VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sanath S Shetty(1BM23CS297)

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

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Sanath S Shetty(1BM23CS297),** 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 Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

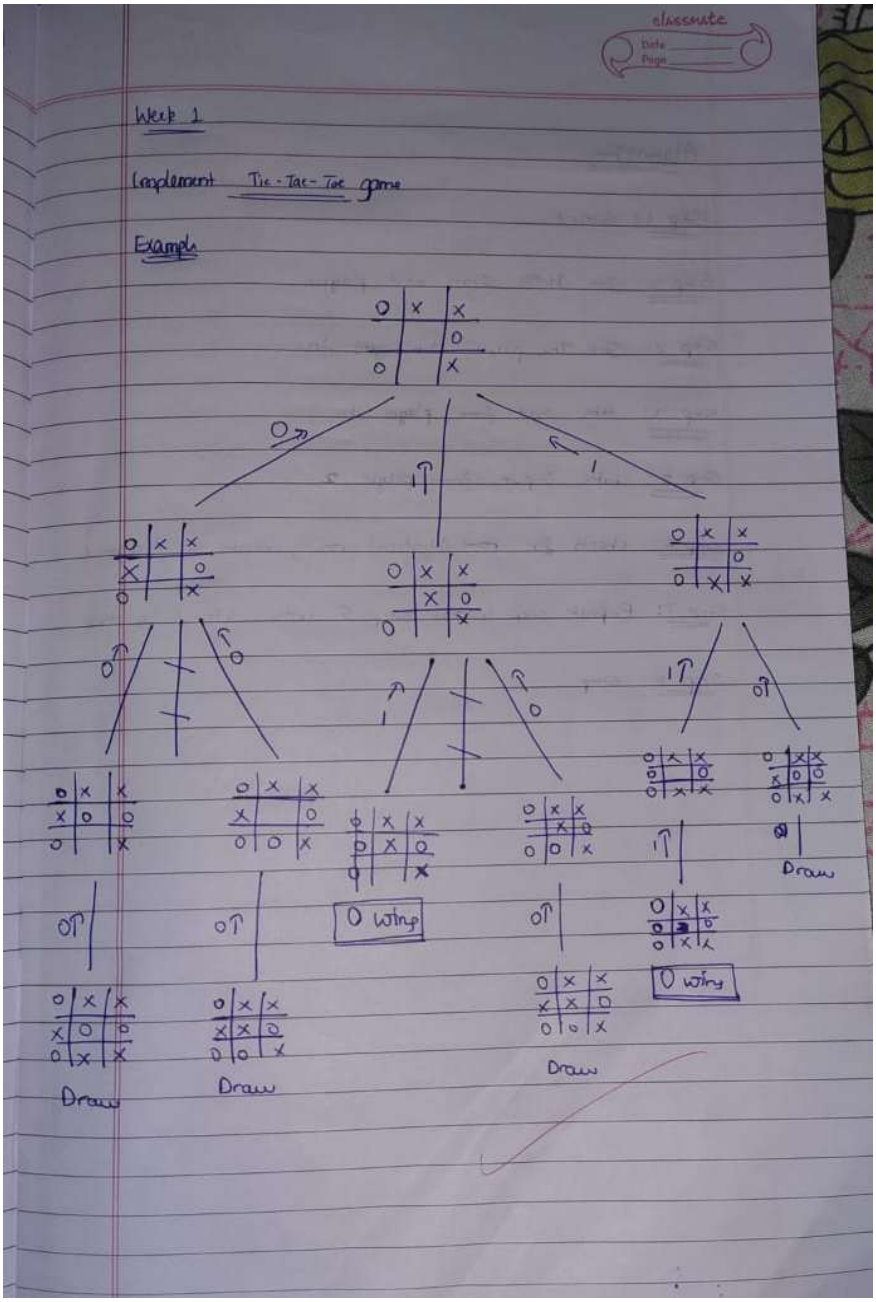
GitHub Link:

https://github.com/Sanath-S-Shetty/1BM23CS297-AI-LAB

Program 1

Implement Tic –Tac –Toe Game

Observation:



Algorithm

Step 1: Start

Step 2: Set initial state and player

Step 3: Set the player who goes first

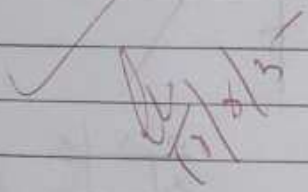
Step 4: take input from player 1

Step 5: take input from player 2

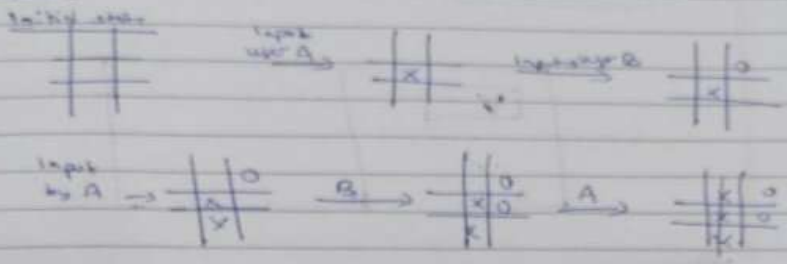
Step 6: check for identical row, column or diagonal

Step 7: Repeat step 4 and step 5 until step 6 is true.

Step 8: stop

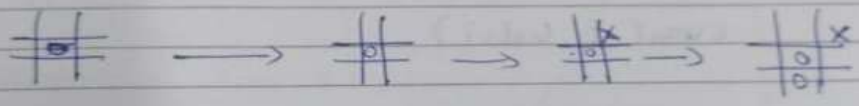


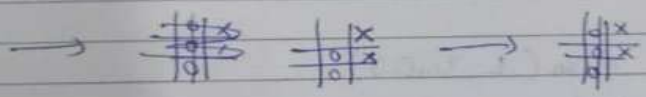
Output



X wins

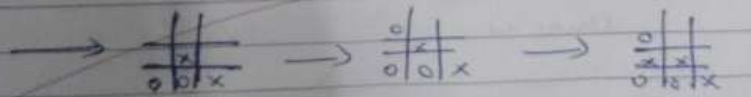
Initial state

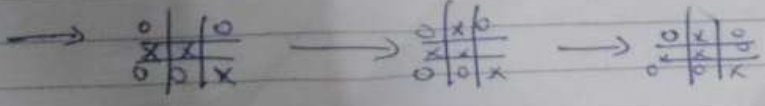




O wins

Initial state





Draw

Code:

```
print("SANATH S SHETTY 1BM23CS297")
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    game_over = False

    while not game_over:
        print_board(board)
        print(f"Player {current_player}'s turn.")

        while True:
            try:
                row = int(input("Enter row (0, 1, 2): "))
                col = int(input("Enter column (0, 1, 2): "))
                if 0 <= row <= 2 and 0 <= col <= 2 and board[row][col] == " ":
                    break
                else:
                    print("Invalid move. Try again.")
```

```
    except ValueError:
        print("Invalid input. Please enter a number.")

board[row][col] = current_player

win = False
for i in range(3):
    if all(board[i][j] == current_player for j in range(3)) or \
       all(board[j][i] == current_player for j in range(3)):
        win = True
        break
if not win and (all(board[i][i] == current_player for i in range(3)) or \
                all(board[i][2 - i] == current_player for i in range(3))):
    win = True

if win:
    print_board(board)
    print(f"Player {current_player} wins!")
    game_over = True
elif all(cell != " " for row in board for cell in row):
    print_board(board)
    print("It's a draw!")
    game_over = True
else:
```

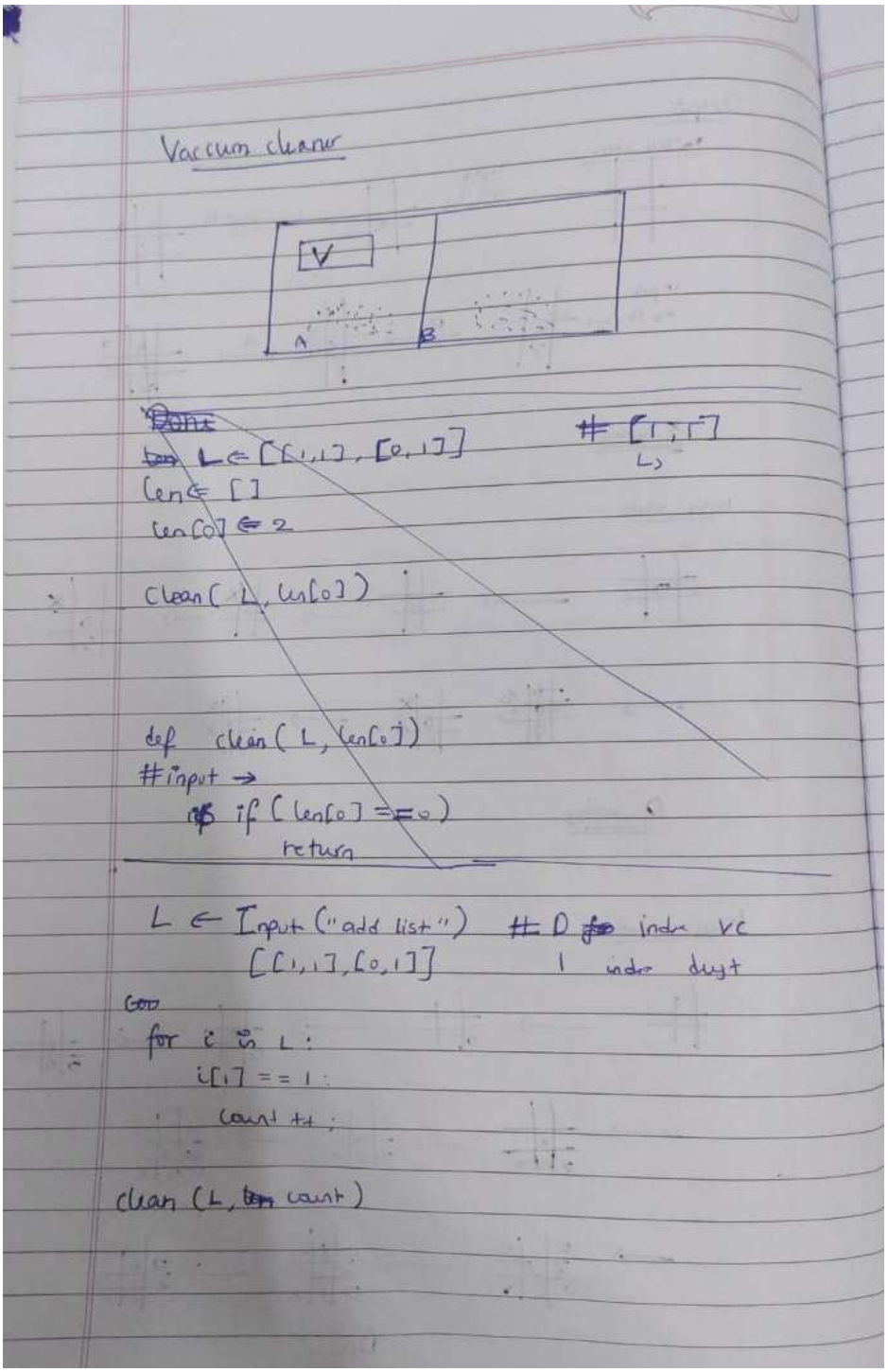
```
            current_player = "O" if current_player == "X" else "X"


if __name__ == "__main__":
    play_game()
```

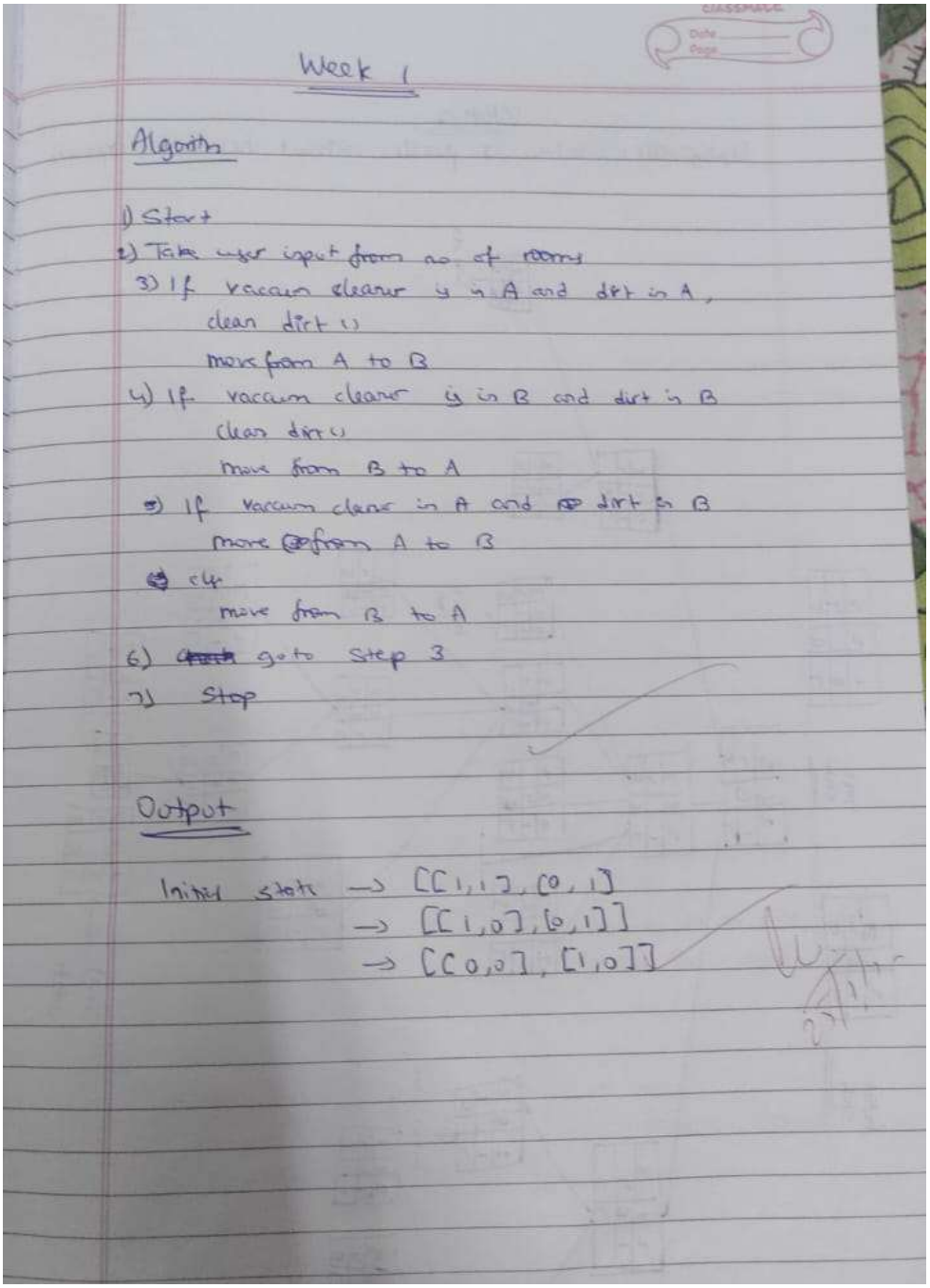
Output:

```
SANATH S SHETTY 1BM23CS297
  |   |
---------
  |   |
---------
  |   |
---------
Player X's turn.
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 0
X |   |
---------
  |   |
---------
  |   |
---------
Player O's turn.
Enter row (0, 1, 2): 2
Enter column (0, 1, 2): 2
X |   |
---------
  |   |
---------
  |   | O
---------
Player X's turn.
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 1
X | X |
---------
  |   |
---------
  |   | O
---------
Player O's turn.
Enter row (0, 1, 2): 1
Enter column (0, 1, 2): 1
X | X |
---------
  | O |
---------
  |   | O
---------
Player X's turn.
Enter row (0, 1, 2): 0
Enter column (0, 1, 2): 2
X | X | X
---------
  | O |
---------
  |   | O
---------
Player X wins!
```

