VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Spurthi Reddy P (1BM23CS338)

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 **Spurthi Reddy P (1BM23CS338),** 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

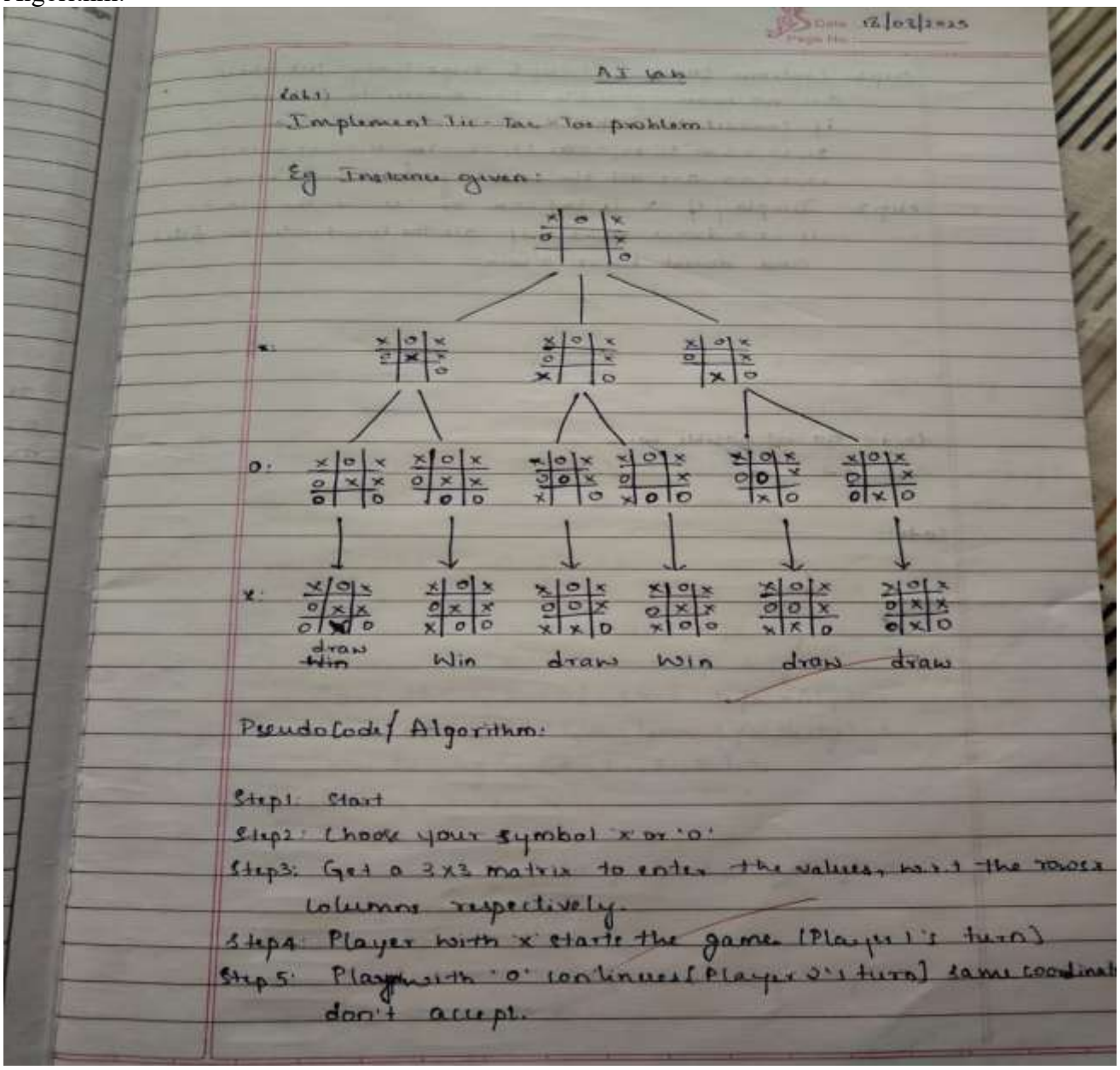
Index

Github Link:
https://github.com/Spurthi-338/AILAB-338

Program 1

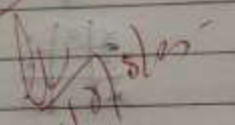
Implement Tic –Tac –Toe Game

Algorithm:



Step6: Continue step 4 and step 5 respectively till there are no more possible coordinates to fill or if coordinates with 11, 22, 33 or 11, 12, 13 or 21, 22, 23 or 31, 32, 33 or 13, 22, 31 or 11, 21, 31 or 12, 22, 32 or 13, 23, 33 are all the same symbols then it wins.

Step7: Display if 'X' is the win or 'O' is the win or it is a draw match if all the coordinates are filled and doesn't have a win.



Cost = no of posible wins

÷2

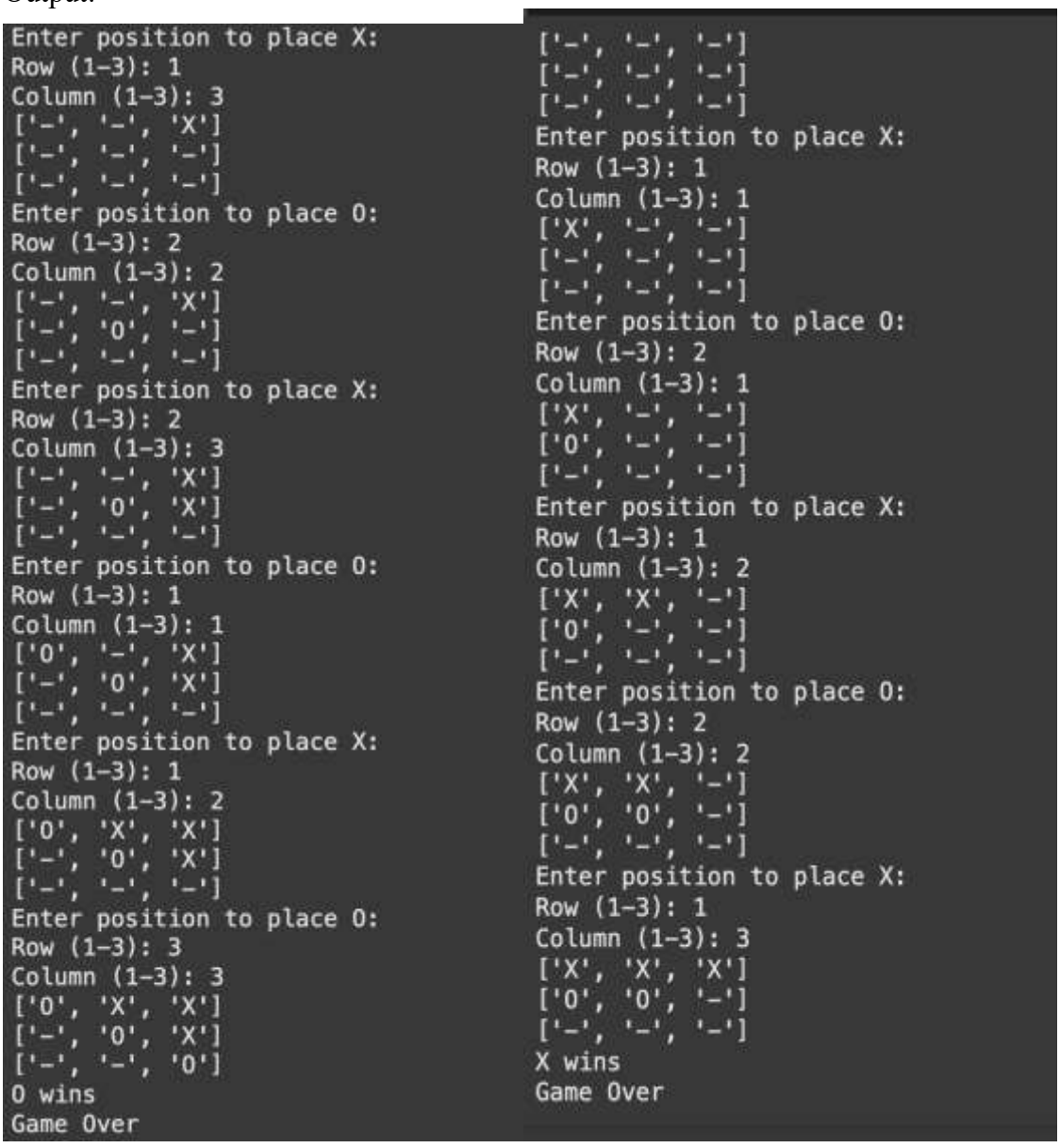
~~Code:~~

Code:

```
board = [['-', '-', '-'],
         ['-', '-', '-'],
         ['-', '-', '-']]

def print_board():
    for row in board:
        print(row)

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

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

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

    return False

def play_game():
    print_board()
    for turn in range(9):
        if turn % 2 == 0:
            player = 'X'
        else:
            player = 'O'

        print(f"Enter position to place {player}: ")
        row = int(input("Row (1-3): ")) - 1
        col = int(input("Column (1-3): ")) - 1

        if board[row][col] == '-':
            board[row][col] = player
        else:
            print("Cell already taken, try again!")
            continue

        print_board()

        if check_winner(player):
            print(f"{player} wins")
```

```
        print("Game Over")
        return

    print("It's a Draw!")
    print("Game Over")

play_game()
```

Output:

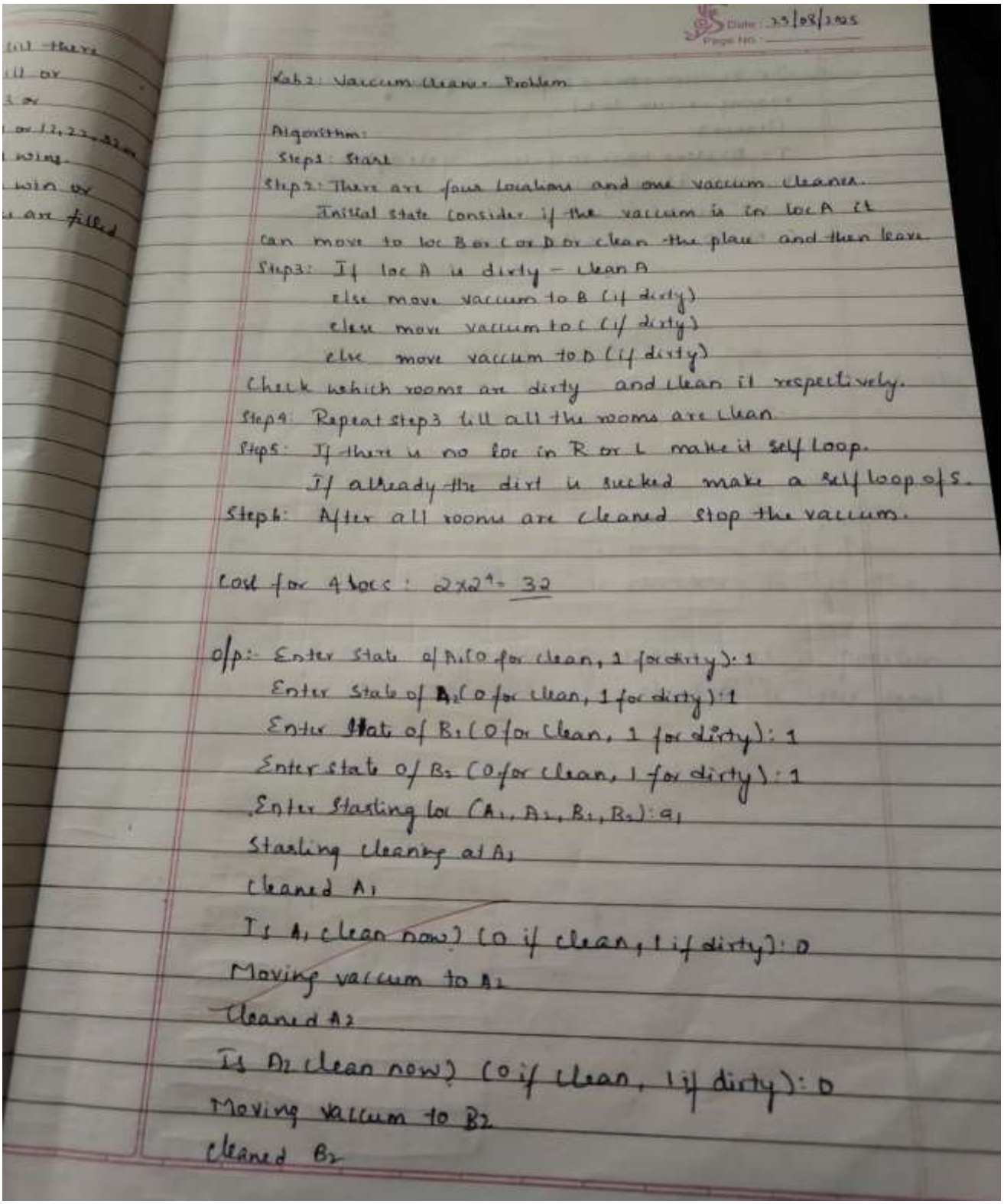


```
Enter position to place X:
Row (1-3): 1
Column (1-3): 3
['-', '-', 'X']
['-', '-', '-']
['-', '-', '-']
Enter position to place O:
Row (1-3): 2
Column (1-3): 2
['-', '-', 'X']
['-', 'O', '-']
['-', '-', '-']
Enter position to place X:
Row (1-3): 2
Column (1-3): 3
['-', '-', 'X']
['-', 'O', 'X']
['-', '-', '-']
Enter position to place O:
Row (1-3): 1
Column (1-3): 1
['O', '-', 'X']
['-', 'O', 'X']
['-', '-', '-']
Enter position to place X:
Row (1-3): 1
Column (1-3): 2
['O', 'X', 'X']
['-', 'O', 'X']
['-', '-', '-']
Enter position to place O:
Row (1-3): 3
Column (1-3): 3
['O', 'X', 'X']
['-', 'O', 'X']
['-', '-', 'O']
O wins
Game Over
```

```
['-', '-', '-']
['-', '-', '-']
['-', '-', '-']
Enter position to place X:
Row (1-3): 1
Column (1-3): 1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
Enter position to place O:
Row (1-3): 2
Column (1-3): 1
['X', '-', '-']
['O', '-', '-']
['-', '-', '-']
Enter position to place X:
Row (1-3): 1
Column (1-3): 2
['X', 'X', '-']
['O', '-', '-']
['-', '-', '-']
Enter position to place O:
Row (1-3): 2
Column (1-3): 2
['X', 'X', '-']
['O', 'O', '-']
['-', '-', '-']
Enter position to place X:
Row (1-3): 1
Column (1-3): 3
['X', 'X', 'X']
['O', 'O', '-']
['-', '-', '-']
X wins
Game Over
```

```
Enter position to place X:
Row (1-3): 1
Column (1-3): 1
['X', '-', '-']
['-', '-', '-']
['-', '-', '-']
Enter position to place O:
Row (1-3): 2
Column (1-3): 2
['X', '-', '-']
['-', 'O', '-']
['-', '-', '-']
Enter position to place X:
Row (1-3): 1
Column (1-3): 2
['X', 'X', '-']
['-', 'O', '-']
['-', '-', '-']
Enter position to place O:
Row (1-3): 1
Column (1-3): 3
['X', 'X', 'O']
['-', 'O', '-']
['-', '-', '-']
Enter position to place X:
Row (1-3): 3
Column (1-3): 1
['X', 'X', 'O']
['-', 'O', '-']
['X', '-', '-']
Enter position to place O:
Row (1-3): 2
Column (1-3): 1
['X', 'X', 'O']
['O', 'O', '-']
['X', '-', '-']
Enter position to place X:
Row (1-3): 2
Column (1-3): 3
['X', 'X', 'O']
['O', 'O', 'X']
['X', '-', '-']
Enter position to place O:
Row (1-3): 3
Column (1-3): 2
['X', 'X', 'O']
['O', 'O', 'X']
['X', 'O', '-']
Enter position to place X:
Row (1-3): 3
Column (1-3): 3
['X', 'X', 'O']
['O', 'O', 'X']
['X', 'O', 'X']
It's a Draw!
Game Over
```

Implement vacuum cleaner agent



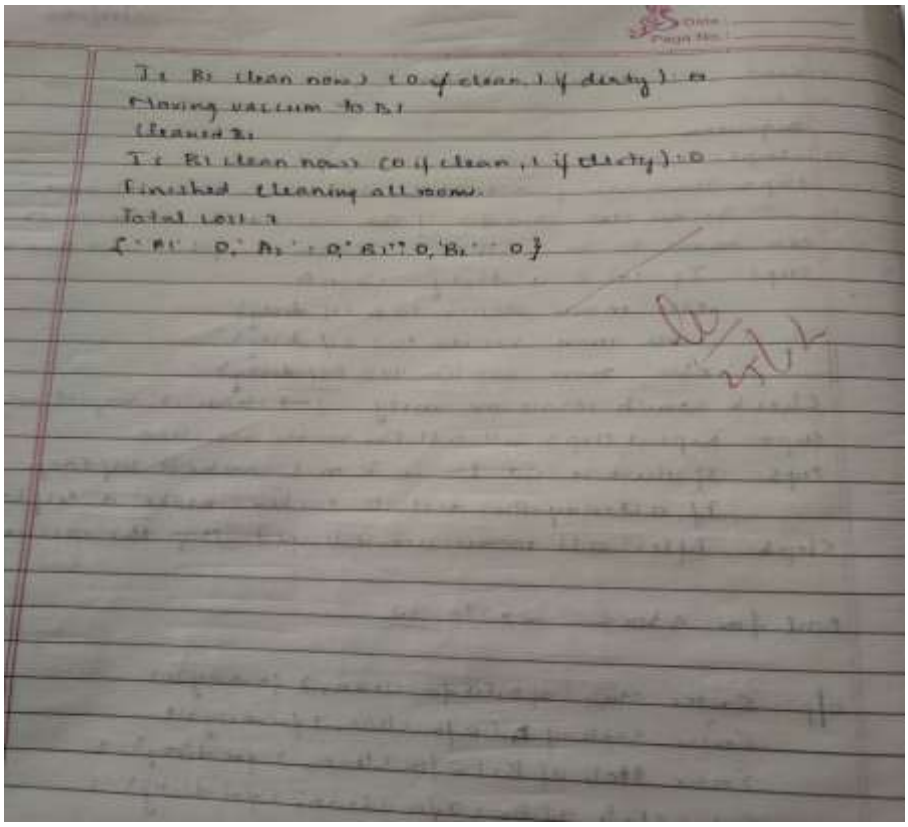
Date: 25/08/2025
Page No.:

Lab2: Vaccum Cleaner Problem

Algorithm:
Step1: Start
Step2: There are four locations and one vaccum cleaner.
Initial state consider if the vaccum is in loc A it can move to loc B or C or D or clean the place and then leave.
Step3: If loc A is dirty - clean A
else move vaccum to B (if dirty)
else move vaccum to C (if dirty)
else move vaccum to D (if dirty)
Check which rooms are dirty and clean it respectively.
Step4: Repeat step3 till all the rooms are clean
Step5: If there is no loc in R or L make it self loop.
If already the dirt is sucked make a self loop of 5.
Step6: After all rooms are cleaned stop the vaccum.

cost for 4 locs: $2 \times 2^4 = 32$

o/p: Enter State of A_1(0 for clean, 1 for dirty): 1
Enter State of A_2(0 for clean, 1 for dirty): 1
Enter State of B_1(0 for clean, 1 for dirty): 1
Enter state of B_2 (0 for clean, 1 for dirty): 1
Enter Starting loc (A_1, A_2, B_1, B_2): a_1
Starting cleaning at A_1
cleaned A_1
Is A_1 clean now) (0 if clean, 1 if dirty): 0
Moving vaccum to A_2
cleaned A_2
Is A_2 clean now) (0 if clean, 1 if dirty): 0
Moving vaccum to B_2
cleaned B_2



Code:

```
def vacuum_agent_2x2():
    state = {
        'A1': int(input("Enter state of A1 (0 for clean, 1 for dirty): ")),
        'A2': int(input("Enter state of A2 (0 for clean, 1 for dirty): ")),
        'B1': int(input("Enter state of B1 (0 for clean, 1 for dirty): ")),
        'B2': int(input("Enter state of B2 (0 for clean, 1 for dirty): "))
    }

    location = input("Enter starting location (A1, A2, B1, B2): ").upper()
    cost = 0

    if all(state[loc] == 0 for loc in state):
        print("All rooms clean. Turning vacuum off.")
        print(f"Cost: {cost}")
        print(state)
        return

    traversal_order = ['A1', 'A2', 'B2', 'B1']

    while traversal_order[0] != location:
        traversal_order.append(traversal_order.pop(0))

    print(f"Starting cleaning at {location}")
```

```
    for idx, loc in enumerate(traversal_order):
        if idx == 0:
            pass
        else:
            print(f"Moving vacuum to {loc}")
            cost += 1

        if state[loc] == 1:
            print(f"Cleaned {loc}.")
            state[loc] = 0
            cost += 1  # cost for cleaning

        print(f"Is {loc} clean now? (0 if clean, 1 if dirty): {state[loc]}")

    print("Finished cleaning all rooms.")
    print(f"Total cost: {cost}")
    print(state)


vacuum_agent_2x2()
```

Output:

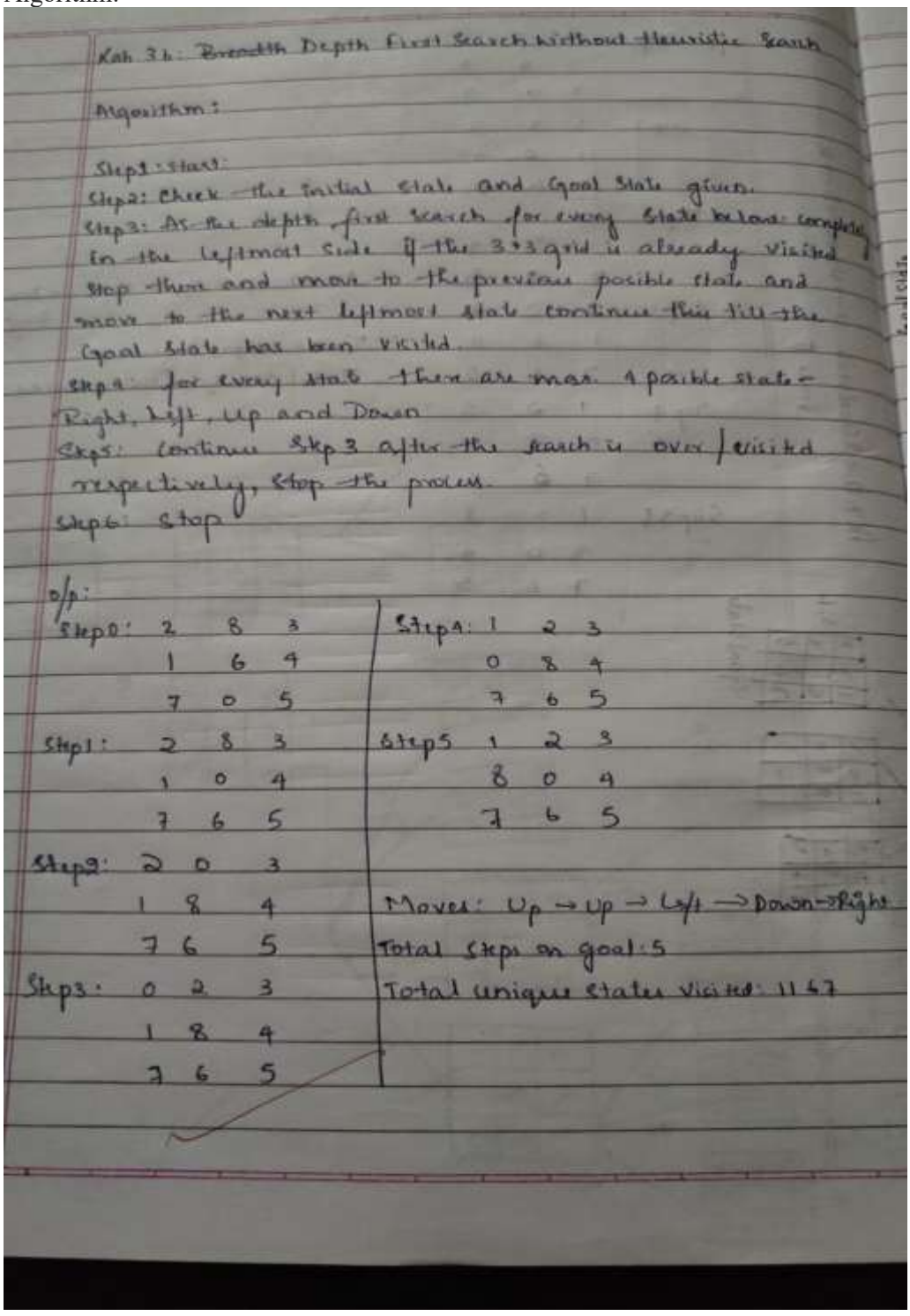
```
Enter state of A1 (0 for clean, 1 for dirty): 1
Enter state of A2 (0 for clean, 1 for dirty): 1
Enter state of B1 (0 for clean, 1 for dirty): 1
Enter state of B2 (0 for clean, 1 for dirty): 1
Enter starting location (A1, A2, B1, B2): a1
Starting cleaning at A1
Cleaned A1.
Is A1 clean now? (0 if clean, 1 if dirty): 0
Moving vacuum to A2
Cleaned A2.
Is A2 clean now? (0 if clean, 1 if dirty): 0
Moving vacuum to B2
Cleaned B2.
Is B2 clean now? (0 if clean, 1 if dirty): 0
Moving vacuum to B1
Cleaned B1.
Is B1 clean now? (0 if clean, 1 if dirty): 0
Finished cleaning all rooms.
Total cost: 7
{'A1': 0, 'A2': 0, 'B1': 0, 'B2': 0}
```

