VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

ANJALI PATEL (1BM23CS037)

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
Sep-2024 to Jan-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 **ANJALI PATEL (1BM23CS037),** who is a 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.

Mamtha mam Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

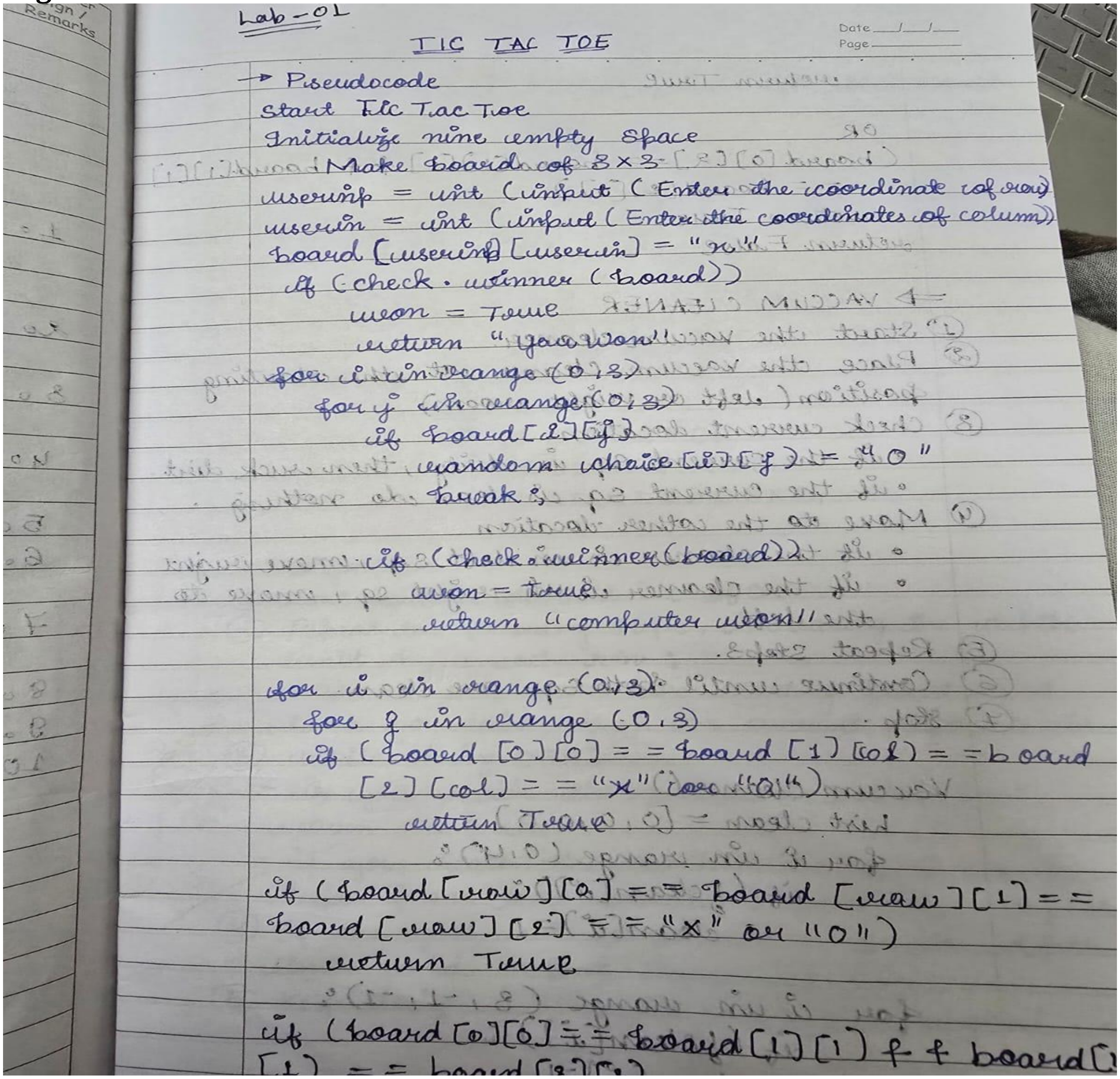
Index

Github Link:
https://github.com/ArchieJain120/AI_lab

Program 1

Implement Tic –Tac –Toe Game

Algorithm:



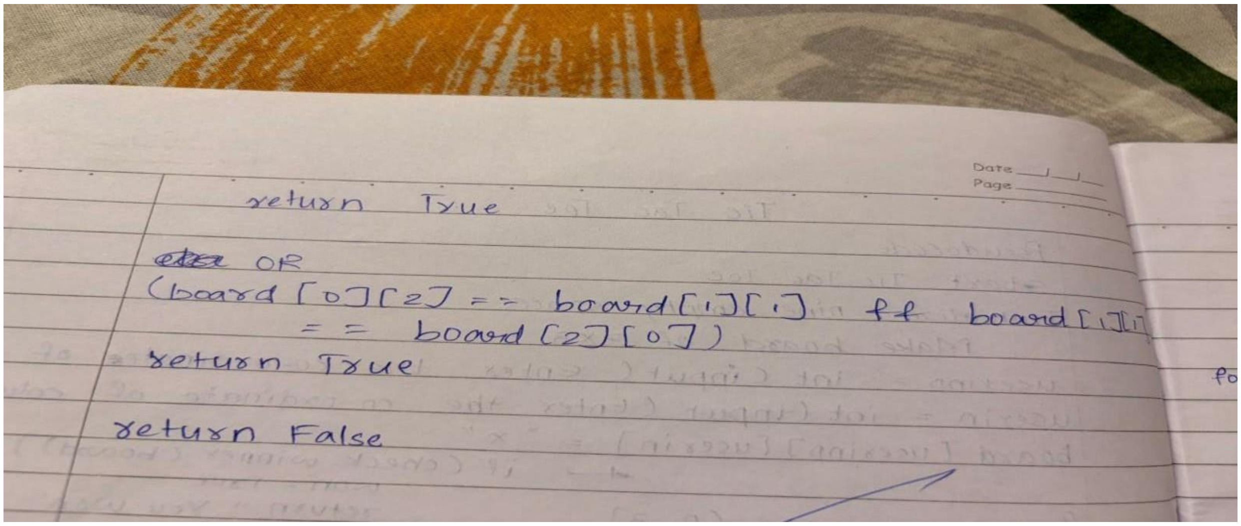
Lab - 01

TIC TAC TOE

Date __/__/__
Page _____

→ Pseudocode

```
Start Tic Tac Toe
Initialize nine empty space
Make board of 3 x 3
userinp = int (input ( Enter the coordinate of row)
userin = int (input ( Enter the coordinates of colum))
board [userinp] [userin] = "x"
 if (check . winner (board))
      won = True
      return "you won"
for i in range (0,3)
   for j in range(0,3)
      if board [i] [j]
      random choice [i] [j] = "0"
         break ;

   if (check . winner (board))
      won = True
         return "computer won"

for i in range (0,3)
   for j in range (0, 3)
   if ( board [0] [0] = = board [1] [col] = = board
      [2] [col] = = "x" (or "0")
         return True

if ( board [row] [0] = = board [row] [1] = =
board [row] [2] = = "x" or "0")
   return True

if (board [0][0] = = board [1] [1] & & board [
[1] = = board [2] [2]
```



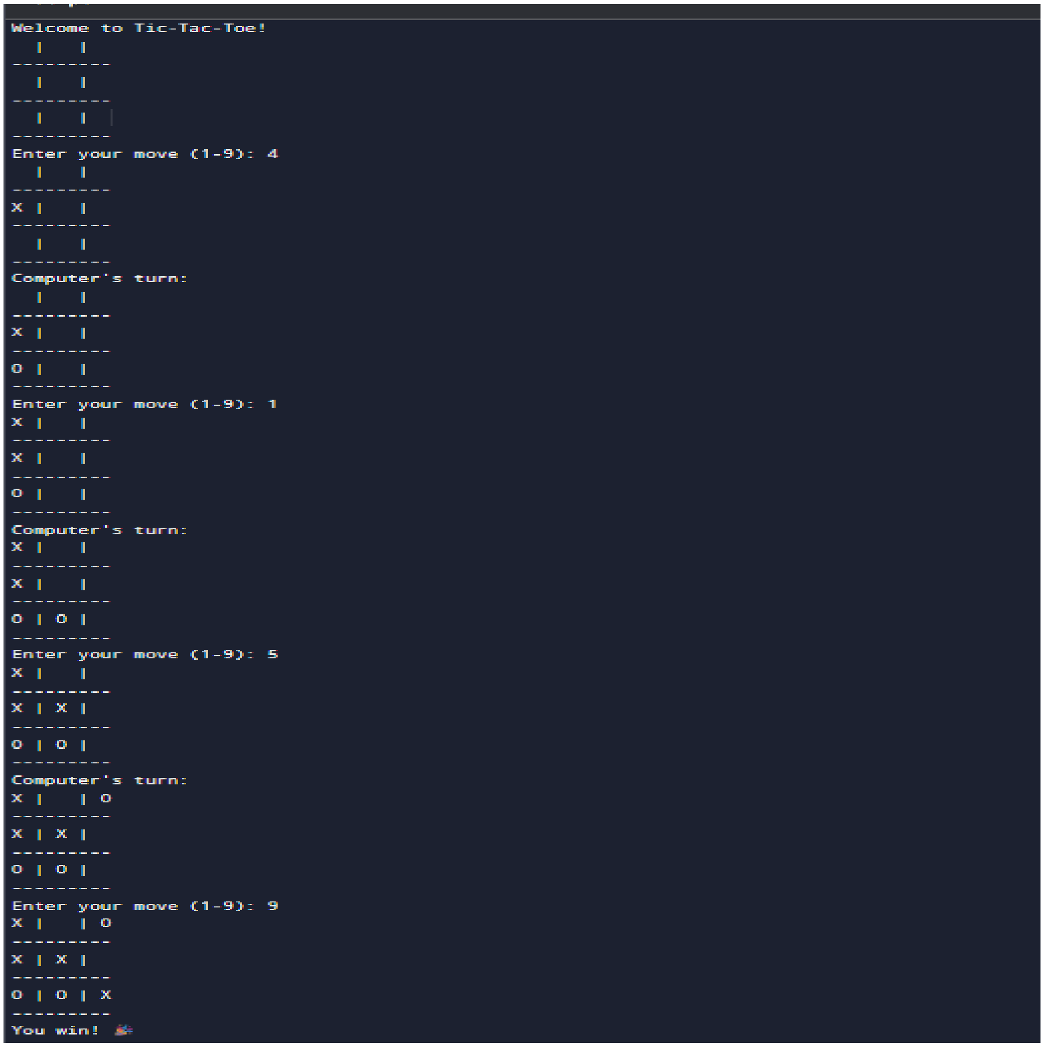
Date / /
Page

```
    return True

~~else~~ OR
(board [0][2] == board[1][1] && board[1][1]
        == board [2][0])
return True

return False
```

Code:

```
import random
board = [' ' for _ in range(9)]

def print_board():
    print()
    for i in range(3):
        print(" " + " | ".join(board[i*3:(i+1)*3]))
        if i < 2:
            print("---+---+---")
    print()
def check_winner(player):
    win_conditions = [
        [0,1,2], [3,4,5], [6,7,8],
        [0,3,6], [1,4,7], [2,5,8],
        [0,4,8], [2,4,6]
    ]
    for cond in win_conditions:
        if all(board[i] == player for i in cond):
            return True
    return False
def is_full():
    return all(cell != ' ' for cell in board)
def player_move():
    while True:
        try:
            move = int(input("Enter your move (1-9): ")) - 1
            if move < 0 or move >= 9:
                print("Invalid move. Choose between 1-9.")
            elif board[move] != ' ':
                print("That spot is taken.")
            else:
                board[move] = 'X'
                break
        except ValueError:
            print("Please enter a valid number.")
def ai_move():
    empty_spots = [i for i, val in enumerate(board) if val == ' ']
    move = random.choice(empty_spots)
    board[move] = 'O'
    print(f"System placed 'O' in position {move+1}")
def play_game():
    print("Welcome to Tic Tac Toe!")
    print_board()
    while True:
        player_move()
        print_board()
        if check_winner('X'):
            print("Congratulations! You win!")
            break
        if is_full():
            print("It's a tie!")
            break
        ai_move()
        print_board()
        if check_winner('O'):
            print("System wins. Better luck next time!")
            break
        if is_full():
            print("It's a tie!")
            break
if _name___== "_main_":
    play_game()
```

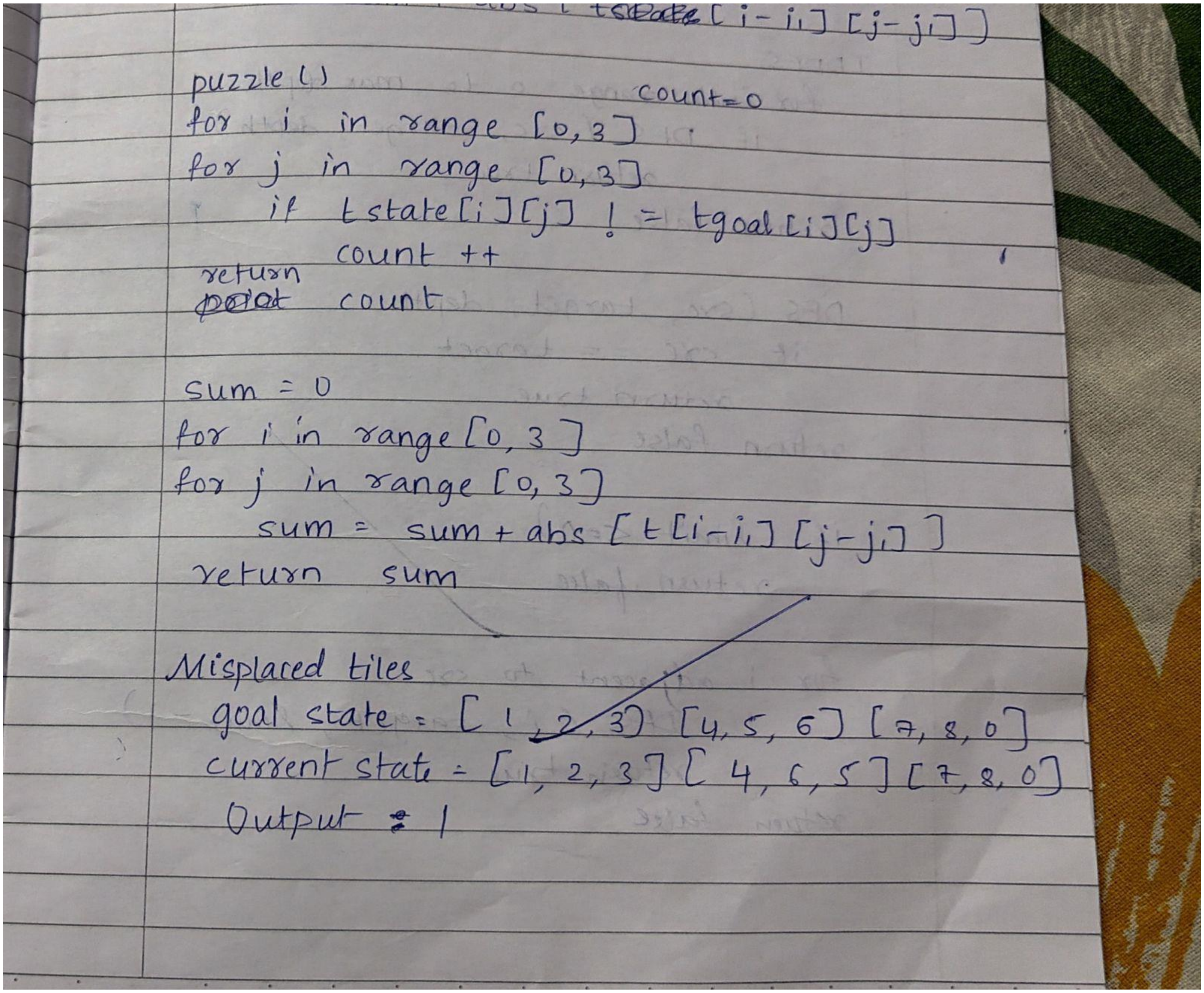
ScreenShots:



```
Welcome to Tic-Tac-Toe!
  |   |
---------
  |   |
---------
  |   |
---------
Enter your move (1-9): 4
  |   |
---------
X |   |
---------
  |   |
---------
Computer's turn:
  |   |
---------
X |   |
---------
O |   |
---------
Enter your move (1-9): 1
X |   |
---------
X |   |
---------
O |   |
---------
Computer's turn:
X |   |
---------
X |   |
---------
O | O |
---------
Enter your move (1-9): 5
X |   |
---------
X | X |
---------
O | O |
---------
Computer's turn:
X |   | O
---------
X | X |
---------
O | O |
---------
Enter your move (1-9): 9
X |   | O
---------
X | X |
---------
O | O | X
---------
You win! 🎉
```

<u>Program 2:</u>

Solve 8 puzzle problems.
Algorithm:



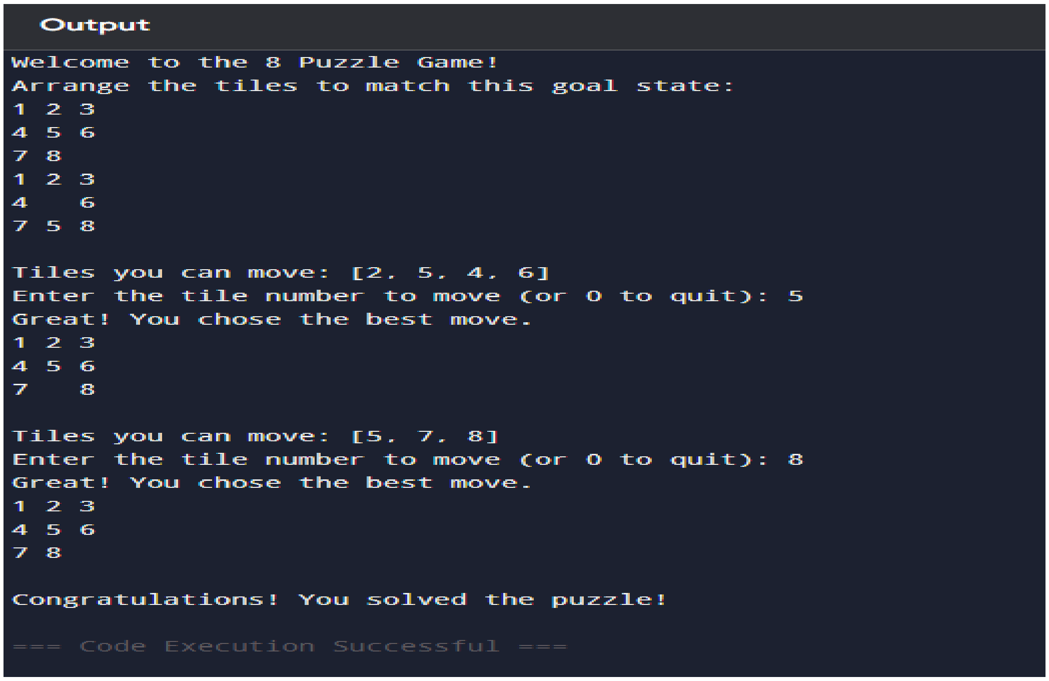
```
abs [ tstate [ i- i,] [j- j,] ]

puzzle ()                       count=0
for  i  in  range [0, 3]
for  j  in  range [0, 3]
        if  tstate[i][j] ! = tgoal [i][j]
            count ++
return
~~print~~   count

sum = 0
for  i  in  range [0, 3]
for  j  in  range [0, 3]
        sum =  sum + abs [ t [i-i,] [j-j,] ]
return   sum
```

Misplaced tiles
goal state = [1, 2, 3] [4, 5, 6] [7, 8, 0]
current state = [1, 2, 3] [4, 6, 5] [7, 8, 0]
Output : 1

Code:

```
import copy
def print_board(board):
    for row in board:
        print(' '.join(str(x) if x != 0 else ' ' for x in row))
    print()
def find_zero(board):
    for i in range(3):
        for j in range(3):
            if board[i][j] == 0:
                return i, j
def is_solved(board):
    solved = [1,2,3,4,5,6,7,8,0]
    flat = [num for row in board for num in row]
    return flat == solved
def valid_moves(zero_pos):
    i, j = zero_pos
    moves = []
    if i > 0: moves.append((i-1, j))
    if i < 2: moves.append((i+1, j))
    if j > 0: moves.append((i, j-1))
    if j < 2: moves.append((i, j+1))
    return moves
def correct_tiles_count(board):
    """Count how many tiles are in their correct position."""
    count = 0
    goal = [1,2,3,4,5,6,7,8,0]
    flat = [num for row in board for num in row]
    for i in range(9):
        if flat[i] != 0 and flat[i] == goal[i]:
            count += 1
    return count
def get_user_move(board):
    zero_pos = find_zero(board)
    moves = valid_moves(zero_pos)
    movable_tiles = [board[i][j] for (i,j) in moves]
    print(f"Tiles you can move: {movable_tiles}")
    while True:
        try:
            move = int(input("Enter the tile number to move (or 0 to quit): "))
            if move == 0:
                return None
            if move in movable_tiles:
                return move
            else:
                print("Invalid tile. Please choose a tile adjacent to the empty space.")
        except ValueError:
            print("Please enter a valid number.")
def evaluate_move(board, tile):
    """Compare user move to all possible moves and tell if it's best/worst."""
    zero_pos = find_zero(board)
    moves = valid_moves(zero_pos)
    movable_tiles = [board[i][j] for (i,j) in moves]
    scores = {}
    for t in movable_tiles:
        temp_board = copy.deepcopy(board)
        make_move(temp_board, t)
        scores[t] = correct_tiles_count(temp_board)
    user_score = scores[tile]
    best_score = max(scores.values())
    worst_score = min(scores.values())
    if user_score == best_score and user_score == worst_score:
        return "Your move is the only possible move."
    elif user_score == best_score:
        return "Great! You chose the best move."
    elif user_score == worst_score:
        return "Oops! You chose the worst move."
    else:
        return "Your move is neither the best nor the worst."
def make_move(board, tile):
    zero_i, zero_j = find_zero(board)
    for i, j in valid_moves((zero_i, zero_j)):
        if board[i][j] == tile:
            board[zero_i][zero_j], board[i][j] = board[i][j], board[zero_i][zero_j]
            return
def main():
    board = [
        [1, 2, 3],
        [4, 0, 6],
        [7, 5, 8]
    ]
    print("Welcome to the 8 Puzzle Game!")
    print("Arrange the tiles to match this goal state:")
    print("1 2 3\n4 5 6\n7 8  ")
    while True:
        print_board(board)
        if is_solved(board):
            print("Congratulations! You solved the puzzle!")
            break
        move = get_user_move(board)
        if move is None:
            print("Game exited. Goodbye!")
            break
        feedback = evaluate_move(board, move)
        print(feedback)
        make_move(board, move)
if _name___== "_main_":
    main()
```

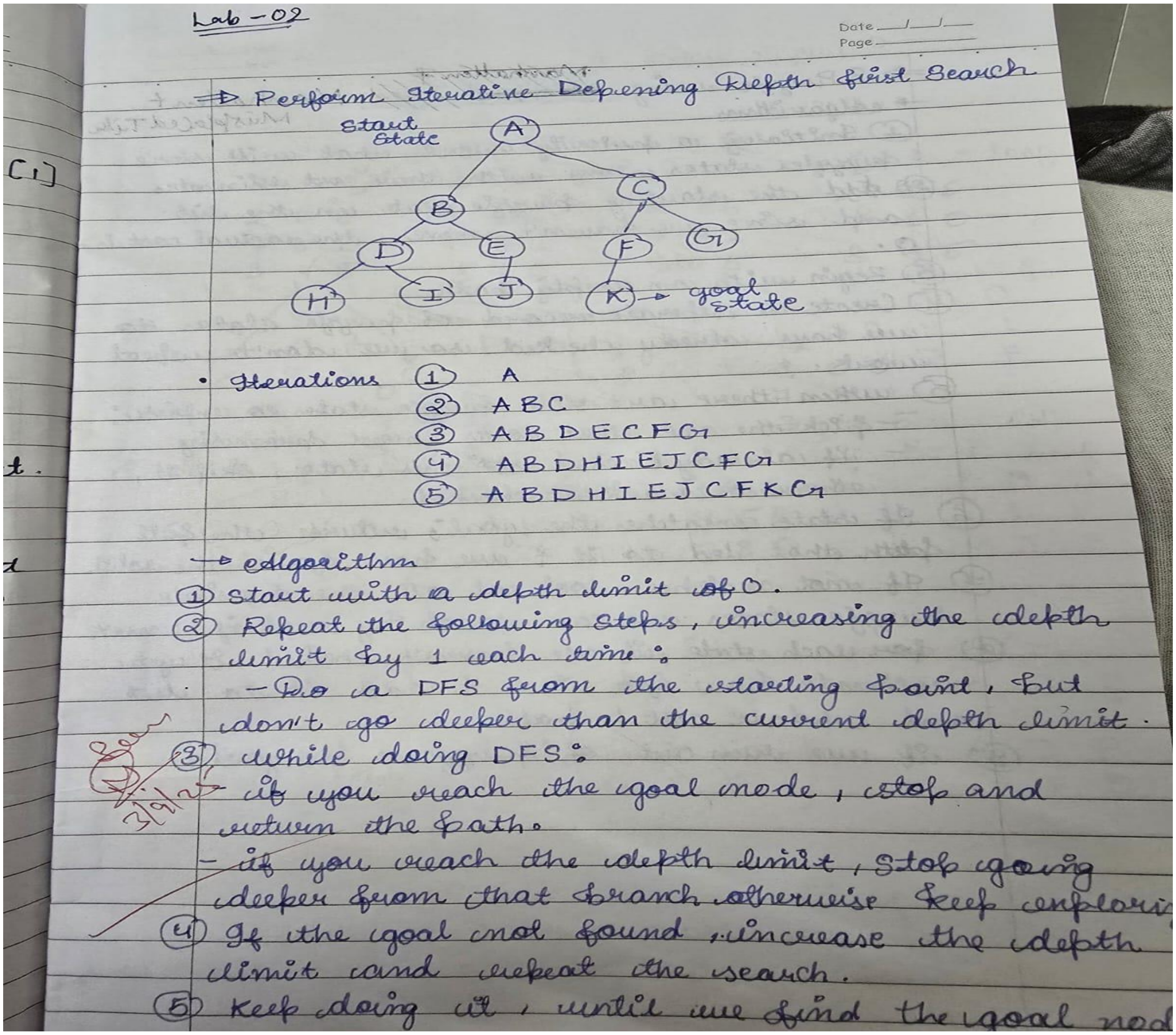
ScreenShot:



```
Welcome to the 8 Puzzle Game!
Arrange the tiles to match this goal state:
1 2 3
4 5 6
7 8
1 2 3
4   6
7 5 8

Tiles you can move: [2, 5, 4, 6]
Enter the tile number to move (or 0 to quit): 5
Great! You chose the best move.
1 2 3
4 5 6
7   8

Tiles you can move: [5, 7, 8]
Enter the tile number to move (or 0 to quit): 8
Great! You chose the best move.
1 2 3
4 5 6
7 8

Congratulations! You solved the puzzle!

=== Code Execution Successful ===
```

<u>Program 3:</u>

Implement Iterative deepening search algorithm.
Algorithm:


Start State
A
B
C
D
E
F
G
H
I
J
K
goal state

Lab - 02

Date __/__/__
Page ____

⇒ Perform Iterative Deepening Depth first Search

• Iterations ① A
② A B C
③ A B D E C F G
④ A B D H I E J C F G
⑤ A B D H I E J C F K G

→ Algorithm

① Start with a depth limit of 0.

② Repeat the following steps, increasing the depth limit by 1 each time:
– Do a DFS from the starting point, but don't go deeper than the current depth limit.

③ While doing DFS:
– if you reach the goal node, stop and return the path.
– if you reach the depth limit, stop going deeper from that branch otherwise keep explori

④ If the goal not found, increase the depth limit and repeat the search.

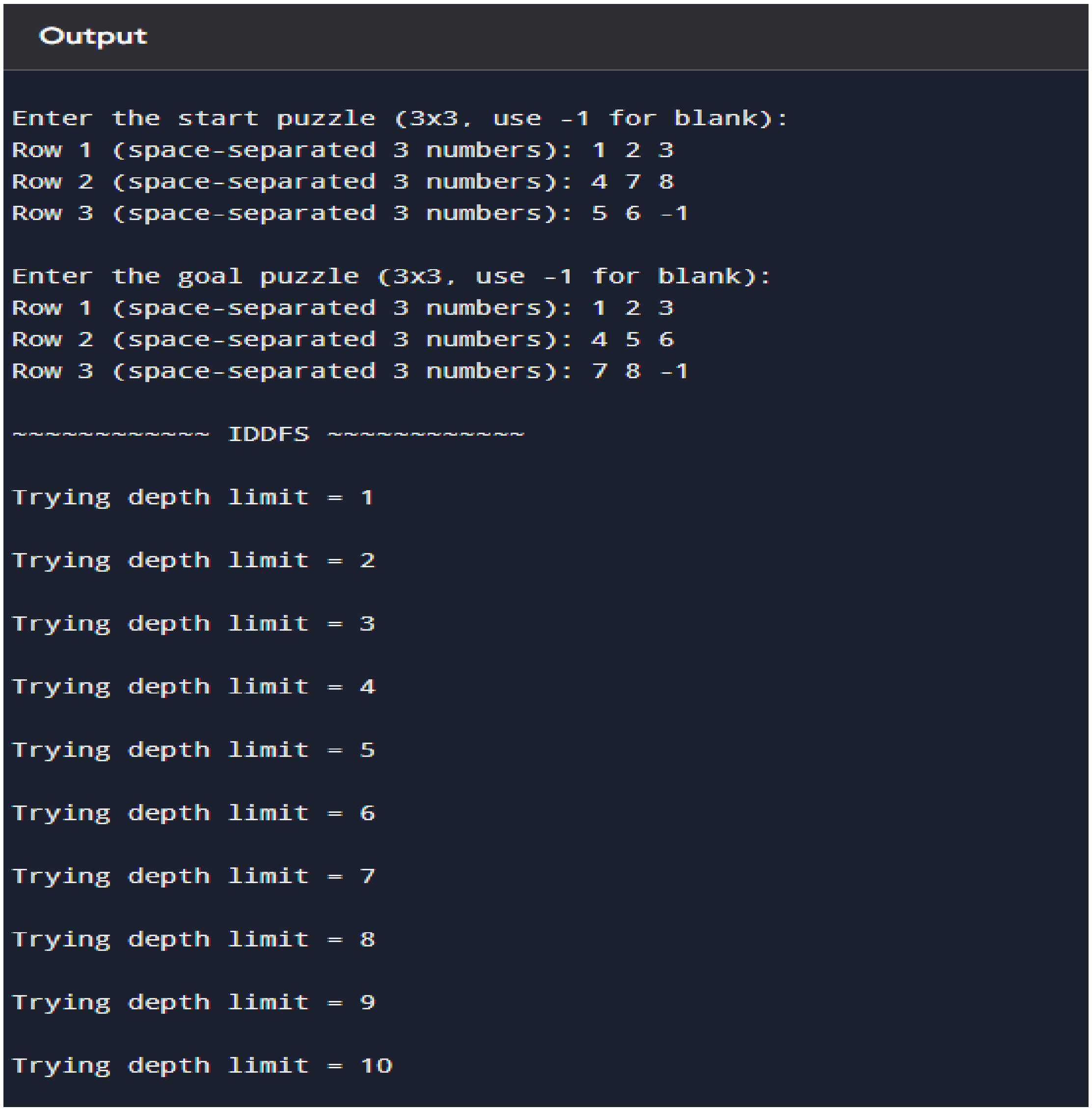
⑤ Keep doing it, until we find the goal nod

Seen 3/9/25

Code:

```
import copy
def get_puzzle(name):
    print(f"\nEnter the {name} puzzle (3x3, use -1 for blank):")
    puzzle = []
    for i in range(3):
        row = list(map(int, input(f"Row {i+1} (space-separated 3 numbers): ").split()))
        puzzle.append(row)
    return puzzle
def move(temp, movement):
    for i in range(3):
        for j in range(3):
            if temp[i][j] == -1:
                if movement == "up" and i > 0:
                    temp[i][j], temp[i-1][j] = temp[i-1][j], temp[i][j]
                elif movement == "down" and i < 2:
                    temp[i][j], temp[i+1][j] = temp[i+1][j], temp[i][j]
                elif movement == "left" and j > 0:
                    temp[i][j], temp[i][j-1] = temp[i][j-1], temp[i][j]
                elif movement == "right" and j < 2:
                    temp[i][j], temp[i][j+1] = temp[i][j+1], temp[i][j]
                return temp
    return temp
def dls(puzzle, depth, limit, last_move, goal):
    if puzzle == goal:
        return True, [puzzle], []
    if depth >= limit:
        return False, [], []
    for move_dir, opposite in [("up","down"), ("left","right"), ("down","up"), ("right","left")]:
        if last_move == opposite:  # avoid direct backtracking
            continue
        temp = copy.deepcopy(puzzle)
        new_state = move(temp, move_dir)
        if new_state != puzzle:  # valid move
            found, path, moves = dls(new_state, depth+1, limit, move_dir, goal)
            if found:
                return True, [puzzle] + path, [move_dir] + moves
    return False, [], []
def ids(start, goal):
    for limit in range(1, 50):  # reasonable max depth
        print(f"\nTrying depth limit = {limit}")
        found, path, moves = dls(start, 0, limit, None, goal)
        if found:
            print("Solution found!")
            for step in path:
                print(step)
            print("Moves:", moves)
            print("Path cost =", len(path)-1)
            return
    print(" Solution not found within depth limit.")
start_puzzle = get_puzzle("start")
goal_puzzle = get_puzzle("goal")
print("\n~~~~~~~~~~~~ IDDFS ~~~~~~~~~~~~")
ids(start_puzzle, goal_puzzle)
```

