VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shubhanshu Raj (1BM23CS325)

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

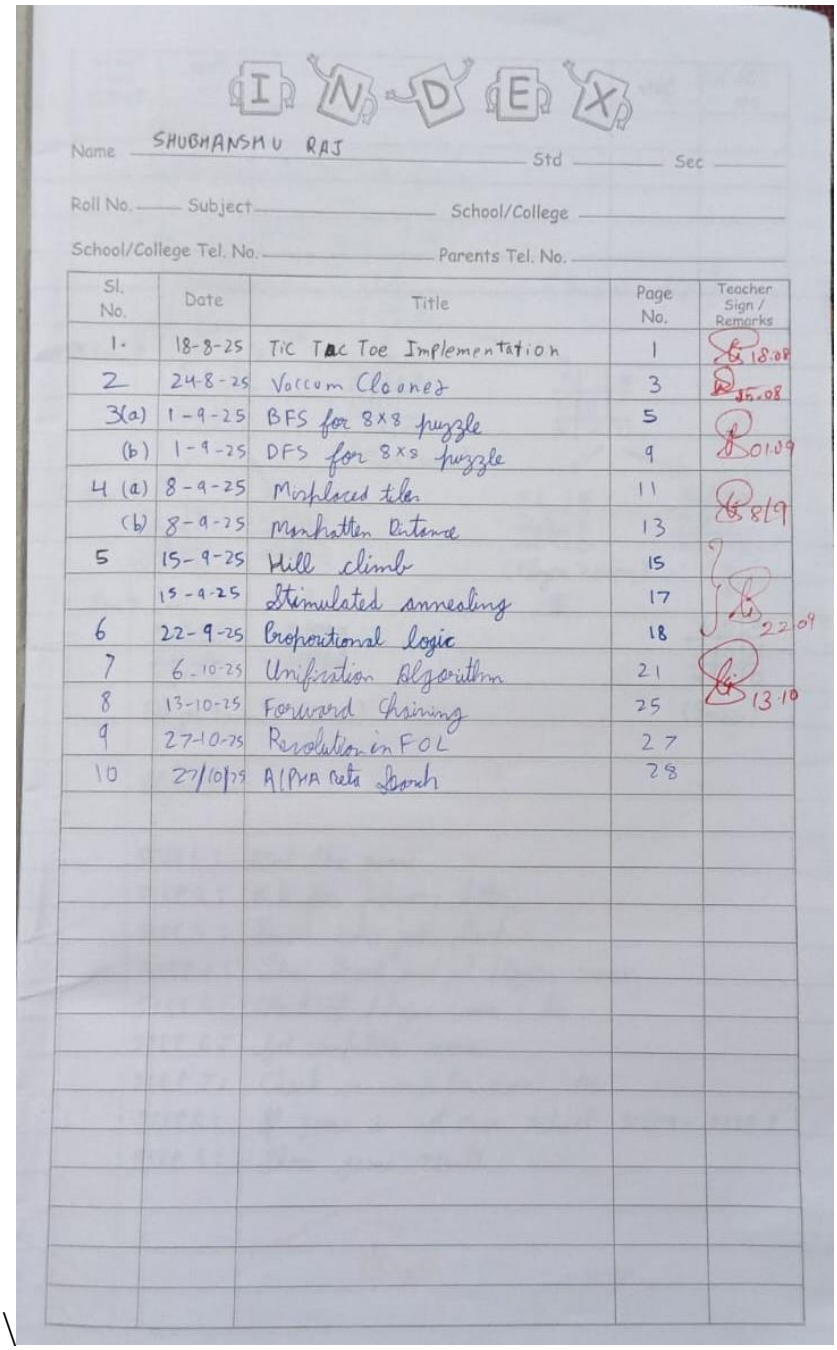
CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Shubhanshu Raj(1BM23CS325),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

Index



INDEX

Name SHUBHANSHU RAJ Std Sec

Roll No. Subject School/College

School/College Tel. No. Parents Tel. No.

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	18-8-25	TIC TAC Toe Implementation	1	18.08
2	24-8-25	Vaccum Cleaner	3	25.08
3(a)	1-9-25	BFS for 8x8 puzzle	5	
(b)	1-9-25	DFS for 8x8 puzzle	9	01.09
4 (a)	8-9-25	Misplaced tiles	11	8/9
(b)	8-9-25	Manhatten Distance	13	
5	15-9-25	Hill climb	15	
	15-9-25	Stimulated annealing	17	22.09
6	22-9-25	Propositional logic	18	
7	6-10-25	Unification Algorithm	21	13.10
8	13-10-25	Forward Chaining	25	
9	27-10-25	Resolution in FOL	27	
10	27/10/25	ALPHA Beta Search	28	

\

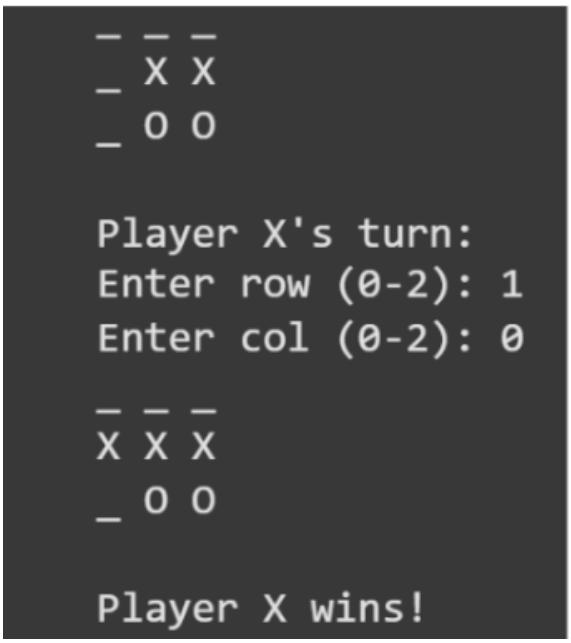
Git repo link: https://github.com/ShubhanshuRaj-BMS/1BM23CS325_ShubhanshuRaj_AI

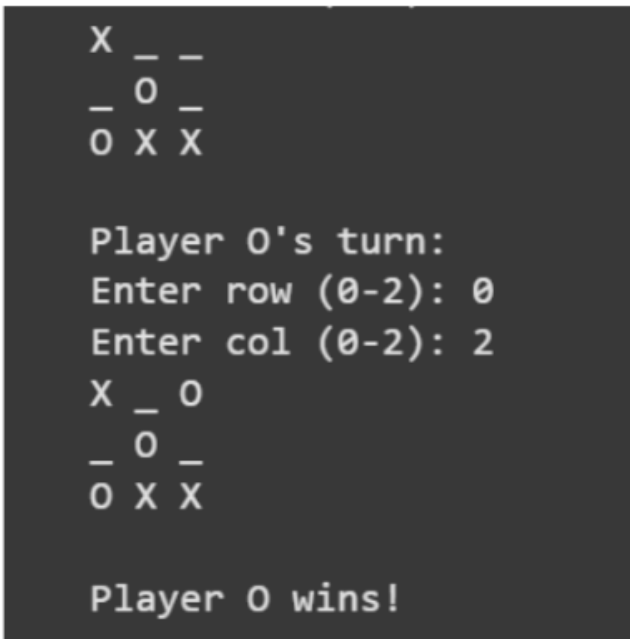
LAB 1

Code

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

def check_winner(board, player):

    for row in board:
        if all(cell == player for cell in row):
            return True

    for c in range(3):
        if all(board[r][c] == player for r in range(3)):
            return True

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

def is_full(board):
    return all(cell != "_" for row in board for cell in row)

def tic_tac_toe():
    board = [["_"] * 3 for _ in range(3)]
    current_player = "X"
```

```
    print("Tic Tac Toe: Player vs Player")
    print_board(board)

    while True:
        print(f"Player {current_player}'s turn:")
        row = int(input("Enter row (0-2): "))
        col = int(input("Enter col (0-2): "))

        if 0 <= row < 3 and 0 <= col < 3 and board[row][col] == "_":
            board[row][col] = current_player
            print_board(board)

            if check_winner(board, current_player):
                print(f"Player {current_player} wins!")
                break
            elif is_full(board):
                print("It's a draw!")
                break

            current_player = "O" if current_player == "X" else "X"
        else:
            print("Invalid move, try again.")

tic_tac_toe()
```


_ _ _
_ X X
_ O O
Player X's turn:
Enter row (0-2): 1
Enter col (0-2): 0
_ _ _
X X X
_ O O
Player X wins!

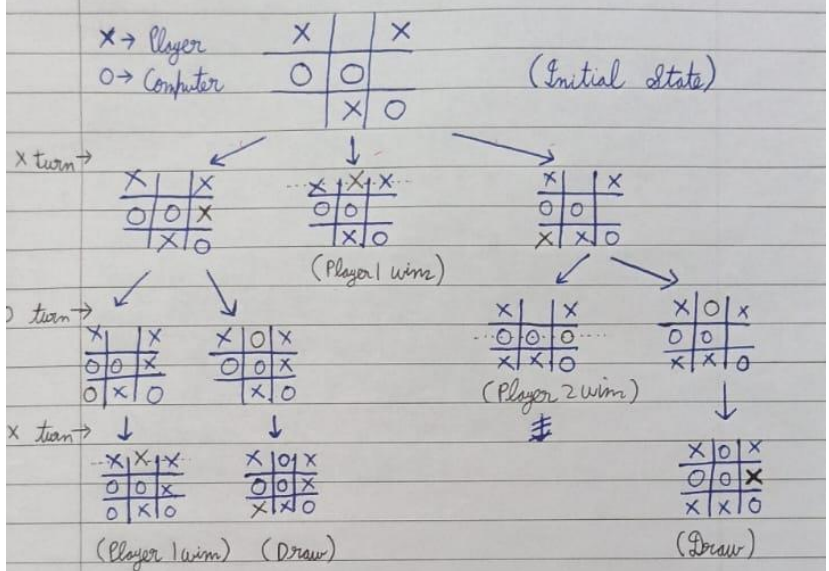

X _ _
_ O _
O X X
Player O's turn:
Enter row (0-2): 0
Enter col (0-2): 2
X _ O
_ O _
O X X
Player O wins!

Date 18/8/25
Page 1

LAB-0

Implement Tic TacToe Game -

X → Player
O → Computer



Algorithm -

STEP 1 : Start the game
STEP 2 : Ask for players letter
STEP 3 : Decide who goes first
STEP 4 : Show board and get players move
STEP 5 : Check if player won / tie
STEP 6 : Get computer's move
STEP 7 : Check if computer won / tie
STEP 8 : If game is not over repeat STEP 4 - STEP 7
STEP 9 : Show game result

18/8

LAB 2:

Code

```
import random

class Room:
    def __init__(self, name):
        self.name = name
        self.is_dirty = random.choice([True, False])

    def __str__(self):
        return f"Room '{self.name}' is {'dirty' if self.is_dirty else
'clean'}"

class VacuumCleanerAgent:
    def __init__(self, rooms):
        self.rooms = rooms
        self.current_room_index = 0
        self.clean_rooms = set()
        self.cost = 0

    def perceive(self):
        return self.rooms[self.current_room_index].is_dirty

    def act(self, percept):
        current_room = self.rooms[self.current_room_index]
        if percept:
            self.suck(current_room)
        else:
            pass
```

```
    def suck(self, room):
        print(f"Sucking dirt in {room.name}")
        room.is_dirty = False
        self.clean_rooms.add(room.name)

    def move_left(self):
        if self.current_room_index > 0:
            self.current_room_index -= 1
            self.cost += 1
            print(f"Moving to the previous room:
{self.rooms[self.current_room_index].name}")
        else:
            print("Already at the first room, can't move left!")

    def move_right(self):
        if self.current_room_index < len(self.rooms) - 1:
            self.current_room_index += 1
            self.cost += 1
            print(f"Moving to the next room:
{self.rooms[self.current_room_index].name}")
        else:
            print("Already at the last room, can't move right!")

    def user_move(self):
        while True:
            current_room = self.rooms[self.current_room_index]

            if current_room.name in self.clean_rooms:
                print(f"Room {current_room.name} is already clean.
Please move to another room.")
```

```
            move = input(f"Do you want to move left or right from
room {current_room.name}? (left/right): ").strip().lower()

            if move == "left" and current_room.name not in
self.clean_rooms:
                self.move_left()
                break
            elif move == "right" and current_room.name not in
self.clean_rooms:
                self.move_right()
                break
            else:
                if move not in ["left", "right"]:
                    print("Invalid input! Please enter 'left' or 'right'.")
                else:
                    print(f"Can't move to room {current_room.name} as
it's already clean. Try moving to a different room.")