Implement vacuum cleaner agent

Observation:





Code:

```
L = []
L = [[1, 1], [0, 1]]
print('Sanath 1BM23CS297')
print(L)
```

```
count = 0
for i in range(len(L)):
    if (L[i][1] == 1):
        count = count + 1

def clean(L, count, index):
    if (count == 0):
        return

    if (index < 0 or index >= len(L)):
        return

    if (L[index][1] == 1):

        L[index][0] = 1
        L[index][1] = 0
        count = count - 1
        print(L)
        L[index][0] = 0

        clean(L, count, index - 1)
        clean(L, count, index + 1)
    else:
        clean(L, count, index + 1)
```

```
        clean(L, count, index - 1)


clean(L, count, 0)
```

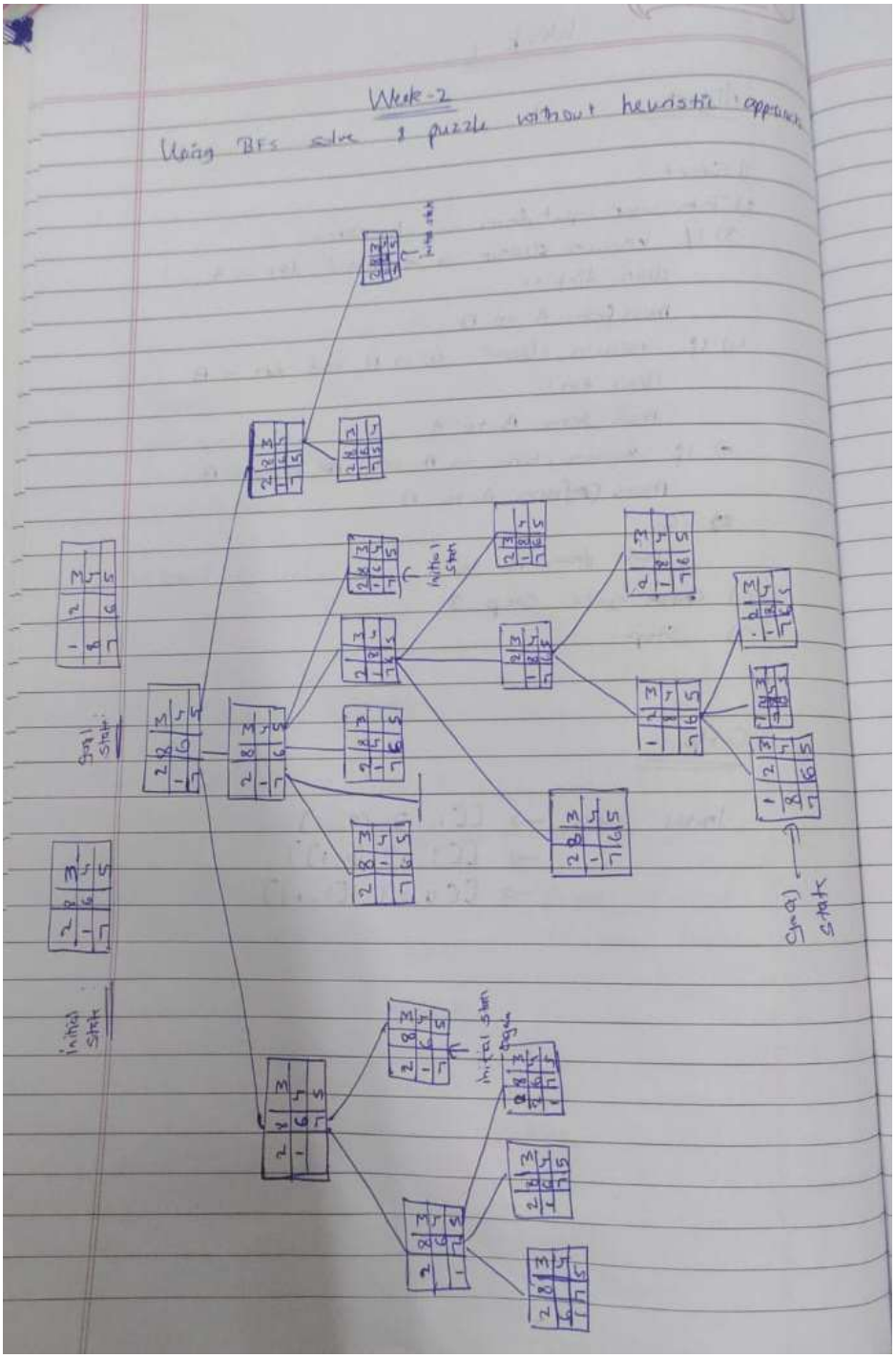
Output:

```
Sanath 1BM23CS297
[[1, 1], [0, 1]]
[[1, 0], [0, 1]]
[[0, 0], [1, 0]]
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Observation:

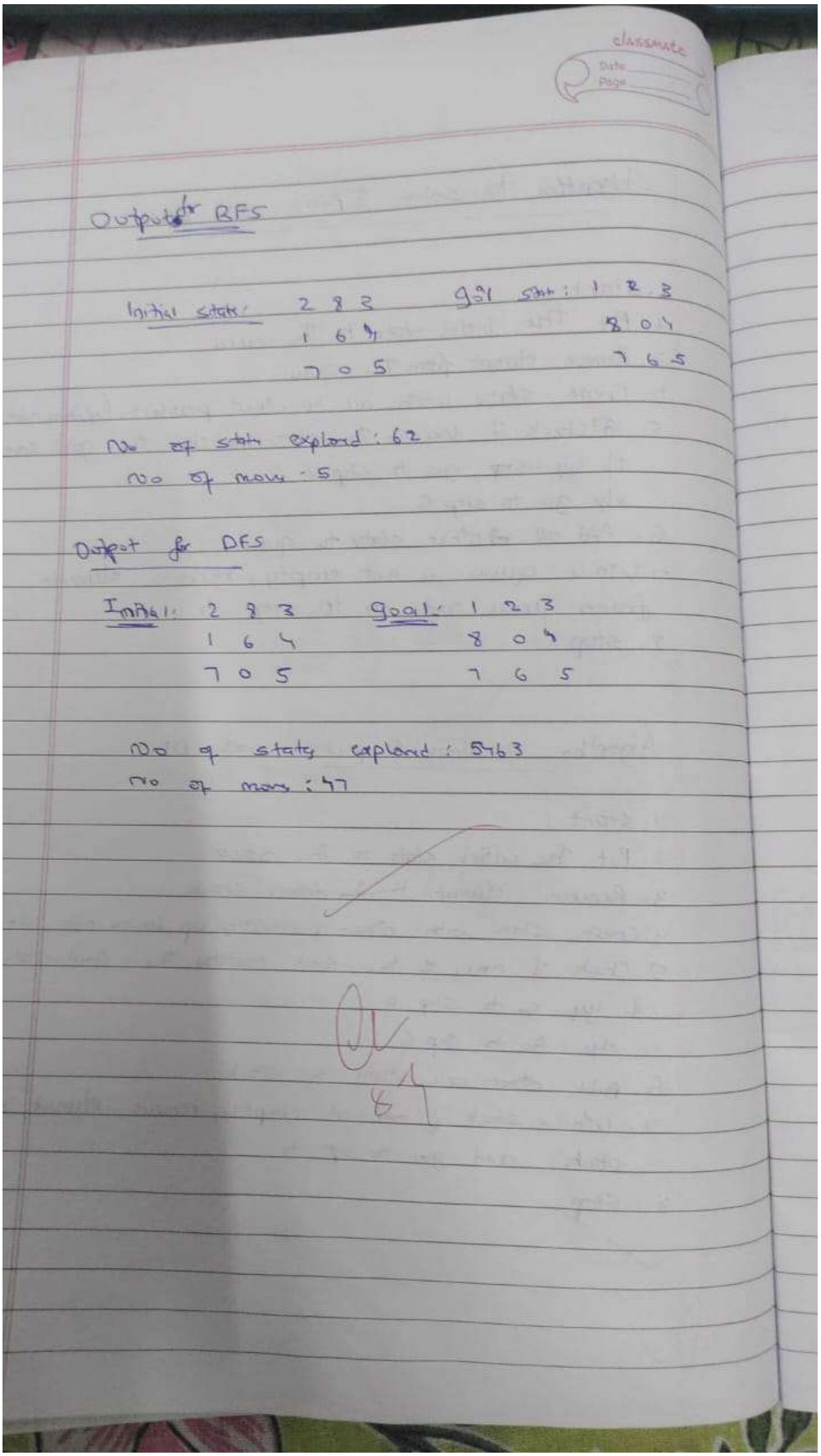


Algorithm to solve 8 Puzzle with BFS

1. Start
2. Put the initial state to the queue.
3. Remove element from the queue
4. Create states with all 4 other possibilities (up, down, right, left).
5. Check if any of the state matches the goal state
   If yes stop, go to step 8
   else go to step 6
6. Add all other states to queue
7. While queue is not empty, remove element from queue and go to step 4
8. Stop.

Algorithm to solve 8 puzzle with DFS

1. Start
2. Put the initial state to the stack.
3. Remove element from stack.
4. Create state with other possibilities (up, down, right, left)
5. Check if any of the state matches the goal state.
   If yes go to step 8
   else go to step 6
6. Add all states to stack.
7. While stack is not empty, remove element from stack and go to step 4
8. Stop



Output for BFS

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

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

No of state explored: 62
no of move: 5

Output for DFS

Initial: 2 8 3
1 6 4
7 0 5

goal: 1 2 3
8 0 4
7 6 5

No of state explored: 5763
no of move: 47

Code:

```
print("Sanath S Shetty")
print("1BM23CS297")

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

# 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("Number of moves:", 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

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)
```

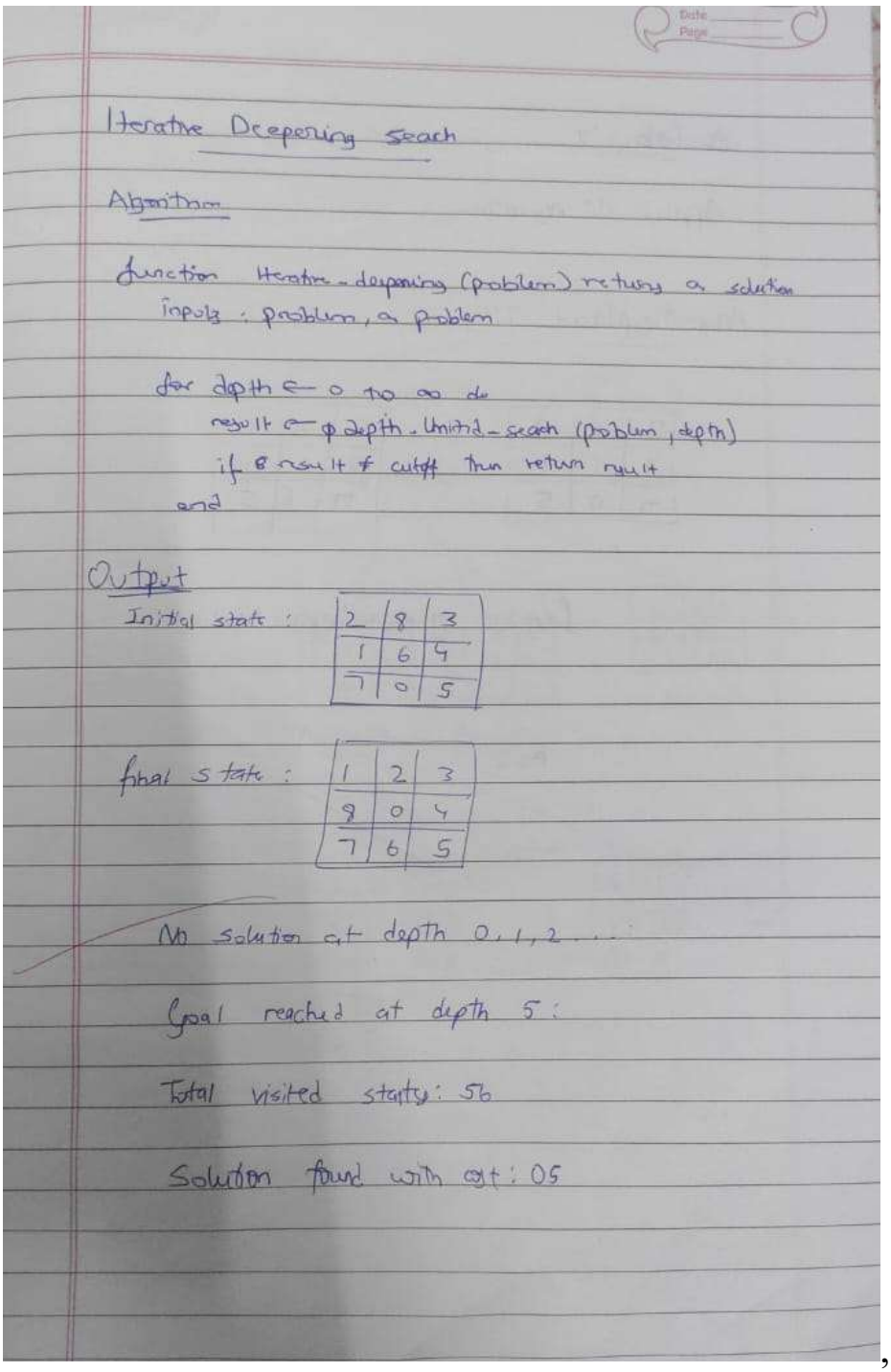
Output:

```
Sanath S Shetty
1BM23CS297
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

Number of states explored: 5463
Number of moves: 47

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

0 2 3
1 8 4
```

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

2 3 4
1 6 5
8 7 0

2 3 4
1 6 0
8 7 5

2 3 0
1 6 4
8 7 5

2 0 3
1 6 4
8 7 5

0 2 3
1 6 4
8 7 5

1 2 3
0 6 4
8 7 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
```

Implement Iterative deepening search algorithm

Observation:



Iterative Deepening Search

Algorithm

function Iterative-deepening (problem) returns a solution
inputs : problem, a problem

for depth ← 0 to ∞ do
result ← depth-limited-search (problem, depth)
if result ≠ cutoff then return result
end

Output

Initial state :

2	8	3
1	6	4
7	0	5

final state :

1	2	3
8	0	4
7	6	5

No solution at depth 0, 1, 2 ...

