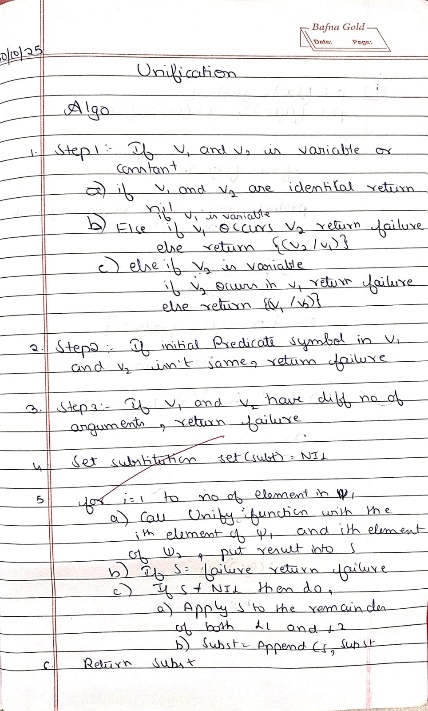
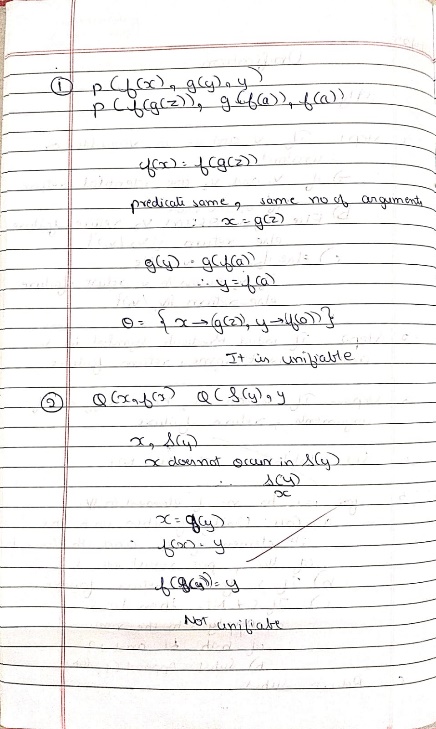
Output:



**Program 7**

Implement unification in first order logic.

Algorithm:

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

subst = {}

# If x or y is a variable or constant if is\_variable(x) or is\_constant(x): if x == y:

return subst elif is\_variable(x):

return unify\_var(x, y, subst) elif is\_variable(y):

return unify\_var(y, x, subst) else:

return None

# If both x and y are compound expressions if is\_compound(x) and is\_compound(y): if x[0] != y[0] or len(x[1]) != len(y[1]):

return None for xi, yi in zip(x[1], y[1]): subst = unify(xi, yi, subst) if subst is None: return None return subst

return None

def is\_variable(x):

return isinstance(x, str) and x.islower() and x.isalpha()

def is\_constant(x):

return isinstance(x, str) and x.isupper() and x.isalpha()

def is\_compound(x): return isinstance(x, tuple) and len(x) == 2 and isinstance(x[0], str) and isinstance(x[1], list)

def unify\_var(var, x, subst): if var in subst:

return unify(subst[var], x, subst) elif x in subst:

return unify(var, subst[x], subst) elif occurs\_check(var, x, subst):

return None else:

subst[var] = x

return subst

def occurs\_check(var, x, subst): if var == x: return True elif is\_variable(x) and x in subst:

return occurs\_check(var, subst[x], subst) elif is\_compound(x):

return any(occurs\_check(var, arg, subst) for arg in x[1]) else:

return False

x = ("P", ["x", "A"])

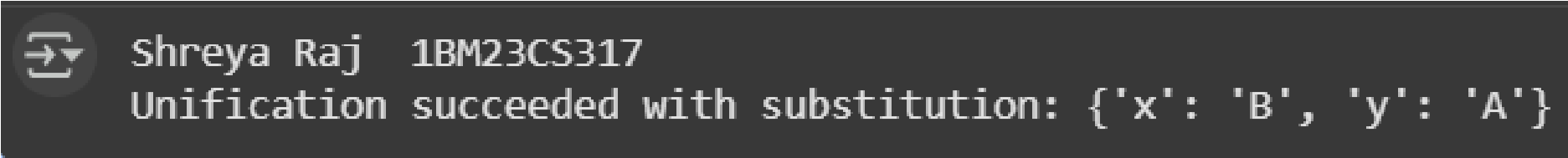
y = ("P", ["B", "y"])

result = unify(x, y) if result is not None:

print("Unification succeeded with substitution:", result) else:

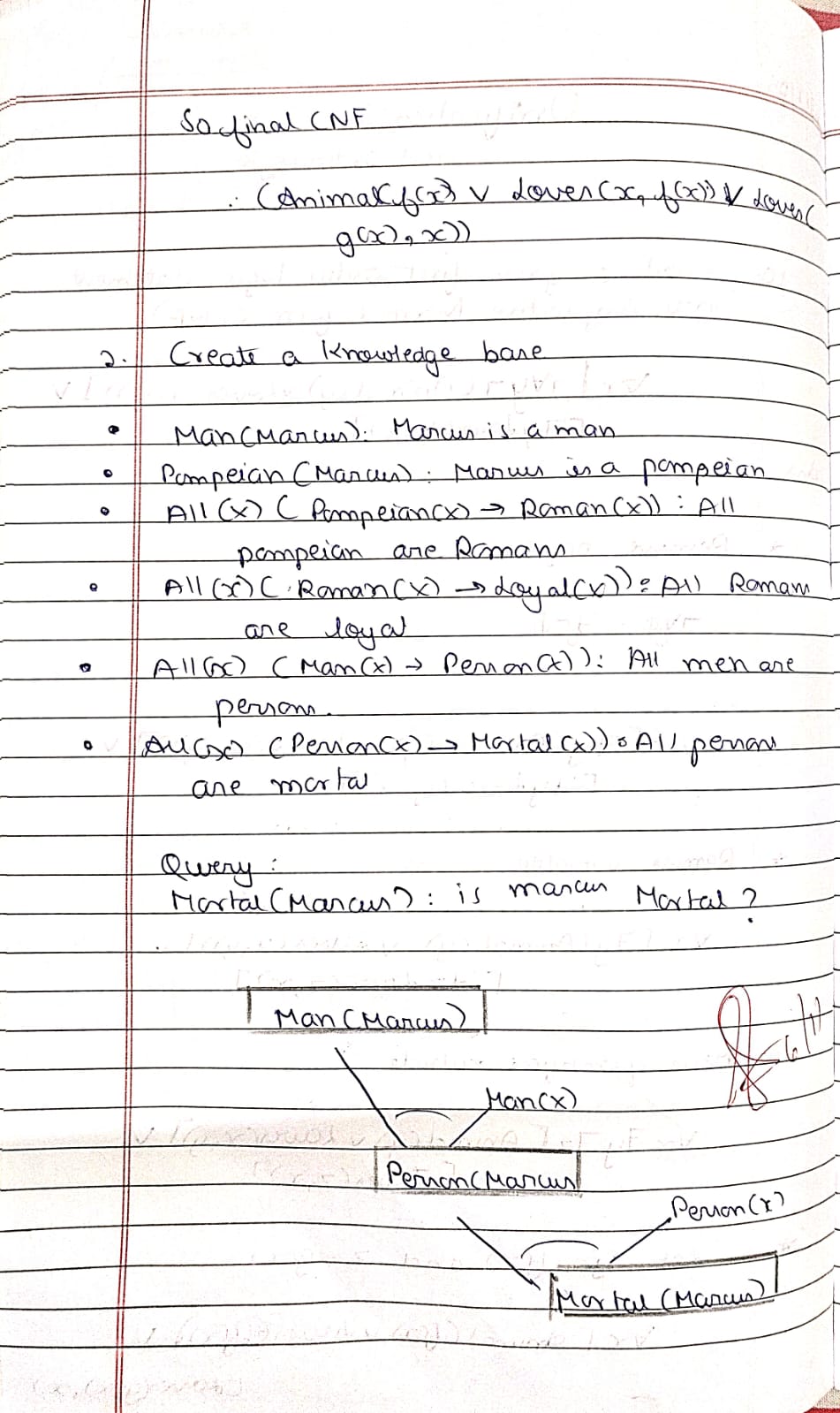
print("Unification failed.")

Output:



**Program 8**

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



Code: facts = {

'American(West)': True,

'Hostile(Nono)': True,

'Missiles(Nono)': True,

} def rule1(facts): if facts.get('American(West)', False) and facts.get('Hostile(Nono)', False):

return 'Criminal(West)'

return None

def rule2(facts): if facts.get('Missiles(Nono)', False) and facts.get('Hostile(Nono)', False):

return 'SoldWeapons(West, Nono)'

def forward\_chaining(facts, rules):

new\_facts = facts.copy() inferred = True while inferred: inferred = False for rule in rules:

result = rule(new\_facts) if result and result not in new\_facts:

new\_facts[result] = True inferred = True print(f"New fact inferred: {result}")

return new\_facts

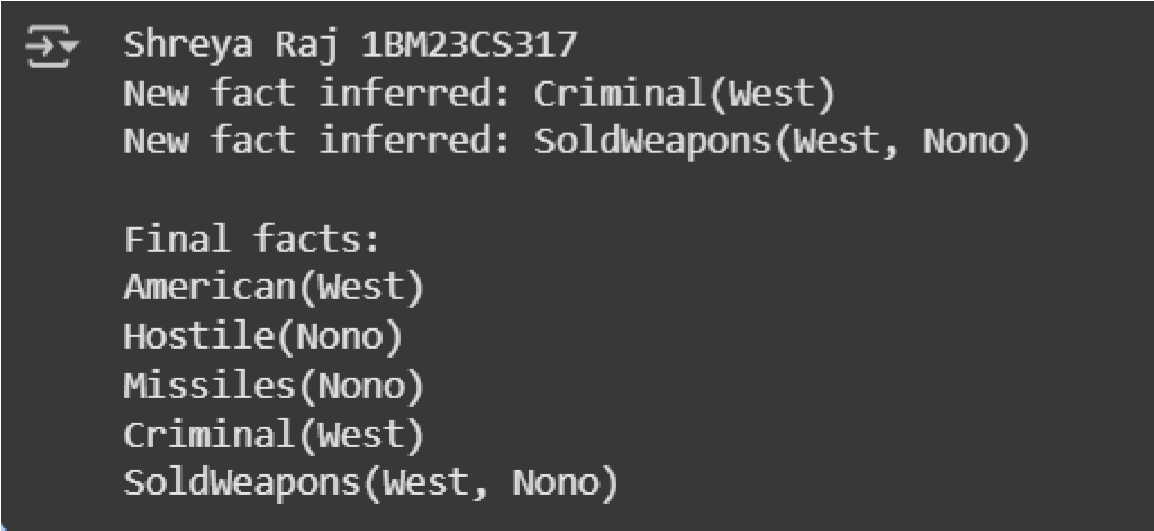
rules = [rule1, rule2]

inferred\_facts = forward\_chaining(facts, rules)

print("\nFinal facts:") for fact in inferred\_facts:

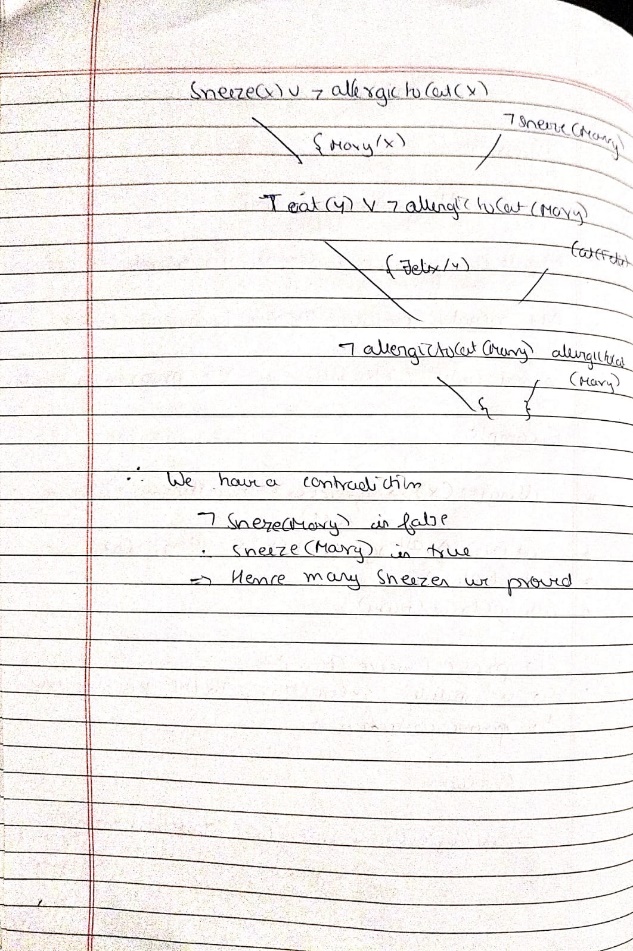
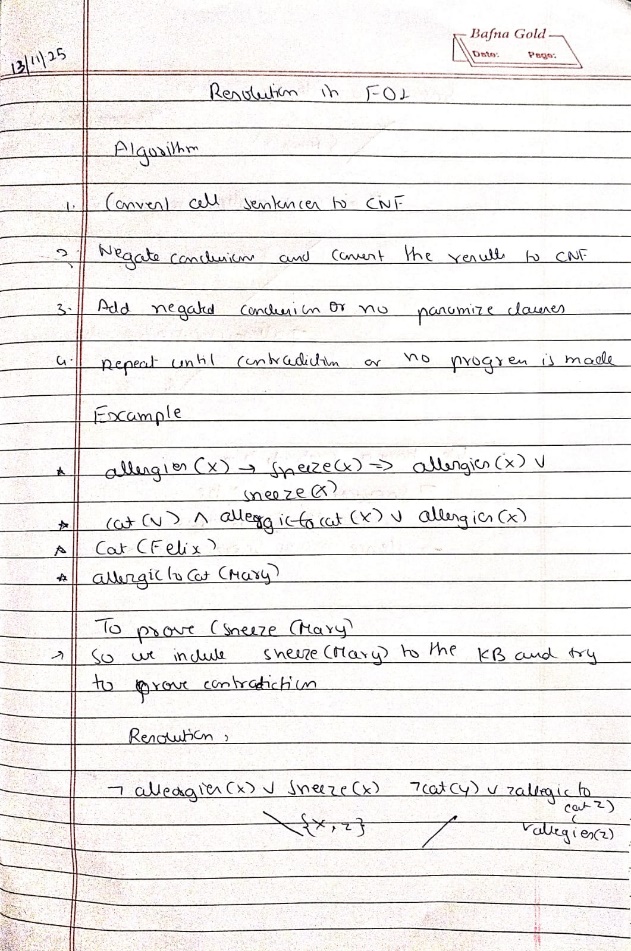
print(fact)

Output:



**Program 9**

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



Code: from collections import deque

import itertools import copy import pprint

# ----- Data structures ----- class Var: def \_\_init\_\_(self, name):

self.name = name def \_\_repr\_\_(self):

return f"Var({self.name})" def \_\_eq\_\_(self, other):

return isinstance(other, Var) and self.name == other.name def \_\_hash\_\_(self):

return hash(('Var', self.name))

class Const: def \_\_init\_\_(self, name):

self.name = name def \_\_repr\_\_(self):

return f"Const({self.name})" def \_\_eq\_\_(self, other):

return isinstance(other, Const) and self.name == other.name def \_\_hash\_\_(self):

return hash(('Const', self.name))

class Func: def \_\_init\_\_(self, name, args):

self.name = name self.args = args def \_\_repr\_\_(self):

return f"Func({self.name}, {self.args})" def \_\_eq\_\_(self, other):

return isinstance(other, Func) and self.name == other.name and self.args == other.args def \_\_hash\_\_(self):

return hash(('Func', self.name, tuple(self.args)))

class Literal:

# predicate\_name: str, args: list of Terms, negated: bool def \_\_init\_\_(self, predicate, args, negated=False):

self.predicate = predicate self.args = tuple(args) self.negated = negated def negate(self):

return Literal(self.predicate, list(self.args), not self.negated) def \_\_repr\_\_(self):

sign = "~" if self.negated else ""

args = ",".join(map(term\_to\_str, self.args)) return f"{sign}{self.predicate}({args})" def \_\_eq\_\_(self, other):

return (self.predicate, self.args, self.negated) == (other.predicate, other.args, other.negated) def \_\_hash\_\_(self):

return hash((self.predicate, self.args, self.negated))

# Clause is frozenset of Literal def clause\_to\_str(cl):

return " OR ".join(map(str, cl)) if cl else "EMPTY"

def term\_to\_str(t): if isinstance(t, Var):

return t.name if isinstance(t, Const):

return t.name if isinstance(t, Func):

return f"{t.name}({','.join(term\_to\_str(a) for a in t.args)})" return str(t)

# ----- Substitution utilities ----- def apply\_subst\_term(term, subst): if isinstance(term, Var): if term in subst:

return apply\_subst\_term(subst[term], subst) else:

return term elif isinstance(term, Const):

return term elif isinstance(term, Func):

return Func(term.name, [apply\_subst\_term(a, subst) for a in term.args]) else:

return term

def apply\_subst\_literal(lit, subst):

return Literal(lit.predicate, [apply\_subst\_term(a, subst) for a in lit.args], lit.negated)

def apply\_subst\_clause(clause, subst):

return frozenset(apply\_subst\_literal(l, subst) for l in clause)

# ----- Unification (Robust, with occurs-check) ----- def occurs\_check(var, term, subst):

term = apply\_subst\_term(term, subst) if term == var: return True if isinstance(term, Func):

return any(occurs\_check(var, arg, subst) for arg in term.args) return False

def unify\_terms(x, y, subst):

# returns updated subst or None on failure x = apply\_subst\_term(x, subst) y = apply\_subst\_term(y, subst)

if isinstance(x, Var): if x == y:

return subst if occurs\_check(x, y, subst):

return None new = subst.copy() new[x] = y return new if isinstance(y, Var):

return unify\_terms(y, x, subst) if isinstance(x, Const) and isinstance(y, Const): return subst if x.name == y.name else None if isinstance(x, Func) and isinstance(y, Func) and x.name == y.name and len(x.args) == len(y.args): for a, b in zip(x.args, y.args):

subst = unify\_terms(a, b, subst) if subst is None: return None return subst

return None

def unify\_literals(l1, l2):

# l1 and l2 must have same predicate and opposite polarity for resolution if l1.predicate != l2.predicate or l1.negated == l2.negated or len(l1.args) != len(l2.args):

return None subst = {} for a, b in zip(l1.args, l2.args):

subst = unify\_terms(a, b, subst) if subst is None: return None

return subst

# ----- Standardize apart variables (to avoid name clashes) -----

\_var\_count = 0 def standardize\_apart(clause):

global \_var\_count varmap = {} new\_literals = [] for lit in clause: new\_args = [] for t in lit.args:

new\_args.append(\_rename\_term\_vars(t, varmap)) new\_literals.append(Literal(lit.predicate, new\_args, lit.negated)) return frozenset(new\_literals)

def \_rename\_term\_vars(term, varmap):

global \_var\_count if isinstance(term, Var): if term.name not in varmap: \_var\_count += 1

varmap[term.name] = Var(f"{term.name}\_{\_var\_count}") return varmap[term.name] if isinstance(term, Const):

return term if isinstance(term, Func):

return Func(term.name, [\_rename\_term\_vars(a, varmap) for a in term.args]) return term

# ----- Resolution operation between two clauses ----- def resolve(ci, cj):

# returns set of resolvent clauses (frozenset of literals) resolvents = set() ci = standardize\_apart(ci) cj = standardize\_apart(cj) for li in ci: for lj in cj: if li.predicate == lj.predicate and li.negated != lj.negated and len(li.args) == len(lj.args):

subst = unify\_literals(li, lj) if subst is not None:

# build resolvent: (Ci - {li}) U (Cj - {lj}) with subst applied

new\_clause = set(apply\_subst\_literal(l, subst) for l in (ci - {li}) | (cj - {lj}))

# remove tautologies: a clause containing P and ~P after subst preds = {} taut = False for l in new\_clause:

key = (l.predicate, tuple(map(term\_to\_str, l.args))) if key in preds and preds[key] != l.negated:

taut = True break preds[key] = l.negated if not taut:

resolvents.add(frozenset(new\_clause)) return resolvents

# ----- Main resolution loop ----- def fol\_resolution(kb\_clauses, query\_clause, max\_iterations=20000):

"""

kb\_clauses: set/list of clauses (each clause is frozenset of Literal) query\_clause: single Literal (to be proved), will be negated and added to KB Returns True if contradiction (empty clause) is derived.

