VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Sushravya R (1WA23CS004)

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 **Sushravya R (1WA23CS004),** 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.

Mamatha M Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

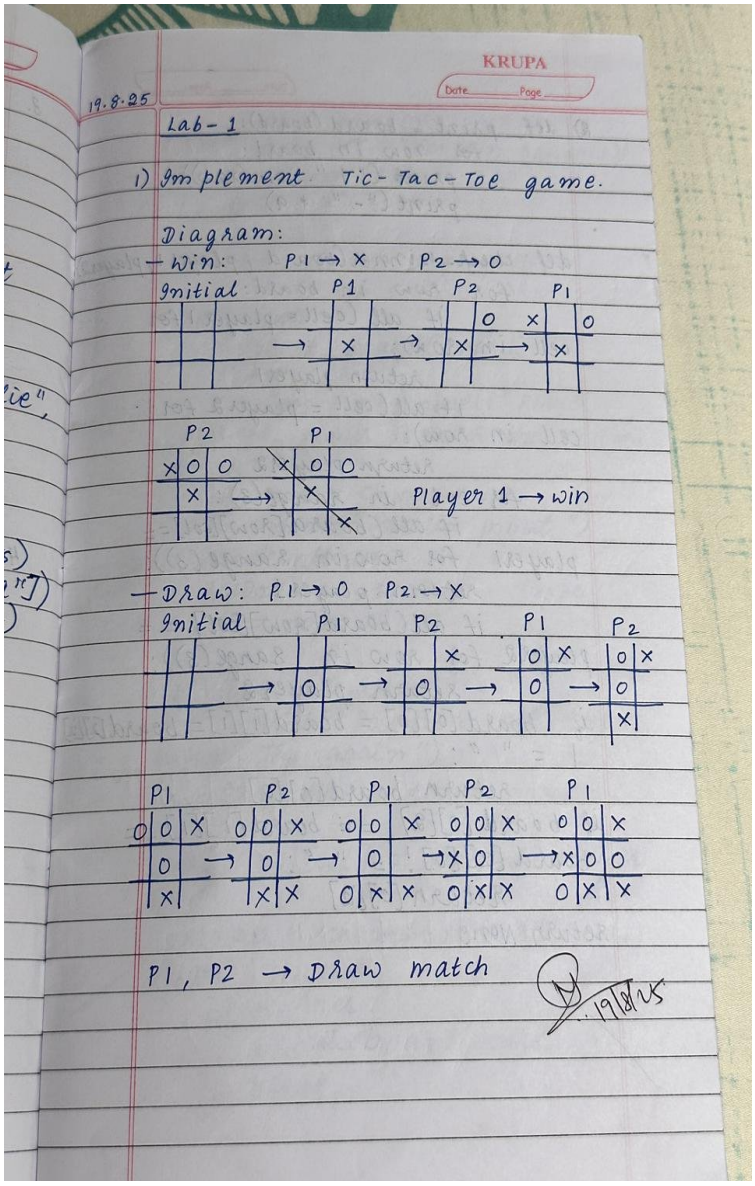
Github Link:

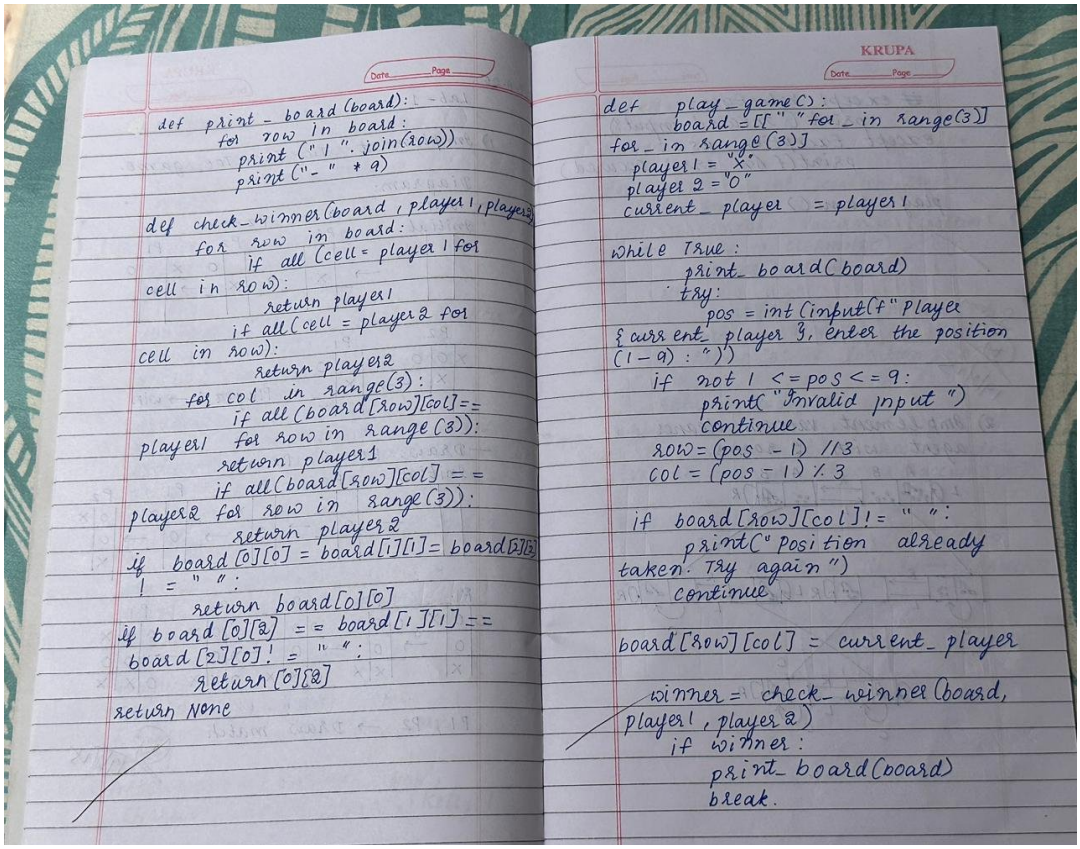
https://github.com/sushravya12/AI-1WA23CS004

Program 1

Implement Tic –Tac –Toe Game

Algorithm:





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

def check_winner(board, player1, player2):
    for row in board:
        if all(cell == player1 for
cell in row):
            return player1
        if all(cell == player2 for
cell in row):
            return player2
    for col in range(3):
        if all(board[row][col]==
player1 for row in range(3)):
            return player1
        if all(board[row][col] ==
player2 for row in range(3)):
            return player2
    if board[0][0] == board[1][1] == board[2][2]
!= " ":
        return board[0][0]
    if board[0][2] == board[1][1] ==
board[2][0] != " ":
        return board[0][2]
    return None
```

KRUPA

```
def play_game():
    board = [[" " for _ in range(3)]
for _ in range(3)]
    player1 = "X"
    player2 = "O"
    current_player = player1

    while True:
        print_board(board)
        try:
            pos = int(input(f"Player
{current_player}, enter the position
(1-9): "))
            if not 1 <= pos <= 9:
                print("Invalid input")
                continue
            row = (pos - 1) //3
            col = (pos - 1) % 3

            if board[row][col] != " ":
                print("Position already
taken. Try again")
                continue

            board[row][col] = current_player

            winner = check_winner(board,
player1, player2)
            if winner:
                print_board(board)
                break.
```

Code:

```
def print_board(board):
    print("\n")
    for i in range(3):
        print(" | ".join(board[i*3:(i+1)*3]))
        if i < 2:
            print("-" * 10)
    print("\n")

def check_winner(board, player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],
        [0, 3, 6], [1, 4, 7], [2, 5, 8],
        [0, 4, 8], [2, 4, 6]

for combo in win_conditions:
 count=0
            for pos in combo:
                if board[pos]==player:
                    count+=1
            if count==3:
                return True
        return False

    board = [" "] * 9
    current_player = "X"
    print_board(board)

    while True:
        while True:
            pos = int(input(f"Player {current_player}, enter your move (1-9): ")) - 1
            if 0 <= pos <= 8 and board[pos] == " ":
                board[pos] = current_player
                break
            else:
                print("Invalid move. Try again.")

        print_board(board)

        if check_winner(board, current_player):
            print(f"Player {current_player} wins!")
            break
        if " " not in board:
            print("It's a draw!")
            break
```

```
    # Switch players
    current_player = "O" if current_player == "X" else "X"
```

Output:

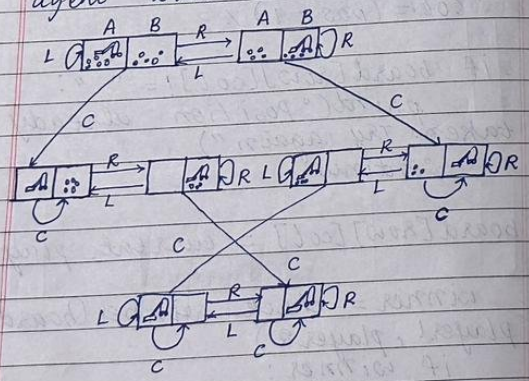
```
  |  |
-------
  |  |
-------
  |  |


Player X, enter your move (1-9): 1
X |  |
-------
  |  |
-------
  |  |
Player O, enter your move (1-9): 2
X | O |
-------
  |  |
-------
  |  |
Player X, enter your move (1-9): 3
X | O | X
-------
  |  |
-------
  |  |
Player O, enter your move (1-9): 4
X | O | X
-------
O |  |
-------
  |  |
Player X, enter your move (1-9): 5
X | O | X
-------
O | X |
-------
  |  |
Player X, enter your move (1-9): 5
X | O | X
-------
O | X |
-------
```

```
  |  |
Player O, enter your move (1-9): 6
X | O | X
- - - - - - -
O | X | O
- - - - - - -
  |  |
Player X, enter your move (1-9): 9

X | O | X
- - - - - - -
O | X | O
- - - - - - -
  |  | X

Player X wins!
```
Implement vacuum cleaner agent

```
except ValueError:
    print("Invalid input")
except Exception as e:
    print(f"An error occured")

play_game()
```

Seen ✓ 19/1/25

2) Implement vaccum cleaner agent with 2 rooms.



KRUPA

Date Page

```
0 = clean , 1 = Dirty
rooms = {'A': 1, 'B': 0}
vaccum_location = "A"

def vaccum_agent(location, state):
    if state == 1:
        print(f"Vaccum in {location}: Room dirty → CLEANING")
        rooms[location] = 0
    elif:
        print(f"Vaccum in {location}: Room clean → MOVE")
        return "B" if location == "A" elsf "A"
    return location
while any(rooms.values()):
    vaccum_location = vaccum_agent(vaccum_location, rooms[vaccum_loc])
```

Output:

```
Vaccum in A : Room dirty → CLEANING
{'A': 0, 'B': 0}
```

Code:

```
# Initial setup
rooms = [1, 1, 1, 1]  # 1 = dirty, 0 = clean
botpos = int(input("Enter Initial Position (1-4): ")) - 1
cost = 0
moves = 0

# Function to clean current room
def clean_room(pos):
    global cost
    if rooms[pos] == 1:
        rooms[pos] = 0
        cost += 1
        print(f"Cleaned room {pos+1}")
    else:
        print(f"Room {pos+1} already clean")

# Function to move bot
def move_bot(pos, direction):
    global moves
    # move right if direction=1 else left
    newpos = pos + direction
    moves += 1
    return newpos