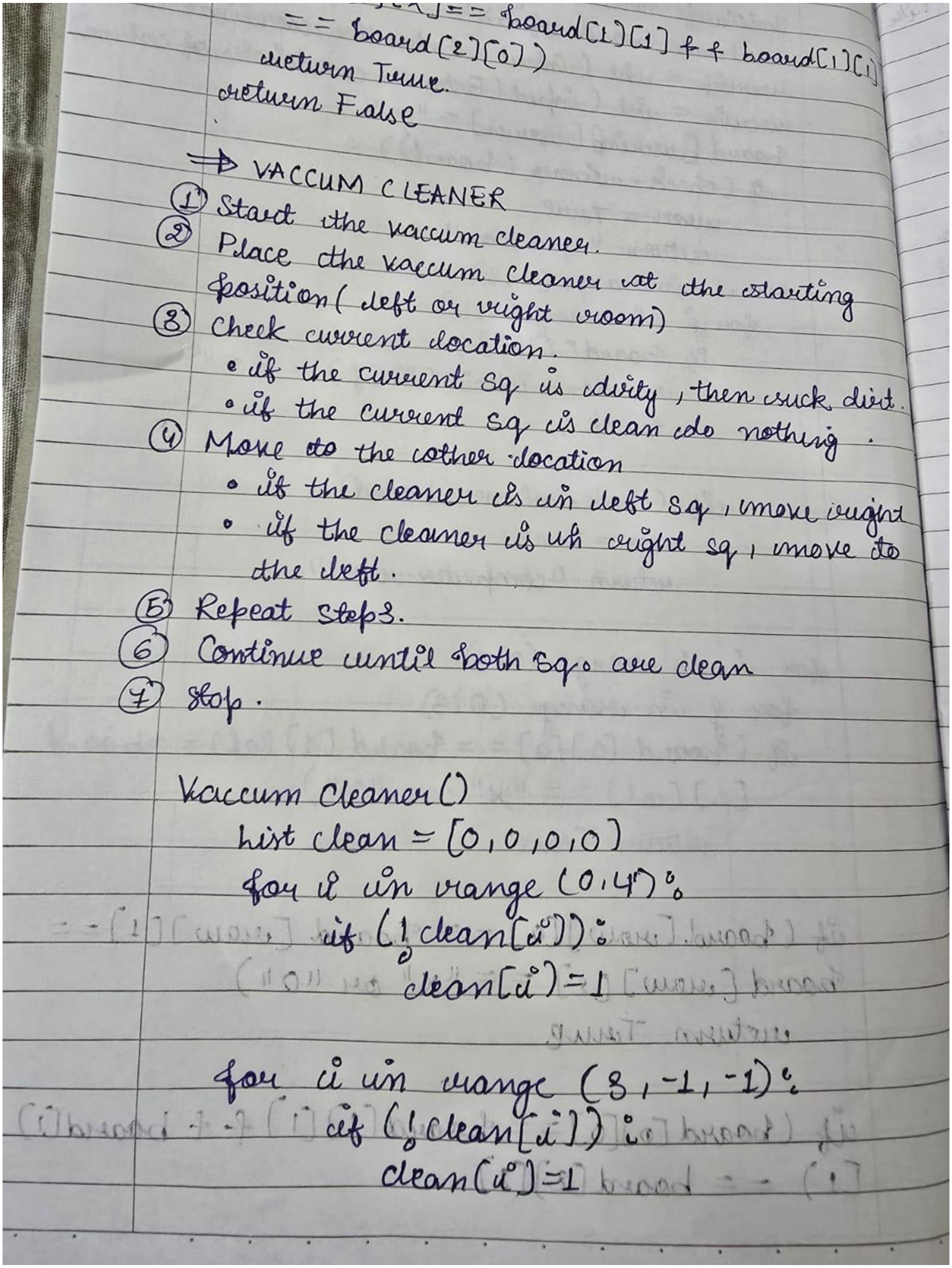
ScreenShot:



```
Enter the start puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 7 8
Row 3 (space-separated 3 numbers): 5 6 -1

Enter the goal puzzle (3x3, use -1 for blank):
Row 1 (space-separated 3 numbers): 1 2 3
Row 2 (space-separated 3 numbers): 4 5 6
Row 3 (space-separated 3 numbers): 7 8 -1

~~~~~~~~~~~~ IDDFS ~~~~~~~~~~~~

Trying depth limit = 1

Trying depth limit = 2

Trying depth limit = 3

Trying depth limit = 4

Trying depth limit = 5

Trying depth limit = 6

Trying depth limit = 7

Trying depth limit = 8

Trying depth limit = 9

Trying depth limit = 10
```

<u>Program 4:</u>

Implement a vacuum cleaner agent.
Algorithm:

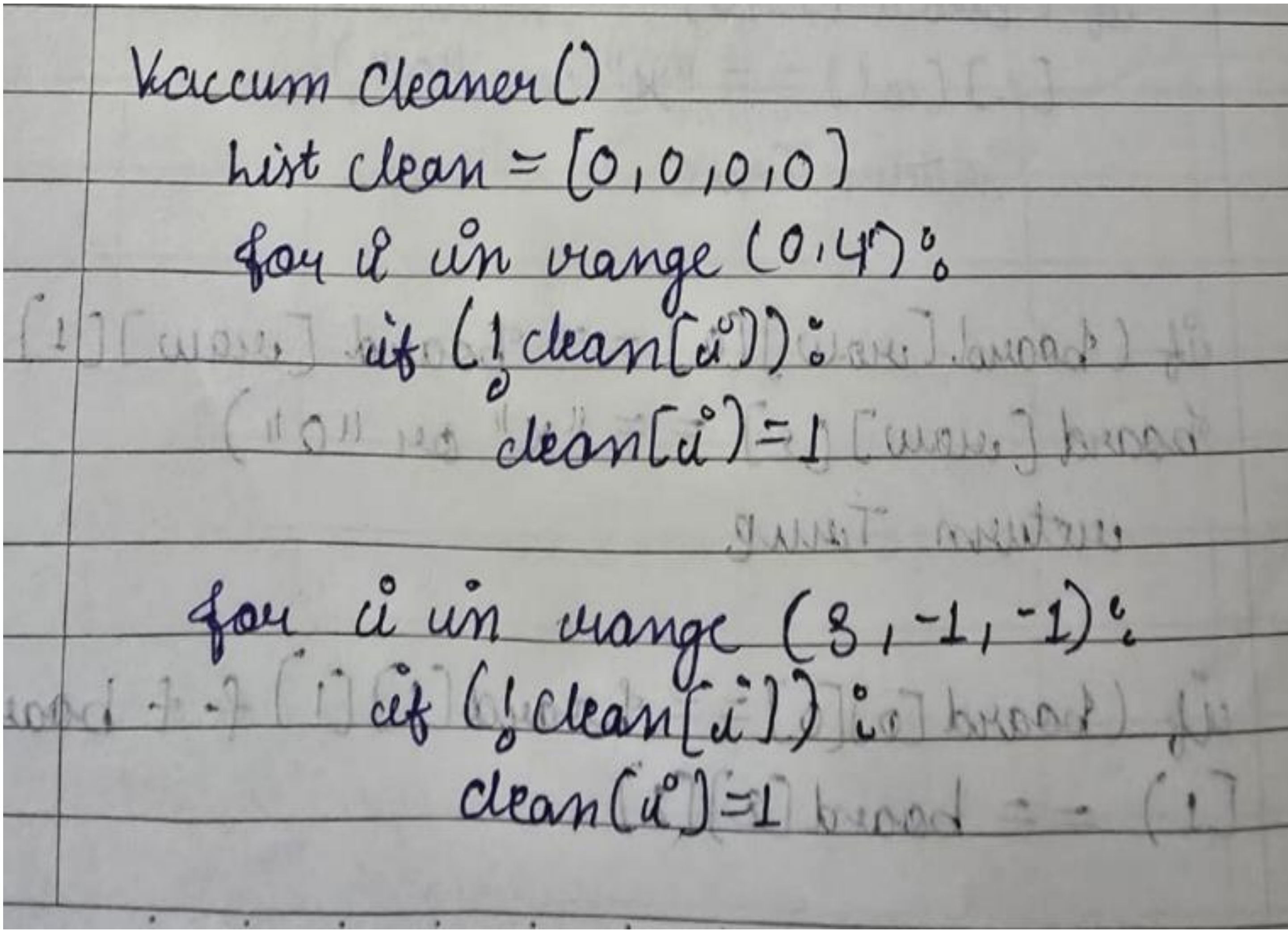


```
        ...]== board[1][1] && board[1][1]
     == board[2][0])
   return True.
return False
```

⇒ VACCUM CLEANER

1. Start the vaccum cleaner.
2. Place the vaccum cleaner at the starting position(left or right room)
3. Check current location.
    - if the current sq is dirty, then suck dirt.
    - if the current sq is clean do nothing.
4. Move to the other location
    - if the cleaner is in left sq, move right
    - if the cleaner is in right sq, move to the left.
5. Repeat step3.
6. Continue until both sq. are clean
7. Stop.

```
Vaccum Cleaner()
   list clean = [0,0,0,0]
   for i in range (0,4):
       if (!clean[i]):
           clean[i] = 1

   for i in range (3,-1,-1):
       if (!clean[i]):
           clean(i] = 1
```

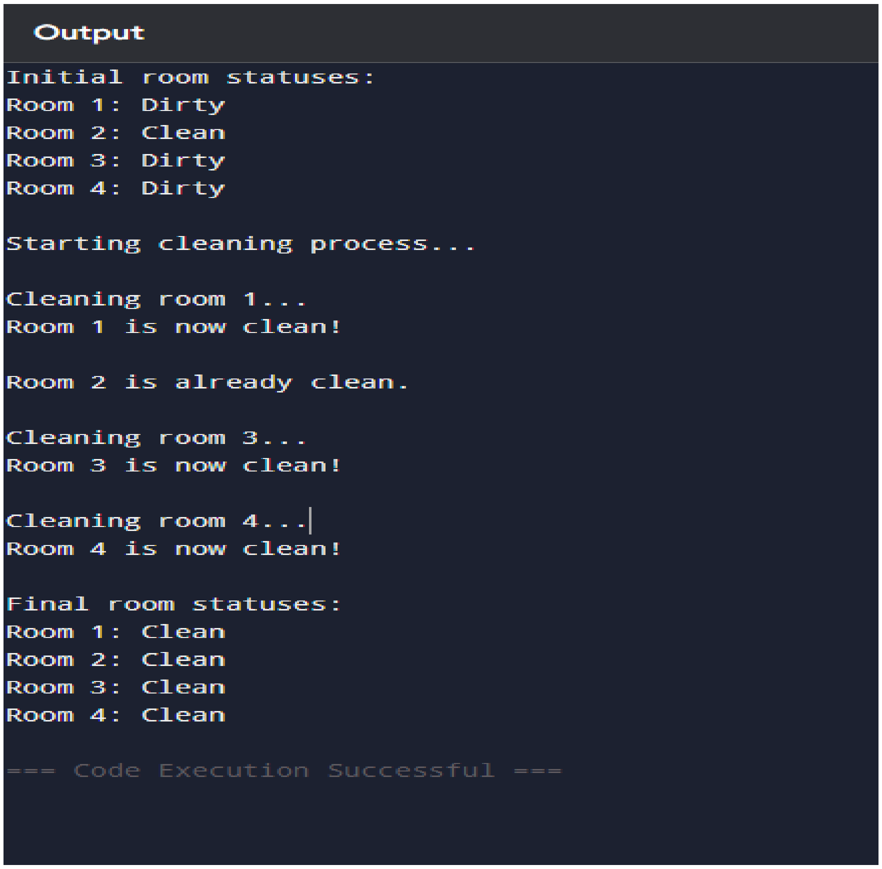


```
Vaccum Cleaner()
    list clean = [0,0,0,0]
    for i in range (0,4):
        if (!clean[i]):
            clean[i)=1

    for i in range (3,-1,-1):
        if (!clean[i]):
            clean(i]=1
```

Code:

```
def show_rooms_status(rooms):
    for room_number, status in rooms.items():
        print(f"Room {room_number}: {'Clean' if status else 'Dirty'}")
def clean_room(rooms, room_number):
    if rooms[room_number]:
        print(f"Room {room_number} is already clean.")
    else:
        print(f"Cleaning room {room_number}...")
        rooms[room_number] = True
        print(f"Room {room_number} is now clean!")
def clean_all_rooms(rooms):
    print("Initial room statuses:")
    show_rooms_status(rooms)
    print("\nStarting cleaning process...\n")
    for room_number in rooms:
        clean_room(rooms, room_number)
        print()
    print("Final room statuses:")
    show_rooms_status(rooms)
if _name___== "_main_":
    rooms = {
        1: False,
        2: True,
        3: False,
        4: False
    }
    clean_all_rooms(rooms)
```

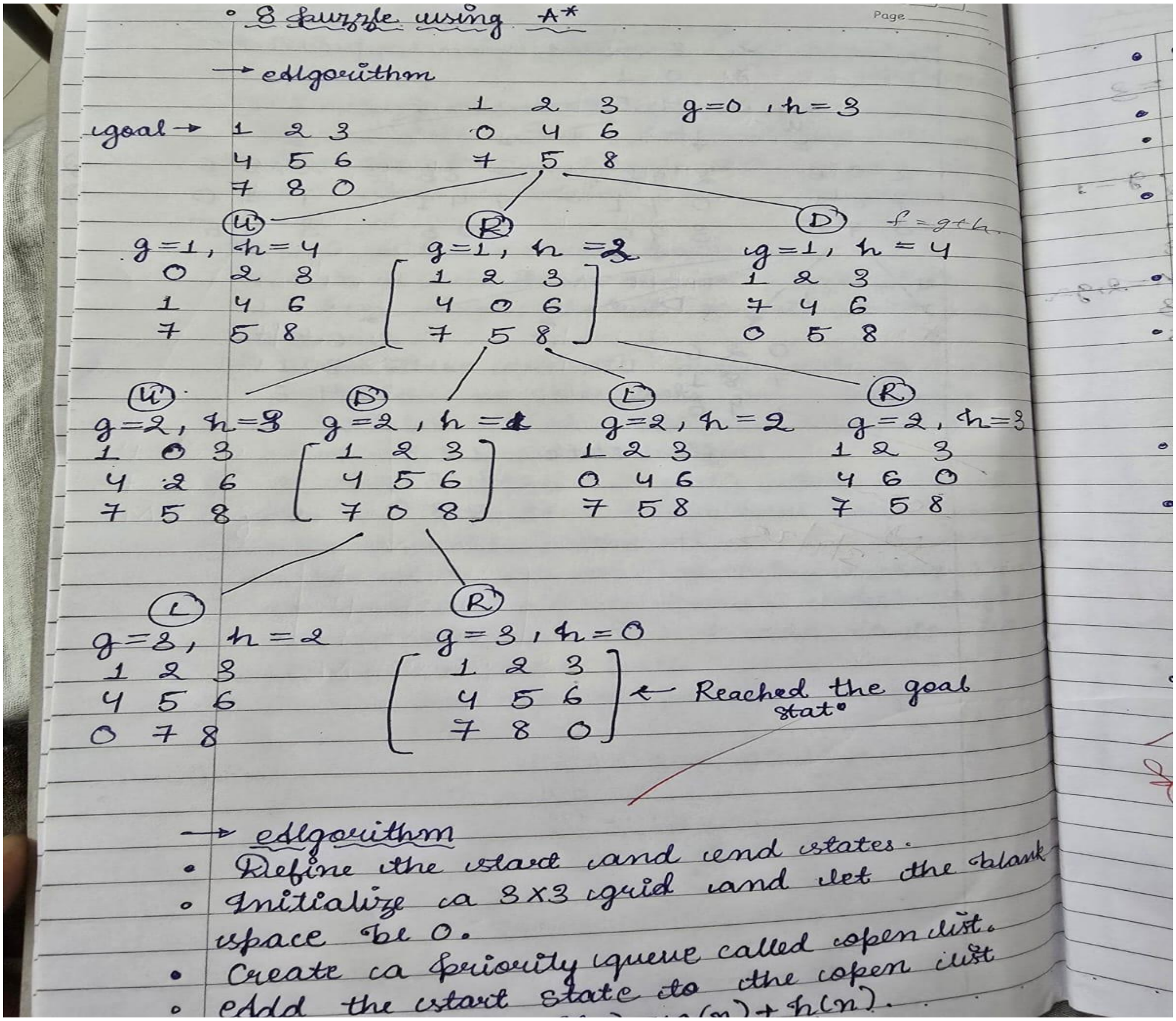
ScreenShot:



```
Initial room statuses:
Room 1: Dirty
Room 2: Clean
Room 3: Dirty
Room 4: Dirty

Starting cleaning process...

Cleaning room 1...
Room 1 is now clean!

Room 2 is already clean.

Cleaning room 3...
Room 3 is now clean!

Cleaning room 4...
Room 4 is now clean!