    def are_all_rooms_clean(self):
        for room in self.rooms:
            if room.is_dirty:
                return False
        return True

if __name__ == "__main__":
    while True:
        num_rooms = int(input("Enter the number of rooms (max 4):
"))
        if 1 <= num_rooms <= 4:
            break
```

```
        else:
            print("Invalid input! Please enter a number between 1 and
4.")

    rooms = [Room(chr(65 + i)) for i in range(num_rooms)]

    agent = VacuumCleanerAgent(rooms)

    print("Initial state of the rooms:")
    for room in rooms:
        print(room)

    print("\nAgent in action:")
    while not agent.are_all_rooms_clean():
        percept = agent.perceive()
        agent.act(percept)

        if agent.are_all_rooms_clean():
            break

        agent.user_move()

    print("\nFinal state of the rooms:")
    for room in rooms:
        print(room)

    print(f"\nAll rooms are now clean!")
    print(f"Total movement cost: {agent.cost}")
    print("\nSHUBHANSHU RAJ")
    print("\n1BM23CS325")
```

```
Enter the number of rooms (max 4): 2
Initial state of the rooms:
Room 'A' is clean
Room 'B' is clean

Agent in action:

Final state of the rooms:
Room 'A' is clean
Room 'B' is clean

All rooms are now clean!
Total movement cost: 0

SHUBHANSHU RAJ

1BM23CS325
```

Date __/__/__
Page 3

LAB #1

Vaccum cleaning for 4 rooms
Input : Current location of vaccum cleaner
If there is dirt in a room (for all rooms)
Target : All dirt in room to be cleared.

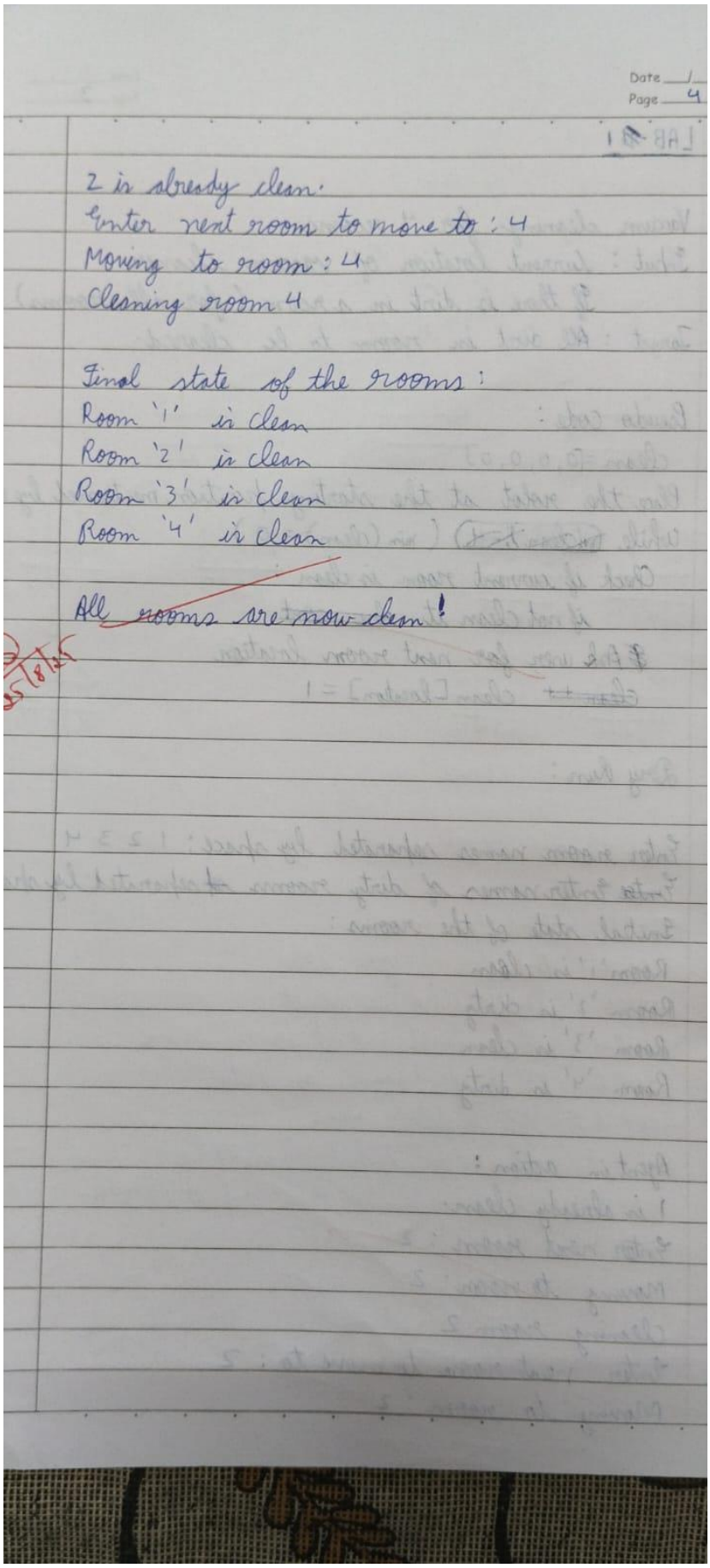
- Pseudo code :

clean = [0, 0, 0, 0]
Place the robot at the starting position mentioned by user
while (min(clean) = 0)
Check if current room is clean :
if not clean it
Ask user for next room location
clean [location] = 1

Dry Run :

Enter room names separated by space : 1 2 3 4
Enter names of dirty rooms separated by space : 2 4
Initial state of the rooms :
Room '1' is clean
Room '2' is dirty
Room '3' is clean
Room '4' is dirty

Agent in action :
1 is already clean.
Enter next room : 2
Moving to room : 2
Cleaning room 2
Enter next room to move to : 2
Moving to room : 2



2 is already clean.
Enter next room to move to : 4
Moving to room : 4
Cleaning room 4

Final state of the rooms :
Room '1' is clean
Room '2' is clean
Room '3' is clean
Room '4' is clean

All rooms are now clean !

LAB 3:

BFS
Code

```
from collections import deque

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

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 bfs(start_state):
    visited = set()
    queue = deque([(start_state, [])])

    while queue:
        current_state, path = queue.popleft()

        if current_state in visited:
            continue

        print("Visited state:")
        print_state(current_state)

        if current_state == goal_state:
            return path

        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
```

```
                queue.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 = bfs(start)

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

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

```
Solution found!
Moves: U U L D R
Number of moves: 5
Shubhanshu Raj

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
```

Date / /
Page 5

LAB 11

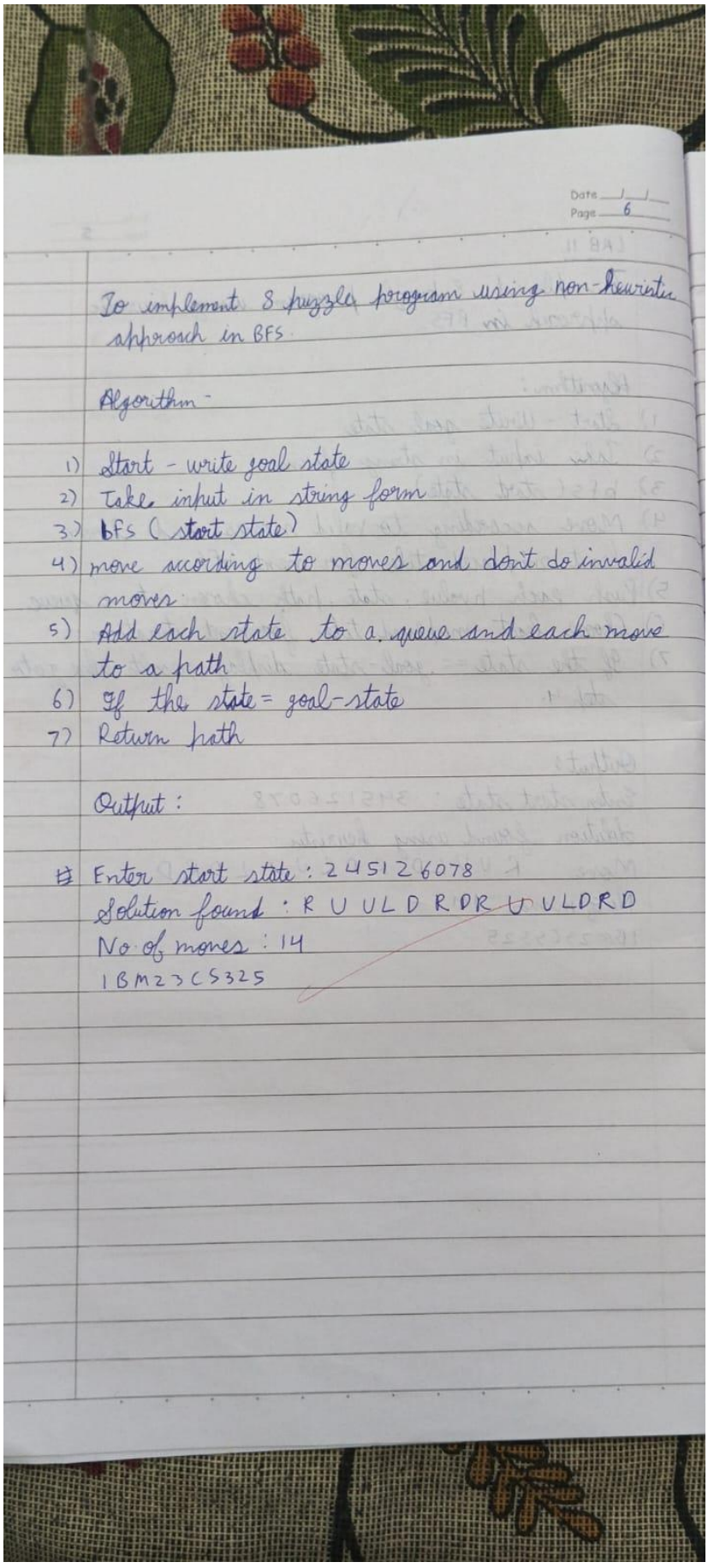
To implement 8 puzzle program using heuristic approach in BFS.

Algorithm:

1) Start - Write goal state
2) Take input in string form
3) bfs(start state)
4) Move according to valid moves and choose least misplaced tile for next bfs.
5) Push each h value, state, path chosen into a queue.
6) Choose least misplaced tile for next iteration.
7) If the state == goal-state display result else goto step 4.

Output:

Enter start state : 345126078
Solution found using heuristic:
Move : R U U L D R D R U U L D R D
No of moves : 14
1BM23CS325



Date / /
Page 6

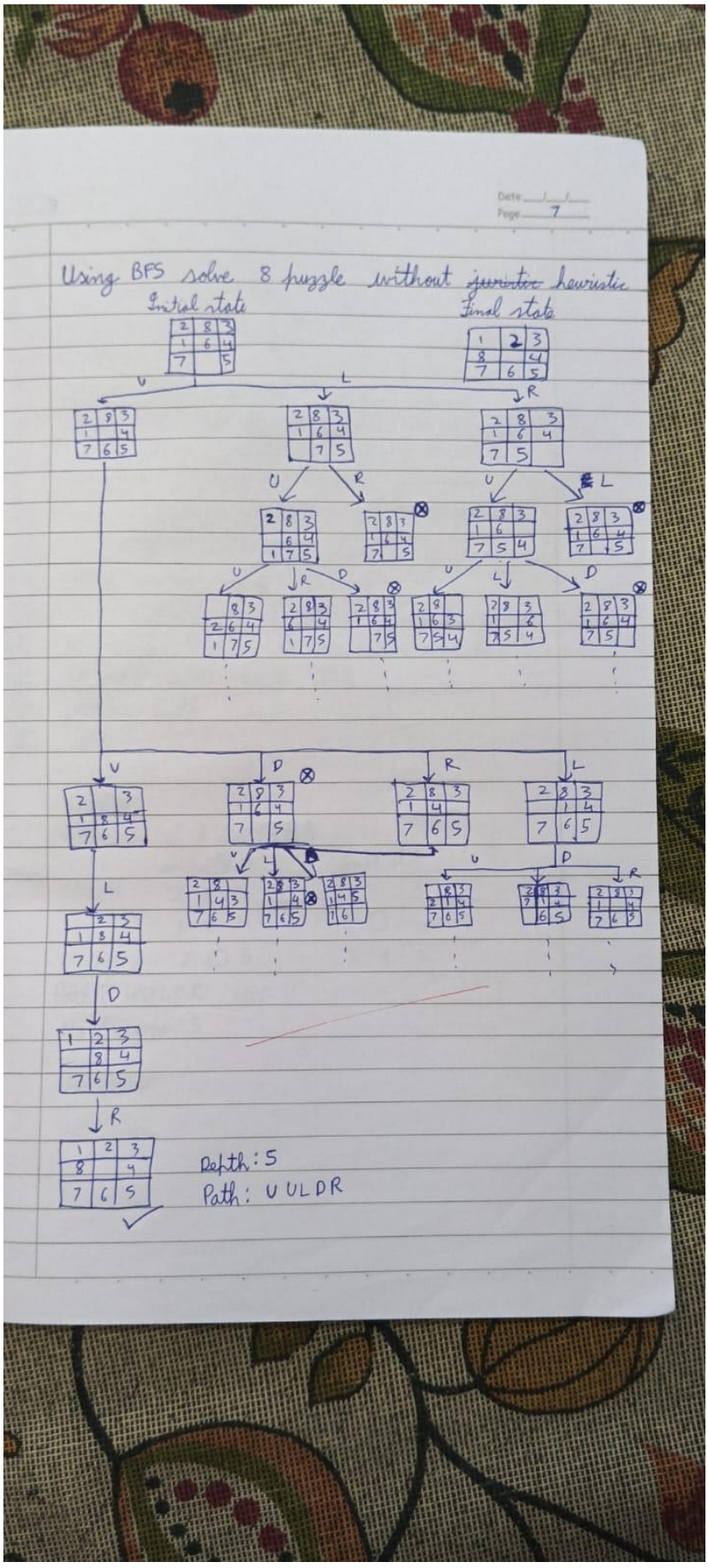
To implement 8 puzzle program using non-heuristic approach in BFS

