**VISVESVARAYA TECHNOLOGICAL UNIVERSITY**

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



**LAB REPORT**

**on**

Artificial Intelligence (23CS5PCAIN)

***Submitted by***

**NAVNEETH K S (1BM22CS174)**

***in partial fulfillment for the award of the degree of***

**BACHELOR OF ENGINEERING**

***in***

**COMPUTER SCIENCE AND ENGINEERING**

**Swathi Sridharan**

**Professor**

**Department of Computer Science and Engineering**



**B.M.S. COLLEGE OF ENGINEERING**

**(Autonomous Institution under VTU)**

**BENGALURU-560019**

**Sep-2024 to Jan-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)” was carried out by NAVNEETH K S **(1BM22CS174),** 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. Kavitha Sooda Professor & HOD

Department of CSE, BMSCE

Swathi Sridharan

Assistant Professor

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 | 4-12 |
| 2 | 7-10-2024 | Implement 8 puzzle problems using Depth First Search (DFS)  Implement Iterative deepening search algorithm | 13-16 |
| 3 | 14-10-2024 | Implement A\* search algorithm | 17-21 |
| 4 | 21-10-2024 | Implement Hill Climbing search algorithm to solve N-Queens problem | 22-34 |
| 5 | 28-10-2024 | Simulated Annealing to Solve 8-Queens problem | 35-40 |
| 6 | 11-11-2024 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 41-47 |
| 7 | 2-12-2024 | Implement unification in first order logic | 48-54 |
| 8 | 2-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 55-58 |
| 9 | 16-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 59-62 |
| 10 | 16-12-2024 | Implement Alpha-Beta Pruning. | 63-66 |

Github Link: https://github.com/navneeth-dev/AI

# Program 1

Implement Tic –Tac –Toe Game Implement vacuum cleaner agent Tic-Tac-Toe

Algorithm:

A piece of paper with writing on it

Description automatically generatedA piece of paper with writing on it

Description automatically generated

Code:

def check\_win(board, r, c):

if board[r - 1][c - 1] == 'X': ch = "O"

else:

ch = "X"

if ch not in board[r - 1] and '-' not in board[r - 1]: return True

elif ch not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]) and '-' not in (board[0][c - 1], board[1][c - 1], board[2][c - 1]):

return True

elif ch not in (board[0][0], board[1][1], board[2][2]) and '-' not in (board[0][0], board[1][1], board[2][2]):

return True

elif ch not in (board[0][2], board[1][1], board[2][0]) and '-' not in (board[0][2], board[1][1], board[2][0]):

return True return False

def displayb(board):

print(board[0]) print(board[1]) print(board[2])

board=[['-','-','-'],['-','-','-'],['-','-','-']]

displayb(board) xo=1

flag=0

while '-' in board[0] or '-' in board[1] or '-' in board[2]:

if xo==1:

print("enter position to place X:") x=int(input())

y=int(input()) if(x>3 or y>3):

print("invalid position") continue

if(board[x-1][y-1]=='-'): board[x-1][y-1]='X' xo=0

displayb(board) else:

print("invalid position") continue if(check\_win(board,x,y)):

print("X wins") flag=1

break else :

print("enter position to place O:") x=int(input())

y=int(input()) if(x>3 or y>3):

print("invalid position") continue

if(board[x-1][y-1]=='-'): board[x-1][y-1]='O' xo=1

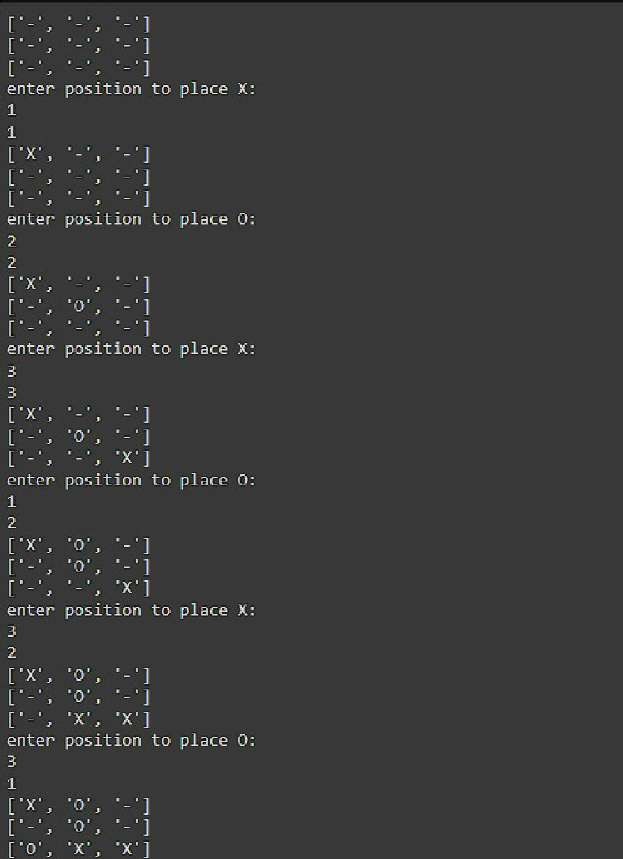
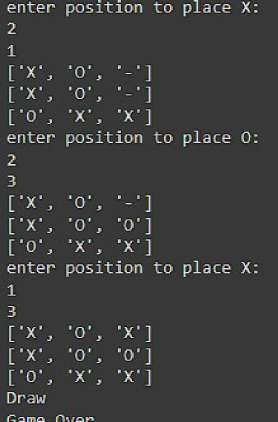
displayb(board) else:

print("invalid position") continue if(check\_win(board,x,y)):

print("0 wins") flag=1

break if flag==0:

print("Draw") print("Game Over")



Vacuum Cleaner

Algorithm:

A paper with writing on it

Description automatically generated

Code:

count = 0

def rec(state, loc): global count

if state['A'] == 0 and state['B'] == 0: print("Turning vacuum off") return

if state[loc] == 1: state[loc] = 0

count += 1 print(f"Cleaned {loc}.")

next\_loc = 'B' if loc == 'A' else 'A'

state[loc] = int(input(f"Is {loc} clean now? (0 if clean, 1 if dirty): ")) if(state[next\_loc]!=1):

state[next\_loc]=int(input(f"Is {next\_loc} dirty? (0 if clean, 1 if dirty): ")) if(state[loc]==1):

rec(state,loc) else:

next\_loc = 'B' if loc == 'A' else 'A' dire="left" if loc=="B" else "right" print(loc,"is clean") print(f"Moving vacuum {dire}")

if state[next\_loc] == 1: rec(state, next\_loc)

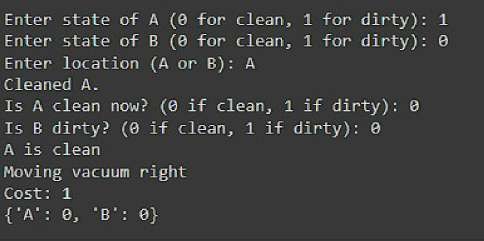
state = {}

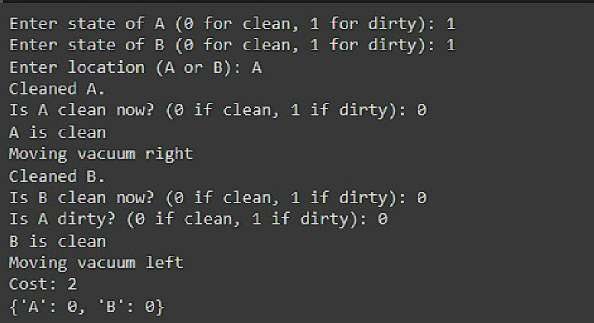
state['A'] = int(input("Enter state of A (0 for clean, 1 for dirty): ")) state['B'] = int(input("Enter state of B (0 for clean, 1 for dirty): ")) loc = input("Enter location (A or B): ")

A computer screen shot of a black screen

Description automatically generatedA computer screen with white text

Description automatically generatedrec(state, loc) print("Cost:",count) print(state



