VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Spandana M R (1BM23CS337)

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 **Spandana M R (1BM23CS337),** 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

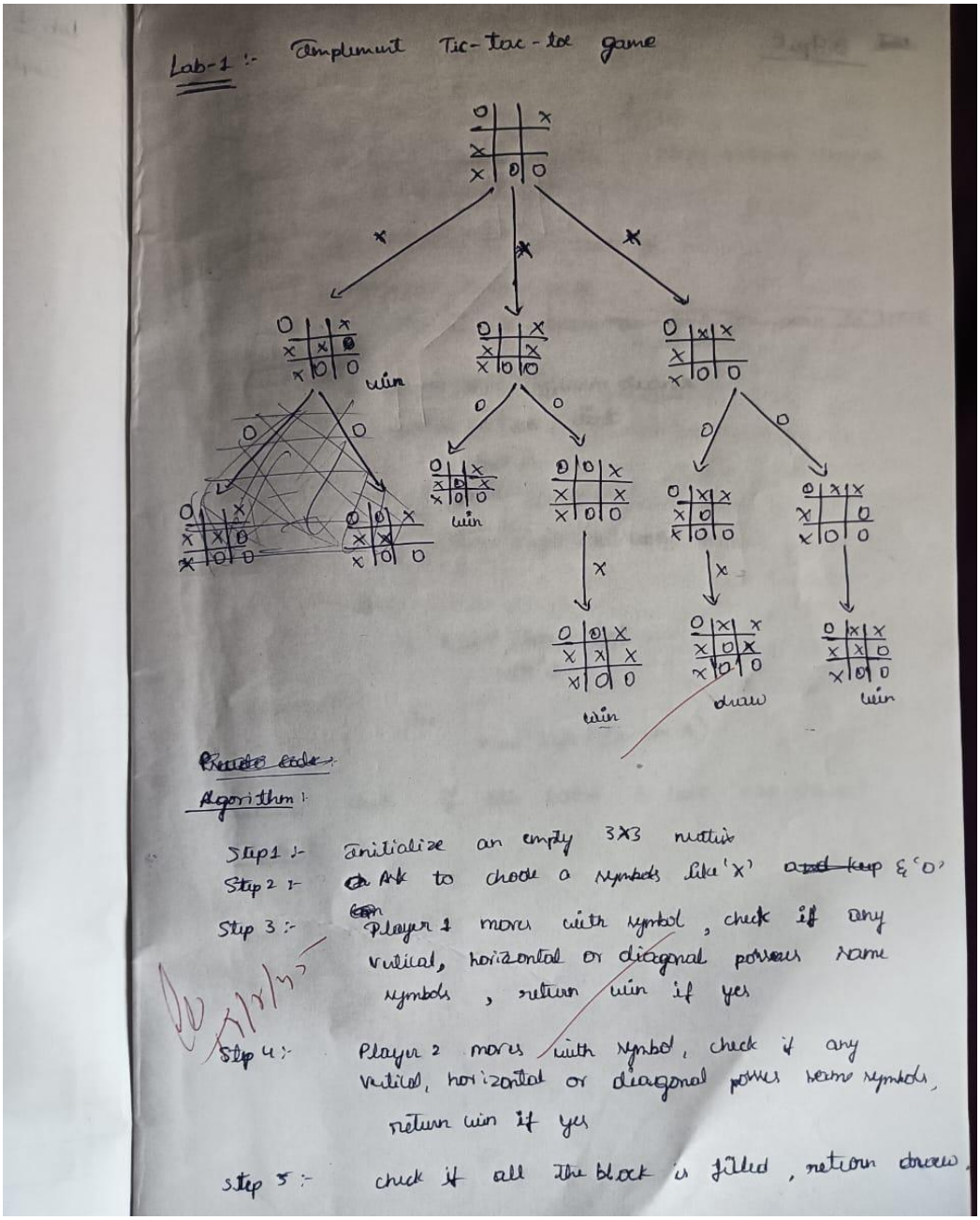
Github Link:
https://github.com/Spandana-mr/AI

Program 1

Implement Tic –Tac –Toe Game I

mplement vacuum cleaner agent

Tic-Tac-Toe:



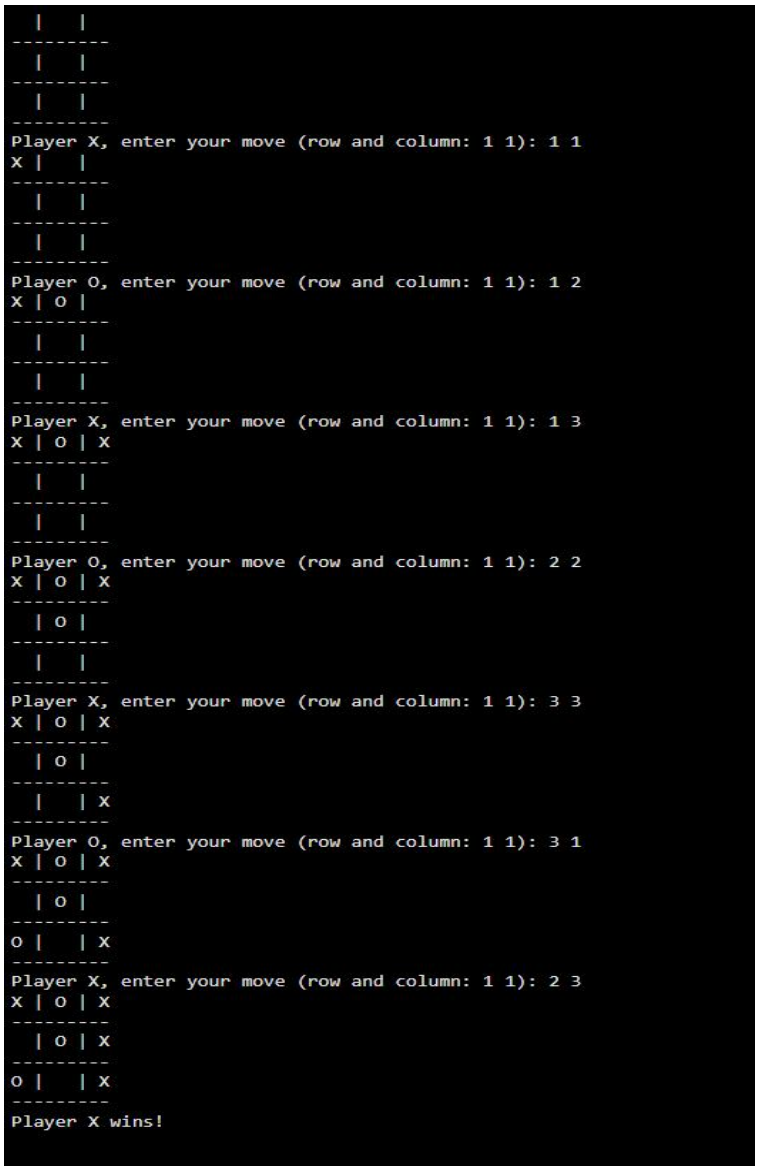
Code:

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

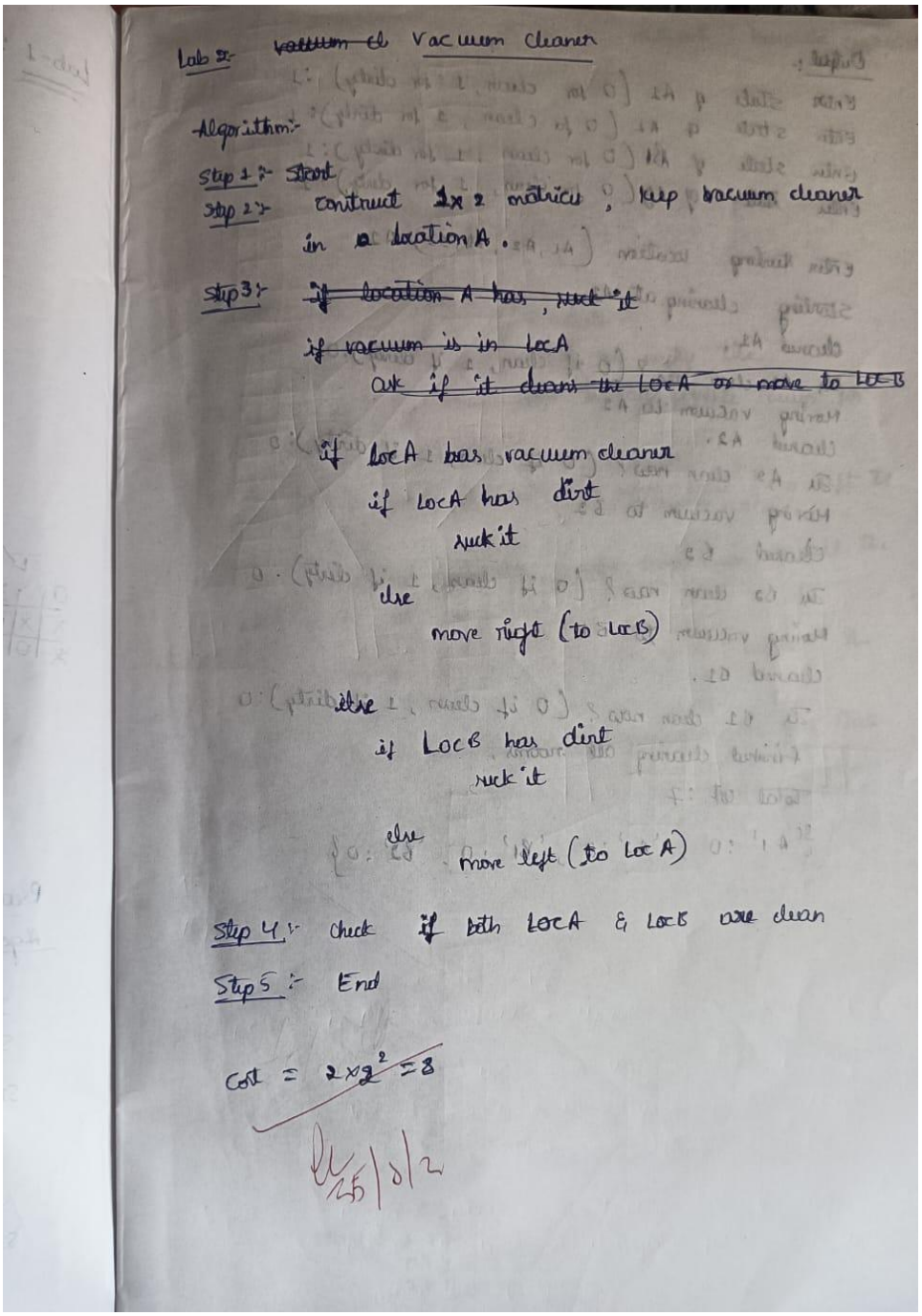
```
def check_winner(board, player):
    # Check rows, columns and diagonals
    for i in range(3):
        if all([cell == player for cell in board[i]]) or \
           all([board[j][i] == player for j in range(3)]):
            return True

    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:



Vacuum Cleaner:



Lab 2:- ~~vacuum cl~~ Vacuum Cleaner

Algorithm:-

Step 1:- Start

Step 2:- construct 1x2 matrix & keep vacuum cleaner in a location A.

Step 3:- ~~if location A has, suck it~~

~~if vacuum is in LocA~~

~~or if it clean the LocA or move to LocB~~

if LocA has vacuum cleaner

if LocA has dirt

suck it

else

move right (to LocB)

else if

if LocB has dirt

suck it

else

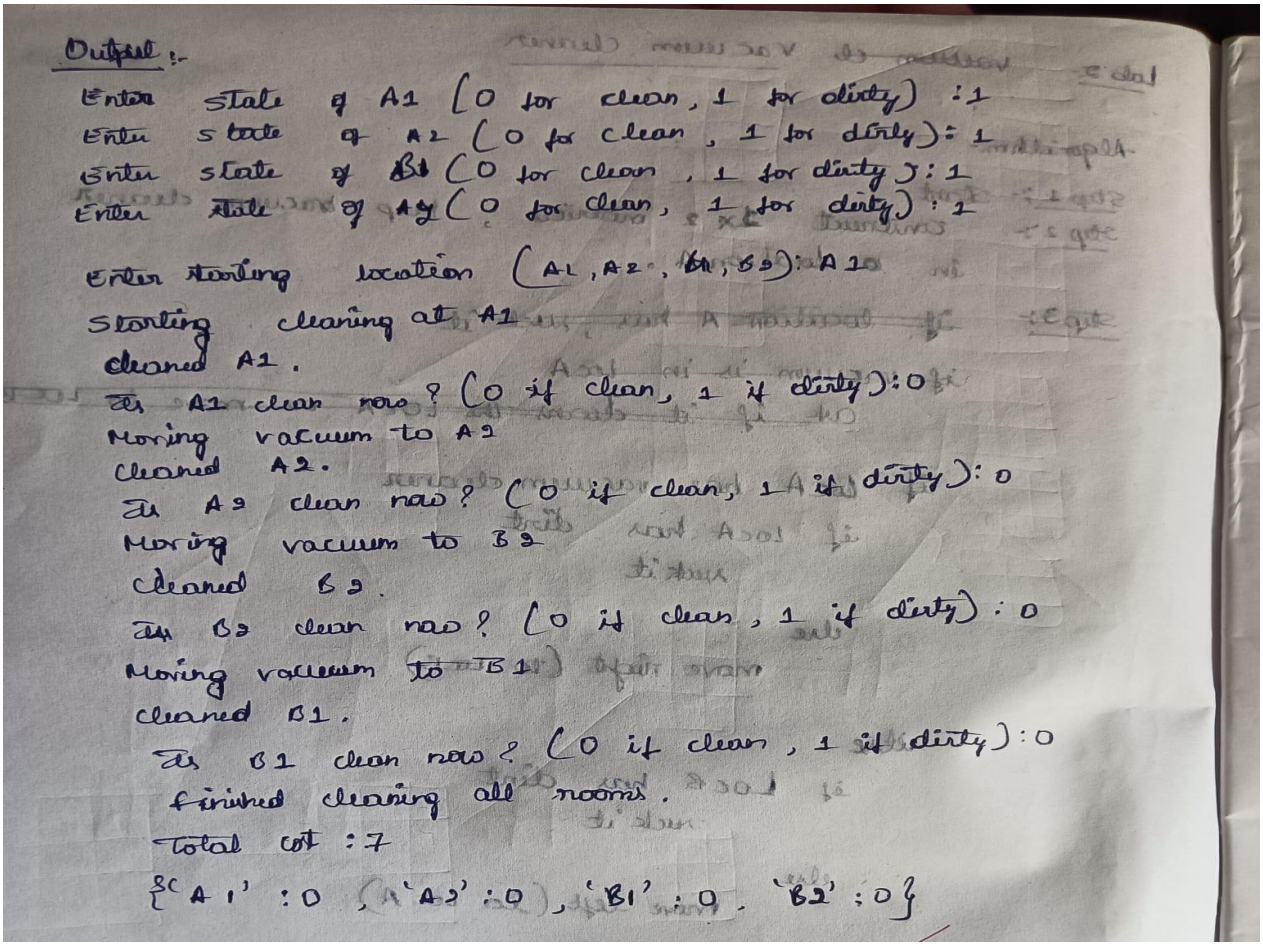
move left (to LocA)

Step 4:- Check if both LocA & LocB are clean

Step 5:- End

Cost = $2 \times 2^2 = 8$

25/8/2



Code:

```
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()
```

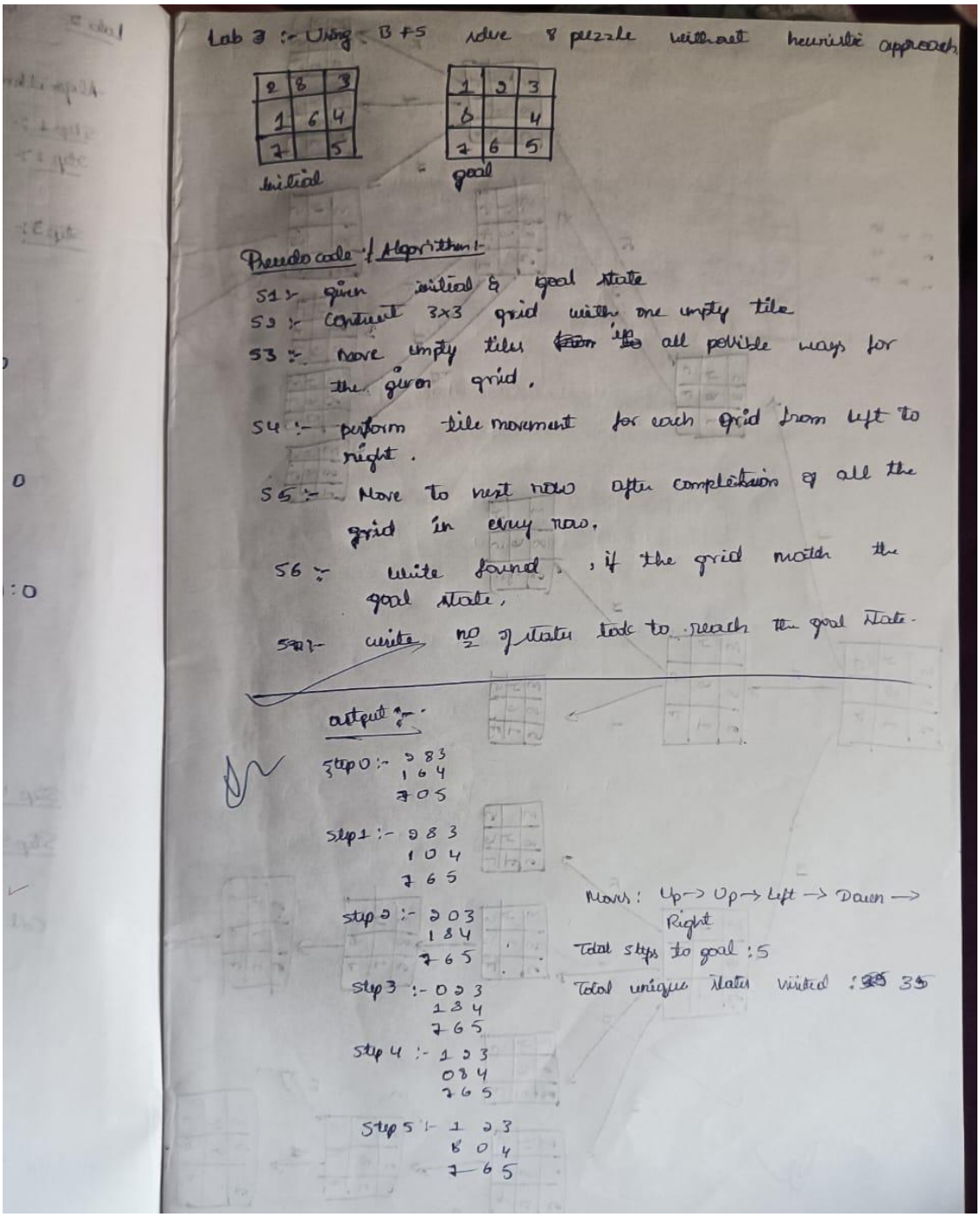
OUTPUT:

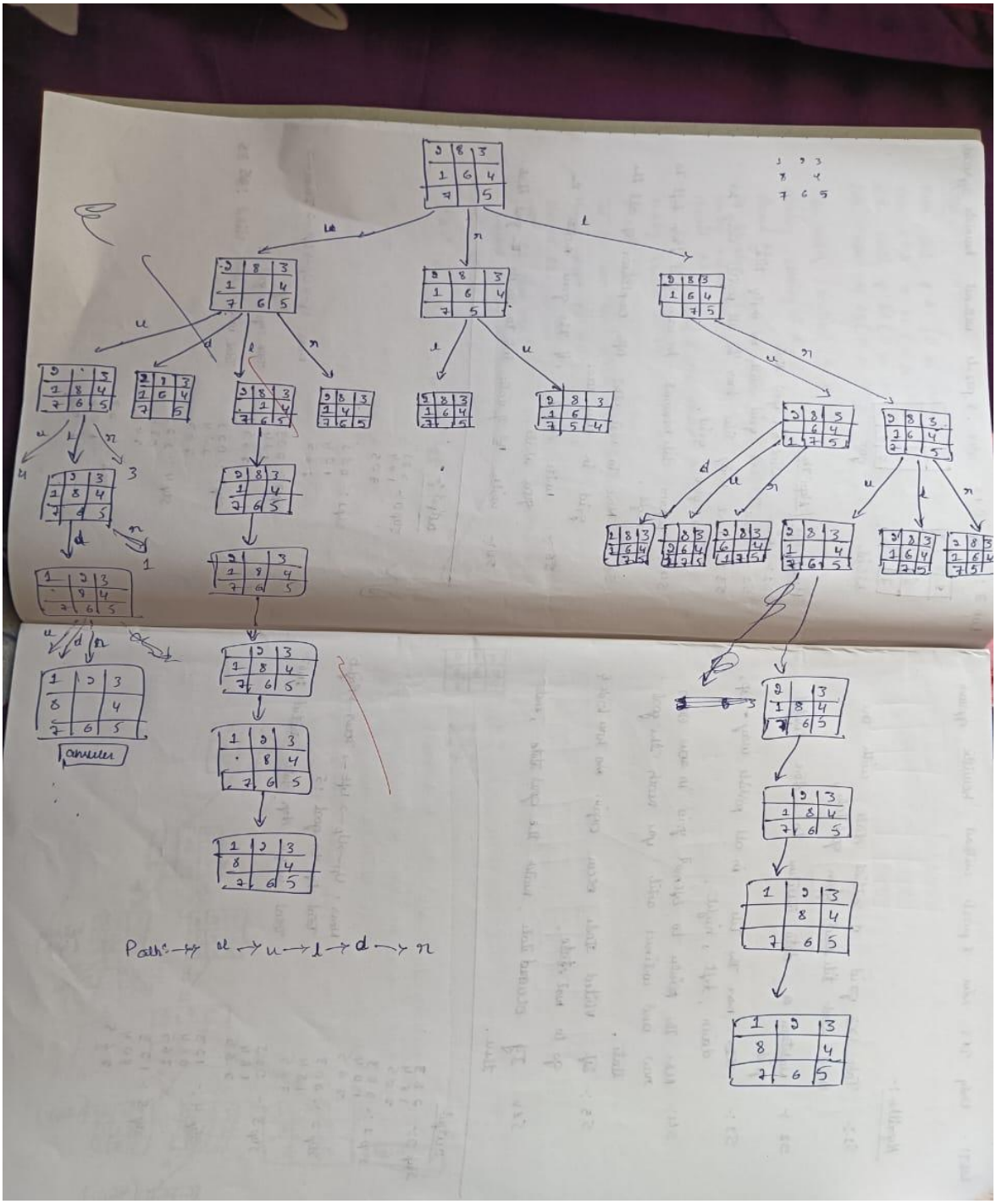
```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaning A.
Turning vacuum off
Cost: 1
{'A': 0, 'B': 0}
```

Program 2

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

8 puzzle problems using Depth First Search (DFS)




Path → u → u → l → d → r

Lab3:- Using DFS solve 8 puzzle without heuristic approach.

Algorithm :-

S1:- Take 3X3 grid of initial state with one empty tile and given goal state,

S2 :- Maintain a data structure to store visited grid.

S3 :- Move the tile in all possible way - up, down, left, right.

S4:- Move the pointer to leftmost grid in each row and continue until you reach the goal state,

S5 :- If visited state occur again then backtrack & go to next state.

S6:- If occured state, match the goal state, end there.

Output

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

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

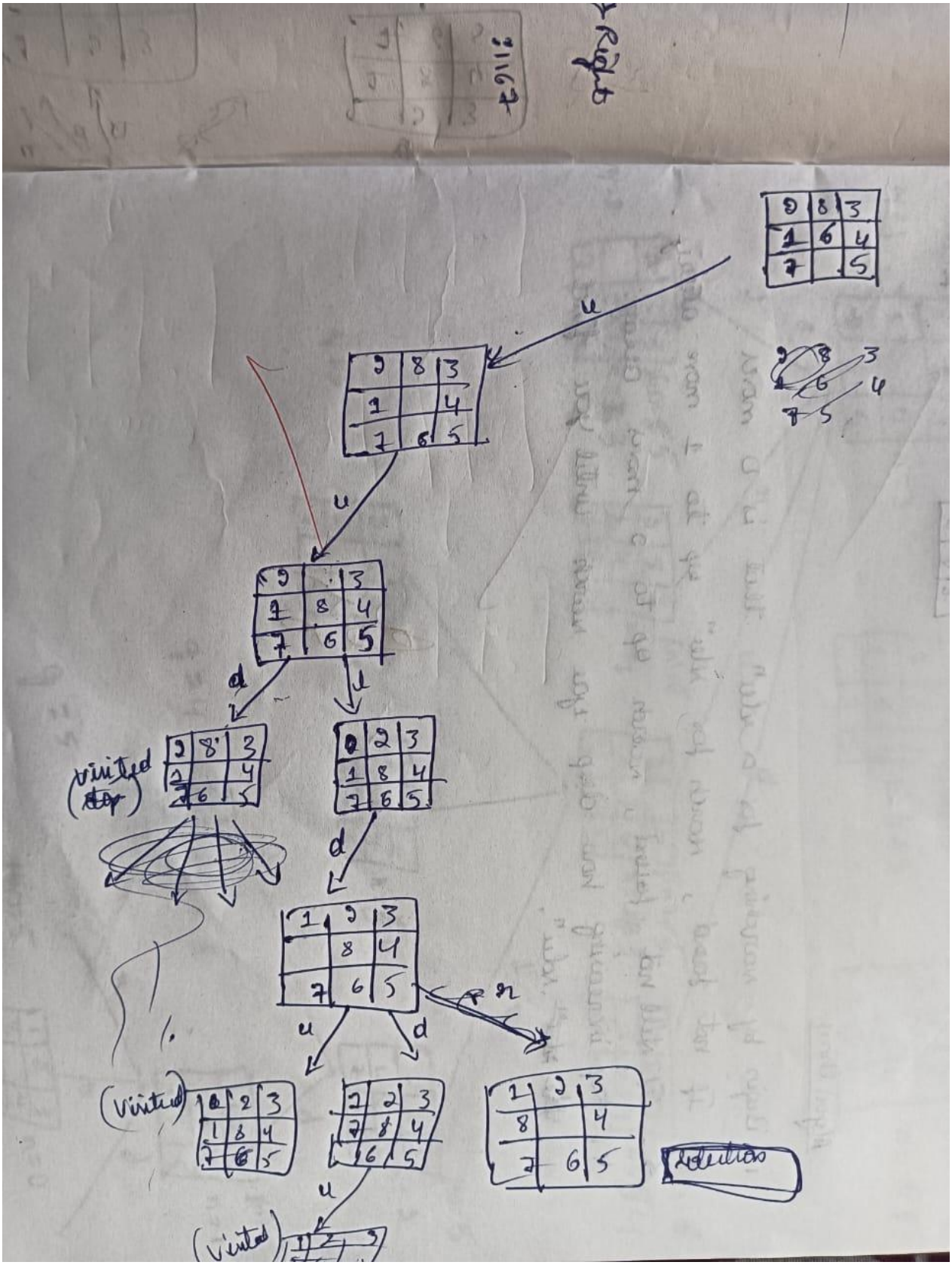
Step 2 :- 2 0 3
1 8 4
7 6 5

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

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

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

Moves : Up → Up → Left → Down → Right
Total steps to goal : 5
Total unique state visited : 1167


u
u
d
u
d
d
u
d
u
visited
visited
visited
Solution

Code:

Usig DFS 8 puzzel without heuristic

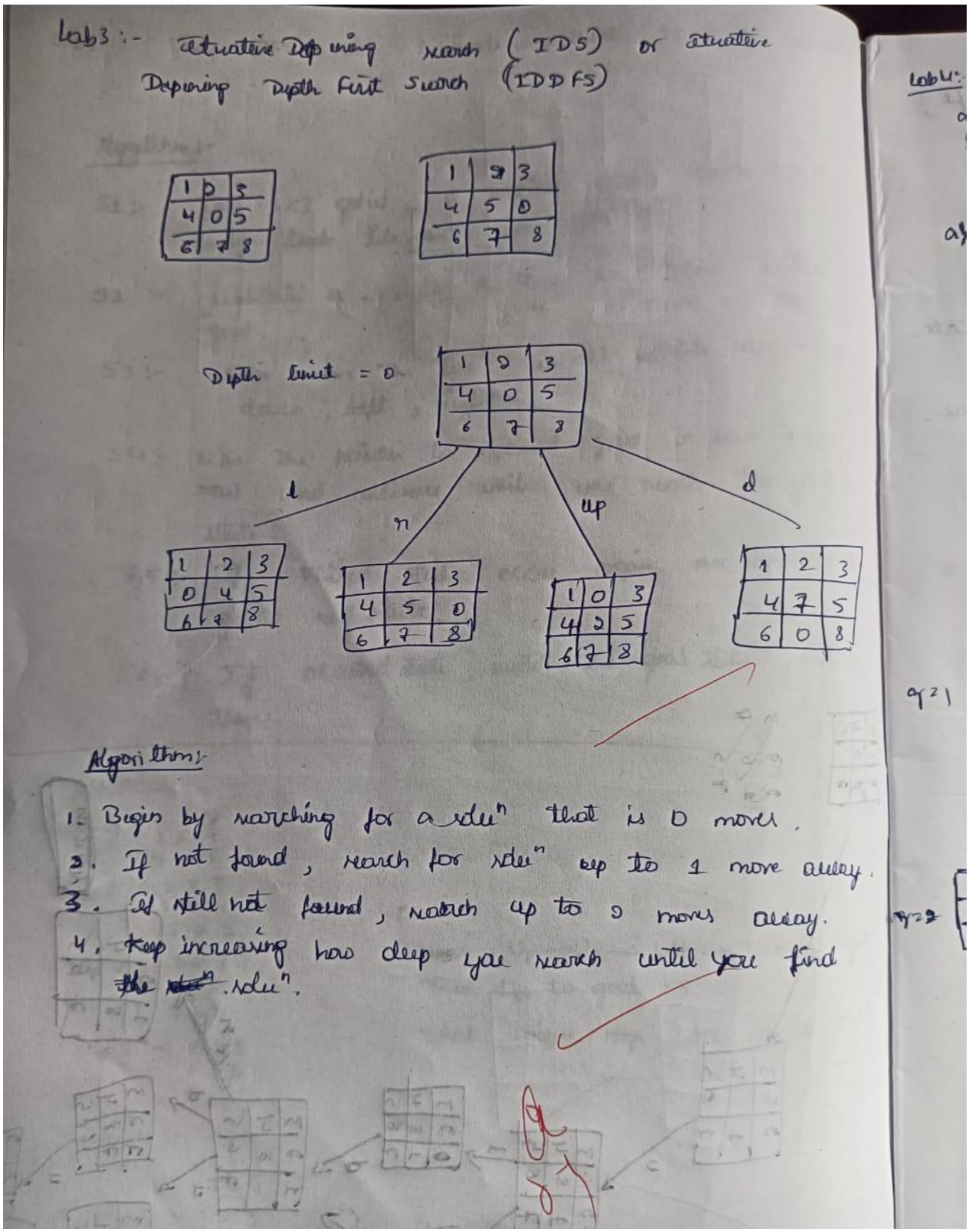
```
# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))

