VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by
Thrisha D (1BM24CS426)

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 **Thrisha D (1BM24CS426),** 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.

Sonika Sharma D Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

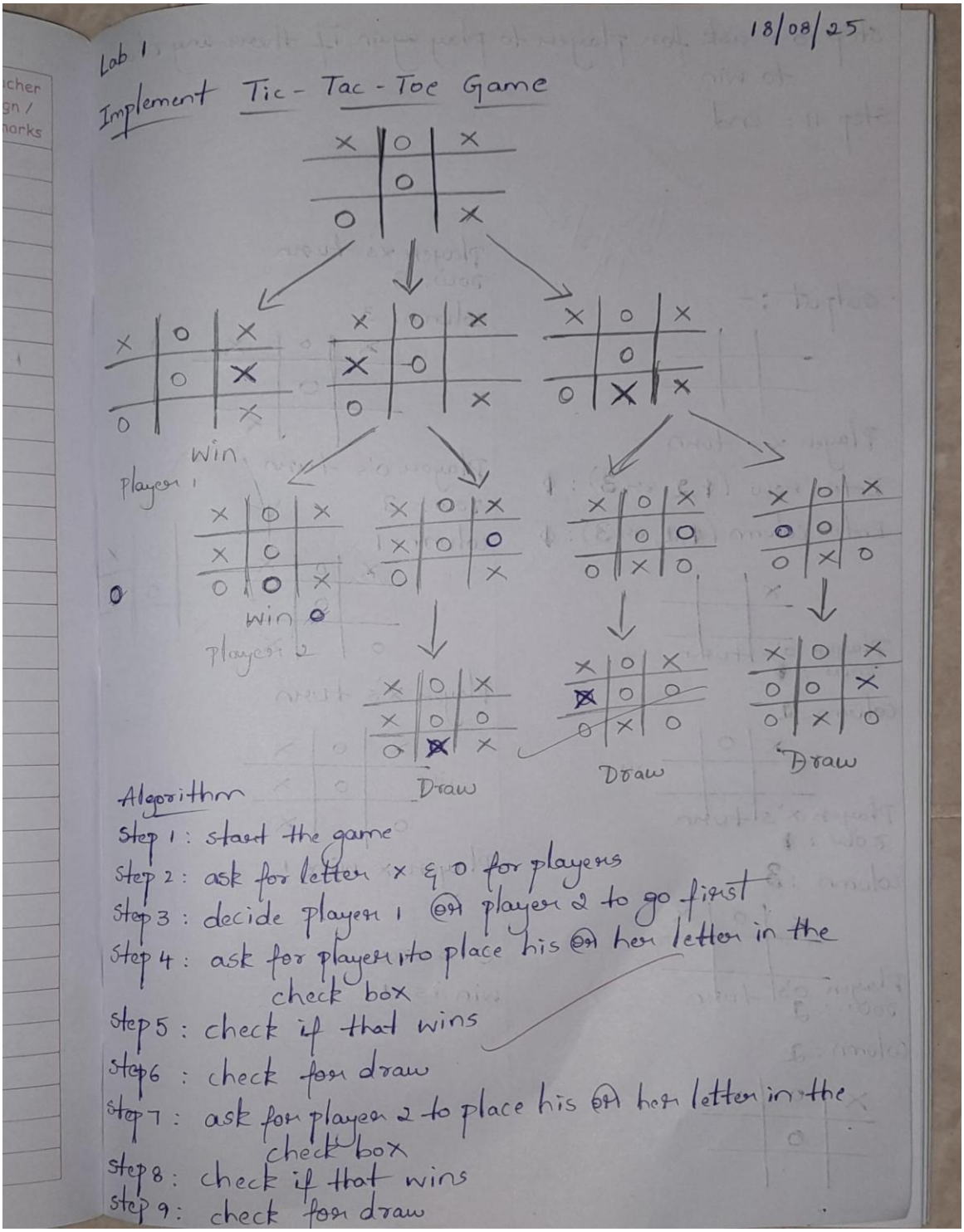
Index

Github Link:
https://github.com/ThrishaD2/ArtificialIntelligence.git

Program 1

Implement Tic –Tac –Toe Game Implement
vacuum cleaner agent

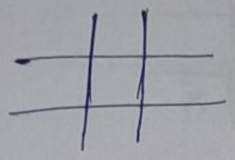
Algorithm:



step 10 : ask for player to play again if there any chance to win

step 11 : end

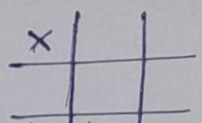
Output :-



Player x's turn

Enter row (1, 2, or 3) : 1

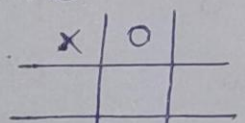
Enter column (1, 2, or 3) : 1



Player o's turn

row : 1

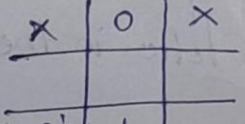
column : 2



Player x's turn

row : 1

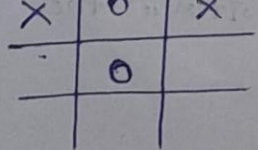
column : 3



Player o's turn

row : 2

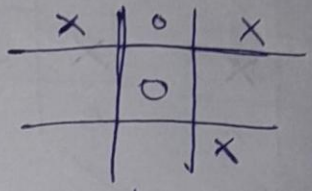
column : 2



Player x's turn

row : 3

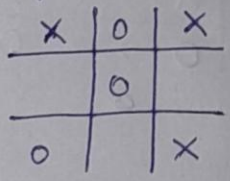
column : 3



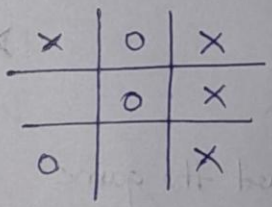
Player o's turn

row : 3

column : 1



Player x's turn



Player x wins!

o is draw

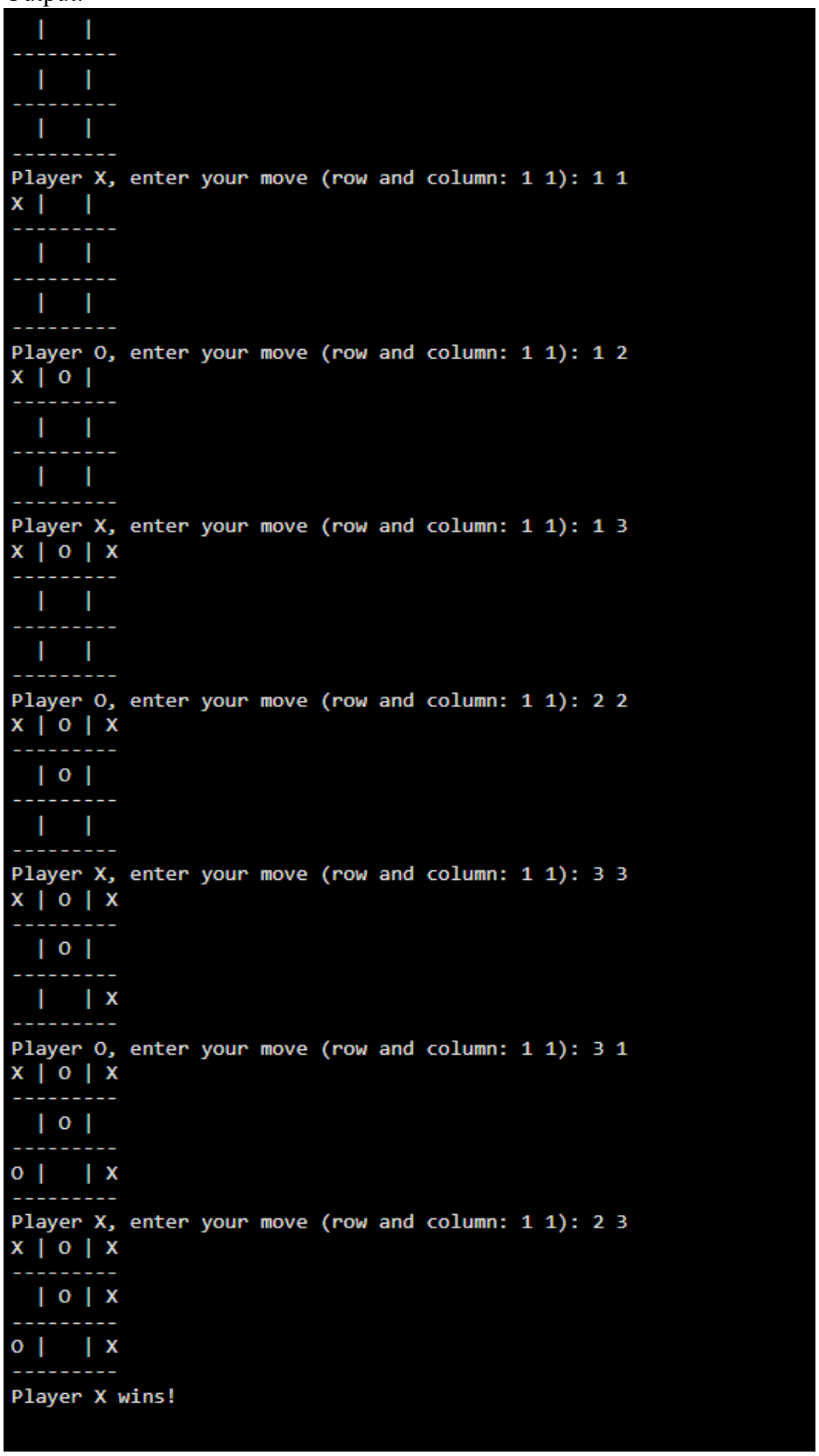
win is 1

Code:
Tic –Tac –Toe Game

```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board, player):
    # Check rows, columns and diagonals
    for i in range(3):
        if all([cell == player for cell in board[i]]) or \
           all([board[j][i] == player for j in range(3)]):
            return True

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

    return False


def is_full(board):
    return all(cell in ['X', 'O'] for row in board for cell in row)


def get_move(player):
    while True:
        try:
            move = input(f"Player {player}, enter your move (row and column: 1 1): ")
            row, col = map(int, move.split())
            if row in [1, 2, 3] and col in [1, 2, 3]:
                return row - 1, col - 1
            else:
                print("Invalid input. Enter numbers between 1 and 3.")
        except ValueError:
            print("Invalid input. Enter two numbers separated by space.")


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

    while True:
        print_board(board)
        row, col = get_move(current_player)
```

```
        if board[row][col] != " ":
            print("That spot is taken. Try again.")
            continue

        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break

        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

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


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

Output:


| |

| |

| |

Player X, enter your move (row and column: 1 1): 1 1
X | |

| |

| |

Player O, enter your move (row and column: 1 1): 1 2
X | O |

| |

| |

Player X, enter your move (row and column: 1 1): 1 3
X | O | X

| |

| |

Player O, enter your move (row and column: 1 1): 2 2
X | O | X

| O |

| |

Player X, enter your move (row and column: 1 1): 3 3
X | O | X

| O |

| | X

Player O, enter your move (row and column: 1 1): 3 1
X | O | X

| O |

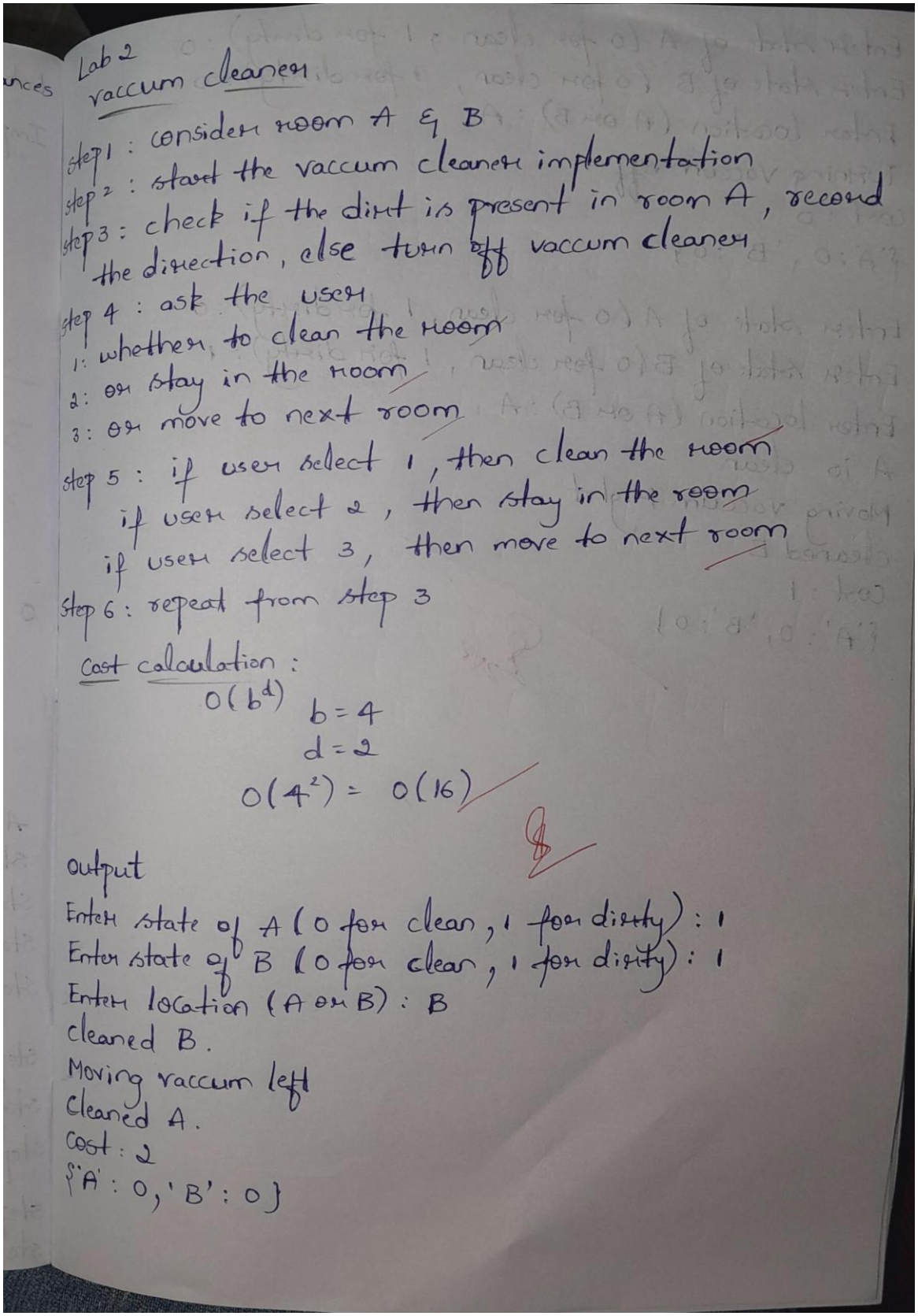
O | | X

Player X, enter your move (row and column: 1 1): 2 3
X | O | X

| O | X

O | | X

Player X wins!



Lab 2

vaccum cleaner

step 1 : consider room A & B

step 2 : start the vaccum cleaner implementation

step 3 : check if the dirt is present in room A, record the direction, else turn off vaccum cleaner

step 4 : ask the user

1: whether to clean the room

2: or stay in the room

3: or move to next room

step 5 : if user select 1, then clean the room

if user select 2, then stay in the room

if user select 3, then move to next room

step 6 : repeat from step 3

Cost calculation :

$O(b^d)$ $b = 4$

$d = 2$

$O(4^2) = O(16)$

Output

Enter state of A (0 for clean, 1 for dirty) : 1

Enter state of B (0 for clean, 1 for dirty) : 1

Enter location (A or B) : B

cleaned B.

Moving vaccum left

Cleaned A.

Cost : 2

{'A' : 0, 'B' : 0}

Enter state of A (0 for clean, 1 for dirty) : 0
Enter state of B (0 for clean, 1 for dirty) : 0
Enter location (A or B) : A
Turning vaccum off
cost : 0
{'A' : 0, 'B' : 0}

Enter state of A (0 for clean, 1 for dirty) : 0
Enter state of B (0 for clean, 1 for dirty) : 1
Enter location (A or B) : A
A is clean
Moving vaccum right
cleaned B
cost : 1
{'A' : 0, 'B' : 0}