"""

# Negate the query and add its literals as separate clauses (each literal is a clause) negated\_query = [query\_clause.negate()] clauses = set(kb\_clauses) for l in negated\_query:

clauses.add(frozenset([l]))

new = set() processed\_pairs = set()

queue = list(clauses)

iterations = 0 while True: pairs = [] clause\_list = list(clauses) n = len(clause\_list)

# iterate over all unordered pairs for i in range(n): for j in range(i+1, n):

pairs.append((clause\_list[i], clause\_list[j]))

something\_added = False for (ci, cj) in pairs: pair\_key = (ci, cj) if pair\_key in processed\_pairs: continue

processed\_pairs.add(pair\_key) resolvents = resolve(ci, cj) iterations += 1 if iterations > max\_iterations:

return False, "max\_iterations\_exceeded" for r in resolvents: if len(r) == 0:

return True, "Derived empty clause (success)" if r not in clauses and r not in new:

new.add(r) something\_added = True if not something\_added: return False, "No new clauses — failure (KB does not entail query)" clauses.update(new) new = set()

# ----- Helper to create easy constants/vars ----- def C(name): return Const(name) def V(name): return Var(name)

def F(name, \*args): return Func(name, list(args)) def L(pred, args, neg=False): return Literal(pred, args, neg)

# Build clauses (using variables V('x'), constants C('Anil'), etc.)

x = V('x') y = V('y')

kb = set()

# 1. ¬Food(x)  Likes(John,x)

kb.add(frozenset([L('Food', [x], neg=True), L('Likes', [C('John'), x], neg=False)]))

# 2a. Food(Apple)

kb.add(frozenset([L('Food', [C('apple')], neg=False)]))

# 2b. Food(vegetable)

kb.add(frozenset([L('Food', [C('vegetable')], neg=False)]))

# 3. ¬Eats(x,y)  Killed(y)  Food(y)

kb.add(frozenset([L('Eats', [x,y], neg=True), L('Killed', [y], neg=False), L('Food', [y], neg=False)]))

# 4a. Eats(Anil,peanuts)

kb.add(frozenset([L('Eats', [C('Anil'), C('peanuts')], neg=False)]))

# 4b. Alive(Anil)

kb.add(frozenset([L('Alive', [C('Anil')], neg=False)]))

# 5. ¬Eats(Anil,x)  Eats(Harry,x)

kb.add(frozenset([L('Eats', [C('Anil'), x], neg=True), L('Eats', [C('Harry'), x], neg=False)]))

# 6. ¬Alive(x)  ¬Killed(x)

kb.add(frozenset([L('Alive', [x], neg=True), L('Killed', [x], neg=True)]))

# 7. Killed(x)  Alive(x)

kb.add(frozenset([L('Killed', [x], neg=True), L('Alive', [x], neg=False)]))

# Query

query = L('Likes', [C('John'), C('peanuts')], neg=False)

def show\_kb(kb):

print("Knowledge base clauses:") for c in kb:

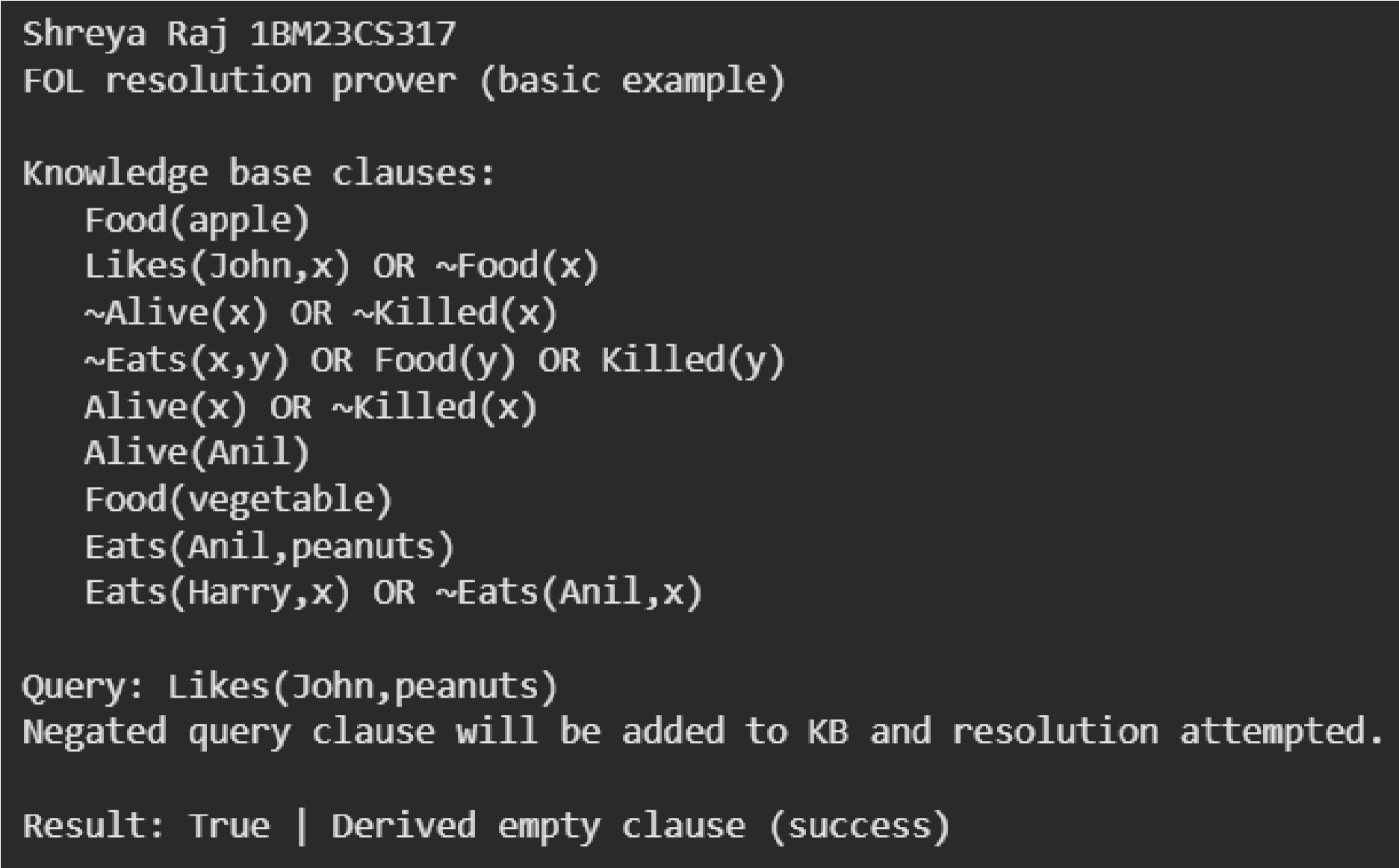
print(" ", clause\_to\_str(c))

