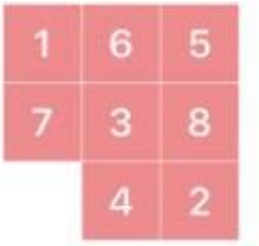
MLA0105- ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS

1) In sliding puzzle, it consists of a 3x3 grid with eight numbered tiles and one empty space. The goal of the

puzzle is to rearrange the tiles from their initial, scrambled state to a goal state where the numbers are

ordered from 1 to 8, with the empty space in the bottom-right corner. Implement the same using python.




```
import heapq
# Goal state
GOAL_STATE = ((1, 2, 3),
              (4, 5, 6),
              (7, 8, 0))
# Possible moves: Up, Down, Left, Right
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]
# Manhattan distance heuristic
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                target_x = (value - 1) // 3
                target_y = (value - 1) % 3
                distance += abs(target_x - i) + abs(target_y - j)
    return distance
```

```
# Generate neighbors
def get_neighbors(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
    neighbors = []
    for dx, dy in MOVES:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

# Solve puzzle using A* search
def solve_puzzle(start_state):
    start = tuple(tuple(row) for row in start_state)
    pq = [(manhattan_distance(start), 0, start)]
    visited = set()

    while pq:
        _, cost, state = heapq.heappop(pq)
        if state in visited:
            continue
        visited.add(state)
        if state == GOAL_STATE:
            return state
        for neighbor in get_neighbors(state):
```

```
            heapq.heappush(pq, (cost + 1 + manhattan_distance(neighbor), cost + 1, neighbor))
    return None


# Print the puzzle state
def print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else '_' for x in row))
    print()


# Take input from user
print("Enter the initial 8-puzzle state (0 for empty tile) row by row, numbers separated by spaces:")
start_state = [list(map(int, input(f"Row {i+1}: ").split())) for i in range(3)]


print("\nInitial State:")
print_state(start_state)


final_state = solve_puzzle(start_state)
if final_state:
    print("Final Solved State:")
    print_state(final_state)
else:
    print("No solution found.")
```

```
=== RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python313/ai 1.py ===
Enter the initial 8-puzzle state (0 for empty tile) row by row, numbers separated by spaces:
Row 1: 1 6 5
Row 2: 7 3 8
Row 3: 0 4 2

Initial State:
1 6 5
7 3 8
_ 4 2

Final Solved State:
1 2 3
4 5 6
7 8 _
```

2) The Towers of Hanoi problem involves three pegs (A, B, C) and a number of disks of different sizes that

can slide onto any peg. The puzzle starts with all the disks stacked on one peg in order of decreasing size,

with the largest at the bottom.

The objective of the problem is to move the entire stack to another peg, following these rules:

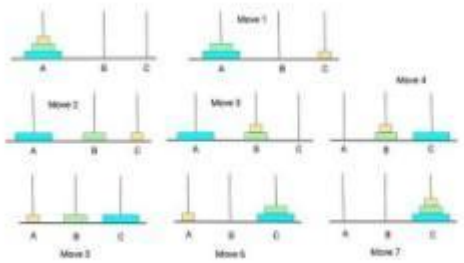
a) Only one disk can be moved at a time.

b) Each move consists of taking the upper disk from one of the stacks and placing it on top of another

stack.

c) No disk may be placed on top of a smaller disk.

Implement the above Towers of Hanoi problem using Python.



```
def towers_of_hanoi(n, source, auxiliary, target):
    if n == 1:
        print(f"Move disk 1 from {source} to {target}")
        return
    # Move n-1 disks from source to auxiliary peg
    towers_of_hanoi(n - 1, source, target, auxiliary)

    # Move the remaining disk from source to target
    print(f"Move disk {n} from {source} to {target}")
```

```
    # Move the n-1 disks from auxiliary to target peg
    towers_of_hanoi(n - 1, auxiliary, source, target)

# Main Program
num_disks = int(input("Enter the number of disks: "))
print(f"\nSteps to solve Tower of Hanoi with {num_disks} disks:\n")
towers_of_hanoi(num_disks, 'A', 'B', 'C')
```

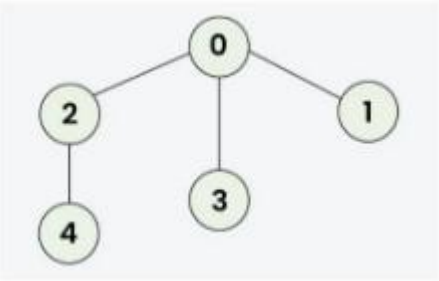
```
=== RESTART: C:/Users/LENOVO/AppData/Local/Programs/Pyt
Enter the number of disks: 3

Steps to solve Tower of Hanoi with 3 disks:

Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C
```

3) Implement a Python program to perform Breadth-First Search (BFS) on a graph. The program should:

• Represent the graph using an adjacency list.

• Start the traversal from a given source node.