```
def vacuum_simulation():
    cost = 0

    # Get initial states and location
    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()

    # Vacuum operation loop
    while True:
        if location == 'A':
            if state_A == 1:
                print("Cleaning A.")
                state_A = 0
                cost += 1
            elif state_B == 1:
                print("Moving vacuum right")
                location = 'B'
                cost += 1
            else:
                print("Turning vacuum off")
                break

        elif location == 'B':
            if state_B == 1:
                print("Cleaning B.")
                state_B = 0
                cost += 1
            elif state_A == 1:
                print("Moving vacuum left")
```

```
            location = 'A'
            cost += 1
        else:
            print("Turning vacuum off")
            break
    print(f"Cost: {cost}")
    print(f"{{'A': {state_A}, 'B': {state_B}}}")


vacuum_simulation()
```

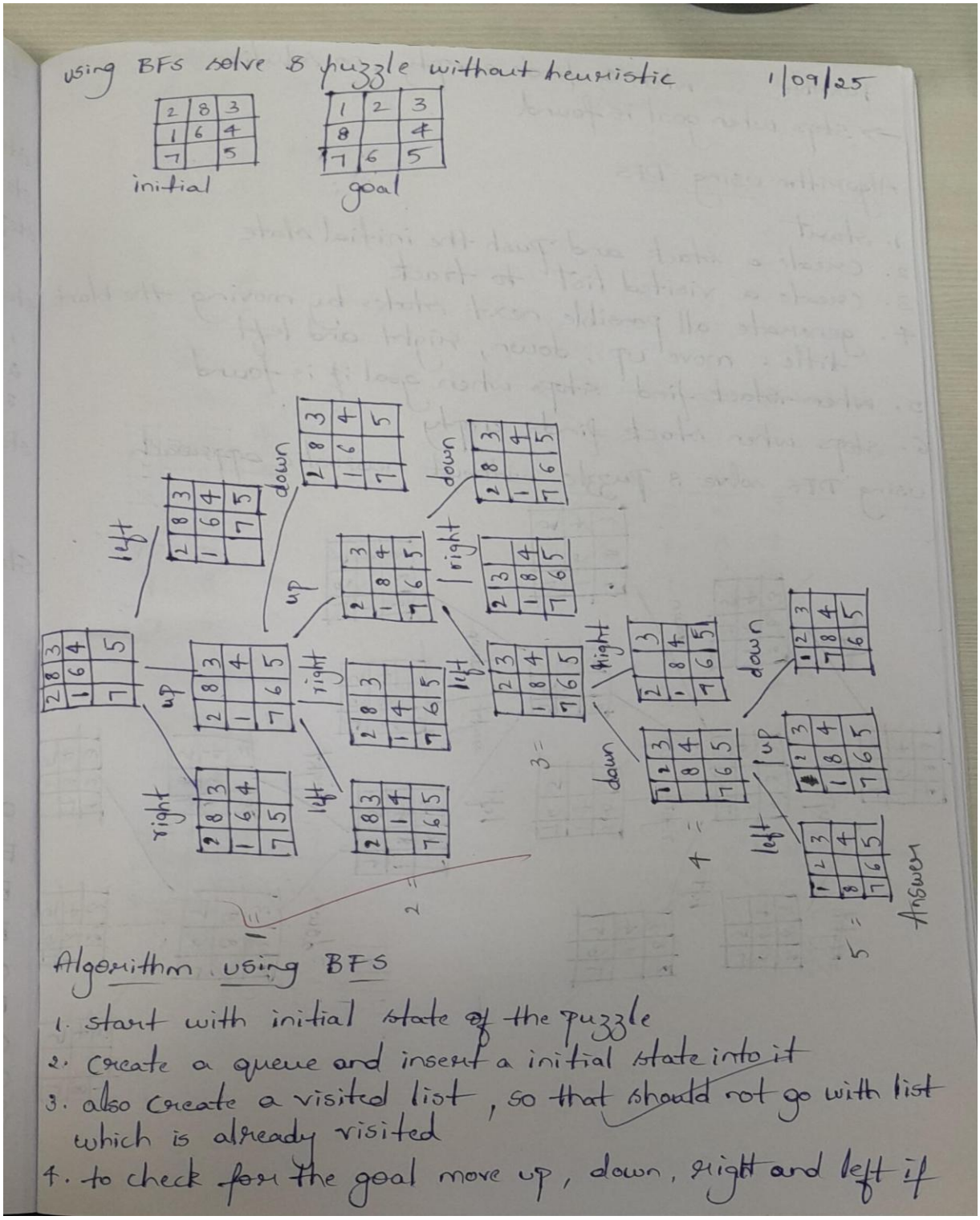
OUTPUT

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaning A.
Turning vacuum off
Cost: 1
{'A': 0, 'B': 0}
```

Program 2

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

Algorithm:



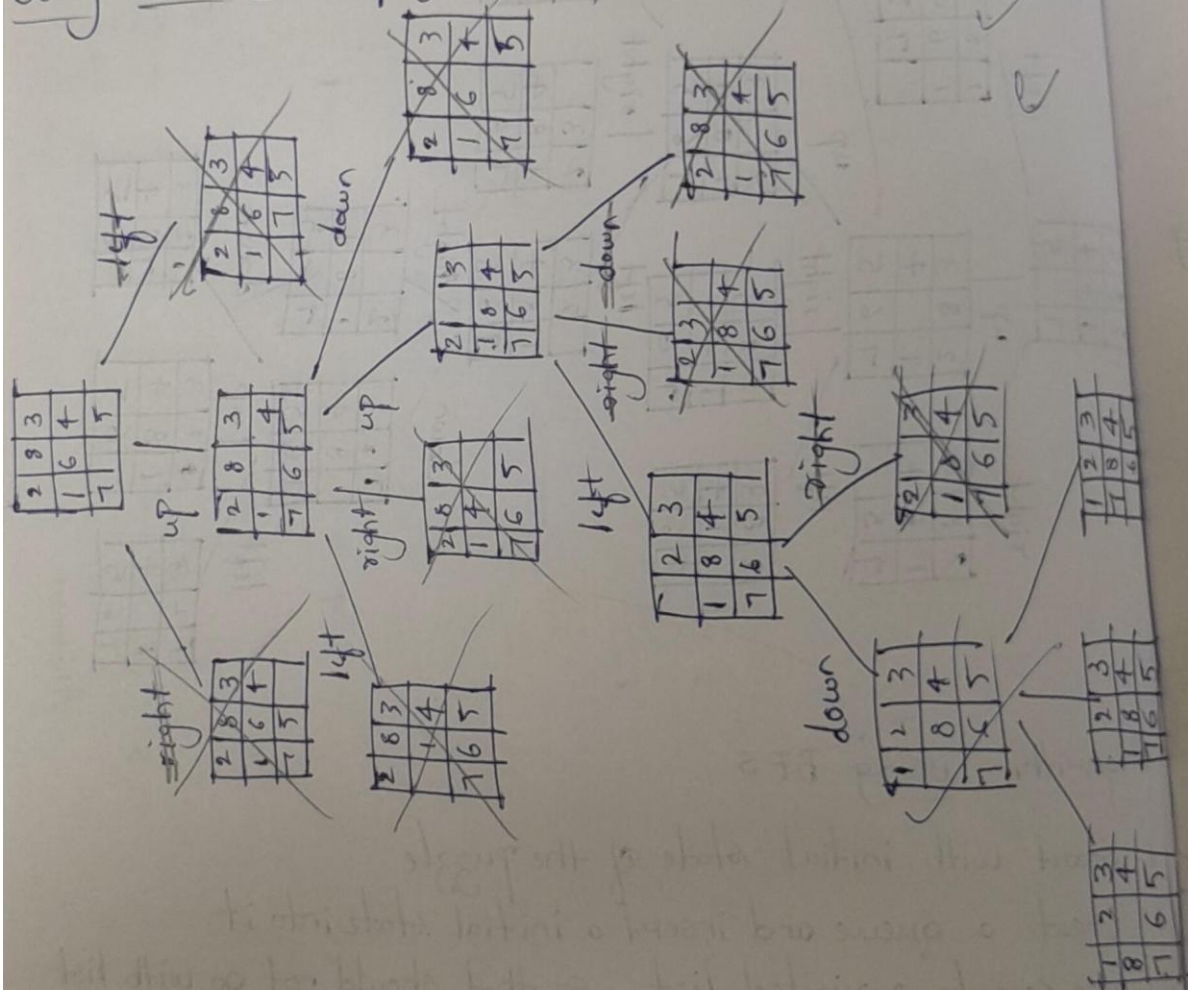
Possible → when Q finds empty, no solution
→ stops when goal is found

DFS solution

Algorithm using DFS

1. start
2. Create a stack and push the initial state
3. Create a visited list to track
4. generate all possible next states by moving the blank title: move up, down, right and left
5. ~~when stack find~~ stops when goal is found
6. stops when stack finds empty

using DFS solve 8 puzzle without heuristic approach



DFS output

solution found in 5 moves:

2	8	3
1	6	4
7	0	5

= initial

↓

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

= goal

BFS output

solution found in 5 moves

2	8	3
1	6	4
7	0	5

= initial

↓

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

↓

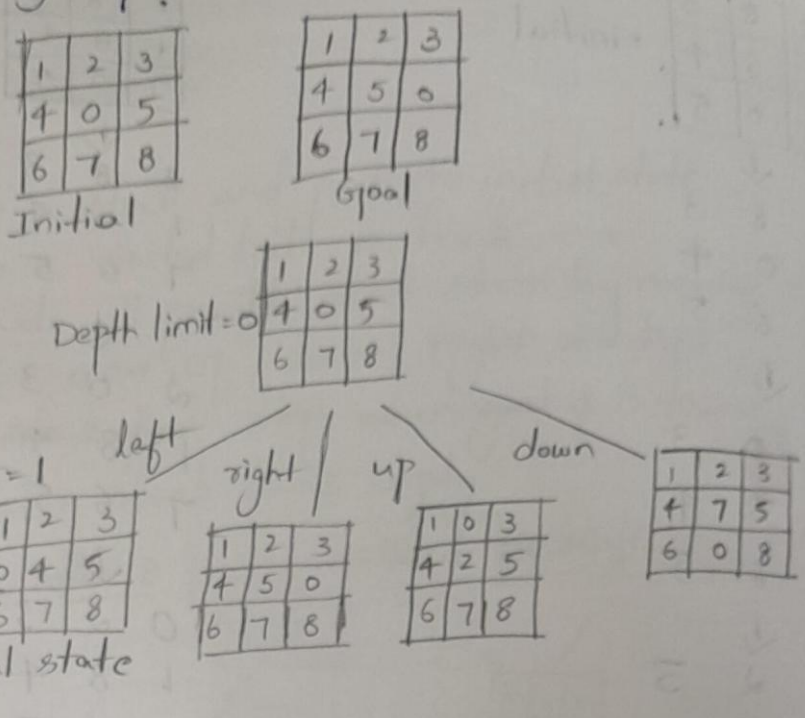
1 2 3
0 8 4
7 6 5

↓

1	2	3
8	0	4
7	6	5

= goal

Iterative Deepening search (IDS) or Iterative Deepening Depth First search (IDDFS)