print()

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

print("FOL resolution prover (basic example)\n") show\_kb(kb) print("Query:", query) print("Negated query clause will be added to KB and resolution attempted.\n") success, info = fol\_resolution(kb, query, max\_iterations=20000) print("Result:", success, "|", info)

Output:



**Program 10**

Implement Alpha-Beta Pruning Algorithm:

Code: def alpha\_beta(node\_index, depth, max\_depth, alpha, beta, is\_max, values, explored, pruned, path):

node\_index : index of current node in conceptual tree

depth : current depth (0 = root) max\_depth : total depth of tree values : list of leaf node values

total\_leaves = len(values)

if depth == max\_depth:

leaf\_index = node\_index - (2 \*\* max\_depth - 1) if 0 <= leaf\_index < total\_leaves: val = values[leaf\_index] explored.append((list(path), val)) return val else: return 0 # Safety fallback if is\_max:

value = float('-inf') for i in range(2): # left & right children child\_index = node\_index \* 2 + i + 1 path.append(child\_index) value = max(value, alpha\_beta(child\_index, depth + 1, max\_depth, alpha, beta, False, values, explored, pruned, path)) path.pop() alpha = max(alpha, value) if beta <= alpha:

pruned.append((node\_index, child\_index, 'Beta cutoff')) break return value else:

value = float('inf') for i in range(2):

child\_index = node\_index \* 2 + i + 1 path.append(child\_index) value = min(value, alpha\_beta(child\_index, depth + 1, max\_depth, alpha, beta, True, values, explored, pruned, path)) path.pop() beta = min(beta, value) if beta <= alpha:

pruned.append((node\_index, child\_index, 'Alpha cutoff')) break return value

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

values = [3, 5, 6, 9, 1, 2, 0, -1] # leaf node values max\_depth = 3 # since 2^3 = 8 leaves explored, pruned = [], [] print("Leaf node values:", values) result = alpha\_beta(0, 0, max\_depth, float('-inf'), float('inf'),

True, values, explored, pruned, [0])

print("\nValue of root node (MAX mode):", result)