Spurthireddyp

Program2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:



Aim 3: Breadth Depth First Search without Heuristic Search

Algorithm:

Step1: Start.
Step2: Check the initial state and Goal state given.
Step3: As the depth first search for every state below: completely in the leftmost side if the 3*3 grid is already visited stop there and move to the previous posible state and move to the next leftmost state continue this till the Goal State has been visited.
Step4: for every state there are max. 4 posible state - Right, Left, Up and Down
Step5: Continue Step 3 after the search is over/visited respectively, Stop the process.
Step6: Stop

o/p:

Step0: 2 8 3
1 6 4
7 0 5

Step1: 2 8 3
1 0 4
7 6 5

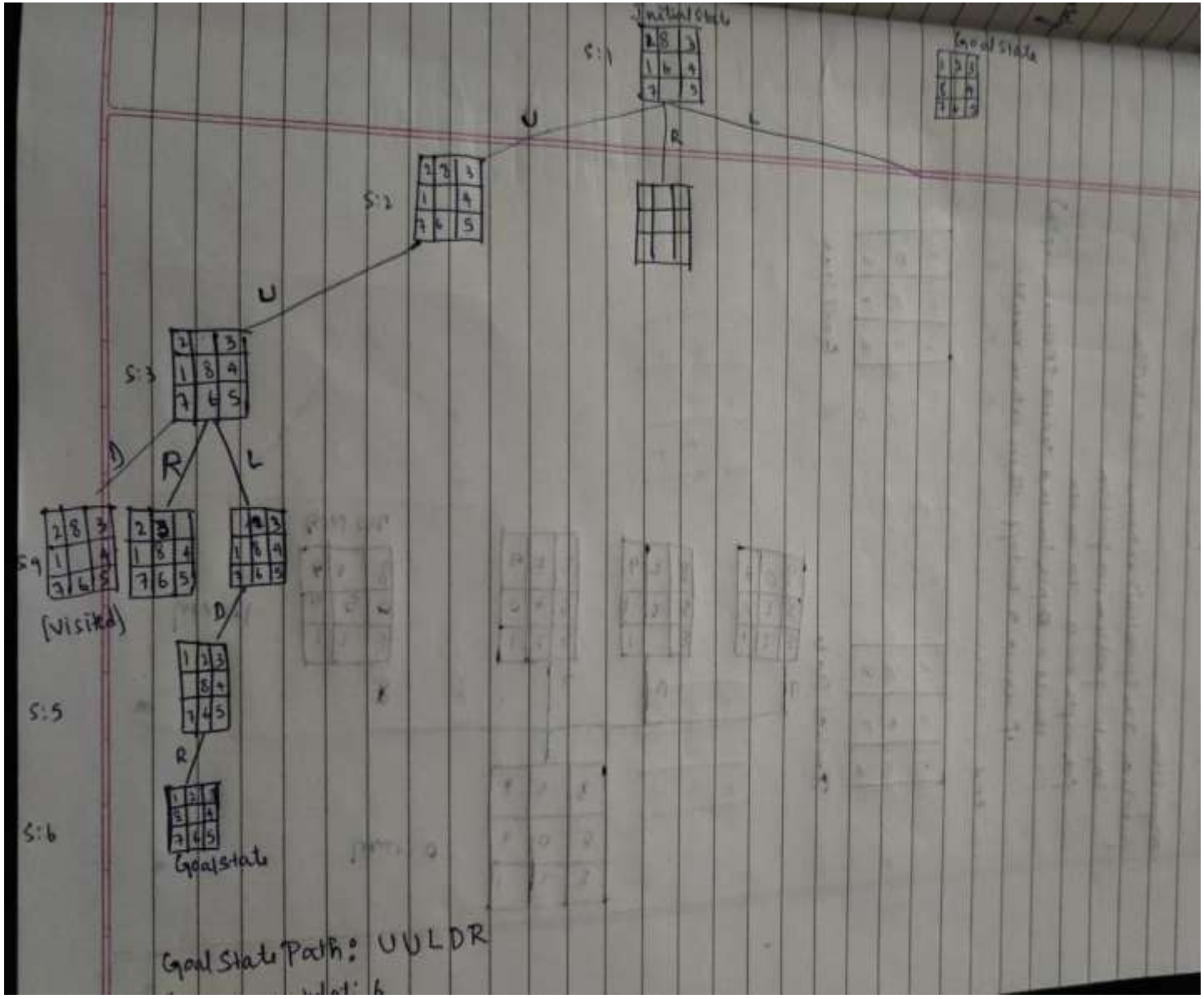
Step2: 2 0 3
1 8 4
7 6 5

Step3: 0 2 3
1 8 4
7 6 5

Step4: 1 2 3
0 8 4
7 6 5

Step5 1 2 3
8 0 4
7 6 5

Moves: Up → Up → Left → Down → Right
Total steps on goal: 5
Total unique states visited: 1147


Initial state
Goal state
S:1
U
R
L
S:2
U
S:3
U
R
L
S:4
(Visited)
D
S:5
R
S:6
Goalstate
Goal State Path: UULDR

Code:

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

def is_goal(state):
    return state == "123804765"

def get_neighbors(state):
    neighbors = []
    moves = {'Up': -3, 'Down': 3, 'Left': -1, 'Right': 1}
    zero_index = state.index('0')

    for move, pos_change in moves.items():
        new_index = zero_index + pos_change

        if move == 'Left' and zero_index % 3 == 0:
            continue
        if move == 'Right' and zero_index % 3 == 2:
            continue
        if move == 'Up' and zero_index < 3:
            continue
        if move == 'Down' and zero_index > 5:
            continue

        state_list = list(state)
        state_list[zero_index], state_list[new_index] = state_list[new_index], state_list[zero_index]
        neighbors.append(("".join(state_list), move))

    return neighbors

def dfs(start_state, max_depth=20):
    visited = set()
    stack = [(start_state, [], [], 0)]  # (state, path, moves, depth)

    while stack:
        current_state, path, moves, depth = stack.pop()

        if current_state in visited:
            continue

        visited.add(current_state)

        if is_goal(current_state):
            steps = path + [current_state]
            print("\nGoal reached (DFS)!\n")
```

```
            for i, step in enumerate(steps):
                print(f"Step {i}:")
                print_state(step)
            print("Moves:", " -> ".join(moves))
            print("Total steps to goal:", len(steps) - 1)
            print("Total unique states visited:", len(visited))
            return

        if depth < max_depth:
            for neighbor, move in reversed(get_neighbors(current_state)):
                if neighbor not in visited:
                    stack.append((neighbor, path + [current_state], moves + [move], depth + 1))

    print(f"No solution found within depth limit of {max_depth}!")
start = "283164705"
dfs(start, max_depth=20)
```

Output:
Goal reached (DFS)!

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:

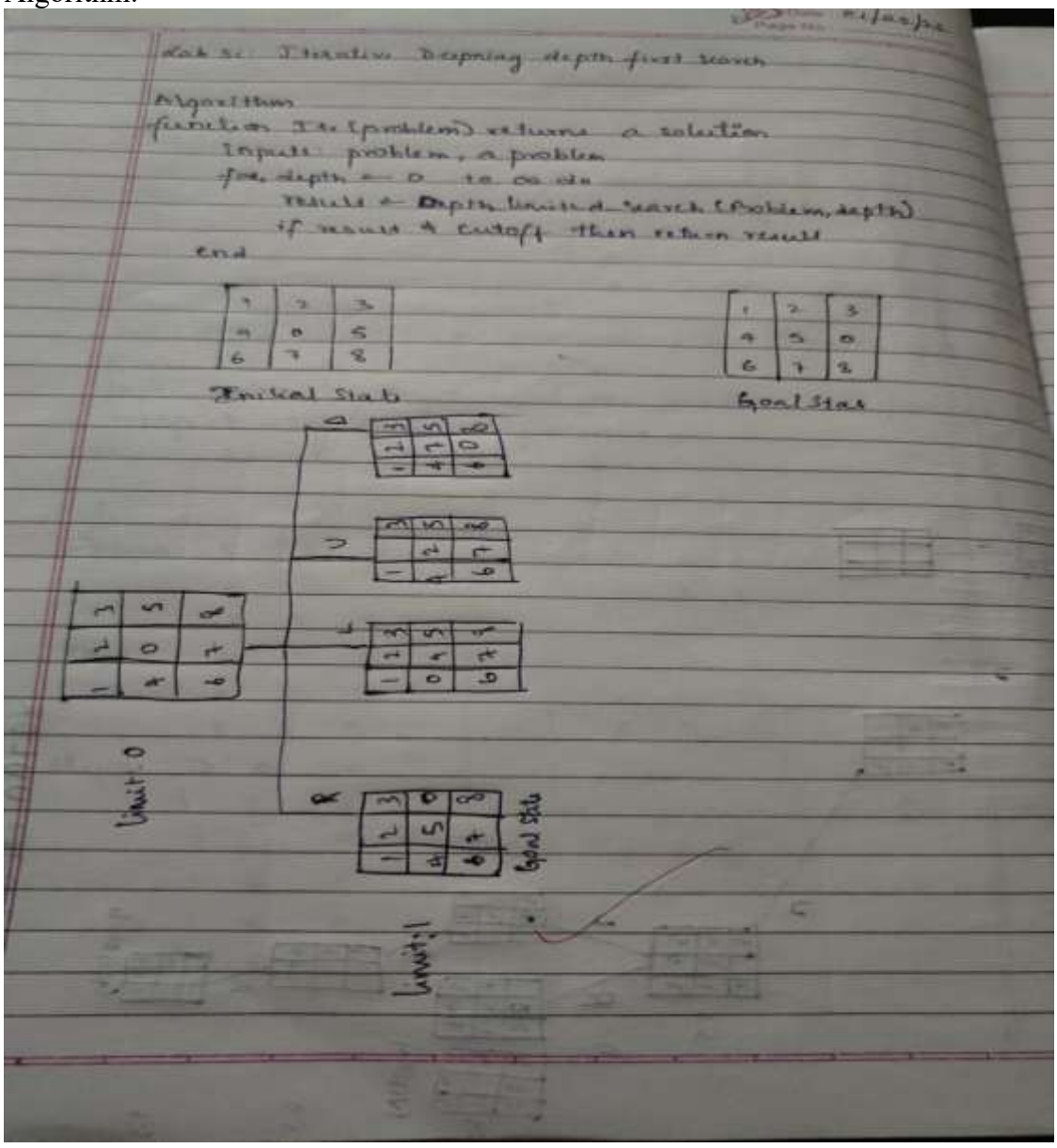
1 2 3
8 0 4
7 6 5

Moves: Up -> Up -> Left -> Down -> Right
Total steps to goal: 5
Total unique states visited: 1167

Spurthi Reddy P

Implement Iterative deepening search algorithm

Algorithm:



o/p:

Searching with depth limit=0

Searching with depth limit=1

Sol found in 1 move

Step0:

1	2	3
4	0	5
6	7	8

Step1:

1	2	3
4	5	0
6	7	8

Code:

```
from collections import deque

# Board size (3x3 puzzle)
N = 3

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

# Helper: Find position of blank tile
def find_blank(state):
    idx = state.index("_")
    return divmod(idx, N)

# Helper: Swap tiles
def swap(state, i1, j1, i2, j2):
    s = list(state)
    idx1, idx2 = i1*N+j1, i2*N+j2
    s[idx1], s[idx2] = s[idx2], s[idx1]
    return tuple(s)

# Expand node: Generate next states
def expand(state):
    x, y = find_blank(state)
    children = []
    for dx, dy in moves:
        nx, ny = x+dx, y+dy
        if 0 <= nx < N and 0 <= ny < N:
            children.append(swap(state, x, y, nx, ny))
    return children

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

    if limit == 0:
        return None

    visited.add(state)
    for child in expand(state):
        if child not in visited:
            result = dls(child, goal, limit-1, path+[child], visited)
            if result is not None:
                return result
    visited.remove(state)
    return None
```

```
def iddfs(start, goal, max_depth=20):
    for depth in range(max_depth):
        visited = set()
        result = dls(start, goal, depth, [start], visited)
        if result is not None:
            return result
    return None


initial = (2,8,3,1,6,4,7,"_",5)
goal = (1,2,3,8,"_",4,7,6,5)

solution = iddfs(initial, goal, max_depth=30)

if solution:
    print("Solution found in", len(solution)-1, "moves:\n")
    for step in solution:
        for i in range(0, 9, 3):
            print(step[i:i+3])
        print()
else:
    print("No solution found within depth limit")
```

Output:
Solution found in 5 moves:

```
(2, 8, 3)
(1, 6, 4)
(7, '_', 5)

(2, 8, 3)
(1, '_', 4)
(7, 6, 5)

(2, '_', 3)
(1, 8, 4)
(7, 6, 5)

('_', 2, 3)
(1, 8, 4)
(7, 6, 5)

(1, 2, 3)
('_', 8, 4)
(7, 6, 5)
```

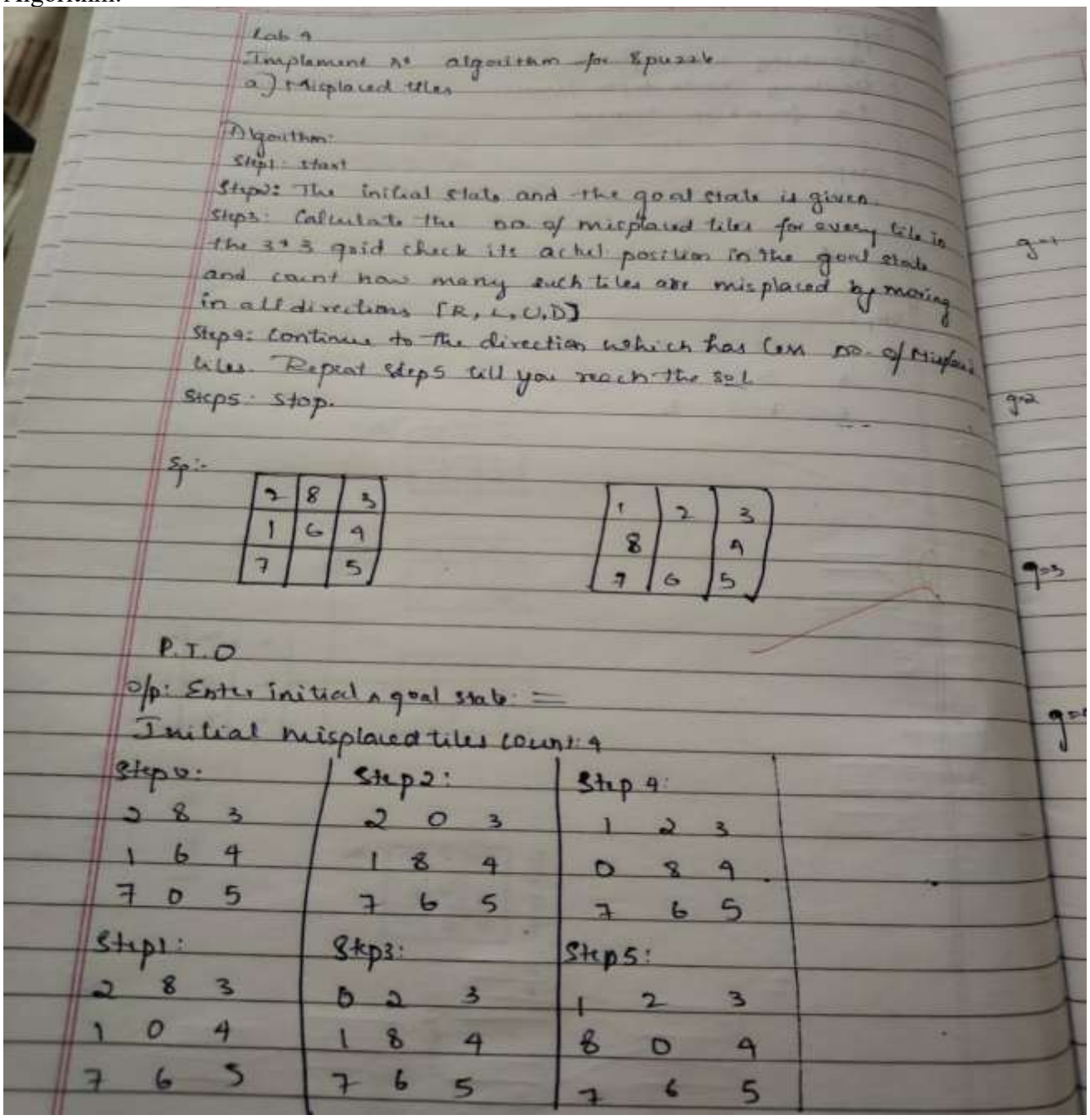
(1, 2, 3)
(8, '_', 4)
(7, 6, 5)

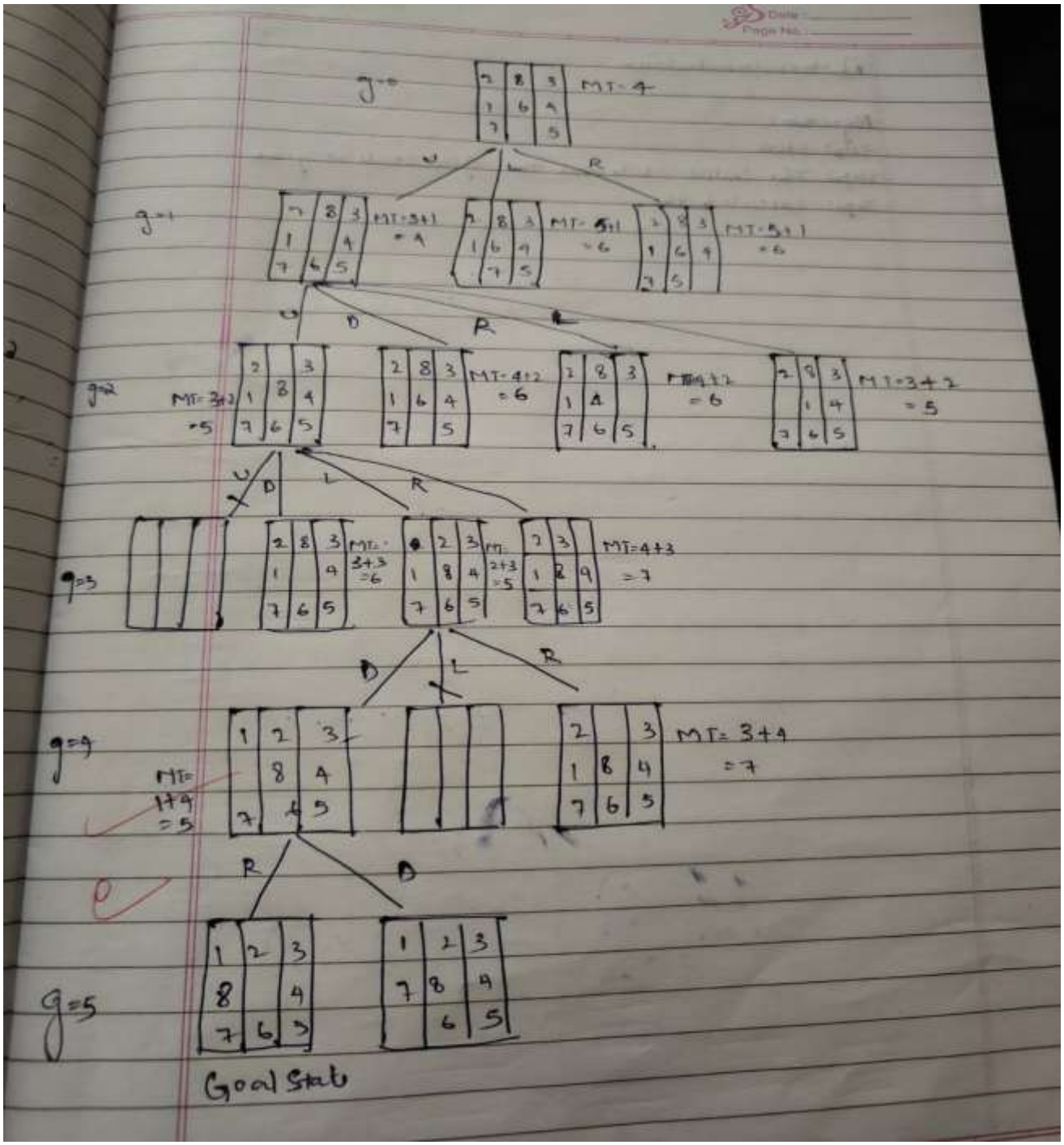
Program3

Implement A* search algorithm

a) Misplaced Tiles

Algorithm:





Code: from collections import deque

```
def solve_puzzle_all_paths(initial, goal, max_depth=50):
    visited = set()
    initial_t = tuple(map(tuple, initial))
    visited.add(initial_t)

    queue = deque()
```

```
    queue.append((initial, [initial]))

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

        if current == goal:
            return path

        if len(path) > max_depth:
            continue

        neighbors_states = neighbors(current)
        misplaced_list = [(state, misplaced_tiles(state, goal)) for state in neighbors_states]

        if not misplaced_list:
            continue

        min_misplaced = min(m for _, m in misplaced_list)

        for state, m in misplaced_list:
            state_t = tuple(map(tuple, state))
            if m == min_misplaced and state_t not in visited:
                visited.add(state_t)
                queue.append((state, path + [state]))

    return None

def main():
    initial_state = input_puzzle("initial")
    goal_state = input_puzzle("goal")

    print("\nInitial State:")
    print_state(initial_state)
    print("Goal State:")
    print_state(goal_state)

    misplaced = misplaced_tiles(initial_state, goal_state)
    print(f"Initial misplaced tiles count: {misplaced}\n")

    max_depth = int(input("Enter maximum search depth (e.g., 50): "))

    solution_path = solve_puzzle_all_paths(initial_state, goal_state, max_depth=max_depth)