Algorithm

1. Begin by searching for a solution that is 0 moves
2. if not found, search for solutions up to 1 move away
3. if still not found, search up to 2 moves away
4. keep increasing how deep you search until you find the solution

output:

searching with depth limit = 0

searching with depth limit = 1

solution found in 1 move

step 0:

1	2	3
4	0	5
6	7	8

step 1:

1	2	3
4	5	0
6	7	8

up

2		3
1	8	4
7	6	5

Code:
Usig DFS 8 puzzel without heuristic

```
# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))

# Moves: Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def get_neighbors(state):
    # Find the empty tile (0)
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break

    neighbors = []
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Swap empty tile with adjacent tile
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

def dfs_limited(start, depth_limit):
    stack = [(start, [start])]
    visited = set([start])

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

        if current == goal:
            return path

        if len(path) - 1 >= depth_limit:  # already reached depth limit
            continue

        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [neighbor]))

    return None
```

```
# Example start state
start = ((2, 8, 3),
         (1, 6, 4),
         (7, 0, 5))

solution_path = dfs_limited(start, depth_limit=5)

if solution_path:
    print(f"Solution found in {len(solution_path) - 1} moves:")
    for state in solution_path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found within 5 moves.")
```

OUTPUT

```
Solution found in 5 moves:
(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)
```

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

```
from copy import deepcopy
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 0],
    [6, 7, 8]
]

# Possible moves of the blank (0) tile: up, down, left, right
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]

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

def is_goal(state):
    return state == GOAL_STATE

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)

    for dx, dy in MOVES:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = deepcopy(state)
            # Swap blank with neighbor
```

```
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def dfs(state, depth, limit, path, visited):
    if is_goal(state):
        return path + [state]
    if depth == limit:
        return None

    for neighbor in get_neighbors(state):
        # To avoid cycles, do not revisit states in current path
        if neighbor not in visited:
            result = dfs(neighbor, depth + 1, limit, path + [state], visited + [neighbor])
            if result is not None:
                return result
    return None

def iterative_deepening_search(initial_state, max_depth=50):
    for depth_limit in range(max_depth):
        print(f"Searching with depth limit = {depth_limit}")
        result = dfs(initial_state, 0, depth_limit, [], [initial_state])
        if result is not None:
            return result
    return None

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

```
if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [6, 7, 8]
    ]

    solution = iterative_deepening_search(initial_state)

    if solution:
        print(f"Solution found in {len(solution)-1} moves!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            print_state(state)
    else:
        print("No solution found.")
```

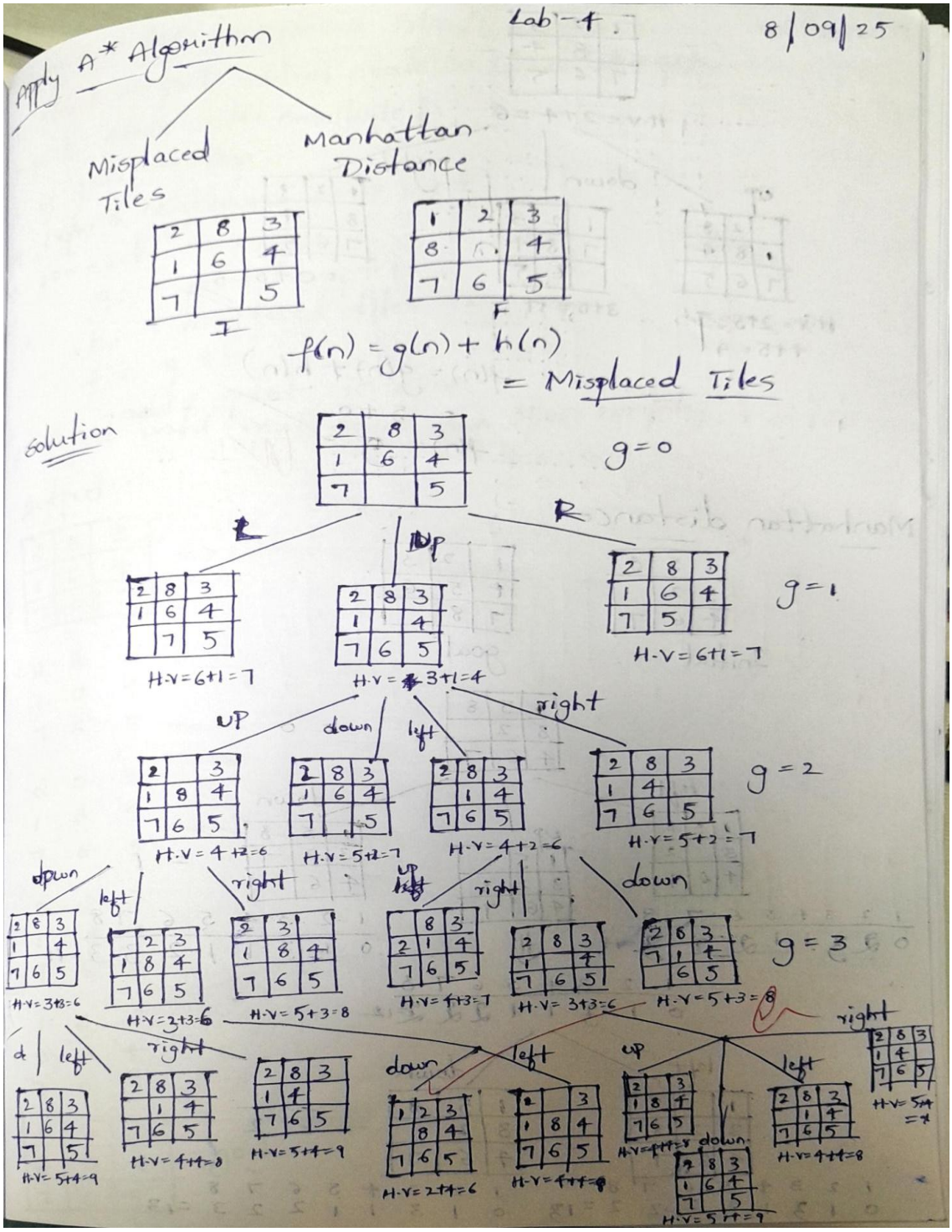
OUTPUT

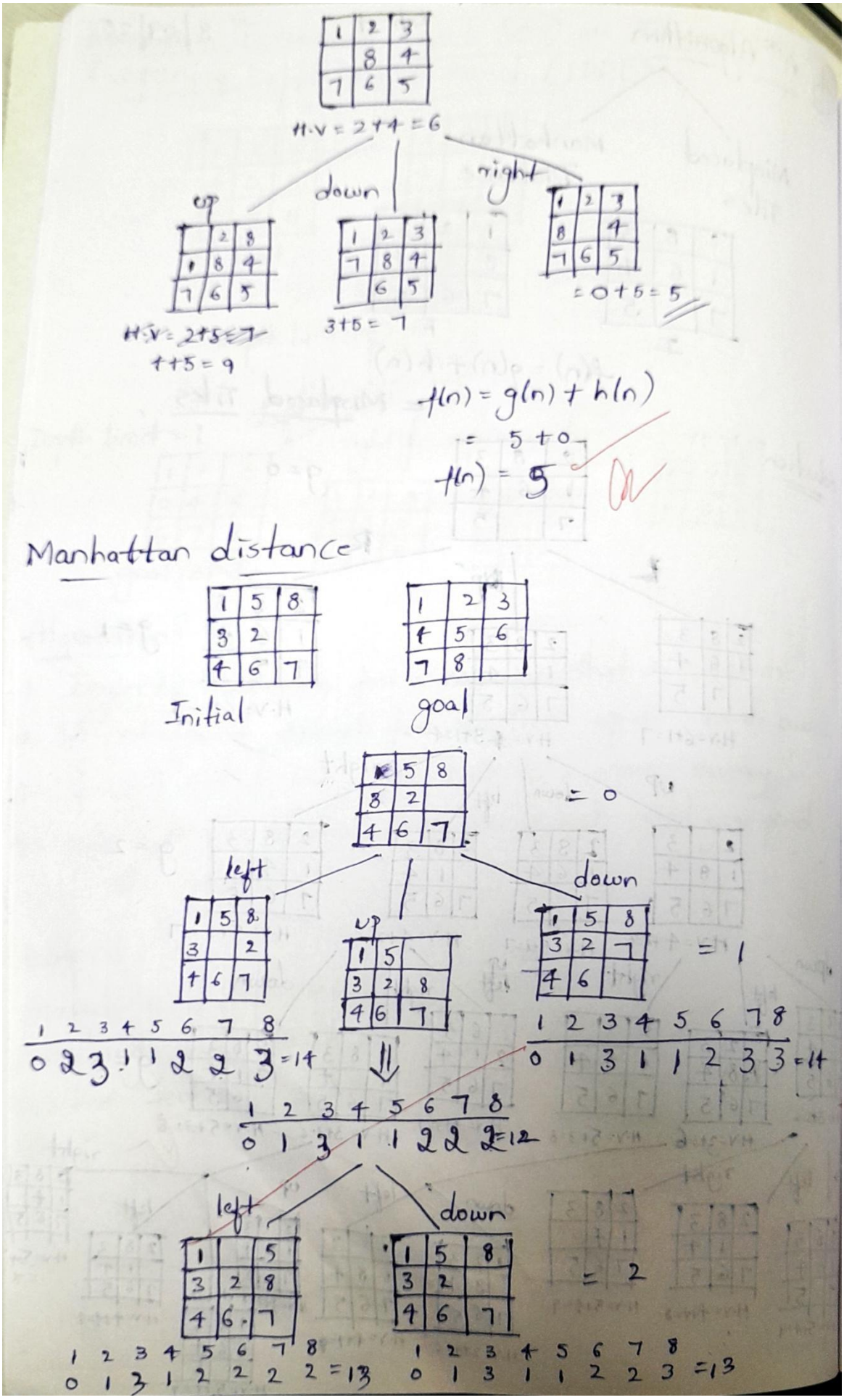
```
Searching with depth limit = 0
Searching with depth limit = 1
Solution found in 1 moves!
Step 0:
1 2 3
4 0 5
6 7 8

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

Program 3

Implement A* search algorithm

Algorithm:




H.V = 2+4 = 6
up
down
right
H.V = 2+5 = 7
1+5 = 9
3+5 = 7
= 0+5 = 5
f(n) = g(n) + h(n)
= 5 + 0
f(n) = 5
Manhattan distance
Initial
goal
= 0
left
up
down
= 1
1 2 3 4 5 6 7 8
0 2 3 1 1 2 2 3 = 14
0 1 3 1 1 2 3 3 = 14
0 1 3 1 1 2 2 2 = 12
left
down
= 2
0 1 3 1 2 2 2 2 = 13
0 1 3 1 1 2 2 3 = 13

Algorithm for (Misplaced Tiles)

1. start → put initial state in OPEN ($f = g+h$)
2. pick state with smallest f
3. if goal → stop
4. Expand neighbours (move blank)
5. For each neighbour
   - $g := \text{parent}.g + 1$
   - $h :=$ misplaced tiles
   - $f = g + h$
6. Repeat add/update in open until goal found ~~out~~ open empty

output:

2	8	3
1	6	4
7	0	5

⇒ initial

2	8	3
1	0	4
7	6	5

2	0	3
1	8	4
7	6	5

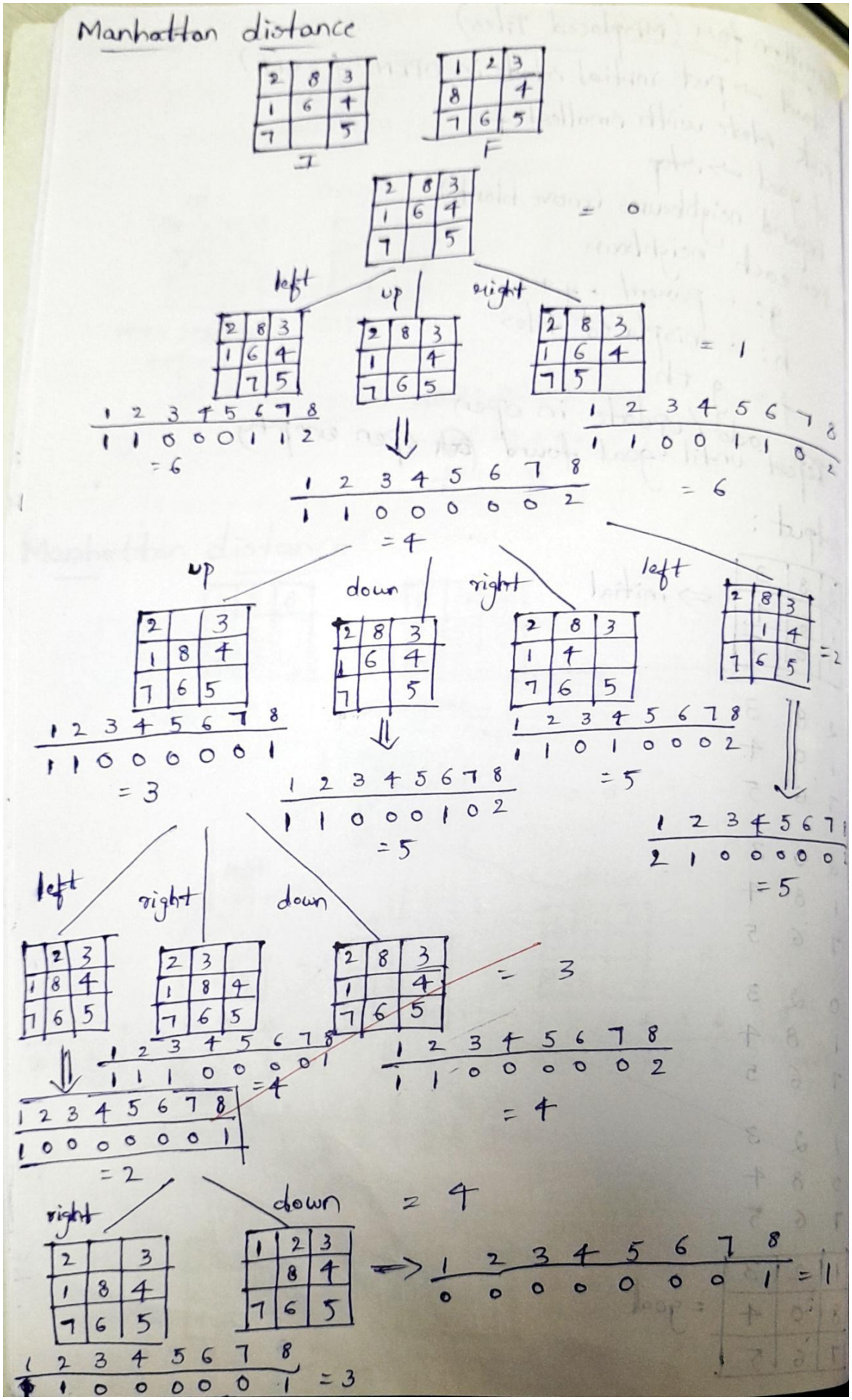
0	2	3
1	8	4
7	6	5

1	2	3
0	8	4
7	6	5

1	2	3
8	0	4
7	6	5

= goal

Manhattan distance



Code:
Misplace Tiles

```
import heapq


# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))


# Moves: Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]


# Heuristic: Misplaced tiles
def misplaced_tiles(state):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal[i][j]:
                count += 1
    return count


# Find blank position (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
# Generate neighbors
def get_neighbors(state):
    neighbors = []
```

```
    x, y = find_blank(state)

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state)
    return neighbors
# A* Search
def astar(start):
    pq = []
    heapq.heappush(pq, (misplaced_tiles(start), 0, start, []))
    visited = set()

    while pq:
        f, g, state, path = heapq.heappop(pq
        if state == goal:
            return path + [state]
        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)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None
```

```
# Example usage
start_state = ((2, 8, 3),
               (1, 6, 4),
               (7, 0, 5))

solution = astar(start_state)

# Print solution path
for step in solution:
    for row in step:
        print(row)
    print("------")
```

OUTPUT

```
(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)
------
```

Manhatten:

```
import heapq
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))


moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                # goal position of this tile
                goal_x = (value - 1) // 3
                goal_y = (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance
def find_blank(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_blank(state)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
```

```
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors
def astar(start):
    pq = []
    heapq.heappush(pq, (manhattan_distance(start), 0, start, []))  # (f, g, state, path)
    visited = set()
    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == goal:
            return path + [state]
        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_distance(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None
start_state = ((2, 8, 3),
               (1, 6, 4),
               (7, 0, 5))
solution = astar(start_state)

if solution is None:
    print("No solution found.")
else:
```

```
    print("Solution path:")
    for step in solution:
        for row in step:
            print(row)
        print("------")
```

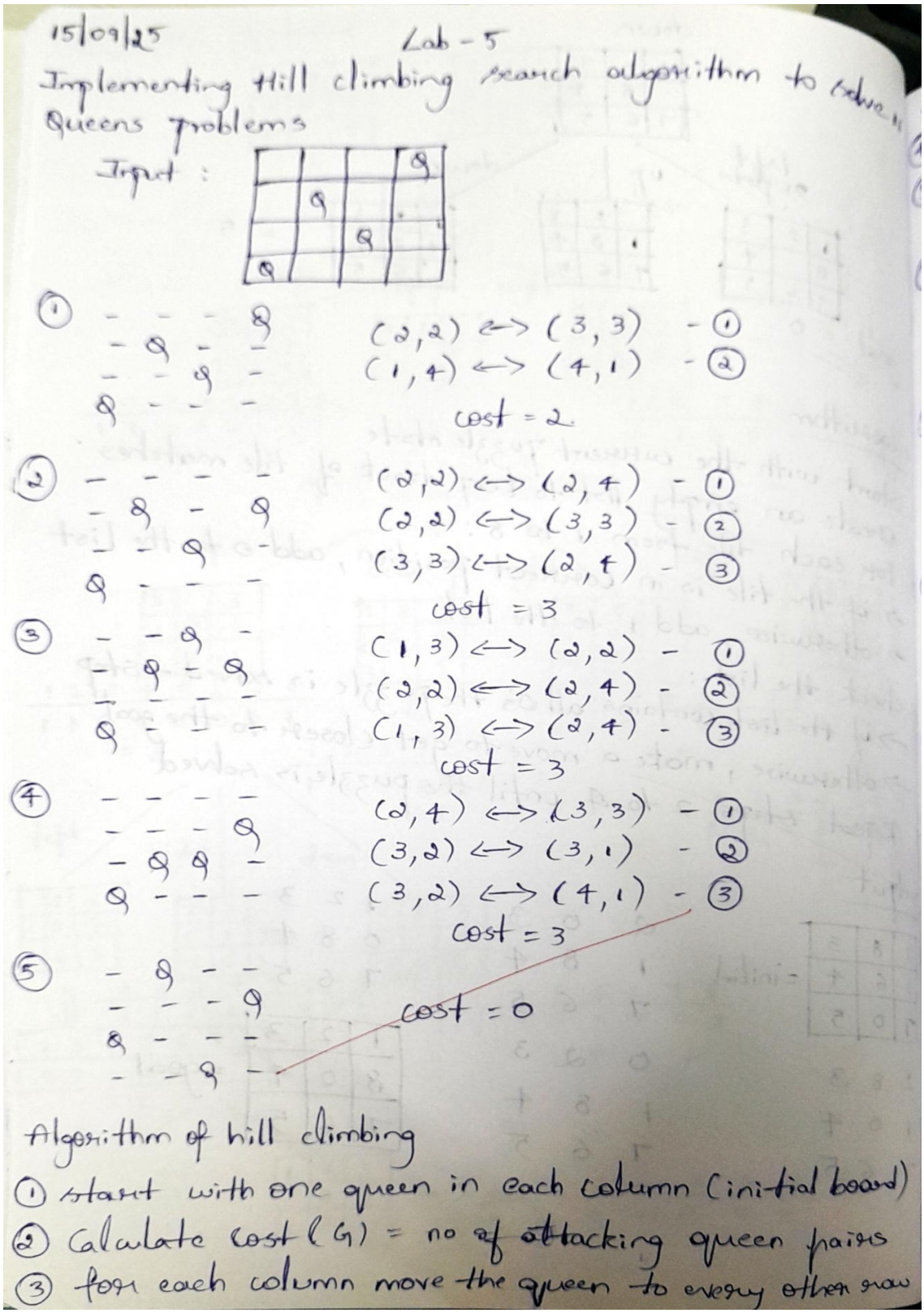
OUTPUT:

```
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)
------
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



and compute next cost.
③ choose the move that gives the lowest cost (best neighbour)
④ if best cost < current cost, move queen there and repeat step 2
⑤ if no neighbour has lowest cost, stop

output:
initial stack

① - - - Q cost = 2
- Q - -
- - Q -
Q - - -

② - - - Q cost = 2
- Q - -
- - - -
Q - Q -

③ - - - Q cost = 1
- Q - -
Q - - -
- - Q -

④ - Q - Q cost = 1
- - - -
Q - - -
- - Q -

⑤ - Q - - cost = 0
- - - Q
Q - - -
- - Q -

sol found in 4 steps

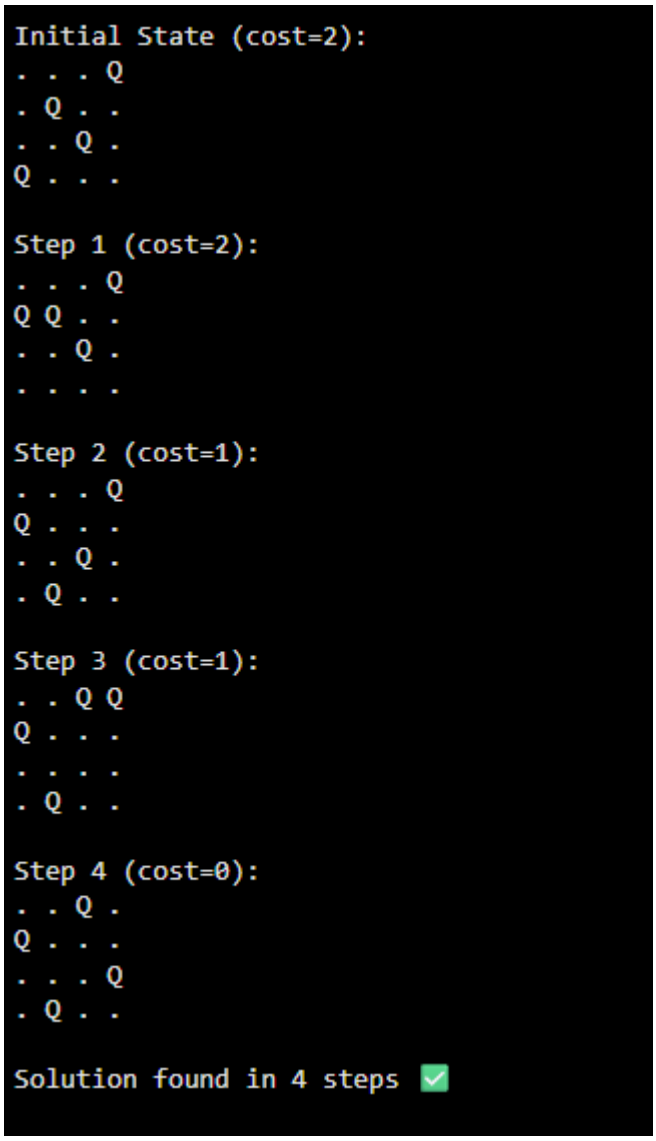
Code:

```
import random


def compute_cost(state):
    n = len(state)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:              # same row
                cost += 1
            elif abs(state[i] - state[j]) == abs(i - j):  # same diagonal
                cost += 1
    return cost



def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):        # pick a column
        for row in range(n):    # try moving queen in this column to another row
            if row != state[col]:
                new_state = state.copy()
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors




def print_board(state):
    n = len(state)
```

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


def hill_climb(initial_state, max_sideways=50):
    current = initial_state
    current_cost = compute_cost(current)
    steps = 0
    sideways_moves = 0

    print("Initial State (cost={}):".format(current_cost))
    print_board(current)

    while True:
        neighbors = get_neighbors(current)
        costs = [compute_cost(n) for n in neighbors]
        min_cost = min(costs)

        if min_cost > current_cost:
            # no better neighbor -> stop
            break

        # pick one of the best neighbors randomly
        best_neighbors = [n for n, c in zip(neighbors, costs) if c == min_cost]
        next_state = random.choice(best_neighbors)
        next_cost = compute_cost(next_state)
```

```
        # handle sideways moves
        if next_cost == current_cost:
            if sideways_moves >= max_sideways:
                break
            else:
                sideways_moves += 1
        else:
            sideways_moves = 0

        current = next_state
        current_cost = next_cost
        steps += 1

        print("Step {} (cost={}):".format(steps, current_cost))
        print_board(current)

        if current_cost == 0:
            print("Solution found in {} steps ☑".format(steps))
            return current

    print("Local minimum reached (cost={}) ❌".format(current_cost))
    return current

initial_state = [3, 1, 2, 0]

final = hill_climb(initial_state, max_sideways=10)
```