Goal reached at depth 5 :

Total visited states : 56

Solution found with cost : 05

,

Code:

```
print("Sanath Shetty")
print("1BM23CS297")



MAX_VISITED_DISPLAY = 10
NUM_INTERMEDIATE_STATES = 3
MAX_DEPTH_LIMIT = 50

def print_state(state):
    for row in state:
        print(' '.join(str(x) for x in row))
    print()

def is_goal(state, goal_state):
    return state == goal_state

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
```

```
def get_neighbors(state):
    neighbors = []
    x, y = find_zero(state)
    directions = [(1,0), (-1,0), (0,1), (0,-1)]
    for dx, dy in directions:
        new_x, new_y = x + dx, y + dy
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[new_x][new_y] = new_state[new_x][new_y],
new_state[x][y]
            neighbors.append(new_state)
    return neighbors


def is_solvable(state):
    flat = [num for row in state for num in row if num != 0]
    inv_count = 0
    for i in range(len(flat)):
        for j in range(i + 1, len(flat)):
            if flat[i] > flat[j]:
                inv_count += 1
    return inv_count % 2 == 0


def dls(current_state, goal_state, depth_limit, path, visited, visited_states_display):
    """
```

```
    Depth-Limited Search helper for IDDFS.
    Returns:
        path if goal found else None
    """
    if len(path) - 1 > depth_limit:
        return None

    visited_states_display.append(current_state)
    if len(visited_states_display) <= MAX_VISITED_DISPLAY:
        print(f"Visited state #{len(visited_states_display)}:")
        print_state(current_state)

    if is_goal(current_state, goal_state):
        return path

    for neighbor in reversed(get_neighbors(current_state)):
        neighbor_tuple = tuple(tuple(row) for row in neighbor)
        if neighbor_tuple not in visited:
            visited.add(neighbor_tuple)
            result = dls(neighbor, goal_state, depth_limit, path + [neighbor], visited,
visited_states_display)
            if result is not None:
                return result
            # Backtrack visited set for other paths:
```

```
            visited.remove(neighbor_tuple)
    return None


def iddfs(start_state, goal_state, max_depth=MAX_DEPTH_LIMIT):
    """
    Iterative Deepening DFS:
    Tries DFS with increasing depth limits until goal found or max depth exceeded.
    """
    print("Starting Iterative Deepening DFS traversal...\n")


    for depth in range(max_depth + 1):
        print(f"Trying depth limit: {depth}")
        visited_states_display = []
        visited = set()
        visited.add(tuple(tuple(row) for row in start_state))
        path = dls(start_state, goal_state, depth, [start_state], visited,
visited_states_display)
        if path is not None:
            print(f"\nGoal reached at depth {depth}!")
            print(f"Total visited states in last iteration: {len(visited_states_display)}")
            return path
        print(f"No solution found at depth {depth}\n")
    print(f"No solution found within max depth limit {max_depth}")
    return None
```

```
def read_state(name):
    print(f"Enter the {name} state, row by row (use space-separated numbers, 0 for empty):")
    state = []
    for _ in range(3):
        row = input().strip().split()
        if len(row) != 3:
            raise ValueError("Each row must have exactly 3 numbers.")
        row = list(map(int, row))
        state.append(row)
    return state

# --- Main Execution ---

initial_state = read_state("initial")
goal_state = read_state("goal")

if not (is_solvable(initial_state) == is_solvable(goal_state)):
    print("The puzzle is unsolvable.")
    exit()

solution_path = iddfs(initial_state, goal_state)
```

```
if solution_path:
    cost = len(solution_path) - 1
    print(f"\nSolution found with cost: {cost}\n")
    print("Solution path:")

    total_steps = len(solution_path) - 1  # number of moves
    print("Initial State:")
    print_state(solution_path[0])

    if total_steps > 1:
        step_indices = list(range(1, total_steps))
        if len(step_indices) > NUM_INTERMEDIATE_STATES:
            interval = len(step_indices) // (NUM_INTERMEDIATE_STATES + 1)
            selected_indices = [step_indices[i * interval] for i in range(1,
NUM_INTERMEDIATE_STATES + 1)]
        else:
            selected_indices = step_indices

        for idx in selected_indices:
            print(f"Intermediate State (Step {idx}):")
            print_state(solution_path[idx])

    print("Final State:")
    print_state(solution_path[-1])
```

```
else:
    print("No solution found")
```

Output:

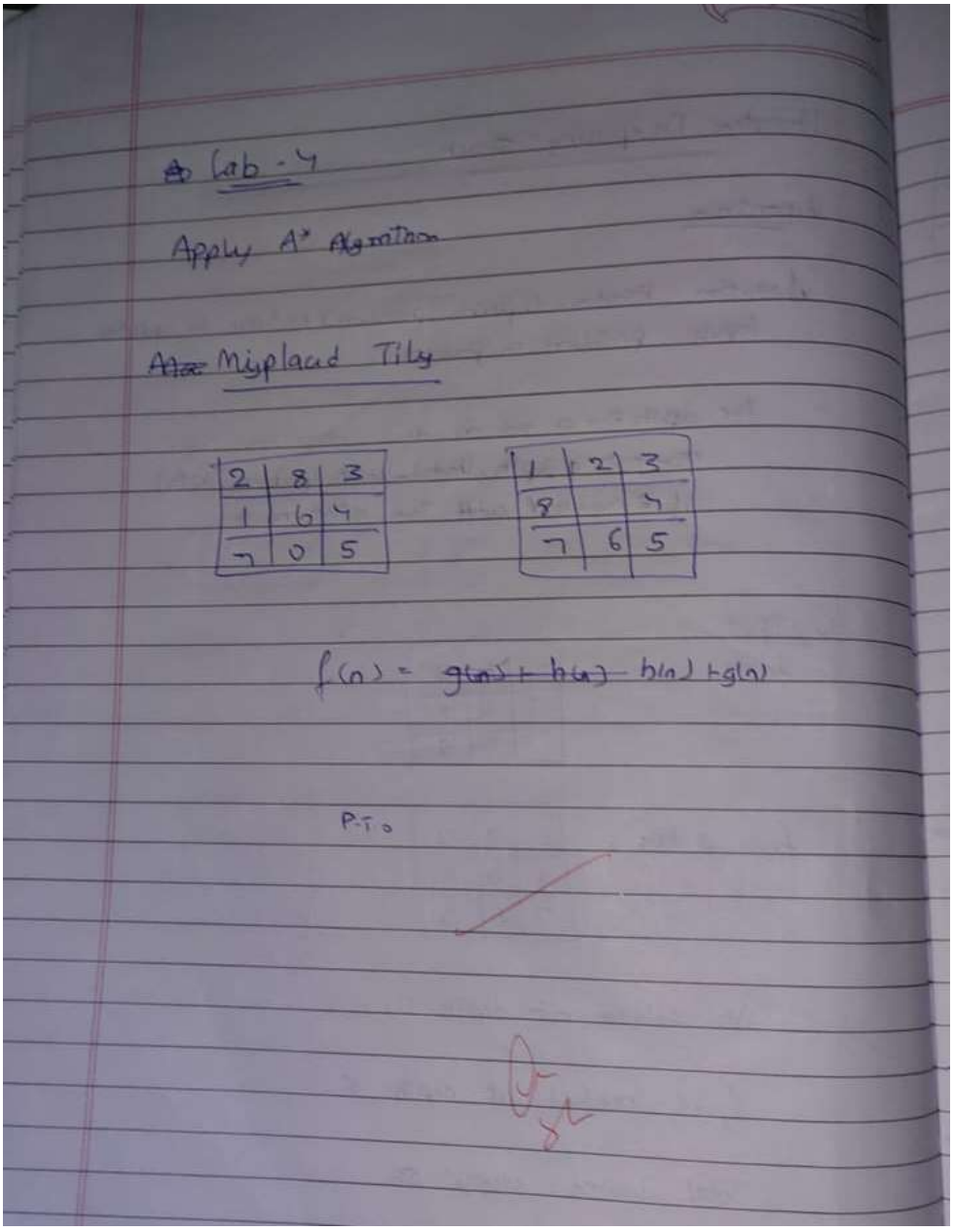
```
Sanath Shetty
1BM23CS297
Enter the initial state, row by row (use space-separated numbers, 0 for empty):
2 8 3
1 6 4
7 0 5
Enter the goal state, row by row (use space-separated numbers, 0 for empty):
1 2 3
8 0 4
7 6 5
Starting Iterative Deepening DFS traversal...

Trying depth limit: 0
Visited state #1:
2 8 3
1 6 4
7 0 5

No solution found at depth 0

Trying depth limit: 1
Visited state #1:
2 8 3
1 6 4
7 0 5

Visited state #2:
2 8 3
1 6 4
0 7 5

Visited state #3:
2 8 3
1 6 4
7 5 0

Visited state #4:
2 8 3
1 0 4
7 6 5
```

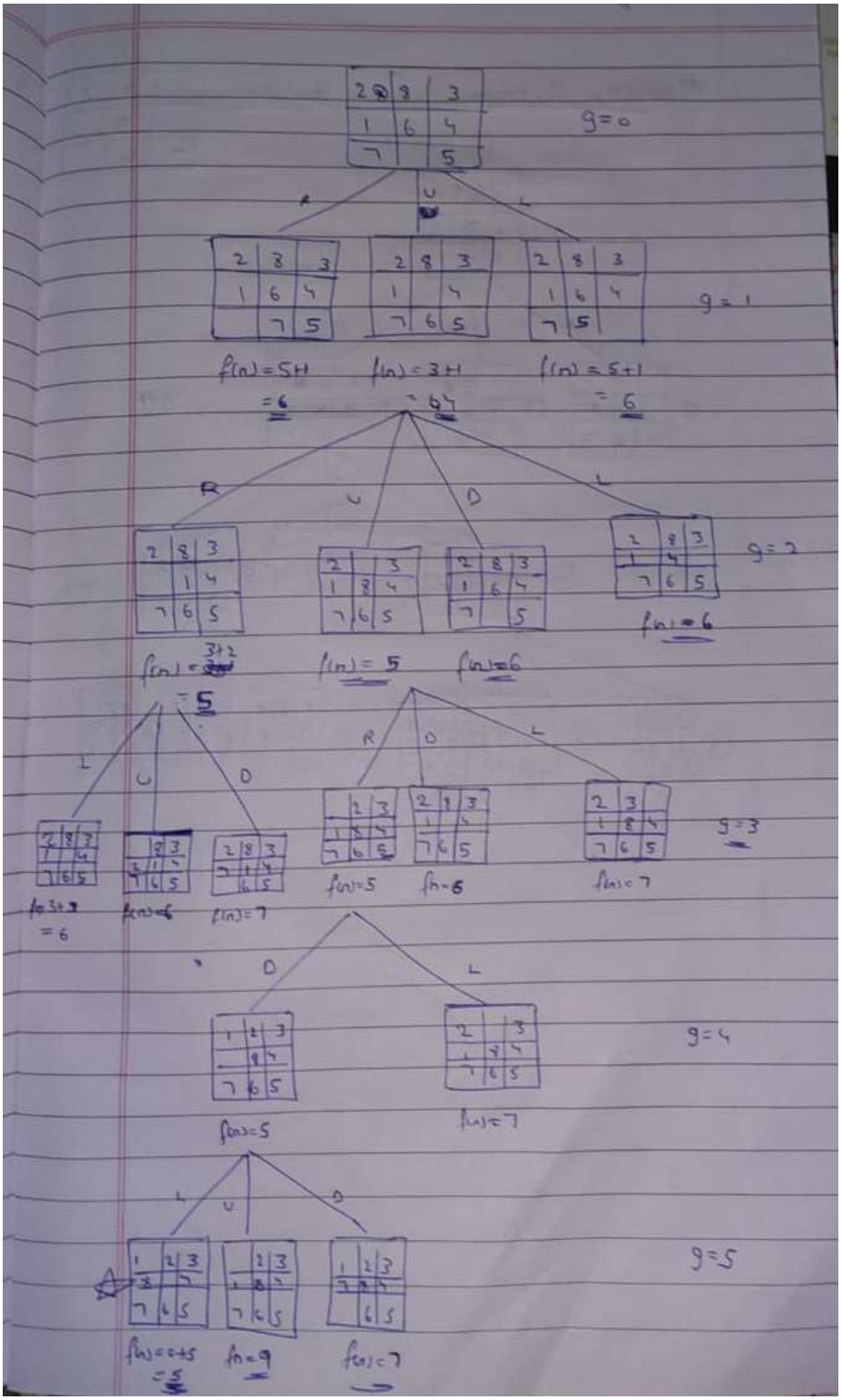
```
Goal reached at depth 5!
Total visited states in last iteration: 56

Solution found with cost: 5

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

Intermediate State (Step 2):
2 0 3
1 8 4
7 6 5

Intermediate State (Step 3):
0 2 3
1 8 4
7 6 5

Intermediate State (Step 4):
1 2 3
0 8 4
7 6 5

Final State:
1 2 3
8 0 4
7 6 5
```

