VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Ayaan Shrestha (1BM23CS056)

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 Ayaan Shrestha **(1BM23CS056),** 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. Sandhya A Kulkarni Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Sl. No.	Date	Experiment Title
1	20-8-2025	Implement Tic –Tac –Toe Game Implement vacuum cleaner agent
2	3-9-202	Implement 8 puzzle problems using Depth First Search (DFS) Implement Iterative deepening search algorithm
3	10-9-2025	Implement A* search algorithm
4	8-10-2025	Implement Hill Climbing search algorithm to solve N-Queens problem
5	8-10-2025	Simulated Annealing to Solve 8-Queens problem
6	15-10-2025	Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.
7	29-10-2025	Implement unification in first order logic
8	29-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.
9	12-11-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution
10	12-11-2025	Implement Alpha-Beta Pruning.

INDEX

NAME: Ayaan Shrestha STD.: 5A SEC.: A ROLL NO.: 56

S. No.	Date	Title	Page No.	Teacher's Sign / Remarks
1.	20/8/25	- Tic Tac Toe - Vacuum Cleaner	8	
2.	3/9/25	IDDFS, DFS, 8-puzzle Manhattan, misplaced tiles	(8)	
3.	10/9/25	- 8 puzzle using A*	10	
4	8/10/25	N-Queens using Hill climbing algorithm	10	
5	8/10/25	N-Queens using simulated annealing	10	
6.	15/10/25	Propositional Logic Knowledge Base using	(9)	
7.	29/10/25	Unification in FOL	10	
8.	29/10/25	FOL - Forward Chaining	10	
9.	12/11/25	FOL → CNF	10	
10.	12/11/25	Alpha-beta Pruning, Min Max	10	

CIE - /10

or

/5

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Ayaan Shrestha

for successfully completing the course

AI-first Software Engineering

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!

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

Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Ayaan Shrestha

for successfully completing the course

Prompt Engineering

on November 23, 2025

Infosys | Springboard

Congratulations! You make us proud!

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

Issued on: Sunday, November 23, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to Highlights and Notes

Ayaan Shrestha

for successfully completing the course

Introduction to OpenAI GPT Models

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!

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

Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys
Navigate your next

COURSE COMPLETION CERTIFICATE

The certificate is awarded to

Ayaan Shrestha

for successfully completing the course

OpenAI Generative Pre-trained Transformer 3 (GPT-3) for developers

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!

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

Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Infosys
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Ayaan Shrestha

for successfully completing

Applied Generative AI Certification

on November 24, 2025

Infosys | Springboard

Congratulations! You make us proud!

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

Issued on: Monday, November 24, 2025
To verify, scan the QR code at https://verify.onwingspan.com

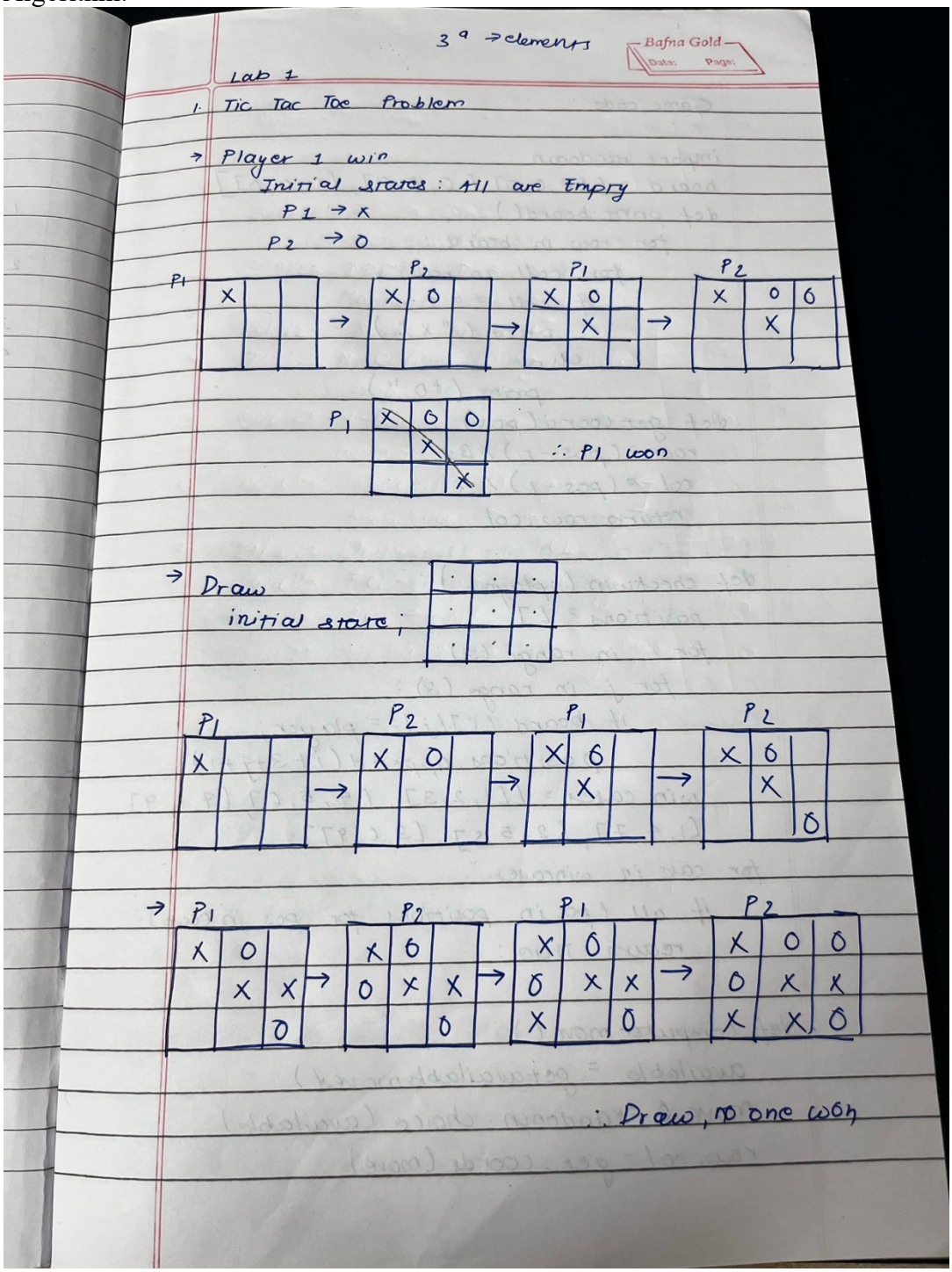
GitHub link : https://github.com/ayaanstha/AI-LABS

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



Game code :

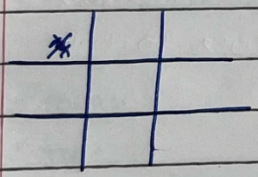
```
import randomn
board = [[0,0,0],[ 0, 0, 0], [0,0,0]]
def print board():
    for row in board:
        for cell in row:
            if cell == 1:
                print ("X"):
            else
                print ("0"):
def get coord(pos):
    row = (pos - 1) // 3
    col = (pos - 1) % 3
    return row, col

def checkwin ( player ):
    positions = []
    for i in range (3):
        for j in range (3):
            if board [i][j] == player;
                positions.append (i+3+j+1)
    win cases = [[1,2,3], [4,5,6], [7,8,9],
    [1,4,7], [2,5,8], [3,6,9]]
for case in wincases
    if all (pos in positions for pos in case):
        return True:

def computer move ( ):
    available = getavailablemoved()
    move 0 = randomn.choice (available)
    row, col = get_coords (move)
```

Bafna Gold
Date: Page:

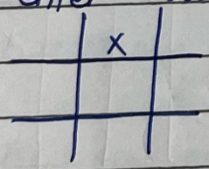
```
board[row][col] = 2

def playgame():
    printboard();
    for turn in range(9):
        if checkwin(1):
            print("You Win")
            return
        elif checkwin(2):
            print("Comp Wins")
            printboard()
    print("Draw")
```

O/p:

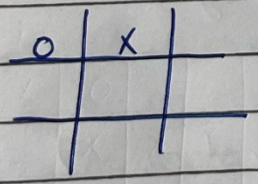
choose your symbol (X → first, O → second): X



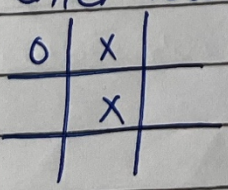
enter row and col (0-2, space seperated): 0 1



AI is making a move..



enter row and col (0-2, space sepr): 1 1



Code:
Tic tac toe:

```
def print_board(board):
    for row in board:
        print(" ".join(row))
    print()

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):
            return True
        if all(board[j][i] == player for j in range(3)):
            return True
    if all(board[i][i] == player for i in range(3)):
        return True
    if all(board[i][2 - i] == player for i in range(3)):
        return True
    return False

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

def minimax(board, is_ai_turn):
    if check_winner(board, 'O'):
        return 1
    if check_winner(board, 'X'):
        return -1
    if is_draw(board):
        return 0

    if is_ai_turn:
        best_score = -float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == '-':
                    board[i][j] = 'O'
                    score = minimax(board, False)
                    board[i][j] = '-'
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for i in range(3):
            for j in range(3):
                if board[i][j] == '-':
                    board[i][j] = 'X'
                    score = minimax(board, True)
```

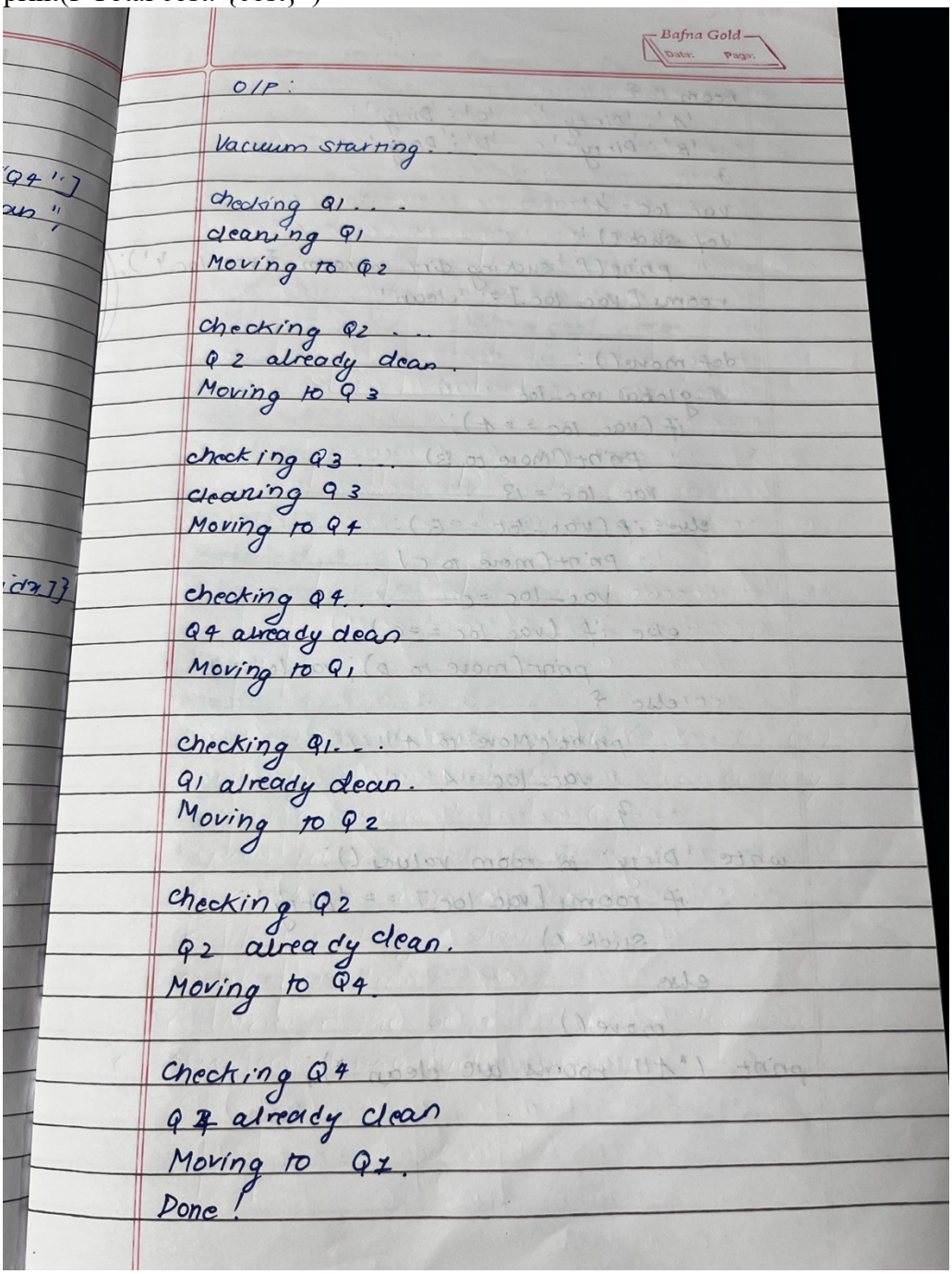
```
                board[i][j] = '-'
                best_score = min(score, best_score)
        return best_score

def manual_game():
    board = [['-' for _ in range(3)] for _ in range(3)]
    print("Initial Board:")
    print_board(board)

    while True:
        while True:
            try:
                x_row = int(input("Enter X row (1-3): ")) - 1
                x_col = int(input("Enter X col (1-3): ")) - 1
                if board[x_row][x_col] == '-':
                    board[x_row][x_col] = 'X'
                    break
                else:
                    print("Cell occupied!")
            except:
                print("Invalid input!")

        print("Board after X move:")
        print_board(board)

        if check_winner(board, 'X'):
            print("X wins!")
            break
        if is_draw(board):
            print("Draw!")
            break

        while True:
            try:
                o_row = int(input("Enter O row (1-3): ")) - 1
                o_col = int(input("Enter O col (1-3): ")) - 1
                if board[o_row][o_col] == '-':
                    board[o_row][o_col] = 'O'
                    break
                else:
                    print("Cell occupied!")
            except:
                print("Invalid input!")

        print("Board after O move:")
        print_board(board)
```

```
                if check_winner(board, 'O'):
                    print("O wins!")
                    break
                if is_draw(board):
                    print("Draw!")
                    break

                cost = minimax(board, True)
                print(f"AI evaluation cost from this position: {cost}")

        manual_game()
```