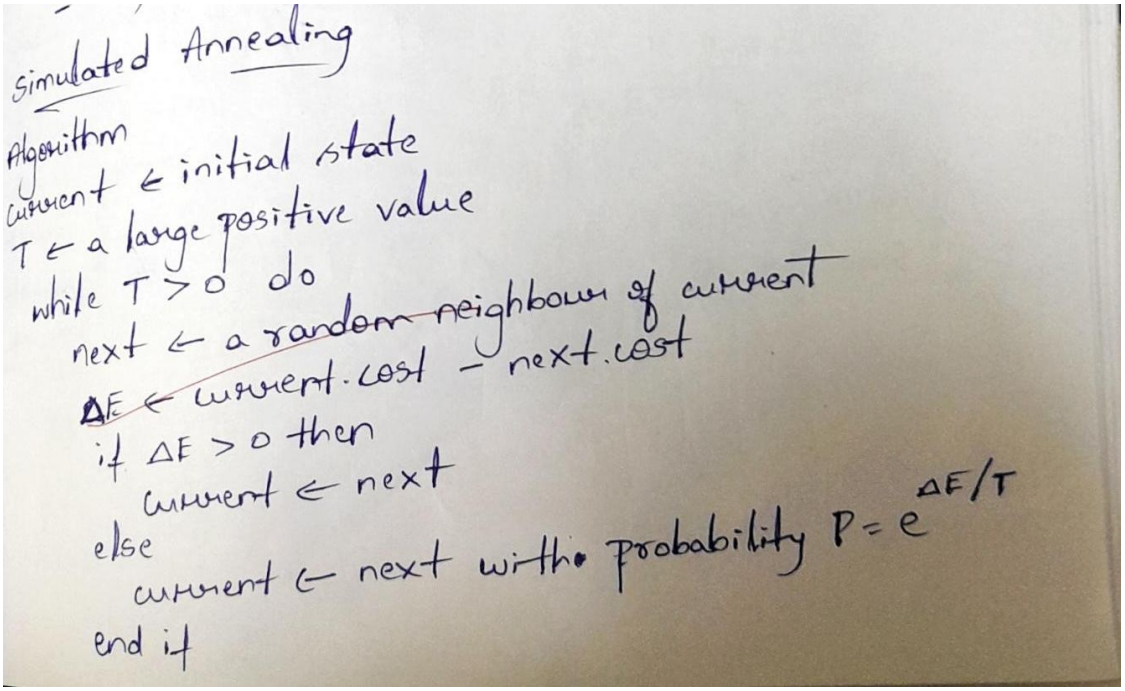
OUTPUT:

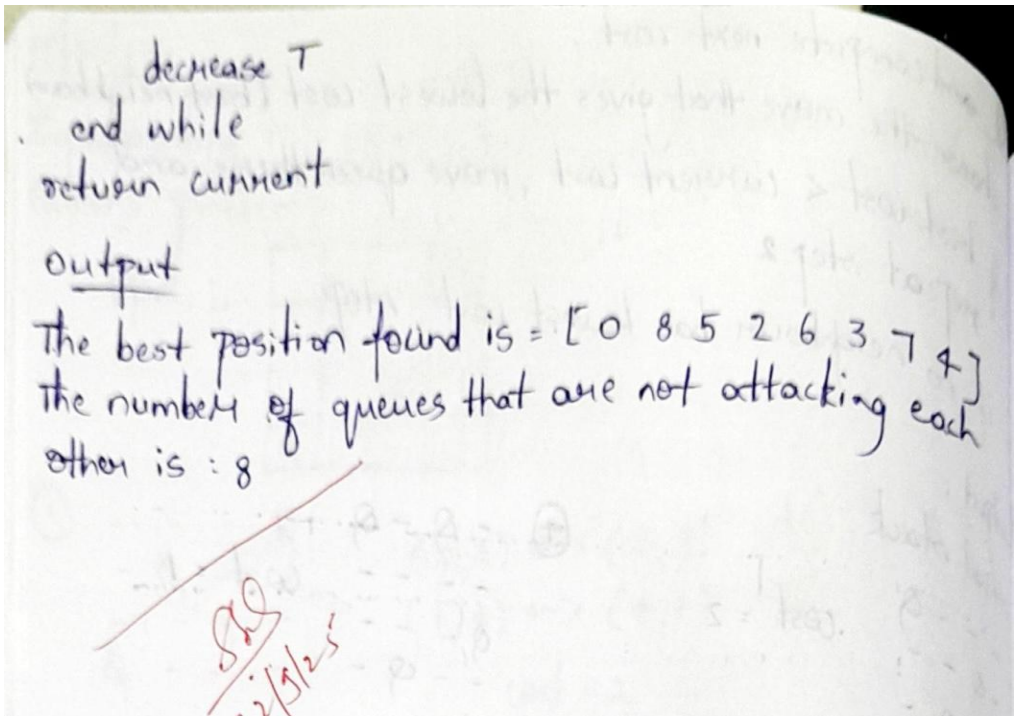

Initial State (cost=2):
. . . Q
. Q . .
. . Q .
Q . . .
Step 1 (cost=2):
. . . Q
Q Q . .
. . Q .
. . . .
Step 2 (cost=1):
. . . Q
Q . . .
. . Q .
. Q . .
Step 3 (cost=1):
. . Q Q
Q . . .
. . . .
. Q . .
Step 4 (cost=0):
. . Q .
Q . . .
. . . Q
. Q . .
Solution found in 4 steps

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:





Code:

```
from scipy.optimize import dual_annealing
import numpy as np

def queens_max(x):
    cols = np.round(x).astype(int)
    n = len(cols)

    if len(set(cols)) < n:
        return 1e6
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if abs(i - j) == abs(cols[i] - cols[j]):
                attacks += 1
    return attacks
n = 8
bounds = [(0, n - 1)] * n
result = dual_annealing(queens_max, bounds)

best_cols = np.round(result.x).astype(int).tolist()
not_attacking = n

print(f"The best position found is: {best_cols}")
print(f"The number of queens that are not attacking each other is: {not_attacking}")
```

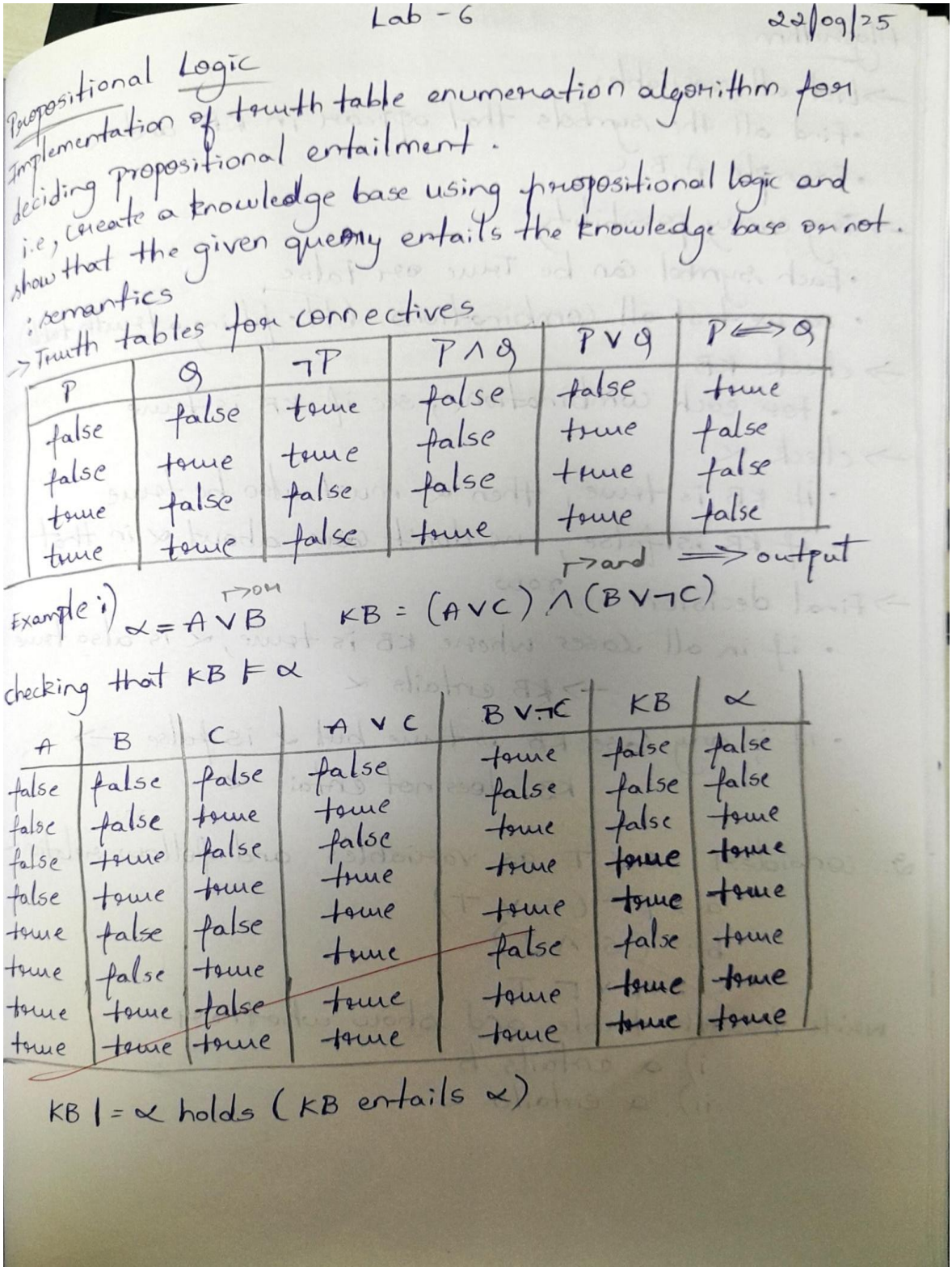
OUTPUT:

```
The best position found is: [7, 4, 6, 1, 3, 5, 0, 2]
The number of queens that are not attacking each other is: 8
```

Program 6

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

Algorithm:



Lab - 6 22/09/25

Propositional Logic

Implementation of truth table enumeration algorithm for deciding propositional entailment.

i.e, create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

: semantics

→ Truth tables 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

↳ and ⟹ output

Example i) $\alpha = A \vee B$ (↳ OR) $KB = (A \vee C) \wedge (B \vee \neg C)$

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

$KB \models \alpha$ holds (KB entails α)

Algorithm

→ List all variables
- Find all the symbols that appear in KB and α
- Example A, B, C

→ Try every possibility
- Each symbol can be True or False
- So we test all combinations (like filling a truth table)

→ check KB
- For each combination, see if KB is true

→ check α
- if KB is true, then α must also be true
- if KB is false, we don't care about α in that row

→ Final decision
- if in all cases where KB is true, α is also true
  → KB entails α
- if in any case KB is true but α is false →
  KB does not entail α

2. consider $S \in T$ as variables and follow relation

a : $\neg (S \vee T)$

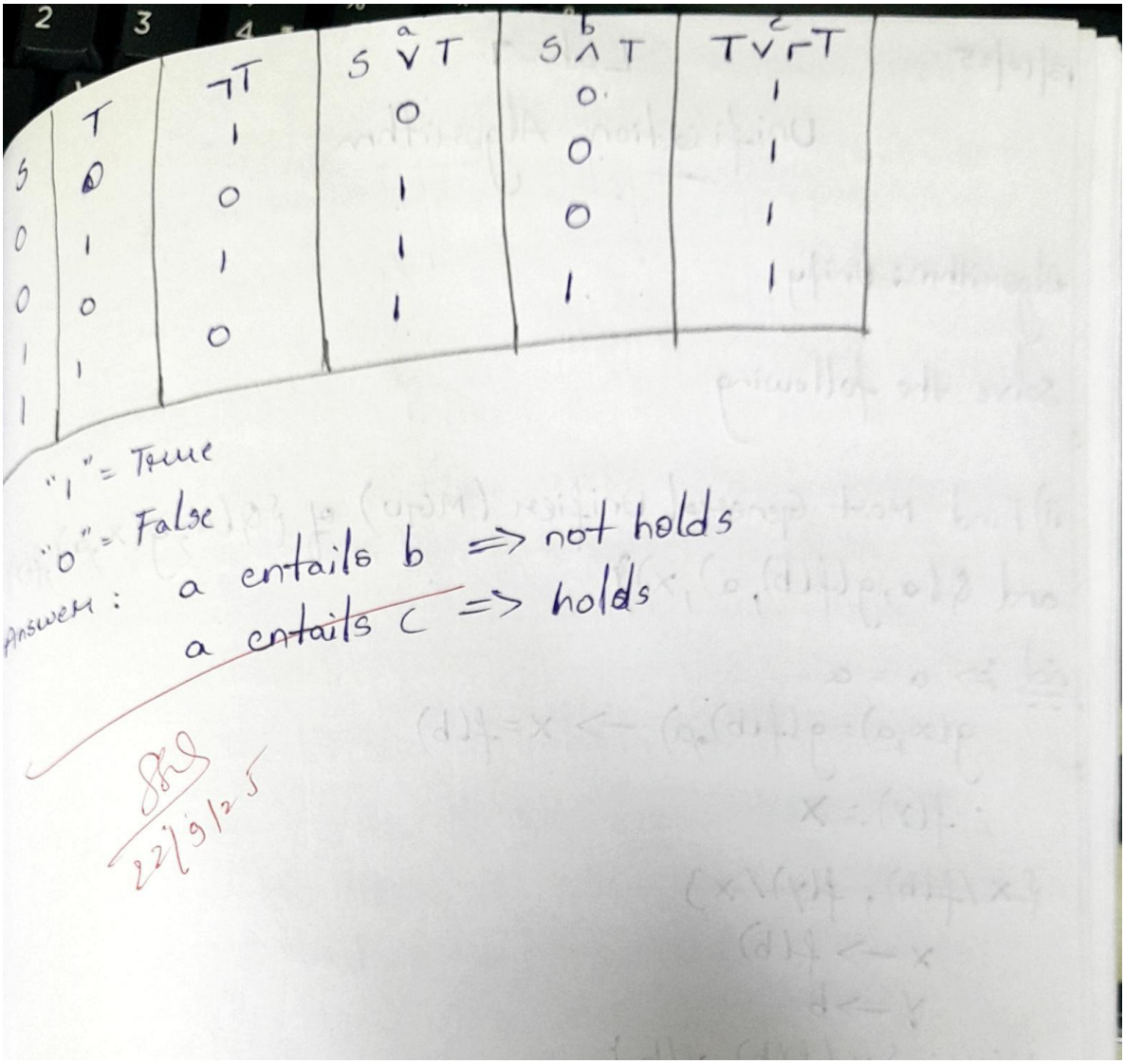
b : $(S \wedge T)$

c : $\top \vee \neg \top$

write truth table and show whether

i) a entails b

ii) a entails c



S	T	¬T	S ∨ T (a)	S ∧ T (b)	T ∨ ¬T (c)
0	0	1	0	0	1
0	1	0	1	0	1
1	0	1	1	0	1
1	1	0	1	1	1

