VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sujan G E (1BM23CS347)

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 **Sujan G E(1BM23CS347),** 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.

Mrs. Seema Patil Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

9	27-10-2025	Create a knowledge base consisting of first order logic statements and prove the given query using Resolution	46
10	27-10-2025	Implement Alpha-Beta Pruning.	51

Sujan

AI

1BM23CS314

INDEX

Name : ..... Subject : .....

Std. : ..... Div. : ..... Roll No. : .....

School / College : .....

Sl No.	Date	Title	Page No.	Teacher Sign/ Remarks
1	18/8/25	Implement Tic - TAC -TOE	10	8/9/25
2	25/8/25	Vaccum cleaner agent		
3A		BFS without Huristic approch	10	01.09
3B		BFC with Huristic approch		
3C		iterative Deepening DFS	10	
4A	8/9	A* with manhattan.	10	8/9
4B	8/9	A* with Misplaced.		
5A	15/9	Hill climbing search	10	15/9
6	22/9	Simulated Annealing		
		Propotional logic		13/10/25
7	13/10	Unification		
		First order logic		
8		FOL - Resolution		27/10/25
		Alpha - beta - search.		

Completed

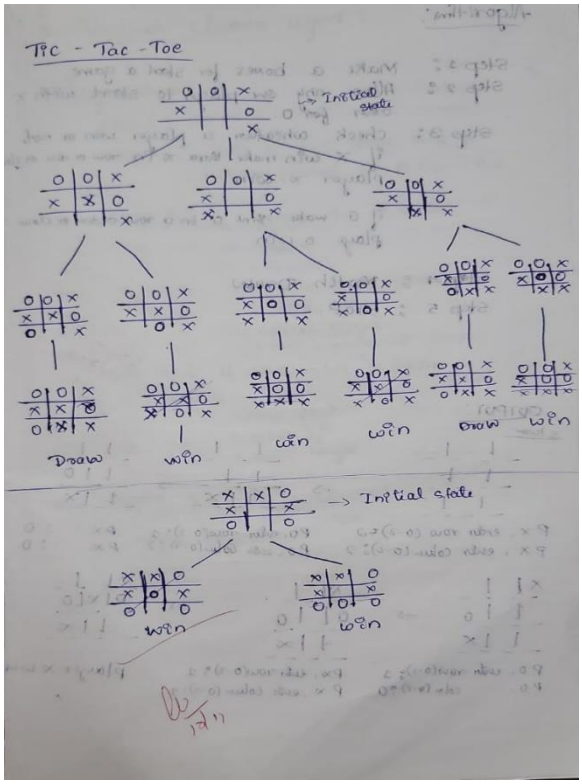
Github Link: https://github.com/Sujan279/AI

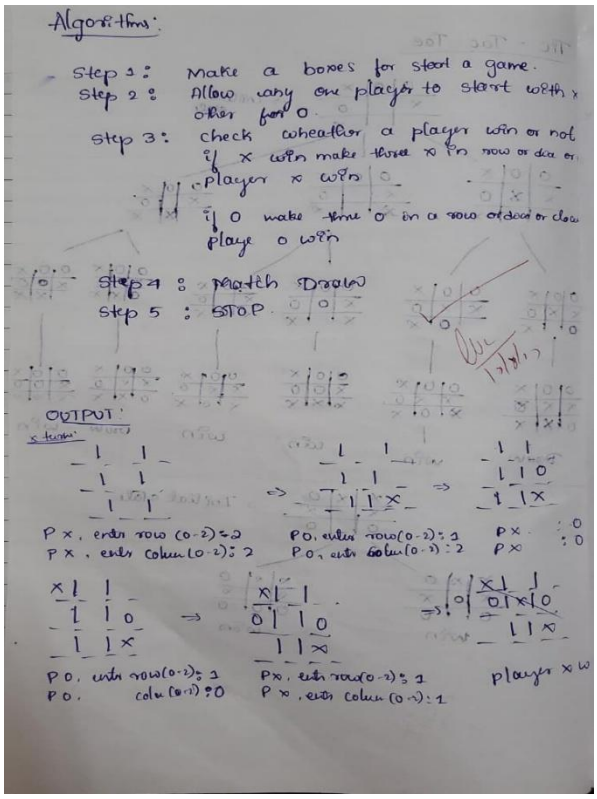
Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

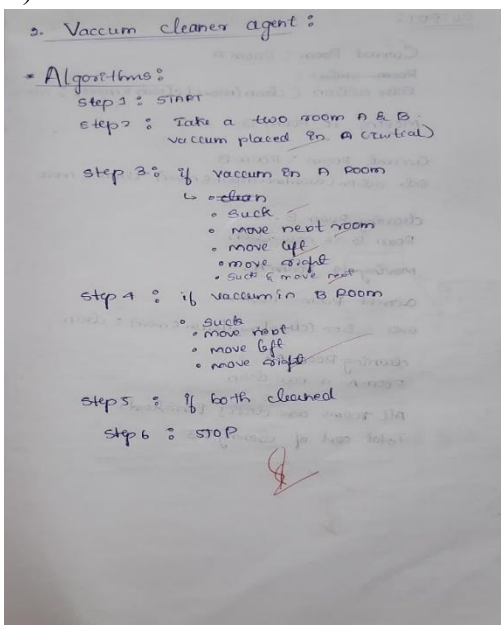
Algorithm:

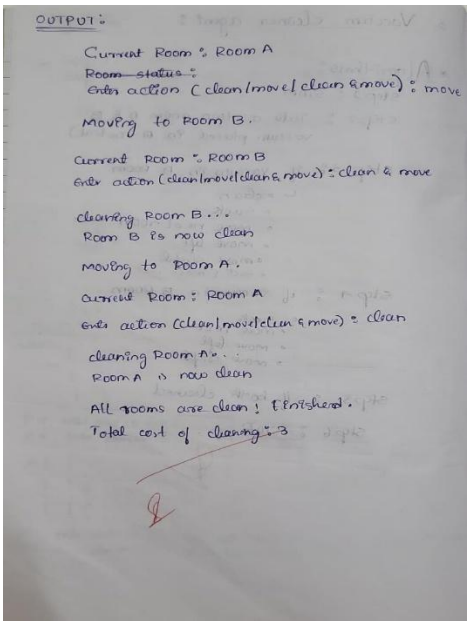
a) Tic Tac Toe





b) Vacuum Cleaner





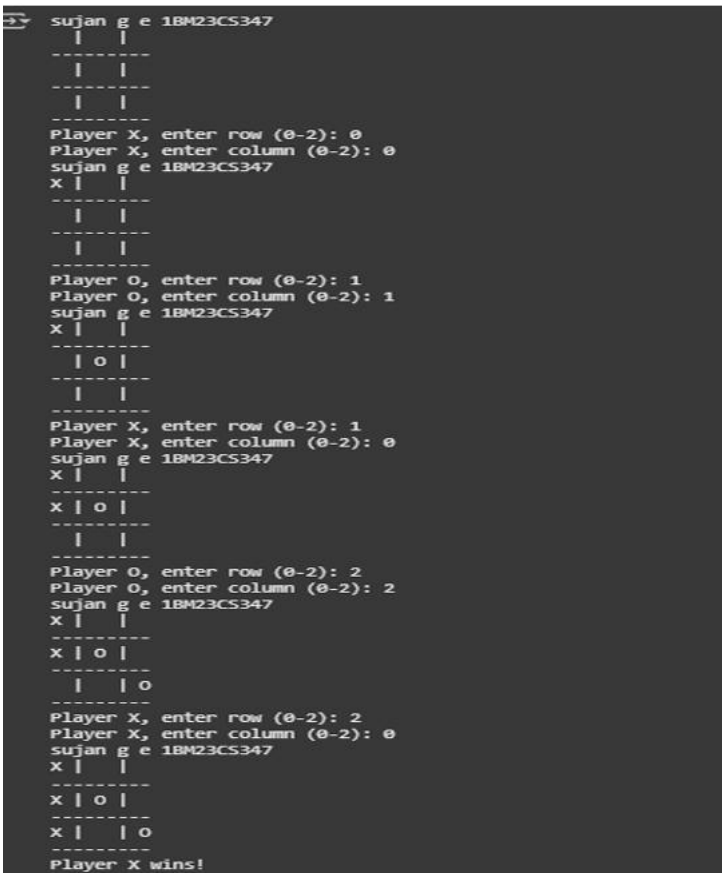
Code:

a)Tic Tac Toe

```
def print_board(board):
    print("sujan g e 1BM23CS347")
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_win(board, 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)]) or all([board[i][2-i] == player for i in range(3)]):
        return True
    return False

def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"
    while True:
        print_board(board)
        try:
```

```
            row = int(input(f"Player {current_player}, enter row (0-2): "))
            col = int(input(f"Player {current_player}, enter column (0-2): "))
            if board[row][col] == " ":
                board[row][col] = current_player
                if check_win(board, current_player):
                    print_board(board)
                    print(f"Player {current_player} wins!")
                    break
                elif all([cell != " " for row in board for cell in row]):
                    print_board(board)
                    print("It's a tie!")
                    break
                current_player = "O" if current_player == "X" else "X"
            else:
                print("That spot is already taken. Try again.")
        except (ValueError, IndexError):
            print("Invalid input. Please enter numbers between 0 and 2.")

play_game()
```