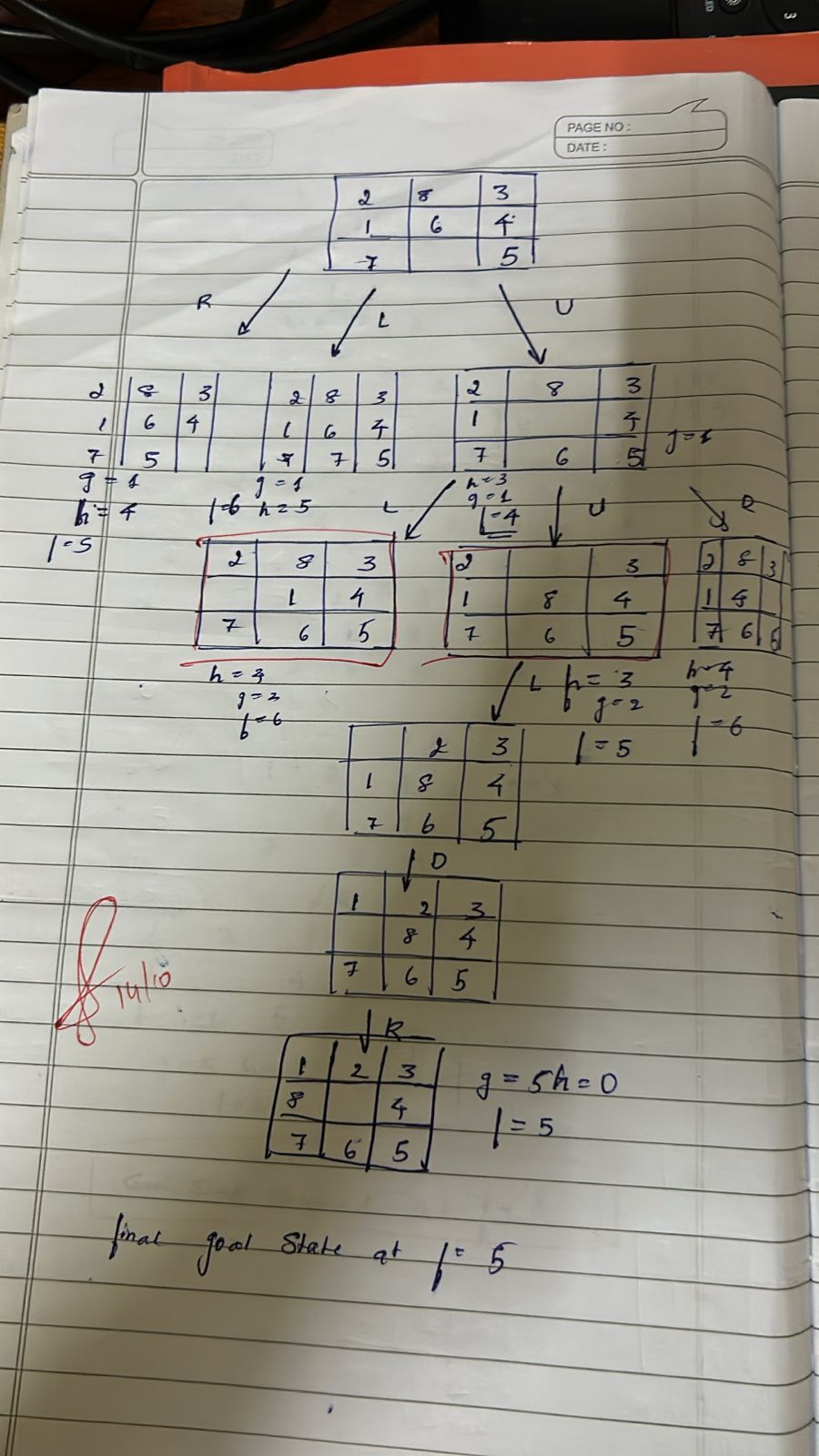


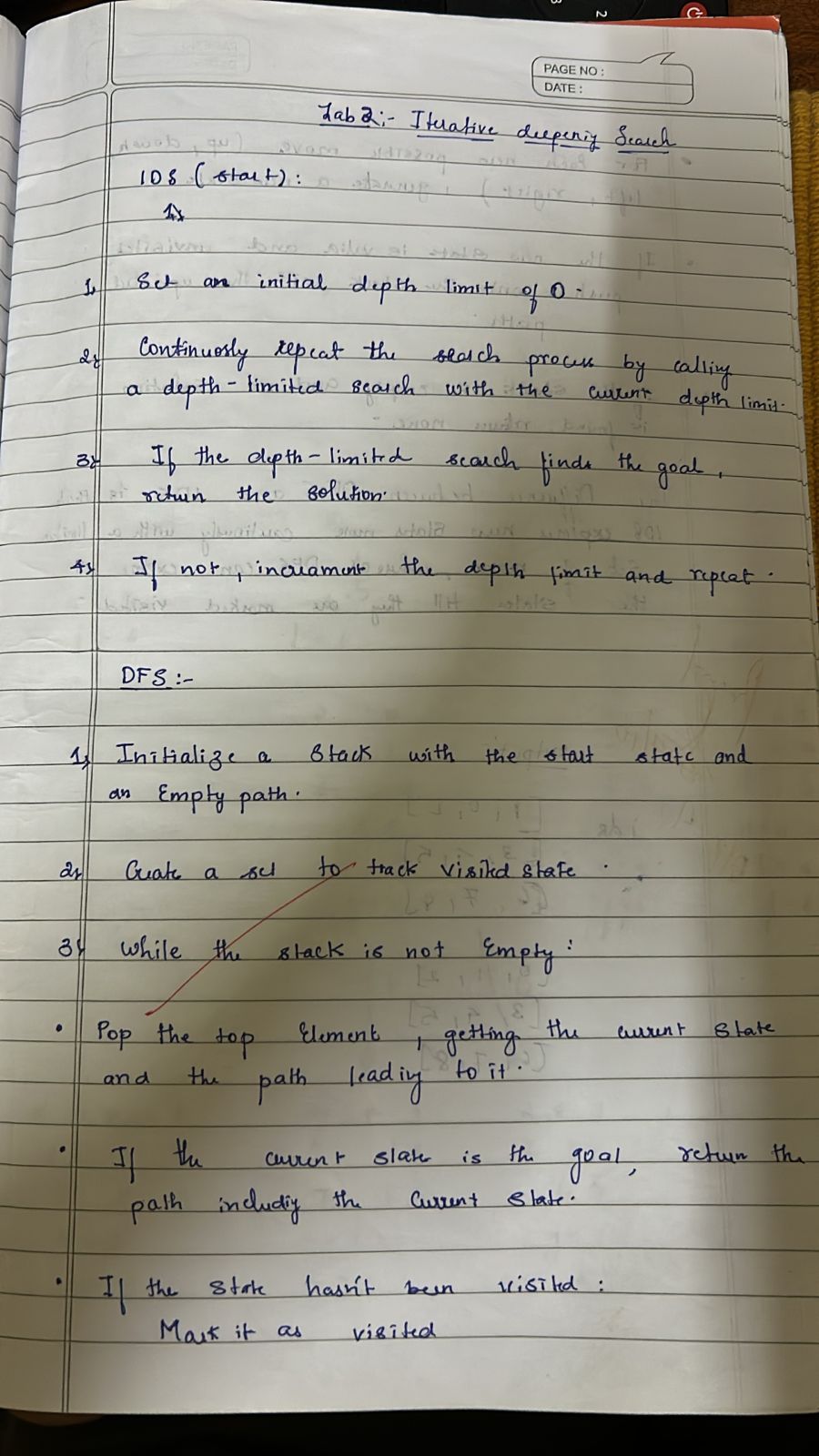
# Program 2

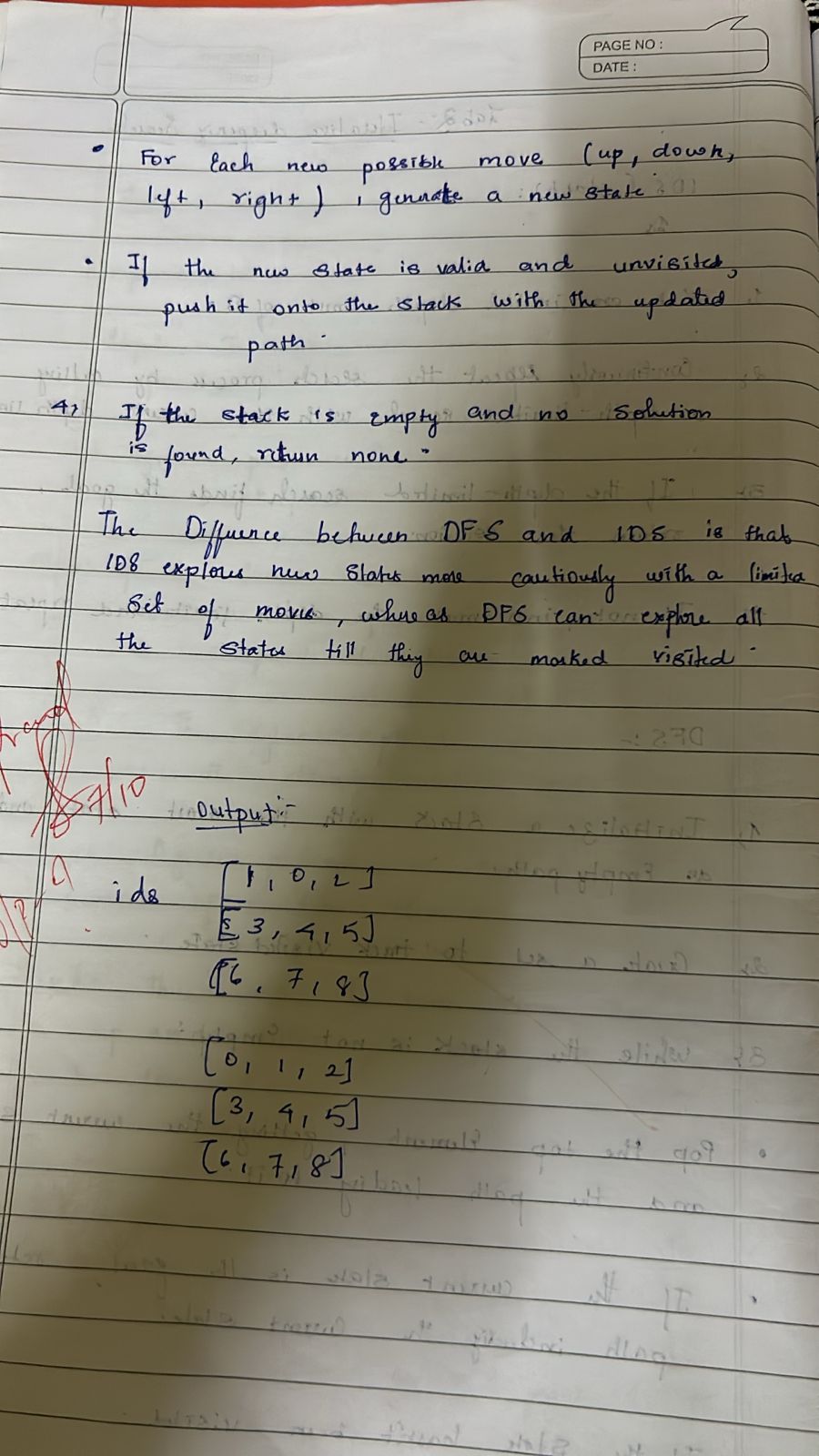
Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm

8 puzzles using DFS

Algorithm:







Code:

def dfs(initial\_board, zero\_pos):

stack = [(initial\_board, zero\_pos, [])] visited = set()

while stack:

current\_board, zero\_pos, moves = stack.pop()

if is\_goal(current\_board):

return moves, len(moves) # Return moves and their count visited.add(tuple(current\_board))

for neighbor\_board, neighbor\_pos in get\_neighbors(current\_board, zero\_pos): if tuple(neighbor\_board) not in visited:

stack.append((neighbor\_board, neighbor\_pos, moves + [neighbor\_board]))

return None, 0 # No solution found, return count as 0

# Initial state of the puzzle initial\_board = [1, 2, 3, 0, 4, 6, 7, 5, 8]

zero\_position = (1, 0) # Position of the empty tile (0)

# Solve the puzzle using DFS

solution, move\_count = dfs(initial\_board, zero\_position)

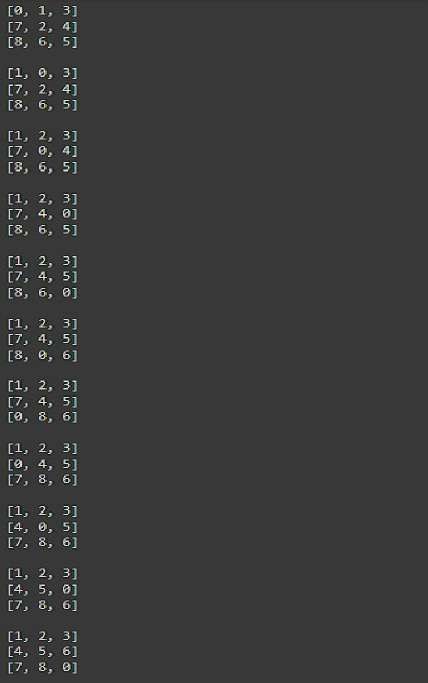
if solution:

print("Solution found with moves ({} moves):".format(move\_count)) for move in solution:

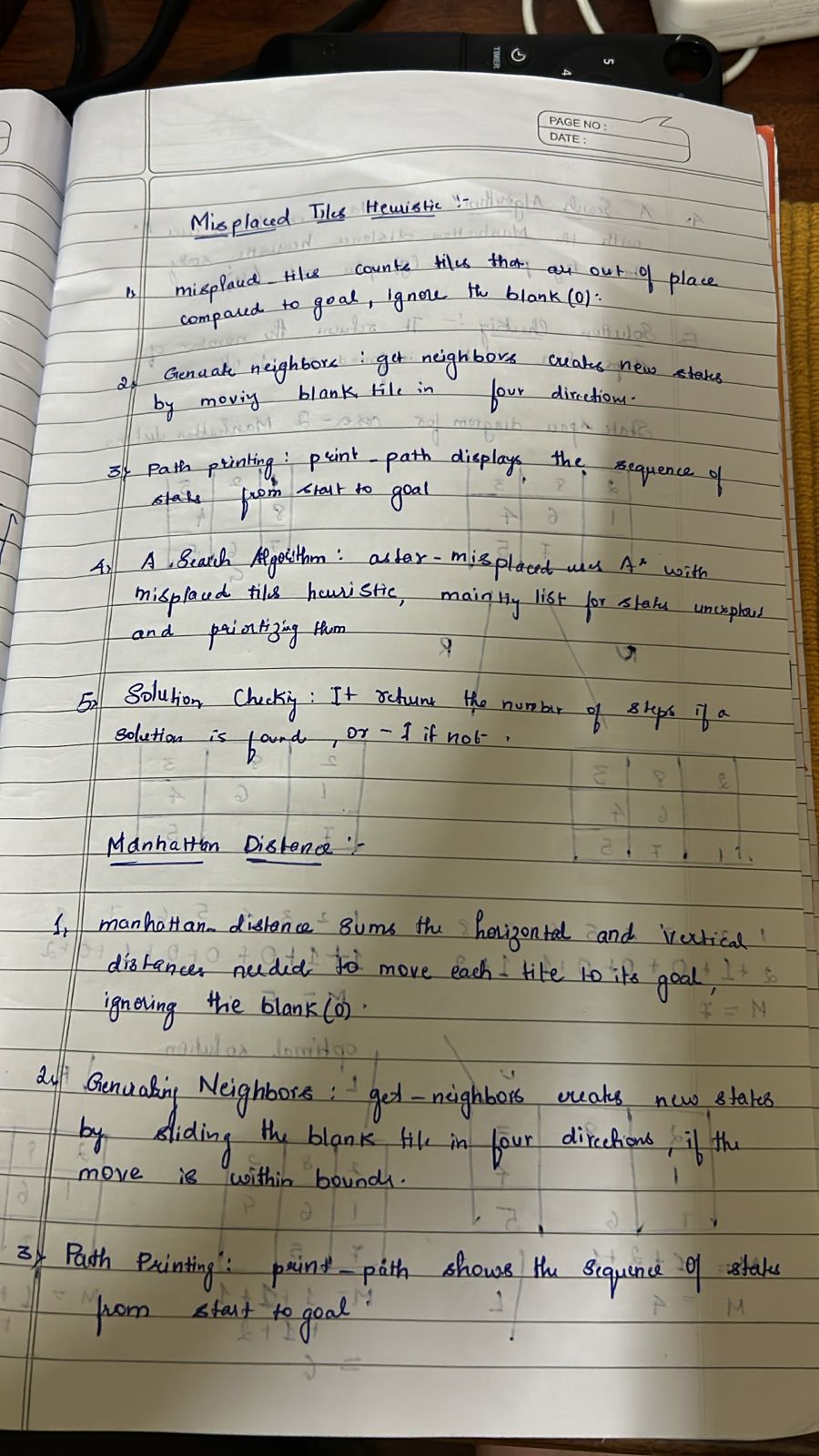
print\_board(move)