"1" = True

"0" = False

Answer: a entails b => not holds

a entails c => holds

22/9/25

Code:

```
import itertools
from sympy import symbols, sympify

A, B, C = symbols('A B C')

alpha_input = input("Enter alpha (example: A | B): ")
kb_input = input("Enter KB (example: (A | C) & (B | ~C)): ")

alpha = sympify(alpha_input, evaluate=False)
kb = sympify(kb_input, evaluate=False)

GREEN = "\033[92m"
RESET = "\033[0m"

print(f"\nTruth Table for α = {alpha_input}, KB = {kb_input}\n")
print(f"{'A':<6}{'B':<6}{'C':<6}{'α':<10}{'KB':<10}")

entailed = True

for values in itertools.product([False, True], repeat=3):
    subs = {A: values[0], B: values[1], C: values[2]}
    alpha_val = alpha.subs(subs)
    kb_val = kb.subs(subs)

    alpha_str = f"{GREEN}{alpha_val}{RESET}" if kb_val else str(alpha_val)
    kb_str = f"{GREEN}{kb_val}{RESET}" if kb_val else str(kb_val)

    print(f"{str(values[0]):<6}{str(values[1]):<6}{str(values[2]):<6}"
          f"{alpha_str:<10}{kb_str:<10}")
```

```
    if kb_val and not alpha_val:
        entailed = False

if entailed:
    print(f"\n KB |= α holds (KB entails α)\n")
else:
    print(f"\n KB does NOT entail α\n")
```

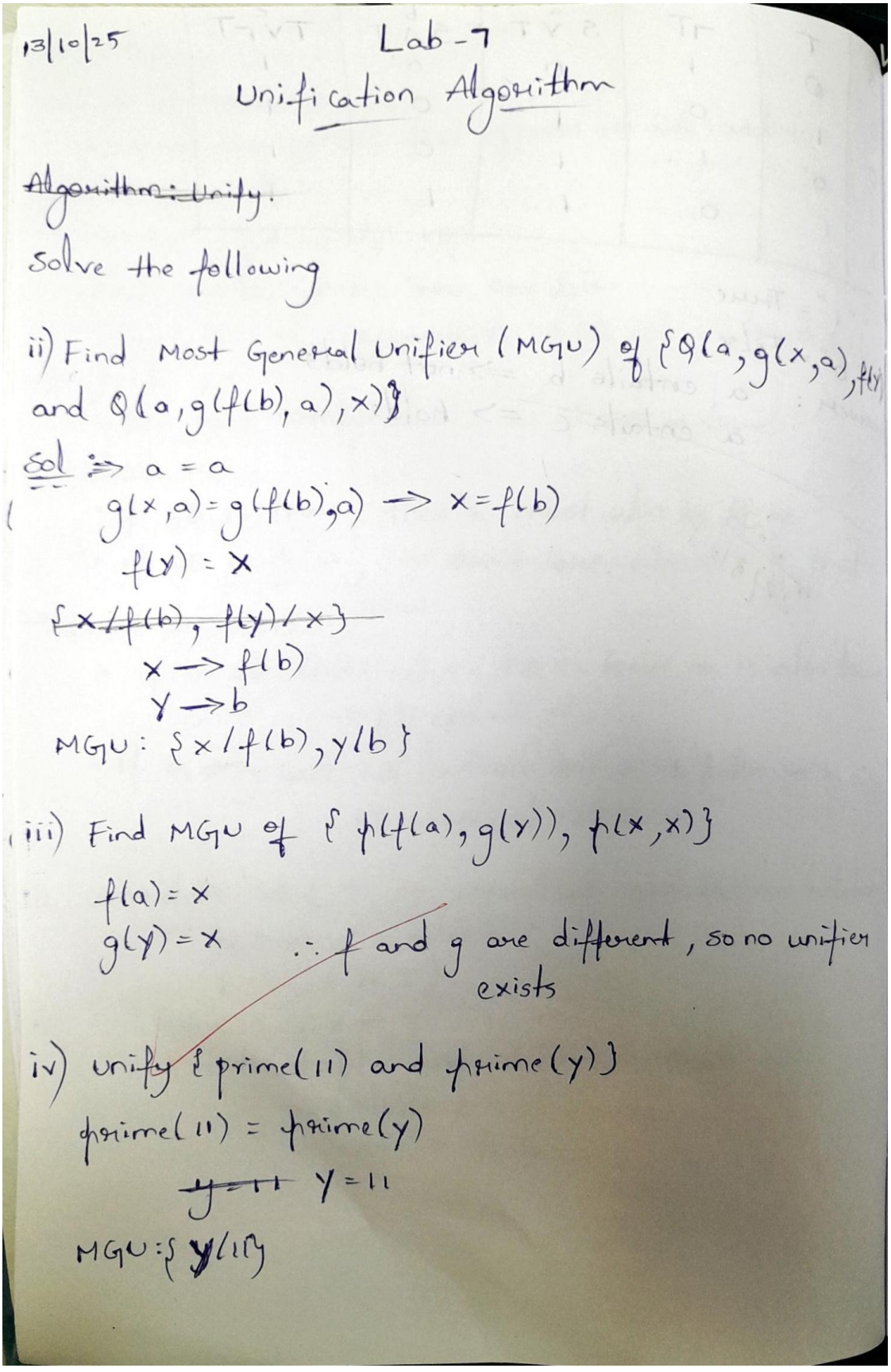
OUTPUT:

```
Enter alpha (example: A | B): A|B
Enter KB (example: (A | C) & (B | ~C)): (A | C) & (B | ~C)

Truth Table for α = A|B, KB = (A | C) & (B | ~C)

A     B     C     α         KB
False False False False     False
False False True  False     False
False True  False True      False
False True  True  TrueTrue
True  False False TrueTrue
True  False True  True      False
True  True  False TrueTrue
True  True  True  TrueTrue

 KB |= α holds (KB entails α)
```

Program 7

Implement unification in first order logic

Algorithm:



13/10/25

Lab - 7

Unification Algorithm

~~Algorithm: Unify.~~

Solve the following

ii) Find Most General Unifier (MGU) of {Q(a, g(x,a), f(y)) and Q(a, g(f(b), a), x)}

Sol ⇒ a = a

g(x,a) = g(f(b),a) → x = f(b)

f(y) = x

~~{x/f(b), f(y)/x}~~

x → f(b)

y → b

MGU: {x/f(b), y/b}

iii) Find MGU of {p(f(a), g(y)), p(x,x)}

f(a) = x

g(y) = x ∴ f and g are different, so no unifier exists

iv) unify {prime(11) and prime(y)}

prime(11) = prime(y)

~~y = 11~~ y = 11

MGU: {y/11}

v) unify: {knows(John, x), knows(y, mother(y))}

knows(John) = y

y = John → John = y

knows(x) = knows(mother(y))

x = mother(John)

MGU: {y/John, x/mother(John)}

vi) unify: {knows(John, x), knows(y, Bill)}

knows(John) = knows(y)

y = John

knows(x) = knows(Bill)

x = Bill

MGU: {y/John, x/Bill}

i) Find MGU of {p(b, x, f(g(z))) and p(z, f(y), f(y))}

b = z → z = b

x → f(y)

f(g(z)) = f(y) → g(z) = y

MGU: {z/b, x/f(y), y/g(z)}

Algorithm

Algorithm: Unify(Ψ_1, Ψ_2)

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

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

b) Else if Ψ_1 is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return $\{(\psi_1 / \psi_1)\}$,

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 in ψ_1 and ψ_2 are not same, then return FAILURE

step 3 : IF ψ_1 and ψ_2 have a different number of argument, 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

output (i)

MGU : {'b': 'z', 'x': ('f', 'y'), 'y': ('g', 'z')}

Code:

```
def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        return any(occurs_check(var, sub) for sub in expr[1:])  # Skip function symbol
    return False
def substitute(expr, subst):
    if isinstance(expr, str):
        # Follow substitution chain until fully resolved
        while expr in subst:
            expr = subst[expr]
        return expr
    # If it's a function term: (f, arg1, arg2, ...)
    return (expr[0],) + tuple(substitute(sub, subst) for sub in expr[1:])
def unify(Y1, Y2, subst=None):
    if subst is None:
        subst = {}
    Y1 = substitute(Y1, subst)
    Y2 = substitute(Y2, subst)

    # Case 1: identical
    if Y1 == Y2:
        return subst

    # Case 2: Y1 is variable
    if isinstance(Y1, str):
        if occurs_check(Y1, Y2):
            return "FAILURE"
        subst[Y1] = Y2
        return subst
```

```
    # Case 3: Y2 is variable
    if isinstance(Y2, str):
        if occurs_check(Y2, Y1):
            return "FAILURE"
        subst[Y2] = Y1
        return subst

    # Case 4: function mismatch
    if Y1[0] != Y2[0] or len(Y1) != len(Y2):
        return "FAILURE"

    # Case 5: unify arguments
    for a, b in zip(Y1[1:], Y2[1:]):
        subst = unify(a, b, subst)
        if subst == "FAILURE":
            return "FAILURE"

    return subst
expr1 = ("p", "X", ("f", "Y"))
expr2 = ("p", "a", ("f", "b"))

output = unify(expr1, expr2)
print(output)
```