vaccum cleaner:

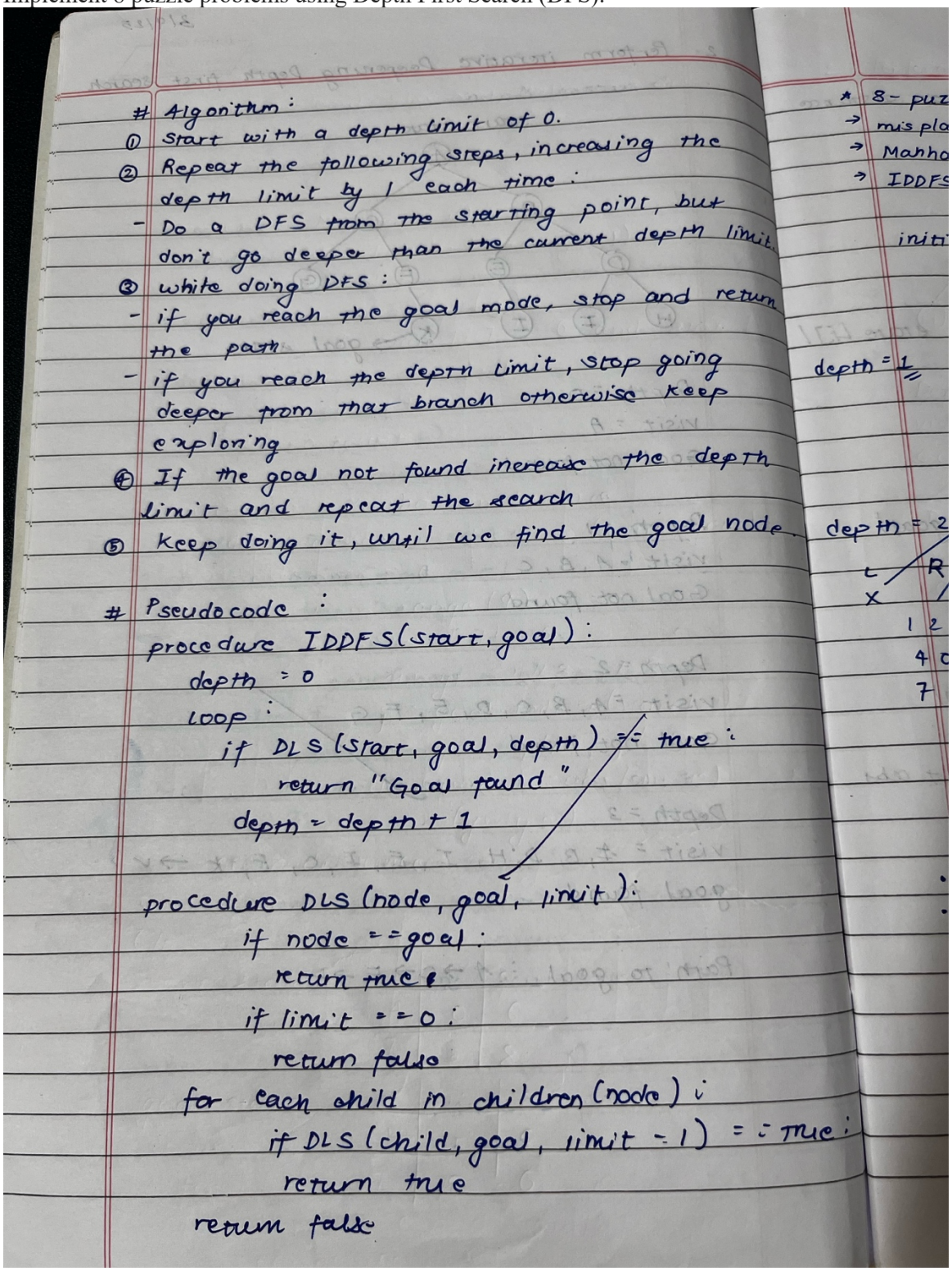
```
rooms = int(input("Enter Number of rooms: "))
Rooms = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
cost = 0
Roomval = {}


for i in range(rooms):
    print(f"Enter Room {Rooms[i]} state (0 for clean, 1 for dirty): ")
    n = int(input())
    Roomval[Rooms[i]] = n

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

while 1 in Roomval.values():
    if Roomval[loc] == 1:
        print(f"Room {loc} is dirty. Cleaning...")
        Roomval[loc] = 0
        cost += 1
    else:
        print(f"Room {loc} is already clean.")

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

    if move == "L":
        if loc != Rooms[0]:
            loc = Rooms[Rooms.index(loc) - 1]
        else:
            print("No room to move left.")
    elif move == "R":
        if loc != Rooms[rooms - 1]:
            loc = Rooms[Rooms.index(loc) + 1]
        else:
            print("No room to move right.")
    elif move == "Q":
```

```
        break
    else:
        print("Invalid input. Please enter L, R, or Q.")

print("\nAll Rooms Cleaned." if 1 not in Roomval.values() else "Exited before cleaning all rooms.")
print(f"Total cost: {cost}")
```



Program 2

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



Algorithm:
① Start with a depth limit of 0.
② Repeat the following steps, increasing the depth limit by 1 each time:
- Do a DFS from the starting point, but don't go deeper than the current depth limit.
③ While doing DFS:
- if you reach the goal node, stop and return the path
- if you reach the depth limit, stop going deeper from that branch otherwise keep exploring
④ If the goal not found increase the depth limit and repeat the search
⑤ Keep doing it, until we find the goal node.

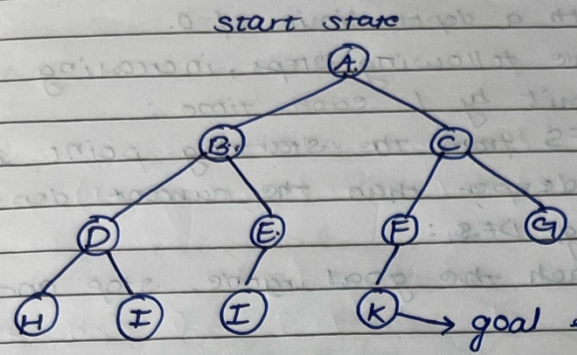
Pseudocode:

```
procedure IDDFS(start, goal):
    depth = 0
    loop:
        if DLS(start, goal, depth) == true:
            return "Goal found"
        depth = depth + 1

procedure DLS(node, goal, limit):
        if node == goal:
            return true
        if limit == 0:
            return false
        for each child in children(node):
            if DLS(child, goal, limit - 1) == true:
                return true
        return false
```

3/9/25

Bafna Gold
Date: Page:

2. Perform iterative Deepening Depth first search



Depth = 0
visit = A
Goal not found

Depth = 1
visit = A, B, C
Goal not found

Depth = 2
visit = A, B, C, D, E, F, G
Goal not found

Depth = 3
visit = A, B, D, H, I, E, I, C, F, K → V
goal found

Path to goal : A → C → F → K

using misplaced tiles and Manhattan distance

Pseudo code :

```
# Function 1 : Misplaced tiles
    def misplaced_tiles (state, goal)
        count = 0
        for i in range (9)
            if state[i] != 0 and state[i]
                != goal[i]
            count += 1
        return count

function 2 : Manhattan distance
    def manhattan_distance (state, goal)
        distance = 0
        for i in range (9) :
            if state[i] != 0
                x1, y1 = i // 3, i % 3
                j = goal.index (state[i])
                x2, y2 = j // 3, j % 3
                distance += abs (x1-x2) + abs
                    (y1-y2)
        return distance
```

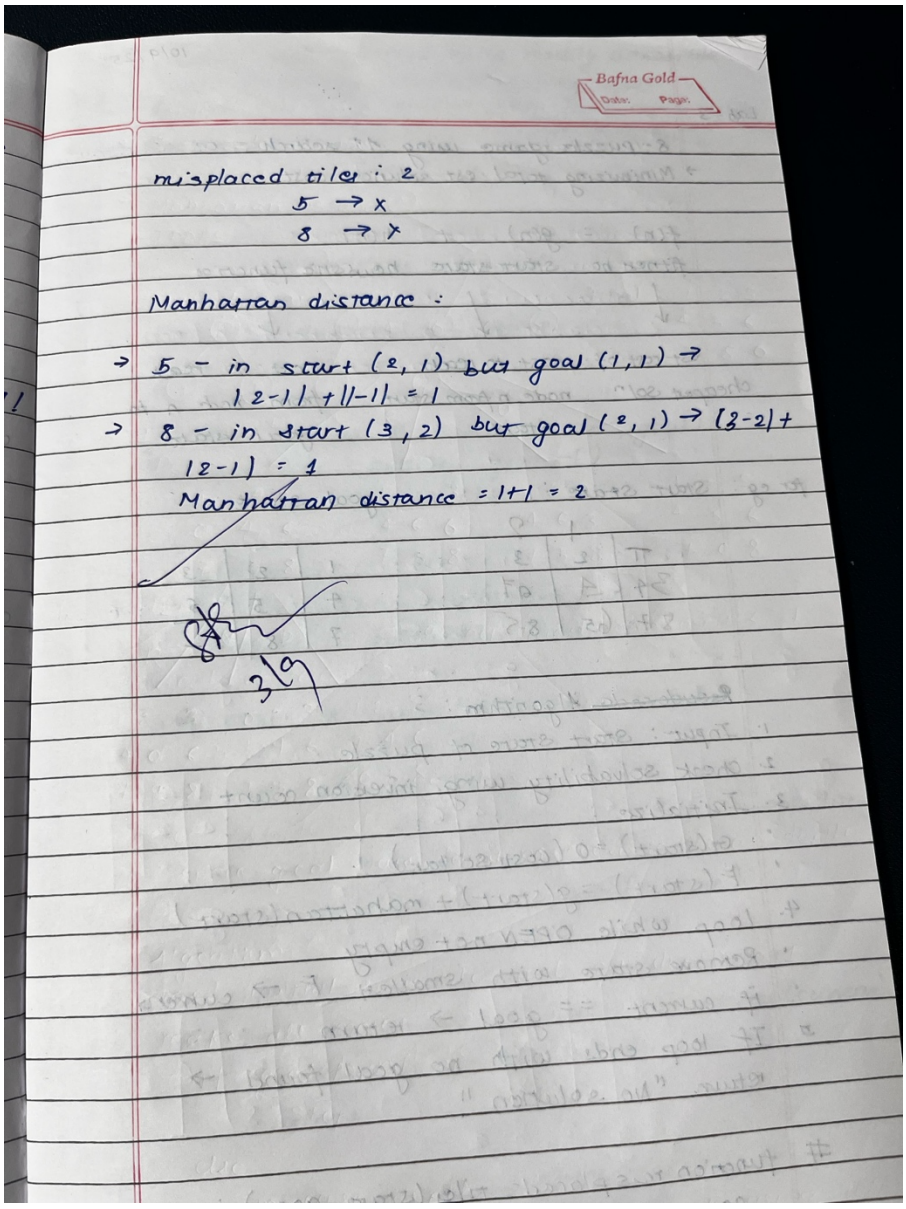
O/p :

start state :