# Main loop
direction = 1  # start moving right
while 1 in rooms: # loop until all rooms are clean
    print("Rooms:", rooms, " Bot at:", botpos+1)
    clean_room(botpos)

    if 1 not in rooms:
        break

    # change direction if at edges
    if botpos == len(rooms)-1:
        direction = -1
    elif botpos == 0:
        direction = 1

    botpos = move_bot(botpos, direction)

print("\n All rooms cleaned!")
print("Cleaning Cost =", cost)
print("Total Moves =", moves)
```

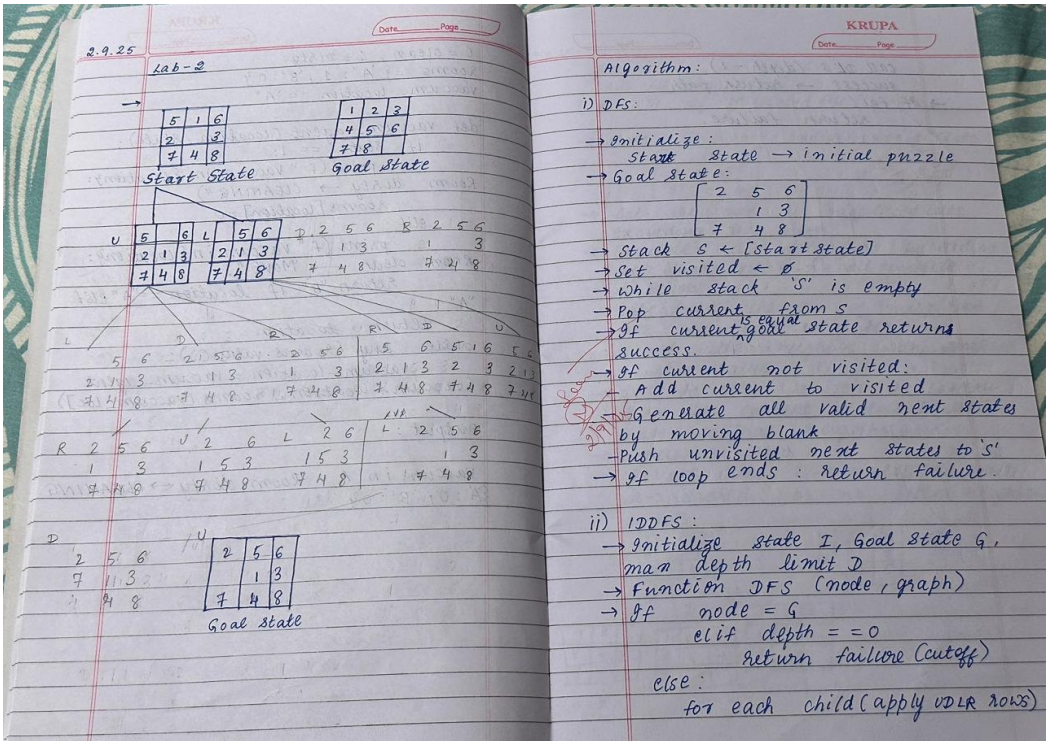
Output:

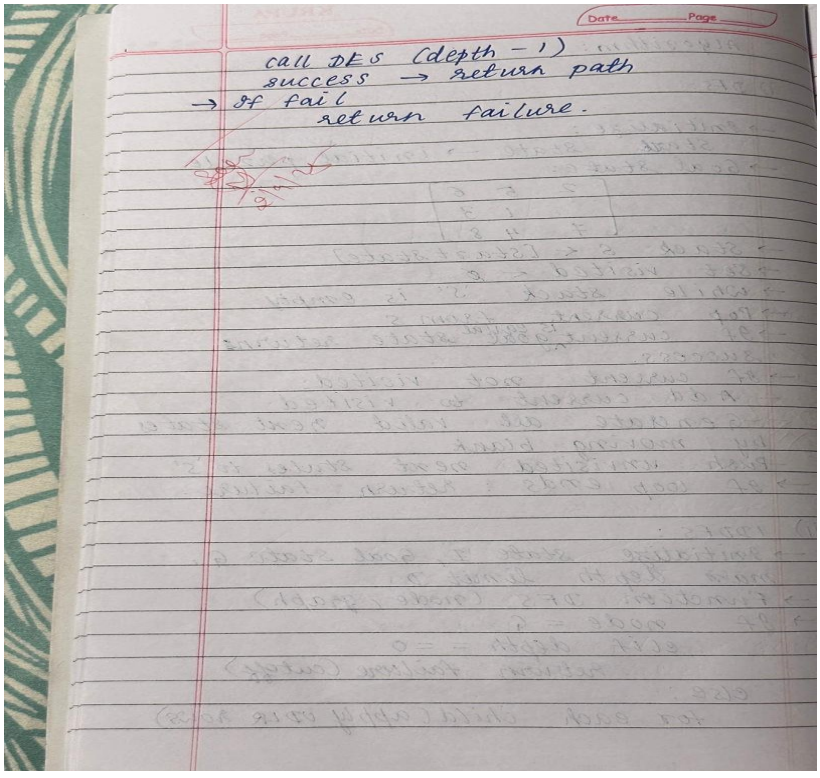
```
Enter Initial Position (1-4): 2
Rooms: [1, 1, 1, 1] Bot at: 2
Cleaned room 2
Rooms: [1, 0, 1, 1] Bot at: 3
Cleaned room 3
Rooms: [1, 0, 0, 1] Bot at: 4
Cleaned room 4
Rooms: [1, 0, 0, 0] Bot at: 3
Room 3 already clean
Rooms: [1, 0, 0, 0] Bot at: 2
Room 2 already clean
Rooms: [1, 0, 0, 0] Bot at: 1
Cleaned room 1

All rooms cleaned!
Cleaning Cost = 4
Total Moves = 5
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm:





Code:

```
from collections import deque
# Define possible moves
MOVES =
   { 'Up': -3,
   'Down': 3,
   'Left': -1,
   'Right': 1
def is_valid_move(zero_index, move):
   if move == 'Left' and zero_index % 3 == 0:
      return False
   if move == 'Right' and zero_index % 3 == 2:
      return False
   if move == 'Up' and zero_index < 3:
      return False
   if move == 'Down' and zero_index > 5:
      return False
   return True

def move_tile(state, move):
   zero_index = state.index('0')
   if not is_valid_move(zero_index, move):
      return None
   new_index = zero_index + MOVES[move]
   state_list = list(state)
   state_list[zero_index], state_list[new_index] = state_list[new_index], state_list[zero_index]
   return ''.join(state_list)

def reconstruct_path(parent, move_record, end_state):
   path = []
   current = end_state
   while parent[current] is not None:
      path.append((move_record[current], current))
      current = parent[current]
   path.reverse()
   return path

# BFS implementation
def bfs(start_state, goal_state):
   queue = deque()
   visited = set()
   parent = {}
   move_record = {}
```

```
    queue.append(start_state)
    visited.add(start_state)
    parent[start_state] = None
    move_record[start_state] = None

    while queue:
        current = queue.popleft()

        if current == goal_state:
            return reconstruct_path(parent, move_record, current)

        for move in MOVES:
            new_state = move_tile(current, move)
            if new_state and new_state not in visited:
                visited.add(new_state)
                queue.append(new_state)
                parent[new_state] = current
                move_record[new_state] = move

    return None
# DFS implementation
def dfs(start_state, goal_state, max_depth=50):
    stack = [(start_state, 0)]
    visited = set()
    parent = {}
    move_record = {}

    parent[start_state] = None
    move_record[start_state] = None

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

        if current == goal_state:
            return reconstruct_path(parent, move_record, current)

        if depth >= max_depth:
            continue

        visited.add(current)

        for move in reversed(list(MOVES)): # Reverse for consistent DFS order
            new_state = move_tile(current, move)
            if new_state and new_state not in visited:
                stack.append((new_state, depth + 1))
            parent[new_state] = current
```

```
                move_record[new_state] = move

    return None

def print_puzzle(state):
    for i in range(0, 9, 3):
        row = state[i:i+3]
        print(' '.join(c if c != '0' else '_' for c in row))
    print()

def print_solution(start_state, goal_state, method='bfs'):
    print(f"\nStart State ({method.upper()}):")
    print_puzzle(start_state)
    print("Goal State:")
    print_puzzle(goal_state)

    if method == 'bfs':
        path = bfs(start_state, goal_state)
    elif method == 'dfs':
        path = dfs(start_state, goal_state)
    else:
        raise ValueError("Method must be 'bfs' or 'dfs'")

    if path is None:
        print("No solution found.")
        return

    print(f"\nSolution found in {len(path)} moves:")
    current = start_state
    for move, state in path:
        print(f"Move: {move}")
        print_puzzle(state)

def get_state_input(prompt):
    while True:
        raw = input(prompt).replace(" ", "").replace("_", "0")
        if len(raw) != 9 or not all(c in "0123456789" for c in raw):
            print("Invalid input. Enter 9 digits from 0-8 (0 for blank).")
            continue
        if sorted(raw) != list("012345678"):
            print("Invalid puzzle: must contain digits 0 through 8 exactly once.")
            continue
        return raw

# Main program
if _name___== "_main_":
    print("8-Puzzle Solver (BFS and DFS)")
```

```
    print("Enter the puzzle state as 9 digits (0 = blank). Example: 123405678")

    start = get_state_input("Enter the initial state: ")
    goal = get_state_input("Enter the goal state: ")

    method = ''
    while method not in ['bfs', 'dfs']:
        method = input("Choose search method (bfs/dfs): ").strip().lower()

    print_solution(start, goal, method)
```

Output:

```
8-Puzzle Solver (BFS and DFS)
Enter the puzzle state as 9 digits (0 = blank). Example: 123405678
Enter the initial state: 283164705
Enter the goal state: 123804765
Choose search method (bfs/dfs): dfs
Start State (DFS):
2 8 3
1 6 4
7 _ 5

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


Solution found in 43 moves:
Move: Up
2 8 3
1 _ 4
7 6 5

Move: Up
2 _ 3
1 8 4
7 6 5

Move: Left
_ 2 3
1 8 4
7 6 5
```

Move: Down
1 2 3
_ 8 4
7 6 5

Move: Down
1 2 3
7 8 4
_ 6 5

Move: Right
1 2 3
7 8 4
6 _ 5

Move: Up
1 2 3
7 _ 4
6 8 5

Move: Up
1 _ 3
7 2 4
6 8 5

Move: Left
_ 1 3
7 2 4
6 8 5

Move: Down
7 1 3
_ 2 4
6 8 5

Move: Down
7 1 3
6 2 4
_ 8 5

Move: Right
7 1 3
6 2 4
8 _ 5

Move: Up
7 1 3

6 _ 4
8 2 5

Move: Up
7 _ 3
6 1 4
8 2 5

Move: Left
_ 7 3
6 1 4
8 2 5

Move: Down
6 7 3
_ 1 4
8 2 5

Move: Down
6 7 3
8 1 4
_ 2 5

Move: Right
6 7 3
8 1 4
2 _ 5

Move: Up
6 7 3
8 _ 4
2 1 5

Move: Up
6 _ 3
8 7 4
2 1 5

Move: Left
_ 6 3
8 7 4
2 1 5

Move: Down
8 6 3
_ 7 4
2 1 5

Move: Down
8 6 3
2 7 4
_ 1 5

Move: Right
8 6 3
2 7 4
1 _ 5

Move: Up
8 6 3
2 _ 4
1 7 5

Move: Up
8 _ 3
2 6 4
1 7 5

Move: Left
_ 8 3
2 6 4
1 7 5

Move: Down
2 8 3
_ 6 4
1 7 5

Move: Right
2 8 3
6 _ 4
1 7 5

Move: Down
2 8 3
6 7 4
1 _ 5

Move: Left
2 8 3
6 7 4
_ 1 5

Move: Up
2 8 3

_ 7 4
6 1 5

Move: Right
2 8 3
7 _ 4
6 1 5

Move: Down
2 8 3
7 1 4
6 _ 5

Move: Left
2 8 3
7 1 4
_ 6 5

Move: Up
2 8 3
_ 1 4
7 6 5

Move: Up
_ 8 3
2 1 4
7 6 5

Move: Right
8 _ 3
2 1 4
7 6 5

Move: Down
8 1 3
2 _ 4
7 6 5

Move: Left
8 1 3
_ 2 4
7 6 5

Move: Up
_ 1 3
8 2 4
7 6 5

Move: Right
1 _ 3
8 2 4
7 6 5

Move: Down
1 2 3
8 _ 4
7 6 5

Implement Iterative deepening search algorithm

Code:

```
import copy

# Goal state
goal_state = [[1, 2, 3],
              [8, 0, 4],
              [7, 6, 5]]

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

# Find the position of the blank tile (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
# Check if the current state is the goal state
def is_goal(state):
    return state == goal_state

# Get all possible neighbors (states) by moving the blank tile
def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors
# Convert the state into a tuple for hashing in the visited set
def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

# Print the current state in a readable format
def print_state(state):
    for row in state:
        print(" ".join(str(num) if num != 0 else "_" for num in row))
    print("              ")
# Depth-limited DFS
def dfs_limited(state, path, depth, limit, visited):
    if depth == limit: # If we reached the depth limit, stop exploring
```

```
        return None

    # Print the current state (optional, can be commented out for cleaner output)
    print_state(state)

    if is_goal(state):
        return path + [state] # Return the solution path if goal is found

    visited.add(state_to_tuple(state)) # Mark current state as visited

    for neighbor in get_neighbors(state):
        key = state_to_tuple(neighbor)
        if key not in visited:
            result = dfs_limited(neighbor, path + [state], depth + 1, limit, visited)
            if result:
                return result # If a valid solution is found, return it
    return None # No solution found at this depth
    # Iterative Deepening DFS
def iddfs(start_state, max_depth=50):
    for limit in range(max_depth + 1):
        print(f"=== Iteration with depth limit {limit} ===")
        visited = set() # Set to track visited states
        path = dfs_limited(start_state, [], 0, limit, visited) # Start DFS with depth 0
        if path:
            print("Goal reached!")
            print("Visited:", len(visited))
            print("Solution depth:", len(path) - 1)
            print("Steps:")
            for step in path:
                print_state(step) # Print the path from start to goal
            return # Solution found, exit the function
    print("No solution found within depth limit.") # No solution found after max_depth

# Example usage
start_state = [[2, 8, 3],
               [1, 6, 4],
               [7, 0, 5]]

iddfs(start_state, max_depth=20) # Start the search with a max depth of 20
```