# 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)



Code:

```
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.")
```

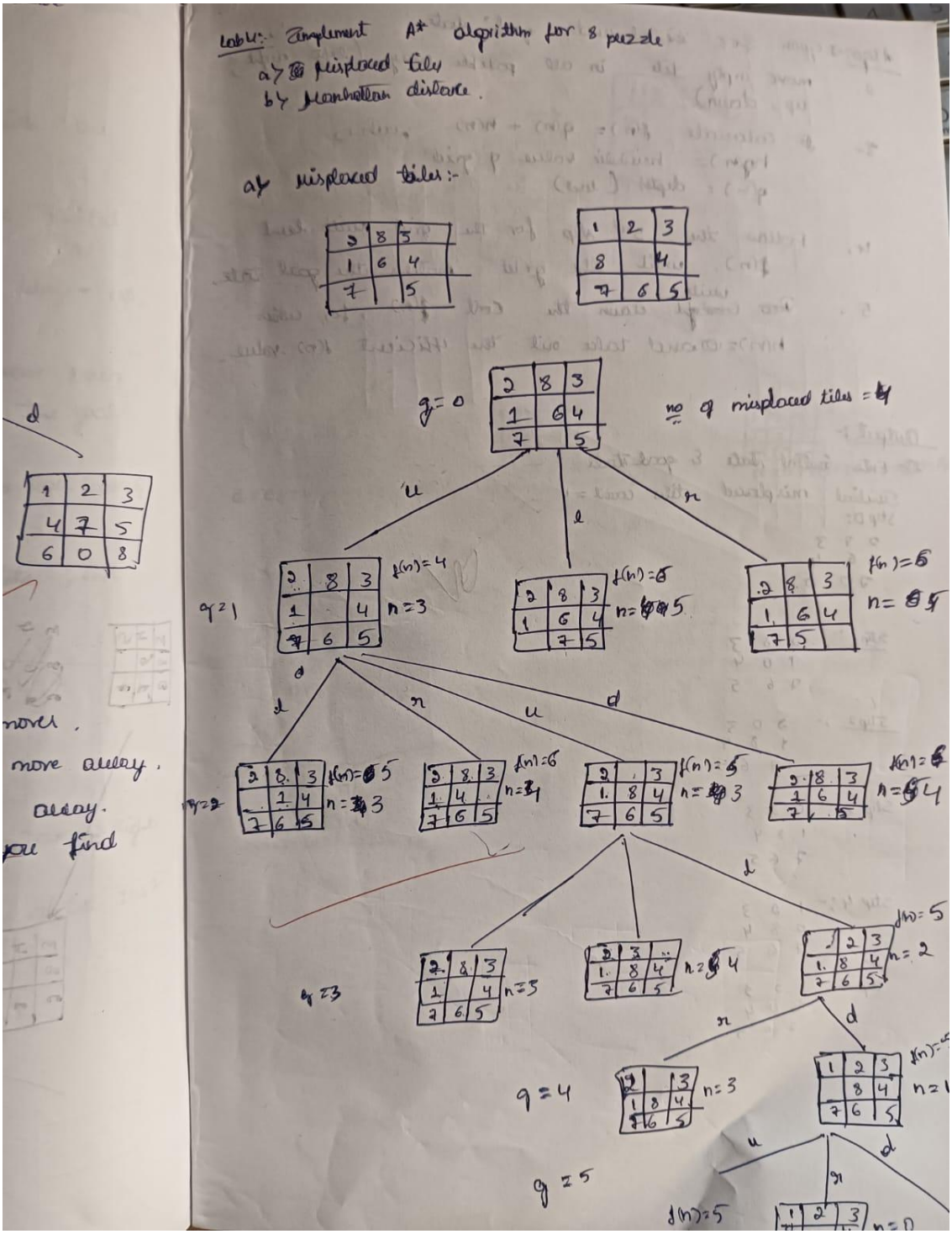
OUTPUT

```
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

1.Misplace Tiles



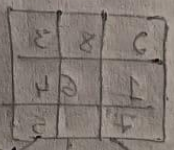
Algo:- 1. Given 3x3 initial and goal state.

2. move empty tile in all possible ways (left, right, up, down)

3. Calculate $f(n) = g(n) + h(n)$, where,
   $h(n)$ = heuristic value of grid
   $g(n)$ = depth (level)

4. Follow the grid step for the grid with least $f(n)$. until the grid matches the goal state.

5. write down the cost $f(n)$, for which $h(n) = 0$ and take out the efficient $f(n)$ value.



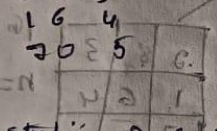
Output:-

Enter initial state & goal state :- ----

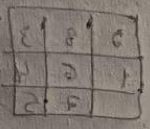
Initial misplaced tiles count = 4

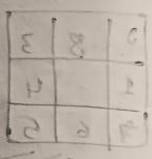
Step 0:

2 8 3
1 6 4
7 0 5

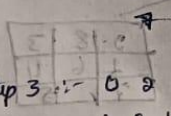


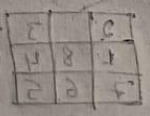
Step 1 :- 2 8 3
1 0 4
7 6 5

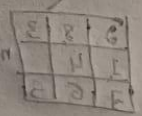


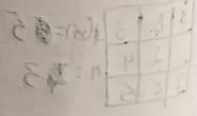


Step 2 :- 2 0 3
1 8 4
7 6 5



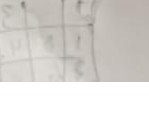






Step 3 :- 0 2 3
1 8 4
7 6 5

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



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

Code:

```
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
```

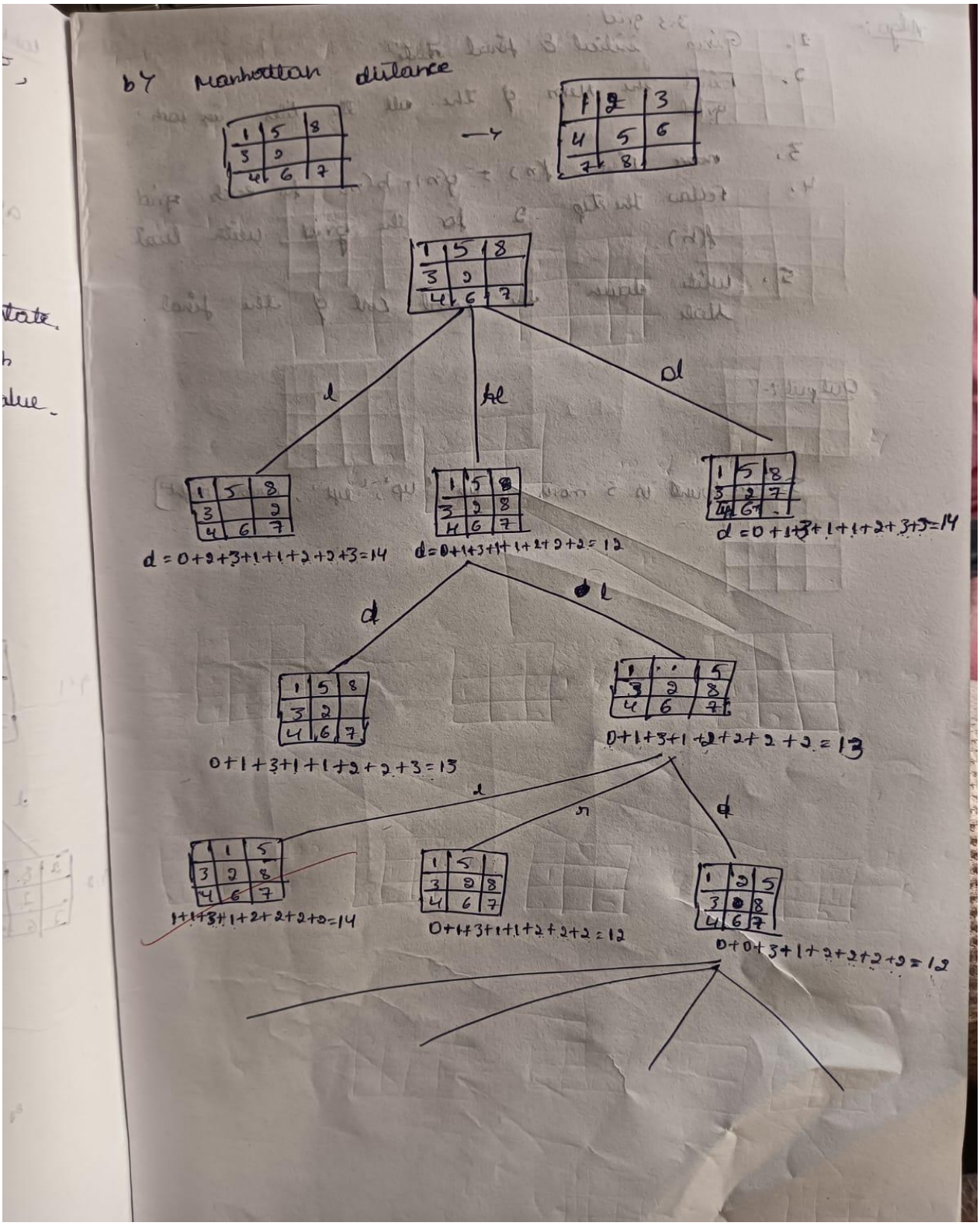
```
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state)
    return neighbors
# A* Search
def astar(start):
    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))
```

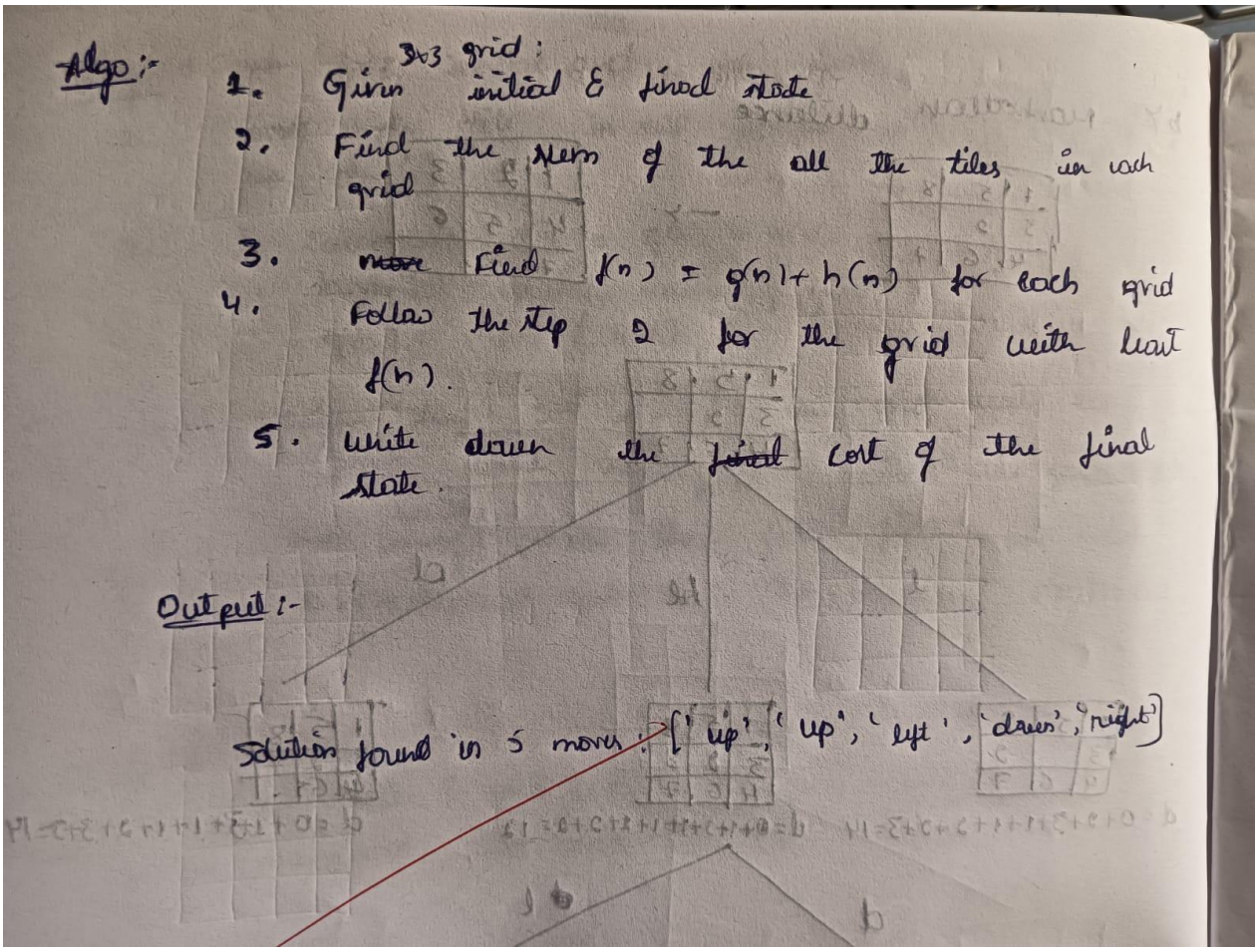
```
solution = astar(start_state)

# 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:


by Manhattan distance
d = 0+2+3+1+1+2+2+3=14
d = 0+1+3+1+1+2+2+2 = 12
d = 0+1+3+1+1+2+2+3=13
0+1+3+1+1+2+2+2+2 = 13
1+1+3+1+2+2+2+2=14
0+1+3+1+1+2+2+2 = 12
0+0+3+1+2+2+2+2 = 12



