VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shaikh Uzair Ahmed (1BM23CS307)

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 **Shaikh Uzair Ahmed (1BM23CS307),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

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

Index

Uzair Ahmed

IV - F

Index

Prg 1 : Implement Tic Tac Toe Problem

Prg 2 : Implement Vacuum cleaner.

Prg 3.a : 8 puzzle (BFS) (without heuristic approach)

Prg 3.b : 8 puzzle (BFS) (with heuristic)

Prg 3.c : " (DFS)

Prg 4.1 : IDDFS

Coupled.

Prg 4.2 Misplaced tile

Prg 4.3 Manhattan distance

Prg 5.1 Hill Climbing Algorithm to solve N-Queens

Prg 5.2 Simulated Annealing to solve 8-Queens

Prg 6. Propositional logic

Prg 7 Unification Algorith

Prg 8 FOL Algorithm

Prg 9 Resolution in FOL

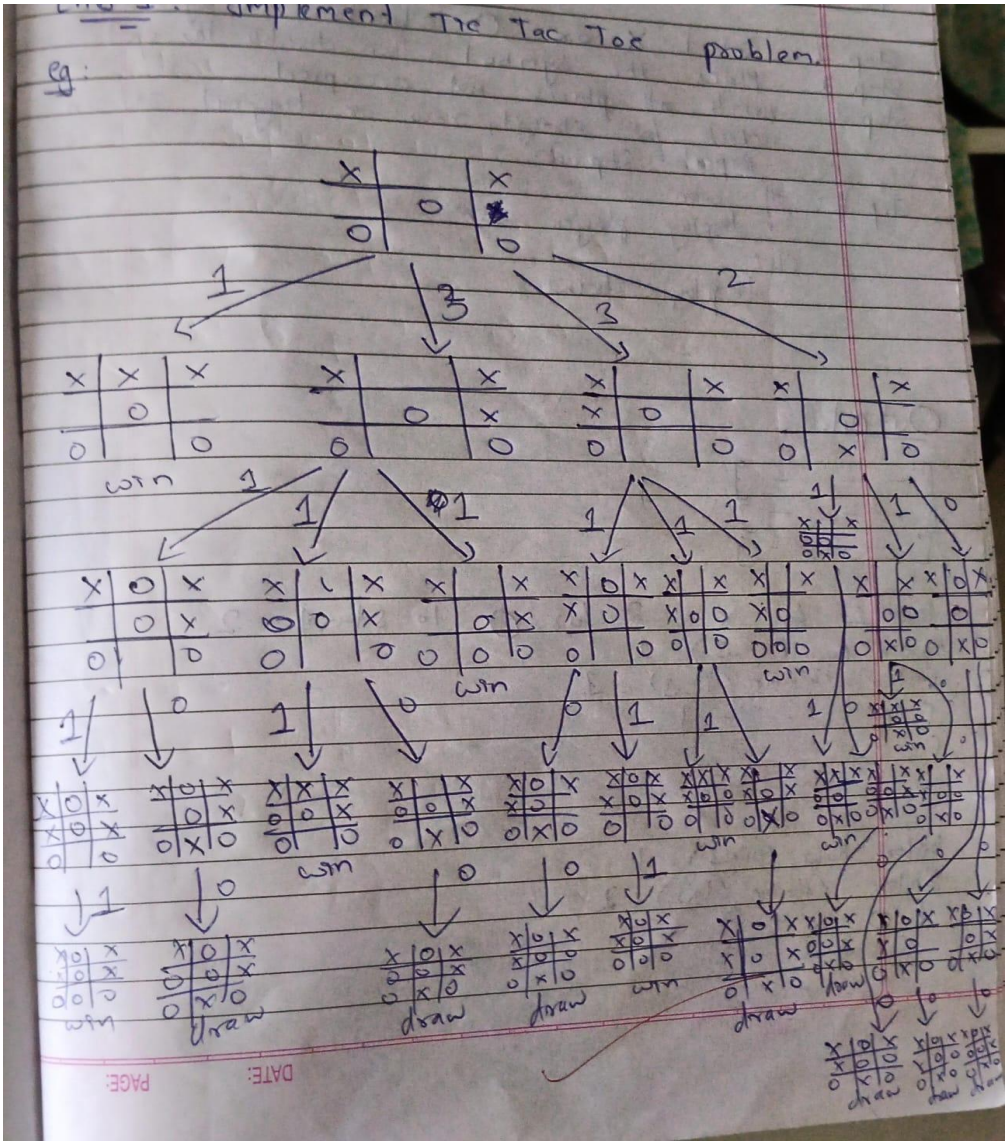
Prg 10 Alpha-beta pruning

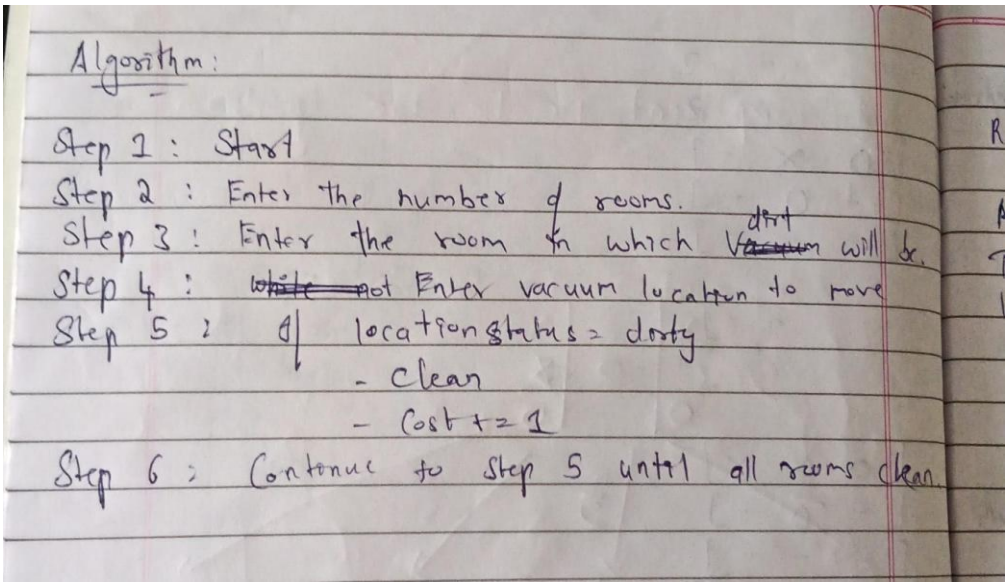
Github Link:
https://github.com/Shaikh-Uzair-Ahmed/AI-LAB-5th-Sem

Program 1

Implement Tic –Tac –Toe Game
Implement vacuum cleaner agent

Algorithm:
a) Tic Tac Toe

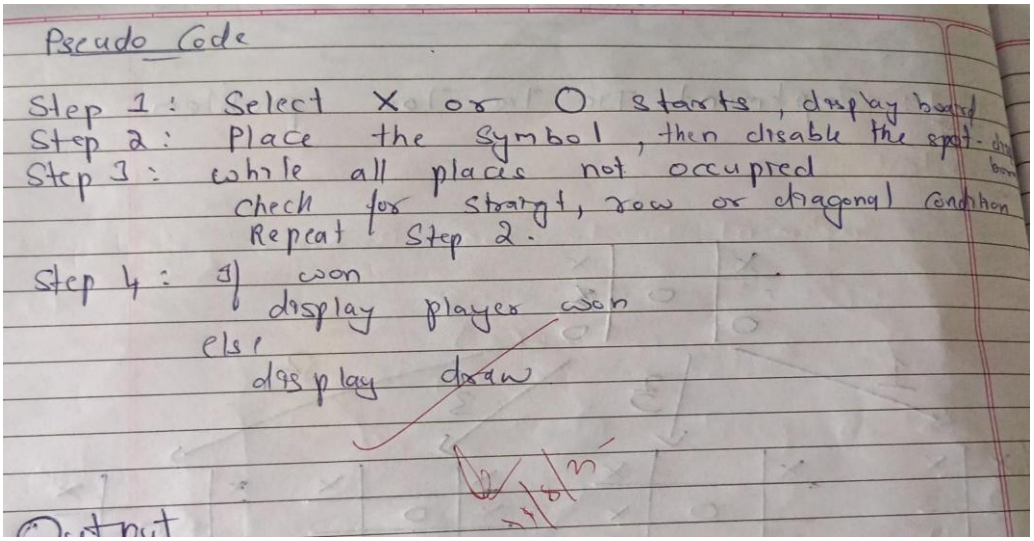




Algorithm:

Step 1 : Start
Step 2 : Enter the number of rooms.
Step 3 : Enter the room in which dirt will be.
Step 4 : ~~while not~~ Enter vacuum location to move
Step 5 : if location status = dirty
- clean
- Cost += 1
Step 6 : Continue to step 5 until all rooms clean.

b) Vacuum Cleaner



Pseudo Code

Step 1 : Select X or O starts, display board
Step 2 : Place the symbol, then disable the spot.
Step 3 : while all places not occupied
check for straight, row or diagonal condition
Repeat Step 2.
Step 4 : if won
display player won
else
display draw

Output

Code:
a)Tic Tac Toe

```
board = [["-","-","-"],["-","-","-"],["-","-","-"]]
Xrow = {"0":0,"1":0,"2":0}
Orow = {"0":0,"1":0,"2":0}
Xcol = {"0":0,"1":0,"2":0}
Ocol = {"0":0,"1":0,"2":0}
win = False

def checkWin():
    global win
    for key in Xrow:
        if Xrow[key] == 3:
            win = True
    for key in Xcol:
        if Xcol[key] == 3:
            win = True
    for key in Orow:
        if Orow[key] == 3:
            win = True
    for key in Ocol:
        if Ocol[key] == 3:
            win = True
    if board[0][0] == board[1][1] == board[2][2] != "-":
        win = True
    if board[0][2] == board[1][1] == board[2][0] != "-":
        win = True

def place(symbol, row, col):
    if board[row][col] != "-":
        print("Can't place at this spot, try again")
        return False

    if symbol == "1":
        board[row][col] = "X"
        Xrow[str(row)] += 1
        Xcol[str(col)] += 1
    elif symbol == "2":
        board[row][col] = "O"
        Orow[str(row)] += 1
        Ocol[str(col)] += 1
```

```
    return True

def displayBoard():
    for row in board:
        print(row)
    print("\n")
# Game loop
displayBoard()
switch = input("Enter 1 for X, 2 for O to start: ")

while any("-" in row for row in board) and not win:
    try:
        row = int(input("Enter Row (1-3): "))
        col = int(input("Enter Column (1-3): "))
    except ValueError:
        print("Invalid input, enter numbers only.")
        continue

    if row > 3 or col > 3 or row < 1 or col < 1:
        print("Invalid input, please try again.")
        continue

    decision = place(switch, row - 1, col - 1)
    if decision:
        displayBoard()
        checkWin()
        if not win:
            switch = "2" if switch == "1" else "1"

if win:
    winner = "X" if switch == "1" else "O"
    print(f"{winner} wins!")
else:
    print("It's a draw!")

print("Uzair 1BM23CS307")
```

Output:

```
['-', '-', '-']
['-', '-', '-']
```

['-', '-', '-']

Enter 1 for X, 2 for O to start: 1
Enter Row (1-3): 2
Enter Column (1-3): 2
['-', '-', '-']
['-', 'X', '-']
['-', '-', '-']

Enter Row (1-3): 1
Enter Column (1-3): 3
['-', '-', 'O']
['-', 'X', '-']
['-', '-', '-']

Enter Row (1-3): 1
Enter Column (1-3): 2
['-', 'X', 'O']
['-', 'X', '-']
['-', '-', '-']

Enter Row (1-3): 2
Enter Column (1-3): 3
['-', 'X', 'O']
['-', 'X', 'O']
['-', '-', '-']

Enter Row (1-3): 3
Enter Column (1-3): 2
['-', 'X', 'O']
['-', 'X', 'O']
['-', 'X', '-']

X wins!
Uzair 1BM23CS307

b) Vacuum Cleaner

```
rooms = int(input("Enter Number of rooms: "))
Rooms = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
cost = 0
Roomval = {}
for i in range(rooms):
    print(f"Enter Room {Rooms[i]} state (0 for clean, 1 for dirty): ")
    n = int(input())
    Roomval[Rooms[i]] = n

loc = input(f"Enter Location of vacuum ({Rooms[:rooms]}): ").upper()

while 1 in Roomval.values():
    if Roomval[loc] == 1:
        print(f"Room {loc} is dirty. Cleaning...")
        Roomval[loc] = 0
        cost += 1
    else:
        print(f"Room {loc} is already clean.")

    move = input("Enter L or R to move left or right (or Q to quit): ").upper()

    if move == "L":
        if loc != Rooms[0]:
            loc = Rooms[Rooms.index(loc) - 1]
        else:
            print("No room to move left.")
    elif move == "R":
        if loc != Rooms[rooms - 1]:
            loc = Rooms[Rooms.index(loc) + 1]
        else:
            print("No room to move right.")
    elif move == "Q":
        break
    else:
        print("Invalid input. Please enter L, R, or Q.")

print("\nAll Rooms Cleaned." if 1 not in Roomval.values() else "Exited before cleaning all rooms.")
print(f"Total cost: {cost}")
```

```
print("1BM23CS307 Uzair")
```

