VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sneha S Bhairappa (1BM23CS333)

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)

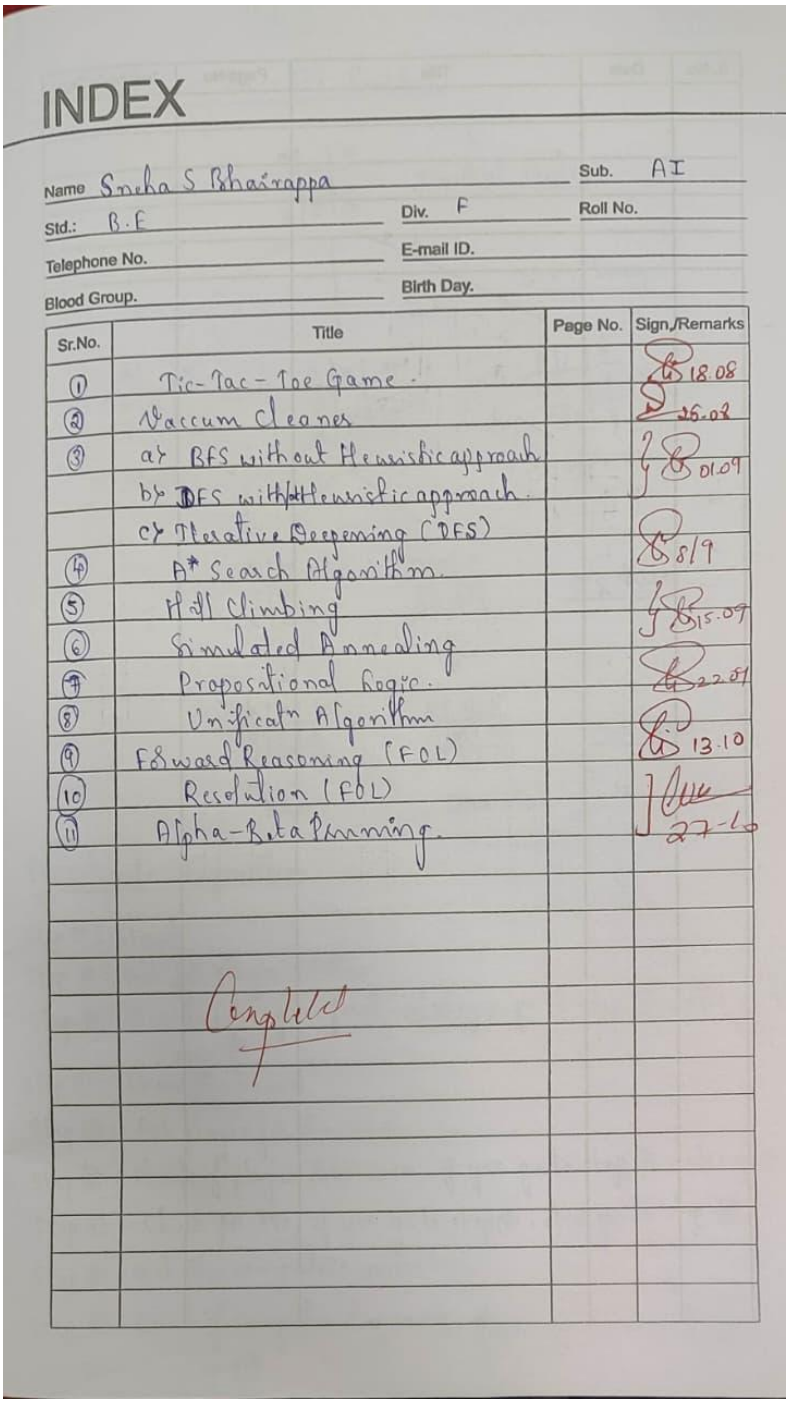
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Sneha S Bhairappa (1BM23CS333),** 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.

Surabhi S Assistant Professor Department of CSE, BMSCE	**Dr. Kavitha Sooda** Professor & HOD Department of CSE, BMSCE

Index



INDEX

Name Sneha S Bhairappa Sub. AI

Std.: B.E Div. F Roll No.

Telephone No. E-mail ID.

Blood Group. Birth Day.

Sr.No.	Title	Page No.	Sign./Remarks
1	Tic-Tac-Toe Game.		18.08
2	Vaccum cleaner		25.08
3	a) BFS without Heuristic approach		01.09
	b) DFS with Heuristic approach.		
	c) Iterative Deepening (DFS)		
4	A* Search Algorithm.		8/9
5	Hill Climbing		15.09
6	Simulated Annealing		22.09
7	Propositional Logic.		
8	Unification Algorithm		13.10
9	Forward Reasoning (FOL)		
10	Resolution (FOL)		27-10
11	Alpha-Beta Pruning.		

Completed

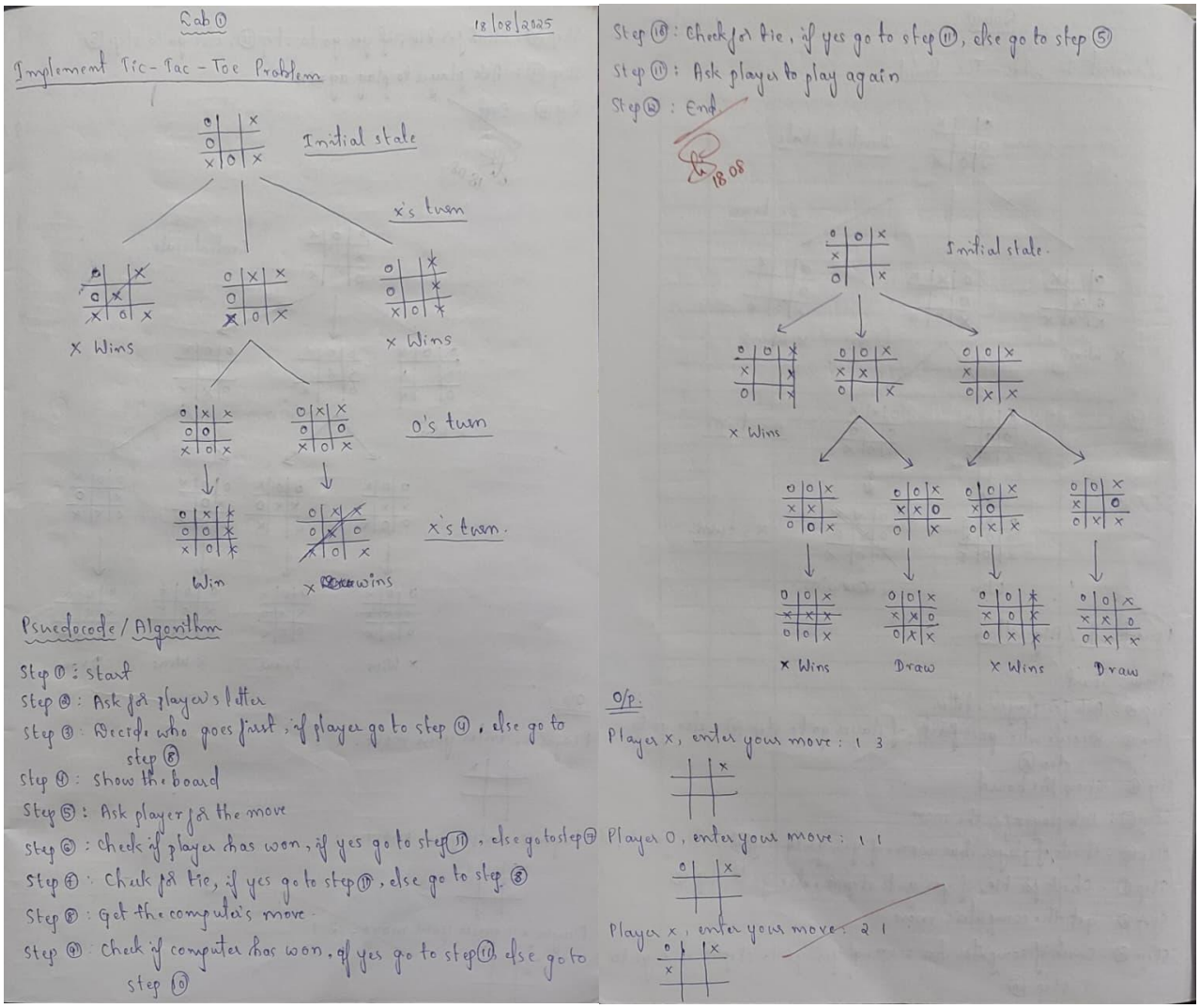
Github Link:
https://github.com/1BM23CS333/AI-LAB.git

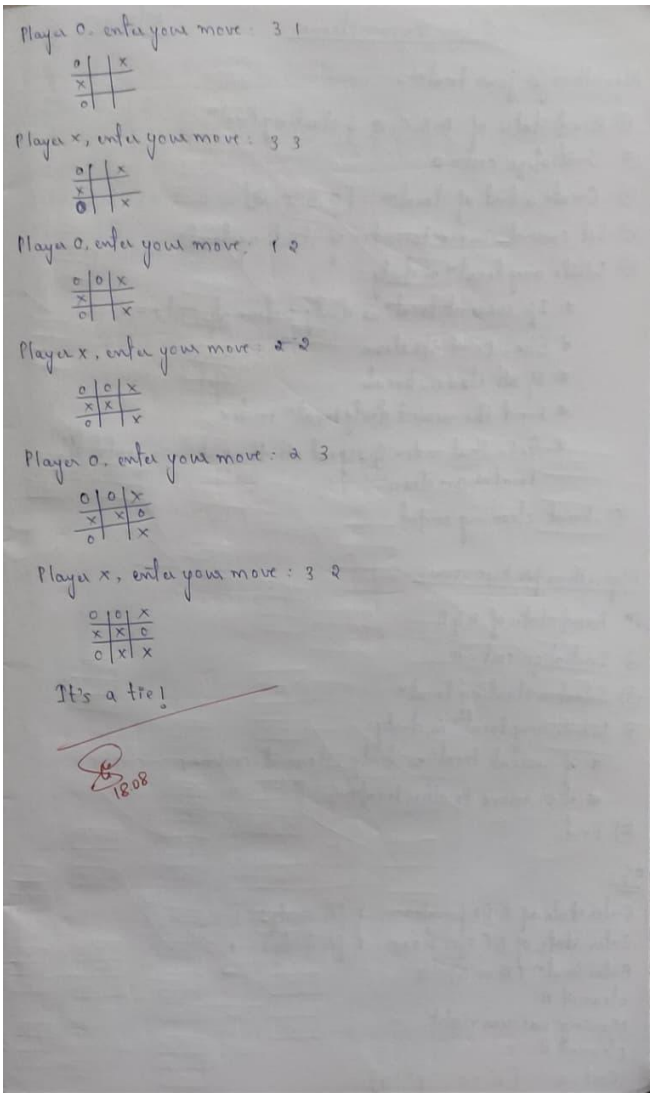
Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

1.TIC TAC TOE GAME

Algorithm:





Code:

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


def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] != " ":
            return True
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return True
```