1	2	3
4	0	6
7	5	8

goal state :

1	2	3
4	5	6
7	8	6



goal state = '123804765'

```
moves = {
   'U': -3,
   'D': 3,
   'L': -1,
   'R': 1
}
count = 0
invalid_moves = {
   0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
   3: ['L'],        5: ['R'],
   6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
```

```
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

def print_state(state):
    for i in range(0, 9, 3):
        print(' '.join(state[i:i+3]).replace('0', ' '))
    print()

def dfs(start_state, max_depth=50):
    visited = set()
    stack = [(start_state, [])]  # Each element: (state, path)

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

        if current_state in visited:
            continue

        # Print every visited state
        print("Visited state:")
        print_state(current_state)

        if current_state == goal_state:
            return path

        visited.add(current_state)
        global count
        count +=1
        if len(path) >= max_depth:
            continue

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                stack.append((new_state, path + [direction]))

    return None

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

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)
```

```
    result = dfs(start)

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("Number of visited states",count)


        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")
            print_state(current_state)
    else:
        print("No solution exists for the given start state or max depth reached.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Implement Iterative deepening search algorithm:

```
goal_state = '123456780'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],        5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)
```

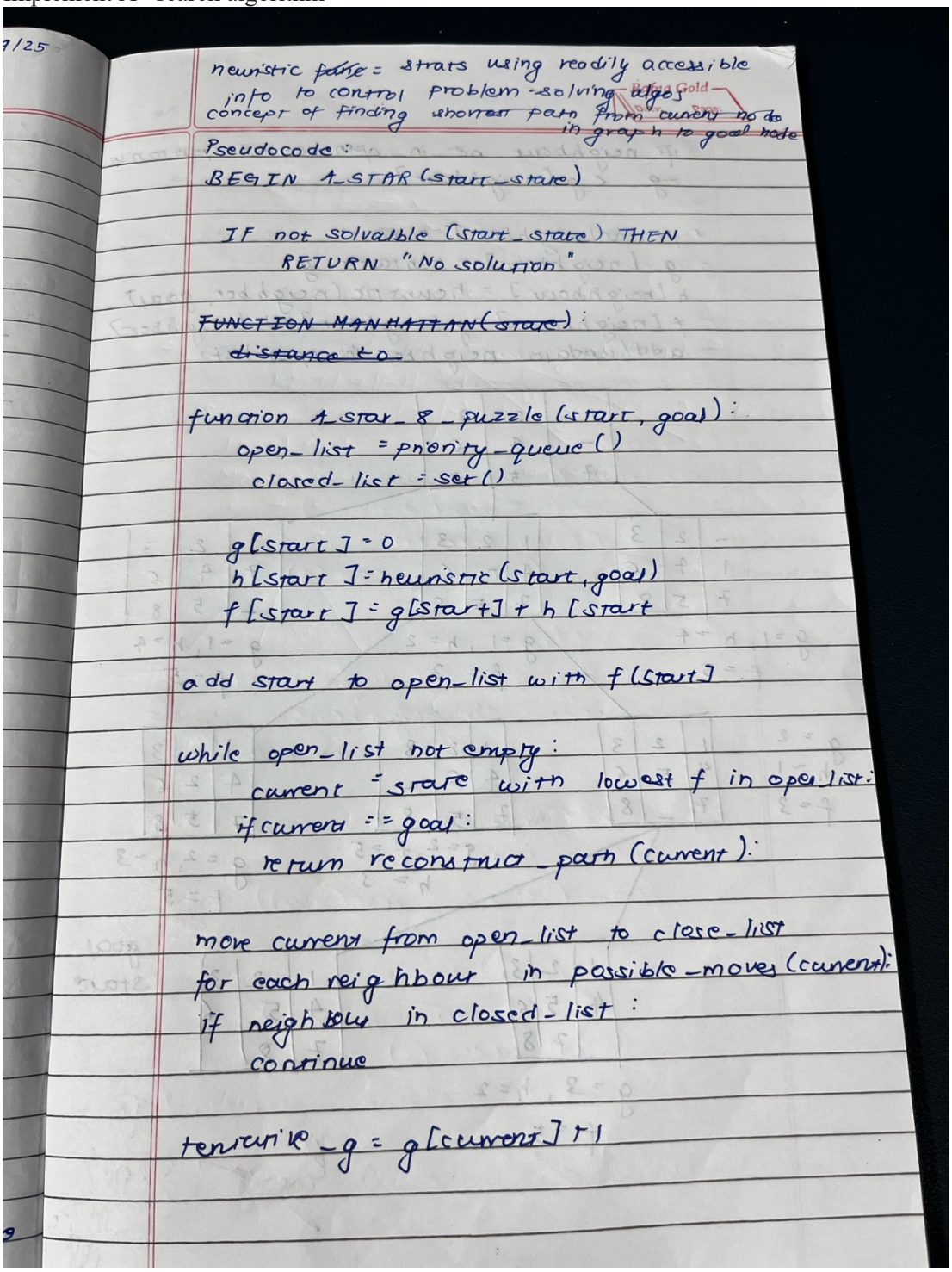
```
def print_state(state):
    for i in range(0, 9, 3):
        print(' '.join(state[i:i+3]).replace('0', ' '))
    print()

def dls(state, depth, path, visited, visited_count):
    visited_count[0] += 1  # Increment visited states count
    if state == goal_state:
        return path

    if depth == 0:
        return None

    visited.add(state)

    for direction in moves:
        new_state = move_tile(state, direction)
        if new_state and new_state not in visited:
            result = dls(new_state, depth - 1, path + [direction], visited, visited_count)
            if result is not None:
                return result

    visited.remove(state)
    return None

def iddfs(start_state, max_depth=50):
    visited_count = [0]  # Using list to pass by reference
    for depth in range(max_depth + 1):
        visited = set()
        result = dls(start_state, depth, [], visited, visited_count)
        if result is not None:
            return result, visited_count[0]
    return None, visited_count[0]

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

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result, visited_states = iddfs(start,15)

    print(f"Total states visited: {visited_states}")

    if result is not None:
```

```
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))


        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")
            print_state(current_state)
    else:
        print("No solution exists for the given start state or max depth reached.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

```
Enter start state (e.g., 724506831): 724506831
Start state:
7 2 4
5   6
8 3 1

Visited state:
7 2 4
5   6
8 3 1

Visited state:
7 2 4
5 6
8 3 1

Visited state:
7 2 4
5 6 1
8 3

Visited state:
7 2 4
5 6 1
8   3
```

Program 3

Implement A* search algorithm



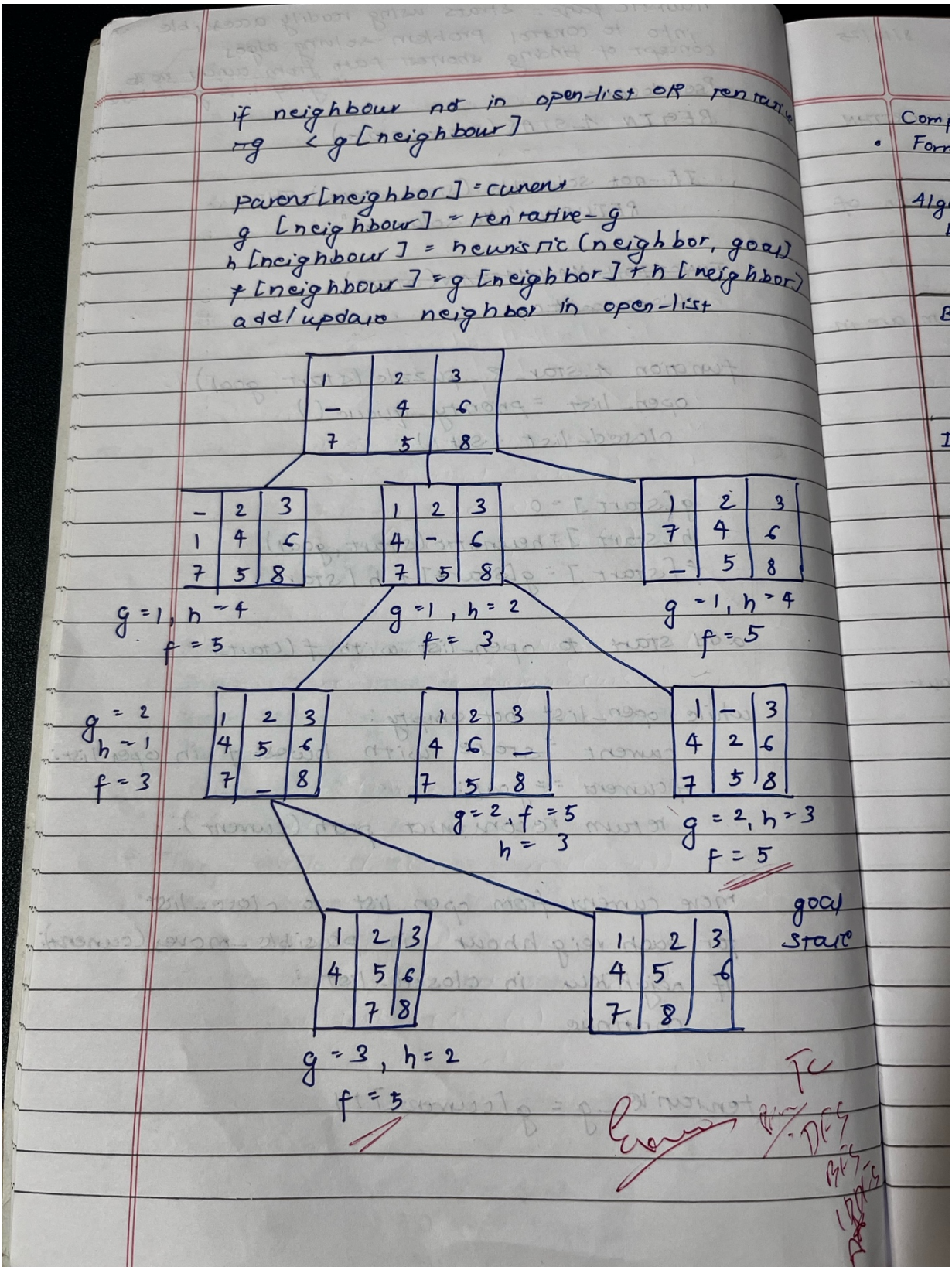
heuristic ~~function~~ = strats using readily accessible info to control problem-solving algos concept of finding shortest path from current node in graph to goal node