• Print the order in which the nodes are visited




```
#BFS
from collections import deque


def bfs(graph, start):
```

```
    visited = []
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.append(node)
            for neighbor in graph.get(node, []):
                if neighbor not in visited:
                    queue.append(neighbor)

# ---- MAIN PROGRAM ----
graph = {}

n = int(input("Enter the number of levels in the graph: "))

print("\nEnter each node and its neighbors.")
print("Example: For node A with neighbors B and C, enter: A B C")
print("         For node 1 with neighbors 2 and 3, enter: 1 2 3\n")

for _ in range(n):
    data = input("Enter node and its neighbors: ").split()
    node = data[0]
    # Convert to int if possible
    try:
        node = int(node)
    except ValueError:
        pass
```

```
    neighbors = []
    for val in data[1:]:
        try:
            neighbors.append(int(val))
        except ValueError:
            neighbors.append(val)

    graph[node] = neighbors

# Display the graph
print("\nGraph (Adjacency List):")
for node, neighbors in graph.items():
    print(f"{node} -> {neighbors}")

# BFS Traversal
start = input("\nEnter the starting node: ")
try:
    start = int(start)
except ValueError:
    pass

if start not in graph:
    print("\nInvalid starting node!")
else:
    print(f"\nBreadth-First Search traversal starting from node {start}:")
    bfs(graph, start)
```

```
Enter the number of levels in the graph: 2

Enter each node and its neighbors.
Example: For node A with neighbors B and C, enter: A B C
         For node 1 with neighbors 2 and 3, enter: 1 2 3

Enter node and its neighbors: 0 2 3 1
Enter node and its neighbors: 2 4

Graph (Adjacency List):
0 -> [2, 3, 1]
2 -> [4]

Enter the starting node: 0

Breadth-First Search traversal starting from node 0:
0 2 3 1 4
```

4) The Monkey and Banana problem is a classic artificial intelligence problem where a monkey needs to

navigate through a room to reach a bunch of bananas hanging from the ceiling. The monkey has to move

a box to stand on it to reach the bananas. Implement the above scenario using python.

```
initial_state = {
 'monkey': 'A',
 'box': 'B',
 'monkey_on_box': False,
 'has_banana': False
}

goal_state = {
 'has_banana': True
}
```

```
banana_position = 'C'  # Bananas are hanging at position C

def display_state(state):
    print(f"Monkey at: {state['monkey']}, Box at: {state['box']}, "
          f"On Box: {state['monkey_on_box']}, Has Banana: {state['has_banana']}")

def move(state, position):
    state['monkey'] = position
    print(f"Monkey moves to {position}")
    return state

def push_box(state, position):
    if state['monkey'] == state['box']:
        state['monkey'] = position
        state['box'] = position
        print(f"Monkey pushes the box to {position}")
    else:
        print("Monkey needs to be at the box to push it!")
    return state

def climb_box(state):
    if state['monkey'] == state['box']:
        state['monkey_on_box'] = True
        print("Monkey climbs on the box")
    else:
        print("Monkey must be at the box to climb it!")
    return state

def grab_banana(state):
```

```
    if state['monkey_on_box'] and state['box'] == banana_position:
        state['has_banana'] = True
        print("Monkey grabs the banana! 🍌")
    else:
        print("Monkey cannot reach the banana yet.")
    return state

# --- Simulation ---
print("Initial State:")
display_state(initial_state)
print("\nActions:")

# Step 1: Move to the box
move(initial_state, 'B')

# Step 2: Push box to bananas
push_box(initial_state, 'C')

# Step 3: Climb onto the box
climb_box(initial_state)

# Step 4: Grab the bananas
grab_banana(initial_state)

# --- Final State ---
print("\nFinal State:")
display_state(initial_state)

if initial_state['has_banana']:
    print("\n✅ Goal achieved! Monkey got the bananas!")
```

else:

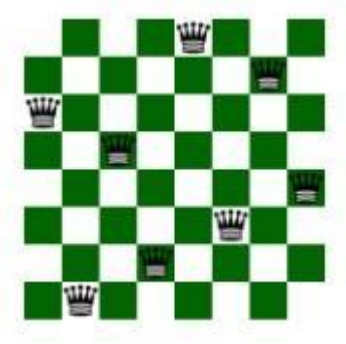
print("\n❌ Goal not achieved.")

```
Initial State:
Monkey at: A, Box at: B, On Box: False, Has Banana: False

Actions:
Monkey moves to B
Monkey pushes the box to C
Monkey climbs on the box
Monkey grabs the banana!

Final State:
Monkey at: C, Box at: C, On Box: True, Has Banana: True

☑ Goal achieved! Monkey got the bananas!
```

5) Write the python program to place eight queens on an 8 x 8 chessboard in such a way that no two queens

threaten each other. In other words, no two queens can share the same row, column, or diagonal. Use

backtracking to explore different possibilities and ensures that no two queens threaten each other.



