VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Soham Arora (1BM23CS334)

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 **Soham Arora (1BM23CS334),** 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.

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

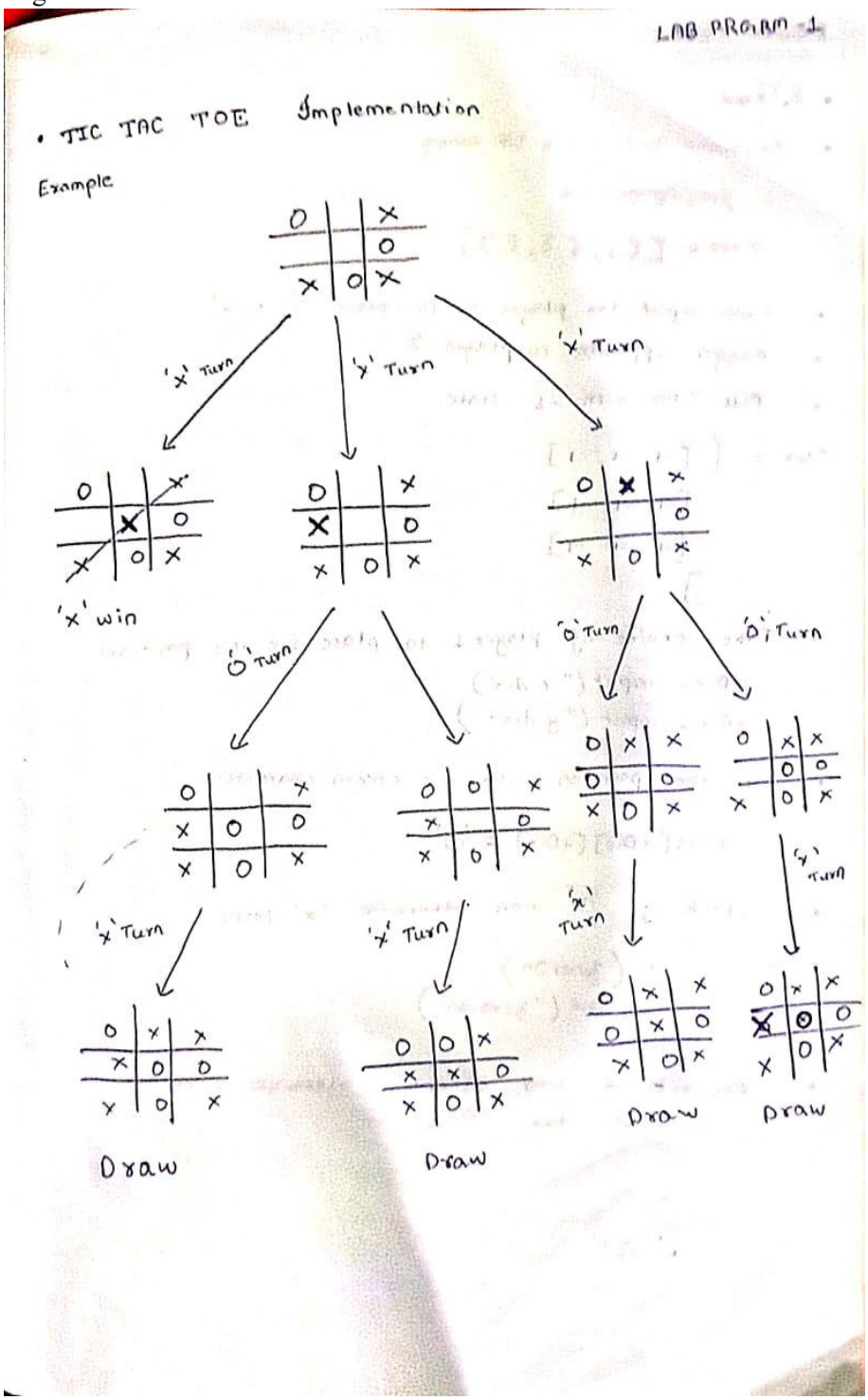
Index

Github Link:https://github.com/sohamArora1605/1BM23CS334AI

Program 1

Implement Tic –Tac –Toe Game Implement
vacuum cleaner agent

Algorithm:



Pseudo Code :

- ~~Start~~
- Implement 3×3 empty 2D array

~~for(i = 0 → 2~~

Tiles = [[ ], [ ], [ ]]

- Take input for player 1 to choose 'O' or 'X'
- Assign opposite to player 2
- Fill Tiles with -1 state

```
Tiles = [ [-1, -1, -1]
          [-1  -1  -1]
          [-1  -1  -1]
        ]
```

- Take choice of player 1 to place i.e it's position

```
xDir = input("x dir")
yDir = input("y dir:")
```

- Fill the position with his chosen character

```
Tiles[xDir][yDir] = 'O'
```

- Check if 'O' won otherwise 'X' turns

```
if (zeroOn)
    print("zerowon:")
```

- Restart if won otherwise alternate turn

draw

1808

- An algorithm for vaccum cleaning for 4 rooms
  - Input :
    - Number of rooms = 4
    - Each room = a grid
    - Dirt there in room/not for each room
    - Starting position
  - Output
    - All dirt in all 4 rooms cleaned

- Pseudo Code :

Start robot at base wherever the robot got input for initial position

For each room in 1 to 4 :
  while (!clean) :
    check options go Left, Right, up, down
    or suck dirt if it's there

    check for boundary condition and loop back if it's on edge

    if dirt is cleaned :
      clean += 1

  if (clean == 4) :
    break

Output :

Enter initial dirt status for rooms (0 for clean, 1 for dirty) separated by spaces 0 0 0 0

Initial state of rooms :
Room 1 is clean ← VC here
Room 2 is clean
Room 3 is dirty
Room 4 is clean

Choose an action:

1. Go left
2. Go right
3. Suck dirt

Enter action: 2

Moving right.

Current state

ROOM1 is clean
ROOM2 is clean ← VC
ROOM3 is dirty
ROOM4 is clean

Choose an action:

1. Go left
2. Go right
3. Suck dirt

Enter action: 2

moving right

Current state

ROOM1 is clean
ROOM2 is clean
ROOM 3 is dirty ← VC
ROOM 4 is clean

Choose

3.

All rooms cleaned

Total cost: ~~2~~ 2 (No. of rooms with dirt is cost)

1BM23CS334

26/8/26

Open in Colab

In [ ]:
```python
def create_board():
    return [' ' for _ in range(9)]  # Flat list, indices 0-8


def display_board(board):
    for i in range(3):
        print(board[i*3] + " | " + board[i*3+1] + " | " + board[i*3+2])
        if i < 2:
            print("--+---+--")

# Check for winner
def check_winner(board, player):
    win_conditions = [
        [0, 1, 2], [3, 4, 5], [6, 7, 8],  # rows
        [0, 3, 6], [1, 4, 7], [2, 5, 8],  # columns
        [0, 4, 8], [2, 4, 6]              # diagonals
    ]
    for condition in win_conditions:
        if all(board[i] == player for i in condition):
            return True
    return False

# Check for draw
def is_draw(board):
    return ' ' not in board

# Switch player
def switch_player(player):
    return 'O' if player == 'X' else 'X'
# Recursive minimax function to calculate cost
def minimax(board, is_maximizing, player):
    opponent = switch_player(player)

    if check_winner(board, player):
        return 1  # Win for current player
    elif check_winner(board, opponent):
        return -1  # Loss for current player
    elif is_draw(board):
        return 0  # Draw

    if is_maximizing:
        best_score = -float('inf')
        for i in range(9):
            if board[i] == ' ':
                board[i] = player
                score = minimax(board, False, player)
                board[i] = ' '
                best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
```

```
        for i in range(9):
            if board[i] == ' ':
                board[i] = opponent
                score = minimax(board, True, player)
                board[i] = ' '
                best_score = min(score, best_score)
        return best_score
# Example: calculating cost from an empty board for player 'X'
board = create_board()
player = 'X'

print("Initial Board:")
display_board(board)

cost = minimax(board, True, player)
print(f"Cost of best move for player {player}: {cost}")
```

```
Initial Board:
  |   |
--+---+--
  |   |
--+---+--
  |   |
Cost of best move for player X: 0
```

In [ ]:

```
def play_game():
    board = create_board()
    player1_symbol = input("Player 1, choose your symbol (X or O): ").upper()
    while player1_symbol not in ['X', 'O']:
        player1_symbol = input("Invalid input. Choose X or O: ").upper()

    player2_symbol = 'O' if player1_symbol == 'X' else 'X'
    print(f"Player 2, your symbol is: {player2_symbol}")

    current_player_symbol = 'X'
    print("Soham Arora 1BM23CS334")

    while True:
        display_board(board)
        print(f"It's {current_player_symbol}'s turn.")

        try:
            move = int(input("Enter your move (1-9): ")) - 1
            if move < 0 or move > 8 or board[move] != ' ':
                print("Invalid move. Try again.")
                continue
        except ValueError:
            print("Invalid input. Enter a number from 1 to 9.")
            continue

        board[move] = current_player_symbol
```

```
        if check_winner(board, current_player_symbol):
            display_board(board)
            print(f"Player {current_player_symbol} wins!")
            break
        elif is_draw(board):
            display_board(board)
            print("It's a draw!")
            break

        current_player_symbol = switch_player(current_player_symbol)


play_game()
```

```
Player 1, choose your symbol (X or O): O
Player 2, your symbol is: X
Soham Arora 1BM23CS334
  |   |
--+---+--
  |   |
--+---+--
  |   |
It's X's turn.
Enter your move (1-9): 5
  |   |
--+---+--
  | X |
--+---+--
  |   |
It's O's turn.
Enter your move (1-9): 7
  |   |
--+---+--
  | X |
--+---+--
O |   |
It's X's turn.
Enter your move (1-9): 1
X |   |
--+---+--
  | X |
--+---+--
O |   |
It's O's turn.
Enter your move (1-9): 8
X |   |
--+---+--
  | X |
--+---+--
O | O |
It's X's turn.
Enter your move (1-9): 9
X |   |
--+---+--
  | X |
--+---+--
O | O | X
Player X wins!
```

Open in Colab

In [ ]:

New Section

In [ ]:
```
import random

rooms = []
room_names = ["Room 1", "Room 2", "Room 3", "Room 4"]

while True:
    user_input = input("Enter initial dirt status for 5 rooms (0 for clean, 1
    dirt_status_str = user_input.split()
    if len(dirt_status_str) == 5:
        try:
            dirt_status = [int(status) for status in dirt_status_str]
            if all(status in [0, 1] for status in dirt_status):
                break
            else:
                print("Error: Invalid input. Please enter only 0 or 1.")
        except ValueError:
            print("Error: Invalid input. Please enter numbers separated by spa
    else:
        print("Error: Invalid input. Please enter 5 numbers, either 0 or 1, se

for i, name in enumerate(room_names):
    is_dirty = dirt_status[i] == 1
    rooms.append({"name": name, "is_dirty": is_dirty})

current_room_index = 0
cost = 0

def are_all_rooms_clean(rooms):
    for room in rooms:
        if room["is_dirty"]:
            return False
    return True
def display_state(rooms, current_room_index):
    print("\nCurrent state of the rooms:")
    for i, room in enumerate(rooms):
        status = "dirty" if room["is_dirty"] else "clean"
        position = "<-- VC Here" if i == current_room_index else ""
        print(f"{room['name']} is {status} {position}")

print("Initial state of the rooms:")
display_state(rooms, current_room_index)

print("\nVacuum cleaner in action:")
while not are_all_rooms_clean(rooms):
    display_state(rooms, current_room_index)
```