print() # Print an empty line between moves

else:

print("No solution found.")

Implement Iterative deepening search algorithm



Algorithm:

Code:

from collections import deque

class PuzzleState:

def init (self, board, zero\_pos, moves=0, previous=None): self.board = board

self.zero\_pos = zero\_pos # Position of the zero tile

self.moves = moves # Number of moves taken to reach this state self.previous = previous # For tracking the path

def is\_goal(self, goal\_state): return self.board == goal\_state

def get\_possible\_moves(self): moves = []

x, y = self.zero\_pos

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)] # Up, Down, Left, Right for dx, dy in directions:

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

if 0 <= new\_x < 3 and 0 <= new\_y < 3: new\_board = [row[:] for row in self.board]

# Swap the zero tile with the adjacent tile

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

moves.append((new\_board, (new\_x, new\_y))) return moves

def ids(initial\_state, goal\_state, max\_depth): for depth in range(max\_depth):

visited = set()

result = dls(initial\_state, goal\_state, depth, visited) if result:

return result return None

def dls(state, goal\_state, depth, visited): if state.is\_goal(goal\_state):

return state if depth == 0:

return None

visited.add(tuple(map(tuple, state.board))) # Mark this state as visited for new\_board, new\_zero\_pos in state.get\_possible\_moves():

new\_state = PuzzleState(new\_board, new\_zero\_pos, state.moves + 1, state) if tuple(map(tuple, new\_board)) not in visited:

result = dls(new\_state, goal\_state, depth - 1, visited) if result:

return result

visited.remove(tuple(map(tuple, state.board))) # Unmark this state return None

def print\_solution(solution): path = []

while solution: path.append(solution.board) solution = solution.previous

for board in reversed(path): for row in board:

print(row) print()

# Define the initial state and goal state initial\_state = PuzzleState(

board=[[1, 2, 3],

[4, 0, 5],

[7, 8, 6]],

zero\_pos=(1, 1)

)

goal\_state = [ [1, 2, 3],

[4, 5, 6],

[7, 8, 0]

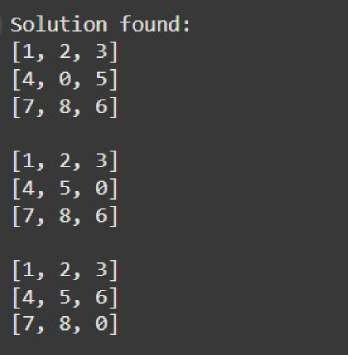
]

# Perform Iterative Deepening Search max\_depth = 20 # You can adjust this value

solution = ids(initial\_state, goal\_state, max\_depth)

if solution: print("Solution found:") print\_solution(solution)

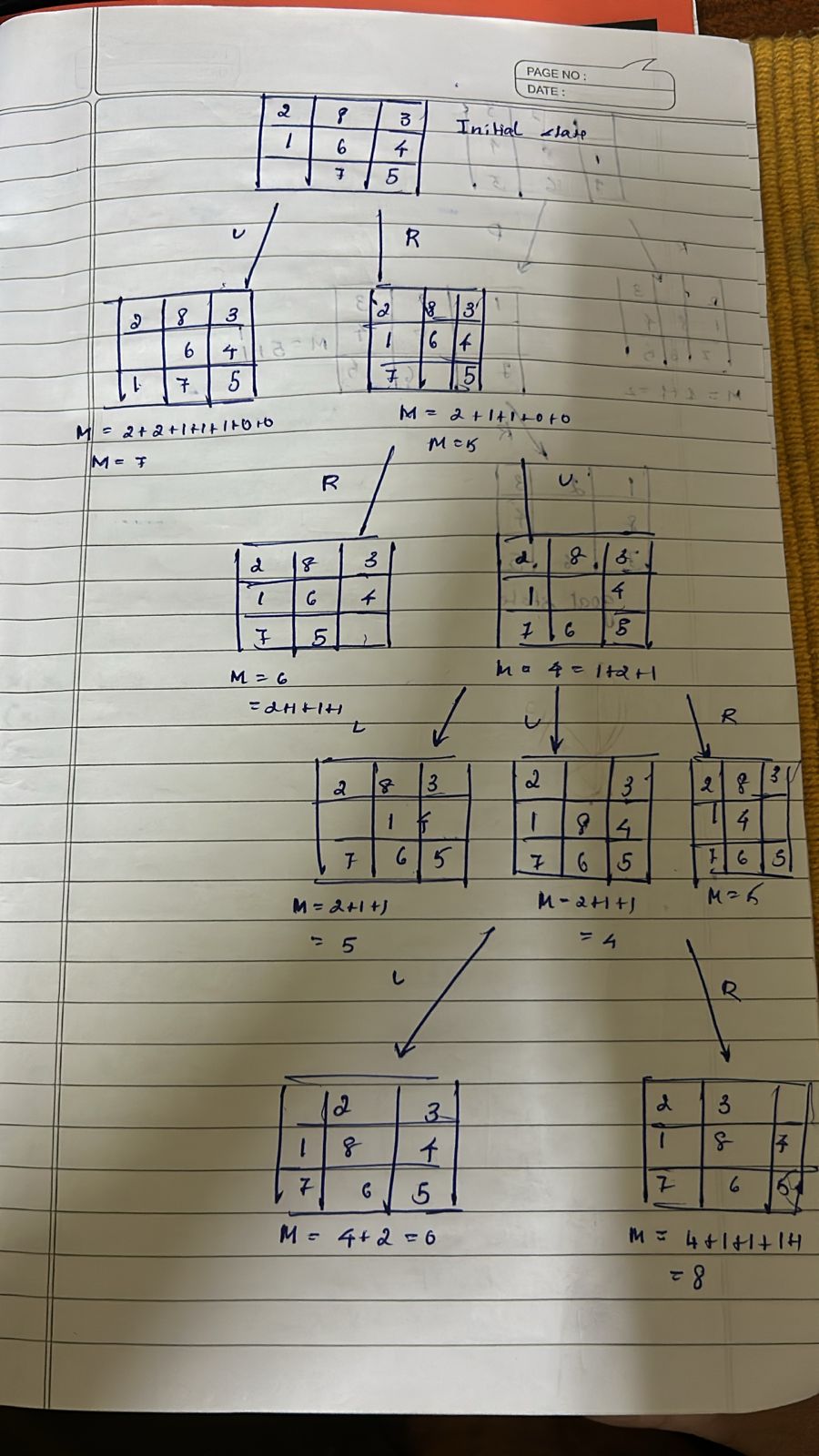
else:

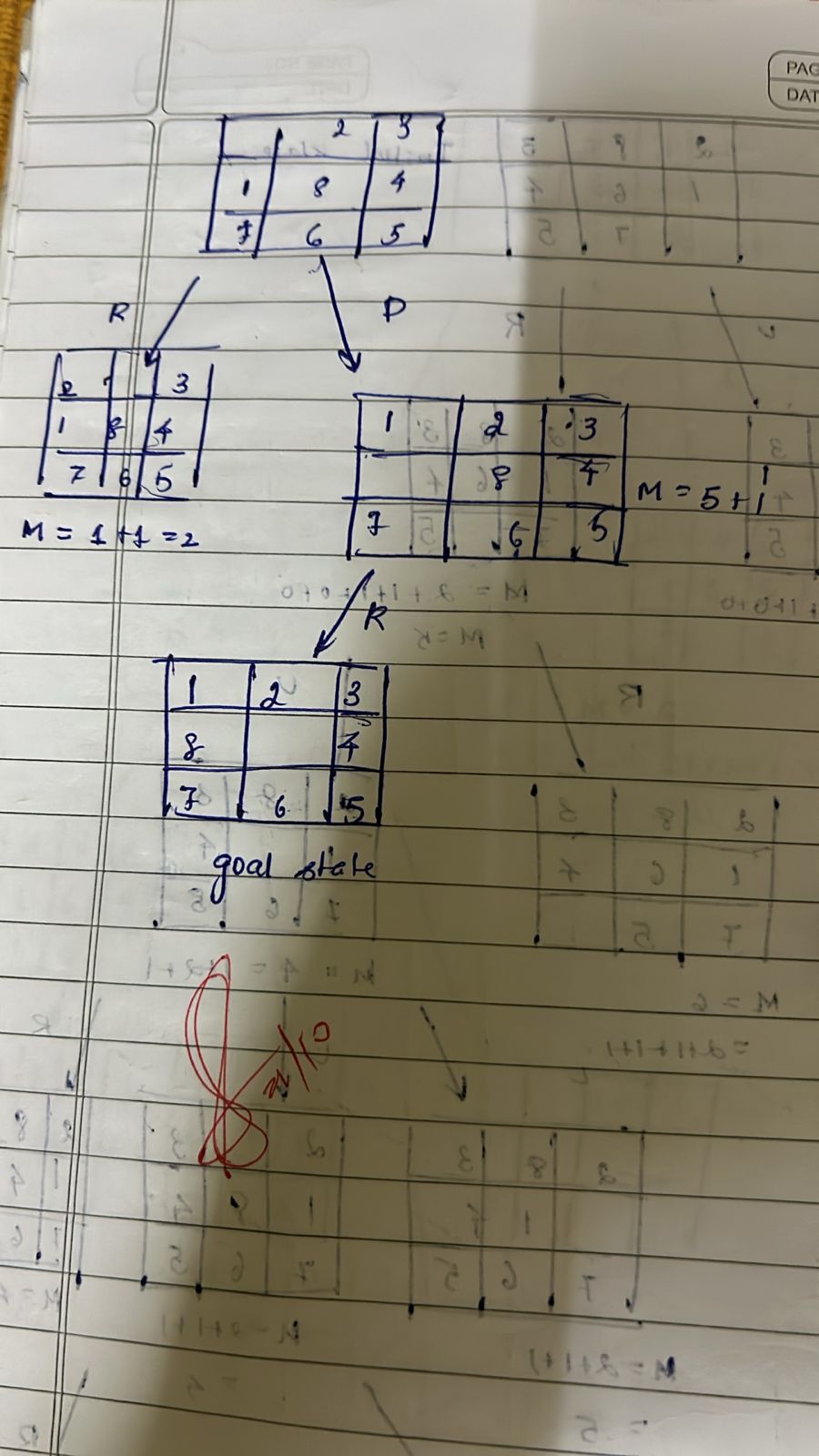
print("No solution found.")