Output:



b) Vacuum Cleaner

```
class VacuumCleanerAgent:
    def __init__(self):
        print("Sujan G E 1BM23CS347")
        self.rooms = {'Room A': 0, 'Room B': 0}
        self.current_room = 'Room A'

    def display_status(self):
        print("\nCurrent Room:", self.current_room)
        print("Room Status:", self.rooms)

    def clean_room(self):
        if self.rooms[self.current_room] == 0:
            print(f"\nCleaning {self.current_room}...")
            self.rooms[self.current_room] = 1
            print(f"{self.current_room} is now clean.")
        else:
            print(f"\n{self.current_room} is already clean.")

    def move_to_next_room(self):
        if self.current_room == 'Room A':
            self.current_room = 'Room B'
            print("\nMoving to Room B.")
        elif self.current_room == 'Room B':
            self.current_room = 'Room A'
            print("\nMoving to Room A.")

    def are_all_rooms_clean(self):
        return all(status == 1 for status in self.rooms.values())

agent = VacuumCleanerAgent()

while not agent.are_all_rooms_clean():
    agent.display_status()
    user_input = input("Enter action (clean/move/clean and move): ").lower()

    if user_input == 'clean':
        agent.clean_room()
    elif user_input == 'move':
        agent.move_to_next_room()
    elif user_input == 'clean and move':
```

```
        agent.clean_room()
        if not agent.are_all_rooms_clean():
            agent.move_to_next_room()
    else:
        print("\nInvalid input. Please enter 'clean', 'move', or 'clean and move'.")

print("\nAll rooms are clean! Simulation finished.")
```

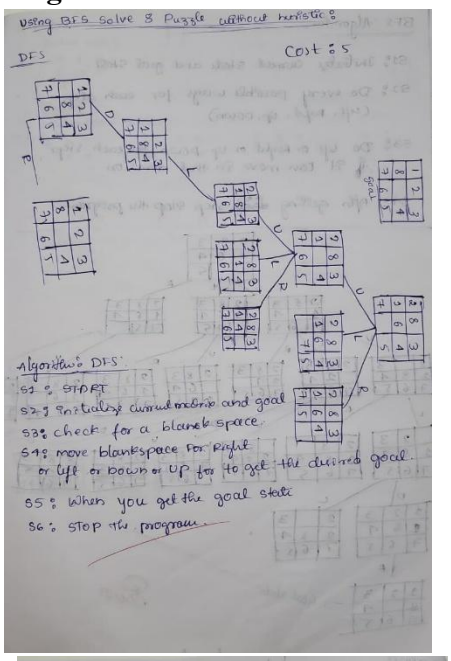
Output:

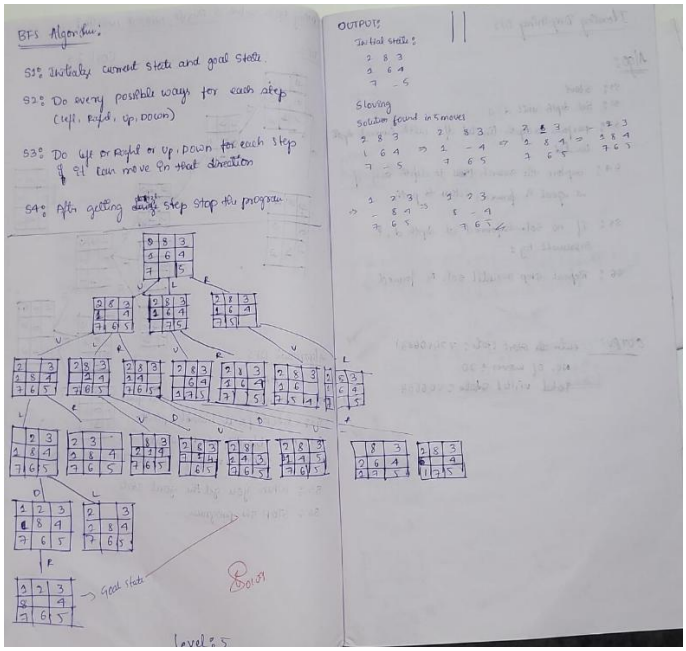
```
Sujan G E 1BM23CS347

Current Room: Room A
Room Status: {'Room A': 0, 'Room B': 0}
Enter action (clean/move/clean and move): move

Moving to Room B.

Current Room: Room B
Room Status: {'Room A': 0, 'Room B': 0}
Enter action (clean/move/clean and move): clean and move

Cleaning Room B...
Room B is now clean.

Moving to Room A.

Current Room: Room A
Room Status: {'Room A': 0, 'Room B': 1}
Enter action (clean/move/clean and move): clean

Cleaning Room A...
Room A is now clean.

All rooms are clean! Simulation finished.
```

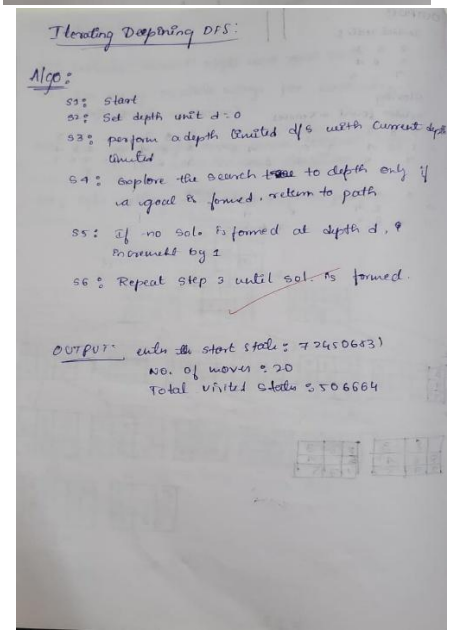
Program 2

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

Algorithm:







Code:

a) DFS goal_state =
'123804765'

```
moves = {
   'U': -3,
   'D': 3,
   'L': -1,
   'R': 1
}

invalid_moves = {
   0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
   3: ['L'],        5: ['R'],
   6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
   index = state.index('0')     if direction in
invalid_moves.get(index, []):
       return None

   new_index = index + moves[direction]
if new_index < 0 or new_index >= 9:
       return None

   state_list = list(state)     state_list[index], state_list[new_index] =
state_list[new_index], state_list[index]     return ''.join(state_list)

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

def dfs(start_state, max_depth=50):
   visited = set()     stack = [(start_state, [])]  # Each
element: (state, path)

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

       if current_state in visited:
```

```
            continue

        # Print every visited state
print("Visited state:")
print_state(current_state)

        if current_state == goal_state:
            return path

        visited.add(current_state)

        if len(path) >= max_depth:
            continue

        for direction in moves:
            new_state = move_tile(current_state, direction)
if new_state and new_state not in visited:
                stack.append((new_state, path + [direction]))

    return None


start = input("Enter the INITIAL state (give '0'for empty space): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("INITIAL state:")
print_state(start)

    result = dfs(start)

    if result is not None:
        print("Solution found!")
print("Moves:", ' '.join(result))
print("Number of moves:", len(result))

        current_state = start         for i, move in
enumerate(result, 1):             current_state =
move_tile(current_state, move)             print(f"Move
{i}: {move}")             print_state(current_state)
else:         print("No solution exists for the given start
state or max depth reached.") else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

print("1BM23CS347 Sujan g e")
Output:
Enter start state (e.g., 724506831): 123456078 Start
state:
1 2 3
4 5 6
  7 8

Visited state:
1 2 3
4 5 6
  7 8

Visited state:
1 2 3
4 5 6
7   8

Visited state:
1 2 3
4 5 6
7 8

Solution found!
Moves: R R
Number of moves: 2
Move 1: R
1 2 3
4 5 6
7   8
Move 2: R
1 2 3
4 5 6
7 8
1BM23CS347 Sujan g e

b) Iterative Deepening Search

goal_state = '123456780'

moves = {

```
    'U': -3,
    'D': 3,
    'L': -1,
    'R': 1
}

invalid_moves = {
    0: ['U', 'L'], 1: ['U'], 2: ['U', 'R'],
    3: ['L'],        5: ['R'],
    6: ['D', 'L'], 7: ['D'], 8: ['D', 'R']
}

def move_tile(state, direction):
    index = state.index('0')     if direction in
invalid_moves.get(index, []):
        return None

    new_index = index + moves[direction]
if new_index < 0 or new_index >= 9:
        return None

    state_list = list(state)     state_list[index], state_list[new_index] =
state_list[new_index], state_list[index]     return ''.join(state_list)

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

def dls(state, depth, path, visited, visited_count):
    visited_count[0] += 1  # Increment visited states count
if state == goal_state:
        return path

    if depth == 0:
return None

    visited.add(state)

    for direction in moves:
        new_state = move_tile(state, direction)
if new_state and new_state not in visited:
```