```
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return True
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return True
    return False


def is_full(board):
    for row in board:
        if " " in row:
            return False
    return True


def tic_tac_toe():
    print("Welcome to Tic Tac Toe!")
    board = [[" " for _ in range(3)] for _ in range(3)]
    print_board(board)
    current_player = "X"
    while True:
        try:
            row, col = map(int, input(f"Player {current_player}, enter your move (row and column: 1 1
for top-left): ").split())
            if row < 1 or row > 3 or col < 1 or col > 3:
                print("Invalid position! Enter numbers between 1 and 3.")
                continue
            if board[row - 1][col - 1] != " ":
                print("Cell already taken! Try again.")
                continue
            board[row - 1][col - 1] = current_player
            print_board(board)
            if check_winner(board):
                print(f"🎉 Player {current_player} wins!")
                break
            if is_full(board):
                print("It's a tie!")
                break
            current_player = "O" if current_player == "X" else "X"
        except ValueError:
            print("Invalid input! Enter row and column numbers separated by a space.")
```

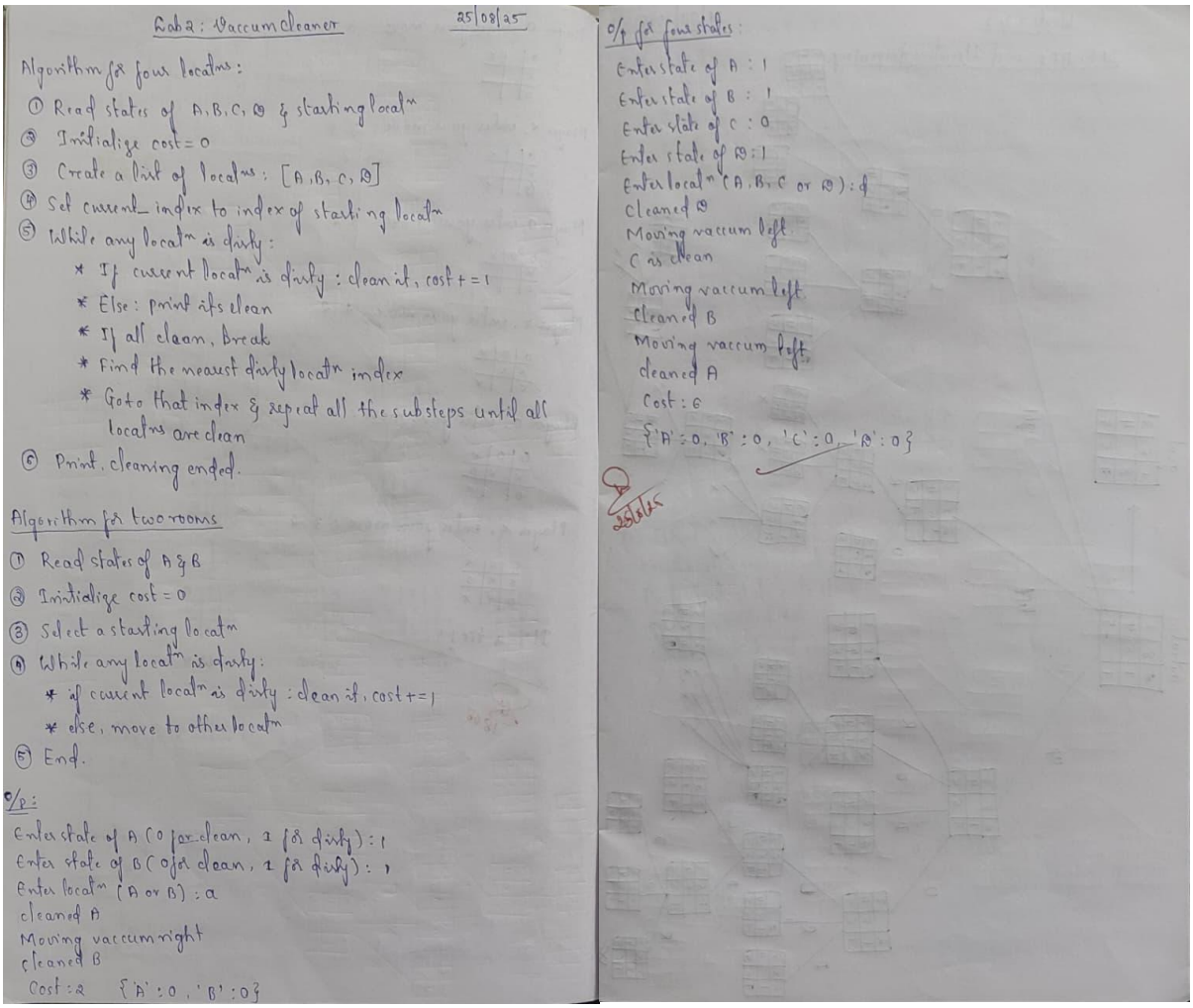
tic_tac_toe()

Output:

```
---------
O |   | X
---------
Player O, enter your move (row and column: 1 1 for top-left): 1 2
O | O | X
---------
X |   |
---------
O |   | X
---------
Player X, enter your move (row and column: 1 1 for top-left): 2 2
O | O | X
---------
X | X |
---------
O |   | X
---------
Player O, enter your move (row and column: 1 1 for top-left): 2 3
O | O | X
---------
X | X | O
---------
O |   | X
---------
Player X, enter your move (row and column: 1 1 for top-left): 3 2
O | O | X
---------
X | X | O
---------
O | X | X
---------
It's a tie! 🤝
```

2.VACCUM CLEANER

Algorithm:



Lab 2: Vaccum cleaner 25/08/25

Algorithm for four locations:

① Read states of A, B, C, D & starting location
② Initialize cost = 0
③ Create a list of locations: [A, B, C, D]
④ Set current_index to index of starting location
⑤ While any location is dirty:
* If current location is dirty: clean it, cost += 1
* Else: print it's clean
* If all clean, Break
* Find the nearest dirty location index
* Go to that index & repeat all the substeps until all locations are clean
⑥ Print, cleaning ended.

Algorithm for two rooms

① Read states of A & B
② Initialize cost = 0
③ Select a starting location
④ While any location is dirty:
* if current location is dirty: clean it, cost += 1
* else, move to other location
⑤ End.

O/p:

Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): a
cleaned A
Moving vaccum right
cleaned B
Cost: 2 {'A': 0, 'B': 0}

O/p for four states:

Enter state of A: 1
Enter state of B: 1
Enter state of C: 0
Enter state of D: 1
Enter location (A, B, C or D): d
cleaned D
Moving vaccum left
C is clean
Moving vaccum left
cleaned B
Moving vaccum left
cleaned A
Cost: 6
{'A': 0, 'B': 0, 'C': 0, 'D': 0}

25 states

Code:

```
def vacuum_world():
    print("1BM23CS333")
    state = {
        'A': int(input("Enter state of A (0 for clean, 1 for dirty): ")),
        'B': int(input("Enter state of B (0 for clean, 1 for dirty): "))
    }
    location = input("Enter location (A or B): ").strip().upper()
    cost = 0

    if location == 'A':
        if state['A'] == 1:
```

```
            print("Cleaned A.")
            state['A'] = 0
            cost += 1
        else:
            print("A is clean")
        if state['B'] == 1:
            print("Moving vacuum right")
            cost += 1
            print("Cleaned B.")
            state['B'] = 0
        else:
            print("Moving vacuum right")

    elif location == 'B':
        if state['B'] == 1:
            print("Cleaned B.")
            state['B'] = 0
            cost += 1
        else:
            print("B is clean")
        if state['A'] == 1:
            print("Moving vacuum left")
            cost += 1
            print("Cleaned A.")
            state['A'] = 0
        else:
            print("Moving vacuum left")

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


vacuum_world()
```

Output:

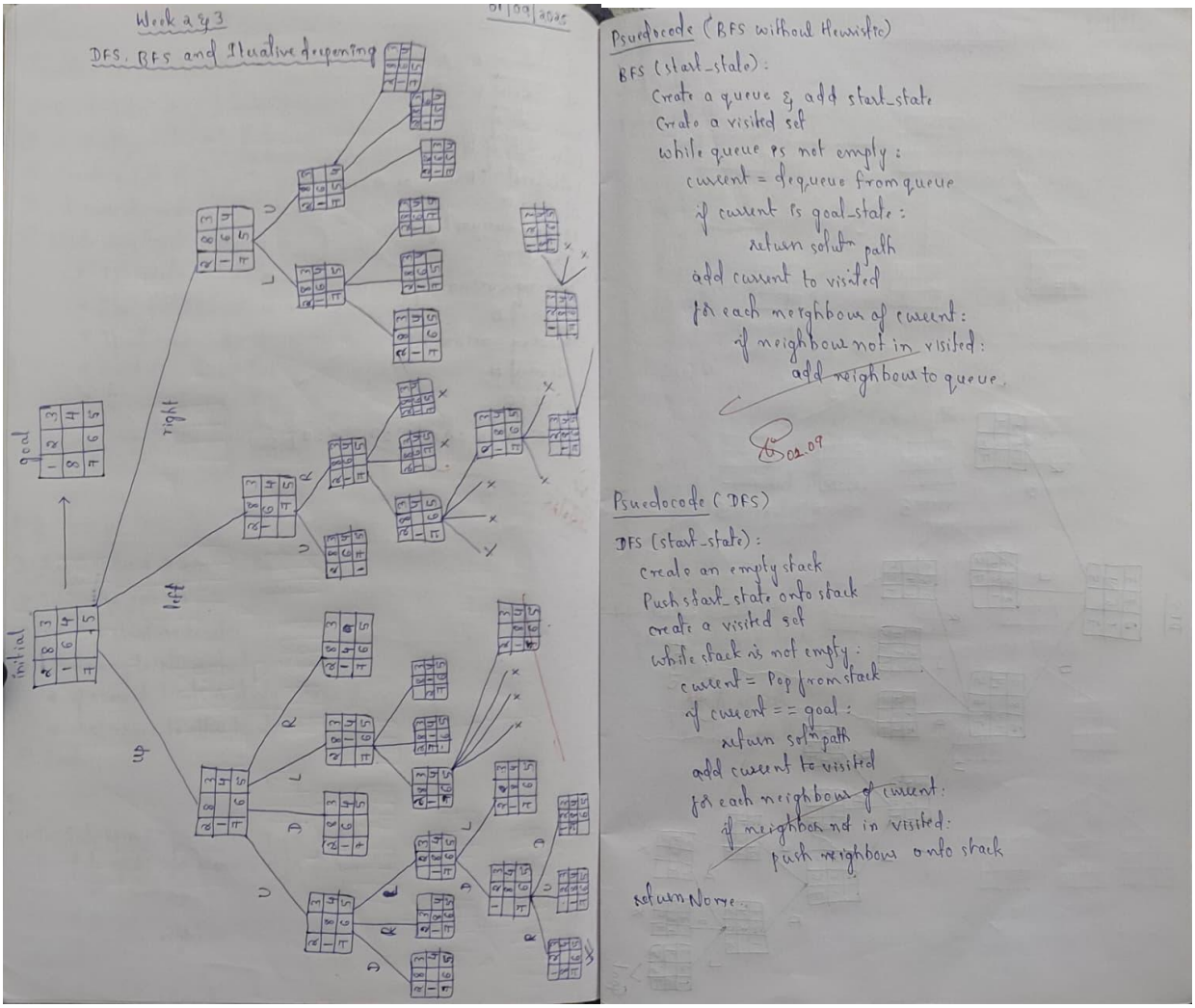
```
1BM23CS333
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): a
Cleaned A.
Moving vacuum right
Cleaned B.
Cost: 2
{'A': 0, 'B': 0}
```

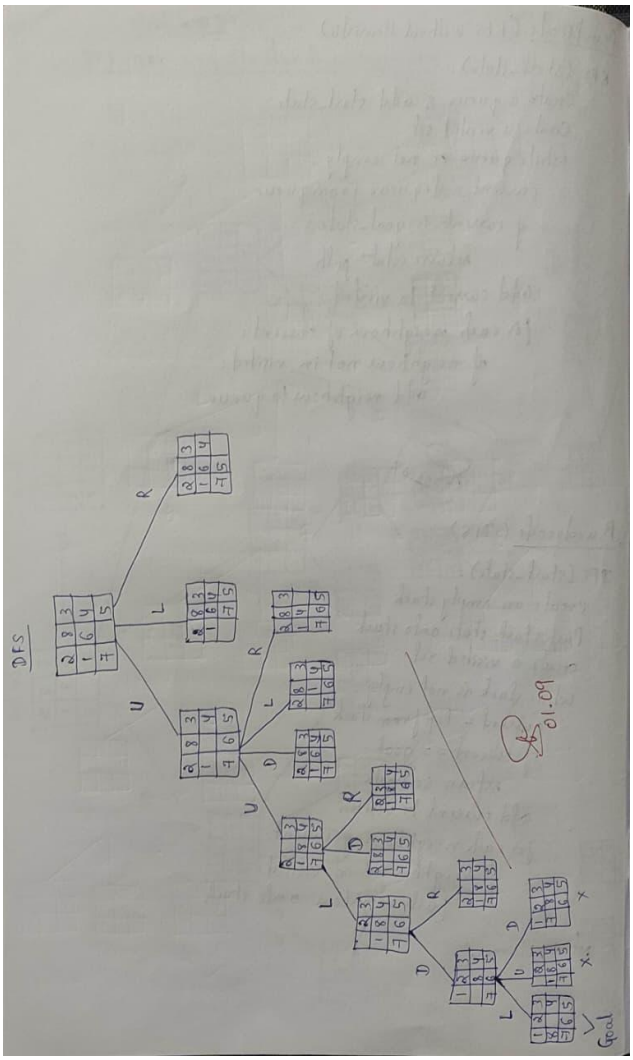
Program 2

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

1.8 PUZZLE USING DFS

Algorithm:





Code:

```
def get_neighbors_dfs(state):
    neighbors = []
    index = state.index('0')
    row, col = divmod(index, 3)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for r, c in moves:
        new_row, new_col = row + r, col + c
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_index = new_row * 3 + new_col
            new_state = list(state)
            new_state[index], new_state[new_index] = new_state[new_index], new_state[index]
            neighbors.append("".join(new_state))
```

```
    return neighbors


def dfs(start_state, goal_state):
    stack = [start_state]
    visited = set()
    parent = {start_state: None}

    while stack:
        current = stack.pop()
        if current == goal_state:
            path = []
            while current:
                path.append(current)
                current = parent[current]
            return path[::-1]
        if current not in visited:
            visited.add(current)
            neighbors = get_neighbors_dfs(current)
            neighbors.reverse()
            for neighbor in neighbors:
                if neighbor not in visited:
                    parent[neighbor] = current
                    stack.append(neighbor)
    return None


print("1BM23CS333")
print("Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):")
initial_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    initial_state_rows.extend(row)
initial_state = "".join(initial_state_rows)

print("\nEnter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):")
goal_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    goal_state_rows.extend(row)
goal_state = "".join(goal_state_rows)
```

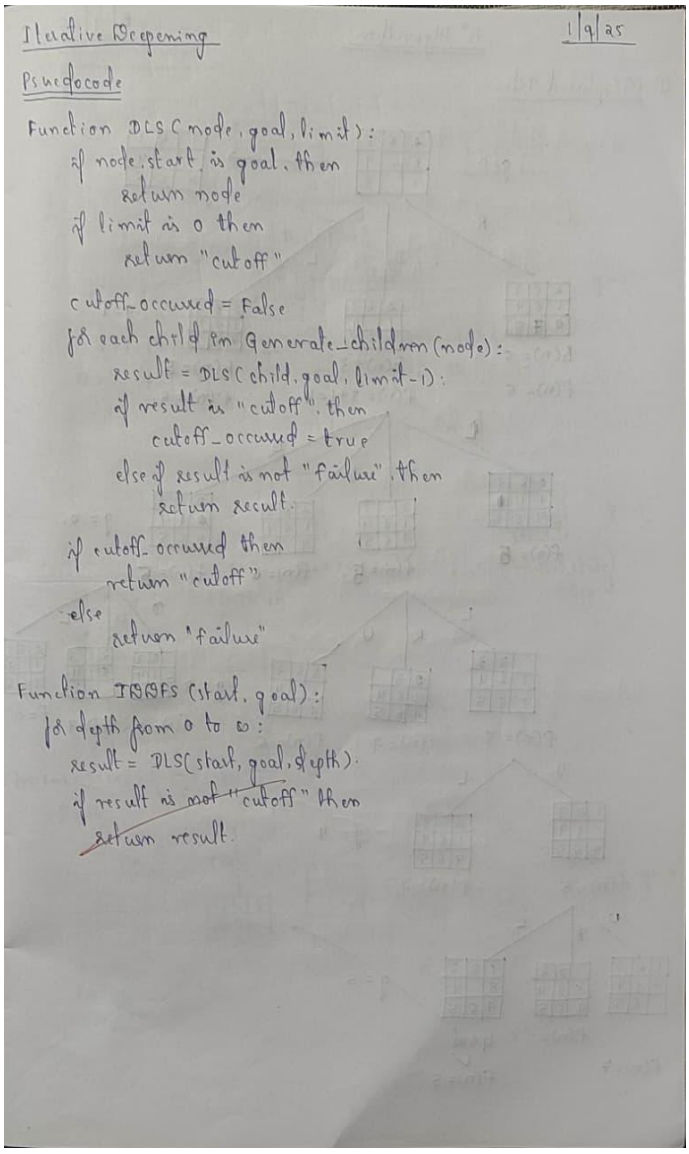
```
solution = dfs(initial_state, goal_state)

if solution:
    print("\nDFS solution path:")
    for s in solution:
        print(s[:3])
        print(s[3:6])
        print(s[6:])
        print()
else:
    print("\nNo solution found.")
```

Output:

```
Streaming output truncated to the last 5000 lines.
164
320
785

160
324
785

106
324
785

126
304
785

126
384
705

126
384
075

126
084
375

026
184
375

206
184
375

286
104
375

286
174
305
```

2.ITERATIVE DEEPENING

Algorithm:


Iterative Deepening 1/9/25

Pseudocode

```
Function DLS(node, goal, limit):
    if node.start is goal. then
        return node
    if limit is 0 then
        return "cutoff"
    cutoff_occured = False
    for each child in Generate_children(node):
        result = DLS(child, goal, limit-1):
        if result is "cutoff". then
            cutoff_occured = true
        else if result is not "failure". then
            return result.
    if cutoff_occured then
        return "cutoff"
    else
        return "failure"
Function IDDFS (start, goal):
    for depth from 0 to ∞:
        result = DLS(start, goal, depth).
        if result is not "cutoff" then
        return result.
```

Code:

```
def get_neighbors(state):
    neighbors = []
    idx = state.index("0")
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    x, y = divmod(idx, 3)

    for dx, dy in moves:
        nx, ny = x + dx, y + dy
```

```
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_idx = nx * 3 + ny
            state_list = list(state)
            state_list[idx], state_list[new_idx] = state_list[new_idx], state_list[idx]
            neighbors.append("".join(state_list))
    return neighbors


def dfs_limit(start_state, goal_state, limit):
    stack = [(start_state, 0)]
    visited = set()
    parent = {start_state: None}
    path = []

    while stack:
        current_state, depth = stack.pop()
        if current_state == goal_state:
            while current_state:
                path.append(current_state)
                current_state = parent[current_state]
            return path[::-1]
        if depth < limit and current_state not in visited:
            visited.add(current_state)
            neighbors = get_neighbors(current_state)
            neighbors.reverse()
            for neighbor in neighbors:
                if neighbor not in visited:
                    parent[neighbor] = current_state
                    stack.append((neighbor, depth + 1))
    return None


def iddfs(start_state, goal_state, max_depth):
    for limit in range(max_depth + 1):
        print(f"Searching with depth limit: {limit}")
        solution = dfs_limit(start_state, goal_state, limit)
        if solution:
            return solution
    return None
```

```
print("1BM23CS333")
print("Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):")
initial_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    initial_state_rows.extend(row)
initial_state = "".join(initial_state_rows)

print("\nEnter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):")
goal_state_rows = []
for i in range(3):
    row = input(f"Row {i+1}: ").split()
    goal_state_rows.extend(row)
goal_state = "".join(goal_state_rows)

max_depth = 50
solution = iddfs(initial_state, goal_state, max_depth)

if solution:
    print("\nIDDFS solution path:")
    for s in solution:
        print(s[:3])
        print(s[3:6])
```

Output:

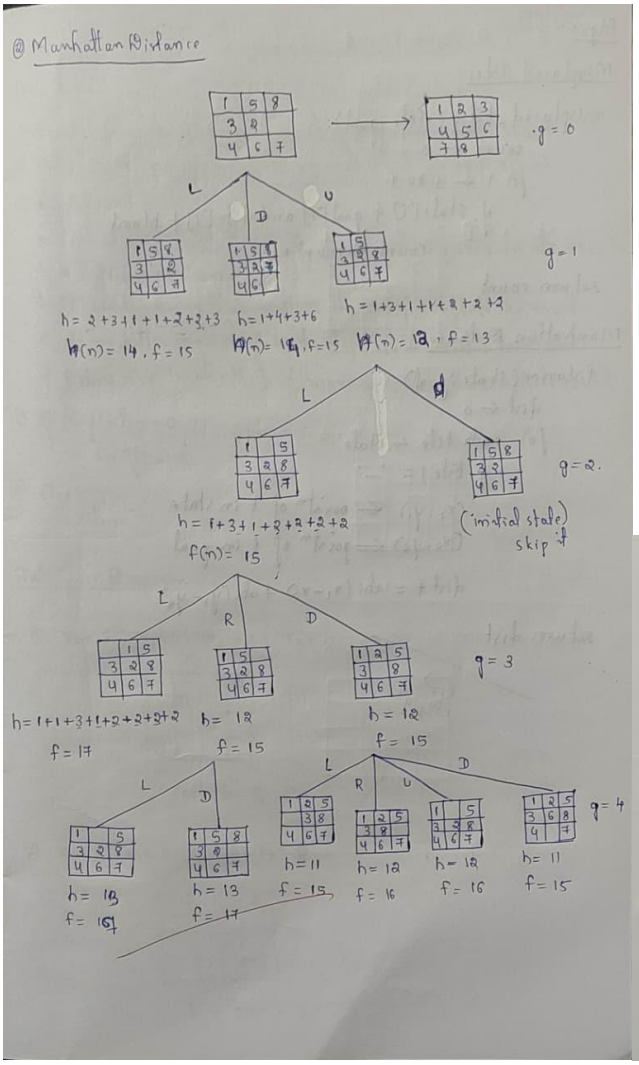
```
1BM23CS333
Enter the initial state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5