8 Queens Problem using Backtracking

N = 8 # Chessboard size (8x8)

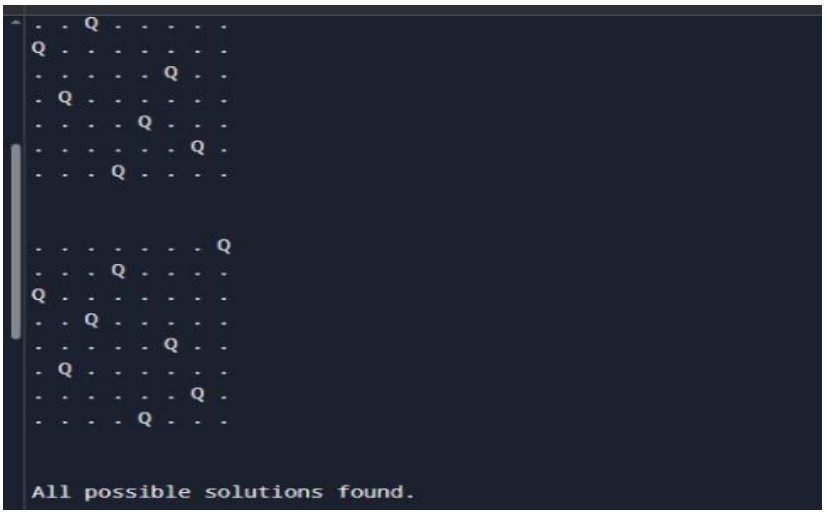
def print_solution(board):

"""Display the chessboard with queens placed safely."""

for row in board:

```
        print(" ".join("Q" if x == 1 else "." for x in row))
    print("\n")

def is_safe(board, row, col):
    """Check if a queen can be safely placed at board[row][col]."""

    # Check this column on upper rows
    for i in range(row):
        if board[i][col] == 1:
            return False

    # Check upper left diagonal
    for i, j in zip(range(row-1, -1, -1), range(col-1, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check upper right diagonal
    for i, j in zip(range(row-1, -1, -1), range(col+1, N)):
        if board[i][j] == 1:
            return False

    return True

def solve_nqueens(board, row):
    """Use backtracking to solve the N Queens problem."""
    # Base case: If all queens are placed
    if row >= N:
        print_solution(board)
        return True
```

```
    success = False
    for col in range(N):
        if is_safe(board, row, col):
            board[row][col] = 1  # Place the queen
            success = solve_nqueens(board, row + 1) or success
            board[row][col] = 0  # Backtrack (remove the queen)

    return success

def solve():
    """Initialize board and trigger solver."""
    board = [[0] * N for _ in range(N)]
    if not solve_nqueens(board, 0):
        print("No solution exists.")
    else:
        print("All possible solutions found.")

# Run the solver
solve()
```

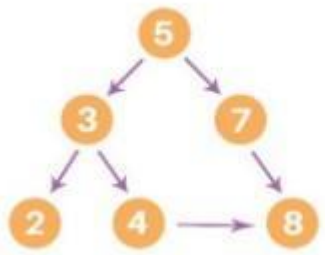


6) Implement a Python program to perform Depth-First Search (DFS) on a graph. The program should:

• Represent the graph using an adjacency list.

• Traverse the graph starting from a given source node.

• Print the order in which the nodes are visited.




```
# Depth-First Search (DFS) using adjacency list
```

Depth-First Search (DFS) using adjacency list

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    # Mark the current node as visited
    visited.add(start)
    print(start, end=" ")

    # Visit all adjacent nodes
    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example graph (Adjacency List)
graph = {
    5: [3, 7],
    3: [2, 4],
```

```
    7: [8],
    2: [],
    4: [],
    8: []
}
# Starting node
start_node = 5

print("DFS Traversal:")
dfs(graph, start_node)
```

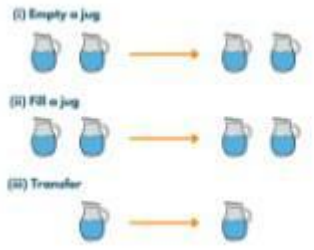
```
===============
DFS Traversal:
5 3 2 4 7 8
```

7. Solve the given problem using python. Consider a Water Jug problem: You are given two jugs, a 4-gallon

one and a 3-gallon one. Neither has any measuring marker on it. There is a pump that can be used to fill

the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug? Explicit

Assumptions: A jug can be filled from the pump, water can be poured out of a jug onto the ground, water



```
from collections import deque

def water_jug_problem():
    jugA, jugB = 4, 3
```

```
visited = set()
queue = deque([(0, 0, [])])

while queue:
    a, b, path = queue.popleft()
    if (a, b) in visited:
        continue
    visited.add((a, b))
    path = path + [(a, b)]
    if a == 2:
        print("Steps to get exactly 2 gallons in the 4-gallon jug:\n")
        for step in path:
            print(f"Jug A: {step[0]} gallons, Jug B: {step[1]} gallons")
        return

    next_states = []
    next_states.append((jugA, b))
    next_states.append((a, jugB))
    next_states.append((0, b))
    next_states.append((a, 0))
    pour = min(a, jugB - b)
    next_states.append((a - pour, b + pour))
    pour = min(b, jugA - a)
    next_states.append((a + pour, b - pour))