Output:

```
=== Iteration with depth limit 0 ===
=== Iteration with depth limit 1 ===
2 8 3
1 6 4
7 _ 5
- - - - - - -
=== Iteration with depth limit 2 ===
2 8 3
1 6 4
7 _ 5
- - - - - - -
2 8 3
1 _ 4
7 6 5
- - - - - - -
2 8 3
1 6 4
_ 7 5
- - - - - - -
2 8 3
1 6 4
7 5 _
=== Iteration with depth limit 3 ===
2 8 3
1 6 4
7 _ 5
- - - - - - -
2 8 3
1 _ 4
7 6 5
- - - - - - -
2 _ 3
1 8 4
7 6 5
- - - - - - -
2 8 3
_ 1 4
7 6 5
- - - - - - -
2 8 3
1 4 _
7 6 5
- - - - - - -
2 8 3
```

1 6 4

_ 7 5
- - - - - - -
2 8 3
_ 6 4
1 7 5
- - - - - - -
2 8 3
1 6 4
7 5 _
- - - - - - -
2 8 3
1 6 _
7 5 4
=== Iteration with depth limit 4 ===
2 8 3
1 6 4
7 _ 5
- - - - - - -
2 8 3
1 _ 4
7 6 5
- - - - - - -
2 _ 3
1 8 4
7 6 5
- - - - - - -
_ 2 3
1 8 4
7 6 5
- - - - - - -
2 3 _
1 8 4
7 6 5
- - - - - - -
2 8 3
_ 1 4
7 6 5
- - - - - - -
_ 8 3
2 1 4
7 6 5
- - - - - - -
2 8 3
7 1 4

_ 6 5

2 8 3
1 4 _
7 6 5
---
2 8 _
1 4 3
7 6 5
---
2 8 3
1 4 5
7 6 _
---
2 8 3
1 6 4
_ 7 5
---
2 8 3
_ 6 4
1 7 5

_ 8 3
2 6 4
1 7 5
---
2 8 3
6 _ 4
1 7 5
---
2 8 3
1 6 4
7 5 _
---
2 8 3
1 6 _
7 5 4
---
2 8 _
1 6 3
7 5 4
---
2 8 3
1 _ 6

=== Iteration with depth limit 5 ===
2 8 3
1 6 4
7 _ 5
- - - - - - -
2 8 3
1 _ 4
7 6 5
- - - - - - -
2 _ 3
1 8 4
7 6 5
- - - - - - -
_ 2 3
1 8 4
7 6 5
- - - - - - -
1 2 3
_ 8 4
7 6 5
- - - - - - -
2 3 _
1 8 4
7 6 5
- - - - - - -
2 3 4
1 8 _
7 6 5
2 8 3
_ 1 4
7 6 5
- - - - - - -
_ 8 3
2 1 4
7 6 5
- - - - - - -
8 _ 3
2 1 4
7 6 5
- - - - - - -
2 8 3
7 1 4
_ 6 5
- - - - - - -
2 8 3
7 1 4

6 _ 5
- - - - - - -
2 8 3
1 4 _
7 6 5
- - - - - - -
2 8 _
1 4 3
7 6 5
- - - - - - -
2 _ 8
1 4 3
7 6 5
- - - - - - -
2 8 3
1 4 5
7 6 _
- - - - - - -
2 8 3
1 4 5
7 _ 6
2 8 3
1 6 4
_ 7 5
- - - - - - -
2 8 3
_ 6 4
1 7 5
- - - - - - -
_ 8 3
2 6 4
1 7 5
- - - - - - -
8 _ 3
2 6 4
1 7 5
- - - - - - -
2 8 3
6 _ 4
1 7 5
2 _ 3
6 8 4
1 7 5
- - - - - - -
2 8 3
6 7 4

1 _ 5
- - - - - - -
2 8 3
6 4 _
1 7 5
- - - - - - -
2 8 3
1 6 4
7 5 _
- - - - - - -
2 8 3
1 6 _
7 5 4
- - - - - - -
2 8 _
1 6 3
7 5 4
- - - - - - -
2 _ 8
1 6 3
7 5 4
- - - - - - -
2 8 3
1 _ 6
7 5 4
- - - - - - -
2 _ 3
1 8 6
7 5 4
- - - - - - -
2 8 3
1 5 6
7 _ 4
- - - - - - -
2 8 3
_ 1 6
7 5 4
- - - - - - -
=== Iteration with depth limit 6 ===
2 8 3
1 6 4
7 _ 5
- - - - - - -
2 8 3
1 _ 4
7 6 5
- - - - - - -

2 _ 3
1 8 4

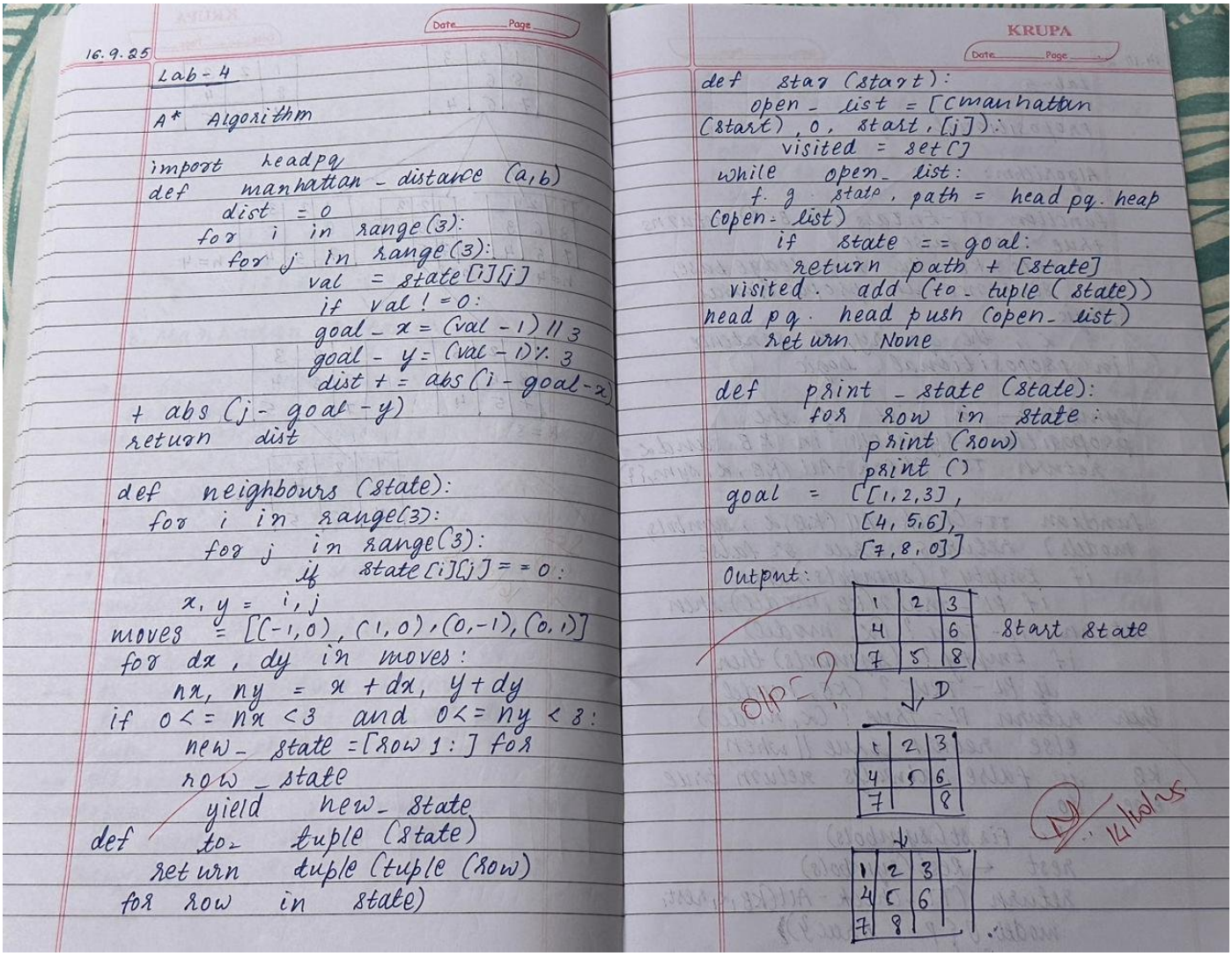
7 6 5
- - - - - - -
_ 2 3
1 8 4
7 6 5
- - - - - - -
1 2 3
_ 8 4
7 6 5
- - - - - - -
1 2 3
7 8 4
_ 6 5
- - - - - - -
1 2 3
8 _ 4
7 6 5
- - - - - - -
Goal reached!
Visited: 6
Solution depth: 5
Steps:
2 8 3
1 6 4
7 _ 5
- - - - - - -
2 8 3
1 _ 4
7 6 5
- - - - - - -
2 _ 3
1 8 4
7 6 5
- - - - - - -
_ 2 3
1 8 4
7 6 5
- - - - - - -
1 2 3
_ 8 4
7 6 5
- - - - - - -

1 2 3
8 _ 4
7 6 5

Program 3

Implement A* search algorithm

Algorithm:



Code:

```
from heapq import heappush, heappop

GOAL = (1, 2, 3,
        8, 0, 4,
        7, 6, 5) # 0 = blank
GOAL_POS = {v: (i // 3, i % 3) for i, v in enumerate(GOAL)}
def manhattan_distance(state):
    dist = 0
    for i, v in enumerate(state):
        if v != 0: # skip blank
            r, c = divmod(i, 3) # current position (row, col)
            gr, gc = GOAL_POS[v] # goal position (row, col)
            dist += abs(r - gr) + abs(c - gc)
    return dist

# Inversion parity (checking solvability)
def inversion_parity(state):
    arr = [v for v in state if v != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv % 2
def is_solvable(start, goal):
    """General solvability: start and goal must have same inversion parity."""
    return inversion_parity(start) == inversion_parity(goal)

# Moves: (name, row_delta, col_delta)
MOVES =
    [ ("up",  -1, 0),
    ("down",  1, 0),
    ("left",  0, -1),
    ("right", 0, 1),
]

def neighbors(state):
    """Generate (next_state, action) pairs by sliding a tile into the blank."""
    zero = state.index(0)
    zr, zc = divmod(zero, 3)

    for name, dr, dc in MOVES:
        nr, nc = zr + dr, zc + dc
        if 0 <= nr < 3 and 0 <= nc <
            3: nidx = nr * 3 + nc
            new_state = list(state)
```

```
            new_state[zero], new_state[nidx] = new_state[nidx], new_state[zero]
            yield tuple(new_state), name

# -------- A* search --------
def a_star(start):
    if not is_solvable(start, GOAL):
        raise ValueError("This puzzle configuration is not solvable.")

    counter = 0
    h0 = manhattan_distance(start)
    frontier = []
    heappush(frontier, (h0, counter, start))

    g = {start: 0} # g(n): cost to reach the current state from the start
    parent = {start: None}
    action_to = {start: None}

    while frontier:
        f, _, s = heappop(frontier)

        if s == GOAL:
            # Reconstruct path
            actions, states = [], []
            cur = s
            while cur is not None:
                states.append(cur)
                actions.append(action_to[cur])
                cur = parent[cur]
            actions = actions[-2::-1] # drop None, reverse
            states = states[::-1]
            return actions, states, g

        for ns, act in neighbors(s):
            tentative_g = g[s] + 1 # g(n) = g(parent) + 1
            if ns not in g or tentative_g < g[ns]:
                g[ns] = tentative_g
                parent[ns] = s
                action_to[ns] = act
                counter += 1
                heappush(frontier, (tentative_g + manhattan_distance(ns), counter, ns))

    raise ValueError("No path found (should not happen for solvable states).")

def print_state(state):
    for r in range(3):
        row = state[3*r:3*r+3]
        print(" ".join(str(x) if x != 0 else "·" for x in row))
```

```
    print()

if __name__ == "__main__":
    start = (2, 8, 3,
             1, 6, 4,
             7, 0, 5)
    print("Start:")
    print_state(start)
    print("Goal:")
    print_state(GOAL)
    try:
        actions, states, g = a_star(start)
        print(f"Solved in {len(actions)} moves using Manhattan distance heuristic.\n")

        # Printing the costs during the search path
        for i, (a, st) in enumerate(zip(actions, states[1:]), start=1):
            g_cost = g[st] # g(n): cost to reach this state from the start
            h_cost = manhattan_distance(st) # h(n): Manhattan distance heuristic
            f_cost = g_cost + h_cost # f(n) = g(n) + h(n)
            print(f"Move {i}: {a}")
            print(f"g(n) = {g_cost}, h(n) = {h_cost}, f(n) = {f_cost}")
            print_state(st)
    except ValueError as e:
        print(e)

from heapq import heappush, heappop
# ---------- Problem setup ----------
GOAL = (1, 2, 3,
        8, 0, 4,
        7, 6, 5) # 0 = blank

# Heuristic: misplaced tiles
def misplaced_tiles(state):
    return sum(1 for i, v in enumerate(state) if v != 0 and v != GOAL[i])

# Inversion count
def inversion_parity(state):
    arr = [v for v in state if v != 0]
    inv = 0
    for i in range(len(arr)):
        for j in range(i + 1, len(arr)):
            if arr[i] > arr[j]:
                inv += 1
    return inv % 2

def is_solvable(start, goal):
    """General solvability: start and goal must have same inversion parity."""
```

```
    return inversion_parity(start) == inversion_parity(goal)

# Moves: (name, row_delta, col_delta)
MOVES =
    [ ("up",  -1, 0),
    ("down",  1, 0),
    ("left",  0, -1),
    ("right", 0, 1),
]

def neighbors(state):
    """Generate (next_state, action) pairs by sliding a tile into the blank."""
    zero = state.index(0)
    zr, zc = divmod(zero, 3)

    for name, dr, dc in MOVES:
        nr, nc = zr + dr, zc + dc
        if 0 <= nr < 3 and 0 <= nc <
            3: nidx = nr * 3 + nc
            new_state = list(state)
            new_state[zero], new_state[nidx] = new_state[nidx], new_state[zero]
            yield tuple(new_state), name

# -------- A* search --------
def a_star(start):
    if not is_solvable(start, GOAL):
        raise ValueError("This puzzle configuration is not solvable.")

    counter = 0
    h0 = misplaced_tiles(start)
    frontier = []
    heappush(frontier, (h0, counter, start))

    g = {start: 0}
    parent = {start: None}
    action_to = {start: None}

    while frontier:
        f, _, s = heappop(frontier)
        if s == GOAL:
            # Reconstruct path
            actions, states = [], []
            cur = s
            while cur is not None:
                states.append(cur)
                actions.append(action_to[cur])
                cur = parent[cur]
```

```
            actions = actions[-2::-1] # drop None, reverse
            states = states[::-1]
            return actions, states, g

        for ns, act in neighbors(s):
            tentative_g = g[s] + 1
            if ns not in g or tentative_g < g[ns]:
                g[ns] = tentative_g
                parent[ns] = s
                action_to[ns] = act
                counter += 1
                heappush(frontier, (tentative_g + misplaced_tiles(ns), counter, ns))

    raise ValueError("No path found (should not happen for solvable states).")

# ---------- Pretty printing ----------
def print_state(state):
    for r in range(3):
        row = state[3*r:3*r+3]
        print(" ".join(str(x) if x != 0 else "·" for x in row))
    print()

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

    print("Start:")
    print_state(start)
    print("Goal:")
    print_state(GOAL)
    try:
        actions, states, g = a_star(start)
        print(f"Solved in {len(actions)} moves using misplaced-tiles heuristic.\n")
        for i, (a, st) in enumerate(zip(actions, states[1:]), start=1):
            g_cost = g[st]
            h_cost = misplaced_tiles(st)
            f_cost = g_cost + h_cost
            print(f"Move {i}: {a}")
            print(f"g={g_cost}, h={h_cost}, f={f_cost}")
            print_state(st)
    except ValueError as e:
        print(e)
```

Output:

```
Start:
```

2 8 3
1 6 4
7 · 5

Goal:
1 2 3
8 · 4
7 6 5

Solved in 5 moves using Manhattan distance heuristic.

Move 1: up
g(n) = 1, h(n) = 4, f(n) = 5
2 8 3
1 · 4
7 6 5

Move 2: up
g(n) = 2, h(n) = 3, f(n) = 5
2 · 3
1 8 4
7 6 5

Move 3: left
g(n) = 3, h(n) = 2, f(n) = 5
· 2 3
1 8 4
7 6 5

Move 4: down
g(n) = 4, h(n) = 1, f(n) = 5
1 2 3
· 8 4
7 6 5

Move 5: right
g(n) = 5, h(n) = 0, f(n) = 5
1 2 3
8 · 4
7 6 5
Sanjana Srinivas-1BM23CS301
Start:
2 8 3
1 6 4
7 · 5
Goal:
1 2 3

8 · 4
7 6 5

Solved in 5 moves using misplaced-tiles heuristic.

Move 1: up
g=1, h=3, f=4
2 8 3
1 · 4
7 6 5

Move 2: up
g=2, h=3, f=5
2 · 3
1 8 4
7 6 5

Move 3: left
g=3, h=2, f=5
· 2 3
1 8 4
7 6 5

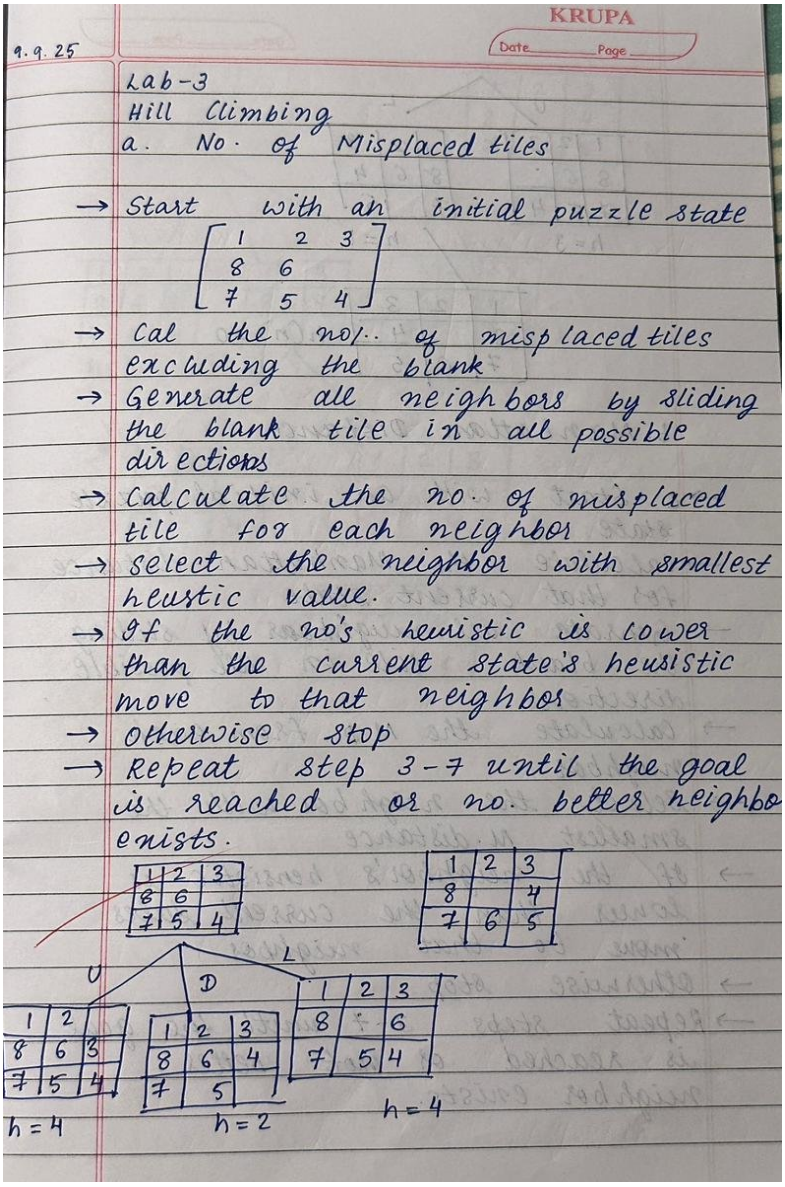
Move 4: down
g=4, h=1, f=5
1 2 3
· 8 4
7 6 5

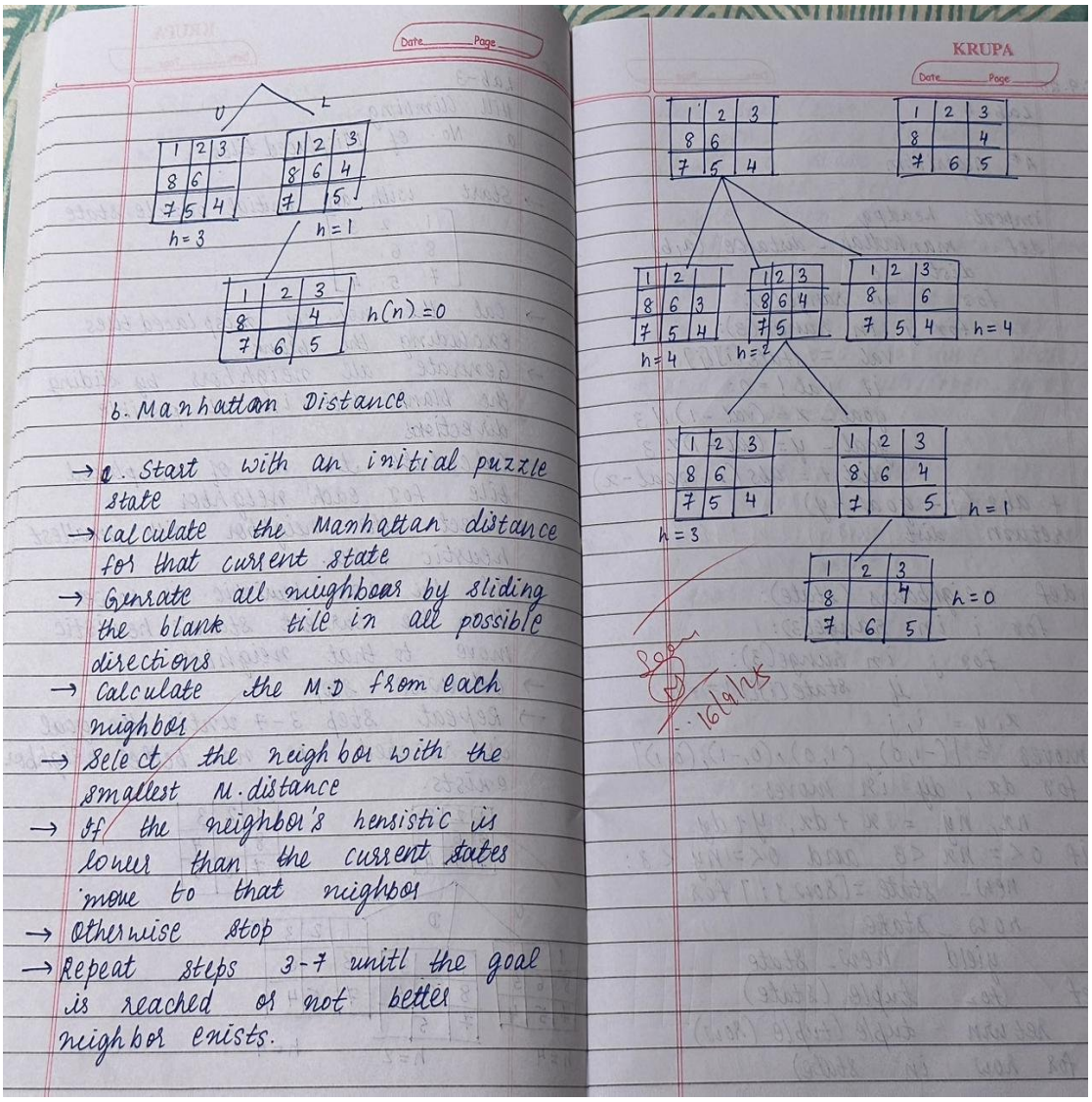
Move 5: right
g=5, h=0, f=5
1 2 3
8 · 4
7 6 5

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:





b. Manhattan Distance

→ a. Start with an initial puzzle state
→ calculate the Manhattan distance for that current state
→ Genrate all neighbors by sliding the blank tile in all possible directions
→ Calculate the M.D from each neighbor
→ Select the neighbor with the smallest M.distance
→ If the neighbor's hensistic is lower than the current states move to that neighbor
→ Otherwise stop
→ Repeat steps 3-7 until the goal is reached or not better neighbor exists.