Enter the goal state (enter 3 digits per row, separated by spaces, 0 for empty):
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5
Searching with depth limit: 0
Searching with depth limit: 1
Searching with depth limit: 2
Searching with depth limit: 3
Searching with depth limit: 4
Searching with depth limit: 5

IDDFS solution path:
283
164
705

283
104
765

203
184
765

023
184
765

123
084
765

123
804
765
```

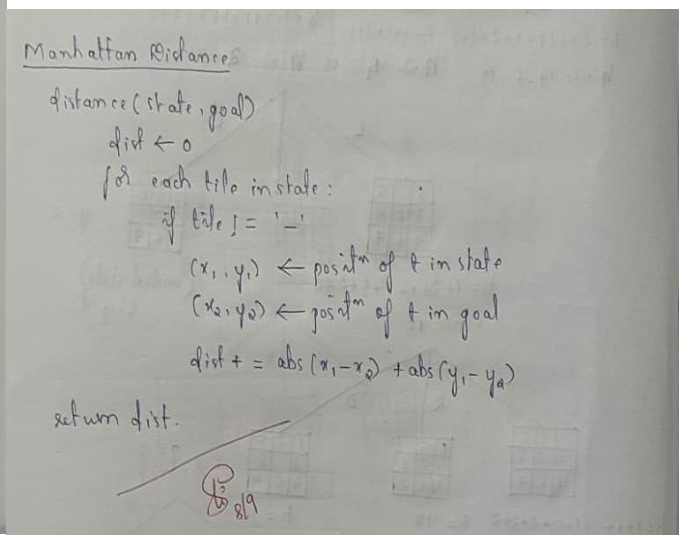
Program 3

Implement A* search algorithm

1.A* SEARCH USING MANNHATTAN DISTANCE METHOD

Algorithm:





Code:

```
import heapq

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
```

```
        self.board = board
        self.moves = moves
        self.previous = previous
        self.size = 3

    def __eq__(self, other):
        return self.board == other.board

    def __hash__(self):
        return hash(self.board)

    def get_neighbors(self):
        neighbors = []
        zero_index = self.board.index(0)
        x, y = divmod(zero_index, self.size)
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
        for dx, dy in directions:
            new_x, new_y = x + dx, y + dy
            if 0 <= new_x < self.size and 0 <= new_y < self.size:
                new_zero_index = new_x * self.size + new_y
                new_board = list(self.board)
                new_board[zero_index], new_board[new_zero_index] = new_board[new_zero_index],
new_board[zero_index]
                neighbors.append(PuzzleState(tuple(new_board), self.moves + 1, self))
        return neighbors

    def misplaced_tiles(self, goal):
        count = 0
        for i in range(len(self.board)):
            if self.board[i] != 0 and self.board[i] != goal.board[i]:
                count += 1
        return count

    def __lt__(self, other):
        return False


def a_star(start, goal):
    open_set = []
    heapq.heappush(open_set, (start.misplaced_tiles(goal), start))
    g_score = {start: 0}
```

```
    f_score = {start: start.misplaced_tiles(goal)}
    closed_set = set()
    while open_set:
        current_f, current = heapq.heappop(open_set)
        if current == goal:
            path = []
            while current:
                path.append(current)
                current = current.previous
            path.reverse()
            return path
        closed_set.add(current)
        for neighbor in current.get_neighbors():
            if neighbor in closed_set:
                continue
            tentative_g = g_score[current] + 1
            if neighbor not in g_score or tentative_g < g_score[neighbor]:
                neighbor.previous = current
                g_score[neighbor] = tentative_g
                f = tentative_g + neighbor.misplaced_tiles(goal)
                f_score[neighbor] = f
                heapq.heappush(open_set, (f, neighbor))
    return None


def print_path(path):
    for state in path:
        for i in range(3):
            print(state.board[i*3:(i+1)*3])
        print()


def get_input_state(prompt):
    print(prompt)
    while True:
        try:
            values = list(map(int, input("Enter 9 numbers (0 for blank) separated by spaces:
").strip().split()))
            if len(values) != 9 or sorted(values) != list(range(9)):
                raise ValueError
            return tuple(values)
```

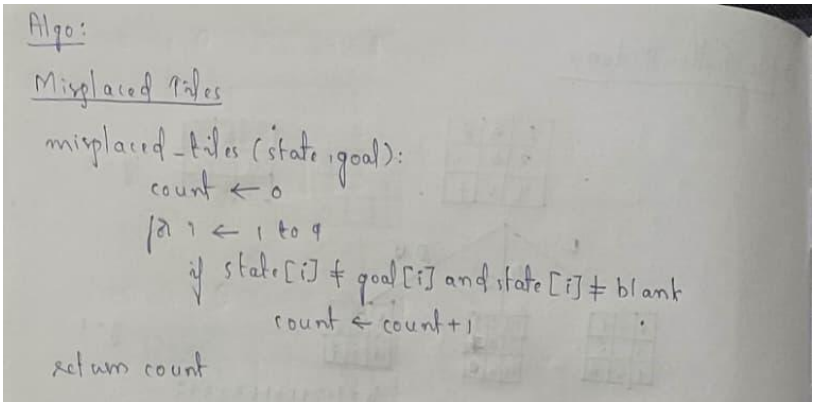
```
        except ValueError:
            print("Invalid input. Please enter numbers 0 to 8 without duplicates.")


start_board = get_input_state("Enter initial state:")
goal_board = get_input_state("Enter goal state:")

start_state = PuzzleState(start_board)
goal_state = PuzzleState(goal_board)

solution_path = a_star(start_state, goal_state)

if solution_path:
    print(f"Solution found in {len(solution_path)-1} moves:")
    print_path(solution_path)
else:
    print("No solution found.")

print("1BM23CS333")
```

Output:

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

1BM23CS333
```

2.A * SEARCH USING MISPLACED TILES METHOD

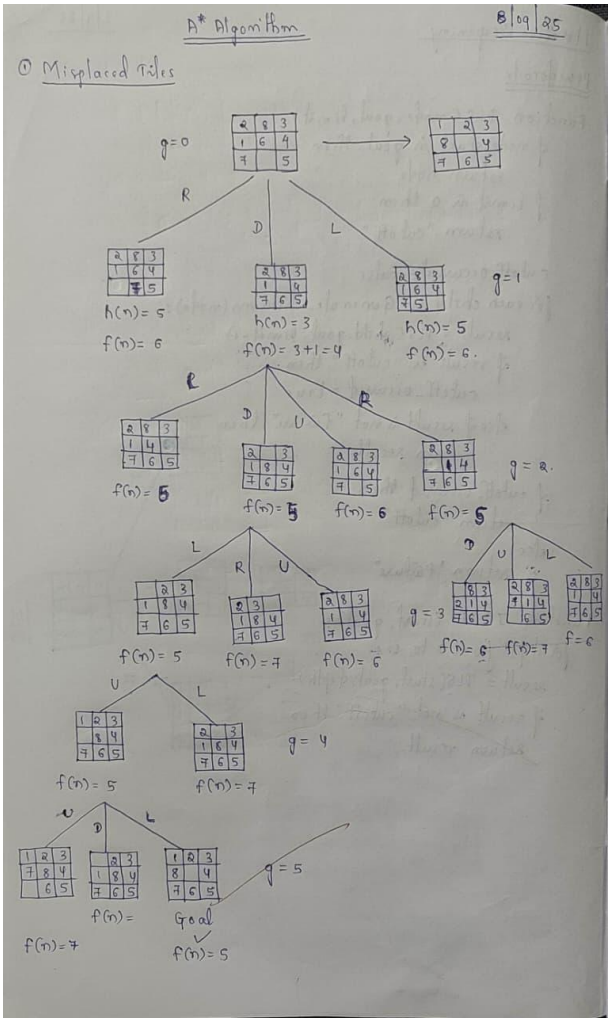
Algorithm:



Algo:

Misplaced Tiles

```
misplaced_tiles (state, goal):
    count ← 0
    for i ← 1 to 9
        if state[i] ≠ goal[i] and state[i] ≠ blank
            count ← count + 1
    return count
```



Code:

```
import heapq

def misplaced_tiles_heuristic(state, goal_state):
    misplaced_count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                misplaced_count += 1
    return misplaced_count

def get_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return -1, -1

def get_neighbors(state):
    neighbors = []
    row, col = get_blank_position(state)
    moves = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    for dr, dc in moves:
        new_row, new_col = row + dr, col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = [list(r) for r in state]
            new_state[row][col], new_state[new_row][new_col] = new_state[new_row][new_col],
new_state[row][col]
            neighbors.append([tuple(r) for r in new_state])
    return neighbors

def a_star(initial_state, goal_state):
    initial_state = tuple(tuple(row) for row in initial_state)
    goal_state = tuple(tuple(row) for row in goal_state)
    open_set = [(misplaced_tiles_heuristic(initial_state, goal_state), 0, initial_state, [])]
    closed_set = set()
    while open_set:
        f_cost, g_cost, current_state, path = heapq.heappop(open_set)
        if current_state == goal_state:
            return path + [current_state], g_cost
```

```
        if current_state in closed_set:
            continue
        closed_set.add(current_state)
        for neighbor_state_list in get_neighbors(current_state):
            neighbor_state = tuple(tuple(row) for row in neighbor_state_list)
            if neighbor_state not in closed_set:
                new_g_cost = g_cost + 1
                new_f_cost = new_g_cost + misplaced_tiles_heuristic(neighbor_state, goal_state)
                heapq.heappush(open_set, (new_f_cost, new_g_cost, neighbor_state, path +
[current_state]))
    return None, -1

def get_user_input_state(state_name):
    print(f"\nEnter the {state_name} state row by row (use 0 for the blank space, space separated):")
    state = []
    for i in range(3):
        while True:
            try:
                row = list(map(int, input(f"Row {i+1}: ").split()))
                if len(row) == 3 and all(0 <= x <= 8 for x in row):
                    state.append(row)
                    break
                else:
                    print("Invalid input. Please enter 3 numbers between 0 and 8.")
            except ValueError:
                print("Invalid input. Please enter numbers separated by spaces.")
    return state

initial_state = get_user_input_state("initial")
goal_state = get_user_input_state("goal")
path, cost = a_star(initial_state, goal_state)

if path:
    print("\nSolution Found!")
    print("Steps:")
    for step in path:
        for row in step:
            print(row)
        print()
    print(f"Cost (number of moves): {cost}")
else:
```

```
print("\nNo solution found for the given states.")
```