Pseudocode:

```
BEGIN A_STAR (start_state)

  IF not solvable (start_state) THEN
      RETURN "No solution"

~~FUNCTION MANHATTAN(state):~~
  ~~distance to~~

function A_star_8_puzzle (start, goal):
    open_list = priority_queue()
    closed_list = set()

    g[start] = 0
    h[start] = heuristic (start, goal)
    f[start] = g[start] + h[start

add start to open_list with f[start]

while open_list not empty:
    current = state with lowest f in open_list:
    if current == goal:
      return reconstruct_path (current):

  move current from open_list to close_list
  for each neighbour in possible_moves (current):
  if neighbour in closed_list:
      continue

tentative_g = g[current] + 1
```

if neighbour not in open-list OR tentative_g < g[neighbour]

parent[neighbor] = current
g[neighbour] = tentative_g
h[neighbour] = heuristic(neighbor, goal)
f[neighbour] = g[neighbor] + h[neighbor]
add/update neighbor in open-list



Complexity Comparision Overview

- Formula Reference Table

Algorithm	Time Complexity	Space Complexity	Notes
DFS	$O(V+E)$	$O(V)$	recursion, $V=V_{ertices}$ $E=Edges$
BFS	$O(V+E)$	$O(V)$	Queue for layer; can larger large gaps
IDDFS	$O(b^d)$	$O(d)$	b = branching factor, d = s^{th} depth
A*	$O(b^d)$	$O(b^d)$	Depends on $h(n)$ quality may keep all nodes

using misplaced tiles and Manhattan distance

Pseudo code :

```
# Function 1 : Misplaced tiles
    def misplaced_tiles (state, goal)
        count = 0
        for i in range (9)
            if state[i] != 0 and state[i]
                = goal[i]
            count += 1
        return count

function 2 : Manhattan distance
    def manhattan_distance (state, goal)
        distance = 0
        for i in range (9):
            if state[i] != 0
                x1, y1 = i // 3, i % 3
                j = goal.index (state[i])
                x2, y2 = j // 3, j % 3
                distance += abs (x1-x2) + abs
                    (y1-y2)
        return distance
```

O/p :

start state

1	2	3
4	0	6
7	5	8

goal state :

1	2	3
4	5	6
7	8	6

```
import heapq

goal_state = '123456780'

moves = {
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],        5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')
    if direction in invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
    if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)
    state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
    return ''.join(state_list)

def print_state(state):
    for i in range(0, 9, 3):
        print(' '.join(state[i:i+3]).replace('0', ' '))
    print()

def manhattan_distance(state):
    distance = 0
    for i, val in enumerate(state):
        if val == '0':
            continue
        goal_pos = int(val) - 1
        current_row, current_col = divmod(i, 3)
        goal_row, goal_col = divmod(goal_pos, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

def a_star(start_state):
```

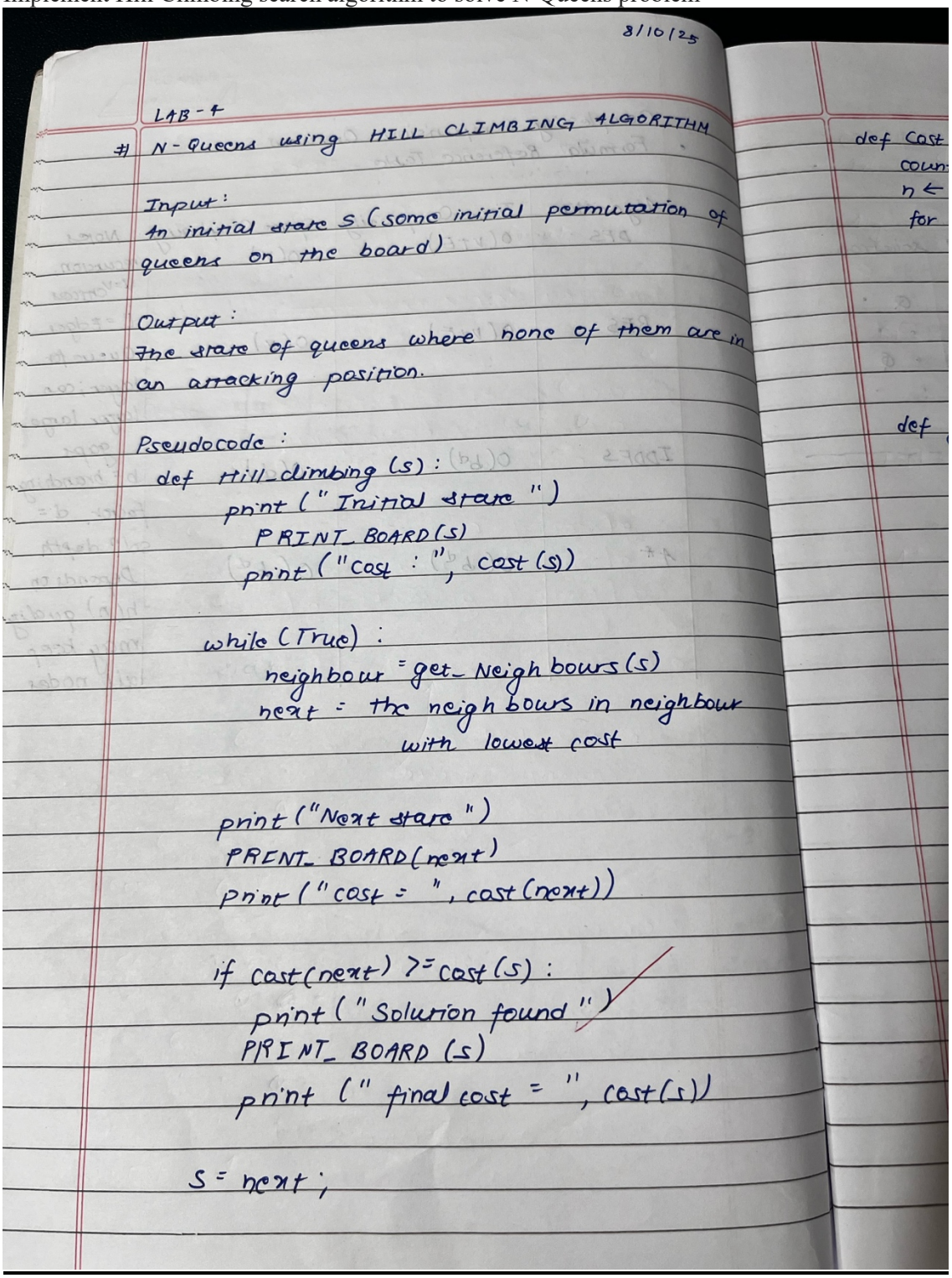
```
    visited_count = 0
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(start_state), 0, start_state, []))
    visited = set()

    while open_set:
        f, g, current_state, path = heapq.heappop(open_set)
        visited_count += 1

        if current_state == goal_state:
            return path, visited_count

        if current_state in visited:
            continue
        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(new_state)
                heapq.heappush(open_set, (new_f, new_g, new_state, path + [direction]))

    return None, visited_count

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

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
    print_state(start)

    result, visited_states = a_star(start)

    print(f"Total states visited: {visited_states}")

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))


        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")
            print_state(current_state)
```

```
    else:
        print("No solution exists for the given start state.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

```
Enter start state (e.g., 724506831): 283164705
Start state:
2 8 3
1 6 4
7   5

Total states visited: 7
Solution found!
Moves: U U L D R
Number of moves: 5
1BM23CS299 sanchit mehta

Move 1: U
2 8 3
1   4
7 6 5

Move 2: U
2   3
1 8 4
7 6 5

Move 3: L
  2 3
1 8 4
7 6 5

Move 4: D
1 2 3
  8 4
7 6 5

Move 5: R
1 2 3
8   4
7 6 5
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem



8/10/25

LAB - 4

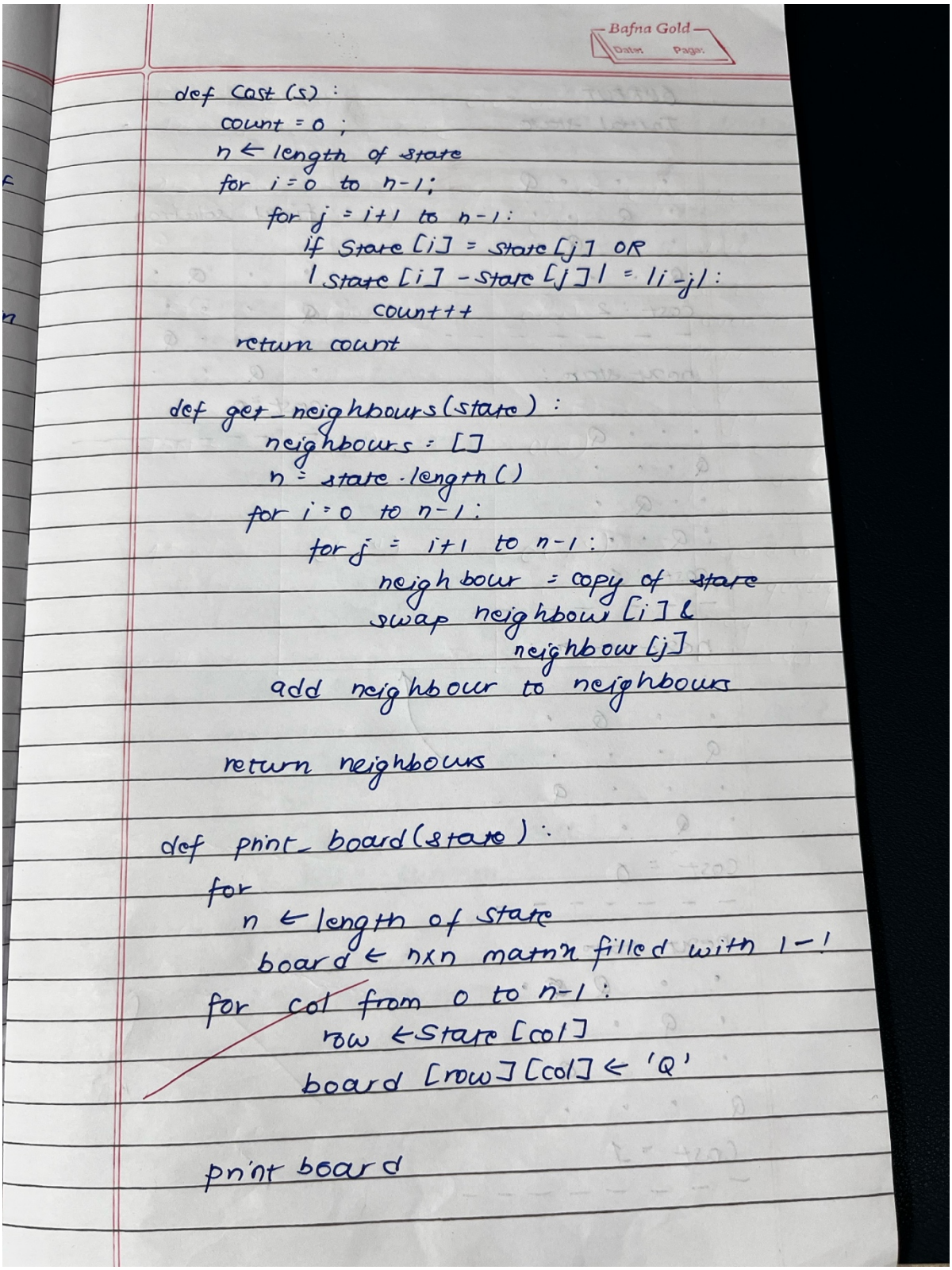
N-Queens using HILL CLIMBING ALGORITHM

Input:
An initial state S (some initial permutation of queens on the board)

Output:
The state of queens where none of them are in an attacking position.

