VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sareddy Poojya Sree (1BM23CS303)

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 **Sareddy Poojya Sree (1BM23CS303),** 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.

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

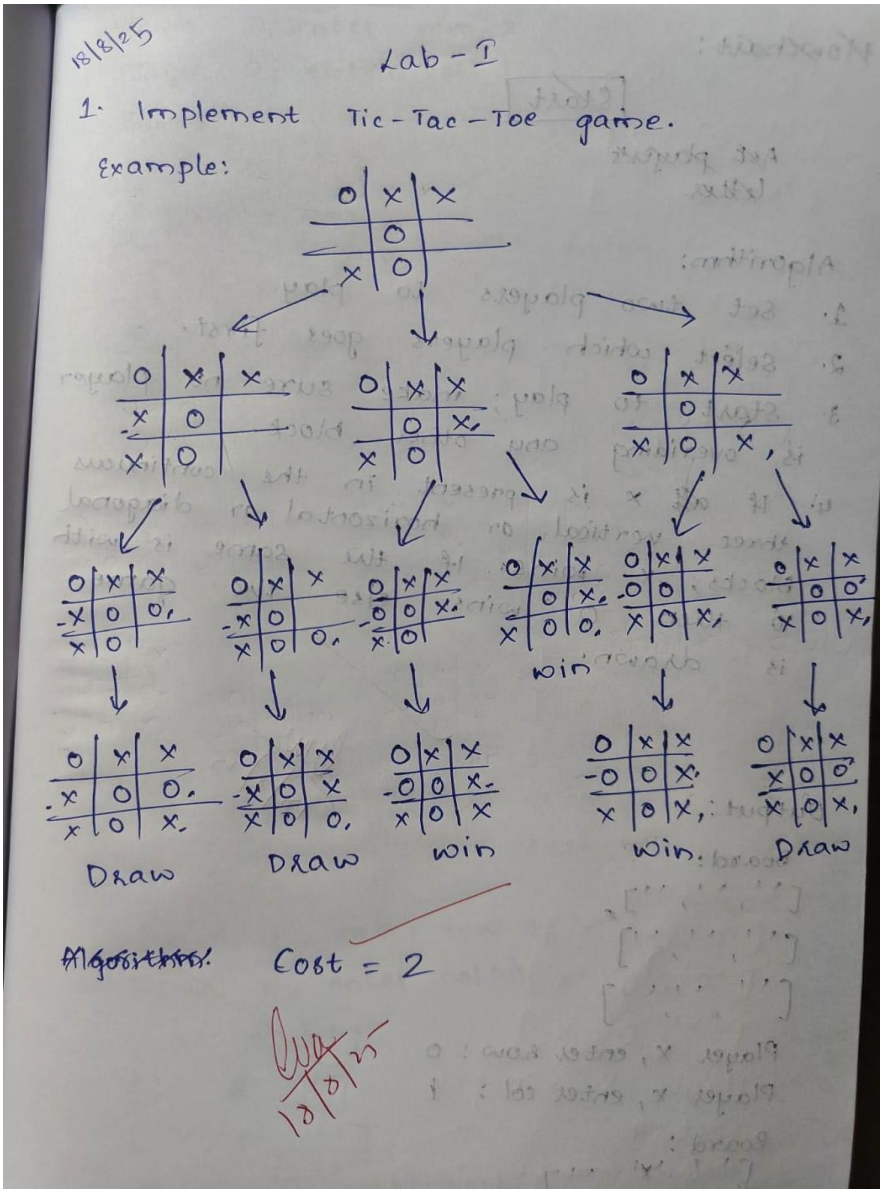
Index

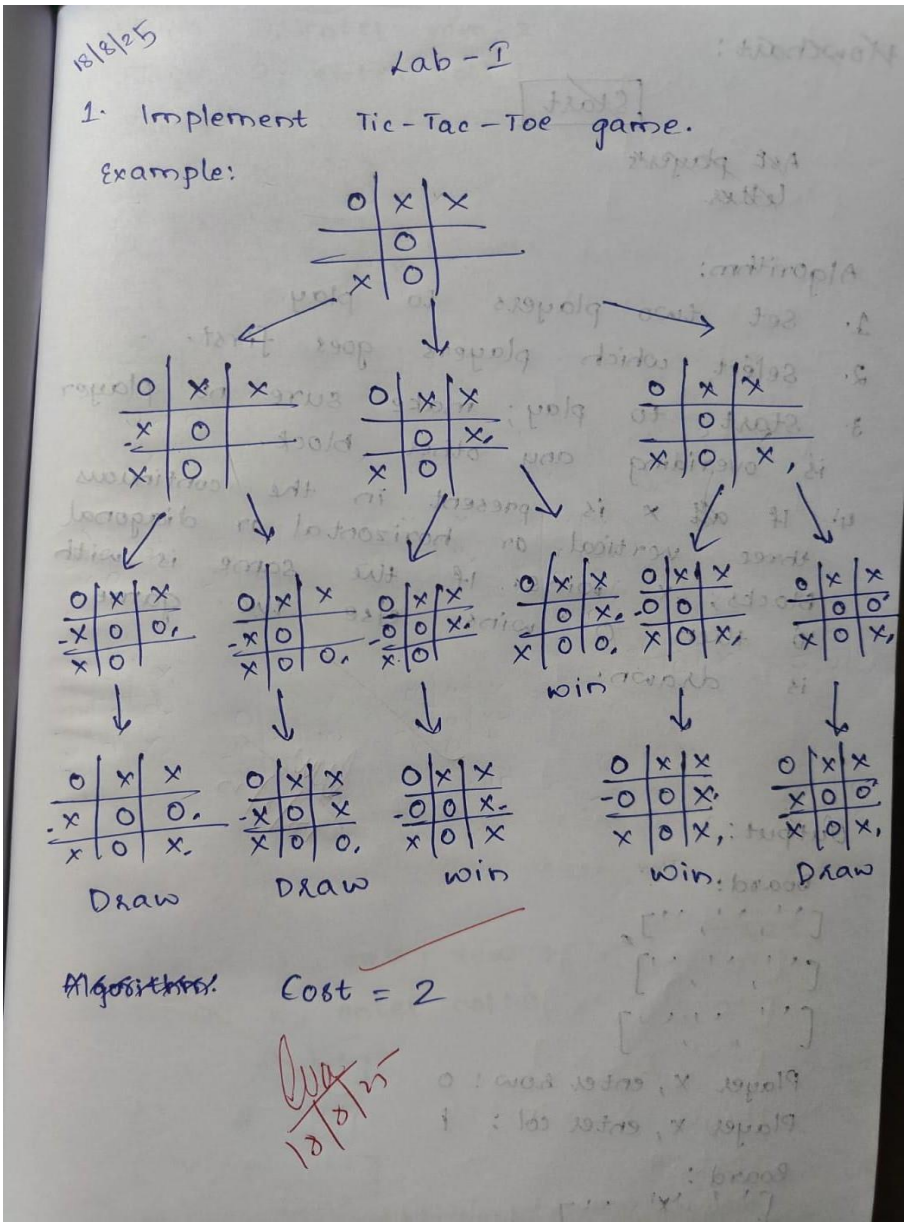
Github Link:
poojya100/1BM23CS303_SAREDDY_POOJYA_SREE-AI_LAB

Week – 1

Program 1 Implement Tic –Tac –Toe Game

Algorithm:




18/8/25
Lab - I
1. Implement Tic-Tac-Toe game.
Example:
win
Draw
Draw
win
win.
Draw
Cost = 2

Output:

```
Player X, enter row (0-2): 2
Player X, enter col (0-2): 0

Current Board:
['O', 'X', 'X']
[' ', ' ', ' ']
['X', 'O', ' ']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

Current Board:
['O', 'X', 'X']
[' ', 'O', ' ']
['X', 'O', ' ']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 2

Current Board:
['O', 'X', 'X']
[' ', 'O', 'X']
['X', 'O', ' ']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 0

Current Board:
['O', 'X', 'X']
['O', 'O', 'X']
['X', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 2

Current Board:
['O', 'X', 'X']
['O', 'O', 'X']
['X', 'O', 'X']

Player X wins!
Total moves (cost): 9
```

```
[' ', ' ', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 1

Current Board:
['O', 'X', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 0

Current Board:
['O', 'X', 'X']
[' ', ' ', ' ']
['X', 'O', ' ']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

Current Board:
['O', 'X', 'X']
[' ', 'O', ' ']
['X', 'O', ' ']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 0

Current Board:
['O', 'X', 'X']
['X', 'O', ' ']
['X', 'O', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 2

Current Board:
['O', 'X', 'X']
['X', 'O', ' ']
['X', 'O', 'O']

Player O wins!
Total moves (cost): 8
```

```
Player O, enter row (0-2): 2
Player O, enter col (0-2): 1

Current Board:
['O', 'X', 'X']
[' ', ' ', ' ']
[' ', 'O', ' ']

Player X, enter row (0-2): 2
Player X, enter col (0-2): 0

Current Board:
['O', 'X', 'X']
[' ', ' ', ' ']
['X', 'O', ' ']

Player O, enter row (0-2): 1
Player O, enter col (0-2): 1

Current Board:
['O', 'X', 'X']
[' ', 'O', ' ']
['X', 'O', ' ']

Player X, enter row (0-2): 1
Player X, enter col (0-2): 0

Current Board:
['O', 'X', 'X']
['X', 'O', ' ']
['X', 'O', ' ']

Player O, enter row (0-2): 2
Player O, enter col (0-2): 2

Current Board:
['O', 'X', 'X']
['X', 'O', ' ']
['X', 'O', 'O']

Player O wins!
Total moves (cost): 8
```

Code:

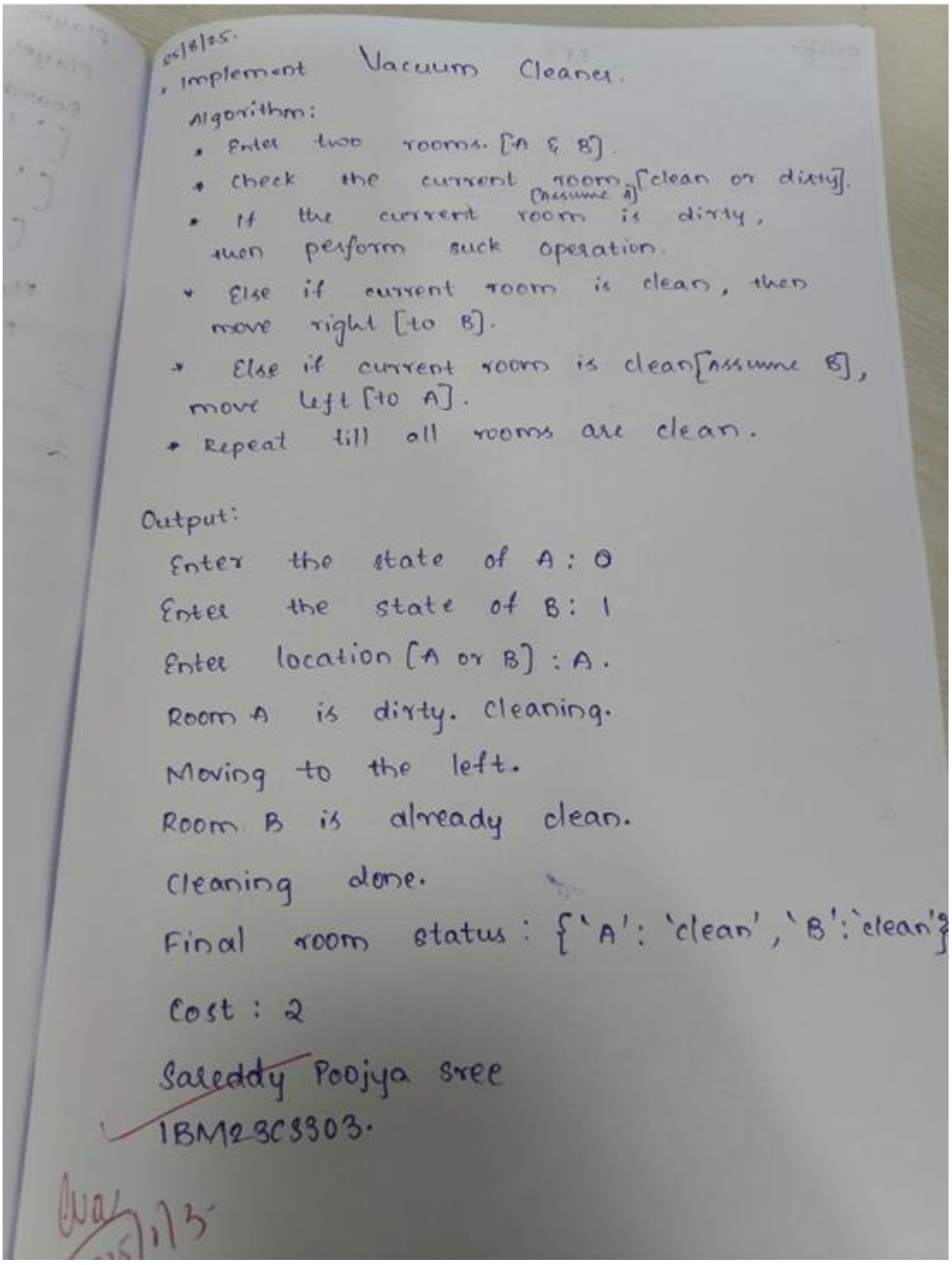
```
def print_board(board):
    print("\nCurrent Board:")
    for row in board:
        print(row)
    print()



def check_winner(board, player):
    # Check rows
    for row in board:
        if all(cell == player for cell in row):
            return True
```

```
    # Check columns
    for col in range(3):
        if all(board[row][col] == player for row in range(3)):
            return True

    # Check diagonals
    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 tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    players = ['X', 'O']
    move_count = 0

    while True:
        current_player = players[move_count % 2]
        print(f"Player {current_player}, enter row (0-2): ", end="")
        row = int(input())
        print(f"Player {current_player}, enter col (0-2): ", end="")
        col = int(input())

        # If cell is empty
        if board[row][col] == ' ':
            board[row][col] = current_player
            move_count += 1
            print_board(board)
```

```
            if check_winner(board, current_player):
                print(f"Player {current_player} wins!")
                print(f"Total moves (cost): {move_count}")
                break

            if move_count == 9:  # Board full
                print("It's a draw!")
                break
        else:
            print("Cell already taken! Try again.")


# Run the game
tic_tac_toe()
```

b. Implement vacuum cleaner:

Algorithm:



05/8/25.

Implement Vacuum Cleaner.

Algorithm:

- Enter two rooms. [A & B].
- Check the current room [assume A] [clean or dirty].
- If the current room is dirty, then perform suck operation.
- Else if current room is clean, then move right [to B].
- Else if current room is clean [assume B], move Left [to A].
- Repeat till all rooms are clean.

Output:

Enter the state of A : 0

Enter the state of B : 1

Enter location [A or B] : A.

Room A is dirty. Cleaning.

Moving to the left.

Room B is already clean.

Cleaning done.

Final room status : {'A': 'clean', 'B': 'clean'}

Cost : 2

Saleddy Poojya Sree

1BM23CS303.