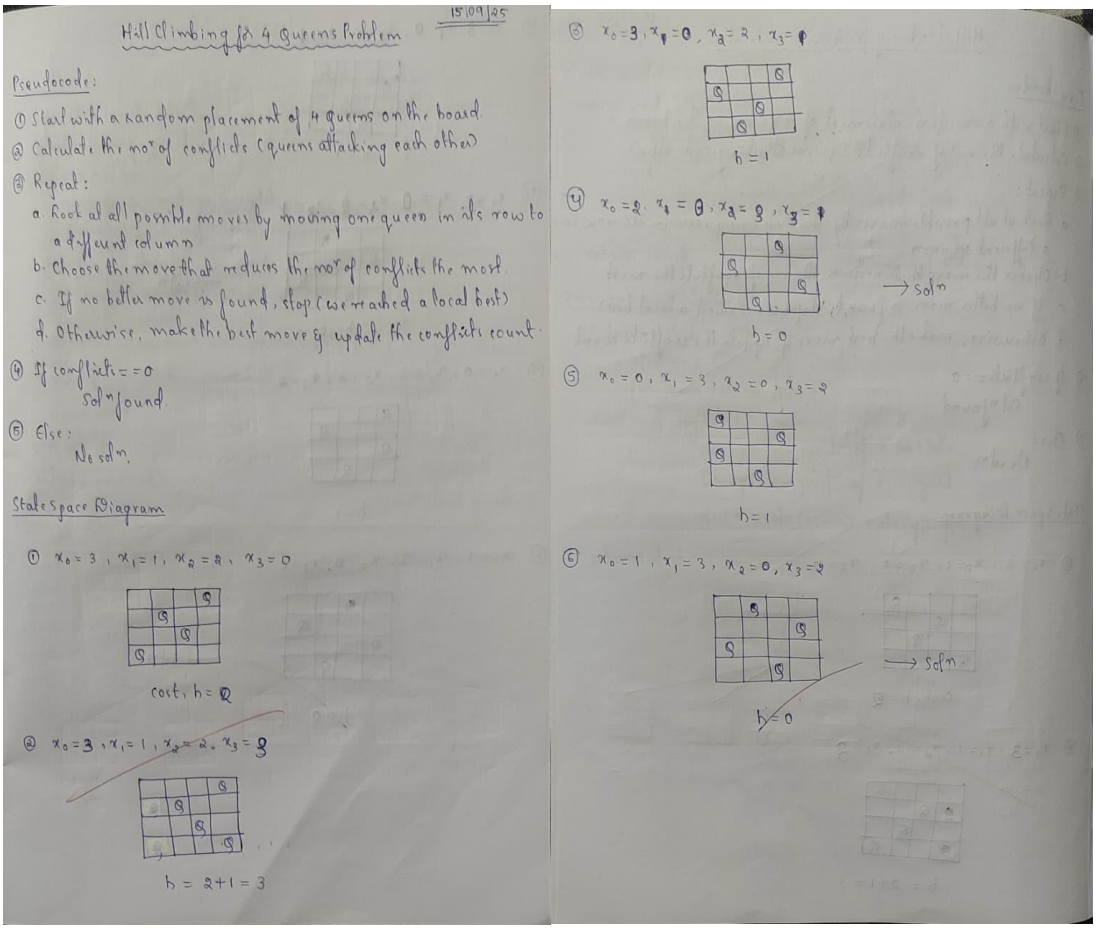
Output:

```
1BM23CS333
Enter the initial state row by row (use 0 for the blank space, space separated):
Row 1: 2 8 3
Row 2: 1 6 4
Row 3: 7 0 5
Enter the goal state row by row (use 0 for the blank space, space separated):
Row 1: 1 2 3
Row 2: 8 0 4
Row 3: 7 6 5

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

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

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

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

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

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

Cost (number of moves): 5
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Code:

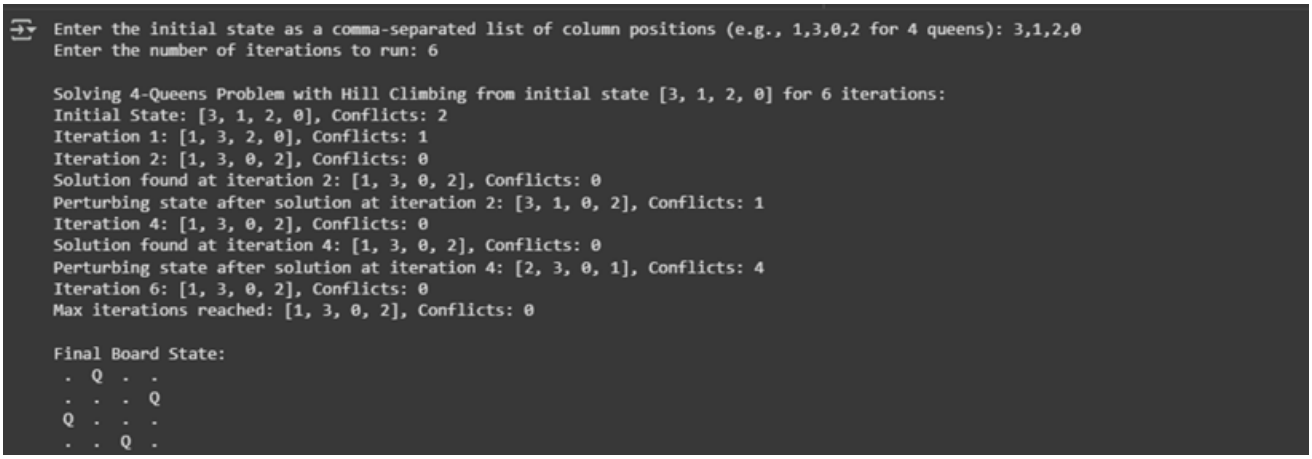
```
import random

def initial_state(n):
    state = list(range(n))
    random.shuffle(state)
    return state

def conflicts(state):
    n = len(state)
    conflict_count = 0
```

```
    for i in range(n):
        for j in range(i + 1, n):
            if abs(state[i] - state[j]) == abs(i - j):
                conflict_count += 1
    return conflict_count

def get_neighbors(state):
    n = len(state)
    neighbors = []
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = list(state)
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

def hill_climbing(initial_state_list, max_iterations=1000):
    current_state = list(initial_state_list)
    current_conflicts = conflicts(current_state)
    print(f"Initial State: {current_state}, Conflicts: {current_conflicts}")
    for i in range(max_iterations):
        if current_conflicts == 0:
            print(f"Solution found at iteration {i}: {current_state}, Conflicts:
{current_conflicts}")
            if i < max_iterations - 1:
                n = len(current_state)
                if n > 1:
                    idx1, idx2 = random.sample(range(n), 2)
                    current_state[idx1], current_state[idx2] = current_state[idx2],
current_state[idx1]
                    current_conflicts = conflicts(current_state)
                    print(f"Perturbing state after solution at iteration {i}: {current_state},
Conflicts: {current_conflicts}")
            continue
        neighbors = get_neighbors(current_state)
        if not neighbors:
            print(f"No neighbors to explore at iteration {i}: {current_state}, Conflicts:
{current_conflicts}")
            for j in range(i + 1, max_iterations):
                print(f"Iteration {j}: {current_state}, Conflicts: {current_conflicts}")
            return current_state
```

```
        best_neighbor = current_state
        best_conflicts = current_conflicts
        for neighbor in neighbors:
            neighbor_conflicts = conflicts(neighbor)
            if neighbor_conflicts < best_conflicts:
                best_conflicts = neighbor_conflicts
                best_neighbor = neighbor
        print(f"Iteration {i+1}: {best_neighbor}, Conflicts: {best_conflicts}")
        if best_conflicts >= current_conflicts:
            for j in range(i + 1, max_iterations):
                print(f"Iteration {j+1}: {current_state}, Conflicts: {current_conflicts}")
            return current_state
        current_state = best_neighbor
        current_conflicts = best_conflicts
    print(f"Max iterations reached: {current_state}, Conflicts: {current_conflicts}")
    return current_state

def print_board(state):
    if state is None:
        print("No solution found.")
        return
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[row] == col:
                line += " Q "
            else:
                line += " . "
        print(line)

initial_state_str = input("Enter the initial state as a comma-separated list of column
positions (e.g., 1,3,0,2 for 4 queens): ")
max_iterations_str = input("Enter the number of iterations to run: ")

try:
    initial_state_list = [int(x.strip()) for x in initial_state_str.split(',')]
    max_iterations = int(max_iterations_str)
    n = len(initial_state_list)
    print(f"\nSolving {n}-Queens Problem with Hill Climbing from initial state
{initial_state_list} for {max_iterations} iterations:")
```

```
    final_state = hill_climbing(initial_state_list, max_iterations)
    print("\nFinal Board State:")
    print_board(final_state)
except ValueError:
    print("Invalid input. Please enter the initial state as a comma-separated list of integers
and the number of iterations as an integer.")
except Exception as e:
    print(f"An error occurred: {e}")
```

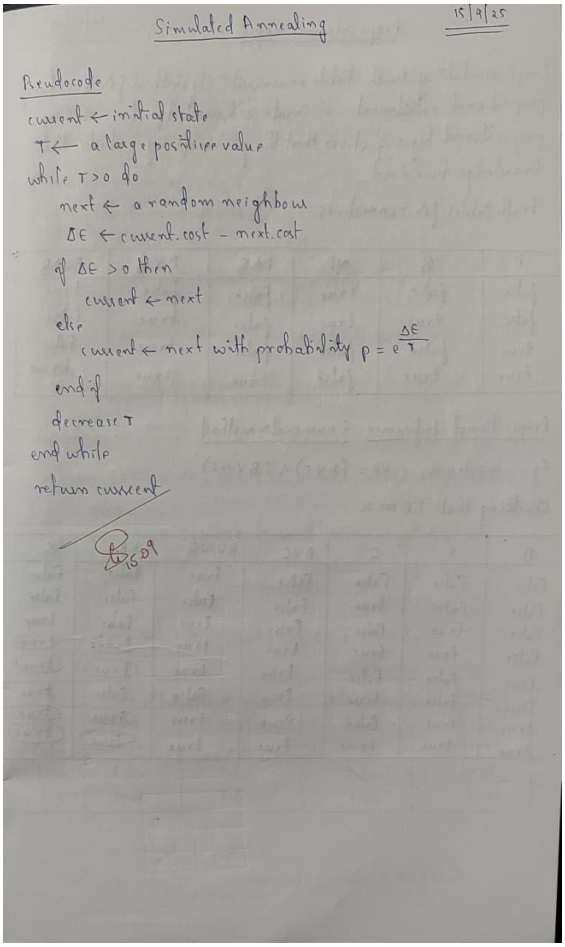
Output:



Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

```
import random
import math

def cost_function(state):
    conflicts = 0
    for i in range(len(state)):
        for j in range(i + 1, len(state)):
            if state[i] == state[j] or state[i] - i == state[j] - j or state[i] + i == state[j] + j:
                conflicts += 1
```

```
    return conflicts

def get_neighbors(state):
    neighbors = []
    for i in range(len(state)):
        for j in range(len(state)):
            if i != j:
                neighbor = list(state)
                neighbor[i] = j
                neighbors.append(neighbor)
    return neighbors

def simulated_annealing(initial_state, temperature, cooling_rate):
    current_state = initial_state
    current_cost = cost_function(current_state)
    best_state = current_state
    best_cost = current_cost
    while temperature > 0.1:
        neighbors = get_neighbors(current_state)
        if not neighbors:
            break
        next_state = random.choice(neighbors)
        next_cost = cost_function(next_state)
        delta_cost = next_cost - current_cost
        if delta_cost < 0 or random.random() < math.exp(-delta_cost / temperature):
            current_state = next_state
            current_cost = next_cost
        if current_cost < best_cost:
            best_state = current_state
            best_cost = current_cost
        temperature *= cooling_rate
    return best_state, best_cost

def solve_8_queens_simulated_annealing():
    n = 8
    initial_state = [random.randint(0, n - 1) for _ in range(n)]
    temperature = 1000
    cooling_rate = 0.95
    solution, cost = simulated_annealing(initial_state, temperature, cooling_rate)
    if cost == 0:
        print("Solution found:")
```