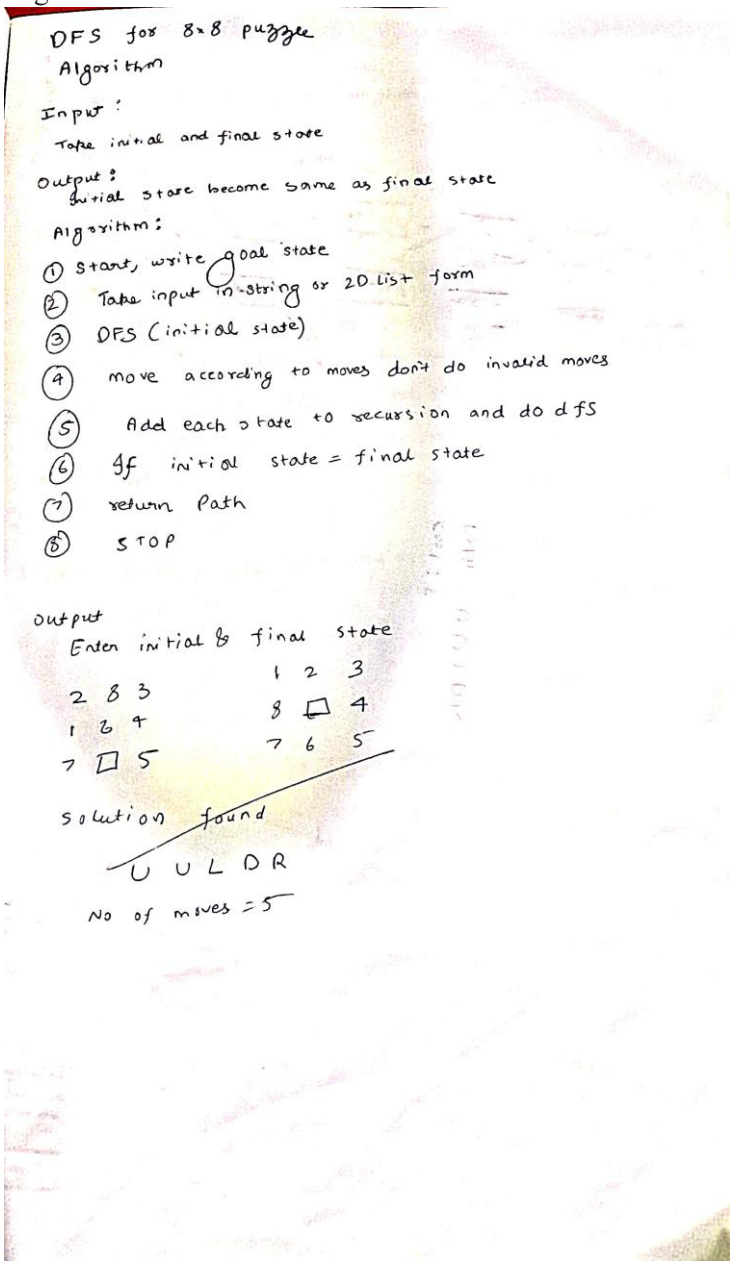
```
    print("\nChoose an action:")
    print("1. Go Left")
    print("2. Go Right")
    print("3. Suck Dirt")

    action = input("Enter action number (1, 2, or 3): ")

    if action == '1':
        if current_room_index > 0:
            current_room_index -= 1
            cost += 1
            print("Moving left.")
        else:
            print("Cannot move left from the first room.")
    elif action == '2':
        if current_room_index < len(rooms) - 1:
            current_room_index += 1
            cost += 1
            print("Moving right.")
        else:
            print("Cannot move right from the last room.")
    elif action == '3':
        if rooms[current_room_index]["is_dirty"]:
            print(f"Sucking dirt in {rooms[current_room_index]['name']}.")
            rooms[current_room_index]["is_dirty"] = False
            cost += 1
        else:
            print(f"{rooms[current_room_index]['name']} is already clean.")
    else:
        print("Invalid action. Please enter 1, 2, or 3.")


print("\nFinal state of the rooms:")
display_state(rooms, current_room_index)

print("\nAll rooms are now clean!")
print(f"Total cost (number of actions): {cost}")

print("\nSoham Arora")
print("\n1BM23CS334")
```

```
Enter initial dirt status for 5 rooms (0 for clean, 1 for dirty), separated by
spaces: 0 0 0 1 0
Initial state of the rooms:

Current state of the rooms:
Room 1 is clean <-- VC Here
Room 2 is clean
Room 3 is clean
Room 4 is dirty
Room 5 is clean

Vacuum cleaner in action:

Current state of the rooms:
Room 1 is clean <-- VC Here
Room 2 is clean
Room 3 is clean
Room 4 is dirty
Room 5 is clean

Choose an action:
1. Go Left
2. Go Right
3. Suck Dirt
Enter action number (1, 2, or 3): 2
Moving right.

Current state of the rooms:
Room 1 is clean
Room 2 is clean <-- VC Here
Room 3 is clean
Room 4 is dirty
Room 5 is clean

Choose an action:
1. Go Left
2. Go Right
3. Suck Dirt
Enter action number (1, 2, or 3): 2
Moving right.

Current state of the rooms:
Room 1 is clean
Room 2 is clean
Room 3 is clean <-- VC Here
Room 4 is dirty
Room 5 is clean

Choose an action:
1. Go Left
2. Go Right
3. Suck Dirt
Enter action number (1, 2, or 3): 2
Moving right.
```

Program 2

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

Algorithm:



Deep iterative BFS

- Input
  Take initial, final state as well as 'd' (depth level)
- Output
  Initial state and final state will be same

Algorithm

- Start, write final state
- Take input instring or 2D list form
- BFS (initial state), ~~do~~ till 'd' level
- ~~if~~ move according to moves don't do invalid moves
- ~~Add~~ If initial state = final state
- return path
- stop

Output Enter initial & final state $d = 5$

2 8 3	1 2 3
1 6 4	8 □ 4
7 □ 5	7 6 5

Solution found

U U L D R

No of moves = 5

In [4]:
```
goal_state = input("Enter goal state : ")

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_li
    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, [])]

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

        if current_state in visited:
            continue

        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 start state : ")

if len(start) == 9 and set(start) == set('012345678'):
    print("Start 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))
        print("1BM23CS334 Soham Arora\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 reach
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 withc
```

```
Streaming output truncated to the last 5000 lines.
Visited state:
8 1
2 7 3
5 6 4

Visited state:
8   1
2 7 3
5 6 4

Visited state:
8 1 3
2 7 4
  5 6

Visited state:
8   3
2 1 4
5 7 6

Visited state:
8 3
2 1 4
5 7 6

Visited state:
  8 3
2 1 4
5 7 6

Visited state:
2 8 3
  1 4
5 7 6

Visited state:
2 8 3
1   4
5 7 6

Visited state:
2 8 3
1 7 4
5   6

Visited state:
2   3
1 8 4
5 7 6

Visited state:
2 8 3
5 1 4
```

```
1 8 4
7 6 5

Visited state:
2 3 4
1 8
7 6 5

Visited state:
2 3 4
1   8
7 6 5

Visited state:
2 3 4
  1 8
7 6 5

Visited state:
2 3 4
1 6 8
7   5

Visited state:
2   4
1 3 8
7 6 5

Visited state:
2 3 4
1 8 5
7 6

Visited state:
2 3 4
1 8 5
7   6

Visited state:
  2 3
1 8 4
7 6 5

Visited state:
1 2 3
  8 4
7 6 5

Visited state:
1 2 3
8   4
7 6 5

Solution found!
```

In [5]:

```
import copy
goal_state = [[1, 2, 3],
              [8, 0, 4],
              [7, 6, 5]]


moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

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

def is_goal(state):
    return state == goal_state

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

def state_to_tuple(state):
    return tuple(tuple(row) for row in state)

def print_state(state):
    for row in state:
        print(" ".join(str(num) if num != 0 else "_" for num in row))
    print("----------")

def dfs_limited(state, path, depth, limit, visited, current_depth):
    if depth == limit:
        return None

    print_state(state)

    if is_goal(state):
        return path + [state]

    visited.add(state_to_tuple(state))

    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, v
            if result:
                return result
```

```
=== 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
```

```
    return None

def iddfs(start_state, max_depth=50):
    for limit in range(max_depth + 1):
        print(f"=== Iteration with depth limit {limit} ===")
        visited = set()
        path = dfs_limited(start_state, [], 0, limit, visited, 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)
            return
    print("No solution found within depth limit.")

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

iddfs(start_state, max_depth=20)
print("Soham Arora - 1BM23CS334")
```

```
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
```