Pseudocode:

```
def Hill_climbing (s):
    print ("Initial state ")
      PRINT_BOARD(s)
    print ("cost :", cost (s))

  while (True) :
      neighbour = get_Neighbours(s)
      next = the neighbours in neighbour
                with lowest cost

  print ("Next state ")
  PRINT_BOARD(next)
  print ("cost = ", cost(next))

  if cost(next) >= cost(s) :
      print ("Solution found")
      PRINT_BOARD (s)
      print (" final cost = ", cost(s))

  S = next;
```



```
def cost(s):
    count = 0;
    n ← length of state
    for i = 0 to n-1;
        for j = i+1 to n-1:
            if state[i] = state[j] OR
            |state[i] - state[j]| = |i-j|:
                count++
    return count

def get_neighbours(state):
    neighbours = []
    n = state.length()
    for i = 0 to n-1:
        for j = i+1 to n-1:
            neighbour = copy of state
            swap neighbour[i] &
                 neighbour[j]
        add neighbour to neighbours

    return neighbours

def print_board(state):
    for
        n ← length of state
        board ← nxn matrix filled with '-'
    for col from 0 to n-1:
            row ← state[col]
            board[row][col] ← 'Q'

    print board
```

```
import random
import time

def print_board(state):
    """Prints the chessboard for a given state."""
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

def compute_heuristic(state):
    """Computes the number of attacking pairs of queens."""
    h = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                h += 1
    return h

def get_neighbors(state):
    """Generates all possible neighbors by moving one queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if row != state[col]:
                neighbor = list(state)
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

def hill_climb_verbose(initial_state, step_delay=0.5):
    """Hill climbing algorithm with verbose output at each step."""
    current = initial_state
    current_h = compute_heuristic(current)
    step = 0

    print(f"Initial state (heuristic: {current_h}):")
    print_board(current)
    time.sleep(step_delay)
```

```
    while True:
        neighbors = get_neighbors(current)
        next_state = None
        next_h = current_h

        for neighbor in neighbors:
            h = compute_heuristic(neighbor)
            if h < next_h:
                next_state = neighbor
                next_h = h

        if next_h >= current_h:
            print(f"Reached local minimum at step {step}, heuristic: {current_h}")
            return current, current_h

        current = next_state
        current_h = next_h
        step += 1
        print(f"Step {step}: (heuristic: {current_h})")
        print_board(current)
        time.sleep(step_delay)

def solve_n_queens_verbose(n, max_restarts=1000):
    """Solves N-Queens problem using hill climbing with restarts and verbose output."""
    for attempt in range(max_restarts):
        print(f"\n=== Restart {attempt + 1} ===\n")
        initial_state = [random.randint(0, n - 1) for _ in range(n)]
        solution, h = hill_climb_verbose(initial_state)
        if h == 0:
            print(f" Solution found after {attempt + 1} restart(s):")
            print_board(solution)
            return solution
        else:
            print(f"No solution in this attempt (local minimum).\n")
    print("Failed to find a solution after max restarts.")
    return None

# --- Run the algorithm ---
if __name__ == "__main__":
    N = int(input("Enter the number of queens (N): "))
    solve_n_queens_verbose(N)
```

Output:

```
Enter the number of queens (N): 4

=== Restart 1 ===
```

Initial state (heuristic: 3):
Q . Q .
. Q . .
. . . Q
. . . .

Step 1: (heuristic: 1)
. . Q .
. Q . .
. . . Q
Q . . .

Reached local minimum at step 1, heuristic: 1
❌ No solution in this attempt (local minimum).

=== Restart 2 ===

Initial state (heuristic: 3):
. Q . .
. . Q .
. . . .
Q . . Q

Step 1: (heuristic: 1)
. Q . .
. . Q .
Q . . .
. . . Q

Reached local minimum at step 1, heuristic: 1
❌ No solution in this attempt (local minimum).

=== Restart 3 ===

Initial state (heuristic: 2):
. . . .
. Q . Q
. . . .
Q . Q .

Step 1: (heuristic: 1)
. Q . .
. . . Q
. . . .

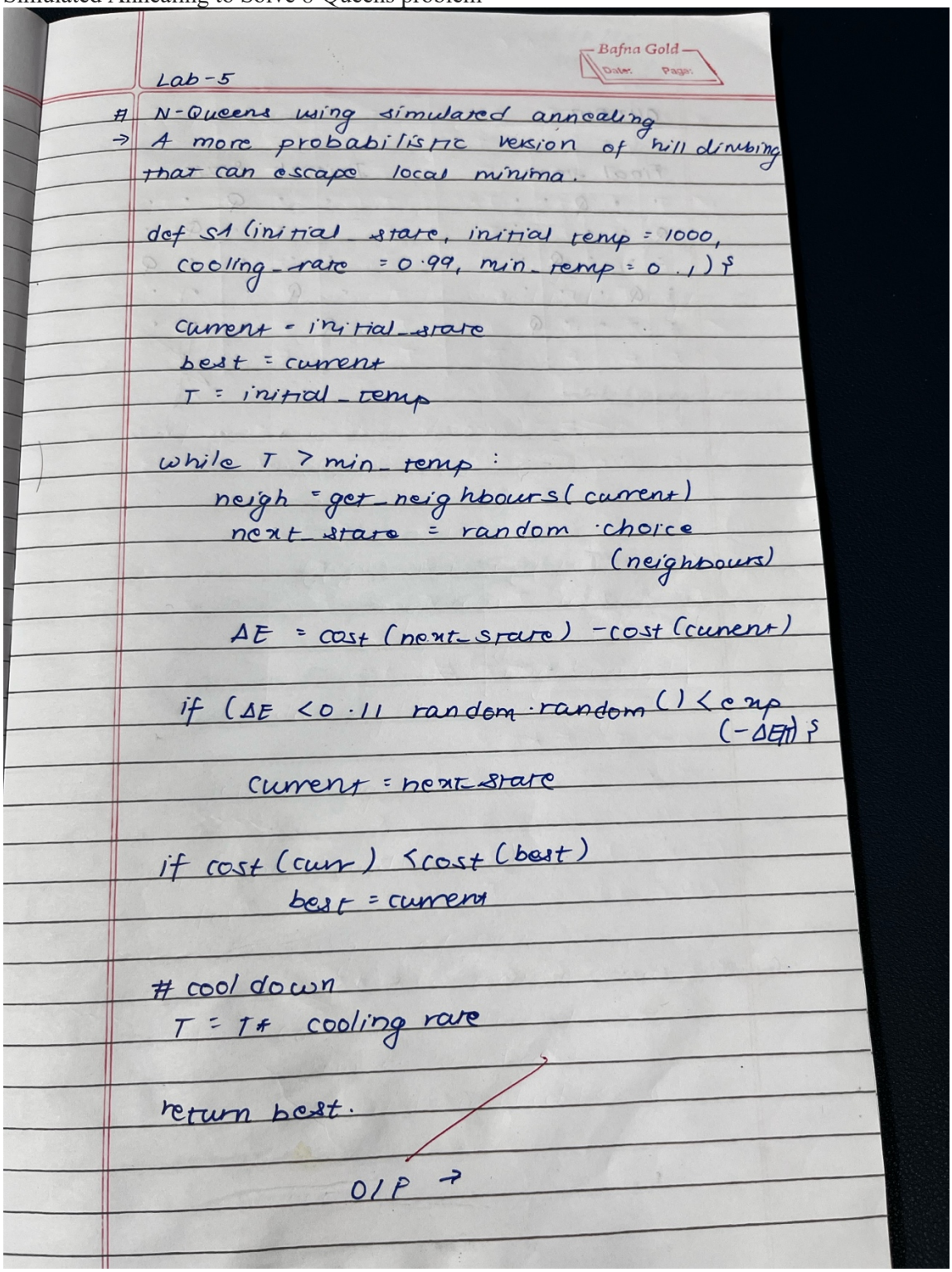
Q . Q .

Step 2: (heuristic: 0)
. Q . .
. . . Q
Q . . .
. . Q .

Reached local minimum at step 2, heuristic: 0
✅ Solution found after 3 restart(s):
. Q . .
. . . Q
Q . . .
. . Q .

Program 5

Simulated Annealing to Solve 8-Queens problem

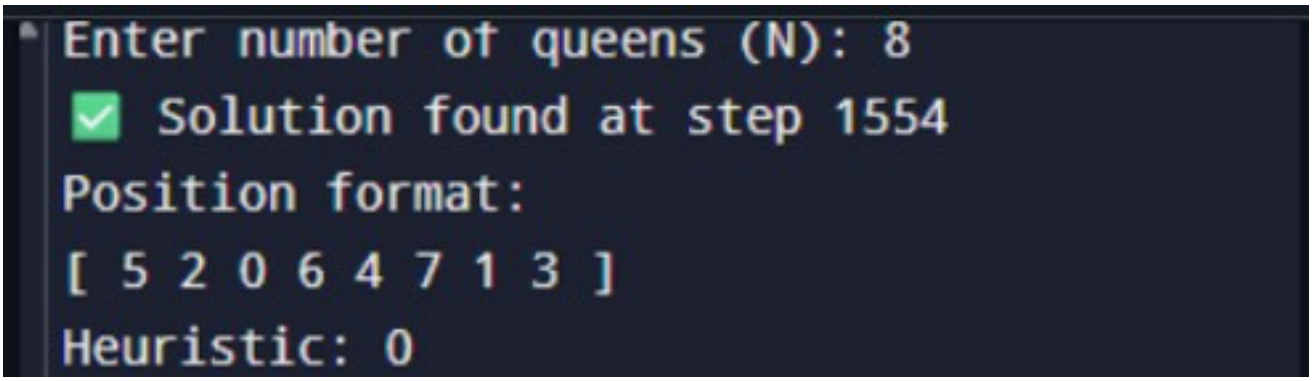


Lab-5

```
# N-Queens using simulated annealing
→ A more probabilistic version of hill climbing
that can escape local minima.

def SA(initial_state, initial_temp = 1000,
  cooling_rate = 0.99, min_temp = 0.1):

  current = initial_state
  best = current
  T = initial_temp

  while T > min_temp :
    neigh = get_neighbours(current)
    next_state = random.choice
                 (neighbours)

    ΔE = cost(next_state) - cost(current)

  if (ΔE < 0.11 random.random() < exp
                 (-ΔE/T)):

    current = next_state

  if cost(curr) < cost(best)
      best = current

# cool down
  T = T* cooling rate

return best.

O/P →
```

```
import random
import math

def compute_heuristic(state):
    """Number of attacking pairs."""
    h = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                h += 1
    return h

def random_neighbor(state):
    """Returns a neighbor by randomly changing one queen's row."""
    n = len(state)
    neighbor = state[:]
    col = random.randint(0, n - 1)
    old_row = neighbor[col]
    new_row = random.choice([r for r in range(n) if r != old_row])
    neighbor[col] = new_row
    return neighbor

def dual_simulated_annealing(n, max_iter=10000, initial_temp=100.0, cooling_rate=0.99):
    """Simulated Annealing with dual acceptance strategy."""
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_h = compute_heuristic(current)
    temperature = initial_temp

    for step in range(max_iter):
        if current_h == 0:
            print(f"✅ Solution found at step {step}")
            return current

        neighbor = random_neighbor(current)
        neighbor_h = compute_heuristic(neighbor)
        delta = neighbor_h - current_h

        if delta < 0:
            current = neighbor
            current_h = neighbor_h
        else:
            # Dual acceptance: standard + small chance of higher uphill move
            probability = math.exp(-delta / temperature)
            if random.random() < probability:
                current = neighbor
                current_h = neighbor_h
```

```
        temperature *= cooling_rate
        if temperature < 1e-5:  # Restart if stuck
            temperature = initial_temp
            current = [random.randint(0, n - 1) for _ in range(n)]
            current_h = compute_heuristic(current)

    print("❌ Failed to find solution within max iterations.")
    return None

# --- Run the algorithm ---
if __name__ == "__main__":
    N = int(input("Enter number of queens (N): "))
    solution = dual_simulated_annealing(N)

    if solution:
        print("Position format:")
        print("[", " ".join(str(x) for x in solution), "]")
        print("Heuristic:", compute_heuristic(solution))
```



Program 6

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

15/10/25
Wednesday

LAB-6

Propositional Logic

Q) Knowledge Base using proportional logic

Algorithm

Q → query

KB → knowledge Base

```
def entails(KB, Q):
    propositions = get_unique_propositions(KB, Q)

# assignments for the propositions
    truth_table = generate_truth_table(propositions)

    for row in truth_table:
        if evaluate(KB, row):
            if not evaluate(Q, row):
                ~~if not evaluate(~~
        ~~KB is~~ return False

            return False
    return True
```

Q) Proportional logic

Q → P

P → -Q

i) Construct a TT that shows the truth value of each sentence in KB and indicate the models in which the KB is true.

ii) Does KB entail R?

iii) " " entail R → P ?

iv) Does KB entail Q → R?