Output:

```
Enter status for Room A (0 = clean, 1 = dirty): 1
Enter status for Room B (0 = clean, 1 = dirty): 1
Enter starting room (A or B): A

Initial Room Statuses:
Room A: Dirty
Room B: Dirty

Vacuum starting in Room A...

Room A is dirty. Performing SUCK action.
Moving to Room B.
Room B is dirty. Performing SUCK action.

Final Room Statuses:
Room A: Clean
Room B: Clean

cost : 3
Sareddy Poojya Sree
1BM23CS303
```

```
Enter status for Room A (0 = clean, 1 = dirty): 1
Enter status for Room B (0 = clean, 1 = dirty): 0
Enter starting room (A or B): A

Initial Room Statuses:
Room A: Dirty
Room B: Clean

Vacuum starting in Room A...

Room A is dirty. Performing SUCK action.
Moving to Room B.
Room B is already clean.

Final Room Statuses:
Room A: Clean
Room B: Clean

cost : 2
Sareddy Poojya Sree
1BM23CS303
```

Code:

```
def vacuum_cleaner():
    # Input the state of rooms A and B
    state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    cost = 0
    rooms = {'A': state_A, 'B': state_B}
```

```
    # Function to clean a room if dirty
    def clean_room(room):
        nonlocal cost
        if rooms[room] == 1:
            print(f"Cleaned {room}.")
            rooms[room] = 0
            cost += 1
        else:
            print(f"{room} is clean.")

    # Start cleaning based on location
    if location == 'A':
        clean_room('A')
10
        print("Moving vacuum right")
        clean_room('B')
    elif location == 'B':
        clean_room('B')
        print("Moving vacuum left")
        clean_room('A')
    else:
        print("Invalid starting location!")

    print(f"Cost: {cost}")
    print(rooms)

if __name___== "__main__":
    vacuum_cleaner()
```

Implement BFS

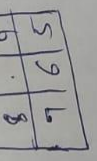
05/8/25. BFS

Algorithm:

* Check the initial state with goal state, if matching break, else move the empty tile to all the possible positions [up, right, left, down].
* Now check all the new states with goal state.
* If any of the state match goal state, break, else move the empty tile to all the possible positions and gain new states.
* Check all the new states with goal state and repeat the process until the goal state is achieved.

~~Output:~~

Using BFS solve 8 puzzle without heuristic.



DFS.

Algorithm:

1. Start with the initial puzzle state.
2. Put this state into a stack.
3. Keep a set of visited states to avoid repeating the same board.
4. While the stack is not empty:
   → Take the top state from the stack.
   → If it matches the goal state, stop.

al state,
the
e positions

with

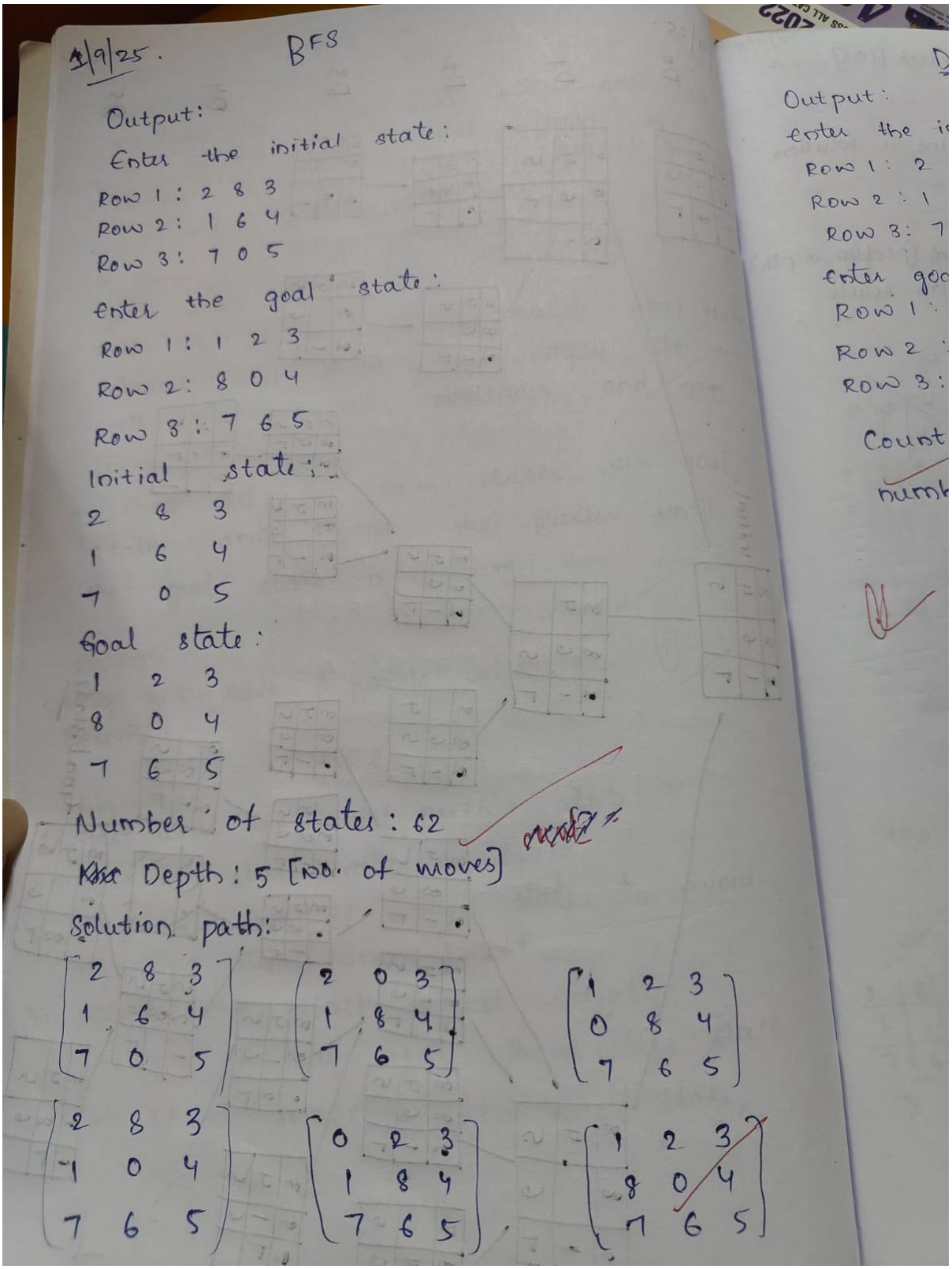
al state,
le to
gain

goal
til

d

ct.

1/9/25.


3/9/25. BFS
Output:
Enter the initial state:
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5
Enter the goal state:
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5
Initial state:
2 8 3
1 6 4
7 0 5
Goal state:
1 2 3
8 0 4
7 6 5
Number of states: 62
Depth: 5 [no. of moves]
Solution path:
Output:
Enter the
Row 1: 2
Row 2: 1
Row 3: 7
Enter goa
Row 1:
Row 2
Row 3:
Count
numb

Output:

```
Enter the INITIAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 2 8 3
Row 2 (3 numbers space-separated): 1 6 4
Row 3 (3 numbers space-separated): 7 0 5

Enter the GOAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 1 2 3
Row 2 (3 numbers space-separated): 8 0 4
Row 3 (3 numbers space-separated): 7 6 5

Initial State:
2 8 3
1 6 4
7 0 5

Goal State:
1 2 3
8 0 4
7 6 5


 Goal reached!
 Number of states explored: 62
  Number of moves: 5

Solution path:
2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5
```

```
2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5

Sareddy Poojya Sree
1BM23CS303
```

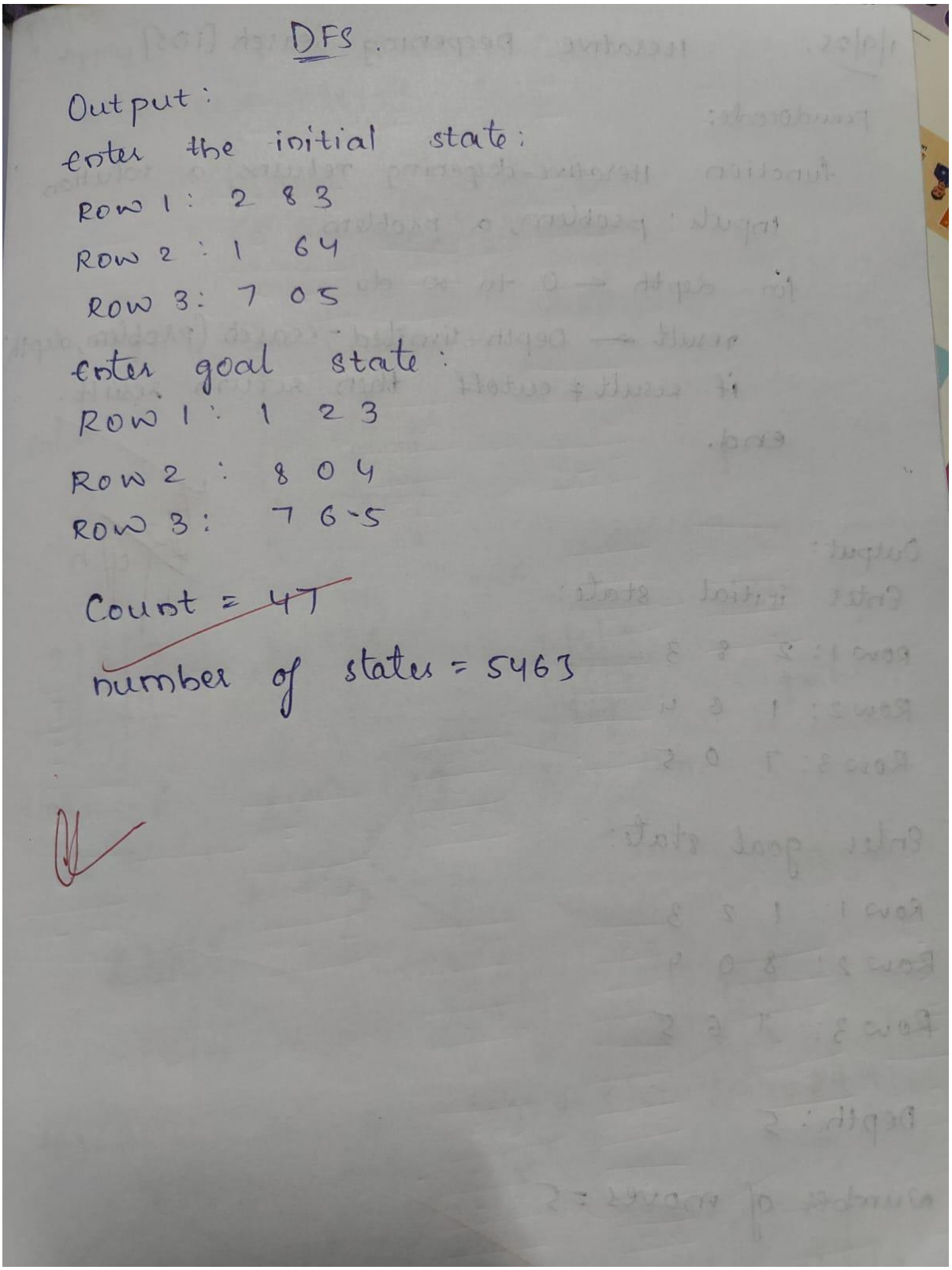
Code:

```
from collections import deque

# Function to display puzzle state
def print_state(state):
    for i in range(0, 9, 3):
        print(" ".join(state[i:i+3]))
    print()

# Generate neighbors
def get_neighbors(state):
    neighbors = []
```

```
    index = state.index("0")  # blank position
    row, col = divmod(index, 3)

    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right

    for dr, dc in moves:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            state_list = list(state)
            # swap blank
            state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
            neighbors.append("".join(state_list))
    return neighbors

# BFS solver
def bfs(start, goal):
    visited = set()
    queue = deque([(start, [])])  # (state, path)
    visited.add(start)
    state_count = 1

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

        if state == goal:
            print("Number of states explored:", state_count)
            print("  Number of moves:", len(path))
            print("\nSolution path:")
            for s in path + [state]:
                print_state(s)
            return path

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append((neighbor, path + [state]))
                state_count += 1

    print("No solution found.")
    return None

# ------------------------------
# MAIN PROGRAM
# ------------------------------
print("Enter the INITIAL 8-puzzle board configuration (use 0 for blank):")
```

```
initial_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    initial_board.extend(row)

print("\nEnter the GOAL 8-puzzle board configuration (use 0 for blank):")
goal_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    goal_board.extend(row)

start_state = "".join(initial_board)
goal_state = "".join(goal_board)

print("\nInitial State:")
print_state(start_state)

print("Goal State:")
print_state(goal_state)

bfs(start_state, goal_state)
```

Implement DFS



Output:

```
Enter the INITIAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 2 8 3
Row 2 (3 numbers space-separated): 1 6 4
Row 3 (3 numbers space-separated): 7 0 5

Enter the GOAL 8-puzzle board configuration (use 0 for blank):
Row 1 (3 numbers space-separated): 1 2 3
Row 2 (3 numbers space-separated): 8 0 4
Row 3 (3 numbers space-separated): 7 6 5

Initial State:
2 8 3
1 6 4
7 0 5

Goal State:
1 2 3
8 0 4
7 6 5

count: 47

Solution path:
2 8 3
1 6 4
7 0 5
```

```
1 2 3
8 6 4
0 7 5

1 2 3
8 6 4
7 0 5

1 2 3
8 0 4
7 6 5

Sareddy Poojya Sree
1BM23CS303
```