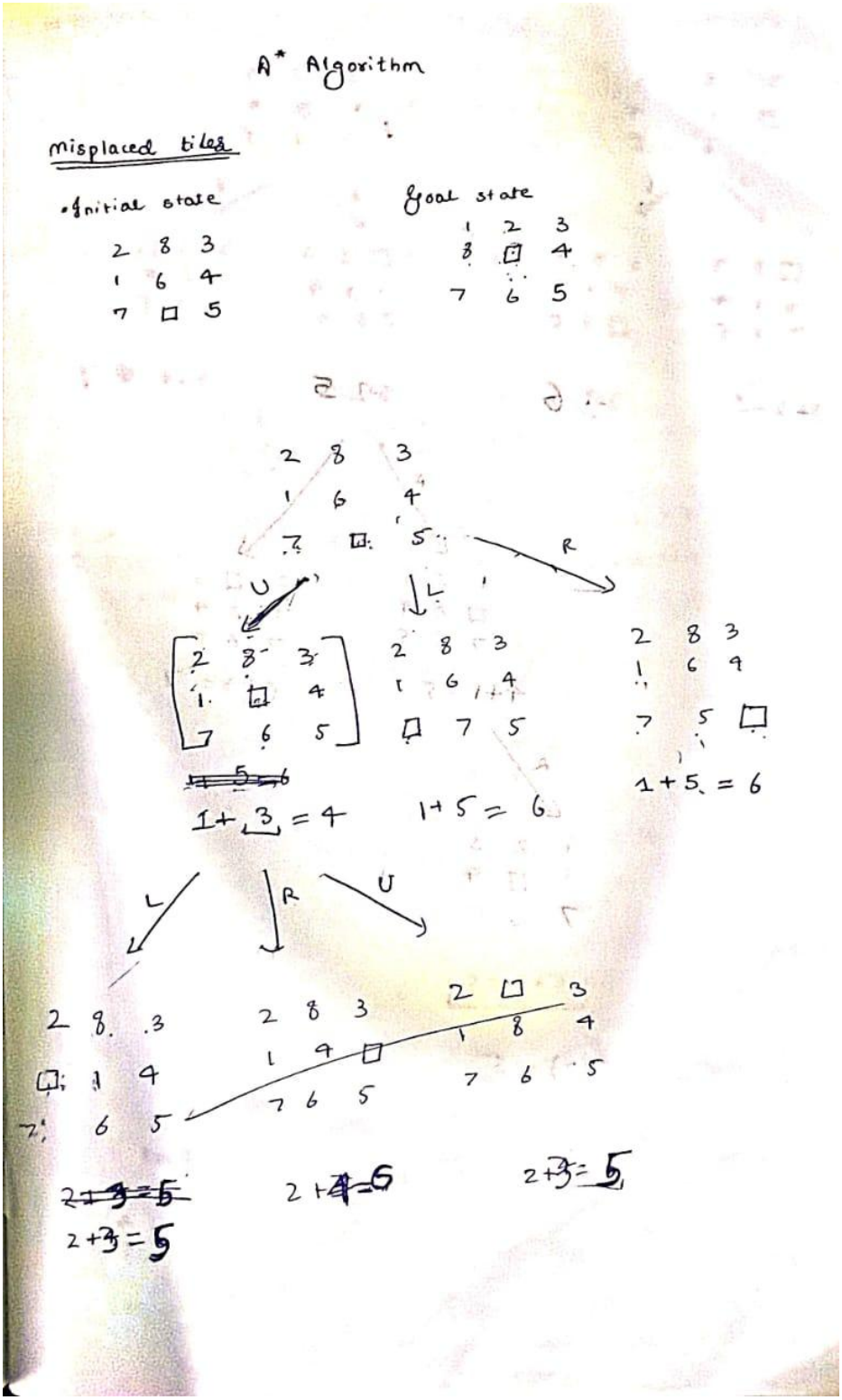
```
7 5 _
----------
2 8 3
1 6 _
7 5 4
----------
=== Iteration with depth limit 4 ===
2 8 3
1 6 4
7 _ 5
----------
2 8 3
1 _ 4
7 6 5
----------
2 _ 3
1 8 4
7 6 5
----------
_ 2 3
1 8 4
7 6 5
----------
2 3 _
1 8 4
7 6 5
----------
2 8 3
_ 1 4
7 6 5
----------
_ 8 3
2 1 4
7 6 5
----------
2 8 3
7 1 4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

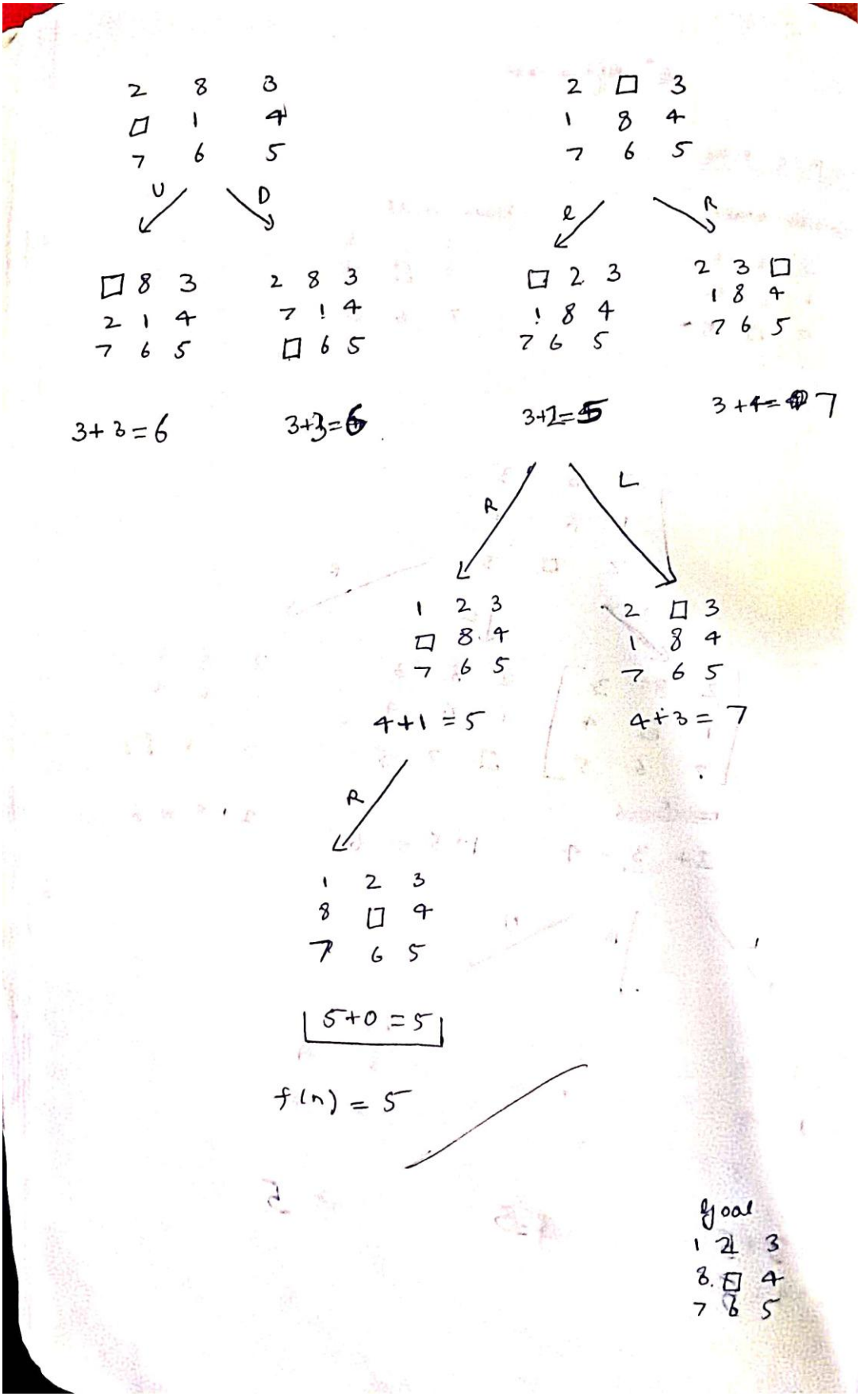
```
=== Iteration with depth limit 6 ===
2 8 3
1 6 4
7 _ 5
----------
2 8 3
1 _ 4
7 6 5
----------
2 _ 3
1 8 4
7 6 5
----------
_ 2 3
1 8 4
7 6 5
----------
1 2 3
_ 8 4
7 6 5
----------
1 2 3
7 8 4
_ 6 5
----------
1 2 3
8 _ 4
7 6 5
----------
Goal reached!
Visited: 6
Solution depth: 5
Steps:
2 8 3
1 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
```

Program 3

Implement A* search algorithm

Algorithm:




2 8 3
□ 1 4
7 6 5
U
D
□ 8 3
2 1 4
7 6 5
2 8 3
7 ! 4
□ 6 5
3+3=6
3+3=6
2 □ 3
1 8 4
7 6 5
L
R
□ 2 3
! 8 4
7 6 5
2 3 □
1 8 4
7 6 5
3+1=5
3+4=7
R
L
1 2 3
□ 8 4
7 6 5
4+1=5
2 □ 3
1 8 4
7 6 5
4+3=7
R
1 2 3
8 □ 4
7 6 5
5+0=5
f(n) = 5
goal
1 2 3
8 □ 4
7 6 5

Algorithm

A* search emulates nodes by combining $g(n)$, the cost reach the node & $h(n)$, the cost get from node to goal

- $f(n) = g(n) + h(n)$
- $f(n)$ is evaluation function which gives cheapest solution cost
- $g(n)$ is exact cost to reach node n from initial state
- $h(n)$ is an estimation of assumed cost from current state to reach goal

8/9

In [8]:

```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calcu
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in ei
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):
        """ Move the blank space in the given direction and if the position va
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data)
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None


    def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
        """ Specifically used to find the position of the blank space """
        for i in range(0,len(self.data)):
            for j in range(0,len(self.data)):
                if puz[i][j] == x:
                    return i,j
```

```
            """ If the difference between current and goal node is 0 we have r
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)


puz = Puzzle(3)
puz.process()
```

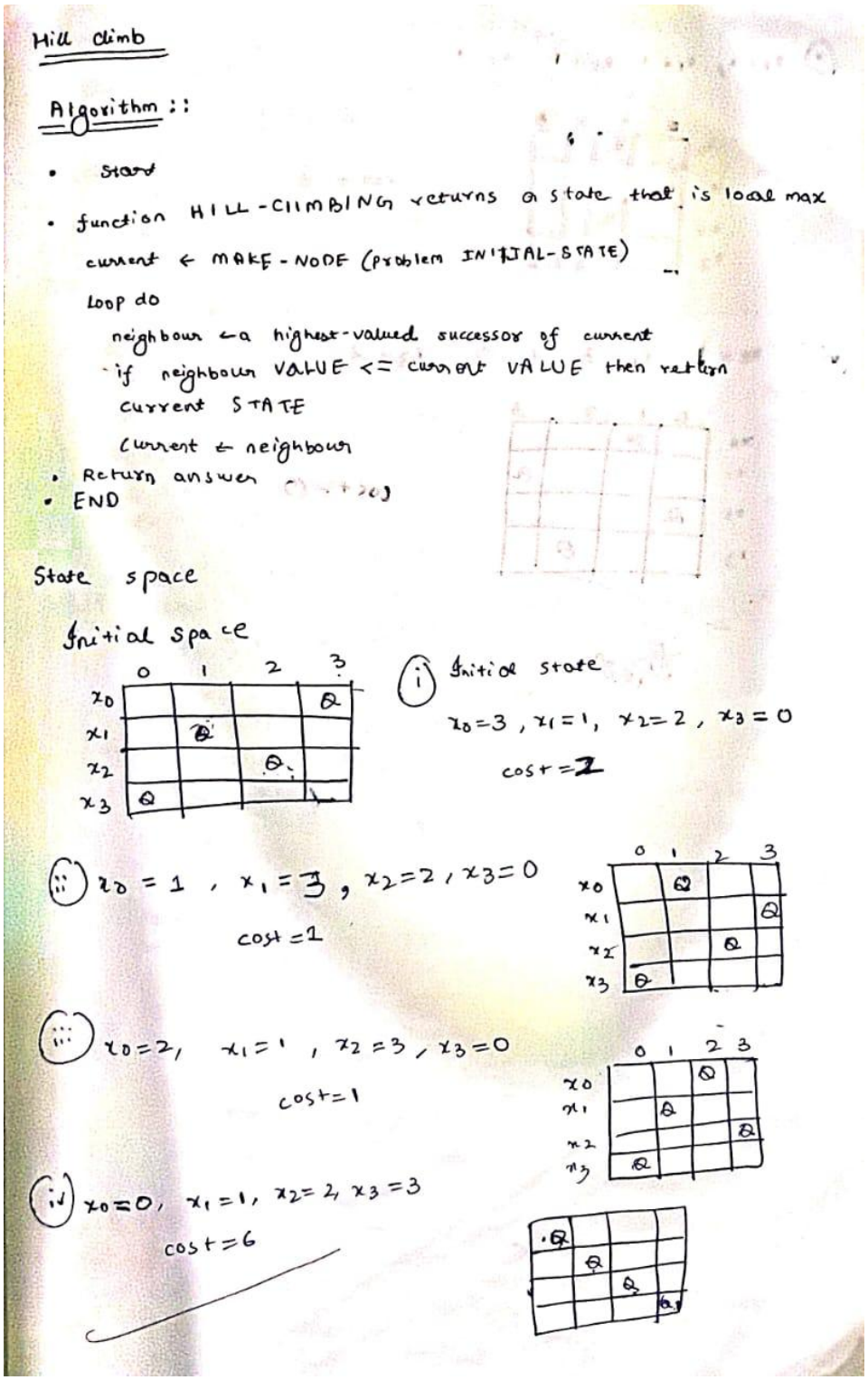
```
Soham arora 1bm23cs334
Enter the start state matrix