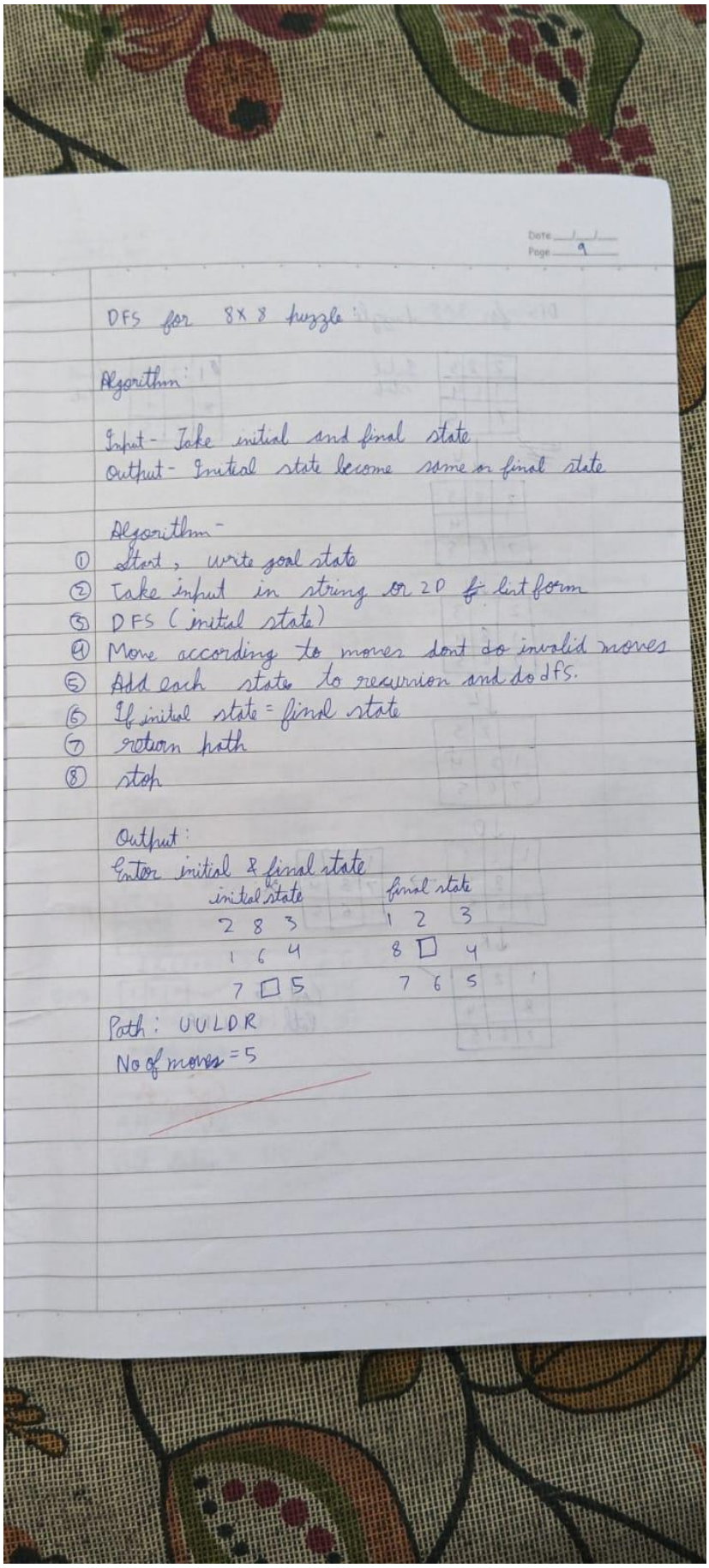
Algorithm -

1) Start - write goal state
2) Take input in string form
3) bFs (start state)
4) move according to moves and dont do invalid moves
5) Add each state to a queue and each move to a path
6) If the state = goal-state
7) Return path

Output :

Enter start state : 245126078
Solution found : R U UL D R DR U ULDRD
No of moves : 14
1BM23CS325


Date
Page 7
Using BFS solve 8 puzzle without ~~heuristic~~ heuristic
Initial state
Final state
Depth: 5
Path: U UL DR



Date / /
Page 9

DFS for 8x8 puzzle:

Algorithm:

Input - Take initial and final state
Output - Initial state become same as final state

Algorithm -
① Start, write goal state
② Take input in string or 2D list form
③ DFS (initial state)
④ Move according to moves dont do invalid moves
⑤ Add each state to recursion and do dfs.
⑥ If initial state = final state
⑦ return path
⑧ stop

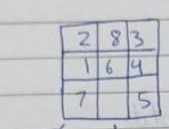
Output:
Enter initial & final state

initial state	final state
2 8 3	1 2 3
1 6 4	8 □ 4
7 □ 5	7 6 5

Path: UULDR
No of moves = 5

Date / /
Page 10

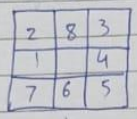
DFS for 8×8 puzzle:



Initial state

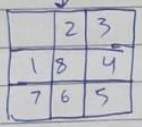
Final state

↓U

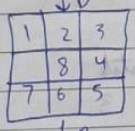


↓U

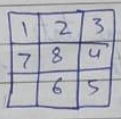
↓L



↓D

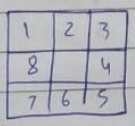


→



×

↓R



✓

Cost: 5
Path: ULLDR

Bolo

Dfs

```
from collections import deque

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

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 bfs(start_state):
    visited = set()
    queue = deque([(start_state, [])])

    while queue:
        current_state, path = queue.popleft()

        if current_state in visited:
            continue

        print("Visited state:")
        print_state(current_state)

        if current_state == goal_state:
            return path

        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                queue.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 = bfs(start)

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

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

```
Solution found!
Moves: U U L D R
Number of moves: 5
Shubhanshu Raj

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
```

LAB 4:

Manhatten distance

```
import heapq

goal_state = '123456780'

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

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

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

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

    state_list = list(state)
```

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

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

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

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

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

```
        if current_state == goal_state:
            return path, visited_count

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

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

    return None, visited_count

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

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

    result, visited_states = a_star(start)

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

    if result is not None:
        print("Solution found!")
```

```
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS325 Shubhanshu Raj\n")

        current_state = start
        g = 0  # initialize cost so far
        for i, move in enumerate(result, 1):
            new_state = move_tile(current_state, move)
            g += 1
            h = manhattan_distance(new_state)
            f = g + h
            print(f"Move {i}: {move}")
            print_state(new_state)
            print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n")
            current_state = new_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):
123678450
Start state:
1 2 3
6 7 8
4 5

Total states visit
ed: 21
Solution found!
Moves: L U L D R R
U L D R
Number of moves: 1
0
1BM23CS325 Shubhan
shu Raj

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

g(n) = 1, h(n) =
9, f(n) = g(n) + h
(n) = 10

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

Misplaced tiles

```
import heapq

goal_state = '123804765'

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 misplaced_tiles(state):
    """Heuristic: count of tiles not in their goal position (excluding
zero)."""
    return sum(1 for i, val in enumerate(state) if val != '0' and val !=
goal_state[i])

def a_star(start_state):
    visited_count = 0
    open_set = []
    heapq.heappush(open_set, (misplaced_tiles(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 + misplaced_tiles(new_state)
                heapq.heappush(open_set, (new_f, new_g, new_state,
path + [direction]))

    return None, visited_count

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

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

    result, visited_states = a_star(start)

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

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS325 Shubhanshu Raj\n")

        current_state = start
        g = 0  # initialize cost so far
        for i, move in enumerate(result, 1):
            new_state = move_tile(current_state, move)
```