    if solution_path is None:
        print("Could not solve the puzzle within the depth limit.")
    else:
        for step, state in enumerate(solution_path):
```

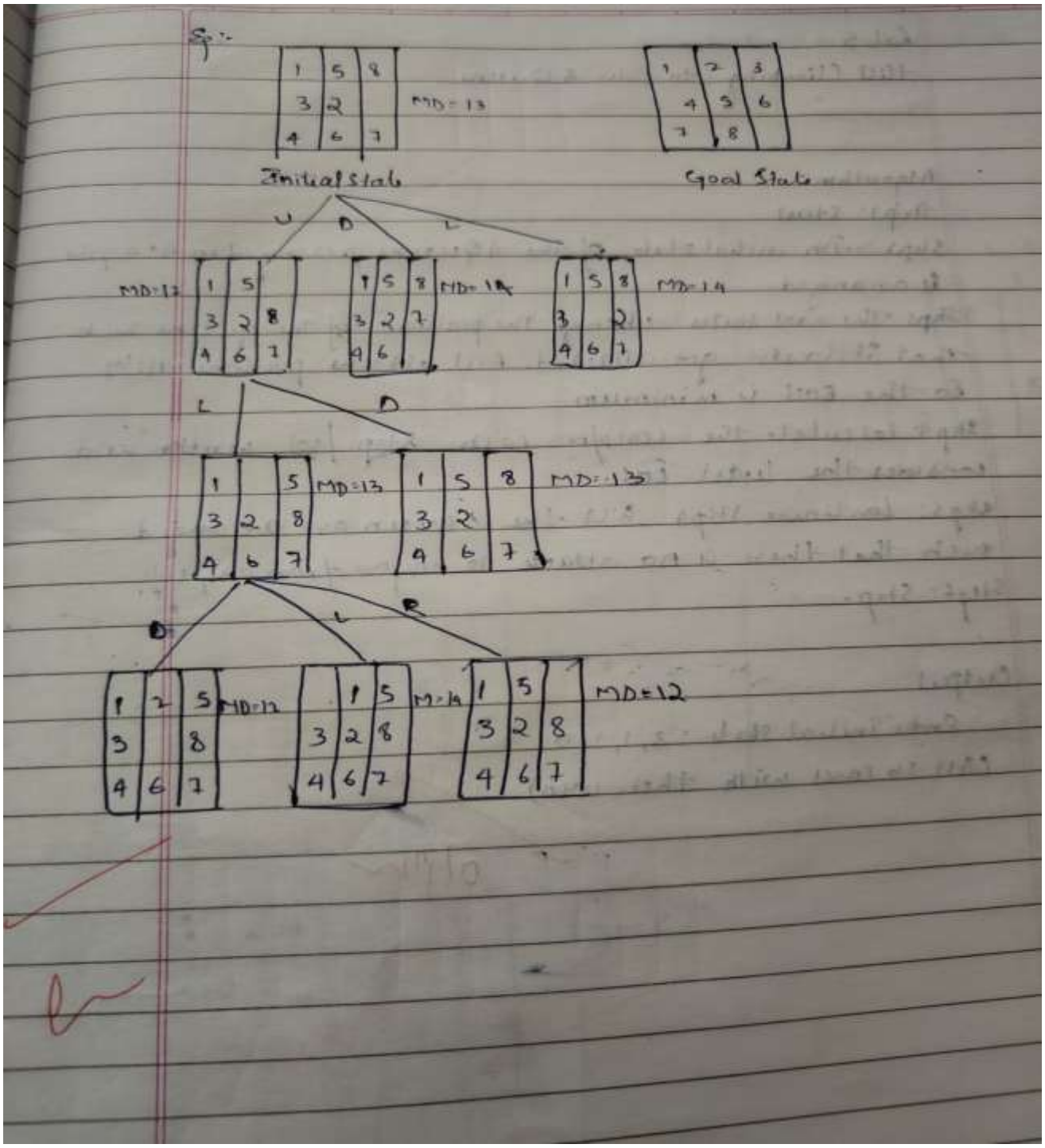
```
        print(f"Step {step}:")
        print_state(state)

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

Output:

```
Enter the initial puzzle state (use 0 for the blank):
Row 1: 1 2 3
Row 2: 4 0 6
Row 3: 7 5 8
Enter the goal puzzle state (use 0 for the blank):
Row 1: 1 2 3
Row 2: 4 5 6
Row 3: 7 8 0

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

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

Initial misplaced tiles count: 3

Enter maximum search depth: 20
```

b) Manhattan Distance

Algorithm:


Sol:-
MD=13
Initial State
Goal State
U
D
L
MD=12
MD=14
L
D
MD=13
MD=13
D
L
R
MD=12
MD=14
MD=12

Code:

```
from heapq import heappush, heappop

class PuzzleState:
    def __init__(self, board, parent=None, move=None, depth=0):
        self.board = board
        self.parent = parent
        self.move = move
        self.depth = depth
        self.zero_pos = self.board.index(0)

    def is_goal(self, goal):
        return self.board == goal

    def get_moves(self):
        moves = []
        zero = self.zero_pos
        row, col = zero // 3, zero % 3

        directions = {
            'Up': (row - 1, col),
            'Down': (row + 1, col),
            'Left': (row, col - 1),
            'Right': (row, col + 1)
        }

        for move, (r, c) in directions.items():
            if 0 <= r < 3 and 0 <= c < 3:
                new_zero = r * 3 + c
                new_board = list(self.board)
                new_board[zero], new_board[new_zero] = new_board[new_zero], new_board[zero]
                moves.append(PuzzleState(tuple(new_board), self, move, self.depth + 1))
        return moves

    def manhattan_distance(self, goal):
        distance = 0
        for i, tile in enumerate(self.board):
            if tile != 0:
                goal_index = goal.index(tile)
                current_row, current_col = i // 3, i % 3
                goal_row, goal_col = goal_index // 3, goal_index % 3
                distance += abs(current_row - goal_row) + abs(current_col - goal_col)
        return distance

    def __lt__(self, other):
        return True
```

```
def a_star(start, goal):
    open_list = []
    closed_set = set()
    start_state = PuzzleState(start)
    heappush(open_list, (start_state.manhattan_distance(goal), start_state))

    while open_list:
        _, current = heappop(open_list)

        if current.is_goal(goal):
            return reconstruct_path(current)

        closed_set.add(current.board)

        for neighbor in current.get_moves():
            if neighbor.board in closed_set:
                continue
            cost = neighbor.depth + neighbor.manhattan_distance(goal)
            heappush(open_list, (cost, neighbor))

    return None

def reconstruct_path(state):
    path = []
    while state.parent is not None:
        path.append(state.move)
        state = state.parent
    path.reverse()
    return path

if __name__ == "__main__":
    start = (2, 8, 3,
             1, 6, 4,
             7, 0, 5)

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

    print("Spurthi Reddy P (1BM23CS338)")
    solution = a_star(start, goal)
    if solution:
        print(f"Solution found in {len(solution)} moves: {solution}")
    else:
        print("No solution found.")
```

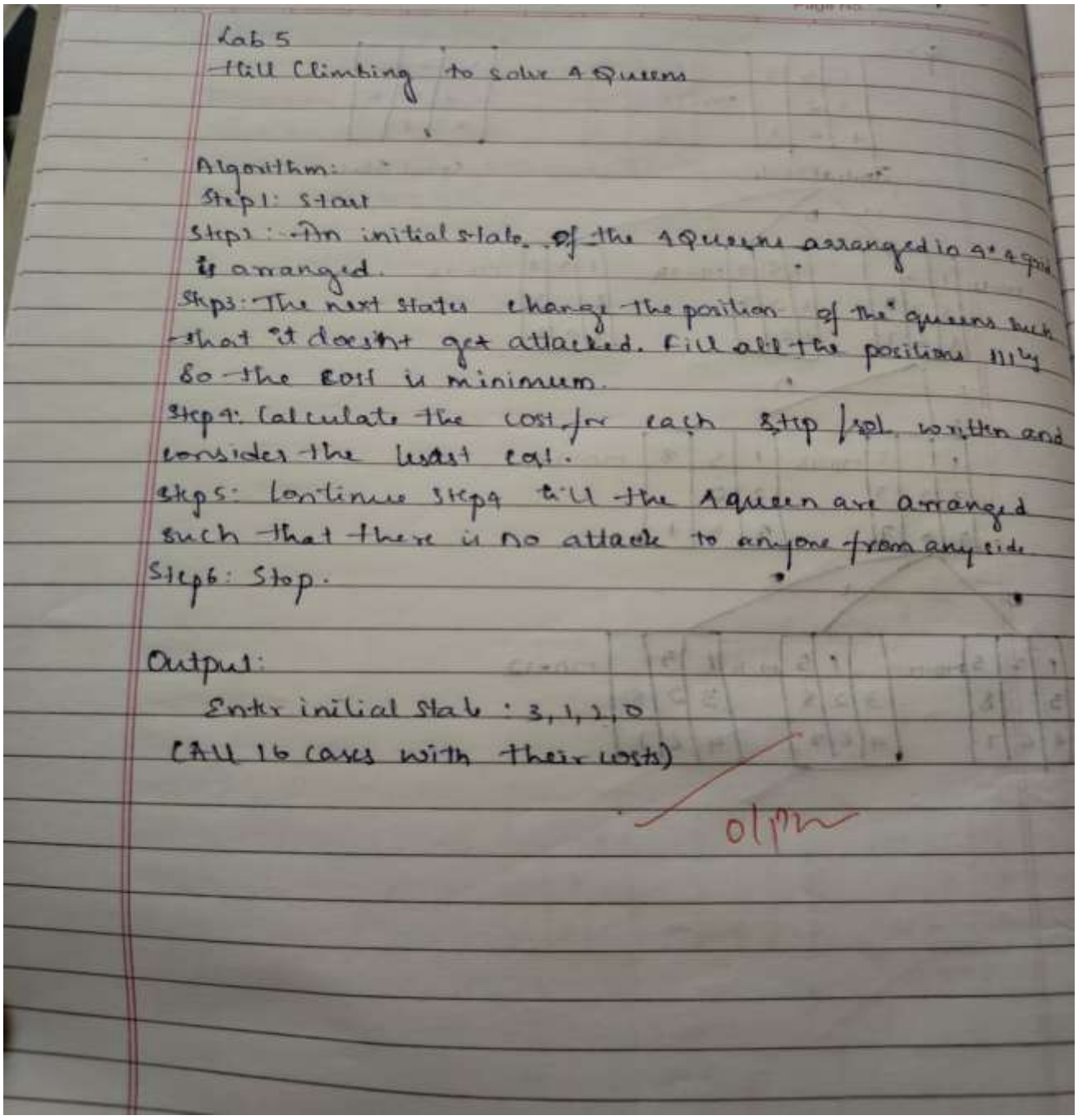
Output:

```
Spurthi Reddy P (1BM23CS338)
Solution found in 5 moves: ['Up', 'Up', 'Left', 'Down', 'Right']
```

Program4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Lab 5

Hill Climbing to solve 4 Queens

Algorithm:

Step1: Start

Step2: An initial state of the 4 Queens arranged in 4*4 grid is arranged.

Step3: The next states change the position of the queens such that it doesn't get attacked. Fill all the positions like so the cost is minimum.

Step4: Calculate the cost for each step / sol. written and consider the least cal.

Step5: Continue Step4 till the 4 queen are arranged such that there is no attack to anyone from any side.

Step6: Stop.

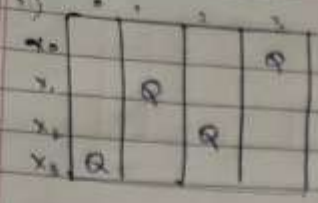
Output:

Enter initial state : 3,1,2,0

(All 16 cases with their costs)

o/p

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

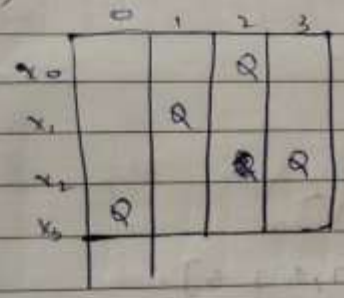


Cost = 2

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

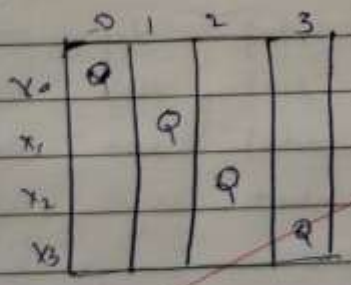
Cost = 1

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



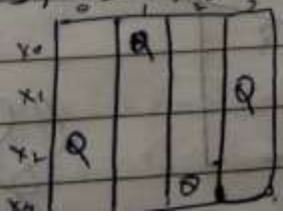
Cost = 1

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



Cost = 6

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



Cost = 0

Code:

```
from itertools import permutations

def compute_cost(state):
    cost = 0
    for i in range(4):
        for j in range(i+1, 4):
            if abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def print_board(state):
    board = ""
    for row in range(4):
        for col in range(4):
            board += " Q " if state[col] == row else " . "
        board += "\n"
    return board

def is_valid_permutation(state):
    return sorted(state) == [0, 1, 2, 3]

def main():
    all_perms = list(permutations(range(4)))
    print(f"Total permutations: {len(all_perms)}\n")

    while True:
        user_input = input("Enter initial state as 4 comma-separated integers from 0 to 3 (e.g. 1,3,0,2):
")
        try:
            user_state = tuple(int(x.strip()) for x in user_input.split(','))
            if len(user_state) != 4 or not is_valid_permutation(user_state):
                print("Invalid input! Please enter a permutation of 0,1,2,3 (each number exactly once).")
                continue
            break
        except ValueError:
            print("Invalid input! Please enter integers separated by commas.")

    cost = compute_cost(user_state)
    print(f"\nYour input state: {user_state} | Diagonal Conflicts Cost: {cost}")
    print(print_board(user_state))
    print("-" * 30)

    print("Showing first 16 permutations with their costs:\n")
    for idx, perm in enumerate(all_perms[:16], 1):
        cost = compute_cost(perm)
        print(f"Case {idx}: {perm} | Diagonal Conflicts Cost: {cost}")
```

```
        print(print_board(perm))
        print("-" * 30)

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

Output:

```
Total permutations: 24

Enter initial state as 4 comma-separated integers from 0 to 3 (e.g. 1,3,0,2): 3,1,2,0

Your input state: (3, 1, 2, 0) | Diagonal Conflicts Cost: 2
 .  .  .  Q
 .  Q  .  .
 .  .  Q  .
 Q  .  .  .