```
        print(solution)
    else:
        print("No perfect solution found, best found with conflicts:", cost)
        print(solution)

solve_8_queens_simulated_annealing()
```

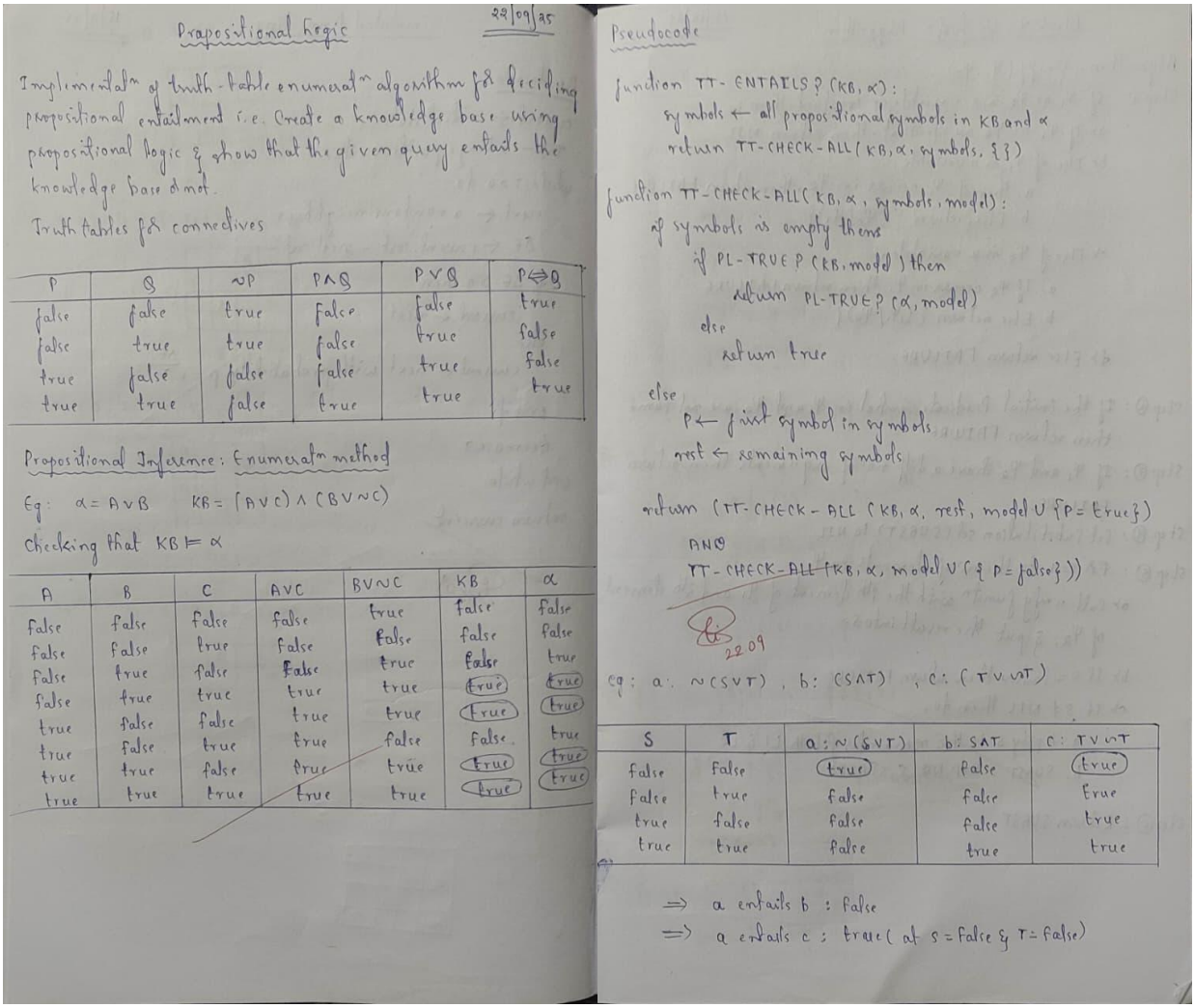
Output:

```
Solution found:
[4, 2, 0, 5, 7, 1, 3, 6]
```

Program 6

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

Algorithm:



Propositional Logic 22/09/25

Implementation of truth-table enumeration algorithm for deciding propositional entailment i.e. Create a knowledge base using propositional logic & show that the given query entails the knowledge base or not.

Truth tables for connectives

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

Propositional Inference: Enumeration method

Eg: α = A∨B KB = (A∨C)∧(B∨¬C)

Checking that KB ⊨ α

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

Pseudocode

```
function TT-ENTAILS?(KB, α):
    symbols ← all propositional symbols in KB and α
    return TT-CHECK-ALL(KB, α, symbols, {})

function TT-CHECK-ALL(KB, α, symbols, model):
    if symbols is empty then
        if PL-TRUE?(KB, model) then
            return PL-TRUE?(α, model)
        else
            return true
    else
        P ← first symbol in symbols
        rest ← remaining symbols
        return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true}))
            AND
            TT-CHECK-ALL(KB, α, model ∪ {P = false}))
```

22/09

eg: a: ¬(S∨T) , b: (S∧T) , c: (T∨¬T)

S	T	a: ¬(S∨T)	b: S∧T	c: T∨¬T
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 & 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")
            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))
    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
    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

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

Output:

```
Truth Table:
```

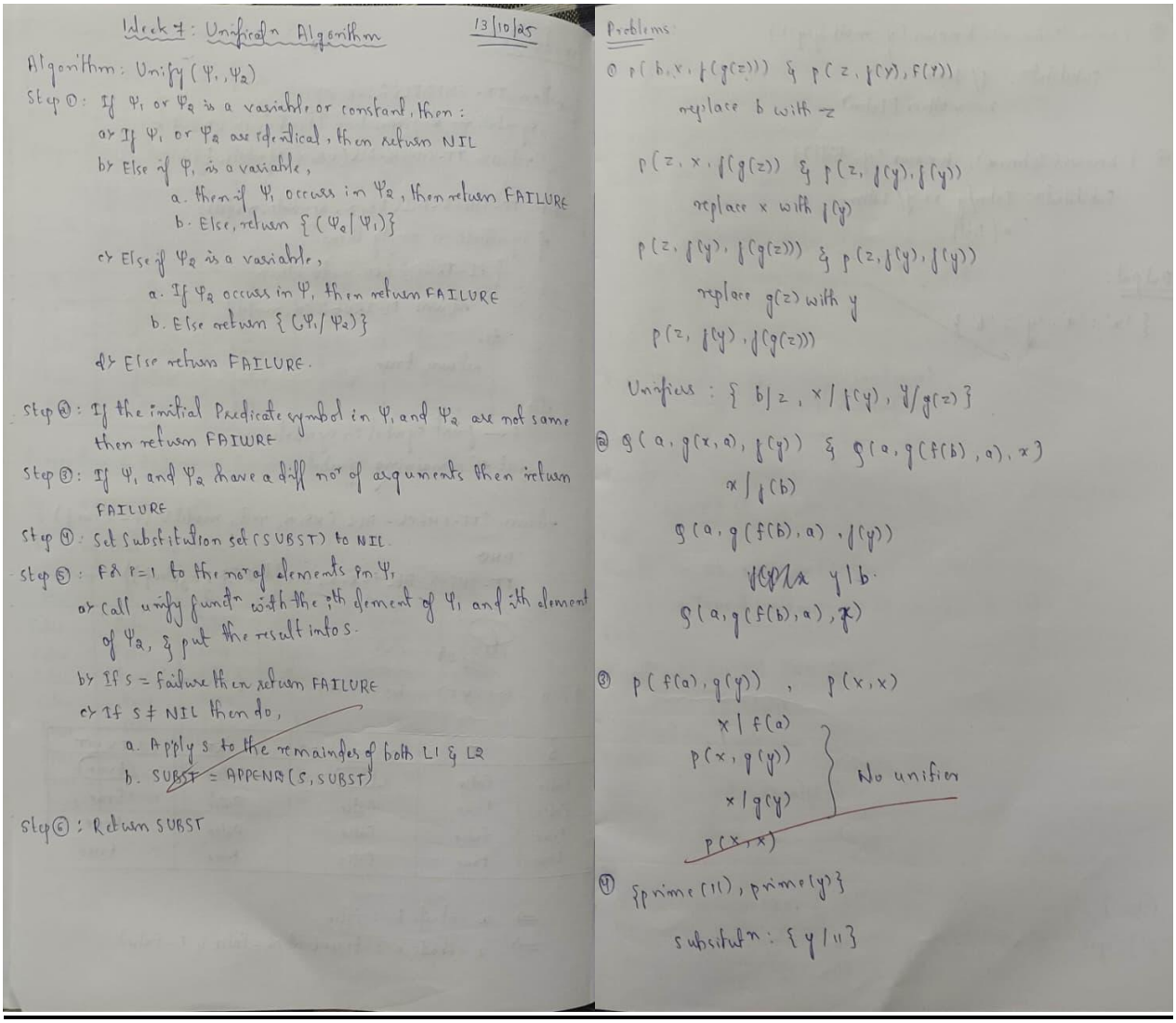
A	B	C	KB	alpha	KB ∧ α
True	True	True	True	True	True
True	True	False	False	True	False
True	False	True	True	True	True
True	False	False	True	True	True
False	True	True	True	True	True
False	True	False	False	False	False
False	False	True	False	True	False
False	False	False	False	False	False