    # Add next states to queue
    for state in next_states:
        if state not in visited:
            queue.append((state[0], state[1], path))
```

```
    print("No solution found.")

water_jug_problem()
```

```
Steps to get exactly 2 gallons in the 4-gallon jug:

Jug A: 0 gallons, Jug B: 0 gallons
Jug A: 4 gallons, Jug B: 0 gallons
Jug A: 1 gallons, Jug B: 3 gallons
Jug A: 1 gallons, Jug B: 0 gallons
Jug A: 0 gallons, Jug B: 1 gallons
Jug A: 4 gallons, Jug B: 1 gallons
Jug A: 2 gallons, Jug B: 3 gallons
```

8. Implement a Python program to perform Backward Chaining on a set of Horn clauses. The program should:

a) Accept a knowledge base defined using Horn clauses (facts and rules).

b) Take a goal (query) as input.

c) Attempt to prove the goal using backward chaining.

d) Print whether the goal can be derived from the knowledge base.

mammal(A) ==> vertebrate(A).

vertebrate(A) ==> animal(A).

vertebrate(A),flying(A) ==> bird(A).

vertebrate("duck").

flying("duck").

mammal("cat").

```
# --------------------------
# Knowledge Base
# --------------------------

# Rules: ("head", ["body1", "body2", ...])
```

```
rules = [
    ("vertebrate(A)", ["mammal(A)"]),
    ("animal(A)", ["vertebrate(A)"]),
    ("bird(A)", ["vertebrate(A)", "flying(A)"])
]

# Facts: simple known facts
facts = [
    'vertebrate("duck")',
    'flying("duck")',
    'mammal("cat")'
]

# ----------------------------
# Unification function
# ----------------------------
def unify(term, fact):
    """
    Unify a term with a fact.
    Returns a substitution dict if successful, else None
    """
    if "(" not in term:  # simple fact
        return {} if term == fact else None

    pred1, arg1 = term.split("(")
    arg1 = arg1.rstrip(")")
    pred2, arg2 = fact.split("(")
    arg2 = arg2.rstrip(")")

    if pred1 != pred2:
```

```
        return None

    # If term has a variable (uppercase)
    if arg1.isupper():  # e.g., A
        return {arg1: arg2}
    elif arg1 == arg2:
        return {}
    else:
        return None

# ----------------------------
# Backward Chaining Algorithm
# ----------------------------
def backward_chain(goal, facts, rules, visited=None):
    if visited is None:
        visited = set()

    if goal in visited:
        return False
    visited.add(goal)

    # Check if goal is a fact
    for fact in facts:
        if unify(goal, fact) is not None:
            return True

    # Check if any rule can derive the goal
    for head, body in rules:
        substitution = unify(head, goal)
        if substitution is not None:
```

```
            # Apply substitution to body
            new_goals = []
            for b in body:
                if "(" in b:
                    pred, arg = b.split("(")
                    arg = arg.rstrip(")")
                    if arg in substitution:
                        arg = substitution[arg]
                    new_goals.append(f"{pred}({arg})")
                else:
                    new_goals.append(b)
            # Recursively check all sub-goals
            if all(backward_chain(g, facts, rules, visited) for g in new_goals):
                return True
    return False

# ---------------------------
# Main Program
# ---------------------------
if __name__ == "__main__":
    print("Backward Chaining using Horn Clauses\n")
    while True:
        query = input("Enter the goal to prove (e.g., animal(\"cat\")): ")
        result = backward_chain(query, facts, rules)
        if result:
            print(f"Goal '{query}' can be derived from the knowledge base.\n")
        else:
            print(f"Goal '{query}' cannot be derived from the knowledge base.\n")

        cont = input("Do you want to try another goal? (y/n): ")
```

if cont.lower() != 'y':

break

```
Backward Chaining using Horn Clauses

Enter the goal to prove (e.g., animal("cat")): animal("cat")
Goal 'animal("cat")' can be derived from the knowledge base.

Do you want to try another goal? (y/n): y
Enter the goal to prove (e.g., animal("cat")): bird("duck")
Goal 'bird("duck")' can be derived from the knowledge base.