None
------------------------------
Showing first 16 permutations with their costs:

Case 1: (0, 1, 2, 3) | Diagonal Conflicts Cost: 6
 Q  .  .  .
 .  Q  .  .
 .  .  Q  .
 .  .  .  Q

None
------------------------------
Case 2: (0, 1, 3, 2) | Diagonal Conflicts Cost: 2
 Q  .  .  .
 .  Q  .  .
 .  .  .  Q
 .  .  Q  .

None
------------------------------
Case 3: (0, 2, 1, 3) | Diagonal Conflicts Cost: 2
 Q  .  .  .
 .  .  Q  .
 .  Q  .  .
 .  .  .  Q

None
------------------------------
Case 4: (0, 2, 3, 1) | Diagonal Conflicts Cost: 1
 Q  .  .  .
 .  .  .  Q
 .  Q  .  .
 .  .  Q  .

None
------------------------------
Case 5: (0, 3, 1, 2) | Diagonal Conflicts Cost: 1
 Q  .  .  .
 .  .  Q  .
 .  .  .  Q
 .  Q  .  .

None
```

Program5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

Date 15/09/2025
Page No.

Simulated Annealing

Algorithm:

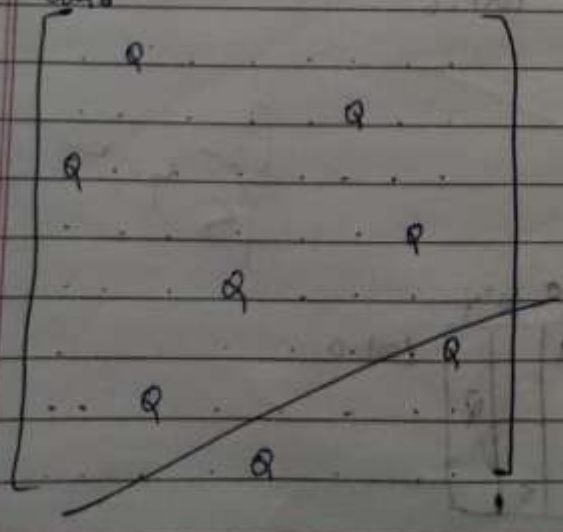
1. current ← initial state
2. T ← a large positive value
3. while T > 0 do
4. next ← a random neighbour of current
5. ΔE ← current.cost() - next.cost()
6. if ΔE > 0 then
7. current ← next
8. else
9. current ← next with probability $p = e^{\frac{\Delta E}{T}}$
10. end if
11. decrease T
12. end while
13.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

output:

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

No. of non attacking pairs: 28

Board



15/9/25

Code:

```
import random
import math

def cost(state):
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

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

def simulated_annealing(n=8, max_iter=10000):
    current = list(range(n))
    random.shuffle(current)
    current_cost = cost(current)

    temperature = 100.0
    cooling_rate = 0.95

    best = current[:]
    best_cost = current_cost

    for _ in range(max_iter):
        if temperature <= 0 or best_cost == 0:
            break

        neighbor = get_neighbor(current)
        neighbor_cost = cost(neighbor)
        delta = current_cost - neighbor_cost

        if delta > 0:
            current, current_cost = neighbor, neighbor_cost
            if neighbor_cost < best_cost:
                best, best_cost = neighbor, neighbor_cost
        else:
            probability = math.exp(delta / temperature)
            if random.random() < probability:
                current, current_cost = neighbor, neighbor_cost
```

```
        temperature *= cooling_rate

    return best, best_cost

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

if __name__ == "__main__":
    n = 8
    solution, cost_val = simulated_annealing(n)

    print("Best position found:", solution)
    print(f"Number of non-attacking pairs: {n*(n-1)//2 - cost_val}")
    print("\nBoard:")
    print_board(solution)
```

Output:

```
Best position found: [3, 1, 7, 5, 0, 2, 4, 6]
Number of non-attacking pairs: 28

Board:
 .  .  .  .  Q  .  .  .
 .  Q  .  .  .  .  .  .
 .  .  .  .  .  Q  .  .
 Q  .  .  .  .  .  .  .
 .  .  .  .  .  .  Q  .
 .  .  .  Q  .  .  .  .
 .  .  .  .  .  .  .  Q
 .  .  Q  .  .  .  .  .
```

Program6

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

Algorithm:

Lab - 6

Propositional Logic

→ Truth table for Connectives

P	Q	¬P	P∧Q	P∨Q	P⇔Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

→ Propositional Inference: Enumeration Method

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

Checking that $KB \models \alpha$

A	B	C	A∨C	B∨¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	true	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	true	false	true	true	true	true
true	true	true	true	true	true	true

- Algorithm:

Step1: Start

Step2: The KB (knowledge base), α (propositional logic), a s
and a equation is given.

Step3: (a) Calculate/Solve the value for both KB and α based on the
cond.

Step4: Check in truth table if both the KB and α values are true.
Step5: If both values are true the o/p is also true
Step6: If both are values KB and α are false or either of them are false then they are ignored.
Step7: Assume a P value where it true once and false once then recure with the rest
Step8: Combine the results with and operation
Step9: If both branches are true then the whole check is
Step10: Stop

Output:

Truth Table

A	B	C	KB	α	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 where KB and 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}

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 (at S = false and T = false)

Code:

```
def pl_true(sentence, model):
    if isinstance(sentence, str):
        return model.get(sentence, False)
    elif isinstance(sentence, tuple):
        operator = sentence[0]
        if operator == 'not':
            return not pl_true(sentence[1], model)
        elif operator == 'and':
            return all(pl_true(arg, model) for arg in sentence[1:])
        elif operator == 'or':
            return any(pl_true(arg, model) for arg in sentence[1:])
        elif operator == '=>':
            return (not pl_true(sentence[1], model)) or pl_true(sentence[2], model)
        elif operator == '<=>':
            return pl_true(sentence[1], model) == pl_true(sentence[2], model)
    return False


def tt_check_all(kb, alpha, symbols, model, true_models, all_symbols, col_widths):
    if not symbols:
        kb_true = pl_true(kb, model)
        alpha_true = pl_true(alpha, model)
        row_values = [str(model.get(s, False)) for s in all_symbols] + [str(kb_true), str(alpha_true),
str(kb_true and alpha_true)]
        formatted_row = "│ " + " │ ".join(f"{val:^{col_widths[i]}}" for i, val in
enumerate(row_values)) + " │"

        if kb_true and alpha_true:
            print(f"\033[92m{formatted_row}\033[0m")  # green for satisfying rows
            true_models.append(model.copy())
        else:
            print(formatted_row)
        return (not kb_true) or alpha_true
    else:
        P = symbols[0]
        rest = symbols[1:]
        model_true = model.copy()
        model_true[P] = True
        model_false = model.copy()
        model_false[P] = False
        return (tt_check_all(kb, alpha, rest, model_true, true_models, all_symbols, col_widths) and
                tt_check_all(kb, alpha, rest, model_false, true_models, all_symbols, col_widths))


def tt_entails(KB, alpha):
    symbols = set()
```

```
    def extract_symbols(sentence):
        if isinstance(sentence, str):
            symbols.add(sentence)
        elif isinstance(sentence, tuple):
            for arg in sentence[1:]:
                extract_symbols(arg)

    extract_symbols(KB)
    extract_symbols(alpha)
    symbols = list(sorted(symbols))  # sorted for consistent order

    header_values = symbols + ["KB", "alpha", "KB ∧ α"]
    col_widths = [max(len(str(s)), 5) for s in header_values]

    total_width = sum(col_widths) + 3 * len(col_widths) + 1
    border = "═" * total_width

    print("\nTruth Table:")
    print(" ┌" + "┬".join("─" * (col_widths[i] + 2) for i in range(len(header_values))) + "┐ ")
    header_string = " │ " + " │ ".join(f"{val:^{col_widths[i]}}" for i, val in enumerate(header_values))
+ " │"
    print(header_string)
    print(" ├" + "┼".join("─" * (col_widths[i] + 2) for i in range(len(header_values))) + "┤ ")

    true_models = []
    result = tt_check_all(KB, alpha, list(symbols), {}, true_models, all_symbols=symbols,
col_widths=col_widths)

    print(" └" + "┴".join("─" * (col_widths[i] + 2) for i in range(len(header_values))) + "┘ ")

    print("\nModels where KB and alpha are true:")
    if true_models:
        for model in true_models:
            print(model)
    else:
        print("None")

    return result


# Example
kb2 = ('and', ('or', 'A', 'B'), ('=>', 'B', 'C'))
alpha2 = ('or', 'A', 'C')
print(f"\nDoes KB entail alpha? {tt_entails(kb2, alpha2)}")
```

Output:

Truth Table:

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 where KB and 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
```

Program7
Implement unification in first order logic

Algorithm:

Lab-7

Unification in FOL

Algorithm:

Step1: 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.

Step2: If the initial predicate symbol in ψ_1 and ψ_2 are not same, then return FAILURE.

Step3: If ψ_1 and ψ_2 have a different no. of arguments, then return FAILURE.

Step4: Set substitution set (SUBST) to NIL.

Step5: For i=1 to the no. of elements

a. Call unify fn with the ith element of ψ_1 and ith element of ψ_2, and put the result into S.

b. If S = ~~FAILURE~~ failure then returns 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).

Step6: Return SUBST

Output:

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

Solve the following:

Find Most General Unifier (MGU) of {p(b, X, f(g(Z)) and

$P(z, f(y), f(y))$ }

⇒ $P(b, x, f(g(z)))$ } both have same number and predicate are same
$P(z, f(y), f(y))$

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

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

Unifies = $\{b/z, x/f(y), y/g(z)\}$

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

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

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

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)
g(y) = x ⇒ f(a) = g(y)
It is not possible
∴ No Unifies

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

prime(11) } both have same pricate and same
prime(y) } no of 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)
~~knows~~ x = mother(John)
∴ knows(John, mother(John)) =
knows(John, mother(John))

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

knows(John, x) Let y = John
knows(y, Bill) x = Bill
∴ knows(John, Bill) =
knows(John, Bill