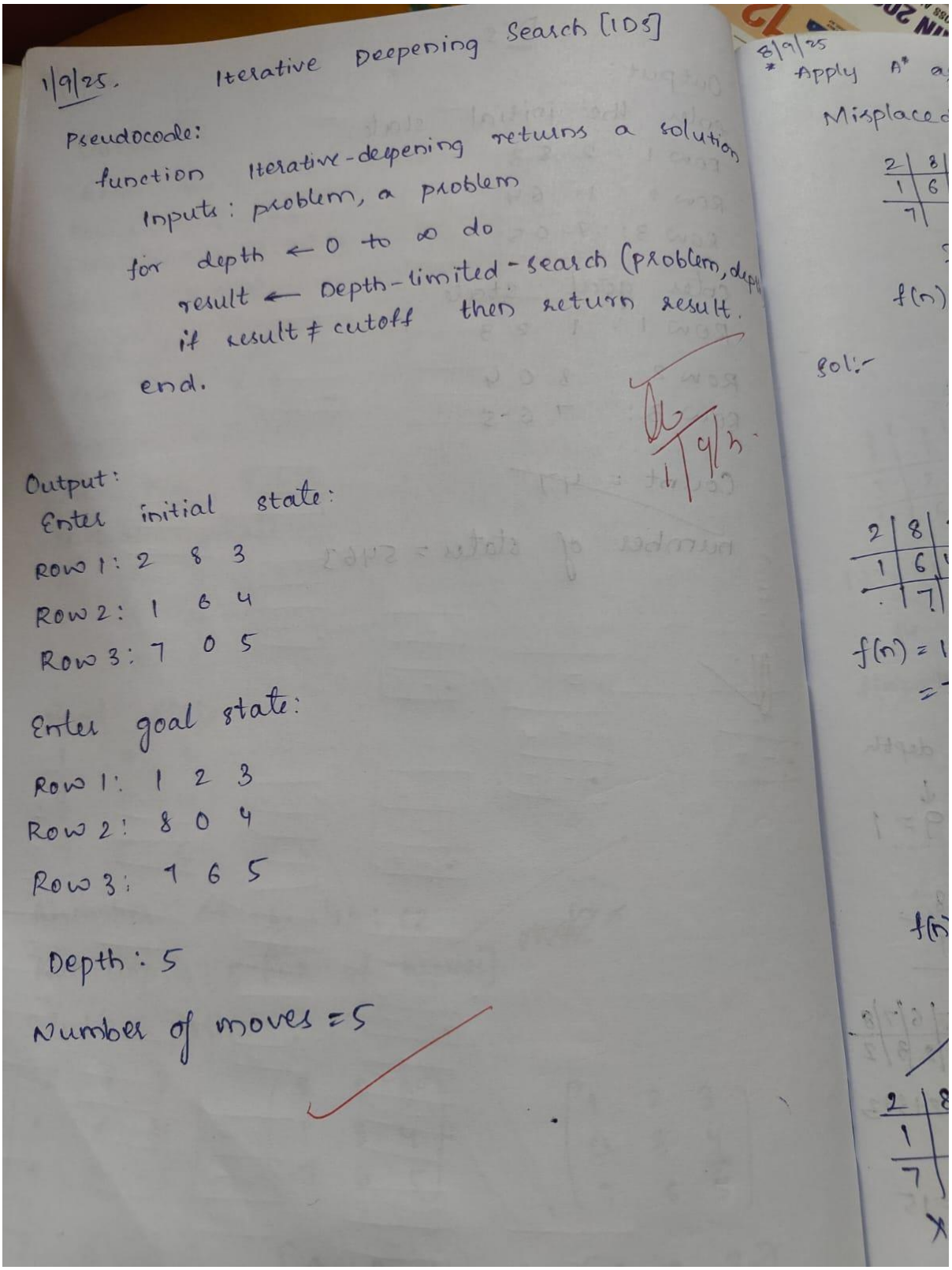
Code:

```
from collections import deque
```

```python
# Function to display puzzle state
def print_state(state):
    for i in range(0, 9, 3):
        print(" ".join(state[i:i+3]))
    print()

# Generate neighbors
def get_neighbors(state):
    neighbors = []
    index = state.index("0")  # blank position
    row, col = divmod(index, 3)

    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # up, down, left, right

    for dr, dc in moves:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            state_list = list(state)
            # swap blank
            state_list[index], state_list[new_index] = state_list[new_index], state_list[index]
            neighbors.append("".join(state_list))
    return neighbors

# DFS solver (recursive)
def dfs(start, goal, visited=None, path=None, state_count=[0], max_depth=50):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    visited.add(start)
    state_count[0] += 1

    if start == goal:

        #print("Number of states explored:", state_count[0])
        print("count:", len(path))
        print("\nSolution path:")
        for s in path + [start]:
            print_state(s)
        return path

    if len(path) >= max_depth:  # prevent infinite recursion
        return None

    for neighbor in get_neighbors(start):
        if neighbor not in visited:
            result = dfs(neighbor, goal, visited, path + [start], state_count, max_depth)
```

```
            if result is not None:
                return result

    return None

# ------------------------------
# MAIN PROGRAM
# ------------------------------
print("Enter the INITIAL 8-puzzle board configuration (use 0 for blank):")
initial_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    initial_board.extend(row)

print("\nEnter the GOAL 8-puzzle board configuration (use 0 for blank):")
goal_board = []
for i in range(3):
    row = input(f"Row {i+1} (3 numbers space-separated): ").split()
    goal_board.extend(row)

start_state = "".join(initial_board)
goal_state = "".join(goal_board)

print("\nInitial State:")
print_state(start_state)

print("Goal State:")
print_state(goal_state)

dfs(start_state, goal_state)

print("Sareddy Poojya Sree")
print("1BM23CS303")
```

Implement Iterative Deepening DFS



1/9/25. Iterative Deepening Search [IDS]

Pseudocode:

function Iterative-deepening returns a solution
inputs: problem, a problem
for depth ← 0 to ∞ do
result ← Depth-limited-search (problem, dep
if result ≠ cutoff then return result.
end.

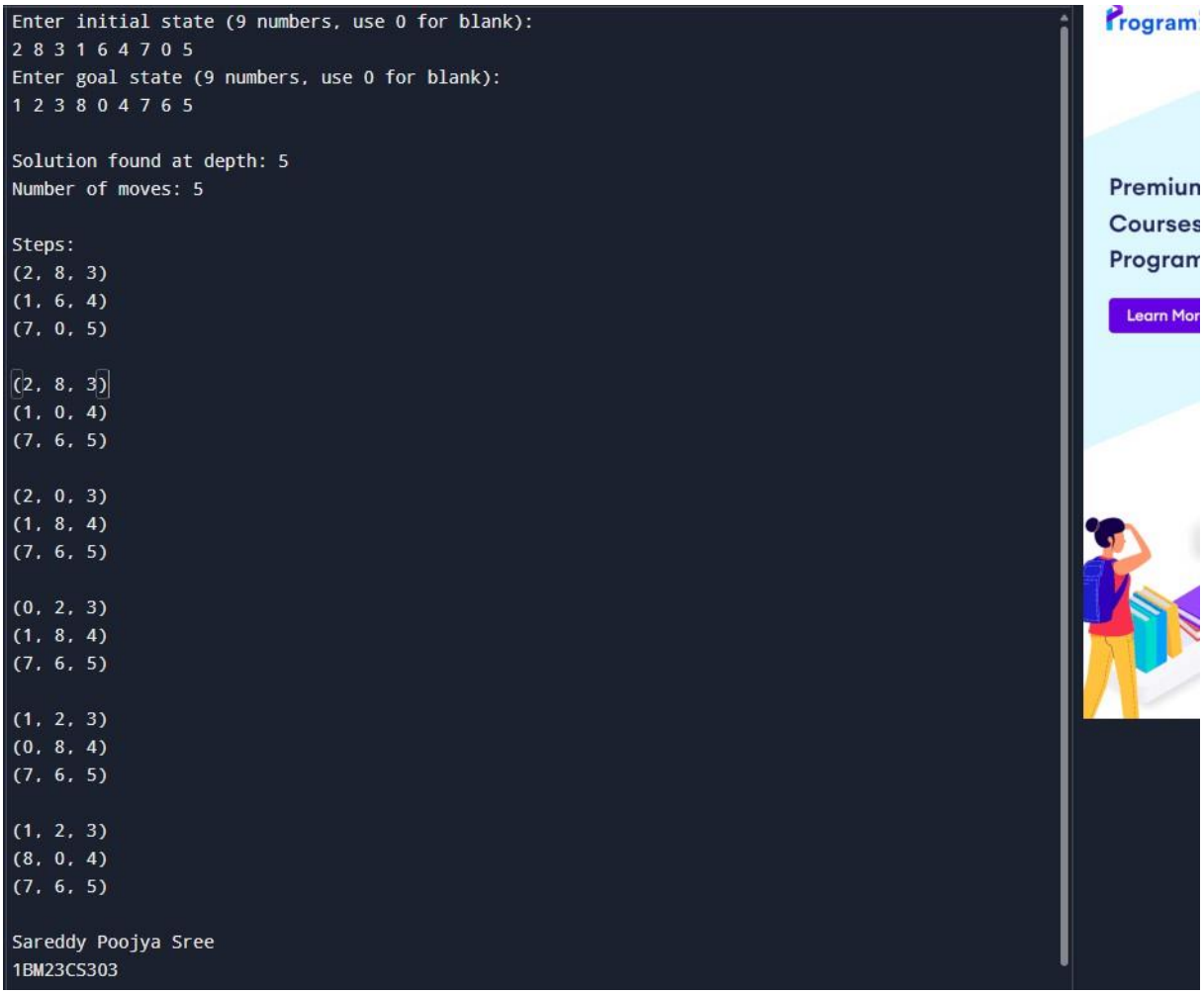
Output:
Enter initial state:
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5

Enter goal state:
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5

Depth: 5
Number of moves = 5

8/9/25
* Apply A*
Misplace
f(n)
Sol:-
f(n) =

Output:

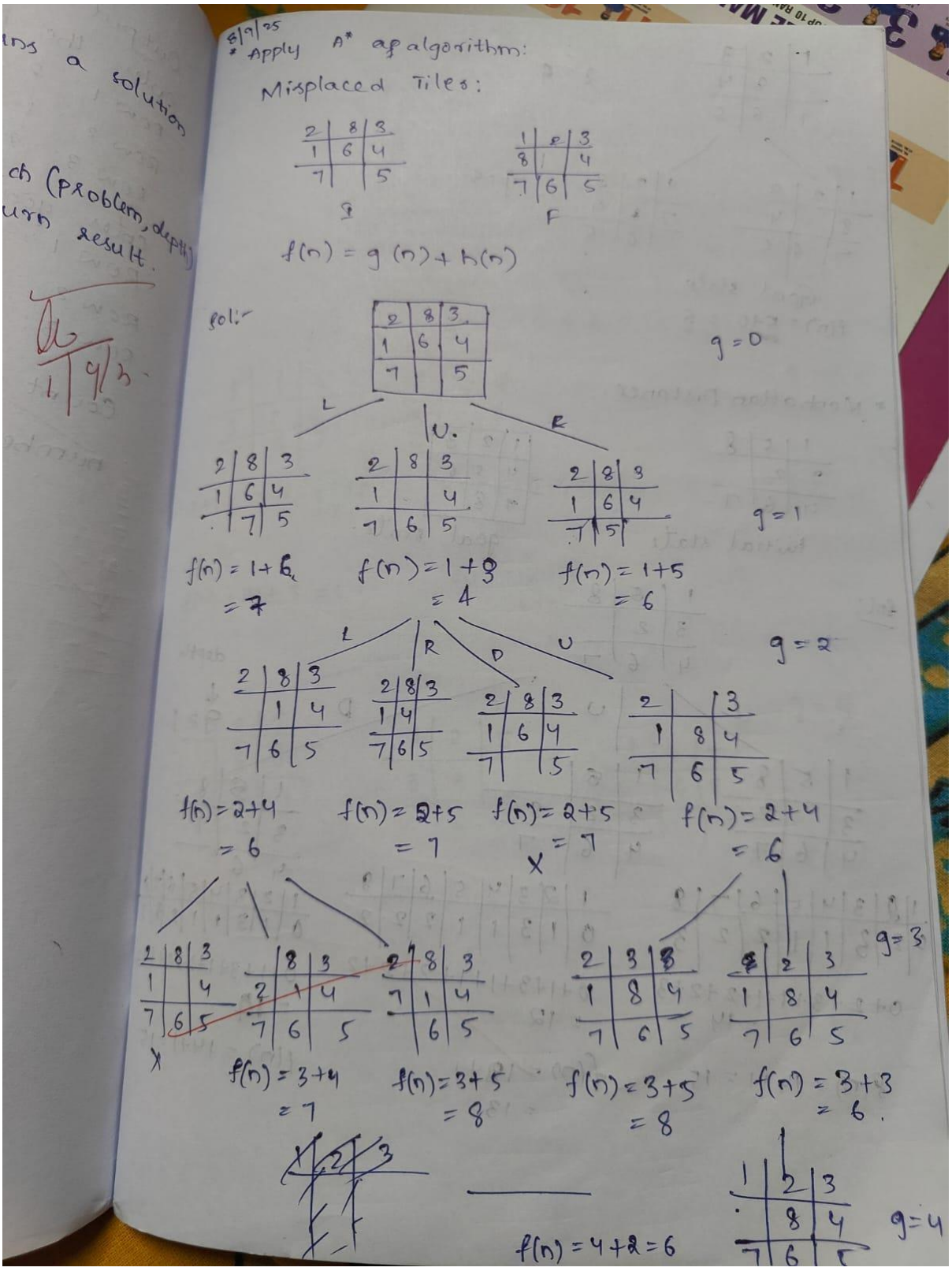


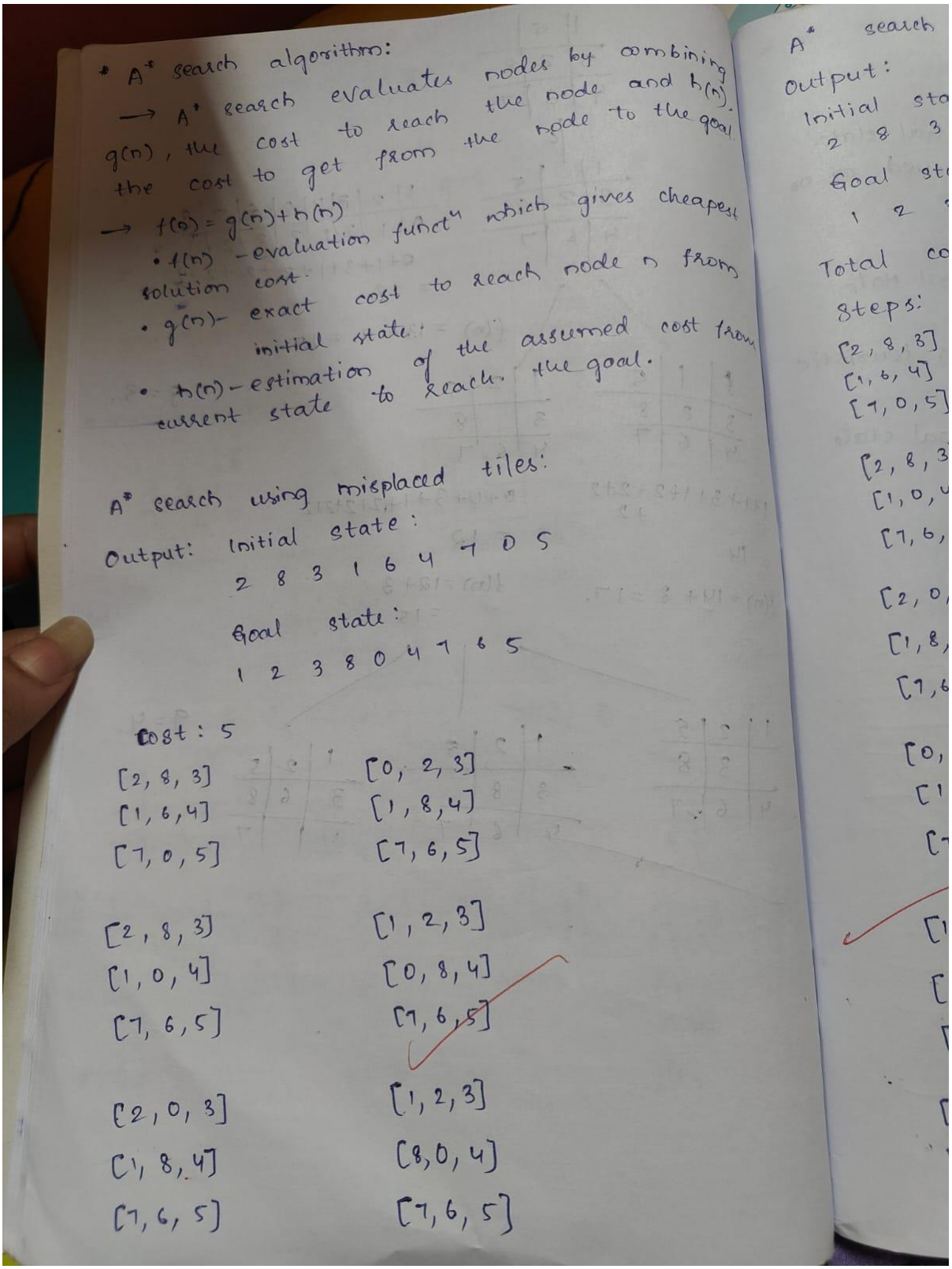
Code:

```
def get_neighbors(state):
    neighbors = []
    blank = state.index(0)
    x, y = divmod(blank, 3)

    moves = [(-1,0), (1,0), (0,-1), (0,1)] # up, down, left, right
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_blank = nx*3 + ny
            new_state = list(state)
            new_state[blank], new_state[new_blank] = new_state[new_blank], new_state[blank]
            neighbors.append(tuple(new_state))
    return neighbors

# Depth Limited Search (recursive)
def depth_limited_search(state, goal, limit, path, visited):
    if state == goal:
        return path
```

```
    if limit == 0:
        return None

    visited.add(state)
    for neighbor in get_neighbors(state):
        if neighbor not in visited:
            result = depth_limited_search(neighbor, goal, limit - 1, path + [neighbor], visited)
            if result is not None:
                return result
    return None

# Iterative Deepening Search
def iterative_deepening_search(initial, goal):
    depth = 0
    while True:
        visited = set()
        result = depth_limited_search(initial, goal, depth, [initial], visited)
        if result is not None:
            return result, depth
        depth += 1

# Print puzzle in 3x3 grid
def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Main
if __name___== "__main__":
    print("Enter initial state (9 numbers, use 0 for blank):")
    initial = tuple(map(int, input().split()))
    print("Enter goal state (9 numbers, use 0 for blank):")
    goal = tuple(map(int, input().split()))

    path, depth = iterative_deepening_search(initial, goal)

    print("\nSolution found at depth:", depth)
    print("Number of moves:", len(path)-1)
    print("\nSteps:")
    for step in path:
        print_state(step)
print("Sareddy Poojya Sree\n1BM23CS303")
```

Implement a* Search using misplaced tiles





* A* search 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 the goal.