# Program 3

Implement A\* search algorithm

Algorithm:





Code:

Misplaced Tiles

def mistil(state, goal): count = 0

for i in range(3): for j in range(3):

if state[i][j] != goal[i][j]: count += 1

return count

def findmin(open\_list, goal): minv = float('inf') best\_state = None

for state in open\_list:

h = mistil(state['state'], goal) f = state['g'] + h

if f < minv: minv = f

best\_state = state open\_list.remove(best\_state) return best\_state

def operation(state): next\_states = []

blank\_pos = find\_blank\_position(state['state']) for move in ['up', 'down', 'left', 'right']:

new\_state = apply\_move(state['state'], blank\_pos, move) if new\_state:

next\_states.append({ 'state': new\_state, 'parent': state, 'move': move,

'g': state['g'] + 1

})

return next\_states

def find\_blank\_position(state): for i in range(3):

for j in range(3):

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

return None

def apply\_move(state, blank\_pos, move): i, j = blank\_pos

new\_state = [row[:] for row in state] if move == 'up' and i > 0:

new\_state[i][j], new\_state[i - 1][j] = new\_state[i - 1][j], new\_state[i][j] elif move == 'down' and i < 2:

new\_state[i][j], new\_state[i + 1][j] = new\_state[i + 1][j], new\_state[i][j] elif move == 'left' and j > 0:

new\_state[i][j], new\_state[i][j - 1] = new\_state[i][j - 1], new\_state[i][j] elif move == 'right' and j < 2:

new\_state[i][j], new\_state[i][j + 1] = new\_state[i][j + 1], new\_state[i][j] else:

return None return new\_state

def print\_state(state): for row in state:

print(' '.join(map(str, row)))

initial\_state = [[2,8,3], [1,6,4], [7,0,5]]

goal\_state = [[1,2,3], [8,0,4], [7,6,5]]

open\_list = [{'state': initial\_state, 'parent': None, 'move': None, 'g': 0}] visited\_states = []

while open\_list:

best\_state = findmin(open\_list, goal\_state) print("Current state:") print\_state(best\_state['state'])

h = mistil(best\_state['state'], goal\_state) f = best\_state['g'] + h

print(f"g(n): {best\_state['g']}, h(n): {h}, f(n): {f}") if best\_state['move'] is not None:

print(f"Move: {best\_state['move']}") print()

if mistil(best\_state['state'], goal\_state) == 0: goal\_state\_reached = best\_state

break visited\_states.append(best\_state['state']) next\_states = operation(best\_state)

for state in next\_states:

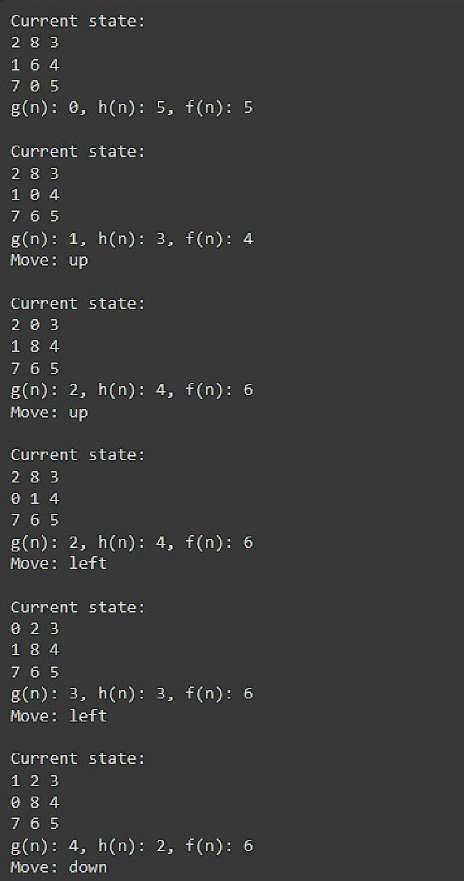
if state['state'] not in visited\_states: open\_list.append(state)

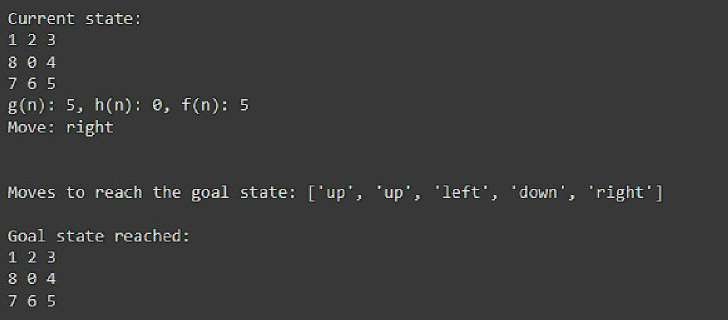
moves = []

while goal\_state\_reached['move'] is not None: moves.append(goal\_state\_reached['move']) goal\_state\_reached = goal\_state\_reached['parent']

moves.reverse()

print("\nMoves to reach the goal state:", moves) print("\nGoal state reached:")

print\_state(goal\_state)



Manhattan Distance

def manhattan\_distance(state, goal): distance = 0

for i in range(3): for j in range(3):

tile = state[i][j]

if tile != 0: # Ignore the blank space (0)

# Find the position of the tile in the goal state for r in range(3):

for c in range(3):

if goal[r][c] == tile: target\_row, target\_col = r, c break

# Add the Manhattan distance (absolute difference in rows and columns) distance += abs(target\_row - i) + abs(target\_col - j)

return distance

def findmin(open\_list, goal): minv = float('inf') best\_state = None

for state in open\_list:

h = manhattan\_distance(state['state'], goal) # Use Manhattan distance here f = state['g'] + h

if f < minv: minv = f

best\_state = state open\_list.remove(best\_state) return best\_state

def operation(state): next\_states = []

blank\_pos = find\_blank\_position(state['state']) for move in ['up', 'down', 'left', 'right']:

new\_state = apply\_move(state['state'], blank\_pos, move) if new\_state:

next\_states.append({ 'state': new\_state, 'parent': state, 'move': move,

'g': state['g'] + 1

})

return next\_states

def find\_blank\_position(state): for i in range(3):

for j in range(3):

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

return i, j return None

def apply\_move(state, blank\_pos, move): i, j = blank\_pos

new\_state = [row[:] for row in state] if move == 'up' and i > 0:

new\_state[i][j], new\_state[i - 1][j] = new\_state[i - 1][j], new\_state[i][j] elif move == 'down' and i < 2:

new\_state[i][j], new\_state[i + 1][j] = new\_state[i + 1][j], new\_state[i][j] elif move == 'left' and j > 0:

new\_state[i][j], new\_state[i][j - 1] = new\_state[i][j - 1], new\_state[i][j] elif move == 'right' and j < 2:

new\_state[i][j], new\_state[i][j + 1] = new\_state[i][j + 1], new\_state[i][j] else:

return None return new\_state

def print\_state(state): for row in state:

print(' '.join(map(str, row)))

# Initial state and goal state initial\_state = [[2,8,3], [1,6,4], [7,0,5]]

goal\_state = [[1,2,3], [8,0,4], [7,6,5]]

# Open list and visited states

open\_list = [{'state': initial\_state, 'parent': None, 'move': None, 'g': 0}] visited\_states = []

while open\_list:

best\_state = findmin(open\_list, goal\_state)

print("Current state:") print\_state(best\_state['state'])

h = manhattan\_distance(best\_state['state'], goal\_state) # Using Manhattan distance here f = best\_state['g'] + h

print(f"g(n): {best\_state['g']}, h(n): {h}, f(n): {f}")

if best\_state['move'] is not None: print(f"Move: {best\_state['move']}")

print()

if h == 0: # Goal is reached if h == 0 goal\_state\_reached = best\_state break

visited\_states.append(best\_state['state']) next\_states = operation(best\_state)

for state in next\_states:

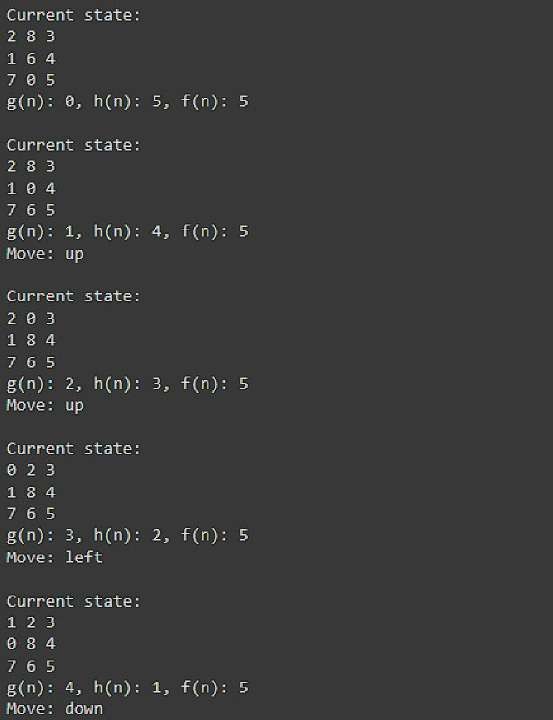
if state['state'] not in visited\_states: open\_list.append(state)

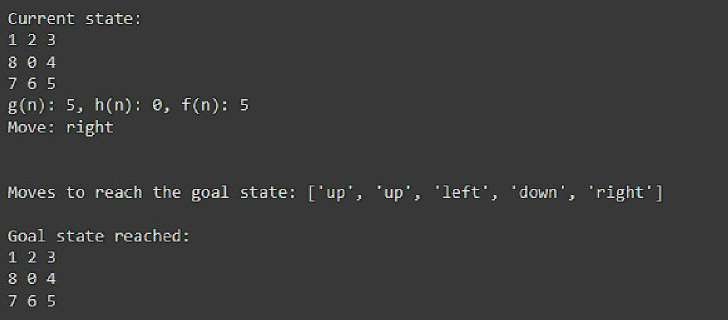
# Reconstruct the path of moves moves = []

while goal\_state\_reached['move'] is not None: moves.append(goal\_state\_reached['move']) goal\_state\_reached = goal\_state\_reached['parent']

moves.reverse()

print("\nMoves to reach the goal state:", moves) print("\nGoal state reached:") print\_state(goal\_state)