Code:
a.

```
def read_state(prompt):
    print(prompt)
    numbers = input().strip().split()
    if len(numbers) != 9 or not all(n.isdigit() for n in numbers):
        print("Enter exactly 9 numbers separated by spaces.")
        return read_state(prompt)
    return [list(map(int, numbers[i*3:(i+1)*3])) for i in range(3)]

def misplaced_tiles(state, goal):
```

```
    return sum(
        1 for i in range(3) for j in range(3)
        if state[i][j] != 0 and state[i][j] != goal[i][j]
    )

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

def get_neighbors(state):
    neighbors = []
    x, y = find_zero(state)
    moves = [(-1,0), (1,0), (0,-1), (0,1)]
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [row[:] for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def state_equal(s1, s2):
    return all(s1[i][j] == s2[i][j] for i in range(3) for j in range(3))

def solve(initial, goal, max_steps=30):
    current = initial
    steps = 0
    visited = []

    print("Starting state:")
    for row in current:
        print(row)

    while not state_equal(current, goal) and steps < max_steps:
        visited.append(current)
        neighbors = get_neighbors(current)
        best = None
        lowest_h = float('inf')
        for neighbor in neighbors:
            if any(state_equal(neighbor, v) for v in visited):
                continue
            h = misplaced_tiles(neighbor, goal)
            if h < lowest_h:
                lowest_h = h
                best = neighbor
        if best is None:
```

```
            print("No more moves or stuck. Aborting.")
            return
        current = best
        steps += 1
        print(f"\nStep {steps}, Misplaced tiles: {lowest_h}")
        for row in current:
            print(row)

    if state_equal(current, goal):
        print("\nSolved in", steps, "steps!")
    else:
        print("\nMax steps reached or stuck, not solved.")

initial_state = read_state("Enter the initial state (use 0 for blank):")
goal_state = read_state("Enter the goal state:")

solve(initial_state, goal_state)
```

Code:

b.

```
from copy import deepcopy

def read_state(prompt):
    while True:
        print(prompt)
        numbers = input().strip().split()
        if len(numbers) != 9 or not all(n.isdigit() for n in numbers):
            print("Error: Enter exactly 9 numbers separated by spaces (0 for blank). Try again.")
            continue
        numbers = list(map(int, numbers))

        if set(numbers) != set(range(9)):
            print("Error: Numbers must be from 0 to 8 with no duplicates. Try again.")
            continue
        return [numbers[i*3:(i+1)*3] for i in range(3)]

def find_position(state, value):
    for i in range(3):
        for j in range(3):
            if state[i][j] == value:
                return i, j
    return None

def manhattan_distance(state, goal):
```

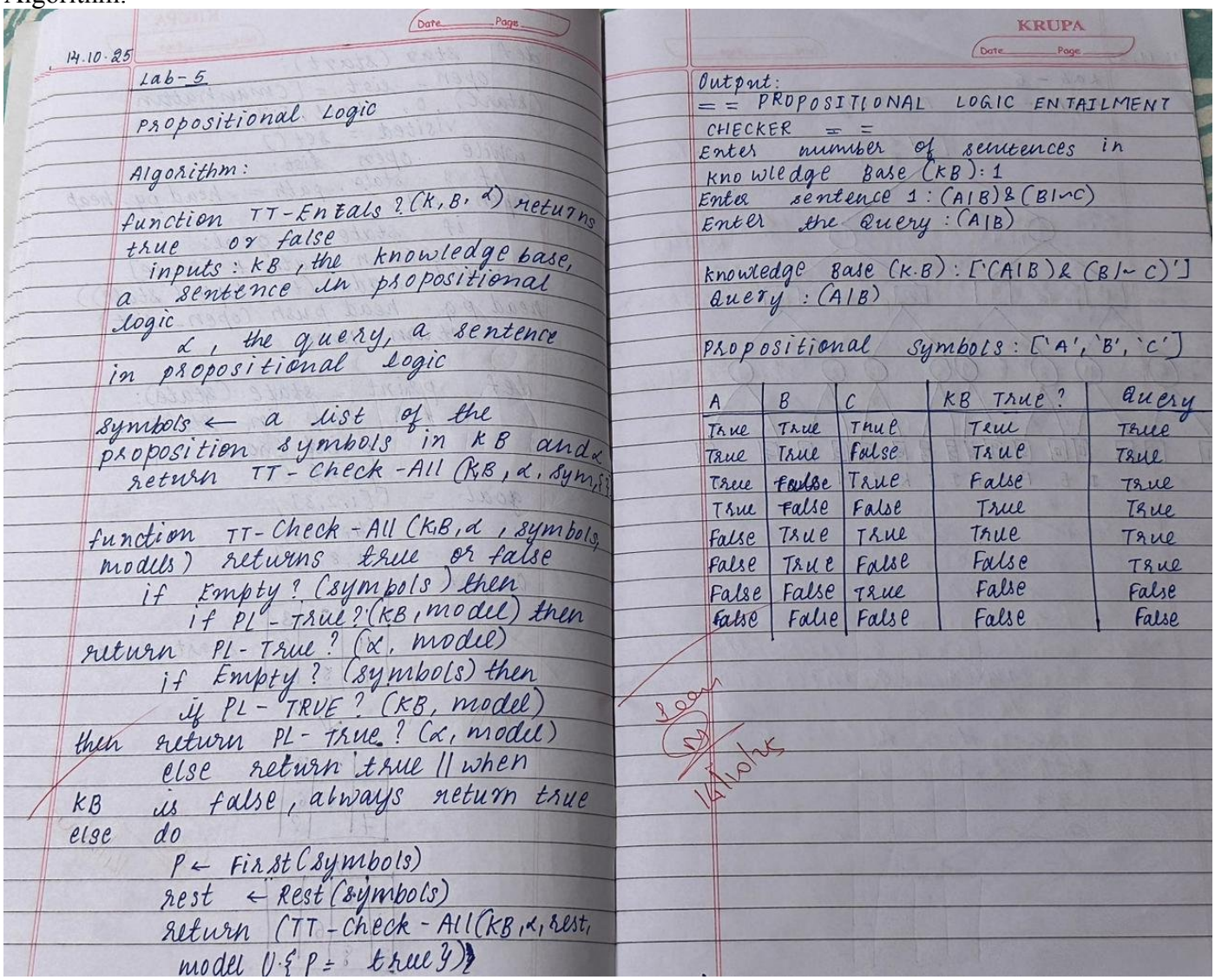
```
    distance = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val != 0:
                goal_i, goal_j = find_position(goal, val)
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

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

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    moves = [(-1, 0, "up"), (1, 0, "down"), (0, -1, "left"), (0, 1, "right")]
    for dx, dy, move in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append((new_state, move))
    return neighbors

def print_state(state):
    for row in state:
        print(row)
    print()

def hill_climbing(initial_state, goal_state, max_iterations=1000):
    current = initial_state
    current_h = manhattan_distance(current, goal_state)
    steps = 0
    path = []

    print(f"Step {steps}: (Manhattan Distance: {current_h})")
    print_state(current)

    while steps < max_iterations:
        neighbors = get_neighbors(current)
        neighbor_h = [(manhattan_distance(n[0], goal_state), n[0], n[1]) for n in neighbors]

        neighbor_h.sort(key=lambda x: x[0])
        best_h, best_state, best_move = neighbor_h[0]
```

```
        if best_h >= current_h:
            return current, current_h, steps, path

        current, current_h = best_state, best_h
        path.append(best_move)
        steps += 1

        print(f"Step {steps}: Move: {best_move} (Manhattan Distance: {current_h})")
        print_state(current)

    return current, current_h, steps, path

if __name__ == "__main__":
    initial_state = read_state("Enter the initial state (9 numbers 0-8 with 0 as blank, separated by
spaces):")
    goal_state = read_state("Enter the goal state (9 numbers 0-8 with 0 as blank, separated by spaces):")

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

    print("Goal State:")
    print_state(goal_state)

    final_state, h, steps, path = hill_climbing(initial_state, goal_state)

    print(f"Final State after Hill Climbing (after {steps} steps):")
    print_state(final_state)
    print(f"Manhattan Distance: {h}")

    if h == 0:
        print("Reached goal state!")
    else:
        print("Stopped at local minimum (not solved).")

    if path:
        print("Final Path of moves:")
        print("Start -> " + " -> ".join(path) + " -> Goal")
    else:
        print("No moves made.")
```

Program 6

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

Algorithm:



14.10.25

Lab-5

Propositional Logic

Algorithm:

function TT-Entails?(KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositional logic

symbols ← a list of the proposition symbols in KB and α
return TT-Check-All(KB, α, symbols, {})

function TT-Check-All(KB, α, symbols, model) returns true or false
if Empty?(symbols) then
if PL-True?(KB, model) then return PL-True?(α, model)
if Empty?(symbols) then
if PL-TRUE?(KB, model) then return PL-True?(α, model)
else return true // when KB is false, always return true
else do
P ← First(symbols)
rest ← Rest(symbols)
return (TT-Check-All(KB, α, rest, model ∪ {P = true}))

KRUPA

Output:

== PROPOSITIONAL LOGIC ENTAILMENT CHECKER ==
Enter number of sentences in knowledge base (KB): 1
Enter sentence 1: (A|B)&(B|~C)
Enter the Query: (A|B)

knowledge base (KB): ['(A|B)&(B|~C)']
Query: (A|B)

propositional symbols: ['A', 'B', 'C']

A	B	C	KB True?	Query
True	True	True	True	True
True	True	False	True	True
True	False	True	False	True
True	False	False	True	True
False	True	True	True	True
False	True	False	False	True
False	False	True	False	False
False	False	False	False	False

Learn
Try
Values

Code:

```
from itertools import product

# --- Utility to parse logical expressions safely ---
def eval_formula(expr, model):
    # Replace propositional symbols with model values
    for sym, val in model.items():
        expr = expr.replace(sym, str(val))

    # Replace logical operators with Python equivalents
    expr = expr.replace("¬", " not ")
    expr = expr.replace("∧", " and ")
    expr = expr.replace("∨", " or ")
    expr = expr.replace("→", " <= ")  # A → B ≡ (not A) or B
    expr = expr.replace("↔", " == ")  # A ↔ B ≡ (A == B)

    return eval(expr)

# --- Main program ---
print("Available logical operators you can use in input formulas:")
print(" Negation (NOT):        ¬")
print(" Conjunction (AND):     ∧")
print(" Disjunction (OR):      ∨")
print(" Implication (IMPLIES):
→") print(" Biconditional (IFF):
↔") print("-" * 50)

symbols = input("Enter symbols separated by space (e.g., P Q R): ").split()
kb_expr = input("Enter Knowledge Base formula (e.g., (P ∧ Q) → R): ")
query_expr = input("Enter Query formula α (e.g., R): ")

print("\nKnowledge Base (KB):", kb_expr)
print("Query (α):", query_expr, "\n")

# Function to print truth table
def print_truth_table(kb_expr, query_expr, symbols):
    header = symbols + ["KB", "α"]
    print(" | ".join(f"{h:^5}" for h in header))
    print("-" * (7 * len(header)))

    for values in product([False, True], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        kb_val = eval_formula(kb_expr, model)
        q_val = eval_formula(query_expr, model)
        row = [model[s] for s in symbols] + [kb_val, q_val]
        print(" | ".join(f"{str(r):^5}" for r in row))
```

```
# Entailment check
def entails(kb_expr, query_expr, symbols):
    for values in product([True, False], repeat=len(symbols)):
        model = dict(zip(symbols, values))
        if eval_formula(kb_expr, model):  # KB true
            if not eval_formula(query_expr, model): # Query false → not entailed
                return False
    return True

# Run
print_truth_table(kb_expr, query_expr, symbols)
result = entails(kb_expr, query_expr, symbols)
print("\nDoes KB entail Query (α)? :", result)
```

Query (α): A ∨ B

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

Does KB entail Query (α)? : True

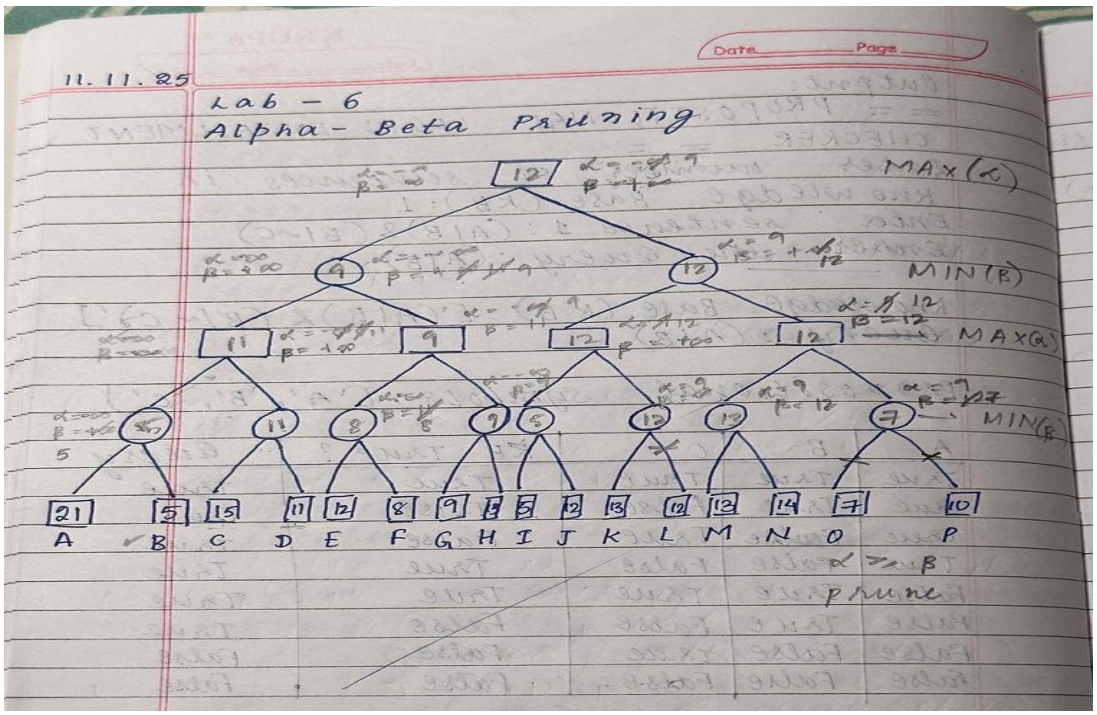
Output:

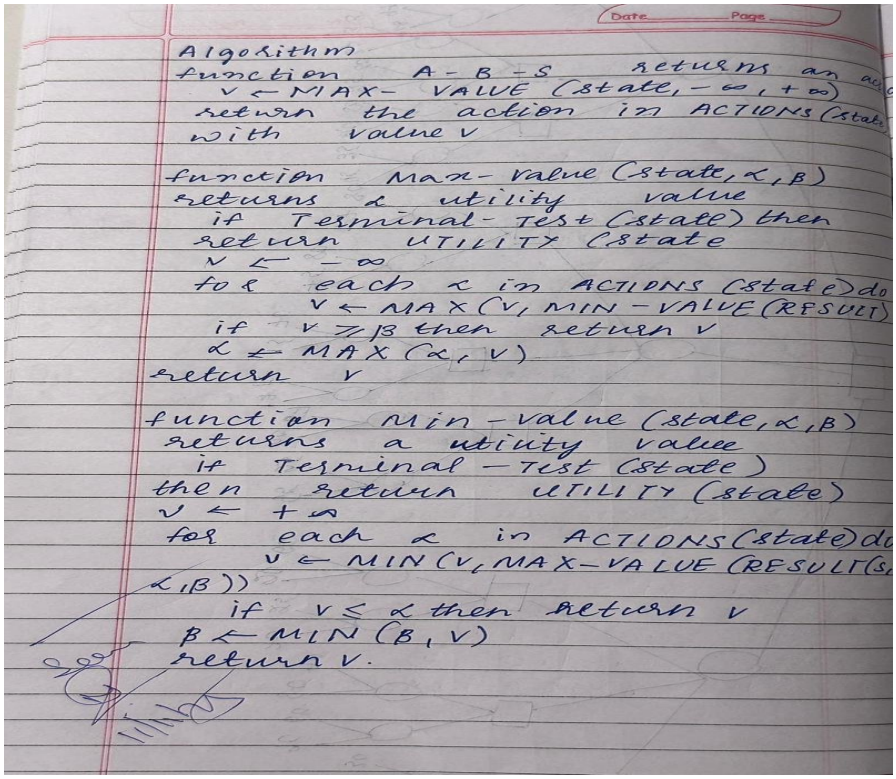
```
Available logical operators you can use in input formulas:
  Negation (NOT):       ¬
  Conjunction (AND):    ∧
  Disjunction (OR):     ∨
  Implication (IMPLIES):
  → Biconditional (IFF): ↔
--------------------------------.
Enter symbols separated by space (e.g., P Q R): A B C
Enter Knowledge Base formula (e.g., (P ∧ Q) → R): (A∨C) ∧ (B∨¬C)
Enter Query formula α (e.g., R): A ∨ B

Knowledge Base (KB): (A∨C) ∧ (B∨¬C)
```

Program 7

Implement alpha-beta pruning

Algorithm:





Code:

```
from typing import List, Optional, Tuple
import math

class Node:
    def _init_(self, name: str, children: Optional[List["Node"]] = None, value: Optional[int] = None):
        self.name = name
        self.children = children or []
        self.value = value

    def is_leaf(self):
        return self.value is not None

    def _repr_(self): if
        self.is_leaf():
            return f"{self.name}({self.value})"
        return f"{self.name}"
EXPANSION_ORDER: List[str] = []
PRUNED_SUBTREES: List[Tuple[str, List[str]]] = []

def alpha_beta(node: Node,
               alpha: float,
               beta: float,
               maximizing_player: bool,
               path: Optional[List[str]] = None) -> Tuple[float, List[str]]:
    if path is None:
        path = []
    path = path + [node.name]
    if node.is_leaf():
        EXPANSION_ORDER.append(node.name)
        return node.value, path
    EXPANSION_ORDER.append(node.name)
    if maximizing_player:
        value = -math.inf
        best_path: List[str] = []
        for child in node.children:
            if value >= beta:
             PRUNED_SUBTREES.append((child.name, path + [child.name]))
             continue
            child_val, child_path = alpha_beta(child, alpha, beta, False, path)
            if child_val > value:
                value = child_val
                best_path = child_path
            alpha = max(alpha, value)
        return value, best_path
    else:
        value = math.inf
```

```
        best_path: List[str] = []
        for child in node.children:
            if value <= alpha:
                PRUNED_SUBTREES.append((child.name, path + [child.name]))
                continue

            child_val, child_path = alpha_beta(child, alpha, beta, True, path)
            if child_val < value:
                value = child_val
                best_path = child_path
            beta = min(beta, value)
        return value, best_path
def build_example_tree() -> Node:
    L_D1 = Node("D1", value=3)
    L_D2 = Node("D2", value=5)
    L_E1 = Node("E1", value=6)
    L_E2 = Node("E2", value=9)

    L_F1 = Node("F1", value=1)
    L_F2 = Node("F2", value=2)
    L_G1 = Node("G1", value=0)
    L_G2 = Node("G2", value=-1)

    L_I1 = Node("I1", value=4)
    L_I2 = Node("I2", value=7)
    L_J1 = Node("J1", value=8)
    L_J2 = Node("J2", value=-2)
    D = Node("D", children=[L_D1, L_D2])  # 5
    E = Node("E", children=[L_E1, L_E2])  # 9

    F = Node("F", children=[L_F1, L_F2])  # 2
    G = Node("G", children=[L_G1, L_G2])  # 0

    I = Node("I", children=[L_I1, L_I2])  # 7
    J = Node("J", children=[L_J1, L_J2])  # 8

    B = Node("B", children=[D, E])        # min(5,9)=5
    C = Node("C", children=[F, G])        # min(2,0)=0
    H = Node("H", children=[I, J])        # min(7,8)=7

    A = Node("A", children=[B, C, H])     # MAX root
    return A
def evaluate_child_without_polluting_trace(child: Node, root_name: str) -> float:

    global EXPANSION_ORDER, PRUNED_SUBTREES
    saved_exp = EXPANSION_ORDER.copy()
    saved_pruned = PRUNED_SUBTREES.copy()
```

```
    # Clear globals for a clean evaluation
    EXPANSION_ORDER.clear()
    PRUNED_SUBTREES.clear()
    val, _ = alpha_beta(child, alpha=-math.inf, beta=math.inf, maximizing_player=False,
path=[root_name])
    EXPANSION_ORDER.clear()
    EXPANSION_ORDER.extend(saved_exp)
    PRUNED_SUBTREES.clear()
    PRUNED_SUBTREES.extend(saved_pruned)
    return val
def run_demo():
    global EXPANSION_ORDER, PRUNED_SUBTREES
    EXPANSION_ORDER = []
    PRUNED_SUBTREES = []

    root = build_example_tree()
    child_values = []
    for child in root.children:
        val = evaluate_child_without_polluting_trace(child, root.name)
        child_values.append((child.name, val))
    best_child_name, best_child_val = max(child_values, key=lambda x: x[1])
    print(f"Best move for MAX: {best_child_name} (value = {best_child_val})\n")
    EXPANSION_ORDER = []
    PRUNED_SUBTREES = []
    root_value, best_path = alpha_beta(root, alpha=-math.inf, beta=math.inf, maximizing_player=True)

    print("Alpha-Beta Pruning Demo (root is MAX):\n")
    print(f"Root value computed: {root_value}")
    print(f"Best path from root to leaf (names): {' -> '.join(best_path)}")

    print("\nNodes expanded in visit order:")
    print(" ", " , ".join(EXPANSION_ORDER))

    if PRUNED_SUBTREES:
        print("\nPruned subtrees (roots and their path from root):")
        for name, path in PRUNED_SUBTREES:
            print(f" Pruned subtree root: {name}, path: {' -> '.join(path)}")
    else:
        print("\nNo pruning occurred for this ordering.")
    # Also display per-child values (useful to see why that child was chosen)
    print("\nImmediate child values at root (MIN nodes evaluated):")
    for name, val in child_values:
        print(f" Child {name}: {val}")

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

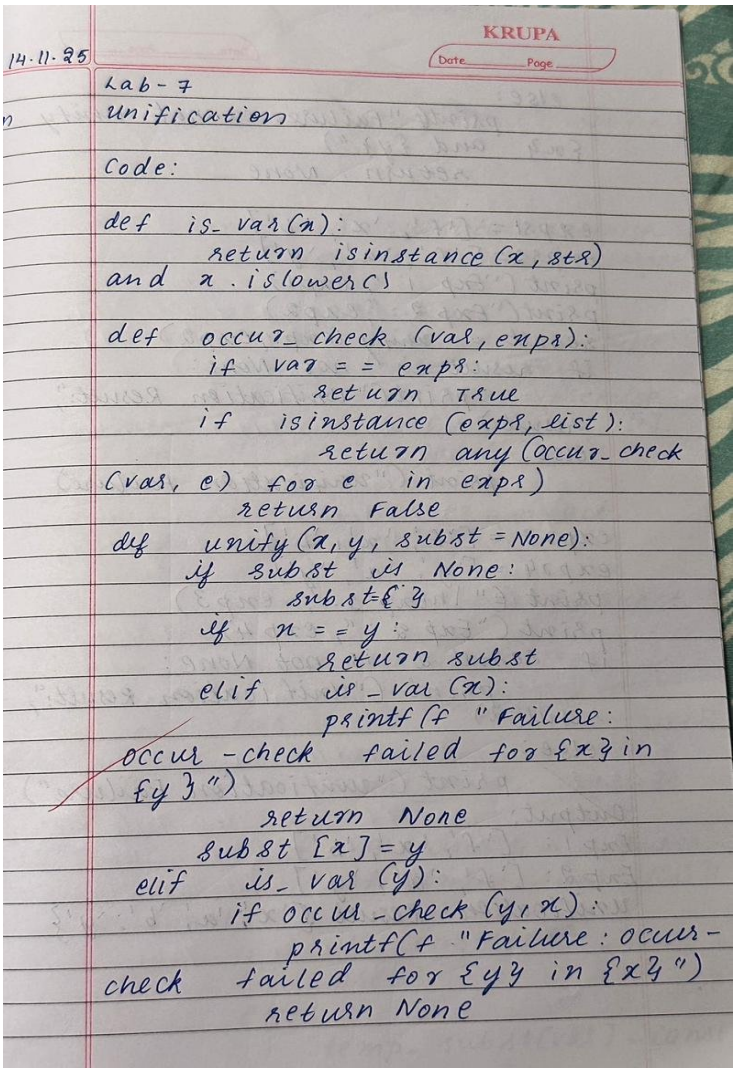
```
Best move for MAX: H (value = 7)

Alpha-Beta Pruning Demo (root is MAX):

Root value computed: 7
Best path from root to leaf (names): A -> H -> I -> I2

Nodes expanded in visit order:
   A , B , D , D1 , D2 , E , E1 , C , F , F1 , F2 , H , I , I1 , I2 , J , J

Pruned subtrees (roots and their path from root):
  Pruned subtree root: E2, path: A -> B -> E -> E2
  Pruned subtree root: G, path: A -> C -> G
  Pruned subtree root: J2, path: A -> H -> J -> J2