```
        result = dls(new_state, depth - 1, path + [direction], visited, visited_count)
if result is not None:
            return result

    visited.remove(state)
return None

def iddfs(start_state, max_depth=50):
    visited_count = [0]  # Using list to pass by reference
for depth in range(max_depth + 1):        visited = set()
result = dls(start_state, depth, [], visited, visited_count)
if result is not None:
        return result, visited_count[0]
return None, visited_count[0]

# Main start = input("Enter start state (e.g.,
724506831): ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start state:")
print_state(start)

    result, visited_states = iddfs(start,15)

    print(f"Total states visited: {visited_states}")

    if result is not None:
        print("Solution found!")
print("Moves:", ' '.join(result))
print("Number of moves:", len(result))
print("1BM23CS347 Sujan G E\n")
current_state = start        for i, move in
enumerate(result, 1):
        current_state = move_tile(current_state, move)            print(f"Move
{i}: {move}")            print_state(current_state)    else:        print("No
solution exists for the given start state or max depth reached.") else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

```
Enter start state (e.g., 724506831): 123450678 Start
state:
1 2 3
```

4 5
6 7 8

Total states visited: 9504 Solution found!
Moves: D L L U R D R U L L D R R
Number of moves: 13
1BM23CS347 Sujan G E

Move 1: D
1 2 3
4 5 8
6 7

Move 2: L
1 2 3
4 5 8
6 7

Move 3: L
1 2 3
4 5 8
 6 7

Move 4: U
1 2 3
 5 8
4 6 7

Move 5: R
1 2 3
5 8
4 6 7

Move 6: D
1 2 3
5 6 8
4 7

Move 7: R
1 2 3
5 6 8

4 7

Move 8: U
1 2 3
5 6
4 7 8

Move 9: L
1 2 3
5 6
4 7 8

Move 10: L
1 2 3
 5 6
4 7 8

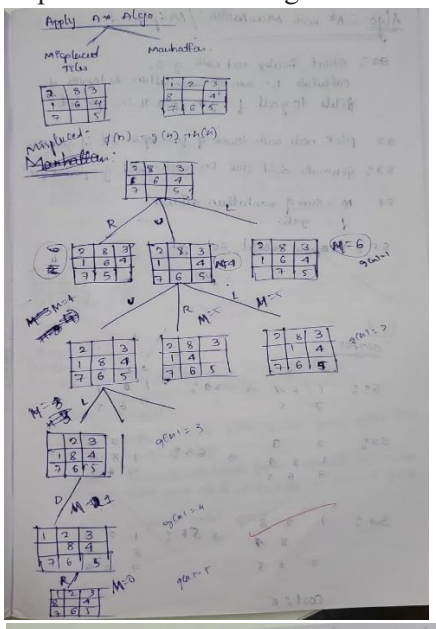
Move 11: D
1 2 3
4 5 6
 7 8

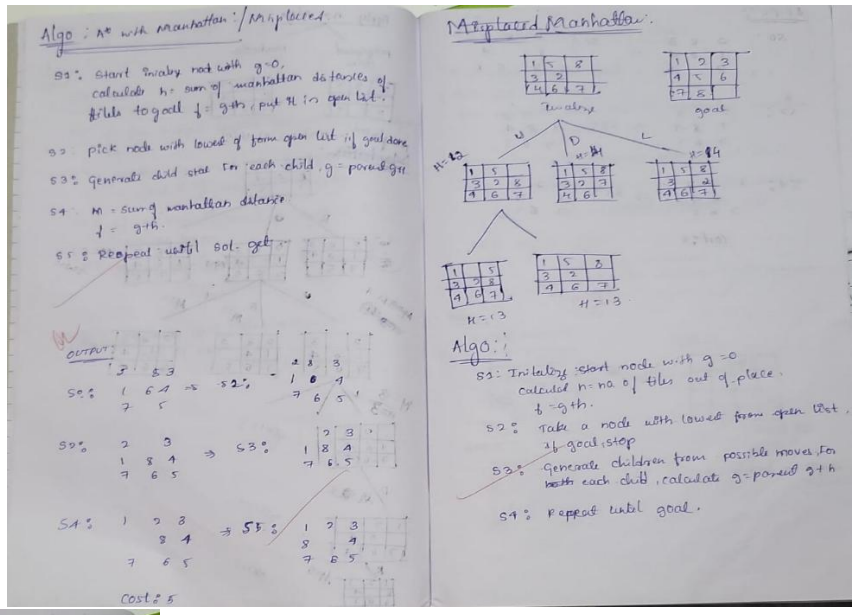
Move 12: R
1 2 3
4 5 6
7 8

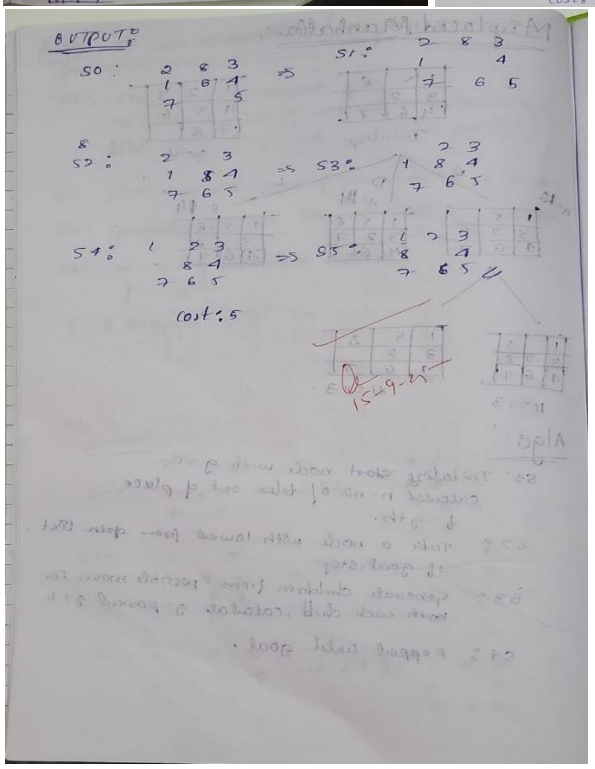
Move 13: R
1 2 3
4 5 6
7 8

Program 3 :

Implement A* search algorithm







Code:

a) Misplaced Tiles

```
import heapq
import time
```

```
class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):
        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def heuristic(self):
        misplaced = 0
        for i, tile in enumerate(self.board):
            if tile != 0 and tile != self.goal[i]:
                misplaced += 1
        return misplaced

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y - 1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)
                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board), self.goal, self.path + move, self.cost +
1))
        return neighbors

def a_star(start, goal):
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
```

```
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map, current_state.board)

        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

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

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

result = a_star(initial_state, final_state)
if result:
    solution_boards, solution_moves = result
    print("Step-by-step solution:\n")
    print("Name:Sujan G E ")
    print("USN:1BM23CS347")
    print()
    for step_num, board in enumerate(solution_boards):
        moves_so_far = "".join(solution_moves[:step_num])
        print(f"Step {step_num}: Moves: {moves_so_far}")
        print_board(board)
        time.sleep(1)
else:
    print("No solution found.")
```

Output:

```
Step-by-step solution:

Name:Sujan G E
USN:1BM23CS347

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

Step 1: Moves: U
2 8 3
1   4
7 6 5

Step 2: Moves: UU
2   3
1 8 4
7 6 5

Step 3: Moves: UUL
  2 3
1 8 4
7 6 5

Step 4: Moves: UULD
1 2 3
  8 4
7 6 5

Step 5: Moves: UULDR
1 2 3
8   4
7 6 5
```

b) Manhattan Distance import
heapq
import heapq
import time

```
class PuzzleState:
    def __init__(self, board, goal, path="", cost=0):
        self.board = board
        self.goal = goal
        self.path = path
        self.cost = cost
        self.zero_pos = self.board.index(0)
        self.size = int(len(board) ** 0.5)

    def __lt__(self, other):

        return (self.cost + self.heuristic()) < (other.cost + other.heuristic())

    def heuristic(self):

        distance = 0
        for i, tile in enumerate(self.board):
            if tile != 0:
                goal_pos = self.goal.index(tile)
                distance += abs(i // self.size - goal_pos // self.size) + abs(i % self.size - goal_pos % self.size)
        return distance

    def get_neighbors(self):
        neighbors = []
        x, y = divmod(self.zero_pos, self.size)
        moves = {'U': (x - 1, y), 'D': (x + 1, y), 'L': (x, y - 1), 'R': (x, y + 1)}

        for move, (nx, ny) in moves.items():
            if 0 <= nx < self.size and 0 <= ny < self.size:
                new_zero_pos = nx * self.size + ny
                new_board = list(self.board)

                new_board[self.zero_pos], new_board[new_zero_pos] = new_board[new_zero_pos],
new_board[self.zero_pos]
                neighbors.append(PuzzleState(tuple(new_board), self.goal, self.path + move, self.cost + 1))
        return neighbors

def a_star(start, goal):
```

```
    start_state = PuzzleState(start, goal)
    frontier = []
    heapq.heappush(frontier, start_state)
    explored = set()
    parent_map = {start_state.board: None}
    move_map = {start_state.board: ""}

    while frontier:
        current_state = heapq.heappop(frontier)

        if current_state.board == goal:
            return reconstruct_path(parent_map, move_map, current_state.board)

        explored.add(current_state.board)

        for neighbor in current_state.get_neighbors():
            if neighbor.board not in explored and neighbor.board not in parent_map:
                parent_map[neighbor.board] = current_state.board
                move_map[neighbor.board] = neighbor.path[-1]
                heapq.heappush(frontier, neighbor)

    return None

def reconstruct_path(parent_map, move_map, state):
    path_boards = []
    path_moves = []
    while parent_map[state] is not None:
        path_boards.append(state)
        path_moves.append(move_map[state])
        state = parent_map[state]
    path_boards.append(state)
    path_boards.reverse()
    path_moves.reverse()
    return path_boards, path_moves

def print_board(board):
    size = int(len(board) ** 0.5)
    for i in range(size):
        row = board[i*size:(i+1)*size]
        print(" ".join(str(x) if x != 0 else " " for x in row))
    print()

if __name__ == "__main__":
    initial_state = (1, 5, 8,
```