Code:

```
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:
```

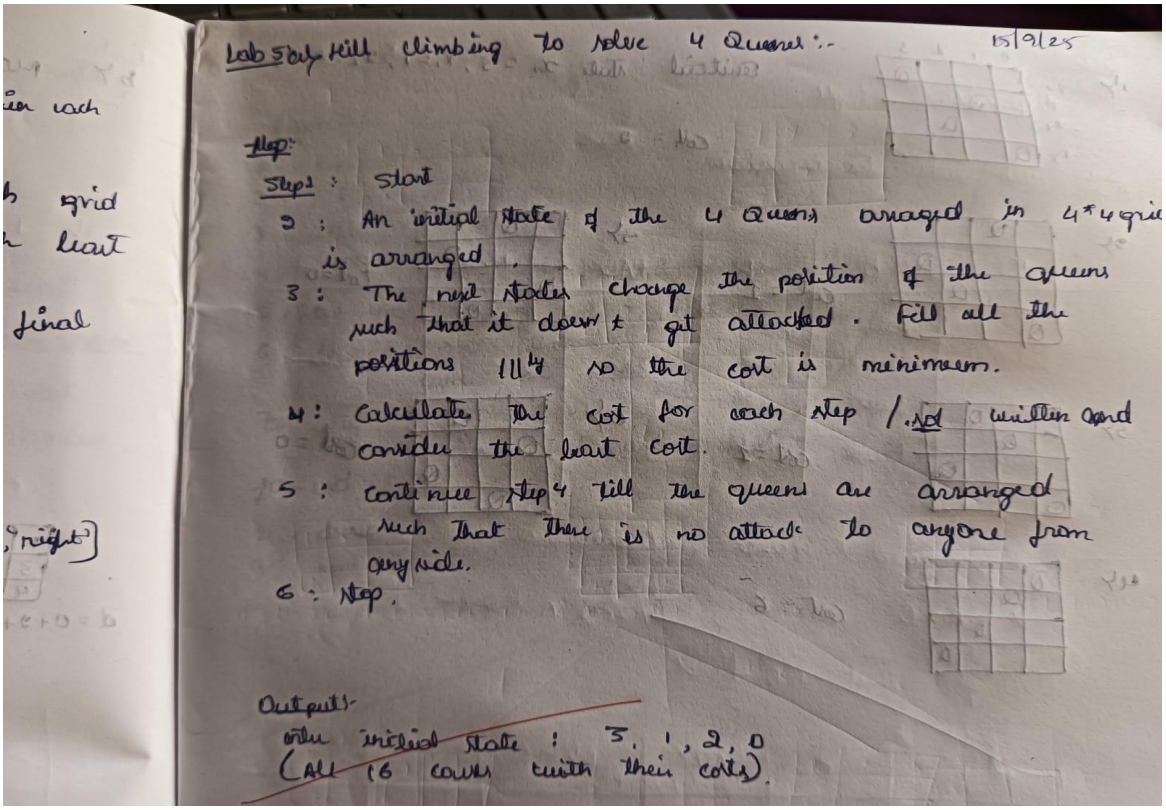
```
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

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

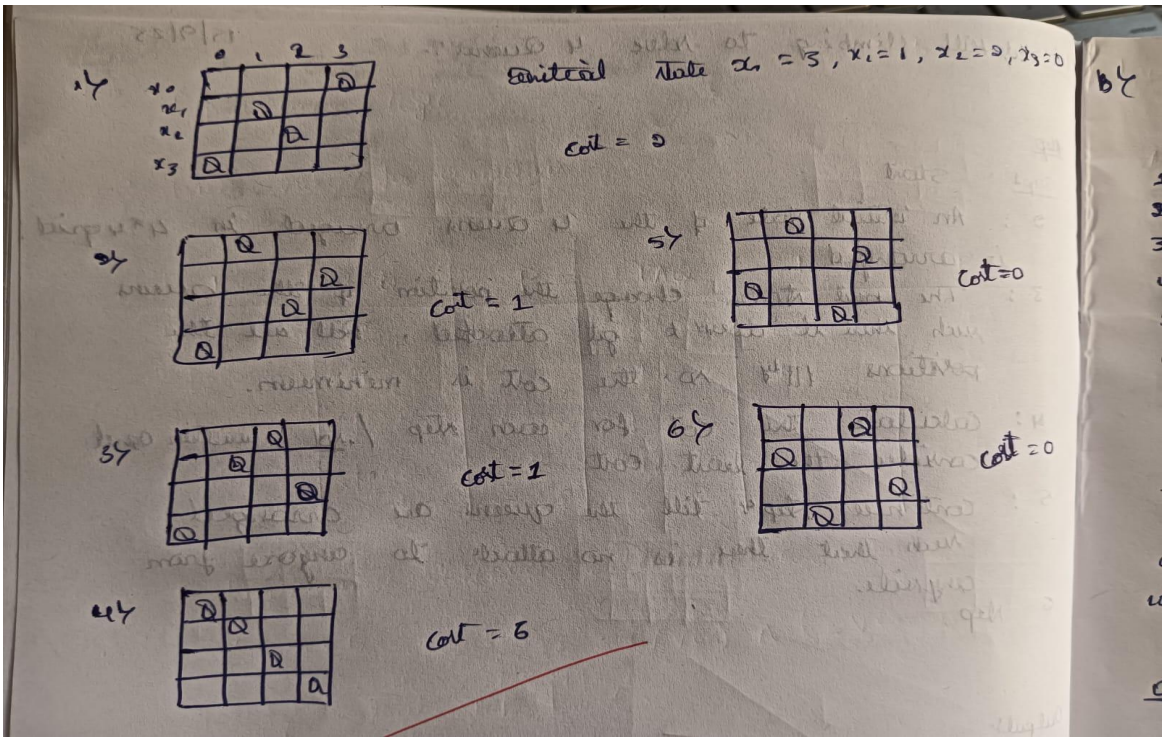
OUTPUT:

```
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





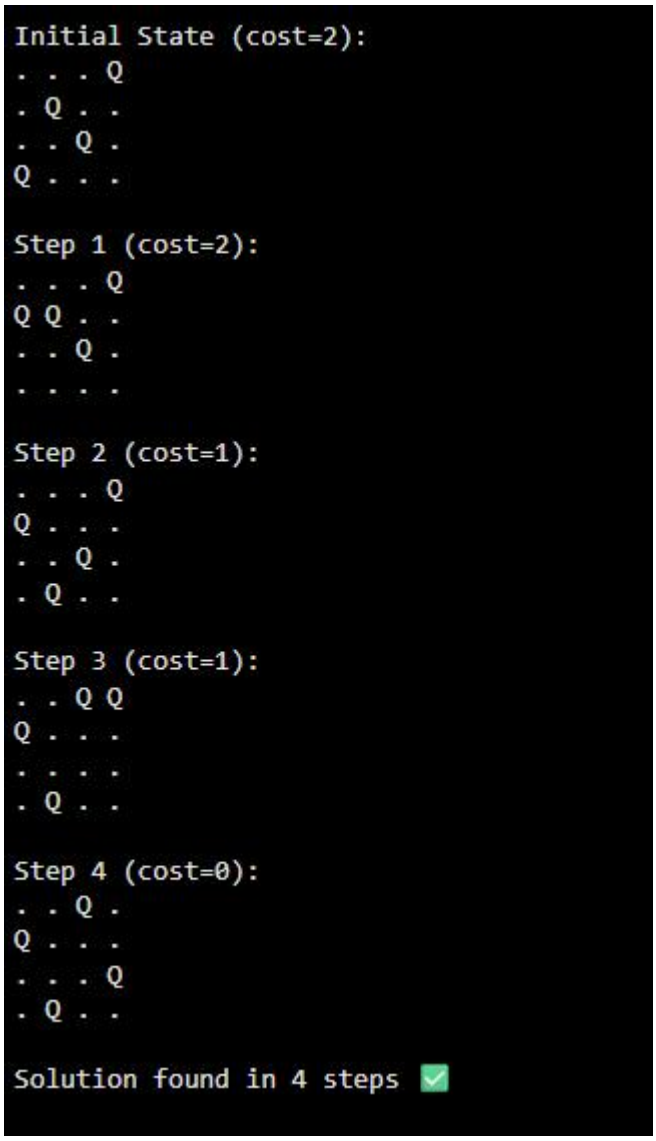
Code:

```
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)
```

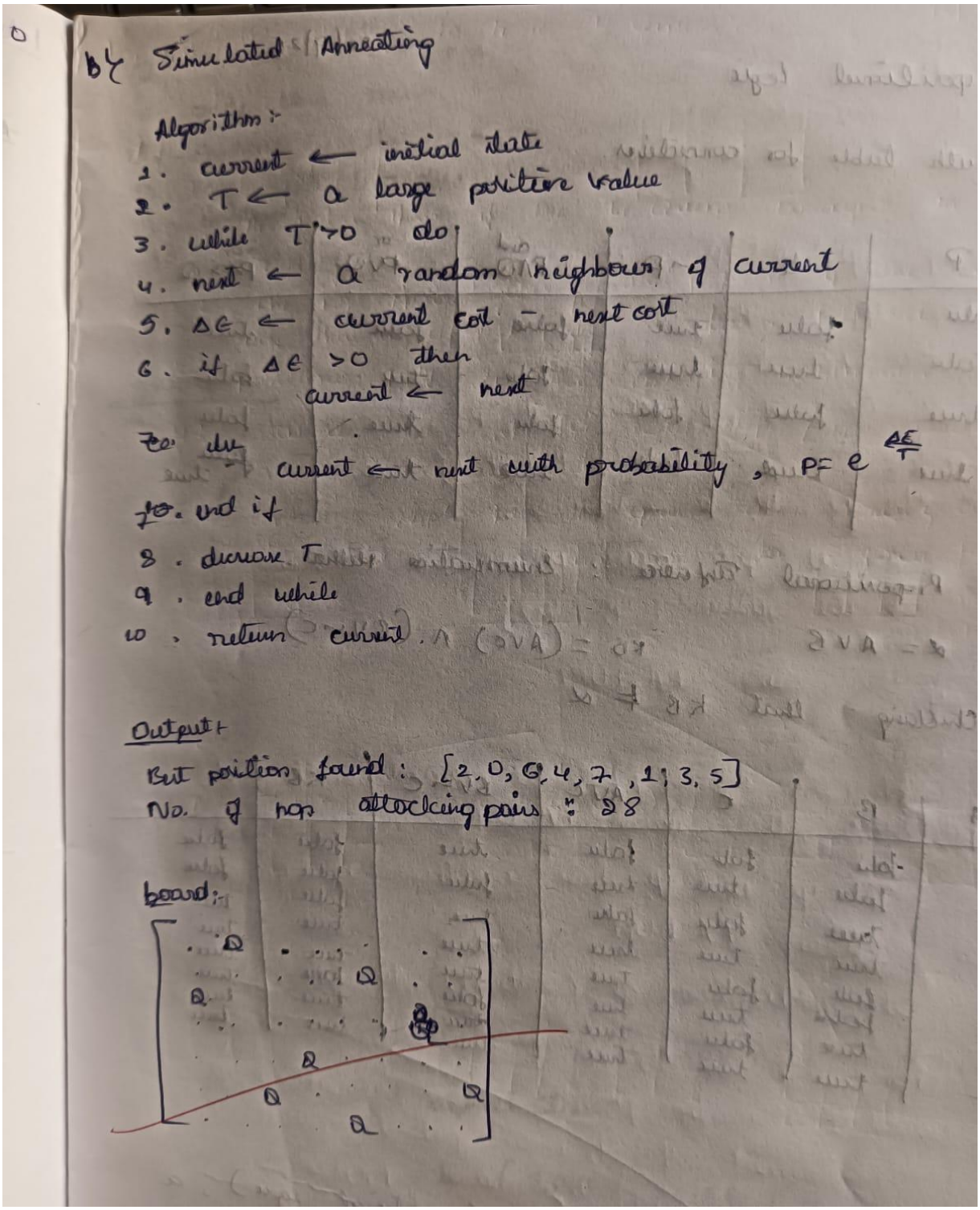
```
        # handle sideways moves
        if next_cost == current_cost:
            if sideways_moves >= max_sideways:
                break
            else:
                sideways_moves += 1
        else:
            sideways_moves = 0

        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



By Simulated Annealing

Algorithm:

1. current ← initial state
2. T ← a large positive value
3. while T > 0 do:
4. next ← a random neighbour of current
5. ΔE ← current cost − next cost
6. if ΔE > 0 then
   current ← next
7. else
   current ← next with probability, $P = e^{\frac{\Delta E}{T}}$
8. end if
9. decrease T
10. end while
11. return current

Output:

Best position found: [2, 0, 6, 4, 7, 1, 3, 5]

No. of non attacking pairs: 28

board:

Code:

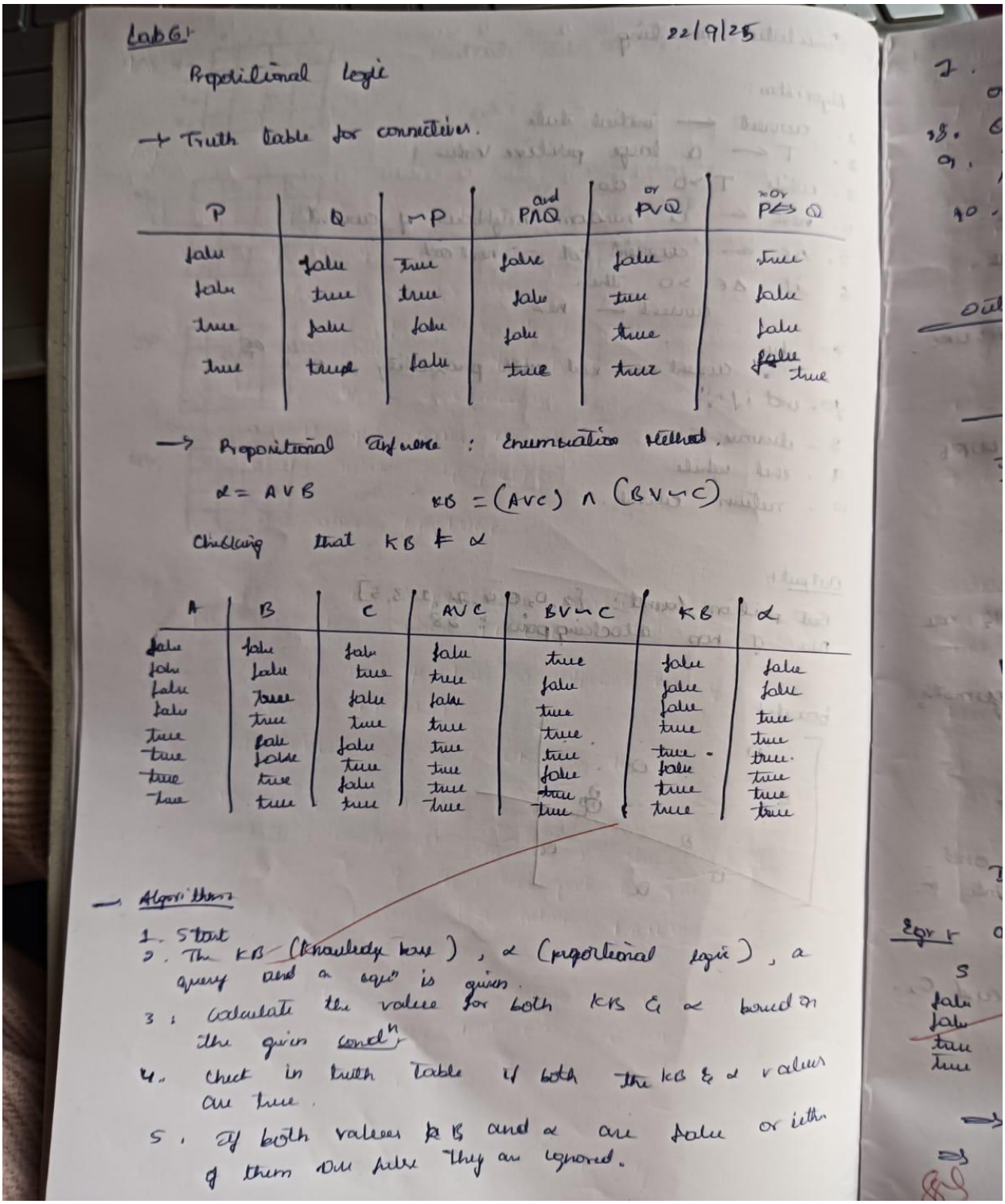
```
from scipy.optimize import dual_annealing
import numpy as np