∧ ∨

OR

TT

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

ans ii) Yes, KB ⊨ R because when KB is true, R is also True

iii) models R P R → P

(T, F, T) T F F

(F, F, T) T F F

∴ R → P ⊭ KB

iv) Q → R ⊨ KB ?

models Q R Q → R

(T, F, T) F T T

(F, F, T) F T T

∴ Q → P ⊨ KB //

```
Enter knowledge base (e.g. (and A (or B C))): A
Enter query (e.g. A): A
Symbols: ['A']
A     KB    α
True      True      True
False     False     False

Does KB entail α? : True

=== Code Execution Successful ===
```

Program 7

Implement unification in first order logic:

29/10/25

Bafna Gold

Date: Page:

Week - 7 Lab

First Order Logic

Implement unification in first order logic

→ Algo

(i) If ψ_1 or ψ_2 is a variable or constant then:

(a) If ψ_1 or ψ_2 are identical, then return NIL

b) Elseif ψ_2 is a variable,

i. then if ψ_1 occurs is ψ_2, then return FAILURE

ii. Else return $\{(\psi_2 / \psi_1)\}$

c) Else if ψ_2 is a variable,

i. if ψ_2 occurs in ψ_1, then return FAILURE

ii. Else return $\{(\psi_2 / \psi_1)\}$

d) Else return FAILURE

(ii) If the initial predicate symbol in ψ_1 & ψ_2 are not the same, then return FAILURE

(iii) If ψ_1 & ψ_2 have diff no of args, then return FAILURE

iv) Set substitution set (SUBST) to NIL

v) For i = 1 to the no of element in ψ_1

a) Call unify function with the i^{th} elements of ψ_1 & i^{th} element of ψ_2 and put the result into S.

b) If S = failure → return FAILURE

c) if S ≠ NIL, then do
i. Apply S to the remainder of both L1 & L2
ii. SUBSET = APPEND (S, SUBSET)

vi) Return SUBST

Questions

① $P(f(x), g(y), y)$
$P(f(g(z)), g(f(a)), f(a))$

find θ (MGU) → Most general unifier

→ ~~$f(x) \to f(g(z))$~~
~~$g(y) = g(f$~~
~~$y =$~~

ans: $f(x) \to f(g(z))$
⇒ $x \to g(z)$

$g(y) \to g(f(a))$
$y = f(a)$

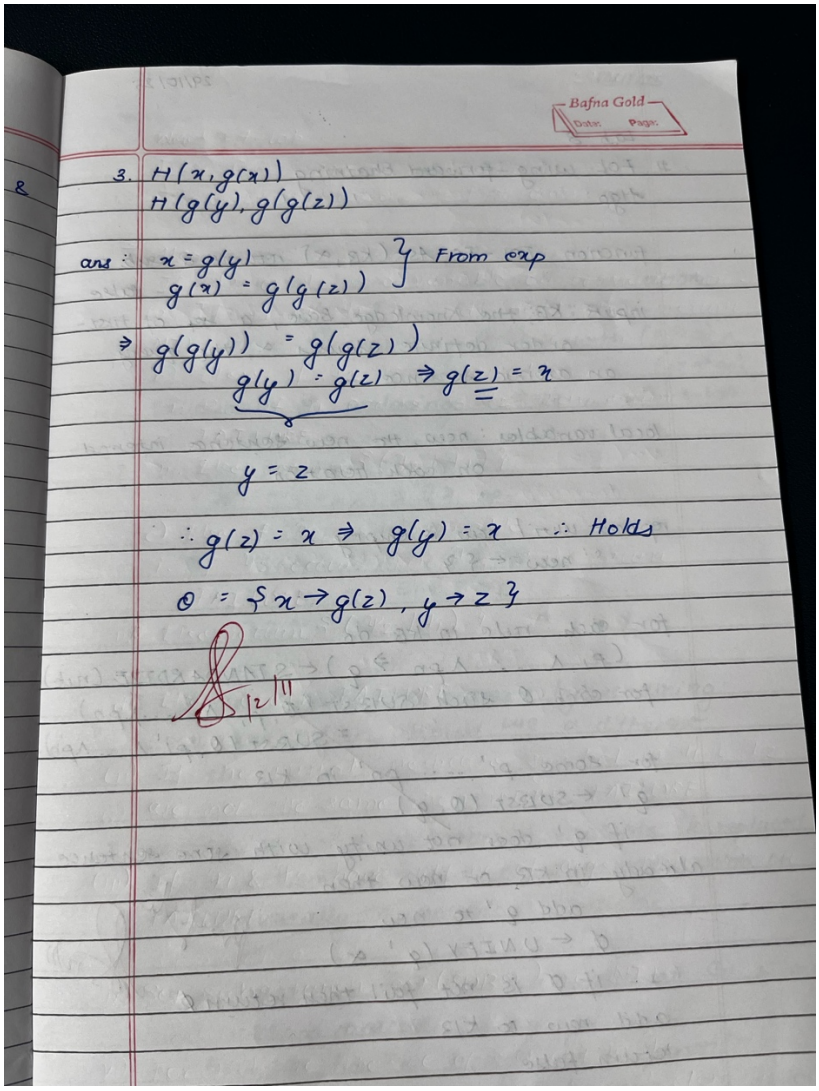
$\theta = \{x/g(z), y/f(a)\}$ Holds

② ~~$Q(x, f(x))$~~ $Q(x, f(x))$
$Q(f(y), y)$

ans: Not defined
$x = f(y)$
$f(x) = y \Rightarrow f(f(y))$
$= y$
↳ cyclic in Nature

3. $H(x$
$H(g$

ans: $x =$
g
⇒ $g($




```
import json

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

def is_variable(term):
    """Checks if a term is a variable (a single capital letter string)."""
    return isinstance(term, str) and len(term) == 1 and 'A' <= term[0] <= 'Z'

def occurs_check(variable, term, sigma):
    """
    Checks if 'variable' occurs anywhere in 'term' under the current substitution 'sigma'.
    This prevents infinite recursion (e.g., unifying X with f(X)).
    """
    term = apply_substitution(term, sigma) # Check the substituted term

    if term == variable:
```

```
        return True

    # If the term is a list (function/predicate), check its arguments recursively
    if isinstance(term, list):
        for arg in term[1:]:
            if occurs_check(variable, arg, sigma):
                return True

    return False

def apply_substitution(term, sigma):
    """
    Applies the current substitution 'sigma' to a 'term' recursively.
    """
    if is_variable(term):
        # If the variable is bound in sigma, apply the binding
        if term in sigma:
            # Recursively apply the rest of the substitutions to the binding's value
            # This is critical for chains like X/f(Y), Y/a -> X/f(a)
            return apply_substitution(sigma[term], sigma)
        return term

    if isinstance(term, list):
        # Apply substitution to the arguments of the function/predicate
        new_term = [term[0]] # Keep the function/predicate symbol
        for arg in term[1:]:
            new_term.append(apply_substitution(arg, sigma))
        return new_term

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

def term_to_string(term):
    """
    Converts the internal list representation of a term into standard logic notation string.
    e.g., ['f', 'Y'] -> "f(Y)"
    """
    if isinstance(term, str):
        return term

    if isinstance(term, list):
        # Term is a function or predicate
        symbol = term[0]
        args = [term_to_string(arg) for arg in term[1:]]
        return f"{symbol}({', '.join(args)})"

    return str(term)
```

```
# --- Main Unification Function ---

def unify(term1, term2):
    """
    Implements the Unification Algorithm to find the MGU for term1 and term2.
    Returns the MGU as a dictionary or None if unification fails.
    """
    # Initialize the substitution set (MGU)
    sigma = {}

    # Initialize the list of pairs to resolve (the difference set)
    diff_set = [[term1, term2]]

    print(f"--- Unification Process Started ---")
    print(f"Initial Terms:")
    print(f"L1: {term_to_string(term1)}")
    print(f"L2: {term_to_string(term2)}")
    print("-" * 35)

    while diff_set:
        # Pop the current pair of terms to unify
        t1, t2 = diff_set.pop(0)

        # 1. Apply the current MGU to the terms before comparison
        t1_prime = apply_substitution(t1, sigma)
        t2_prime = apply_substitution(t2, sigma)

        print(f"Attempting to unify: {term_to_string(t1_prime)} vs {term_to_string(t2_prime)}")


        # 2. Check if terms are identical
        if t1_prime == t2_prime:
            print(f"  -> Identical. Current MGU: {term_to_string(sigma)}")
            continue

        # 3. Handle Variable-Term unification
        if is_variable(t1_prime):
            var, term = t1_prime, t2_prime
        elif is_variable(t2_prime):
            var, term = t2_prime, t1_prime
        else:
            var, term = None, None

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

```
        if is_variable(term):
            print(f"  -> Both are variables. Skipping {var} / {term}")
            # Ensure they are added back if not identical (which is caught by step 2).
            # If V1 != V2, we add V1/V2 or V2/V1 to sigma. Since step 2 handles V/V, this means V1
!= V2 here.
            if var != term:
                sigma[var] = term
                print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")
        # Occurs Check: Fail if the variable occurs in the term it's being bound to
        elif occurs_check(var, term, sigma):
            print(f"  -> OCCURS CHECK FAILURE: Variable {var} occurs in
{term_to_string(term)}")
            return None

        # Create a new substitution {var / term}
        else:
            sigma[var] = term
            print(f"  -> Variable binding added: {var} / {term_to_string(term)}. New MGU:
{term_to_string(sigma)}")

    # 4. Handle Complex Term (Function/Predicate) unification
    elif isinstance(t1_prime, list) and isinstance(t2_prime, list):
        # Check functor/predicate symbol and arity (number of arguments)
        if t1_prime[0] != t2_prime[0] or len(t1_prime) != len(t2_prime):
            print(f"  -> FUNCTOR/ARITY MISMATCH: {t1_prime[0]} != {t2_prime[0]} or arity
mismatch.")
            return None

        # Add corresponding arguments to the difference set
        # Start from index 1 (after the symbol)
        for arg1, arg2 in zip(t1_prime[1:], t2_prime[1:]):
            diff_set.append([arg1, arg2])
        print(f"  -> Complex terms matched. Adding arguments to difference set.")