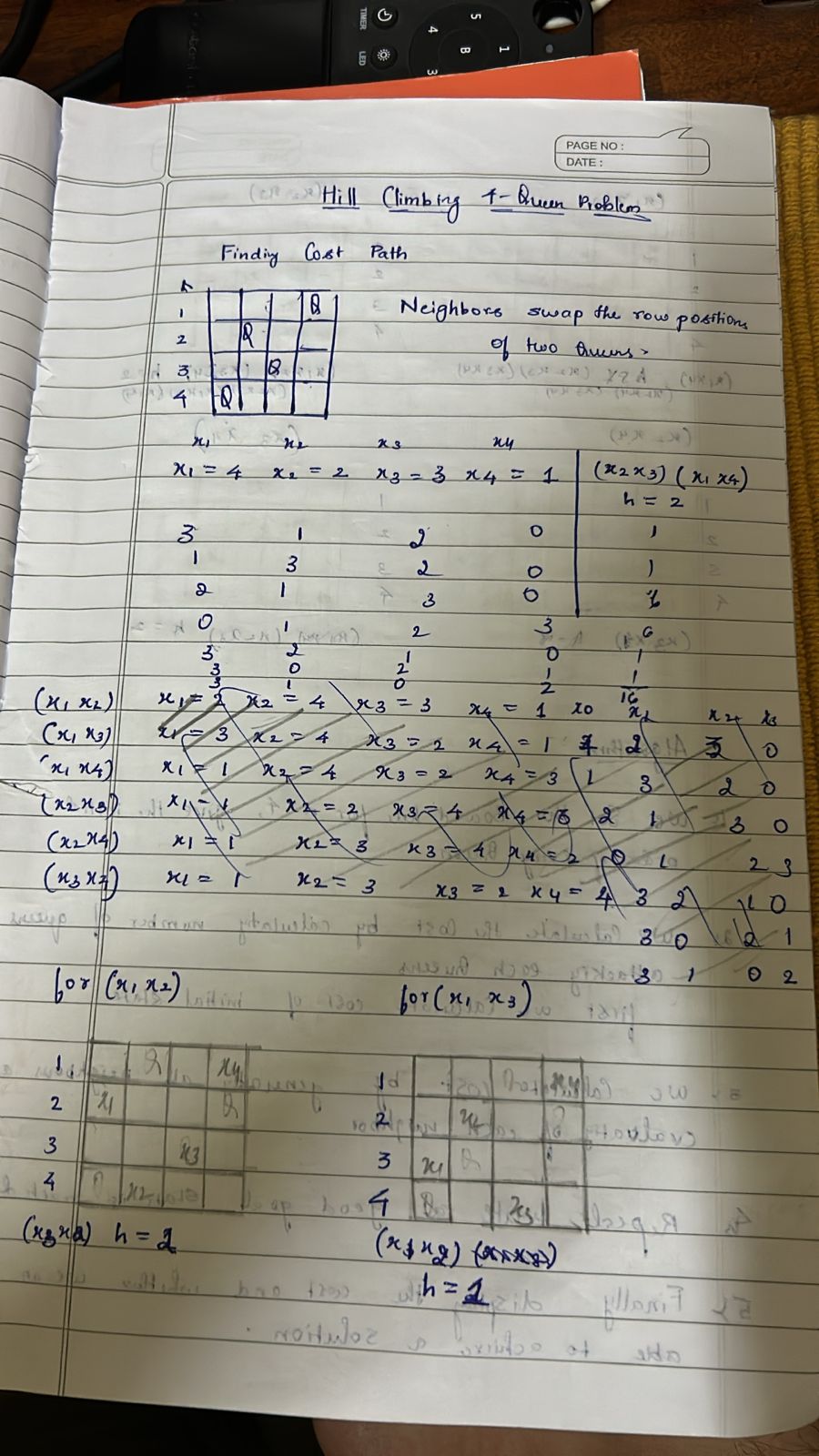


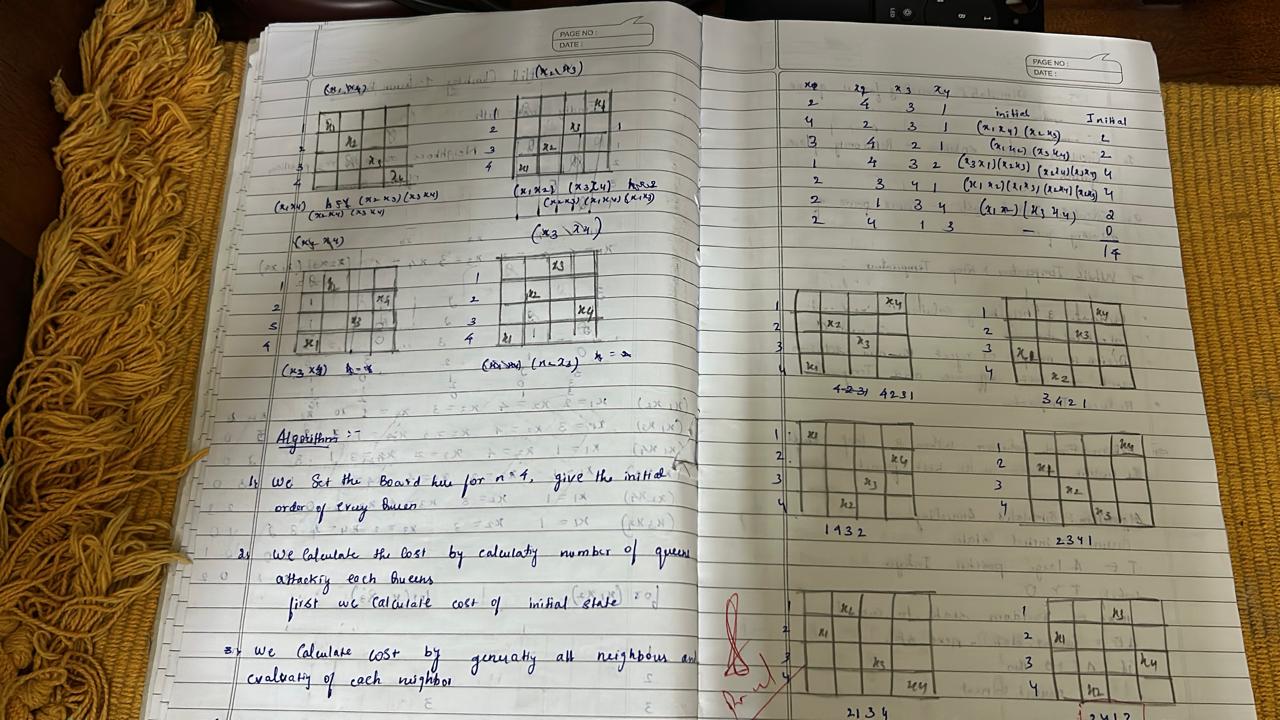


# Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





Code:

import random

def calculate\_conflicts(board): conflicts = 0

n = len(board) for i in range(n):

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

if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j): conflicts += 1

return conflicts

def hill\_climbing(n): cost=0

while True:

# Initialize a random board current\_board = list(range(n)) random.shuffle(current\_board)

current\_conflicts = calculate\_conflicts(current\_board)

while True:

# Generate neighbors by moving each queen to a different position found\_better = False

for i in range(n): for j in range(n):

if j != current\_board[i]: # Only consider different positions neighbor\_board = list(current\_board)

neighbor\_board[i] = j

neighbor\_conflicts = calculate\_conflicts(neighbor\_board) if neighbor\_conflicts < current\_conflicts:

print\_board(current\_board) print(current\_conflicts) print\_board(neighbor\_board) print(neighbor\_conflicts) current\_board = neighbor\_board current\_conflicts = neighbor\_conflicts cost+=1

found\_better = True break

if found\_better: break

# If no better neighbor found, stop searching if not found\_better:

break

# If a solution is found (zero conflicts), return the board if current\_conflicts == 0:

return current\_board, current\_conflicts, cost

def print\_board(board): n = len(board)

for i in range(n): row = ['.'] \* n

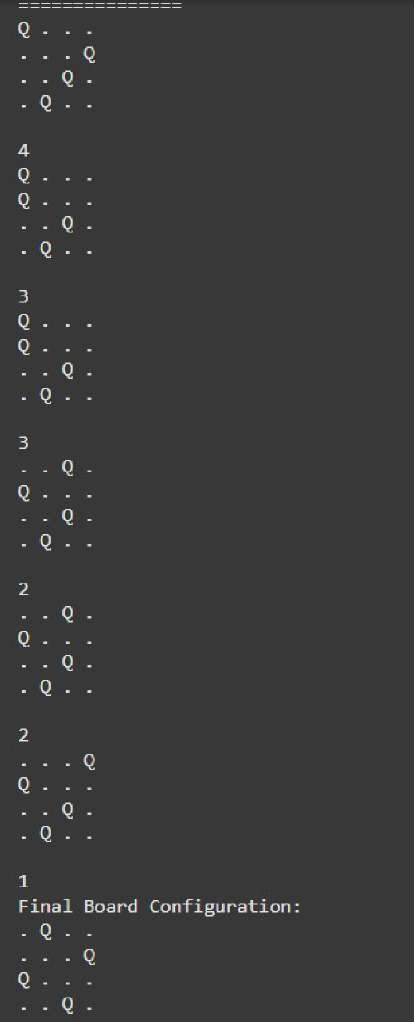
row[board[i]] = 'Q' # Place a queen print(' '.join(row))

print()

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

# Example Usage n = 4

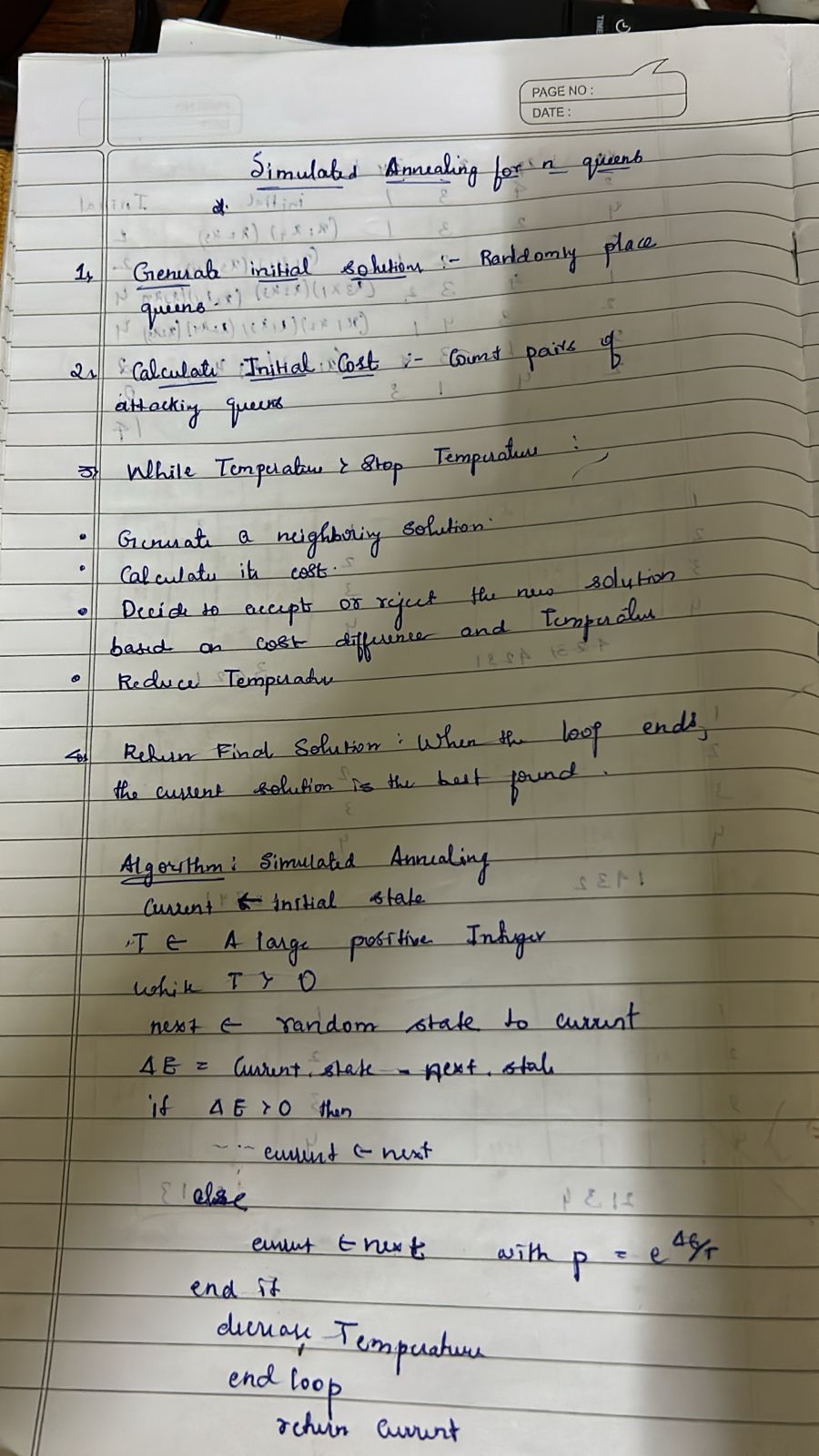
solution, conflicts, cost = hill\_climbing(n) print("Final Board Configuration:") print\_board(solution)

print("Number of Cost:", cost)

# Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

import numpy as np

from scipy.optimize import dual\_annealing

def queens\_max(position):

# This function calculates the number of pairs of queens that are not attacking each other position = np.round(position).astype(int) # Round and convert to integers for queen positions n = len(position)

queen\_not\_attacking = 0

for i in range(n - 1): no\_attack\_on\_j = 0 for j in range(i + 1, n):

# Check if queens are on the same row or on the same diagonal

if position[i] != position[j] and abs(position[i] - position[j]) != (j - i): no\_attack\_on\_j += 1

if no\_attack\_on\_j == n - 1 - i: queen\_not\_attacking += 1

if queen\_not\_attacking == n - 1: queen\_not\_attacking += 1

return -queen\_not\_attacking # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard) bounds = [(0, 8) for \_ in range(8)]

# Use dual\_annealing for simulated annealing optimization result = dual\_annealing(queens\_max, bounds)

# Display the results

best\_position = np.round(result.x).astype(int)

best\_objective = -result.fun # Flip sign to get the number of non-attacking queens

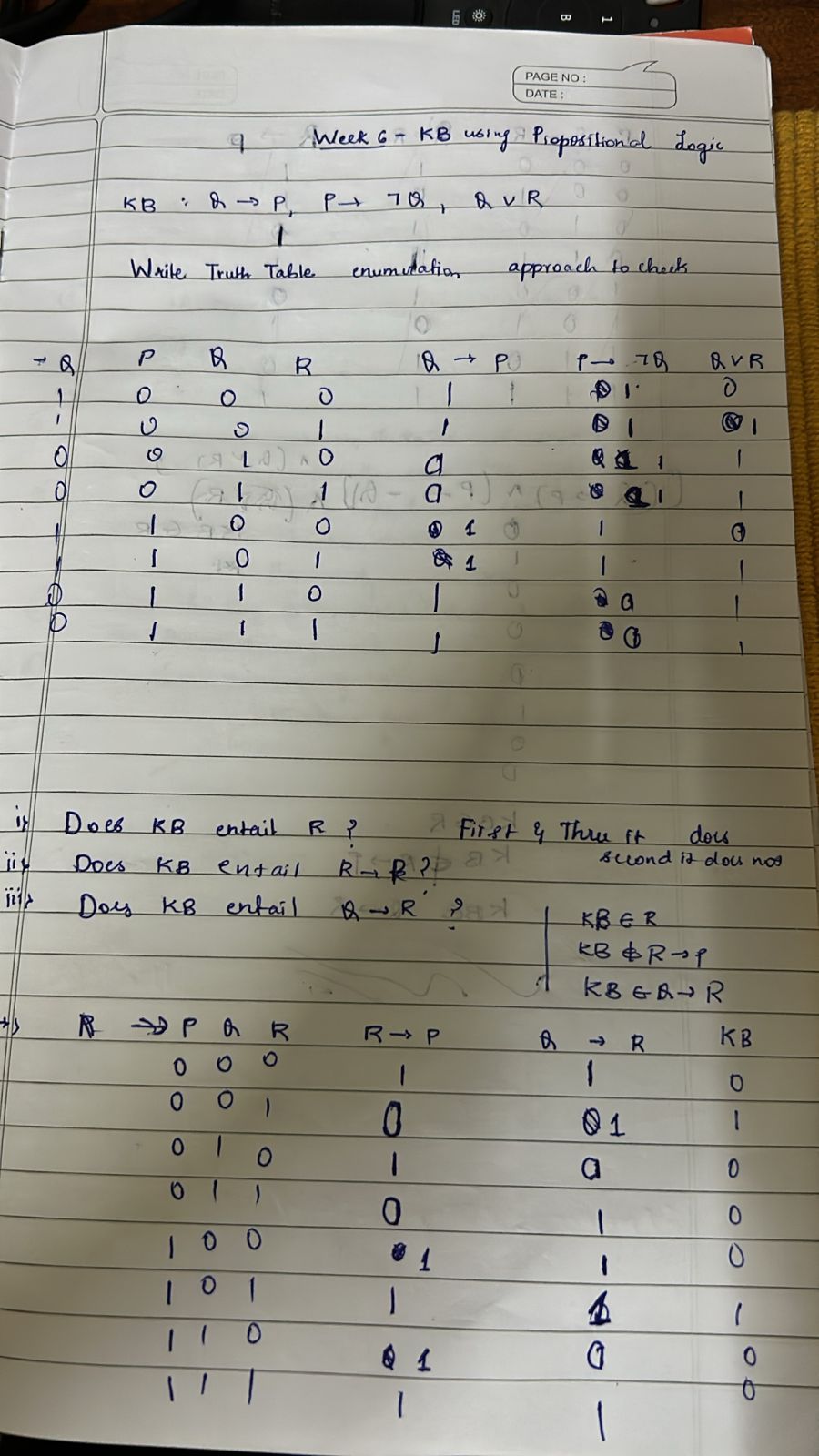
print('The best position found is:', best\_position)

print('The number of queens that are not attacking each other is:', best\_objective)

# Program 6

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

Algorithm:



Code:

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

import itertools

# Function to evaluate an expression

def evaluate\_expression(a, b, c, expression):

# Use eval() to evaluate the logical expression return eval(expression)

# Function to generate the truth table and evaluate a logical expression def truth\_table\_and\_evaluation(kb, query):

# All possible combinations of truth values for a, b, and c truth\_values = [True, False]

combinations = list(itertools.product(truth\_values, repeat=3))

# Reverse the combinations to start from the bottom (False -> True) combinations.reverse()

# Header for the full truth table

print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

# Evaluate the expressions for each combination

for combination in combinations: a, b, c = combination

# Evaluate the knowledge base (KB) and query expressions kb\_result = evaluate\_expression(a, b, c, kb)

query\_result = evaluate\_expression(a, b, c, query)

# Replace True/False with string "True"/"False" kb\_result\_str = "True" if kb\_result else "False" query\_result\_str = "True" if query\_result else "False"

# Convert boolean values of a, b, c to "True"/"False" a\_str = "True" if a else "False"

b\_str = "True" if b else "False" c\_str = "True" if c else "False"

# Print the results for the knowledge base and the query

print(f"{a\_str:<5} {b\_str:<5} {c\_str:<5} {kb\_result\_str:<20} {query\_result\_str:<20}")

# Additional output for combinations where both KB and query are true print("\nCombinations where both KB and Query are True:") print(f"{'a':<5} {'b':<5} {'c':<5} {'KB':<20}{'Query':<20}")

# Print only the rows where both KB and Query are True for combination in combinations:

a, b, c = combination

# Evaluate the knowledge base (KB) and query expressions kb\_result = evaluate\_expression(a, b, c, kb)

query\_result = evaluate\_expression(a, b, c, query)

# If both KB and query are True, print the combination if kb\_result and query\_result:

a\_str = "True" if a else "False" b\_str = "True" if b else "False" c\_str = "True" if c else "False"

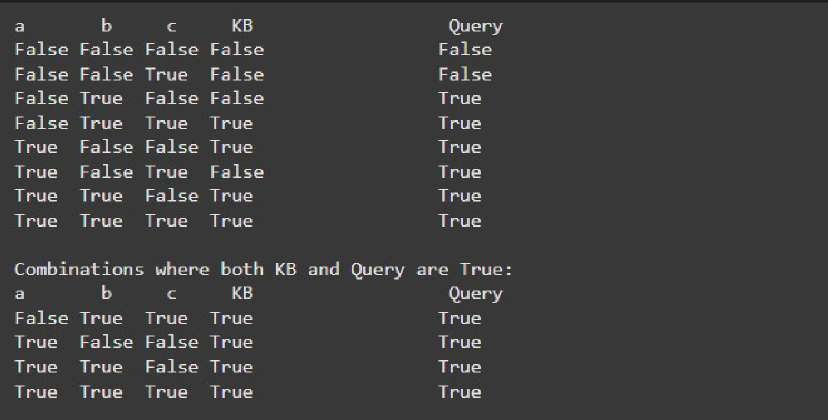
kb\_result\_str = "True" if kb\_result else "False" query\_result\_str = "True" if query\_result else "False"

print(f"{a\_str:<5} {b\_str:<5} {c\_str:<5} {kb\_result\_str:<20} {query\_result\_str:<20}")

# Define the logical expressions as strings

kb = "(a or c) and (b or not c)" # Knowledge Base query = "a or b" # Query to evaluate

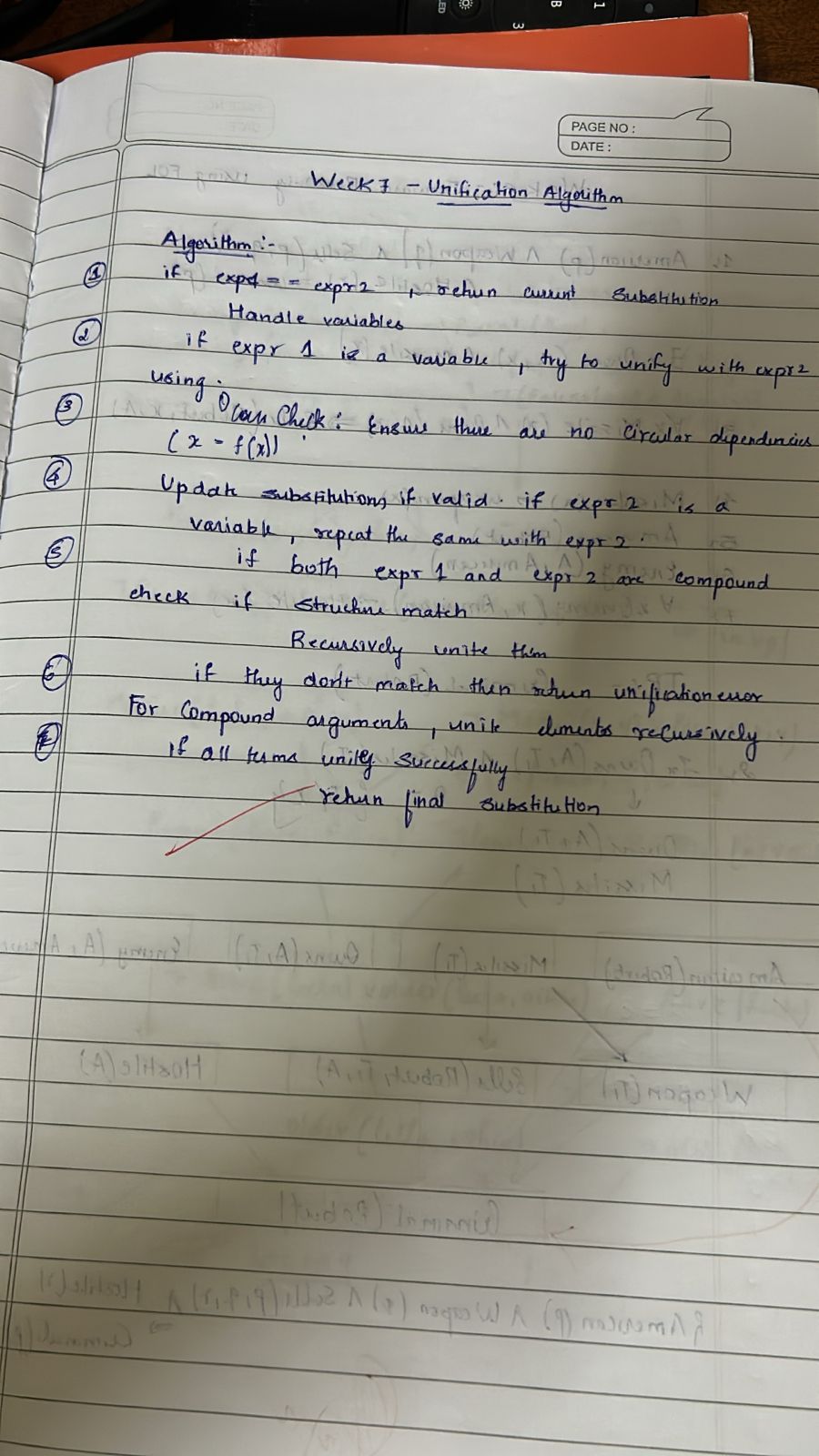
# Generate the truth table and evaluate the knowledge base and query truth\_table\_and\_evaluation(kb, query)