2 8 3
1 6 4
7 _ 5
Enter the goal state matrix

1 2 3
8 _ 4
7 6 5




  |
  |
 \'/

2 8 3
1 6 4
7 _ 5

  |
  |
 \'/

2 8 3
1 _ 4
7 6 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

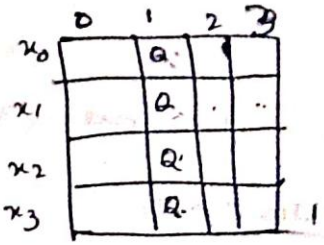
Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

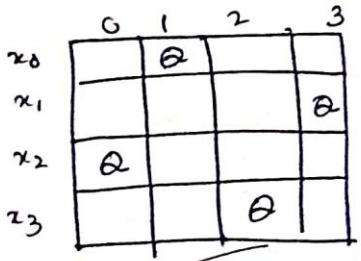


(v) $x_0 = 1,\ x_1 = 1,\ x_2 = 1,\ x_3 = 1$



cost = 6

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



cost = 0

15.09

In [1]:

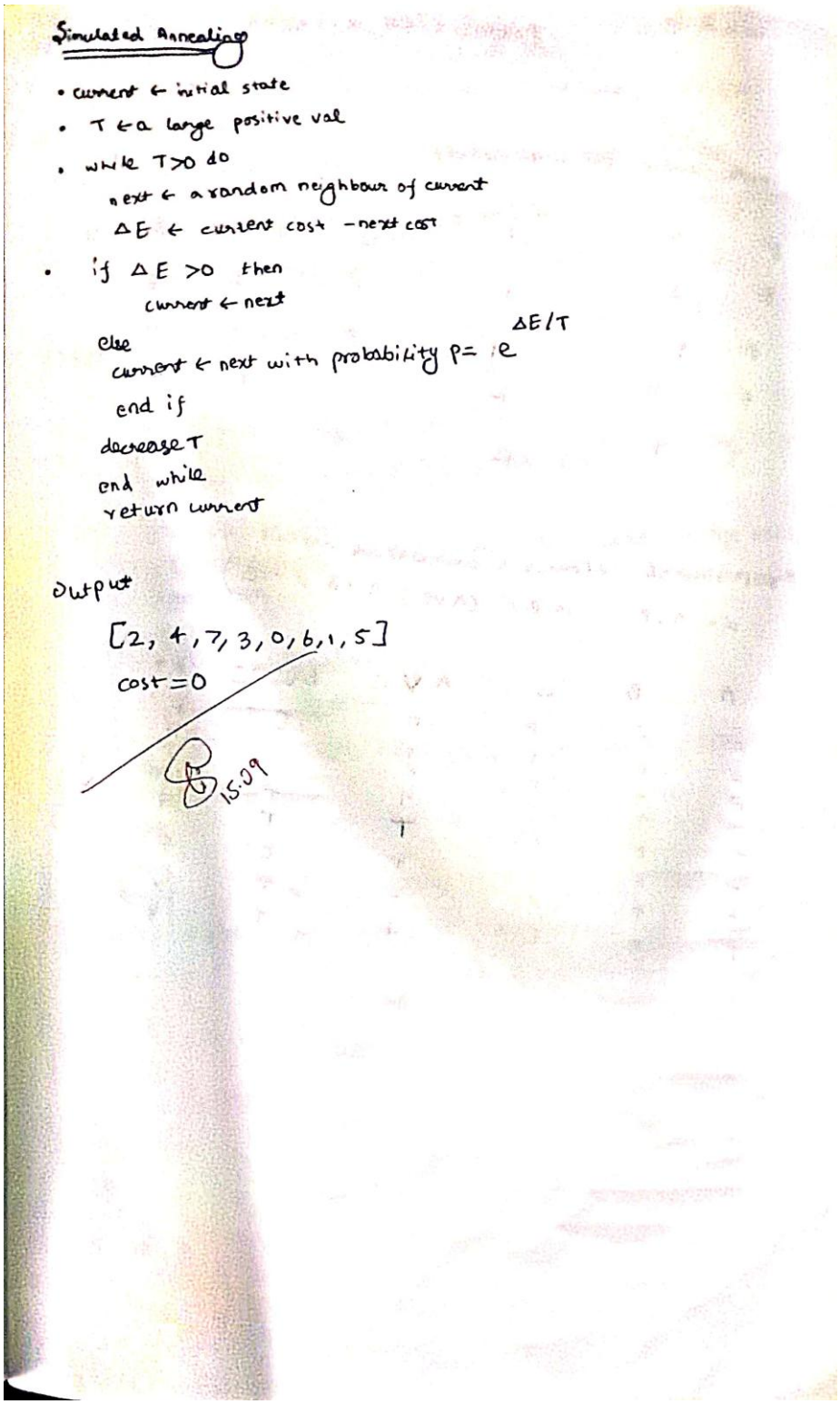
```python
import random

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

def generate_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = list(state)
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors

def hill_climbing(initial_state):
    print("1bm23cs334 soham arora")
    current = initial_state
    steps = []
    step_count = 0
    while True:
        current_cost = calculate_cost(current)
        steps.append((current, current_cost))
        print(f"Step {step_count}: State={current}, Cost={current_cost}") # Pr
        step_count += 1

        neighbors = generate_neighbors(current)
        neighbor_costs = [(n, calculate_cost(n)) for n in neighbors]


        best_neighbor, best_cost = min(neighbor_costs, key=lambda x: x[1])

        if best_cost >= current_cost:
            break

        current = best_neighbor

    return steps

initial_state = [0, 1, 2, 3]
steps = hill_climbing(initial_state)
```

```
1bm23cs334 soham arora
Step 0: State=[0, 1, 2, 3], Cost=6
Step 1: State=[1, 1, 2, 3], Cost=4
Step 2: State=[1, 0, 2, 3], Cost=2
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



In [ ]:

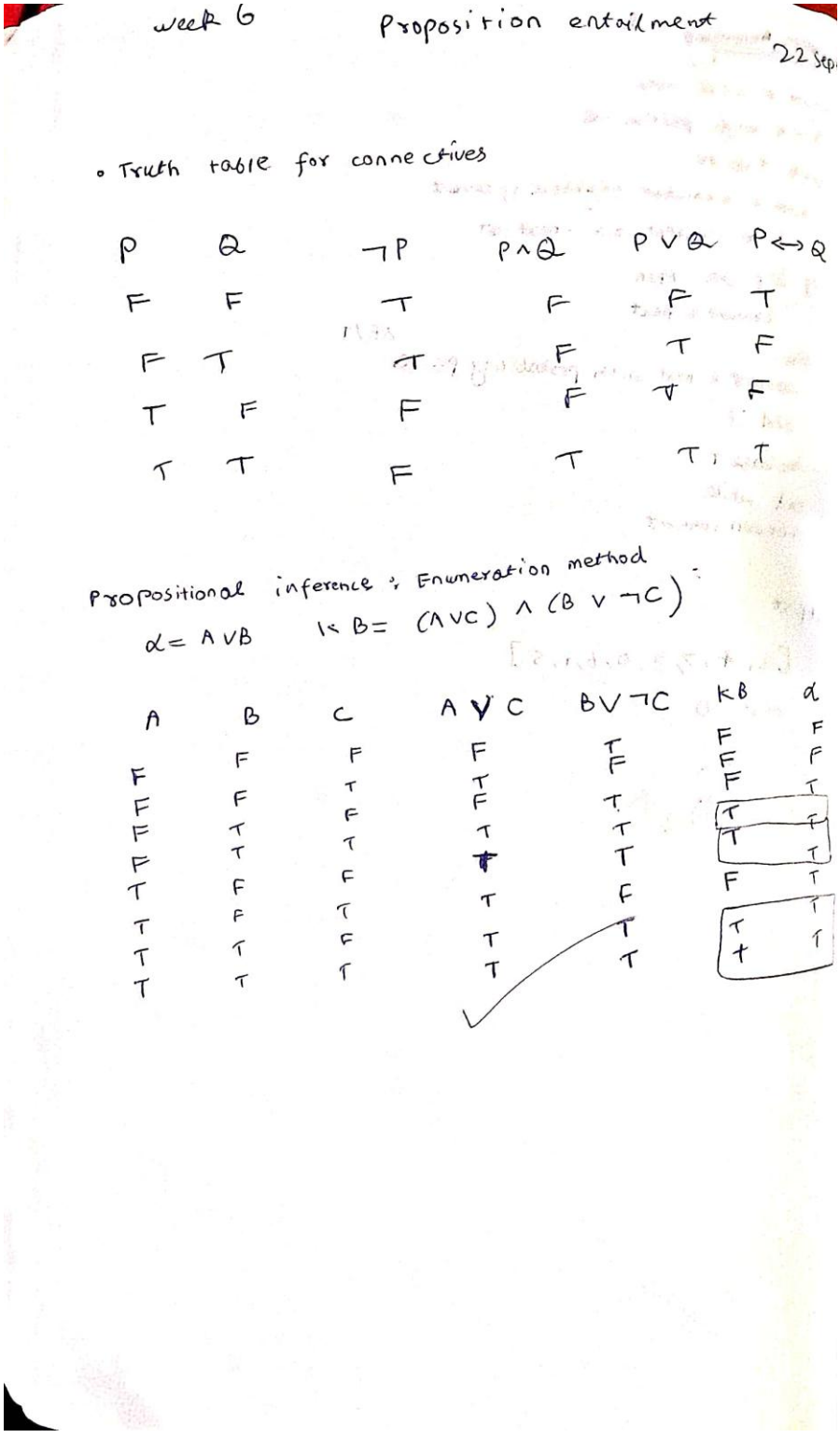
```python
import random
import math

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

def get_random_neighbor(state):
    n = len(state)
    new_state = list(state)
    col = random.randint(0, n - 1)
    row = random.randint(0, n - 1)
    new_state[col] = row
    return new_state

def simulated_annealing(n=8, max_iterations=10000, initial_temp=100.0, cooling

    current = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current)
    best = current
    best_cost = current_cost
    temperature = initial_temp

    for _ in range(max_iterations):
        if current_cost == 0:
            break  # found solution

        neighbor = get_random_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current, current_cost = neighbor, neighbor_cost

            if current_cost < best_cost:
                best, best_cost = current, current_cost

        temperature *= cooling_rate
        if temperature < 1e-6:
            break

    return best, best_cost

best_state, best_cost = simulated_annealing()

print("The best position found:", best_state)
print("cost =", best_cost)
print("soham arora 1bm23cs334")
```