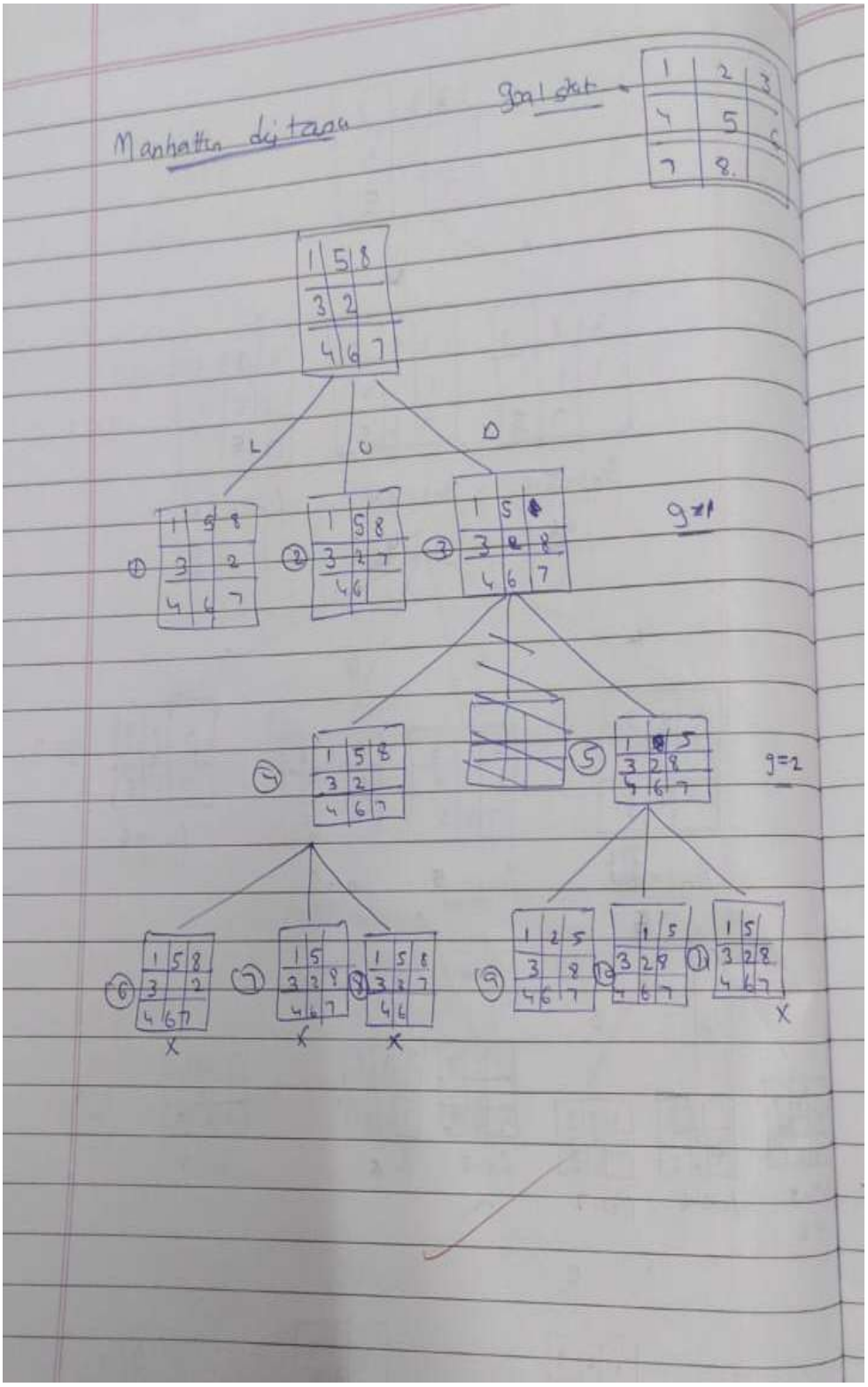
Program 3

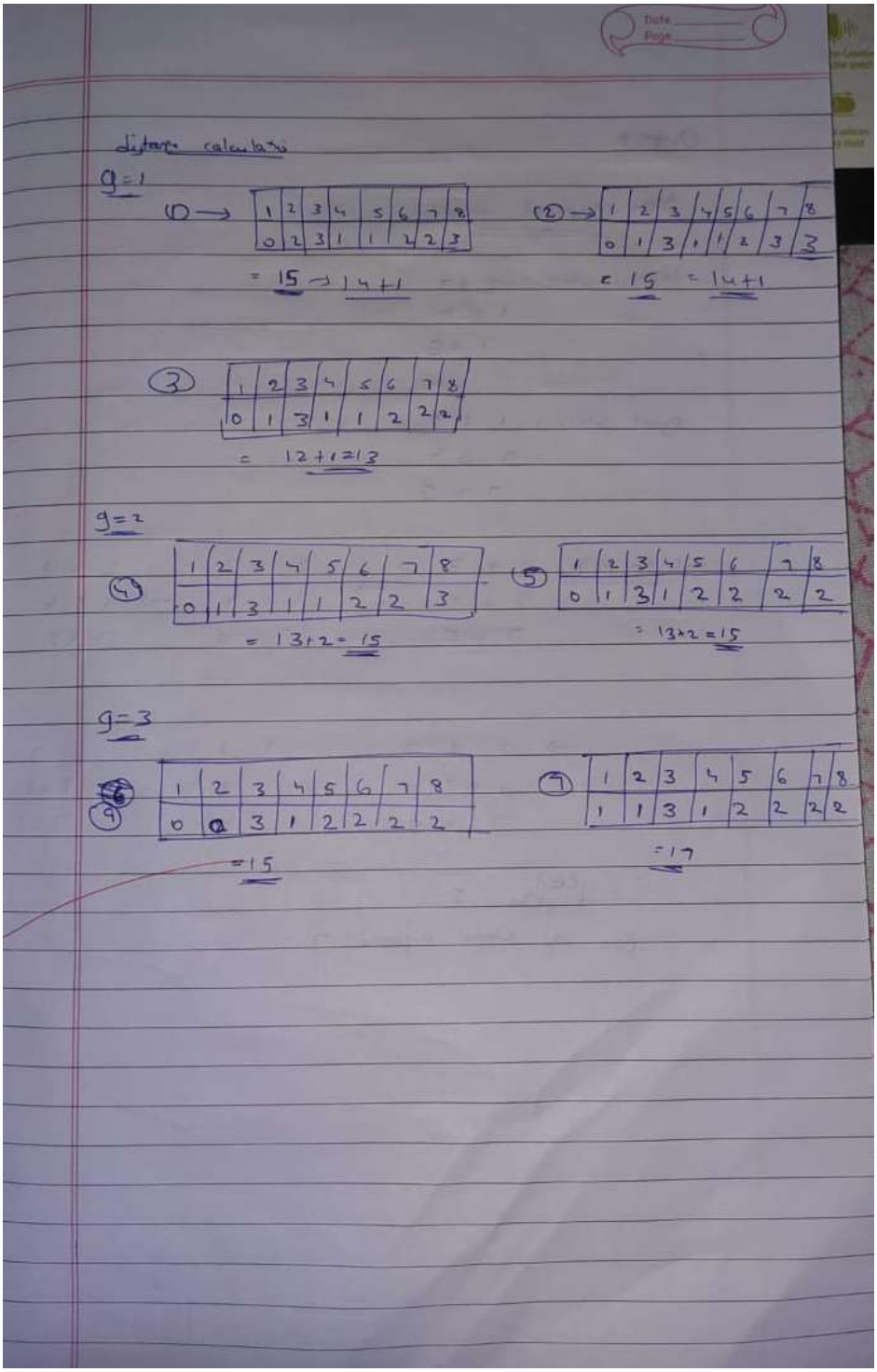
Implement A* search algorithm

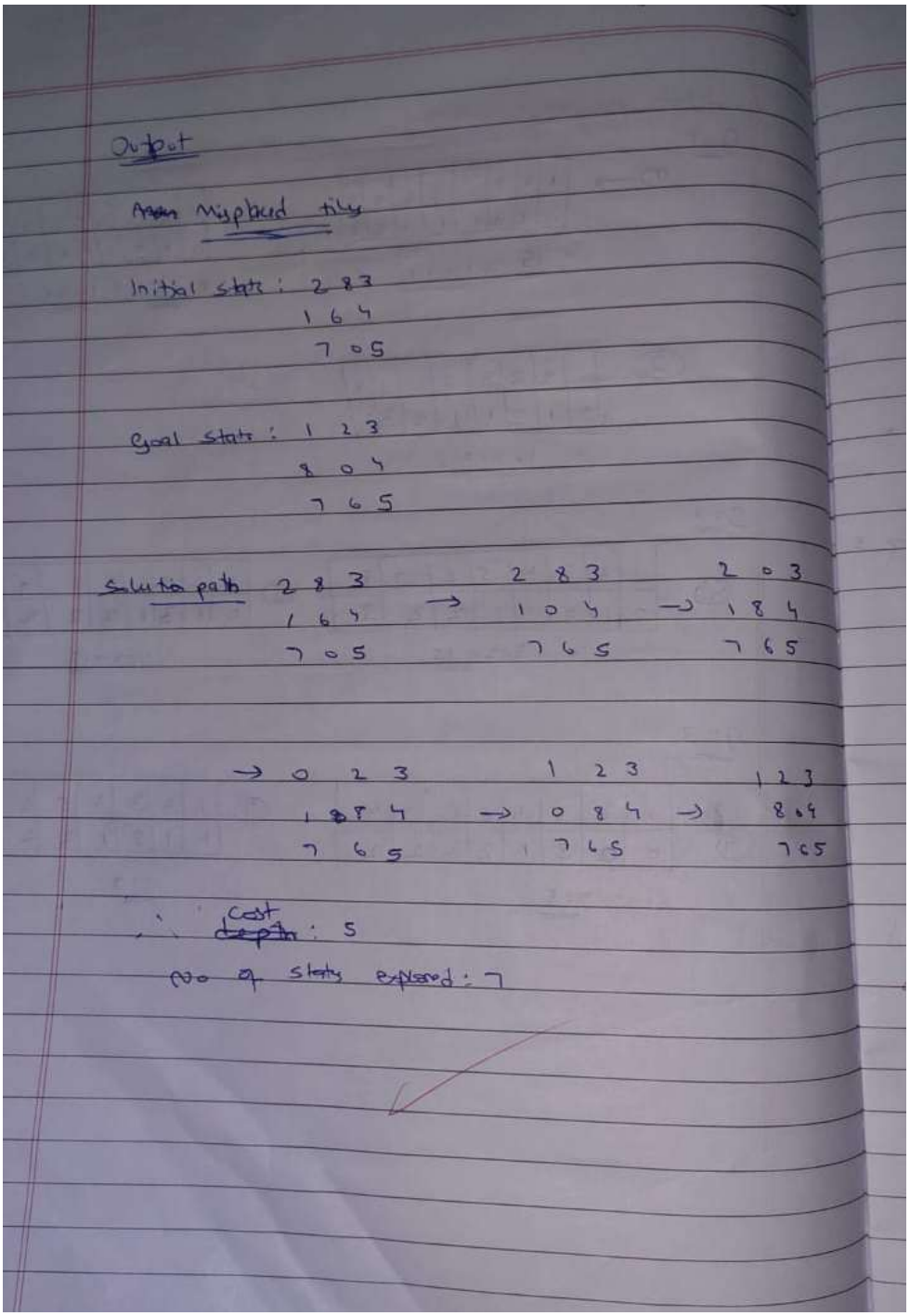
Observation:




g=0
g=1
f(n)=5+1 =6
f(n)=3+1 =6
f(n)=5+1 =6
g=2
f(n)=3+2 =5
f(n)=5
f(n)=6
g=3
f(n)=5
f(n)=6
f(n)=7
f(n)=3+3 =6
f(n)=6
f(n)=7
g=4
f(n)=5
f(n)=7
g=5
f(n)=6+5 =5
f(n)=9
f(n)=7


Manhattan distance
Goal state
L
U
D
g=1
g=2
X
X
X
X


distance calculation
g=1
(1) → 1 2 3 4 5 6 7 8
0 2 3 1 1 4 2 3
= 15 → 14+1
(2) → 1 2 3 4 5 6 7 8
0 1 3 1 1 2 3 3
= 15 = 14+1
(3) 1 2 3 4 5 6 7 8
0 1 3 1 1 2 2 2
= 12+1=13
g=2
(4) 1 2 3 4 5 6 7 8
0 1 3 1 1 2 2 3
= 13+2= 15
(5) 1 2 3 4 5 6 7 8
0 1 3 1 2 2 2 2
= 13+2=15
g=3
(6) (8) 1 2 3 4 5 6 7 8
0 a 3 1 2 2 2 2
=15
(7) 1 2 3 4 5 6 7 8
1 1 3 1 2 2 2 2
=17



Output

Misplaced tile

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

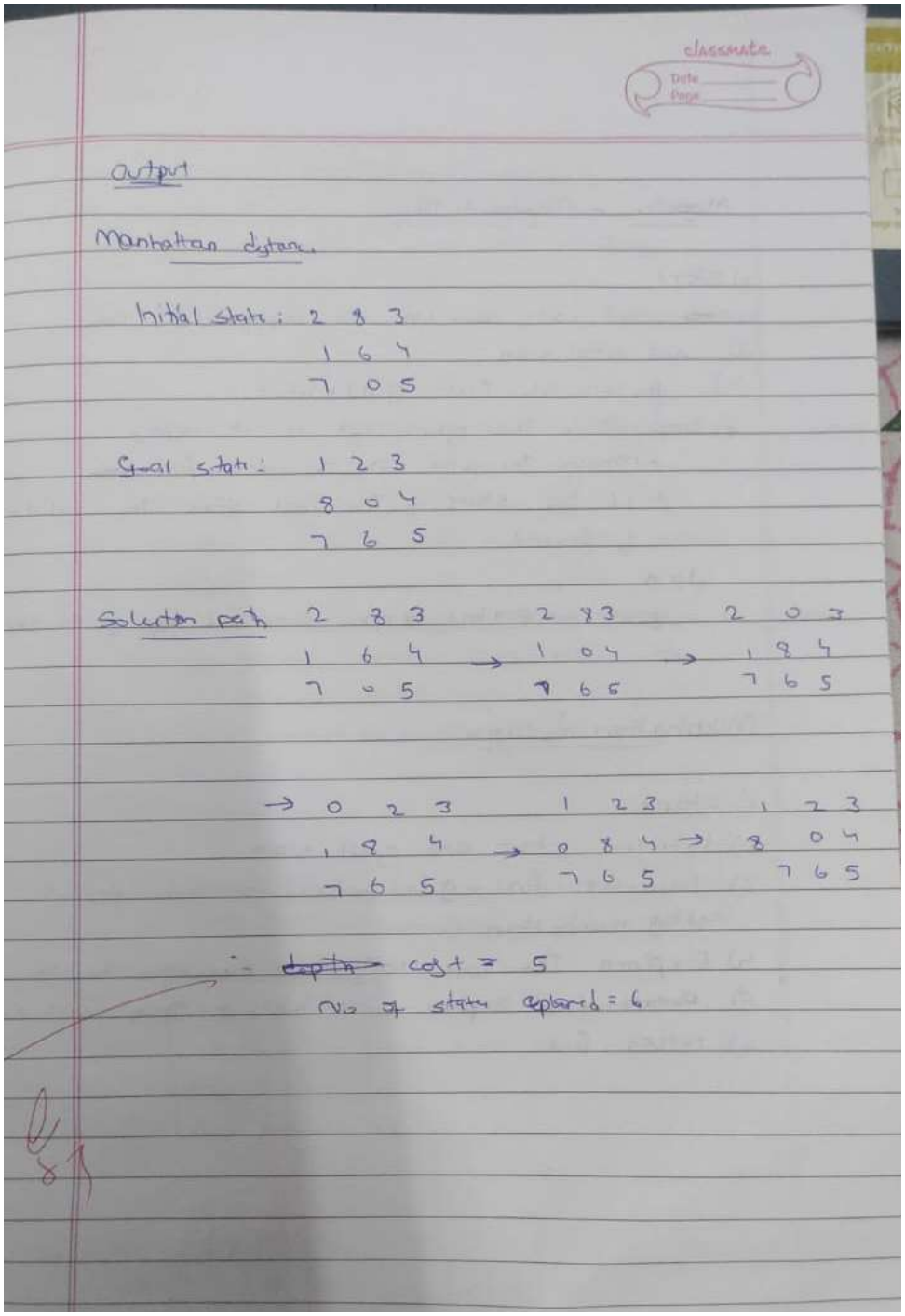
Goal state : 1 2 3
8 0 4
7 6 5

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

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

∴ cost : 5

No of states explored : 7



Output

Manhattan distance.

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

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

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

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

∴ ~~depth~~ cost = 5
No of states explored = 6

Algorithm - Misplaced tiles

1) Start
2) start with open list and empty closed list.
3) Add initial state
4) fn calculate $f(n) = g(n) + h(n)$.
5) loop, while the open list is not empty:
   - remove the state with the lowest $f(n)$
   - If this state is the goal state, then solution is found
   else
   generate all possible other states and go to step

Manhattan distance

1) Start
2) Initialize start and goal state
3) Calculate $f(n) = g(n) + h(n)$ for each possible using manhattan
4) Explore the node with least manhattan heuristic
5) Return find Explore until $h(n) = 0$ then $f(n) = g(n)$
6) return $f(n)$

Code:

```
import heapq

print("Sanath S Shetty")
print("1BM23CS297")

def print_state(state):
    """Prints the 8-puzzle state in a 3x3 grid."""
    for i in range(0, 9, 3):
        print(" ".join(str(x) for x in state[i:i+3]))
    print()

def misplaced_tiles(current_state, goal_state):
    """Calculates the number of misplaced tiles heuristic."""
    misplaced = 0
    for i in range(9):
        if current_state[i] != 0 and current_state[i] != goal_state[i]:
            misplaced += 1
    return misplaced

def find_zero(state):
    """Finds the index of the blank tile (0)."""
    return state.index(0)
```

```
def get_neighbors(state):
    """Generates possible next states (neighbors) from the current state."""
    neighbors = []
    zero_index = find_zero(state)
    row, col = divmod(zero_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
            new_state_list = list(state)
            new_state_list[zero_index], new_state_list[new_index] =
new_state_list[new_index], new_state_list[zero_index]
            neighbors.append(tuple(new_state_list))
    return neighbors

def astar_misplaced(start_state, goal_state):
    """Solves the 8-puzzle using A* with the misplaced tiles heuristic."""
    open_set = []
    heapq.heappush(open_set, (misplaced_tiles(start_state, goal_state), 0,
start_state, [])) # (f_cost, g_cost, state, path)
    visited = set()
    visited.add(start_state)
```

```
    state_count = 0

    while open_set:
        f_cost, g_cost, current_state, path = heapq.heappop(open_set)
        state_count += 1

        if current_state == goal_state:
            print("\nGoal reached!")
            print("Number of states explored:", state_count)
            print("depth:", g_cost)
            print("\nSolution path:")
            for s in path + [current_state]:
                print_state(s)
            return path

        for neighbor in get_neighbors(current_state):
            if neighbor not in visited:
                visited.add(neighbor)
                new_g_cost = g_cost + 1
                new_f_cost = new_g_cost + misplaced_tiles(neighbor, goal_state)
                heapq.heappush(open_set, (new_f_cost, new_g_cost, neighbor, path +
[current_state]))

    print("No solution found.")
```

```
    return None

# --- Main Execution ---
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(map(int, 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(map(int, row))

start_state = tuple(initial_board)
goal_state = tuple(goal_board)

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

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

astar_misplaced(start_state, goal_state)

Output:

```
Sanath S Shetty
1BM23CS297
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: 7
depth: 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

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

[(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)]
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Observation:

Program-6 Week-5

4 queens problem using hill climbing.

Hill climbing search algorithm.