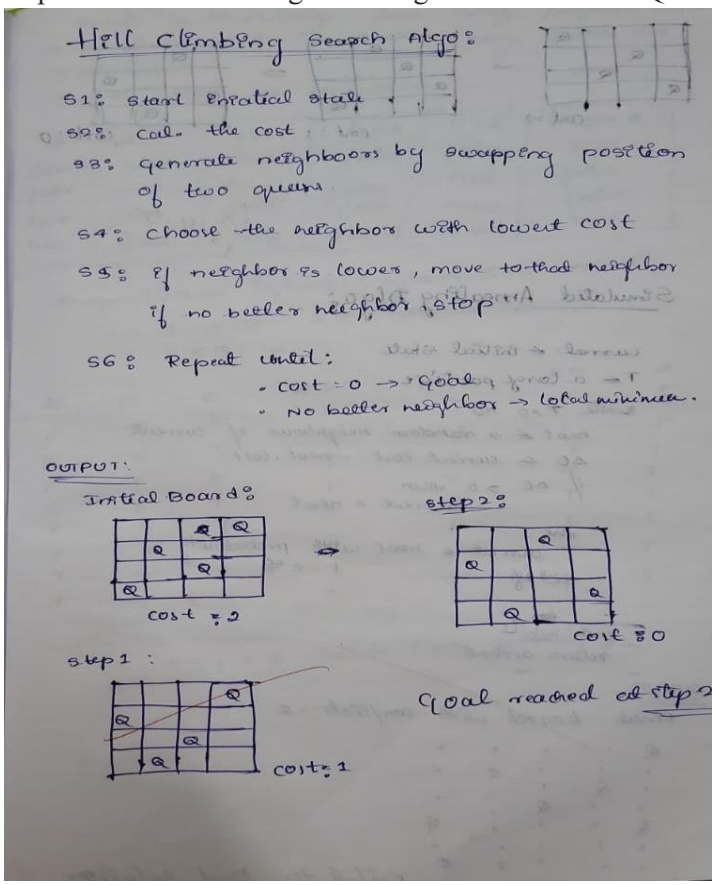
```
                3, 2, 0,
                4, 6, 7)

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

    result = a_star(initial_state, final_state)
    if result:
        solution_boards, solution_moves = result
        print("Step-by-step solution:\n")
        for step_num, board in enumerate(solution_boards):
            moves_so_far = "".join(solution_moves[:step_num])
            print(f"Step {step_num}: Moves: {moves_so_far}")
            print_board(board)
            time.sleep(1)
    else:
        print("Name:Sujan G E\nUSN:1BM23CS347")
        print("No solution found.")
```

Output:

```
Name:Sujan G E
No solution found.
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem



Code:

```
 \def
calculate_conflicts(boar
d):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1,
n):
            if board[i] ==
board[j] or abs(board[i]
- board[j]) == j - i:
                conflicts += 1
    return conflicts
```

```
def print_board(board):
    n = len(board)
    for row in range(n):
        line = ['Q' if col ==
board[row] else '.' for
col in range(n)]
        print(' '.join(line))
    print()

def
hill_climbing_step_by_s
tep(board):
    n = len(board)
    current_state =
board[:]
    current_conflicts =
calculate_conflicts(curre
nt_state)

    step = 0
    print("Name:sujan g
e\nUSN:1BM23CS347\
n")
    print(f"Initial board
with conflicts =
{current_conflicts}:")

print_board(current_stat
e)

    while
current_conflicts > 0:
        step += 1
        print(f"Step
{step}:")
        best_state =
current_state[:]
        best_conflicts =
current_conflicts

        for row in range(n):
```

```
            original_col =
current_state[row]
            for col in
range(n):
                if col !=
original_col:

current_state[row] = col
                    conflicts =
calculate_conflicts(curre
nt_state)

                    if conflicts
< best_conflicts:

best_conflicts =
conflicts
                        best_state
= current_state[:]


current_state[row] =
original_col

        if best_conflicts ==
current_conflicts:
            print("No better
neighbor found, stuck at
local optimum.")
            break

        current_state =
best_state
        current_conflicts =
best_conflicts

        print(f"Board with
conflicts =
{current_conflicts}:")

print_board(current_stat
e)
```

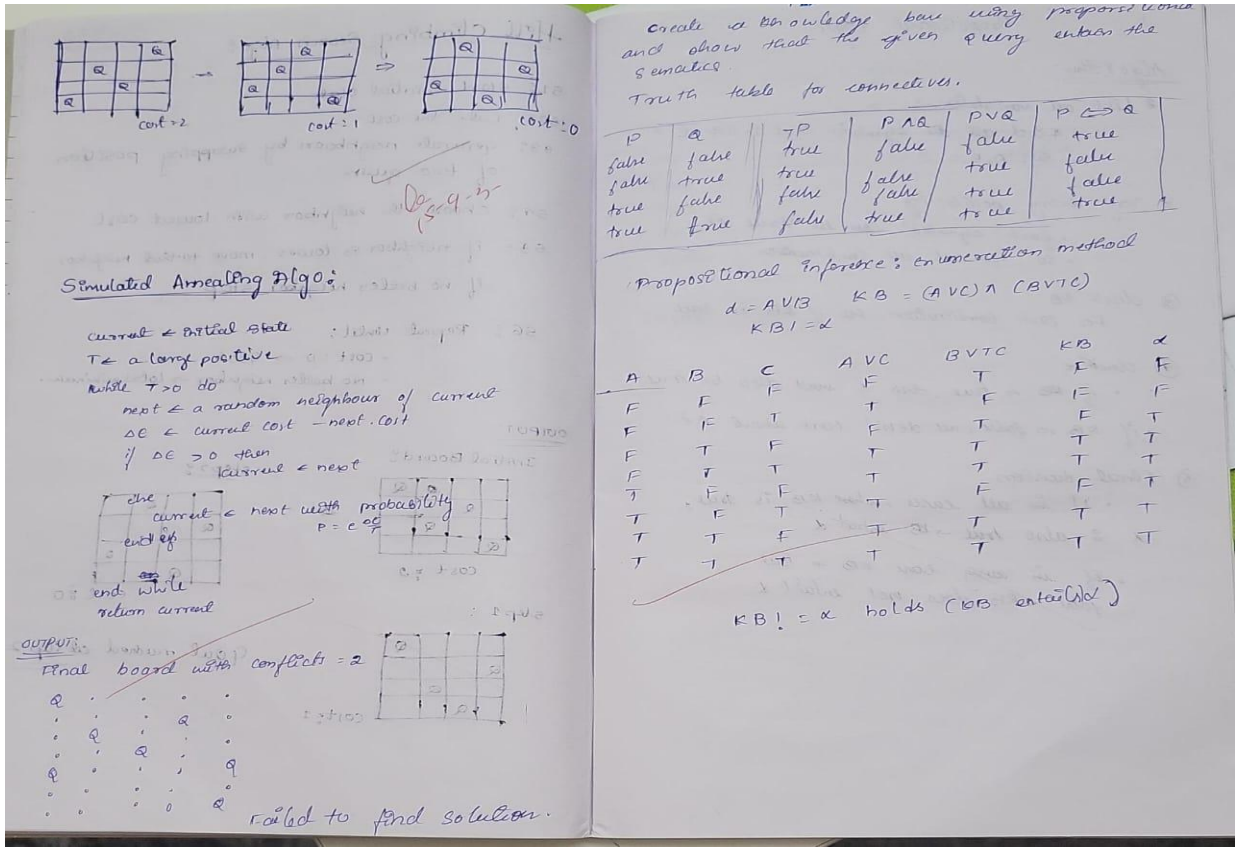
```
    if current_conflicts ==
0:
        print("Solution
found!")
    else:
        print("No solution
found.")
    return current_state

initial_board = [3, 0, 1,
2]
solution =
hill_climbing_step_by_s
tep(initial_board)
```

Output:

```
Name:sujan g e
USN:1BM23CS347

Initial board with conflicts = 4:
. . . Q
Q . . .
. Q . .
. . Q .