# Program 7

Implement unification in first order logic

Algorithm:



Code:

import re

def occurs\_check(var, x):

"""Checks if var occurs in x (to prevent circular substitutions).""" if var == x:

return True

elif isinstance(x, list): # If x is a compound expression (like a function or predicate) return any(occurs\_check(var, xi) for xi in x)

return False

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

"""Handles unification of a variable with another term."""

if var in subst: # If var is already substituted return unify(subst[var], x, subst)

elif isinstance(x, (list, tuple)) and tuple(x) in subst: # Handle compound expressions return unify(var, subst[tuple(x)], subst)

elif occurs\_check(var, x): # Check for circular references return "FAILURE"

else:

# Add the substitution to the set (convert list to tuple for hashability) subst[var] = tuple(x) if isinstance(x, list) else x

return subst

def unify(x, y, subst=None): """

Unifies two expressions x and y and returns the substitution set if they can be unified. Returns 'FAILURE' if unification is not possible.

"""

if subst is None:

subst = {} # Initialize an empty substitution set

# Step 1: Handle cases where x or y is a variable or constant if x == y: # If x and y are identical

return subst

elif isinstance(x, str) and x.islower(): # If x is a variable return unify\_var(x, y, subst)

elif isinstance(y, str) and y.islower(): # If y is a variable return unify\_var(y, x, subst)

elif isinstance(x, list) and isinstance(y, list): # If x and y are compound expressions (lists) if len(x) != len(y): # Step 3: Different number of arguments

return "FAILURE"

# Step 2: Check if the predicate symbols (the first element) match if x[0] != y[0]: # If the predicates/functions are different

return "FAILURE"

# Step 5: Recursively unify each argument

for xi, yi in zip(x[1:], y[1:]): # Skip the predicate (first element) subst = unify(xi, yi, subst)

if subst == "FAILURE": return "FAILURE"

return subst

else: # If x and y are different constants or non-unifiable structures return "FAILURE"

def unify\_and\_check(expr1, expr2): """

Attempts to unify two expressions and returns a tuple: (is\_unified: bool, substitutions: dict or None)

"""

result = unify(expr1, expr2) if result == "FAILURE":

return False, None return True, result

def display\_result(expr1, expr2, is\_unified, subst): print("Expression 1:", expr1)

print("Expression 2:", expr2) if not is\_unified:

print("Result: Unification Failed") else:

print("Result: Unification Successful")

print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

def parse\_input(input\_str):

"""Parses a string input into a structure that can be processed by the unification algorithm.""" # Remove spaces and handle parentheses

input\_str = input\_str.replace(" ", "")

# Handle compound terms (like p(x, f(y)) -> ['p', 'x', ['f', 'y']]) def parse\_term(term):

# Handle the compound term if '(' in term:

match = re.match(r'([a-zA-Z0-9\_]+)(.\*)', term) if match:

predicate = match.group(1) arguments\_str = match.group(2)

arguments = [parse\_term(arg.strip()) for arg in arguments\_str.split(',')] return [predicate] + arguments

return term

return parse\_term(input\_str)

# Main function to interact with the user def main():

while True:

# Get the first and second terms from the user

expr1\_input = input("Enter the first expression (e.g., p(x, f(y))): ") expr2\_input = input("Enter the second expression (e.g., p(a, f(z))): ")

# Parse the input strings into the appropriate structures expr1 = parse\_input(expr1\_input)

expr2 = parse\_input(expr2\_input)

# Perform unification

is\_unified, result = unify\_and\_check(expr1, expr2)

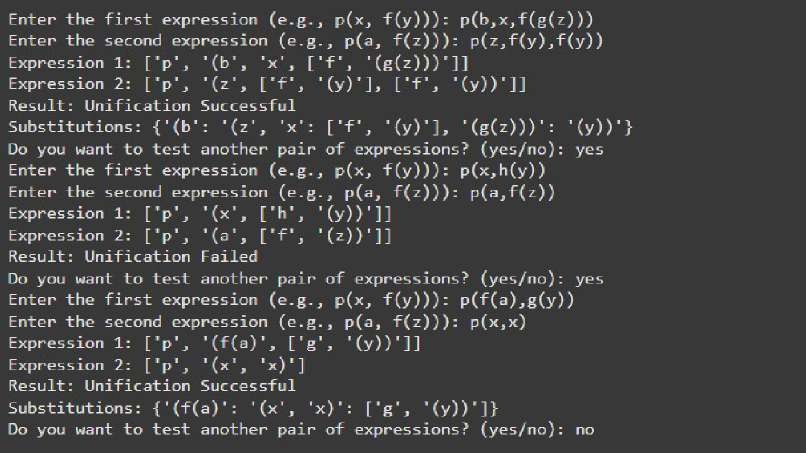
# Display the results

display\_result(expr1, expr2, is\_unified, result)

# Ask the user if they want to run another test

another\_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower() if another\_test != 'yes':

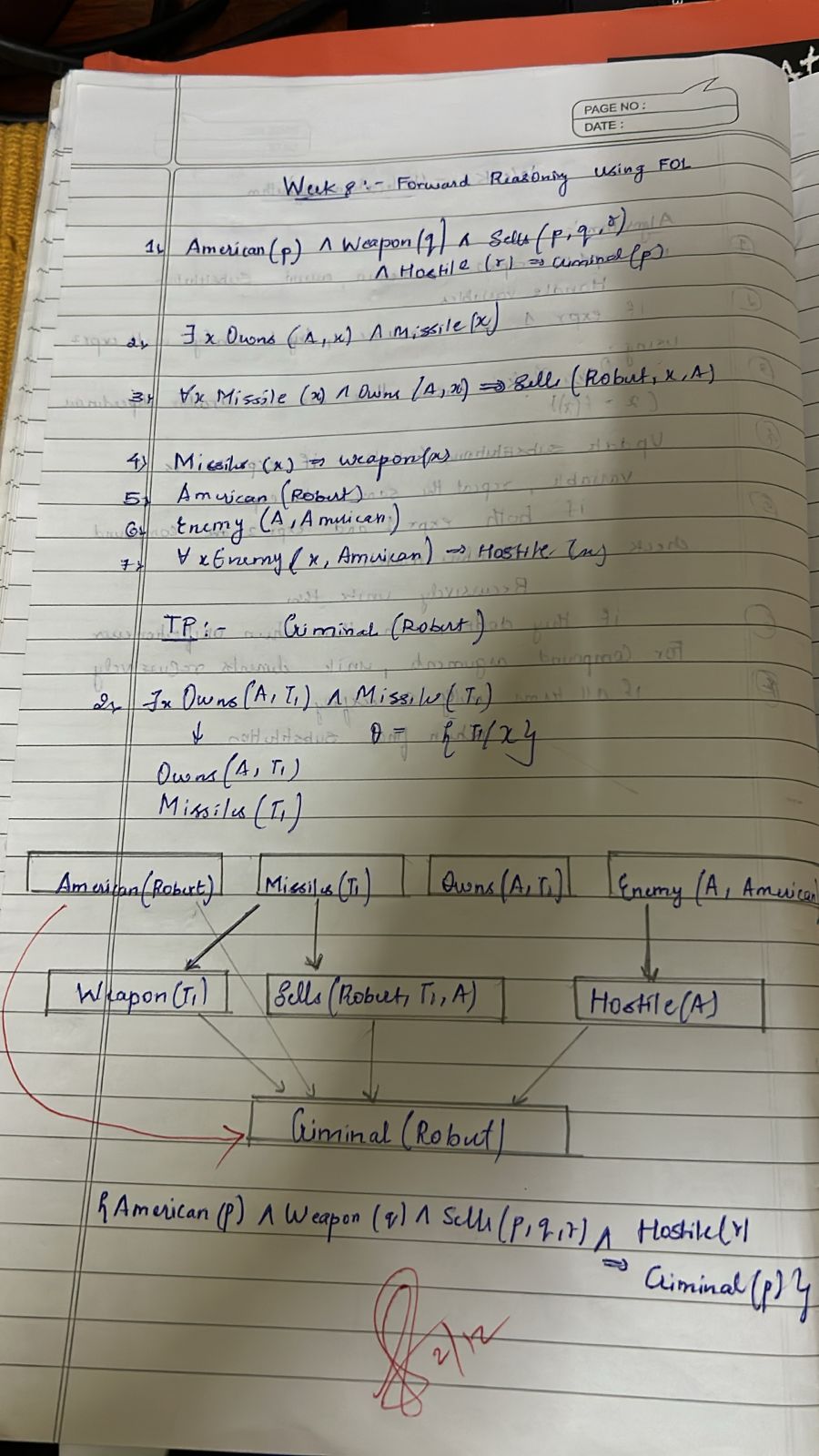
break

if name == " main ": main()

# Program 8

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

Algorithm:

:

Code:

# Define the knowledge base (KB) as a set of facts KB = set()

# Premises based on the provided FOL problem KB.add('American(Robert)') KB.add('Enemy(America, A)')

KB.add('Missile(T1)')

KB.add('Owns(A, T1)')

# Define inference rules

def modus\_ponens(fact1, fact2, conclusion):

""" Apply modus ponens inference rule: if fact1 and fact2 are true, then conclude conclusion """ if fact1 in KB and fact2 in KB:

KB.add(conclusion) print(f"Inferred: {conclusion}")

def forward\_chaining():

""" Perform forward chaining to infer new facts until no more inferences can be made """ # 1. Apply: Missile(x) → Weapon(x)

if 'Missile(T1)' in KB: KB.add('Weapon(T1)') print(f"Inferred: Weapon(T1)")

# 2. Apply: Sells(Robert, T1, A) from Owns(A, T1) and Weapon(T1) if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:

KB.add('Sells(Robert, T1, A)') print(f"Inferred: Sells(Robert, T1, A)")

# 3. Apply: Hostile(A) from Enemy(A, America) if 'Enemy(America, A)' in KB:

KB.add('Hostile(A)') print(f"Inferred: Hostile(A)")

# 4. Now, check if the goal is reached (i.e., if 'Criminal(Robert)' can be inferred)

if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:

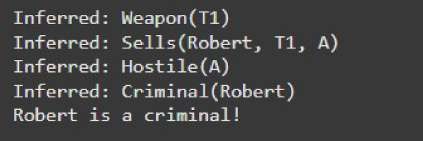
KB.add('Criminal(Robert)') print("Inferred: Criminal(Robert)")

# Check if we've reached our goal if 'Criminal(Robert)' in KB:

print("Robert is a criminal!") else:

print("No more inferences can be made.")