```
The best position found: [2, 4, 7, 3, 0, 6, 1, 5]
cost = 0
```

Program 6

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

Algorithm:



Algorithm

function TT-ENTAILS?(KB, α) returns true or false

input: KB, the knowledge base, a sentence in propositional logic

α the query a sentence in propositional logic

symbols ← a list of proposition symbols in KB & α

return TT-CHECK ALL (KB, α, symbols, {})

function TT-CHECK ALL (KB, α, symbols, model)

returns true or false

if symbols == Empty then

if PL == TRUE (KB, model) 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-CHECKALL (KB, α, rest, model ∪ {P= true}) and TT-CHECK-ALL (KB, α, rest model ∪ {P= false}))

// In short

take input for KB, α expressions and give output (KB ⇒ α), create a truth table for entire scenario

2209

S, T as variables

a: $\vdash(S \vee T)$

b: $(S \wedge T)$

C: $T \vee \vdash T$

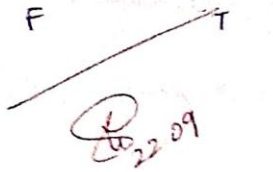
write TT

(1) a entails b

(2) a entails C

S	T	a $\vdash(S \vee T)$	b	C
T	T	F	T	T
T	F	F	F	T
F	T	F	F	T
F	F	T	F	T

a ⊨ b	a ⊨ C
F	F
F	F
F	F
F	T



Open in Colab

In [ ]:

In [ ]:
```python
import itertools

def parse_expression(expression):
    print("Soham Arora 1BM23CS334")
    expression = expression.replace('V', ' and ')
    expression = expression.replace('U', ' or ')
    expression = expression.replace('~', ' not ')
    import re
    expression = re.sub(r'([A-Z])', r'(\1)', expression)

    return expression

def evaluate_expression(expression, truth_assignment):
    for symbol, value in truth_assignment.items():
        expression = expression.replace(f'({symbol})', str(value))
    try:
        return eval(expression)
    except Exception as e:
        print(f"Error evaluating expression: {expression} with assignment {tru
        return False


def propositional_inference_enumeration_single(literals, knowledge_base_expr,
    symbols = sorted(literals)
    truth_assignments = list(itertools.product([True, False], repeat=len(symbc
    header = symbols + [knowledge_base_expr, alpha_expr, "KB True", "KB => Alp
    print("\t".join(header))
    print("-" * (8 * len(header)))

    entails = True
    for assignment_values in truth_assignments:
        truth_assignment = dict(zip(symbols, assignment_values))
        parsed_kb_expr = parse_expression(knowledge_base_expr)
        kb_evaluation = evaluate_expression(parsed_kb_expr, truth_assignment)

        parsed_alpha_expr = parse_expression(alpha_expr)
        alpha_evaluation = evaluate_expression(parsed_alpha_expr, truth_assigr


        kb_true = kb_evaluation
        kb_implies_alpha = not kb_true or alpha_evaluation

        row_values = list(assignment_values) + [kb_evaluation, alpha_evaluatic


        if kb_true and alpha_evaluation:
            print("\t".join(f"\033[1m{v}\033[0m" for v in row_values))
        else:
            print("\t".join(str(v) for v in row_values))
```

```
        if kb_true and not alpha_evaluation:
            entails = False

    return entails


num_literals = int(input("Enter the number of literals: "))
literals = [input(f"Enter literal {i+1}: ") for i in range(num_literals)]
knowledge_base_expr = input("Enter the knowledge base expression (use V for AN
alpha_expr = input("Enter the alpha expression (use V for AND, U for OR, ~ for

print("\n--- Truth Table ---")
result = propositional_inference_enumeration_single(literals, knowledge_base_e

if result:
    print(f"\nResult: Knowledge base entails alpha.")
else:
    print(f"\nResult: Knowledge base does not entail alpha.")
```

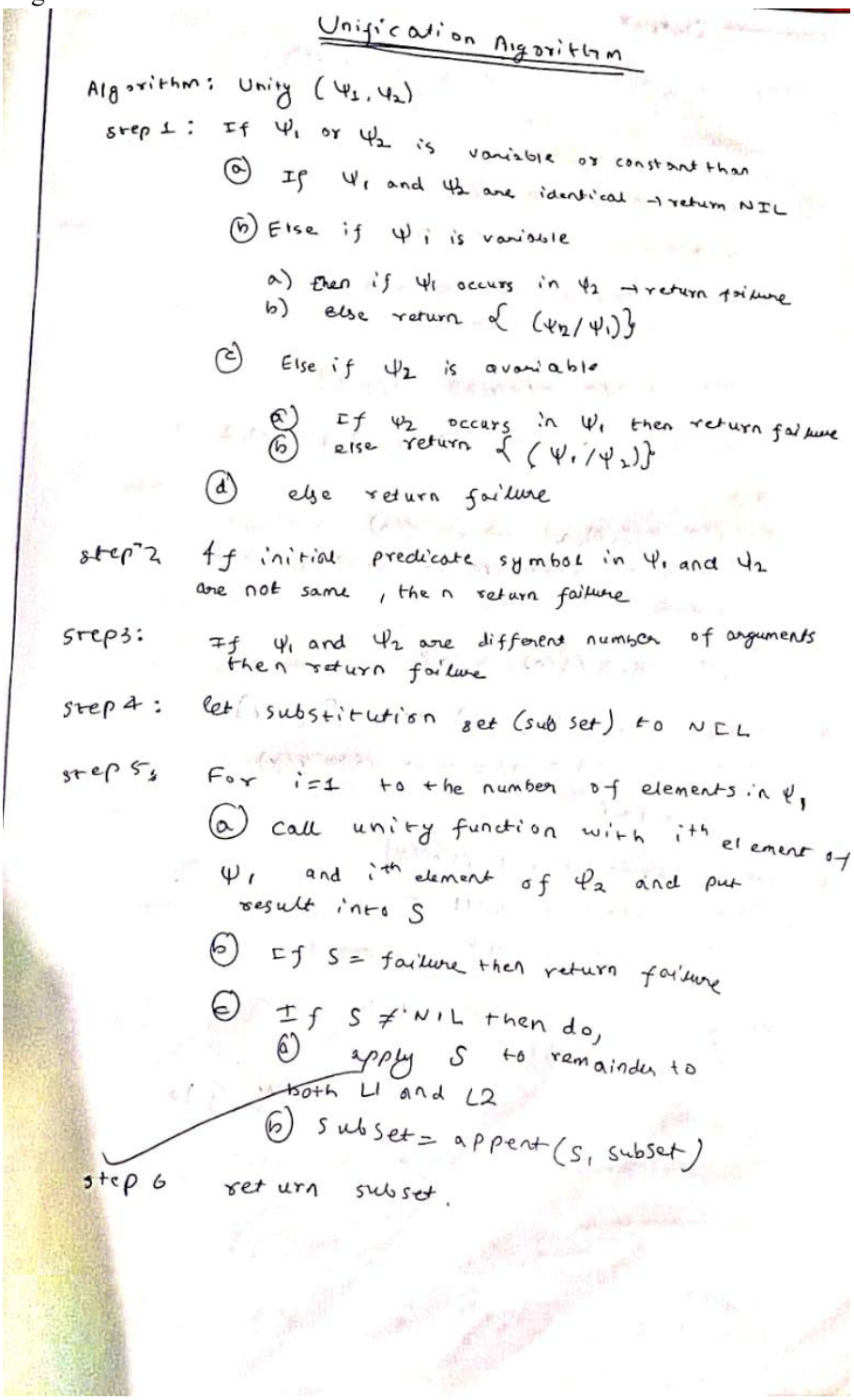
In [1]:
```
def alpha_beta(node, depth, alpha, beta, isMax, values, maxDepth):

    if depth == maxDepth:

        first_leaf_index = (2 ** maxDepth) - 1
        leaf_idx = node - first_leaf_index
        return values[leaf_idx]

    if isMax:
        best = float('-inf')
        for i in range(2):
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, False, values
            best = max(best, val)
            alpha = max(alpha, best)
            print(f"MAX: Depth={depth}, Node={node}, Alpha={alpha}, Beta={beta

            if beta <= alpha:
                print(f"🚫 PRUNED at MAX node {node} (α ≥ β)")
                break
        return best
    else:
        best = float('inf')
        for i in range(2):
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, True, values,
            best = min(best, val)
            beta = min(beta, best)
            print(f"MIN: Depth={depth}, Node={node}, Alpha={alpha}, Beta={beta

            if beta <= alpha:
                print(f"🚫 PRUNED at MIN node {node} (α ≥ β)")
                break
        return best


print("🧠 ALPHA–BETA PRUNING — Interactive Demo")
print("=========================================\n")


maxDepth = int(input("Enter maximum depth of the game tree: "))


num_leaves = 2 ** maxDepth
print(f"For depth {maxDepth}, the tree will have {num_leaves} leaf nodes.\n")


values = []
print("Enter the leaf node values from LEFT to RIGHT:")
for i in range(num_leaves):
    val = int(input(f"Value of leaf {i + 1}: "))
    values.append(val)
```

```
Enter the number of literals: 3
Enter literal 1: A
Enter literal 2: B
Enter literal 3: C
Enter the knowledge base expression (use V for AND, U for OR, ~ for NOT): A V B
U C
Enter the alpha expression (use V for AND, U for OR, ~ for NOT): A V B

--- Truth Table ---
A        B        C        A V B U C        A V B        KB True        KB => A
lpha
-----------------------------------------------------------
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
True        True        True        True        True        True        True
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
True        True        False        True        True        True        True
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
True        False        True        True        False        True        False
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
True        False        False        False        False        False        Tr
ue
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
False        True        True        True        False        True        False
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
False        True        False        False        False        False        Tr
ue
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
False        False        True        True        False        True        Fals
e
Soham Arora 1BM23CS334
Soham Arora 1BM23CS334
False        False        False        False        False        False        T
rue