Final room statuses:
Room 1: Clean
Room 2: Clean
Room 3: Clean
Room 4: Clean

=== Code Execution Successful ===
```

<u>Program 5:</u>

Implement A* search algorithm.
Algorithm:



8 puzzle using A*

→ Algorithm

goal →
1 2 3
4 5 6
7 8 0

g=0, h=3
1 2 3
0 4 6
7 5 8

f = g+h

U
g=1, h=4
0 2 8
1 4 6
7 5 8

R
g=1, h=2
1 2 3
4 0 6
7 5 8

D
g=1, h=4
1 2 3
7 4 6
0 5 8

U
g=2, h=3
1 0 3
4 2 6
7 5 8

D
g=2, h=1
1 2 3
4 5 6
7 0 8

L
g=2, h=2
1 2 3
0 4 6
7 5 8

R
g=2, h=3
1 2 3
4 6 0
7 5 8

L
g=3, h=2
1 2 3
4 5 6
0 7 8

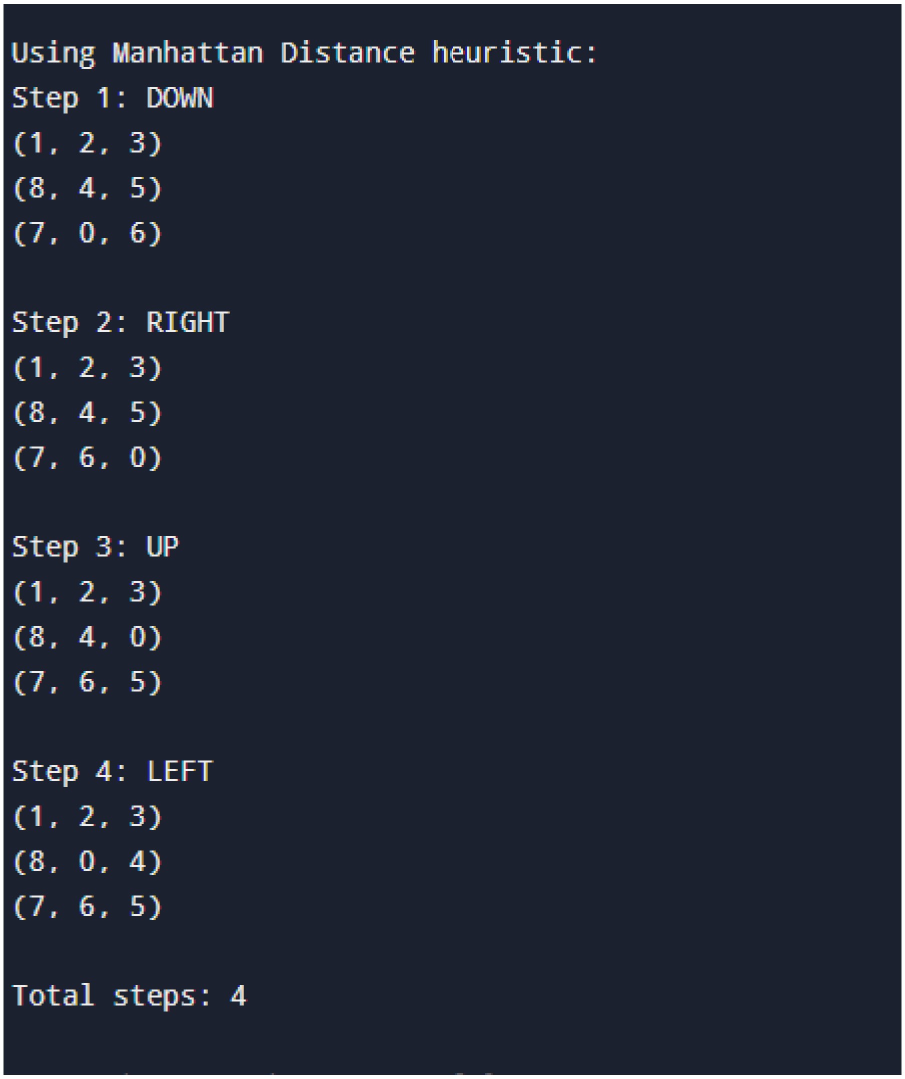
R
g=3, h=0
1 2 3
4 5 6
7 8 0
← Reached the goal state

→ Algorithm
- Define the start and end states.
- Initialize a 3x3 grid and let the blank space be 0.
- Create a priority queue called open list.
- Add the start state to the open list
g(n)+h(n).

Code:

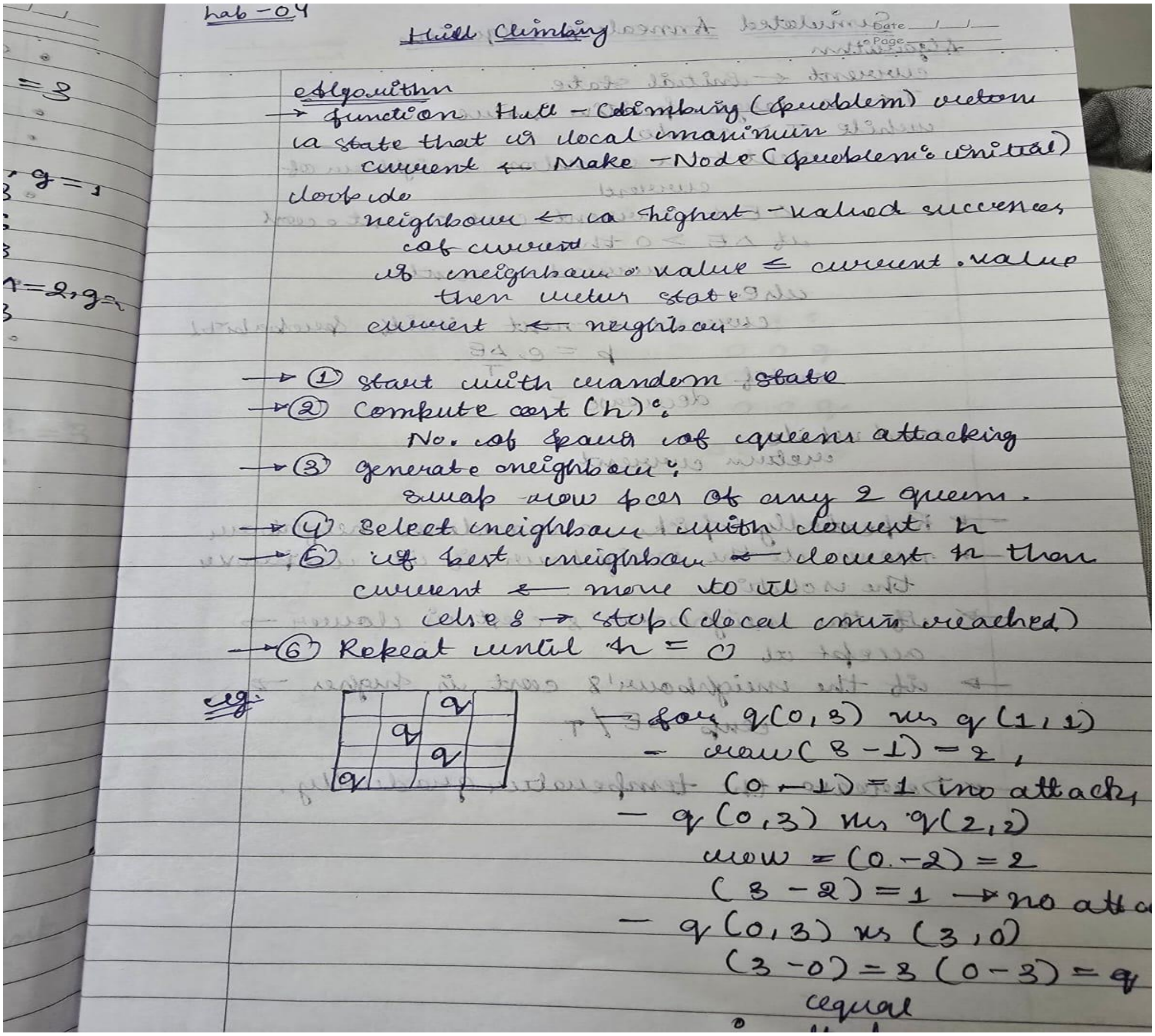
```
from heapq import heappush, heappop
goal_state = [
    [1, 2, 3],
    [8, 0, 4],
    [7, 6, 5]
]
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
direction_names = ["UP", "DOWN", "LEFT", "RIGHT"]
def misplaced_tiles(state):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                count += 1
    return count
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_x, goal_y = divmod(tile - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance
def get_neighbors_with_actions(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break
    for (dx, dy), action in zip(directions, direction_names):
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append((new_state, action))
    return neighbors
def state_to_tuple(state):
    return tuple(tuple(row) for row in state)
def reconstruct_path(came_from, current):
    actions = []
    states = []
    while current in came_from:
        prev_state, action = came_from[current]
        actions.append(action)
        states.append(current)
        current = prev_state
    states.append(current)
    actions.reverse()
    states.reverse()
    return actions, states
def a_star_search_with_steps(initial_state, heuristic_func):
    open_list = []
    closed_set = set()
    g_score = {state_to_tuple(initial_state): 0}
    f_score = {state_to_tuple(initial_state): heuristic_func(initial_state)}
    came_from = {}
    heappush(open_list, (f_score[state_to_tuple(initial_state)], initial_state))
    while open_list:
        _, current_state = heappop(open_list)
        current_t = state_to_tuple(current_state)
        if current_state == goal_state:
            return reconstruct_path(came_from, current_t)
        closed_set.add(current_t)
        for neighbor, action in get_neighbors_with_actions(current_state):
            neighbor_t = state_to_tuple(neighbor)
            if neighbor_t in closed_set:
                continue
            tentative_g = g_score[current_t] + 1
            if neighbor_t not in g_score or tentative_g < g_score[neighbor_t]:
                came_from[neighbor_t] = (current_t, action)
                g_score[neighbor_t] = tentative_g
                f_score[neighbor_t] = tentative_g + heuristic_func(neighbor)
                heappush(open_list, (f_score[neighbor_t], neighbor))
    return None, None
def print_path(actions, states):
    for i, (action, state) in enumerate(zip(actions, states[1:]), 1):
        print(f"Step {i}: {action}")
        for row in state:
            print(row)
        print()
initial_state = [
     [1, 2, 3],
    [8, 0, 5],
    [7, 4, 6]
]
print("Using Misplaced Tiles heuristic:")
actions, states = a_star_search_with_steps(initial_state, misplaced_tiles)
if actions:
    print_path(actions, states)
    print("Total steps:", len(actions))
else:
    print("No solution found.")
print("\nUsing Manhattan Distance heuristic:")
actions, states = a_star_search_with_steps(initial_state, manhattan_distance)
if actions:
    print_path(actions, states)
    print("Total steps:", len(actions))
else:
    print("No solution found.")
```

ScreenShot:



```
Using Manhattan Distance heuristic:
Step 1: DOWN
(1, 2, 3)
(8, 4, 5)
(7, 0, 6)

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

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

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

Total steps: 4
```

b. Implement Hill Climbing Algorithm
Algorithm:



Lab - 04

Hill Climbing

Algorithm

```
→ function Hill-Climbing(problem) return
a state that is local maximum
    current ← Make-Node(problem.initial)
loop do
    neighbour ← a highest-valued successor
        of current
    if neighbour.value ≤ current.value
        then return state
    current ← neighbour
```

→ ① start with random state

→ ② Compute cost (h):
No. of pairs of queens attacking

→ ③ generate neighbours:
swap row pos of any 2 queens.

→ ④ Select neighbour with lowest h

→ ⑤ if best neighbour ← lowest h than
current ← move to it
else: → stop (local min reached)

→ ⑥ Repeat until h = 0

eg:

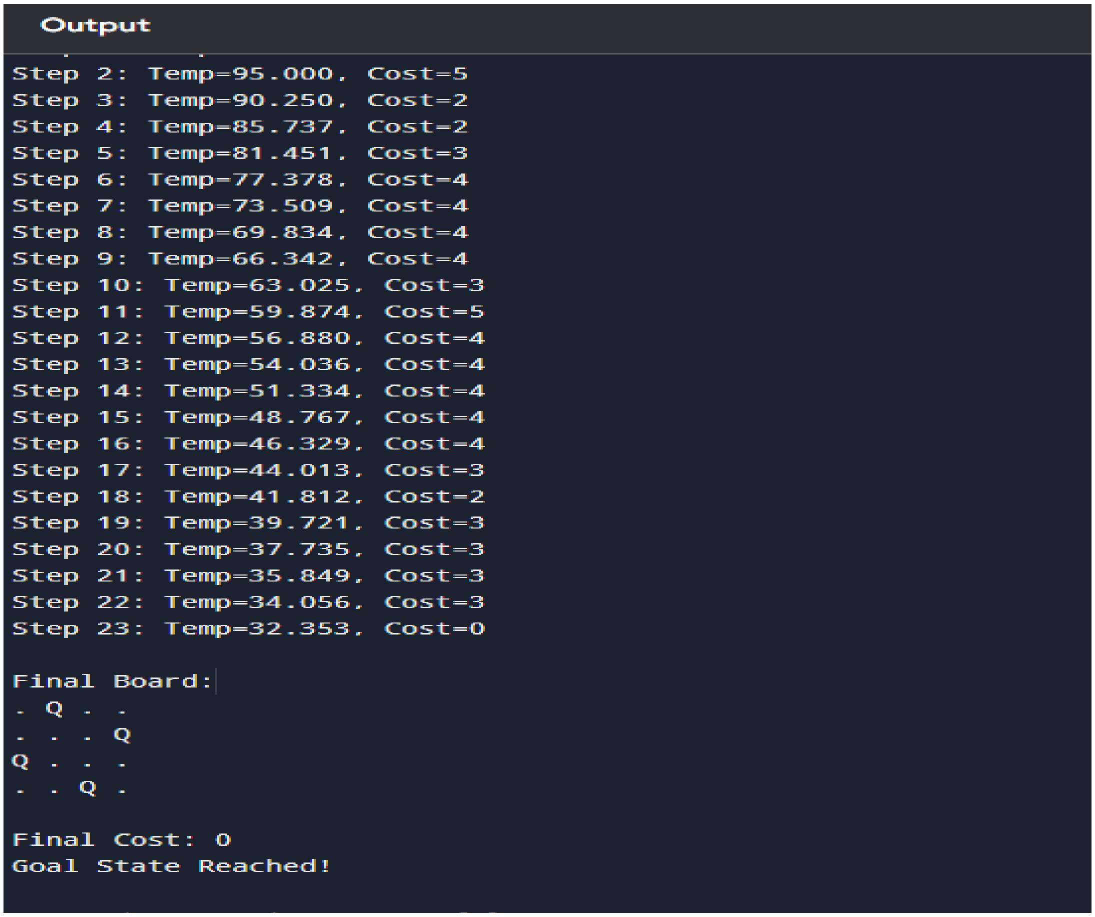
		q	
	q		
		q	
q			

→ for q(0,3) vs q(1,1)
- row (3-1) = 2,
(0-1) = 1 no attack,
- q(0,3) vs q(2,2)
row = (0-2) = 2
(3-2) = 1 → no atta
- q(0,3) vs (3,0)
(3-0) = 3 (0-3) = 3
equal

Code:

```
import random
import time
def generate_initial_state(n=4):
    return [random.randint(0, n - 1) for _ in range(n)]
def calculate_conflicts(state):
    conflicts = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:
                conflicts += 1
            if abs(state[i] - state[j]) == abs(i - j):
                conflicts += 1
    return conflicts
def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
        for row in range(n):
            if state[col] != row:
                neighbor = state.copy()
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors
def print_board(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 hill_climbing_with_steps(n=4, max_restarts=100):
    for restart in range(max_restarts):
        current = generate_initial_state(n)
        step = 0
        print(f"Restart #{restart+1}: Initial state (Conflicts = {calculate_conflicts(current)})")
        print_board(current)
        while True:
            current_conflicts = calculate_conflicts(current)
            if current_conflicts == 0:
                print(f"Solution found in {step} steps!")
                return current
            neighbors = get_neighbors(current)
            neighbor_conflicts = [calculate_conflicts(nbr) for nbr in neighbors]
            min_conflict = min(neighbor_conflicts)
            if min_conflict >= current_conflicts:
                print("Reached local minimum, restarting...\n")
                break
            best_neighbor = neighbors[neighbor_conflicts.index(min_conflict)]
            step += 1
            print(f"Step {step}: Conflicts = {min_conflict}")
            print_board(best_neighbor)
            current = best_neighbor
    return None
solution = hill_climbing_with_steps()
if solution:
    print("Final Solution:")
    print_board(solution)
else:
    print("No solution found.")
```

ScreenShot:



```
Step 2: Temp=95.000, Cost=5
Step 3: Temp=90.250, Cost=2
Step 4: Temp=85.737, Cost=2
Step 5: Temp=81.451, Cost=3
Step 6: Temp=77.378, Cost=4
Step 7: Temp=73.509, Cost=4
Step 8: Temp=69.834, Cost=4
Step 9: Temp=66.342, Cost=4
Step 10: Temp=63.025, Cost=3
Step 11: Temp=59.874, Cost=5
Step 12: Temp=56.880, Cost=4
Step 13: Temp=54.036, Cost=4
Step 14: Temp=51.334, Cost=4
Step 15: Temp=48.767, Cost=4
Step 16: Temp=46.329, Cost=4
Step 17: Temp=44.013, Cost=3
Step 18: Temp=41.812, Cost=2
Step 19: Temp=39.721, Cost=3
Step 20: Temp=37.735, Cost=3
Step 21: Temp=35.849, Cost=3
Step 22: Temp=34.056, Cost=3
Step 23: Temp=32.353, Cost=0

Final Board:
. Q . .
. . . Q
Q . . .
. . Q .

Final Cost: 0
Goal State Reached!
```

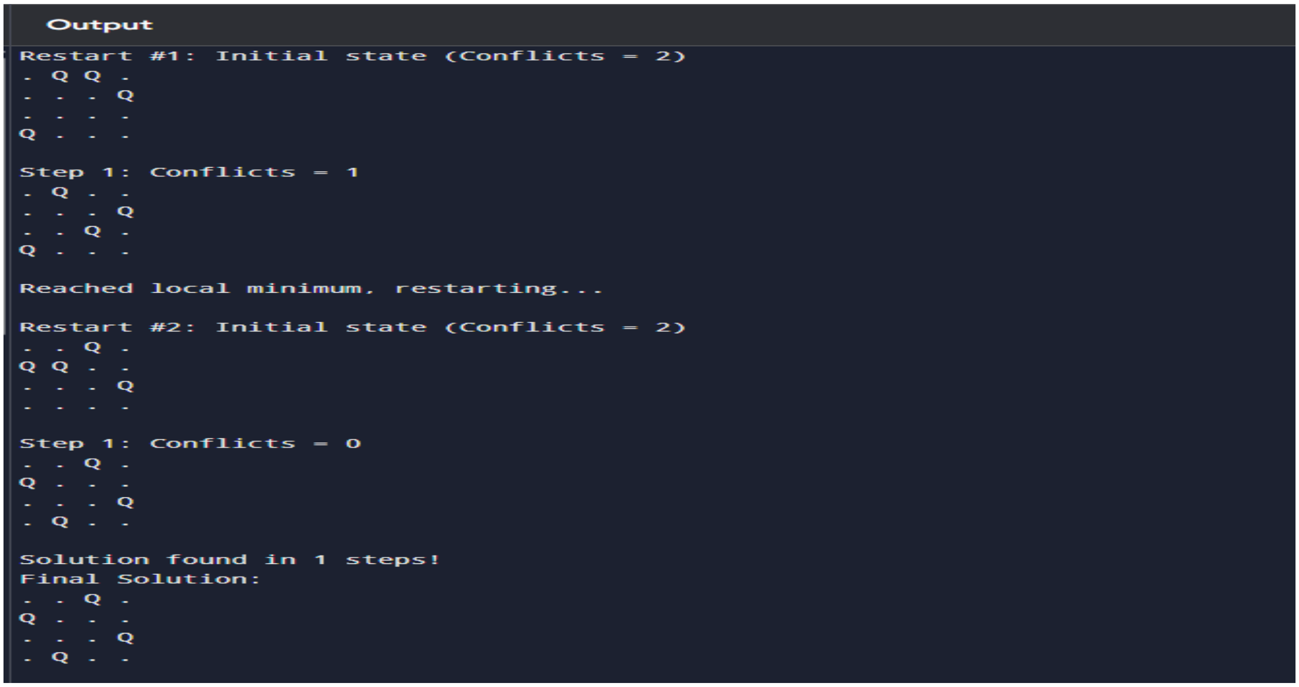
Program 6:

Write a program to implement Simulated Annealing Algorithm

Code:

```
import random
import math
def print_board(board):
    n = len(board)
    for i in range(n):
        row = ["Q" if board[i] == j else "." for j in range(n)]
        print(" ".join(row))
    print()
def calculate_cost(board):
    """Heuristic: number of pairs of queens attacking each other"""
    n = len(board)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                cost += 1
    return cost
def random_neighbor(board):
    """Generate a random neighboring board by moving one queen"""
    n = len(board)
    neighbor = list(board)
    row = random.randint(0, n - 1)
    col = random.randint(0, n - 1)
    neighbor[row] = col
    return neighbor
def simulated_annealing(n, initial_temp=100, cooling_rate=0.95, stopping_temp=1):
    current_board = [random.randint(0, n - 1) for _ in range(n)]
    current_cost = calculate_cost(current_board)
    temperature = initial_temp
    step = 1
    print("Initial Board:")
    print_board(current_board)
    print(f"Initial Cost: {current_cost}\n")
    while temperature > stopping_temp and current_cost > 0:
        neighbor = random_neighbor(current_board)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost
        if delta < 0 or random.random() < math.exp(-delta / temperature):
            current_board = neighbor
            current_cost = neighbor_cost
        print(f"Step {step}: Temp={temperature:.3f}, Cost={current_cost}")
        step += 1
        temperature *= cooling_rate
    print("\nFinal Board:")
    print_board(current_board)
    print(f"Final Cost: {current_cost}")
    if current_cost == 0:
        print("Goal State Reached!")
    else:
        print("Terminated before reaching goal.")
simulated_annealing(4)
```

ScreenShot:

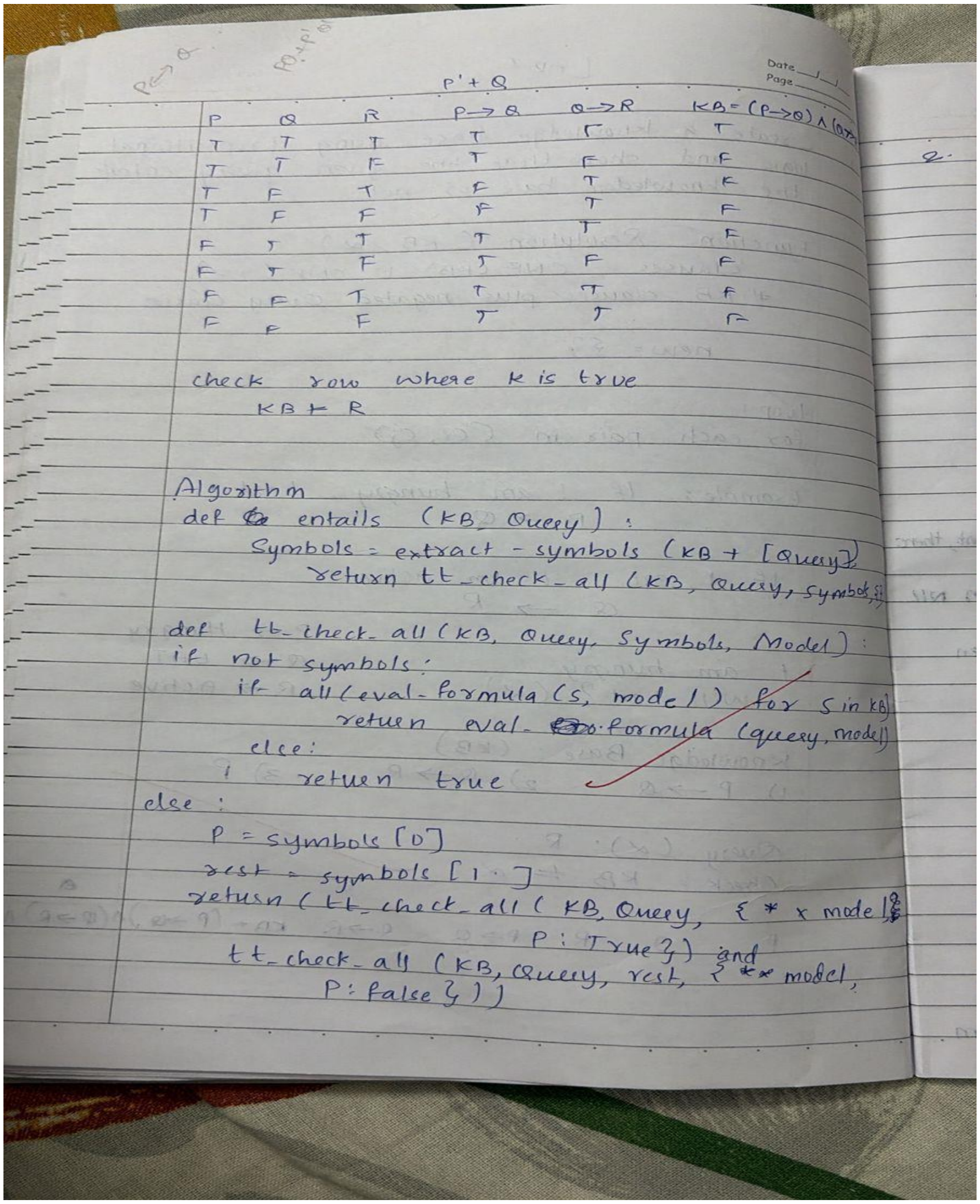


```
Restart #1: Initial state (Conflicts = 2)
. Q Q .
. . . Q
. . . .
Q . . .

Step 1: Conflicts = 1
. Q . .
. . . Q
. . Q .
Q . . .

Reached local minimum, restarting...

Restart #2: Initial state (Conflicts = 2)
. . Q .
Q Q . .
. . . Q
. . . .

Step 1: Conflicts = 0
. . Q .
Q . . .
. . . Q
. Q . .

Solution found in 1 steps!
Final Solution:
. . Q .
Q . . .
. . . Q
. Q . .
```

<u>Program 7:</u>

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

Algorithm:



P' + Q

P	Q	R	P→Q	Q→R	KB = (P→Q) ∧ (Q→
T	T	T	T	T	T
T	T	F	T	F	F
T	F	T	F	T	F
T	F	F	F	T	F
F	T	T	T	T	F
F	T	F	T	F	F
F	F	T	T	T	F
F	F	F	T	T	F

check row where k is true
KB ⊢ R

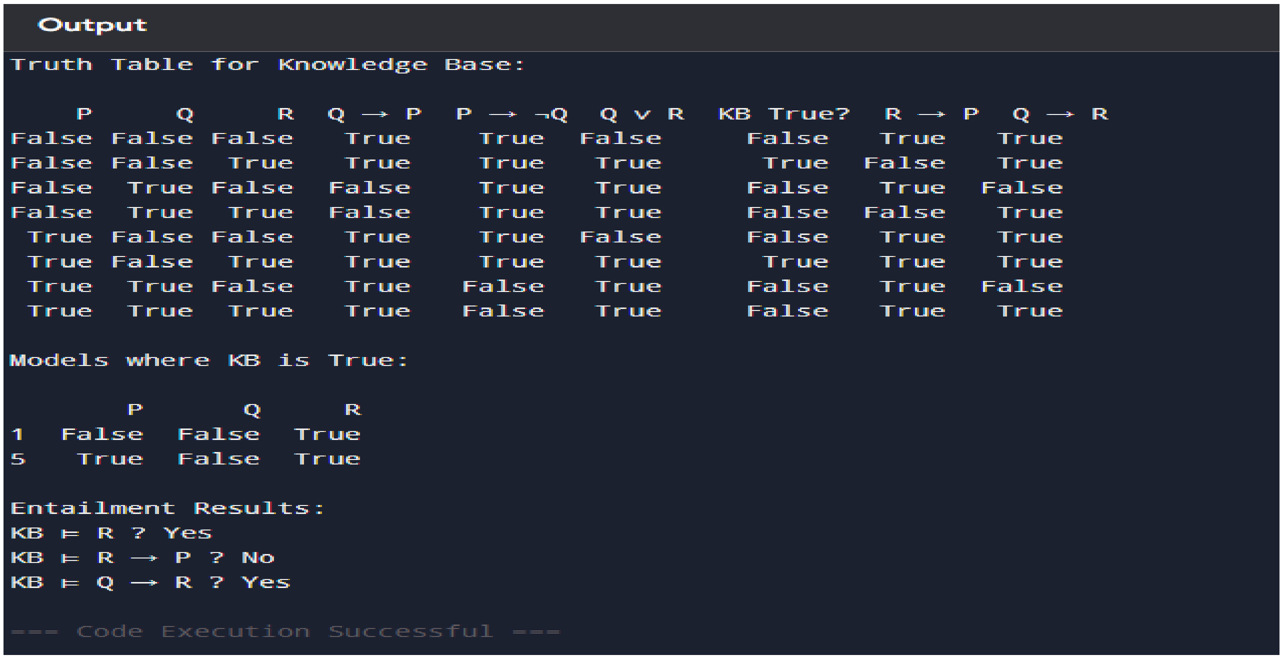
Algorithm

```
def entails (KB, Query) :
    Symbols = extract - symbols (KB + [Query])
      return tt_check_all (KB, Query, Symbols, {})

def tt_check_all (KB, Query, Symbols, Model) :
if not symbols :
    if all (eval_formula (s, model)) for s in KB)
        return eval_formula (query, model)
    else:
        return true
else :
    P = symbols [0]
    rest = symbols [1 : ]
  return (tt_check_all ( KB, Query, { * x model,
                          P : True }) and
      tt_check_all (KB, Query, rest, { ** model,
          P : false } ) )
```

Code:

```
import itertools
import pandas as pd
variables = ['P', 'Q', 'R']
combinations = list(itertools.product([False, True], repeat=3))
rows = []
for (P, Q, R) in combinations:
    s1 = (not Q) or P
    s2 = (not P) or (not Q)  # P → ¬Q
    s3 = Q or R              # Q ∨   R
    KB = s1 and s2 and s3
    entail_R = R
    entail_R_imp_P = (not R) or P
    entail_Q_imp_R = (not Q) or R
    rows.append({
        'P': P, 'Q': Q, 'R': R,
        'Q → P': s1,
        'P → ¬Q': s2,
        'Q ∨  R': s3,
        'KB True?': KB,
        'R': entail_R,
        'R → P': entail_R_imp_P,
        'Q → R': entail_Q_imp_R
    })
df = pd.DataFrame(rows)
print("Truth Table for Knowledge Base:\n")
print(df.to_string(index=False))
models_true = df[df['KB True?'] == True]
print("\nModels where KB is True:\n")
print(models_true[['P', 'Q', 'R']])
def entails(column):
    """Check if KB entails the given statement."""
    return all(models_true[column])
print("\nEntailment Results:")
print(f"KB ⊨ R ? {'Yes' if entails('R') else 'No'}")
print(f"KB ⊨ R → P ? {'Yes' if entails('R → P') else 'No'}")
print(f"KB ⊨ Q → R ? {'Yes' if entails('Q → R') else 'No'}")
```

ScreenShot:

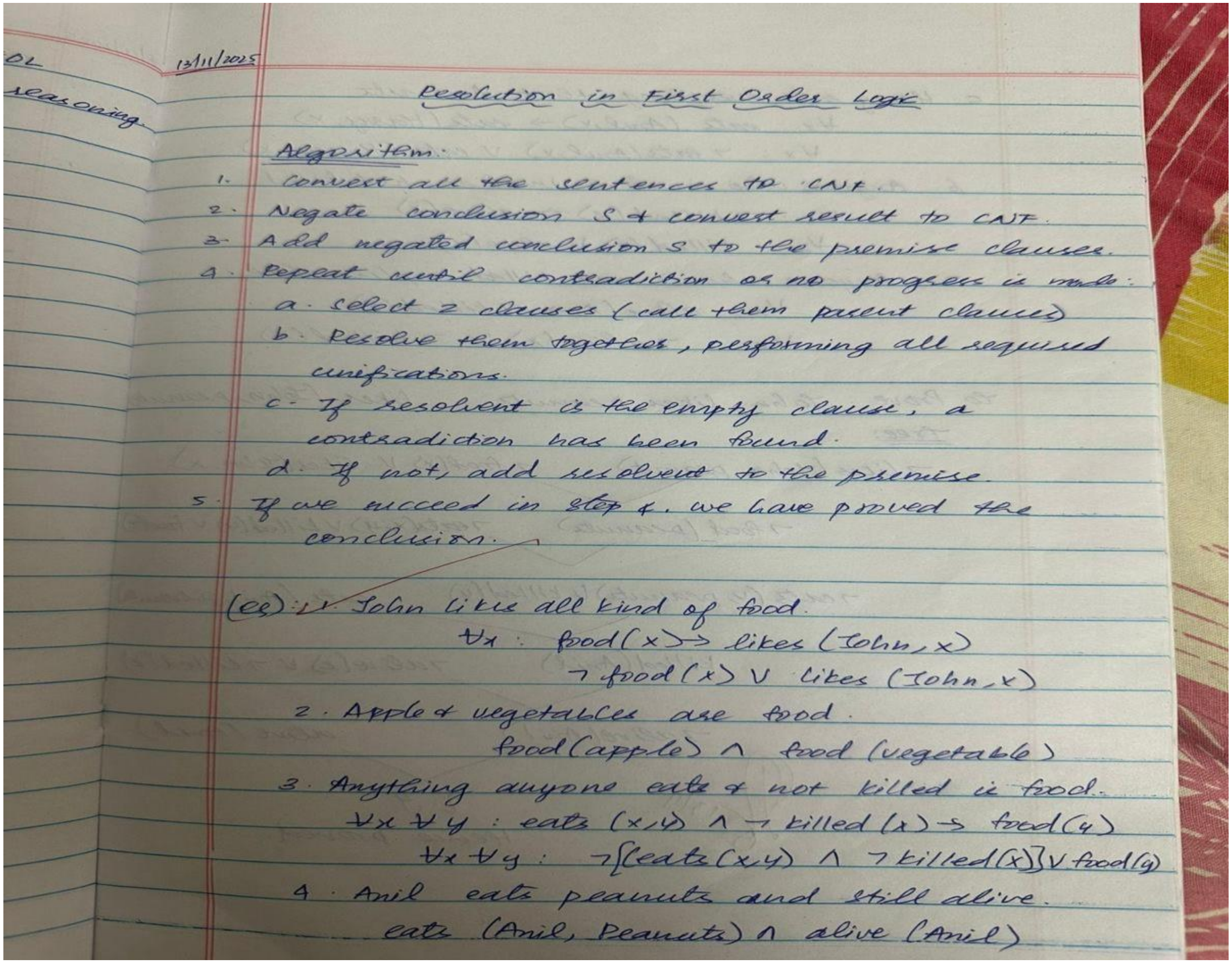


```
Truth Table for Knowledge Base:

    P     Q     R Q → P P → ¬Q Q ∨ R KB True? R → P Q → R
False False False  True   True False    False  True  True
False False  True  True   True  True     True False  True
False  True False False   True  True    False  True False
False  True  True False   True  True    False False  True
 True False False  True   True False    False  True  True
 True False  True  True   True  True     True  True  True
 True  True False  True  False  True    False  True False
 True  True  True  True  False  True    False  True  True

Models where KB is True:

       P      Q     R
1  False  False  True
5   True  False  True

Entailment Results:
KB ⊨ R ? Yes
KB ⊨ R → P ? No
KB ⊨ Q → R ? Yes

=== Code Execution Successful ===
```

Program 8:

Create a knowledge base using prepositional logic and prove the given query using resolution.

Algorithm:



OL

reasoning.

13/11/2025

Resolution in First Order Logic

Algorithm:

1. Convert all the 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.
   d. If not, add resolvent to the premise.
5. If we succeed in step 4. we have proved the conclusion.

(eg):- 1. John likes all kind of food.

$\forall x : food(x) \rightarrow likes(John, x)$

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

2. Apple & vegetables are food.

$food(apple) \wedge food(vegetable)$

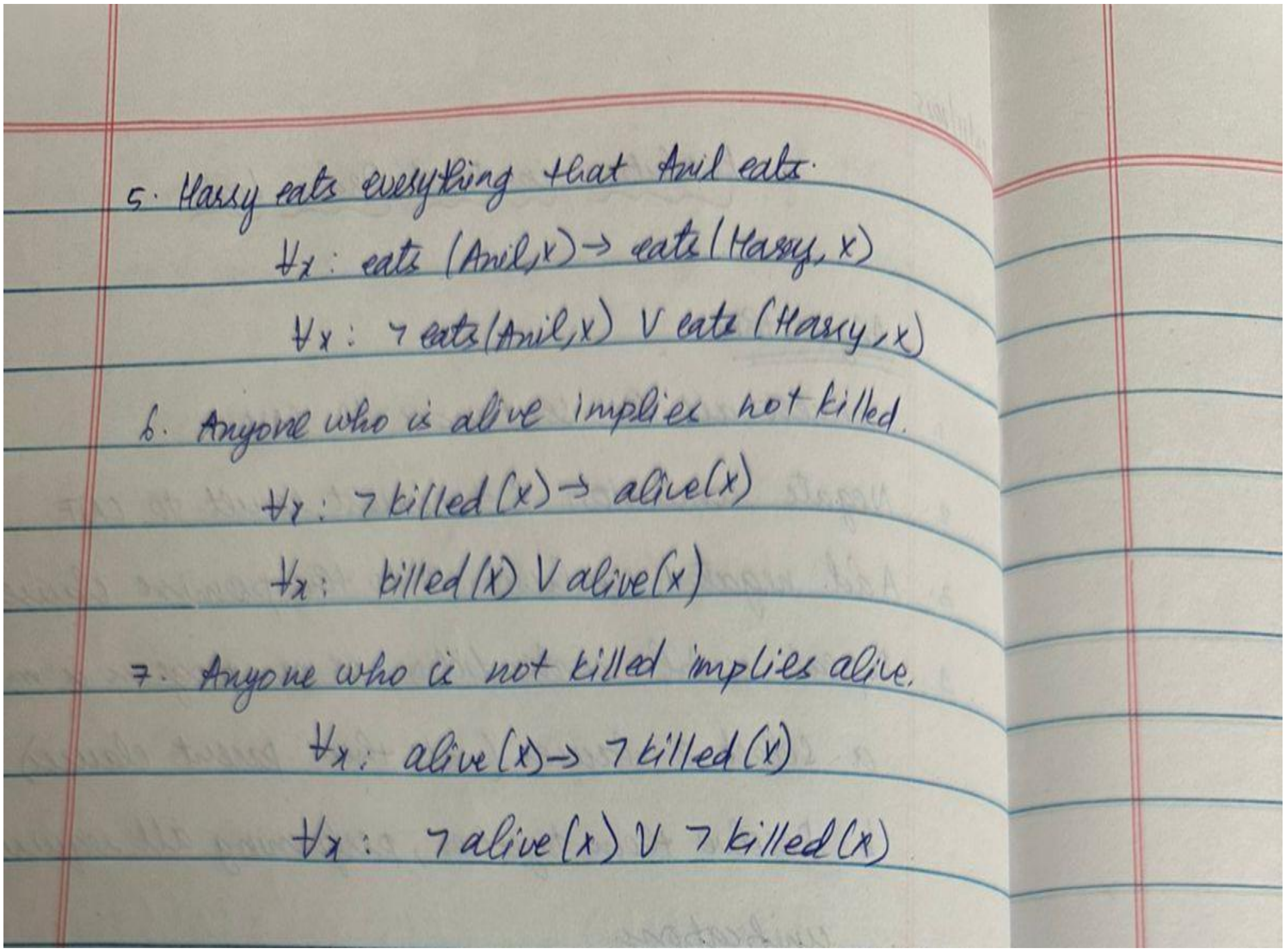
3. Anything anyone eats & not killed is food.

$\forall x \forall y : eats(x,y) \wedge \neg killed(x) \rightarrow food(y)$

$\forall x \forall y : \neg[eats(x,y) \wedge \neg killed(x)] \vee food(y)$

4. Anil eats peanuts and still alive.

$eats(Anil, Peanuts) \wedge alive(Anil)$



5. Harry eats everything that Anil eats.

$\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$

$\forall x : \neg eats(Anil, x) \vee eats(Harry, x)$

6. Anyone who is alive implies not killed.

$\forall x : \neg killed(x) \rightarrow alive(x)$

$\forall x : killed(x) \vee alive(x)$

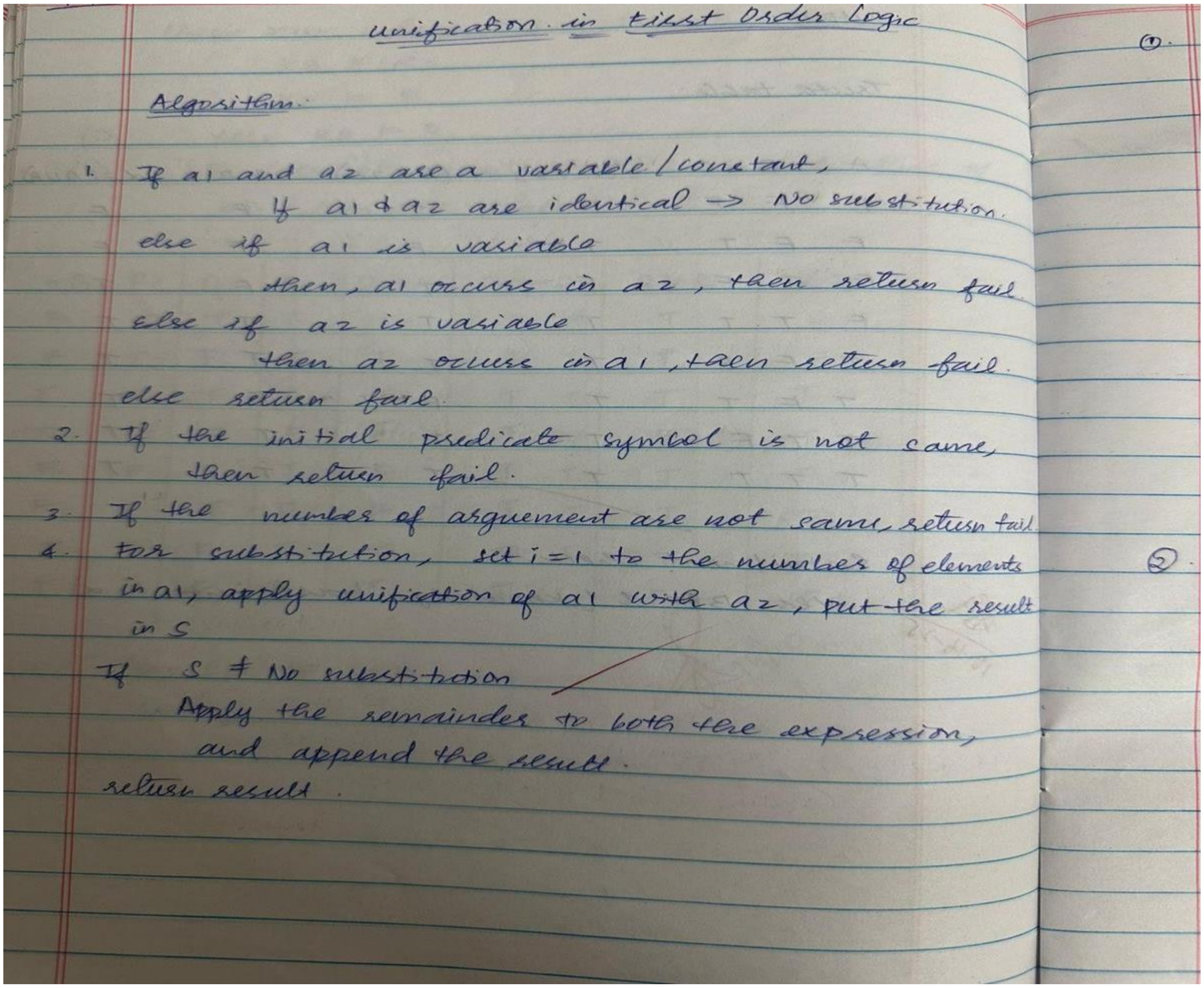
7. Anyone who is not killed implies alive.

$\forall x : alive(x) \rightarrow \neg killed(x)$

$\forall x : \neg alive(x) \vee \neg killed(x)$

Program 9:

Implement unification in first order logic.
Algorithm:

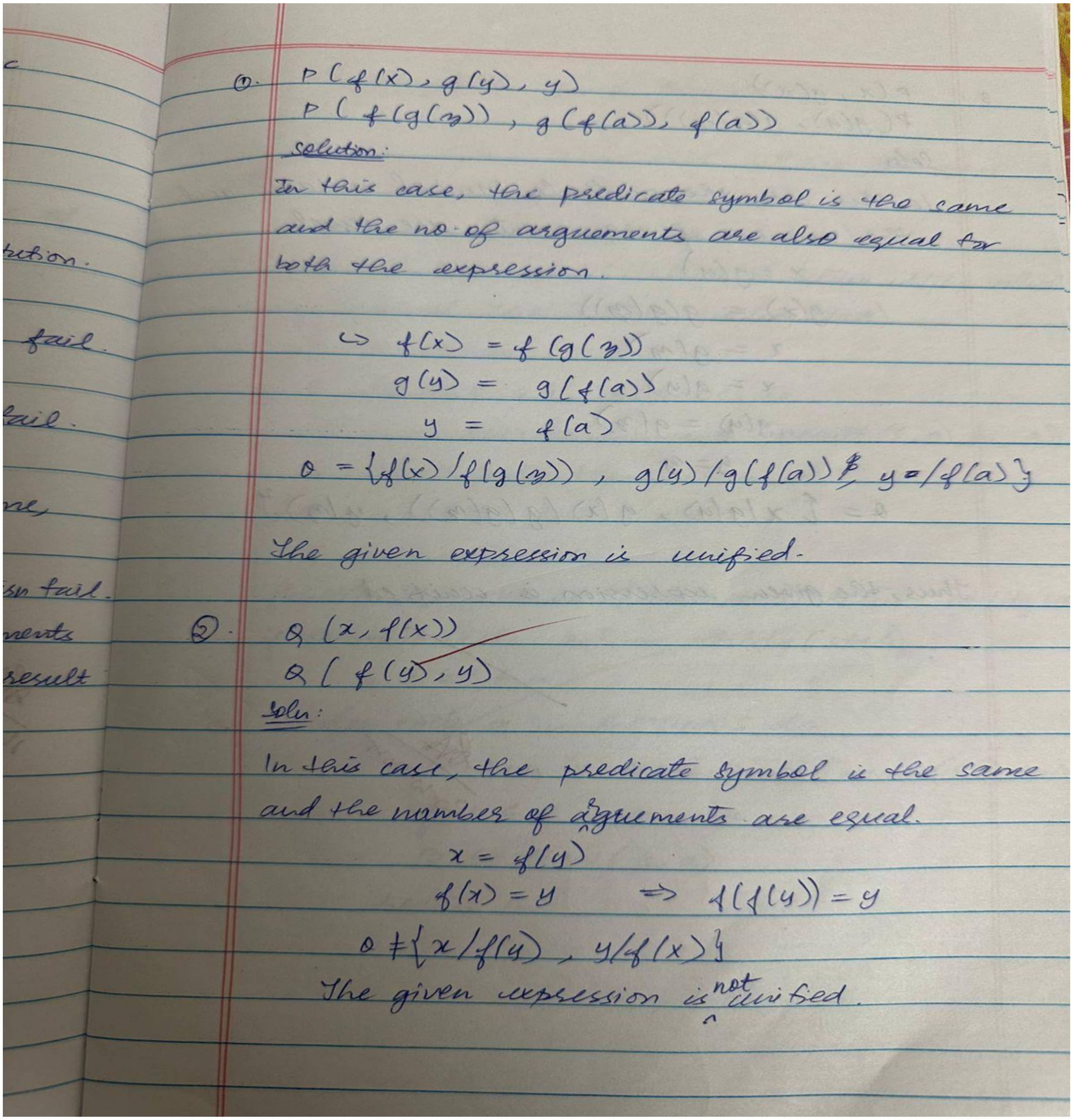


unification in First Order Logic

(1).

Algorithm:

```
1.  If a1 and a2 are a variable/constant,
          If a1 & a2 are identical -> No substitution.
    else if a1 is variable
          then, a1 occurs in a2, then return fail.
    else if a2 is variable
          then a2 occurs in a1, then return fail.
    else return fail.
2.  If the initial predicate symbol is not same,
       then return fail.
3.  If the number of arguement are not same, return fail.
4.  For substitution, set i=1 to the number of elements   (2).
    in a1, apply unification of a1 with a2, put the result
    in S
    If S ≠ No substitution
       Apply the remainder to both the expression,
       and append the result.
    return result.
```



1. $P(f(x), g(y), y)$

$P(f(g(z)), g(f(a)), f(a))$

solution:

In this case, the predicate symbol is the same and the no. of arguements are also equal for both the expression.

$\Rightarrow f(x) = f(g(z))$

$g(y) = g(f(a))$

$y = f(a)$

$\theta = \{f(x)/f(g(z)), g(y)/g(f(a)), y=/f(a)\}$

The given expression is unified.

2. $Q(x, f(x))$

$Q(f(y), y)$

Soln:

In this case, the predicate symbol is the same and the number of arguements are equal.

$x = f(y)$

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

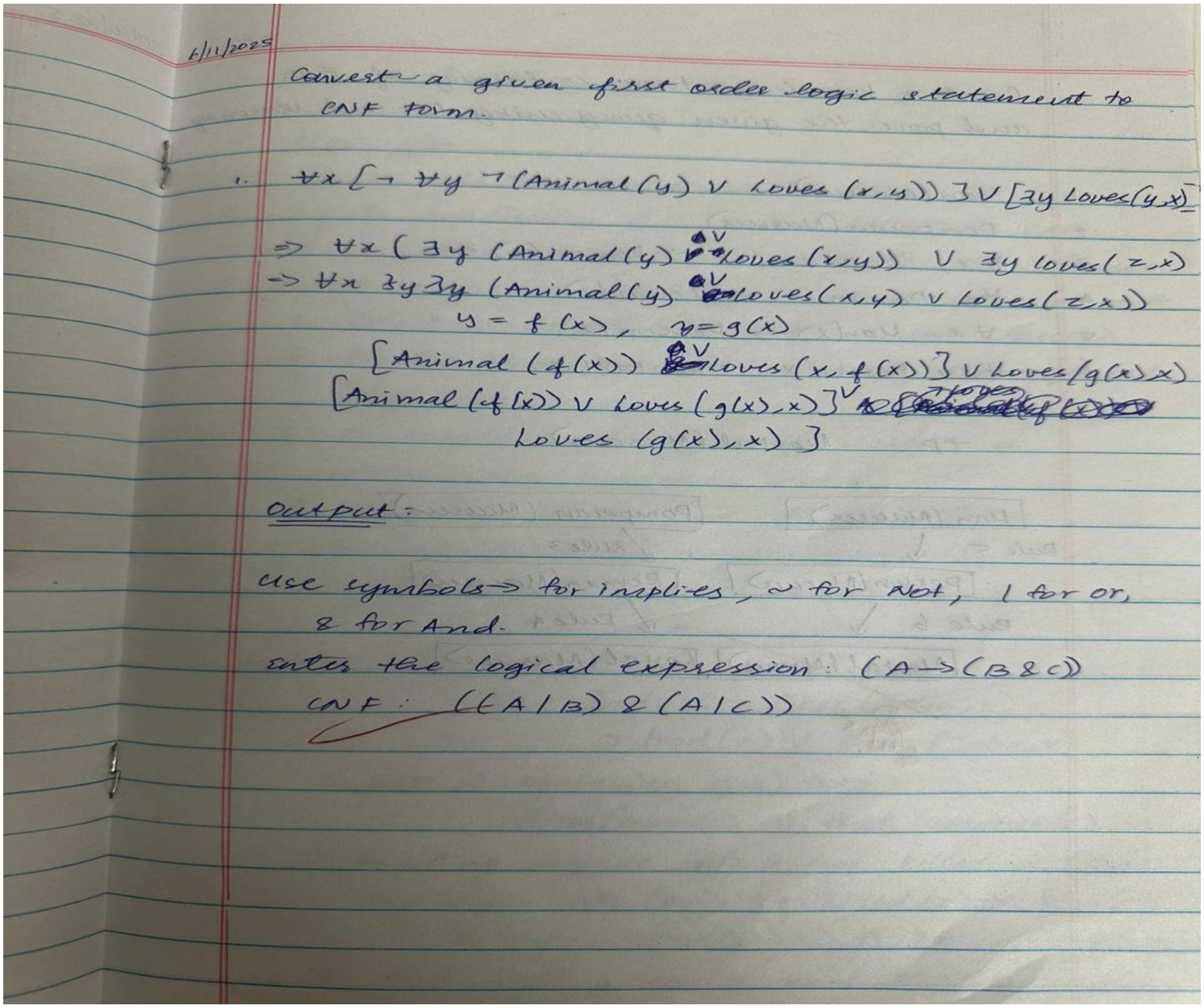
$\theta \neq \{x/f(y), y/f(x)\}$

The given expression is not unified.