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.



Lab 6:

22/9/25

Propositional logic

→ Truth table for connectives.

P	Q	¬P	P ∧ Q (and)	P ∨ Q (or)	P ⊻ Q (xor)
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	false

→ Propositional Inference : Enumeration Method.

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

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. Start
2. The KB (knowledge base), α (propositional logic), a query and a sqe" is given.
3. Calculate the values for both KB & α based on the given condⁿ
4. Check in truth table if both the KB & α values are true.
5. If both values of KB and α are false or either of them are false they are ignored.

7. Assume a P value where is 'true', once then recur with the rest. once & 'false'
8. Combine the results with and operation.
9. If both branches are false then the whole check succeed.
10. Stop.

Output

A	B	C	KB	alpha	KB ∧ α
true	true	true	true	true	true
true	true	false	false	true	false
true	false	true	true	true	true
true	false	false	true	true	true
false	true	true	true	true	true
false	true	false	false	false	false
false	false	true	false	true	false
false	false	false	false	false	false

Models when KB & alpha are true:

{'A': true, 'B': true, 'C': true}

{'A': true, 'B': false, 'C': true}

{'A': true, 'B': false, 'C': false}

{'A': false, 'B': true, 'C': true}

Does KB entail alpha? true

Eg:- a: ¬(S∨T), b: (S∧T), c: (T∨¬T)

S	T	a	b	c
false	false	true	false	true
false	true	false	false	true
true	false	false	false	true
true	true	false	true	true

⇒ a entails b: false

⇒ a entails c: true (coz S: false & T: false)

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}")
```

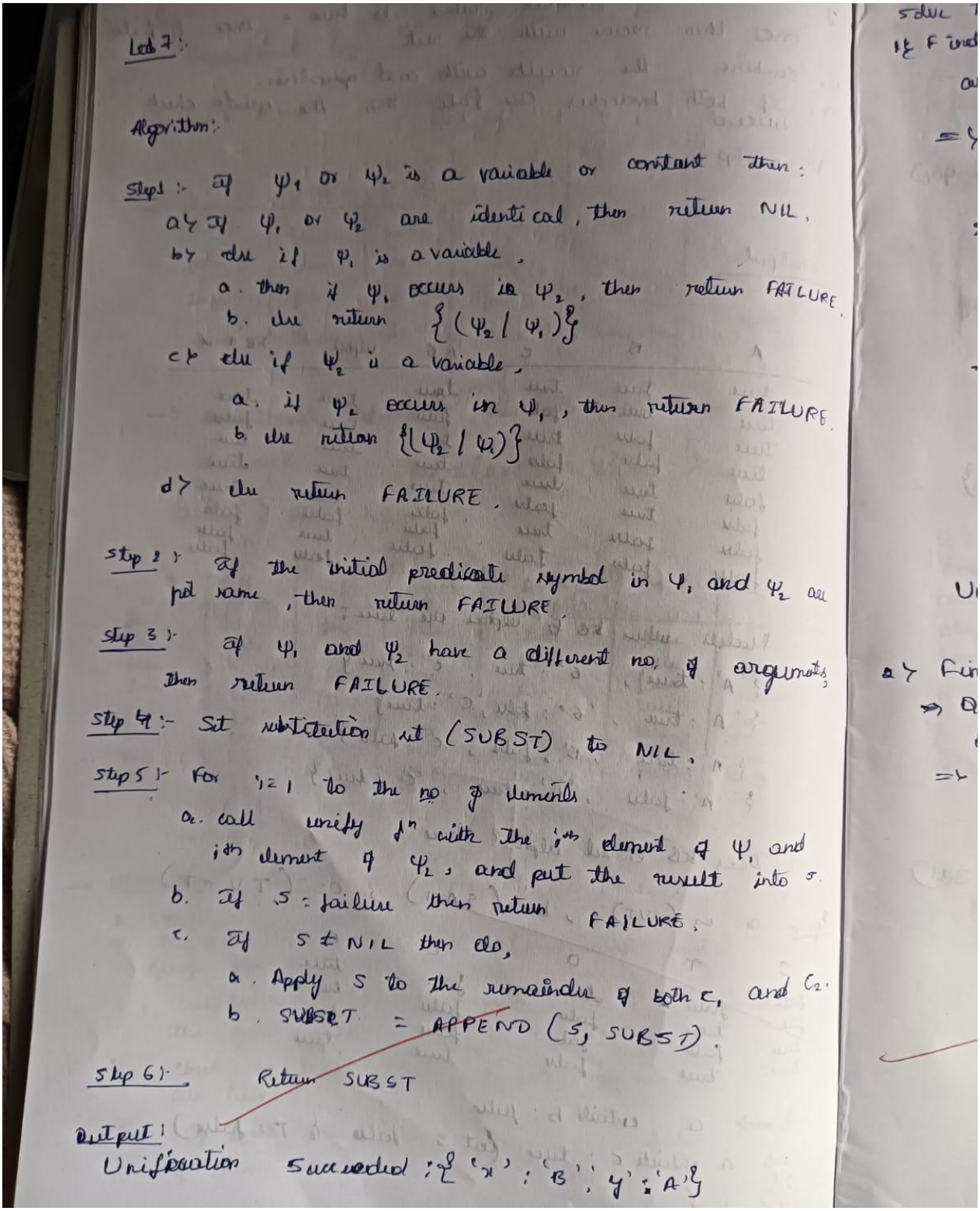
```
    if kb_val and not alpha_val:
        entailed = False

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



Lab 7:

Algorithm:

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

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

b) else if ψ_1 is a variable,

a. then if ψ_1 occurs 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 $\{(\psi_2 / \psi_1)\}$

d) else return FAILURE.

Step 2 :- If the initial predicate symbol in ψ_1 and ψ_2 are not same, then return FAILURE.

Step 3 :- If ψ_1 and ψ_2 have a different no. of arguments, then return FAILURE.

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

Step 5 :- For i = 1 to the no. of elements

a. call unify function with the i^{th} element of ψ_1 and i^{th} element of ψ_2, and put the result into S.

b. If S = failure then return FAILURE.

c. If S ≠ NIL then do,

a. Apply S to the remainder of both L_1 and L_2.

b. SUBST = APPEND (S, SUBST).

Step 6 :- Return SUBST

Output :

Unification Succeeded : { 'x' : 'B', 'y' : 'A' }

Solve the following:

1) Find Most General Unifier (MGU) of $\{p(b, x, f(g(z)))$ and $p(z, f(y), f(y))\}$

$\Rightarrow \{p(b, x, f(g(z)))$
$p(z, f(y), f(y))\}$ both have arguments & predicates our name

So, $b = z$
$x = f(y)$
$f(g(z)) = f(y)$
$\therefore g(z) = y$

$\Rightarrow z = b$
$x = f(y)$
$y = g(z)$
$x = f(y)$ but $y = g(z)$
$\therefore x = f(g(z))$

$p(z, f(y), f(g(z))) \Rightarrow b = z, x = f(y)$
$p(z, f(y), f(g(z))) \Rightarrow f(y) = f(g(y))$
Unifiers $= \{b/z, x/f(y), y/g(z)\}$

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

Q

$\Rightarrow Q(a, g(x,a), f(y))$
$Q(a, g(f(b),a), b)$

$Q(a, g(f(b),a), f(y))$ $\quad x = f(b)$
$Q(a, g(f(b),a), b)$ $\quad f(y) = x$
$f(y) = f(b)$
$\therefore y = b$

$Q(a, g(f(b),a), f(b))$
$Q(a, g(f(b),a), f(b))$
Unifiers : $\{x / f(b)\}$

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

p(f(a), g(y))

p(x, x)

if { x = f(a) and x = g(y) } ∴ x = f(a)

it is not possible g(y) ≠ x ⇒ f(a) ≠ g(y)

∴ No Unifiers

4) Unify { prime(11) and prime(y) }

prime(11) } both have same predicate & same no of
prime(y) } arguments.