Do you want to try another goal? (y/n): n
```

9) Implement a Python program to create a Feedforward Neural Network. The program should:

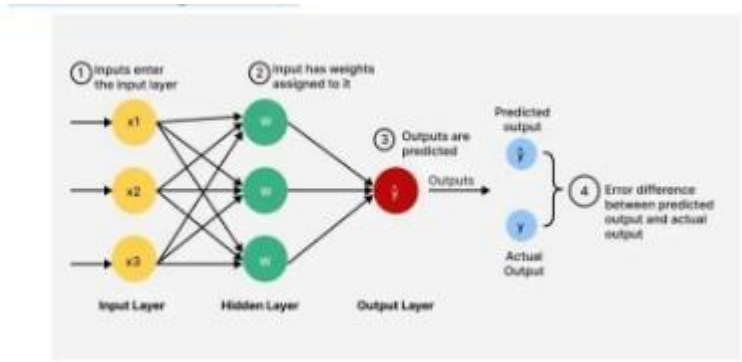
• Define an input layer, one hidden layer, and an output layer.

• Use activation functions such as Sigmoid or ReLU.

• Perform forward propagation to compute the output values.

• Demonstrate the working of the network on a small sample dataset (e.g., XOR or binary

classification problem).



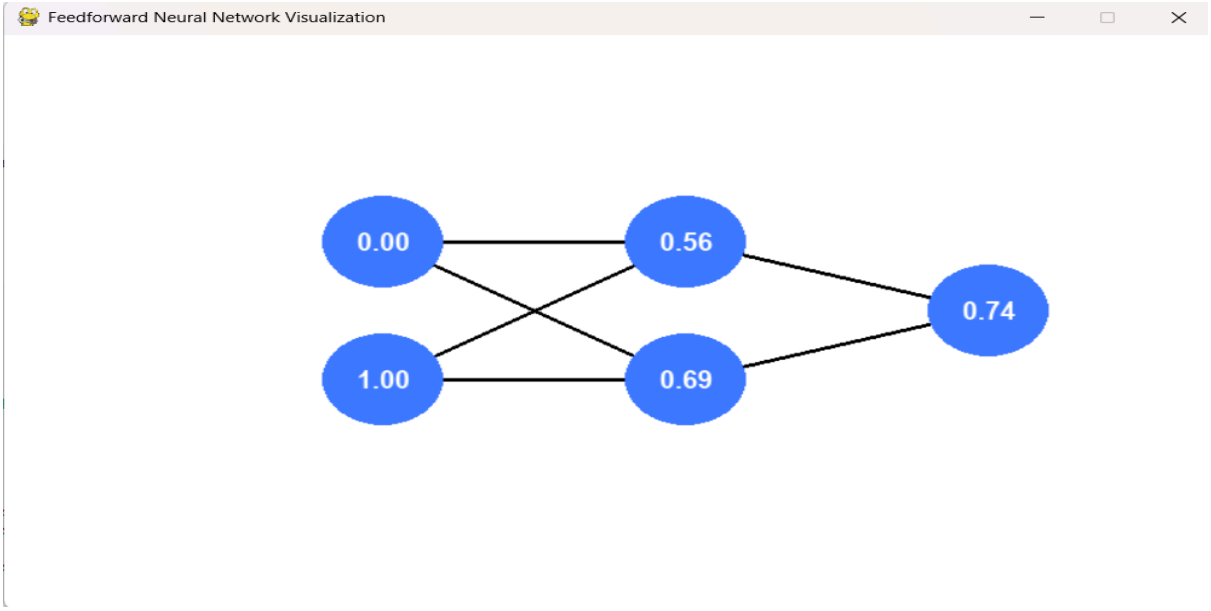

```
import pygame
import numpy as np
import sys


# ---------------------------------
# Neural Network (Simple Forward Pass)
```

```
# ---------------------------------
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

class FeedforwardNN:
    def __init__(self):
        # 2 input → 2 hidden → 1 output
        self.w1 = np.random.randn(2, 2)
        self.b1 = np.zeros((2,))

        self.w2 = np.random.randn(2, 1)
        self.b2 = np.zeros((1,))

    def forward(self, inputs):
        self.z1 = np.dot(inputs, self.w1) + self.b1
        self.a1 = sigmoid(self.z1)

        self.z2 = np.dot(self.a1, self.w2) + self.b2
        self.a2 = sigmoid(self.z2)

        return self.a1, self.a2



# ---------------------------------
# Pygame Visualization
# ---------------------------------
pygame.init()
WIDTH, HEIGHT = 800, 500
win = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Feedforward Neural Network Visualization")
```

```
font = pygame.font.SysFont("Arial", 22, bold=True)

WHITE = (255, 255, 255)
BLUE = (60, 120, 255)
BLACK = (0, 0, 0)

nn = FeedforwardNN()

# Node positions
nodes = {
    "input":  [(250, 180), (250, 300)],
    "hidden": [(450, 180), (450, 300)],
    "output": [(650, 240)]
}

def draw_node(x, y, value):
    pygame.draw.circle(win, BLUE, (x, y), 40)
    text = font.render(f"{value:.2f}", True, WHITE)
    win.blit(text, (x - text.get_width()//2, y - text.get_height()//2))

def draw_connection(p1, p2):
    pygame.draw.line(win, BLACK, p1, p2, 3)

inputs_list = [
    np.array([0, 0]),
    np.array([0, 1]),
    np.array([1, 0]),
    np.array([1, 1])
]
```

```
index = 0
clock = pygame.time.Clock()

# ---------------------------------
# Main Loop
# ---------------------------------
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    # Cycle through inputs slowly
    x_in = inputs_list[index]
    hidden_vals, out_val = nn.forward(x_in)

    index = (index + 1) % 4
    pygame.time.delay(1000)

    win.fill(WHITE)