Code:

```
print('Spurthi Reddy P  1BM23CS338')
def unify(x, y, subst=None):
    if subst is None:
        subst = {}

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

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


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


def is_constant(x):
    return isinstance(x, str) and x.isupper() and x.isalpha()


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


def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
```

```
        return None
    else:
        subst[var] = x
        return subst


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


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

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

Output:
Spurthi Reddy P 1BM23CS338
Unification succeeded with substitution: {'x': 'B', 'y': 'A'}

Program8
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:

Lab - 8

Create a Knowledge Based consisting of First order logic statements and prove the given query using forward reasoning.

Eg: Primitive Conclusions

$P \Rightarrow Q$

$L \wedge M \Rightarrow P$] Rules

$B \wedge L \Rightarrow M$

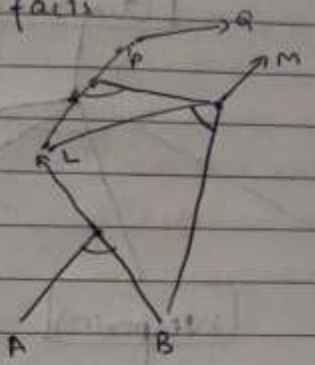
$A \wedge P \Rightarrow L$

$A \wedge B \Rightarrow L$

A] facts

B

Prove Q



Eg1: 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".

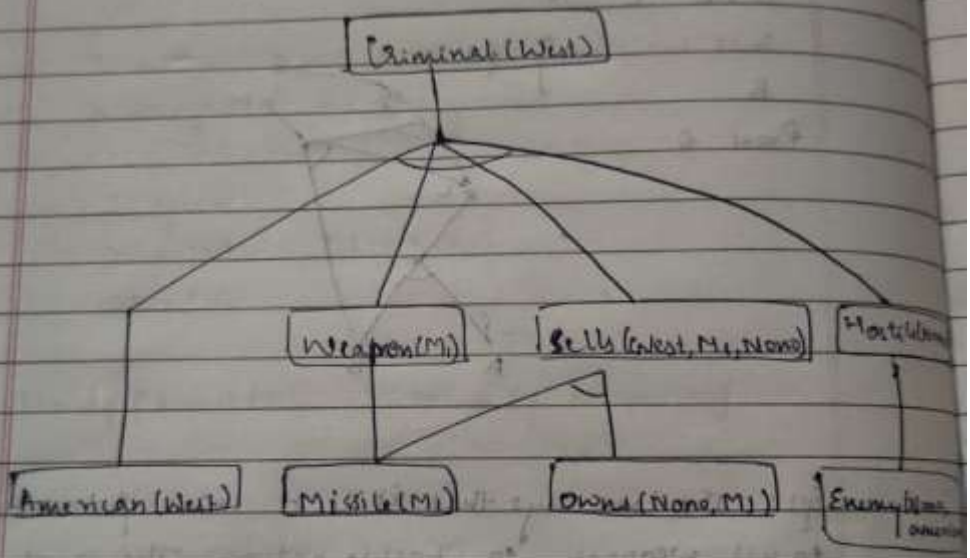
P.T "West is criminal"

Forward Chaining:

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

2. ∀ x Missile(x) ∧ Owns(Nono, x) ⇒ Sells(West, x, Nono)
3. ∀ x Enemy(x, America) ⇒ Hostile(x)
4. ∀ x Missile(x) ⇒ Weapon(x)
5. American(West)
6. Enemy(Nono, America)
7. Owns(Nono, M1) and
8. Missile(M1)

Prove suleit using (3, 6, 2, 8) Facts and (3, 6, 9) Rules.



Forward Reasoning 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 vars: new, the new sentences inferred on each iter
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, 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:

New fact inferred : Criminal (West)

New fact inferred : SoldWeapons (West, Nono)

Final facts :

American (West)

Hostile (Nono)

Missiles (Nono)

Criminal (West)

SoldWeapons (West, Nono)

13/10/25

Code:

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

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


def forward_chaining(facts, rules):
   new_facts = facts.copy()
   inferred = True
   while inferred:
      inferred = False
      for rule in rules:
         result = rule(new_facts)
         if result and result not in new_facts:-
            new_facts[result] = True
            inferred = True
            print(f"New fact inferred: {result}")
   return new_facts
rules = [rule1, rule2]

inferred_facts = forward_chaining(facts, rules)

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

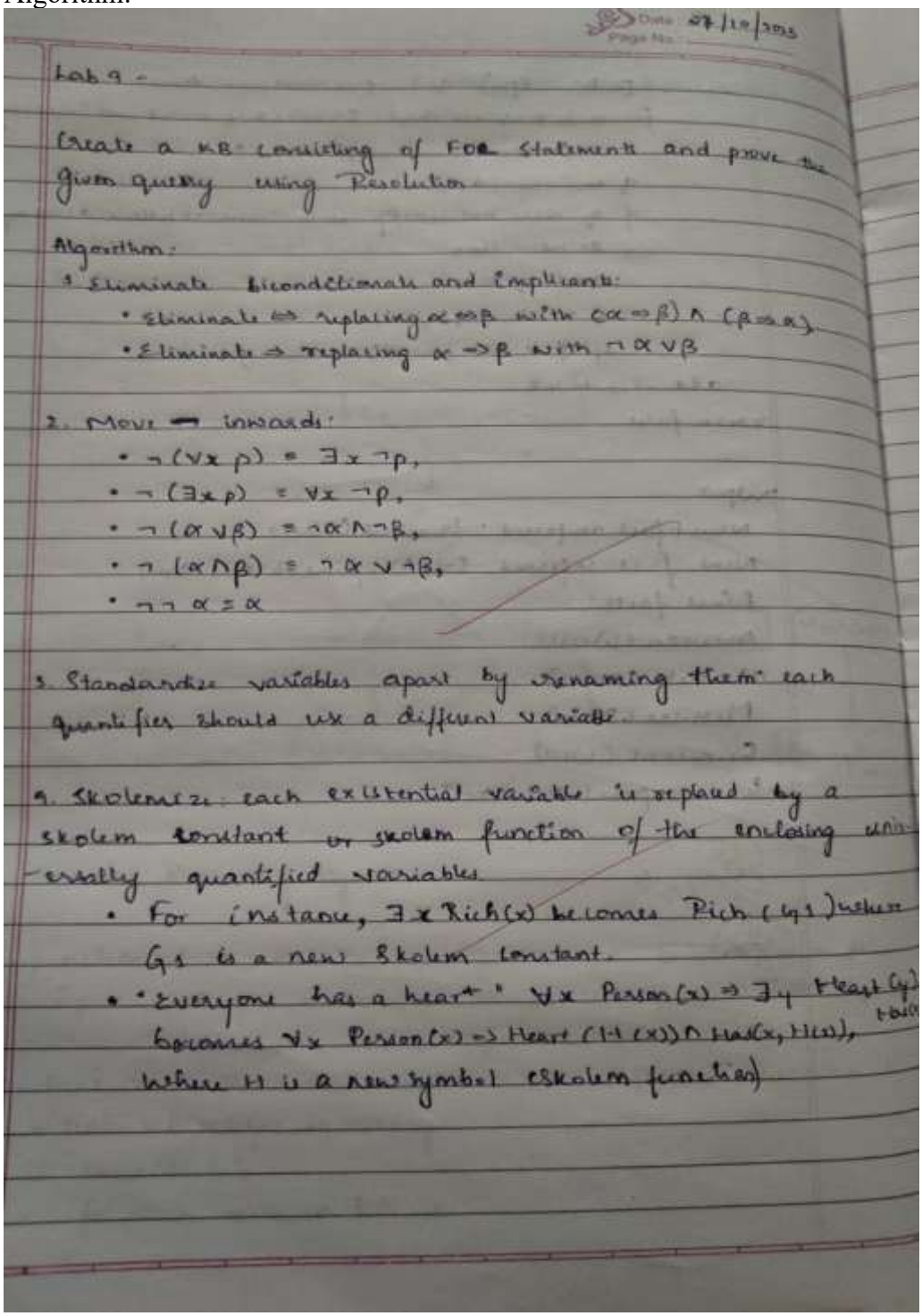
Output:

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

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

Program9

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

Algorithm:



Lab 9 -

Create a KB consisting of FOL statements and prove the given query using Resolution.

Algorithm:

1. Eliminate biconditionals and implications:
   - Eliminate ⇔ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
   - Eliminate ⇒ replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$

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

3. Standardize variables apart by renaming them: each quantifier should use a different variable.

4. Skolemize: each existential variable is replaced by a skolem constant or skolem function of the enclosing universally quantified variables
   - For instance, $\exists x$ Rich(x) becomes Rich(G1) where G1 is a new Skolem constant.
   - "Everyone has a heart" $\forall x$ Person(x) ⇒ $\exists y$ Heart(y) ∧ Has(x, y) becomes $\forall x$ Person(x) ⇒ Heart(H(x)) ∧ Has(x, H(x)), where H is a new symbol (Skolem function)

Pseudocode:

Proof by resolution

Given KB on 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

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) $\forall x \neg food(x) \vee Likes(John, x)$

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)

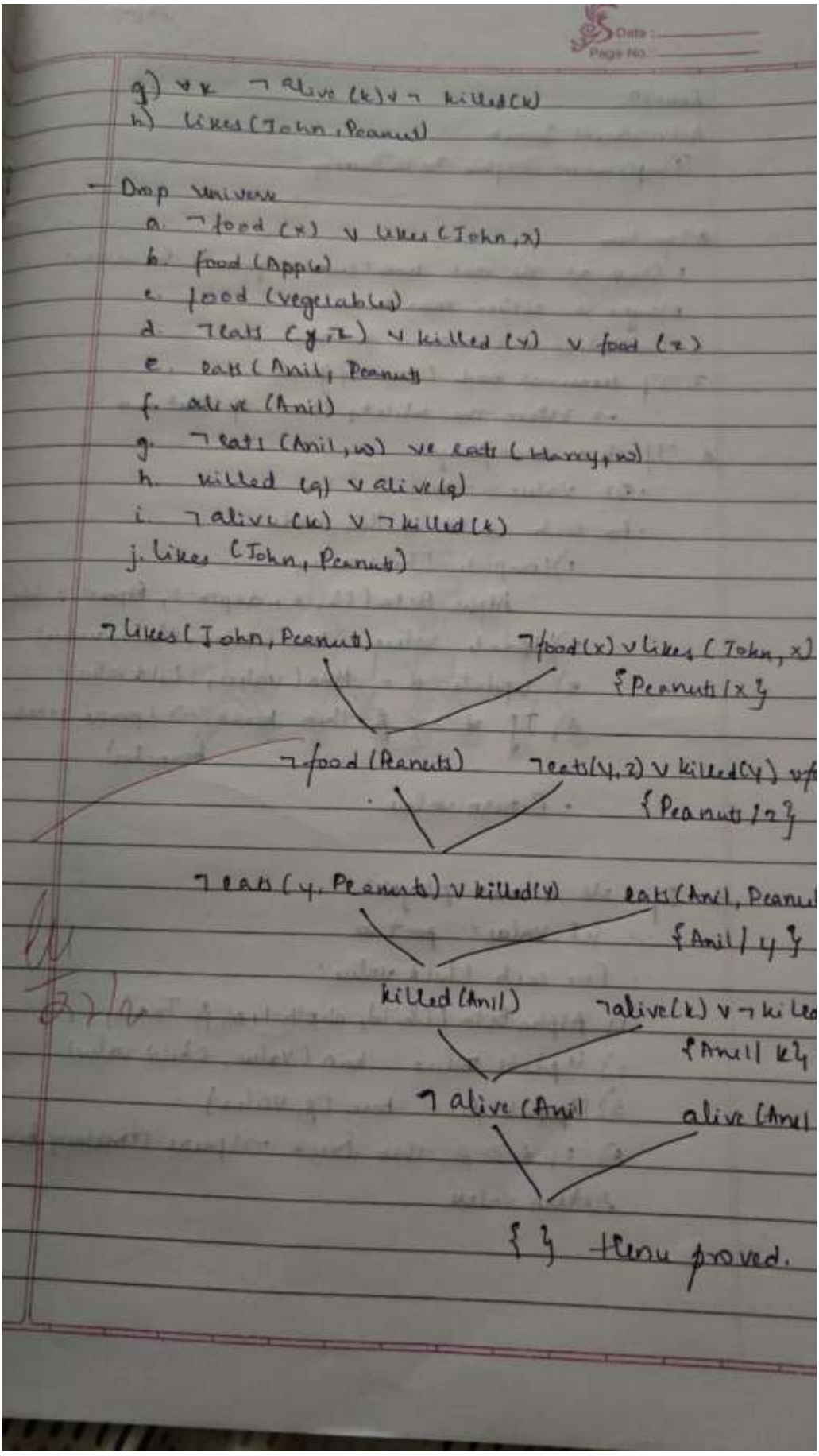
- Move negation ($\neg$) inwards and rewrite
a) $\forall x \neg food(x) \vee likes(John, x)$
b) food (Apple) $\wedge$ food (vegetables)
c) $\forall x \forall y \neg eats(x, y) \vee killed(x) \vee food(y)$
d) eats (Anil, Peanuts) $\wedge$ alive (x)
e) $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
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) $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f) $\forall x [\neg killed(x)] \vee alive(x)$
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) ∀x ¬ alive(k) ∨ ¬ killed(k)
h) likes(John, Peanuts)

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



Code:

```
def negate_literal(lit):
    return lit[1:] if lit.startswith("~") else "~" + lit

def resolve(ci, cj):
    ci = set(ci)
    cj = set(cj)
    resolvents = []

    for di in ci:
        ndi = negate_literal(di)
        if ndi in cj:
            new_clause = (ci - {di}) | (cj - {ndi})
            if len(new_clause) == 0:
                resolvents.append({"res": set(), "pair": (ci, cj)})
            else:
                resolvents.append({"res": new_clause, "pair": (ci, cj)})
    return resolvents