Step 1:
Board with conflicts = 2:
. . . Q
Q . . .
Q . . .
. . Q .

Step 2:
Board with conflicts = 1:
. . . Q
. Q . .
Q . . .
. . Q .

Step 3:
No better neighbor found, stuck at local optimum.
No solution found.
```

Program 5 :
Simulated Annealing to Solve 8-Queens problem

Algorithm:



Code:

```
import random
import math

def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts
```

```
def print_board(board):
    n = len(board)
    for row in range(n):
        line = ['Q' if col == board[row] else '.' for col in range(n)]
        print(' '.join(line))
    print()

def simulated_annealing(n=8, max_iter=10000, initial_temp=100, cooling_rate=0.95):
    current_state = [random.randint(0, n - 1) for _ in range(n)]
    current_conflicts = calculate_conflicts(current_state)
    temperature = initial_temp
    iteration = 0

    while current_conflicts > 0 and iteration < max_iter and temperature > 0.1:
        iteration += 1
        neighbor = current_state[:]
        row = random.randint(0, n - 1)
        new_col = random.randint(0, n - 1)
        while new_col == neighbor[row]:
            new_col = random.randint(0, n - 1)
        neighbor[row] = new_col

        neighbor_conflicts = calculate_conflicts(neighbor)
        delta = neighbor_conflicts - current_conflicts

        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / temperature):
            current_state = neighbor
            current_conflicts = neighbor_conflicts

        temperature *= cooling_rate

    return current_state, current_conflicts

solution, conflicts = simulated_annealing(n=8)
print("Final board with conflicts =", conflicts)
print("Name:SUJAN G E\nUSN:1BM23CS347\n")
print_board(solution)

if conflicts == 0:
    print("Solution found!")
else:
    print("Failed to find a solution.")
```

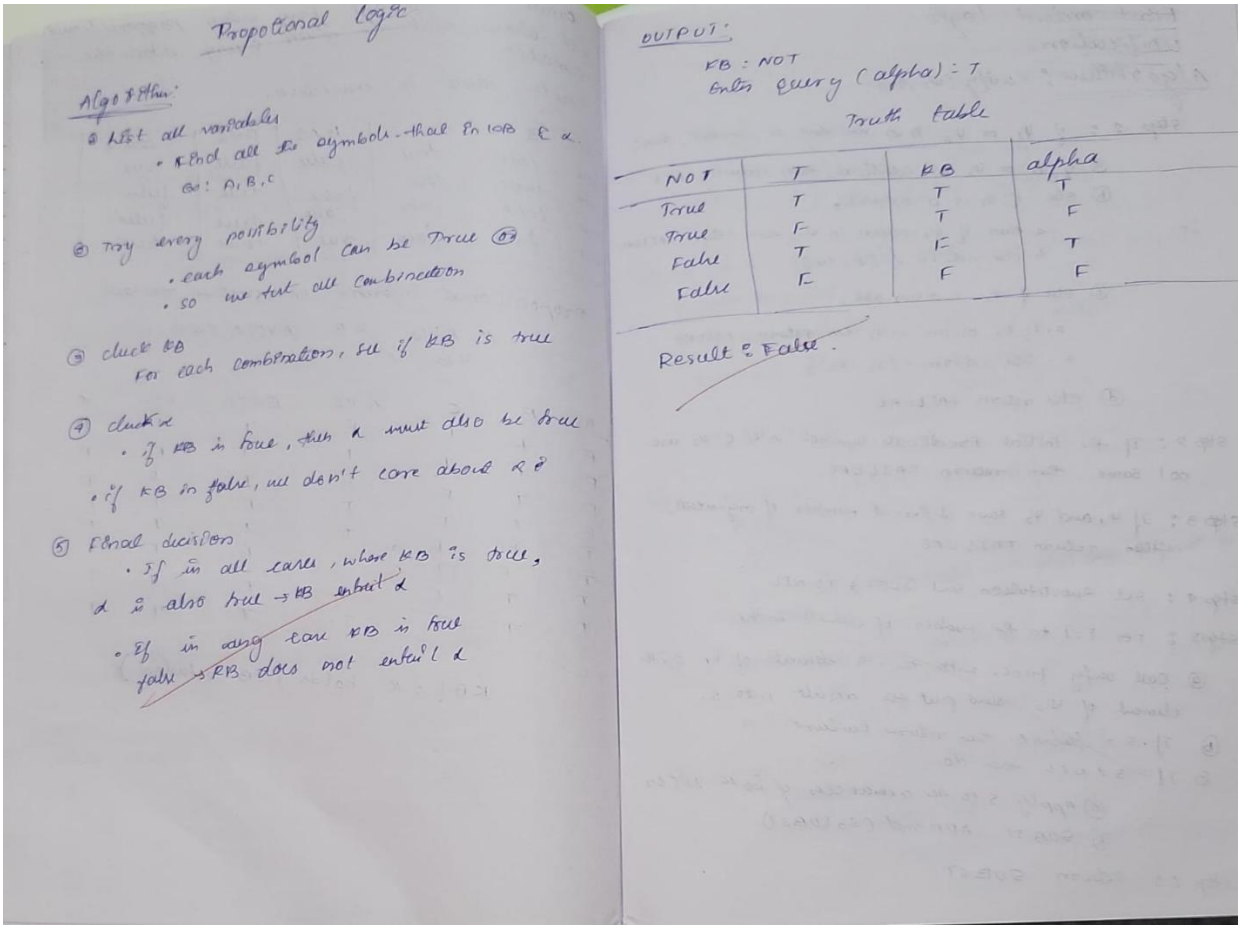
Output:

```
Final board with conflicts = 2
Name:SUJAN G E
USN:1BM23CS347

Q . . . . . . .
. . . . Q . . .
. Q . . . . . .
. . . . . . . Q
. . . . . Q . .
. . . Q . . . .
. . . . . . Q .
Q . . . . . . .

Failed to find a solution.
```

Program 6

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

Algorithm:



Code:

```
import itertools
import pandas as pd
import re

def replace_implications(expr):
    """
    Replace every X => Y with (not X or Y).
    This uses regex with a callback to avoid partial string overwrites.
    """
    # Pattern: capture left side and right side around =>
    # Made more flexible to handle various expressions
    pattern = r'([^=><]+?)\s*=>\s*([^=><]+?)(?=\s|$|[&|])'
    while re.search(pattern, expr):
        expr = re.sub(pattern,
                      lambda m: f"(not {m.group(1).strip()} or {m.group(2).strip()})",
                      expr,
                      count=1)
    return expr

def pl_true(sentence, model):
    expr = sentence.strip()
    expr = expr.replace("<=>", "==")
    expr = replace_implications(expr)

    # Replace propositional symbols with their truth values safely
    for sym, val in model.items():
        expr = re.sub(rf'\b{sym}\b', str(val), expr)

    # Clean up spacing and add proper spacing for boolean operators
    expr = re.sub(r'\s+', ' ', expr)  # Remove extra spaces
    expr = expr.replace(" and ", " and ").replace(" or ", " or ").replace(" not ", " not ")

    return eval(expr)

def get_symbols(KB, alpha):
    symbols = set()
    for sentence in KB + [alpha]:
        # Find all alphabetic tokens (propositional variables)
        for token in re.findall(r'\b[A-Za-z]+\b', sentence):
            if token not in ['and', 'or', 'not']:  # Exclude boolean operators
```

```
            symbols.add(token)
    return sorted(list(symbols))

def tt_entails(KB, alpha):
    symbols = get_symbols(KB, alpha)
    rows = []
    entails = True

    for values in itertools.product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))

        try:
            kb_val = all(pl_true(sentence, model) for sentence in KB)
            alpha_val = pl_true(alpha, model)

            rows.append({**model, "KB": kb_val, "alpha": alpha_val})

            if kb_val and not alpha_val:
                entails = False
        except Exception as e:
            print(f'Error evaluating with model {model}: {e}")
            return False

    df = pd.DataFrame(rows)

    # Create a beautiful formatted table
    print("\n" + "="*50)
    print("               TRUTH TABLE")
    print("="*50)

    # Get column widths for proper alignment
    col_widths = {}
    for col in df.columns:
        col_widths[col] = max(len(str(col)), df[col].astype(str).str.len().max())

    # Calculate total table width
    table_width = sum(col_widths.values()) + len(df.columns) * 3 - 1

    # Print top border
    print(" ┌" + "─" * table_width + "┐ ")

    # Print header
```

```
    header = " │ "
    for col in df.columns:
        header += f" {col:^{col_widths[col]}} │ "
    print(header)

    # Print separator
    separator = " ├"
    for col in df.columns:
        separator += "─" * (col_widths[col] + 2) + "┼"
    separator = separator[:-1] + "┤ "
    print(separator)

    # Print rows
    for _, row in df.iterrows():
        row_str = " │ "
        for col in df.columns:
            value = str(row[col])
            row_str += f" {value:^{col_widths[col]}} │ "
        print(row_str)