→ f(n) = g(n) + h(n)

- f(n) - evaluation funct^n which gives cheapest solution cost.
- g(n) - exact cost to reach node n from initial state.
- h(n) - estimation of the assumed cost from current state to reach the goal.

A* search using misplaced tiles:

Output: Initial state:

2 8 3 1 6 4 7 0 5

Goal state:

1 2 3 8 0 4 7 6 5

cost: 5

[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

A* search

Output:

Initial sta
2 8 3

Goal st
1 2

Total co

steps:

[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3
[1, 0,
[7, 6,

[2, 0,
[1, 8,
[7, 6

[0,
[1
[7

[1

[

[

[

Output:

```
Enter initial state (9 numbers, use 0 for blank):
2 8 3 1 6 4 7 0 5
Enter goal state (9 numbers, use 0 for blank):
1 2 3 8 0 4 7 6 5

Solution found in 5 moves.
Total cost: 5

Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Sareddy Poojya Sree
1BM23CS303
```

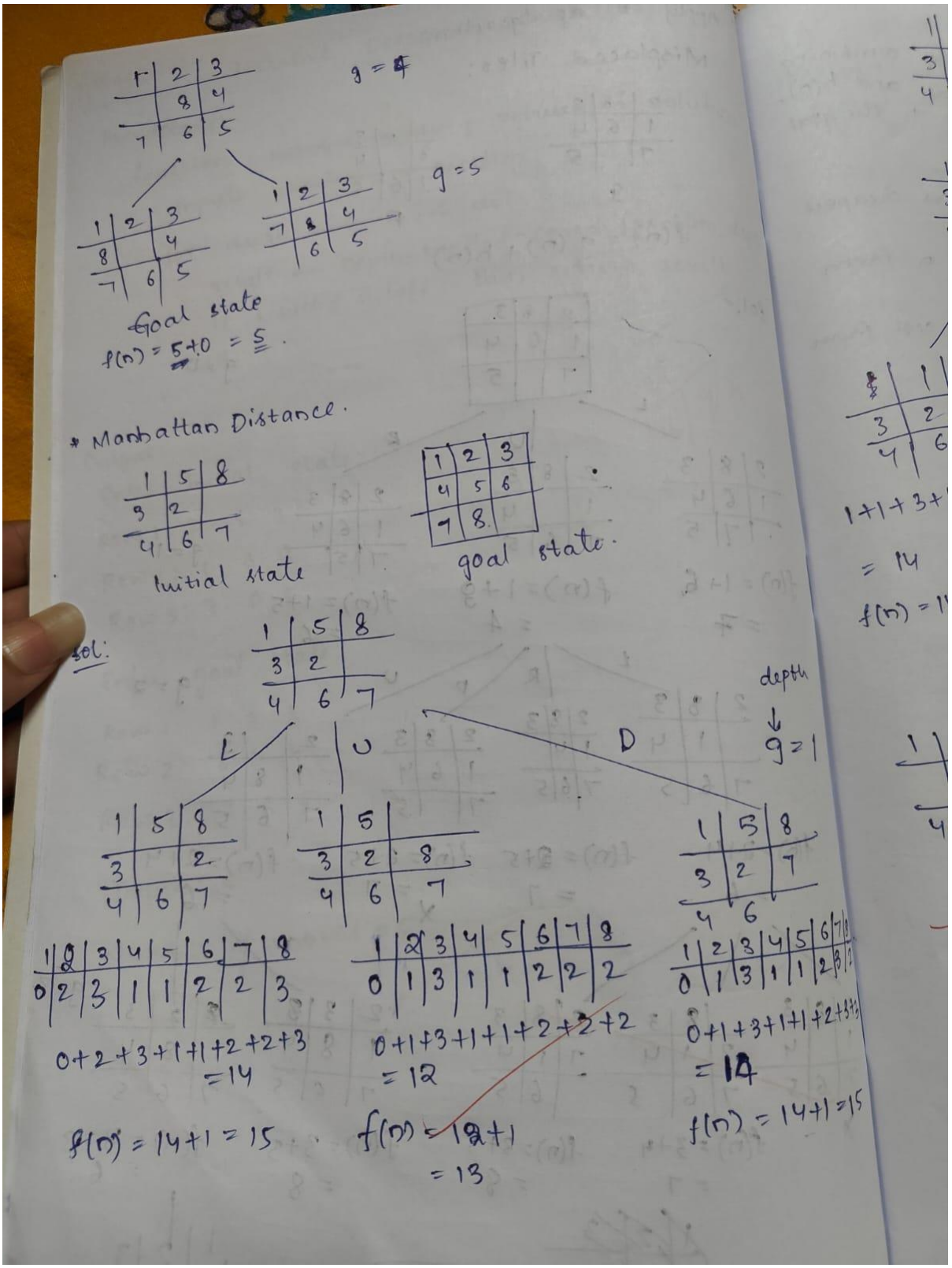
Code:

```
import heapq

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

# Generate possible next states
def get_neighbors(state):
    neighbors = []
```

```
    blank = state.index(0)
    x, y = divmod(blank, 3)

    moves = [(-1,0), (1,0), (0,-1), (0,1)] # up, down, left, right
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_blank = nx*3 + ny
            new_state = list(state)
            new_state[blank], new_state[new_blank] = new_state[new_blank],
new_state[blank]
            neighbors.append(tuple(new_state))
    return neighbors

# A* Search
def a_star(initial, goal):
    frontier = []
    heapq.heappush(frontier, (misplaced_tiles(initial, goal), 0, initial, [initial]))
    visited = set()

    while frontier:
        f, g, state, path = heapq.heappop(frontier)

        if state == goal:
            return path, g  # return path and cost

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

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + misplaced_tiles(neighbor, goal)
                heapq.heappush(frontier, (new_f, new_g, neighbor, path + [neighbor]))
    return None, None

# Print puzzle in 3x3 grid
def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# Main
if __name___== "__main__":
    print("Enter initial state (9 numbers, use 0 for blank):")
    initial = tuple(map(int, input().split()))
    print("Enter goal state (9 numbers, use 0 for blank):")
    goal = tuple(map(int, input().split()))
```

```
    path, cost = a_star(initial, goal)

    if path:
        print("\nSolution found in", len(path)-1, "moves.")
        print("Total cost:", cost)
        print("\nSteps:")
        for step in path:
            print_state(step)
    else:
        print("No solution found!")
print("Sareddy Poojya Sree\n1BM23CS303")
```

Implement a* using Manhattan Distance:


g = 4
g = 5
Goal state
f(n) = 5+0 = 5
* Manhattan Distance.
Initial state
goal state.
Sol:
depth
g = 1
L
U
D
0+2+3+1+1+2+2+3
= 14
f(n) = 14+1 = 15
0+1+3+1+1+2+2+2
= 12
f(n) = 12+1
= 13
0+1+3+1+1+2+3+3
= 14
f(n) = 14+1 = 15

1		5
3	2	8
4	6	7

L

1		5
3	2	8
4	6	7

g = 2

1	2	3	4	5	6	7	8
0	1	3	1	2	2	2	1

0+1+3+1+2+2+2+2
= 13

f(n) = 13 + 2 = 15

g = 3.

1	1	5
3	2	8
4	6	7

1+1+3+1+2+2+2
+2

= 14

f(n) = 14 + 3 = 17.

1	2	5
3		8
4	6	7

0+0+3+1+2+2+2+2

= 12

f(n) = 12 + 3
= 15

g = 4.

1	2	5
	3	8
4	6	7

1	2	5
3	8	
4	6	7

1	2	5
3	6	8
4		7

depth

↓

g = 1

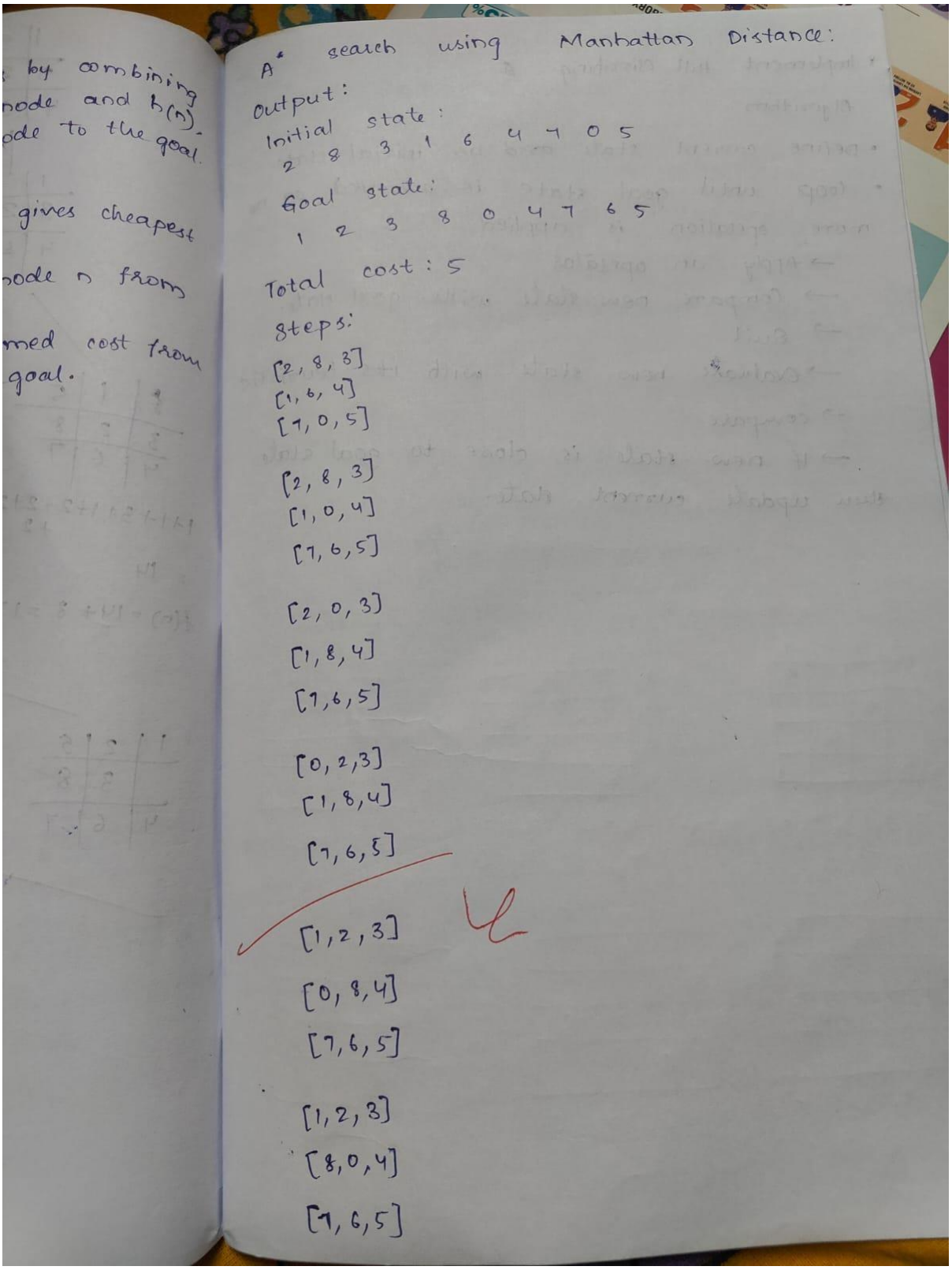
8

7

5	6	7	8
1	2	3	2

+1+2+3+3

+1 = 15



by combining
node and h(n).
ode to the goal.

gives cheapest

node n from

med cost from
goal.

A* search using Manhattan Distance:

Output:

Initial state:
2 8 3 1 6 4 7 0 5

Goal state:
1 2 3 8 0 4 7 6 5

Total cost: 5

steps:

[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]

Output:

```
Enter initial state (9 numbers, use 0 for blank):
2 8 3 1 6 4 7 0 5
Enter goal state (9 numbers, use 0 for blank):
1 2 3 8 0 4 7 6 5

Solution found in 5 moves.
Total cost: 5

Steps:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Sareddy Poojya Sree
1BM23CS303
```

Code:

```
import heapq

# Heuristic: Manhattan distance
def manhattan(state, goal):
    distance = 0
    for i in range(1, 9):  # ignore blank (0)
        x1, y1 = divmod(state.index(i), 3)
        x2, y2 = divmod(goal.index(i), 3)
        distance += abs(x1 - x2) + abs(y1 - y2)
```

```
    return distance

# Generate possible next states
def get_neighbors(state):
    neighbors = []
    blank = state.index(0)
    x, y = divmod(blank, 3)

    moves = [(-1,0), (1,0), (0,-1), (0,1)] # up, down, left, right
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_blank = nx*3 + ny
            new_state = list(state)
            new_state[blank], new_state[new_blank] = new_state[new_blank],
new_state[blank]
            neighbors.append(tuple(new_state))
    return neighbors

# A* Search
def a_star(initial, goal):
    frontier = []
    heapq.heappush(frontier, (manhattan(initial, goal), 0, initial, [initial]))
    visited = set()

    while frontier:
        f, g, state, path = heapq.heappop(frontier)

        if state == goal:
            return path, g  # return path and cost

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

        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + manhattan(neighbor, goal)
                heapq.heappush(frontier, (new_f, new_g, neighbor, path + [neighbor]))
    return None, None

# Print puzzle in 3x3 grid
def print_state(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
```

```
    print()

# Main
if __name___== "__main__":
    print("Enter initial state (9 numbers, use 0 for blank):")
    initial = tuple(map(int, input().split()))
    print("Enter goal state (9 numbers, use 0 for blank):")
    goal = tuple(map(int, input().split()))

    path, cost = a_star(initial, goal)

    if path:
        print("\nSolution found in", len(path)-1, "moves.")
        print("Total cost:", cost)
        print("\nSteps:")
        for step in path:
            print_state(step)
    else:
        print("No solution found!")
print("Sareddy Poojya Sree\n1BM23CS303")
```

Week - 4

Implement Hill Climbing

Algorithm:

Week-4

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

Algorithm/Pseudocode:

function Hill-climbing returns a state that is a local maximum.

current ← Make-node (problem.Initial-state)

loop do

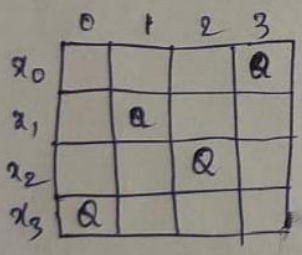
neighbour ← a highest-valued successor of current

if neighbour.Value ≤ current.value

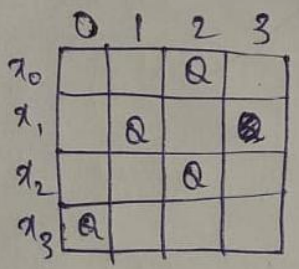
then return current.State

current ← neighbour;

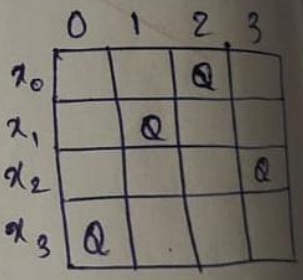
Solve 4-Queens:



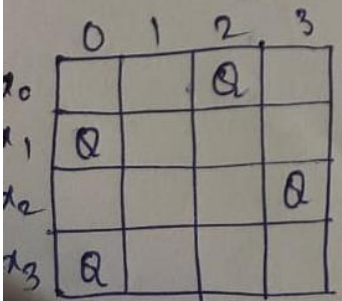
$x_0 = 3, x_1 = 1, x_2 = 2, x_3 = 0$



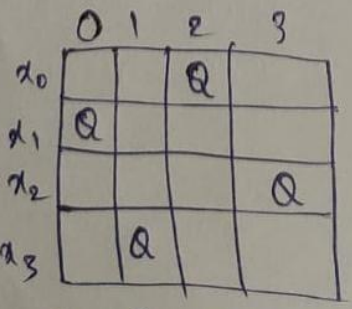
$x_0 = 2, x_1 = 1$
$x_2 = 2, x_3 = 0$



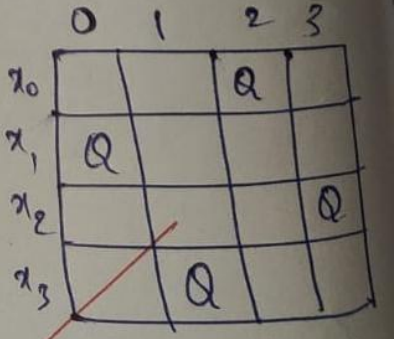
$x_0 = 2, x_1 = 1$
$x_2 = 3, x_3 = 0$



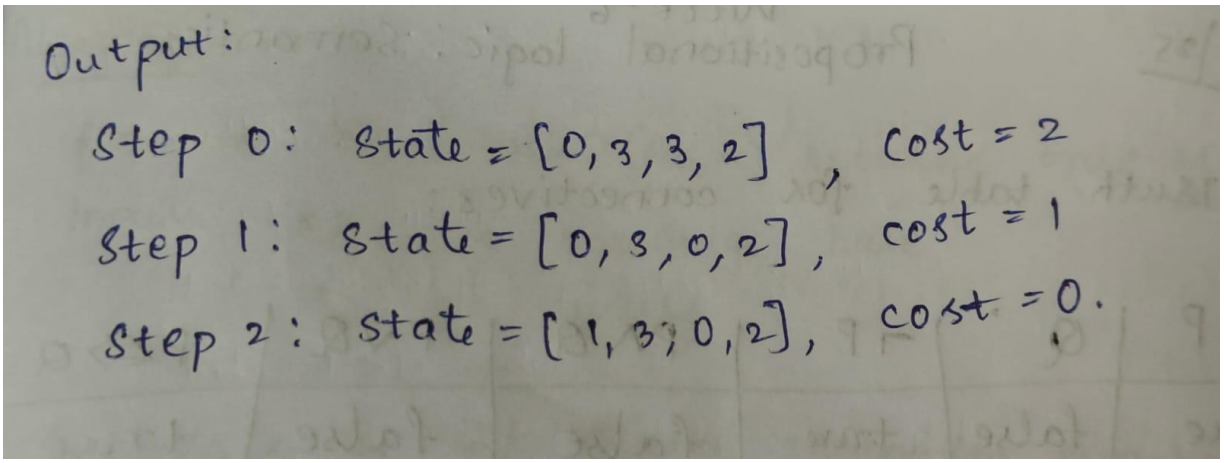
$x_0 = 2, x_1 = 0$
$x_2 = 3, x_3 = 0$



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



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



Output:

```
Step 0: State=[0, 3, 3, 2], Cost=2
Step 1: State=[0, 3, 0, 2], Cost=1
Step 2: State=[1, 3, 0, 2], Cost=0
Sareddy Poojya Sree
1BM23CS303
```

Code:

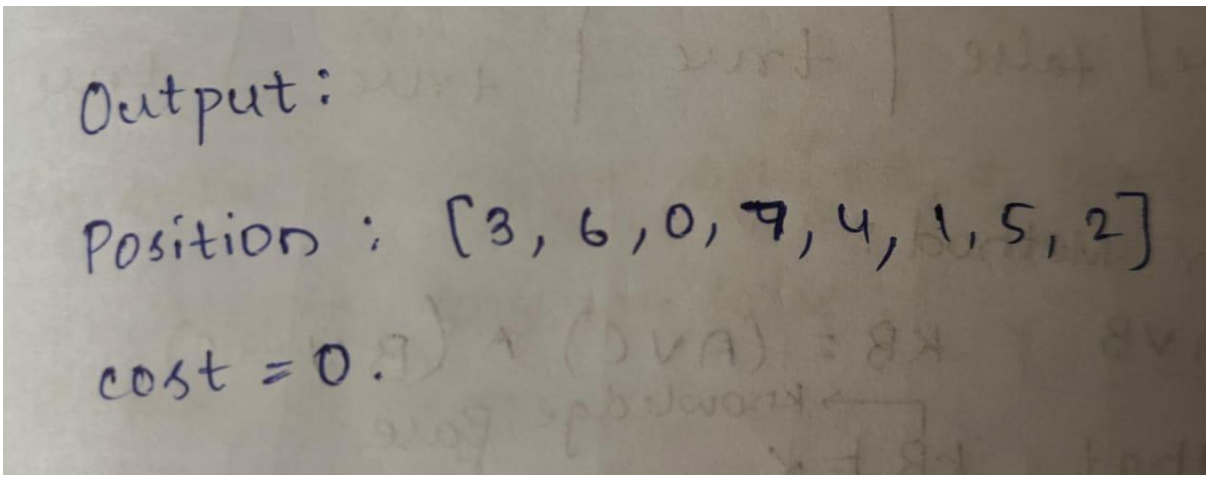
```
def hill_climbing_with_restarts(n):
    max_restarts = 1000  # limit restarts to avoid infinite loop
    restart_count = 0

    while restart_count < max_restarts:
        # Start with a random initial state
        current = [random.randint(0, n-1) for _ in range(n)]
        steps = []
        step_count = 0

        while True:
            current_cost = calculate_cost(current)
            steps.append((current, current_cost))
            print(f"Step {step_count}: State={current},
Cost={current_cost}")
            step_count += 1

            if current_cost == 0:  # solution found

                return steps

            neighbors = generate_neighbors(current)
            neighbor_costs = [(n, calculate_cost(n)) for n in
neighbors]
```

```
            best_neighbor, best_cost = min(neighbor_costs, key=lambda
x: x[1])

            if best_cost >= current_cost:
                # no improvement, restart with a new random state
                print(f"Restarting after {step_count} steps, cost stuck
at {current_cost}")
                restart_count += 1
                break

            current = best_neighbor

    print("Failed to find a solution after max restarts.")
    return None

steps = hill_climbing_with_restarts(4)
print("Sareddy Poojya Sree\n1BM23CS303")
```

Week – 5

Implement Simulated Annealing:

Algorithm:



Output:

```
The best position found: [1, 6, 4, 7, 0, 3, 5, 2]
cost = 0
Sareddy Poojya Sree
1BM23CS303
```

Code:

```
import random
import math# Heuristic: number of attacking pairs
def calculate_cost(state):
    cost = 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):
                cost += 1
    return cost


# Generate a random neighbor
def get_random_neighbor(state):
    n = len(state)
```

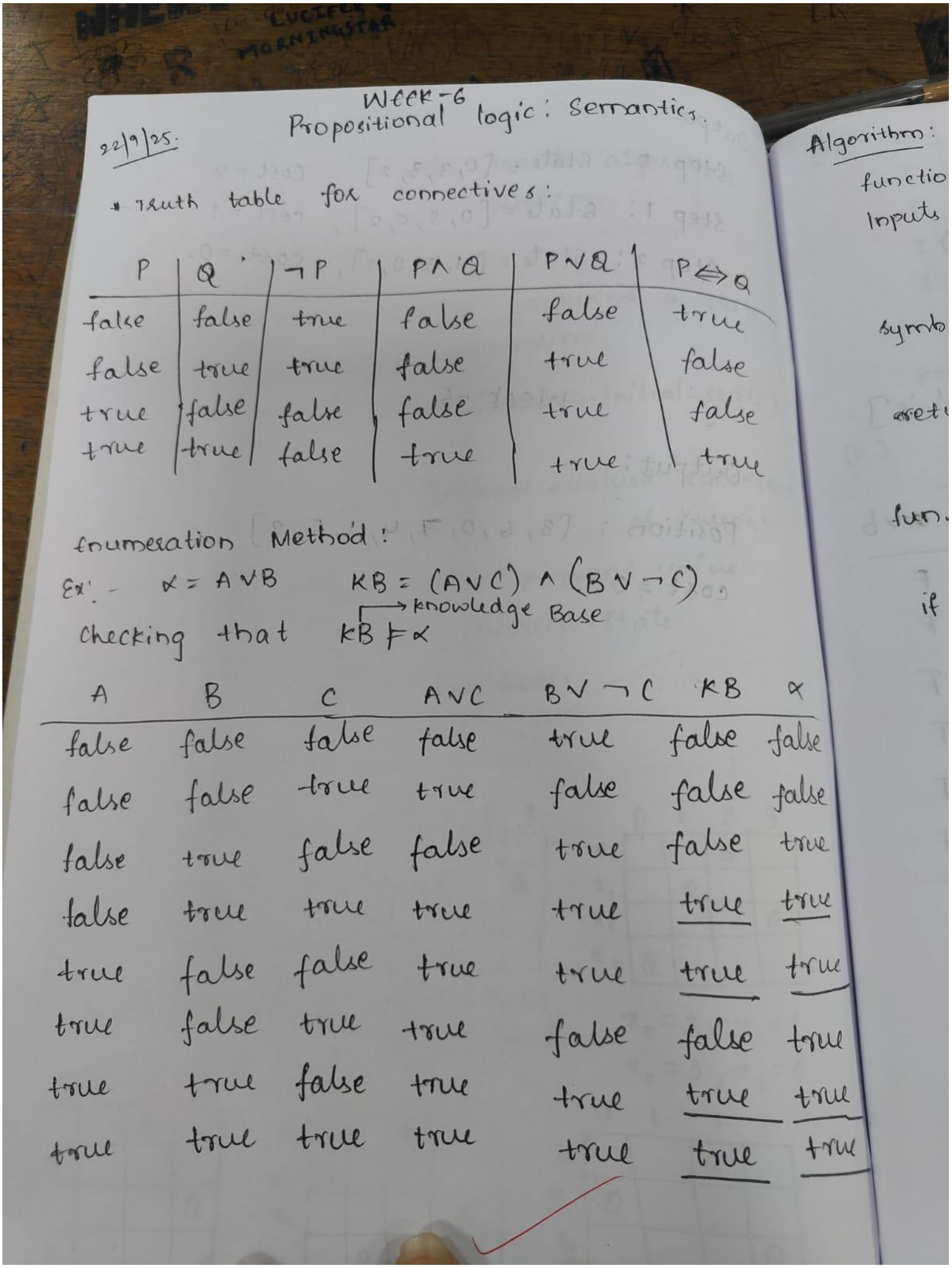
```
    new_state = list(state)
    col = random.randint(0, n - 1)   # pick random column
    row = random.randint(0, n - 1)   # new row
    new_state[col] = row
    return new_state

def simulated_annealing(n=8, max_iterations=10000, initial_temp=100.0, cooling_rate=0.99):
    # start with a random state
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current)
    best = current
    best_cost = current_cost
    temperature = initial_temp

    for _ in range(max_iterations):
        if current_cost == 0:
            break  # found solution

        neighbor = get_random_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current, current_cost = neighbor, neighbor_cost

            if current_cost < best_cost:
                best, best_cost = current, current_cost

        temperature *= cooling_rate
        if temperature < 1e-6:
            break
```

```
    return best, best_cost

best_state, best_cost = simulated_annealing()

print("The best position found:", best_state)
print("cost =", best_cost)
```

Week - 6

Propositional Logic: Semantic

Algorithm:



WEEK-6

22/9/25.

Propositional logic: Semantics.

* Truth table for connectives:

P	Q	¬P	P∧Q	P∨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	true

Enumeration Method:

Ex:- $\alpha = A \vee B$ $KB = (A \vee C) \wedge (B \vee \neg C)$

KB → knowledge Base

Checking that $KB \models \alpha$

A	B	C	A∨C	B∨¬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

Algorithm:

functio

Inputs

symb

retu

fun

if

Algorithm:

function TT-Entails? (KB, α) returns true or false
Inputs : KB, the knowledge base
α, the query

symbols ← a list of the proposition symbols in KB and α.
return TT-check-All (KB, α, symbols, {})

function TT-Check-All (KB, α, symbols, model)
return true or false
if EMPTY? (symbols) then
if PL-TRUE? (KB, model)
else return true
else do
P ← first (symbols)
rest ← Rest (symbols)
return (TT-check-All (KB, α, rest, model ∪ {P = true}))
and
TT-check-All (KB, α, rest, model ∪ {P = false}))

Output:

Enter proportional formulas in kb:
(a∨c), (b∨¬c)
Enter combination of KB(AND/OR):
AND
Enter the query formula:
a∨b
proposition variables found: ['a', 'b', 'c']

Truth Table:

a	b	c	(a∨c)	(b∨¬c)	KB	a∨b
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

Knowledge Base entails the query: YES

Consider: S & T as variables and following relation:

a: ¬(s∨t) b: (s∧t) c: T∨¬T.

Write TT and show whether

i) a entails b

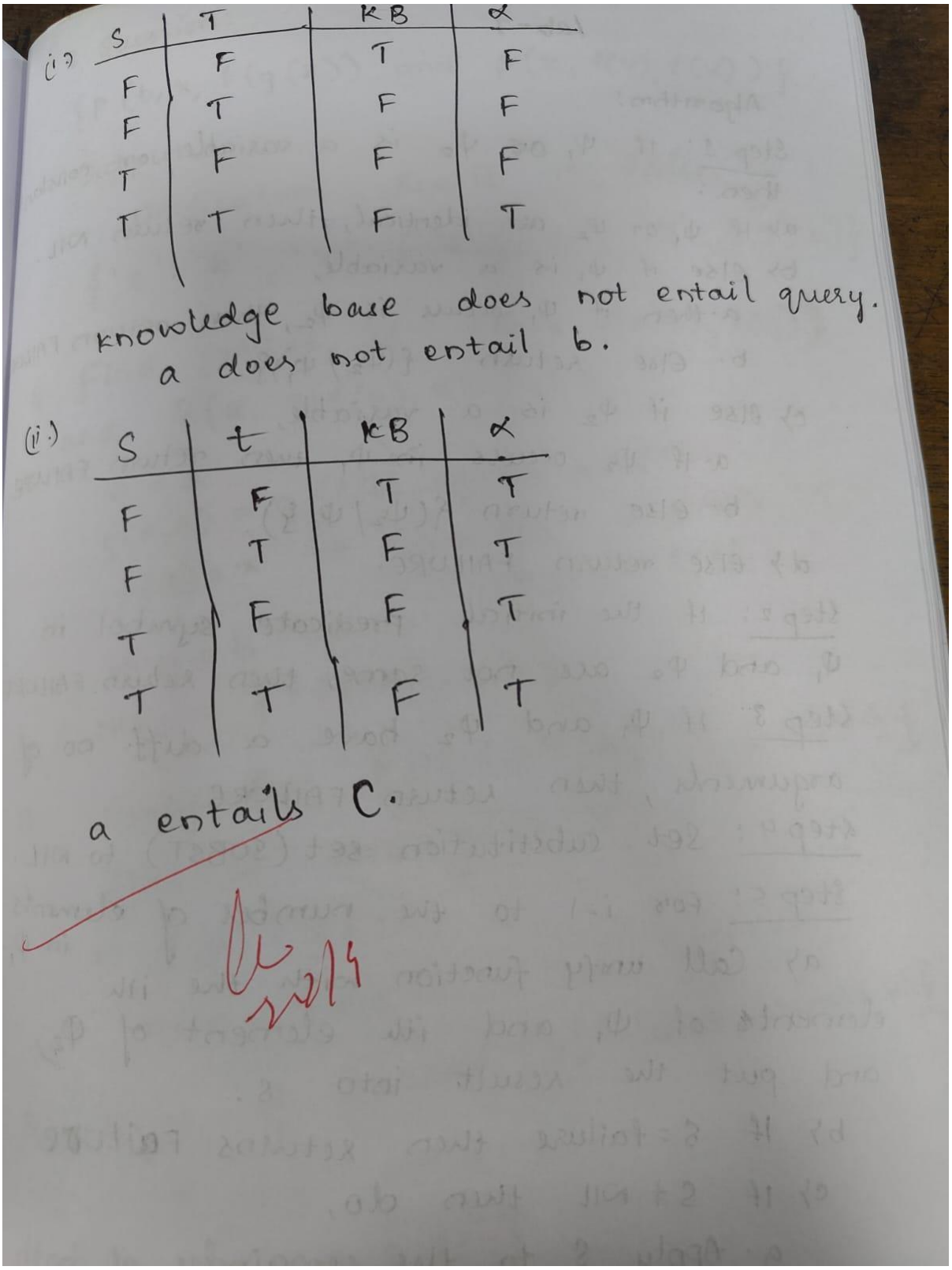
ii) a ∨ c

(i)

S
F
F
T
T

(ii)

S
F
F



(i)

S	T	KB	α
F	F	T	F
F	T	F	F
T	F	F	F
T	T	F	T

knowledge base does not entail query.
a does not entail b.