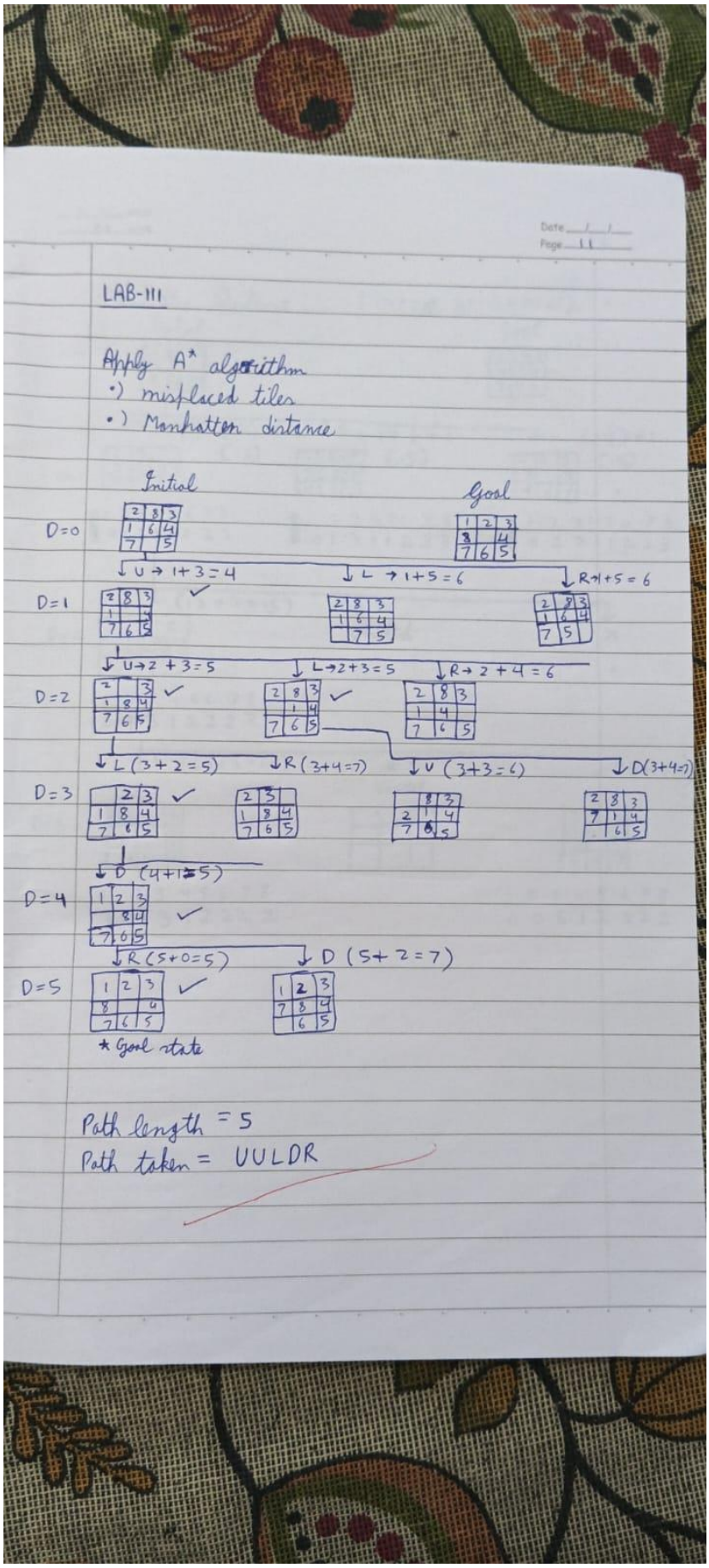
```
            g += 1
            h = misplaced_tiles(new_state)
            f = g + h
            print(f"Move {i}: {move}")
            print_state(new_state)
            print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n")
            current_state = new_state
    else:
        print("No solution exists for the given start state.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

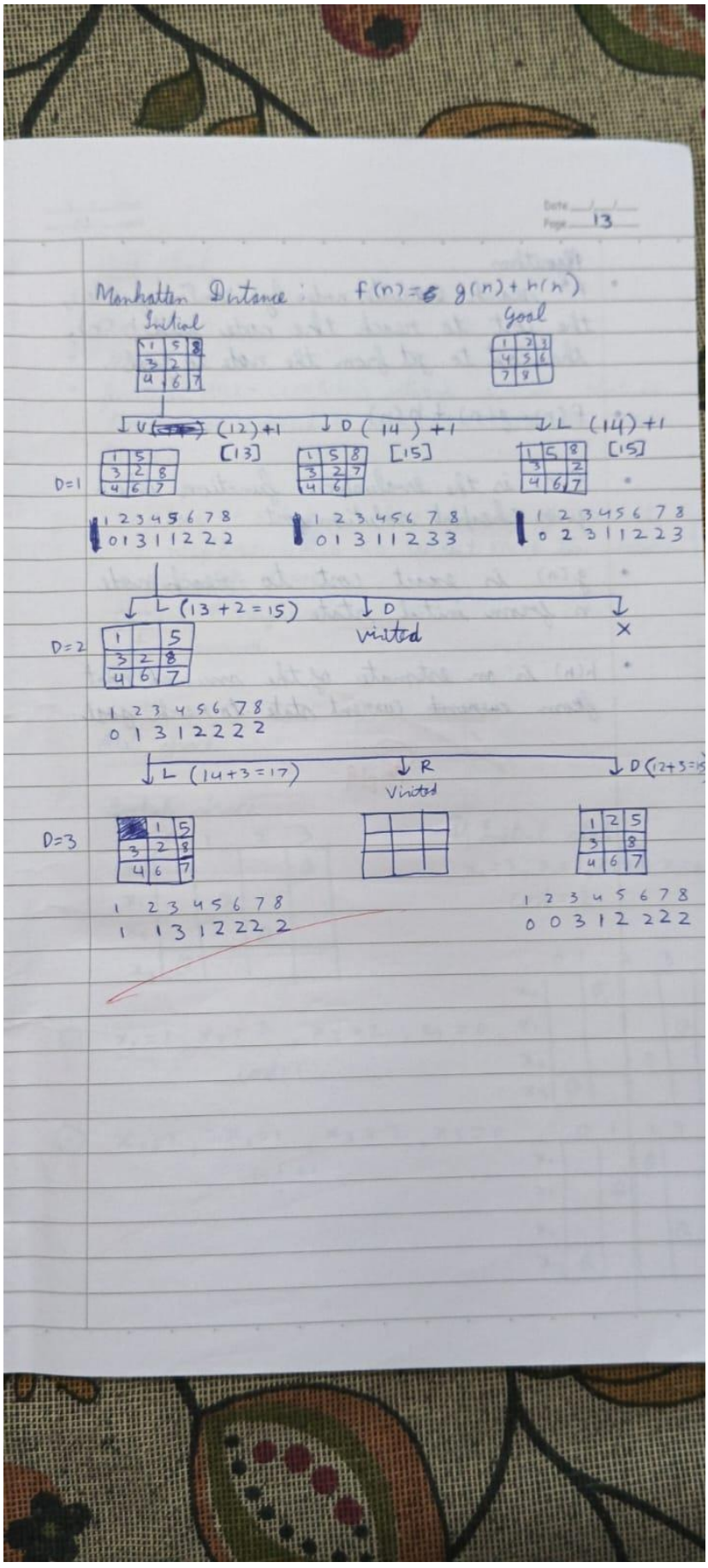
```
Total states visited: 7
Solution found!
Moves: U U L D R
Number of moves: 5
1BM23CS325 Shubhanshu Raj

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

g(n) = 1, h(n) = 3, f(n) = g
(n) + h(n) = 4

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

g(n) = 2, h(n) = 3, f(n) = g
(n) + h(n) = 5

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

g(n) = 3, h(n) = 2, f(n) = g
(n) + h(n) = 5

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

Algorithm

- A* search evaluates nodes by combining $g(n)$, the cost to reach the node and $h(n)$, the cost to get from the node to goal
- $f(n) = g(n) + h(n)$
- $f(n)$ is the evaluation function which gives cheapest solution cost
- $g(n)$ is exact cost to reach node n from initial state
- $h(n)$ is an estimate of the assumed cost from ~~current~~ current state to reach goal.

8 | 9


LAB-III
Apply A* algorithm
•) misplaced tiles
•) Manhattan distance
Initial
Goal
D=0
U → 1+3=4
L → 1+5=6
R → 1+5=6
D=1
U → 2+3=5
L → 2+3=5
R → 2+4=6
D=2
L (3+2=5)
R (3+4=7)
U (3+3=6)
D (3+4=7)
D=3
D (4+1=5)
D=4
R (5+0=5)
D (5+2=7)
D=5
* Goal state
Path length = 5
Path taken = UULDR


Date
Page 13
Manhattan Distance: f(n) = g(n) + h(n)
Initial
Goal
U (12)+1
D (14) +1
L (14)+1
[13]
[15]
[15]
D=1
1 2 3 4 5 6 7 8
0 1 3 1 1 2 2 2
1 2 3 4 5 6 7 8
0 1 3 1 1 2 3 3
1 2 3 4 5 6 7 8
0 2 3 1 1 2 2 3
L (13 + 2 = 15)
D
visited
X
D=2
1 2 3 4 5 6 7 8
0 1 3 1 2 2 2 2
L (14+3=17)
R
Visited
D=3
1 2 3 4 5 6 7 8
1 1 3 1 2 2 2 2
1 2 3 4 5 6 7 8
0 0 3 1 2 2 2 2

Lab 5:

Code

```
import random

def cost(state):
    """Calculate the number of attacking pairs of queens in the
current state."""
    attacking_pairs = 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):
                attacking_pairs += 1
    return attacking_pairs

def print_board(state):
    """Represent the state as a 4x4 board."""
    n = len(state)
    board = [['.' for _ in range(n)] for _ in range(n)]
    for i in range(n):
        board[state[i]][i] = 'Q'

    for row in board:
        print(" ".join(row))

def get_neighbors(state):
    """Generate all possible neighbors by swapping two queens."""
    neighbors = []
    n = len(state)
```

```
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = list(state)
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  # Swap queens
            neighbors.append(tuple(neighbor))
    return neighbors

def hill_climbing(initial_state):
    """Hill climbing algorithm to solve the N-Queens problem."""
    current = initial_state
    print(f"Initial state:")
    print_board(current)
    print(f"Cost: {cost(current)}")
    print('-' * 20)

    while True:
        neighbors = get_neighbors(current)
        # Select the neighbor with the lowest cost
        next_state = min(neighbors, key=lambda x: cost(x))
        print(f"Next state:")
        print_board(next_state)
        print(f"Cost: {cost(next_state)}")
        print('-' * 20)

        if cost(next_state) >= cost(current):
            # If no better state is found, return the current state as the solution
            print(f"Solution found:")
            print_board(current)
            print(f"Cost: {cost(current)}")
```

```
            return current
        current = next_state

if __name__ == "__main__":
    # Initial state for 4-Queens, random placement
    initial_state = (3, 1, 2, 0)  # Example initial state, where each index represents a column

    # Run Hill Climbing algorithm
    solution = hill_climbing(initial_state)
print("Shubhanshu Raj 1BM23CS325")
```

```
--------------------
Next state:
. . Q .
Q . . .
. . . Q
. Q . .
Cost: 0
--------------------
Next state:
. . Q .
. Q . .
. . . Q
Q . . .
Cost: 1
--------------------
Solution found:
. . Q .
Q . . .
. . . Q
. Q . .
Cost: 0
Shubhanshu Raj 1BM23CS325
```

Date / /
Page 15

Hill climb

Algorithm :

- Start
- function HILL-CLIMBING returns a state that is local max current ← MAKE-NODE (problem INITIAL-STATE)
  loop do
  neighbour ← a highest valued successor of current
  if neighbour VALUE <= current VALUE then return current ~~state~~ STATE
  current ← neighbour
  return answer
  END

State space

Initial space

	0	1	2	3
x_0				Q
x_1		Q		
x_2			Q	
x_3	Q			

① Initial state

$x_0 = 3, x_1 = 1, x_2 = 2, x_3 = 0$

cost = 2

	0	1	2	3
x_0		Q		
x_1				Q
x_2			Q	
x_3	Q			

ii) $x_0 = 1, x_1 = 3, x_2 = 2, x_3 = 0$

cost = 1

	0	1	2	3
x_0			Q	
x_1		Q		
x_2				Q
x_3	Q			

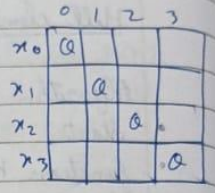
iii) $x_0 = 2, x_1 = 1, x_2 = 3, x_3 = 0$

cost = 1

Date / /
Page 16

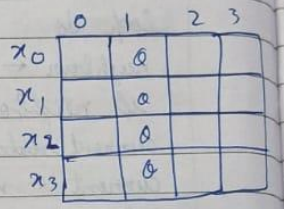
(iv) $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$

Cost = 6



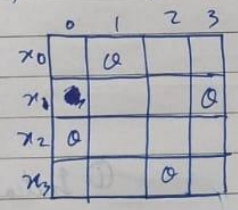
(v) $x_0 = 1$, $x_1 = 1$, $x_2 = 1$, $x_3 = 1$

Cost = 6



(vi) $x_0 = 1$, $x_1 = 3$, $x_2 = 0$, $x_3 = 2$

Cost = 0



Lab 6:
Code

```
import pandas as pd
from itertools import product
import re

def tokenize(sentence):
    # Tokenize based on logical operators, parentheses, and symbols
    # Tokens are: '(', ')', 'and', 'or', 'not', variables (letters),
whitespace ignored
    token_pattern = r'\w+|[()]'
    return re.findall(token_pattern, sentence)

def pl_true(sentence, model):
    # Tokenize the sentence
    tokens = tokenize(sentence)

    # Logical operators in Python are lowercase
    logical_ops = {'and', 'or', 'not'}

    evaluated_tokens = []
    for token in tokens:
        token_lower = token.lower()
        if token_lower in logical_ops:
            # Keep operators as is (lowercase)
            evaluated_tokens.append(token_lower)
        elif token in model:
            # Replace symbol with 'True' or 'False' string
```