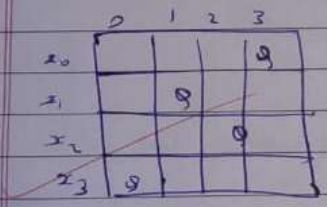
function hc(problem) returns a state that is local max

current ← make-node(problem, initial state)
loop do
neighbour ← a highest valued successor of current
if neighbour.value ≤ current.value then
return current.state
current ← neighbour

State space

Initial space

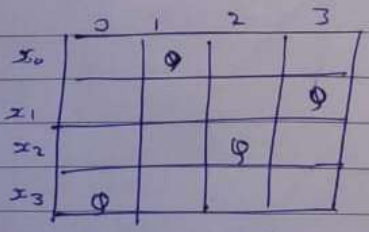
1)



cost = 2

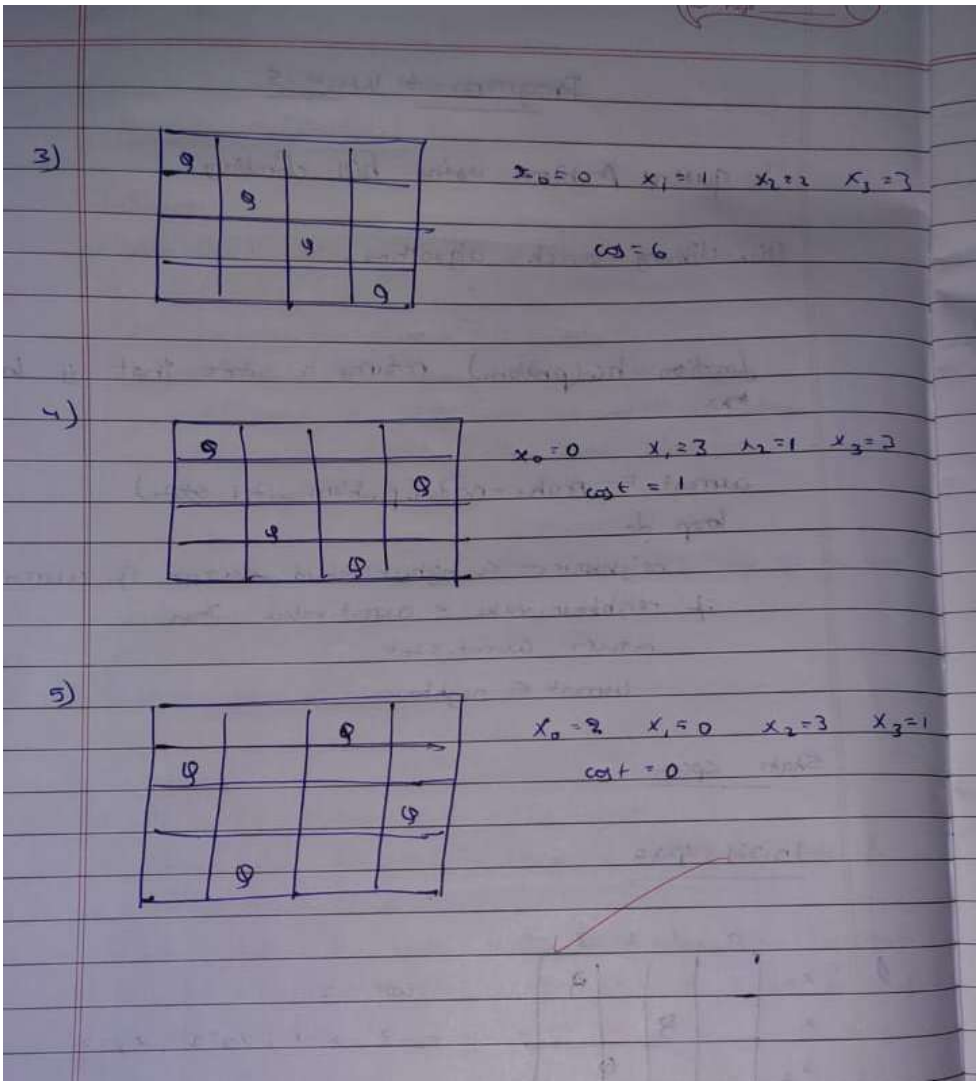
$x_0=3$, $x_1=1$, $x_2=2$, $x_3=0$

2)



cost = 1

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



Code:

```
print("Sanath S Shetty")
print("1BM23CS297")


def print_board(state):
    n = len(state)
    for row in range(n):
        line = ""
```

```
        for col in range(n):
            if state[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

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]) == j - i:
                cost += 1
    return cost

def get_neighbors(state):
    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]
            neighbors.append((neighbor, (i, j)))
```

```
    return neighbors

def hill_climbing(initial_state):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    print("Start State:")
    print_board(current_state)
    print(f"Cost: {current_cost}\n")

    path = [(current_state, current_cost, None)]

    while True:
        neighbors = get_neighbors(current_state)
        neighbor_costs = [(tuple(neighbor), calculate_cost(neighbor), swap) for
neighbor, swap in
neighbors]

        neighbor_costs.sort(key=lambda x: (x[1], x[2]))

        best_neighbor, best_cost, best_swap = neighbor_costs[0]

        print(f"Neighbors of {current_state} with costs:")
        for neighbor, cost, swap in neighbor_costs:
            print(f"Swap columns {swap}:")
            print_board(neighbor)
            print(f"Cost: {cost}\n")
```

```
        if best_cost < current_cost:
            print(f"Moving to better neighbor by swapping columns {best_swap}:")
            print_board(best_neighbor)
            print(f"Cost: {best_cost}\n")
            current_state, current_cost = best_neighbor, best_cost
            path.append((current_state, current_cost, best_swap))
        else:
            print("/nReached goal state.")
            break

    return path


def get_initial_state():
    print("Enter the initial positions of the 4 queens (row for each column, 0-
indexed):")
    positions = []
    for col in range(4):
        while True:
            try:
                pos = int(input(f"Column {col}: "))
                if 0 <= pos < 4:
                    positions.append(pos)
                    break
                else:
                    print("Invalid input. Enter a number between 0 and 3.")
```

```
            except ValueError:
                print("Invalid input. Please enter an integer.")
    return tuple(positions)


initial_state = get_initial_state()
path = hill_climbing(initial_state)

print("Final path:")
for i, (state, cost, swap) in enumerate(path):
    print(f"Step {i}:")
    print_board(state)
    print(f"Cost: {cost}")
    if swap is not None:
        print(f"Swap columns: {swap}")
    print("--------------------")
```

Output:

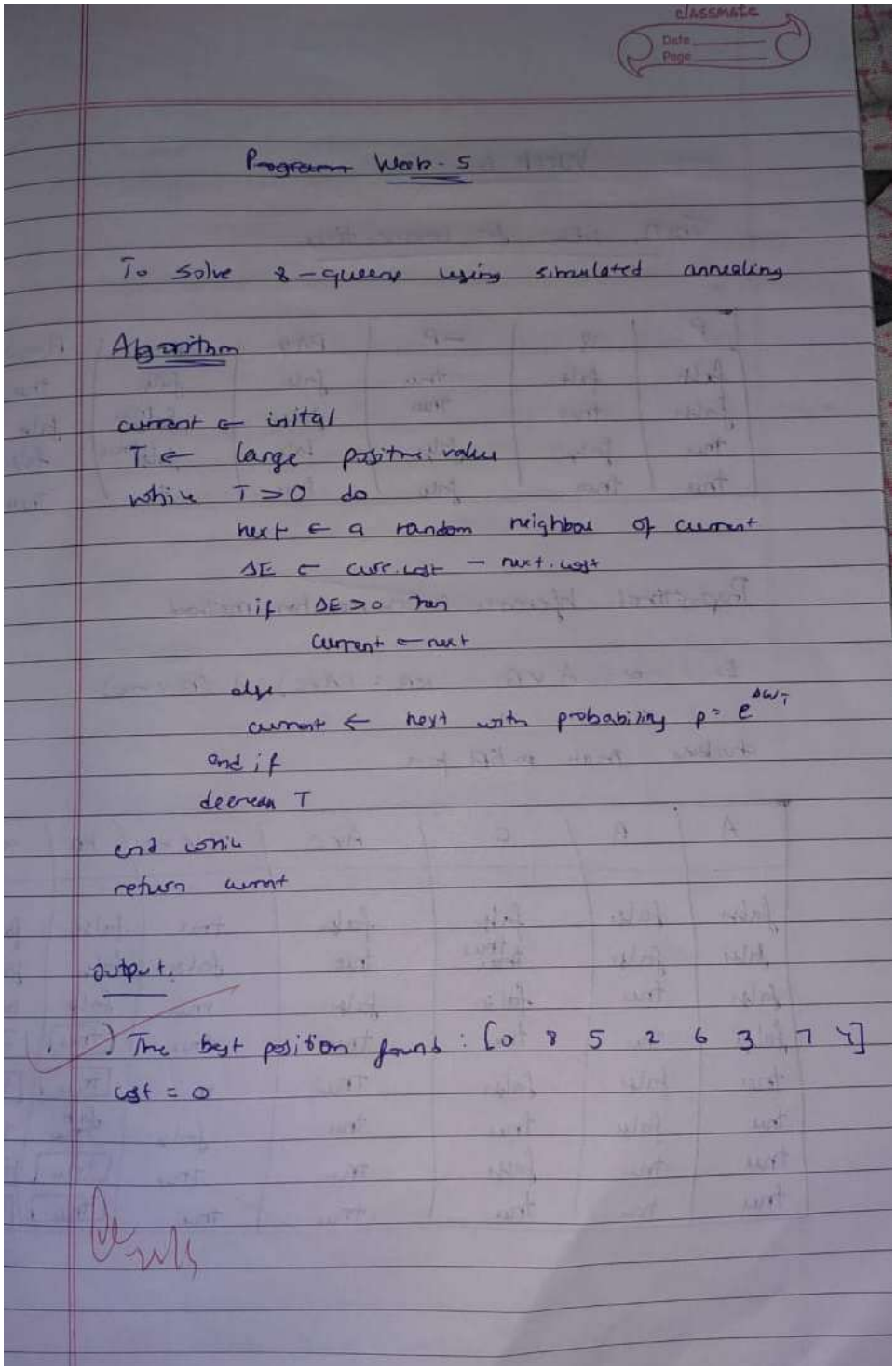
```
Sanath S Shetty
1BM23CS297
Enter the initial positions of the 4 queens (row for each column, 0-indexed):
Column 0: 3
Column 1: 2
Column 2: 1
Column 3: 0
Start State:
. . . Q
. . Q .
. Q . .
Q . . .

Cost: 6

Neighbors of (3, 2, 1, 0) with costs:
Swap columns (0, 1):
. . . Q
. . Q .
Q . . .
. Q . .

Cost: 2

Swap columns (0, 3):
Q . . .
. . Q .
. Q . .
. . . Q

Cost: 2

Swap columns (1, 2):
. . . Q
. Q . .
. . Q .
Q . . .

Cost: 2
```

```
Cost: 4

Swap columns (1, 2):
. Q . .
Q . . .
. . . Q
. . Q .

Cost: 4

/nReached goal state.
Final path:
Step 0:
. . . Q
. . Q .
. Q . .
Q . . .

Cost: 6
--------------------
Step 1:
. . . Q
. . Q .
Q . . .
. Q . .

Cost: 2
Swap columns: (0, 1)
--------------------
Step 2:
. . . Q
Q . . .
. . Q .
. Q . .

Cost: 1
Swap columns: (0, 2)
--------------------
Step 3:
. . Q .
Q . . .
. . . Q
. Q . .

Cost: 0
Swap columns: (2, 3)
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Observaion:



Program Work-5

To solve 8-queens using simulated annealing

Algorithm

current ← inital
T ← large positive value
while T > 0 do
    next ← a random neighbor of current
    ΔE ← curr.cost − next.cost
    if ΔE > 0 then
        current ← next
    else
        current ← next with probability $p = e^{\Delta E/T}$
    end if
    decrease T
end while
return current

output:

The best position found: [0 4 7 5 2 6 1 3]
cost = 0

Code:

```
print("Sanath S Shetty")
print("1BM23CS297")
import random
import math

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

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]) == j - i:
```

```
            cost += 1
    return cost


def get_neighbor(state):
    n = len(state)
    neighbor = list(state)
    i, j = random.sample(range(n), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return tuple(neighbor), (i, j)


def simulated_annealing(initial_state, initial_temp=1000, cooling_rate=0.95,
min_temp=1e-3, max_iter=1000):
    current_state = initial_state
    current_cost = calculate_cost(current_state)
    temperature = initial_temp
    path = [(current_state, current_cost, None)]

    print("Initial State:")
    print_board(current_state)
    print(f"Cost: {current_cost}\n")

    iteration = 0
    while temperature > min_temp and current_cost > 0 and iteration < max_iter:
        neighbor, swap = get_neighbor(current_state)
```

```
        neighbor_cost = calculate_cost(neighbor)

        cost_diff = neighbor_cost - current_cost

        if cost_diff < 0 or math.exp(-cost_diff / temperature) > random.random():
            current_state, current_cost = neighbor, neighbor_cost
            path.append((current_state, current_cost, swap))
            print(f"Iteration {iteration}: Swap columns {swap}")
            print_board(current_state)
            print(f"Cost: {current_cost}, Temperature: {temperature:.4f}\n")

        temperature *= cooling_rate
        iteration += 1

    print("Terminated.")
    return path

def get_initial_state():
    print("Enter the initial positions of the 4 queens (row for each column, 0-
indexed):")
    positions = []
    for col in range(4):
        while True:
            try:
```

```
            pos = int(input(f'Column {col}: '))
            if 0 <= pos < 4:
                positions.append(pos)
                break
            else:
                print("Invalid input. Enter a number between 0 and 3.")
        except ValueError:
            print("Invalid input. Please enter an integer.")
    return tuple(positions)

initial_state = get_initial_state()
solution_path = simulated_annealing(initial_state)

print("Final path:")
for i, (state, cost, swap) in enumerate(solution_path):
    print(f'Step {i}:')
    print_board(state)
    print(f'Cost: {cost}')
    if swap is not None:
        print(f'Swap columns: {swap}')
    print("--------------------")
```

Output:

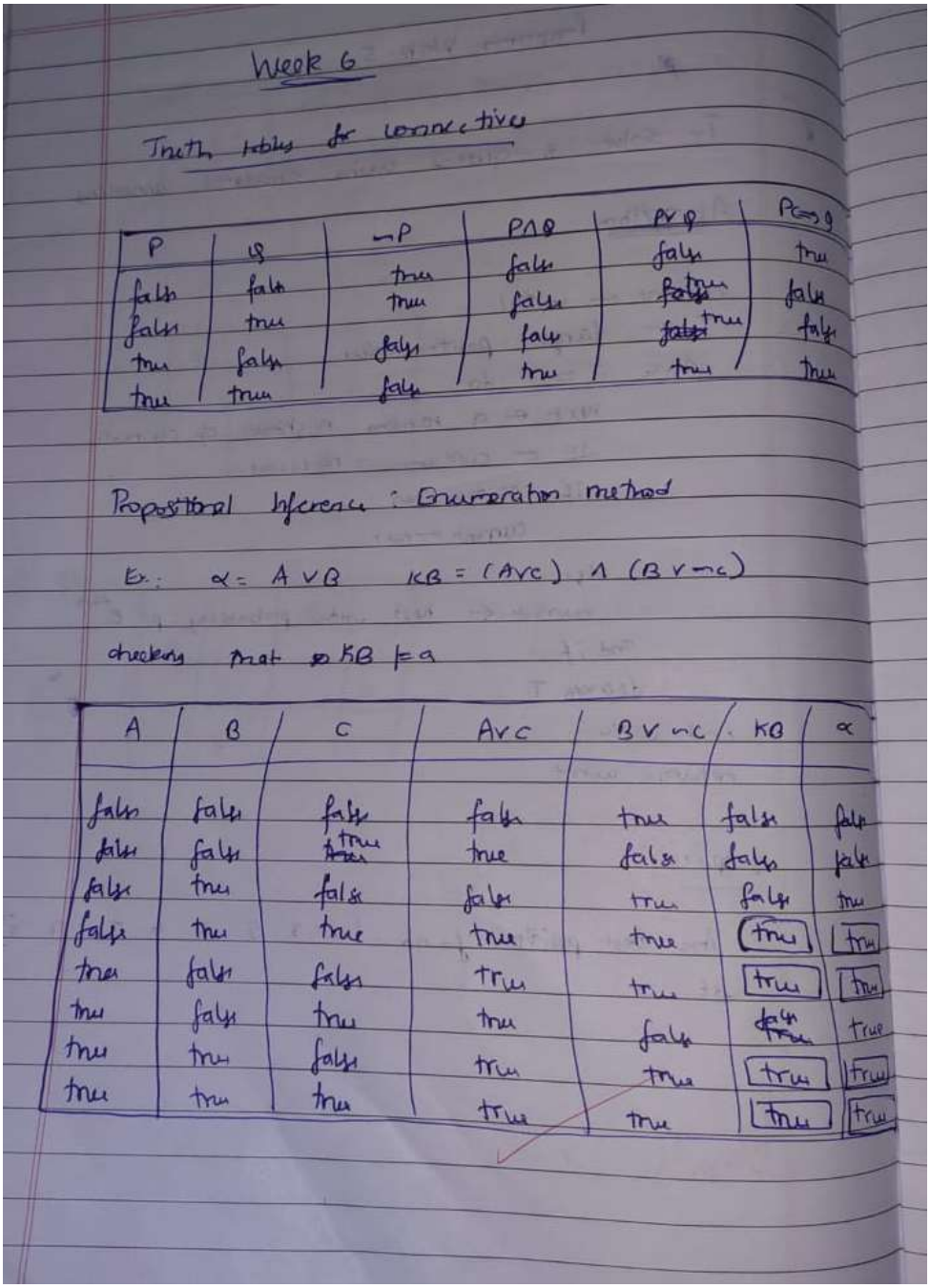
```
Sanath S Shetty
1BM23CS297
Enter the initial positions of the 4 queens (row for each column, 0-indexed):
Column 0: 3
Column 1: 2
Column 2: 1
Column 3: 0
Initial State:
. . . Q
. . Q .
. Q . .
Q . . .