# CLAUSES (CNF)
clauses = [
    {"~Food(x)", "Likes(John,x)"},
    {"~Eats(x,y)", "~Killed(y)", "Food(y)"},
    {"Eats(Anil,Peanut)"},
    {"Alive(Anil)"},
    {"~Alive(z)", "~Killed(z)"},
    {"~Likes(John,Peanut)"}   # Negated Query
]

# Resolution Steps Storage
steps = []

changed = True
while changed:
    changed = False
    new = []

    for i in range(len(clauses)):
        for j in range(i+1, len(clauses)):
            ci = clauses[i]
            cj = clauses[j]
            resolvents = resolve(ci, cj)

            for r in resolvents:
                res = r["res"]
                if res == set(): # NIL Found
```

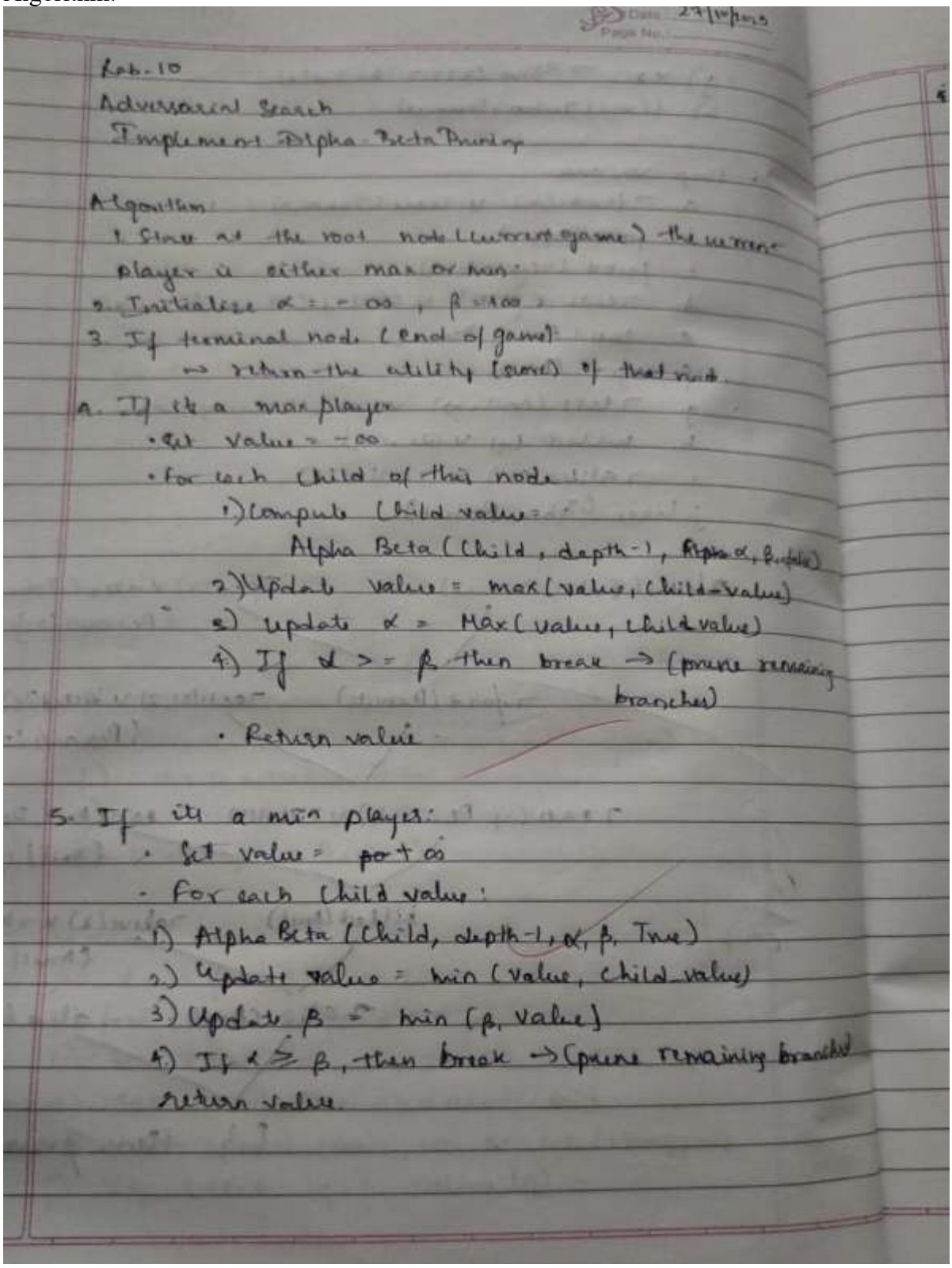
```
                steps.append((ci, cj, set()))
                print("\n☑ NIL (Contradiction Found)")
                print("=> Query Proven TRUE")
                changed = False
                break

            if res not in clauses and res not in new:
                new.append(res)
                steps.append((ci, cj, res))
                changed = True
        if changed is False and resolvents and res == set():
            break

    for c in new:
        clauses.append(c)


print("      RESOLUTION STEPS")

for (a, b, c) in steps:
    if c == set():
        print(f"{a}  +  {b}  =>  NIL")
    else:
        print(f"{a}  +  {b}  =>  {c}")

# PRINT CLAUSES AFTER DERIVATION
print("\nRemaining Clauses:")
for c in clauses:
    print(c)
```

Output:

```
FOL resolution, 1BM23CS338: SPURTHI REDDY P

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

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

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

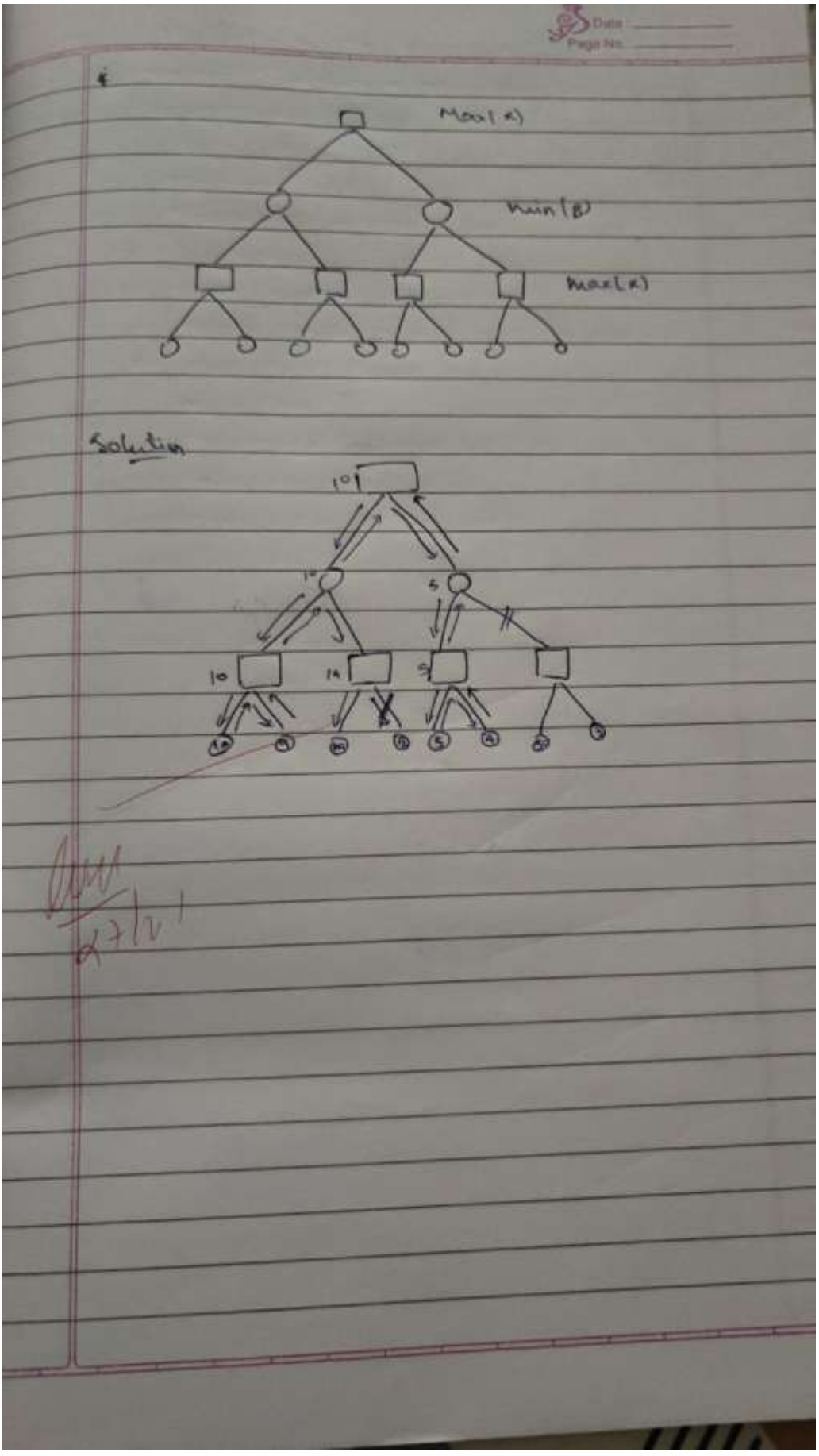
Program10
Implement Alpha-Beta Pruning.

Algorithm:


Lab. 10
Adversarial Search
Implement Alpha Beta Pruning

Algorithm:
1. Since at the root node (current game) the maximum 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 it a max player
   • Set value = $-\infty$
   • For each child of this node:
      1) Compute Child value:
         Alpha Beta (Child, depth-1, Alpha α, β, False)
      2) Update value = max(value, Child-value)
      3) update α = Max(value, Child value)
      4) If $\alpha >= \beta$ 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)
      4) If $\alpha \geq \beta$, then break → (prune remaining branches)
   return value.


Max(A)
Min(B)
Max(A)
Solution
10
10
5
10
10
5

Code:

```
def alpha_beta(node_index, depth, max_depth, alpha, beta, is_max, values, explored, pruned, path):


    # Number of leaves = 2^max_depth
    total_leaves = len(values)

    # If we reached a leaf node, return its value
    if depth == max_depth:
        leaf_index = node_index - (2 ** max_depth - 1)
        if 0 <= leaf_index < total_leaves:
            val = values[leaf_index]
            explored.append((list(path), val))
            return val
        else:
            return 0  # Safety fallback

    if is_max:
        value = float('-inf')
        for i in range(2):  # left & right children
            child_index = node_index * 2 + i + 1
            path.append(child_index)
            value = max(value, alpha_beta(child_index, depth + 1, max_depth,
                                          alpha, beta, False, values, explored, pruned, path))
            path.pop()
            alpha = max(alpha, value)
            if beta <= alpha:
                pruned.append((node_index, child_index, 'Beta cutoff'))
                break
        return value

    else:
        value = float('inf')
        for i in range(2):
            child_index = node_index * 2 + i + 1
            path.append(child_index)
            value = min(value, alpha_beta(child_index, depth + 1, max_depth,
                                          alpha, beta, True, values, explored, pruned, path))
            path.pop()
            beta = min(beta, value)
            if beta <= alpha:
                pruned.append((node_index, child_index, 'Alpha cutoff'))
                break
        return value


if __name__ == "__main__":
```

```
values = [3, 5, 6, 9, 1, 2, 0, -1]  # leaf node values
max_depth = 3                       # since 2^3 = 8 leaves
explored, pruned = [], []
print("Spurthi Reddy P : 1BM23CS338")
print("Leaf node values:", values)

result = alpha_beta(0, 0, max_depth, float('-inf'), float('inf'),
                    True, values, explored, pruned, [0])

print("\nValue of root node (MAX mode):", result)
print("\nExplored leaf paths:")
for p, val in explored:
    print(f"Path {p} -> Value {val}")

print("\nPruned branches:")
for item in pruned:
    print(item)
```

Output:

```
Spurthi Reddy P : 1BM23CS338
Leaf node values: [3, 5, 6, 9, 1, 2, 0, -1]

Value of root node (MAX mode): 5

Explored leaf paths:
Path [0, 1, 3, 7] -> Value 3
Path [0, 1, 3, 8] -> Value 5
Path [0, 1, 4, 9] -> Value 6
Path [0, 2, 5, 11] -> Value 1
Path [0, 2, 5, 12] -> Value 2

Pruned branches:
(4, 9, 'Beta cutoff')
(2, 5, 'Alpha cutoff')
```