(ii)

S	t	KB	α
F	F	T	T
F	T	F	T
T	F	F	T
T	T	F	T

a entails C.

Output:

```
Enter propositional formulas in the knowledge base (separated by commas):
(a v c), (b v -c)
Enter how to combine KB formulas (AND / OR):
AND
Enter the query formula:
a v b

Propositional Variables found: ['A', 'B', 'C']

Full Truth Table:

   A    |    B    |    C    | (a v c) | (b v -c) | KB (AND) |  a v b
-------------------------------------------------------------------
   F    |    F    |    F    |    F    |    T     |    F     |    F
   F    |    F    |    T    |    T    |    F     |    F     |    F
   F    |    T    |    F    |    F    |    T     |    F     |    T
   F    |    T    |    T    |    T    |    T     |    T     |    T
   T    |    F    |    F    |    T    |    T     |    T     |    T
   T    |    F    |    T    |    T    |    F     |    F     |    T
   T    |    T    |    F    |    T    |    T     |    T     |    T
   T    |    T    |    T    |    T    |    T     |    T     |    T

Knowledge Base entails the query: YES
Sareddy Poojya Sree
1BM23CS303
```

Code:

```python
import itertools

def preprocess_formula(formula: str) -> str:
    formula = formula.upper()
    formula = formula.replace('^', '&')
    formula = formula.replace('V', '|')
    formula = formula.replace('v', '|')
    formula = formula.replace('-', '~')
    return formula

def tokenize(s):
    tokens = []
    i = 0
    while i < len(s):
        c = s[i]
        if c == ' ':
            i += 1
            continue
        if c in ('(', ')', '~', '&', '|'):
            tokens.append(c)
            i += 1
        elif c == '-':
            if i+1 < len(s) and s[i+1] == '>':
                tokens.append('->')
                i += 2
            else:
```

```
                raise ValueError("Invalid token: '-' not followed by
'>'")
        elif c == '<':
            if s[i:i+3] == '<->':
                tokens.append('<->')
                i += 3
            else:
                raise ValueError("Invalid token starting with '<'")
        elif c.isalpha() and c.isupper():
            tokens.append(c)
            i += 1
        else:
            raise ValueError(f"Invalid character: {c}")
    return tokens

def parse_formula(tokens):
    def parse_expr():
        return parse_biconditional()
    def parse_biconditional():
        left = parse_implication()
        while tokens and tokens[0] == '<->':
            tokens.pop(0)
            right = parse_implication()
            left = ('<->', left, right)
        return left
    def parse_implication():
        left = parse_or()
        while tokens and tokens[0] == '->':
            tokens.pop(0)
            right = parse_implication()
            left = ('->', left, right)
        return left
    def parse_or():
        left = parse_and()
        while tokens and tokens[0] == '|':
            tokens.pop(0)
            right = parse_and()
            left = ('|', left, right)
        return left
    def parse_and():
        left = parse_not()
        while tokens and tokens[0] == '&':
            tokens.pop(0)
            right = parse_not()
            left = ('&', left, right)
        return left
    def parse_not():
        if tokens and tokens[0] == '~':
```

```
            tokens.pop(0)
            operand = parse_not()
            return ('~', operand)
        else:
            return parse_atom()
    def parse_atom():
        if not tokens:
            raise ValueError("Unexpected end of input")
        token = tokens.pop(0)
        if token == '(':
            node = parse_expr()
            if not tokens or tokens.pop(0) != ')':
                raise ValueError("Missing closing parenthesis")
            return node
        elif token.isalpha():
            return ('var', token)
        else:
            raise ValueError(f"Unexpected token: {token}")
    return parse_expr()

def eval_parsed_formula(node, valuation):
    op = node[0]
    if op == 'var':
        return valuation[node[1]]
    elif op == '~':
        return not eval_parsed_formula(node[1], valuation)
    elif op == '&':
        return eval_parsed_formula(node[1], valuation) and
eval_parsed_formula(node[2], valuation)
    elif op == '|':
        return eval_parsed_formula(node[1], valuation) or
eval_parsed_formula(node[2], valuation)
    elif op == '->':
        return (not eval_parsed_formula(node[1], valuation)) or
eval_parsed_formula(node[2], valuation)
    elif op == '<->':
        return eval_parsed_formula(node[1], valuation) ==
eval_parsed_formula(node[2], valuation)
    else:
        raise ValueError(f"Unknown node type: {op}")

def extract_vars(formulas):
    vars = set()
    def helper(node):
        if node[0] == 'var':
            vars.add(node[1])
        elif node[0] in ('~', '&', '|', '->', '<->'):
            if isinstance(node[1], tuple):
```

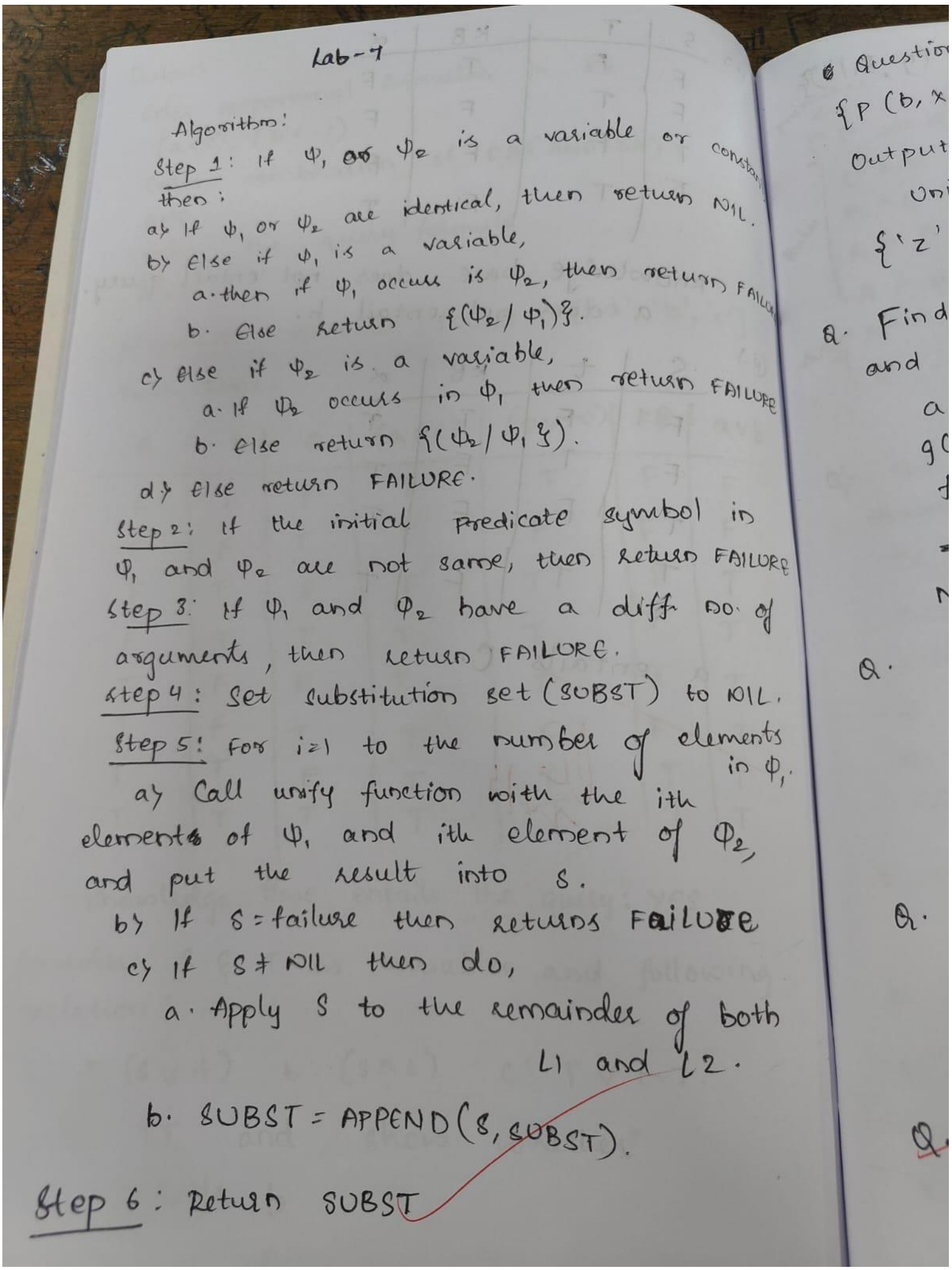
```
                helper(node[1])
            if len(node) > 2 and isinstance(node[2], tuple):
                helper(node[2])
    for f in formulas:
        helper(f)
    return sorted(vars)

def main():
    print("Enter propositional formulas in the knowledge base
(separated by commas):")
    kb_input = input().strip()
    kb_raw = [f.strip() for f in kb_input.split(',') if f.strip()]
    kb_processed = [preprocess_formula(f) for f in kb_raw]

    print("Enter how to combine KB formulas (AND / OR):")
    comb = input().strip().upper()
    if comb not in ('AND', 'OR'):
        print("Invalid combination choice, defaulting to AND.")
        comb = 'AND'

    print("Enter the query formula:")
    query_input = input().strip()
    query_processed = preprocess_formula(query_input)

    kb_parsed = []
    for formula in kb_processed:
        tokens = tokenize(formula)
        parsed = parse_formula(tokens)
        if tokens:
            raise ValueError("Extra tokens after parsing formula")
        kb_parsed.append(parsed)

    tokens_query = tokenize(query_processed)
    query_parsed = parse_formula(tokens_query)
    if tokens_query:
        raise ValueError("Extra tokens after parsing query")

    all_formulas = kb_parsed + [query_parsed]
    vars = extract_vars(all_formulas)

    print("\nPropositional Variables found:", vars)

    col_width = 7
    headers = vars + kb_raw + [f'KB ({comb})', query_input]
    print("\nFull Truth Table:\n")
    print(" | ".join(h.center(col_width) for h in headers))
    print("-" * (len(headers) * (col_width + 3) - 3))
```

```
    entails = True
    for values in itertools.product([False, True], repeat=len(vars)):
        valuation = dict(zip(vars, values))
        kb_vals = [eval_parsed_formula(f, valuation) for f in
kb_parsed]

        if comb == 'AND':
            kb_combined = all(kb_vals)
        else:
            kb_combined = any(kb_vals)

        query_val = eval_parsed_formula(query_parsed, valuation)

        row_vals = [('T' if val else 'F').center(col_width) for val in
values]
        row_vals += [('T' if val else 'F').center(col_width) for val in
kb_vals]
        row_vals.append(('T' if kb_combined else
'F').center(col_width))
        row_vals.append(('T' if query_val else 'F').center(col_width))

        print(" | ".join(row_vals))

        # Check entailment condition:
        if kb_combined and not query_val:
            entails = False

    print("\nKnowledge Base entails the query:" , "YES" if entails else
"NO")

if __name__== "__main__":
    main()

print("Sareddy Poojya Sree\n1BM23CS303")
```

Week – 7

Implement unification in first order logic.

Algorithm:



Lab-7

Algorithm:

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 is 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 $\{(\Psi_2 / \Psi_1)\}$.

d) Else return 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 a diff no. of arguments, then return FAILURE.

Step 4: Set substitution set (SUBST) to NIL.

Step 5: For i=1 to the number of elements in Ψ_1.

a) Call unify function with the ith element of Ψ_1 and ith element of Ψ_2, and put the result into S.

b) If S = failure then returns FAILURE

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST = APPEND(S, SUBST).

Step 6: Return SUBST

* Question:
{P(b, x, f(g(z))) and P(z, f(y), f(y))}.

Output:
Unification Result is ~~false~~.
{'z': 'b', 'x': ['f', 'y'], 'y': ['g', b]}

Q. Find MGU of {Q(a, g(x,a), f(y)) and Q(a, g(f(b), a), x)}.

$a = a$
$g(x,a) = g(f(b), a) \Rightarrow x = f(b)$
$f(y) = x \rightarrow$ substitue $x = f(b)$
$\rightarrow f(y) = f(b)$

$\Rightarrow y = b$
MGU = {x(f(b)), y/b}.