    # Print bottom border
    print(" └" + "─" * table_width + "┘ ")

    # Print result with styling
    print("\n" + "="*50)
    result_text = f"KB ENTAILS ALPHA: {'✓ YES' if entails else '✗ NO'}"
    print(f"{result_text:^50}")
    print("="*50)
    return entails

# --- Interactive input ---
print("Name :sujan g e \nUSN:1BM23CS347")
print("Enter Knowledge Base (KB) sentences, separated by commas.")
print("Use symbols like A, B, C and operators: and, or, not, =>, <=>")
kb_input = input("KB: ").strip()
KB = [x.strip() for x in kb_input.split(",")]
alpha = input("Enter query (alpha): ").strip()
result = tt_entails(KB, alpha)
print(f"Result: {result}")
#
```

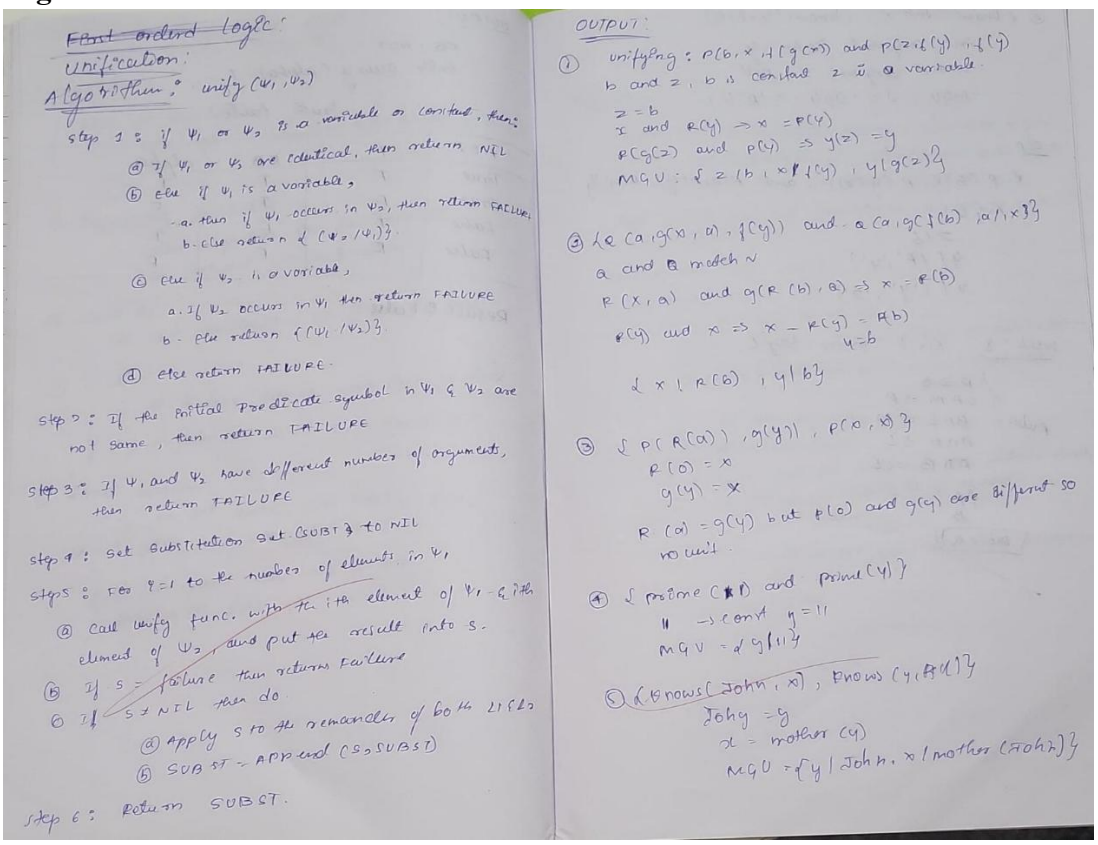
Output:

```
Name :sujan g e
USN:1BM23CS347
Enter Knowledge Base (KB) sentences, separated by commas.
Use symbols like A, B, C and operators: and, or, not, =>, <=>
KB: NOT
Enter query (alpha): T

================================================
                TRUTH TABLE
================================================
┌─────────┬─────────┬─────────┬─────────┐
│   NOT   │    T    │   KB    │  alpha  │
├─────────┼─────────┼─────────┼─────────┤
│ True    │ True    │ True    │ True    │
│ True    │ False   │ True    │ False   │
│ False   │ True    │ False   │ True    │
│ False   │ False   │ False   │ False   │
└─────────┴─────────┴─────────┴─────────┘

================================================
             KB ENTAILS ALPHA: ✗ NO
================================================
Result: False
```

Program 7

Implement unification in first order logic

Algorithm:



Code:

```
import re
from collections import namedtuple

Var = namedtuple('Var', ['name'])
Const = namedtuple('Const', ['name'])
Func = namedtuple('Func', ['name', 'args'])

def parse(s):
    s = s.strip()
    if '(' in s:
        n, rest = s[:s.index('(')], s[s.index('(')+1:-1]
        args = []
        depth = 0; current = []
        for c in rest + ',':
            if c == ',' and depth == 0:
                args.append(''.join(current).strip())
                current = []
            else:
                if c == '(': depth += 1
                elif c == ')': depth -= 1
                current.append(c)
        return Func(n, [parse(a) for a in args])
    if re.fullmatch(r'[a-z][a-z0-9]*', s): return Var(s)
    return Const(s)

def occurs(v, x, s):
    x = subst(x, s)
    if v == x: return True
    if isinstance(x, Func):
        return any(occurs(v, a, s) for a in x.args)
    return False

def subst(t, s):
    while isinstance(t, Var) and t.name in s:
        t = s[t.name]
```

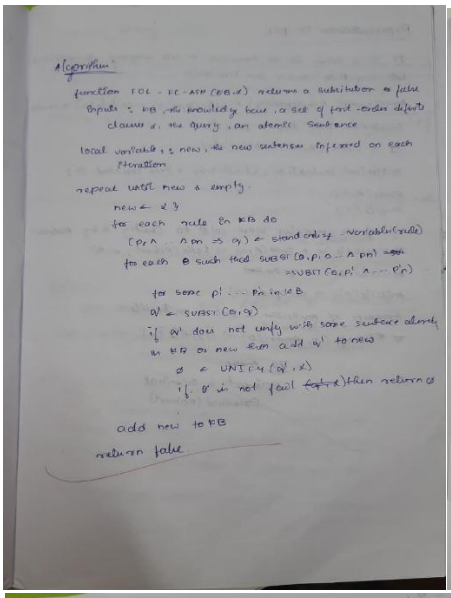
```
    if isinstance(t, Func):
        return Func(t.name, [subst(a, s) for a in t.args])
    return t

def unify(t1, t2, s=None):
    if s is None: s = {}
    t1, t2 = subst(t1, s), subst(t2, s)
    if t1 == t2: return s
    if isinstance(t1, Var):
        if occurs(t1, t2, s): return None
        s[t1.name] = t2
        return s
    if isinstance(t2, Var):
        if occurs(t2, t1, s): return None
        s[t2.name] = t1
        return s
    if isinstance(t1, Func) and isinstance(t2, Func):
        if t1.name != t2.name or len(t1.args) != len(t2.args): return None
        for a1, a2 in zip(t1.args, t2.args):
            s = unify(a1, a2, s)
            if s is None: return None
        return s
    if isinstance(t1, Const) and isinstance(t2, Const) and t1.name == t2.name:
        return s
    return None

def to_str(t):
    if isinstance(t, Var) or isinstance(t, Const):
        return t.name
    return f"{t.name}({', '.join(to_str(a) for a in t.args)})"

def show_subs(s):
    if s is None:
        print("Unification failed.")
    elif not s:
        print("No substitution needed.")
    else:
        for k,v in s.items():
            print(f"{k} = {to_str(v)}")
print("Name:Sujan g e\nUSN:1BM23CS347\n\n")
tests = [
    ("p(b,X,f(g(Z)))", "p(z,f(Y),f(Y))"),
```

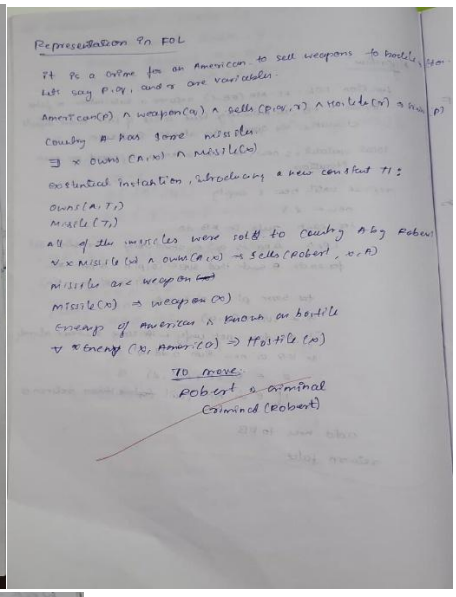
```
    ("Q(a,g(x,a),f(y))", "Q(a,g(f(b),a),x)"),
    ("p(f(a),g(Y))", "p(X,X)"),
    ("prime(11)", "prime(y)"),
    ("knows(John,x)", "knows(y,mother(y))"),
    ("knows(John,x)", "knows(y,Bill)")
]

