AI LAB RECORD

S.NO	TITLE	
01	8x8 puzzle using BFS	
02	DFS ,BFS, UCS & IDS on weighted graph	
03	8 puzzle using A* , RBFS	
04	Random Restart Hill Climbing using TSP,8 queens,8x8 puzzle	
05	Map Coloring using CSP	
06	9x9 Sudoko Puzzle Back Jumping With &Without Heuristics	
07	Propositional Logic Using Resolution Refutation	
08	TicTacToe Using Minimax Alpha Beta Pruning Algorithm	
09	Unification & Fwd Chaining Algo	
10	Inference On Bayesian Networks	
11	Diff 4 Sampling Methods On Bayesian Network	

WEEK1- Implement 8 puzzle using BFS:

```python
from collections import deque
N = 3
class PuzzleState:
    def __init__(self, board, x, y, depth, path):
          # constructor
        self.board = board
        self.x = x
        self.y = y
        self.depth = depth
        # list of moves
        self.path = path
 # directions
row = [0, 0, -1, 1]
col = [-1, 1, 0, 0]
moves = ['left', 'right', 'up', 'down']

def isGoalState(board):
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    return board == goal

def isValidState(x, y):
    return 0 <= x < N and 0 <= y < N

def printBoard(board):
    for row in board:
        # convert first into str and then give spaces in btwn for clarity
        print(' '.join(map(str, row)))

def issolvable(board):
    # remove zero
    flat_list = [num for row in board for num in row if num != 0]
    inversions = 0
    for i in range(len(flat_list)):
        for j in range(i + 1, len(flat_list)):
            if flat_list[i] > flat_list[j]:
                inversions += 1

    # solvable if inversions are even else odd.
    return inversions % 2 == 0
```

```python
def solvePuzzleBfs(start, x, y):
    q = deque()
    visited = set()
    q.append(PuzzleState(start, x, y, 0, []))
    # 2d tuple thing
    visited.add(tuple(map(tuple, start)))

    while q:
         # must add braces after popleft
        curr = q.popleft()
        # after evry iteration 1st check whether it has reached the goal
        if isGoalState(curr.board):
             # withut f str it wont evaluate depth variable
            print(f'\nGoal Reached at depth: {curr.depth}')
            print(f'Solution path: {curr.path}')
            print(f'Number of steps: {len(curr.path)}')
            return

        for i in range(4):
            newx = curr.x + row[i]
            newy = curr.y + col[i]

            if isValidState(newx, newy):
                 #    individually adds the each row
                newboard = [r[:] for r in curr.board]
                  # swap 0 with the target cell
                newboard[curr.x][curr.y], newboard[newx][newy] = newboard[newx][newy], newboard[curr.x]
[curr.y]
                if tuple(map(tuple, newboard)) not in visited:
                    visited.add(tuple(map(tuple, newboard)))
                    # Append current move to path
                    new_path = curr.path + [moves[i]]
                    q.append(PuzzleState(newboard, newx, newy, curr.depth + 1, new_path))

    print(' No solution found (BFS reached depth limit)')

if __name__ == '__main__':
    start = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]
    goal = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
    x, y = 1, 1
    print('Initial State:')
    printBoard(start)
    print('\nGoal State:')
    printBoard(goal)
    solvePuzzleBfs(start, x, y)
    issolvable(start)
```

OUTPUT:

```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/main.py
Initial State:
1 2 3
4 0 5
6 7 8

Goal State:
1 2 3
4 5 6
7 8 0

Goal Reached at depth: 14
Solution path: ['right', 'down', 'left', 'left', 'up', 'right', 'down', 'right', 'up', 'left', 'left',
'down', 'right', 'right']
Number of steps: 14
```

WEEK2:

To generate a weighted graph of n nodes (n = 1000 or higher) and run DFS, BFS, UCS, and IDS on it. The output must be a comparison table of these algorithms in terms of the number of nodes generated and time taken along with the shortest path between the source and destination.
The results can be reported as an average of multiple source and destination nodes chosen at random.
You have to ensure that the graph is connected after generating it and write a function to check if you have to use a loop to generate and check until a connected graph is generated.

```python
#to create&manage graphs
import networkx as nx
#to pick random nodes&generate random wts
import random
#to get execution times of algos
import time
#not used
import pandas as pd
#used in ucs
from queue import PriorityQueue
#used in bfs for efficient algo
from collections import deque

# Ensure&generate the connected graph
def generate_connected_graph(n, edge_prob=0.01):
    while True:
        #generating the random graph
        G = nx.erdos_renyi_graph(n, edge_prob)
        #checking connectivity of graph
        #we can also check the connectivity of a graph thru traversals(bfs/dfs) but the diff is who des
it internally.now our import does it.
        if nx.is_connected(G):
            #if connected it assigns the random wts from 1-10 &return it
            for (u, v) in G.edges():
                G.edges[u, v]['weight'] = random.randint(1, 10)
            return G

# BFS
def bfs(graph, start, goal):
    visited, queue = set(), deque([(start, [start])])
    nodes_generated = 0
    while queue:
        vertex, path = queue.popleft()
        nodes_generated += 1
        if vertex == goal:
            return path, nodes_generated
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in graph.neighbors(vertex):
                queue.append((neighbor, path + [neighbor]))
    return None, nodes_generated

# DFS
def dfs(graph, start, goal):
    visited, stack = set(), [(start, [start])]
    nodes_generated = 0
    while stack:
        vertex, path = stack.pop()
        nodes_generated += 1
        if vertex == goal:
            return path, nodes_generated
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in graph.neighbors(vertex):
                stack.append((neighbor, path + [neighbor]))
    return None, nodes_generated
```

```python
# UCS
def ucs(graph, start, goal):
    visited = set()
    pq = PriorityQueue()
    pq.put((0, start, [start]))
    nodes_generated = 0
    while not pq.empty():
        cost, vertex, path = pq.get()
        nodes_generated += 1
        if vertex == goal:
            return path, nodes_generated, cost
        if vertex not in visited:
            visited.add(vertex)
            for neighbor in graph.neighbors(vertex):
                edge_cost = graph.edges[vertex, neighbor]['weight']
                pq.put((cost + edge_cost, neighbor, path + [neighbor]))
    return None, nodes_generated, float('inf')

# IDS
def dls(graph, current, goal, limit, path, visited, nodes_generated):
    nodes_generated[0] += 1
    if current == goal:
        return path
    if limit <= 0:
        return None
    visited.add(current)
    for neighbor in graph.neighbors(current):
        if neighbor not in visited:
            result = dls(graph, neighbor, goal, limit - 1, path + [neighbor], visited,
nodes_generated)
            if result:
                return result
    visited.remove(current)
    return None

def ids(graph, start, goal):
    depth = 0
    nodes_generated = [0]
    while True:
        visited = set()
        result = dls(graph, start, goal, depth, [start], visited, nodes_generated)
        if result:
            return result, nodes_generated[0]
        depth += 1
```

```python
# Main
n = 1000
G = generate_connected_graph(n)
#no of random source-dest pairs(3)
trials = 3
#dictionary with K-V pairs
results = {"Algorithm": [], "Avg Time": [], "Avg Nodes": [], "Avg Cost": []}

# store exetime,nonodes generated,cost
bfs_times, dfs_times, ucs_times, ids_times = [], [], [], []
bfs_nodes, dfs_nodes, ucs_nodes, ids_nodes = [], [], [], []
ucs_costs = []

for _ in range(trials):
    #consider random source-dest nodes
    src, dest = random.sample(range(n), 2)

    #storing execution time and no of nodes explored
    #path = path from src to dest
    #nodes = number of nodes explored
    #cost = total cost (sum of edge weights) of the path
    start_time = time.time()
    path, nodes = bfs(G, src, dest)
    bfs_times.append(time.time() - start_time)
    bfs_nodes.append(nodes)

    start_time = time.time()
    path, nodes = dfs(G, src, dest)
    dfs_times.append(time.time() - start_time)
    dfs_nodes.append(nodes)

    start_time = time.time()
    path, nodes, cost = ucs(G, src, dest)
    ucs_times.append(time.time() - start_time)
    ucs_nodes.append(nodes)
    ucs_costs.append(cost)

    start_time = time.time()
    path, nodes = ids(G, src, dest)
    ids_times.append(time.time() - start_time)
    ids_nodes.append(nodes)

# Prepare results
results["Algorithm"] = ["BFS", "DFS", "UCS", "IDS"]
results["Avg Time"] = [
    sum(bfs_times)/trials, sum(dfs_times)/trials, sum(ucs_times)/trials, sum(ids_times)/trials
]
results["Avg Nodes"] = [
    sum(bfs_nodes)/trials, sum(dfs_nodes)/trials, sum(ucs_nodes)/trials, sum(ids_nodes)/trials
]
#cost is only meaningful for ucs
results["Avg Cost"] = [
    None, None, sum(ucs_costs)/trials, None
]

# Print C-style table
print("\n=== Average Results ===")
print(f"{'Algorithm':<10} {'Avg Time (s)':<15} {'Avg Nodes':<12} {'Avg Cost':<10}")
print("-" * 50)
for i in range(len(results["Algorithm"])):
    print(f"{results['Algorithm'][i]:<10} {results['Avg Time'][i]:<15.6f} {results['Avg Nodes'][i]:
<12.2f} {str(results['Avg Cost'][i] or '-'): <10}")
```

OUTPUT:

```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/secondpgm.py

=== Average Results ===
Algorithm  Avg Time (s)    Avg Nodes    Avg Cost
--------------------------------------------------
BFS        0.009151        1413.33      -
DFS        0.000000        302.00       -
UCS        0.018305        1435.00      12.666666666666666
IDS        0.000000        1593.67      -
```

WEEK-3: Solve 8-puzzle problem with A* algorithm and RBFS

```python
import heapq
import math
#for calcula execution time
import time
# Goal state
goal_state = [[1,2,3],[4,5,6],[7,8,0]]
goal_positions = {goal_state[i][j]:(i,j) for i in range(3) for j in range(3)}

# Heuristic: Manhattan distance
def manhattan(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            val = state[i][j]
            if val != 0:
                gi, gj = goal_positions[val]
                distance += abs(i-gi) + abs(j-gj)
    return distance
```

```python
#Get possible moves and then swap with neighbors
#get_neighbors = "from current board, generate all boards after moving the blank tile in each legal direction".
def get_neighbors(state):
    neighbors = []
    #loop through and find the blank tile
    i, j = next((i,j) for i in range(3) for j in range(3) if state[i][j]==0)
    #DURL directions it can move
    moves = [(1,0),(-1,0),(0,1),(0,-1)]
    for di, dj in moves:
        #new blank positions
        ni, nj = i+di, j+dj
        #validation.preventing the out of bounds
        if 0 <= ni < 3 and 0 <= nj < 3:
            #copy of the current state
            new_state = [row[:] for row in state]
            #swwap the blank tile with other neighbor
            new_state[i][j], new_state[ni][nj] = new_state[ni][nj], new_state[i][j]
            neighbors.append(new_state)
    return neighbors

# Convert state to tuple of tuples to put in set (for hashing)
def to_tuple(state):
    return tuple(tuple(row) for row in state)

# ---------------- A* Search ---------------- #
#start positin of puzzle
def astar(start):
    #unexplored nodes
    frontier = []
    #priority q               f(n)           g(n) curr  path till now
    heapq.heappush(frontier, (manhattan(start), 0, start, []))
    explored = set()

    while frontier:
        #pick the best/min one
        f, g, state, path = heapq.heappop(frontier)
        #if curr is goal then return
        if state == goal_state:
            #ret path
            return path+[state]
        explored.add(to_tuple(state))
        #add possible states
        for neighbor in get_neighbors(state):
            if to_tuple(neighbor) not in explored:
                heapq.heappush(frontier,(g+1+manhattan(neighbor), g+1, neighbor, path+[state]))
    return None #unsolvable
#g(n) = how many steps you already walked.
#h(n) = how many steps you think are left.
#f(n) = total journey (walked + remaining).
```

```python
# ---------------- RBFS ---------------- #
#Think of RBFS like “explore the best path first, but keep an eye on the 2nd best in case you need to backtrack.”
def rbfs(start):
    def rbfs_recursive(state, path, g, f_limit):
        #checks if goal
        if state == goal_state:
            return path+[state], 0
        successors = []
        for neighbor in get_neighbors(state):
            #ensures f doesn’t go backward (
            fval = max(g+1+manhattan(neighbor), g+1)
            successors.append([neighbor, path+[state], g+1, fval])
        if not successors:
            return None, math.inf
        while True:
            successors.sort(key=lambda x:x[3])
            #[3] comes coz you have indexes in successors
            best = successors[0]
            if best[3] > f_limit:
                return None, best[3]
            alternative = successors[1][3] if len(successors)>1 else math.inf
            result, best[3] = rbfs_recursive(best[0], best[1], best[2], min(f_limit, alternative))
            if result is not None:
                return result, 0
            #Start recursion with empty path, cost=0, and infinite f-limit
    return rbfs_recursive(start, [], 0, math.inf)[0]

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

print("Solving with A*:")
t1 = time.time()
path_astar = astar(start_state)
t2 = time.time()
for s in path_astar:
    for row in s: print(row)
    print()

print("Solving with RBFS:")
t3 = time.time()
path_rbfs = rbfs(start_state)
t4 = time.time()
for s in path_rbfs:
    for row in s: print(row)
    print()

# ---------------- Performance Summary ---------------- #
# Cost = number of steps (path length - 1)
cost_astar = len(path_astar) - 1 if path_astar else None
cost_rbfs = len(path_rbfs) - 1 if path_rbfs else None
time_astar = (t2 - t1) * 1000
time_rbfs = (t4 - t3) * 1000

# Print results as a simple table (no extra libraries needed)
print("\n------ Performance Comparison ------")
print(f"{'Algorithm':<10} | {'Cost (steps)':<12} | {'Execution Time (ms)':<20}")
print("-" * 50)
print(f"{'A*':<10} | {cost_astar:<12} | {time_astar:<20.3f}")
print(f"{'RBFS':<10} | {cost_rbfs:<12} | {time_rbfs:<20.3f}")
```