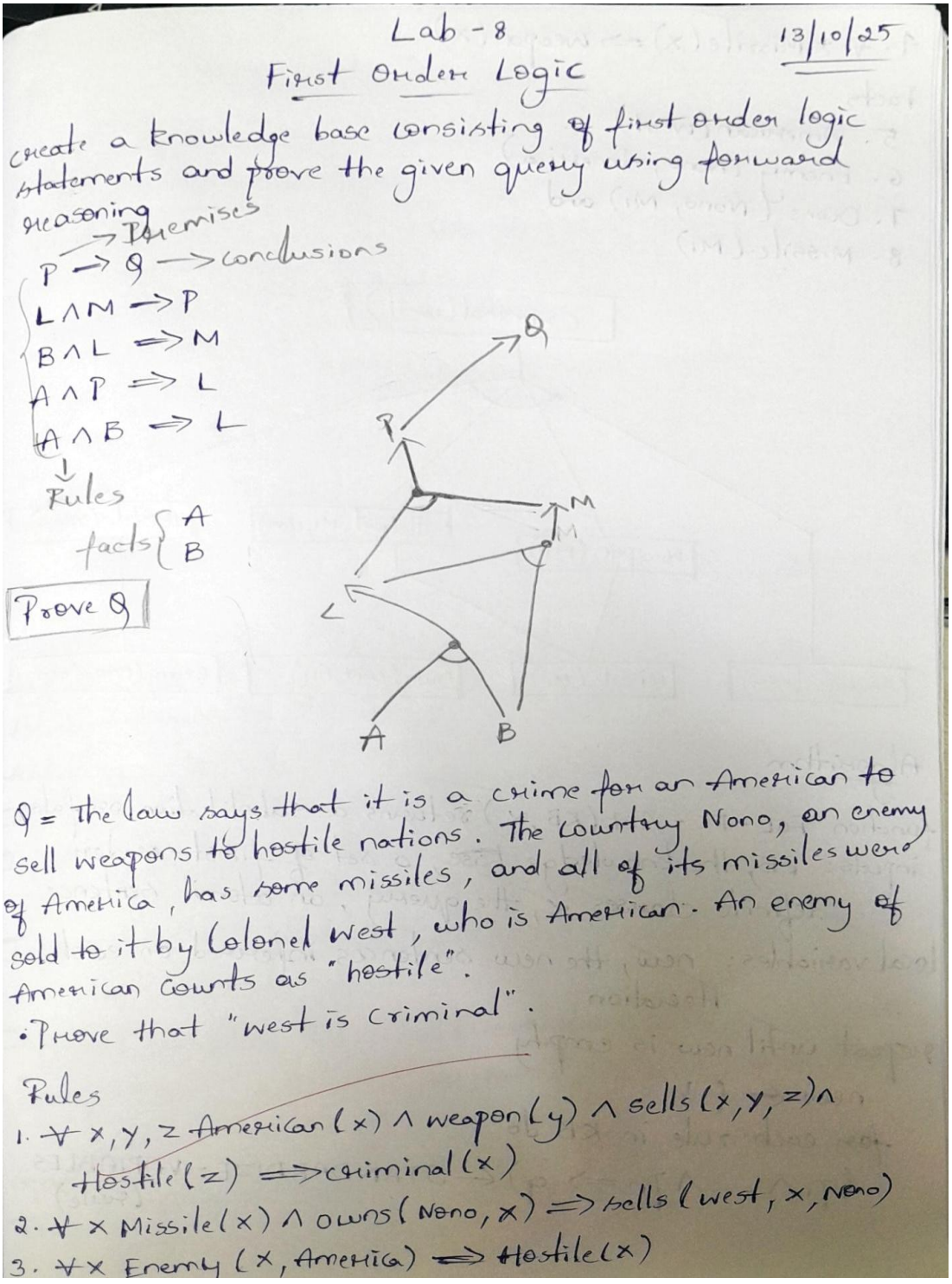
OUTPUT:

```
{'X': 'a', 'Y': 'b'}
```

Program 8

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

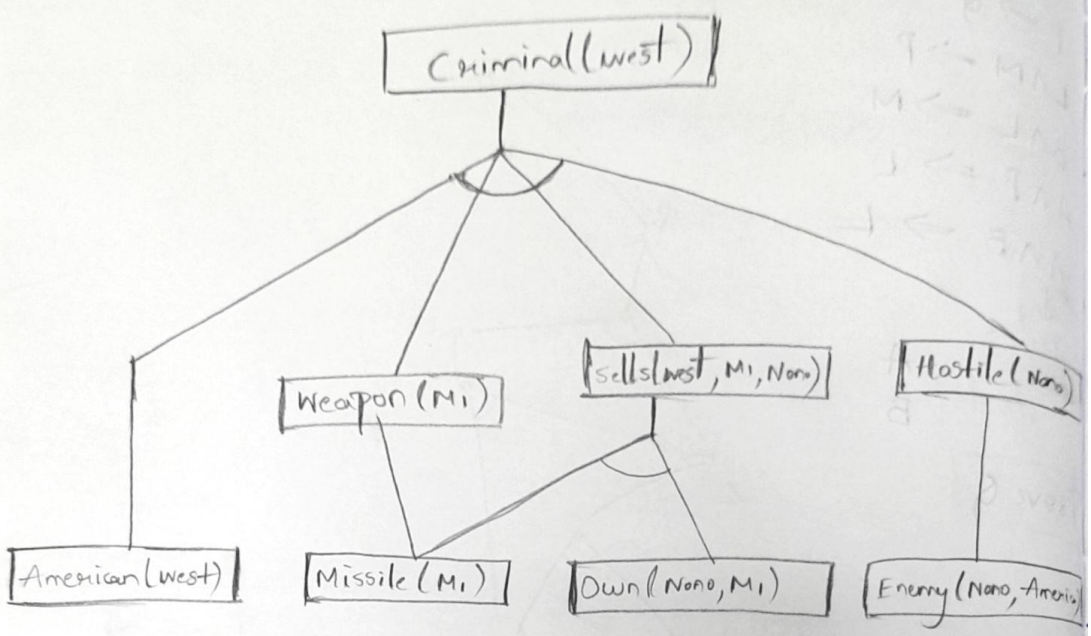
Algorithm:



4. $\forall x \; Missile(x) \Rightarrow Weapon(x)$

Facts

5. American(west)
6. Enemy(Nono, America)
7. Owns(Nono, M1) and
8. Missile(M1)



Algorithm

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

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

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

repeat until new is empty

new $\leftarrow$ { }

for each rule in KB do

$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ STANDARDIZE-VARIABLES (rule)

for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
for some p'1, ..., p'n in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new
φ ← UNIFY(q', α)
if φ is not fail then return φ
add new to KB
return false

output:

Adding fact : American(west)
Adding fact : Enemy(Nono, America)
Adding fact : Missile(M1)
Adding fact : Owns(Nono, M1)
Inferred new fact : weapon(M1) from ['Missile(M1)'] => weapon(M1)
Inferred new fact : Sells(west, M1, Nono) from ['missile(M1)', 'Owns(Nono, M1)'] => sells(west, M1, Nono)
Inferred new fact : Hostile(Nono) from ['Enemy(Nono, America)'] => Hostile(Nono)
Inferred new fact : Criminal(west) from ['American(west)', 'weapon(M1)', 'Sells(west, M1, Nono)', 'Hostile(Nono)'] => Criminal(west)
Goal Reached : west is Criminal
True

13/10/21

Code:

```
from collections import deque
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
        self.inferred = set()

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

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

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

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

            for (premises, conclusion) in self.rules:
                if all(p in self.inferred for p in premises):
                    if conclusion not in self.facts:
```

```
                print(f"Inferred new fact: {conclusion} from {premises} => {conclusion}")
                self.facts.add(conclusion)
                agenda.append(conclusion)

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

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

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

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

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

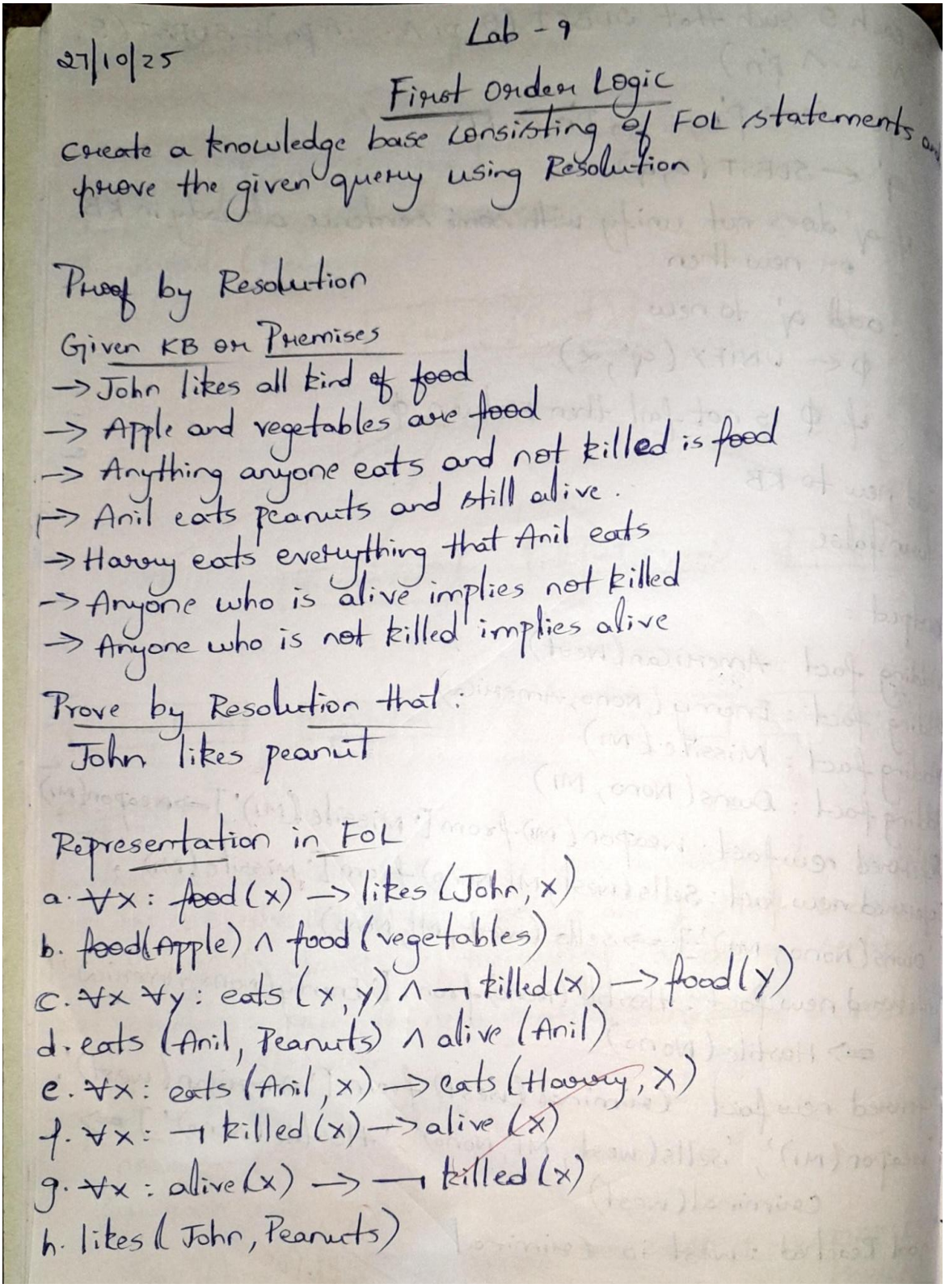
OUTPUT:

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

✅ Goal Reached: West is Criminal
True
```

Program 9

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

Algorithm:



Lab - 9

27/10/25

First Order Logic

Create a knowledge base consisting of FOL statements and prove the given query using Resolution

Proof by Resolution

Given KB or Premises

-> John likes all kind of food
-> Apple and vegetables are food
-> Anything anyone eats and not killed is food
-> Anil eats peanuts and still alive.
-> Harry eats everything that Anil eats
-> Anyone who is alive implies not killed
-> Anyone who is not killed implies alive

Prove by Resolution that:
John likes peanut

Representation in FOL

a. $\forall x : food(x) \rightarrow likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f. $\forall x : \neg killed(x) \rightarrow alive(x)$
g. $\forall x : alive(x) \rightarrow \neg killed(x)$
h. $likes(John, Peanuts)$