Immediate child values at root (MIN nodes evaluated):
  Child B: 5
  Child C: 0
  Child H: 7
```

Output:

Program 8

Implement unification in first order logic

Algorithm:



14.11.25

KRUPA

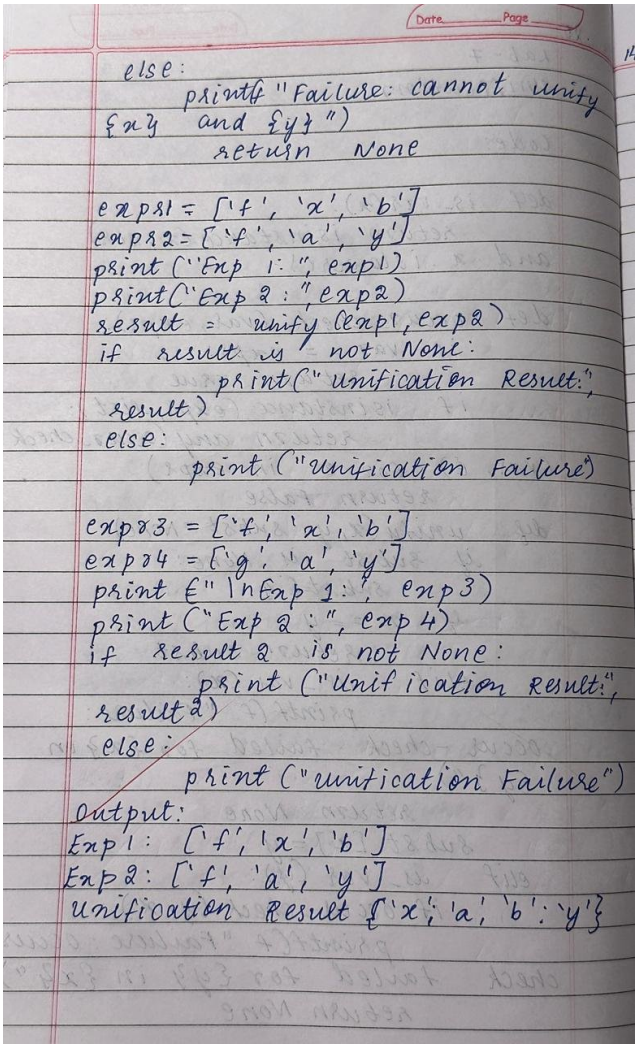
Date Page

Lab - 7

unification

Code:

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

def occur_check (var, exps):
    if var == exps:
        return True
    if isinstance (exps, list):
        return any (occur_check
(var, e) for e in exps)
    return False
dif unify(x, y, subst = None):
    if subst is None:
        subst={}
    if x == y:
        return subst
    elif is_var (x):
        printf (f "Failure:
occur-check failed for {x} in
{y}")
        return None
    subst [x] = y
elif is_var (y):
    if occur_check (y, x):
        printf(f "Failure: occur-
check failed for {y} in {x}")
        return None
```



Date Page

```
    else:
        printf "Failure: cannot unify
{x} and {y}")
        return None

exps1 = ['f', 'x', 'b']
exps2 = ['f', 'a', 'y']
print ("Exp 1: ", exp1)
print("Exp 2: ", exp2)
result = unify (exp1, exp2)
if result is not None:
    print(" unification Result:",
result)
else:
    print ("unification Failure")

exps3 = ['f', 'x', 'b']
exps4 = ['g', 'a', 'y']
print €" \nExp 1: ", exp3)
print ("Exp 2 :", exp 4)
if result 2 is not None:
    print ("Unification Result:",
result2)
else:
    print ("unification Failure")
```

Output:

```
Exp 1: ['f', 'x', 'b']
Exp 2: ['f', 'a', 'y']
Unification Result {'x': 'a', 'b': 'y'}
```

Code:

```
def unify(x, y, substitutions=None):
    if substitutions is None:
        substitutions = {}

    if x == y:
        return substitutions

    # If x is a variable
    if is_variable(x):
        return unify_var(x, y, substitutions)

    # If y is a variable
    if is_variable(y):
        return unify_var(y, x, substitutions)

    # If both are compound terms (like f(a), g(X), etc.)
    if isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x[1]) != len(y[1]):
            return None
        for xi, yi in zip(x[1], y[1]):
            substitutions = unify(apply(substitutions, xi), apply(substitutions, yi), substitutions)
            if substitutions is None:
                return None
        return substitutions
    return None


def unify_var(var, x, substitutions):
    if var in substitutions:
        return unify(substitutions[var], x, substitutions)
    elif occurs_check(var, x, substitutions):
        return None
    else:
        substitutions[var] = x
        return substitutions
```

```
def occurs_check(var, x, substitutions):
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, substitutions) for xi in x[1])
    elif isinstance(x, str) and x in substitutions:
        return occurs_check(var, substitutions[x], substitutions)
    return False


def apply(substitutions, expr):
    if isinstance(expr, str):
        return substitutions.get(expr, expr)
    elif isinstance(expr, tuple):
        return (expr[0], [apply(substitutions, e) for e in expr[1]])
    else:
        return expr


def is_variable(x):
    # A variable is a single lowercase letter (e.g., x, y, z)
    return isinstance(x, str) and len(x) == 1 and x.islower()


def parse(expr):
    expr = expr.replace(" ", "")
    if '(' not in expr:
        return expr
    functor = expr.split('(')[0]
    args = expr[len(functor) + 1:-1]
    parts, depth, start = [], 0, 0
    for i, c in enumerate(args):
        if c == ',' and depth == 0:
            parts.append(args[start:i])
            start = i + 1
        elif c == '(':
            depth += 1
        elif c == ')':
            depth -= 1
    parts.append(args[start:])
    return (functor, [parse(p) for p in parts])


# -------- TEST CASES --------
tests = [
    ("p(b,x,f(g(z)))", "p(z,f(y),f(y))"), # fixed the extra parenthesis
```

```
    ("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)")
]

print(" UNIFICATION RESULTS \n")
for i, (a, b) in enumerate(tests, start=1):
    e1 = parse(a)
    e2 = parse(b)
    result = unify(e1, e2)
    print(f"Q{i}: {a} AND {b}")
    if result is None:
        print("MGU = FAIL\n")
    else:
        print("MGU =", result, "\n")
```

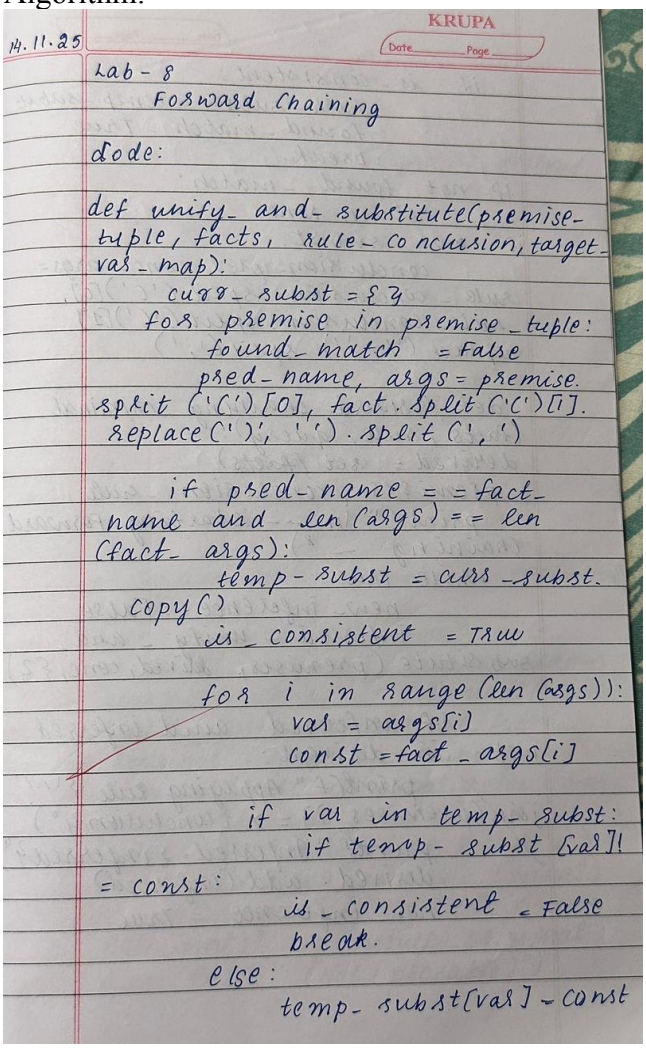
Output:

```
UNIFICATION RESULTS

Q1: p(b,x,f(g(z))) AND p(z,f(y),f(y)) MGU
= {'b': 'z', 'x': ('f', ['y']), 'y': ('g', ['z'])}

Q2: Q(a,g(x,a),f(y)) AND Q(a,g(f(b),a),x)
MGU = {'x': ('f', ['b']), 'y': 'b'}

Q3: p(f(a),g(Y)) AND p(X,X)
MGU = FAIL

Q4: prime(11) AND prime(y)
MGU = {'y': '11'}
Q5: knows(John,x) AND knows(y,mother(y))
MGU = {'y': 'John', 'x': ('mother', ['John'])}

Q6: knows(John,x) AND knows(y,Bill)
MGU = {'y': 'John', 'x': 'Bill'}
```

Program 9

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

Algorithm:



14.11.25

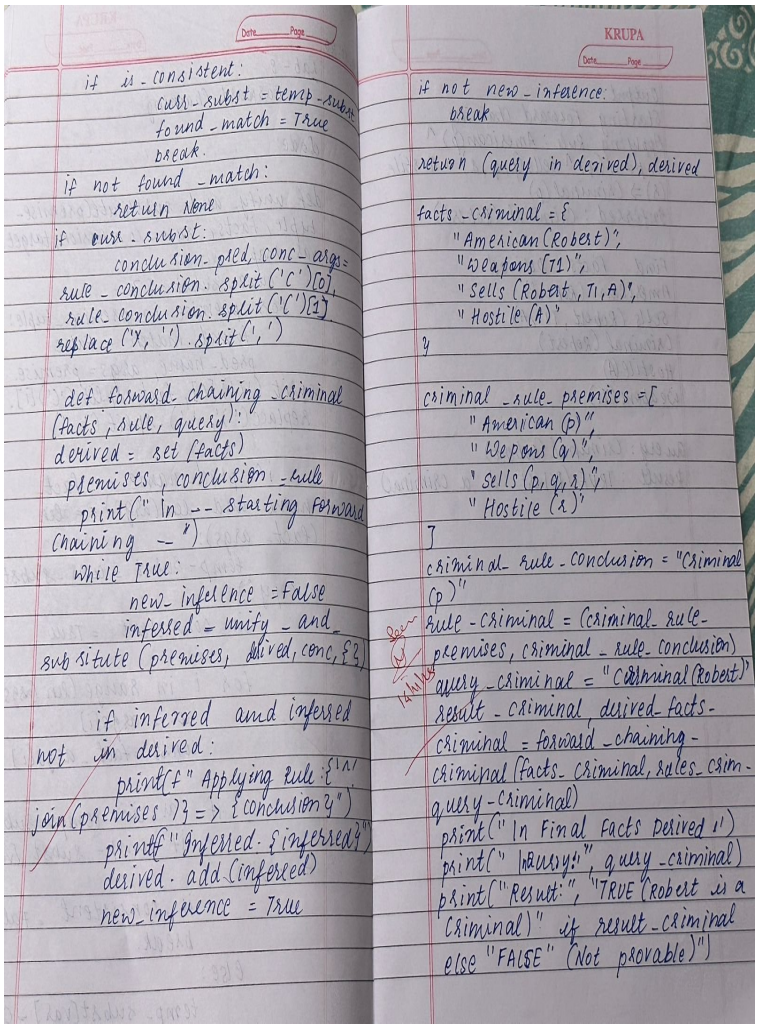
KRUPA

Lab - 8

Forward Chaining

Code:

```
def unify_and_substitute(premise_tuple, facts, rule_conclusion, target_vars_map):
    curr_subst = {}
    for premise in premise_tuple:
        found_match = False
        pred_name, args = premise.split('(')[0], fact.split('(')[1].replace(')', '').split(',')

        if pred_name == fact_name and len(args) == len(fact_args):
            temp_subst = curr_subst.copy()
            is_consistent = True

            for i in range(len(args)):
                var = args[i]
                const = fact_args[i]

                if var in temp_subst:
                    if temp_subst[var] != const:
                        is_consistent = False
                        break
                else:
                    temp_subst[var] = const
```



```
        if is_consistent:
            curr_subst = temp_subst
            found_match = True
            break
    if not found_match:
        return None
    if curr_subst:
        conclusion_pred, conc_args = rule_conclusion.split('(')[0], rule_conclusion.split('(')[1].replace(')', '').split(',')

def forward_chaining_criminal(facts, rule, query):
    derived = set(facts)
    premises, conclusion = rule
    print("\n--- Starting Forward Chaining ---")
    while True:
        new_inference = False
        inferred = unify_and_substitute(premises, derived, conc, {})

        if inferred and inferred not in derived:
            print(f"Applying rule: {' ∧ '.join(premises)} => {conclusion}")
            print(f"Inferred: {inferred}")
            derived.add(inferred)
            new_inference = True
        if not new_inference:
            break

    return (query in derived), derived

facts_criminal = {
    "American(Robert)",
    "Weapons(T1)",
    "Sells(Robert, T1, A)",
    "Hostile(A)"
}

criminal_rule_premises = [
    "American(p)",
    "Weapons(q)",
    "Sells(p, q, r)",
    "Hostile(r)"
]
criminal_rule_conclusion = "Criminal(p)"
rule_criminal = (criminal_rule_premises, criminal_rule_conclusion)
query_criminal = "Criminal(Robert)"
result_criminal, derived_facts_criminal = forward_chaining_criminal(facts_criminal, rule_criminal, query_criminal)
print("\n--- Final Facts Derived ---")
print("Query:", query_criminal)
print("Result:", "TRUE (Robert is a Criminal)" if result_criminal else "FALSE (Not provable)")
```