Q. Unify for {P{f(a), g(y)), P(x, x)}.
$f(a) = x$ and $g(y) = x$
$\Rightarrow f(a) = g(y)$
MGU = FAIL

Q. MGU of {prime (11) and prime(y)}
$11 \rightarrow y \rightarrow y = 11$
MGU = {y/11}

Q. MGU {knows (John, x), knows(y, mother(y))}

John = y x = (mother (y))
$\Rightarrow x$ = mother (John)
MGU = {y/John, x/mother (John)}

Output:

```
Unification succeeded with substitution: {'x': 'B', 'y': 'A'}
```

Code:

```
def unify(x, y, subst=None):
    if subst is None:
        subst = {}

    # If x or y is a variable or constant
    if is_variable(x) or is_constant(x):
        if x == y:
            return subst
        elif is_variable(x):
            return unify_var(x, y, subst)
        elif is_variable(y):
            return unify_var(y, x, subst)
        else:
            return None

    # If both x and y are compound expressions
    if is_compound(x) and is_compound(y):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
        for xi, yi in zip(x[1], y[1]):
            subst = unify(xi, yi, subst)
            if subst is None:
                return None
        return subst
    return None

def is_variable(x):
    return isinstance(x, str) and x.islower() and x.isalpha()

def is_constant(x):
```

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

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

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst
```



```
def occurs_check(var, x, subst):
    if var == x:
        return True
    elif is_variable(x) and x in subst:
        return occurs_check(var, subst[x], subst)
    elif is_compound(x):
        return any(occurs_check(var, arg, subst) for arg in x[1])
    else:
        return False

# Example usage:
# Let's say we want to unify P(x, A) and P(B, y)
x = ("P", ["x", "A"])
y = ("P", ["B", "y"])
```

```
result = unify(x, y)
if result is not None:
    print("Unification succeeded with substitution:", result)
else:
    print("Unification failed.")
```

Week – 8

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

Algorithm:

Bill

WEEK - 8.

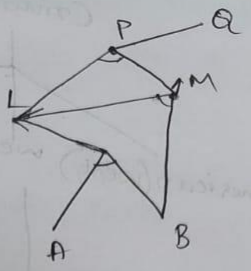
premises → conclusion

$P \Rightarrow Q$

Rules:
- $L \wedge M \Rightarrow P$ ✓
- $B \wedge L \Rightarrow M$ ✓
- $A \wedge P \Rightarrow L$
- $A \wedge B \Rightarrow L$ ✓

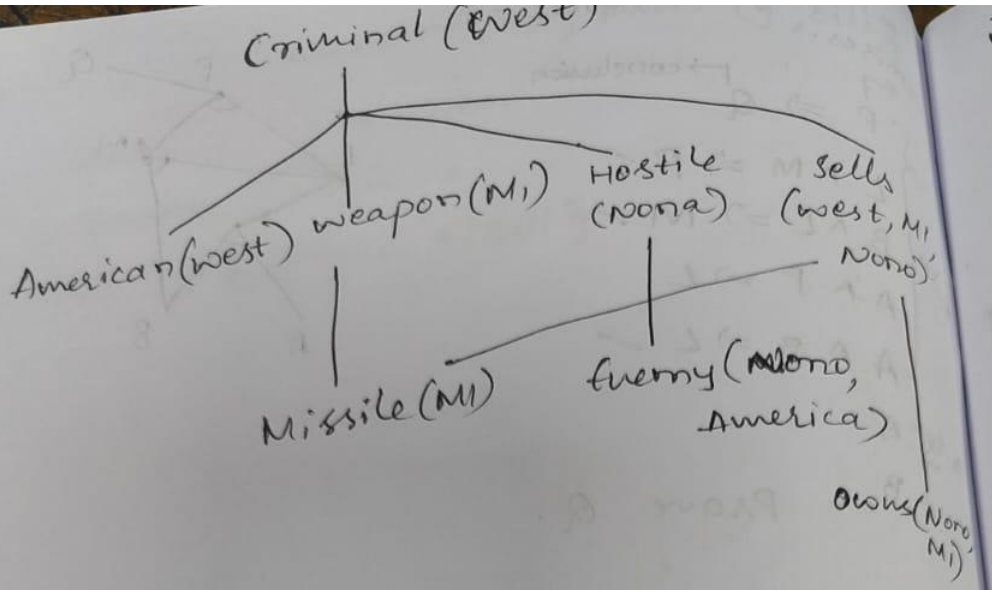
facts:
- A
- B

Prove Q.



Q) The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American. An enemy of America counts as "hostile".
Prove that "West is criminal".

1. $\forall x, y, z$ $\text{America}(x) \wedge \text{Weapon}(y) \wedge \text{Sells}(x, y, z) \wedge \text{Hostile}(z) \Rightarrow \text{Criminal}(x)$
2. $\forall x$ $\text{missile}(x) \wedge \text{owns}(\text{Nono}, x) \Rightarrow \text{Sells}(\text{West}, x, \text{Nono})$
3. $\forall x$ $\text{Enemy}(x, \text{America}) \Rightarrow \text{Hostile}(x)$
4. $\forall x$ $\text{missile}(x) \Rightarrow \text{weapon}(x)$
5. American (West)
6. Enemy (Nono, America)
7. owns (Nono, M_1) and
8. Missile (M_1)



Algorithm:

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

inputs: KB, knowledge Base, a set of first-order definite clauses.

α, the query, an atomic sentence.

local variables: new, the new sentences inferred on each iteration.

repeat until new is empty

new $\leftarrow$ {}

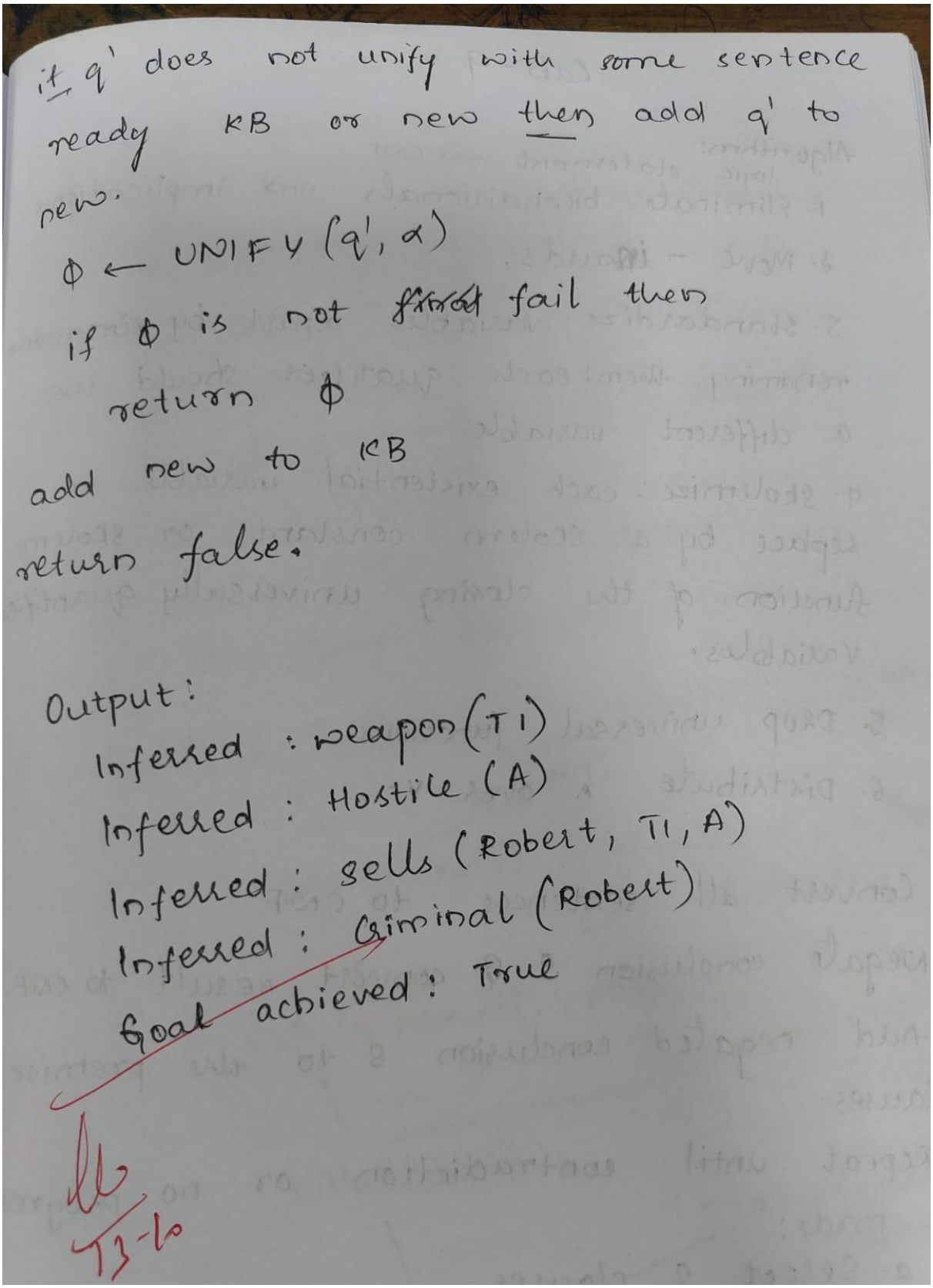
for each rule in KB do

$(p_1 \wedge p_2 \wedge \dots \wedge p_n \Rightarrow q) \leftarrow$ Standardize-Variable (rule)

for each θ such that SUBST (θ, $p_1 \wedge \dots \wedge p_n$) = SUBST ($\theta$, $p'_1 \wedge \dots \wedge p'_n$)

for some $p'_1, \dots p'_n$ in KB

$q' \leftarrow$ SUBST (θ, q)


if q' does not unify with some sentence
ready 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.
Output:
Inferred : weapon (T1)
Inferred : Hostile (A)
Inferred : sells (Robert, T1, A)
Inferred : Criminal (Robert)
Goal achieved : True

Output:

```
New fact inferred: Criminal(West)
New fact inferred: SoldWeapons(West, Nono)

Final facts:
American(West)
Hostile(Nono)
Missiles(Nono)
Criminal(West)
SoldWeapons(West, Nono)
```

Code:

```
facts = {
    'American(West)': True,
    'Hostile(Nono)': True,
    'Missiles(Nono)': True,
}
def rule1(facts):
    if facts.get('American(West)', False) and facts.get('Hostile(Nono)', False):
        return 'Criminal(West)'
    return None


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


def forward_chaining(facts, rules):
    new_facts = facts.copy()
    inferred = True
    while inferred:
        inferred = False
        for rule in rules:
            result = rule(new_facts)
            if result and result not in new_facts:
                new_facts[result] = True
```

```
            inferred = True
            print(f"New fact inferred: {result}")
    return new_facts
rules = [rule1, rule2]

inferred_facts = forward_chaining(facts, rules)

print("\nFinal facts:")
for fact in inferred_facts:
    print(fact)
```

Week – 9

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

Algorithm:

Lab-9

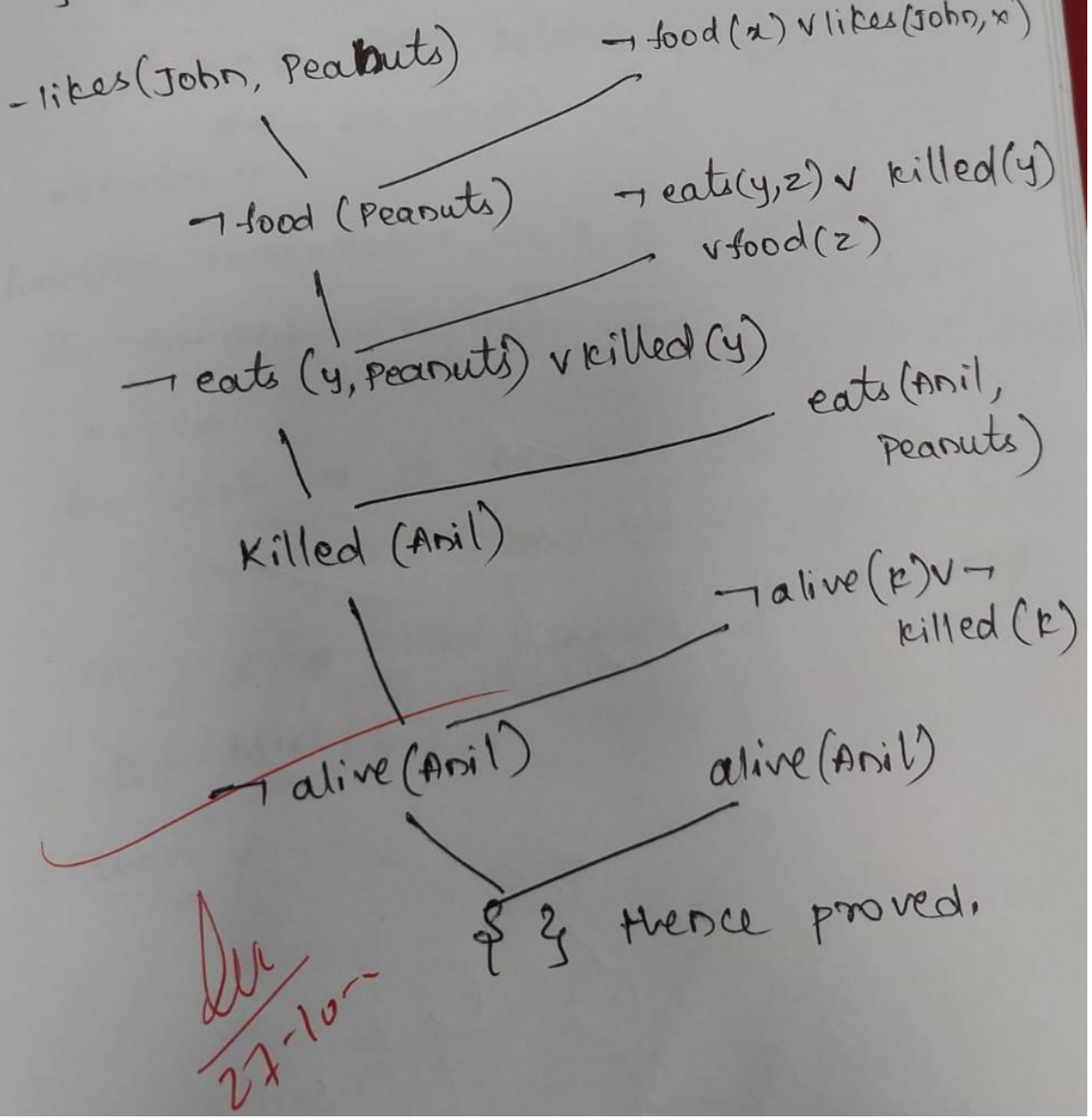
Algorithm: logic statement → CNF

1. Eliminate biconditionals and implications.
2. Move ¬ inwards:
3. Standardize variables apart by renaming them: each quantifier should use a different variable.
4. Skolemize: each existential variable is replace by a skolem constant or skolem function of the closing universally quantified variables.
5. Drop universal functions.
6. Distribute ∧ over ∨.

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF.
3. Add negated conclusion S to the premise clauses.
4. Repeat until contradiction or no progress is made:
   a. Select 2 clauses.
   b. Resolve them together, performing all required unifications.
   c. If resolvent 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.

a. food (x) ∨ likes (John, x)
b. food (Apple)
c. food (vegetables)
d. ¬ eats (y, z) ∨ killed (y) ∨ food (z)
e. eats (Anil, Peanuts)
f. alive (Anil)
g. ¬ eats (Anil, w) ∨ eats (Harry, w)
h. killed (g) ∨ alive (g)
i. ¬ alive (k) ∨ ¬ killed (k)
j. likes (John, Peanuts).



27-10-

Output:

```
FOL resolution prover (basic example)

Knowledge base clauses:
   Food(apple)
   Likes(John,x) OR ~Food(x)
   ~Alive(x) OR ~Killed(x)
   ~Eats(x,y) OR Food(y) OR Killed(y)
   Alive(x) OR ~Killed(x)
   Alive(Anil)
   Food(vegetable)
   Eats(Anil,peanuts)
   Eats(Harry,x) OR ~Eats(Anil,x)

Query: Likes(John,peanuts)
Negated query clause will be added to KB and resolution attempted.