for e1, e2 in tests:
    print(f"Unifying: {e1} and {e2}")
    s = unify(parse(e1), parse(e2))
    show_subs(s)
    print('-'*40)
```

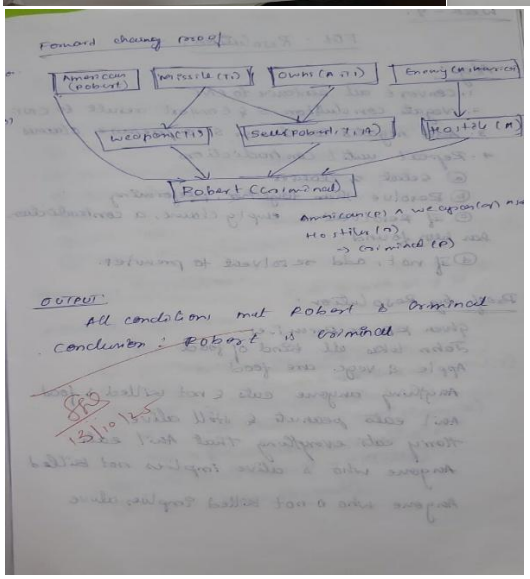
Output:

```
Name:Sujan g e
USN:1BM23CS347


Unifying: p(b,X,f(g(Z))) and p(z,f(Y),f(Y))
Unification failed.
----------------------------------------
Unifying: Q(a,g(x,a),f(y)) and Q(a,g(f(b),a),x)
x = f(b)
y = b
----------------------------------------
Unifying: p(f(a),g(Y)) and p(X,X)
Unification failed.
----------------------------------------
Unifying: prime(11) and prime(y)
y = 11
----------------------------------------
Unifying: knows(John,x) and knows(y,mother(y))
y = John
x = mother(John)
----------------------------------------
Unifying: knows(John,x) and knows(y,Bill)
y = John
x = Bill
----------------------------------------
```

FOL:


Algorithm:
function FOL - FC - ASK (KB, α) returns a substitution or false
Inputs : KB, the knowledge base, a set of first-order definite clauses
α, the query, an atomic sentence
local variables : new, the new sentences inferred on each iteration
repeat until new is empty
new ← { }
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
ø ← UNIFY(q', α)
if ø is not fail then return ø
add new to KB
return false


Representation in FOL
It is a crime for an American to sell weapons to hostile nations
Let say p, q, and r are variables
American(p) ∧ weapon(q) ∧ sells(p, q, r) ∧ Hostile(r) ⇒ Criminal(p)
Country A has some missiles
∃ x Owns(A, x) ∧ Missile(x)
Existential instantiation, introducing a new constant T1:
Owns(A, T1)
Missile(T1)
All of the missiles were sold to country A by Robert
∀ x Missile(x) ∧ Owns(A, x) ⇒ Sells(Robert, x, A)
Missiles are weapons
Missile(x) ⇒ Weapon(x)
Enemy of America is known as hostile
∀ x Enemy(x, America) ⇒ Hostile(x)
TO prove:
Robert is criminal
Criminal(Robert)


Forward chaining proof
American (Robert)
Missile (T1)
Owns (A, T1)
Enemy (A, America)
Weapon (T1)
Sells (Robert, T1, A)
Hostile (A)
Robert (Criminal)
American(p) ∧ weapon(q) ∧
Hostile(r)
⇒ Criminal(p)
OUTPUT:
All conditions met Robert is Criminal
Conclusion : Robert is Criminal

Code :

```
class Person:
    def __init__(self, name, nationality):
        self.name = name
        self.nationality = nationality

class Country:
    def __init__(self, name, hostile_to=None):
        self.name = name
        self.hostile_to = hostile_to if hostile_to else []

class Weapon:
    def __init__(self, name, owner=None):
        self.name = name
        self.owner = owner

robert = Person("Robert", "American")
countryA = Country("CountryA", hostile_to=["America"])

missiles = [
    Weapon("Missile1", owner=countryA),
    Weapon("Missile2", owner=countryA),
]

def sold_by(person, weapon):
    return weapon.owner == countryA and person == robert

def is_hostile(buyer, seller_country_name):
    return seller_country_name in buyer.hostile_to

def is_weapon(item):
    return isinstance(item, Weapon)

def prove_robert_criminal(person):
    print(f"Step 1: Check if {person.name} is American.")
    if person.nationality == "American":
        print(f"  {person.name} is American.")
    else:
        print(f"  {person.name} is NOT American. Proof ends here.")
        return False

    print(f"Step 2: Check if CountryA is hostile to America.")
    if is_hostile(countryA, "America"):
```

```
        print(f"  CountryA is hostile to America.")
    else:
        print(f"  CountryA is NOT hostile to America. Proof ends here.")
        return False

    print(f"Step 3: Check missiles owned by CountryA.")
    for missile in missiles:
        print(f"  Missile '{missile.name}' owned by {missile.owner.name}")

    print(f"Step 4: Check if {person.name} sold these missiles.")
    for missile in missiles:
        if sold_by(person, missile):
            print(f"  {person.name} sold {missile.name}.")
        else:
            print(f"  {person.name} did NOT sell {missile.name}. Proof ends here.")
            return False

    print(f"Step 5: Confirm missiles are weapons.")
    for missile in missiles:
        if is_weapon(missile):
            print(f"  {missile.name} is a weapon.")
        else:
            print(f"  {missile.name} is NOT a weapon. Proof ends here.")
            return False

    print(f"Step 6: Apply the law: American selling weapons to hostile nations is criminal.")
    print(f"Step 7: All conditions met, so {person.name} is criminal.")
    return True

if prove_robert_criminal(robert):
    print("\nConclusion: Robert is criminal.")
else:
    print("\nConclusion: Robert is NOT criminal.")
```

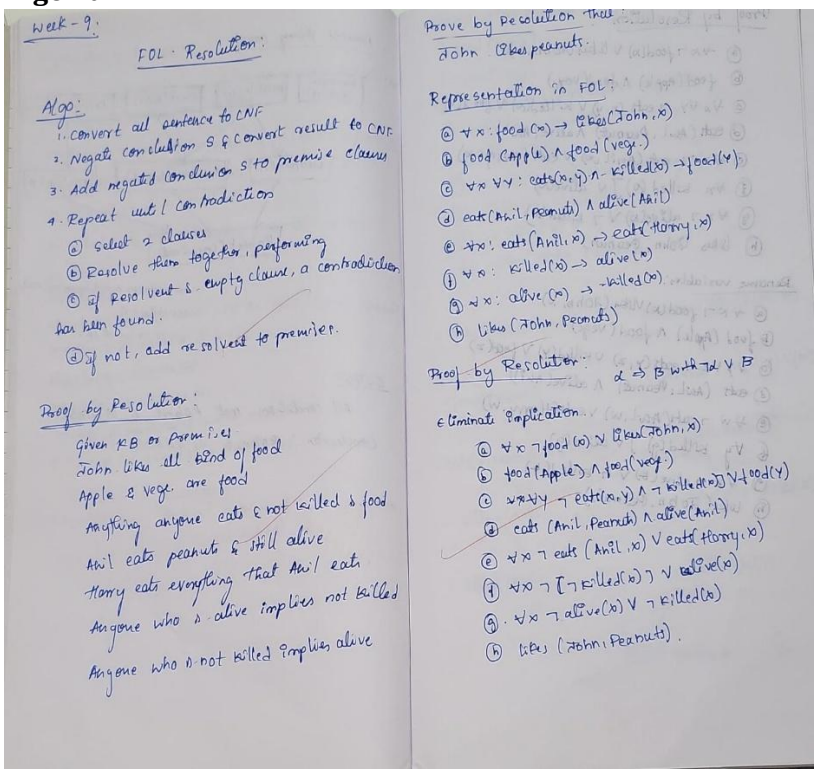
Output:

```
Step 1: Check if Robert is American.
  Robert is American.
Step 2: Check if CountryA is hostile to America.
  CountryA is hostile to America.
Step 3: Check missiles owned by CountryA.
  Missile 'Missile1' owned by CountryA
  Missile 'Missile2' owned by CountryA
Step 4: Check if Robert sold these missiles.
  Robert sold Missile1.
  Robert sold Missile2.
Step 5: Confirm missiles are weapons.
  Missile1 is a weapon.
  Missile2 is a weapon.
Step 6: Apply the law: American selling weapons to hostile nations is criminal.
Step 7: All conditions met, so Robert is criminal.