    # Draw connections
    for i_pos in nodes["input"]:
        for h_pos in nodes["hidden"]:
            draw_connection(i_pos, h_pos)

    for h_pos in nodes["hidden"]:
        draw_connection(h_pos, nodes["output"][0])

    # Draw nodes with values
```

```
draw_node(nodes["input"][0][0], nodes["input"][0][1], x_in[0])
draw_node(nodes["input"][1][0], nodes["input"][1][1], x_in[1])

draw_node(nodes["hidden"][0][0], nodes["hidden"][0][1], hidden_vals[0])
draw_node(nodes["hidden"][1][0], nodes["hidden"][1][1], hidden_vals[1])

draw_node(nodes["output"][0][0], nodes["output"][0][1], out_val[0])

pygame.display.update()
clock.tick(60)
```



10. Implement a Python program for a Checkers AI agent using the Minimax algorithm with Alpha–Beta

pruning. The program should:

a) Represent the Checkers board and its legal moves.

b) Allow two players (Human vs AI) or (AI vs AI) to play.

c) Use Minimax with Alpha–Beta pruning for the AI's decision-making.

d) Display the board after every move and declare the winner at the end.

```
import copy
import math
import random

# --------------------------------------------------------
# Checkers Game Representation
# --------------------------------------------------------

EMPTY = "."
RED = "r"
RED_KING = "R"
BLACK = "b"
BLACK_KING = "B"

DIRECTIONS = {
    RED: [(-1, -1), (-1, 1)],
    BLACK: [(1, -1), (1, 1)],
    RED_KING: [(-1,-1),(-1,1),(1,-1),(1,1)],
    BLACK_KING: [(-1,-1),(-1,1),(1,-1),(1,1)]
}

def create_board():
    board = [[EMPTY]*8 for _ in range(8)]

    for i in range(3):
        for j in range(8):
```

```
            if (i+j)%2==1:
                board[i][j] = BLACK

    for i in range(5,8):
        for j in range(8):
            if (i+j)%2==1:
                board[i][j] = RED

    return board

def print_board(board):
    print("\n    0 1 2 3 4 5 6 7")
    print("    ----------------")
    for i, row in enumerate(board):
        print(f"{i} | " + " ".join(row))
    print()

def in_bounds(r,c):
    return 0 <= r < 8 and 0 <= c < 8

def opponent(player):
    return RED if player in (BLACK,BLACK_KING) else BLACK

def king(piece):
    return piece in (RED_KING, BLACK_KING)

def make_king(piece):
    return RED_KING if piece in (RED, RED_KING) else BLACK_KING
```

```
# ------------------------------------------------------------
# Move Generation
# ------------------------------------------------------------
def get_all_moves(board, player):
    """Return all legal moves for the current player."""
    moves = []
    captures = []

    for r in range(8):
        for c in range(8):
            if board[r][c].lower() == player:
                caps = get_piece_captures(board, r, c)
                if caps:
                    captures.extend(caps)
                else:
                    moves.extend(get_piece_moves(board, r, c))

    return captures if captures else moves

def get_piece_moves(board, r, c):
    piece = board[r][c]
    moves = []

    for dr, dc in DIRECTIONS[piece]:
        nr, nc = r + dr, c + dc
        if in_bounds(nr, nc) and board[nr][nc] == EMPTY:
            moves.append(((r,c),(nr,nc)))

    return moves
```

```
def get_piece_captures(board, r, c):
    piece = board[r][c]
    captures = []

    for dr, dc in DIRECTIONS[piece]:
        mid_r, mid_c = r + dr, c + dc
        end_r, end_c = r + 2*dr, c + 2*dc

        if (in_bounds(mid_r, mid_c) and in_bounds(end_r, end_c) and
            board[mid_r][mid_c].lower() == opponent(piece) and
            board[end_r][end_c] == EMPTY):

            # Perform capture simulation
            new_board = copy.deepcopy(board)
            move_piece(new_board, (r,c), (end_r,end_c))
            new_board[mid_r][mid_c] = EMPTY

            # Chain captures
            further = get_piece_captures(new_board, end_r, end_c)
            if further:
                captures.extend([( (r,c), (end_r,end_c), chain ) for chain in further])
            else:
                captures.append(((r,c),(end_r,end_c)))

    return captures

def move_piece(board, start, end):
    (r,c),(nr,nc) = start,end
    piece = board[r][c]
    board[r][c] = EMPTY
```

```
    # Promotion
    if piece == RED and nr == 0:
        piece = RED_KING
    if piece == BLACK and nr == 7:
        piece = BLACK_KING

    board[nr][nc] = piece



# ------------------------------------------------------------
# Winner Check
# ------------------------------------------------------------
def check_winner(board):
    red_exists = any(piece in (RED, RED_KING) for row in board for piece in row)
    black_exists = any(piece in (BLACK, BLACK_KING) for row in board for piece in row)
    if not red_exists:
        return "BLACK wins"
    if not black_exists:
        return "RED wins"
    return None