Cost: 6

Iteration 0: Swap columns (0, 2)
. . . Q
Q . . .
. Q . .
. . Q .

Cost: 4, Temperature: 1000.0000

Iteration 1: Swap columns (1, 3)
. Q . .
Q . . .
. . . Q
. . Q .

Cost: 4, Temperature: 950.0000

Iteration 2: Swap columns (0, 3)
```

```
Terminated.
Final path:
Step 0:
. . . Q
. . Q .
. Q . .
Q . . .

Cost: 6
--------------------
Step 1:
. . . Q
Q . . .
. Q . .
. . Q .

Cost: 4
Swap columns: (0, 2)
--------------------
Step 2:
. Q . .
Q . . .
. . . Q
. . Q .

Cost: 4
Swap columns: (1, 3)
--------------------
Step 3:
. Q . .
. . . Q
Q . . .
. . Q .

Cost: 0
Swap columns: (0, 3)
--------------------
```

Program 6

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

Observation:



Week 6

Truth tables for connectives

P	Q	¬P	P∧Q	P∨Q	P⇒Q
false	false	true	false	false	true
false	true	true	false	false	false
true	false	false	false	true	false
true	true	false	true	true	true

Propositional Inference : Enumeration method

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

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

1. List all symbols in the knowledge base and query
2. Go through all possible truth assignments
3. for each assignment
   - If KB is true, check if α is also true
   - If KB is false, ignore
4. If α is true every time KB is true → KB entails α
5. If not → it does not entail α

Output

~~Sample~~

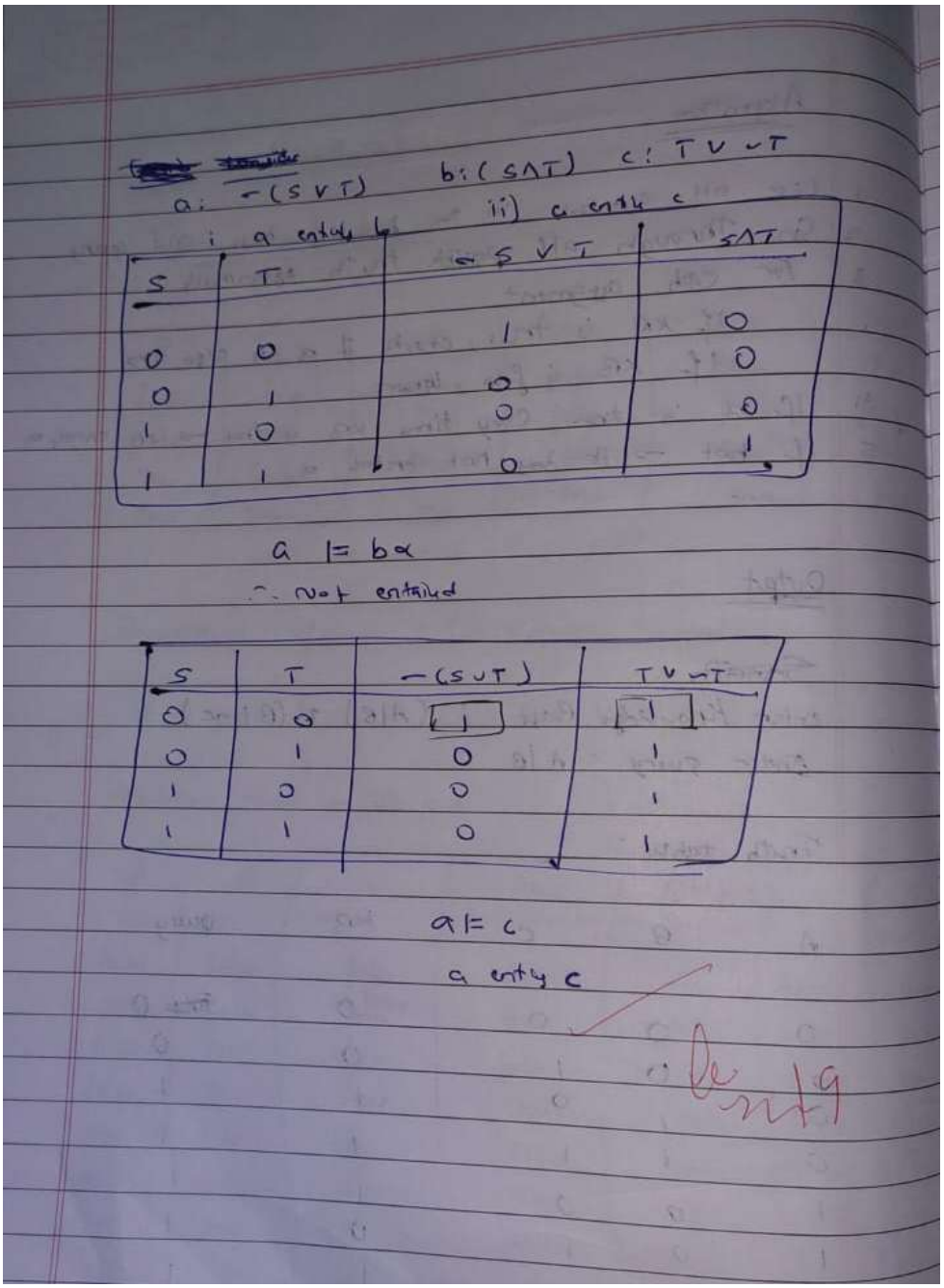
Enter Knowledge Base : (A|B) & (B|~C)

Enter query : A|B

Truth table :

A	B	C	KB	Query
0	0	0	0	~~False~~ 0
0	0	1	0	0
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	1	1
1	1	1	1	1

KB entails Query



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

i) a entails b

S	T	$\neg S \vee T$	$S \wedge T$
0	0	1	0
0	1	1	0
1	0	0	0
1	1	1	1

$a \models b$

∴ not entailed

ii) a entails c

S	T	$\neg(S \vee T)$	$T \vee \neg T$
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	1

$a \models c$

a entails c

∴ b entails a

Code:

```
import itertools
print("Sanath S Shetty")
print("1BM23CS297")
def eval_expr(expr, model):
    try:
        return eval(expr, {}, model)
    except:
        return False

def tt_entails(KB, query):
    symbols = sorted(set([ch for ch in KB + query if ch.isalpha()]))

    print("\nTruth Table:")
    print(" | ".join(symbols) + " | KB | Query")
    print("-" * (6 * len(symbols) + 20))

    entails = True
    for values in itertools.product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        kb_val = eval_expr(KB, model)
        query_val = eval_expr(query, model)

        row = " | ".join(["T" if model[s] else "F" for s in symbols])
```

```
        print(f"{row} |  {kb_val}  |  {query_val}")

        if kb_val and not query_val:
            entails = False

    return entails

KB = input("Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): ")
query = input("Enter Query: ")

result = tt_entails(KB, query)

print("\nResult:")
if result:
    print("KB entails Query (True in all cases).")
else:
    print("KB does NOT entail Query.")
```

Output:

```
Sanath S Shetty
1BM23CS297
Enter Knowledge Base (use &, |, ~ for AND, OR, NOT): (A | B) & (B | ~C)
Enter Query: A|B

Truth Table:
A | B | C | KB | Query
-------------------------------------
F | F | F |  0  |  False
F | F | T |  0  |  False
F | T | F |  1  |  True
F | T | T |  1  |  True
T | F | F |  1  |  True
T | F | T |  0  |  True
T | T | F |  1  |  True
T | T | T |  1  |  True

Result:
KB entails Query (True in all cases).
```

Program 7

Implement unification in first order logic

Observation

Program - 7

Unification algorithm

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

(1) Unify (Knows (John, x), Knows (John, Jane))

$\theta = x/Jane$

Unify (Knows (John, Jane), Knows (John, Jane))

(2) Unify (Knows (John, x), Knows (y, Bill))

$\theta = y/John$

Knows (John, x), Knows (John, Bill)

(3) Find mgu of

{p (b, x, P(g(z))}

{p (z, P(y), f(y)}

{p(z, x, f(g(z))}

$\theta = b/z$

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

{p(z, x, f(g(z))}

{p(z, x, x}

$\theta = f(y)/x$,

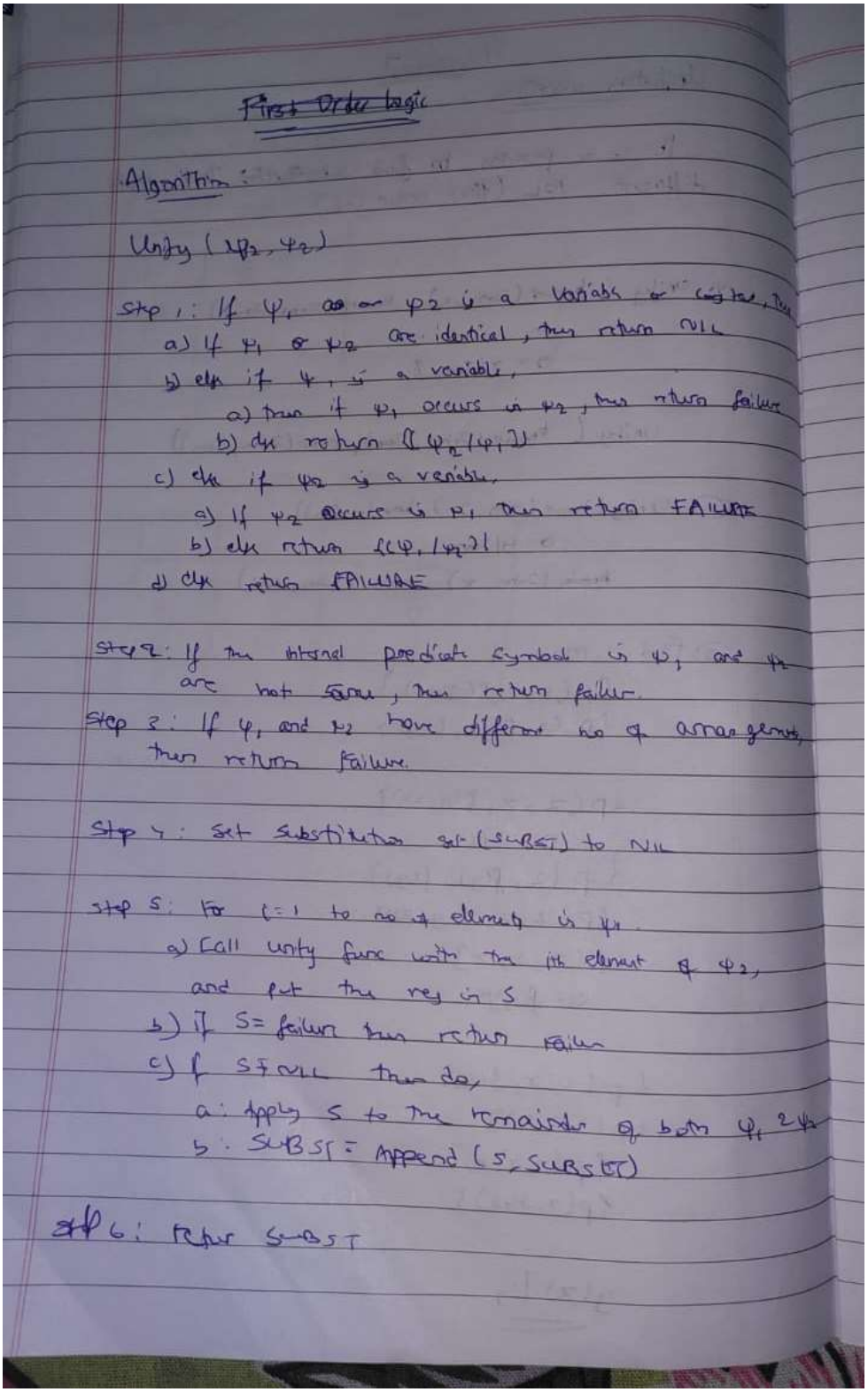
{p (z, x, f(y)}

<p(z, z, x)}

{p(z, x, z)}

{p(z, x, x}}

$g(z)/y$



First Order logic

Algorithm:

Unify (ψ_1, ψ_2)

Step 1: If ψ_1 or ψ_2 is a variable or constant, then

a) If ψ_1 & 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

Step 2: If the initial predicate symbol is ψ_1 and ψ_2 are not same, then return failure.

Step 3: If ψ_1 and ψ_2 have different no of arrangements, then return failure.

Step 4: Set substitution set (SUBST) to NIL

Step 5: For $i = 1$ to no of elements in ψ_1

a) Call unify func with the ith element of ψ_1 & ψ_2, and put the result in S

b) If S = failure then return failure

c) If S ≠ NIL then do,

a: Apply S to the remainder of both ψ_1 & ψ_2

b: SUBST = Append (S, SUBST)

Step 6: Return SUBST

Output:

Most general unifier: {'z': 'b', 'x': ('a', 'y'), 'y': ('y', 'z')}

② {p(f(a), g(y)), p(x, x)}
failure

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

④ Unify {Knows (John, x), Knows (y, mother(y))}

θ : y/John

Unify {Knows (John, x), Knows (John, mother (John))}

θ = x, mother (John)

Unify {Knows (John, mother (John)), Knows (John, mother (John))}

Code

```
print("Sanath S Shetty")
print("1BM23CS297")
class UnificationError(Exception):
    pass


def occurs_check(var, term):
    """Check if a variable occurs in a term (to prevent infinite recursion)."""
    if var == term:
        return True
    if isinstance(term, tuple):  # Term is a compound (function term)
        return any(occurs_check(var, subterm) for subterm in term)
    return False


def unify(term1, term2, substitutions=None):
    """Try to unify two terms, return the MGU (Most General Unifier)."""
    if substitutions is None:
        substitutions = {}