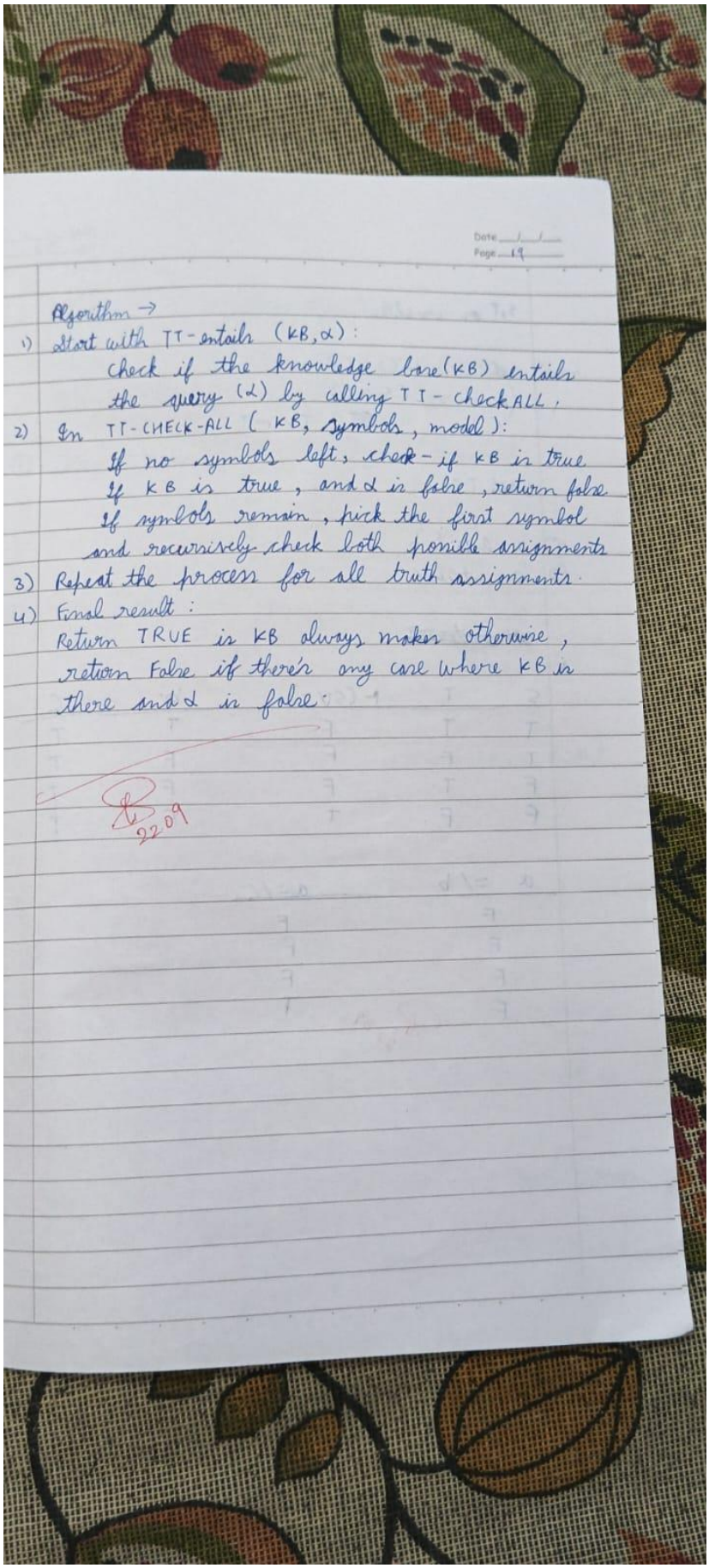
```
            evaluated_tokens.append(str(model[token]))
        else:
            # Parentheses or unknown tokens are kept as is
            evaluated_tokens.append(token)

    # Join tokens with spaces for safe eval
    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 or C': model_dict.get('A', False) or model_dict.get('C',
False),
            'B or not 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):
    # Find unique uppercase letters as symbols
    return sorted(set(re.findall(r'[A-Z]', kb + alpha)))

# Example usage:

kb = "(A or C) and (B or not C)"
alpha = "A or B"

symbols = get_symbols(kb, alpha)

result, truth_table = tt_entails(kb, alpha, symbols)

print("Truth Table:")
display(truth_table)

if result:
    print("\nKB entails α")
else:
    print("\nKB does not entail α")
print("Shubhanshu Raj 1BM23CS325")
```

Truth Table:

	A	B	C	A or C	B or not C	KB	α
0	False	False	False	False	True	False	False
1	False	False	True	True	False	False	False
2	False	True	False	False	True	False	True
3	False	True	True	True	True	True	True
4	True	False	False	True	True	True	True
5	True	False	True	True	False	False	True
6	True	True	False	True	True	True	True
7	True	True	True	True	True	True	True

KB entails α
Shubhanshu Raj 1BM23CS325

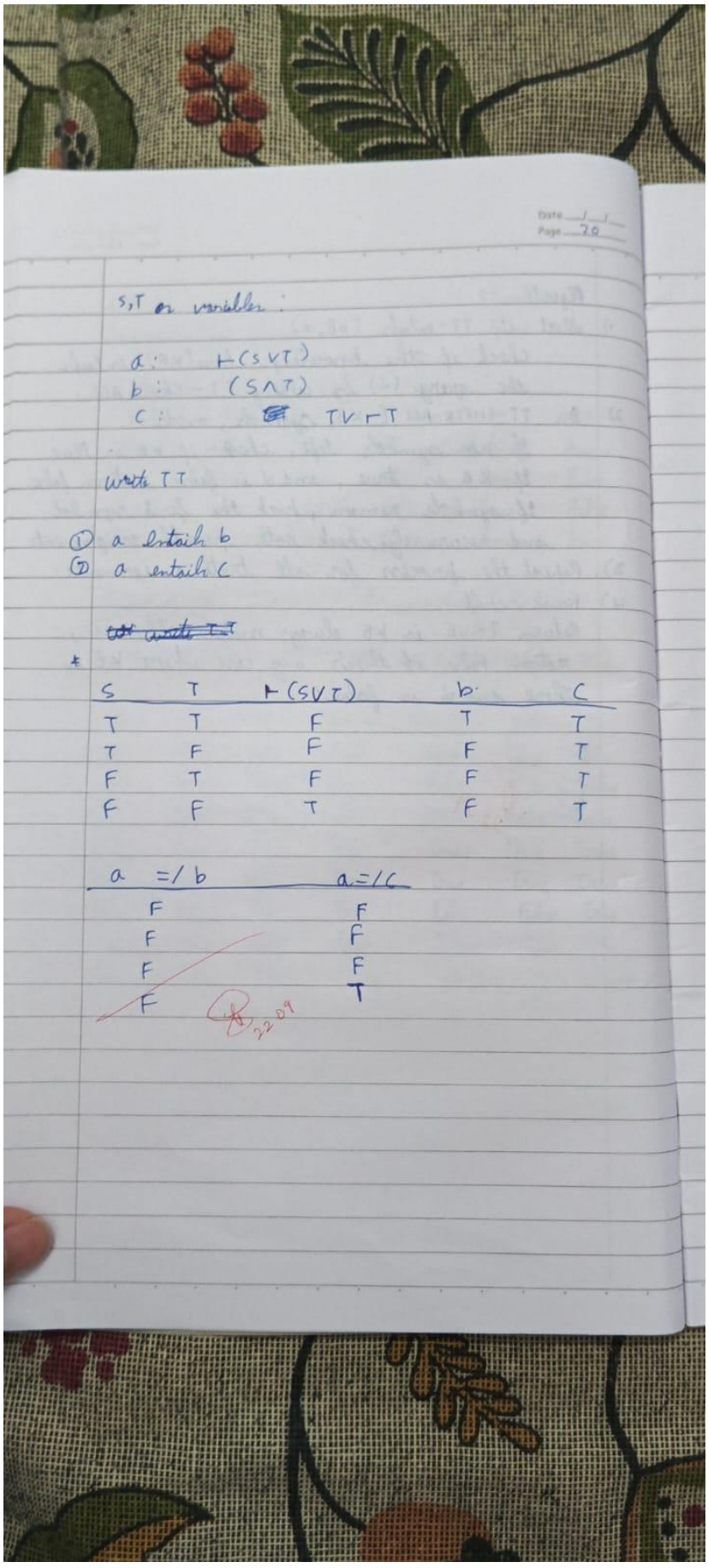


Date / /
Page 19

Algorithm →

1) Start with TT-entails (KB, α):
   Check if the knowledge base (KB) entails the query (α) by calling TT-checkALL.
2) In TT-CHECK-ALL (KB, symbols, model):
   If no symbols left, check - if KB is true.
   If KB is true, and α is false, return false.
   If symbols remain, pick the first symbol and recursively check both possible assignments.
3) Repeat the process for all truth assignments.
4) Final result:
   Return TRUE is KB always makes otherwise, return False if there's any case where KB is there and α is false.

22/09



Date / /
Page 70

S, T are variables:

a: $\vdash(S \vee T)$
b: $(S \wedge T)$
c: $T \vee \neg T$

with T T

① a entails b
② a entails c

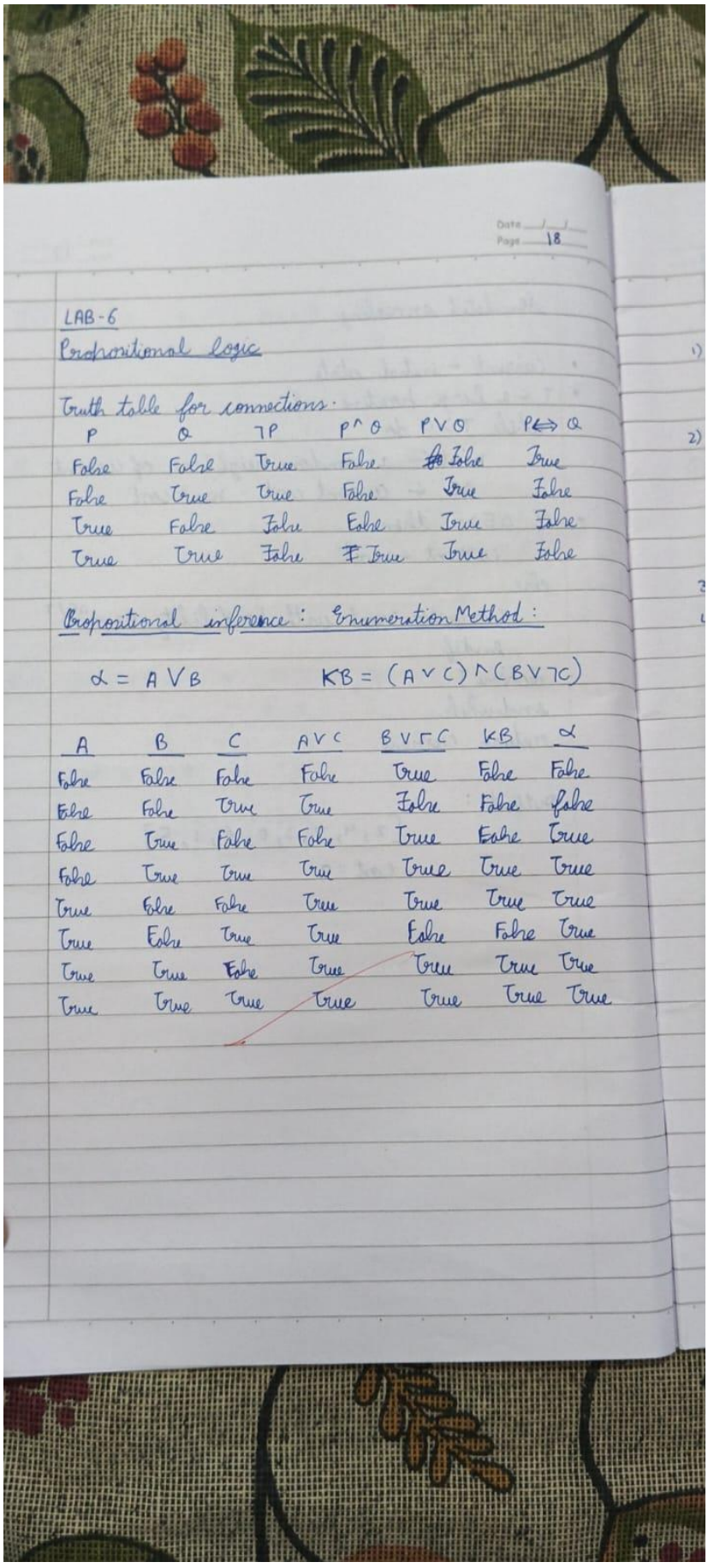
~~at most 1~~

*

S	T	$\vdash(S \vee T)$	b	c
T	T	F	T	T
T	F	F	F	T
F	T	F	F	T
F	F	T	F	T

a	=/ b	a =/ c
F		F
F		F
F		F
F		T

22.09



Date / /
Page 18

LAB-6

Propositional logic

Truth table for connections:

P	Q	¬P	P ^ Q	P V Q	P ⇔ Q
False	False	True	False	False	True
False	True	True	False	True	False
True	False	False	False	True	False
True	True	False	True	True	False

Propositional inference: Enumeration Method:

$\alpha = A \vee B$ $\quad KB = (A \vee C) \wedge (B \vee \neg C)$

A	B	C	A V C	B V ¬C	KB	α
False	False	False	False	True	False	False
False	False	True	True	False	False	False
False	True	False	False	True	False	True
False	True	True	True	True	True	True
True	False	False	True	True	True	True
True	False	True	True	False	False	True
True	True	False	True	True	True	True
True	True	True	True	True	True	True