# ------------------------------------------------------------
# Evaluation Function (Heuristic)
# ------------------------------------------------------------
def evaluate(board):
    score = 0
    for row in board:
        for p in row:
```

```
            if p == RED: score += 1
            elif p == RED_KING: score += 2
            elif p == BLACK: score -= 1
            elif p == BLACK_KING: score -= 2
    return score


# ------------------------------------------------------------
# Minimax with Alpha-Beta
# ------------------------------------------------------------
def minimax(board, depth, alpha, beta, maximizing, player):
    win = check_winner(board)
    if win or depth == 0:
        return evaluate(board), None

    moves = get_all_moves(board, player)

    if not moves:
        return evaluate(board), None

    best_move = None

    if maximizing:
        max_eval = -math.inf
        for mv in moves:
            new_board = simulate_move(board, mv)
            ev, _ = minimax(new_board, depth-1, alpha, beta, False, opponent(player))
            if ev > max_eval:
                max_eval = ev
                best_move = mv
```

```
            alpha = max(alpha, ev)
            if beta <= alpha:
                break
        return max_eval, best_move

    else:
        min_eval = math.inf
        for mv in moves:
            new_board = simulate_move(board, mv)
            ev, _ = minimax(new_board, depth-1, alpha, beta, True, opponent(player))
            if ev < min_eval:
                min_eval = ev
                best_move = mv

            beta = min(beta, ev)
            if beta <= alpha:
                break
        return min_eval, best_move

def simulate_move(board, move):
    new_board = copy.deepcopy(board)

    if len(move) == 2:
        # Normal move
        move_piece(new_board, move[0], move[1])
    else:
        # Capture
        (r,c),(nr,nc), chain = move
        move_piece(new_board, (r,c), (nr,nc))
```

```
            mid_r = (r+nr)//2
            mid_c = (c+nc)//2
            new_board[mid_r][mid_c] = EMPTY
        return new_board


# ------------------------------------------------------------
# Main Game Loop
# ------------------------------------------------------------
def play_game():
    board = create_board()
    print_board(board)

    mode = input("Choose mode (1 = Human vs AI, 2 = AI vs AI): ")

    current_player = RED

    while True:
        winner = check_winner(board)
        if winner:
            print(winner)
            break

        print(f"{current_player.upper()}'s turn")
        print_board(board)

        moves = get_all_moves(board, current_player)

        if not moves:
            print(f"{opponent(current_player).upper()} wins!")
```

```
            break

        if mode == "1" and current_player == RED:
            # HUMAN MOVE
            print("Available moves (start_r start_c end_r end_c):")
            for mv in moves:
                print(mv)

            sr = int(input("Start row: "))
            sc = int(input("Start col: "))
            er = int(input("End row: "))
            ec = int(input("End col: "))

            chosen = ((sr,sc),(er,ec))
        else:
            print("AI thinking...")
            _, chosen = minimax(board, 4, -math.inf, math.inf, True, current_player)

        # Apply the move
        board = simulate_move(board, chosen)

        current_player = opponent(current_player)

# ------------------------------------------------------------
# Run the game
# ------------------------------------------------------------
play_game()
```

```
     0 1 2 3 4 5 6 7
     ---------------
0  |  . b . b . b . b
1  |  b . b . b . b .
2  |  . b . b . b . b
3  |  . . . . . . . .
4  |  . . . . . . . .
5  |  r . r . r . r .
6  |  . r . r . r . r
7  |  r . r . r . r .

Choose mode (1 = Human vs AI, 2 = AI vs AI): 1
R's turn

     0 1 2 3 4 5 6 7
     ---------------
0  |  . b . b . b . b
1  |  b . b . b . b .
2  |  . b . b . b . b
3  |  . . . . . . . .
4  |  . . . . . . . .
5  |  r . r . r . r .
6  |  . r . r . r . r
7  |  r . r . r . r .

Available moves (start_r start_c end_r end_c):
((5, 0), (4, 1))
((5, 2), (4, 1))
((5, 2), (4, 3))
((5, 4), (4, 3))
((5, 4), (4, 5))
((5, 6), (4, 5))
((5, 6), (4, 7))
Start row: 2
Start col: 4
End row: 4
End col: 4
B's turn
```

```
     0 1 2 3 4 5 6 7
     ---------------
0  |  . b . b . b . b
1  |  b . b . b . b .
2  |  . b . b . b . b
3  |  . . . . . . . .
4  |  . . . . . . . .
5  |  r . r . r . r .
6  |  . r . r . r . r
7  |  r . r . r . r .

AI thinking...
R's turn

     0 1 2 3 4 5 6 7
     ---------------
0  |  . b . b . b . b
1  |  b . b . b . b .
2  |  . . . b . b . b
3  |  b . . . . . . .
4  |  . . . . . . . .
5  |  r . r . r . r .
6  |  . r . r . r . r
7  |  r . r . r . r .