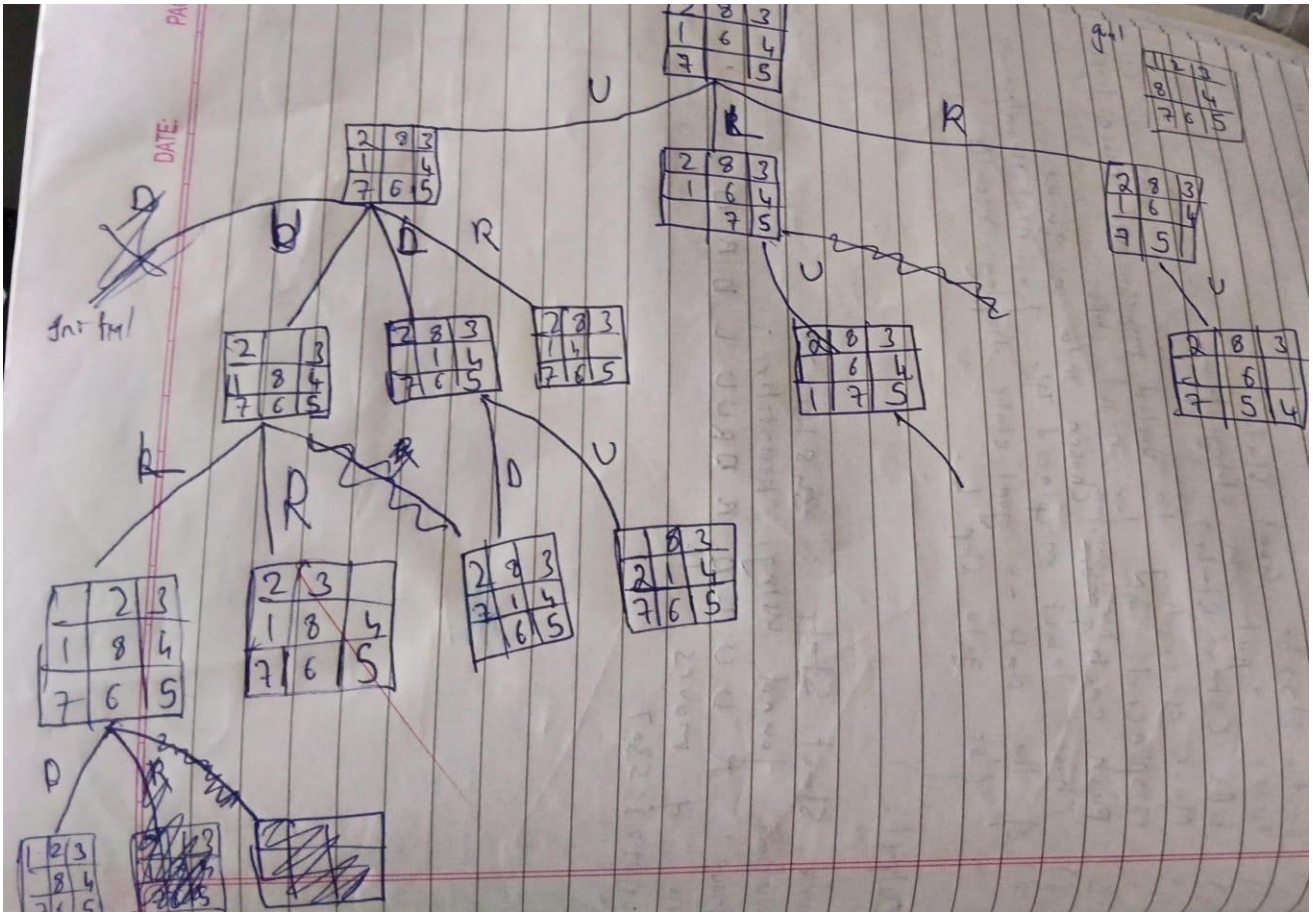
Output:

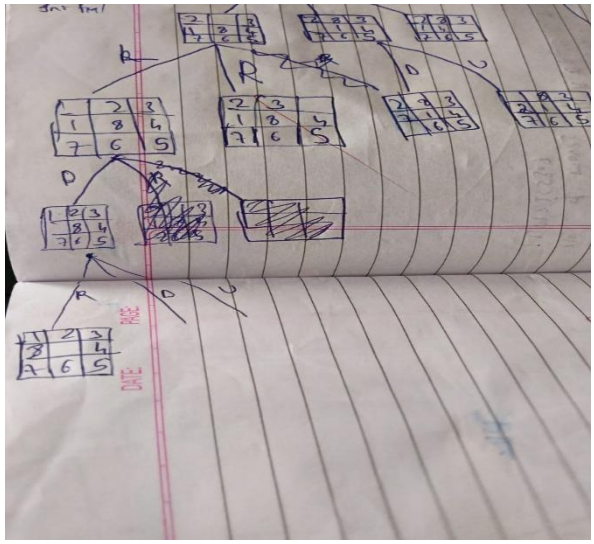
```
Enter Number of rooms: 3
Enter Room A state (0 for clean, 1 for dirty):
1
Enter Room B state (0 for clean, 1 for dirty):
0
Enter Room C state (0 for clean, 1 for dirty):
1
Enter Location of vacuum (ABC): B
Room B is already clean.
Enter L or R to move left or right (or Q to quit): L
Room A is dirty. Cleaning...
Enter L or R to move left or right (or Q to quit): R
Room B is already clean.
Enter L or R to move left or right (or Q to quit): R
Room C is dirty. Cleaning...
Enter L or R to move left or right (or Q to quit): R
No room to move right.

All Rooms Cleaned.
Total cost: 2
1BM23CS307 Uzair
```

Program 2

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

Algorithm:





LaB 2.3

Implement DFS in 8 puzzle program

Algorithm:

1) Start, Write goal state
2) Take input in string form
3) DFS (start state)
4) If not visited
   - add in state, path queue
5) If goal-state == state
   return no of visits, path
   else go to step 4
6) Display each stage

Output

Lab 3.1

1) Implement Iterative deepening search for 8 puzzle problem

Algorithm:

function: IDDFS (problem) returns a solution
input: problem, a problem

for depth ← 0 to ∞ do
   result ← depth-limited-search (problem, depth)
   if result ≠ cutoff then return result
end

Code:
a) DFS

```
goal_state = '123456780'

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

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

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

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

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

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

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

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

```
        if current_state in visited:
            continue

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

        if current_state == goal_state:
            return path

        visited.add(current_state)

        if len(path) >= max_depth:
            continue

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

    return None

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

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("1BM23CS307 Uzair\n")

        current_state = start
        for i, move in enumerate(result, 1):
            current_state = move_tile(current_state, move)
            print(f"Move {i}: {move}")
```

```
            print_state(current_state)
    else:
        print("No solution exists for the given start state or max depth reached.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

```
Enter start state (e.g., 724506831): 123456078
Start state:
1 2 3
4 5 6
  7 8

Visited state:
1 2 3
4 5 6
  7 8

Visited state:
1 2 3
4 5 6
7   8

Visited state:
1 2 3
4 5 6
7 8

Solution found!
Moves: R R
Number of moves: 2
1BM23CS307 Uzair

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

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

b) Iterative Deepening Search

```
goal_state = '123456780'

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

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

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

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

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

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

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

    if depth == 0:
```

```
        return None

    visited.add(state)

    for direction in moves:
        new_state = move_tile(state, direction)
        if new_state and new_state not in visited:
            result = dls(new_state, depth - 1, path + [direction], visited, visited_count)
            if result is not None:
                return result

    visited.remove(state)
    return None

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

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

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

    result, visited_states = iddfs(start,15)

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

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS307 Uzair\n")

        current_state = start
```

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

Output:

```
Enter start state (e.g., 724506831): 123405678
Start state:
1 2 3
4   5
6 7 8

Total states visited: 24298
Solution found!
Moves: R D L L U R D R U L L D R R
Number of moves: 14
1BM23CS307 Uzair

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

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

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

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

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

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

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

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

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

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

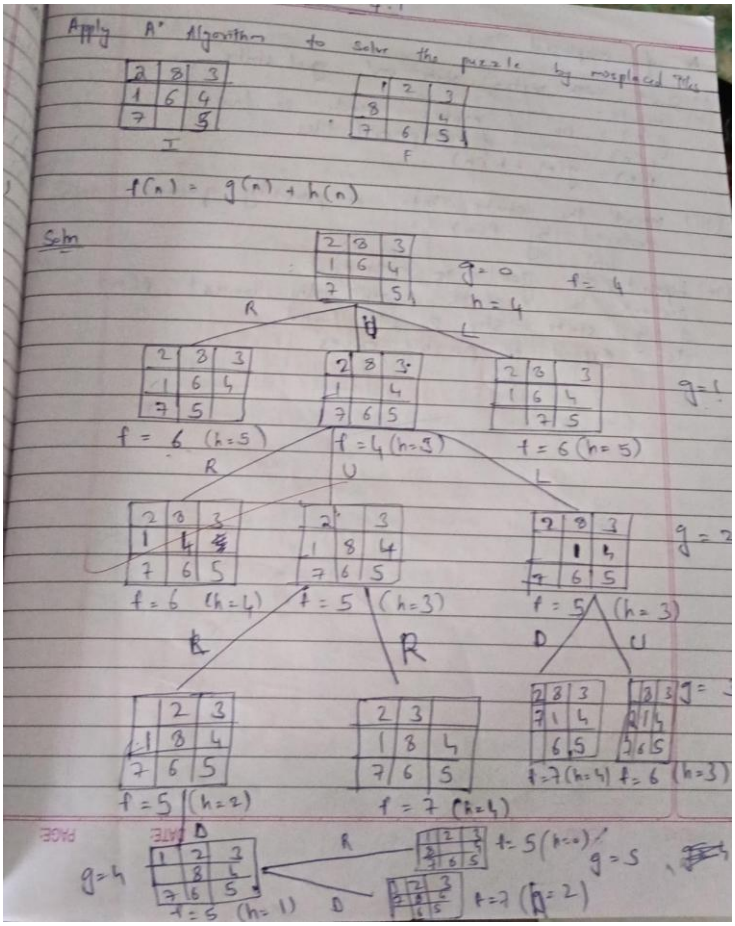
Move 11: L
1 2 3
  5 6
4 7 8

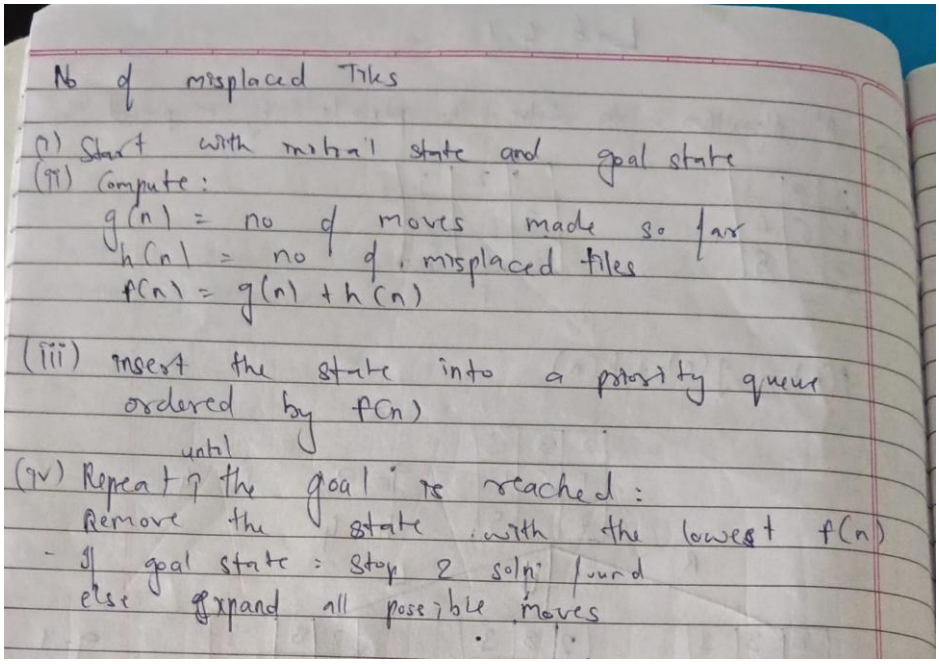
Move 12: D
1 2 3
4 5 6
  7 8

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

Move 14: R
1 2 3
4 5 6
7 8

Program 3
Implement A* search algorithm



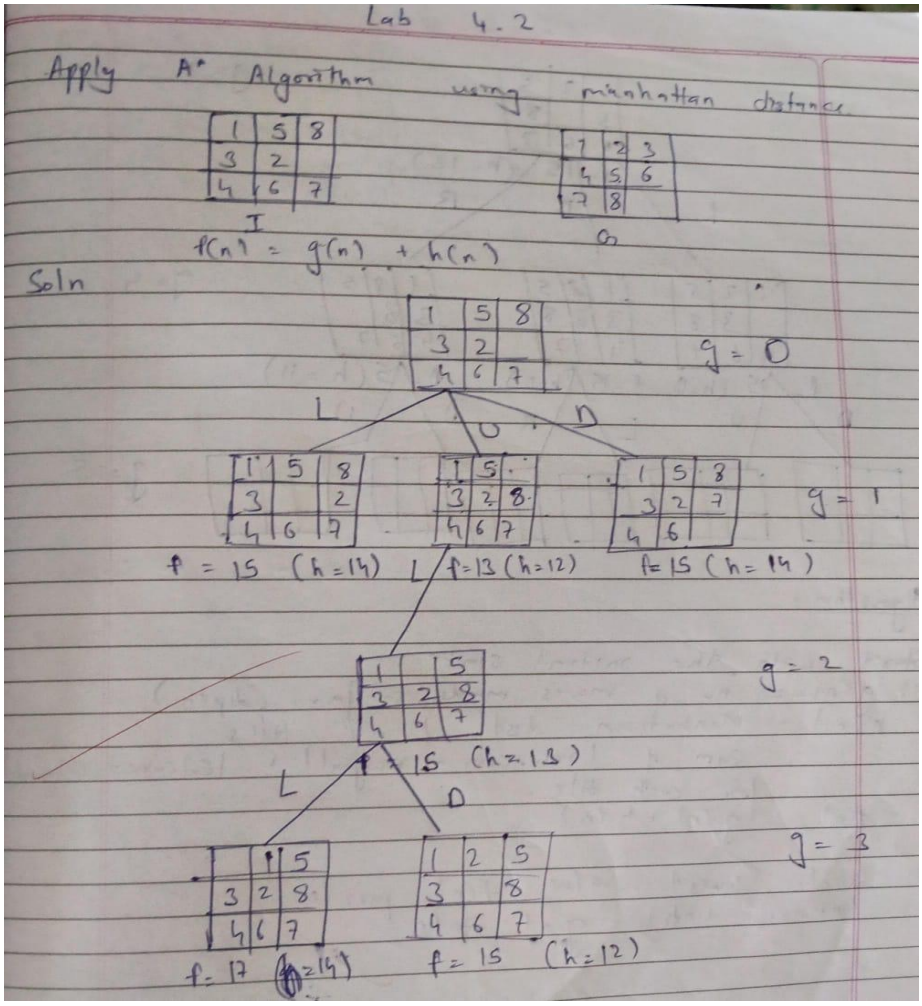


No of misplaced Tiles

(i) Start with initial state and goal state
(ii) Compute:
g(n) = no of moves made so far
h(n) = no of misplaced tiles
f(n) = g(n) + h(n)