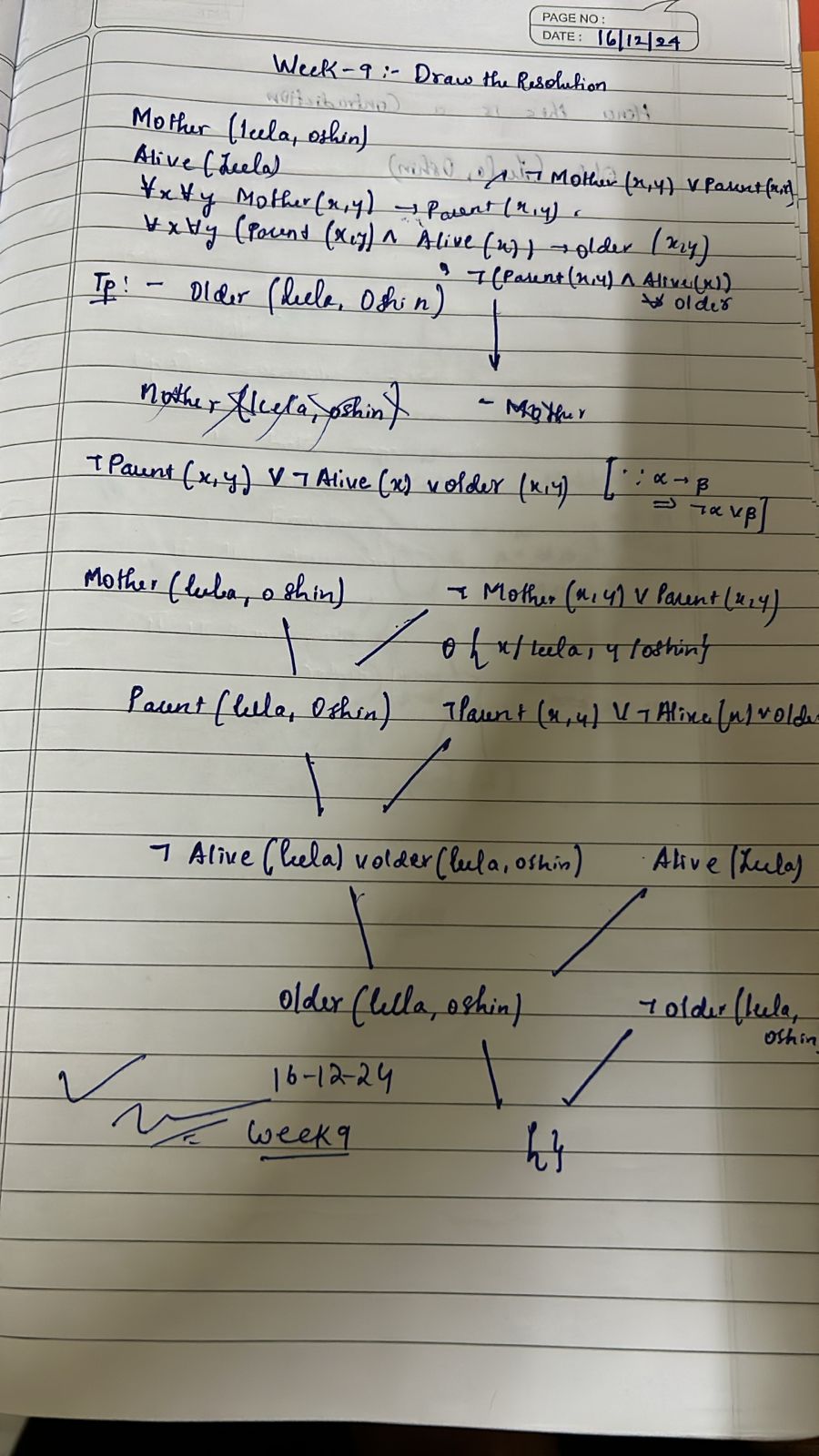
# Run forward chaining to attempt to derive the conclusion forward\_chaining()



# Program 9

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

Algorithm:



Code:

# Define the knowledge base (KB) KB = {

"food(Apple)": True, "food(vegetables)": True, "eats(Anil, Peanuts)": True, "alive(Anil)": True,

"likes(John, X)": "food(X)", # Rule: John likes all food

"food(X)": "eats(Y, X) and not killed(Y)", # Rule: Anything eaten and not killed is food "eats(Harry, X)": "eats(Anil, X)", # Rule: Harry eats what Anil eats

"alive(X)": "not killed(X)", # Rule: Alive implies not killed "not killed(X)": "alive(X)", # Rule: Not killed implies alive

}

# Function to evaluate if a predicate is true based on the KB def resolve(predicate):

# If it's a direct fact in KB

if predicate in KB and isinstance(KB[predicate], bool): return KB[predicate]

# If it's a derived rule if predicate in KB:

rule = KB[predicate]

if " and " in rule: # Handle conjunction sub\_preds = rule.split(" and ")

return all(resolve(sub.strip()) for sub in sub\_preds) elif " or " in rule: # Handle disjunction

sub\_preds = rule.split(" or ")

return any(resolve(sub.strip()) for sub in sub\_preds) elif "not " in rule: # Handle negation

sub\_pred = rule[4:] # Remove "not " return not resolve(sub\_pred.strip())

else: # Handle single predicate return resolve(rule.strip())

# If the predicate is a specific query (e.g., likes(John, Peanuts)) if "(" in predicate:

func, args = predicate.split("(") args = args.strip(")").split(", ")

if func == "food" and args[0] == "Peanuts":

return resolve("eats(Anil, Peanuts)") and not resolve("killed(Anil)") if func == "likes" and args[0] == "John" and args[1] == "Peanuts":

return resolve("food(Peanuts)")

# Default to False if no rule or fact applies return False

# Query to prove: John likes Peanuts query = "likes(John, Peanuts)"

result = resolve(query)

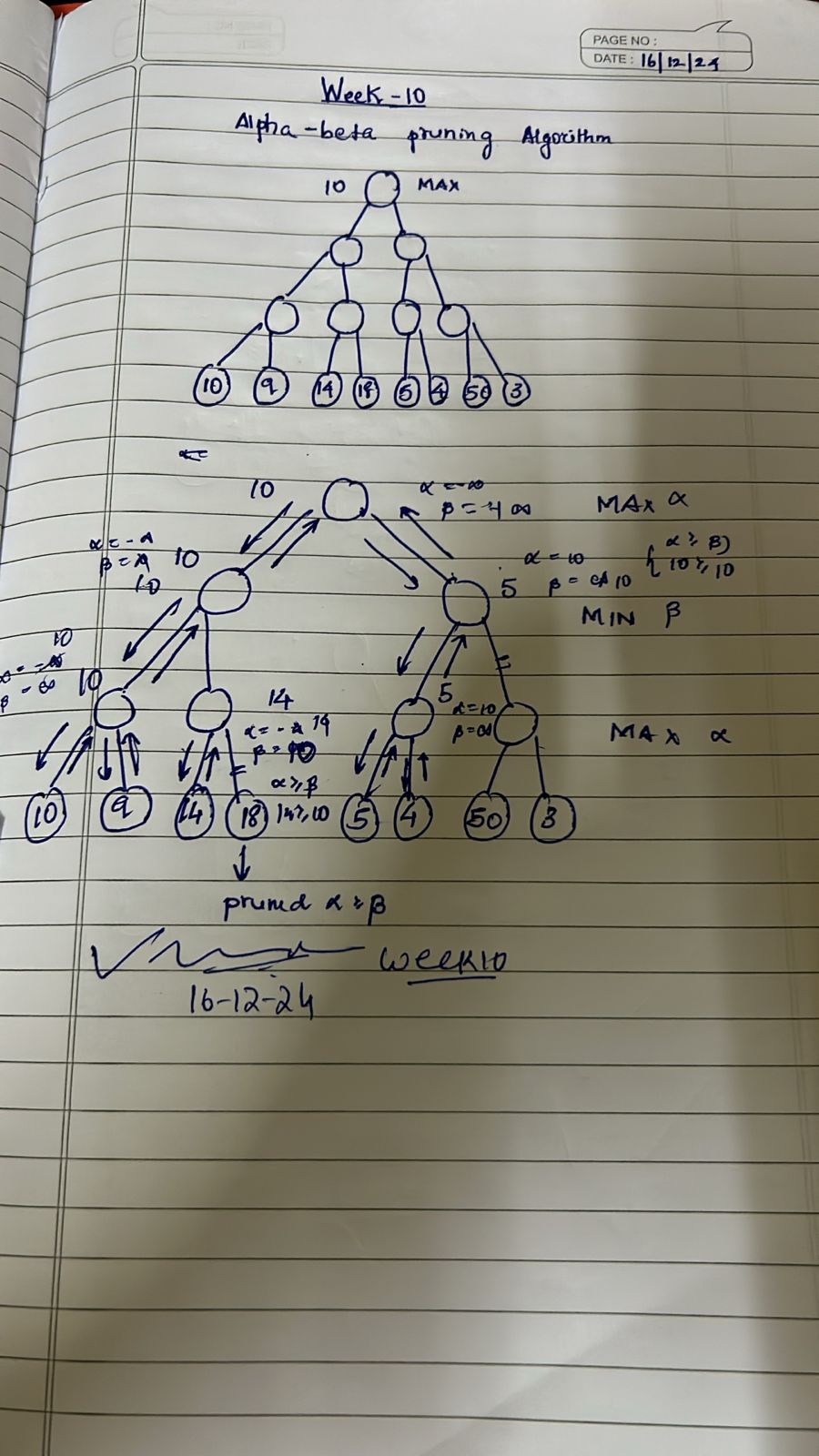
# Print the result

print(f"Does John like peanuts? {'Yes' if result else 'No'}")

# Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Code:

# Alpha-Beta Pruning Implementation

def alpha\_beta\_pruning(node, alpha, beta, maximizing\_player):

# Base case: If it's a leaf node, return its value (simulating evaluation of the node) if type(node) is int:

return node

# If not a leaf node, explore the children if maximizing\_player:

max\_eval = -float('inf')

for child in node: # Iterate over children of the maximizer node eval = alpha\_beta\_pruning(child, alpha, beta, False) max\_eval = max(max\_eval, eval)

alpha = max(alpha, eval) # Maximize alpha if beta <= alpha: # Prune the branch

break return max\_eval

else:

min\_eval = float('inf')

for child in node: # Iterate over children of the minimizer node

eval = alpha\_beta\_pruning(child, alpha, beta, True) min\_eval = min(min\_eval, eval)

beta = min(beta, eval) # Minimize beta if beta <= alpha: # Prune the branch

break return min\_eval

# Function to build the tree from a list of numbers def build\_tree(numbers):

# We need to build a tree with alternating levels of maximizers and minimizers # Start from the leaf nodes and work up

current\_level = [[n] for n in numbers]

while len(current\_level) > 1: next\_level = []

for i in range(0, len(current\_level), 2): if i + 1 < len(current\_level):

next\_level.append(current\_level[i] + current\_level[i + 1]) # Combine two nodes else:

next\_level.append(current\_level[i]) # Odd number of elements, just carry forward current\_level = next\_level

return current\_level[0] # Return the root node, which is a maximizer # Main function to run alpha-beta pruning

def main():

# Input: User provides a list of numbers

numbers = list(map(int, input("Enter numbers for the game tree (space-separated): ").split()))

# Build the tree with the given numbers tree = build\_tree(numbers)

# Parameters: Tree, initial alpha, beta, and the root node is a maximizing player alpha = -float('inf')

beta = float('inf')

maximizing\_player = True # The root node is a maximizing player

# Perform alpha-beta pruning and get the final result

result = alpha\_beta\_pruning(tree, alpha, beta, maximizing\_player) print("Final Result of Alpha-Beta Pruning:", result)

if name == " main ": main()