Result: Knowledge base does not entail alpha.
```

Program 7

Implement unification in first order logic

Algorithm:



Unification Algorithm

Algorithm: Unify (Ψ_1, Ψ_2)

step 1: If Ψ_1 or Ψ_2 is variable or constant then

(a) If Ψ_1 and Ψ_2 are identical → return NIL

(b) Else if Ψ_1 is variable

a) then if Ψ_1 occurs in Ψ_2 → return failure
b) else return $\{(\Psi_2/\Psi_1)\}$

(c) Else if Ψ_2 is variable

(a) If Ψ_2 occurs in Ψ_1 then return failure
(b) else return $\{(\Psi_1/\Psi_2)\}$

(d) else return failure

step 2 If initial predicate symbol in Ψ_1 and Ψ_2 are not same, then return failure

step 3: If Ψ_1 and Ψ_2 are different number of arguments then return failure

step 4: Let substitution set (sub set) to NIL

step 5: For i=1 to the number of elements in Ψ_1

(a) Call unify function with i^{th} element of Ψ_1 and i^{th} element of Ψ_2 and put result into S

(b) If S = failure then return failure

(c) If S ≠ NIL then do,

(a) apply S to remainder to both L1 and L2

(b) subset = append (S, subset)

step 6 return subset.

~~Algorithm~~ Output

- Unify $\{f(b, x, f(g(z)), h(z, f(y), f(y))\}$

compare

$b = z \rightarrow z = b$

$x = f(y) \rightarrow x = f(y)$

$f(g(z)) = f(y) \rightarrow y = g(z)$

Substitute $y = g(z)$ into $x = f(y) =$
$x = f(g(z))$ No contradiction

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

2. Unify $Q(a, g(x,a), f(y))$ and $Q(g(f(b), a), x)$

$a = a$

$g(x,a) = g(f(b), a) \rightarrow x = f(b)$

$f(y) = x \rightarrow$ substitute $x = f(b) \rightarrow f(y) = f(b)$

$\Rightarrow y = b$

$mGU = \{ x/f(b), y/b \}$

3. Unify $P(f(a), g(y))$ and $P(x, x)$

$f(a) = x$ and $g(y) = x \Rightarrow f(a) = g(y)$

$mGU =$ fail

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

$11 = y \rightarrow y = 11$

$mGU = \{ y/11 \}$

5. Unify knows(John, x) and knows(y, mother(y))

John $\rightarrow y$

$x =$ mother$(y) \rightarrow x =$ mother(John)

~~x = mother~~

$mGU = \{ y/John, x/mother(John) \}$

27-10

6. Output

mGU $\{ 'John' : 'y', 'x' : 'Bill' \}$

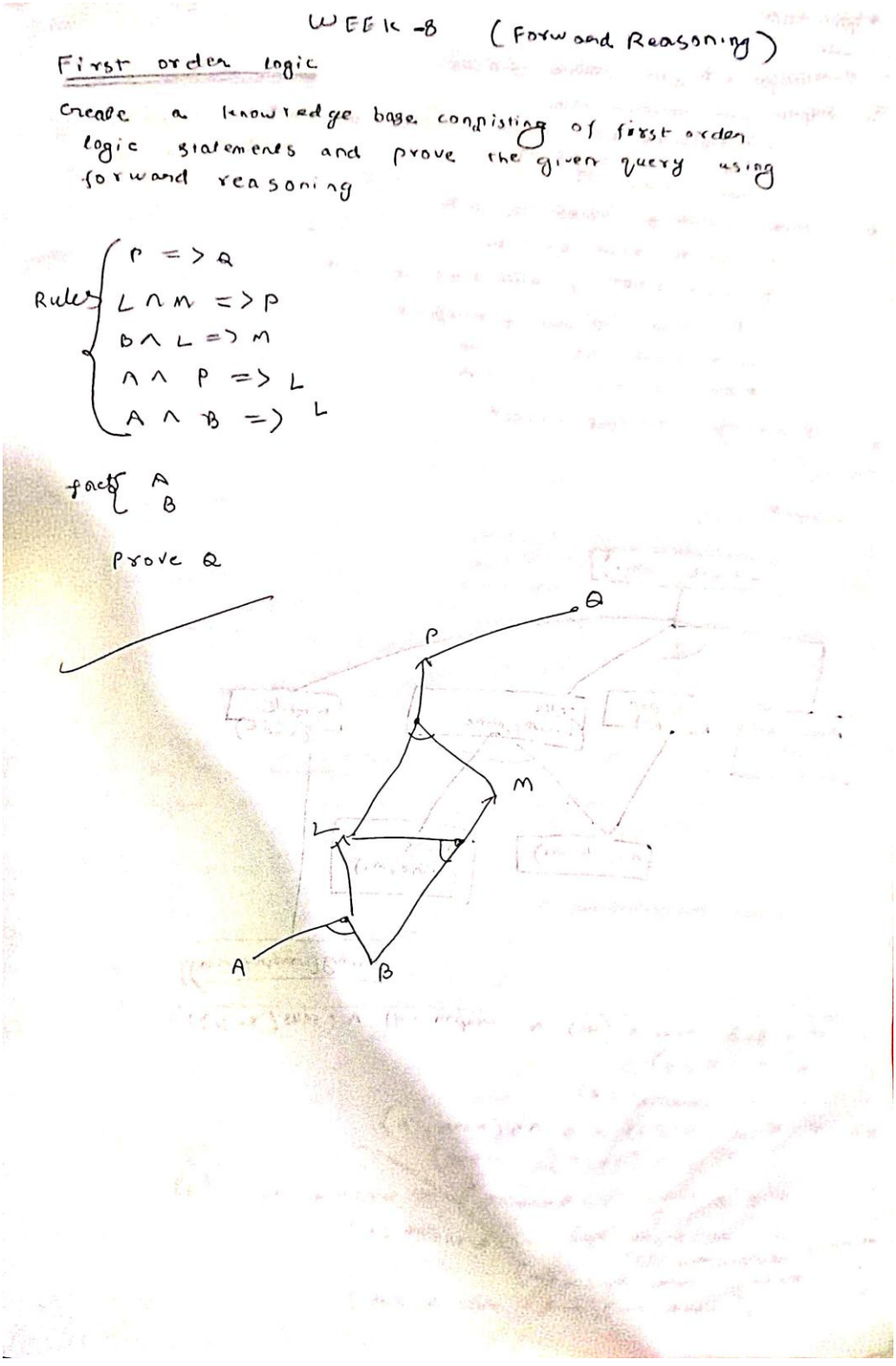
In [4]:

```python
def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        return any(occurs_check(var, sub) for sub in expr)
    return False

def substitute(expr, subst):
    if isinstance(expr, str):
        return subst.get(expr, expr)
    return tuple(substitute(sub, subst) for sub in expr)

def unify(Y1, Y2, subst=None):
    if subst is None:
        subst = {}

    Y1 = substitute(Y1, subst)
    Y2 = substitute(Y2, subst)

    if Y1 == Y2:
        return subst

    if isinstance(Y1, str):
        if occurs_check(Y1, Y2):
            return "FAILURE"
        subst[Y1] = Y2
        return subst

    if isinstance(Y2, str):
        if occurs_check(Y2, Y1):
            return "FAILURE"
        subst[Y2] = Y1
        return subst

    if Y1[0] != Y2[0] or len(Y1) != len(Y2):
        return "FAILURE"

    for a, b in zip(Y1[1:], Y2[1:]):
        subst = unify(a, b, subst)
        if subst == "FAILURE":
            return "FAILURE"

    return subst


expr1 = ('p', 'b', 'X', ('f', ('g', 'Z')))
expr2 = ('p', 'z', ('f', 'Y'), ('f', 'Y'))

result = unify(expr1, expr2)
print("MGU:", result)
print("1BM23CS334")
```

```
MGU: {'b': 'z', 'X': ('f', 'Y'), 'Y': ('g', 'Z')}
1BM23CS334
```

Program 8

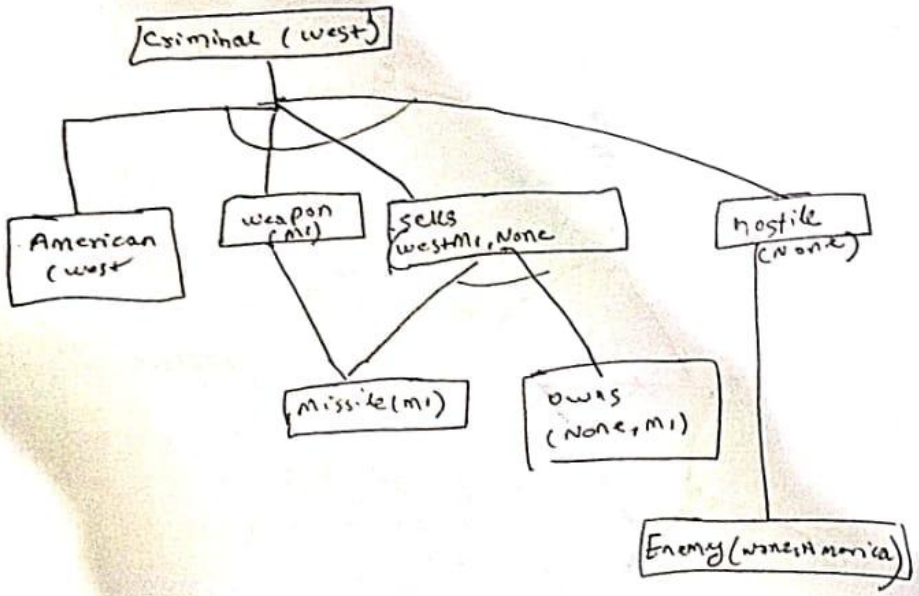
Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.



Algorithm

1. Start
2. Initialize KB with known facts
3. Define the inference rule
   - if (American (x) AND Hostile (y) AND SELLS_weapon (x, y) Then Crime (x)
4. check matching facts in KB
   - x such that American (x) is true
   - y such that Hostile (y) is true
5. If all conditions are satisfied
   - Infer Crime (x) is True
   - Add crime (x) to KB
6. Display inferred result
7. End

Output



1. $\forall x, y, z$ America $(x) \wedge$ (weapon $(y) \wedge$ (sells (x, y, z)) $\wedge$ Hostile (z) $\Rightarrow$ criminal (x)
2. $\forall x$ Missile $(x) \wedge$ owns (None, x) $\Rightarrow$ sells (west, x, None)
3. $\forall x$ Enemy $(x$, America) $\Rightarrow$ Hostile (x)
4. $\forall x$ Missile $(x) \Rightarrow$ weapon (x)
5. America (west)
6. Enemy (None, America)
7. owns (None, M1) and Missile (M1)

In [6]:

```
from collections import deque

class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
        self.inferred = set()

    def add_fact(self, fact):
        if fact not in self.facts:
            print(f"Adding fact: {fact}")
            self.facts.add(fact)
            return True
        return False

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def forward_chain(self):
        agenda = deque(self.facts)

        while agenda:
            fact = agenda.popleft()
            if fact in self.inferred:
                continue
            self.inferred.add(fact)

            for (premises, conclusion) in self.rules:
                if all(p in self.inferred for p in premises):
                    if conclusion not in self.facts:
                        print(f"Inferred new fact: {conclusion} from {premises
                        self.facts.add(conclusion)
                        agenda.append(conclusion)

                        if conclusion == 'Criminal(West)':
                            print("\n✅ Goal Reached: West is Criminal")
                            return True
        return False
kb = KnowledgeBase()


kb.add_fact('American(West)')
kb.add_fact('Enemy(Nono, America)')
kb.add_fact('Missile(M1)')
kb.add_fact('Owns(Nono, M1)')

kb.add_rule(premises=['Missile(M1)'], conclusion='Weapon(M1)')

kb.add_rule(premises=['Missile(M1)', 'Owns(Nono, M1)'], conclusion='Sells(West

kb.add_rule(premises=['Enemy(Nono, America)'], conclusion='Hostile(Nono)')

kb.add_rule(premises=['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)',
```