    # If both terms are equal, no further substitution is needed
    if term1 == term2:
        return substitutions
```

```
# If term1 is a variable, we substitute it with term2
elif isinstance(term1, str) and term1.isupper():
    # If term1 is already substituted, recurse
    if term1 in substitutions:
        return unify(substitutions[term1], term2, substitutions)
    elif occurs_check(term1, term2):
        raise UnificationError(f"Occurs check fails: {term1} in {term2}")
    else:
        substitutions[term1] = term2
        return substitutions

# If term2 is a variable, we substitute it with term1
elif isinstance(term2, str) and term2.isupper():
    # If term2 is already substituted, recurse
    if term2 in substitutions:
        return unify(term1, substitutions[term2], substitutions)
    elif occurs_check(term2, term1):
        raise UnificationError(f"Occurs check fails: {term2} in {term1}")
    else:
        substitutions[term2] = term1
        return substitutions

# If both terms are compound (i.e., functions), unify their parts recursively
elif isinstance(term1, tuple) and isinstance(term2, tuple):
```

```
        for subterm1, subterm2 in zip(term1, term2):
            substitutions = unify(subterm1, subterm2, substitutions)

        return substitutions

    else:
        raise UnificationError(f"Cannot unify: {term1} with {term2}")

# Define the terms as tuples
term1 = ('p', 'b', 'X', ('f', ('g', 'Z')))
term2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

try:
    # Find the MGU
    result = unify(term1, term2)
    print("Most General Unifier (MGU):")
    print(result)
except UnificationError as e:
    print(f"Unification failed: {e}")
```

Output:

```
Sanath S Shetty
1BM23CS297
Most General Unifier (MGU):
{'Z': 'b', 'X': ('f', 'Y'), 'Y': ('g', 'Z')}
```

Program 8

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

reasoning.

Observation:

Week 8

First Order Logic

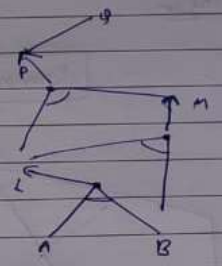
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Premises

Conclusion

Rules
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$B \wedge L \Rightarrow M$

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: West is criminal

Rules

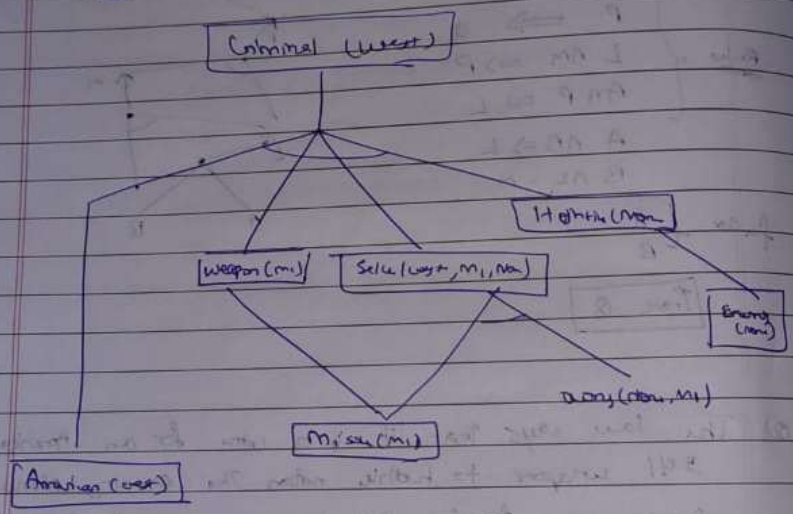
1) $\forall x, y, z$ American(x) $\wedge$ Weapon(y) $\wedge$ Sells(x, y, z) $\wedge$ Hostile(z) $\Rightarrow$ Criminal(x)

2) $\forall x$ missile(x) $\wedge$ Owns(Nono, x) $\Rightarrow$ Sells(West, x, Nono)

3) $\forall x$ Enemy(x, American) $\Rightarrow$ Hostile(x)

4) ∀ x Missile(x) ⇒ Weapon(x)
5) American(West)
6) Enemy(NONO, America)
7) Owns(NONO, M1) and
8) Missile(M1)

Proof



Algorithm

Function FOL-FC-ASK(KB, α) returns a substitution or false
inputs: KB, set of first order,
α, the query, an atomic sentence

local variables: new, new sentences inferred on each iteration
repeat until new is empty
new ← { }
for each rule in KB do
(p1 ∧ ... ∧ pn ⇒ q) ← Standardize-variables(rule)

for each θ such that subst($\theta, p_1, \ldots p_n$) = subst($\theta, p_1', \ldots p_n'$)
for some $p_1', \ldots p_n'$ in KB
$q' = subst(\theta, q)$
if q' does not unify with some sentence already in KB or new then,
add q' to new
$\phi \leftarrow unify(q', \alpha)$

if ϕ is not fail then return ϕ
add new to KB
return fail

Output:

Robert Robert is a criminal.

Code:

```
print("Sanath S Shetty")
print("1BM23CS297")
def FOL_FC_ASK(KB, alpha):
# Initialize the new sentences to be inferred in this iteration
new = set()

while new: # Repeat until new is empty
new = set() # Clear new sentences on each iteration

# For each rule in KB
for rule in KB:
# Standardize the rule variables to avoid conflicts
standardized_rule = standardize_variables(rule)
p1_to_pn, q = standardized_rule # Premises (p1, ..., pn) and conclusion (q)

# For each substitution θ such that Subst(θ, p1, ..., pn) matches the premises
for theta in get_matching_substitutions(p1_to_pn, KB):
q_prime = apply_substitution(theta, q)

# If q_prime does not unify with some sentence already in KB or new
if not any(unify(q_prime, sentence) != 'FAILURE' for sentence in KB.union(new)):
new.add(q_prime) # Add q_prime to new

# Unify q_prime with the query (alpha)
phi = unify(q_prime, alpha)
if phi != 'FAILURE':
return phi # Return the substitution if it unifies

# Add newly inferred sentences to the knowledge base
KB.update(new)

return False # Return false if no substitution is found


def standardize_variables(rule):
```

```
"""
Standardize variables in the rule to avoid variable conflicts.
Rule is assumed to be a tuple (premises, conclusion).
"""
premises, conclusion = rule
# Apply standardization logic here (for simplicity, assume no conflict in this case)
return (premises, conclusion)


def get_matching_substitutions(premises, KB):
"""
Get matching substitutions for the premises in the KB.
This is a placeholder to represent how substitutions would be found.
"""
# Implement substitution matching here
return [] # This should return a list of valid substitutions


def apply_substitution(theta, expression):
"""
Apply a substitution θ to an expression.
This function will replace variables in expression with their corresponding terms
from θ.
"""
if isinstance(expression, str) and expression.startswith('?'):
return theta.get(expression, expression) # Apply substitution to variable
elif isinstance(expression, tuple):
return tuple(apply_substitution(theta, arg) for arg in expression)
return expression


def unify(psi1, psi2, subst=None):
"""Unification algorithm (simplified)"""
if subst is None:
subst = {}

def apply_subst(s_map, expr):
```

```
if isinstance(expr, str) and expr.startswith('?'):
return s_map.get(expr, expr)
elif isinstance(expr, tuple):
return tuple(apply_subst(s_map, arg) for arg in expr)
return expr

def is_variable(expr):
return isinstance(expr, str) and expr.startswith('?')

_psi1 = apply_subst(subst, psi1)
_psi2 = apply_subst(subst, psi2)

if is_variable(_psi1) or is_variable(_psi2) or not isinstance(_psi1, tuple) or not
isinstance(_psi2, tuple):
if _psi1 == _psi2:
return subst
elif is_variable(_psi1):
if _psi1 in str(_psi2):
return 'FAILURE'
return {**subst, _psi1: _psi2}
elif is_variable(_psi2):
if _psi2 in str(_psi1):
return 'FAILURE'
return {**subst, _psi2: _psi1}
else:
return 'FAILURE'

if _psi1[0] != _psi2[0] or len(_psi1) != len(_psi2):
return 'FAILURE'

for arg1, arg2 in zip(_psi1[1:], _psi2[1:]):
s = unify(arg1, arg2, subst)
if s == 'FAILURE':
return 'FAILURE'
subst = s

return subst
```

```
# Knowledge Base (KB)
KB = set()

# Adding facts to KB:
KB.add(('american', 'Robert')) # Robert is an American
KB.add(('hostile_nation', 'Country_A')) # Country A is a hostile nation
KB.add(('sell_weapons', 'Robert', 'Country_A')) # Robert sold weapons to Country
A

# Adding the rule (the law):
KB.add((('american(x)', 'hostile_nation(y)', 'sell_weapons(x, y)'),
'criminal(x)'))

# Goal: Prove that Robert is a criminal
goal = 'criminal(Robert)'

# Calling FOL_FC_ASK to prove the goal
result = FOL_FC_ASK(KB, goal)

if result != 'FAILURE':
print("Robert is a criminal!")
else:
print("Robert is not a criminal.")
```

Output:

```
Sanath S Shetty
1BM23CS297
Robert is a criminal!
```

Program 9

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

Observation:

Week - 9

Resolution in FOL

1) Eliminate bicondition and implication.
Eliminate $\leftrightarrow$ replacing $\alpha \leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$

2) Move $\neg$ inwards
$\neg(\forall x\, p) \equiv \exists x \neg p$
$\neg(\exists x\, p) \equiv \forall x \neg p$
$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$
$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$
$\neg\neg\alpha \equiv \alpha$

3) Standardize variables apart by renaming them, each quantifier should use different variable.

4. Skolemize: each existential variable is replaced by its Skolem constant or Skolem function of the enclosing universally quantified variable.

- For instance, $\exists x$ Rich(x) becomes Rich(H1) where H1 is new Skolem constant

5. Drop universal quantifiers
- $\forall$ person(x) becomes Person(x)

6. Distribute $\wedge$ over $\vee$:
$(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Algorithm for resolution

Basic steps for proving a conclusion S given premises $P_1, P_2 \ldots P_n$ (all expressed in FOL);

1. Convert all sentences to CNF
2. Negate conclusion S and convert result to CNF
3. Add negated conclusion and to the premise clauses
4. Repeat until contradiction or no progress is made
   a. Select 2 clauses (call them parent 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

Proof Exp 4:

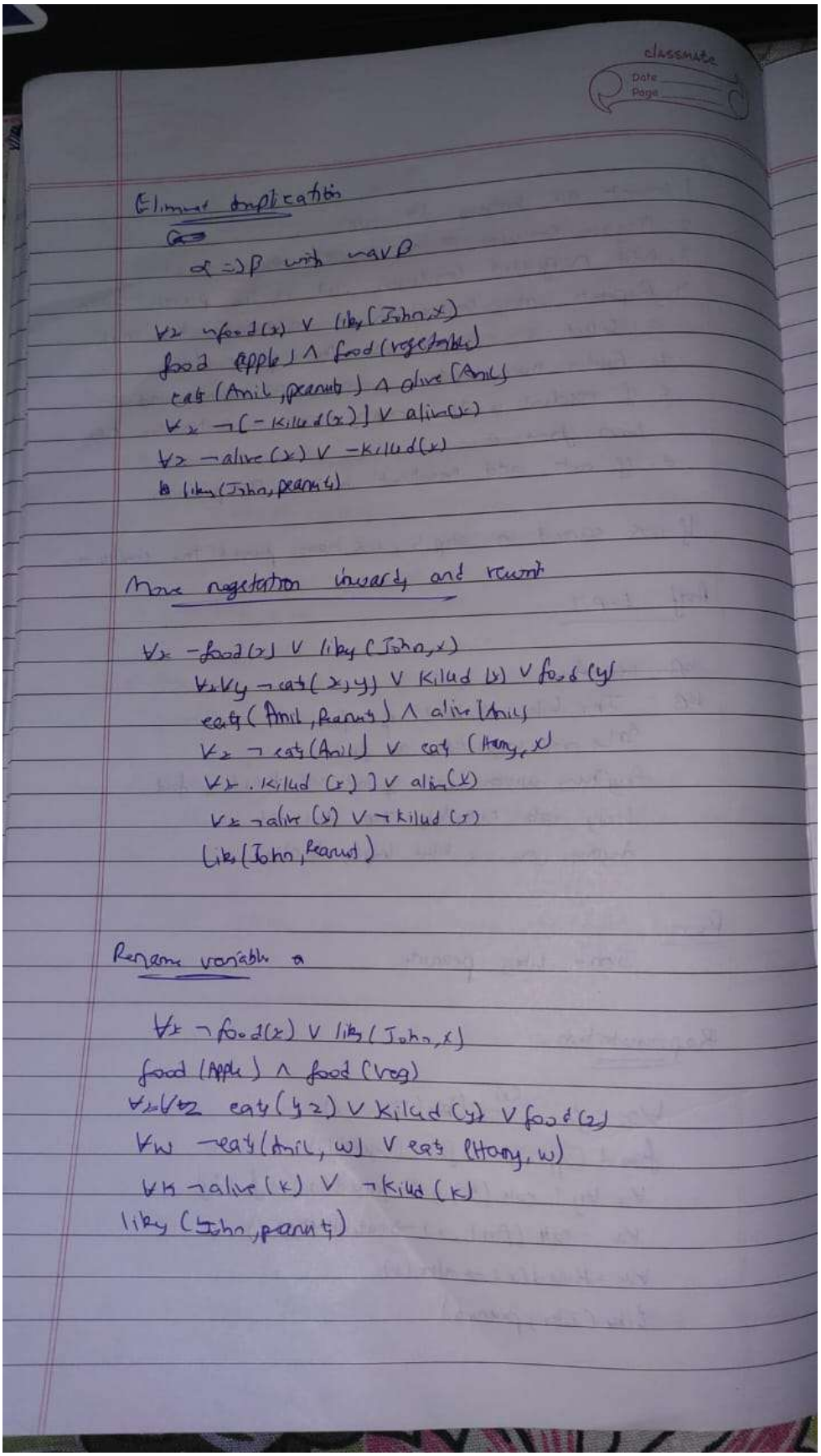
~~KB ⊨ A~~

KB = John Like all kind of food
Apple and vegetables are food
Anything anyone eats and not killed is food
Harry eats everything that Anil eats
Anyone who is killed implies not alive.

Prove
John likes peanuts.

Representation.

$\forall x : food(x) \rightarrow likes(John, x)$
food(apple) ∧ food(vegetables)
$\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
$\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
$\forall x : killed(x) \rightarrow alive(x)$
likes(John, peanuts)



Eliminate implication

$\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

$\forall x \neg food(x) \vee likes(John, x)$
food(apple) $\wedge$ food(vegetables)
eats(Anil, peanuts) $\wedge$ alive(Anil)
$\forall x \neg[\neg killed(x)] \vee alive(x)$
$\forall x \neg alive(x) \vee \neg killed(x)$
likes(John, peanuts)

Move negation inwards and recurse

$\forall x \neg food(x) \vee likes(John, x)$
$\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$
eats(Anil, peanuts) $\wedge$ alive(Anil)
$\forall x \neg eats(Anil) \vee eats(Harry, x)$
$\forall x \neg[killed(x)] \vee alive(x)$
$\forall x \neg alive(x) \vee \neg killed(x)$
likes(John, peanuts)

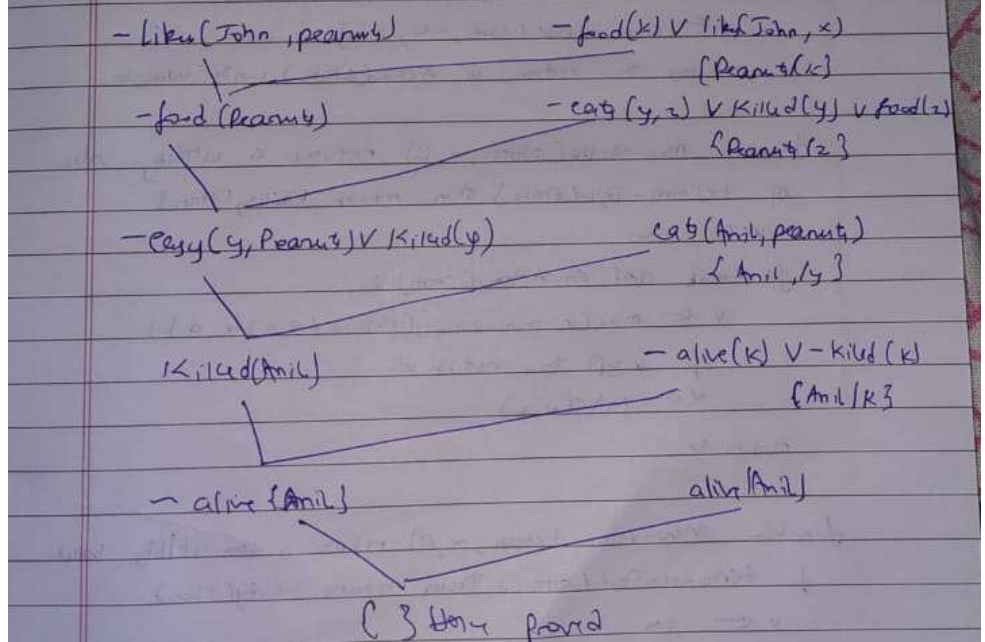
Rename variables

$\forall x \neg food(x) \vee likes(John, x)$
food(Apple) $\wedge$ food(veg)
$\forall y \forall z$ eats$(y, z) \vee killed(y) \vee food(z)$
$\forall w \neg eats(Anil, w) \vee eats(Harry, w)$
$\forall k \neg alive(k) \vee \neg killed(k)$
likes(John, peanuts)

Deep universal

food(x) ∨ likes(John, x)
food(Apple)
food(veg)
−eats(y, z) ∨ killed(y) ∨ food(z)
eats(Anil, peanuts)
alive(Anil)
killed(g) ∨ alive(g)
~~likes~~ likes(John, Peanuts)

Graph



Code:

```
print("Sanath S Shetty")
print("1BM23CS297")

# a. ¬food(x) V likes(John, x)
clause_a = {"NOT food(x)", "likes(John, x)"}
# d. ¬eats(y, z) V killed(y) V food(z)
clause_d = {"NOT eats(y, z)", "killed(y)", "food(z)"}
# e. eats(Anil, Peanuts)
clause_e = {"eats(Anil, Peanuts)"}
# f. alive(Anil)
clause_f = {"alive(Anil)"}
# g. ¬eats(Anil, w) V eats(Harry, w)
clause_g = {"NOT eats(Anil, w)", "eats(Harry, w)"}
# h. killed(g) V alive(g)
clause_h = {"killed(g)", "alive(g)"}
# i. ¬alive(k) V ¬killed(k)
clause_i = {"NOT alive(k)", "NOT killed(k)"}
negated_goal = {"NOT likes(John, Peanuts)"}


def resolve(clause1, literal1, clause2, literal2):
    new_clause = clause1.union(clause2)
```

```
    # Remove the two complementary literals
    new_clause.remove(literal1)
    new_clause.remove(literal2)
    return new_clause

print("Starting Resolution Proof...\n")
print(f"Goal: Prove 'likes(John, Peanuts)'")
print(f"Negated Goal: {negated_goal}\n")
#   {NOT likes(John, Peanuts)}  (Negated Goal)
#   {NOT food(x), likes(John, x)}  (Clause a)
# Unification: {x / Peanuts}
# Result: {NOT food(Peanuts)}
print("Step 1: Resolving Negated Goal with Clause a")
print(f"  {negated_goal}")
print(f"  {clause_a}  [Unify x=Peanuts]")
resolvent_1 = resolve(negated_goal, "NOT likes(John, Peanuts)",
                      clause_a, "likes(John, x)")
# Manually apply unification
resolvent_1.remove("NOT food(x)")
resolvent_1.add("NOT food(Peanuts)")
print(f"  Result 1: {resolvent_1}\n")

#   {NOT food(Peanuts)}  (Result 1)
#   {NOT eats(y, z), killed(y), food(z)} (Clause d)
```

```
# Unification: {z / Peanuts}
# Result: {NOT eats(y, Peanuts), killed(y)}
print("Step 2: Resolving Result 1 with Clause d")
print(f"  {resolvent_1}")
print(f"  {clause_d}  [Unify z=Peanuts]")
resolvent_2 = resolve(resolvent_1, "NOT food(Peanuts)",
                      clause_d, "food(z)")
# Manually apply unification
resolvent_2.remove("NOT eats(y, z)")
resolvent_2.remove("killed(y)")
resolvent_2.add("NOT eats(y, Peanuts)")
resolvent_2.add("killed(y)")
print(f"  Result 2: {resolvent_2}\n")
#   {NOT eats(y, Peanuts), killed(y)} (Result 2)
#   {eats(Anil, Peanuts)} (Clause e)
# Unification: {y / Anil}
# Result: {killed(Anil)}
print("Step 3: Resolving Result 2 with Clause e")
print(f"  {resolvent_2}")
print(f"  {clause_e}  [Unify y=Anil]")
resolvent_3 = resolve(resolvent_2, "NOT eats(y, Peanuts)",
                      clause_e, "eats(Anil, Peanuts)")
# Manually apply unification
resolvent_3.remove("killed(y)")
```

```
resolvent_3.add("killed(Anil)")
print(f"  Result 3: {resolvent_3}\n")

#   {killed(Anil)} (Result 3)
#   {NOT alive(k), NOT killed(k)} (Clause i)
# Unification: {k / Anil}
# Result: {NOT alive(Anil)}
print("Step 4: Resolving Result 3 with Clause i")
print(f"  {resolvent_3}")
print(f"  {clause_i}  [Unify k=Anil]")
resolvent_4 = resolve(resolvent_3, "killed(Anil)",
                      clause_i, "NOT killed(k)")
# Manually apply unification
resolvent_4.remove("NOT alive(k)")
resolvent_4.add("NOT alive(Anil)")
print(f"  Result 4: {resolvent_4}\n")

# --- Proof Step 5 ---
# Resolving:
#   {NOT alive(Anil)} (Result 4)
#   {alive(Anil)} (Clause f)
# Unification: {}
# Result: {} (Empty Clause)
print("Step 5: Resolving Result 4 with Clause f")
```

```
print(f"  {resolvent_4}")
print(f"  {clause_f}")
resolvent_5 = resolve(resolvent_4, "NOT alive(Anil)",
                      clause_f, "alive(Anil)")
print(f"  Result 5: {resolvent_5}\n")


    print("-------------------------------------------------")
    print("Conclusion: An empty clause {} was derived.")
    print("This is a contradiction, which means the negated goal is false.")
    print("Therefore, the original goal 'likes(John, Peanuts)' is TRUE.")
    print("Hence proved. ")
else:
    print("Proof failed. Could not derive the empty clause.")
```

Output:

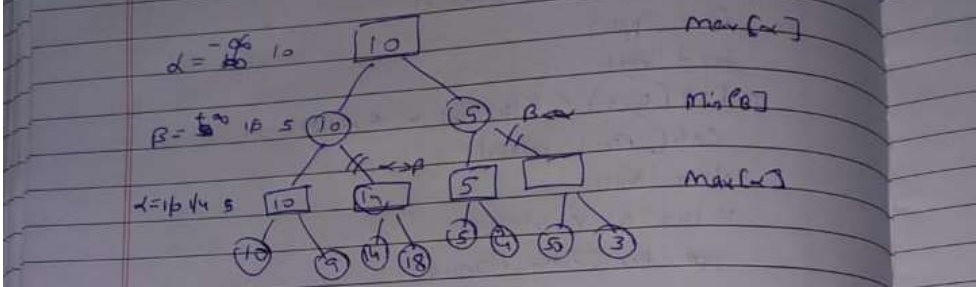
```
Sanath S Shetty
1BM23CS297
Starting Resolution Proof...

Goal: Prove 'likes(John, Peanuts)'
Negated Goal: {'NOT likes(John, Peanuts)'}

Step 1: Resolving Negated Goal with Clause a
  {'NOT likes(John, Peanuts)'}
  {'likes(John, x)', 'NOT food(x)'}  [Unify x=Peanuts]
  Result 1: {'NOT food(Peanuts)'}

Step 2: Resolving Result 1 with Clause d
  {'NOT food(Peanuts)'}
  {'killed(y)', 'NOT eats(y, z)', 'food(z)'}  [Unify z=Peanuts]
  Result 2: {'killed(y)', 'NOT eats(y, Peanuts)'}

Step 3: Resolving Result 2 with Clause e
  {'killed(y)', 'NOT eats(y, Peanuts)'}
  {'eats(Anil, Peanuts)'}  [Unify y=Anil]
  Result 3: {'killed(Anil)'}

Step 4: Resolving Result 3 with Clause i
  {'killed(Anil)'}
  {'NOT alive(k)', 'NOT killed(k)'}  [Unify k=Anil]
  Result 4: {'NOT alive(Anil)'}

Step 5: Resolving Result 4 with Clause f
  {'NOT alive(Anil)'}
  {'alive(Anil)'}
  Result 5: set()

--------------------------------------------------
Conclusion: An empty clause {} was derived.
This is a contradiction, which means the negated goal is false.
Therefore, the original goal 'likes(John, Peanuts)' is TRUE.
Hence proved.
```

Program 10

Implement Alpha-Beta Pruning.

Observation:

Week 10 α-β - pruning



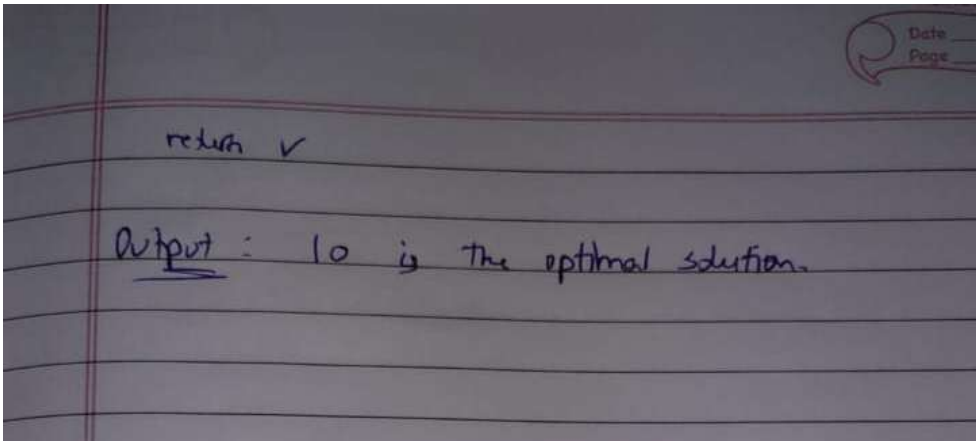
Algorithm

function A-B-S (state) returns an action.
  $v \leftarrow$ MAX-VALUE (state, $-\infty$, $+\infty$)
  return the action in Actions(state) with value v.

function MAX-VALUE(state, α, β) returns a utility value
  if terminal-Test(state) then return Utility(state)
  $v \leftarrow -\infty$
  for each a in Actions(state) do
    $v \leftarrow$ max(v, MIN-VALUE(Result(s,a), α, β))
    if $v \geq \beta$ then return v
    $\alpha \leftarrow$ MAX(α, v)
  return v

function MIN-VALUE (state, α, β) returns a utility value
  if terminal-Test(state) then return Utility(state)
  $v \leftarrow +\infty$

for each a in Actions(state) do
  $v \leftarrow$ MIN(v, MAX-VALUE(Result(s,a), α, β))
  if $v \leq \alpha$ then return v
  $\beta \leftarrow$ MIN(β, v)



Code:

```
import math

print("Sanath S Shetty")
print("1BM23CS297")

tree = []
pruned_nodes_list = []

def get_node_by_path(path_indices):
    node = tree
    try:
        for index in path_indices:
            node = node[index]
        if isinstance(node, int):
            return f"Leaf node ({node})"
```

```
        else:
            if all(isinstance(n, int) for n in node):
                return f"Branch node (children: {node})"
            else:
                return f"Branch node (children: {len(node)})"
    except (IndexError, TypeError):
        return "Path Error"


def max_value(node, alpha, beta, path_indices=[]):
    if isinstance(node, int):
        return node, []
    v = -math.inf
    best_path = []
    for i, child in enumerate(node):
        child_v, child_path = min_value(child, alpha, beta, path_indices + [i])
        if child_v > v:
            v = child_v
            best_path = [i] + child_path
        if v >= beta:
            for j in range(i + 1, len(node)):
                pruned_path = path_indices + [j]
                pruned_nodes_list.append(f"Path {pruned_path} (Node:
{get_node_by_path(pruned_path)})")
            print(f"PRUNING (MAX): At path {path_indices}, v={v} >= beta={beta}.
Pruning remaining children.")
```

```
            return v, best_path
        alpha = max(alpha, v)
    return v, best_path


def min_value(node, alpha, beta, path_indices=[]):
    if isinstance(node, int):
        return node, []
    v = math.inf
    best_path = []
    for i, child in enumerate(node):
        child_v, child_path = max_value(child, alpha, beta, path_indices + [i])
        if child_v < v:
            v = child_v
            best_path = [i] + child_path
        if v <= alpha:
            for j in range(i + 1, len(node)):
                pruned_path = path_indices + [j]
                pruned_nodes_list.append(f"Path {pruned_path} (Node:
{get_node_by_path(pruned_path)})")
            print(f"PRUNING (MIN): At path {path_indices}, v={v} <=
alpha={alpha}. Pruning remaining children.")
            return v, best_path
        beta = min(beta, v)
    return v, best_path
```

```
if __name__ == "__main__":
    tree = [
        [
            [10, 9],
            [14, 18]
        ],
        [
            [5, 4],
            [50, 3]
        ]
    ]


    print("Starting Alpha-Beta Search...")
    root_value, optimal_path_indices = max_value(tree, -math.inf, math.inf,
path_indices=[])
    print("\n--- Search Complete ---")
    print(f"\n## Value of the Root Node")
    print(f"The final optimal value for the root node is: {root_value}")
    print(f"\n## Optimal Path")
    print("The path to achieve this value is:")
    path_str = "Root"
    current_node = tree
    print(f"  {path_str} (Value: {root_value})")
    for index in optimal_path_indices:
```

```
    current_node = current_node[index]
    if isinstance(current_node, int):
        path_str += f" -> [Leaf {current_node}]"
    else:
        path_str += f" -> [Node at index {index}]"
    print(f"  {path_str}")
print(f"\n## Pruned Paths")
if pruned_nodes_list:
    print("The following paths were pruned (not explored):")
    for p in pruned_nodes_list:
        print(f"- {p}")
else:
    print("No paths were pruned.")
```

Output:

```
Sanath S Shetty
1BM23CS297
Starting Alpha-Beta Search...
PRUNING (MAX): At path [0, 1], v=14 >= beta=10. Pruning remaining children.
PRUNING (MIN): At path [1], v=5 <= alpha=10. Pruning remaining children.

--- Search Complete ---

## Value of the Root Node
The final optimal value for the root node is: 10

## Optimal Path
The path to achieve this value is:
  Root (Value: 10)
  Root -> [Node at index 0]
  Root -> [Node at index 0] -> [Node at index 0]
  Root -> [Node at index 0] -> [Node at index 0] -> [Leaf 10]

## Pruned Paths
The following paths were pruned (not explored):
- Path [0, 1, 1] (Node: Leaf node (18))
- Path [1, 1] (Node: Branch node (children: [50, 3]))
```