```
import re

def getAttributes(expression):
    """
    Extracts the arguments/attributes from a predicate expression (e.g., P(a,b) -> ['a', 'b']).
    Note: This simplified version will struggle with nested functions like P(f(x), y).
    """
    expression = expression.split("(")[1:]
    expression = "(".join(expression)
    expression = expression.split(")")[:-1]
    expression = ")".join(expression)
    attributes = expression.split(',')
    return attributes

def getInitialPredicate(expression):
    """
    Extracts the predicate symbol (e.g., P(a,b) -> P).
    """
    return expression.split("(")[0]

def isConstant(char):
    """
    Checks if a string is a constant (single uppercase letter).
    """
    return char.isupper() and len(char) == 1

def isVariable(char):
    """
    Checks if a string is a variable (single lowercase letter).
    """
    return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):
    """
    Applies a single substitution (old -> new) to the attributes of an expression.
    """
    attributes = getAttributes(exp)
    predicate = getInitialPredicate(exp)
    for index, val in enumerate(attributes):
        if val == old:
            attributes[index] = new
    return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):
    """
    Applies a list of substitutions sequentially to an expression.
    """
    for new, old in substitutions:
        exp = replaceAttributes(exp, old, new)
    return exp

def checkOccurs(var, exp):
    """
    Performs the Occurs Check: returns True if 'var' is found in 'exp'.
    """
    if exp.find(var) == -1:
        return False
    return True

def getFirstPart(expression):
    """
    Gets the first argument of the predicate.
    """
    attributes = getAttributes(expression)
    return attributes[0]

def getRemainingPart(expression):
    """
    Returns the predicate with all arguments EXCEPT the first one.
    """
    predicate = getInitialPredicate(expression)
    attributes = getAttributes(expression)
    newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"
    return newExpression

def unify(exp1, exp2):
    """
    The main Unification Algorithm (recursive).
    """
    # Step 1: Base Case - Identical
    if exp1 == exp2:
        return []

    # Step 1: Base Case - Constants
    if isConstant(exp1) and isConstant(exp2):
        if exp1 != exp2:
            print(f"{exp1} and {exp2} are constants. Cannot be unified")
        return []

    # Step 1: Base Case - One is a Constant, One is a Variable/Term
    # (The original code handles simple constants, but this block is incomplete
    # and likely intended for constant-term/variable unification.)
    if isConstant(exp1):
        # NOTE: This assumes exp2 is a variable or constant, but if exp2 is complex, this fails.
        return [(exp1, exp2)]
    if isConstant(exp2):
        # NOTE: This assumes exp1 is a variable or constant, but if exp1 is complex, this fails.
        return [(exp2, exp1)]

    # Step 1: Variable Check (W1 is a variable)
    if isVariable(exp1):
        # returns [(exp2, exp1)] on success, or [] on failure (Occurs Check)
        return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

    # Step 1: Variable Check (W2 is a variable)
    if isVariable(exp2):
        # returns [(exp1, exp2)] on success, or [] on failure (Occurs Check)
        return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

    # Step 2: Predicate Check
    if getInitialPredicate(exp1) != getInitialPredicate(exp2):
        print("Cannot be unified as the predicates do not match!")
        return []

    # Step 3: Arity Check
    attributeCount1 = len(getAttributes(exp1))
    attributeCount2 = len(getAttributes(exp2))
    if attributeCount1 != attributeCount2:
        print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot be unified")
        return []

    # Steps 5 & 6: Recursive Unification

    # Unify the first parts (heads)
    head1 = getFirstPart(exp1)
    head2 = getFirstPart(exp2)
    initialSubstitution = unify(head1, head2)

    if not initialSubstitution and (head1 != head2): # Note: Added head1!=head2 check for clarity
        return []

    # If only one attribute remains, we're done
    if attributeCount1 == 1:
        return initialSubstitution

    # Prepare the remaining parts (tails)
    tail1 = getRemainingPart(exp1)
    tail2 = getRemainingPart(exp2)

    # Apply substitution to the remaining parts (Crucial step for consistency)
    if initialSubstitution:
        tail1 = apply(tail1, initialSubstitution)
        tail2 = apply(tail2, initialSubstitution)

    # Recursively unify the remaining parts
    remainingSubstitution = unify(tail1, tail2)

    if remainingSubstitution is False: # Check if remaining unification failed
        return []

    # Compose the substitutions
    return initialSubstitution + remainingSubstitution

if __name__== "__main__":
    print("Unification Algorithm (Simple String-based)")
    print("-" * 35)
    print("Enter the first expression (e.g., P(x,f(y)))")
    e1 = input()

    print("Enter the second expression (e.g., P(g(z),f(a)))")
    e2 = input()

    # Note: This implementation has limitations with complex nested terms due to simple string parsing.
    substitutions = unify(e1, e2)

    print("-" * 35)
    print("The substitutions (MGU) are:")
    if substitutions is False or substitutions == []:
        print("Unification Failed or No Substitutions Needed.")
    else:
        # Reformat the output for clarity (e.g., ('g(z)', 'x') -> g(z) / x)
        print([f"{new} / {old}" for new, old in substitutions])
```

Program 10:

Convert a given first order logic statement into Conjunctive Normal Form (CNF). Algorithm:



6/11/2025

Convert a given first order logic statement to CNF form.

1. ∀x [¬ ∀y ¬ (Animal(y) ∨ Loves (x, y))] ∨ [∃y Loves(y, x)]

⇒ ∀x (∃y (Animal(y) ∨ Loves (x,y)) ∨ ∃y loves(z,x)

→ ∀x ∃y ∃y (Animal(y) ∨ Loves(x,y) ∨ Loves(z,x))

y = f (x), y = g(x)

[Animal (f(x)) ∨ Loves (x, f(x))] ∨ Loves(g(x),x)

[Animal (f(x)) ∨ Loves (g(x), x)] ∨ [
Loves (g(x), x)]

Output :

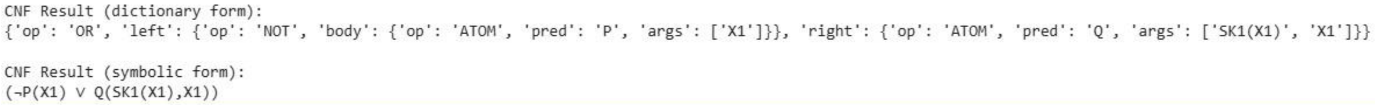
use symbols → for implies , ~ for Not, | for or, & for And.

enter the logical expression : (A→(B&C))

CNF : ((A|B) & (A|C))

```
import itertools

# --------------------------------------------------------------------
# Use ASCII strings as operator names
# --------------------------------------------------------------------
FORALL = 'FORALL'       # ∀
EXISTS = 'EXISTS'       # ∃
NOT = 'NOT'             # ¬
AND = 'AND'             # ∧
OR = 'OR'               # ∨
IMPLIES = 'IMPLIES'     # →
IFF = 'IFF'             # ↔
ATOM = 'ATOM'

# --------------------------------------------------------------------
# Unicode symbols for pretty printing only
# --------------------------------------------------------------------
unicode_symbols = {
    FORALL: '∀',
    EXISTS: '∃',
    NOT: '¬',
    AND: '∧',
    OR: '∨',
    IMPLIES: '→',
    IFF: '↔'
}

# ------------------------------------------------------------------
# Utility counters for skolem and variables
# ------------------------------------------------------------------
_skolem_counter = itertools.count(1)
_var_counter = itertools.count(1)

def new_skolem_name():
    return f"SK{next(_skolem_counter)}"

def new_var_name():
    return f"X{next(_var_counter)}"

# ------------------------------------------------------------------
# STEP 1: Eliminate ↔ and →
# ------------------------------------------------------------------
def eliminate_implications(F):
    if F is None:
        return None

    op = F['op']

    if op == ATOM:
        return F
    elif op == IMPLIES:
        # (α → β) ≡ (¬α ∨ β)
        return {'op': OR,
                'left': {'op': NOT, 'body': eliminate_implications(F['left'])},
                'right': eliminate_implications(F['right'])}
    elif op == IFF:
        # (α ↔ β) ≡ (α → β) ∧ (β → α)
        p = eliminate_implications(F['left'])
        q = eliminate_implications(F['right'])
        return {'op': AND,
                'left': {'op': OR, 'left': {'op': NOT, 'body': p}, 'right': q},
                'right': {'op': OR, 'left': {'op': NOT, 'body': q}, 'right': p}}
    elif op == NOT:
        return {'op': NOT, 'body': eliminate_implications(F['body'])}
    elif op in (AND, OR):
        return {'op': op,
                'left': eliminate_implications(F['left']),
                'right': eliminate_implications(F['right'])}
    elif op in (FORALL, EXISTS):
        return {'op': op, 'var': F['var'], 'body': eliminate_implications(F['body'])}
    return F

# ------------------------------------------------------------------
# STEP 2: Move negations inward
# ------------------------------------------------------------------
def move_negations_inward(F):
    if F['op'] == ATOM:
        return F
    elif F['op'] == NOT:
        sub = F['body']
        if sub['op'] == ATOM:
            return F
        elif sub['op'] == NOT:
            return move_negations_inward(sub['body'])
        elif sub['op'] == AND:
            return {'op': OR,
                    'left': move_negations_inward({'op': NOT, 'body': sub['left']}),
                    'right': move_negations_inward({'op': NOT, 'body': sub['right']})}
        elif sub['op'] == OR:
            return {'op': AND,
                    'left': move_negations_inward({'op': NOT, 'body': sub['left']}),
                    'right': move_negations_inward({'op': NOT, 'body': sub['right']})}
        elif sub['op'] == FORALL:
            return {'op': EXISTS, 'var': sub['var'],
                    'body': move_negations_inward({'op': NOT, 'body': sub['body']})}
        elif sub['op'] == EXISTS:
            return {'op': FORALL, 'var': sub['var'],
                    'body': move_negations_inward({'op': NOT, 'body': sub['body']})}
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': move_negations_inward(F['left']),
                'right': move_negations_inward(F['right'])}
    elif F['op'] in (FORALL, EXISTS):
        return {**F, 'body': move_negations_inward(F['body'])}
    return F

# ------------------------------------------------------------------
# STEP 3: Standardize variables
# ------------------------------------------------------------------
def standardize_variables(F, mapping=None):
    if mapping is None:
        mapping = {}
    if F['op'] == ATOM:
        args = [mapping.get(a, a) for a in F['args']]
        return {'op': ATOM, 'pred': F['pred'], 'args': args}
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': standardize_variables(F['left'], mapping),
                'right': standardize_variables(F['right'], mapping)}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': standardize_variables(F['body'], mapping)}
    elif F['op'] in (FORALL, EXISTS):
        new_var = new_var_name()
        mapping2 = mapping.copy()
        mapping2[F['var']] = new_var
        return {'op': F['op'], 'var': new_var, 'body': standardize_variables(F['body'], mapping2)}
    return F

# ------------------------------------------------------------------
# STEP 4: Skolemize (remove ∃)
# ------------------------------------------------------------------
def skolemize(F, scope_vars=None):
    if scope_vars is None:
        scope_vars = []
    if F['op'] == ATOM:
        return F
    elif F['op'] == FORALL:
        return {'op': FORALL, 'var': F['var'], 'body': skolemize(F['body'], scope_vars + [F['var']])}
    elif F['op'] == EXISTS:
        skolem_name = new_skolem_name()
        skolem_term = skolem_name if not scope_vars else f"{skolem_name}({','.join(scope_vars)})"
        return skolemize(substitute(F['body'], F['var'], skolem_term), scope_vars)
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': skolemize(F['left'], scope_vars),
                'right': skolemize(F['right'], scope_vars)}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': skolemize(F['body'], scope_vars)}
    return F

def substitute(F, var, term):
    if F['op'] == ATOM:
        new_args = [term if a == var else a for a in F['args']]
        return {'op': ATOM, 'pred': F['pred'], 'args': new_args}
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': substitute(F['left'], var, term),
                'right': substitute(F['right'], var, term)}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': substitute(F['body'], var, term)}
    elif F['op'] in (FORALL, EXISTS):
        if F['var'] == var:
            return F
        return {**F, 'body': substitute(F['body'], var, term)}
    return F

# ------------------------------------------------------------------
# STEP 5: Drop ∀ quantifiers
# ------------------------------------------------------------------
def drop_universal_quantifiers(F):
    if F['op'] == FORALL:
        return drop_universal_quantifiers(F['body'])
    elif F['op'] in (AND, OR):
        return {'op': F['op'],
                'left': drop_universal_quantifiers(F['left']),
                'right': drop_universal_quantifiers(F['right'])}
    elif F['op'] == NOT:
        return {'op': NOT, 'body': drop_universal_quantifiers(F['body'])}
    return F

# ------------------------------------------------------------------
# STEP 6: Distribute ∨ over ∧
# ------------------------------------------------------------------
def distribute_or_over_and(F):
    if F['op'] == OR:
        A = distribute_or_over_and(F['left'])
        B = distribute_or_over_and(F['right'])
        if A['op'] == AND:
            return {'op': AND,
                    'left': distribute_or_over_and({'op': OR, 'left': A['left'], 'right': B}),
                    'right': distribute_or_over_and({'op': OR, 'left': A['right'], 'right': B})}
        elif B['op'] == AND:
            return {'op': AND,
                    'left': distribute_or_over_and({'op': OR, 'left': A, 'right': B['left']}),
                    'right': distribute_or_over_and({'op': OR, 'left': A, 'right': B['right']})}
        else:
            return {'op': OR, 'left': A, 'right': B}
    elif F['op'] == AND:
        return {'op': AND,
                'left': distribute_or_over_and(F['left']),
                'right': distribute_or_over_and(F['right'])}
    else:
        return F

# ------------------------------------------------------------------
# PRETTY PRINTING (SYMBOLIC OUTPUT)
# ------------------------------------------------------------------
def pretty(F):
    """Convert CNF structure to readable symbolic formula."""
    op = F['op']
    if op == ATOM:
        args = ",".join(F['args'])
        return f"{F['pred']}({args})"
    elif op == NOT:
        return f"¬{pretty(F['body'])}"
    elif op in (AND, OR):
        return f"({pretty(F['left'])} {unicode_symbols[op]} {pretty(F['right'])})"
    return str(F)

# ------------------------------------------------------------------
# MAIN WRAPPER
# ------------------------------------------------------------------
def to_CNF(F):
    F1 = eliminate_implications(F)
    F2 = move_negations_inward(F1)
    F3 = standardize_variables(F2)
    F4 = skolemize(F3)
    F5 = drop_universal_quantifiers(F4)
    F6 = distribute_or_over_and(F5)
    return F6

# ------------------------------------------------------------------
# EXAMPLE
# ------------------------------------------------------------------
if __name__ == "__main__":
    # ∀x ( P(x) → ∃y Q(y,x) )
    formula = {
        'op': FORALL, 'var': 'x',
        'body': {
            'op': IMPLIES,
            'left': {'op': ATOM, 'pred': 'P', 'args': ['x']},
            'right': {
                'op': EXISTS, 'var': 'y',
                'body': {'op': ATOM, 'pred': 'Q', 'args': ['y', 'x']}
            }
        }
    }

    cnf = to_CNF(formula)
```

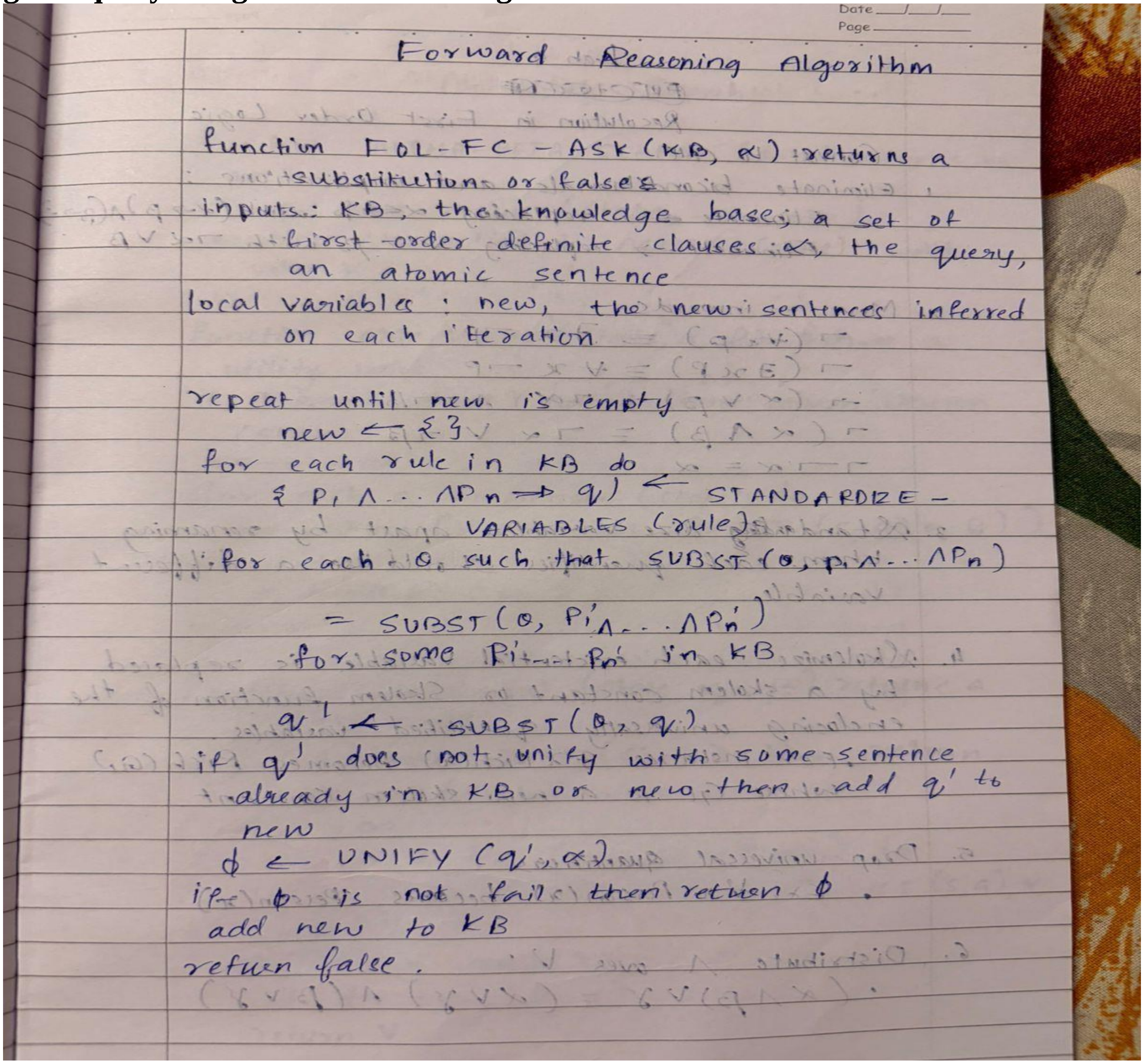
```
print("CNF Result (dictionary form):")
print(cnf)
print("\nCNF Result (symbolic form):")
print(pretty(cnf))
```



```
CNF Result (dictionary form):
{'op': 'OR', 'left': {'op': 'NOT', 'body': {'op': 'ATOM', 'pred': 'P', 'args': ['X1']}}, 'right': {'op': 'ATOM', 'pred': 'Q', 'args': ['SK1(X1)', 'X1']}}