OUTPUT:

```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/thirdpgm.py
Solving with A*:
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]

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

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

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

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

Solving with RBFS:
[1, 2, 3]
[5, 0, 6]
[4, 7, 8]

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

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

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

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


------ Performance Comparison ------
Algorithm  | Cost (steps) | Execution Time (ms)
-------------------------------------------------
A*         | 4            | 0.000
RBFS       | 4            | 0.000
PS D:\PythonAIcodes>
```

WEEK-4:

```python
import random
import math
#taking a copy of nested lists
import copy
import time
#used to print tables in a nice formatted way (instead of just raw lists).
from tabulate import tabulate

# ---------------------- Random Restart Hill Climbing ----------------------
#The problem object (TSP, 8-puzzle, or 8-queens).
#no of times to restart the hill climbing from diff random initial state
#Maximum steps allowed per restart (to prevent infinite loops).
def random_restart_hill_climb(problem, restarts=10, max_steps=1000):
    #Stores the overall best solution found across all restarts.
    best_solution = None
    #Stores the lowest cost found so far, initialized to infinity so any real solution will be better.
    best_cost = float('inf')

    for r in range(restarts):
        #This is the key idea of random restart → avoid local minima.
        current = problem.random_state()
        for step in range(max_steps):
            neighbors = problem.get_neighbors(current)
            #stop climbing.
            if not neighbors:
                break
            #Choose the best neighbor (lowest cost) among all neighbors.
            next_state = min(neighbors, key=problem.cost)
            if problem.cost(next_state) < problem.cost(current):
                current = next_state
            else:
                break

        current_cost = problem.cost(current)
        if current_cost < best_cost:
            best_solution = current
            best_cost = current_cost

    return best_solution, best_cost
```

```python
# ---------------------- Travelling Salesman Problem ----------------------
class TSP:
    #constructor
    def __init__(self, cities):
        #cities: List of city names
        self.cities = cities
        #A distance matrix between cities, generated randomly by _generate_distances()
        self.distances = self._generate_distances()

    def _generate_distances(self):
        #Number of cities.
        n = len(self.cities)
        #2d list
        distances = [[0]*n for _ in range(n)]
        for i in range(n):
            for j in range(i+1, n):
                d = random.randint(1, 50)
                #distances[i][j] = distances[j][i] = d (because distance from A→B = B→A)
                distances[i][j] = distances[j][i] = d
        return distances

    def random_state(self):
        #Creates a random order of visiting cities (a permutation).
        state = list(range(len(self.cities)))
        random.shuffle(state)
        return state

    def cost(self, state):
        total = 0
        for i in range(len(state)):
            #state[(i+1) % len(state)]: Wraps around so the last city connects back to the first (making
it a cycle).
            total += self.distances[state[i]][state[(i+1) % len(state)]]
        return total
#If state = [0, 2, 1], cost = dist[0][2] + dist[2][1] + dist[1][0].
#If state = [0, 1, 2], neighbors are:
#Swap(0,1) → [1, 0, 2]
#Swap(0,2) → [2, 1, 0]
#Swap(1,2) → [0, 2, 1]
#A state = one possible route visiting all cities.
# Why is this useful?
#Searching all possible routes directly is too expensive (factorial growth: n! possibilities).
#So we use local search → explore only small modifications (neighbors).
#Swapping gives us a structured way to explore different paths step by step.
    def get_neighbors(self, state):
        neighbors = []
        for i in range(len(state)):
            for j in range(i+1, len(state)):
                neighbor = state[:]
                neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
                neighbors.append(neighbor)
        return neighbors
```

```python
# --------------------- 8 Queens Problem ---------------------
class EightQueens:
    #Creates a random board configuration.
#Representation: a list of length 8 → state[col] = row.
    def random_state(self):
        return [random.randint(0, 7) for _ in range(8)]
#Goal: Count number of attacking queen pairs.
#Loop through all pairs of queens (i, j):
#state[i] == state[j] → both in same row → conflict.
#abs(state[i] - state[j]) == abs(i - j) → diagonal conflict.
#conflicts = total pairs of queens attacking each other.
#Lower cost = better. Goal = cost = 0 (solution).
    def cost(self, state):
        conflicts = 0
        for i in range(len(state)):
            for j in range(i+1, len(state)):
                if state[i] == state[j] or abs(state[i]-state[j]) == abs(i-
j):                 conflicts += 1
        return conflicts

    def get_neighbors(self, state):
        neighbors = []
        #pick a queen's column.
        for col in range(8):
            #try placing that queen in every row.
            for row in range(8):
                #don't keep the same position.
                if state[col] != row:
                    #copy current state
                    neighbor = state[:]
                    #move queen in that column to a different row
                    neighbor[col] = row
                    neighbors.append(neighbor)
        return neighbors
```

```python
# ---------------------- 8 Puzzle Problem ----------------------
class EightPuzzle:
    goal_state = [[1,2,3],[4,5,6],[7,8,0]]

    def random_state(self):
        state = [i for i in range(9)]
        #to generate a new arrangement.
        random.shuffle(state)
        #Convert the 1-D shuffled list into a 3x3 matrix.
        return [state[0:3], state[3:6], state[6:9]]

    def cost(self, state):
        #This is the "cost" of how far the current state is from the goal.
        cost = 0
        for i in range(3):
            for j in range(3):
                val = state[i][j]
                if val != 0:
                    goal_x, goal_y = divmod(val-1, 3)
                    cost += abs(goal_x - i) + abs(goal_y - j)
        return cost

    def get_neighbors(self, state):
        neighbors = []
        x = y = 0
        for i in range(3):
            for j in range(3):
                if state[i][j] == 0:
                    x, y = i, j
        moves = [(0,1),(1,0),(0,-1),(-1,0)]
        for dx, dy in moves:
            nx, ny = x+dx, y+dy
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_state = copy.deepcopy(state)
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x]
[y]             neighbors.append(new_state)
        return neighbors
    #random_state() → gives random puzzle.
#cost(state) → gives heuristic Manhattan distance.
#get_neighbors(state) → gives all states after sliding one tile.
```

```python
def main():
    results = []

    # Travelling Salesman Problem
    cities = ["A","B","C","D","E"]
    problem = TSP(cities)
    t1 = time.time()
    solution, cost = random_restart_hill_climb(problem, restarts=10, max_steps=500)
    t2 = time.time()
    results.append(["TSP", cost, f"{(t2 - t1)*1000:.3f} ms"])

    # 8 Queens Problem
    problem = EightQueens()
    t3 = time.time()
    solution, cost = random_restart_hill_climb(problem, restarts=10, max_steps=500)
    t4 = time.time()
    results.append(["8-Queens", cost, f"{(t4 - t3)*1000:.3f} ms"])

    # 8 Puzzle Problem
    problem = EightPuzzle()
    t5 = time.time()
    solution, cost = random_restart_hill_climb(problem, restarts=10, max_steps=500)
    t6 = time.time()
    results.append(["8-Puzzle", cost, f"{(t6 - t5)*1000:.3f} ms"])

    # Print performance comparison table
    headers = ["Algorithm", "Cost", "Execution Time"]
    print("\n------ Performance Comparison ------")
    print(tabulate(results, headers=headers, tablefmt="grid"))

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

OUTPUT:

```
------ Performance Comparison ------
+-------------+--------+------------------+
| Algorithm   |   Cost | Execution Time   |
+=============+========+==================+
| TSP         |    105 | 2.012 ms         |
+-------------+--------+------------------+
| 8-Queens    |      0 | 28.190 ms        |
+-------------+--------+------------------+
| 8-Puzzle    |      8 | 0.000 ms         |
+-------------+--------+------------------+
```

WEEK-5: Solve Map coloring problem using CSP- Backtracking search, using AI textbook notation (assignment, csp, variables, values). Generate a planar graph of 100 nodes and find the coloring such that no two adjacent vertices have the same color. Execute this algorithm on planar graphs of varying number of vertices (1000, 10000, etc) and tabulate the algorithm statistics in terms of time and memory.

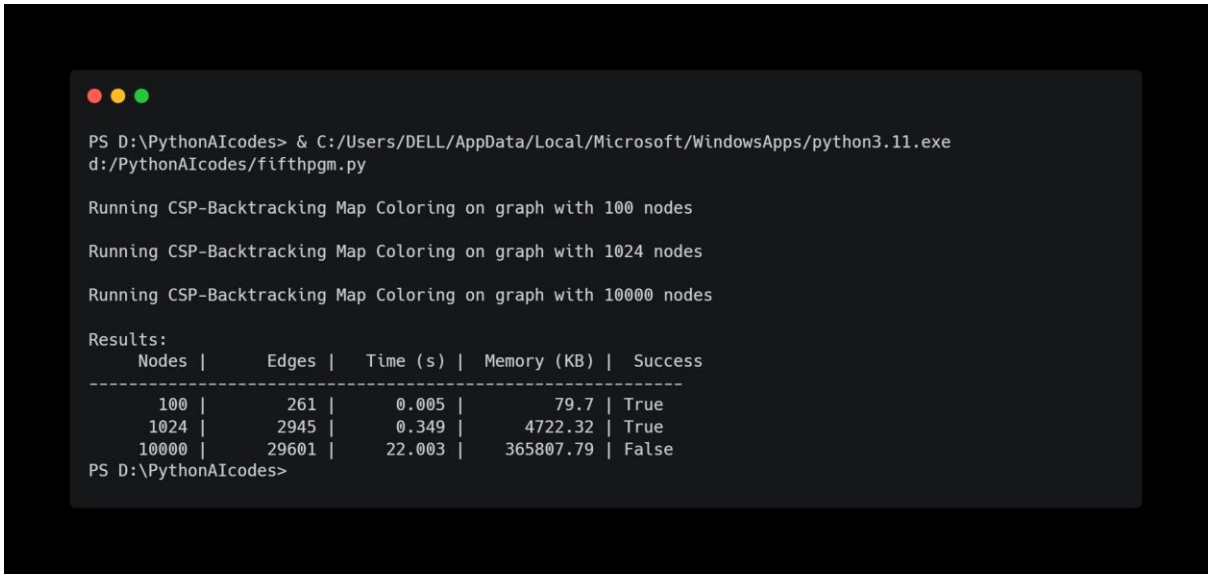
```
#A planar graph is a graph that can be drawn on a flat plane (a sheet of
paper) without any edges crossing each other
#Map coloring problem uses planar graphs:
#Each country is a node, borders are edges. Planar property ensures that 4
colors are enough to color any map (the Four Color Theorem).
import time
import tracemalloc
import sys

# Increase recursion limit,generally does upto 100
sys.setrecursionlimit(50000)

# ---------------- Graph Generator ----------------
def generate_planar_graph(rows=10, cols=10):
    #rows * cols = number of nodes.
    n = rows * cols
    #dictionary key→ each node has a set of neighbors
    adjacency = {i: set() for i in range(n)}
    #{0: {1}, 1: {0}, 2: set(), 3: set()}

    # Add edges in a grid  building a 2d grid
    for r in range(rows):
        for c in range(cols):
            #row_index * number_of_columns + column_index  uniq id of each
node
            #Why does this work?Each row has cols elements.So if you’re on row
r, you’ve already "skipped" r * cols cells from the rows above.Then, you add
the column index c to reach the exact position.
            node = r * cols + c
            #If we are not in the last column(both directions.as it is
undirected)
            if c + 1 < cols:
                right = node + 1
                adjacency[node].add(right)
                adjacency[right].add(node)
                #If we are not in the last row,connect this node to the one
below it.
            if r + 1 < rows:
                down = node + cols
                adjacency[node].add(down)
                adjacency[down].add(node)
```

```
    # add Diagonals edges (checkerboard rule)
    #We stop at rows - 1 and cols - 1(This would cause index out of range
errors.) because each diagonal needs a 2×2 square block.
    for r in range(rows - 1):
        for c in range(cols - 1):
            # Label the corners of the 2×2 square
            a = r * cols + c  #TL----
            b = a + 1          #TR
            d = a + cols       #BL
            e = d + 1          #BR
          #If r+c is even → connect a ↔ e (top-left ↔ bottom-right). If r+c is
odd → connect b ↔ d (top-right ↔ bottom-left).
          #Instead, we connect only one diagonal per 2×2 block, alternating
like a checkerboard.
            if (r + c) % 2 == 0:
                adjacency[a].add(e)
                adjacency[e].add(a)
            else:
                adjacency[b].add(d)
                adjacency[d].add(b)
                #Example with (r+c) % 2
#2×2 grid block with (r=0,c=0):
#a(0,0) —— b(0,1)
# |          |
#c(1,0) —— d(1,1)
#Here (r+c) = 0+0 = 0 (even) → connect a–d.
#Next block (r=0,c=1):
#b(0,1) —— x(0,2)
# |           |
#d(1,1) —— y(1,2)
#Here (r+c) = 0+1 = 1 (odd) → connect b–d instead.
#So it alternates diagonals like a checkerboard

    return adjacency


# ---------------- CSP Backtracking ----------------
def is_consistent(node, color, assignment, graph):
    #Check if assigning 'color' to 'node' is valid wrt neighbors.
    for neighbor in graph[node]:
        if neighbor in assignment and assignment[neighbor] == color:
            return False
    return True
#assignment → dictionary storing nodes with assigned colors.
#variables → list of all nodes.
#domains → possible colors for each node.
#graph → adjacency list.
```

```
#start + timeout → to stop if it takes too long.
#timeout → is the maximum time allowed (in seconds).
def backtrack(assignment, variables, domains, graph, start, timeout=10):
    """Recursive backtracking search with MRV + timeout."""
    #If all nodes are assigned → we found a valid solution.
    if len(assignment) == len(variables):
        return assignment

    #If it runs longer than timeout seconds → abort search.
    if time.time() - start > timeout:
        return None
    # MRV heuristic: pick variable with fewest remaining colors
    unassigned = [v for v in variables if v not in assignment]
    node = min(unassigned, key=lambda var: len(domains[var]))

    for color in domains[node]:
        if is_consistent(node, color, assignment, graph):
            assignment[node] = color
            result = backtrack(assignment, variables, domains, graph, start,
timeout)
            if result is not None:
                return result
            del assignment[node]  # backtrack

    return None
#--------solver--------
def solve_map_coloring(graph, colors=4, timeout=10):
    variables = list(graph.keys())
    domains = {v: list(range(colors)) for v in variables}
    assignment = {}
    start = time.time()#current time in sec
    return backtrack(assignment, variables, domains, graph, start, timeout)

# ---------------- Benchmark Function ---------------
def benchmark(rows, cols, colors=4, timeout=10):
    graph = generate_planar_graph(rows, cols)
    n = len(graph)

    print(f"\nRunning CSP-Backtracking Map Coloring on graph with {n} nodes")

    tracemalloc.start()
    start = time.time()

    solution = solve_map_coloring(graph, colors, timeout)

    end = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
```

```
    return {
        "Nodes": n,
        #graph is your adjacency list (dict: node → set of neighbors)For each
node, len(v) = number of neighbors (degree of that node).So sum(len(v) for v
in graph.values()) = sum of all degrees in the graph.Handshaking Lemma
Sum of degrees of all vertices=2×Number of edgesAdjacency list:
#{
 # 0: {1, 2},
  #1: {0, 2},
  #2: {0, 1}
#}
#Degrees: len(0)=2, len(1)=2, len(2)=2 → sum = 6
#But actual edges = {0-1, 1-2, 0-2} → only 3
#So sum(...) // 2 = 6 // 2 = 3
        "Edges": sum(len(v) for v in graph.values()) // 2,
        "Time (s)": round(end - start, 3),#round(x, 3) → round to 3 decimal
places.
        "Memory (KB)": round(peak / 1024, 2),
        "Success": solution is not None#did we solve it or not(just a boolean
flag to record whether the backtracking search actually found a solution.)
    }

# ---------------- Run Experiments ----------------
results = []
results.append(benchmark(10, 10, timeout=10))     # ~100 nodes
results.append(benchmark(32, 32, timeout=15))     # ~1000 nodes
results.append(benchmark(100, 100, timeout=20))   # ~10000 nodes (may timeout)

# ---------------- Tabulate Results ----------------
print("\nResults:")
print(f"{'Nodes':>10} | {'Edges':>10} | {'Time (s)':>10} | {'Memory (KB)':>12}
| {'Success':>8}")
print("-"*60)
for row in results:
    print(f"{row['Nodes']:>10} | {row['Edges']:>10} | {row['Time (s)']:>10} |
{row['Memory (KB)']:>12} | {row['Success']}")
```

OUTPUT:



```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/fifthpgm.py

Running CSP-Backtracking Map Coloring on graph with 100 nodes

Running CSP-Backtracking Map Coloring on graph with 1024 nodes

Running CSP-Backtracking Map Coloring on graph with 10000 nodes

Results:
     Nodes |      Edges |   Time (s) |  Memory (KB) |  Success
------------------------------------------------------------
       100 |        261 |      0.005 |         79.7 | True
      1024 |       2945 |      0.349 |      4722.32 | True
     10000 |      29601 |     22.003 |    365807.79 | False
PS D:\PythonAIcodes>
```

WEEK-6: 6.3.3 implement with and without heuristics.compare seaRch space and tc implement 9 9 sudoko

```python
#But in Sudoku (dense constraints), sometimes they tie.

#for execution time calc
import time
#to make its own copy without modifying the original)
import copy
#for memory usage calc
import tracemalloc

class SudokuSolver:
    def __init__(self, board):
        self.board = board
        #number of recursive calls
        self.steps = 0

    #is_valid checks if placing num in (row, col) is allowed.
    def is_valid(self, row, col, num):
        # Row & Column check
        for i in range(9):
            if self.board[row][i] == num or self.board[i][col] == num:
                return False

        # 3x3 Subgrid check
        #for top left corner
        sr, sc = 3 * (row // 3), 3 * (col // 3)
```

```
        for i in range(3):
            for j in range(3):
                if self.board[sr+i][sc+j] == num:
                    return False
        return True


# ------------------- 1. Simple Backtracking -------------------
class SimpleBacktracking(SudokuSolver):
    def solve(self):
        self.steps += 1
        #loos thru eah cell.if cell is empty(0) then try numbers in it.
        for row in range(9):
            for col in range(9):
                if self.board[row][col] == 0:
                    #1-9
                    for num in range(1, 10):
                        if self.is_valid(row, col, num):
                            self.board[row][col] = num
                            #Calls recursion: if later success → propagate
True.Otherwise, undo assignment (backtrack).
                            if self.solve():
                                return True
                            self.board[row][col] = 0
                    return False
        return True


# ------------------- 2. Heuristic Backtracking (MRV) -------------------
class HeuristicBacktracking(SudokuSolver):
    def find_mrv(self):
        #best stores best cell.min_cand=10 (larger than max possible
candidates=9)
        best = None
        min_cand = 10
        #For each empty cell, compute list of valid numbers (cand).
        for row in range(9):
            for col in range(9):
                if self.board[row][col] == 0:
                    cand = [num for num in range(1, 10) if self.is_valid(row,
col, num)]
                    #Keep track of the cell with fewest options.Returns (row,
col, candidates).
                    if len(cand) < min_cand:
                        min_cand = len(cand)
                        best = (row, col, cand)
        return best

    def solve(self):
```

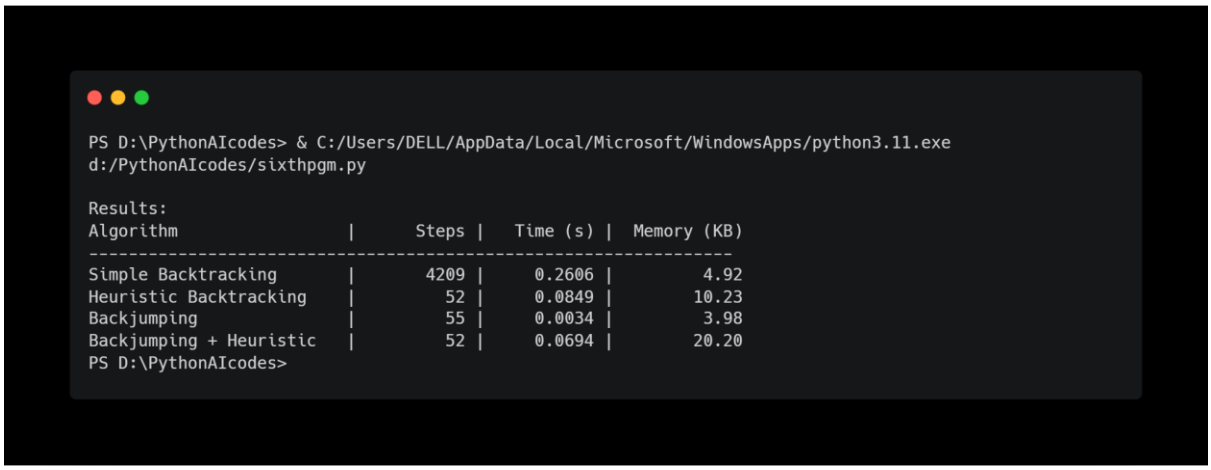
```
        #Each recursion increments steps.Finds best cell using MRV.
        self.steps += 1
        cell = self.find_mrv()
        #If no empty cell found → solved.
        if not cell:
            return True
        #Tries each candidate, backtracking if needed.
        row, col, cand = cell
        for num in cand:
            self.board[row][col] = num
            if self.solve():
                return True
            self.board[row][col] = 0
        return False


# ------------------- 3. Backjumping (Fixed) -------------------
#Backjumping is an improved version of backtracking that tries to “jump back” multiple steps at once instead of undoing just the most recent move.
class Backjumping(SudokuSolver):
    #__init__ calls parent constructor to set self.board and self.steps
    def __init__(self, board):
        super().__init__(board)
        #self.empty_cells is a precomputed list of coordinates for all empty cells, in a fixed linear order. This index-based ordering lets us refer to "variable index idx".
        self.empty_cells = [(i, j) for i in range(9) for j in range(9) if self.board[i][j] == 0]
         # conflict_sets[idx] will collect indices (or markers) that represent which earlier variables contributed to failures encountered when trying to fill empty_cells[idx].
        self.conflict_sets = [set() for _ in range(len(self.empty_cells))]


    #assigns values to the idx th empty cell
    def recursive_ibt(self, idx):
        self.steps += 1

        # BASE CASE:if idx equals the number of empty cells, all were filled successfully → puzzle solved.
        if idx == len(self.empty_cells):
            return True, idx   # return tuple both success + jump index

        #Retrieve coordinates (row, col) for the current empty cell.
        row, col = self.empty_cells[idx]

        # try all numbers 1-9
        for num in range(1, 10):
            if self.is_valid(row, col, num):
```

```
                self.board[row][col] = num
                solved, jump_to = self.recursive_ibt(idx + 1)#next empty cell
                if solved:
                    return True, jump_to

                # merge conflict sets
                self.conflict_sets[idx] |= self.conflict_sets[jump_to] #union
the set at jump_to into the set at idx
                self.board[row][col] = 0 #backtrack
                #After merging, if the current index idx appears in its own
conflict set, that means current variable is implicated in the failure and
cannot resolve it by trying other values.So we signal a backjump by returning
(False, idx) — saying: "I failed and the conflict points to idx (so callers
should consider idx)."
                if idx in self.conflict_sets[idx]:
                    # must backjump
                    return False, idx

        # ❌ no candidate worked → backjump
        #If loop finished with no num leading to success, no candidate works
here.Mark current index as conflicting (self.conflict_sets[idx].add(idx)) to
record that this variable itself caused failure.Return (False, idx - 1) —
communicates failure and suggests jumping back to idx-1.
        self.conflict_sets[idx].add(idx)
        return False, idx - 1   # jump back to earlier cell

#Public method to start recursion from the first empty cell.Unpacks the
returned pair and returns only the boolean solved to caller.
    def solve(self):
        solved, _ = self.recursive_ibt(0)
        return solved

# ------------------- 4. Backjumping + Heuristic (Fixed) -------------------
class BackjumpingHeuristic(SudokuSolver):
    def __init__(self, board):
        super().__init__(board)
        self.conflict_sets = {}   # track conflicts per cell

    def find_mrv(self):
        """Pick cell with Minimum Remaining Values (MRV)."""
        best = None
        min_cand = 10
        candidates = None
        for row in range(9):
            for col in range(9):
                if self.board[row][col] == 0:
                    cand = [num for num in range(1, 10) if self.is_valid(row,
col, num)]
```

```
                    if len(cand) < min_cand:
                        min_cand = len(cand)
                        best = (row, col)
                        candidates = cand
        return best, candidates

    def recursive_bjh(self, path):
        """path = list of assigned cells in order"""
        self.steps += 1

        # ✅ all cells filled → solved
        cell, cand = self.find_mrv()
        if not cell:
            return True, None   # solved, no jump

        row, col = cell
        for num in cand:
            if self.is_valid(row, col, num):
                self.board[row][col] = num
                solved, jump_cell = self.recursive_bjh(path + [(row, col)])
                if solved:
                    return True, None
                # merge conflict info
                self.conflict_sets[(row, col)] = self.conflict_sets.get((row,
col), set())
                self.conflict_sets[(row, col)].add(num)
                self.board[row][col] = 0

                # If conflict points to an earlier cell → backjump
                if jump_cell and jump_cell != (row, col):
                    return False, jump_cell

        # ❌ no candidate worked
        if path:
            # backjump to the *earliest conflict in path*
            return False, path[-1]
        else:
            return False, None

    def solve(self):
        solved, _ = self.recursive_bjh([])
        return solved

# ------------------ Run & Compare ------------------
#Runs any solver class (Cls) on a fresh copy of board.Measures runtime using
time.time().Also tracks memory usage with tracemalloc.
def run_solver(Cls, board, name):
    solver = Cls(copy.deepcopy(board))
```

```
    tracemalloc.start()
    start = time.time()
    solver.solve()
    end = time.time()
    current, peak = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return {
        "Algorithm": name,
        "Steps": solver.steps,
        "Time (s)": end - start,
        "Memory (KB)": round(peak / 1024, 2)   # convert bytes → KB
    }


if __name__ == "__main__":
    # Example Sudoku (0 = empty)
    board = [
        [5, 3, 0, 0, 7, 0, 0, 0, 0],
        [6, 0, 0, 1, 9, 5, 0, 0, 0],
        [0, 9, 8, 0, 0, 0, 0, 6, 0],
        [8, 0, 0, 0, 6, 0, 0, 0, 3],
        [4, 0, 0, 8, 0, 3, 0, 0, 1],
        [7, 0, 0, 0, 2, 0, 0, 0, 6],
        [0, 6, 0, 0, 0, 0, 2, 8, 0],
        [0, 0, 0, 4, 1, 9, 0, 0, 5],
        [0, 0, 0, 0, 8, 0, 0, 7, 9]
    ]
   #Runs all four solvers.Appends results to list.
    results = []
    results.append(run_solver(SimpleBacktracking, board, "Simple
Backtracking"))
    results.append(run_solver(HeuristicBacktracking, board, "Heuristic
Backtracking"))
    results.append(run_solver(Backjumping, board, "Backjumping"))
    results.append(run_solver(BackjumpingHeuristic, board, "Backjumping +
Heuristic"))

    # ---------------- Tabulate Results ----------------
    print("\nResults:")
    #:<25 → left-align algorithm name in 25 spaces.   :>10 → right-align
numbers in 10 spaces.
    print(f"{'Algorithm':<25} | {'Steps':>10} | {'Time (s)':>10} | {'Memory
(KB)':>12}")
    print("-"*65)
    for row in results:
        print(f"{row['Algorithm']:<25} | {row['Steps']:>10} | {row['Time
(s)']:>10.4f} | {row['Memory (KB)']:>12.2f}")
```

OUTPUT:



```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/sixthpgm.py

Results:
Algorithm                |      Steps |   Time (s) |  Memory (KB)
-----------------------------------------------------------------
Simple Backtracking      |       4209 |     0.2606 |         4.92
Heuristic Backtracking   |         52 |     0.0849 |        10.23
Backjumping              |         55 |     0.0034 |         3.98
Backjumping + Heuristic  |         52 |     0.0694 |        20.20
PS D:\PythonAIcodes>
```

WEEK-7:

```python
# Assignment-7
import re
from typing import List, Set, Tuple, Optional, Union

class PropositionalLogic:
    def __init__(self):
        self.variables = set()

    class Node:
        def __init__(self, value, left=None, right=None):
            self.value = value
            self.left = left
            self.right = right

        def __str__(self):
            if self.left is None and self.right is None:
                return self.value
            elif self.left is not None and self.right is None:  # Negation
                return f"~{self.left}"
            else:
                left_str = f"({self.left})" if self.left and self.left.left
and self.left.right else str(self.left)
                right_str = f"({self.right})" if self.right and
self.right.left and self.right.right else str(self.right)
                return f"{left_str} {self.value} {right_str}"

    def parse_formula(self, formula_str: str) -> Node:
        """Parse propositional logic formula into tree structure"""
```

```
        formula_str = formula_str.replace(' ', '')

        # Tokenize
        tokens = []
        i = 0
        while i < len(formula_str):
            if formula_str[i] in '()~&|':
                tokens.append(formula_str[i])
                i += 1
            elif formula_str[i:i+2] == '->':
                tokens.append('->')
                i += 2
            elif formula_str[i:i+3] == '<->':
                tokens.append('<->')
                i += 3
            elif formula_str[i].isupper():
                tokens.append(formula_str[i])
                self.variables.add(formula_str[i])
                i += 1
            else:
                raise ValueError(f"Invalid character: {formula_str[i]}")
        return self._parse_expression(tokens)

    def _parse_expression(self, tokens: List[str]) -> Node:
        if not tokens:
            raise ValueError("Empty expression")
        return self._parse_implication(tokens)

    def _parse_implication(self, tokens: List[str]) -> Node:
        left = self._parse_disjunction(tokens)
        if tokens and tokens[0] == '->':
            tokens.pop(0)
            right = self._parse_implication(tokens)
            return self.Node('->', left, right)
        elif tokens and tokens[0] == '<->':
            tokens.pop(0)
            right = self._parse_implication(tokens)
            return self.Node('<->', left, right)
        return left

    def _parse_disjunction(self, tokens: List[str]) -> Node:
        left = self._parse_conjunction(tokens)
        while tokens and tokens[0] == '|':
            tokens.pop(0)
            right = self._parse_conjunction(tokens)
            left = self.Node('|', left, right)
        return left
```

```
    def _parse_conjunction(self, tokens: List[str]) -> Node:
        left = self._parse_unary(tokens)
        while tokens and tokens[0] == '&':
            tokens.pop(0)
            right = self._parse_unary(tokens)
            left = self.Node('&', left, right)
        return left

    def _parse_unary(self, tokens: List[str]) -> Node:
        if tokens and tokens[0] == '~':
            tokens.pop(0)
            operand = self._parse_unary(tokens)
            return self.Node('~', operand)
        elif tokens and tokens[0] == '(':
            tokens.pop(0)
            expr = self._parse_expression(tokens)
            if not tokens or tokens[0] != ')':
                raise ValueError("Missing closing parenthesis")
            tokens.pop(0)
            return expr
        elif tokens and tokens[0].isupper():
            var = tokens.pop(0)
            return self.Node(var)
        else:
            raise ValueError(f"Unexpected token: {tokens[0] if tokens else
'EOF'}")

    def eliminate_equivalence(self, node: Node) -> Node:
        if node is None:
            return None
        if node.value == '<->':
            left_impl = self.Node('->', node.left, node.right)
            right_impl = self.Node('->', node.right, node.left)
            return self.Node('&', self.eliminate_equivalence(left_impl),
self.eliminate_equivalence(right_impl))
        elif node.value == '->':
            neg_left = self.Node('~', self.eliminate_equivalence(node.left))
            return self.Node('|', neg_left,
self.eliminate_equivalence(node.right))
        elif node.value in ['&', '|']:
            return self.Node(node.value,
self.eliminate_equivalence(node.left), self.eliminate_equivalence(node.right))
        elif node.value == '~':
            return self.Node('~', self.eliminate_equivalence(node.left))
        else:
            return node

    def apply_demorgan(self, node: Node) -> Node:
```

```
        if node is None:
            return None
        if node.value == '~':
            if node.left.value == '&':
                return self.Node('|',
                                 self.apply_demorgan(self.Node('~',
node.left.left)),
                                 self.apply_demorgan(self.Node('~',
node.left.right)))
            elif node.left.value == '|':
                return self.Node('&',
                                 self.apply_demorgan(self.Node('~',
node.left.left)),
                                 self.apply_demorgan(self.Node('~',
node.left.right)))
            elif node.left.value == '~':
                return self.apply_demorgan(node.left.left)
        elif node.value in ['&', '|']:
            return self.Node(node.value, self.apply_demorgan(node.left),
self.apply_demorgan(node.right))
        return node

    def distribute_disjunction(self, node: Node) -> Node:
        if node is None:
            return None
        if node.value == '|':
            if node.right and node.right.value == '&':
                left_dist = self.Node('|', node.left, node.right.left)
                right_dist = self.Node('|', node.left, node.right.right)
                return self.Node('&',
                                 self.distribute_disjunction(left_dist),
                                 self.distribute_disjunction(right_dist))
            elif node.left and node.left.value == '&':
                left_dist = self.Node('|', node.left.left, node.right)
                right_dist = self.Node('|', node.left.right, node.right)
                return self.Node('&',
                                 self.distribute_disjunction(left_dist),
                                 self.distribute_disjunction(right_dist))
        if node.value in ['&', '|']:
            return self.Node(node.value,
self.distribute_disjunction(node.left),
self.distribute_disjunction(node.right))
        return node

    def simplify_cnf(self, node: Node) -> Node:
        if node is None:
            return None
        if node.value == '&':
```

```
            left_simplified = self.simplify_cnf(node.left)
            right_simplified = self.simplify_cnf(node.right)
            if self._trees_equal(left_simplified, right_simplified):
                return left_simplified
            return self.Node('&', left_simplified, right_simplified)
        elif node.value == '|':
            left_simplified = self.simplify_cnf(node.left)
            right_simplified = self.simplify_cnf(node.right)
            if self._trees_equal(left_simplified, right_simplified):
                return left_simplified
            return self.Node('|', left_simplified, right_simplified)
        return node

    def _trees_equal(self, tree1: Node, tree2: Node) -> bool:
        if tree1 is None and tree2 is None:
            return True
        if tree1 is None or tree2 is None:
            return False
        if tree1.value != tree2.value:
            return False
        return (self._trees_equal(tree1.left, tree2.left) and
self._trees_equal(tree1.right, tree2.right))

    def cnfConvert(self, formula_str: str) -> List[List[str]]:
        formula_tree = self.parse_formula(formula_str)
        step1 = self.eliminate_equivalence(formula_tree)
        step2 = self.apply_demorgan(step1)
        step3 = self.distribute_disjunction(step2)
        step4 = self.simplify_cnf(step3)
        return self._tree_to_clauses(step4)

    def _tree_to_clauses(self, node: Node) -> List[List[str]]:
        if node is None:
            return []
        if node.value == '&':
            left_clauses = self._tree_to_clauses(node.left)
            right_clauses = self._tree_to_clauses(node.right)
            return left_clauses + right_clauses
        elif node.value == '|':
            literals = self._extract_literals(node)
            return [literals]
        else:
            return [[self._node_to_literal(node)]]

    def _extract_literals(self, node: Node) -> List[str]:
        if node is None:
            return []
        if node.value == '|':
```

```
            left_literals = self._extract_literals(node.left)
            right_literals = self._extract_literals(node.right)
            return left_literals + right_literals
        return [self._node_to_literal(node)]

    def _node_to_literal(self, node: Node) -> str:
        if node.value == '~':
            return f"~{node.left.value}"
        else:
            return node.value


class ResolutionProver:
    def __init__(self):
        self.steps = 0
        self.max_clauses = 0
        self.proof_sequence = []

    def plResolution(self, premises: List[str], goal: str, strategy: int = 0,
                     max_steps: int = 1000, max_clauses: int = 10000) ->
Tuple[bool, int, int, List[str]]:
        self.steps = 0
        self.max_clauses = 0
        self.proof_sequence = []

        pl = PropositionalLogic()
        clauses = []
        for premise in premises:
            clauses.extend(pl.cnfConvert(premise))

        negated_goal = f"~({goal})"
        goal_clauses = pl.cnfConvert(negated_goal)

        sos = set(self._clause_to_tuple(clause) for clause in goal_clauses)
        other_clauses = set(self._clause_to_tuple(clause) for clause in
clauses)
        all_clauses = sos | other_clauses

        self._update_max_clauses(len(all_clauses))

        while sos and self.steps < max_steps and len(all_clauses) <
max_clauses:
            sos_clause_tuple = next(iter(sos))
            sos_clause = list(sos_clause_tuple)
            sos.remove(sos_clause_tuple)

            new_clauses = set()
            for other_clause_tuple in all_clauses - {sos_clause_tuple}:
                other_clause = list(other_clause_tuple)
```

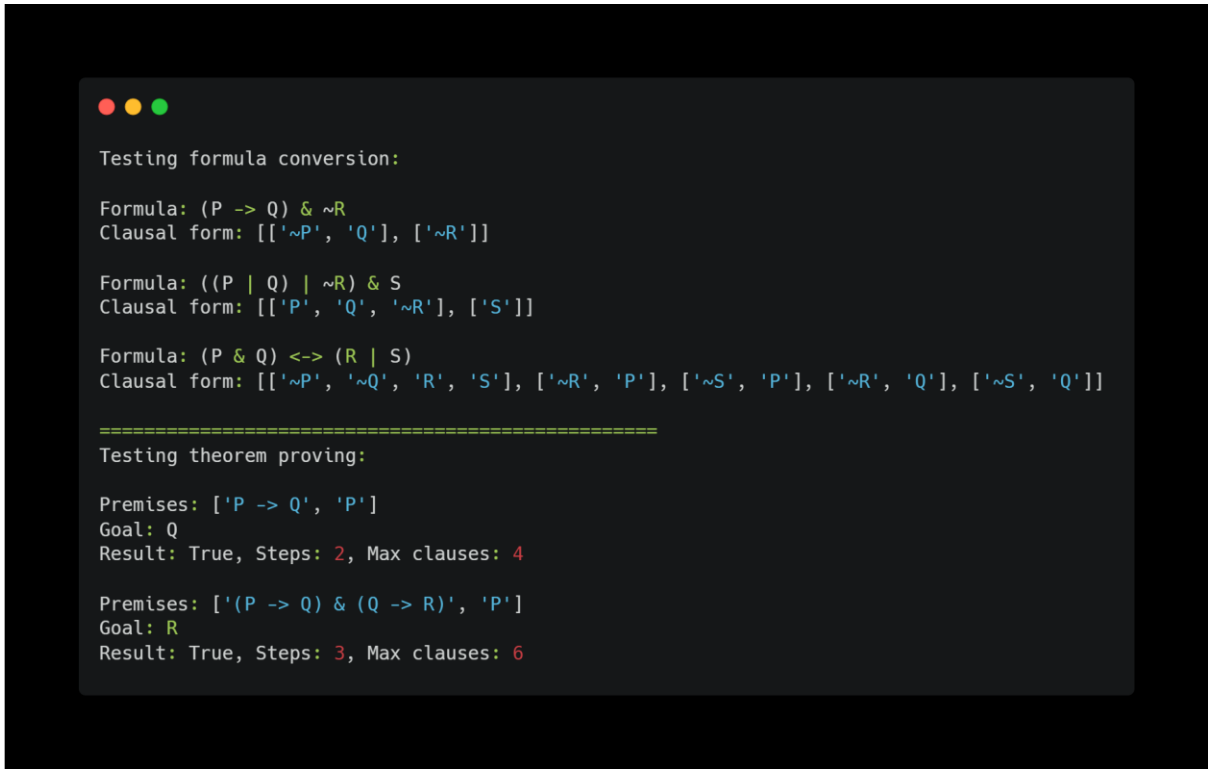
```
                resolvents = self._resolve_clauses(sos_clause, other_clause)
                for resolvent in resolvents:
                    resolvent_tuple = tuple(sorted(resolvent))
                    if not resolvent:
                        self.steps += 1
                        self.proof_sequence.append(f"Resolved {sos_clause}
with {other_clause} -> EMPTY")
                        self._update_max_clauses(len(all_clauses))
                        return True, self.steps, self.max_clauses,
self.proof_sequence
                    if resolvent_tuple not in all_clauses and resolvent_tuple
not in new_clauses:
                        new_clauses.add(resolvent_tuple)
                        self.proof_sequence.append(f"Resolved {sos_clause}
with {other_clause} -> {resolvent}")
            self.steps += 1
            if strategy == 1:
                new_clauses = self._simplify_clauses(new_clauses)
                all_clauses = self._simplify_clauses(all_clauses)
            for new_clause in new_clauses:
                sos.add(new_clause)
                all_clauses.add(new_clause)
            self._update_max_clauses(len(all_clauses))
            if len(all_clauses) >= max_clauses:
                raise MemoryError("Maximum clause limit exceeded")

        if self.steps >= max_steps:
            raise TimeoutError("Maximum step limit exceeded")
        return False, self.steps, self.max_clauses, self.proof_sequence

    def _resolve_clauses(self, clause1: List[str], clause2: List[str]) ->
List[List[str]]:
        resolvents = []
        for literal1 in clause1:
            for literal2 in clause2:
                if (literal1.startswith('~') and literal1[1:] == literal2) or
\
                   (literal2.startswith('~') and literal2[1:] == literal1):
                    resolvent = [l for l in clause1 if l != literal1] + [l for
l in clause2 if l != literal2]
                    resolvent = list(dict.fromkeys(resolvent))
                    if not self._is_valid_clause(resolvent):
                        resolvents.append(resolvent)
        return resolvents

    def _is_valid_clause(self, clause: List[str]) -> bool:
        for literal in clause:
            if literal.startswith('~'):
```

```
                if literal[1:] in clause:
                    return True
            else:
                if f"~{literal}" in clause:
                    return True
        return False

    def _simplify_clauses(self, clauses: Set[Tuple[str]]) -> Set[Tuple[str]]:
        simplified = set()
        clause_list = [set(clause) for clause in clauses]
        i = 0
        while i < len(clause_list):
            clause1 = clause_list[i]
            if self._has_complementary_pair(clause1):
                i += 1
                continue
            subsumed = False
            j = 0
            while j < len(clause_list) and not subsumed:
                if i != j:
                    clause2 = clause_list[j]
                    if clause1.issuperset(clause2):
                        subsumed = True
                j += 1
            if not subsumed:
                simplified.add(tuple(sorted(clause1)))
            i += 1
        return simplified

    def _has_complementary_pair(self, clause: Set[str]) -> bool:
        for literal in clause:
            if literal.startswith('~'):
                if literal[1:] in clause:
                    return True
            else:
                if f"~{literal}" in clause:
                    return True
        return False

    def _clause_to_tuple(self, clause: List[str]) -> Tuple[str]:
        return tuple(sorted(clause))

    def _update_max_clauses(self, current_count: int):
        if current_count > self.max_clauses:
            self.max_clauses = current_count


def main():
    print("Propositional Logic Theorem Prover")
```

```
    print("Enter premises (one per line, empty line to finish):")
    premises = []
    while True:
        line = input().strip()
        if not line:
            break
        premises.append(line)

    print("Enter goal formula:")
    goal = input().strip()
    print("Select strategy (0 - set-of-support, 1 - set-of-support +
simplification):")
    strategy = int(input().strip())
    print("Enter maximum steps (default 1000):")
    try:
        max_steps = int(input().strip())
    except:
        max_steps = 1000
    print("Enter maximum clauses (default 10000):")
    try:
        max_clauses = int(input().strip())
    except:
        max_clauses = 10000

    prover = ResolutionProver()
    try:
        result, steps, max_clauses_used, proof_seq =
prover.plResolution(premises, goal, strategy, max_steps, max_clauses)
        print("\n" + "=" * 50)
        if result:
            print("RESULT: Goal is proven")
        else:
            print("RESULT: Goal cannot be proven")
        print(f"Steps: {steps}")
        print(f"Maximum clauses in memory: {max_clauses_used}")
        print("\nProof sequence:")
        for step in proof_seq:
            print(f"  {step}")
    except MemoryError as e:
        print(f"ERROR: {e}")
    except TimeoutError as e:
        print(f"ERROR: {e}")
    except Exception as e:
        print(f"ERROR: {e}")


if __name__ == "__main__":
    print("Testing formula conversion:")
    pl = PropositionalLogic()
```

```
    test_formulas = [
        "(P -> Q) & ~R",
        "((P | Q) | ~R) & S",
        "(P & Q) <-> (R | S)"
    ]
    for formula in test_formulas:
        print(f"\nFormula: {formula}")
        try:
            clauses = pl.cnfConvert(formula)
            print(f"Clausal form: {clauses}")
        except Exception as e:
            print(f"Error: {e}")

    print("\n" + "=" * 50)
    print("Testing theorem proving:")
    prover = ResolutionProver()
    premises1 = ["P -> Q", "P"]
    goal1 = "Q"
    print(f"\nPremises: {premises1}")
    print(f"Goal: {goal1}")
    try:
        result, steps, max_clauses, proof = prover.plResolution(premises1,
goal1)
        print(f"Result: {result}, Steps: {steps}, Max clauses: {max_clauses}")
    except Exception as e:
        print(f"Error: {e}")

    premises2 = ["(P -> Q) & (Q -> R)", "P"]
    goal2 = "R"
    print(f"\nPremises: {premises2}")
    print(f"Goal: {goal2}")
    try:
        result, steps, max_clauses, proof = prover.plResolution(premises2,
goal2)
        print(f"Result: {result}, Steps: {steps}, Max clauses: {max_clauses}")
    except Exception as e:
        print(f"Error: {e}")

    # main()
```

OUTPUT:



```
Testing formula conversion:

Formula: (P -> Q) & ~R
Clausal form: [['~P', 'Q'], ['~R']]

Formula: ((P | Q) | ~R) & S
Clausal form: [['P', 'Q', '~R'], ['S']]

Formula: (P & Q) <-> (R | S)
Clausal form: [['~P', '~Q', 'R', 'S'], ['~R', 'P'], ['~S', 'P'], ['~R', 'Q'], ['~S', 'Q']]

================================================
Testing theorem proving:

Premises: ['P -> Q', 'P']
Goal: Q
Result: True, Steps: 2, Max clauses: 4

Premises: ['(P -> Q) & (Q -> R)', 'P']
Goal: R
Result: True, Steps: 3, Max clauses: 6
```

WEEK-8:

```python
#to pick a random move for the "human"
import random
#to measure execution time of the algorithms
import time

# ------------------ Board Utilities ------------------
def print_board(board):
    for row in board:
        print("|".join(row))
        print("-"*5)

#Checks if there are any empty spaces left on the board.
# Returns True if at least one cell is empty.
def is_moves_left(board):
    return any(cell == ' ' for row in board for cell in row)

#Evaluates any player has won the Tic-Tac-Toe game.
# Returns:10 → Computer ('X') wins.-10 → Human ('O') wins.0 → No one has won
yet.
def evaluate(board):
    # Check rows and columns
    for i in range(3):
```

```
        #checks if all 3 cells in row i are equal and not empty.
        if board[i][0] == board[i][1] == board[i][2] != ' ':
            return 10 if board[i][0] == 'X' else -10
        if board[0][i] == board[1][i] == board[2][i] != ' ':
            return 10 if board[0][i] == 'X' else -10

    # Check diagonals
    #main diagonal (top-left to bottom-right).
    if board[0][0] == board[1][1] == board[2][2] != ' ':
        return 10 if board[0][0] == 'X' else -10
    #anti-diagonal (top-right to bottom-left).
    if board[0][2] == board[1][1] == board[2][0] != ' ':
        return 10 if board[0][2] == 'X' else -10
    #no winner- draw case or still running.
    return 0

# ------------------ Minimax Algorithm ------------------
#to track how many nodes the algorithm evaluates.
minimax_nodes = 0
#is_max → True if it's the computer's turn, False for the human.
##Computer tries all possible moves and chooses the one that maximizes its
chances of winning.
def minimax(board, is_max):
    global minimax_nodes
    minimax_nodes += 1

    score = evaluate(board)
    #Base case: if the game is over, return the score.If the board is full and
no winner → draw → return 0.
    if score == 10 or score == -10:
        return score
    if not is_moves_left(board):
        return 0

#Try all empty cells.Place 'X' in that cell and recursively call minimax for
human.
# Keep track of the maximum score (best move for computer).
    if is_max:
        best = -1000
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'X'
                    best = max(best, minimax(board, False))
                    board[i][j] = ' '#backtracking
        return best
    #For human's turn:Place 'O' and recursively call minimax for computer.
    # Keep track of the minimum score (worst-case for computer).
```

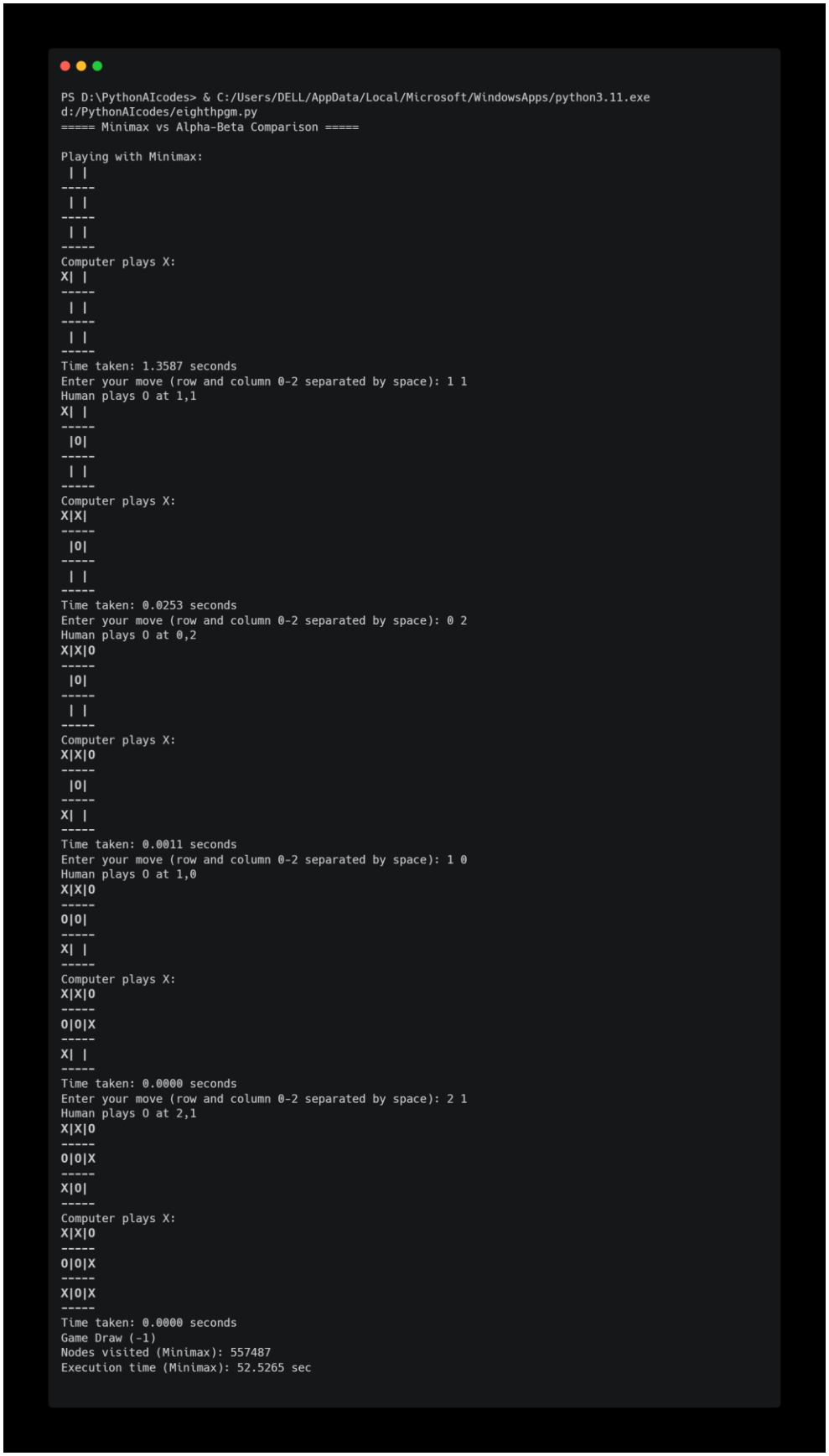
```
    #Human is assumed to play perfectly and will always try to minimize the
computer's score.
    else:
        best = 1000
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                    best = min(best, minimax(board, True))
                    board[i][j] = ' '
        return best

#Iterates all possible moves for the computer.Calls minimax to get the value
of that move.
# Returns the move with the highest score.
def find_best_move(board):
    best_val = -1000
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X'
                move_val = minimax(board, False)
                board[i][j] = ' '
                if move_val > best_val:
                    best_val = move_val
                    best_move = (i, j)
    return best_move

# ------------------ Alpha-Beta Pruning ------------------
ab_nodes = 0
#alpha-maximiser
#beta-minimiser
def minimax_ab(board, alpha, beta, is_max):
    global ab_nodes
    ab_nodes += 1

    score = evaluate(board)
    if score == 10 or score == -10:
        return score
    if not is_moves_left(board):
        return 0

    if is_max:
        best = -1000
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
```

```
                    board[i][j] = 'X'
                    best = max(best, minimax_ab(board, alpha, beta, False))
                    board[i][j] = ' '
                    alpha = max(alpha, best)
                    #If beta <= alpha → prune the remaining branches (skip
them).
                    if beta <= alpha:
                        break
        return best
    else:
        best = 1000
        for i in range(3):
            for j in range(3):
                if board[i][j] == ' ':
                    board[i][j] = 'O'
                    best = min(best, minimax_ab(board, alpha, beta, True))
                    board[i][j] = ' '
                    beta = min(beta, best)
                    if beta <= alpha:
                        break
        return best

def find_best_move_ab(board):
    best_val = -1000
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == ' ':
                board[i][j] = 'X'
                move_val = minimax_ab(board, -1000, 1000, False)
                board[i][j] = ' '
                if move_val > best_val:
                    best_val = move_val
                    best_move = (i, j)
    return best_move

# ------------------ Game Loop ------------------
#Creates an empty board.
#use_alpha_beta → Determines whether to use Minimax or Alpha-Beta.
def play_game(use_alpha_beta=True):
    global minimax_nodes, ab_nodes
    board = [[' ']*3 for _ in range(3)]
    print_board(board)

    for turn in range(9):
        if turn % 2 == 0:
            # Computer Move
            start = time.time()
```

```
            if use_alpha_beta:
                best_move = find_best_move_ab(board)
            else:
                best_move = find_best_move(board)
            end = time.time()
            board[best_move[0]][best_move[1]] = 'X'
            print("Computer plays X:")
            print_board(board)
            print(f"Time taken: {end-start:.4f} seconds")
        else:
            # Human Move (manual input)
            #keep asking for input until the user gives a valid move.
            while True:
                try:
                    #splits that into a list of two strings → ['1', '2']
                    # map(int, ...) converts them to integers
                    r, c = map(int, input("Enter your move (row and column 0-2
separated by space): ").split())

                    #Row and column numbers are between 0 and 2 (since it's a
3×3 board).
                    # The selected cell is empty (' ').
                    if 0 <= r <= 2 and 0 <= c <= 2 and board[r][c] == ' ':
                        board[r][c] = 'O'
                        #Then we break the loop to continue the game.
                        break
                    else:
                        #if error we print an error and re ask.
                        print("Invalid move! Try again.")
                except ValueError:
                    print("Please enter two integers between 0 and 2.")
            print(f"Human plays O at {r},{c}")
            print_board(board)

        score = evaluate(board)
        if score == 10:
            print("Computer wins!")
            break
        elif score == -10:
            print("Human wins!")
            break
    else:
        print("Game Draw (-1)")

# ------------------ Run and Compare ------------------
print("===== Minimax vs Alpha-Beta Comparison =====")
minimax_nodes = 0
ab_nodes = 0
```

```
# Minimax Test
print("\nPlaying with Minimax:")
start_time = time.time()
play_game(use_alpha_beta=False)
minimax_time = time.time() - start_time
print(f"Nodes visited (Minimax): {minimax_nodes}")
print(f"Execution time (Minimax): {minimax_time:.4f} sec")

# Alpha-Beta Test
print("\nPlaying with Alpha-Beta Pruning:")
start_time = time.time()
play_game(use_alpha_beta=True)
ab_time = time.time() - start_time
print(f"Nodes visited (Alpha-Beta): {ab_nodes}")
print(f"Execution time (Alpha-Beta): {ab_time:.4f} sec")


#Create a 3×3 board.Evaluate board after every move.Minimax explores all
possible moves recursively.
#Alpha-Beta prunes unnecessary branches → fewer nodes visited.
#Compare nodes visited and execution time → see efficiency gains.

# ------------------ Summary Section ------------------
improvement_nodes = ((minimax_nodes - ab_nodes) / minimax_nodes * 100) if
minimax_nodes else 0
improvement_time = ((minimax_time - ab_time) / minimax_time * 100) if
minimax_time else 0

print("\n===== SUMMARY")
print(f"NODES (Space Used):")
print(f"   Minimax expanded {minimax_nodes} nodes")
print(f"   Alpha-Beta expanded {ab_nodes} nodes")
print(f"   Space reduction: {improvement_nodes:.2f}% fewer nodes expanded\n")

print(f"EXECUTION TIME (Speed):")
print(f"   Minimax took {minimax_time:.4f} seconds")
print(f"   Alpha-Beta took {ab_time:.4f} seconds")
print(f"   Speed improvement: {improvement_time:.2f}% faster\n")

print("=LAST QUESTION O/P")
print("- Alpha-Beta pruning reduces both time and space requirements
drastically.")
print("- It achieves the same optimal result as Minimax but explores fewer
nodes.")
print("- The best-case time complexity improves from O(b^d) to O(b^(d/2)).")
#exploring the most promising nodes first.
print("- Node reordering (evaluating promising moves first) can further
improve pruning efficiency.")
```

OUTPUT:



```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/eighthpgm.py
===== Minimax vs Alpha-Beta Comparison =====

Playing with Minimax:
 | |
-----
 | |
-----
 | |
-----
Computer plays X:
X| |
-----
 | |
-----
 | |
-----
Time taken: 1.3587 seconds
Enter your move (row and column 0-2 separated by space): 1 1
Human plays O at 1,1
X| |
-----
 |O|
-----
 | |
-----
Computer plays X:
X|X|
-----
 |O|
-----
 | |
-----
Time taken: 0.0253 seconds
Enter your move (row and column 0-2 separated by space): 0 2
Human plays O at 0,2
X|X|O
-----
 |O|
-----
 | |
-----
Computer plays X:
X|X|O
-----
 |O|
-----
X| |
-----
Time taken: 0.0011 seconds
Enter your move (row and column 0-2 separated by space): 1 0
Human plays O at 1,0
X|X|O
-----
O|O|
-----
X| |
-----
Computer plays X:
X|X|O
-----
O|O|X
-----
X| |
-----
Time taken: 0.0000 seconds
Enter your move (row and column 0-2 separated by space): 2 1
Human plays O at 2,1
X|X|O
-----
O|O|X
-----
X|O|
-----
Computer plays X:
X|X|O
-----
O|O|X
-----
X|O|X
-----
Time taken: 0.0000 seconds
Game Draw (-1)
Nodes visited (Minimax): 557487
Execution time (Minimax): 52.5265 sec
```

```
Playing with Alpha-Beta Pruning:
 | |
-----
 | |
-----
 | |
-----
Computer plays X:
X| |
-----
 | |
-----
 | |
-----
Time taken: 0.2089 seconds
Enter your move (row and column 0-2 separated by space): 1 0
Human plays O at 1,0
X| |
-----
O| |
-----
 | |
-----
Computer plays X:
X|X|
-----
O| |
-----
 | |
-----
Time taken: 0.0154 seconds
Enter your move (row and column 0-2 separated by space): 0 2
Human plays O at 0,2
X|X|O
-----
O| |
-----
 | |
-----
Computer plays X:
X|X|O
-----
O|X|
-----
 | |
-----
Time taken: 0.0014 seconds
Enter your move (row and column 0-2 separated by space): 2 1
Human plays O at 2,1
X|X|O
-----
O|X|
-----
 |O|
-----
Computer plays X:
X|X|O
-----
O|X|
-----
 |O|X
-----
Time taken: 0.0000 seconds
Computer wins!
Nodes visited (Alpha-Beta): 84343
Execution time (Alpha-Beta): 23.5576 sec

===== SUMMARY
NODES (Space Used):
   Minimax expanded 557487 nodes
   Alpha–Beta expanded 84343 nodes
   Space reduction: 84.87% fewer nodes expanded

EXECUTION TIME (Speed):
   Minimax took 52.5265 seconds
   Alpha–Beta took 23.5576 seconds
   Speed improvement: 55.15% faster

=LAST QUESTION O/P
- Alpha–Beta pruning reduces both time and space requirements drastically.
- It achieves the same optimal result as Minimax but explores fewer nodes.
- The best-case time complexity improves from O(b^d) to O(b^(d/2)).
- Node reordering (evaluating promising moves first) can further improve pruning efficiency.
```

WEEK-9:

```
# ================================================================
# AI LAB TASK - EXERCISE 9.19
# Implementing Unification Algorithm + Forward Chaining Algorithm
# ================================================================

# computes Cartesian products
from itertools import product

# --------------------------------
# Unification algorithm

#theta – a dictionary (a list of variable replacements we already know).
def unify(x, y, theta=None):
    #If no substitution list was given, start with an empty one.
    if theta is None:
        theta = {}
    #If both things are exactly the same, then just return what we already
have.
    if x == y:
        return theta

    #lowercase = variable, uppercase/other = constant or functor name
    #A functor is the name of a function or relation symbol in logic
expressions.It’s basically the main operator or predicate name.

    #instance-It’s a built-in Python function that checks the type of a
variable.
    #isinstance(object, type)
    #It returns:
    #True – if the object is of that given type
    #False – otherwise
    if isinstance(x, str) and x.islower():  # variable
        return unify_var(x, y, theta)
    elif isinstance(y, str) and y.islower():  # variable
        return unify_var(y, x, theta)
    #"Father", "John") → means Father(John)
    # tuples represent predicates or functions with arguments.
    #If both sides are structured expressions of the same shape (same number
of arguments),then try to unify them element by element
    elif isinstance(x, tuple) and isinstance(y, tuple) and len(x) == len(y):
        #unify the corresponding components pairwise.
        #The built-in zip() pairs up each element from both tuples.
        for a, b in zip(x, y):
            theta = unify(a, b, theta)
            if theta is None:
                return None
        return theta
```

```
    else:
        return None

#decides how a variable should be bound during unification.
def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif occurs_check(var, x, theta):
        return None
    else:
        theta[var] = x
        return theta

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

# ------------------------------------------------------------
# Forward Chaining algorithm for first-order Horn clauses

def forward_chain_first_order(KB, query, max_iterations=15):
    terms = set(["John"])  # known constantsymbols
    facts = set()#storing all the facts inferred so far
    iteration = 0#to avoid  infinite loops

    print(f"Initial known terms: {list(terms)}")
    print(f"Query: {pretty(query)}")
    print("------------------------------------------------------------------
------")

    while iteration < max_iterations:
        iteration += 1
        new_facts = set()
        print(f"\n--- Iteration {iteration} ---")
        if facts:
            print("Known ground facts:")
            for f in facts:
                print(f"   {pretty(f)}")
        else:
            print("Known ground facts:\n  (none yet)")

        # Apply rules
        for rule in KB:
```

```
            ## If the rule doesn't have an implication (=>), skip it
            # Example of a rule with => : { "=>": ([premises], [conclusion]) }
            if "=>" not in rule:
                continue
            lhs, rhs = rule["=>"]
            vars_in_rule = list(collect_vars(lhs) | collect_vars(rhs))
            #try every combination now
            for assignment in product(list(terms), repeat=len(vars_in_rule)):
                subs = dict(zip(vars_in_rule, assignment))
                lhs_inst = substitute(lhs, subs)
                rhs_inst = substitute(rhs, subs)
                # Check if all premises in LHS are already true (in known
facts)
        #   - For each premise, see if it unifies with any known fact
                if all(any(unify(prem, f, {}) is not None for f in facts) for
prem in lhs_inst):
                    new_fact = make_hashable(rhs_inst[0])
                    if new_fact not in facts:
                        new_facts.add(new_fact)
                        terms |= collect_terms(new_fact)

        #stop if nothing new inferred.query cant be proved.
        if not new_facts:
             print("\nNo new facts inferred — stopping.")
             print(f"Facts = {{ {', '.join(pretty(fa) for fa in facts)}  }}")
             print(f"Terms = {{ {', '.join(sorted(terms))} }}")
             print("Result: NO ❌\n")
             return False

        print("\nNew facts added this iteration:")
        for f in new_facts:
            print(f"   {pretty(f)}")

        for f in new_facts:
            theta = unify(f, query, {})
            if theta is not None:
                print(f"\nFacts = {{ {', '.join(pretty(fa) for fa in facts |
new_facts)} }}")
                print(f"Terms = {{ {', '.join(sorted(terms))} }}")
                print(f"\n✅ Query satisfied by fact: {pretty(f)}")
                # Format substitution nicely
                sub_str = ", ".join(f"{v}: {k}" for v, k in theta.items())
                print(f"   substitution: {{ {sub_str} }}")
                print("Result: YES ✅\n")
                return True  # ✅ must align with the 'if' (4 spaces more than
for)

        # ✅ this must be OUTSIDE the for-loop but INSIDE the while-loop
```

```
        facts |= new_facts


    print("\nMax iterations reached — stopping.")
    print("Result: UNKNOWN \n")
    return False

# ------------------------------------------------------------
#  Helper functions

def substitute(predicate, subs):
    # ✅ FIXED VERSION (keeps output consistent and hashable)
    if isinstance(predicate, tuple):
        return tuple(substitute(arg, subs) if isinstance(arg, tuple) else
subs.get(arg, arg) for arg in predicate)
    elif isinstance(predicate, list):
        return [substitute(p, subs) for p in predicate]
    else:
        return subs.get(predicate, predicate)

def collect_vars(expr):
    if isinstance(expr, str) and expr.islower():
        return {expr}
    elif isinstance(expr, tuple):
        s = set()
        for e in expr:
            s |= collect_vars(e)
        return s
    elif isinstance(expr, list):
        s = set()
        for e in expr:
            s |= collect_vars(e)
        return s
    return set()

def collect_terms(expr):
    if isinstance(expr, str) and not expr.islower():
        return {expr}
    elif isinstance(expr, tuple):
        s = set()
        for e in expr:
            s |= collect_terms(e)
        return s
    elif isinstance(expr, list):
        s = set()
        for e in expr:
            s |= collect_terms(e)
        return s
    return set()
```

```
def make_hashable(x):
    ## If x is a list, convert each element into a hashable form (recursively)
    # and then make it a tuple.
    if isinstance(x, list):
        return tuple(make_hashable(i) for i in x)
    ## If x is a tuple, check if it contains lists inside — and convert them
too.
    elif isinstance(x, tuple):
        return tuple(make_hashable(i) for i in x)
    ## Otherwise (string, number, etc.) — already hashable, so return it as
is.
    return x

def pretty(expr):
    if isinstance(expr, tuple):
        functor = expr[0]
        args = expr[1:]
        if not args:
            return str(functor)
        pretty_args = [pretty(a) for a in args]
        return f"{functor}({', '.join(pretty_args)})"
    else:
        return str(expr)

# ----------------------------------------------------------
# Knowledge Base (KB)
KB = [
    {"=>": ([("Ancestor", "x", "y"), ("Ancestor", "y", "z")],
             [("Ancestor", "x", "z")])},

    {"=>": ([], [("Ancestor", ("Mother", "x"), "x")])}
]

# ----------------------------------------------------------
#  Queries from Exercise 9.19
queries = [
    ("Ancestor", ("Mother", "y"), "John"),
    ("Ancestor", ("Mother", ("Mother", "y")), "John"),
    ("Ancestor", ("Mother", ("Mother", ("Mother", "y"))), ("Mother", "y")),
    ("Ancestor", ("Mother", "John"), ("Mother", ("Mother", "John")))
]

# ----------------------------------------------------------
# Run all queries
#“Go through every query in the list called queries,and while doing so, also
give me a counter i starting from 1
for i, q in enumerate(queries, start=1):
```

```
    print("====================")
    print(f"Query ({i}): {pretty(q)}")
    result = forward_chain_first_order(KB, q)
```

OUTPUT:

```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/ninethnewpgm.py
====================
Query (1): Ancestor(Mother(y), John)
Initial known terms: ['John']
Query: Ancestor(Mother(y), John)
------------------------------------------------------------------------

--- Iteration 1 ---
Known ground facts:
  (none yet)

New facts added this iteration:
   Ancestor(Mother(John), John)

Facts = { Ancestor(Mother(John), John) }
Terms = { Ancestor, John, Mother }

✅ Query satisfied by fact: Ancestor(Mother(John), John)
   substitution: { y: John }
Result: YES ✅

====================
Query (2): Ancestor(Mother(Mother(y)), John)
Initial known terms: ['John']
Query: Ancestor(Mother(Mother(y)), John)
------------------------------------------------------------------------

--- Iteration 1 ---
Known ground facts:
  (none yet)

New facts added this iteration:
   Ancestor(Mother(John), John)

--- Iteration 2 ---
Known ground facts:
   Ancestor(Mother(John), John)

New facts added this iteration:
   Ancestor(Mother(Ancestor), Ancestor)
   Ancestor(Mother(Mother), Mother)

--- Iteration 3 ---
Known ground facts:
   Ancestor(Mother(Ancestor), Ancestor)
   Ancestor(Mother(John), John)
   Ancestor(Mother(Mother), Mother)

No new facts inferred — stopping.
Facts = { Ancestor(Mother(Ancestor), Ancestor), Ancestor(Mother(John), John), Ancestor(Mother(Mother),
Mother)  }
Terms = { Ancestor, John, Mother }
Result: NO ❌
```

```
====================
Query (3): Ancestor(Mother(Mother(Mother(y))), Mother(y))
Initial known terms: ['John']
Query: Ancestor(Mother(Mother(Mother(y))), Mother(y))
------------------------------------------------------------------------

--- Iteration 1 ---
Known ground facts:
  (none yet)

New facts added this iteration:
   Ancestor(Mother(John), John)

--- Iteration 2 ---
Known ground facts:
   Ancestor(Mother(John), John)

New facts added this iteration:
   Ancestor(Mother(Ancestor), Ancestor)
   Ancestor(Mother(Mother), Mother)

--- Iteration 3 ---
Known ground facts:
   Ancestor(Mother(Ancestor), Ancestor)
   Ancestor(Mother(John), John)
   Ancestor(Mother(Mother), Mother)

No new facts inferred — stopping.
Facts = { Ancestor(Mother(Ancestor), Ancestor), Ancestor(Mother(John), John), Ancestor(Mother(Mother),
Mother)  }
Terms = { Ancestor, John, Mother }
Result: NO ❌

====================
Query (4): Ancestor(Mother(John), Mother(Mother(John)))
Initial known terms: ['John']
Query: Ancestor(Mother(John), Mother(Mother(John)))
------------------------------------------------------------------------

--- Iteration 1 ---
Known ground facts:
  (none yet)

New facts added this iteration:
   Ancestor(Mother(John), John)

--- Iteration 2 ---
Known ground facts:
   Ancestor(Mother(John), John)

New facts added this iteration:
   Ancestor(Mother(Ancestor), Ancestor)
   Ancestor(Mother(Mother), Mother)

--- Iteration 3 ---
Known ground facts:
   Ancestor(Mother(Ancestor), Ancestor)
   Ancestor(Mother(John), John)
   Ancestor(Mother(Mother), Mother)

No new facts inferred — stopping.
Facts = { Ancestor(Mother(Ancestor), Ancestor), Ancestor(Mother(John), John), Ancestor(Mother(Mother),
Mother)  }
Terms = { Ancestor, John, Mother }
Result: NO ❌
```

WEEK-10

```python
#to perform infreence on bn and verify conditional independence.
import itertools #for looping for generating t/f combiations

# the Bayesian Network using a dictionary
bn = {
    "Burglary": {
        "parents": [],
        "cpt": {True: 0.001, False: 0.999}
    },
    "Earthquake": {
        "parents": [],
        "cpt": {True: 0.002, False: 0.998}
    },
    "Alarm": {
        "parents": ["Burglary", "Earthquake"],
        "cpt": {
            (True, True): 0.95,
            (True, False): 0.94,
            (False, True): 0.29,
            (False, False): 0.001
        }
    },
    "JohnCalls": {
        "parents": ["Alarm"],
        "cpt": {True: 0.90, False: 0.05}
    },
    "MaryCalls": {
        "parents": ["Alarm"],
        "cpt": {True: 0.70, False: 0.01}
    }
}

# Helper function: probability of a variable given evidence
#Get the parents and CPT of the variable.
def P(var, value, evidence, bn):
    parents = bn[var]["parents"]
    cpt = bn[var]["cpt"]
    if not parents:  # root node:if no parents directly return its prior prob
        return cpt[value]
    #if parents ,truth values ina tuple  → key = (True, False)
    #create a tuple of parents
    key = tuple(evidence[p] for p in parents)
    if len(key) == 1:
        key = key[0] # from (True,) → True
    #If we ask for P(Alarm=True | Burglary, Earthquake), return that.
    # If we ask for P(Alarm=False | ...), return 1 minus that.
    p_true = cpt[key]
```

```
    return p_true if value else 1 - p_true


# Recursive enumeration over all hidden variables
#t sums over all values of the hidden variables to compute the total prob of
your query.
def enumerate_all(vars_list, evidence, bn):
    #If there are no more variables left to process, we return 1.
    if not vars_list:
        return 1.0
    #vars_list = ['Burglary', 'Earthquake', 'Alarm', 'JohnCalls', 'MaryCalls']
    # Y = 'Burglary'
    # rest = ['Earthquake', 'Alarm', 'JohnCalls', 'MaryCalls']
    Y = vars_list[0]#first var
    rest = vars_list[1:]#remaining list of vars
#We check if we already know the value of Y from the evidence.
#if evidence = { 'Burglary': True }, then when Y = 'Burglary' → known
    if Y in evidence:
        #we don't need to sum over its possible values as we alreday know
        #builds up the total joint
probability:p(x1,x2,x3)=p(x1|parents)*p(x2,x3|x1)
        #P(Alarm=True) × P(JohnCalls, MaryCalls | Alarm=True)
        return P(Y, evidence[Y], evidence, bn) * enumerate_all(rest, evidence,
bn)
    else:
        total = 0
        #marginalisatin
        for y in [True, False]:
            total += P(Y, y, evidence, bn) * enumerate_all(rest, {**evidence,
Y: y}, bn)
        return total


# Query function for any variable
def query(X, evidence, bn):
    dist = {}#dist will store the unnormalized probabilities of X=True and
X=False.
    vars_list = list(bn.keys())

    for x in [True, False]:
        #{**evidence, X: x} means:
        # take the current evidence
        # add/overwrite X with its value (True or False).
        # Then we call enumerate_all(...) to compute the total joint
probability of everything consistent with that case.
        dist[x] = enumerate_all(vars_list, {**evidence, X: x}, bn)

    # Normalize
    total = dist[True] + dist[False]
```

```
    for k in dist:
        #we convert from joint probabilities to conditional probabilities:
        dist[k] = round(dist[k] / total, 4)
    return dist



# (a)
print("P(JohnCalls | Burglary, Earthquake) =",
      query("JohnCalls", {"Burglary": True, "Earthquake": True}, bn)[True])

# (b)
print("P(Alarm | Burglary) =",
      query("Alarm", {"Burglary": True}, bn)[True])

# (c)
print("P(Earthquake | MaryCalls) =",
      query("Earthquake", {"MaryCalls": True}, bn)[True])

# (d)
print("P(Burglary | Alarm) =",
      query("Burglary", {"Alarm": True}, bn)[True])

# Compute joint P(JohnCalls, MaryCalls | Alarm)
p_joint = query("JohnCalls", {"MaryCalls": True, "Alarm": True}, bn)[True]

# Compute individual P(JohnCalls | Alarm)
p_indep = query("JohnCalls", {"Alarm": True}, bn)[True]

print("P(JohnCalls and MaryCalls | Alarm) =", p_joint)
print("P(JohnCalls | Alarm) =", p_indep)

#Once we know whether the alarm rang, does Mary calling tell us anything new
about whether John calls?
if abs(p_joint - p_indep) < 1e-3: #are equal almost
    print("JohnCalls and MaryCalls are conditionally independent given
Alarm.")
else:
    print("They are NOT independent.")
```

OUTPUT:

```
PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/tenthpgm.py
P(JohnCalls | Burglary, Earthquake) = 0.8575
P(Alarm | Burglary) = 0.94
P(Earthquake | MaryCalls) = 0.0359
P(Burglary | Alarm) = 0.3736
P(JohnCalls and MaryCalls | Alarm) = 0.9
P(JohnCalls | Alarm) = 0.9
JohnCalls and MaryCalls are conditionally independent given Alarm.
PS D:\PythonAIcodes>
```

WEEK-11

```
import random#Python's pseudo-random generator
from collections import defaultdict#like a normal dictionary, but it
automatically provides default values for new keys – avoiding KeyErrors.

class BayesianNetworkInference:
    def __init__(self, bn):
        self.bn = bn
        self.variables = list(bn.keys())#ensure bn was defined in topological
order.

    # --- Helper: sample from Bernoulli(p)
    def sample_boolean(self, prob):
        #random.random()-This function returns a random floating-point number
between 0.0 and 1.0.
        #prob-This is the probability of an event being True
        return random.random() < prob

    # --- Helper: get P(X=True | parents)
    #reads the correct probability from the CPT based on the current evidence.
    def get_prob(self, var, evidence):
        node = self.bn[var]
        #It builds a tuple of parent values in the same order listed in
node['parents'] and reads the CPT.
        parents = node['parents']
        parent_vals = tuple(evidence[p] for p in parents)
        #evidence (or partial sample) must already contain values for all
parents; otherwise KeyError.
        return node['cpt'][parent_vals]

    # PRIOR SAMPLING
```

```
    #N: number of samples to generate
    #query: the variable we want to estimate probability for
    #The goal is to estimate P(query=True) and P(query=False) from simulated
samples
    #so we generate random samples,diff cases,let us know at depends on our
var , and change accordingly in a random way,which is joint prob .
    def prior_sampling(self, N, query):
        #Will store counts like: True: number_of_times_query_was_True, False:
number_of_times_query_was_False}
        counts = defaultdict(int)
        for _ in range(N):
            sample = {}#sample will hold the truth values (True/False) of all
variables for one full sample.
           #The list self.variables must be in topological order (parents
before children), or else get_prob will fail because parent values won't be
available yet.
            for var in self.variables:
                #Looks up the CPT entry for that variable based on the
already-sampled parent values.get_prob('Alarm', sample) →
bn['Alarm']['cpt'][(True, False)] = 0.94.
                prob = self.get_prob(var, sample)
                #Performs a Bernoulli trial: returns True with probability
prob, else False
                sample[var] = self.sample_boolean(prob)
            counts[sample[query]] += 1
            #Normalize to get probabilities
        total = sum(counts.values())
        #Converts counts to probabilities:{True: count(True)/N, False:
count(False)/N}
        return {k: v / total for k, v in counts.items()}

#We now want P(query | evidence) → posterior probability (after observing
something).So, we can't just use prior sampling — we must condition on
evidence.But instead of directly conditioning during sampling,we'll generate
all samples like before, and then reject the ones that don't match the
evidence.That's why it's called rejection sampling.

    #  REJECTION SAMPLING
    def rejection_sampling(self, N, query, evidence):
        counts = defaultdict(int)
        for _ in range(N):
            sample = {}
            for var in self.variables:
                prob = self.get_prob(var, sample)
                sample[var] = self.sample_boolean(prob)

            # reject inconsistent samples
            if all(sample[e] == evidence[e] for e in evidence):
```

```
                counts[sample[query]] += 1

        total = sum(counts.values())
        if total == 0:
            return None
        return {k: v / total for k, v in counts.items()}

#Compute P(query | evidence) again – but avoid wasting samples like rejection
sampling does.
# In rejection sampling, most samples are thrown away because they don't match
the evidence → inefficient In likelihood weighting, we force the evidence
variables to stay fixed and weigh each sample by how likely that evidence was.

    # LIKELIHOOD WEIGHTING
    def likelihood_weighting(self, N, query, evidence):
        weighted_counts = defaultdict(float)
        for _ in range(N):
            weight = 1.0
            sample = {}
            for var in self.variables:
                prob = self.get_prob(var, sample)
                #If this variable is part of the evidence,we don't sample it
(we already know its value).
                #But – we adjust the weight of this sample based on how likely
that evidence was:
                if var in evidence:
                    sample[var] = evidence[var]
                    #If the evidence says var=True, multiply weight by
P(var=True | parents) If it says var=False, multiply weight by P(var=False |
parents)
                    weight *= prob if evidence[var] else (1 - prob)
                else:
                    sample[var] = self.sample_boolean(prob)
            weighted_counts[sample[query]] += weight
        total = sum(weighted_counts.values())
        return {k: v / total for k, v in weighted_counts.items()}



    # GIBBS SAMPLING-This is a Markov Chain Monte Carlo (MCMC) method – more
efficient for networks with many variables.
    #→ using sampling, but instead of generating full samples from scratch
each time,we iteratively update one variable at a time – keeping evidence
fixed.This creates a "chain" of samples that eventually reflects the true
posterior distribution.
    def gibbs_sampling(self, N, query, evidence):
        non_evidence = [v for v in self.variables if v not in evidence]
        state = dict(evidence)
        # random initialization
```

```
        for var in non_evidence:
            state[var] = random.choice([True, False])

        counts = defaultdict(int)
        for _ in range(N):
            for var in non_evidence:
                prob_true = self.markov_blanket_prob(var, state)
                state[var] = self.sample_boolean(prob_true)
            counts[state[query]] += 1
#For each iteration:
#Go through each non-evidence variable.
# Compute the probability that this variable should be True given everything
else (its Markov Blanket).
# Sample a new value for it (True/False) using that probability.
# Move on to the next variable.
        total = sum(counts.values())
        return {k: v / total for k, v in counts.items()}


    # --- Helpers for Gibbs
    def markov_blanket_prob(self, var, state):
        #It asks: "What's the probability this variable is True vs False given
current state?
        p_true = self.local_prob(var, True, state)
        p_false = self.local_prob(var, False, state)
        return p_true / (p_true + p_false)

#Computes joint probability of var=value with its children given the current
state
    def local_prob(self, var, value, state):
        #Make a copy of the current world (state) and temporarily assume that
variable has the new value (True or False).
        temp = state.copy()
        temp[var] = value
        p = self.get_prob(var, temp)
        p = p if value else (1 - p)
        # multiply over children
        for child, node in self.bn.items():
            if var in node['parents']:
                prob = self.get_prob(child, temp)
                p *= prob if temp[child] else (1 - prob)
        return p


# ---------------------------
# Example: ALARM NETWORK
# ---------------------------
bn = {
    'Burglary': {'parents': [], 'cpt': {(): 0.001}},
```

```
    'Earthquake': {'parents': [], 'cpt': {(): 0.002}},
    'Alarm': {
        'parents': ['Burglary', 'Earthquake'],
        'cpt': {
            (True, True): 0.95,
            (True, False): 0.94,
            (False, True): 0.29,
            (False, False): 0.001
        }
    },
    'JohnCalls': {'parents': ['Alarm'], 'cpt': {(True,): 0.90, (False,):
0.05}},
    'MaryCalls': {'parents': ['Alarm'], 'cpt': {(True,): 0.70, (False,):
0.01}}
}

# Create object
bn_inf = BayesianNetworkInference(bn)

# ----------------------------
# Run different sampling methods
# ----------------------------
print("🔹 Prior Sampling P(Alarm):")
print(bn_inf.prior_sampling(10000, 'Alarm'))

print("\n🔹 Rejection Sampling P(Alarm | JohnCalls=True):")
print(bn_inf.rejection_sampling(10000, 'Alarm', {'JohnCalls': True}))

print("\n🔹 Likelihood Weighting P(Alarm | JohnCalls=True):")
print(bn_inf.likelihood_weighting(10000, 'Alarm', {'JohnCalls': True}))

print("\n🔹 Gibbs Sampling P(Alarm | JohnCalls=True, MaryCalls=True):")
print(bn_inf.gibbs_sampling(5000, 'Alarm', {'JohnCalls': True, 'MaryCalls':
True}))
```

OUTPUT:


PS D:\PythonAIcodes> & C:/Users/DELL/AppData/Local/Microsoft/WindowsApps/python3.11.exe
d:/PythonAIcodes/eleventhpgm.py
🔹 Prior Sampling P(Alarm):
{False: 0.9966, True: 0.0034}
🔹 Rejection Sampling P(Alarm | JohnCalls=True):
{False: 0.9535315985130112, True: 0.046468401486988845}
🔹 Likelihood Weighting P(Alarm | JohnCalls=True):
{False: 0.9753858175424925, True: 0.024614182457507426}
🔹 Gibbs Sampling P(Alarm | JohnCalls=True, MaryCalls=True):
{False: 0.2396, True: 0.7604}