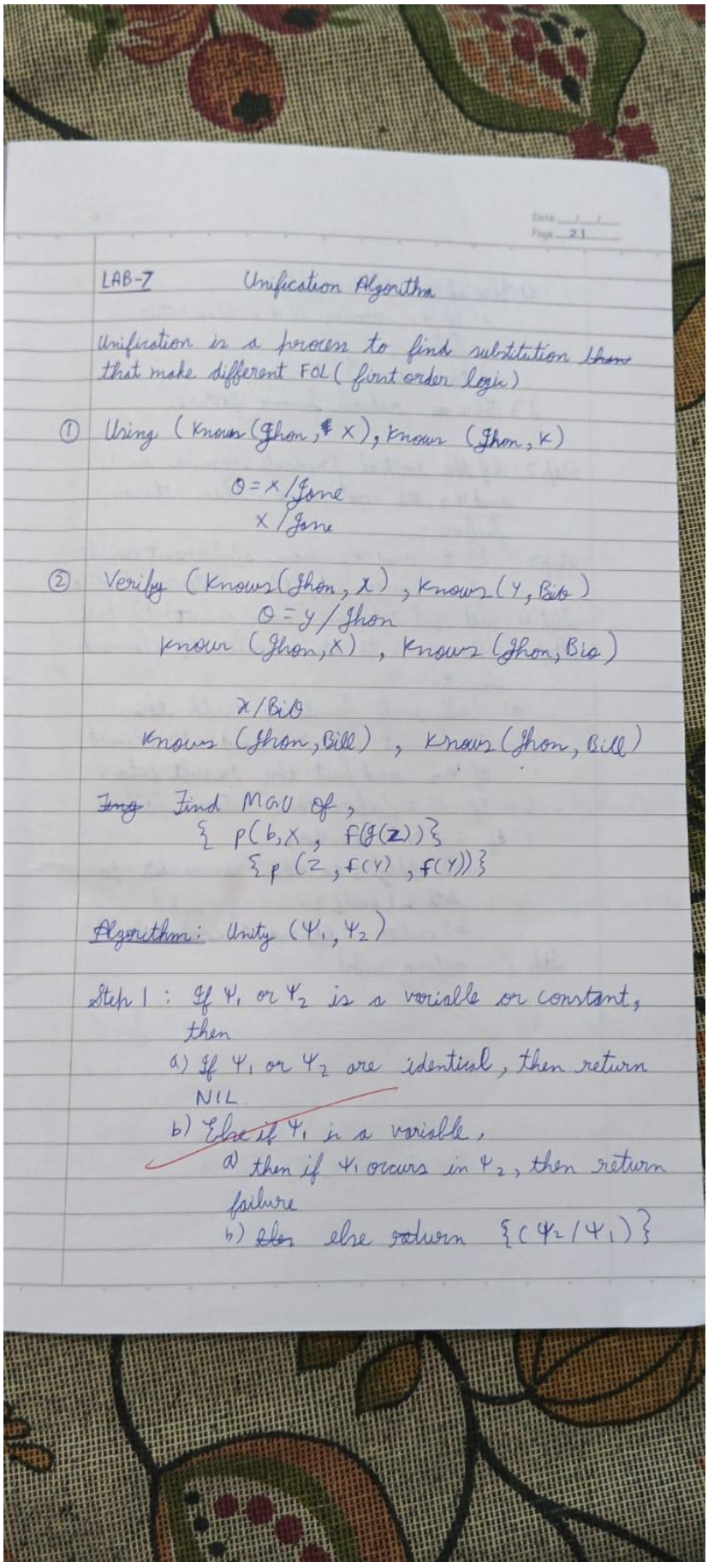
Lab 7:

Code

```
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []

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

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def infer(self):
        new_inferred = True
        while new_inferred:
            new_inferred = False
            for premises, conclusion in self.rules:
                # Check if all premises are true (in facts)
                if all(premise in self.facts for premise in premises):
                    if conclusion not in self.facts:
                        print(f"Inferred: {conclusion} from {premises}")
                        self.facts.add(conclusion)
                        new_inferred = True
        return self.facts

# Define the KB
kb = KnowledgeBase()
```

```
# Add facts
kb.add_fact(('American', 'Robert'))
kb.add_fact(('Hostile', 'CountryA'))
kb.add_fact(('Missile', 'm1'))
kb.add_fact(('Sells', 'Robert', 'CountryA', 'm1'))

# Add rules
# Rule 1: Missile is a Weapon
kb.add_rule([('Missile', 'm1')], ('Weapon', 'm1'))

# Rule 2: If American(x) & Weapon(w) & Hostile(n) & Sells(x,n,w)
=> Criminal(x)
kb.add_rule(
    [('American', 'Robert'), ('Weapon', 'm1'), ('Hostile', 'CountryA'),
('Sells', 'Robert', 'CountryA', 'm1')],
    ('Criminal', 'Robert')
)

# Run inference
final_facts = kb.infer()

print("\nFinal facts:")
for fact in final_facts:
    print(fact)

# Check if Robert is criminal
if ('Criminal', 'Robert') in final_facts:
    print("\nConclusion: Robert is criminal.")
else:
    print("\nConclusion: Robert is NOT criminal.")
```

```
print("Shubhanshu Raj 1BM23CS325")

Inferred: ('Weapon', 'm1') from [('Missile', 'm1')]
Inferred: ('Criminal', 'Robert') from [('American', 'Robert'),
('Weapon', 'm1'), ('Hostile', 'CountryA'), ('Sells', 'Robert',
'CountryA', 'm1')]

Final facts:
('American', 'Robert')
('Missile', 'm1')
('Criminal', 'Robert')
('Sells', 'Robert', 'CountryA', 'm1')
('Weapon', 'm1')
('Hostile', 'CountryA')

Conclusion: Robert is criminal.
Shubhanshu Raj 1BM23CS325
```



LAB-7 Unification Algorithm

Unification is a process to find substitution that make different FOL (first order logic)

① Using (Knows (Jhon, X), Knows (Jhon, X)

$\theta = X/Jane$

$X/Jane$

② Verify (Knows(Jhon, X), Knows(Y, Bill))

$\theta = Y/Jhon$

Knows (Jhon, X), Knows (Jhon, Bill)

$X/Bill$

Knows (Jhon, Bill), Knows (Jhon, Bill)

Find MGU of,

{ p(b, X, f(g(z)) }

{ p (z, f(Y), f(Y)) }

Algorithm: Unify (Ψ_1, Ψ_2)

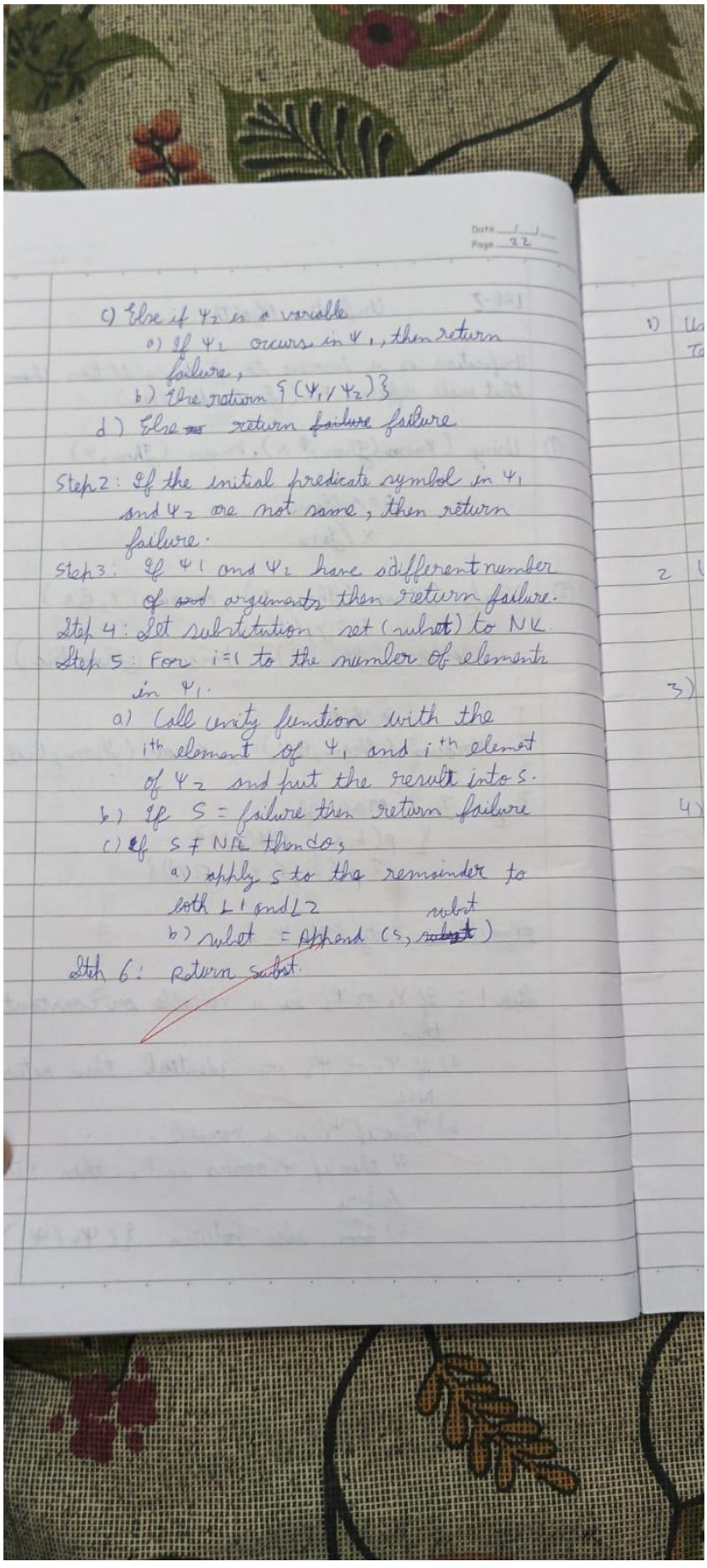
Step 1 : If Ψ_1 or Ψ_2 is a variable or constant, then

a) If Ψ_1 or Ψ_2 are identical, then return NIL

b) Else if Ψ_1 is a variable,

a) then if Ψ_1 occurs in Ψ_2, then return failure

b) else return $\{(\Psi_2/\Psi_1)\}$



c) Else if Ψ_2 is a variable
   a) If Ψ_2 occurs in Ψ_1, then return failure,
   b) Else return $\{(\Psi_1 / \Psi_2)\}$
d) Else ~~return failure~~ failure

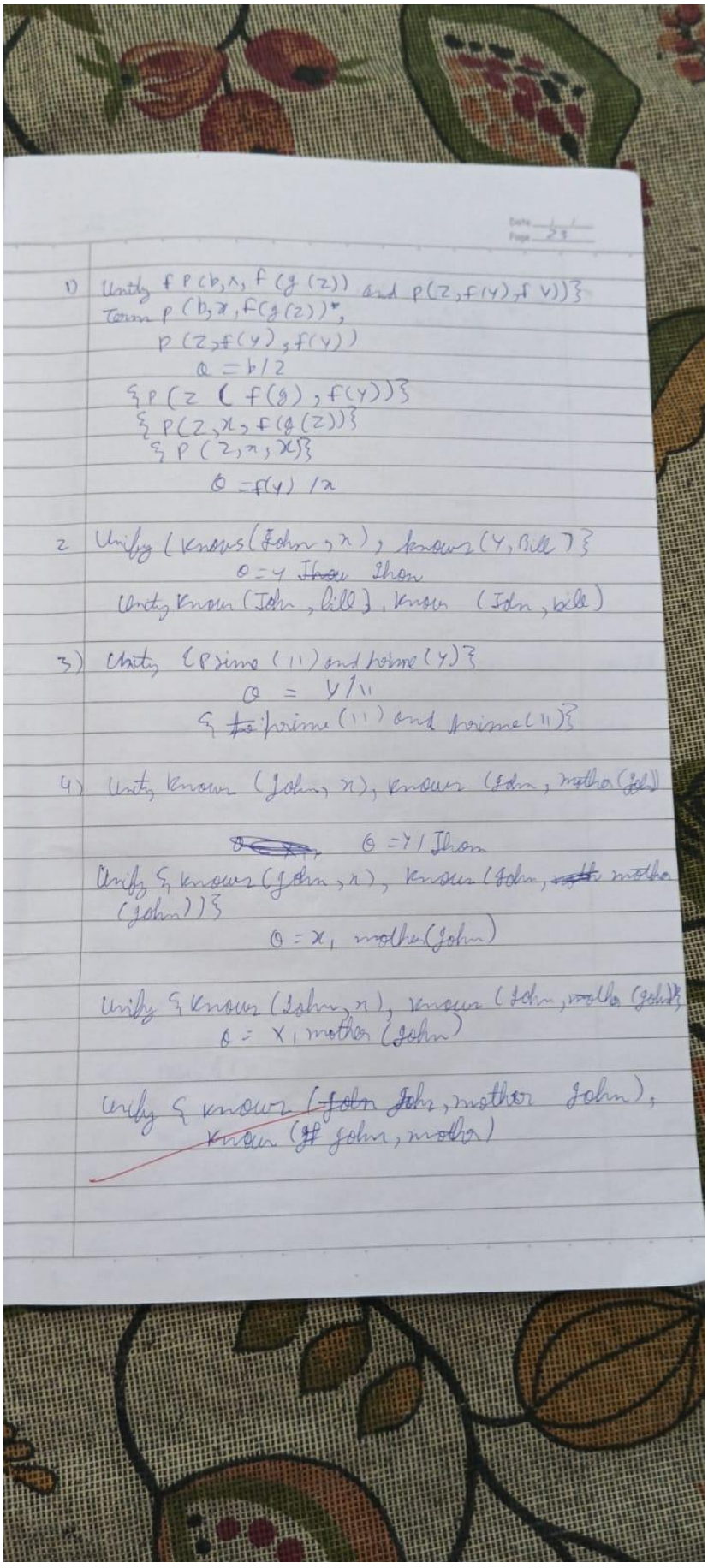
Step 2: If the initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return failure.