Conclusion: Robert is criminal.
```

Program 9

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

Algorithm:



Week - 9:

FOL - Resolution:

Algo:

1. Convert all sentence to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to premise clauses
4. Repeat until contradiction
   a. Select 2 clauses
   b. Resolve them together, performing
   c. If Resolvent is empty clause, a contradiction has been found.
   d. If not, add resolvent to premises.

Proof by Resolution:

Given KB or Premises
John likes all kind of food
Apple & Vege. are food
Anything anyone eats & not killed is food
Anil eats peanuts & still alive
Harry eats everything that Anil eats
Anyone who is alive implies not killed
Anyone who is not killed implies alive

Prove by Resolution that:
John likes peanuts.

Representation in FOL:

a. $\forall x : food(x) \rightarrow likes(John, x)$
b. $food(Apple) \wedge food(Vegy.)$
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. $\forall x : killed(x) \rightarrow alive(x)$
g. $\forall x : alive(x) \rightarrow \neg killed(x)$
h. $likes(John, Peanuts)$

Proof by Resolution:

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

Eliminate Implication.

a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(Vegy.)$
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. $likes(John, Peanuts)$.

Code:

```
from typing import List, Set

class Predicate:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __eq__(self, other):
        return self.name == other.name and self.args == other.args
    def __hash__(self):
        return hash((self.name, tuple(self.args)))
    def __repr__(self):
        return f"{self.name}({', '.join(self.args)})"
```

```
def negate(pred):
    if pred.name.startswith("~"):
        return Predicate(pred.name[1:], pred.args)
    else:
        return Predicate("~" + pred.name, pred.args)

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x[0].islower():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y[0].islower():
        return unify_var(y, x, subst)
    elif isinstance(x, Predicate) and isinstance(y, Predicate):
        if x.name != y.name or len(x.args) != len(y.args):
            return None
        for a, b in zip(x.args, y.args):
            subst = unify(a, b, subst)
        return subst
    else:
        return None

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)
    else:
        subst[var] = x
        return subst

def resolution(kb: List[Set[Predicate]], query: Predicate):
    clauses = kb.copy()
    clauses.append({negate(query)})
    print("\nInitial Clauses:")
    for c in clauses:
        print(c)
    while True:
        new = []
        n = len(clauses)
```

```
        for i in range(n):
            for j in range(i + 1, n):
                resolvents = resolve(clauses[i], clauses[j])
                if set() in resolvents:
                    print("\nDerived empty clause {}. Hence, Query is PROVED.")
                    return True
                for res in resolvents:
                    if res not in clauses and res not in new:
                        new.append(res)
        if not new:
            print("\nNo new clauses derived. Query CANNOT be proved.")
            return False
        for c in new:
            clauses.append(c)

def resolve(ci: Set[Predicate], cj: Set[Predicate]):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di.name == "~" + dj.name or "~" + di.name == dj.name:
                subst = unify(di, negate(dj), {})
                if subst is not None:
                    new_clause = (ci.union(cj) - {di, dj})
                    new_clause = {apply_substitution(p, subst) for p in new_clause}
                    resolvents.append(new_clause)
    return resolvents

def apply_substitution(pred, subst):
    new_args = [subst.get(arg, arg) for arg in pred.args]
    return Predicate(pred.name, new_args)

KB = [
    {Predicate("~Food", ["x"]), Predicate("Likes", ["John", "x"])},
    {Predicate("Food", ["Apple"])},
    {Predicate("Food", ["Vegetable"])},
    {Predicate("~Eats", ["x", "y"]), Predicate("~Killed", ["x"]), Predicate("Food", ["y"])},
    {Predicate("Eats", ["Anil", "Peanut"])},
    {Predicate("Alive", ["Anil"])},
    {Predicate("~Eats", ["Anil", "x"]), Predicate("Eats", ["Harry", "x"])},
    {Predicate("~Alive", ["x"]), Predicate("~Killed", ["x"])},
    {Predicate("Killed", ["x"]), Predicate("Alive", ["x"])},
]
```

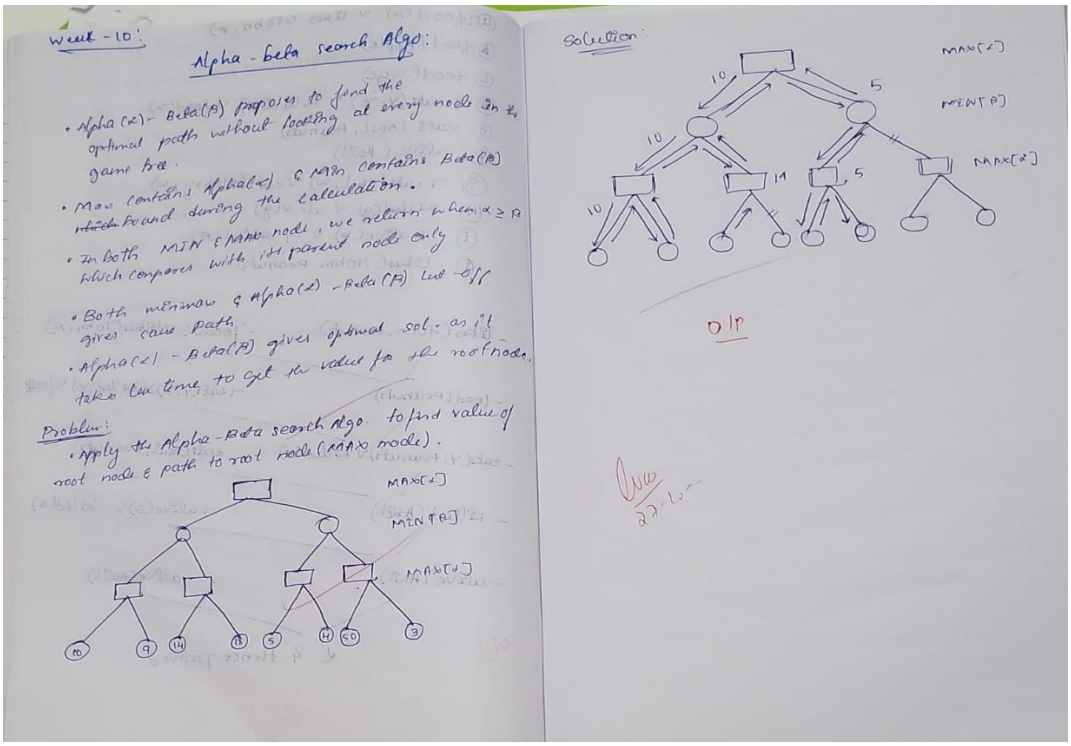
```
query = Predicate("Likes", ["John", "Peanut"])
print("Name:SUJAN G E\nUSN:1BM23CS347\n")
print("RESOLUTION PROCESS ")
proved = resolution(KB, query)
print("\nRESULT:", "Query is TRUE (proved by resolution)" if proved else "Query is FALSE (not provable)")
```

Output:

```
Name:SUJAN G E
USN:1BM23CS347

RESOLUTION PROCESS

Initial Clauses:
{Likes(John, x), ~Food(x)}
{Food(Apple)}
{Food(Vegetable)}
{~Killed(x), Food(y), ~Eats(x, y)}
{Eats(Anil, Peanut)}
{Alive(Anil)}
{Eats(Harry, x), ~Eats(Anil, x)}
{~Killed(x), ~Alive(x)}
{Killed(x), Alive(x)}
{~Likes(John, Peanut)}

Derived empty clause {}. Hence, Query is PROVED.

RESULT: Query is TRUE (proved by resolution)
```

Program 10 Implement
Alpha-Beta Pruning.

Algorithm:



Code :

```
import math

def
alpha_beta(d
epth,
node_index,
maximizing_
player,
values, alpha,
beta,
max_depth):
    if depth ==
max_depth:
```

```
        return
values[node_
index]

    if
maximizing_
player:
        best = -
math.inf
        for i in
range(2):
            val =
alpha_beta(d
epth + 1,
node_index *
2 + i, False,
values, alpha,
beta,
max_depth)
            best =
max(best,
val)
            alpha
= max(alpha,
best)
            if beta
<= alpha:

print(f"Prune
d at depth
{depth},
node
{node_index
}, α={alpha},
β={beta}")

break
        return
best
    else:
        best =
math.inf
```

```
        for i in
range(2):
            val =
alpha_beta(d
epth + 1,
node_index *
2 + i, True,
values, alpha,
beta,
max_depth)
            best =
min(best, val)
            beta =
min(beta,
best)
            if beta
<= alpha:

print(f"Prune
d at depth
{depth},
node
{node_index
}, α={alpha},
β={beta}")

break
        return
best


values = [10,
9, 14, 18, 5,
4, 50, 3]
max_depth =
3
print("Name:
SUJAN G
E\nUSN:1B
M23CS347\n
")
```

```
print("ALPHA–BETA PRUNING PROCESS\n")
optimal_value = alpha_beta(0, 0, True, values, -math.inf, math.inf, max_depth)
print("\nOptimal value (Root Node):", optimal_value)
```

Output:

```
Name:SUJAN G E
USN:1BM23CS347

ALPHA–BETA PRUNING PROCESS

Pruned at depth 2, node 1, α=14, β=10
Pruned at depth 1, node 1, α=10, β=5

Optimal value (Root Node): 10
```