Result: True | Derived empty clause (success)
```

Code:

```
from collections import deque

import itertools

import copy

import pprint

print('Shreya Raj 1BM23CS317')

# ----- Data structures -----

class Var:

    def __init__(self, name):

        self.name = name

    def __repr__(self):

        return f"Var({self.name})"

    def __eq__(self, other):

        return isinstance(other, Var) and self.name == other.name

    def __hash__(self):

        return hash(('Var', self.name)) class Const:

    def __init__(self, name):

        self.name = name
```

```
    def __repr__(self):
        return f"Const({self.name})"
    def __eq__(self, other):
        return isinstance(other, Const) and self.name == other.name
    def __hash__(self):
        return hash(('Const', self.name))

class Func:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __repr__(self):
        return f"Func({self.name}, {self.args})"
    def __eq__(self, other):
        return isinstance(other, Func) and self.name == other.name and self.args == other.args
    def __hash__(self):
        return hash(('Func', self.name, tuple(self.args)))

class Literal:
    # predicate_name: str, args: list of Terms, negated: bool
    def __init__(self, predicate, args, negated=False):
        self.predicate = predicate
        self.args = tuple(args)
        self.negated = negated
    def negate(self):
        return Literal(self.predicate, list(self.args), not self.negated)
    def __repr__(self):
        sign = "~" if self.negated else ""
```



```
        args = ",".join(map(term_to_str, self.args))
        return f"{sign}{self.predicate}({args})"
```

```
    def __eq__(self, other):
        return (self.predicate, self.args, self.negated) == (other.predicate, other.args, other.negated)
    def __hash__(self):
        return hash((self.predicate, self.args, self.negated))

# Clause is frozenset of Literal
def clause_to_str(cl):
    return " OR ".join(map(str, cl)) if cl else "EMPTY"

def term_to_str(t):
    if isinstance(t, Var):
        return t.name
    if isinstance(t, Const):
        return t.name
    if isinstance(t, Func):
        return f"{t.name}({','.join(term_to_str(a) for a in t.args)})"
    return str(t)

# ----- Substitution utilities -----
def apply_subst_term(term, subst):
    if isinstance(term, Var):
        if term in subst:
            return apply_subst_term(subst[term], subst)
        else:
            return term
    elif isinstance(term, Const):
        return term
    elif isinstance(term, Func):
        return Func(term.name, [apply_subst_term(a, subst) for a in term.args])
    else:
        return term
```

```
def apply_subst_literal(lit, subst):
    return Literal(lit.predicate, [apply_subst_term(a, subst) for a in lit.args], lit.negated)

def apply_subst_clause(clause, subst):
    return frozenset(apply_subst_literal(l, subst) for l in clause)

# ----- Unification (Robust, with occurs-check) -----
def occurs_check(var, term, subst):
    term = apply_subst_term(term, subst)
    if term == var:
        return True
    if isinstance(term, Func):
        return any(occurs_check(var, arg, subst) for arg in term.args)
    return False
def unify_terms(x, y, subst):
    # returns updated subst or None on failure
    x = apply_subst_term(x, subst)
    y = apply_subst_term(y, subst)

    if isinstance(x, Var):
        if x == y:
            return subst
        if occurs_check(x, y, subst):
            return None
        new = subst.copy()
        new[x] = y
        return new
    if isinstance(y, Var):
        return unify_terms(y, x, subst)
    if isinstance(x, Const) and isinstance(y, Const):
```

```
        return subst if x.name == y.name else None
    if isinstance(x, Func) and isinstance(y, Func) and x.name == y.name and len(x.args) == len(y.args):
        for a, b in zip(x.args, y.args):
            subst = unify_terms(a, b, subst)
            if subst is None:
                return None
        return subst
    return None

def unify_literals(l1, l2):
    # l1 and l2 must have same predicate and opposite polarity for resolution
    if l1.predicate != l2.predicate or l1.negated == l2.negated or len(l1.args) != len(l2.args):
        return None
    subst = {}
    for a, b in zip(l1.args, l2.args):
        subst = unify_terms(a, b, subst)
        if subst is None:
            return None
    return subst

# ----- Standardize apart variables (to avoid name clashes) -----
_var_count = 0
def standardize_apart(clause):
    global _var_count
    varmap = {}
    new_literals = []
    for lit in clause:
        new_args = []
        for t in lit.args:
            new_args.append(_rename_term_vars(t, varmap))
        new_literals.append(Literal(lit.predicate, new_args, lit.negated))
```

```
    return frozenset(new_literals)


def _rename_term_vars(term, varmap):
    global _var_count
    if isinstance(term, Var):
        if term.name not in varmap:
            _var_count += 1
            varmap[term.name] = Var(f"{term.name}_{_var_count}")
        return varmap[term.name]
    if isinstance(term, Const):
        return term
    if isinstance(term, Func):
        return Func(term.name, [_rename_term_vars(a, varmap) for a in term.args])
    return term


# ----- Resolution operation between two clauses -----
def resolve(ci, cj):
    # returns set of resolvent clauses (frozenset of literals)
    resolvents = set()
    ci = standardize_apart(ci)
    cj = standardize_apart(cj)
    for li in ci:
        for lj in cj:
            if li.predicate == lj.predicate and li.negated != lj.negated and len(li.args) == len(lj.args):
                subst = unify_literals(li, lj)
                if subst is not None:
                    # build resolvent: (Ci - {li}) U (Cj - {lj}) with subst applied
                    new_clause = set(apply_subst_literal(l, subst) for l in (ci - {li}) | (cj - {lj}))
                    # remove tautologies: a clause containing P and ~P after subst
                    preds = {}
```

```
            taut = False
            for l in new_clause:
                key = (l.predicate, tuple(map(term_to_str, l.args)))
                if key in preds and preds[key] != l.negated:
                    taut = True
                    break
                preds[key] = l.negated
            if not taut:
                resolvents.add(frozenset(new_clause))
    return resolvents

# ----- Main resolution loop -----
def fol_resolution(kb_clauses, query_clause, max_iterations=20000):
    """
    kb_clauses: set/list of clauses (each clause is frozenset of Literal)
    query_clause: single Literal (to be proved), will be negated and added to KB
    Returns True if contradiction (empty clause) is derived.
    """
```



```
    # Negate the query and add its literals as separate clauses (each literal is a clause)
    negated_query = [query_clause.negate()]
    clauses = set(kb_clauses)
    for l in negated_query:
        clauses.add(frozenset([l]))

    new = set()
    processed_pairs = set()
    queue = list(clauses)

    iterations = 0
    while True:
```

```
        pairs = []
        clause_list = list(clauses)
        n = len(clause_list)
        # iterate over all unordered pairs
        for i in range(n):
            for j in range(i+1, n):
                pairs.append((clause_list[i], clause_list[j]))
something_added = False
        for (ci, cj) in pairs:
            pair_key = (ci, cj)
            if pair_key in processed_pairs:
                continue
            processed_pairs.add(pair_key)
            resolvents = resolve(ci, cj)
            iterations += 1
            if iterations > max_iterations:
                return False, "max_iterations_exceeded"
            for r in resolvents:
                if len(r) == 0:
                    return True, "Derived empty clause (success)"
                if r not in clauses and r not in new:
                    new.add(r)
                    something_added = True
        if not something_added:
            return False, "No new clauses — failure (KB does not entail query)"
        clauses.update(new)
        new = set()

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

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

# Build clauses (using variables V('x'), constants C('Anil'), etc.)
60
x = V('x')
y = V('y')

kb = set()

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

# 2a. Food(Apple)
kb.add(frozenset([L('Food', [C('apple')], neg=False)]))
# 2b. Food(vegetable)
kb.add(frozenset([L('Food', [C('vegetable')], neg=False)]))

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

# 4a. Eats(Anil,peanuts)
kb.add(frozenset([L('Eats', [C('Anil'), C('peanuts')], neg=False)]))
# 4b. Alive(Anil)
kb.add(frozenset([L('Alive', [C('Anil')], neg=False)]))

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

# 6. ¬Alive(x) ∨ ¬Killed(x)
kb.add(frozenset([L('Alive', [x], neg=True), L('Killed', [x], neg=True)]))
```

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

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

def show_kb(kb):
    print("Knowledge base clauses:")
    for c in kb:
        print("  ", clause_to_str(c))
    print()

if __name___== "__main__":
    print("FOL resolution prover (basic example)\n")
    show_kb(kb)
    print("Query:", query)
    print("Negated query clause will be added to KB and resolution attempted.\n")
    success, info = fol_resolution(kb, query, max_iterations=20000)
    print("Result:", success, "|", info)
```

WEEK – 10

ALPHA-BETA PRUNING :

Algorithm:

Lab-10.

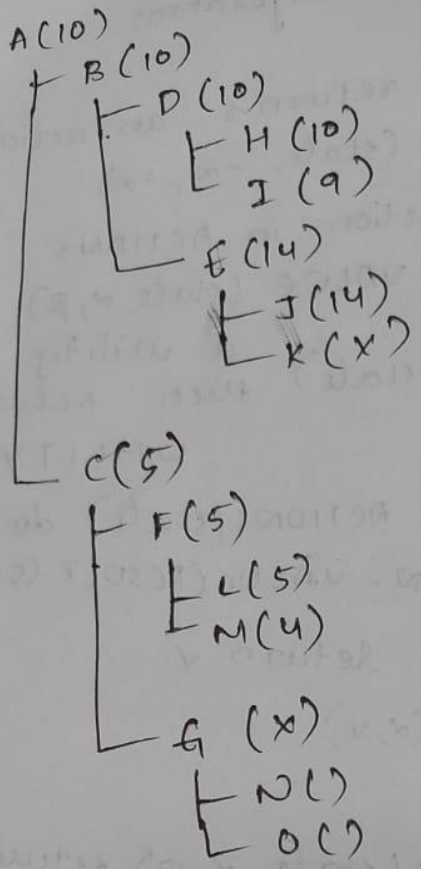
Alpha-beta search algorithm:

function Alpha returns an action.
v ← MAX-VALUE (state, −∞, +∞)
return the action in ACTIONS

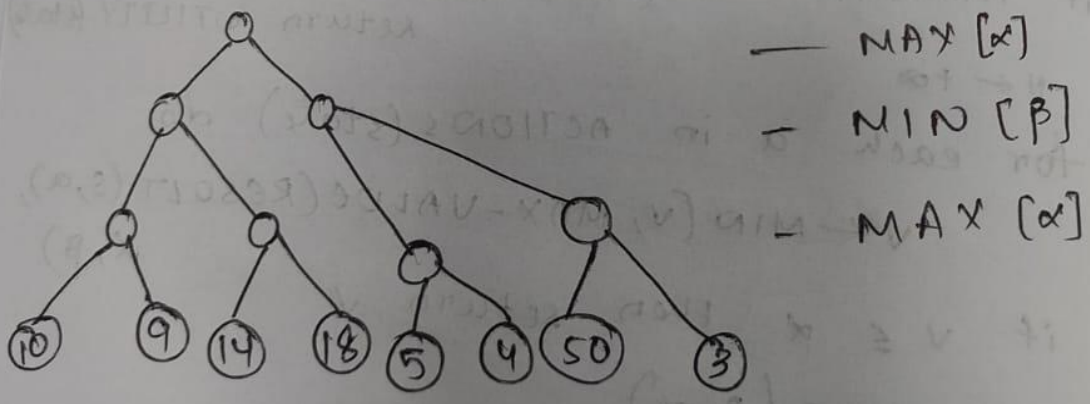
function MAX-VALUE (state, α, β) returns a utility value.
if TERMINAL-TEST (state) then return UTILITY (state)
v ← −∞
for each a in ACTIONS (state) do
v ← MAX (v, MIN-VALUE (RESULT (s,a), α, β))
if v ≥ β then return v
α ← MAX (α, v)
return v

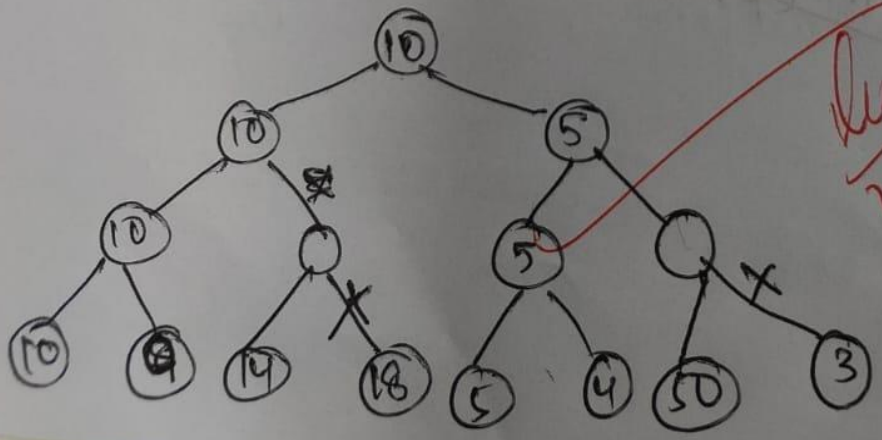
function MIN-VALUE (state, α, β) returns a utility value.
if TERMINAL-TEST (state) then return UTILITY (state)
N ← +∞
for each a in ACTIONS (state) do
v ← MIN (v, MAX-VALUE (RESULT (s,a), α, β)
if v ≤ α then return v
β ← MIN (β, v)
return v

Output:

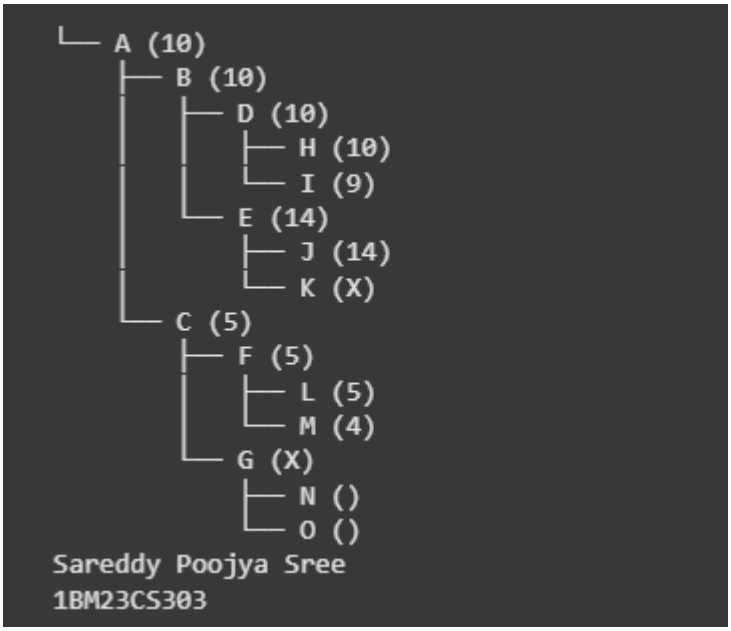


Question:





Output:



Code:

```
import math


tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': ['H', 'I'],
    'E': ['J', 'K'],
    'F': ['L', 'M'],
    'G': ['N', 'O'],
    'H': [], 'I': [], 'J': [], 'K': [],
    'L': [], 'M': [], 'N': [], 'O': []
}


# Leaf node values
values = {
    'H': 10, 'I': 9,
```

```
    'J': 14, 'K': 18,
    'L': 5, 'M': 4,
    'N': 50, 'O': 3
}

# to store final display values
node_values = {}

def get_children(node):
    return tree.get(node, [])

def is_terminal(node):
    return len(get_children(node)) == 0

def evaluate(node):
    return values[node]

def alpha_beta(node, depth, alpha, beta, maximizing):
    if is_terminal(node) or depth == 0:
        val = evaluate(node)
        node_values[node] = val
        return val

    if maximizing:
        value = -math.inf
        for child in get_children(node):
            val = alpha_beta(child, depth - 1, alpha, beta, False)
            value = max(value, val)
            alpha = max(alpha, val)
```

```
            if beta <= alpha:
                # mark remaining children as pruned
                for rem in get_children(node)[get_children(node).index(child)+1:]:
                    node_values[rem] = "X"
                break
        node_values[node] = value
        return value
    else:
        value = math.inf
        for child in get_children(node):
            val = alpha_beta(child, depth - 1, alpha, beta, True)
            value = min(value, val)
            beta = min(beta, val)
            if beta <= alpha:
                for rem in get_children(node)[get_children(node).index(child)+1:]:
                    node_values[rem] = "X"
                break
        node_values[node] = value
        return value

# Run pruning
alpha_beta('A', depth=4, alpha=-math.inf, beta=math.inf, maximizing=True)

def print_tree(node, prefix="", is_last=True):
    connector = " └── " if is_last else " ├── "
    value = node_values.get(node, "")
    print(prefix + connector + f"{node} ({value})")
    children = get_children(node)
    for i, child in enumerate(children):
```

```
        new_prefix = prefix + ("   " if is_last else "│   ")
        print_tree(child, new_prefix, i == len(children)-1)

# Display the final tree
print("\nFINAL TREE\n" )
print_tree('A')

print("Sareddy Poojya Sree\n1BM23CS303")
```