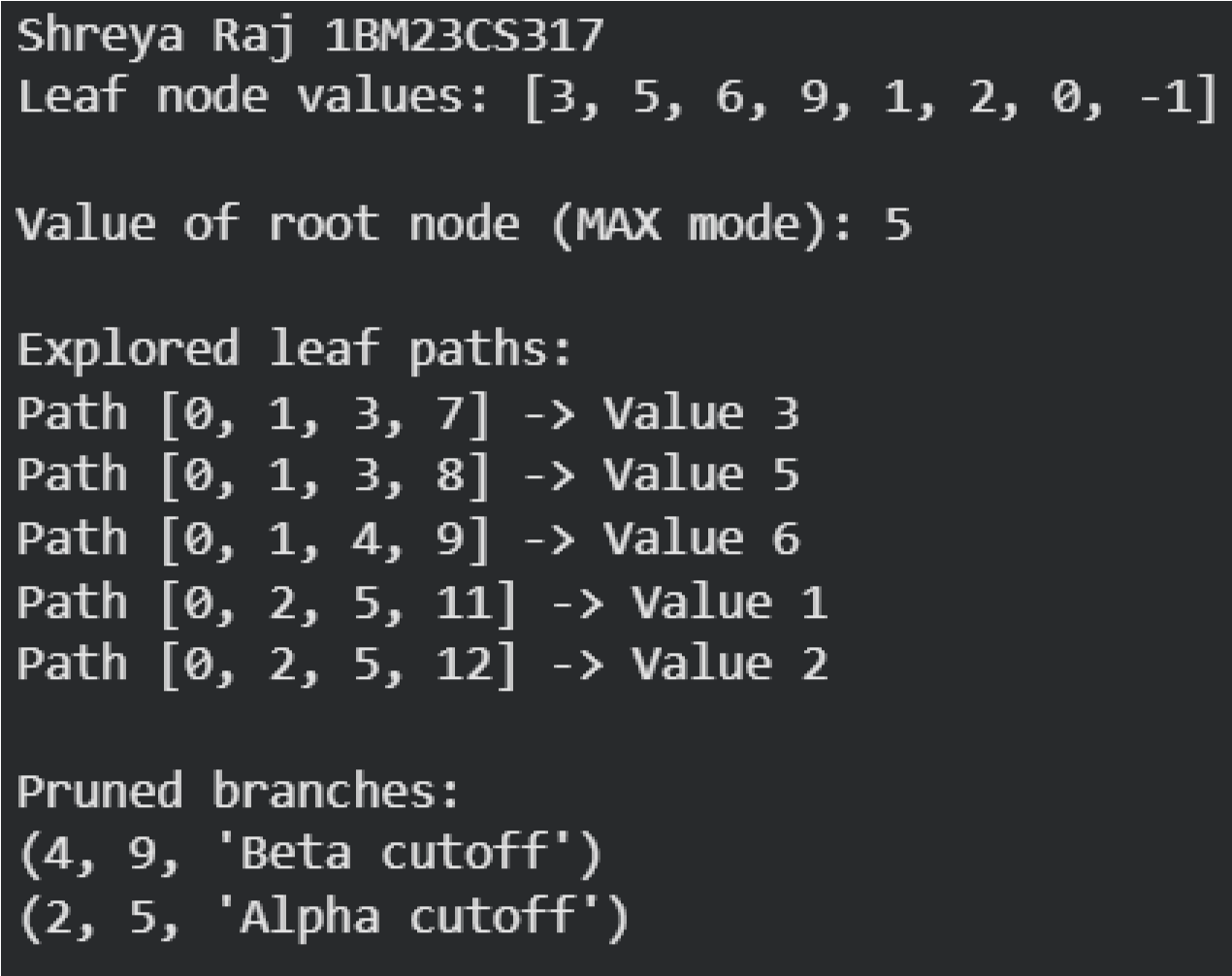
print("\nExplored leaf paths:") for p, val in explored:

print(f"Path {p} -> Value {val}")

print("\nPruned branches:") for item in pruned:

print(item)

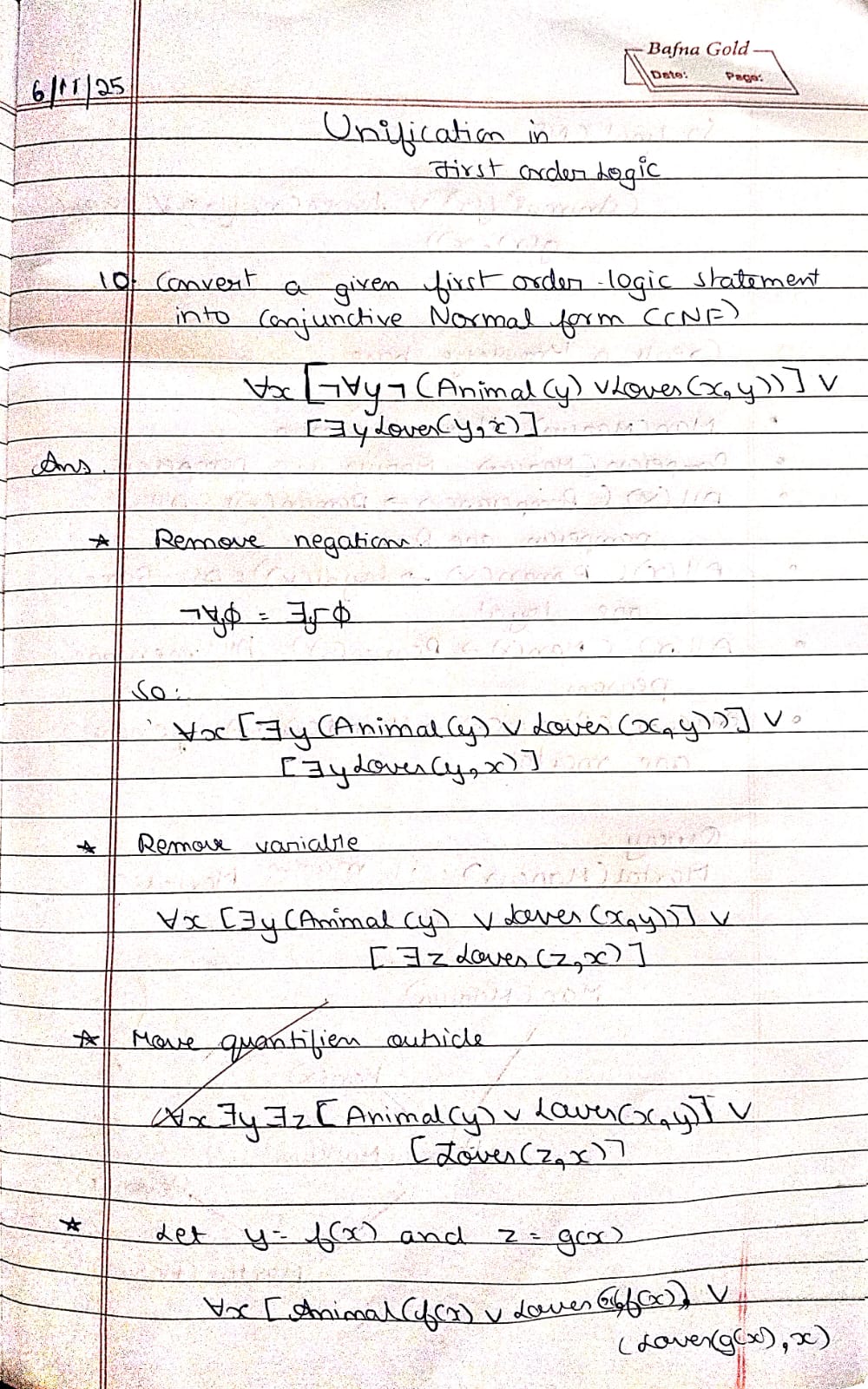
Output:



**Program 11:**

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm:



Code:

import re

def getAttributes(string):

expr = r'\([^)]+\)'

matches = re.findall(expr, string)

return [m for m in str(matches) if m.isalpha()]

def getPredicates(string):

expr = r'[A-Za-z~]+\([A-Za-z,]+\)'

return re.findall(expr, string)

def DeMorgan(sentence):

string = ''.join(list(sentence).copy())

string = string.replace('~~', '')

flag = '[' in string

string = string.replace('~[', '')

string = string.strip(']')

for predicate in getPredicates(string):

string = string.replace(predicate, f'~{predicate}')

s = list(string)

for i, c in enumerate(s):

if c == 'V':

s[i] = '^'

elif c == '^':

s[i] = 'V'

string = ''.join(s)

string = string.replace('~~', '')

return f'[{string}]' if flag else string

def Skolemization(sentence):

SKOLEM\_CONSTANTS = [f'{chr(c)}' for c in range(ord('A'), ord('Z')+1)]

statement = ''.join(list(sentence).copy())

matches = re.findall('[∀∃].', statement)

for match in matches[::-1]:

statement = statement.replace(match, '')

statements = re.findall(r'\[\[[^]]+\]\]', statement)

for s in statements:

statement = statement.replace(s, s[1:-1])

for predicate in getPredicates(statement):

attributes = getAttributes(predicate)

if ''.join(attributes).islower():

statement = statement.replace(predicate, predicate)

else:

aL = [a for a in attributes if a.islower()]

aU = [a for a in attributes if not a.islower()][0] if attributes else ''

if aU:

statement = statement.replace(aU, f'{SKOLEM\_CONSTANTS.pop(0)}({aL[0] if len(aL) else match[1]})')

return statement

def clean\_output(expr):

# Remove multiple brackets and redundant negations

expr = expr.replace('~~', '')

while '[[' in expr or ']]' in expr:

expr = expr.replace('[[', '[').replace(']]', ']')

expr = expr.strip('[] ')

# Remove redundant outer brackets like [(p | q)] -> p | q

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

expr = expr[1:-1]

# Replace internal redundant patterns

expr = re.sub(r'\s+', ' ', expr)

return expr

def fol\_to\_cnf(fol):

statement = fol.replace("<=>", "\_")

while '\_' in statement:

i = statement.index('\_')

new\_statement = '[' + statement[:i] + '=>' + statement[i+1:] + ']^[' + statement[i+1:] + '=>' + statement[:i] + ']'

statement = new\_statement

statement = statement.replace("=>", "-")

expr = r'\[([^]]+)\]'

statements = re.findall(expr, statement)

for i, s in enumerate(statements):

if '[' in s and ']' not in s:

statements[i] += ']'

for s in statements:

statement = statement.replace(s, fol\_to\_cnf(s))

while '-' in statement:

i = statement.index('-')

br = statement.index('[') if '[' in statement else 0

new\_statement = '~' + statement[br:i] + 'V' + statement[i+1:]

statement = statement[:br] + new\_statement if br > 0 else new\_statement

while '~∀' in statement:

i = statement.index('~∀')

statement = list(statement)

statement[i], statement[i+1], statement[i+2] = '∃', statement[i+2], '~'

statement = ''.join(statement)

while '~∃' in statement:

i = statement.index('~∃')

s = list(statement)

s[i], s[i+1], s[i+2] = '∀', s[i+2], '~'

statement = ''.join(s)

statement = statement.replace('~[∀', '[~∀')

statement = statement.replace('~[∃', '[~∃')

expr = r'(~[∀V∃].)'

statements = re.findall(expr, statement)

for s in statements:

statement = statement.replace(s, fol\_to\_cnf(s))

expr = r'~\[[^]]+\]'

statements = re.findall(expr, statement)

for s in statements:

statement = statement.replace(s, DeMorgan(s))

return statement

def main():

print("=== FOL to CNF Converter (Simplified Output) ===")

print("Supports ∀, ∃, ~, &, |, >>, <=>, brackets [] () {}")

print("Example 1: ∀x[~∀y~(Animal(y) | Loves(x,y)) | ∃y Loves(y,x)]")

print("Example 2: ~(p >> q) | (r & s)")

print("------------------------------------------------------------")

fol = input("Enter FOL formula: ").strip()

try:

result = Skolemization(fol\_to\_cnf(fol))

print("\nSimplified CNF form:")

print(clean\_output(result))

except Exception as e:

print("\nError: Could not parse the formula.")

print("Details:", e)

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

main()

Output:

