VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence

(23CS5PCAIN)

Submitted by:

***GHANSHYAM SHARMA*(1BM23CS100)**

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

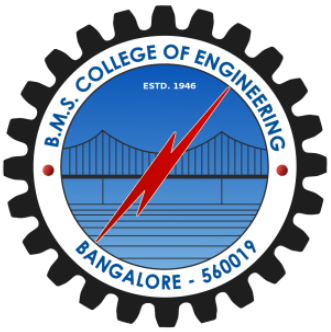
(Autonomous Institution under VTU)

BENGALURU-560019

February 2025 - June 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**" carried out by **GHANSHYAM SHARMA (1BM23CS100),** 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 during the year 2025-26. The Lab report has been approved as it satisfies the academic requirements in respect of **Artificial Intelligence - (23CS5PCAIN)** work prescribed for the said degree.

Prof. Sheetal V.A.
Assistant Professor
Department of CSE
BMSCE, Bengaluru

Dr. Kavitha Sooda
Professor and Head
Department of CSE
BMSCE, Bengaluru

Table Of Contents

Course Outcomes (COs):

CO1: Apply knowledge of agent architecture, searching and reasoning techniques for different applications

CO2: Analyse Searching and Inferencing Techniques

CO3: Design a reasoning and gaming system for a given requirement

CO4: Conduct practical experiments for demonstrating agents, searching and inferencing

1.IMPLEMENT TIC TAC TOE GAME

PseudoCode:

Tic Tac Toe Algorithm

0	1	2
3	4	5
6	7	8

My approach:-

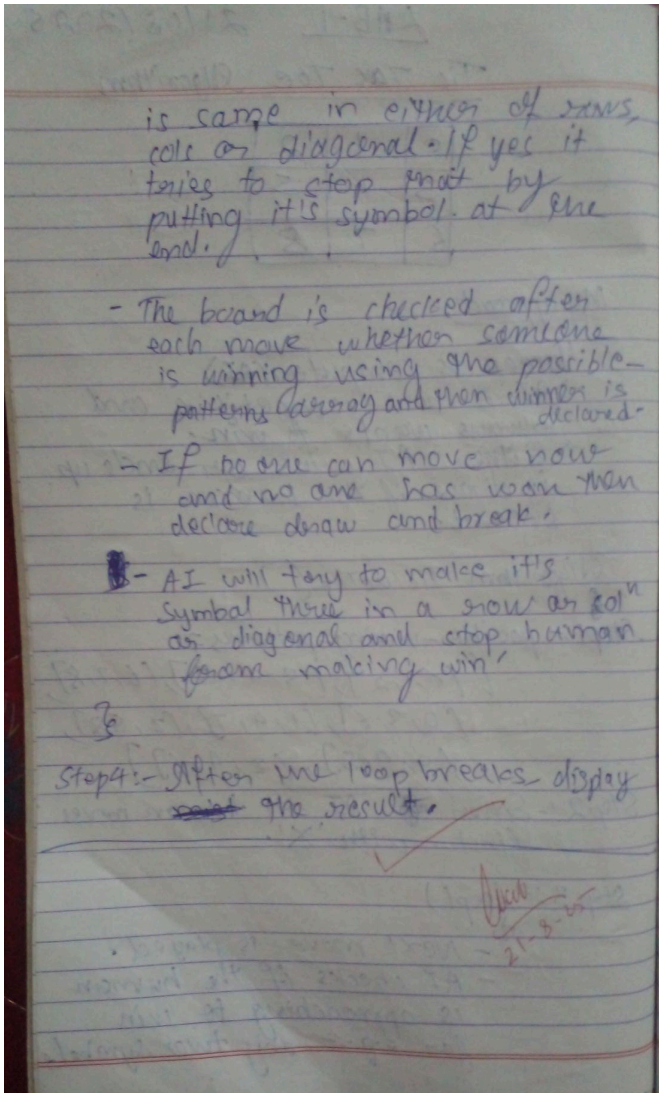
Human vs computer(AI)
→AI is more competitive and always wants to win.
→Either AI wins or ends up in draw if the human is smart.

Algorithm TicTacToe()

Step1:- possible_winning_patterns =
[[0,1,2],[3,4,5],[6,7,8],
[0,3,6],[1,4,7],[2,5,8],
[0,4,8],[2,4,6]]

Step2:- Either AI or human moves first with 'X'.

Step3:- Loop()
{
- Next move is played.
- AI checks if the human is approaching to win for e.g.:- any two symbols



Code:

```
from tkinter import *
from tkinter import messagebox
import random

Player1 = 'X'
stop_game = False
ai_mode = "Dumb"

def clicked(r, c):
    global Player1, stop_game

    if Player1 == "X" and states[r][c] == 0 and not stop_game:
        b[r][c].configure(text="X")
        states[r][c] = 'X'
```

```
        Player1 = 'O'
        check_if_win()

        if not stop_game:
            ai_move()

def ai_move():
    global Player1, stop_game

    if stop_game:
        return

    if ai_mode == "Dumb":
        dumb_ai_move()
    else:
        smart_ai_move()

    Player1 = 'X'
    check_if_win()

def dumb_ai_move():
    empty_cells = [(r, c) for r in range(3) for c in range(3) if
states[r][c] == 0]
    if empty_cells:
        r, c = random.choice(empty_cells)
        b[r][c].configure(text='O')
        states[r][c] = 'O'

def smart_ai_move():
    best_score = -float('inf')
    best_move = None

    for r in range(3):
        for c in range(3):
            if states[r][c] == 0:
                states[r][c] = 'O'
```

```
                score = minimax(states, False)
                states[r][c] = 0
                if score > best_score:
                    best_score = score
                    best_move = (r, c)

    if best_move:
        r, c = best_move
        b[r][c].configure(text='O')
        states[r][c] = 'O'

def minimax(board, is_maximizing):
    # Check for terminal states
    winner = check_winner_for_minimax(board)
    if winner == 'O':
        return 1
    elif winner == 'X':
        return -1
    elif winner == 'Tie':
        return 0

    if is_maximizing:
        best_score = -float('inf')
        for r in range(3):
            for c in range(3):
                if board[r][c] == 0:
                    board[r][c] = 'O'
                    score = minimax(board, False)
                    board[r][c] = 0
                    best_score = max(score, best_score)
        return best_score
    else:
        best_score = float('inf')
        for r in range(3):
            for c in range(3):
                if board[r][c] == 0:
```

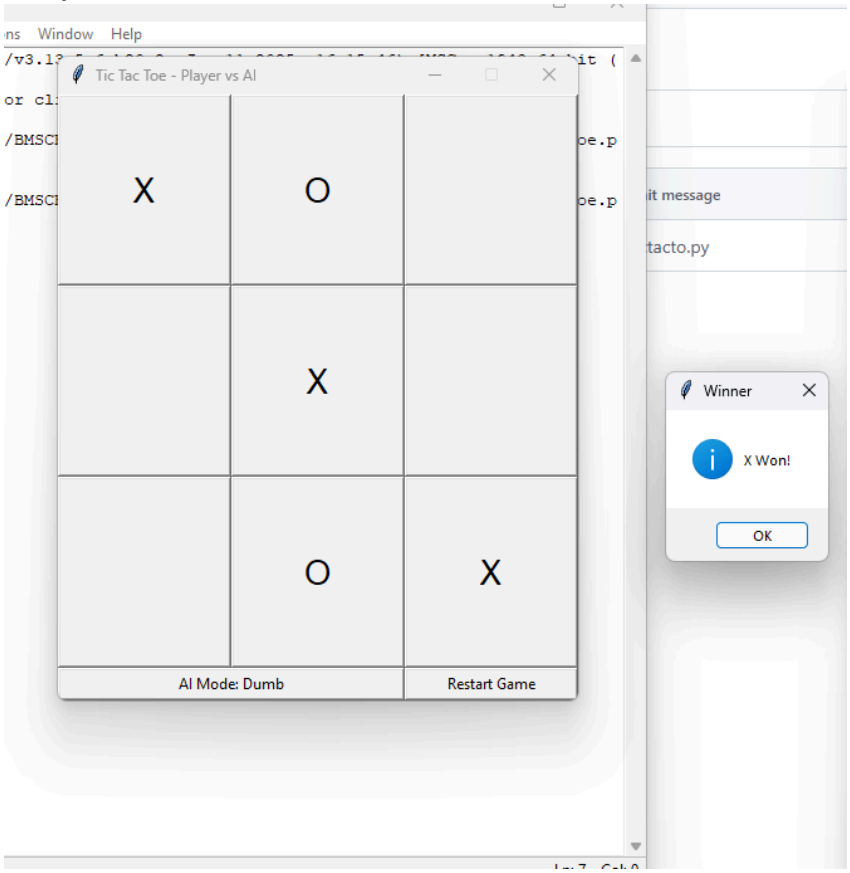
```
                    board[r][c] = 'X'
                    score = minimax(board, True)
                    board[r][c] = 0
                    best_score = min(score, best_score)
        return best_score

def check_winner_for_minimax(board):
    # Rows and Columns
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != 0:
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] != 0:
            return board[0][i]

    # Diagonals
    if board[0][0] == board[1][1] == board[2][2] != 0:
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] != 0:
        return board[0][2]

    # Check tie
    if all(board[r][c] != 0 for r in range(3) for c in
range(3)):
        return 'Tie'

    return None

def check_if_win():
    global stop_game

    for i in range(3):
        if states[i][0] == states[i][1] == states[i][2] != 0:
            stop_game = True
            messagebox.showinfo("Winner", states[i][0] + "
Won!")
            return
```

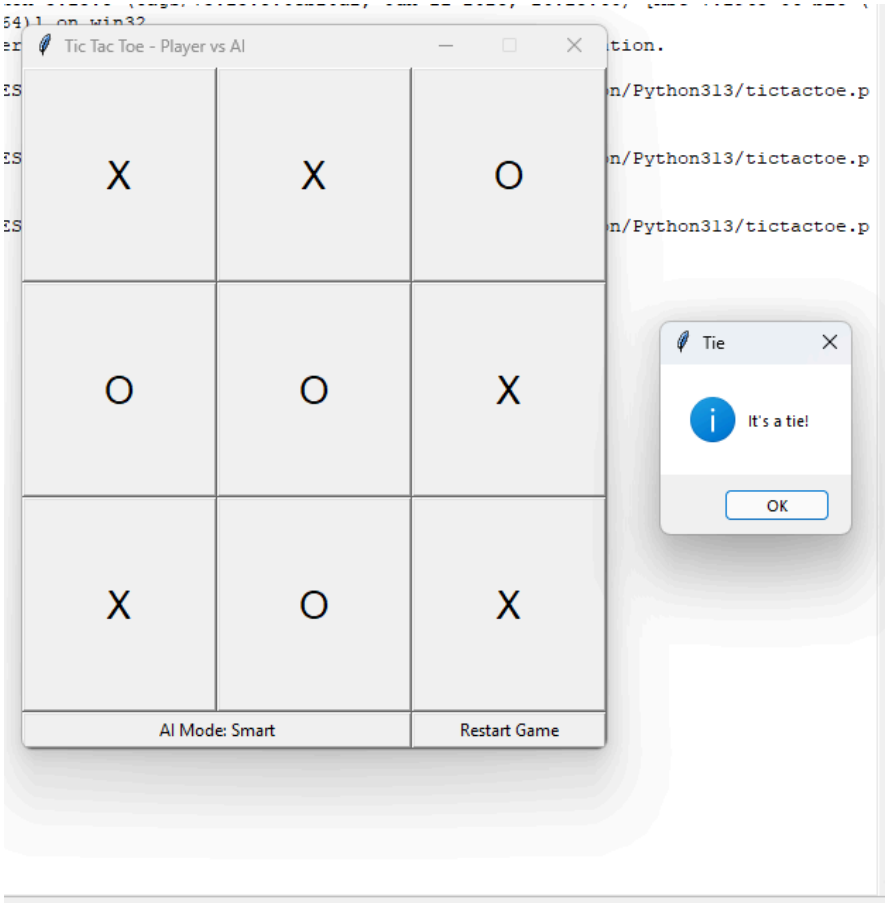
```
        if states[0][i] == states[1][i] == states[2][i] != 0:
            stop_game = True
            messagebox.showinfo("Winner", states[0][i] + "
Won!")
            return

    if states[0][0] == states[1][1] == states[2][2] != 0:
        stop_game = True
        messagebox.showinfo("Winner", states[0][0] + " Won!")
        return

    if states[0][2] == states[1][1] == states[2][0] != 0:
        stop_game = True
        messagebox.showinfo("Winner", states[0][2] + " Won!")
        return

    if all(states[r][c] != 0 for r in range(3) for c in
range(3)):
        stop_game = True
        messagebox.showinfo("Tie", "It's a tie!")
        return

def switch_ai_mode():
    global ai_mode, switch_button, stop_game

    if stop_game:
        messagebox.showinfo("Game Over", "Please restart the
game to change AI mode.")
        return

    if ai_mode == "Dumb":
        ai_mode = "Smart"
    else:
        ai_mode = "Dumb"
```

```
    switch_button.config(text=f"AI Mode: {ai_mode}")

def restart_game():
    global stop_game, Player1, states
    stop_game = False
    Player1 = 'X'
    for i in range(3):
        for j in range(3):
            states[i][j] = 0
            b[i][j].config(text="")
    switch_button.config(state=NORMAL)

# Design window
root = Tk()
root.title("Tic Tac Toe - Player vs AI")
root.resizable(0, 0)

b = [[0 for _ in range(3)] for _ in range(3)]
states = [[0 for _ in range(3)] for _ in range(3)]

for i in range(3):
    for j in range(3):
        b[i][j] = Button(root,
                         height=4, width=8,
                         font=("Helvetica", "20"),
                         command=lambda r=i, c=j: clicked(r, c))
        b[i][j].grid(row=i, column=j)

switch_button = Button(root, text=f"AI Mode: {ai_mode}",
command=switch_ai_mode)
switch_button.grid(row=3, column=0, columnspan=2, sticky="we")

restart_button = Button(root, text="Restart Game",
command=restart_game)
restart_button.grid(row=3, column=2, sticky="we")
```

```
root.mainloop()
```

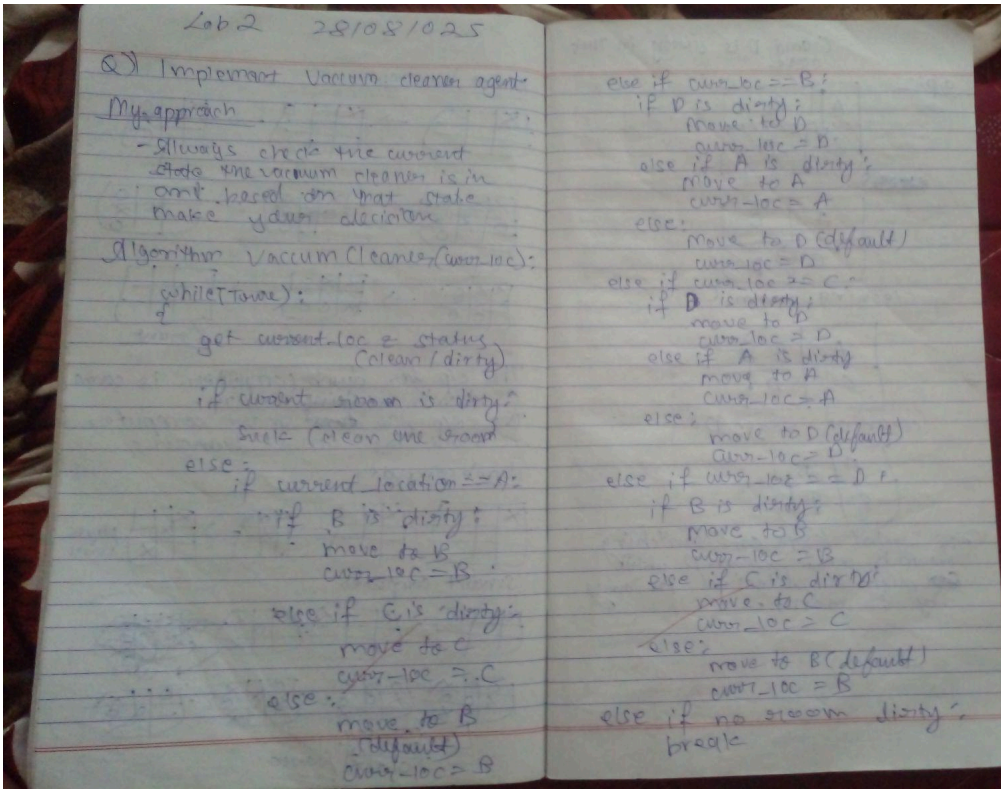
Output:




Tic Tac Toe - Player vs AI
X
X
O
O
O
X
X
O
X
AI Mode: Smart
Restart Game
Tie
It's a tie!
OK

2.SOLVE 8 PUZZLE PROBLEM

PseudoCode:



Code:

```
from collections import deque

# Goal State
GOAL_STATE = ((1, 2, 3), (4, 5, 6), (7, 8, 0))

# Start State from your image
START_STATE = ((5, 0, 2), (4, 6, 3), (1, 7, 8))

def get_neighbors(state):
    """
    Generates all valid neighboring states by moving the blank space (0).
    """
    neighbors = []

    # Find the position of 0 (blank)
    # r = row, c = col
    for r in range(3):
```

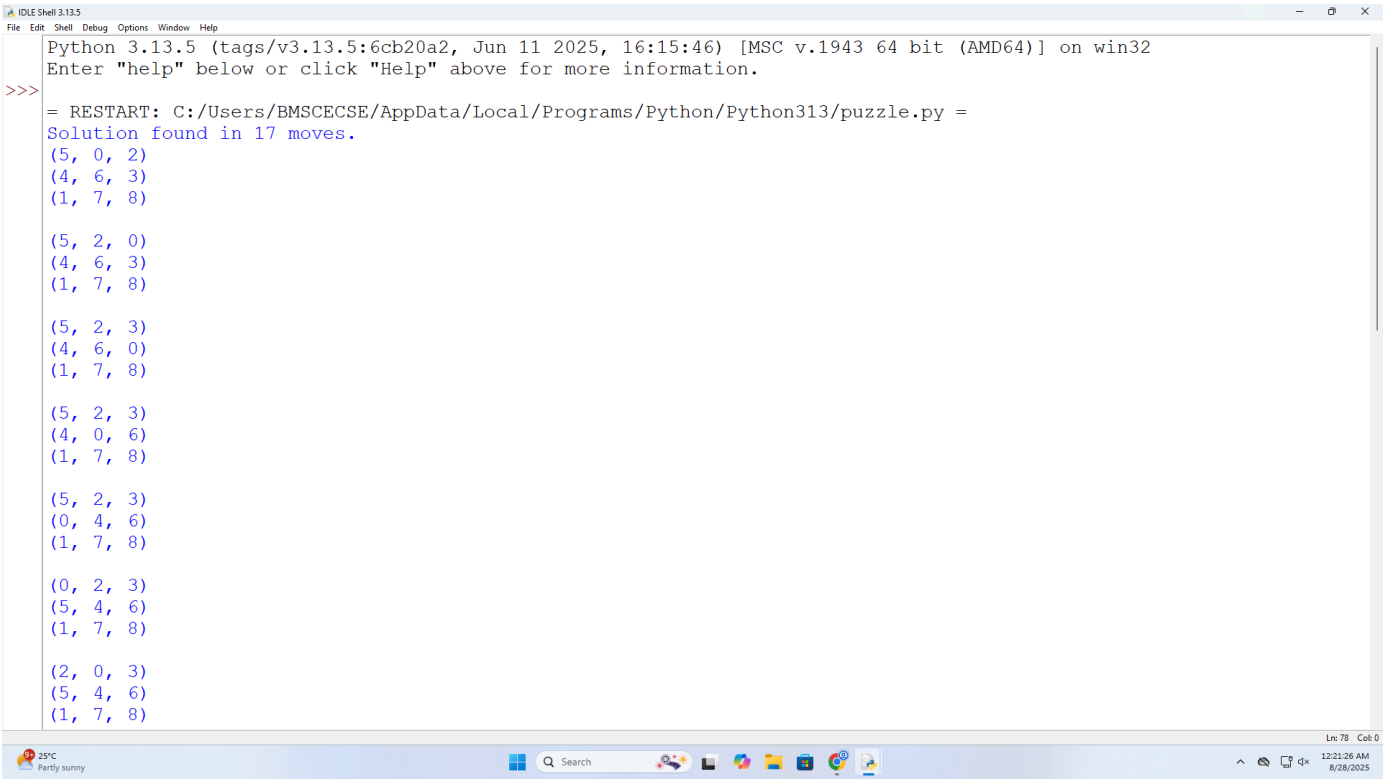
```
        for c in range(3):
            if state[r][c] == 0:
                zero_row, zero_col = r, c
                break

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

    for dr, dc in directions:
        new_r, new_c = zero_row + dr, zero_col + dc

        # Check bounds
        if 0 <= new_r < 3 and 0 <= new_c < 3:
            # Create a mutable copy of the board (list of lists)
            new_board = [list(row) for row in state]

            # Swap 0 with the target cell
            new_board[zero_row][zero_col],
new_board[new_r][new_c] = \
                new_board[new_r][new_c],
new_board[zero_row][zero_col]

            # Convert back to tuple of tuples for
immutability/hashing
            neighbors.append(tuple(tuple(row) for row in
new_board))

    return neighbors

def reconstruct_path(meet_node, parent_fwd, parent_bwd):
    """
    Joins the path from start->meet_node and meet_node->goal.
    """
    # 1. Trace path from Start -> Meeting Node
    path_fwd = []
    curr = meet_node
```

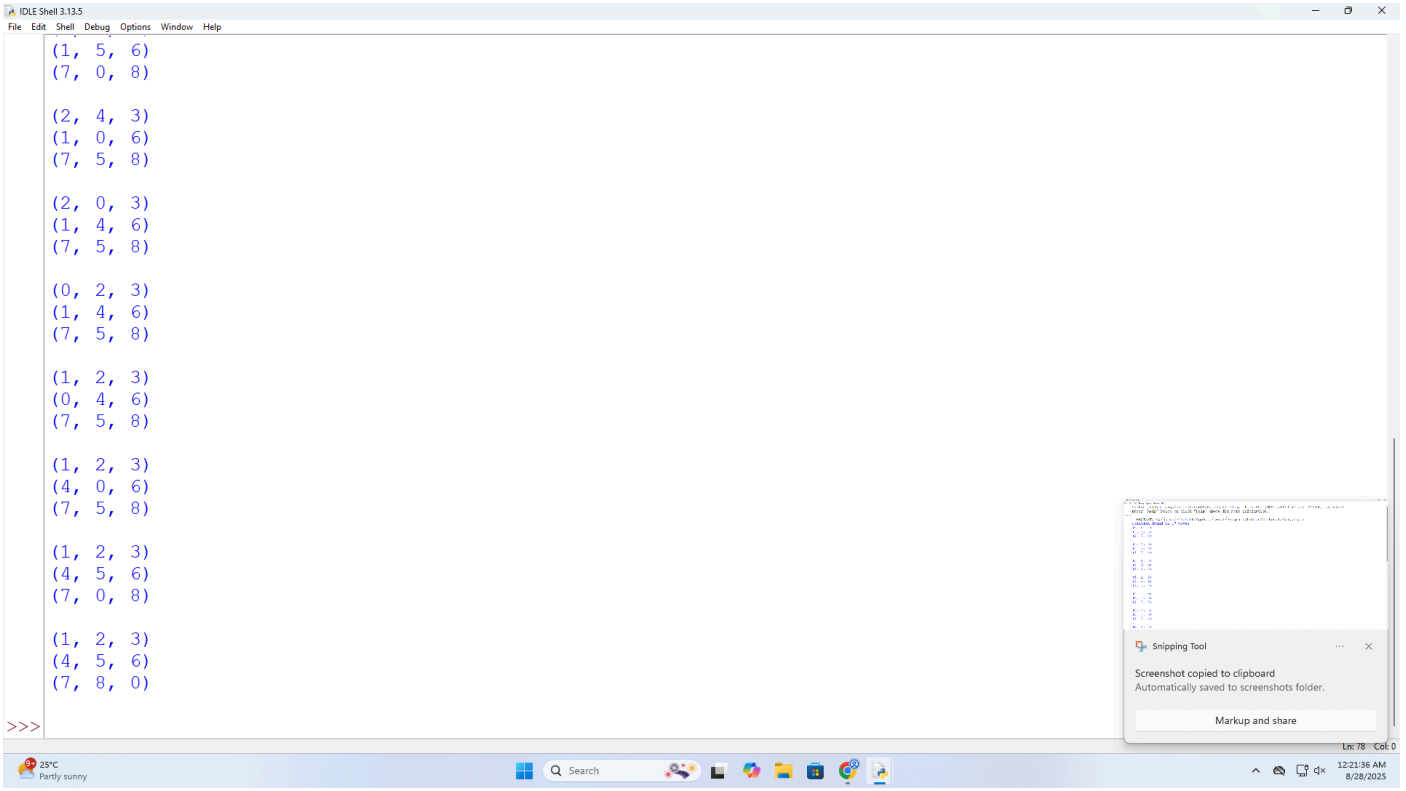
```
    while curr is not None:
        path_fwd.append(curr)
        curr = parent_fwd[curr]
    path_fwd.reverse() # Start is at the end, so reverse to get
Start -> Meet

    # 2. Trace path from Meeting Node -> Goal
    path_bwd = []
    curr = parent_bwd[meet_node] # Start from parent to avoid
duplicating meet_node
    while curr is not None:
        path_bwd.append(curr)
        curr = parent_bwd[curr]

    # Combine paths
    return path_fwd + path_bwd

def bidirectional_search(start, goal):
    """
    Performs Bidirectional BFS to find the shortest path.
    """
    # Queues for BFS
    queue_fwd = deque([start])
    queue_bwd = deque([goal])

    # Dictionaries to store visited nodes and their parents
(Child -> Parent)
    # This helps in path reconstruction and visited checking
    parent_fwd = {start: None}
    parent_bwd = {goal: None}

    while queue_fwd and queue_bwd:
        # --- Forward Search Step ---
        if queue_fwd:
            current_fwd = queue_fwd.popleft()
```

```
            # Check if we met the backward search
            if current_fwd in parent_bwd:
                return reconstruct_path(current_fwd, parent_fwd,
parent_bwd)

            for neighbor in get_neighbors(current_fwd):
                if neighbor not in parent_fwd:
                    parent_fwd[neighbor] = current_fwd
                    queue_fwd.append(neighbor)

        # --- Backward Search Step ---
        if queue_bwd:
            current_bwd = queue_bwd.popleft()

            # Check if we met the forward search
            if current_bwd in parent_fwd:
                return reconstruct_path(current_bwd, parent_fwd,
parent_bwd)

            for neighbor in get_neighbors(current_bwd):
                if neighbor not in parent_bwd:
                    parent_bwd[neighbor] = current_bwd
                    queue_bwd.append(neighbor)

    return None # No solution found

def print_solution(path):
    if not path:
        print("No solution found.")
        return

    # The number of moves is steps - 1 (since path includes
start state)
    print(f"Solution found in {len(path) - 1} moves.")

    for state in path:
```

```
        for row in state:
            print(row)
        print() # Empty line between states for readability

if __name__ == "__main__":
    solution_path = bidirectional_search(START_STATE,
GOAL_STATE)
    print_solution(solution_path)
```

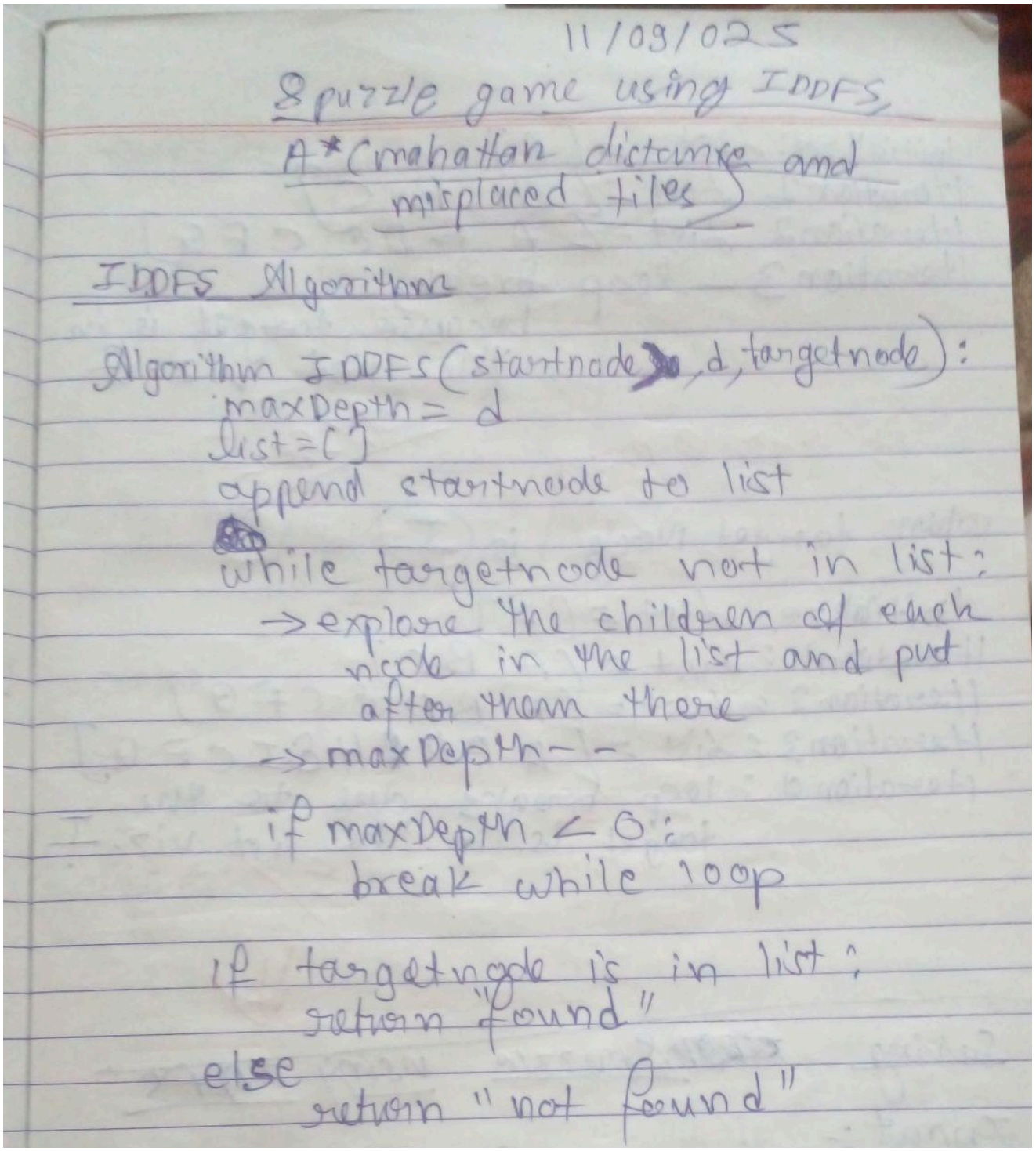
Output:


IDLE Shell 3.13.5
File Edit Shell Debug Options Window Help
Python 3.13.5 (tags/v3.13.5:6cb20a2, Jun 11 2025, 16:15:46) [MSC v.1943 64 bit (AMD64)] on win32
Enter "help" below or click "Help" above for more information.
>>>
= RESTART: C:/Users/BMSCECSE/AppData/Local/Programs/Python/Python313/puzzle.py =
Solution found in 17 moves.
(5, 0, 2)
(4, 6, 3)
(1, 7, 8)
(5, 2, 0)
(4, 6, 3)
(1, 7, 8)
(5, 2, 3)
(4, 6, 0)
(1, 7, 8)
(5, 2, 3)
(4, 0, 6)
(1, 7, 8)
(5, 2, 3)
(0, 4, 6)
(1, 7, 8)
(0, 2, 3)
(5, 4, 6)
(1, 7, 8)
(2, 0, 3)
(5, 4, 6)
(1, 7, 8)
Ln: 78 Col: 0
25°C
Partly sunny
Search
12:21:26 AM
8/28/2025


IDLE Shell 3.13.5
File Edit Shell Debug Options Window Help
(1, 5, 6)
(7, 0, 8)
(2, 4, 3)
(1, 0, 6)
(7, 5, 8)
(2, 0, 3)
(1, 4, 6)
(7, 5, 8)
(0, 2, 3)
(1, 4, 6)
(7, 5, 8)
(1, 2, 3)
(0, 4, 6)
(7, 5, 8)
(1, 2, 3)
(4, 0, 6)
(7, 5, 8)
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
>>>
Snipping Tool
Screenshot copied to clipboard
Automatically saved to screenshots folder.
Markup and share
Ln: 78 Col: 0
25°C
Partly sunny
Search
12:21:36 AM
8/28/2025

3.Implement Iterative deepening search algorithm

PseudoCode:



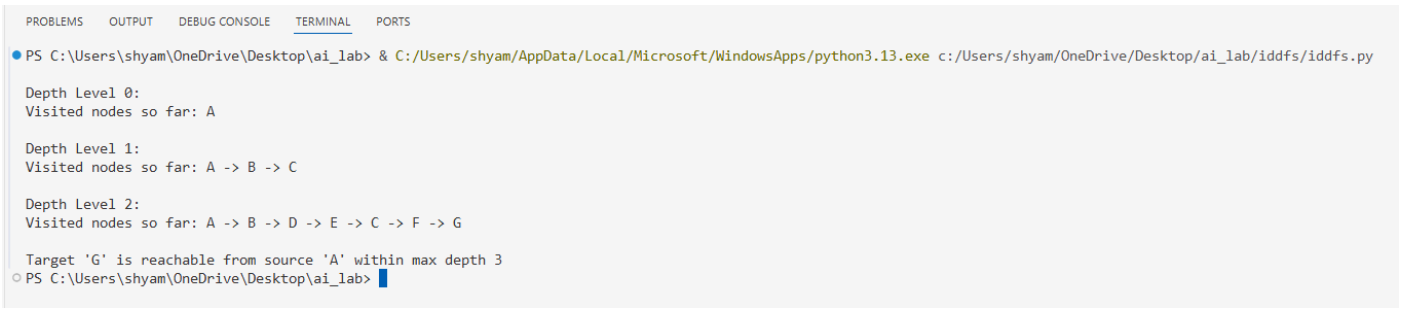
Code:

```
from collections import defaultdict

class Graph:

    def __init__(self, vertices):
```

```
        self.V = vertices
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DLS(self, src, target, maxDepth, depth=0, visited=None):
        if visited is None:
            visited = []

        visited.append(src)  # record visited node

        if src == target:
            return True, visited

        if maxDepth <= 0:
            return False, visited

        for i in self.graph[src]:
            found, visited = self.DLS(i, target, maxDepth - 1,
depth + 1, visited)
            if found:
                return True, visited

        return False, visited

    def IDDFS(self, src, target, maxDepth):
        for i in range(maxDepth + 1):  # maxDepth included
            print(f"\nDepth Level {i}:")
            found, visited = self.DLS(src, target, i)
            print("Visited nodes so far:", " -> ".join(visited))
            if found:
                return True
        return False
```

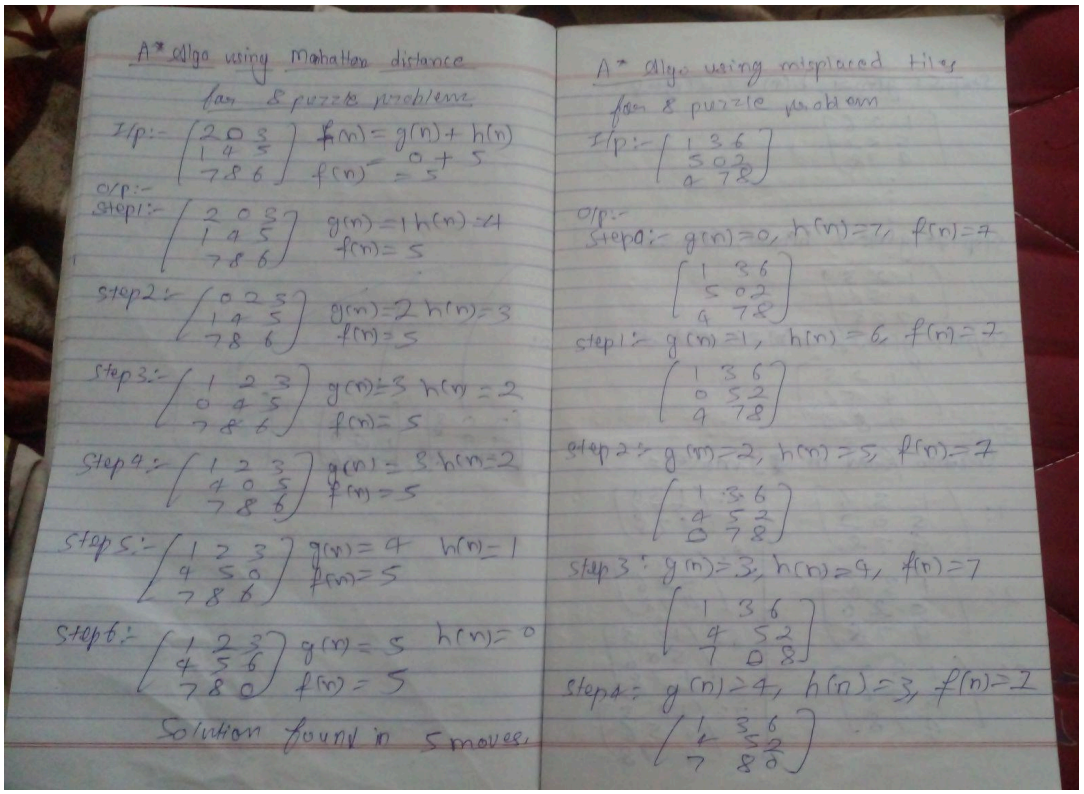
```
# Create a graph with alphabet nodes
g = Graph(7)

g.addEdge('A', 'B')
g.addEdge('A', 'C')
g.addEdge('B', 'D')
g.addEdge('B', 'E')
g.addEdge('C', 'F')
g.addEdge('C', 'G')

target = 'G'
maxDepth = 3
src = 'A'

if g.IDDFS(src, target, maxDepth):
    print(f"\nTarget '{target}' is reachable from source '{src}'
within max depth {maxDepth}")
else:
    print(f"\nTarget '{target}' is NOT reachable from source
'{src}' within max depth {maxDepth}")
```

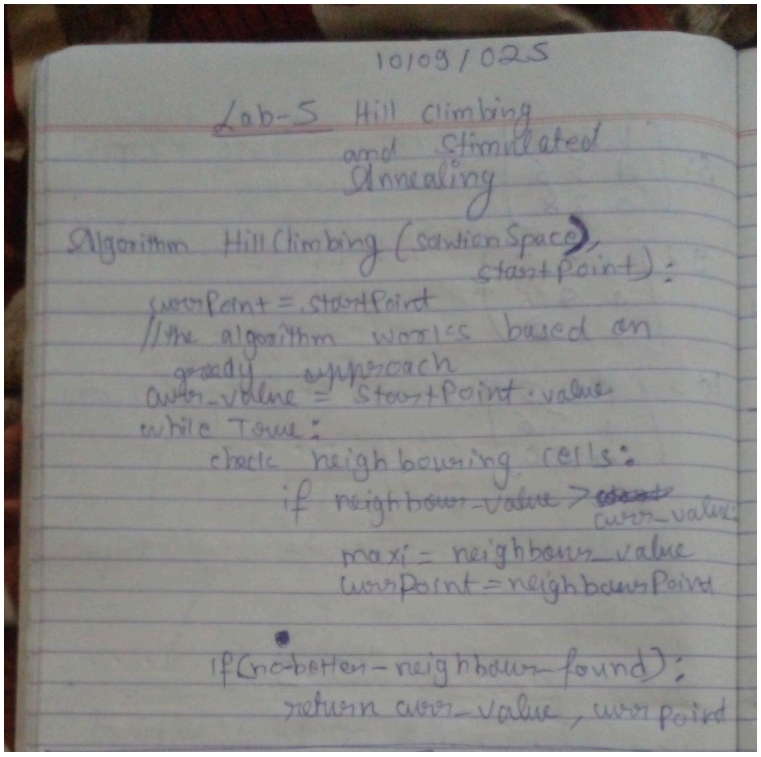
Output:



4.Implement A* search algorithm.Implement Hill Climbing Algorithm

PseudoCode:





Code:

```
# Python3 implementation of the
# above approach
from random import randint
```

```
N = 4

# A utility function that configures
# the 2D array "board" and
# array "state" randomly to provide
# a starting point for the algorithm.
def configureRandomly(board, state):

    # Iterating through the
    # column indices
    for i in range(N):

        # Getting a random row index
        state[i] = randint(0, 100000) % N;

        # Placing a queen on the
        # obtained place in
        # chessboard.
        board[state[i]][i] = 1;

# A utility function that prints
# the 2D array "board".
def printBoard(board):

    for i in range(N):
        print(*board[i])

# A utility function that prints
# the array "state".
def printState( state):
    print(*state)

# A utility function that compares
# two arrays, state1 and state2 and
# returns True if equal
```

```
# and False otherwise.
def compareStates(state1, state2):


    for i in range(N):
        if (state1[i] != state2[i]):
            return False;

    return True;

# A utility function that fills
# the 2D array "board" with
# values "value"
def fill(board, value):

    for i in range(N):
        for j in range(N):
            board[i][j] = value;

# This function calculates the
# objective value of the
# state(queens attacking each other)
# using the board by the
# following logic.
def calculateObjective( board, state):

    # For each queen in a column, we check
    # for other queens falling in the line
    # of our current queen and if found,
    # any, then we increment the variable
    # attacking count.

    # Number of queens attacking each other,
    # initially zero.
    attacking = 0;
```

```
# Variables to index a particular
# row and column on board.
for i in range(N):

    # At each column 'i', the queen is
    # placed at row 'state[i]', by the
    # definition of our state.

    # To the left of same row
    # (row remains constant
    # and col decreases)
    row = state[i]
    col = i - 1;
    while (col >= 0 and board[row][col] != 1) :
        col -= 1

    if (col >= 0 and board[row][col] == 1) :
        attacking += 1;

    # To the right of same row
    # (row remains constant
    # and col increases)
    row = state[i]
    col = i + 1;
    while (col < N and board[row][col] != 1):
        col += 1;

    if (col < N and board[row][col] == 1) :
        attacking += 1;

    # Diagonally to the left up
    # (row and col simultaneously
    # decrease)
    row = state[i] - 1
    col = i - 1;
    while (col >= 0 and row >= 0 and board[row][col] != 1) :
```

```
        col-= 1;
        row-= 1;

    if (col >= 0 and row >= 0  and board[row][col] == 1) :
        attacking+= 1;

    # Diagonally to the right down
    # (row and col simultaneously
    # increase)
    row = state[i] + 1
    col = i + 1;
    while (col < N and row < N  and board[row][col] != 1) :
        col+= 1;
        row+= 1;

    if (col < N and row < N and board[row][col] == 1) :
        attacking += 1;

    # Diagonally to the left down
    # (col decreases and row
    # increases)
    row = state[i] + 1
    col = i - 1;
    while (col >= 0 and row < N  and board[row][col] != 1) :
        col -= 1;
        row += 1;

    if (col >= 0 and row < N and board[row][col] == 1) :
        attacking += 1;

    # Diagonally to the right up
    # (col increases and row
    # decreases)
    row = state[i] - 1
    col = i + 1;
    while (col < N and row >= 0  and board[row][col] != 1) :
```

```
            col += 1;
            row -= 1;

        if (col < N and row >= 0 and board[row][col] == 1) :
            attacking += 1;

    # Return pairs.
    return int(attacking / 2);

# A utility function that
# generates a board configuration
# given the state.
def generateBoard( board, state):
    fill(board, 0);
    for i in range(N):
        board[state[i]][i] = 1;

# A utility function that copies
# contents of state2 to state1.
def copyState( state1, state2):

    for i in range(N):
        state1[i] = state2[i];

# This function gets the neighbour
# of the current state having
# the least objective value
# amongst all neighbours as
# well as the current state.
def getNeighbour(board, state):

    # Declaring and initializing the
    # optimal board and state with
    # the current board and the state
    # as the starting point.
    opBoard = [[0 for _ in range(N)] for _ in range(N)]
```

```
opState = [0 for _ in range(N)]

copyState(opState, state);
generateBoard(opBoard, opState);

# Initializing the optimal
# objective value
opObjective  = calculateObjective(opBoard, opState);

# Declaring and initializing
# the temporary board and
# state for the purpose
# of computation.
NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]

NeighbourState = [0 for _ in range(N)]
copyState(NeighbourState, state);
generateBoard(NeighbourBoard, NeighbourState);

# Iterating through all
# possible neighbours
# of the board.
for i in range(N):
    for j in range(N):

        # Condition for skipping the
        # current state
        if (j != state[i]) :

            # Initializing temporary
            # neighbour with the
            # current neighbour.
            NeighbourState[i] = j;
            NeighbourBoard[NeighbourState[i]][i] = 1;
            NeighbourBoard[state[i]][i] = 0;
```

```
                # Calculating the objective
                # value of the neighbour.
                temp = calculateObjective( NeighbourBoard,
NeighbourState);

                # Comparing temporary and optimal
                # neighbour objectives and if
                # temporary is less than optimal
                # then updating accordingly.

                if (temp <= opObjective) :
                    opObjective = temp;
                    copyState(opState, NeighbourState);
                    generateBoard(opBoard, opState);

                # Going back to the original
                # configuration for the next
                # iteration.
                NeighbourBoard[NeighbourState[i]][i] = 0;
                NeighbourState[i] = state[i];
                NeighbourBoard[state[i]][i] = 1;

    # Copying the optimal board and
    # state thus found to the current
    # board and, state since c+= 1 doesn't
    # allow returning multiple values.
    copyState(state, opState);
    fill(board, 0);
    generateBoard(board, state);

def hillClimbing(board, state):

    # Declaring  and initializing the
    # neighbour board and state with
    # the current board and the state
    # as the starting point.
```

```
    neighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
    neighbourState = [0 for _ in range(N)]

    copyState(neighbourState, state);
    generateBoard(neighbourBoard, neighbourState);

    while True:

        # Copying the neighbour board and
        # state to the current board and
        # state, since a neighbour
        # becomes current after the jump.

        copyState(state, neighbourState);
        generateBoard(board, state);

        # Getting the optimal neighbour

        getNeighbour(neighbourBoard, neighbourState);

        if (compareStates(state, neighbourState)) :

            # If neighbour and current are
            # equal then no optimal neighbour
            # exists and therefore output the
            # result and break the loop.

            printBoard(board);
            break;

        elif (calculateObjective(board, state) ==
calculateObjective( neighbourBoard,neighbourState)):

            # If neighbour and current are
            # not equal but their objectives
```

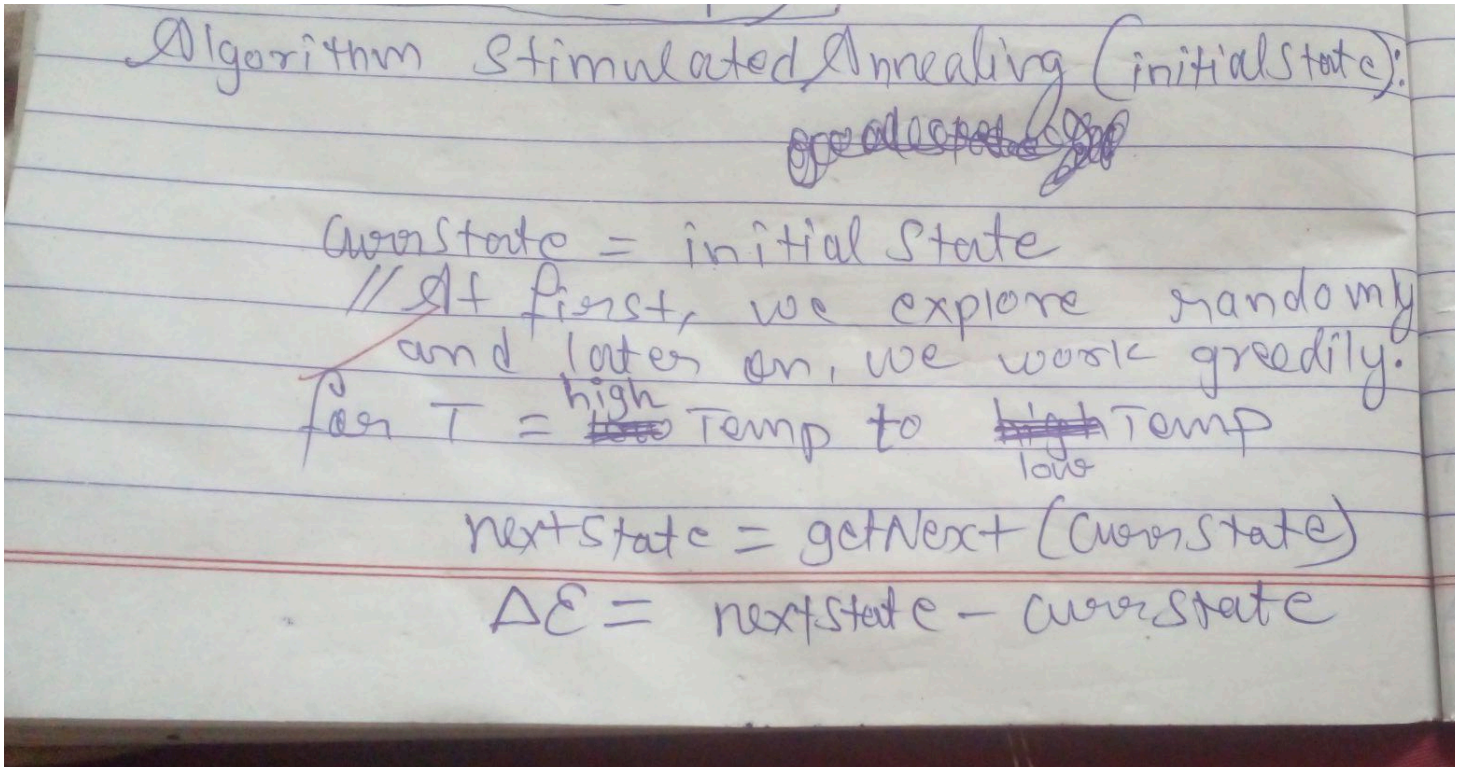
```
            # are equal then we are either
            # approaching a shoulder or a
            # local optimum, in any case,
            # jump to a random neighbour
            # to escape it.

            # Random neighbour
            neighbourState[randint(0, 100000) % N]  = randint(0,
100000) % N;
            generateBoard(neighbourBoard, neighbourState);

# Driver code
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]

# Getting a starting point by
# randomly configuring the board
configureRandomly(board, state);

# Do hill climbing on the
# board obtained
hillClimbing(board, state);

# This code is contributed by phasing17.
```

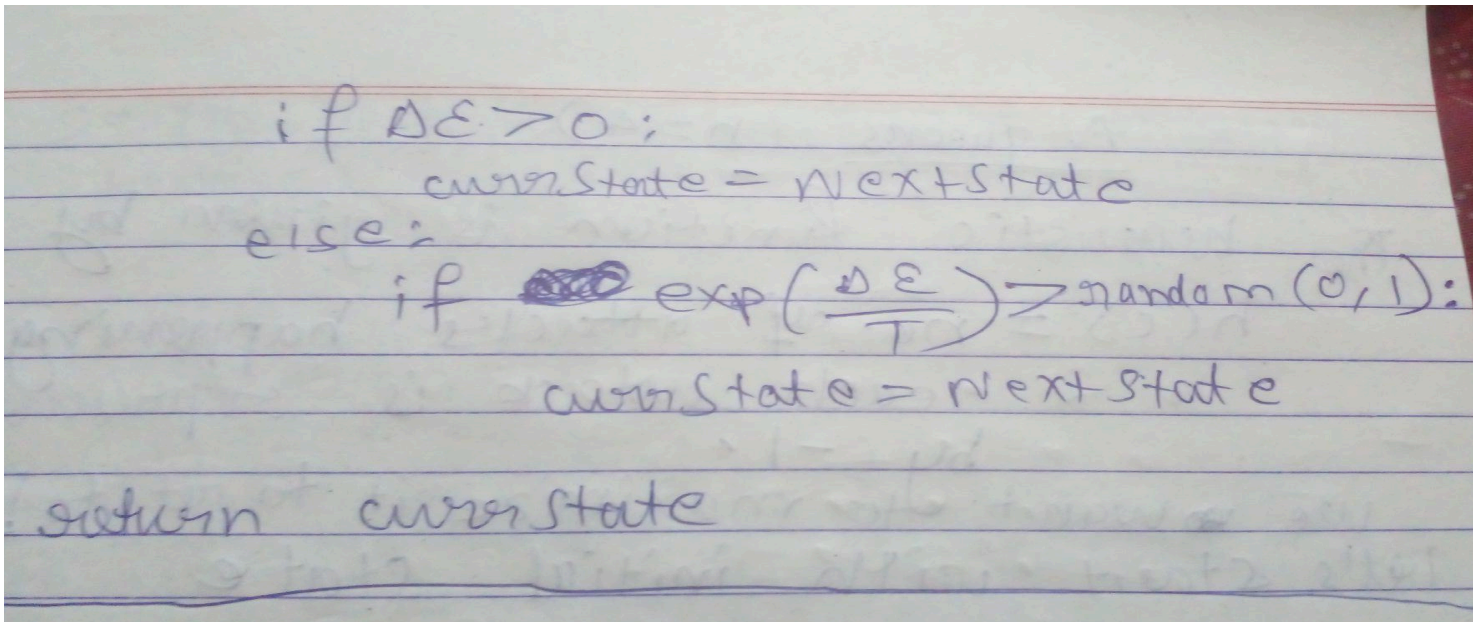
Output:

```
PS C:\Users\shyam\OneDrive\Desktop\ai_lab> & C:/Users/shyam/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/shyam/OneDrive/Desktop/ai_lab/hill climbing/n_queen
using_hill_climbing.py"
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
PS C:\Users\shyam\OneDrive\Desktop\ai_lab>
```

5.Write a program to implement Simulated Annealing Algorithm

PseudoCode:





Code:

```
import random
import math
def calculate_conflicts(board):
    """Calculates the number of conflicts (attacking pairs of
queens) on the board."""
```

```
    n = len(board)
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            # Check for same row (not possible with current
representation)
            # Check for same column
            if board[i] == board[j]:
                conflicts += 1
            # Check for diagonals
            if abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def get_random_neighbor(board):
    """Generates a random neighbor state by moving one queen to
a different row within its column."""
    n = len(board)
    neighbor = list(board)  # Create a copy to avoid modifying
the original
    col_to_move = random.randint(0, n - 1)
    new_row = random.randint(0, n - 1)
    while new_row == neighbor[col_to_move]:  # Ensure the queen
actually moves
        new_row = random.randint(0, n - 1)
    neighbor[col_to_move] = new_row
    return neighbor

def simulated_annealing_n_queens(n, initial_temperature=1000,
cooling_rate=0.99, min_temperature=1):
    """Solves the N-Queens problem using Simulated Annealing."""
    # Initial state: queens placed randomly in each column
    current_board = [random.randint(0, n - 1) for _ in range(n)]
    current_conflicts = calculate_conflicts(current_board)

    best_board = list(current_board)
```

```
    best_conflicts = current_conflicts

    temperature = initial_temperature

    while temperature > min_temperature and best_conflicts > 0:
        neighbor_board = get_random_neighbor(current_board)
        neighbor_conflicts = calculate_conflicts(neighbor_board)

        delta_e = neighbor_conflicts - current_conflicts

        if delta_e < 0:  # If neighbor is better, accept it
            current_board = neighbor_board
            current_conflicts = neighbor_conflicts
            if current_conflicts < best_conflicts:
                best_conflicts = current_conflicts
                best_board = list(current_board)
        else:  # If neighbor is worse, accept with a probability
            probability = math.exp(-delta_e / temperature)
            if random.random() < probability:
                current_board = neighbor_board
                current_conflicts = neighbor_conflicts

        temperature *= cooling_rate

    return best_board, best_conflicts

# Example usage:
if __name__ == "__main__":
    n_queens = 4  # Number of queens (and board size)
    solution_board, conflicts =
simulated_annealing_n_queens(n_queens)

    print(f"N-Queens problem for N={n_queens}")
    if conflicts == 0:
        print("Solution found:")
        for row in solution_board:
```

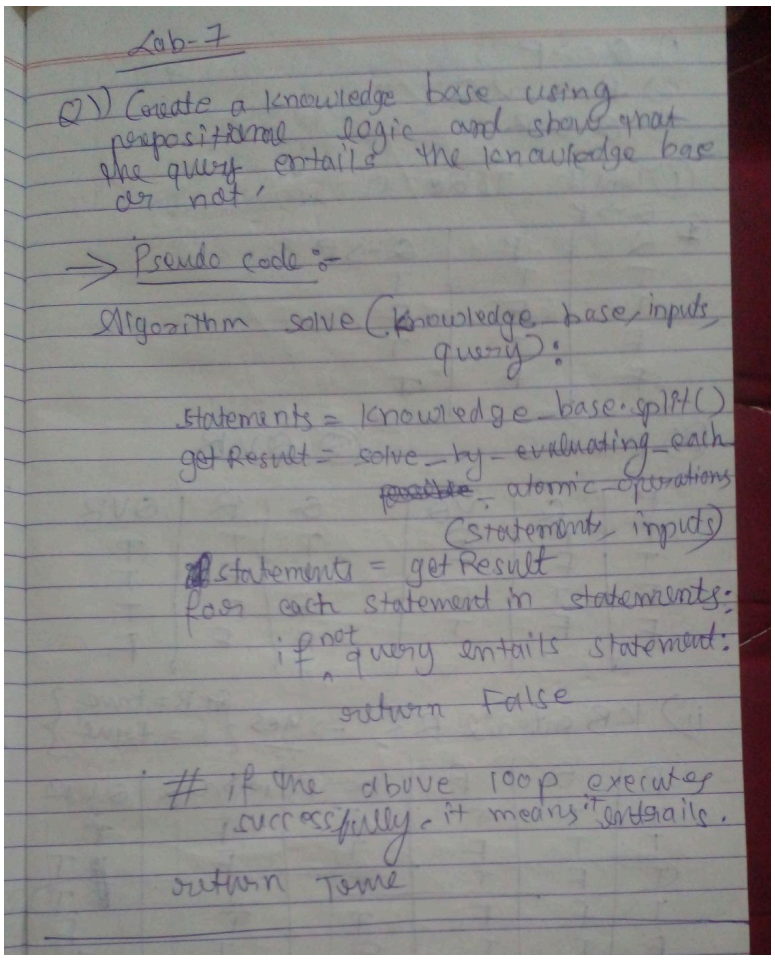
```
            print(" ".join(["Q" if i == row else "." for i in
range(n_queens)]))
    else:
        print(f"Could not find a perfect solution. Best found
has {conflicts} conflicts.")
        for row in solution_board:
            print(" ".join(["Q" if i == row else "." for i in
range(n_queens)]))
```

Output:

```
PS C:\Users\shyam\OneDrive\Desktop\ai_lab> & C:/Users/shyam/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c
ueens_using_simulated_annealing.py"
N-Queens problem for N=4
Solution found:
. Q . .
. . . Q
Q . . .
. . Q .
PS C:\Users\shyam\OneDrive\Desktop\ai_lab>
```

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

PseudoCode:


Lab-7

Q1) Create a knowledge base using propositional logic and show that the query entails the knowledge base or not.

→ Pseudo code :-

Algorithm solve (knowledge_base, inputs, query):

Statements = knowledge_base.split()
getResult = solve_by_evaluating_each atomic_operations (statement, inputs)
statements = getResult
for each statement in statements:
if not query entails statement:
return False

if the above loop executes successfully, it means it entails.
return True

Code:

```
p=[True]*4+[False]*4
q=[True]*2+[False]*2+[True]*2+[False]*2
r=[True,False,True,False,True,False,True,False]
def imply(a,b):
    if a and not b:
        return False
    return True
def Or(a,b):
    return a or b
def And(a,b,c):
    return a and b and c
q_imply_p=[imply(q[i],p[i]) for i in range(len(p))]
p_implynot_q=[imply(p[i],not q[i]) for i in range(len(p))]
```

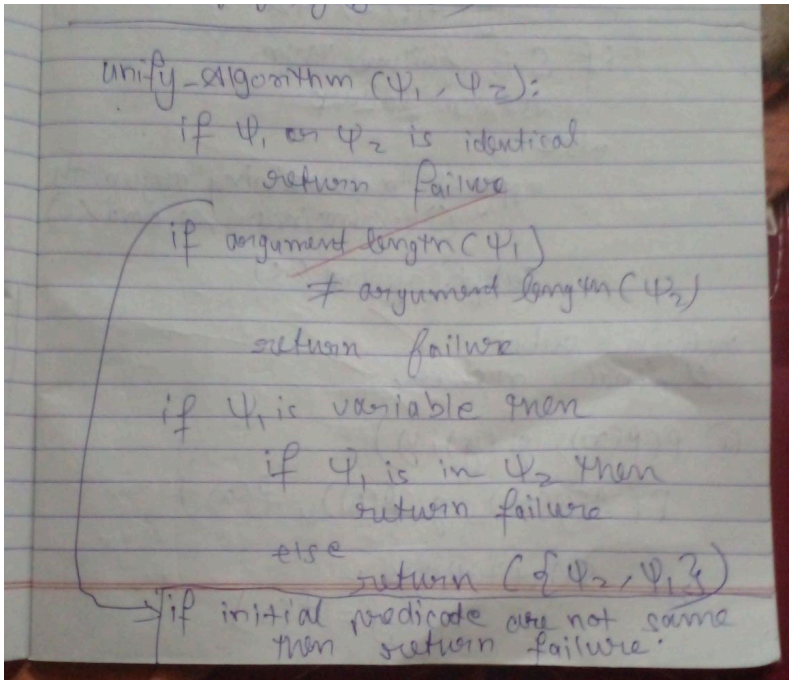
```
q_or_r=[Or(q[i],r[i]) for i in range(len(p))]
r_imply_p=[imply(r[i],p[i]) for i in range(len(p))]
q_imply_r=[imply(q[i],r[i]) for i in range(len(p))]
kb=[And(q_imply_p[i],p_implynot_q[i],q_or_r[i]) for i in
range(len(p))]
kb_reduced=[i for i,j in enumerate(kb) if j]
r_entails=[True for i in kb_reduced if r[i]]
r_imply_p_entails=[True for i in kb_reduced if r_imply_p[i]]
q_imply_r_entails=[True for i in kb_reduced if q_imply_r[i]]
print("q_imply_p")
print(q_imply_p)
print("p_implynot_q")
print(p_implynot_q)
print("q_or_r")
print(q_or_r)
print("r_imply_p")
print(r_imply_p)
print("q_imply_r")
print(q_imply_r)
print("r")
print(r)
print("kb")
print(kb)
print("r entails" if r_entails.count(True)==len(kb_reduced) else
"r doesnot entail")
print("r_imply_p_entails" if
r_imply_p_entails.count(True)==len(kb_reduced) else "r_imply_p
doesnot entail")
print("q_imply_r_entails" if
q_imply_r_entails.count(True)==len(kb_reduced) else "q_imply_r
doesnot entail")
```

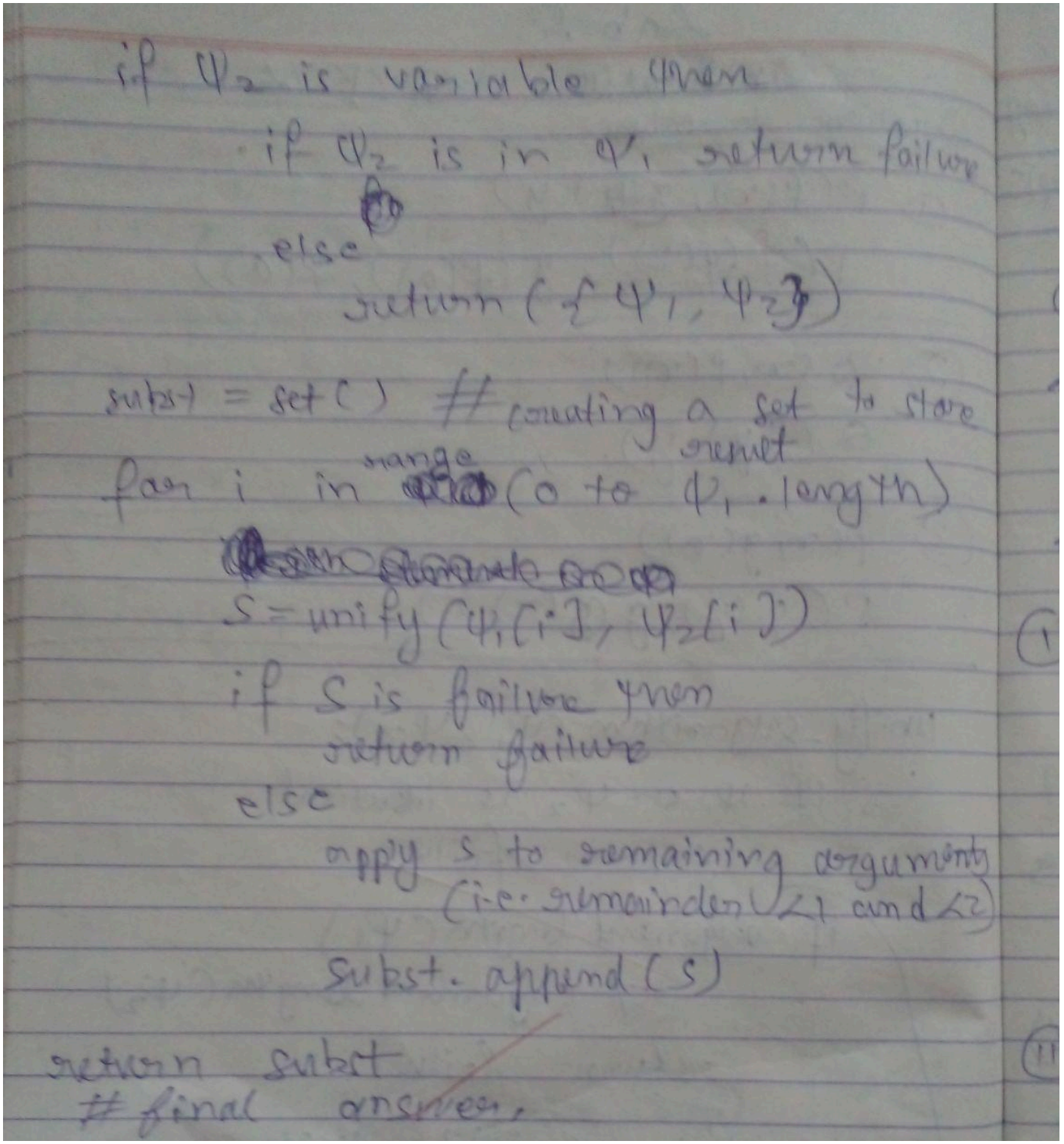
Output:

```
PS C:\Users\shyam\OneDrive\Desktop\ai_lab> & C:/Users/shyam/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/shyam/OneDrive/Desktop/ai_lab/knowledge base using
epositional logic/knowledge_base.py"
q_imply_p
[True, True, True, True, False, False, True, True]
p_implynot_q
[False, False, True, True, True, True, True, True]
q_or_r
[True, True, True, False, True, True, True, False]
r_imply_p
[True, True, True, True, False, True, False, True]
q_imply_r
[True, False, True, True, True, False, True, True]
r
[True, False, True, False, True, False, True, False]
kb
[False, False, True, False, False, False, True, False]
r entails
r_imply_p doesnot entail
q_imply_r_entails
PS C:\Users\shyam\OneDrive\Desktop\ai_lab>
```

7.Implement unification in first order logic

PseudoCode:





Code:

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

    def __eq__(self, other):
```

```
        return isinstance(other, Variable) and self.name ==
other.name

    def __hash__(self):
        return hash(self.name)

    def __repr__(self):
        return self.name

class Constant:
    def __init__(self, value):
        self.value = value

    def __eq__(self, other):
        return isinstance(other, Constant) and self.value ==
other.value

    def __hash__(self):
        return hash(self.value)

    def __repr__(self):
        return str(self.value)

class Function:
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __eq__(self, other):
        return (isinstance(other, Function) and
                self.name == other.name and
                len(self.args) == len(other.args) and
                all(a == b for a, b in zip(self.args,
other.args)))

    def __hash__(self):
```

```
        return hash((self.name, tuple(self.args)))

    def __repr__(self):
        return f"{self.name}({', '.join(map(str, self.args))})"

def unify(term1, term2, substitution=None):
    """
    Unifies two first-order logic terms and returns the MGU
(substitution)
    or None if unification is not possible.
    """
    if substitution is None:
        substitution = {}

    # Apply existing substitutions
    term1 = substitute(term1, substitution)
    term2 = substitute(term2, substitution)

    if term1 == term2:
        return substitution
    elif isinstance(term1, Variable):
        return unify_var(term1, term2, substitution)
    elif isinstance(term2, Variable):
        return unify_var(term2, term1, substitution)
    elif isinstance(term1, Function) and isinstance(term2,
Function):
        if term1.name != term2.name or len(term1.args) !=
len(term2.args):
            return None  # Function symbols or arity don't match
        for arg1, arg2 in zip(term1.args, term2.args):
            substitution = unify(arg1, arg2, substitution)
            if substitution is None:
                return None  # Sub-unification failed
        return substitution
    else:
```

```
        return None  # Cannot unify different types (e.g.,
Constant and Function)

def unify_var(var, x, substitution):
    """Handles unification when one of the terms is a
variable."""
    if var in substitution:
        return unify(substitution[var], x, substitution)
    elif x in substitution:
        return unify(var, substitution[x], substitution)
    elif occurs_check(var, x, substitution):
        return None  # Occurs check fails
    else:
        substitution[var] = x
        return substitution

def occurs_check(var, term, substitution):
    """Checks if a variable occurs within a term, preventing
infinite substitutions."""
    term = substitute(term, substitution)  # Apply current
substitutions
    if var == term:
        return True
    elif isinstance(term, Function):
        return any(occurs_check(var, arg, substitution) for arg
in term.args)
    return False

def substitute(term, substitution):
    """Applies a given substitution to a term."""
    if isinstance(term, Variable):
        return substitution.get(term, term)
    elif isinstance(term, Function):
        return Function(term.name, [substitute(arg,
substitution) for arg in term.args])
    return term
```

```
# Example Usage:
if __name__ == "__main__":
    # Define terms
    x, y, z = Variable('x'), Variable('y'), Variable('z')
    a, b = Constant('a'), Constant('b')
    f = Function('f', [x, Constant('b')])
    g = Function('g', [Constant('a'), y])
    h = Function('h', [z])

    # Test cases
    print(f"Unify(f(x, b), f(a, y)): {unify(Function('f', [x,
b]), Function('f', [a, y]))}")
    print(f"Unify(g(a, y), g(a, b)): {unify(Function('g', [a,
y]), Function('g', [a, b]))}")
    print(f"Unify(x, f(x, b)): {unify(x, Function('f', [x,
b]))}") # Occurs check failure
    print(f"Unify(f(x, y), f(a, g(z))): {unify(Function('f', [x,
y]), Function('f', [a, Function('g', [z]))])}")
    print(f"Unify(P(x, A), P(B, y)): {unify(Function('P', [x,
Constant('A')]), Function('P', [Constant('B'), y]))}")
```

Output:

Output

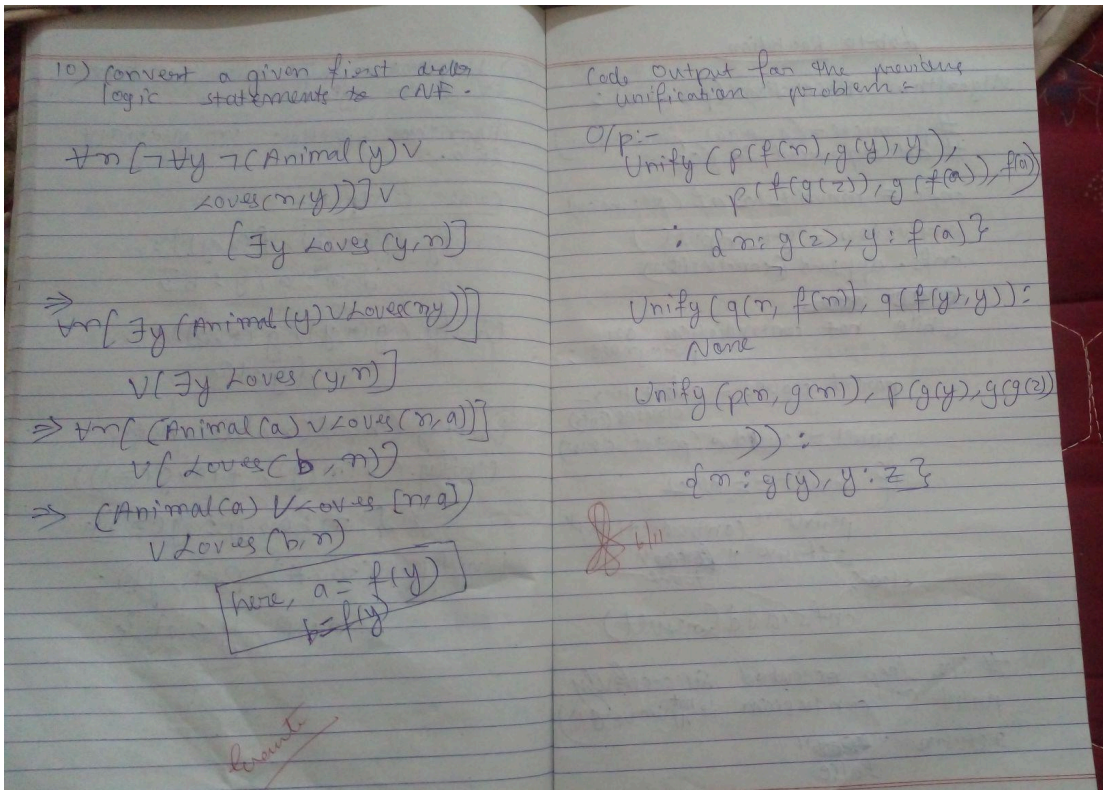
```
Unify(f(x, b), f(a, y)): {x: a, y: b}
Unify(g(a, y), g(a, b)): {y: b}
Unify(x, f(x, b)): None
Unify(f(x, y), f(a, g(z))): {x: a, y: g(z)}
Unify(P(x, A), P(B, y)): {x: B, y: A}

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

Output

```
Unify(f(x, b), f(a, y)): {x: a, y: b}
Unify(g(a, y), g(a, b)): {y: b}
Unify(x, f(x, b)): None
Unify(f(x, y), f(a, g(z))): {x: a, y: g(z)}
Unify(P(x, A), P(B, y)): {x: B, y: A}

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

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

PseudoCode:



Code:

```
# ----------------------------------------

# Convert First Order Logic (FOL) formula to CNF

# ----------------------------------------


import itertools


# --------- FOL Expression Classes ----------


class Var:

    def __init__(self, name):
```

```
        self.name = name
    def __str__(self): return self.name

class Const:
    def __init__(self, name):
        self.name = name
    def __str__(self): return self.name

class Func:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __str__(self): return f"{self.name}({','.join(map(str,self.args))})"

class Pred:
    def __init__(self, name, args):
        self.name = name
        self.args = args
    def __str__(self): return f"{self.name}({','.join(map(str,self.args))})"

# Logical connectives

class Not:
    def __init__(self, op):
        self.op = op
```

```
    def __str__(self): return f"¬{self.op}"

class And:
    def __init__(self, left, right):
        self.left, self.right = left, right
    def __str__(self): return f"({self.left} ∧ {self.right})"

class Or:
    def __init__(self, left, right):
        self.left, self.right = left, right
    def __str__(self): return f"({self.left} ∨ {self.right})"

class Impl:
    def __init__(self, left, right):
        self.left, self.right = left, right
    def __str__(self): return f"({self.left} → {self.right})"

class Iff:
    def __init__(self, left, right):
        self.left, self.right = left, right
    def __str__(self): return f"({self.left} ↔ {self.right})"

class ForAll:
    def __init__(self, var, body):
        self.var = var
```

```
        self.body = body
    def __str__(self): return f"∀{self.var}.{self.body}"

class Exists:
    def __init__(self, var, body):
        self.var = var
        self.body = body
    def __str__(self): return f"∃{self.var}.{self.body}"


# ------------------- CNF Conversion Steps -------------------

def eliminate_iff(formula):
    """Eliminate ↔ and →."""
    if isinstance(formula, Iff):
        A = eliminate_iff(formula.left)
        B = eliminate_iff(formula.right)
        return And(Impl(A, B), Impl(B, A))

    if isinstance(formula, Impl):
        A = eliminate_iff(formula.left)
        B = eliminate_iff(formula.right)
        return Or(Not(A), B)

    if isinstance(formula, (And, Or)):
```

```
        return type(formula)(eliminate_iff(formula.left),
                             eliminate_iff(formula.right))

    if isinstance(formula, Not):
        return Not(eliminate_iff(formula.op))

    if isinstance(formula, (ForAll, Exists)):
        return type(formula)(formula.var, eliminate_iff(formula.body))

    return formula


def push_negation(formula):
    """Push negations inward (De Morgan + quantifier switching)."""
    if isinstance(formula, Not):

        if isinstance(formula.op, Not):
            return push_negation(formula.op.op)

        if isinstance(formula.op, And):
            return Or(push_negation(Not(formula.op.left)),
                      push_negation(Not(formula.op.right)))

        if isinstance(formula.op, Or):
            return And(push_negation(Not(formula.op.left)),
```

```
                push_negation(Not(formula.op.right)))

        if isinstance(formula.op, ForAll):
            return Exists(formula.op.var,
                          push_negation(Not(formula.op.body)))

        if isinstance(formula.op, Exists):
            return ForAll(formula.op.var,
                          push_negation(Not(formula.op.body)))

    if isinstance(formula, (And, Or)):
        return type(formula)(push_negation(formula.left),
                             push_negation(formula.right))

    if isinstance(formula, (ForAll, Exists)):
        return type(formula)(formula.var,
                             push_negation(formula.body))

    return formula


skolem_counter = itertools.count()

def skolemize(formula, vars_in_scope=None):
    """Remove ∃ by replacing with Skolem functions."""
```

```
if vars_in_scope is None:
    vars_in_scope = []

if isinstance(formula, ForAll):
    return ForAll(formula.var,
                  skolemize(formula.body, vars_in_scope + [formula.var]))

if isinstance(formula, Exists):
    sk_name = f"Sk{next(skolem_counter)}"
    sk_term = Func(sk_name, vars_in_scope)
    return skolemize(substitute_var(formula.body,
                                    formula.var,
                                    sk_term),
                     vars_in_scope)

if isinstance(formula, (And, Or)):
    return type(formula)(skolemize(formula.left, vars_in_scope),
                         skolemize(formula.right, vars_in_scope))

if isinstance(formula, Not):
    return Not(skolemize(formula.op, vars_in_scope))

return formula
```

```
def substitute_var(formula, var, term):
    """Replace variable with term."""
    if isinstance(formula, Var):
        return term if formula.name == var.name else formula

    if isinstance(formula, Pred):
        return Pred(formula.name,
                    [substitute_var(a, var, term) for a in formula.args])

    if isinstance(formula, Func):
        return Func(formula.name,
                    [substitute_var(a, var, term) for a in formula.args])

    if isinstance(formula, (And, Or)):
        return type(formula)(substitute_var(formula.left, var, term),
                             substitute_var(formula.right, var, term))

    if isinstance(formula, Not):
        return Not(substitute_var(formula.op, var, term))

    if isinstance(formula, (ForAll, Exists)):
        return type(formula)(formula.var,
                             substitute_var(formula.body, var, term))

    return formula
```

```
def drop_universal(formula):
    """Remove universal quantifiers."""
    if isinstance(formula, ForAll):
        return drop_universal(formula.body)

    if isinstance(formula, (And, Or)):
        return type(formula)(drop_universal(formula.left),
                             drop_universal(formula.right))

    if isinstance(formula, Not):
        return Not(drop_universal(formula.op))

    return formula


def distribute_or(formula):
    """Apply distribution: (A ∨ (B ∧ C)) = (A ∨ B) ∧ (A ∨ C)."""
    if isinstance(formula, Or):

        A = distribute_or(formula.left)
        B = distribute_or(formula.right)

        if isinstance(A, And):
```

```
            return And(distribute_or(Or(A.left, B)),
                       distribute_or(Or(A.right, B)))

        if isinstance(B, And):
            return And(distribute_or(Or(A, B.left)),
                       distribute_or(Or(A, B.right)))

        return Or(A, B)

    if isinstance(formula, And):
        return And(distribute_or(formula.left),
                   distribute_or(formula.right))

    return formula


def to_cnf(formula):
    formula = eliminate_iff(formula)
    formula = push_negation(formula)
    formula = skolemize(formula)
    formula = drop_universal(formula)
    formula = distribute_or(formula)
    return formula
```

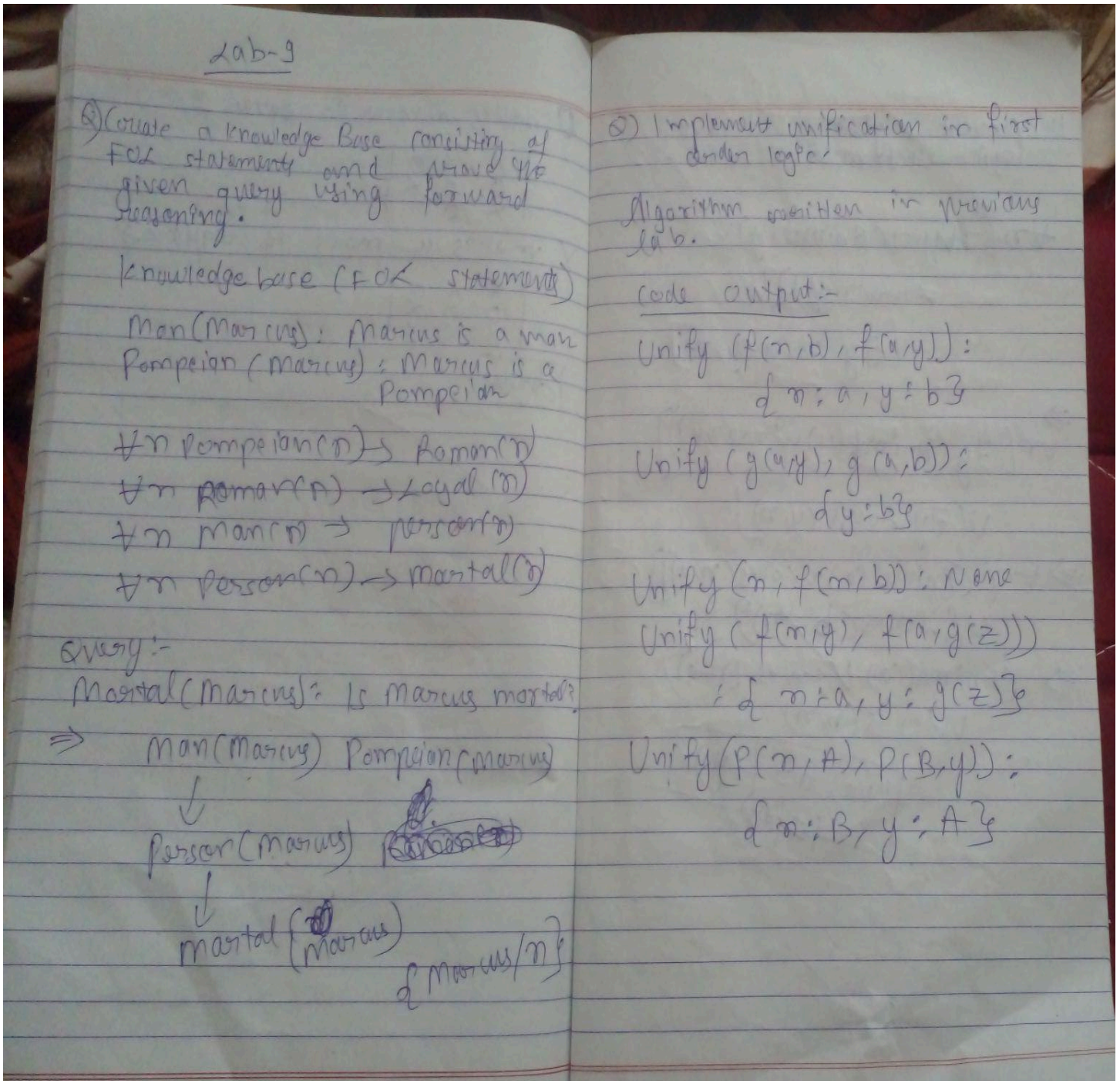
```
# ---------------- Example usage ------------------

# ∀x (P(x) → ∃y Q(x,y))

formula = ForAll(Var("x"),
            Impl(Pred("P", [Var("x")]),
                Exists(Var("y"),
                    Pred("Q", [Var("x"), Var("y")]))))

cnf = to_cnf(formula)
print("CNF:", cnf)
```

Output:

```
CNF: (¬P(x) ∨ Q(x,Sk0(x)))

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

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

PseudoCode:



Code:

```
# --------------------------------

# Forward Chaining in Python

# For the KB:
```

```
#   Parent(x,y) ⟵ Father(x,y)
#   Grandparent(x,z) ⟵ Parent(x,y) ⋀ Parent(y,z)
# Facts:
#   Father(John,Mary)
#   Parent(Mary,Susan)
# Query:
#   Grandparent(John,Susan)
# ------------------------------


from copy import deepcopy


def unify(a, b, subst=None):
    """Unify two literals a and b with a substitution."""
    if subst is None:
        subst = {}

    if a == b:
        return subst

    if isinstance(a, str) and a[0].islower():  # variable in a
        return unify_var(a, b, subst)

    if isinstance(b, str) and b[0].islower():  # variable in b
        return unify_var(b, a, subst)

    if isinstance(a, tuple) and isinstance(b, tuple):
```

```
        if a[0] != b[0] or len(a) != len(b):
            return None
        for x, y in zip(a[1:], b[1:]):
            subst = unify(x, y, subst)
            if subst is None:
                return None
        return subst

    return None


def unify_var(var, x, subst):
    """Unify variable with another term."""
    if var in subst:
        return unify(subst[var], x, subst)
    elif x in subst:
        return unify(var, subst[x], subst)
    else:
        subst[var] = x
        return subst


def substitute(term, subst):
    """Apply substitution to a literal."""
    if isinstance(term, str):
        return subst.get(term, term)
```

```
    return (term[0],) + tuple(substitute(arg, subst) for arg in term[1:])



# Horn Rule = (head, [premises...])
rules = [
    (("Parent", "x", "y"), [("Father", "x", "y")]),
    (("Grandparent", "x", "z"), [("Parent", "x", "y"), ("Parent", "y", "z")])
]

# Initial facts
facts = [
    ("Father", "John", "Mary"),
    ("Parent", "Mary", "Susan")
]

query = ("Grandparent", "John", "Susan")



def forward_chain(rules, facts, query):
    """Perform forward chaining and return True if query is derived."""
    known = set(facts)

    added_new_fact = True

    while added_new_fact:
        added_new_fact = False
```

```
for head, premises in rules:
    # Try to match all premises
    substitutions = [{}]

    for prem in premises:
        new_subs = []
        for subst in substitutions:
            prem_inst = substitute(prem, subst)
            for fact in known:
                s = unify(prem_inst, fact, deepcopy(subst))
                if s is not None:
                    new_subs.append(s)
        substitutions = new_subs

    # Add new facts derived from head
    for subst in substitutions:
        new_fact = substitute(head, subst)
        if new_fact not in known:
            print(f"Derived new fact: {new_fact}")
            known.add(new_fact)
            added_new_fact = True

            if new_fact == query:
                print("\nQuery proven by forward chaining!")
                return True
```

```
    print("\nQuery NOT provable from given KB.")
    return False
```

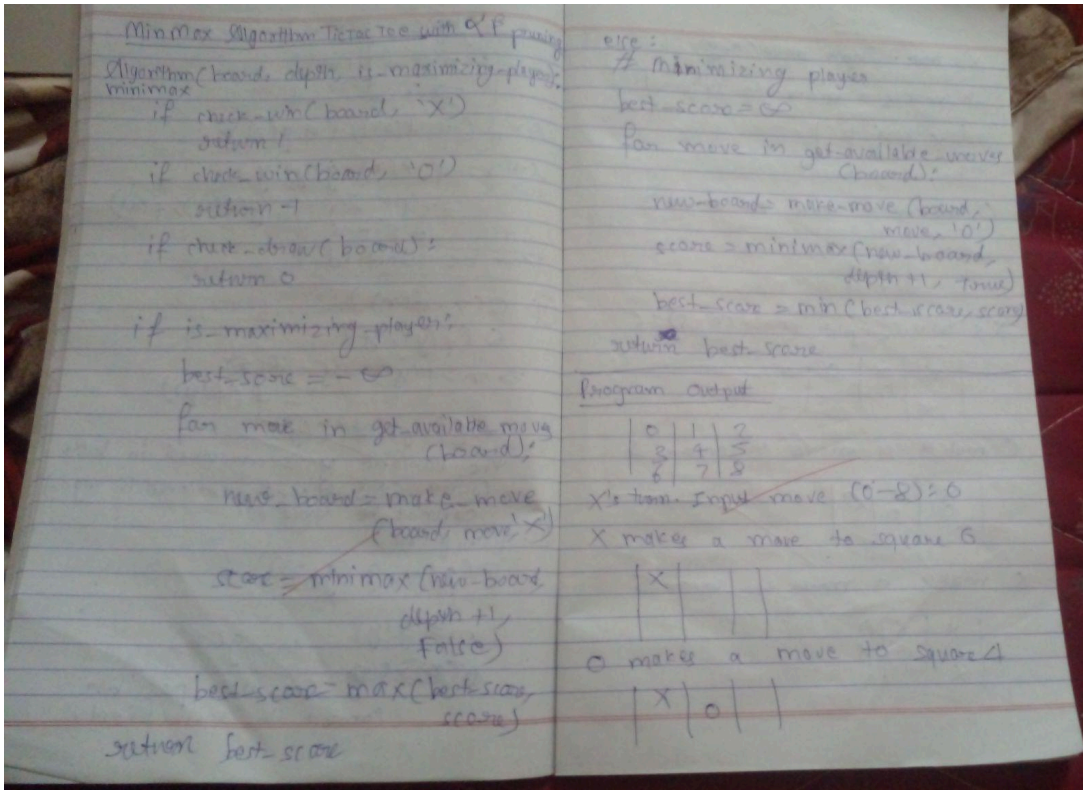
```
# Run forward chaining
forward_chain(rules, facts, query)
```

Output:

```
Derived new fact: ('Parent', 'John', 'Mary')
Derived new fact: ('Grandparent', 'John', 'Susan')

Query proven by forward chaining!

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

10.Implement Alpha-Beta Pruning

PseudoCode:



Code:

```
import math

# --- Game Board and Logic ---
class TicTacToe:
    def __init__(self):
        self.board = [' ' for _ in range(9)]  # Represents the
3x3 board
        self.current_winner = None  # Keeps track of the winner

    def print_board(self):
        for row in [self.board[i*3:(i+1)*3] for i in range(3)]:
            print('| ' + ' | '.join(row) + ' |')

    @staticmethod
    def print_board_nums():
        # 0 | 1 | 2
```

```
        # ---+---+---
        # 3 | 4 | 5
        # ---+---+---
        # 6 | 7 | 8
        number_board = [[str(i) for i in range(j*3, (j+1)*3)]
for j in range(3)]
        for row in number_board:
            print('| ' + ' | '.join(row) + ' |')

    def available_moves(self):
        return [i for i, spot in enumerate(self.board) if spot
== ' ']

    def empty_squares(self):
        return ' ' in self.board

    def num_empty_squares(self):
        return self.board.count(' ')

    def make_move(self, square, player):
        if self.board[square] == ' ':
            self.board[square] = player
            if self.check_winner(player):
                self.current_winner = player
            return True
        return False

    def check_winner(self, player):
        # Check rows
        for row in range(3):
            if all(s == player for s in
self.board[row*3:(row+1)*3]):
                return True
        # Check columns
        for col in range(3):
```

```
            if all(s == player for s in [self.board[col+i*3] for
i in range(3)]):
                return True
        # Check diagonals
        if all(s == player for s in [self.board[i] for i in [0,
4, 8]]):
            return True
        if all(s == player for s in [self.board[i] for i in [2,
4, 6]]):
            return True
        return False

# --- Minimax Algorithm ---
def minimax(state, player, tree_nodes, depth=0, parent_id=None,
alpha=-math.inf, beta=math.inf):
    """
    Minimax with alpha-beta pruning.

    Args:
        state: TicTacToe board state
        player: current player ('X' or 'O')
        tree_nodes: list to record explored nodes for
visualization
        depth: current depth in the tree
        parent_id: id of parent node in tree_nodes
        alpha: best already explored option along path to
maximizer
        beta: best already explored option along path to
minimizer

    Returns:
        dict with 'position' and 'score'
    """
    max_player = 'X'  # Our AI player
    other_player = 'O'  # Opponent of the AI (used for terminal
scoring)
```

```
    # Terminal states
    if state.current_winner == max_player:
        return {'position': None, 'score': 1 *
(state.num_empty_squares() + 1)}
    elif state.current_winner == other_player:
        return {'position': None, 'score': -1 *
(state.num_empty_squares() + 1)}
    elif not state.empty_squares():
        return {'position': None, 'score': 0}

    # Initialize best
    if player == max_player:
        best = {'position': None, 'score': -math.inf}
    else:
        best = {'position': None, 'score': math.inf}

    current_node_id = len(tree_nodes)
    tree_nodes.append({'id': current_node_id, 'board':
list(state.board), 'player': player, 'depth': depth, 'parent':
parent_id, 'score': None})

    # Determine the next player
    next_player = other_player if player == max_player else
max_player

    for possible_move in state.available_moves():
        # Simulate move
        new_state = TicTacToe()
        new_state.board = list(state.board)
        new_state.make_move(possible_move, player)

        # Recurse with alpha-beta
        sim_score = minimax(new_state, next_player, tree_nodes,
depth + 1, current_node_id, alpha, beta)
```

```
            # Update best and alpha/beta
            if player == max_player:
                if sim_score['score'] > best['score']:
                    best['score'] = sim_score['score']
                    best['position'] = possible_move
                alpha = max(alpha, best['score'])
            else:
                if sim_score['score'] < best['score']:
                    best['score'] = sim_score['score']
                    best['position'] = possible_move
                beta = min(beta, best['score'])

            # Alpha-beta pruning
            if beta <= alpha:
                break

    # Update the score of the current node in the tree
    tree_nodes[current_node_id]['score'] = best['score']

    return best

# --- Game Play ---
def play_game(game, x_player, o_player, print_game=True):
    if print_game:
        game.print_board_nums()

    letter = 'X'  # Starting player
    tree_nodes = [] # To store the game tree for visualization

    while game.empty_squares():
        if letter == 'O' and o_player == 'AI':
            square = minimax(game, letter,
tree_nodes)['position']
        elif letter == 'X' and x_player == 'AI':
            square = minimax(game, letter,
tree_nodes)['position']
```

```
        else:
            valid_move = False
            while not valid_move:
                try:
                    square = int(input(f"{letter}'s turn. Input
move (0-8): "))
                    if square not in game.available_moves():
                        raise ValueError
                    valid_move = True
                except ValueError:
                    print("Invalid square. Try again.")

        if game.make_move(square, letter):
            if print_game:
                print(f"{letter} makes a move to square
{square}")
                game.print_board()
                print('')

            if game.current_winner:
                if print_game:
                    print(f"{game.current_winner} wins!")
                return game.current_winner, tree_nodes

            letter = 'O' if letter == 'X' else 'X'  # Switch
player

    if print_game:
        print("It's a tie!")
    return 'Tie', tree_nodes

# --- Tree Visualization (Simple Text-based) ---
def visualize_tree(tree_nodes):
    print("\n--- Game Tree Visualization ---")
    for node in tree_nodes:
        indent = "  " * node['depth']
```

```
        board_str = "".join(node['board'])
        parent_info = f" (Parent: {node['parent']})" if
node['parent'] is not None else ""
        score_info = f" (Score: {node['score']})" if
node['score'] is not None else ""
        print(f"{indent}Node {node['id']}{parent_info}: Player
{node['player']} - Board: {board_str}{score_info}")

if __name__ == '__main__':
    game = TicTacToe()

    # Choose players: 'Human' or 'AI'
    x_player_type = 'Human'
    o_player_type = 'AI'

    winner, tree = play_game(game, x_player_type, o_player_type)

    # Visualize a portion of the generated game tree
    visualize_tree(tree[:50]) # Limiting to first 50 nodes for
readability
```

Output:

```
PS C:\Users\shyam\OneDrive\Desktop\ai_lab> & C:/Users/shyam/AppData/Local/Microsoft/WindowsApps/python3.13.exe "c:/Users/shyam/OneDrive/Desktop/ai_lab/alpha beta pruning/alpha_beta_pruning_tic_tac_toe.py"
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |
X's turn. Input move (0-8): 0
X makes a move to square 0
| X |   |   |
|   |   |   |
|   |   |   |

O makes a move to square 4
| X |   |   |
|   | O |   |
|   |   |   |

X's turn. Input move (0-8): 1
X makes a move to square 1
| X | X |   |
|   | O |   |
|   |   |   |

O makes a move to square 2
| X | X | O |
|   | O |   |
|   |   |   |

X's turn. Input move (0-8): 2
Invalid square. Try again.
X's turn. Input move (0-8): 3
X makes a move to square 3
| X | X | O |
| X | O |   |
|   |   |   |

O makes a move to square 6
| X | X | O |
| X | O |   |
| O |   |   |

O wins!
```