    # 5. Handle Constant-Constant or other mismatches (Fail)
    else:
        print(f"  -> TYPE/CONSTANT MISMATCH: {term_to_string(t1_prime)} and
{term_to_string(t2_prime)} cannot be unified.")
        return None

  print("-" * 35)
  print("--- Unification Successful ---")

  # Final cleanup to ensure all bindings are fully resolved
  final_mgu = {k: apply_substitution(v, sigma) for k, v in sigma.items()}
  return final_mgu
```

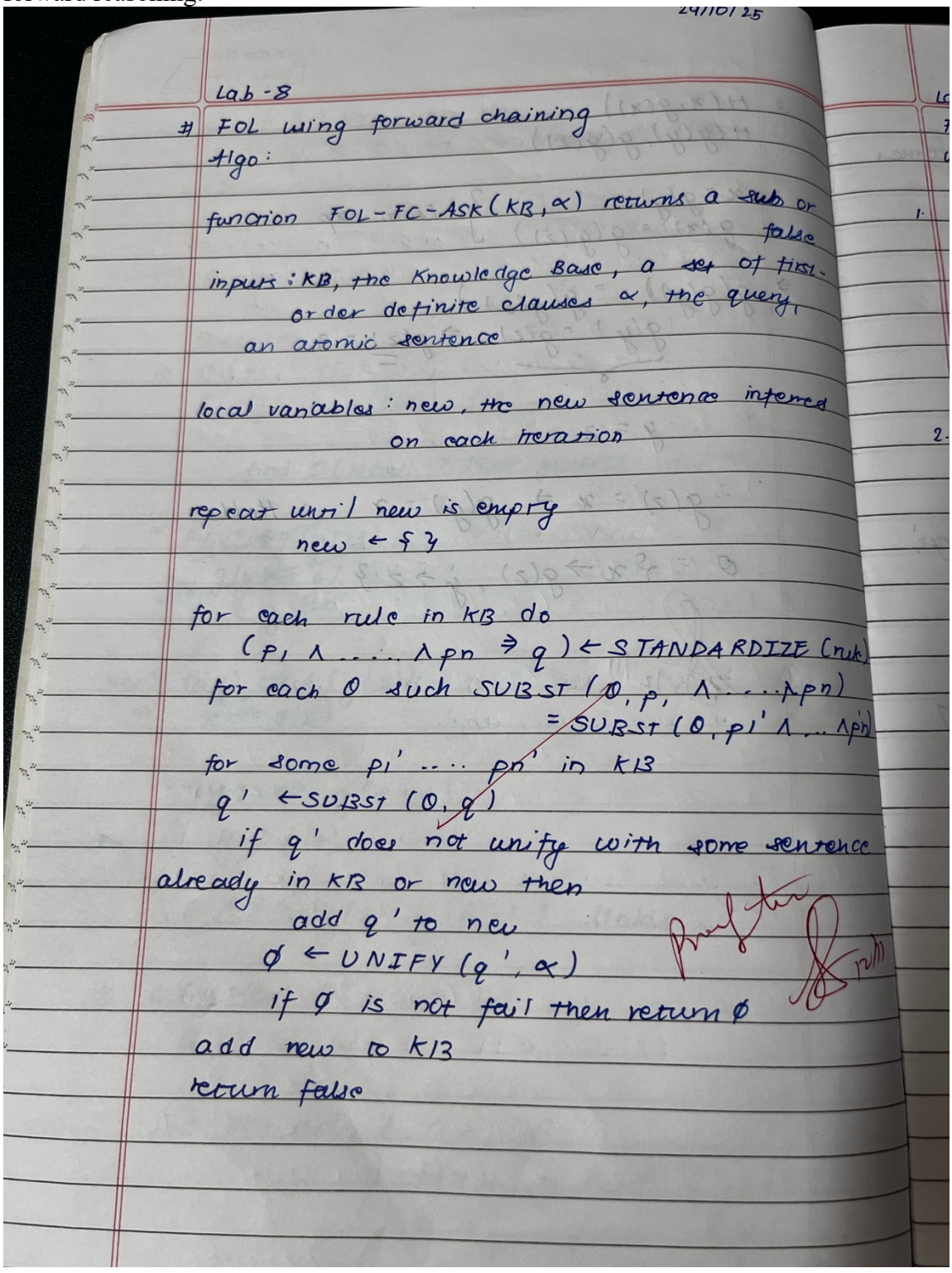
```
# --- Define the Input Terms ---

# L1 = Q(a, g(X, a), f(Y))
literal1 = ['Q', 'a', ['g', 'X', 'a'], ['f', 'Y']]

# L2 = Q(a, g(f(b), a), X)
literal2 = ['Q', 'a', ['g', ['f', 'b'], 'a'], 'X']

# --- Run the Unification ---

mgu_result = unify(literal1, literal2)

if mgu_result is not None:
    print("\n[ Final MGU Result ]")

    # Format the final MGU for display using the new helper function
    clean_mgu = {k: term_to_string(v) for k, v in mgu_result.items()}
    final_output = ', '.join([f"{k} / {v}" for k, v in clean_mgu.items()])
    print(f"Final MGU: {{ {final_output} }}")

    # --- Verification ---
    print("\n[ Verification ]")
    unified_l1 = apply_substitution(literal1, mgu_result)
    unified_l2 = apply_substitution(literal2, mgu_result)

    print(f"L1 after MGU: {term_to_string(unified_l1)}")
    print(f"L2 after MGU: {term_to_string(unified_l2)}")

    if unified_l1 == unified_l2:
        print("-> SUCCESS: L1 and L2 are identical after applying the MGU.")
    else:
        print("-> ERROR: Unification failed verification.")
else:
    print("\nUnification FAILED.")
```

Output

```
--- Unification Process Started ---
Initial Terms:
L1: Q(a, g(X, a), f(Y))
L2: Q(a, g(f(b), a), X)
----------------------------------
Attempting to unify: Q(a, g(X, a), f(Y)) vs Q(a, g(f(b), a), X)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {}
Attempting to unify: g(X, a) vs g(f(b), a)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: f(Y) vs X
  -> Variable binding added: X / f(Y). New MGU: {'X': ['f', 'Y']}
Attempting to unify: f(Y) vs f(b)
  -> Complex terms matched. Adding arguments to difference set.
Attempting to unify: a vs a
  -> Identical. Current MGU: {'X': ['f', 'Y']}
Attempting to unify: Y vs b
  -> Variable binding added: Y / b. New MGU: {'X': ['f', 'Y'], 'Y': 'b'}
----------------------------------
--- Unification Successful ---

[ Final MGU Result ]
Final MGU: { X / f(b), Y / b }

[ Verification ]
L1 after MGU: Q(a, g(f(b), a), f(b))
L2 after MGU: Q(a, g(f(b), a), f(b))
-> SUCCESS: L1 and L2 are identical after applying the MGU.
```

Program 8

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



Lab-8

FOL using forward chaining

Algo:

function FOL-FC-ASK(KB, α) returns a sub or false

inputs: KB, the Knowledge Base, a set of first-order definite clauses α, the query, an atomic sentence

local variables: new, the new sentence inferred on each iteration

repeat until new is empty

new ← { }

for each rule in KB do

(p1 ∧ .... ∧ pn ⇒ q) ← STANDARDIZE (rule)

for each θ such SUBST (θ, p1 ∧ .... ∧ pn) = SUBST (θ, p1' ∧ .. ∧ pn')

for some p1' .... pn' in KB

q' ← SUBST (θ, q)

if q' does not unify with some sentence already in KB or new then

add q' to new

φ ← UNIFY (q', α)

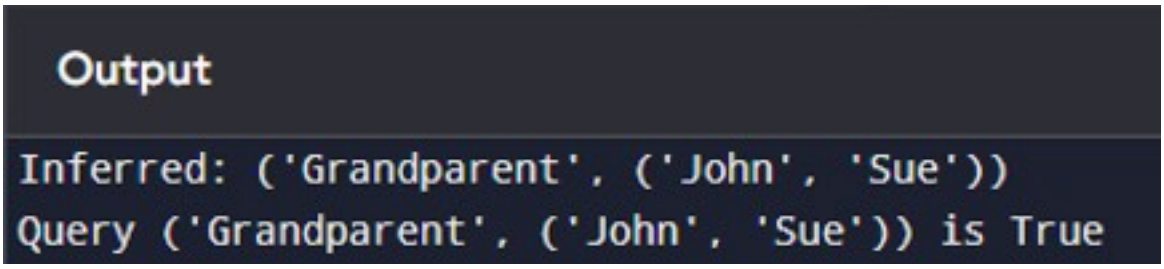
if φ is not fail then return φ

add new to KB

return false

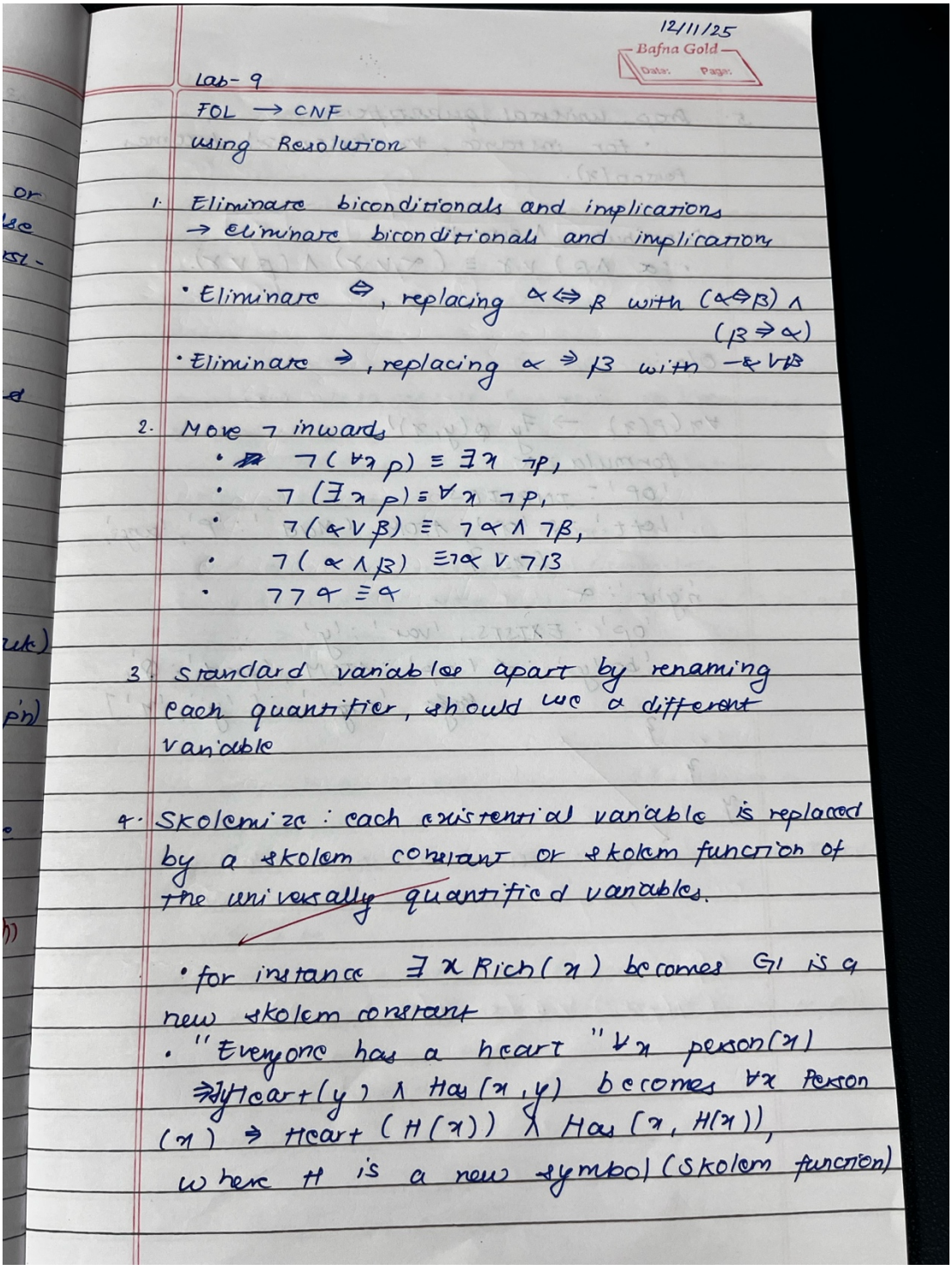
```
from typing import List, Tuple, Dict, Set, Union

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

class Rule:
    def __init__(self, head: Predicate, body: List[Predicate]):
        self.head = head
        self.body = body

    def __repr__(self):
        body_str = ', '.join(f"{p[0]}{p[1]}" for p in self.body)
        return f"{body_str} => {self.head[0]}{self.head[1]}"

# Knowledge base
class KnowledgeBase:
    def __init__(self):
        self.facts: Set[Predicate] = set()
        self.rules: List[Rule] = []

    def add_fact(self, fact: Predicate):
        self.facts.add(fact)

    def add_rule(self, rule: Rule):
        self.rules.append(rule)

    def forward_chain(self, query: Predicate) -> bool:
        inferred = set(self.facts)
        added = True

        while added:
            added = False
            for rule in self.rules:
                if all(self._match_fact(body_pred, inferred) for body_pred in rule.body):
                    if not self._match_fact(rule.head, inferred):
                        inferred.add(rule.head)
                        added = True
                        print(f"Inferred: {rule.head}")

                        if self._match_fact(query, inferred):
                            return True
        return self._match_fact(query, inferred)

    def _match_fact(self, pred: Predicate, fact_set: Set[Predicate]) -> bool:
        return pred in fact_set

# --- Example usage ---
if __name__ == "__main__":
```

```
kb = KnowledgeBase()

kb.add_fact(("Parent", ("John", "Mary")))
kb.add_fact(("Parent", ("Mary", "Sue")))

facts_list = list(kb.facts)
for f1 in facts_list:
    for f2 in facts_list:
        if f1[0] == "Parent" and f2[0] == "Parent":
            if f1[1][1] == f2[1][0]:
                head = ("Grandparent", (f1[1][0], f2[1][1]))
                body = [f1, f2]
                kb.add_rule(Rule(head, body))

query = ("Grandparent", ("John", "Sue"))
result = kb.forward_chain(query)

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



Program 9

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



12/11/25

Bafna Gold

Lab- 9

FOL → CNF

using Resolution

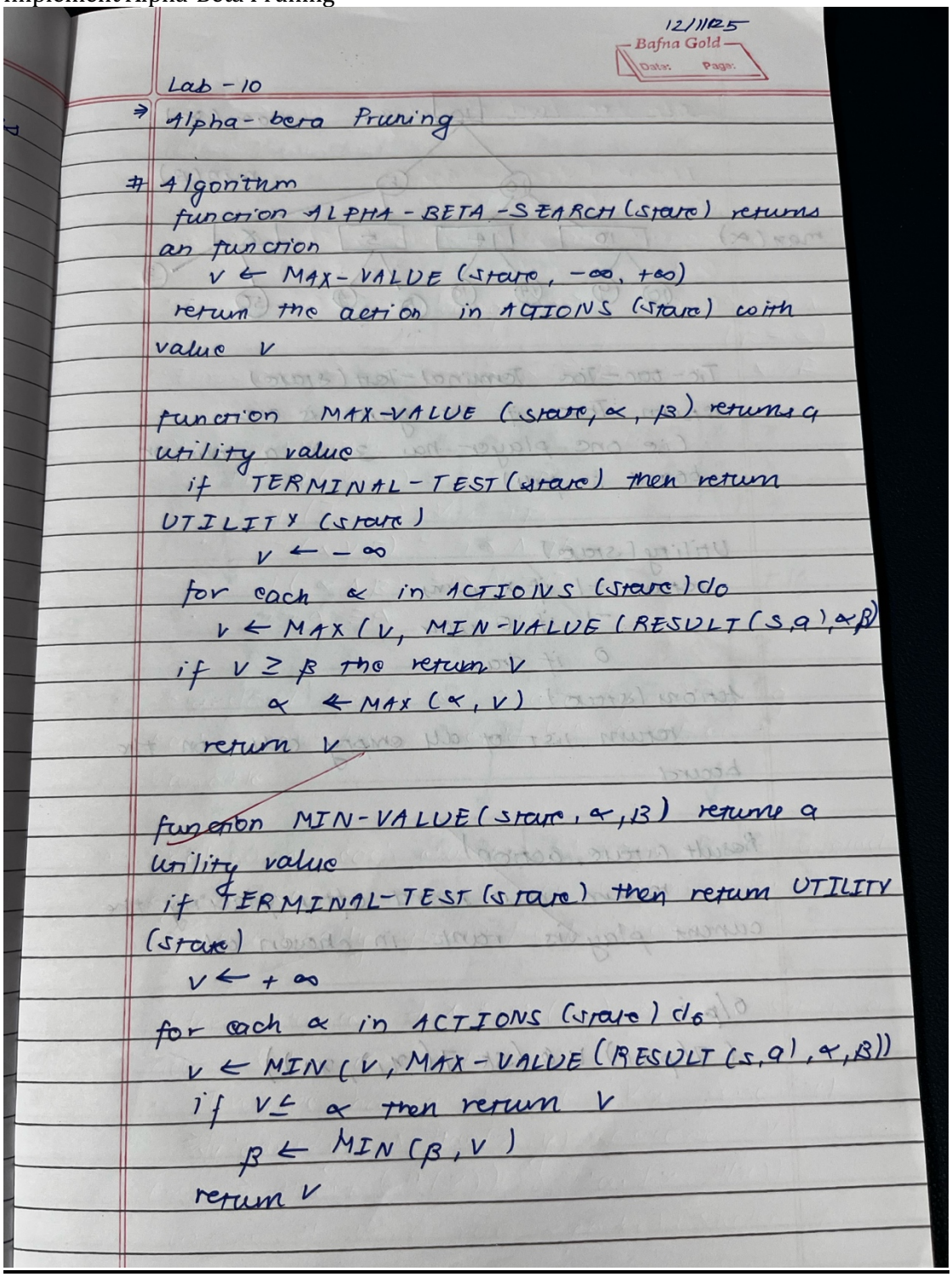
1. Eliminate biconditionals and implications
   → eliminate biconditionals and implications
   - Eliminate ⇔, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
   - Eliminate ⇒, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$
2. Move ¬ inwards
   - $\neg(\forall x\, p) \equiv \exists x\, \neg p$,
   - $\neg(\exists x\, p) \equiv \forall x\, \neg p$,
   - $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$,
   - $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
   - $\neg\neg\alpha \equiv \alpha$
3. Standard variables apart by renaming each quantifier, should use a different variable
4. Skolemize: each existential variable is replaced by a skolem constant or skolem function of the universally quantified variables.
   - for instance $\exists x$ Rich(x) becomes G1 is a new skolem constant
   - "Everyone has a heart" $\forall x$ person(x) $\Rightarrow$ Heart(y) $\wedge$ Has(x, y) becomes $\forall x$ Person(x) $\Rightarrow$ Heart(H(x)) $\wedge$ Has(x, H(x)), where H is a new symbol (Skolem function)

Output

```
Knowledge Base CNF Clauses:
   Parent(John,Mary)
   ~Parent(x,y)|Ancestor(x,y)
   Parent(Mary,Sam)
   ~Parent(x,y)|~Ancestor(y,z)|Ancestor(x,z)

Query: Ancestor(John, Sam) => TRUE
```

Program 10

Implement Alpha-Beta Pruning



12/11/25

Bafna Gold

Date: Page:

Lab - 10

⇒ Alpha - beta Pruning

Algorithm

function ALPHA - BETA - SEARCH (state) returns an function

v ← MAX - VALUE (state, −∞, +∞)

return the action in ACTIONS (state) with value v

function MAX-VALUE (state, α, β) returns a utility value

if TERMINAL - TEST (state) then return UTILITY (state)

v ← −∞

for each a in ACTIONS (state) do

v ← MAX (v, MIN-VALUE (RESULT (s, a), α, β))

if v ≥ β then return v

α ← MAX (α, v)

return v

function MIN-VALUE (state, α, β) returns a utility value

if TERMINAL-TEST (state) then return UTILITY (state)

v ← +∞

for each a in ACTIONS (state) do

v ← MIN (v, MAX-VALUE (RESULT (s, a), α, β))

if v ≤ α then return v

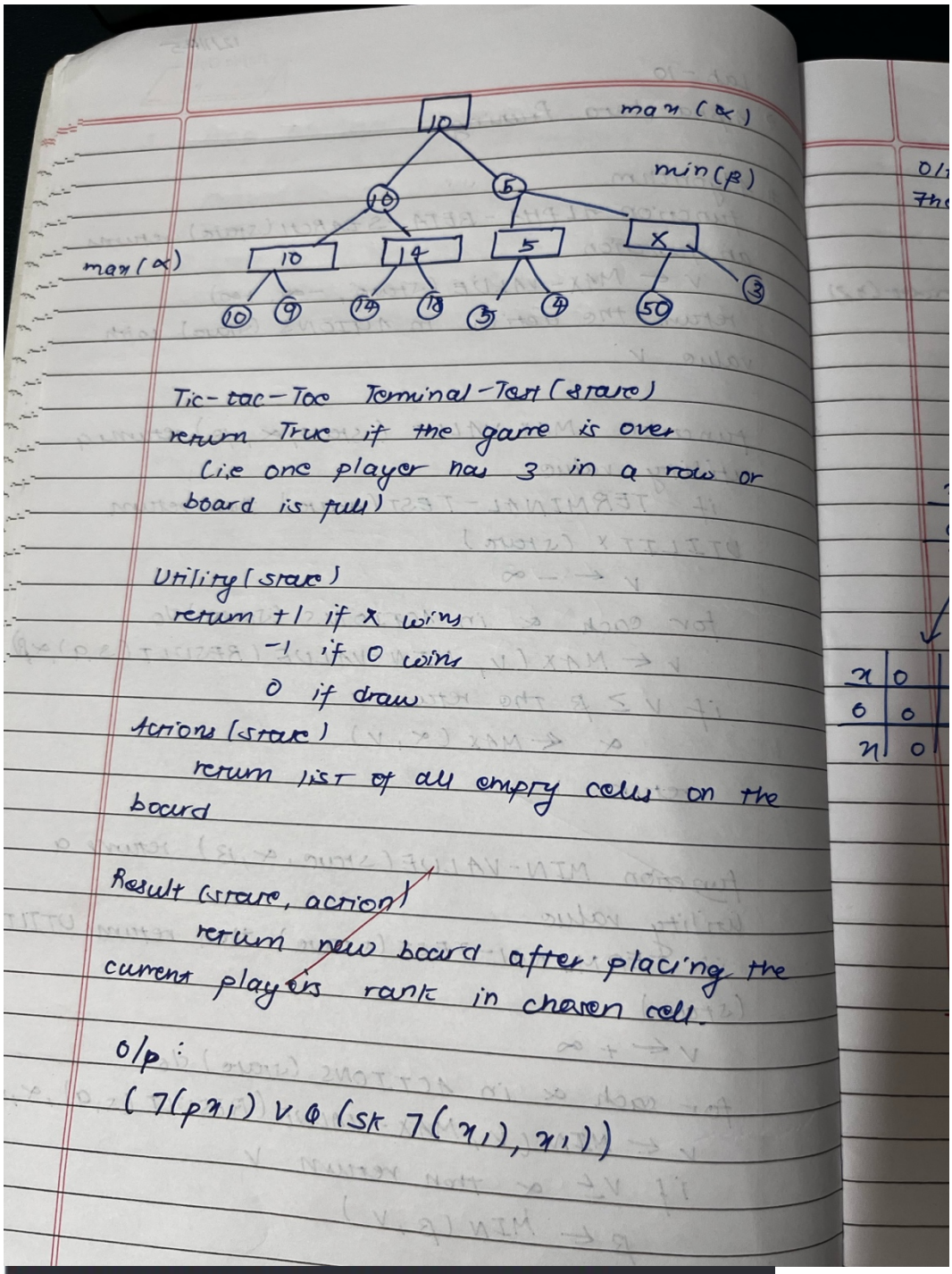
β ← MIN (β, v)

return v

Code:

```
import math

def alpha_beta_search(state):
    return max_value(state, -math.inf, math.inf)

def max_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state)
    v = -math.inf
    for a in actions(state):
        v = max(v, min_value(result(state, a), alpha, beta))
        if v >= beta:
            return v
        alpha = max(alpha, v)
    return v

def min_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state)
    v = math.inf
    for a in actions(state):
        v = min(v, max_value(result(state, a), alpha, beta))
        if v <= alpha:
            return v
        beta = min(beta, v)
    return v

values = [3, 5, 6, 9, 1, 2, 0, -1]
max_depth = 3

def terminal_test(state):
    return state >= len(values) // 2**(max_depth - depth[state])

def utility(state):
    return values[state]

def actions(state):
    if depth[state] == max_depth:
        return []
    return [0, 1]

def result(state, action):
    child = state * 2 + 1 + action
    depth[child] = depth[state] + 1
    return child
```

```
depth = {0: 0}
print("Optimal value:", alpha_beta_search(0))
print("")
```




```
Best value for maximizer: 3
```