CNF Result (symbolic form):
(¬P(X1) ∨ Q(SK1(X1),X1))
```

Program 11:

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



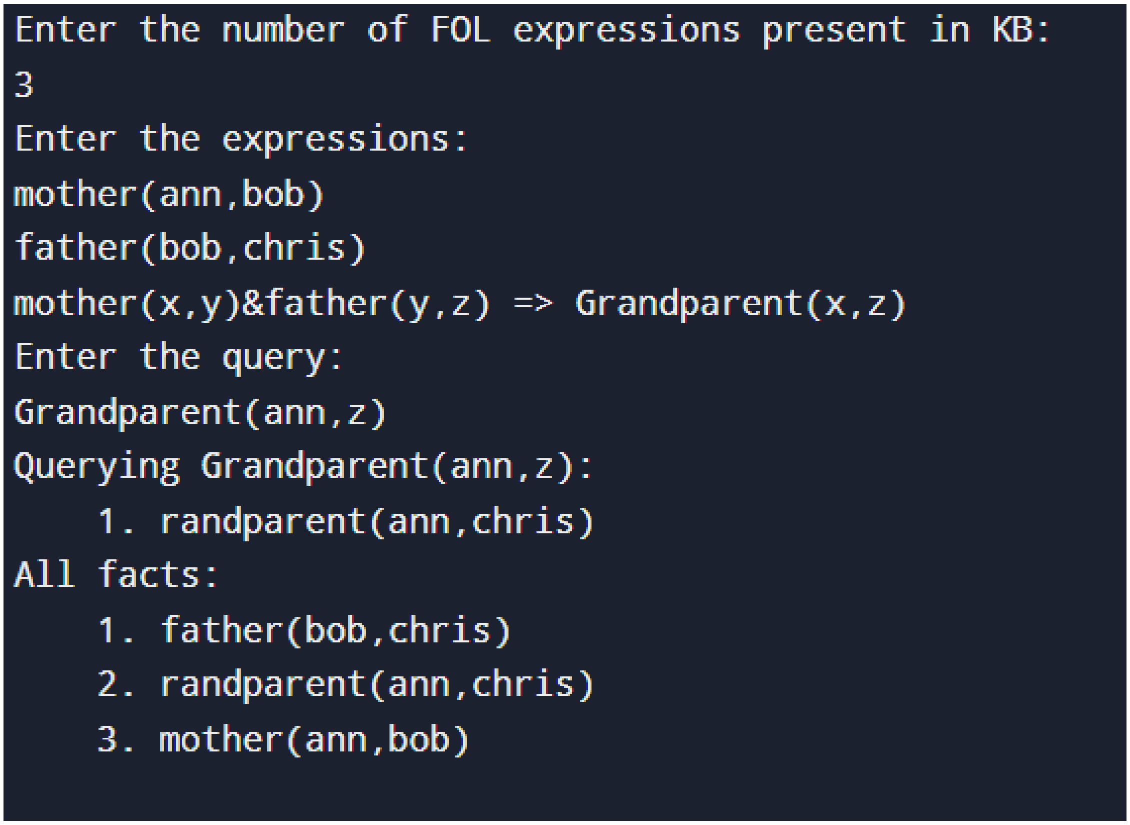
Date ___/___/___
Page ________

Forward Reasoning Algorithm

```
function FOL-FC-ASK(KB, $\alpha$) returns a
    substitution or false
inputs: KB, the knowledge base, a set of
    first-order definite clauses; $\alpha$, the query,
        an atomic sentence
local variables: new, the new sentences inferred
        on each iteration

repeat until new is empty
        new ← {}
    for each rule in KB do
            ($P_1 \wedge \dots \wedge P_n \Rightarrow q$) ← STANDARDIZE-
                        VARIABLES (rule)
        for each $\theta$ such that SUBST($\theta$, $P_1 \wedge \dots \wedge P_n$)
                = SUBST($\theta$, $P_1' \wedge \dots \wedge P_n'$)
            for some $P_1', \dots, P_n'$ in KB
            $q'$ ← SUBST($\theta$, $q$)
    if $q'$ does not unify with some sentence
    already in KB or new then add $q'$ to
        new
        $\phi$ ← UNIFY($q'$, $\alpha$)
    if $\phi$ is not fail then return $\phi$.
    add new to KB
return false.
```

Algorithm:

```
import re

def isVariable(x):
    """
    Heuristically checks if a string is a variable (single, lowercase, alphabetic).
    """
    return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):
    """
    Extracts the parenthesized attribute strings. (e.g., 'P(a,b)' -> ['(a,b)'])
    Note: This uses simple regex and will struggle with nested functions.
    """
    expr = r'\([^)]+\)'
    matches = re.findall(expr, string)
    return matches

def getPredicates(string):
    """
    Extracts the predicate symbol. (e.g., 'P(a,b)' -> ['P'])
    """
    expr = r'([a-z~]+)\([^&|]+\)'
    return re.findall(expr, string)

class Fact:
    """
    Represents a single fact in the KB (e.g., 'Mother(a,b)').
    """
    def __init__(self, expression):
        self.expression = expression
        predicate, params = self.splitExpression(expression)
        self.predicate = predicate
        self.params = params
        # result is true if the fact contains at least one constant
        self.result = any(self.getConstants())

    def splitExpression(self, expression):
        """
        Splits the expression into predicate and parameters.
        """
        predicate = getPredicates(expression)[0]
        params = getAttributes(expression)[0].strip('()').split(',')
        return [predicate, params]

    def getResult(self):
        """
        Returns True if the fact has at least one constant.
        (Usage seems specific to this simplified inference process).
        """
        return self.result

    def getConstants(self):
        """
        Returns a list of constants (or None for variables).
        """
        return [None if isVariable(c) else c for c in self.params]

    def getVariables(self):
        """
        Returns a list of variables (or None for constants).
        """
        return [v if isVariable(v) else None for v in self.params]

    def substitute(self, constants):
        """
        Applies a substitution (constants) to create a new grounded Fact.
        (Seems unused in the evaluate method, but provided here.)
        """
        c = constants.copy()
        f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in self.params])})"
        return Fact(f)

class Implication:
    """
    Represents an implication rule (e.g., 'Mother(x,y)&Father(y,z) => Grandparent(x,z)').
    """
    def __init__(self, expression):
        self.expression = expression
        l = expression.split('=>')
        # LHS is a list of Facts connected by '&'
        self.lhs = [Fact(f) for f in l[0].split('&')]
        self.rhs = Fact(l[1])

    def evaluate(self, facts):
        """
        Attempts to apply the rule (forward chaining) using existing facts.
        This simplified version finds constants by position matching.
        """
        constants = {}
        new_lhs = [] # Facts that successfully matched LHS predicates

        for fact in facts:
            for val in self.lhs:
                if val.predicate == fact.predicate:
                    # Collect constants based on variable positions
                    for i, v in enumerate(val.getVariables()):
                        if v:
                            constants[v] = fact.getConstants()[i]
                    new_lhs.append(fact)

        # Check if we matched all LHS facts and if all matched facts are "true" (have constants)
        if len(new_lhs) == len(self.lhs) and all([f.getResult() for f in new_lhs]):
            # Construct the RHS fact by applying the collected constants
            predicate, attributes = getPredicates(self.rhs.expression)[0],
str(getAttributes(self.rhs.expression)[0])
            for key in constants:
                if constants[key]:
                    # Simple string replacement for substitution
                    attributes = attributes.replace(key, constants[key])

            expr = f'{predicate}{attributes}'
            return Fact(expr)

        return None

class KB:
    """
    The Knowledge Base containing facts and implications.
    """
    def __init__(self):
        self.facts = set()
        self.implications = set()

    def tell(self, e):
        """
        Adds a new fact or implication to the KB and performs inference.
        """
        if '=>' in e:
            self.implications.add(Implication(e))
        else:
            self.facts.add(Fact(e))

        # Run forward chaining (simple form: only one pass)
        for i in self.implications:
            res = i.evaluate(self.facts)
            if res:
                self.facts.add(res)

    def ask(self, e):
        """
        Searches the current facts for facts matching the query's predicate.
        """
        facts = set([f.expression for f in self.facts])
        i = 1
        print(f'Querying {e}:')
        for f in facts:
            if Fact(f).predicate == Fact(e).predicate:
                print(f'\t{i}. {f}')
                i += 1

    def display(self):
        """
        Prints all unique facts currently in the KB.
        """
        print("All facts: ")
        for i, f in enumerate(set([f.expression for f in self.facts])):
            print(f'\t{i+1}. {f}')

def main():
    """
    Main execution function to run the KB system.
    """
    kb = KB()
    print("Enter the number of FOL expressions present in KB:")
    # Example input: 2
    n = int(input())

    print("Enter the expressions:")
    # Example inputs:
    # Mother(ANN,BOB)
    # Mother(x,y)&Father(y,z) => Grandparent(x,z)
    for i in range(n):
        fact = input()
        kb.tell(fact)

    print("Enter the query:")
    # Example input: Grandparent(ANN,z)
    query = input()

    kb.ask(query)
    kb.display()

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

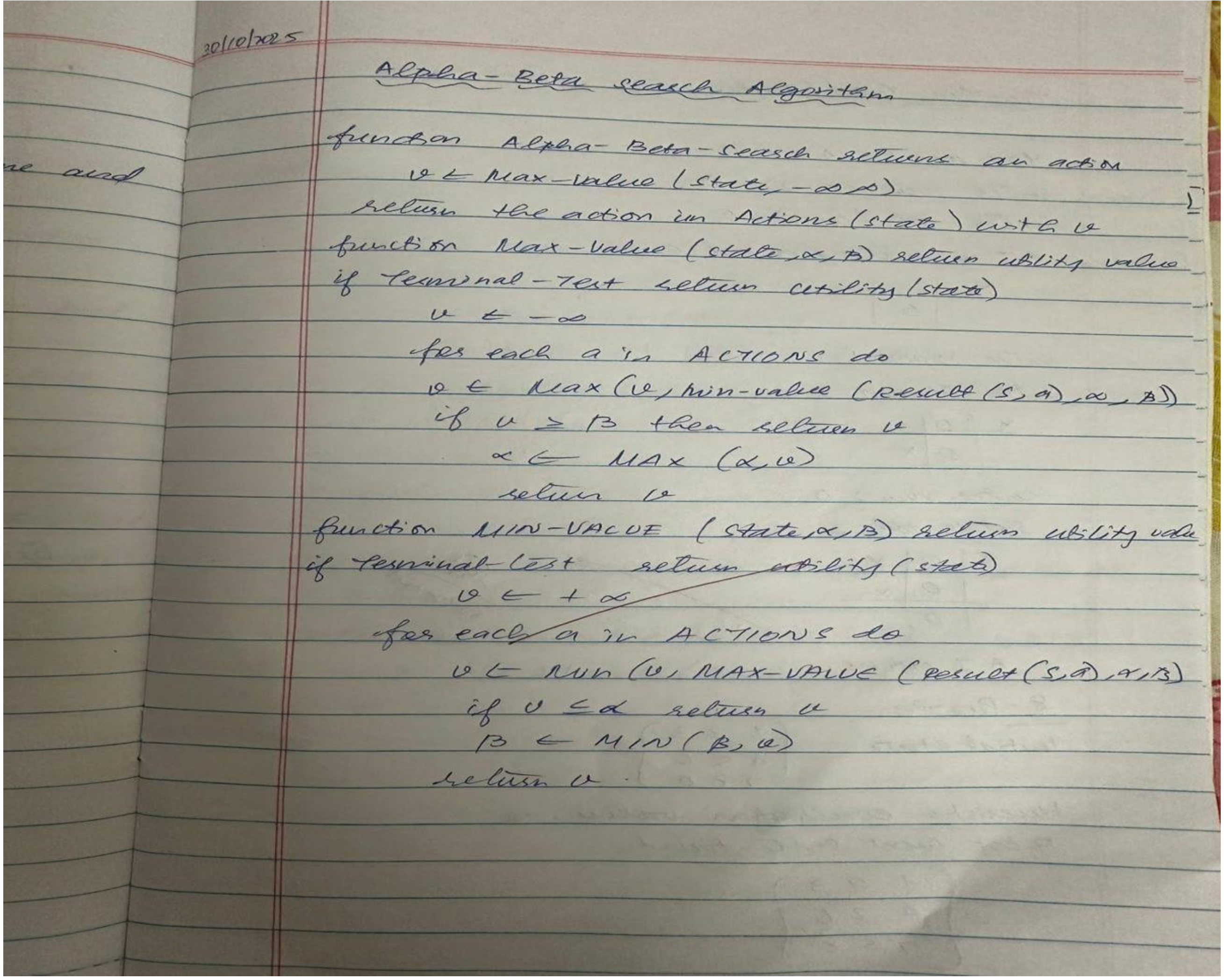


```
Enter the number of FOL expressions present in KB:
3
Enter the expressions:
mother(ann,bob)
father(bob,chris)
mother(x,y)&father(y,z) => Grandparent(x,z)
Enter the query:
Grandparent(ann,z)
Querying Grandparent(ann,z):
    1. randparent(ann,chris)
All facts:
    1. father(bob,chris)
    2. randparent(ann,chris)
    3. mother(ann,bob)
```

Program 12:

Implement Alpha-Beta Pruning.

Algorithm:



30/10/2025

ne and

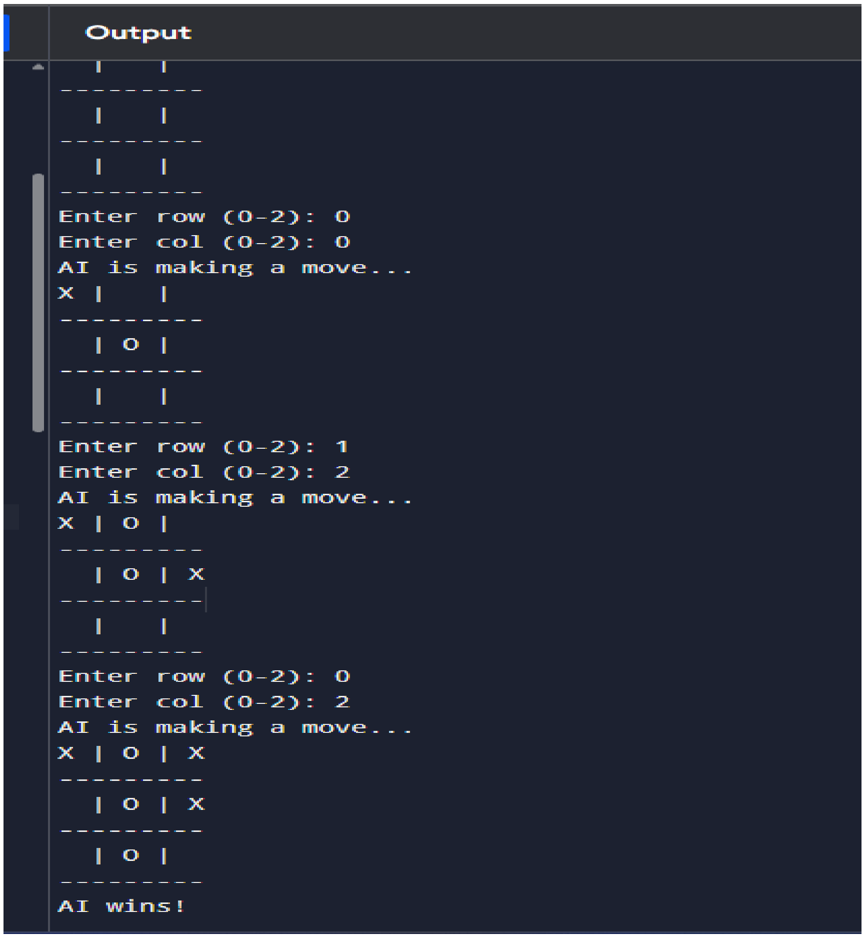
Alpha-Beta search Algorithm

```
function Alpha-Beta-Search returns an action
    v ← Max-value (state, -∞ ∞)
 return the action in Actions (state) with v
function Max-Value (state, α, β) return utility value
if Terminal-Test return utility (state)
        v ← -∞
      for each a in ACTIONS do
        v ← Max (v, min-value (Result (s, a), α, β))
        if v ≥ β then return v
            α ← MAX (α, v)
            return v
function MIN-VALUE (state, α, β) return utility value
if terminal-test   return utility (state)
        v ← + ∞
    for each a in ACTIONS do
          v ← min (v, MAX-VALUE (result (s, a), α, β)
          if v ≤ α return v
            β ← MIN (β, v)
        return v.
```

Code:

```
import math
PLAYER = "X"  # Human
AI = "O"      # Computer
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)
def available_moves(board):
    """Return list of available (row, col) moves."""
    moves = []
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                moves.append((i, j))
    return moves
def check_winner(board):
    """Return 'X' if X wins, 'O' if O wins, or None otherwise."""
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != " ":
            return board[0][i]
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return board[0][2]
    return None
def is_full(board):
    return all(cell != " " for row in board for cell in row)
def minimax(board, depth, is_maximizing):
    winner = check_winner(board)
    if winner == AI:
        return 1
    elif winner == PLAYER:
        return -1
    elif is_full(board):
        return 0
    if is_maximizing:
        best_score = -math.inf
        for (i, j) in available_moves(board):
            board[i][j] = AI
            score = minimax(board, depth + 1, False)
            board[i][j] = " "
            best_score = max(score, best_score)
        return best_score
    else:
        best_score = math.inf
        for (i, j) in available_moves(board):
            board[i][j] = PLAYER
            score = minimax(board, depth + 1, True)
            board[i][j] = " "
            best_score = min(score, best_score)
        return best_score
def best_move(board):
    """Find the best move for the AI."""
    best_score = -math.inf
    move = None
    for (i, j) in available_moves(board):
        board[i][j] = AI
        score = minimax(board, 0, False)
        board[i][j] = " "
        if score > best_score:
            best_score = score
            move = (i, j)
    return     move
def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Tic Tac Toe - You are X, AI is O")
    print_board(board)
    while True:
        row = int(input("Enter row (0-2): "))
        col = int(input("Enter col (0-2): "))
        if board[row][col] != " ":
            print("Cell taken, try again.")
            continue
        board[row][col] = PLAYER
        if check_winner(board) == PLAYER:
            print_board(board)
            print("You win!")
            break
        elif is_full(board):
            print_board(board)
            print("It's a draw!")
            break
        print("AI is making a move...")
        move = best_move(board)
        if move:
            board[move[0]][move[1]] = AI
        print_board(board)
        if check_winner(board) == AI:
            print("AI wins!")
            break
        elif is_full(board):
            print("It's a draw!")
            break
if _name___== "_main_":
    play_game()
```

ScreenShot:



```
  |   |
---------
  |   |
---------
  |   |
---------
Enter row (0-2): 0
Enter col (0-2): 0
AI is making a move...
X |   |
---------
  | O |
---------
  |   |
---------
Enter row (0-2): 1
Enter col (0-2): 2
AI is making a move...
X | O |
---------
  | O | X
---------
  |   |
---------
Enter row (0-2): 0
Enter col (0-2): 2
AI is making a move...
X | O | X
---------
  | O | X
---------
  | O |
---------
AI wins!
```