∴ y = 11

prime(11) Sub: { y/11 }

prime(11)

5) Unify { knows(John, x), knows(y, mother(y)) }

knows(John, x)

knows(y, mother(y))

Let y = John and x = mother(y)

x = mother(John)

∴ knows(John, mother(John)) =

knows(John, mother(John))

6) Unify { knows(John, x), knows(y, Bill) }

knows(John, x)

knows(y, Bill)

Let y = John

x = Bill

knows(John, Bill) = knows(John, Bill)

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
```

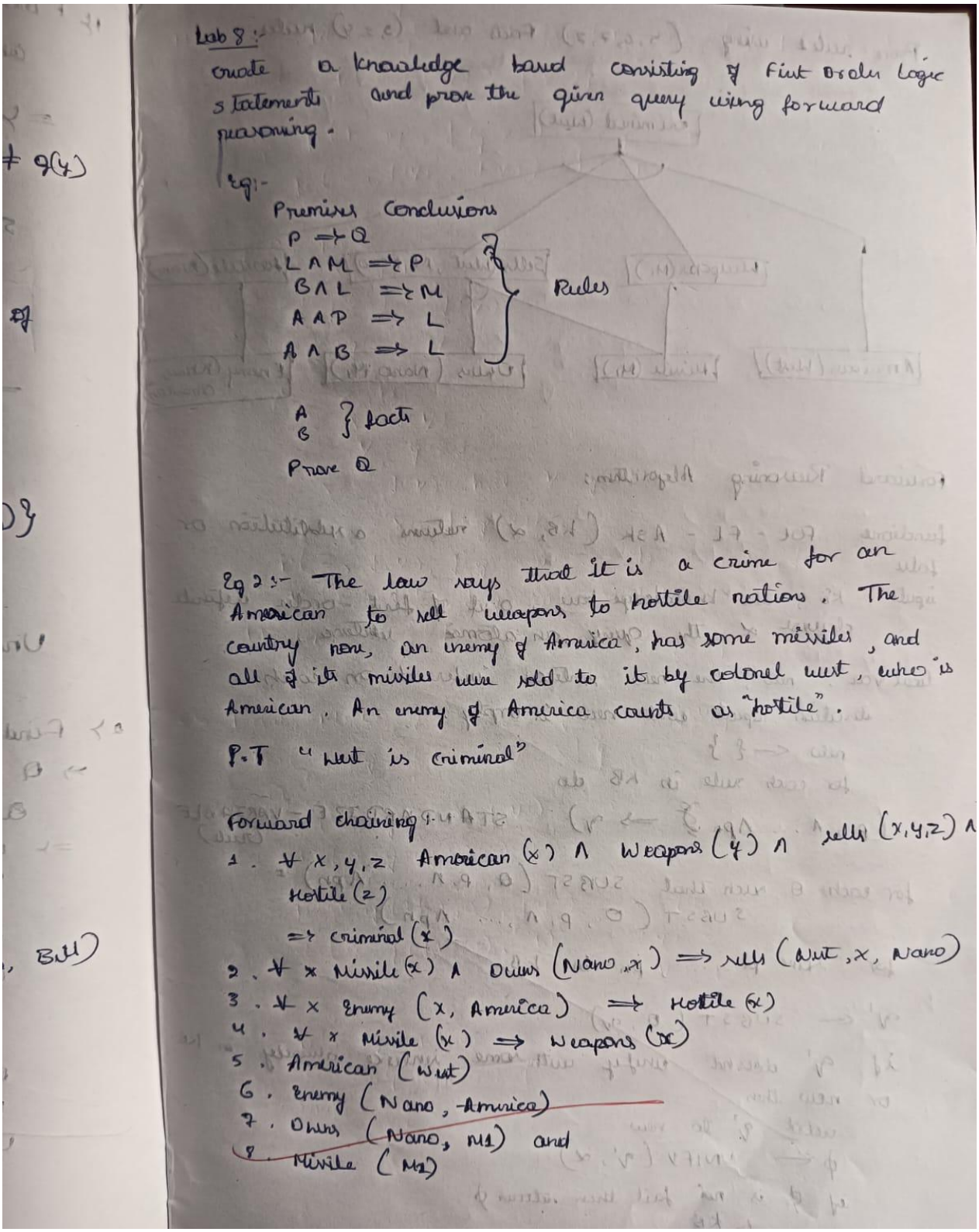
```
        return subst

    # 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.



Lab 8 :-

Create a knowledge based consisting of First Order Logic statements and prove the given query using forward reasoning.

Eg:-

Premises Conclusions

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$

$B \wedge L \Rightarrow M$

$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$ } Rules

A

B } facts

Prove Q

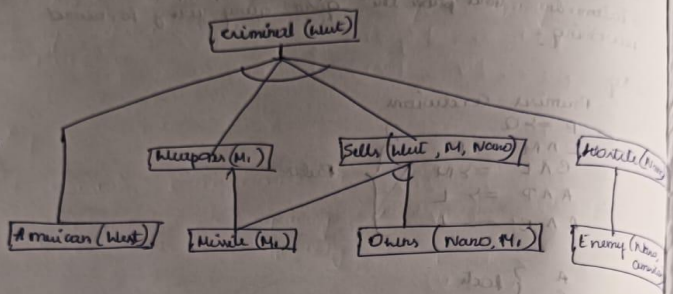
Eg 2:- The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by colonel West, who is American. An enemy of America counts as "hostile".

P.T "West is criminal"

Forward chaining :-

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

Prove rules using (5, 6, 7, 8) Facts and (3, 3, 4) rules.



Forward Reasoning Algorithm:

function FOL - FL - 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 var: new, the 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 (θ, p1 ∧ ... ∧ pn) = SUBST (θ, p1' ∧ ... ∧ pn')

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

q' ← SUBST (θ, q)

if q' doesn't 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:

New fact inferred : Criminal (West)

New fact inferred : SoldWeapons (West, Nono)

Final facts:

American (West)

Hostile (Nono)

Missile (Nono)

Criminal (West)

Sold Weapons (West, Nono)

SRD 27/10/25

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:
```

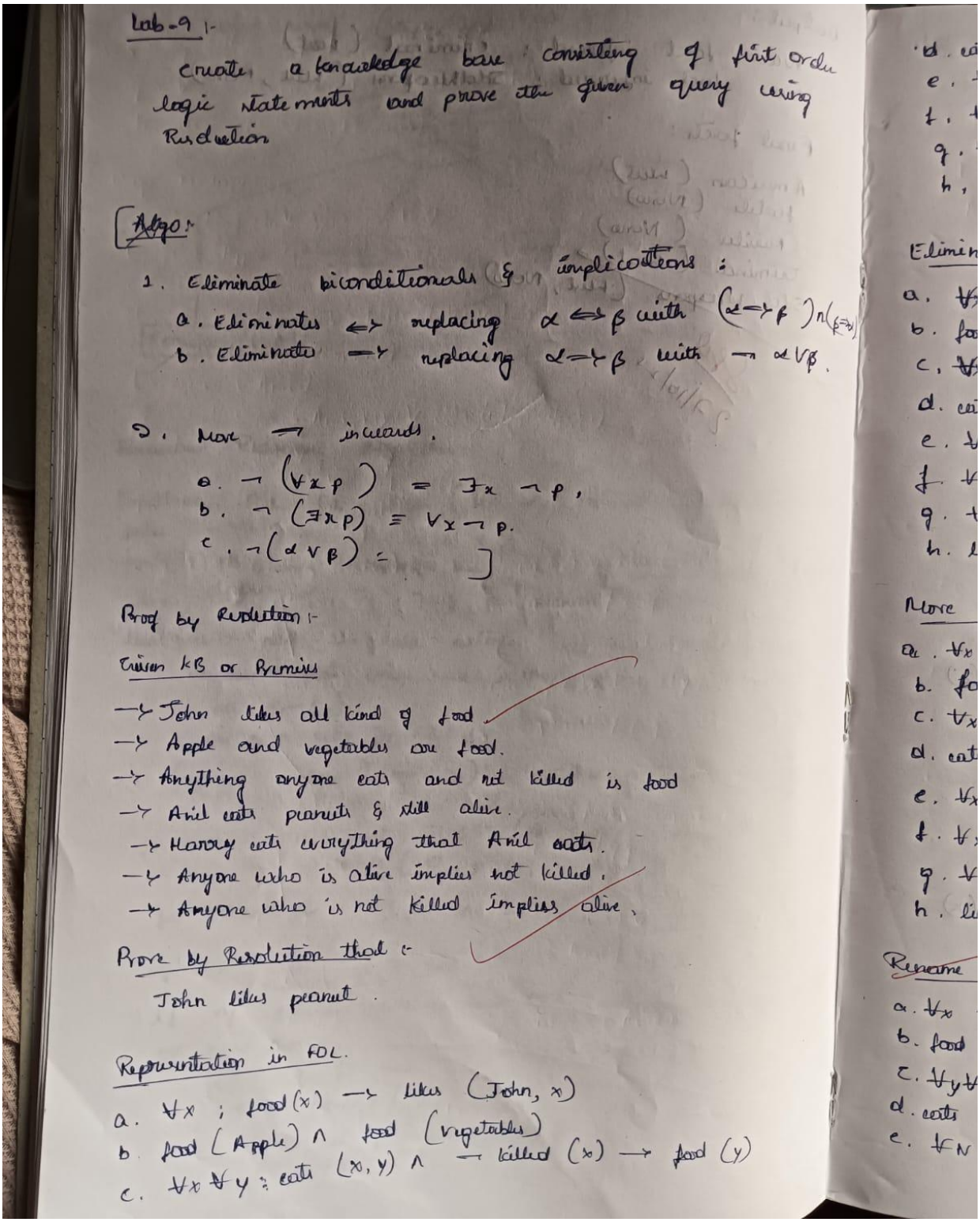
```
                print(f"Inferred new fact: {conclusion} from {premises} => {conclusion}")
                self.facts.add(conclusion)
                agenda.append(conclusion)

                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



Lab-9 :-

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

Algo :-

1. Eliminate biconditionals & implications :
   a. Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$
   b. Eliminate $\rightarrow$ replacing $\alpha \rightarrow \beta$ with $\neg \alpha \vee \beta$.