(iii) Insert the state into a priority queue ordered by f(n)

(iv) Repeat until the goal is reached:
Remove the state with the lowest f(n)
If goal state : Stop & soln found
else expand all possible moves



Lab 4.2

Apply A* Algorithm using manhattan distance

1	5	8
3	2	
4	6	7

I

1	2	3
4	5	6
7	8	

G

f(n) = g(n) + h(n)

Soln

g = 0

L U D

f = 15 (h=14) | f=13 (h=12) | f=15 (h=14)

g = 1

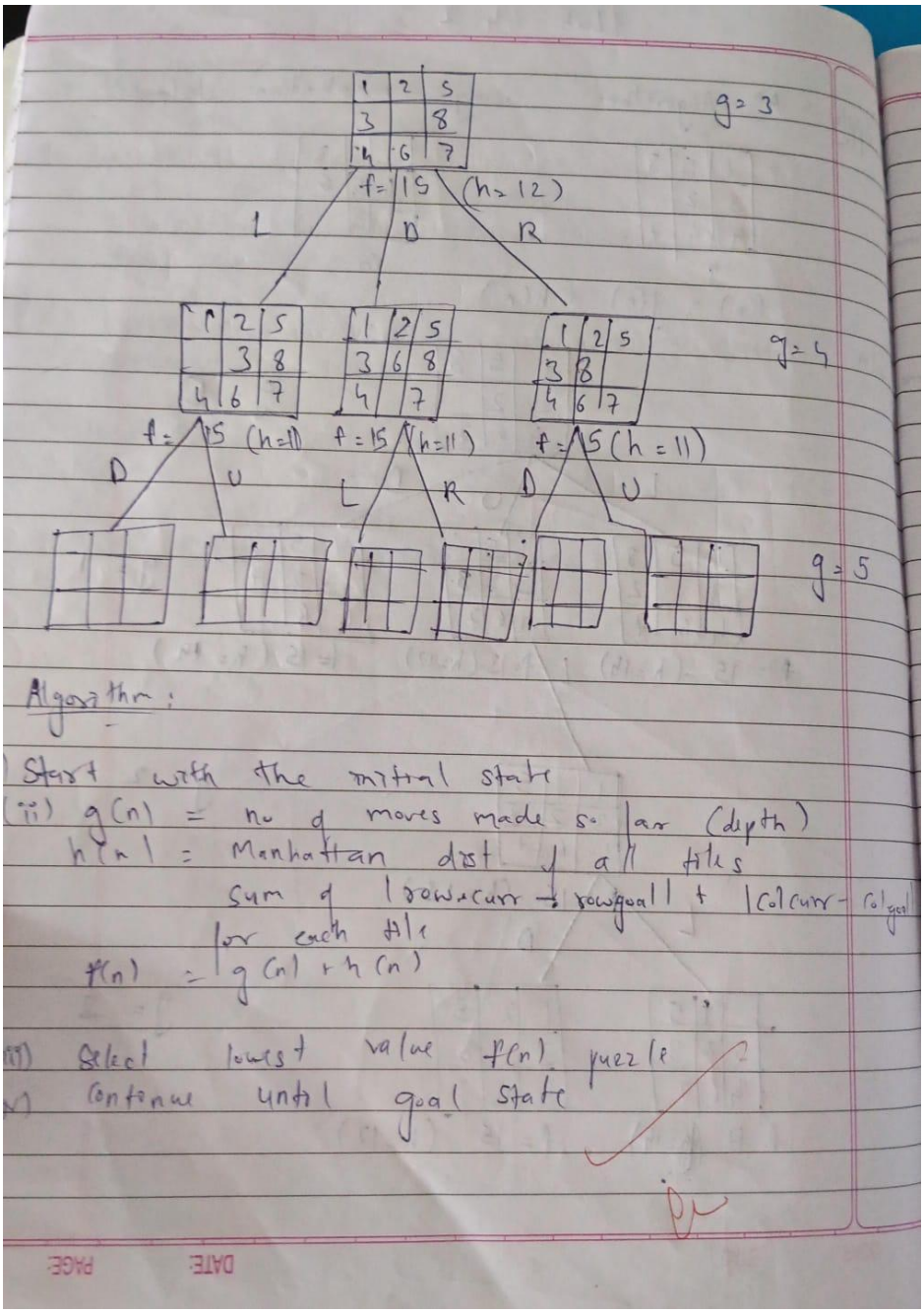
f = 15 (h=13)

g = 2

L D

f=17 (h=14) | f=15 (h=12)

g = 3



Code:
a) Misplaced Tiles

```
import heapq

goal_state = '123804765'

moves = {
    'U': -3,
```

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

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

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

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

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

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

def misplaced_tiles(state):
    """Heuristic: count of tiles not in their goal position (excluding zero)."""
    return sum(1 for i, val in enumerate(state) if val != '0' and val != goal_state[i])

def a_star(start_state):
    visited_count = 0
    open_set = []
    heapq.heappush(open_set, (misplaced_tiles(start_state), 0, start_state, []))
    visited = set()

    while open_set:
        f, g, current_state, path = heapq.heappop(open_set)
        visited_count += 1

        if current_state == goal_state:
            return path, visited_count
```

```
        if current_state in visited:
            continue
        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                new_g = g + 1
                new_f = new_g + misplaced_tiles(new_state)
                heapq.heappush(open_set, (new_f, new_g, new_state, path + [direction]))

    return None, visited_count

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

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

    result, visited_states = a_star(start)

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

    if result is not None:
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS307 Uzair\n")

        current_state = start
        g = 0  # initialize cost so far
        for i, move in enumerate(result, 1):
            new_state = move_tile(current_state, move)
            g += 1
            h = misplaced_tiles(new_state)
            f = g + h
            print(f"Move {i}: {move}")
            print_state(new_state)
            print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n")
            current_state = new_state
    else:
        print("No solution exists for the given start state.")
else:
```

```
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

```
Enter start state (e.g., 724506831): 283164705
Start state:
2 8 3
1 6 4
7   5

Total states visited: 7
Solution found!
Moves: U U L D R
Number of moves: 5
1BM23CS307 Uzair

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

g(n) = 1, h(n) = 3, f(n) = g(n) + h(n) = 4

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

g(n) = 2, h(n) = 3, f(n) = g(n) + h(n) = 5

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

g(n) = 3, h(n) = 2, f(n) = g(n) + h(n) = 5

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

g(n) = 4, h(n) = 1, f(n) = g(n) + h(n) = 5

Move 5: R
1 2 3
```

8 4
7 6 5

g(n) = 5, h(n) = 0, f(n) = g(n) + h(n) = 5

b) Manhattan Distance

```
import heapq

goal_state = '123456780'

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

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

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

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

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

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

def manhattan_distance(state):
    distance = 0
    for i, val in enumerate(state):
        if val == '0':
```

```
            continue
        goal_pos = int(val) - 1
        current_row, current_col = divmod(i, 3)
        goal_row, goal_col = divmod(goal_pos, 3)
        distance += abs(current_row - goal_row) + abs(current_col - goal_col)
    return distance

def a_star(start_state):
    visited_count = 0
    open_set = []
    heapq.heappush(open_set, (manhattan_distance(start_state), 0, start_state, []))
    visited = set()

    while open_set:
        f, g, current_state, path = heapq.heappop(open_set)
        visited_count += 1

        if current_state == goal_state:
            return path, visited_count

        if current_state in visited:
            continue
        visited.add(current_state)

        for direction in moves:
            new_state = move_tile(current_state, direction)
            if new_state and new_state not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(new_state)
                heapq.heappush(open_set, (new_f, new_g, new_state, path + [direction]))

    return None, visited_count

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

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

    result, visited_states = a_star(start)

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

    if result is not None:
```

```
        print("Solution found!")
        print("Moves:", ' '.join(result))
        print("Number of moves:", len(result))
        print("1BM23CS307 Uzair\n")

        current_state = start
        g = 0  # initialize cost so far
        for i, move in enumerate(result, 1):
            new_state = move_tile(current_state, move)
            g += 1
            h = manhattan_distance(new_state)
            f = g + h
            print(f"Move {i}: {move}")
            print_state(new_state)
            print(f"g(n) = {g}, h(n) = {h}, f(n) = g(n) + h(n) = {f}\n")
            current_state = new_state
    else:
        print("No solution exists for the given start state.")
else:
    print("Invalid input! Please enter a 9-digit string using digits 0-8 without repetition.")
```

Output:

```
Enter start state (e.g., 724506831): 123678450
Start state:
1 2 3
6 7 8
4 5

Total states visited: 21
Solution found!
Moves: L U L D R R U L D R
Number of moves: 10
1BM23CS307 Uzair

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

g(n) = 1, h(n) = 9, f(n) = g(n) + h(n) = 10

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

g(n) = 2, h(n) = 8, f(n) = g(n) + h(n) = 10

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

g(n) = 3, h(n) = 7, f(n) = g(n) + h(n) = 10

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

g(n) = 4, h(n) = 6, f(n) = g(n) + h(n) = 10

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

g(n) = 5, h(n) = 5, f(n) = g(n) + h(n) = 10

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

g(n) = 6, h(n) = 4, f(n) = g(n) + h(n) = 10

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

g(n) = 7, h(n) = 3, f(n) = g(n) + h(n) = 10

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

g(n) = 8, h(n) = 2, f(n) = g(n) + h(n) = 10

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

g(n) = 9, h(n) = 1, f(n) = g(n) + h(n) = 10

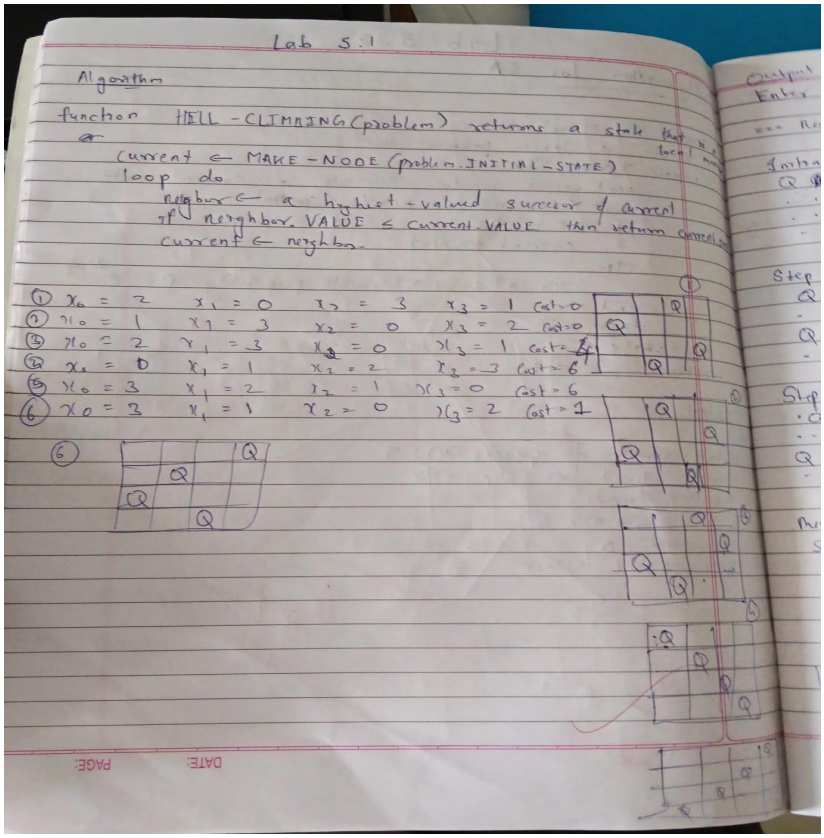
Move 10: R
1 2 3
4 5 6
7 8

g(n) = 10, h(n) = 0, f(n) = g(n) + h(n) = 10

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Code:

import random

```
import time

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

def compute_heuristic(state):
    """Computes the number of attacking pairs of queens."""
    h = 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):
                h += 1
    return h

def get_neighbors(state):
    """Generates all possible neighbors by moving one queen in its column."""
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if row != state[col]:
                neighbor = list(state)
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors

def hill_climb_verbose(initial_state, step_delay=0.5):
    """Hill climbing algorithm with verbose output at each step."""
    current = initial_state
    current_h = compute_heuristic(current)
```

```
    step = 0

    print(f"Initial state (heuristic: {current_h}):")
    print_board(current)
    time.sleep(step_delay)

    while True:
        neighbors = get_neighbors(current)
        next_state = None
        next_h = current_h

        for neighbor in neighbors:
            h = compute_heuristic(neighbor)
            if h < next_h:
                next_state = neighbor
                next_h = h

        if next_h >= current_h:
            print(f"Reached local minimum at step {step}, heuristic: {current_h}")
            return current, current_h

        current = next_state
        current_h = next_h
        step += 1
        print(f"Step {step}: (heuristic: {current_h})")
        print_board(current)
        time.sleep(step_delay)

def solve_n_queens_verbose(n, max_restarts=1000):
    """Solves N-Queens problem using hill climbing with restarts and verbose output."""
    for attempt in range(max_restarts):
        print(f"\n=== Restart {attempt + 1} ===\n")
        initial_state = [random.randint(0, n - 1) for _ in range(n)]
        solution, h = hill_climb_verbose(initial_state)
        if h == 0:
            print(f"✅ Solution found after {attempt + 1} restart(s):")
            print_board(solution)
            return solution
        else:
            print(f"❌ No solution in this attempt (local minimum).\n")
    print("Failed to find a solution after max restarts.")
```

```
    return None

# --- Run the algorithm ---
if __name__ == "__main__":
    N = int(input("Enter the number of queens (N): "))
    solve_n_queens_verbose(N)
    print("1BM23CS307 Uzair")
```

Output:

```
Enter the number of queens (N): 4

=== Restart 1 ===

Initial state (heuristic: 3):
Q . Q .
. Q . .
. . . Q
. . . .