Code:

```
import re

def match_pattern(pattern, fact):
    pat_pred, pat_args = re.match(r'(\w+)', pattern).groups()
    fact_pred, fact_args = re.match(r'(\w+)', fact).groups()

    # Predicate names must match
    if pat_pred != fact_pred:
        return None

    pat_args = [a.strip() for a in pat_args.split(",")]
    fact_args = [a.strip() for a in fact_args.split(",")]

    if len(pat_args) != len(fact_args):
        return None

    subst = {}
    for p_arg, f_arg in zip(pat_args, fact_args):
        # Variables are lowercase identifiers (e.g., p, x, y)
        if re.fullmatch(r'[a-z]\w*', p_arg):
            subst[p_arg] = f_arg
        elif p_arg != f_arg:
            return None
    return subst

def apply_substitution(expr, subst):
    for var, val in subst.items():
        expr = re.sub(rf'\b{var}\b', val, expr)
    return expr


rules = [
    (["American(p)", "Weapon(q)", "Sells(p,q,r)", "Hostile(r)"], "Criminal(p)"),
    (["Missile(x)"], "Weapon(x)"),
    (["Enemy(x, America)"], "Hostile(x)"),
    (["Missile(x)", "Owns(A, x)"], "Sells(Robert, x, A)")
]
facts =
    { "American(Robert)",
    "Enemy(A, America)",
    "Owns(A, T1)",
    "Missile(T1)"
}

goal = "Criminal(Robert)"
```

```
def forward_chain(rules, facts, goal):
    added = True
    while added:
        added = False
        for premises, conclusion in rules:
            possible_substs = []

            # Check each premise of the rule
            for p in premises:
                matched = False
                for f in facts:
                    subst = match_pattern(p, f)
                    if subst:
                        possible_substs.append(subst)
                        matched = True
                        break
                if not matched:
                    break # one premise fails → skip rule
            else:
                # Combine substitutions into one dictionary
                combined = {}
                for s in possible_substs:
                    combined.update(s)
                new_fact = apply_substitution(conclusion, combined)
                if new_fact not in facts:
                    facts.add(new_fact)
                    print(f"Inferred: {new_fact}")
                    added = True
                    if new_fact == goal:
                        return True
    return goal in facts

print("Goal achieved:", forward_chain(rules, facts, goal))
```

Output:

```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Goal achieved: True
```

Program 10

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

Code:

```
import re
import copy

def _init_(self, predicate_string): self.predicate_string
      = predicate_string.strip()
      self.name, self.arguments, self.negative = self.parse_predicate(self.predicate_string)

   def parse_predicate(self, predicate_string):

      s = predicate_string.strip()
      neg = False
      while s.startswith('~'):
         neg = not neg
         s = s[1:].strip()
      m = re.match(r"^([A-Za-z_][A-Za-z0-9_]*)$", s)
      if not m:
         raise ValueError(f"Invalid predicate: {predicate_string}")
      name, args = m.groups()
      args = [a.strip() for a in args.split(",") if a.strip() != ""]
      return name, args, neg

   def negate(self):
      self.negative = not self.negative
      self.predicate_string = ("~" if self.negative else "") + self.name + "(" + ",".join(self.arguments) +
")"
   def unify_with_predicate(self, other):
      if self.name != other.name or len(self.arguments) != len(other.arguments):
         return False
      subs = {}

      def is_var(x):
         return len(x) > 0 and x[0].islower()

      for a, b in zip(self.arguments, other.arguments):
         if a == b:
            continue
         if is_var(a) and is_var(b):
            if a in subs and subs[a] != b:
               return False
            if b in subs and subs[b] != a:
               return False
            subs[a] = b
         elif is_var(a):
```

```
            if a in subs and subs[a] != b:
                return False
            subs[a] = b
        elif is_var(b):
            if b in subs and subs[b] != a:
                return False
            subs[b] = a
        else:
            return False
    return subs

def substitute(self, subs):
    self.arguments = [subs.get(a, a) for a in self.arguments]
    self.predicate_string = ("~" if self.negative else "") + self.name + "(" + ",".join(self.arguments) +
")"

def _repr_(self):
    return self.predicate_string

def _eq_(self, other):
    return isinstance(other, Predicate) and self.name == other.name and self.arguments ==
other.arguments and self.negative == other.negative
def _hash_(self):
    return hash((self.name, tuple(self.arguments), self.negative))

class Statement:
def _init_(self, statement_string):
    self.statement_string = statement_string.strip()
    self.predicate_set = self.parse_statement(self.statement_string)

def parse_statement(self, statement_string):
    parts = [p.strip() for p in statement_string.split('|') if p.strip() != '']
    predicates = [Predicate(p) for p in parts]
    return frozenset(predicates)

def add_statement_to_KB(self, KB, KB_HASH):
    if self in KB:
        return
    KB.add(self)
    for predicate in self.predicate_set:
        key = predicate.name
        if key not in KB_HASH:
            KB_HASH[key] = set()
        KB_HASH[key].add(self)
def get_resolving_clauses(self, KB_HASH):
    resolving_clauses = set()
    for predicate in self.predicate_set:
```

```
            key = predicate.name
            if key in KB_HASH:
                resolving_clauses |= KB_HASH[key]
        return resolving_clauses

    def resolve(self, other):
        new_statements = set()
        for p1 in self.predicate_set:
            for p2 in other.predicate_set:
                if p1.name == p2.name and p1.negative != p2.negative:
                    subs = p1.unify_with_predicate(p2)
                    if subs is False:
                        continue
                    new_pred_set = set()
                    for pred in (set(self.predicate_set) | set(other.predicate_set)):
                        if pred == p1 or pred == p2:
                            continue
                        pred_copy = copy.deepcopy(pred)
                        pred_copy.substitute(subs)
                        new_pred_set.add(pred_copy)
                    if not new_pred_set:
                        return False # contradiction (empty clause)
                    sorted_preds = sorted([str(p) for p in new_pred_set])
                    new_stmt = Statement('|'.join(sorted_preds))
                    new_statements.add(new_stmt)
        return new_statements
    def _repr_(self):
        sorted_preds = sorted([str(p) for p in self.predicate_set])
        return '|'.join(sorted_preds)

    def _eq_(self, other):
        return isinstance(other, Statement) and self.predicate_set == other.predicate_set

    def _hash_(self):
        return hash(self.predicate_set)

def fol_to_cnf_clauses(sentence):

    sentence = sentence.replace(' ', '')
    if '=>' in sentence:
        lhs, rhs = sentence.split('=>')
        parts = [p for p in lhs.split('&') if p != '']
        negated_lhs = []
        for p in parts:
            if p.startswith('~'):
                negated_lhs.append(p[1:])
            else:
```

```
            negated_lhs.append('~' + p)
        disjunction = '|'.join(negated_lhs + [rhs])
        return [disjunction]
    if '&' in sentence:
        return [c for c in sentence.split('&') if c != '']
    return [sentence]

KILL_LIMIT = 8000

def prepare_knowledgebase(fol_sentences):
    KB = set()
    KB_HASH = {}
    for sentence in fol_sentences:
        clauses = fol_to_cnf_clauses(sentence)
        for clause in clauses:
            stmt = Statement(clause)
            stmt.add_statement_to_KB(KB, KB_HASH)
    return KB, KB_HASH
def FOL_Resolution(KB, KB_HASH, query, verbose=True):
    KB = set(KB)
    KB_HASH = {k: set(v) for k, v in KB_HASH.items()}

    query.add_statement_to_KB(KB, KB_HASH)
    if verbose:
        print("\nInitial KB clauses (including negated query):")
        for s in KB:
            print(" ", s)

    iterations = 0
    while True:
        iterations += 1
        if len(KB) > KILL_LIMIT:
            if verbose:
                print("Reached KILL_LIMIT, stopping.")
            return False
        new_statements = set()
        kb_list = list(KB)
        for s1 in kb_list:
            for s2 in s1.get_resolving_clauses(KB_HASH):
                if s1 == s2:
                    continue
                resolvents = s1.resolve(s2)
                if resolvents is False:
                    if verbose:
                        print(f"\nCONTRADICTION derived by resolving:\n {s1}\n {s2}\n=> empty clause
(proof found).")
                    return True
```

```
                new_statements |= resolvents
        if new_statements.issubset(KB):
            if verbose:
                print("\nNo new clauses derived. Resolution failed (query not proved).")
            return False
        added = new_statements - KB
        if verbose and added:
            print(f"\nIteration {iterations}: Derived {len(added)} new clause(s):")
            for st in sorted(added, key=lambda x: str(x)):
                print(" ", st)
        for st in added:
            st.add_statement_to_KB(KB, KB_HASH)
def main():
    fol_sentences = [
        "Food(x) => Likes(John,x)",
        "Food(Apple)",
        "Food(Vegetables)",
        "Eats(p,y)&~Killed(y) => Food(y)",
        "Eats(Anil,Peanuts)",
        "Alive(Anil)",
        "Eats(Anil,y) => Eats(Harry,y)",
        "Alive(x) => ~Killed(x)",
        "~Killed(x) => Alive(x)"
    ]

    goal = "Likes(John,Peanuts)"
    KB, KB_HASH = prepare_knowledgebase(fol_sentences)
    neg_goal_pred = Predicate(goal)
    neg_goal_pred.negate()
    neg_goal_stmt = Statement(str(neg_goal_pred))

    proved = FOL_Resolution(KB, KB_HASH, neg_goal_stmt, verbose=True)

    print("\nResult: Query 'John likes peanuts' is", "TRUE (proved)" if proved else "FALSE (not
proved)")

if _name____== "_main_":
    main()
```

Output:

```
Initial KB clauses (including negated query):
  Alive(x)|Killed(x)
  ~Alive(x)|~Killed(x)
  ~Likes(John,Peanuts)
  Food(Apple)
  Alive(Anil)
```

```
   Eats(Anil,Peanuts)
   Food(Vegetables)
   Likes(John,x)|~Food(x)
   Eats(Harry,y)|~Eats(Anil,y)
   Food(y)|Killed(y)|~Eats(p,y)

Iteration 1: Derived 13 new clause(s):
    Alive(x)|~Alive(x)
    Eats(Harry,Peanuts)
    Food(Peanuts)|Killed(Peanuts)
    Food(x)|~Alive(x)|~Eats(p,x)
    Food(y)|Killed(y)|~Eats(Anil,y)
    Food(y)|~Alive(y)|~Eats(p,y)
    Killed(x)|Likes(John,x)|~Eats(p,x)
    Killed(x)|~Killed(x)
    Killed(y)|Likes(John,y)|~Eats(p,y)
    Likes(John,Apple)
    Likes(John,Vegetables)
    ~Food(Peanuts)
    ~Killed(Anil)
Iteration 2: Derived 23 new clause(s):
    Alive(x)
    Food(Anil)|~Eats(Anil,Anil)
    Food(Anil)|~Eats(p,Anil)
    Food(Peanuts)|~Alive(Peanuts)
    Food(x)|Killed(x)|~Eats(Anil,x)
    Food(x)|Killed(x)|~Eats(p,x)
    Food(x)|~Alive(x)|~Eats(Anil,x)
    Food(x)|~Eats(p,x)
    Food(y)|~Alive(y)|~Eats(Anil,y)
    Killed(Peanuts)
    Killed(Peanuts)|Likes(John,Peanuts)
    Killed(Peanuts)|~Eats(Anil,Peanuts)
    Killed(Peanuts)|~Eats(p,Peanuts)
    Killed(x)
    Killed(x)|Likes(John,x)|~Eats(Anil,x)
    Killed(y)|Likes(John,y)|~Eats(Anil,y)
    Likes(John,Anil)|~Eats(p,Anil)
    Likes(John,x)|~Alive(x)|~Eats(p,x)
    Likes(John,x)|~Eats(p,x)
    Likes(John,y)|~Alive(y)|~Eats(p,y)
    ~Alive(Peanuts)|~Eats(p,Peanuts)
    ~Alive(x)
    ~Killed(x)

CONTRADICTION derived by resolving:
   Killed(x)
```

Killed(x)|~Killed(x)
=> empty clause (proof found).

Result: Query 'John likes peanuts' is TRUE (proved)