```
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
```

Program 7

Implement unification in first order logic

Algorithm:



Ideck 7: Unification Algorithm 13/10/25

Algorithm: Unify(Ψ_1, Ψ_2)

Step ①: If Ψ_1 or Ψ_2 is a variable or constant, then:
- a) If Ψ_1 or Ψ_2 are identical, then return NIL
- b) Else if Ψ_1 is a variable,
  - a. then if Ψ_1 occurs in Ψ_2, then return FAILURE
  - b. Else, return $\{(\Psi_2/\Psi_1)\}$
- c) Else if Ψ_2 is a variable,
  - a. If Ψ_2 occurs in Ψ_1, then return FAILURE
  - b. Else return $\{(\Psi_1/\Psi_2)\}$
- d) Else return FAILURE.

Step ②: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same then return FAIWRE

Step ③: If Ψ_1 and Ψ_2 have a diff no of arguments then return FAILURE

Step ④: Set Substitution set (SUBST) to NIL

Step ⑤:
- a) For i=1 to the no of elements in Ψ_1
  - Call unify funcn with the ith element of Ψ_1 and ith element of Ψ_2, & 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 L1 & L2
  - b. SUBST = APPEND(S, SUBST)

Step ⑥: Return SUBST

Problems

① p(b, x, f(g(z))) & p(z, f(y), f(y))

replace b with z

p(z, x, f(g(z))) & p(z, f(y), f(y))

replace x with f(y)

p(z, f(y), f(g(z))) & p(z, f(y), f(y))

replace g(z) with y

p(z, f(y), f(g(z)))

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

② g(a, g(x, a), f(y)) & g(a, g(f(b), a), x)

x/f(b)

g(a, g(f(b), a), f(y))

y/b

g(a, g(f(b), a), x)

③ p(f(a), g(y)) , p(x, x)

x/f(a)

p(x, g(y))

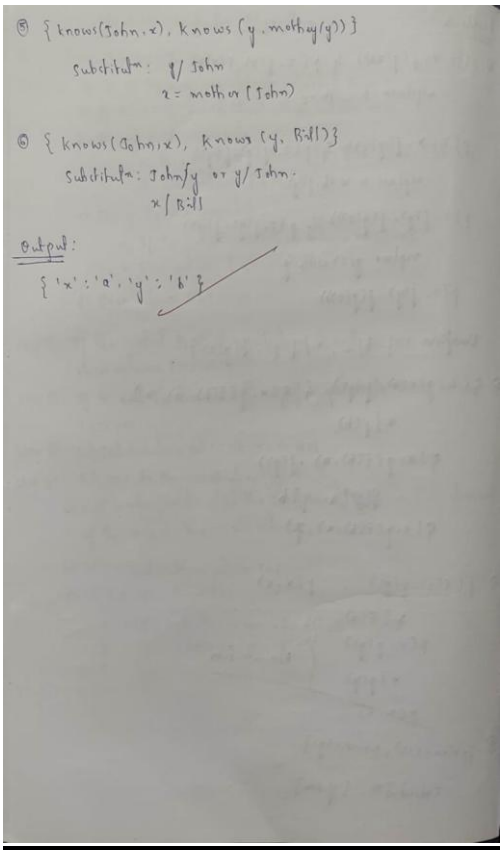
x/g(y)

No unifier

p(x, x)

④ {prime(11), prime(y)}

substitutn: {y/11}



Code:

```
def unify(psi1, psi2):
    if is_variable_or_constant(psi1) or is_variable_or_constant(psi2):
        if psi1 == psi2:
            return {}
        elif is_variable(psi1):
            if occurs_in(psi1, psi2):
                return "FAILURE"
            else:
                return {psi1: psi2}
        elif is_variable(psi2):
            if occurs_in(psi2, psi1):
                return "FAILURE"
            else:
                return {psi2: psi1}
        else:
            return "FAILURE"

    if predicate_symbol(psi1) != predicate_symbol(psi2):
        return "FAILURE"
```

```
    if len(psi1['args']) != len(psi2['args']):
        return "FAILURE"

    SUBST = {}

    for i in range(len(psi1['args'])):
        S = unify(psi1['args'][i], psi2['args'][i])
        if S == "FAILURE":
            return "FAILURE"
        if S:
            psi1 = apply_substitution(S, psi1)
            psi2 = apply_substitution(S, psi2)
            SUBST.update(S)

    return SUBST


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


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


def occurs_in(var, expr):
    if var == expr:
        return True
    if isinstance(expr, dict):
        return any(occurs_in(var, arg) for arg in expr.get('args', []))
    return False


def predicate_symbol(expr):
    if isinstance(expr, dict) and 'pred' in expr:
        return expr['pred']
    return None


def apply_substitution(subst, expr):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    elif isinstance(expr, dict):
        return {
            'pred': expr['pred'],
```

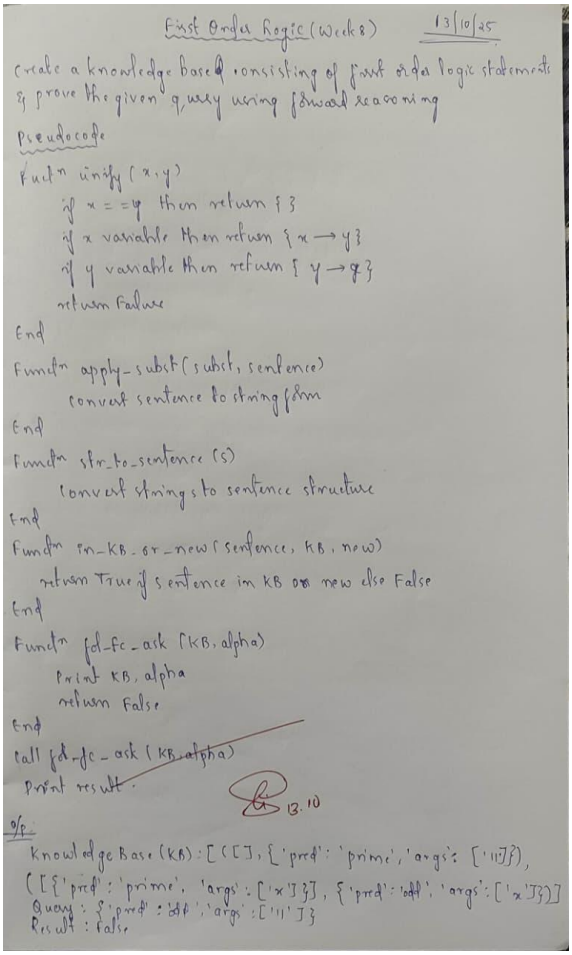
```
            'args': [apply_substitution(subst, arg) for arg in expr.get('args', [])]
        }
    return expr


# Example usage
psi1 = {'pred': 'P', 'args': ['x', 'y']}
psi2 = {'pred': 'P', 'args': ['a', 'b']}
result = unify(psi1, psi2)
print(result)
```

Output:

Program 8

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

Algorithm:


First Order Logic (Week 8) 13/10/25

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

Pseudocode

Fuctn unify (x, y)
  if x == y then return {}
  if x variable then return {x → y}
  if y variable then return {y → x}
  return Failure
End
Functn apply_subst (subst, sentence)
  convert sentence to string form
End
Functn str_to_sentence (s)
  convert strings to sentence structure
End
Functn in_KB_or_new (sentence, KB, new)
  return True if sentence in KB or new else False
End
Functn fol_fc_ask (KB, alpha)
  Print KB, alpha
  return False
End
call fol_fc_ask (KB, alpha)
Print result.

13.10

o/p:
Knowledge Base (KB): [([], {'pred': 'prime', 'args': ['11']}),
([{'pred': 'prime', 'args': ['x']}, {'pred': 'odd', 'args': ['x']}])]
Query: {'pred': 'odd', 'args': ['11']}
Result: False.

Code:

```
def unify(x, y):
    # A simple unifier (can be replaced by your detailed unify function)
    if x == y:
        return {}
    if isinstance(x, str) and x.islower():
        return {x: y}
    if isinstance(y, str) and y.islower():
        return {y: x}
    return "FAILURE"
```

```
def apply_substitution(subst, sentence):
    if isinstance(sentence, str):
        return subst.get(sentence, sentence)
    elif isinstance(sentence, dict):
        return {
            'pred': sentence['pred'],
            'args': [apply_substitution(subst, arg) for arg in sentence.get('args', [])]
        }
    return sentence


def sentence_to_str(sentence):
    if isinstance(sentence, str):
        return sentence
    elif isinstance(sentence, dict):
        args_str = ",".join(sentence_to_str(arg) for arg in sentence.get('args', []))
        return f"{sentence['pred']}({args_str})"
    return str(sentence)


def str_to_sentence(s):
    # Very basic parser assuming format pred(arg1,arg2,...)
    pred_end = s.find("(")
    if pred_end == -1:
        return s
    pred = s[:pred_end]
    args_str = s[pred_end+1:-1]
    args = args_str.split(",") if args_str else []
    return {'pred': pred, 'args': args}


def sentence_in_KB_or_new(sentence, KB, new):
    s_str = sentence_to_str(sentence)
    for rule in KB:
        _, concl = rule
        if sentence_to_str(concl) == s_str:
            return True
    if s_str in new:
        return True
    return False


def find_substitutions_for_premises(premises, KB):
    # For demo: if no premises, return [{}] (empty substitution)
    if not premises:
```

```
            return [{}]
    # Otherwise, just return empty for simplicity
    return [{}]


# Placeholder for the fol_fc_ask function
def fol_fc_ask(KB, alpha):
    """
    A placeholder function for forward chaining inference.
    Replace with your actual implementation.
    """
    print("fol_fc_ask function called.")
    print("Knowledge Base (KB):", KB)
    print("Query (alpha):", alpha)
    # This is where your forward chaining logic would go.
    # For demonstration, let's assume it returns False.
    return False


# Example knowledge base: list of (premises, conclusion)
KB = [
    ([], {'pred': 'prime', 'args': ['11']}),  # Fact: prime(11)
    ([{'pred': 'prime', 'args': ['x']}], {'pred': 'odd', 'args': ['x']})  # Rule: prime(x) => odd(x)
]

alpha = {'pred': 'odd', 'args': ['11']}  # Query: odd(11)

result = fol_fc_ask(KB, alpha)

print("Result:", result)
```

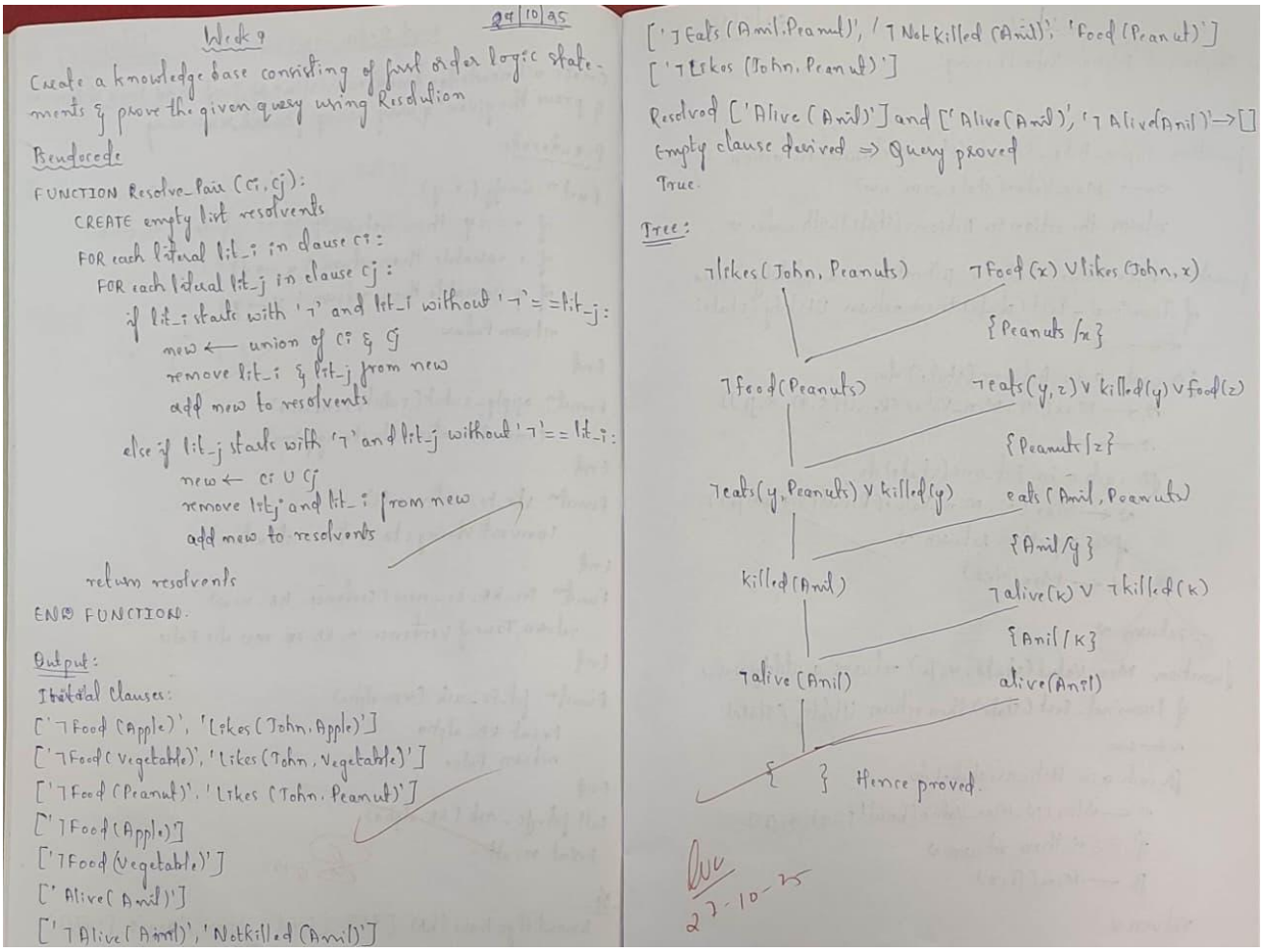
Output:

```
fol_fc_ask function called.
Knowledge Base (KB): [([], {'pred': 'prime', 'args': ['11']}), ([{'pred': 'prime', 'args': ['x']}], {'pred': 'odd', 'args': ['x']})]
Query (alpha): {'pred': 'odd', 'args': ['11']}
Result: False
```

Program 9

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

Algorithm:



Code:

```
from copy import deepcopy

def resolve_pair(ci, cj):
    resolvents = []
    for lit_i in ci:
        for lit_j in cj:
            if lit_i.startswith('¬') and lit_i[1:] == lit_j:
                new_clause = list(set(ci + cj))
                new_clause.remove(lit_i)
```

```
            new_clause.remove(lit_j)
            resolvents.append(new_clause)
        elif lit_j.startswith('¬') and lit_j[1:] == lit_i:
            new_clause = list(set(ci + cj))
            new_clause.remove(lit_j)
            new_clause.remove(lit_i)
            resolvents.append(new_clause)
    return resolvents


def resolution(KB, query):
    clauses = deepcopy(KB)
    clauses.append(['¬' + query])  # Add negated query

    print("Initial Clauses:")
    for c in clauses:
        print(c)

    new = set()

    while True:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j]) for i in range(n) for j in range(i + 1, n)]

        for (ci, cj) in pairs:
            resolvents = resolve_pair(ci, cj)
            for res in resolvents:
                if not res:
                    print(f"\nResolved {ci} and {cj} -> []")
                    print("☑ Empty clause derived ⇒ Query PROVED!")
                    return True
                new.add(tuple(sorted(res)))

        new_clauses = [list(x) for x in new if list(x) not in clauses]

        if not new_clauses:
            print("\nNo new clauses ⇒ Query cannot be proved.")
            return False

        for c in new_clauses:
            clauses.append(c)


# --- Grounded Knowledge Base ---
KB = [
```

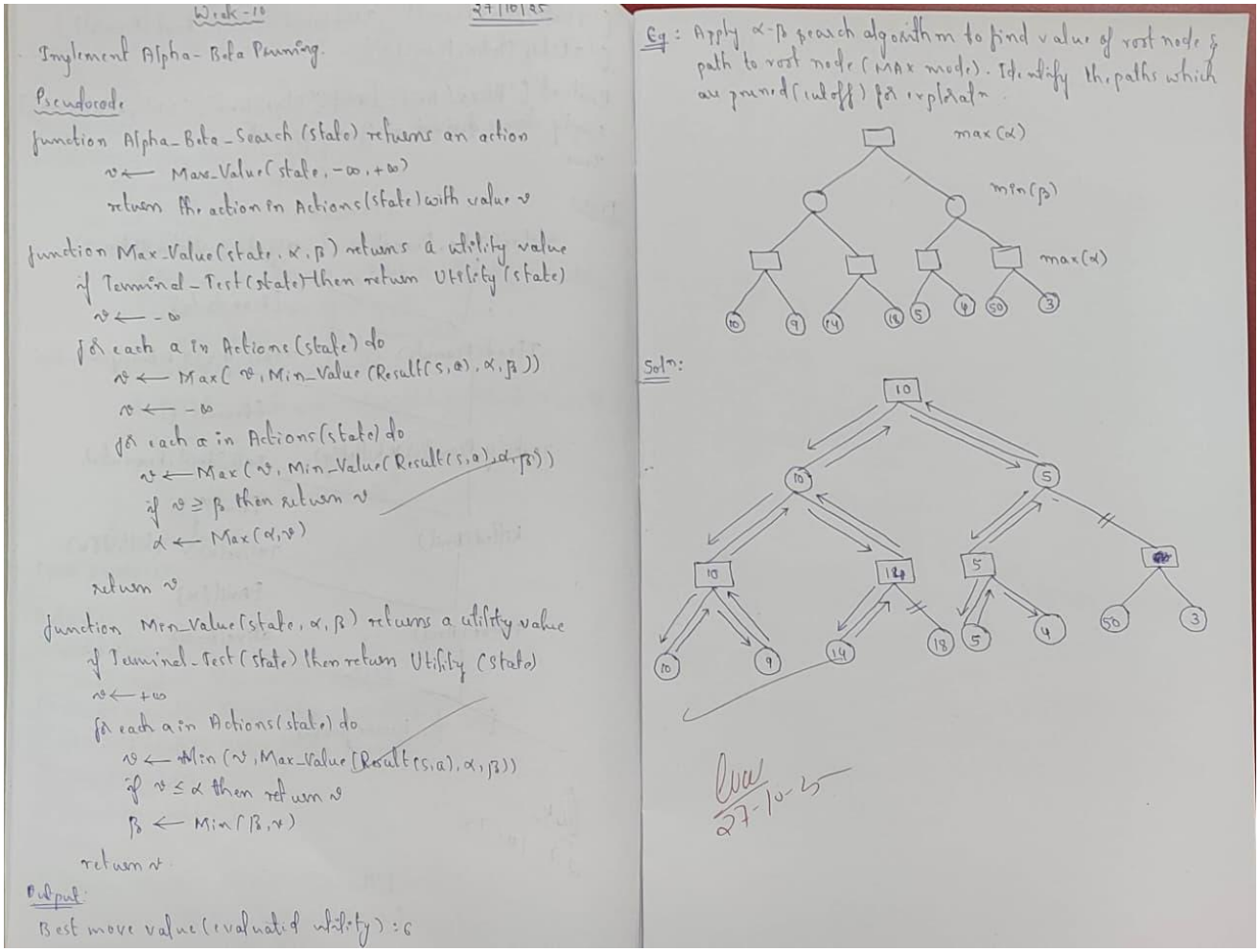
```
    ['¬Food(Apple)', 'Likes(John,Apple)'],       # John likes all kinds of food
    ['¬Food(Vegetable)', 'Likes(John,Vegetable)'],
    ['¬Food(Peanut)', 'Likes(John,Peanut)'],     # All foods liked by John
    ['Food(Apple)'],                             # Apple is food
    ['Food(Vegetable)'],                         # Vegetable is food
    ['Alive(Anil)'],                             # Anil is alive
    ['¬Alive(Anil)', 'NotKilled(Anil)'],         # Alive ⇒ NotKilled
    ['¬NotKilled(Anil)', 'Alive(Anil)'],         # NotKilled ⇒ Alive
    ['Eats(Anil,Peanut)'],                       # Anil eats peanuts
    ['¬Eats(Anil,Peanut)', '¬NotKilled(Anil)', 'Food(Peanut)'],  # Eats & NotKilled ⇒ Food
]

query = 'Likes(John,Peanut)'

# --- Run resolution ---
resolution(KB, query)
```

Output:

Program 10

Implement Alpha-Beta Pruning.

Algorithm:


Week-10 27/10/25

Implement Alpha-Beta Pruning.

Pseudocode

function Alpha_Beta_Search (state) returns an action
    v ← Max_Value(state, -∞, +∞)
    return the action in Actions(state) with value v

function Max_Value(state, α, β) returns a utility value
    if Terminal_Test(state) then return Utility(state)
    v ← -∞
    for each a in Actions(state) do
        v ← Max(v, Min_Value(Result(s,a), α, β))
        v ← -∞
        for each a in Actions(state) do
            v ← Max(v, Min_Value(Result(s,a), α, β))
            if v ≥ β then return v
            α ← Max(α, v)
    return v

function Min_Value(state, α, β) returns a utility value
    if Terminal_Test(state) then return Utility(state)
    v ← +∞
    for each a in Actions(state) do
        v ← Min(v, Max_Value(Result(s,a), α, β))
        if v ≤ α then return v
        β ← Min(β, v)
    return v

Output:
Best move value (evaluated utility) : 6

Eg: Apply α-β search algorithm to find value of root node & path to root node (MAX mode). Identify the paths which are pruned (cutoff) for explanation

Soln:

27-10-25

Code:

```
def alpha_beta_search(state):
    """
    Returns the best action and its evaluated value using Alpha-Beta pruning.
    """
    value, move = max_value(state, float('-inf'), float('inf'))
    return value, move


def max_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state), None
```

```
    value = float('-inf')
    best_move = None

    for action in actions(state):
        v, _ = min_value(result(state, action), alpha, beta)
        if v > value:
            value = v
            best_move = action
        if value >= beta:
            return value, best_move  # Beta cutoff
        alpha = max(alpha, value)

    return value, best_move


def min_value(state, alpha, beta):
    if terminal_test(state):
        return utility(state), None

    value = float('inf')
    best_move = None

    for action in actions(state):
        v, _ = max_value(result(state, action), alpha, beta)
        if v < value:
            value = v
            best_move = action
        if value <= alpha:
            return value, best_move  # Alpha cutoff
        beta = min(beta, value)

    return value, best_move


# --------------------------
# Example Toy Game Functions
# --------------------------

def actions(state):
    """Return all possible actions from this state."""
    return state.get('actions', [])


def result(state, action):
```

```
    """Return the next state after performing action."""
    return action['next']


def terminal_test(state):
    """Check if this is a terminal (leaf) state."""
    return state.get('terminal', False)


def utility(state):
    """Return the utility (score) of a terminal state."""
    return state.get('utility', 0)


# --------------------------
# Example Game Tree
# --------------------------
if __name__ == "__main__":
    # Leaf nodes with known utilities
    leaf1 = {'terminal': True, 'utility': 3}
    leaf2 = {'terminal': True, 'utility': 5}
    leaf3 = {'terminal': True, 'utility': 6}
    leaf4 = {'terminal': True, 'utility': 9}

    # Intermediate nodes
    B = {'actions': [{'next': leaf1}, {'next': leaf2}]}  # MIN layer
    C = {'actions': [{'next': leaf3}, {'next': leaf4}]}  # MIN layer

    # Root node (MAX layer)
    root = {'actions': [{'next': B}, {'next': C}]}

    # Run alpha-beta search
    best_value, best_move = alpha_beta_search(root)
    print("Best move value (evaluated utility):", best_value)
```

Output:

```
Best move value (evaluated utility): 6
```