Step 1: (heuristic: 1)
. . Q .
. Q . .
. . . Q
Q . . .

Reached local minimum at step 1, heuristic: 1
```

❌ No solution in this attempt (local minimum).

```
=== Restart 2 ===

Initial state (heuristic: 3):
. Q . .
. . Q .
. . . .
Q . . Q

Step 1: (heuristic: 1)
. Q . .
. . Q .
Q . . .
```

. . . Q

Reached local minimum at step 1, heuristic: 1
❌ No solution in this attempt (local minimum).

=== Restart 3 ===

Initial state (heuristic: 2):
. . . .
. Q . Q
. . . .
Q . Q .

Step 1: (heuristic: 1)
. Q . .
. . . Q
. . . .
Q . Q .

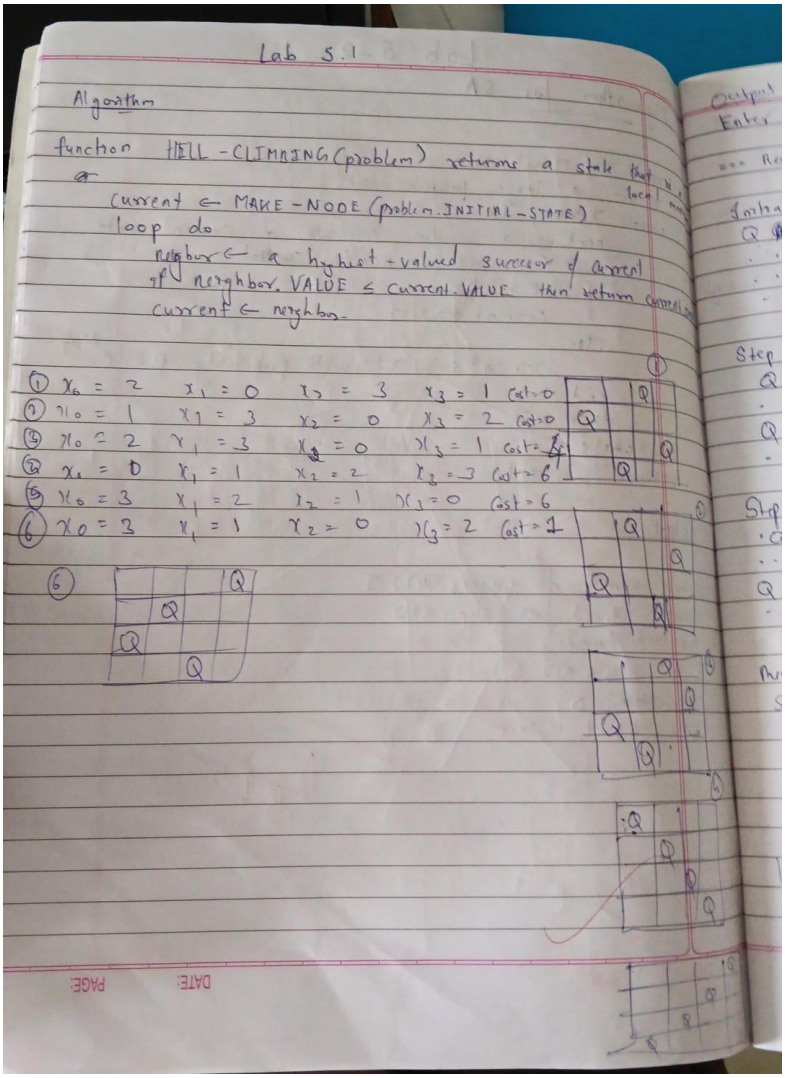
Step 2: (heuristic: 0)
. Q . .
. . . Q
Q . . .
. . Q .

Reached local minimum at step 2, heuristic: 0
✅ Solution found after 3 restart(s):
. Q . .
. . . Q
Q . . .
. . Q .

1BM23CS307 Uzair

Program 5

Simulated Annealing to Solve 8-Queens problem



Code:

```
import random
import math

def compute_heuristic(state):
    """Number of attacking pairs."""
    h = 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):
```

```
            h += 1
    return h

def random_neighbor(state):
    """Returns a neighbor by randomly changing one queen's row."""
    n = len(state)
    neighbor = state[:]
    col = random.randint(0, n - 1)
    old_row = neighbor[col]
    new_row = random.choice([r for r in range(n) if r != old_row])
    neighbor[col] = new_row
    return neighbor

def dual_simulated_annealing(n, max_iter=10000, initial_temp=100.0, cooling_rate=0.99):
    """Simulated Annealing with dual acceptance strategy."""
    current = [random.randint(0, n - 1) for _ in range(n)]
    current_h = compute_heuristic(current)
    temperature = initial_temp

    for step in range(max_iter):
        if current_h == 0:
            print(f"✅ Solution found at step {step}")
            return current

        neighbor = random_neighbor(current)
        neighbor_h = compute_heuristic(neighbor)
        delta = neighbor_h - current_h

        if delta < 0:
            current = neighbor
            current_h = neighbor_h
        else:
            # Dual acceptance: standard + small chance of higher uphill move
            probability = math.exp(-delta / temperature)
            if random.random() < probability:
                current = neighbor
                current_h = neighbor_h

        temperature *= cooling_rate
        if temperature < 1e-5:  # Restart if stuck
            temperature = initial_temp
```

```
            current = [random.randint(0, n - 1) for _ in range(n)]
            current_h = compute_heuristic(current)

    print("❌ Failed to find solution within max iterations.")
    return None

# --- Run the algorithm ---
if __name__ == "__main__":
    N = int(input("Enter number of queens (N): "))
    solution = dual_simulated_annealing(N)

    if solution:
        print("Position format:")
        print("[", " ".join(str(x) for x in solution), "]")
        print("Heuristic:", compute_heuristic(solution))
        print("1BM23CS307 Uzair")
```

Output:

Enter number of queens (N): 8

✅ Solution found at step 675

Position format:

[3 0 4 7 5 2 6 1]

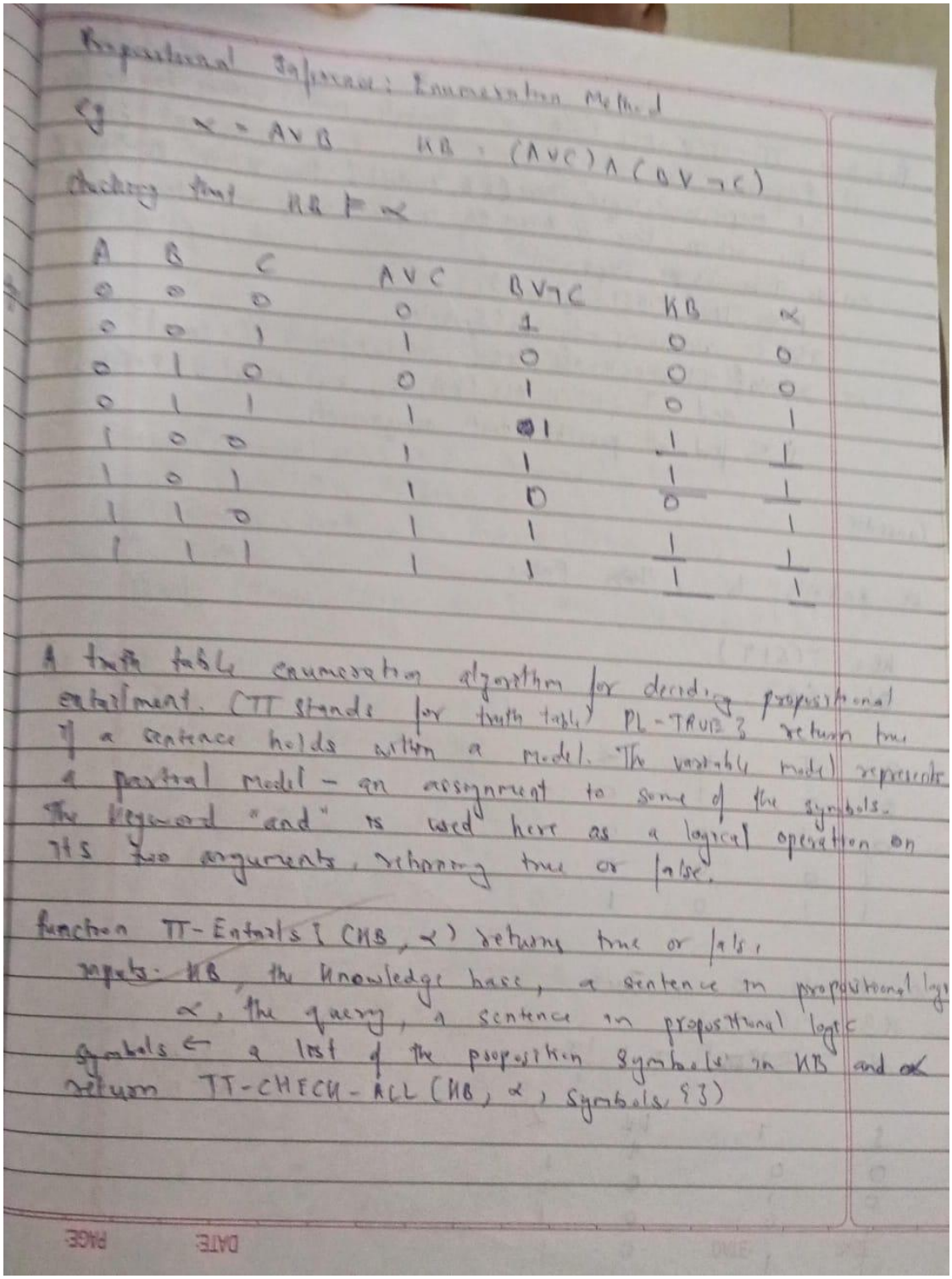
Heuristic: 0

1BM23CS307 Uzair

Program 6

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

Algorithm:



Propositional Inference: Enumeration Method

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

Checking that $KB \models \alpha$

A	B	C	A ∨ C	B ∨ ¬C	KB	α
0	0	0	0	1	0	0
0	0	1	1	0	0	0
0	1	0	0	1	0	0
0	1	1	1	1	1	1
1	0	0	1	1	1	1
1	0	1	1	0	0	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

A truth table enumeration algorithm for deciding propositional entailment. (TT stands for truth table) PL-TRUE? return true if a sentence holds within a model. The variable model represents a partial model - an assignment to some of the symbols. The keyword "and" is used here as a logical operation on its two arguments, returning true or false.

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 Symbols then

if propositional-logic-True (KB, model) then return PL-True

else return true // when KB is false, always return true

else do // recursive steps

P ← TAKE_FIRST (Symbols)

rest ← remaining_Symbols (Symbols)

return (TT-CHECK-ALL (KB, α, rest, model ∪ {P = true})

and TT-CHECK-ALL (KB, α, rest, model ∪ {P = false}))

// To find all possible truth assignments.

Consider

a entails b , True False

KB = ~(S|T)

α = S&T

S	T	KB	α
0	0	1	0
0	1	0	0
1	0	0	0
1	1	0	1

c entails c ; True

KB = ~(S|T)

α = ~~T~~ T | !T

S	T	KB	α
0	0	1	1
0	1	0	1
1	0	0	1
1	1	0	1

PAGE:

DATE:

Code:

```
from itertools import product

# ----------------- Propositional Logic Symbols -----------------
class Symbol:
    def __init__(self, name):
        self.name = name

    def __invert__(self):  # ~P
        return Not(self)

    def __and__(self, other):  # P & Q
        return And(self, other)

    def __or__(self, other):  # P | Q
        return Or(self, other)

    def __rshift__(self, other):  # P >> Q (implication)
        return Or(Not(self), other)

    def __eq__(self, other):  # P == Q (biconditional)
        return And(Or(Not(self), other), Or(Not(other), self))

    def eval(self, model):
        return model[self.name]

    def symbols(self):
        return {self.name}

    def __repr__(self):
        return self.name

class Not:
    def __init__(self, operand):
        self.operand = operand

    def eval(self, model):
        return not self.operand.eval(model)

    def symbols(self):
        return self.operand.symbols()
```

```
    def __repr__(self):
        return f"~{self.operand}"

    def __invert__(self):  # allow ~~A
        return Not(self)


class And:
    def __init__(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) and self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def __repr__(self):
        return f"({self.left} & {self.right})"

    def __invert__(self):  # allow ~(A & B)
        return Not(self)


class Or:
    def __init__(self, left, right):
        self.left, self.right = left, right

    def eval(self, model):
        return self.left.eval(model) or self.right.eval(model)

    def symbols(self):
        return self.left.symbols() | self.right.symbols()

    def __repr__(self):
        return f"({self.left} | {self.right})"

    def __invert__(self):  # allow ~(A | B)
        return Not(self)
```

```
# ----------------- Truth Table Entailment -----------------
def tt_entails(kb, alpha, show_table=False):
    symbols = sorted(list(kb.symbols() | alpha.symbols()))
    if show_table:
        print_truth_table(kb, alpha, symbols)
    return tt_check_all(kb, alpha, symbols, {})


def tt_check_all(kb, alpha, symbols, model):
    if not symbols:  # all symbols assigned
        if kb.eval(model):  # KB is true
            return alpha.eval(model)
        else:
            return True  # if KB is false, entailment holds
    else:
        P, rest = symbols[0], symbols[1:]

        model_true = model.copy()
        model_true[P] = True
        result_true = tt_check_all(kb, alpha, rest, model_true)

        model_false = model.copy()
        model_false[P] = False
        result_false = tt_check_all(kb, alpha, rest, model_false)

        return result_true and result_false


# ----------------- Truth Table Printer -----------------
def print_truth_table(kb, alpha, symbols):
    header = symbols + ["KB", "Query"]
    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 = kb.eval(model)
        alpha_val = alpha.eval(model)
        row = [str(model[s]) for s in symbols] + [str(kb_val), str(alpha_val)]
        print(" | ".join(f"{r:^5}" for r in row))
```