```
kb.forward_chain()
```

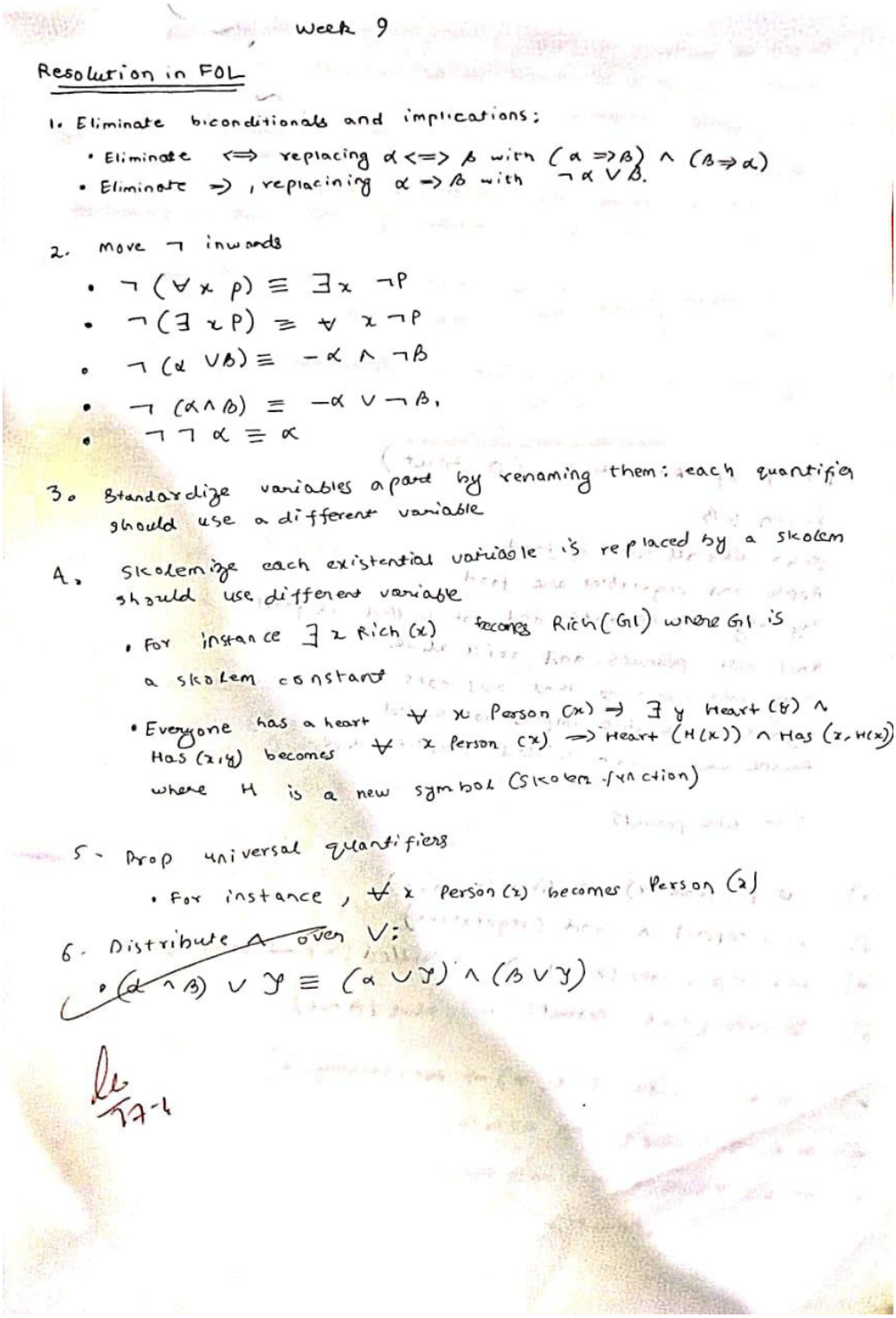
```
Adding fact: American(West)
Adding fact: Enemy(Nono, America)
Adding fact: Missile(M1)
Adding fact: Owns(Nono, M1)
Inferred new fact: Hostile(Nono) from ['Enemy(Nono, America)'] => Hostile(Nono)
Inferred new fact: Weapon(M1) from ['Missile(M1)'] => Weapon(M1)
Inferred new fact: Sells(West, M1, Nono) from ['Missile(M1)', 'Owns(Nono, M1)']
=> Sells(West, M1, Nono)
Inferred new fact: Criminal(West) from ['American(West)', 'Weapon(M1)', 'Sell
s(West, M1, Nono)', 'Hostile(Nono)'] => Criminal(West)

✅ Goal Reached: West is Criminal
```

Out[6]: True

In [ ]:

Program 9

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



Week 9

Resolution in FOL

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

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

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

4. Skolemize each existential variable is replaced by a skolem should use different variable
   - For instance $\exists x$ Rich(x) becomes Rich(G1) where G1 is a skolem constant
   - Everyone has a heart $\forall x$ Person(x) $\Rightarrow \exists y$ Heart(y) $\wedge$ Has(x,y) becomes $\forall x$ Person(x) $\Rightarrow$ Heart(H(x)) $\wedge$ Has(x, H(x)) where H is a new symbol (Skolem function)

5. Drop universal quantifiers
   - For instance, $\forall x$ Person(x) becomes Person(x)

6. Distribute $\wedge$ over $\vee$:
   - $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

11-4

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to premise clauses
4. Repeat until contradiction or no progress is made:
   (a) Select 2 clauses (call them parent clauses)
   (b) Resolve them together, performing all required unifications
   (c) If resolvent is empty clause a contradiction has been found (i.e S follows from premises)
   (d) If not, add resolvent to premises

Proof of Resolution (Output)

Given KB

John likes all kind of food
Apple and vegetables are food
Anything anyone eats and not killed is food
Anil eats peanuts and still alive
Harry eats everything that anil eats
Anyone who is alive implies not killed
Anyone who is not killed implies alive

John like peanuts

(a) $\forall x : food(x) \rightarrow likes(John, x)$
(b) $food(Apple) \wedge food(vegetables)$
(c) $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
(d) $eats(Anil, Peanuts) \wedge alive(Anil)$
(e) $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
(f) $\forall x : \neg killed(x) \rightarrow alive(x)$
(g) $\forall x : alive(x) \rightarrow \neg killed(x)$
(h) $likes(John, Peanuts)$

Proof of Resolution

- Rename variables or standardise variable

(a) $\forall x \neg food(x) \vee likes(John, x)$

(b) $food(APPle) \wedge food(vegetables)$

(c) $\forall y \forall z \neg eats(y, z) \vee killed(y) \vee food(z)$

(d) $eats(Anil, peanuts) \wedge alive(anil)$

(e) $\forall w \neg eats(Anil, w) \vee eats(Harry, w)$

(f) $\forall g [killed(g)] \vee alive(g)$

(g) $\forall k \neg alive(k) \vee \neg killed(k)$

(h) $likes(John, peanuts)$

(a) $\neg food(x) \vee likes(John, x)$

(b) $food(APPle)$

(c) $food(vegetables)$

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

(e) $eats(Anil, peanuts)$

(f) $alive(Anil)$

(g) $\neg eats(Anil, w) \vee eats(Harry, w)$

(k) $killed(g) \vee alive(g)$

(i) $\neg alive(k) \vee \neg killed(k)$

(j) $likes(John, peanuts)$

Proof by Resolution

(a) $\neg food(x) \vee likes(John, x)$

(b) $food(Apple)$

(c) $food(vegetables)$

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

(e) $eats(Anil, Peanuts)$

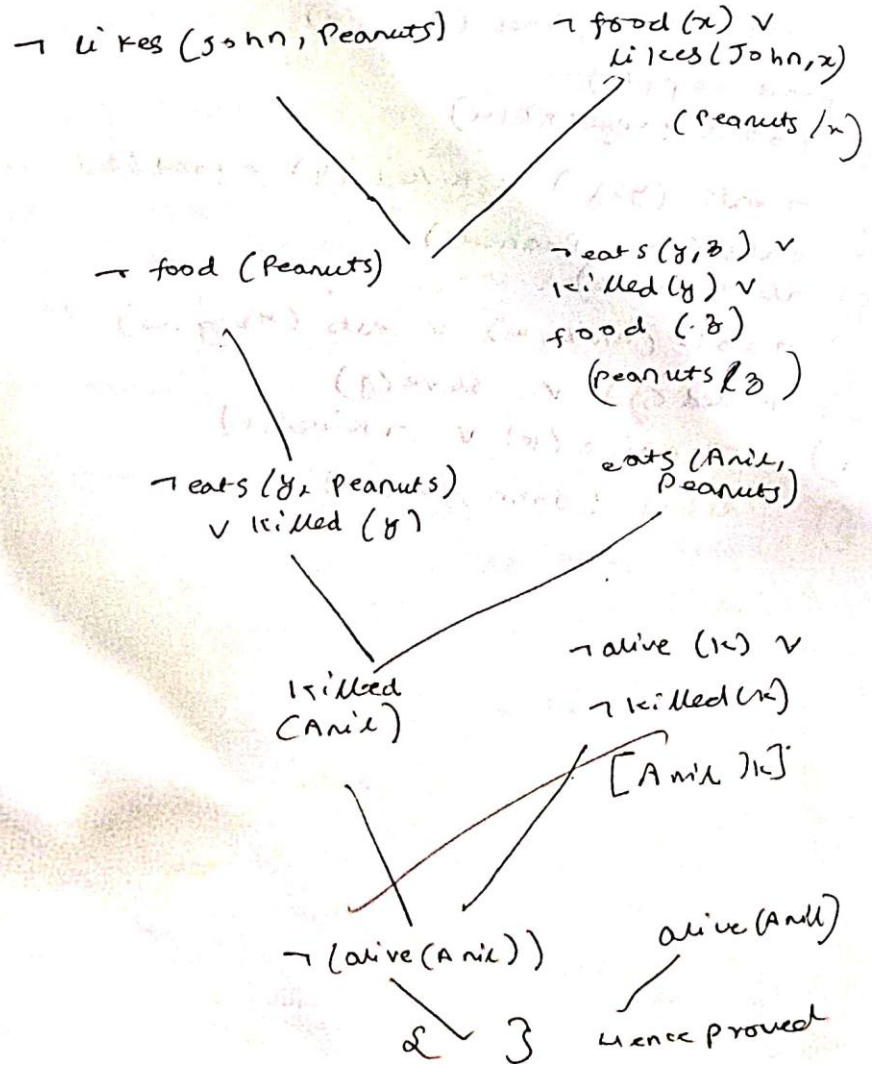
(f) $alive(Anil)$

(g) $\neg eats(Anil, w) \vee eats(Harry, w)$

(h) $killed(g) \vee alive(g)$

(i) $\neg alive(k) \vee \neg killed(k)$

(j) $likes(John, Peanuts)$



In [7]:

```python
from itertools import combinations

def get_clauses():
    n = int(input("Enter number of clauses in Knowledge Base: "))
    clauses = []
    for i in range(n):
        clause = input(f"Enter clause {i+1}: ")
        clause_set = set(clause.replace(" ", "").split("v"))
        clauses.append(clause_set)
    return clauses

def resolve(ci, cj):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di == ('~' + dj) or dj == ('~' + di):
                new_clause = (ci - {di}) | (cj - {dj})
                resolvents.append(new_clause)
    return resolvents

def resolution_algorithm(kb, query):
    kb.append(set(['~' + query]))
    derived = []
    clause_id = {frozenset(c): f"C{i+1}" for i, c in enumerate(kb)}

    step = 1
    while True:
        new = []
        for (ci, cj) in combinations(kb, 2):
            resolvents = resolve(ci, cj)
            for res in resolvents:
                if res not in kb and res not in new:
                    cid_i, cid_j = clause_id[frozenset(ci)], clause_id[frozens
                    clause_name = f"R{step}"
                    derived.append((clause_name, res, cid_i, cid_j))
                    clause_id[frozenset(res)] = clause_name
                    new.append(res)
                    print(f"[Step {step}] {clause_name} = Resolve({cid_i}, {ci
                    step += 1

                    # If empty clause found → proof complete
                    if res == set():
                        print("\n✅ Query is proved by resolution (empty claus
                        print("\n--- Proof Tree ---")
                        print_tree(derived, clause_name)
                        return True
        if not new:
            print("\n❌ Query cannot be proved by resolution.")
            return False
        kb.extend(new)

def print_tree(derived, goal):
    tree = {name: (parents, clause) for name, clause, *parents in [(r[0], r[1]
```

```
    def show(node, indent=0):
        if node not in tree:
            print(" " * indent + node)
            return
        parents, clause = tree[node]
        print(" " * indent + f"{node}: {set(clause) or '{}'}")
        for p in parents:
            show(p, indent + 4)

    show(goal)

print("=== FOL Resolution Demo with Proof Tree ===")
kb = get_clauses()
query = input("Enter query to prove: ")
resolution_algorithm(kb, query)
```

```
=== FOL Resolution Demo with Proof Tree ===
Enter number of clauses in Knowledge Base: 3
Enter clause 1: P
Enter clause 2: ~P v Q
Enter clause 3: ~Q
Enter query to prove: Q
[Step 1] R1 = Resolve(C1, C2) → {'Q'}
[Step 2] R2 = Resolve(C2, C4) → {'~P'}
[Step 3] R3 = Resolve(C1, R2) → {}

✅ Query is proved by resolution (empty clause found).