• Eliminate implication

a. $\forall x \neg food(x) \vee likes(John, x)$

and b. $food(Apple) \wedge food(vegetables)$

c. $\forall x \forall y \neg [eats(x,y) \wedge \neg killed(x)] \vee food(y)$

d. $eats(Anil, Peanuts) \wedge alive(Anil)$

e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$

f. $\forall x \neg [\neg killed(x)] \vee alive(x)$

g. $\forall x \neg alive(x) \vee \neg killed(x)$

h. $likes(John, Peanuts)$

• Move negation ($\neg$) inwards and rewrite

a. $\forall x \neg food(x) \vee likes(John, x)$

b. $food(Apple) \wedge food(vegetables)$

c. $\forall x \forall y \neg eats(x,y) \vee killed(x) \vee food(y)$

d. $eats(Anil, Peanuts) \wedge alive(Anil)$

e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$

f. $\forall x killed(x)] \vee alive(x)$

g. $\forall x \neg alive(x) \vee \neg killed(x)$

h. $likes(John, Peanuts)$

• Rename variable or standardize variables

a. $\forall x \neg food(x) \vee likes(John, x)$

b. $food(Apple) \wedge food(vegetables)$

c. $\forall y \forall z \neg eats(y, z) \vee killed(y) \vee food(z)$

d. $eats(Anil, Peanuts) \wedge alive(Anil)$

e. $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$

f. $\forall g killed(g)] \vee alive(g)$

g. $\forall k \neg alive(k) \vee \neg killed(k)$

h. $likes(John, Peanuts)$

• Drop universe

a. $\neg food(x) \vee likes(John, x)$

b. $food(Apple)$

c. $food(vegetables)$

d. $\neg eats(y, z) \vee killed(y) \vee food(z)$

e. $eats(Anil, Peanuts)$

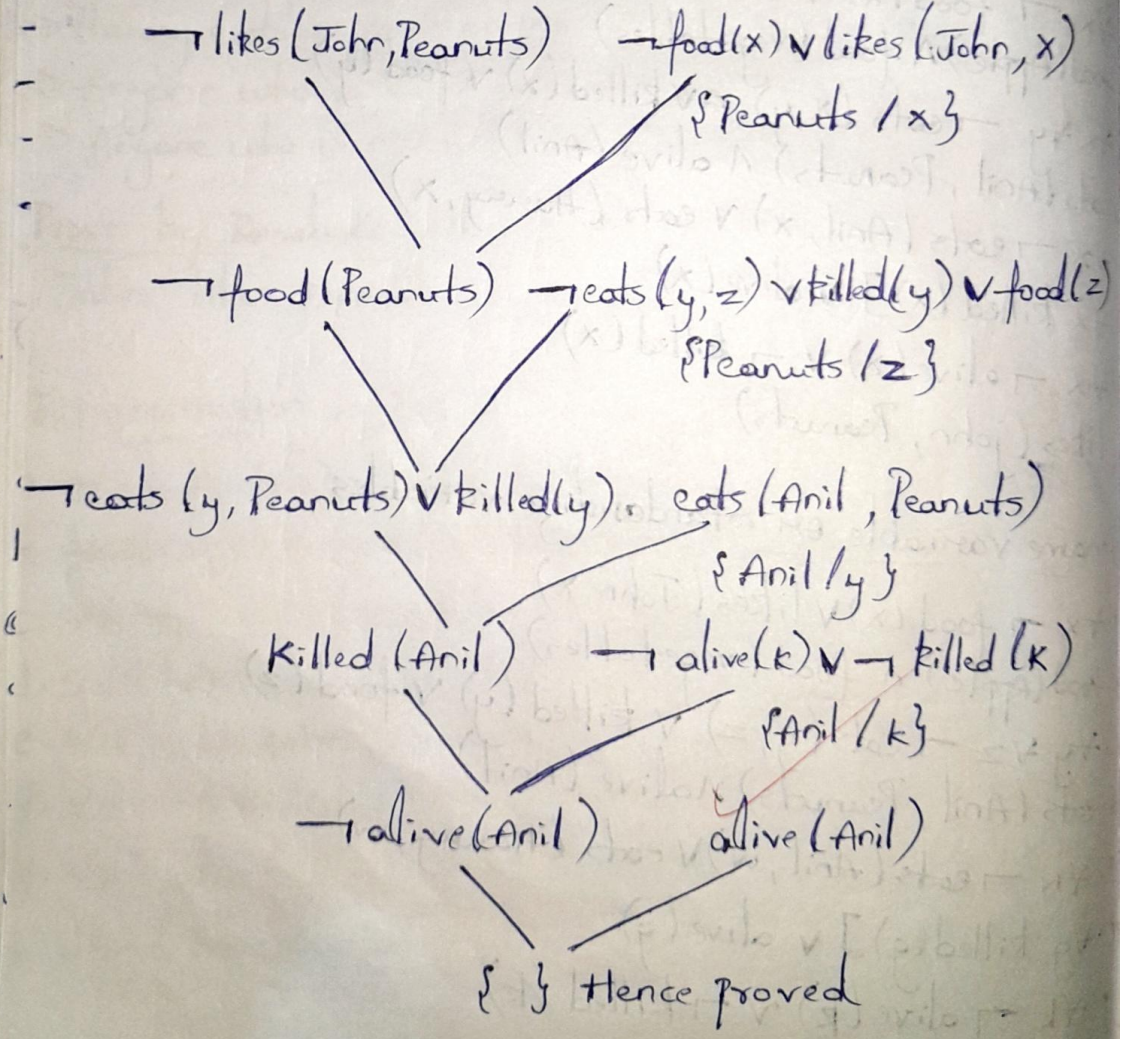
f. $alive(Anil)$

g. $\neg eats(Anil, w) \vee eats(Harry, w)$

h. $killed(g) \vee alive(g)$

i. $\neg alive(k) \vee \neg killed(k)$

j. $likes(John, Peanuts)$



Algorithm

1. Input:
   - Knowledge Base (KB)
   - Query (Q)
2. Convert KB and $\neg Q$ to clausal form:
   - Eliminate implications
   - Move negations inward
   - standardize variables
   - skolemize (remove $\exists$ quantifiers)
   - Drop universal quantifiers
   - Convert to CNF.
3. Apply Resolution:
   - Repeatedly resolve pairs of clauses that contain complementary literals
   - Add new clauses to the KB
   - stop if:
     - Empty clause ($\perp$) is derived $\rightarrow Q$ is true
     - No new clauses can be added $\rightarrow Q$ is False
4. output True/False for query Q

Code:

```
from itertools import combinations

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

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

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

    step = 1
    while True:
        new = []
        for (ci, cj) in combinations(kb, 2):
```

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

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


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

    def show(node, indent=0):
        if node not in tree:
            print(" " * indent + node)
            return
```

```
        parents, clause = tree[node]
        print(" " * indent + f"{node}: {set(clause) or '{}'}")
        for p in parents:
            show(p, indent + 4)

    show(goal)
```

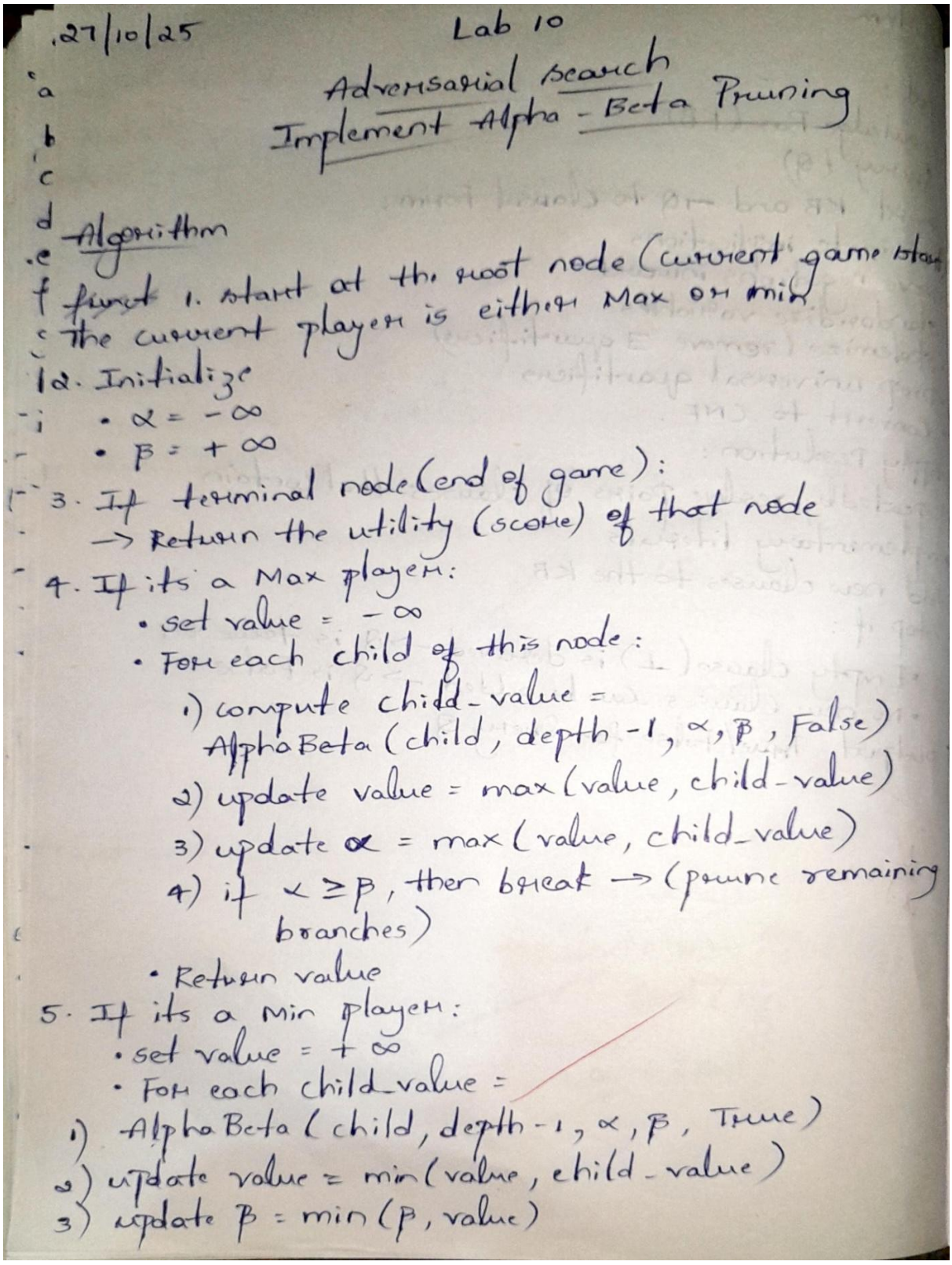
OUPUT:

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

✅ Query is proved by resolution (empty clause found).

--- Proof Tree ---
R3: {}
    C1
    R2: {'~P'}
        C2
        C4
True
```

Program 10

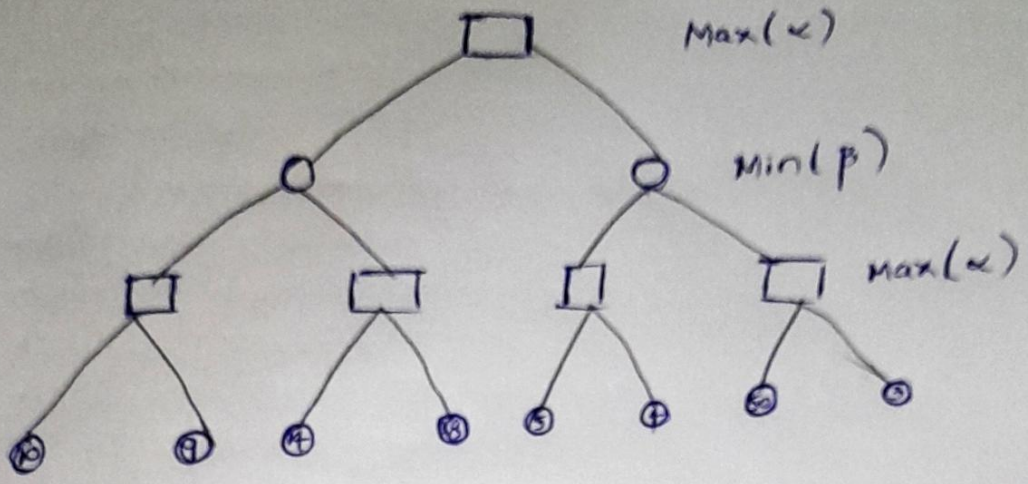
Implement Alpha-Beta Pruning.

Algorithm:

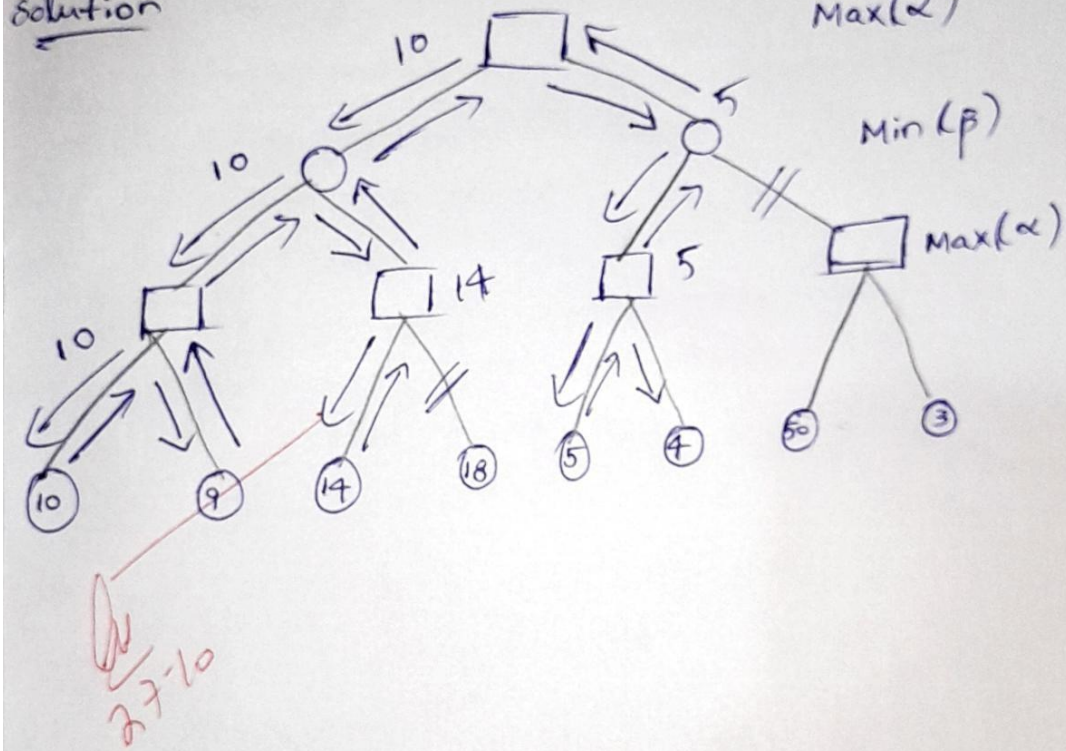


4) if $\alpha \geq \beta$, then break -> (prune remaining branches)

. Return value



solution



Code:

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.value = None
        self.pruned = False

def alpha_beta(node, depth, maximizing, values, alpha, beta, index):
    # Terminal node
    if depth == 3:
        node.value = values[index[0]]
        index[0] += 1
        return node.value

    if maximizing:
        best = float('-inf')
        for i in range(2): # 2 children
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, False, values, alpha, beta, index)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best
    else:
        best = float('inf')
```

```
        for i in range(2):
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, True, values, alpha, beta, index)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best

def print_tree(node, indent=0):
    prune_mark = " [PRUNED]" if node.pruned else ""
    val = f" = {node.value}" if node.value is not None else ""
    print(" " * indent + f"{node.name}{val}{prune_mark}")
    for child in node.children:
        print_tree(child, indent + 4)

# --- main ---
print("=== Alpha-Beta Pruning with Tree ===")
values = list(map(int, input("Enter 8 leaf node values separated by spaces: ").split()))

root = Node("R")
alpha_beta(root, 0, True, values, float('-inf'), float('inf'), [0])

print("\n--- Game Tree ---")
print_tree(root)

print("\nOptimal Value at Root:", root.value)
```

OUTPUT:

```
=== Alpha-Beta Pruning with Tree ===
Enter 8 leaf node values separated by spaces: 3 5 6 9 1 2 0 7

--- Game Tree ---
R = 5
    R0 = 5
        R00 = 5
            R000 = 3
            R001 = 5
        R01 = 6 [PRUNED]
            R010 = 6
    R1 = 2 [PRUNED]
        R10 = 9
            R100 = 9
            R101 = 1
        R11 = 2
            R110 = 2
            R111 = 0

Optimal Value at Root: 5
```

INDEX

Name Thrisha.D Std V Sec F

Roll No. Subject AI Lab School/College

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

Sl. No.	Date	Title	Page No.	Teacher Sign / Remarks
1.	18/08/25	Implement Tic Tac Toe	10	
2.	25/08/25	Vaccum cleaner	8	
3.	1/09/25	using BFS and DFS solve 8 puzzle without heuristic approach	10	
4.	1/09/25	Iterative Deepining search	10	
5.	8/09/25	A* Algorithm - Manhattan A* Algorithm - Missplaced Tiles	10	
6.	15/09/25	Hill climbing Simulated Annealing	10	
7.	22/09/25	Propositional Logic	10	
8.	13/10/25	Unification (FOL)	10	
9.	13/10/25	First order Logic		
10.	27/10/25	FOL (Resolution)	10	
11.	27/10/25	Adversarial search (Alpha-Beta Pruning)		