```
    print()


# ----------------- Example -----------------
S = Symbol("S")
C = Symbol("C")
T = Symbol("T")


c = T | ~T

# KB: P → Q
kb1 = ~ (S|T)
# Query: Q
alpha1 = S & T


print("Knowledge Base:", kb1)
print("Query:", alpha1)
print()
result = tt_entails(kb1, alpha1, show_table=True)
print("Does KB entail Query?", result)
```

Output:

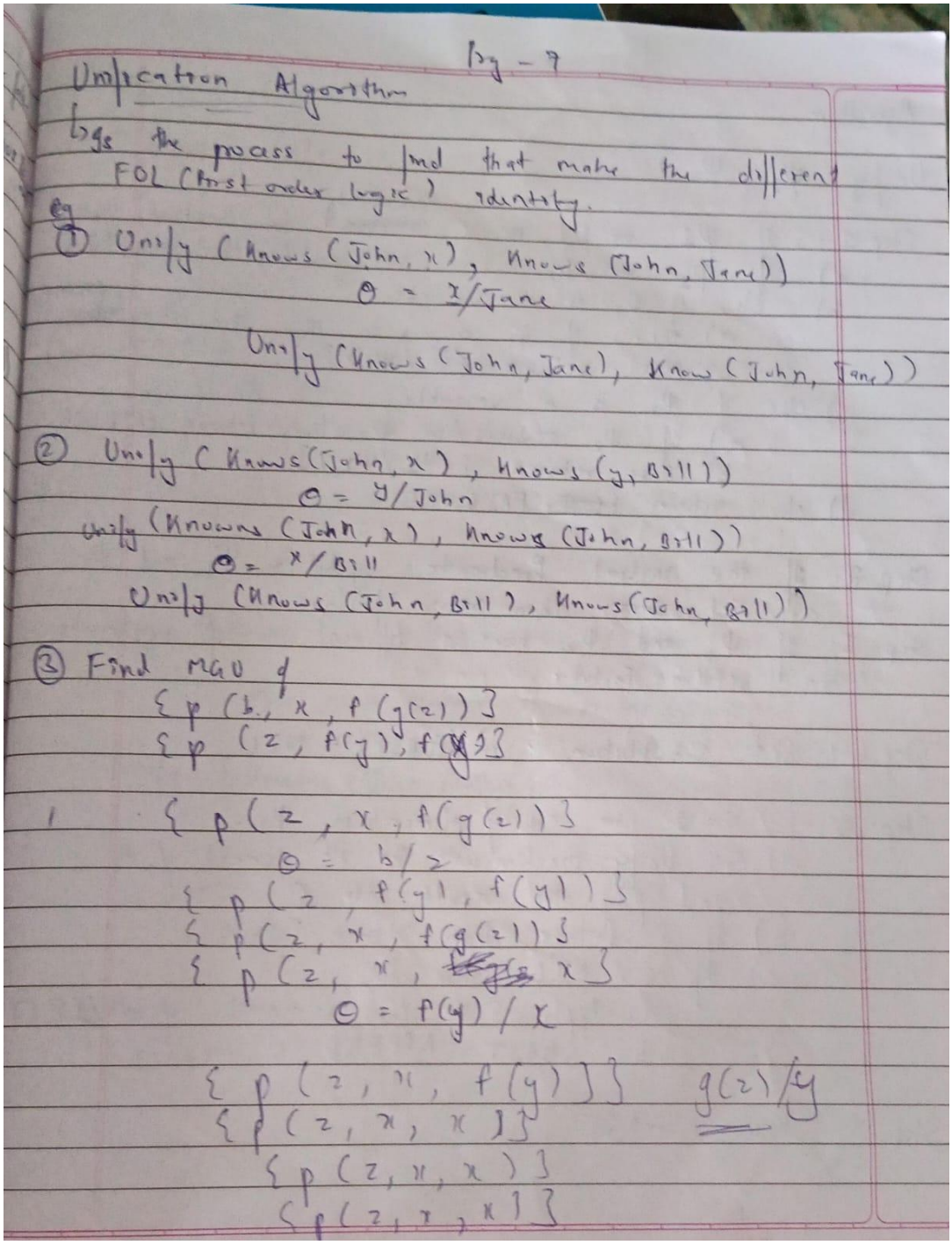
```
Knowledge Base: (P | (Q & P))
Query: (Q | P)

  P   |   Q   |  KB   | Query
-----------------------------
False | False | False | False
False | True  | False | True
True  | False | True  | True
True  | True  | True  | True

Does KB entail Query? True
```

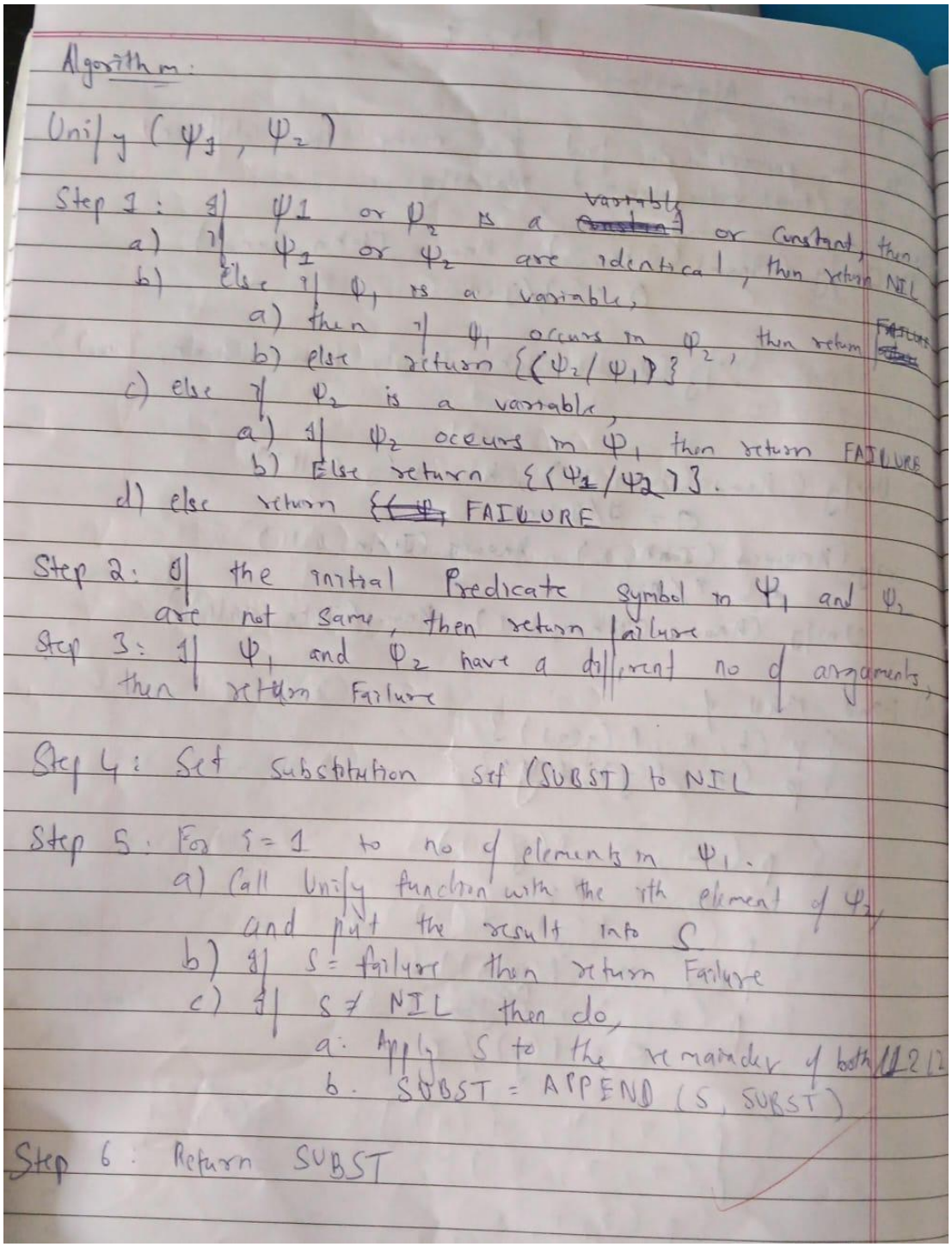
1BM23CS307 Uzair

Program 7

Implement unification in first order logic

Algorithm:





Code:

```
class UnificationError(Exception):
    pass

def occurs_check(var, term):
```

```
    """Check if a variable occurs in a term (to prevent infinite recursion)."""
    if var == term:
        return True
    if isinstance(term, tuple):  # Term is a compound (function term)
        return any(occurs_check(var, subterm) for subterm in term)
    return False

def unify(term1, term2, substitutions=None):
    """Try to unify two terms, return the MGU (Most General Unifier)."""
    if substitutions is None:
        substitutions = {}

    # If both terms are equal, no further substitution is needed
    if term1 == term2:
        return substitutions

    # If term1 is a variable, we substitute it with term2
    elif isinstance(term1, str) and term1.isupper():
        # If term1 is already substituted, recurse
        if term1 in substitutions:
            return unify(substitutions[term1], term2, substitutions)
        elif occurs_check(term1, term2):
            raise UnificationError(f"Occurs check fails: {term1} in {term2}")
        else:
            substitutions[term1] = term2
            return substitutions

    # If term2 is a variable, we substitute it with term1
    elif isinstance(term2, str) and term2.isupper():
        # If term2 is already substituted, recurse
        if term2 in substitutions:
            return unify(term1, substitutions[term2], substitutions)
        elif occurs_check(term2, term1):
            raise UnificationError(f"Occurs check fails: {term2} in {term1}")
        else:
            substitutions[term2] = term1
            return substitutions

    # If both terms are compound (i.e., functions), unify their parts recursively
    elif isinstance(term1, tuple) and isinstance(term2, tuple):
        # Ensure that both terms have the same "functor" and number of arguments
```

```
        # if len(term1) != len(term2):
        #     raise UnificationError(f"Function arity mismatch: {term1} vs {term2}")

        for subterm1, subterm2 in zip(term1, term2):
            substitutions = unify(subterm1, subterm2, substitutions)

        return substitutions

    else:
        raise UnificationError(f"Cannot unify: {term1} with {term2}")

# Define the terms as tuples
term1 = ('p', 'b', 'X', ('f', ('g', 'Z')))
term2 = ('p', 'Z', ('f', 'Y'), ('f', 'Y'))

try:
    # Find the MGU
    result = unify(term1, term2)
    print("Most General Unifier (MGU):")
    print(result)
except UnificationError as e:
    print(f"Unification failed: {e}")
finally:
    print("1BM23CS307 UZAIR")
```

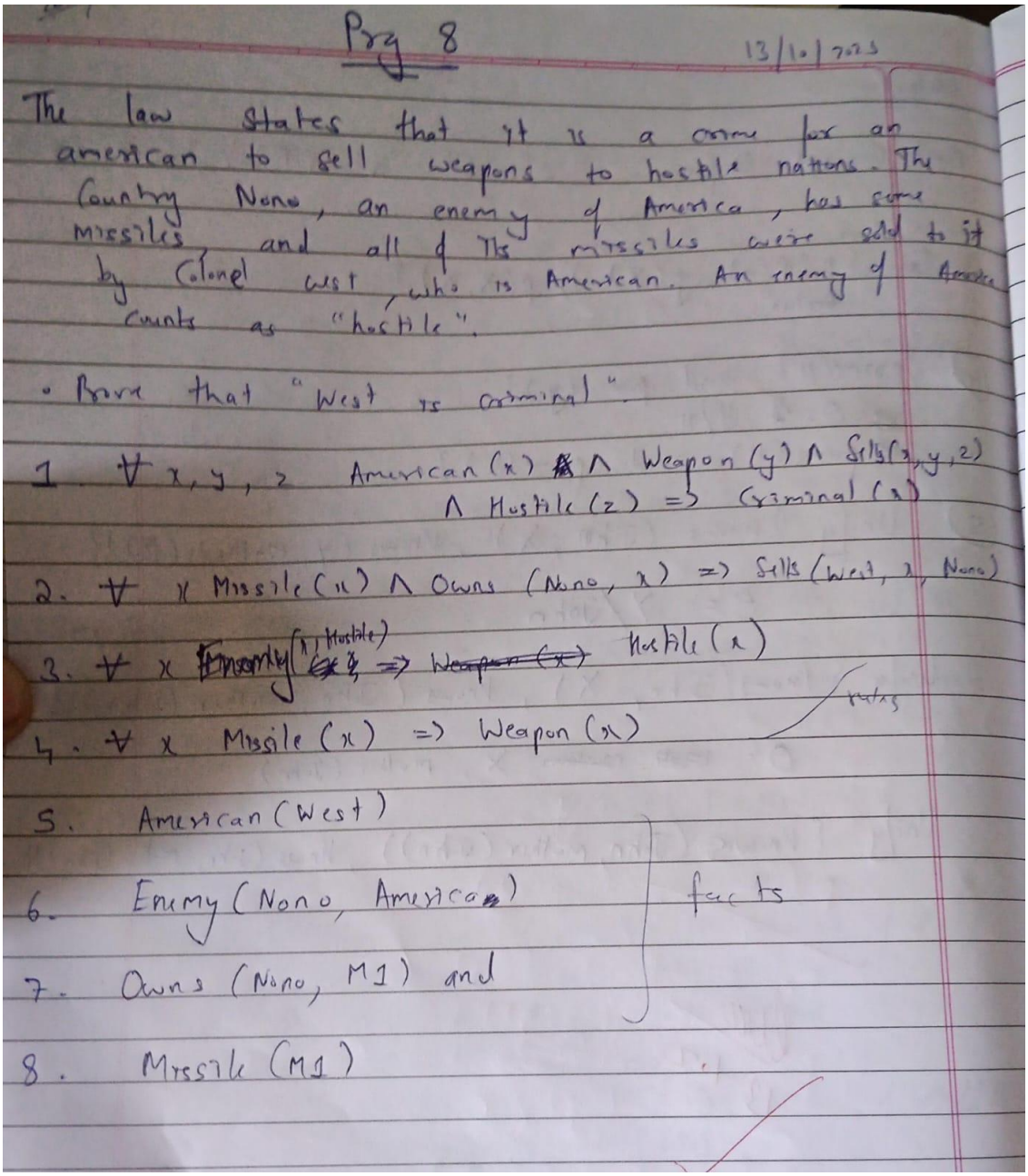
Output:

```
Most General Unifier (MGU):
{'Z': 'b', 'X': ('f', 'Y'), 'Y': ('g', 'Z')}
1BM23CS307 UZAIR
```

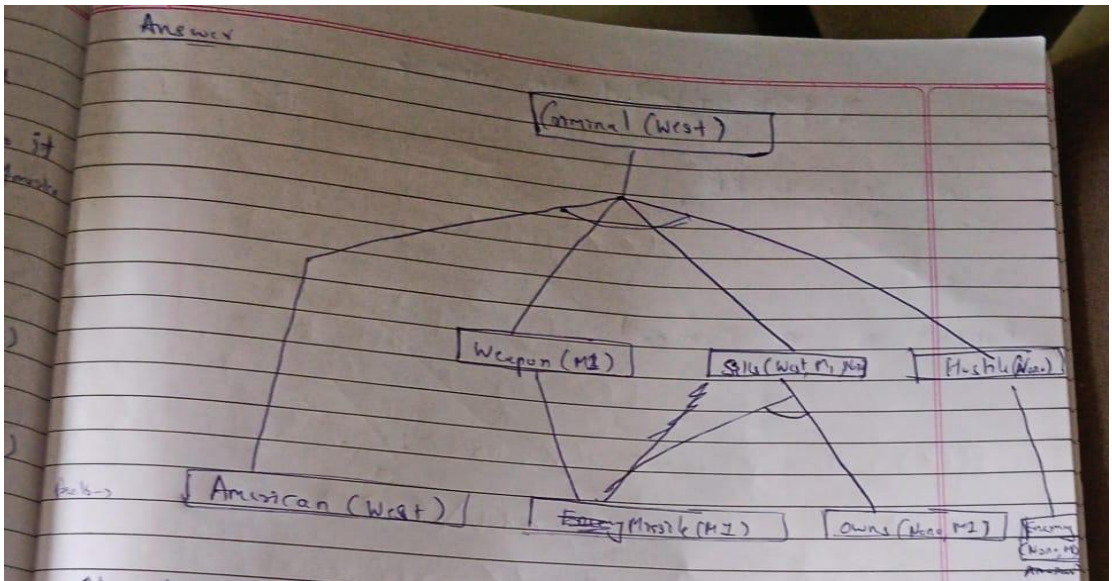
Program 8

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

Algorithm:



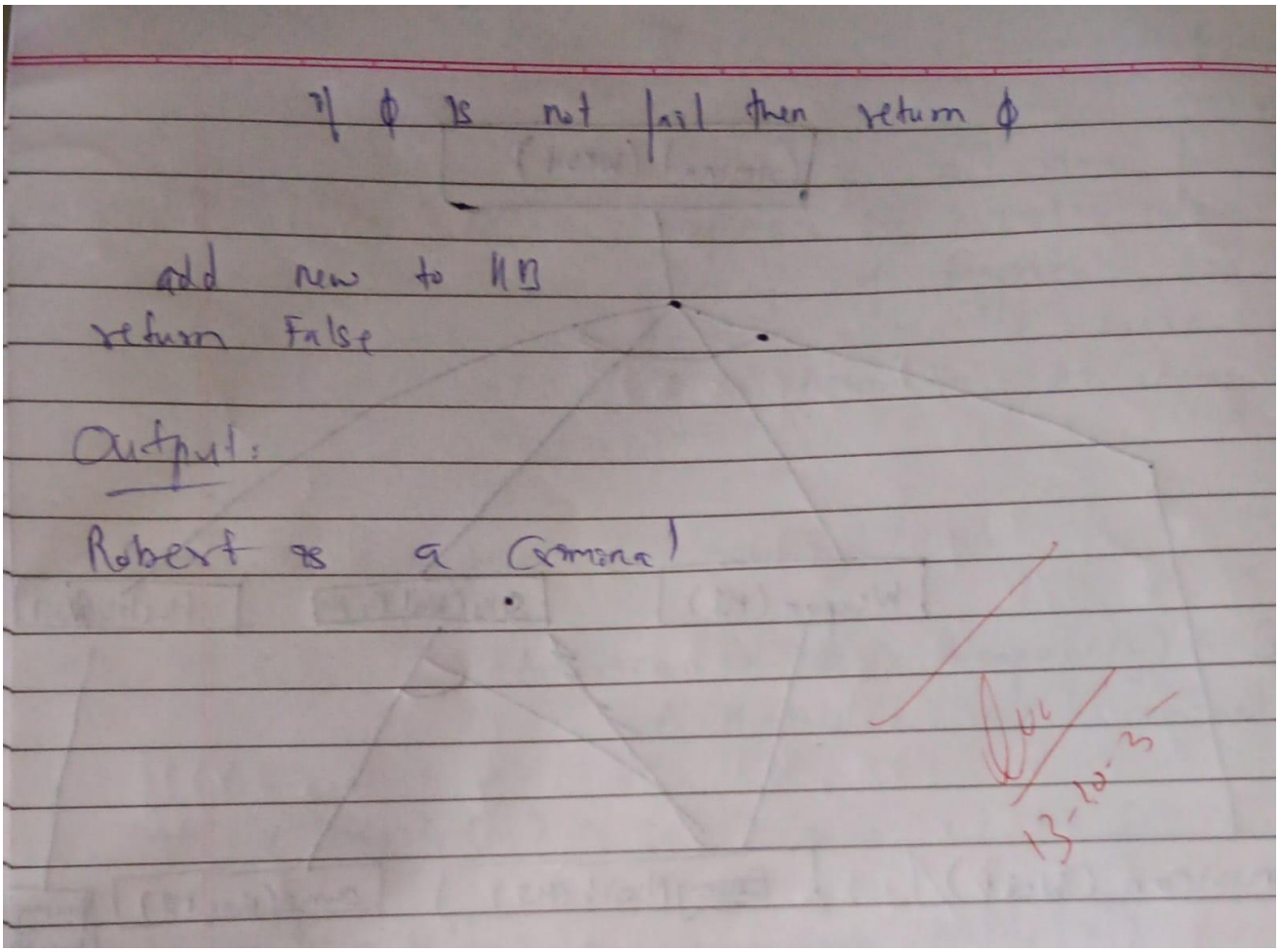
Answer



Algorithm

function FOL-FC-ASK (KB, α) returns a substitution or false
inputs: KB, set of first order
α, the query, an atomic sentence

local variables: new, new sentences inferred on each iteration
repeat until new is empty
new $\leftarrow$ { }
for each rule in KB do
$(p_1 \wedge \ldots \wedge p_n \Rightarrow q) \leftarrow$ Standardize-Variables(rule)
for each θ such that SUBST(θ, $p_1 \wedge \ldots \wedge p_n$)
= SUBST(θ, $p_1' \wedge \ldots \wedge p_n'$)
for some $p_1', \ldots, p_n'$ in KB
$q' \leftarrow$ SUBST(θ, q)
if q' does not unify with some sentence already
in KB or new then
add q' to new
$\phi \leftarrow$ Unify(q', α)



Code:

```
# Define the knowledge base
facts = {
    'American(Robert)': True,  # Robert is an American
    'Hostile(A)': True,        # Country A is hostile to America
    'Sells_Weapons(Robert, A)': True  # Robert sold weapons to Country A
}

# Define the law/rule: If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
def forward_reasoning(facts):
    # Apply the rule: If American(X) and Hostile(Y) and Sells_Weapons(X, Y), then Crime(X)
    if facts.get('American(Robert)', False) and facts.get('Hostile(A)', False) and
facts.get('Sells_Weapons(Robert, A)', False):
        facts['Crime(Robert)'] = True  # Robert is a criminal