2. Move $\neg$ inwards.
   a. $\neg (\forall x\, p) \equiv \exists x \neg p$,
   b. $\neg (\exists x\, p) \equiv \forall x \neg p$.
   c. $\neg (\alpha \vee \beta) \equiv$]

Proof by Resolution :-

Given KB or Premise

→ John likes all kind of food.
→ Apple and vegetables are food.
→ Anything anyone eats and not killed is food
→ Anil eats peanuts & 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) → likes (John, x)
b. food (Apple) ∧ food (vegetables)
c. $\forall x \forall y$: eats (x, y) ∧ ¬ killed (x) → food (y)

d. eats (Anil, Peanuts) ∧ alive (Anil)
e. ∀x : eats (Anil, x) → eats (Harry, x)
f. ∀x : ¬killed (x) → alive (x)
g. ∀x : alive (x) → ¬killed (x)
h. likes (John, Peanuts)

Elimination Implication

a. ∀x ¬ food(x) ∨ likes (John, x)
b. food (Apple) ∧ food (Vegitable)
c. ∀x ∀y ¬[eats (x,y) ∧ ¬ killed (x)] ∨ food (y)
d. eats (Anil, Peanuts) ∧ alive (Anil)
e. ∀x ¬ eats (Anil, x) ∨ eats (Harry, x)
f. ∀x ¬ [¬ killed (x)] ∨ alive(x)
g. ∀x ¬ alive (x) ∨ ¬ killed (x)
h. likes (John, Peanuts)

More negation (¬) inwards and rewrite

a. ∀x ¬ food(x) ∨ likes (John, x)
b. food(Apple) ∧ food (vegitable)
c. ∀x ∀y ¬ eats (x,y) ∨ killed (x) ∨ food(y)
d. eats (Anil, Peanuts) ∧ alive (Anil)
e. ∀x ¬ eats (Anil, x) ∨ eats (Harry, x)
f. ∀x [killed (x)] ∨ alive(x)
g. ∀x ¬ alive (x) ∨ ¬ killed (x)
h. likes (John, Peanuts)

Rename Variable or standardize variables

a. ∀x ¬ food (x) ∨ likes (John, x)
b. food (Apple) ∧ food (vegetable)
c. ∀y ∀z ¬ eats (y, z) ∨ killed (y) ∨ food (z)
d. eats (Anil, Peanuts) ∧ alive (Anil)
e. ∀w ⟹ ¬ eats (Anil, w) ∨ eats (Harry, w)

f. $\forall q$ killed (q)] v alive (q)

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

h. likes (John, Peanuts)

Drop universal:

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

b. food (Apple)

c. food (vegetables)

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

e. eats (Anil, Peanuts)

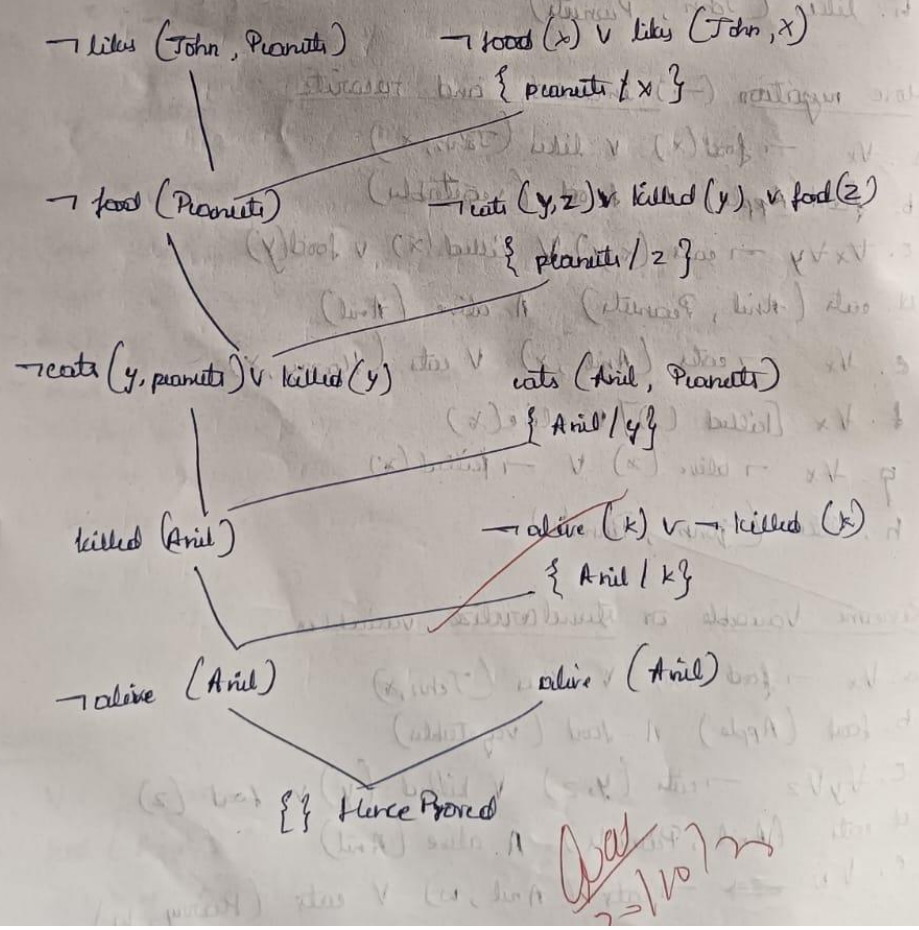
f. alive (Anil)

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

h. killed (g) v alive (g)

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

j. likes (John, Peanuts)



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
```

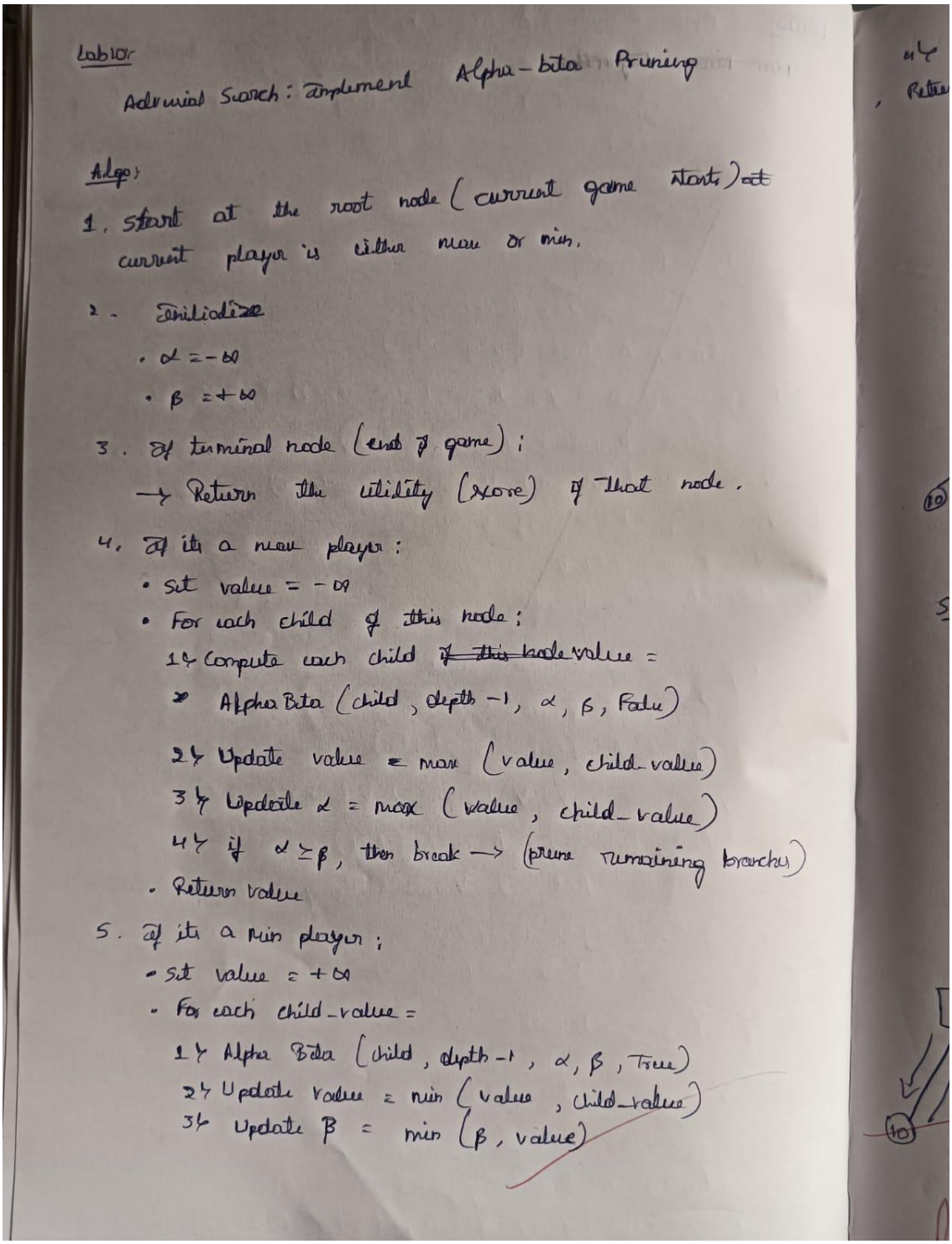
```
    parents, clause = tree[node]
    print("  " * indent + f"{node}: {set(clause) or '{}'}")
    for p in parents:
        show(p, indent + 4)

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.



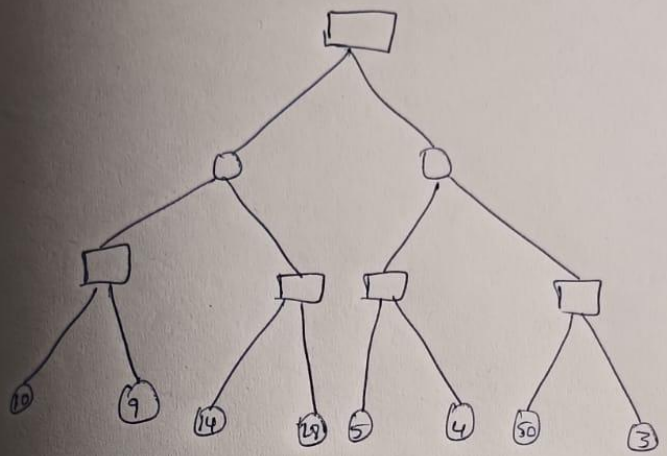
Lab 10

Adversial Search: Implement Alpha-beta Pruning

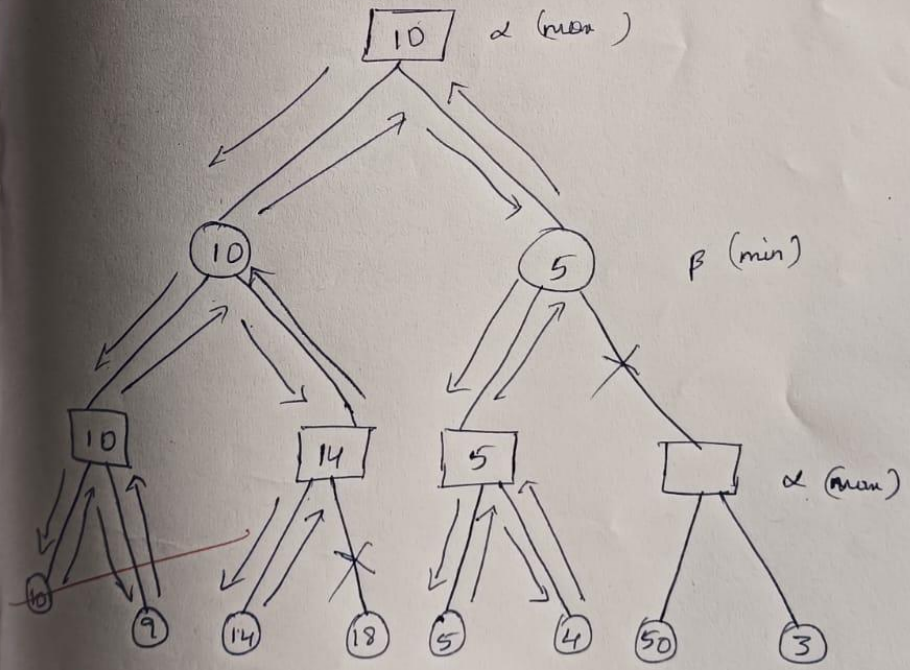
Algo:

1. Start at the root node (current game state) set current player is either max or min.
2. Initialize
   - $\alpha = -\infty$
   - $\beta = +\infty$
3. If terminal node (end of game):
   → Return the utility (score) of that node.
4. If its a max player:
   - Set value = $-\infty$
   - For each child of this node:
     1. Compute each child ~~if this node~~ value =
        Alpha Beta (child, depth −1, α, β, False)
     2. Update value = max (value, child_value)
     3. Update α = max (value, child_value)
     4. if $\alpha \geq \beta$, then break → (prune remaining branches)
   - Return value
5. If its a min player:
   - Set value = $+\infty$
   - For each child_value =
     1. Alpha Beta (child, depth −1, α, β, True)
     2. Update value = min (value, child_value)
     3. Update β = min (β, value)

If $\alpha \geq \beta$, then → (prune remaining branches)
Return Value



Soln:-



Quiz
5/12/23

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
```