--- Proof Tree ---
R3: {}
    C1
    R2: {'~P'}
        C2
        C4
```

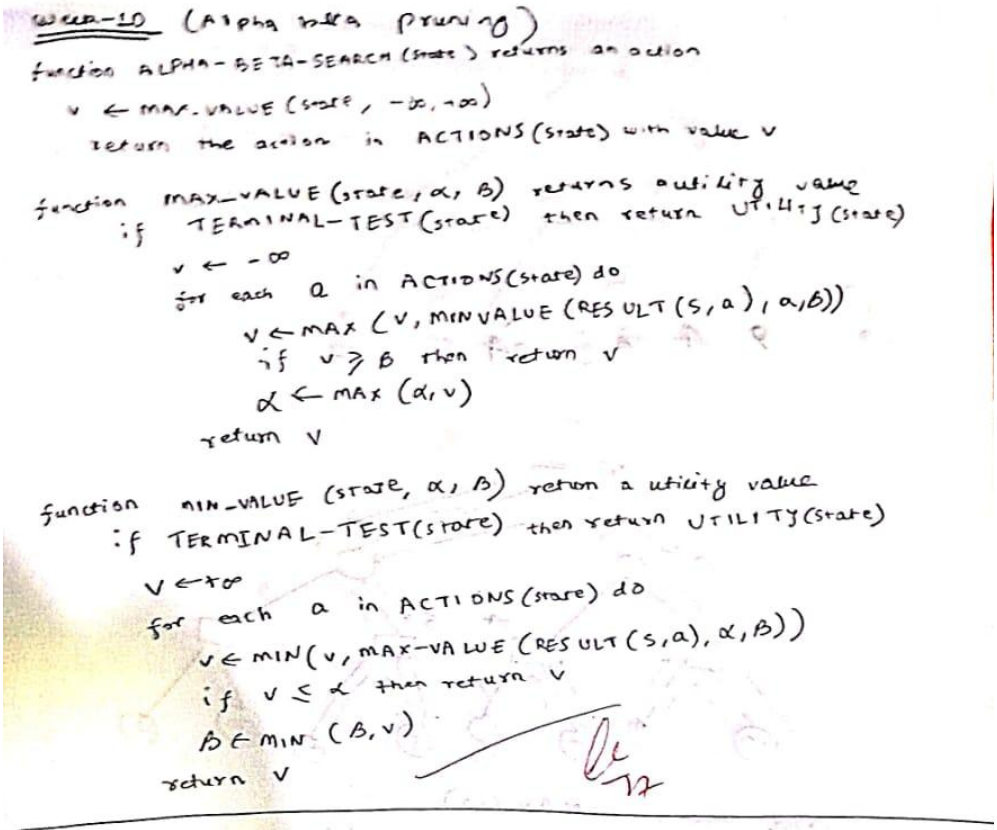
Out[7]: True

In [ ]:

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Week-10 (Alpha beta pruning)

function ALPHA-BETA-SEARCH (state) returns an action

v ← MAX-VALUE (state, −∞, +∞)

return the action in ACTIONS (state) with value v

function MAX-VALUE (state, α, β) returns utility value

if TERMINAL-TEST (state) then return UTILITY (state)

v ← −∞

for each a in ACTIONS (state) do

v ← MAX (v, MINVALUE (RESULT (s, a), α, β))

if v ≥ β then return v

α ← MAX (α, v)

return v

function MIN-VALUE (state, α, β) retun a utility value

if TERMINAL-TEST (state) then return UTILITY (state)

v ← +∞

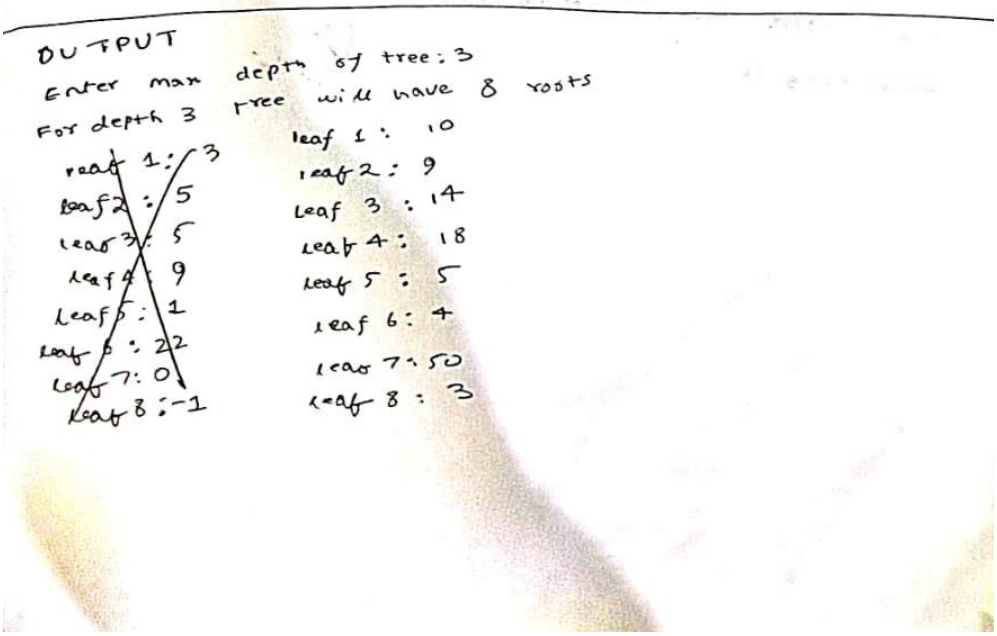
for each a in ACTIONS (state) do

v ← MIN (v, MAX-VALUE (RESULT (s, a), α, β))

if v ≤ α then return v

β ← MIN (β, v)

return v



OUTPUT

Enter max depth of tree: 3

For depth 3 tree will have 8 roots

leaf 1: 3

leaf 2: 5

leaf 3: 5

leaf 4: 9

leaf 5: 1

leaf 6: 2

leaf 7: 0

leaf 8: -1

leaf 1: 10

leaf 2: 9

leaf 3: 14

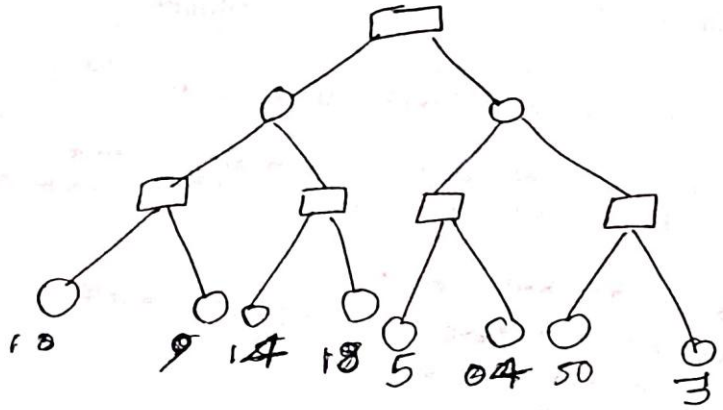
leaf 4: 18

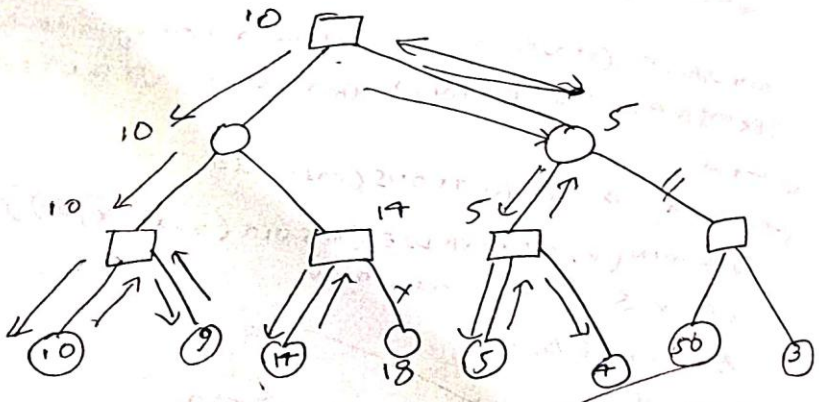
leaf 5: 5

leaf 6: 4

leaf 7: 50

leaf 8: 3





Best value for root MAX (10)

total moves 11

```python
def alpha_beta(node, depth, alpha, beta, isMax, values, maxDepth):

    if depth == maxDepth:

        first_leaf_index = (2 ** maxDepth) - 1
        leaf_idx = node - first_leaf_index
        return values[leaf_idx]

    if isMax:
        best = float('-inf')
        for i in range(2):
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, False, values, maxDepth)
            best = max(best, val)
            alpha = max(alpha, best)
            print(f"MAX: Depth={depth}, Node={node}, Alpha={alpha}, Beta={beta}")

            if beta <= alpha:
                print(f"🚫 PRUNED at MAX node {node} (α ≥ β)")
                break
        return best
    else:
        best = float('inf')
        for i in range(2):
            child_node = node * 2 + i + 1
            val = alpha_beta(child_node, depth + 1, alpha, beta, True, values, maxDepth)
            best = min(best, val)
            beta = min(beta, best)
            print(f"MIN: Depth={depth}, Node={node}, Alpha={alpha}, Beta={beta}")

            if beta <= alpha:
                print(f"🚫 PRUNED at MIN node {node} (α ≥ β)")
                break
        return best
```

```python
print("🧠 ALPHA-BETA PRUNING — Interactive Demo")
print("=========================================\n")


maxDepth = int(input("Enter maximum depth of the game tree: "))


num_leaves = 2 ** maxDepth
print(f"For depth {maxDepth}, the tree will have {num_leaves} leaf nodes.\n")


values = []
print("Enter the leaf node values from LEFT to RIGHT:")
for i in range(num_leaves):
    val = int(input(f"Value of leaf {i + 1}: "))
    values.append(val)

print("\n🔁 Running Alpha-Beta pruning...\n")
result = alpha_beta(0, 0, float('-inf'), float('inf'), True, values, maxDepth)


print("\n✅ Final Result:")
print("soham arora 1bm23cs334")
print(f"Value of the root node (best achievable for MAX): {result}")
```

```
🧠 ALPHA-BETA PRUNING — Interactive Demo
========================================

Enter maximum depth of the game tree: 3
For depth 3, the tree will have 8 leaf nodes.

Enter the leaf node values from LEFT to RIGHT:
Value of leaf 1: 1
Value of leaf 2: 2
Value of leaf 3: 3
Value of leaf 4: 4
Value of leaf 5: 5
Value of leaf 6: 6
Value of leaf 7: 8
Value of leaf 8: 9

🔁 Running Alpha-Beta pruning...

MAX: Depth=2, Node=3, Alpha=1, Beta=inf
MAX: Depth=2, Node=3, Alpha=2, Beta=inf
MIN: Depth=1, Node=1, Alpha=-inf, Beta=2
MAX: Depth=2, Node=4, Alpha=3, Beta=2
🚫 PRUNED at MAX node 4 (α ≥ β)
MIN: Depth=1, Node=1, Alpha=-inf, Beta=2
MAX: Depth=0, Node=0, Alpha=2, Beta=inf
MAX: Depth=2, Node=5, Alpha=5, Beta=inf
MAX: Depth=2, Node=5, Alpha=6, Beta=inf
MIN: Depth=1, Node=2, Alpha=2, Beta=6
MAX: Depth=2, Node=6, Alpha=8, Beta=6
🚫 PRUNED at MAX node 6 (α ≥ β)
MIN: Depth=1, Node=2, Alpha=2, Beta=6
MAX: Depth=0, Node=0, Alpha=6, Beta=inf

✅ Final Result:
soham arora 1bm23cs334
Value of the root node (best achievable for MAX): 6
```