# Perform forward reasoning to see if we can deduce that Robert is a criminal
forward_reasoning(facts)
```

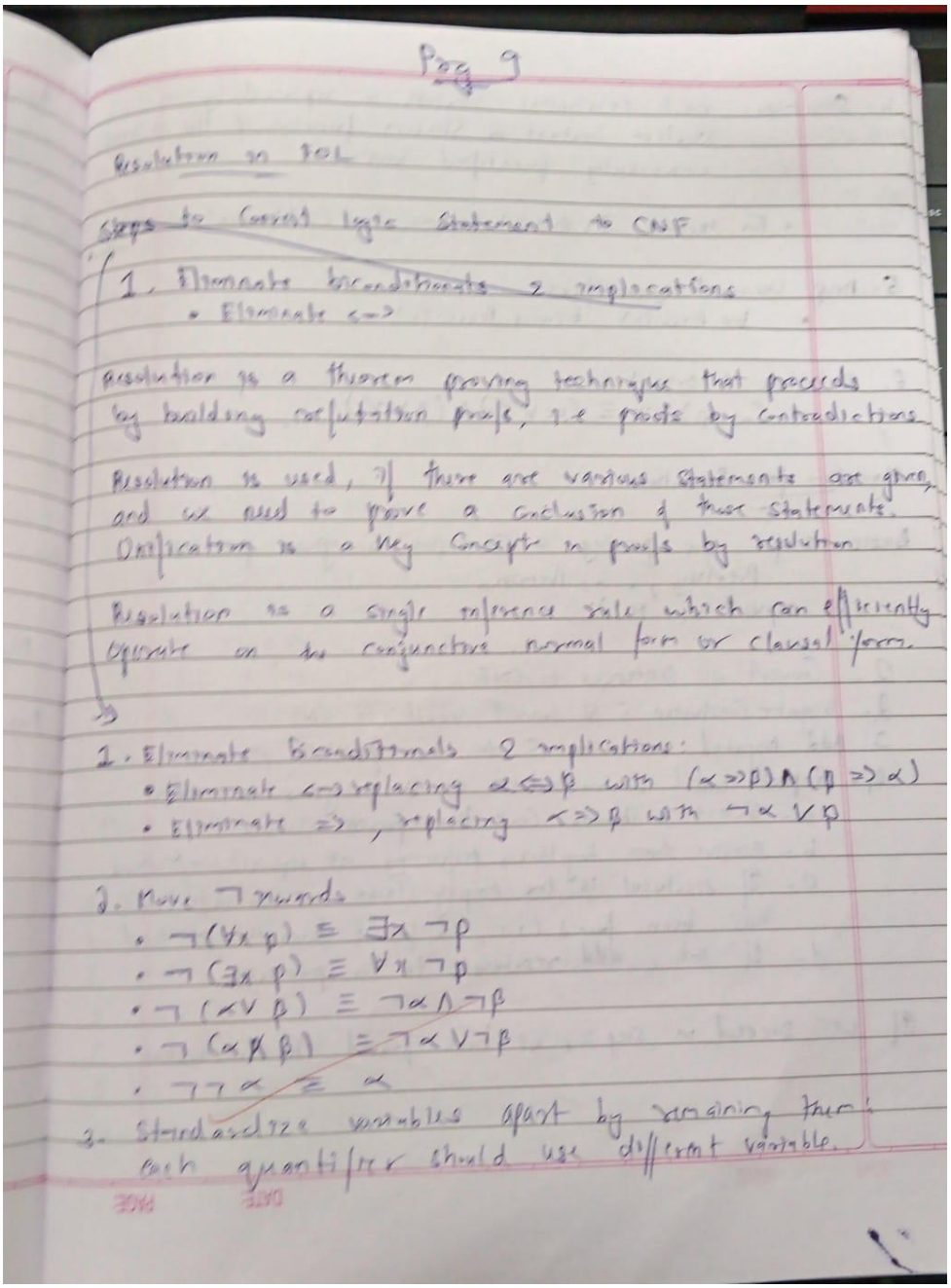
```
# Output the result based on the fact derived
if facts.get('Crime(Robert)', False):
    print("Robert is a criminal.")
    print("Uzair 1BM23CS307")
else:
    print("Robert is not a criminal.")
```

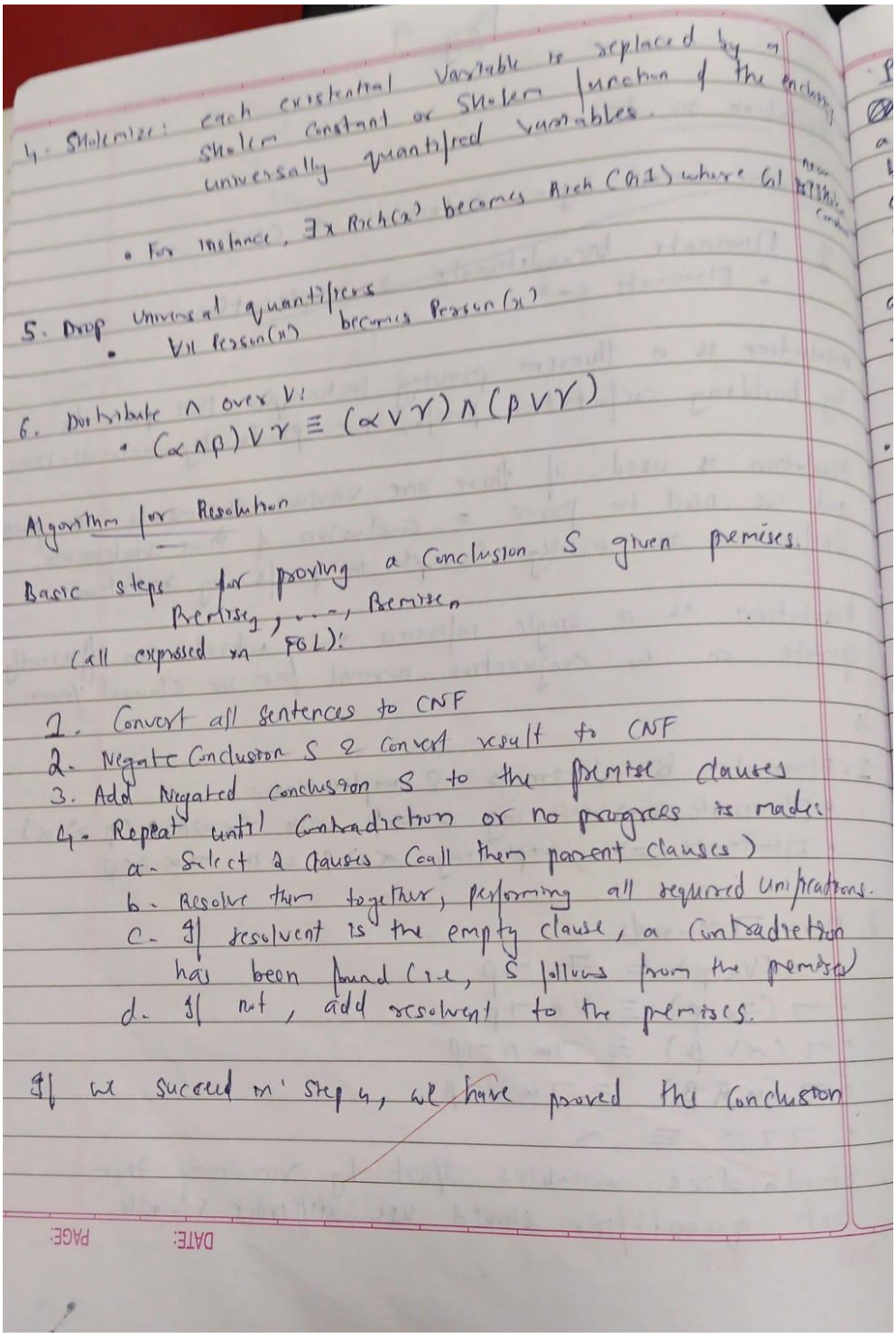
Output:
Robert is a criminal.
Uzair 1BM23CS307

Program 9

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

Algorithm:





4. Skolemize: each existential variable is replaced by a Skolem constant or Skolem function of the enclosing universally quantified variables.
   - For instance, $\exists x\ Rich(x)$ becomes $Rich(G1)$ where $G1$ is a new Skolem constant.

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

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

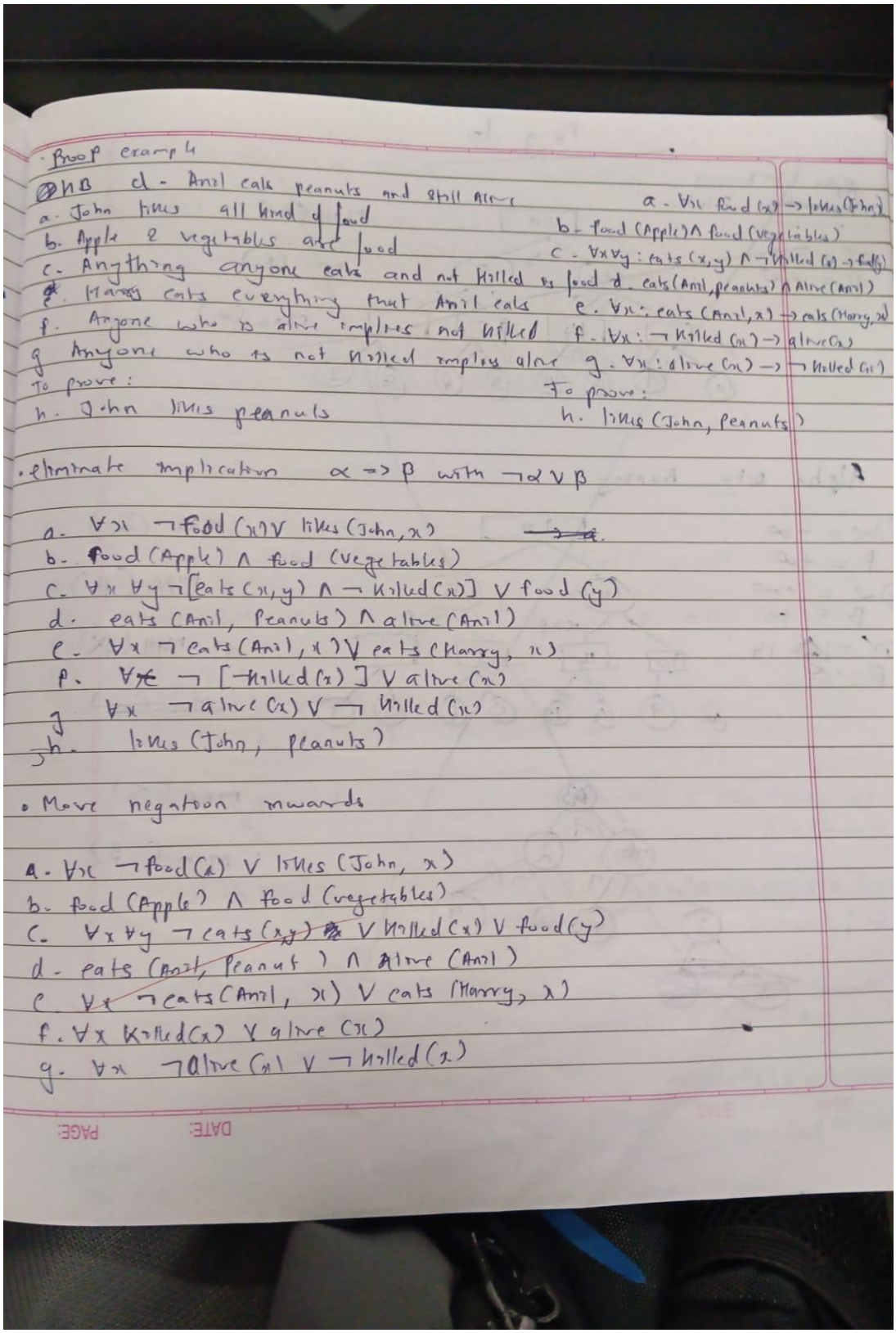
Algorithm for Resolution

Basic steps for proving a conclusion S given premises.
Premise$_1$, ..., Premise$_n$
(all expressed in FOL):

1. Convert all sentences to CNF
2. Negate conclusion S & convert result to CNF
3. Add negated conclusion S to the 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 the empty clause, a contradiction has been found (i.e., S follows from the premises)
   d. If not, add resolvent to the premises.

If we succeed in step 4, we have proved the conclusion.

DATE: PAGE:



Proof example

d - Anil eats peanuts and still Alive

a. John likes all kind of food
b. Apple & vegetables are food
c. Anything anyone eats and not killed is food
e. Harry eats everything that Anil eats
f. Anyone who is alive implies not killed
g. Anyone who is not killed implies alive

To prove:
h. John likes peanuts

a. $\forall x$ food $(x) \rightarrow$ likes (John)
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)

To prove:
h. likes (John, peanuts)

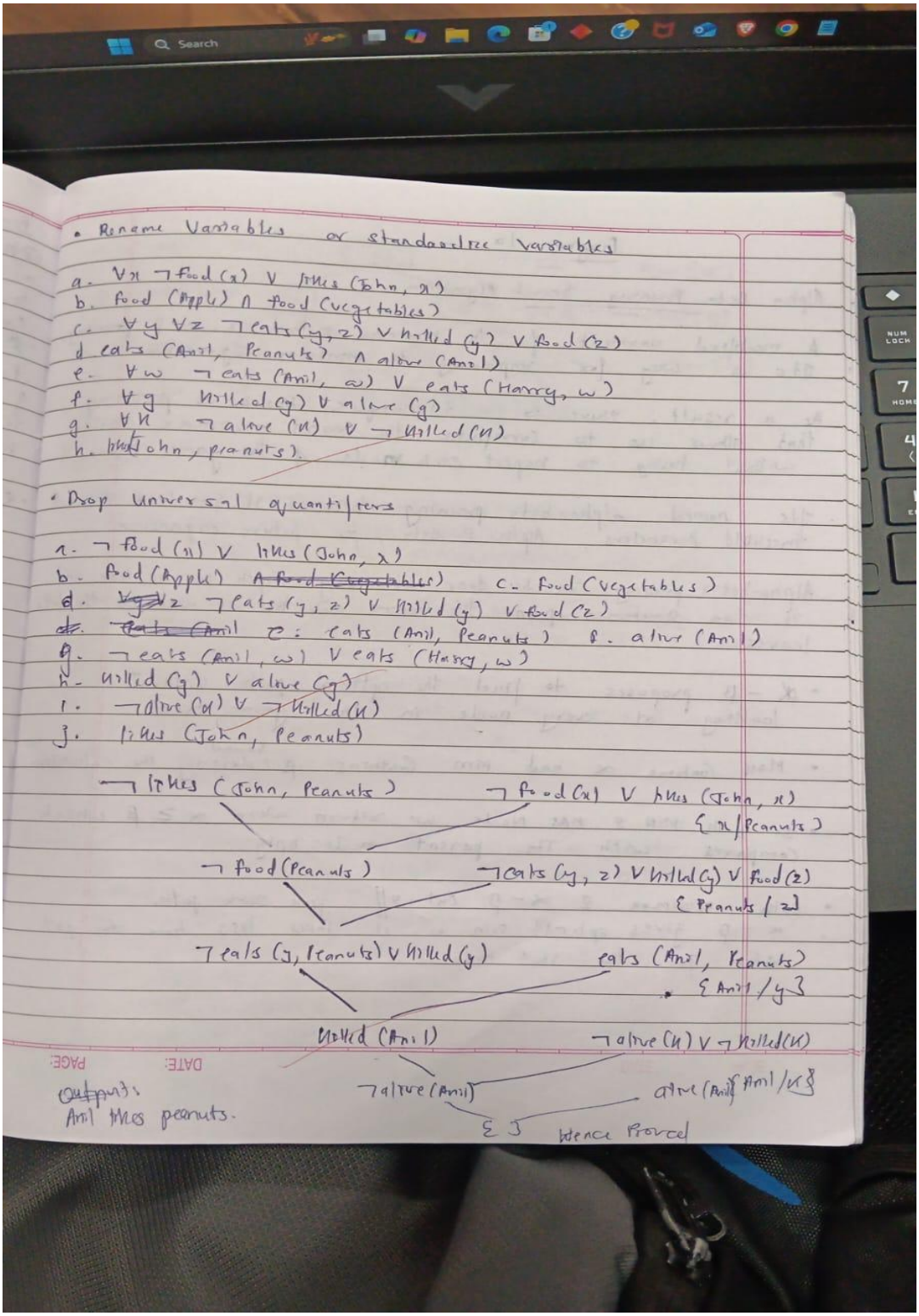
- eliminate implication $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

a. $\forall x \neg$ food $(x) \vee$ likes (John, x)
b. food (Apple) $\wedge$ food (vegetables)
c. $\forall x \forall y \neg$ [eats $(x,y) \wedge \neg$ killed (x)] $\vee$ food (y)
d. eats (Anil, peanuts) $\wedge$ alive (Anil)
e. $\forall x \neg$ eats (Anil, x) $\vee$ eats (Harry, x)
f. $\forall x \neg$ [killed (x)] $\vee$ alive (x)
g. $\forall x \neg$ alive $(x) \vee \neg$ killed (x)
h. likes (John, peanuts)

- Move negation inwards

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

DATE: PAGE:


• Rename Variables or standardize variables
a. ∀x ¬food(x) ∨ likes(John, x)
b. food(Apple) ∧ food(vegetables)
c. ∀y ∀z ¬eats(y, z) ∨ killed(y) ∨ food(z)
d. eats(Anil, Peanuts) ∧ alive(Anil)
e. ∀w ¬eats(Anil, w) ∨ eats(Harry, w)
f. ∀g killed(g) ∨ alive(g)
g. ∀k ¬alive(k) ∨ ¬killed(k)
h. likes(John, peanuts).
• Drop universal quantifiers
i. ¬food(x) ∨ likes(John, x)
b. food(Apple) food(vegetables) c. food(vegetables)
d. ¬eats(y, z) ∨ killed(y) ∨ food(z)
e. eats(Anil, Peanuts) f. alive(Anil)
g. ¬eats(Anil, w) ∨ eats(Harry, w)
h. killed(g) ∨ alive(g)
i. ¬alive(k) ∨ ¬killed(k)
j. likes(John, Peanuts)
¬likes(John, Peanuts)
¬food(x) ∨ likes(John, x)
{x/Peanuts}
¬food(Peanuts)
¬eats(y, z) ∨ killed(y) ∨ food(z)
{Peanuts/z}
¬eats(y, Peanuts) ∨ killed(y)
eats(Anil, Peanuts)
{Anil/y}
killed(Anil)
¬alive(k) ∨ ¬killed(k)
¬alive(Anil)
alive(Anil)
{Anil/k}
{}
Hence Proved
DATE:
PAGE:
Anil likes peanuts.

Code:

```
import re
import copy

class Predicate:
    def __init__(self, predicate_string):
        self.predicate_string = predicate_string
        self.name, self.arguments, self.negative = self.parse_predicate(predicate_string)

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

    def negate(self):
        self.negative = not self.negative
        if self.predicate_string.startswith('~'):
            self.predicate_string = self.predicate_string[1:]
        else:
            self.predicate_string = '~' + self.predicate_string

    def unify_with_predicate(self, other):
        """Attempt to unify two predicates; return substitution dict or False."""
        if self.name != other.name or len(self.arguments) != len(other.arguments):
            return False
        subs = {}
        for a, b in zip(self.arguments, other.arguments):
            if a == b:
                continue
            if a[0].islower():
                subs[a] = b
            elif b[0].islower():
                subs[b] = a
            else:
                return False
```

```
        return subs

    def substitute(self, subs):
        """Apply substitution dictionary."""
        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
class Statement:
    def __init__(self, statement_string):
        self.statement_string = statement_string
        self.predicate_set = self.parse_statement(statement_string)

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

    def add_statement_to_KB(self, KB, KB_HASH):
        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):
```

```
        """Resolve two statements; return new derived statements or False if contradiction."""
        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 self.predicate_set.union(other.predicate_set):
                        if pred not in (p1, p2):
                            pred_copy = copy.deepcopy(pred)
                            pred_copy.substitute(subs)
                            new_pred_set.add(pred_copy)
                    if not new_pred_set:
                        return False  # contradiction
                    new_stmt = Statement('|'.join(sorted([str(p) for p in new_pred_set])))
                    new_statements.add(new_stmt)
        return new_statements

    def __repr__(self):
        return self.statement_string

def fol_to_cnf_clauses(sentence):
    """
    Convert simple implications and conjunctions into CNF.
    Example:
        "A(x,y) => B(x,y)" becomes "~A(x,y)|B(x,y)"
        "A(x,y) & B(y,z) => C(x,z)" becomes "~A(x,y)|~B(y,z)|C(x,z)"
    """
    sentence = sentence.replace(' ', '')

    if '=>' in sentence:
        lhs, rhs = sentence.split('=>')
        parts = lhs.split('&')
        negated_lhs = ['~' + p for p in parts]
        disjunction = '|'.join(negated_lhs + [rhs])
        return [disjunction]

    # Split conjunctions into separate clauses
    if '&' in sentence:
```

```
        return sentence.split('&')

    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):
    KB2 = set()
    query.add_statement_to_KB(KB2, KB_HASH)
    query.add_statement_to_KB(KB, KB_HASH)
    while True:
        new_statements = set()
        if len(KB) > KILL_LIMIT:
            return False
        for s1 in KB:
            for s2 in s1.get_resolving_clauses(KB_HASH):
                if s1 == s2:
                    continue
                resolvents = s1.resolve(s2)
                if resolvents is False:
                    return True
                new_statements |= resolvents
        if new_statements.issubset(KB):
            return False
        new_statements -= KB
        KB |= new_statements

def main():
    fol_sentences = [
        "Parent(John, Mary)",
```

```
        "Parent(Mary, Sam)",
        "Parent(x, y) => Ancestor(x, y)",
        "Parent(x, y) & Ancestor(y, z) => Ancestor(x, z)"
    ]
    queries = ["Ancestor(John, Sam)"]

    KB, KB_HASH = prepare_knowledgebase(fol_sentences)

    print("\nKnowledge Base CNF Clauses:")
    for stmt in KB:
        print("  ", stmt)

    for query_str in queries:
        query_predicate = Predicate(query_str)
        query_predicate.negate()
        query_stmt = Statement(str(query_predicate))
        satisfiable = FOL_Resolution(copy.deepcopy(KB), copy.deepcopy(KB_HASH), query_stmt)
        print(f"\nQuery: {query_str} =>", "TRUE" if satisfiable else "FALSE")


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

Output:

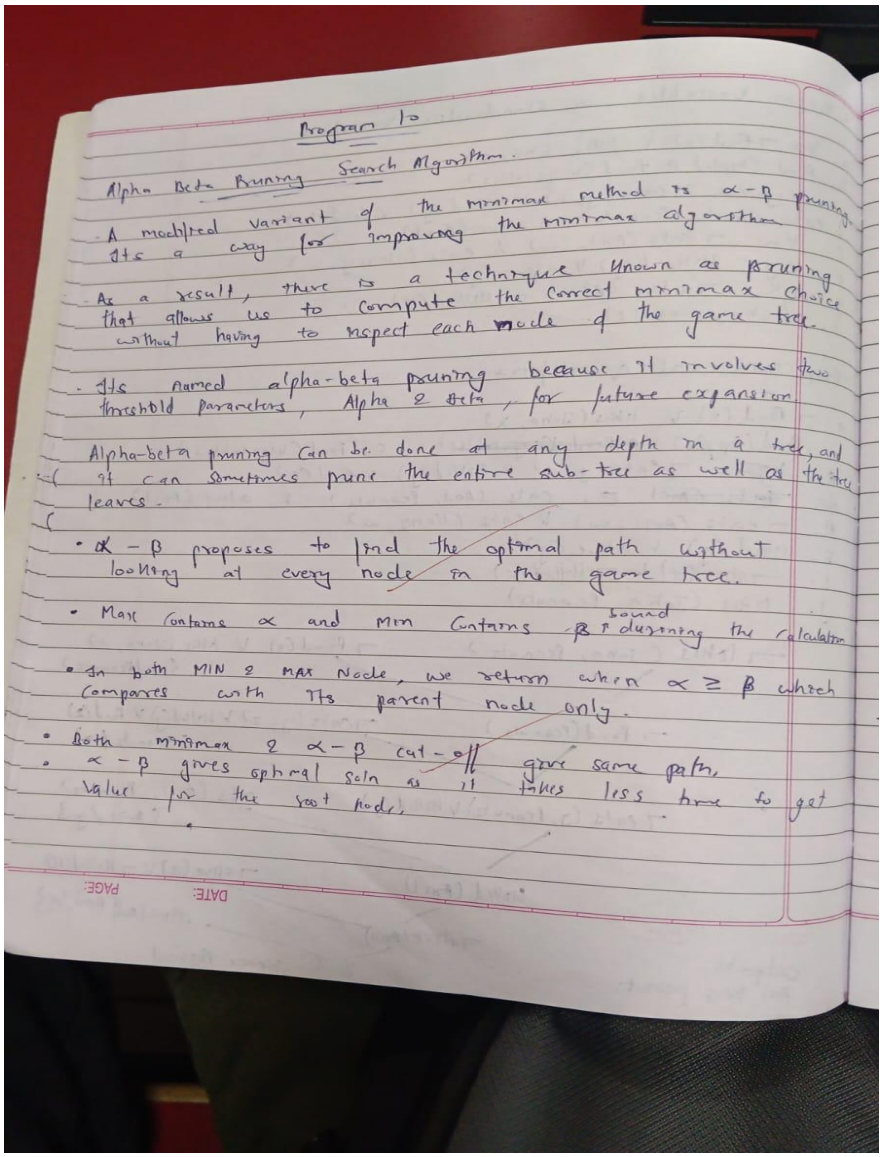
```
Knowledge Base CNF Clauses:
  ~Parent(x,y)|Ancestor(x,y)
  Parent(John,Mary)
  Parent(Mary,Sam)
  ~Parent(x,y)|~Ancestor(y,z)|Ancestor(x,z)

Query: Ancestor(John, Sam) => TRUE
```

1BM23CS307 Uzair

Program 10

Implement Alpha-Beta Pruning.

Algorithm:

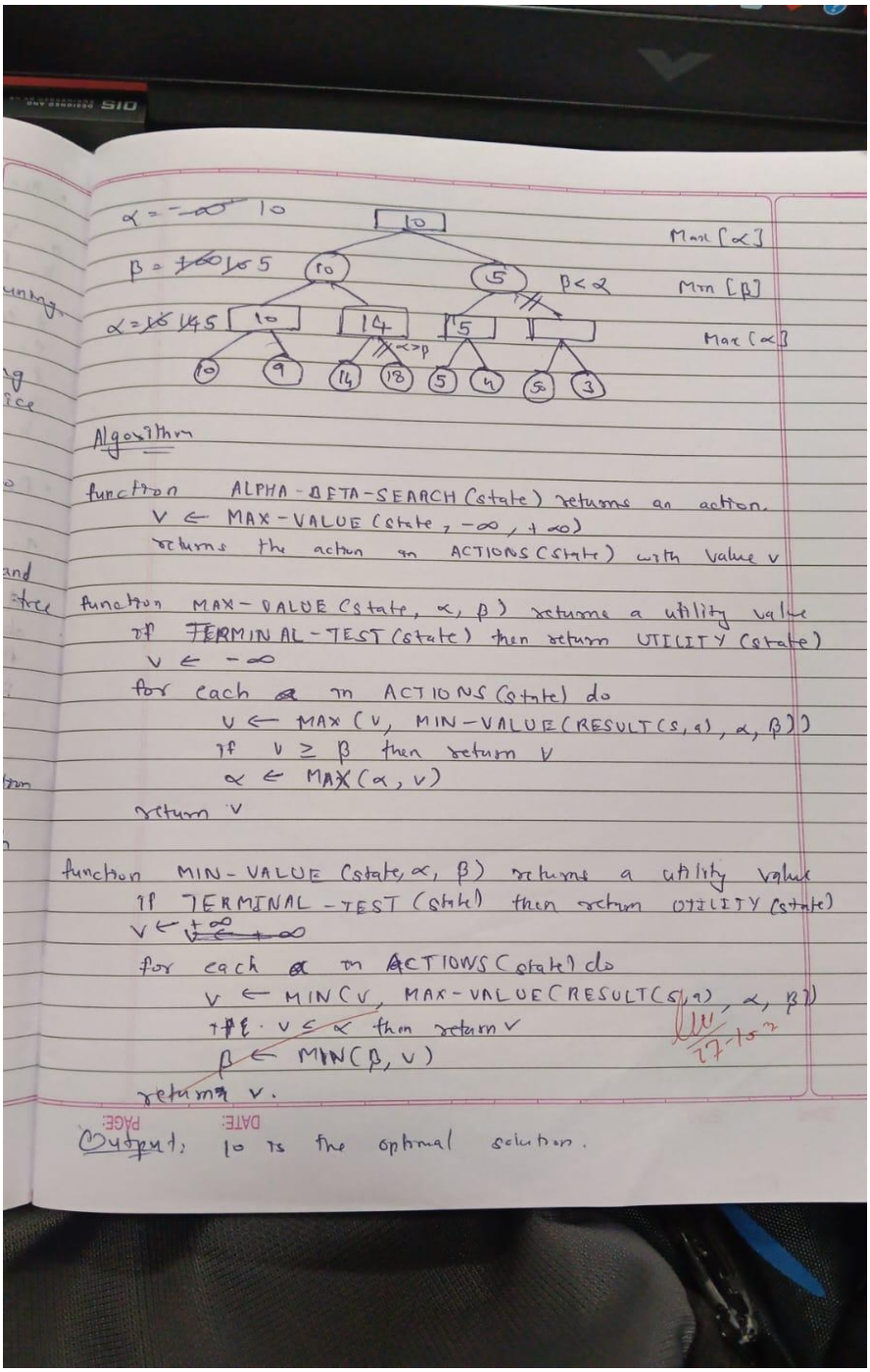


Program 10

Alpha Beta Pruning Search Algorithm.

- A modified variant of the minimax method is $\alpha - \beta$ pruning. It's a way for improving the minimax algorithm.
- As a result, there is a technique known as pruning that allows us to compute the correct minimax choice without having to inspect each node of the game tree.
- It's named alpha-beta pruning because it involves two threshold parameters, Alpha & Beta, for future expansion.
- Alpha-beta pruning can be done at any depth in a tree, and it can sometimes prune the entire sub-tree as well as the tree leaves.
- $\alpha - \beta$ proposes to find the optimal path without looking at every node in the game tree.
- Max contains α and Min contains β bound during the calculation reducing
- In both MIN & MAX Node, we return when $\alpha \geq \beta$ which compares with its parent node only.
- Both minimax & $\alpha - \beta$ cut-off give same path.
- $\alpha - \beta$ gives optimal soln as it takes less time to get value for the root node.

PAGE: DATE:



Code:

```
import math

def alpha_beta(node, depth, alpha, beta, maximizingPlayer, game_tree):
    """
    Alpha-Beta pruning search algorithm
```

```
    node: current node in game tree
    depth: current depth
    alpha: best value for maximizer
    beta: best value for minimizer
    maximizingPlayer: True if maximizer's turn
    game_tree: dictionary representing tree {node: children or value}
    """
    # If leaf node or depth 0, return its value
    if depth == 0 or isinstance(game_tree[node], int):
        return game_tree[node]

    if maximizingPlayer:
        maxEval = -math.inf
        for child in game_tree[node]:
            eval = alpha_beta(child, depth-1, alpha, beta, False, game_tree)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
                break  # Beta cut-off
        return maxEval
    else:
        minEval = math.inf
        for child in game_tree[node]:
            eval = alpha_beta(child, depth-1, alpha, beta, True, game_tree)
            minEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break  # Alpha cut-off
        return minEval

# Example game tree
game_tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': 3,
    'E': 5,
    'F': 2,
    'G': 9
}
```

```
best_value = alpha_beta('A', depth=3, alpha=-math.inf, beta=math.inf, maximizingPlayer=True,
game_tree=game_tree)
print("Best value for maximizer:", best_value)
```

Output:
Best value for maximizer: 3