Step 3: If Ψ_1 and Ψ_2 have different number of ~~and~~ arguments then return failure.

Step 4: Let substitution set (~~subst~~) to NIL.

Step 5: For i=1 to the number of elements in Ψ_1.
   a) Call unify function with the i^{th} element of Ψ_1 and i^{th} element of Ψ_2 and put the result into S.
   b) If S = failure then return failure
   c) If S ≠ NIL then do;
      a) apply S to the remainder to both L_1 and L_2
      b) subst = Append (S, ~~subst~~ subst)

Step 6: Return ~~subst~~.



1) Unify { p(b, x, f(g(z)) and p(z, f(y), f(y))}
Term p(b, x, f(g(z))*,
p(z, f(y), f(y))
θ = b/z
{p(z (f(g), f(y))}
{p(z, x, f(g(z))}
{p(z, x, x)}
θ = f(y)/x

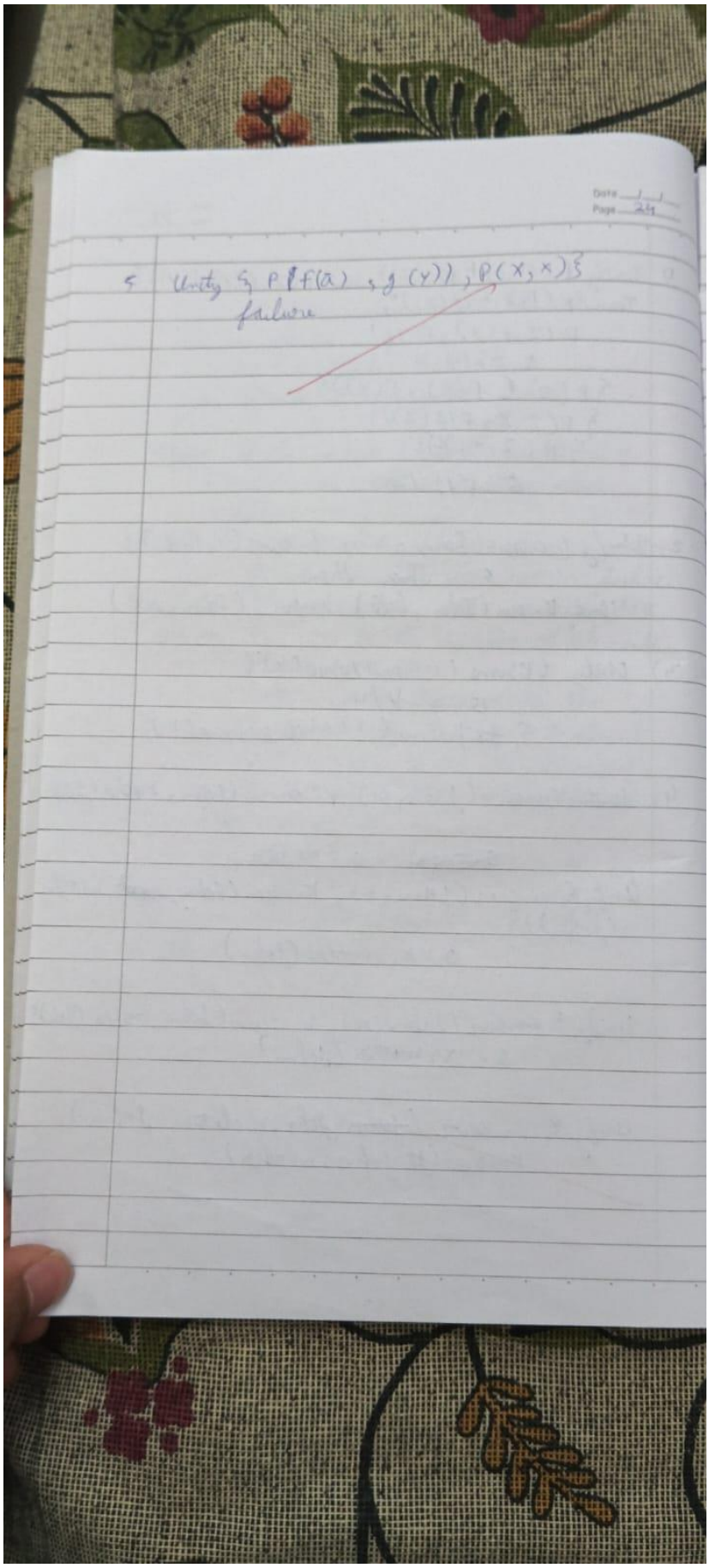
2 Unify {knows(John, x), knows(y, Bill)}
θ = y John
Unify knows(John, Bill), knows(John, Bill)

3) Unify {prime(11) and prime(y)}
θ = y/11
{prime(11) and prime(11)}

4) Unify knows(John, x), knows(John, mother(John))
θ = y/John
Unify {knows(John, x), knows(John, mother(John))}
θ = x, mother(John)

Unify {knows(John, x), knows(John, mother(John))}
θ = x, mother(John)

Unify {knows John, mother John),
knows(John, mother)


Date
Page
Unify {P(f(a), g(y)), P(x, x)}
failure

Lab 8:

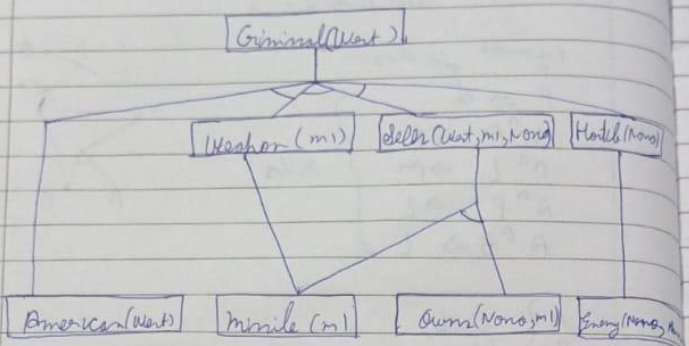
Code

```
from collections import deque

class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
        self.inferred = set()

    def add_fact(self, fact):
        if fact not in self.facts:
            print(f"Adding fact: {fact}")
            self.facts.add(fact)
            return True
        return False

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def forward_chain(self):
        agenda = deque(self.facts)

        while agenda:
            fact = agenda.popleft()
            if fact in self.inferred:
                continue
            self.inferred.add(fact)
```

```
            for (premises, conclusion) in self.rules:
                if all(p in self.inferred for p in premises):
                    if conclusion not in self.facts:
                        print(f"Inferred new fact: {conclusion} from
{premises} => {conclusion}")
                        self.facts.add(conclusion)
                        agenda.append(conclusion)

                        if conclusion == 'Criminal(West)':
                            print("\n☑ Goal Reached: West is Criminal")
                            return True
        return False
kb = KnowledgeBase()


kb.add_fact('American(West)')
kb.add_fact('Enemy(Nono, America)')
kb.add_fact('Missile(M1)')
kb.add_fact('Owns(Nono, M1)')

kb.add_rule(premises=['Missile(M1)'], conclusion='Weapon(M1)')

kb.add_rule(premises=['Missile(M1)', 'Owns(Nono, M1)'],
conclusion='Sells(West, M1, Nono)')

kb.add_rule(premises=['Enemy(Nono, America)'],
conclusion='Hostile(Nono)')

kb.add_rule(premises=['American(West)', 'Weapon(M1)',
'Sells(West, M1, Nono)', 'Hostile(Nono)'],
conclusion='Criminal(West)')
```

kb.forward_chain()

```
Adding fact: American(West)
Adding fact: Enemy(Nono, America)
Adding fact: Missile(M1)
Adding fact: Owns(Nono, M1)
Inferred new fact: Hostile(Nono) from ['Enemy(Nono, America)'] => Hostile(Nono)
Inferred new fact: Weapon(M1) from ['Missile(M1)'] => Weapon(M1)
Inferred new fact: Sells(West, M1, Nono) from ['Missile(M1)', 'Owns(Nono, M1)'] => Sells(West, M1, Nono)
Inferred new fact: Criminal(West) from ['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'] => Criminal
(West)

✅ Goal Reached: West is Criminal
Out[6]:  True
```



Algorithm:

function: FOL-FC-ASK(KB, α) return a substitution or false
input: KB, the knowledge base, a set of first order definite clauses α, the query, an atomic sentence
local variable: new, the new sentences inferred on each iteration
repeat until new is empty
new ← { }
for each rule in KB do
$(p_1 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES(rule)
for each θ such that SUBST(θ, $p_1 \wedge \dots \wedge p_n$) = SUBST(θ, $p_1' \wedge \dots \wedge p_n'$)
for some $p_1' \dots p_n'$ in KB
$q' \rightarrow$ SUBST(θ, q)
If q' does not unify with some sentence already in KB or ~~#~~ new then ~~add~~ add q' to new
$\phi \leftarrow$ Unify(q', α)
if ϕ is not fail then return ϕ
add new to KB
return false

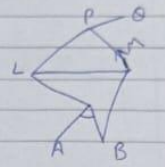
B 13.10

Date / /
Page 25

Lab program - 8

Premises → conclusion

$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$ } rules



A
B } facts

Prove Q

Forward chaining - First order Logic - Solved example

Law states that it is a crime for american to sell weapons to hostile nation. Country Nono, enemy of America has some missiles, and all missiles were sold to it by colonel West, who is American An enemy of America counts as "hostile"

Prove "West is criminal

1. $\forall x, y, z$ American$(x) \wedge$ weapon$(y) \wedge$ sells$(x, y, z) \wedge$ Hostile$(z) \Rightarrow$ Criminal(x)
2. $\forall x$ Missile$(x) \wedge$ Owns$(Nono, x) \Rightarrow$ sells$(West, x, Nono)$
3. $\forall x$ Enemy$(x, America) \Rightarrow$ Hostile(x)
4. $\forall x$ missile$(x) \Rightarrow$ weapon(x)
5. American (West)
6. Enemy (Nono, America)
7. Owns (Nono, M1) and
8. Missile (M1)

lAB 9:

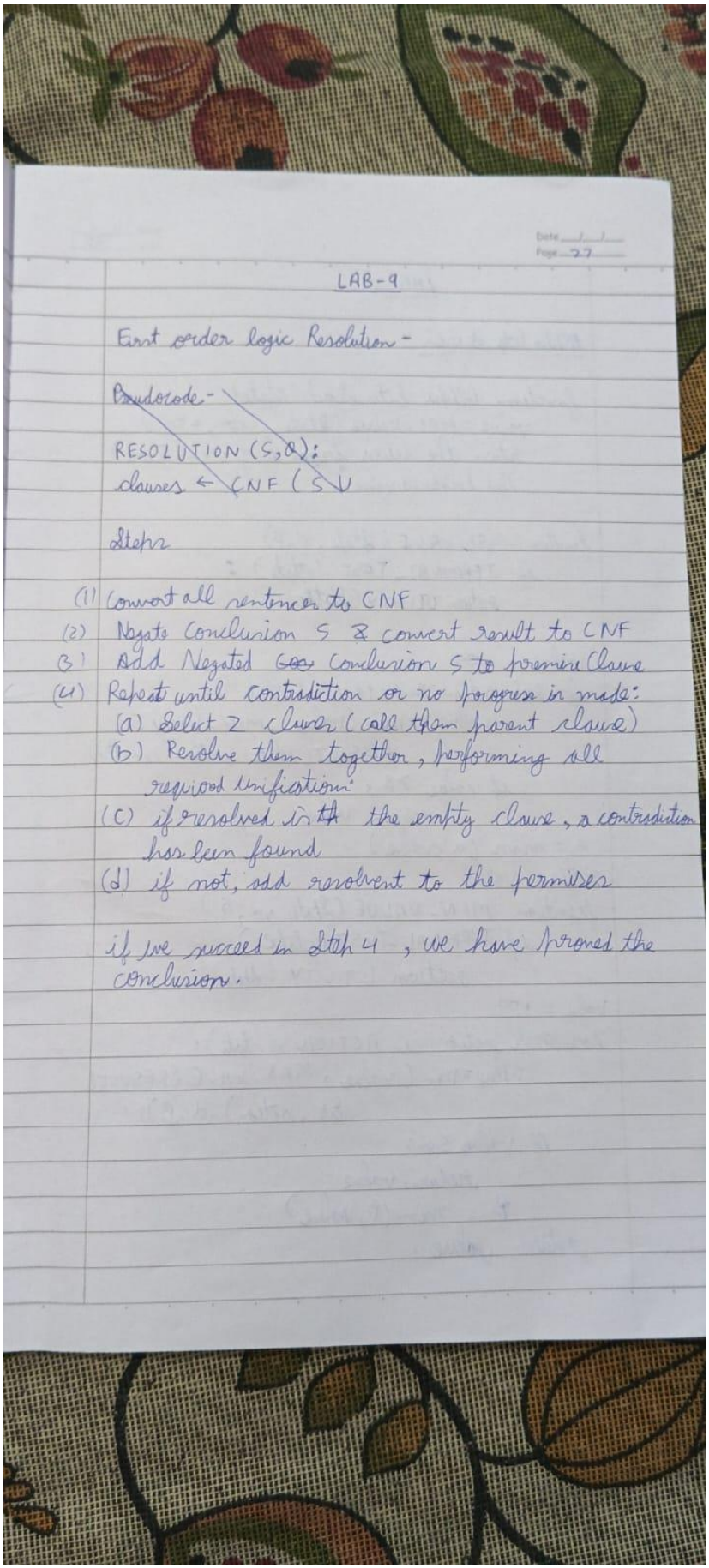
code

```
from itertools import combinations

def get_clauses():
    n = int(input("Enter number of clauses in Knowledge Base: "))
    clauses = []
    for i in range(n):
        clause = input(f"Enter clause {i+1}: ")
        clause_set = set(clause.replace(" ", "").split("v"))
        clauses.append(clause_set)
    return clauses

def resolve(ci, cj):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di == ('~' + dj) or dj == ('~' + di):
                new_clause = (ci - {di}) | (cj - {dj})
                resolvents.append(new_clause)
    return resolvents

def resolution_algorithm(kb, query):
    kb.append(set(['~' + query]))
    derived = []
    clause_id = {frozenset(c): f"C{i+1}" for i, c in enumerate(kb)}

    step = 1
    while True:
```

```
        new = []
        for (ci, cj) in combinations(kb, 2):
            resolvents = resolve(ci, cj)
            for res in resolvents:
                if res not in kb and res not in new:
                    cid_i, cid_j = clause_id[frozenset(ci)],
clause_id[frozenset(cj)]
                    clause_name = f"R{step}"
                    derived.append((clause_name, res, cid_i, cid_j))
                    clause_id[frozenset(res)] = clause_name
                    new.append(res)
                    print(f"[Step {step}] {clause_name} =
Resolve({cid_i}, {cid_j}) → {res or '{}'}")
                    step += 1

                    # If empty clause found → proof complete
                    if res == set():
                        print("\n☑ Query is proved by resolution (empty
clause found).")
                        print("\n--- Proof Tree ---")
                        print_tree(derived, clause_name)
                        return True
        if not new:
            print("\n❌ Query cannot be proved by resolution.")
            return False
        kb.extend(new)

def print_tree(derived, goal):
    tree = {name: (parents, clause) for name, clause, *parents in
[(r[0], r[1], r[2:][0], r[2:][1]) for r in derived]}
```

```
    def show(node, indent=0):
        if node not in tree:
            print(" " * indent + node)
            return
        parents, clause = tree[node]
        print(" " * indent + f"{node}: {set(clause) or '{}'}")
        for p in parents:
            show(p, indent + 4)

    show(goal)

print("=== FOL Resolution Demo with Proof Tree ===")
kb = get_clauses()
query = input("Enter query to prove: ")
resolution_algorithm(kb, query)
```

```
=== FOL Resolution Demo with Proof Tree ===
Enter number of clauses in Knowledge Base: 3
Enter clause 1: P
Enter clause 2: ~P v Q
Enter clause 3: ~Q
Enter query to prove: Q
[Step 1] R1 = Resolve(C1, C2) → {'Q'}
[Step 2] R2 = Resolve(C2, C4) → {'~P'}
[Step 3] R3 = Resolve(C1, R2) → {}
```



Date / /
Page 27

LAB-9

First order logic Resolution -

Pseudocode -

RESOLUTION (S, Q):
clauses ← CNF (S ∨

Steps

(1) Convert all sentences to CNF
(2) Negate Conclusion S & convert result to CNF
(3) Add Negated Conclusion S to premise clause
(4) Repeat until contradiction or no progress is made:
  (a) Select 2 clause (call them parent clause)
  (b) Resolve them together, performing all required unification
  (c) if resolved is the empty clause, a contradiction has been found
  (d) if not, add resolvent to the premises

if we succeed in Step 4, we have proved the conclusion.

LAB 10:

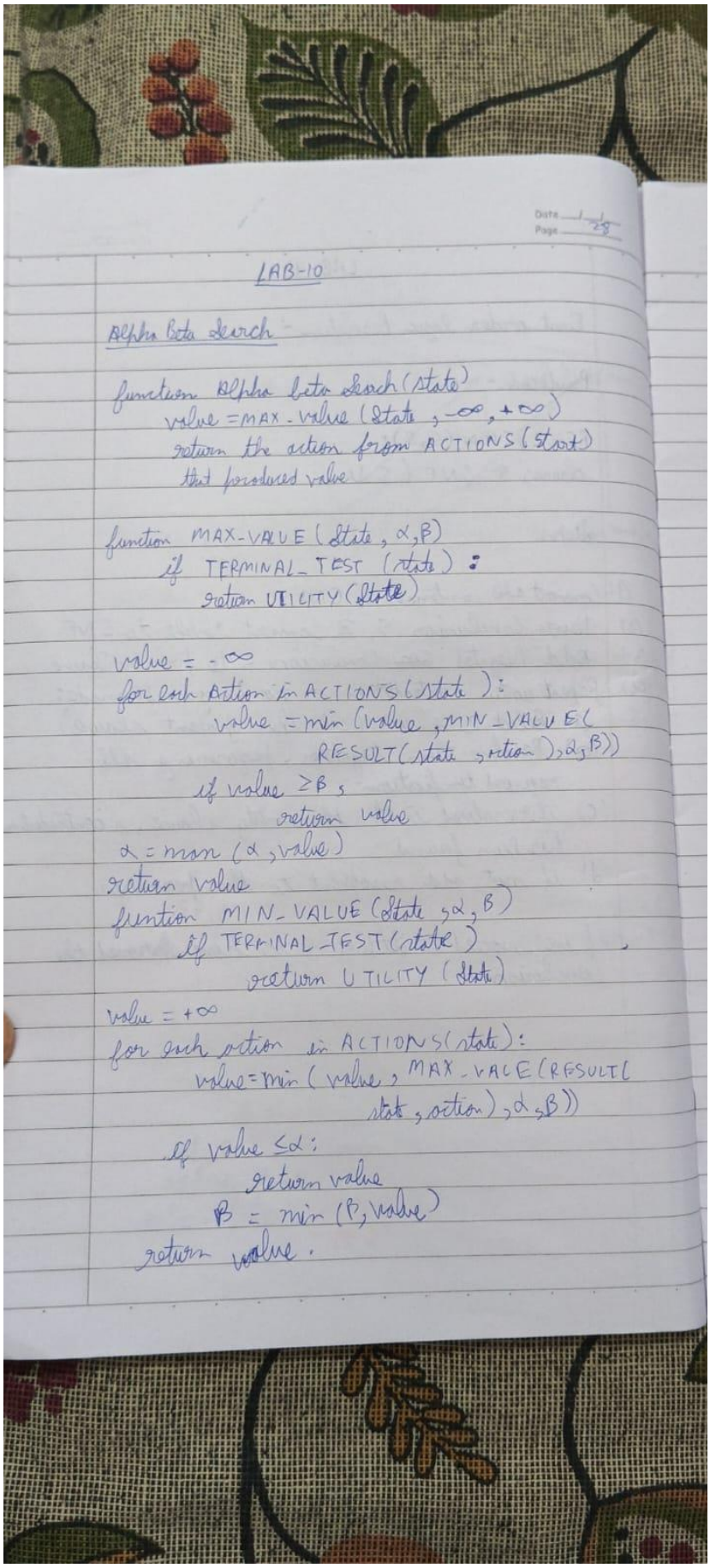
Code

```
def alpha_beta(node, depth, alpha, beta, isMax, values, maxDepth):

    if depth == maxDepth:

        first_leaf_index = (2 ** maxDepth) - 1
        leaf_idx = node - first_leaf_index
        return values[leaf_idx]

    if isMax:
        best = float('-inf')
        for i in range(2):
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, False,
values, maxDepth)
            best = max(best, val)
            alpha = max(alpha, best)
            print(f"MAX: Depth={depth}, Node={node},
Alpha={alpha}, Beta={beta}")

            if beta <= alpha:
                print(f" ⃠ PRUNED at MAX node {node} (α ≥ β)")
                break
        return best
    else:
```

```
        best = float('inf')
        for i in range(2):
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, True,
values, maxDepth)
            best = min(best, val)
            beta = min(beta, best)
            print(f"MIN: Depth={depth}, Node={node},
Alpha={alpha}, Beta={beta}")

            if beta <= alpha:
                print(f" 🚫 PRUNED at MIN node {node} (α ≥ β)")
                break
        return best


print("🤔 ALPHA–BETA PRUNING — Interactive Demo")
print("==========================================\n
")


maxDepth = int(input("Enter maximum depth of the game tree: "))


num_leaves = 2 ** maxDepth
print(f"For depth {maxDepth}, the tree will have {num_leaves}
leaf nodes.\n")


values = []
print("Enter the leaf node values from LEFT to RIGHT:")
```

```
for i in range(num_leaves):
    val = int(input(f"Value of leaf {i + 1}: "))
    values.append(val)

print("\n🔁 Running Alpha–Beta pruning...\n")
result = alpha_beta(0, 0, float('-inf'), float('inf'), True, values,
maxDepth)


print(f"Value of the root node (best achievable for MAX):
{result}")
```

```
🧠 ALPHA-BETA PRUNING — Interactive Demo
========================================

Enter maximum depth of the game tree: 3
For depth 3, the tree will have 8 leaf nodes.

Enter the leaf node values from LEFT to RIGHT:
Value of leaf 1: 1
Value of leaf 2: 2
Value of leaf 3: 3
Value of leaf 4: 4
Value of leaf 5: 5
Value of leaf 6: 6
Value of leaf 7: 8
Value of leaf 8: 9

🔁 Running Alpha-Beta pruning...

MAX: Depth=2, Node=3, Alpha=1, Beta=inf
MAX: Depth=2, Node=3, Alpha=2, Beta=inf
MIN: Depth=1, Node=1, Alpha=-inf, Beta=2
MAX: Depth=2, Node=4, Alpha=3, Beta=2
🚫 PRUNED at MAX node 4 (α ≥ β)
MIN: Depth=1, Node=1, Alpha=-inf, Beta=2
MAX: Depth=0, Node=0, Alpha=2, Beta=inf
MAX: Depth=2, Node=5, Alpha=5, Beta=inf
MAX: Depth=2, Node=5, Alpha=6, Beta=inf
MIN: Depth=1, Node=2, Alpha=2, Beta=6
MAX: Depth=2, Node=6, Alpha=8, Beta=6
🚫 PRUNED at MAX node 6 (α ≥ β)
MIN: Depth=1, Node=2, Alpha=2, Beta=6
MAX: Depth=0, Node=0, Alpha=6, Beta=inf

Value of the root node (best achievable for MAX): 6
```



Date / /
Page 25

LAB-10

Alpha Beta Search

```
function Alpha beta search (state)
    value = MAX-value (State, -∞, +∞)
    return the action from ACTIONS (State)
     that produced value

function MAX-VALUE (State, α, β)
    if TERMINAL_TEST (state):
        return UTILITY (State)

value = -∞
for each action in ACTIONS (state):
        value = min (value, MIN_VALUE (
                RESULT (state, action), α, β))
    if value ≥ β,
        return value
α = max (α, value)
return value
function MIN_VALUE (State, α, β)
    if TERMINAL_TEST (state)
        return UTILITY (State)
value = +∞
for each action in ACTIONS (state):
        value = min (value, MAX_VALUE (RESULT (
                state, action), α, β))
    if value ≤ α:
        return value
    β = min (β, value)
return value.
```