Available moves (start_r start_c end_r end_c):
((5, 0), (4, 1))
((5, 2), (4, 1))
((5, 2), (4, 3))
((5, 4), (4, 3))
((5, 4), (4, 5))
((5, 6), (4, 5))
((5, 6), (4, 7))
Start row:
```

11. Implement a Python program to build and evaluate a Decision Tree classifier.

The program should :

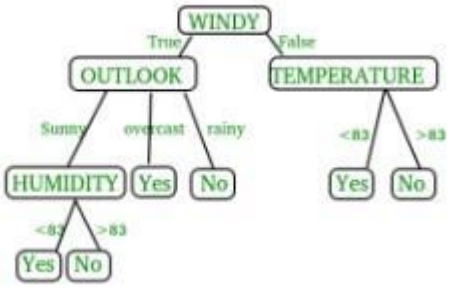
a) Take a dataset as input from the given figure.

b) Build the Decision Tree using a splitting criterion.

c) Display the tree structure in a readable form (rules or indented hierarchy).

d) Classify new instances based on the trained tree.

e) Print the accuracy of the model on test data.




```
# Dataset
dataset = [
    ['Sunny', 'Hot', 'High', 'False', 'No'],
    ['Sunny', 'Hot', 'High', 'True', 'No'],
    ['Overcast', 'Hot', 'High', 'False', 'Yes'],
    ['Rainy', 'Mild', 'High', 'False', 'Yes'],
    ['Rainy', 'Cool', 'Normal', 'False', 'Yes'],
    ['Rainy', 'Cool', 'Normal', 'True', 'No'],
    ['Overcast', 'Cool', 'Normal', 'True', 'Yes'],
    ['Sunny', 'Mild', 'High', 'False', 'No'],
    ['Sunny', 'Cool', 'Normal', 'False', 'Yes'],
    ['Rainy', 'Mild', 'Normal', 'False', 'Yes'],
    ['Sunny', 'Mild', 'Normal', 'True', 'Yes'],
    ['Overcast', 'Mild', 'High', 'True', 'Yes'],
    ['Overcast', 'Hot', 'Normal', 'False', 'Yes'],
    ['Rainy', 'Mild', 'High', 'True', 'No']
]
```

```
features = ['Outlook', 'Temperature', 'Humidity', 'Windy']

import math

# Step 1: Entropy function
def entropy(rows):
    total = len(rows)
    yes = len([r for r in rows if r[-1] == 'Yes'])
    no = total - yes
    if yes == 0 or no == 0:
        return 0
    p1 = yes / total
    p2 = no / total
    return -p1 * math.log2(p1) - p2 * math.log2(p2)

# Step 2: Split dataset
def split_dataset(rows, column, value):
    set1 = [r for r in rows if r[column] == value]
    set2 = [r for r in rows if r[column] != value]
    return set1, set2

# Step 3: Find best split
def best_split(rows):
    best_gain = 0
    best_column = None
    best_value = None
    current_entropy = entropy(rows)
    n_features = len(rows[0]) - 1
```

```
    for col in range(n_features):
        values = set([r[col] for r in rows])
        for v in values:
            set1, set2 = split_dataset(rows, col, v)
            if len(set1) == 0 or len(set2) == 0:
                continue

            p = len(set1) / len(rows)
            gain = current_entropy - p * entropy(set1) - (1 - p) * entropy(set2)

            if gain > best_gain:
                best_gain = gain
                best_column = col
                best_value = v

    return best_gain, best_column, best_value


# Step 4: Node class
class DecisionNode:
    def __init__(self, column=None, value=None, true_branch=None, false_branch=None,
result=None):
        self.column = column
        self.value = value
        self.true_branch = true_branch
        self.false_branch = false_branch
        self.result = result

# Step 5: Build tree
def build_tree(rows):
    gain, column, value = best_split(rows)
    if gain == 0:
```

```
        yes = len([r for r in rows if r[-1] == 'Yes'])
        no = len(rows) - yes
        return DecisionNode(result='Yes' if yes >= no else 'No')

    set1, set2 = split_dataset(rows, column, value)
    true_branch = build_tree(set1)
    false_branch = build_tree(set2)

    return DecisionNode(column, value, true_branch, false_branch)

# Step 6: Print tree
def print_tree(node, spacing=""):
    if node.result is not None:
        print(spacing + "Predict:", node.result)
        return

    print(f"{spacing}{features[node.column]} == {node.value}?")
    print(spacing + "-> True:")
    print_tree(node.true_branch, spacing + "    ")
    print(spacing + "-> False:")
    print_tree(node.false_branch, spacing + "    ")

# Step 7: Classify new instance
def classify(observation, node):
    if node.result is not None:
        return node.result

    if observation[node.column] == node.value:
        return classify(observation, node.true_branch)
    else:
```

```
        return classify(observation, node.false_branch)

# Step 8: Build the tree
tree = build_tree(dataset)

print("\n=== DECISION TREE STRUCTURE ===\n")
print_tree(tree)

# Step 9: Test classification
test_data = ['Rainy', 'Mild', 'Normal', 'False']   # Expected Yes
prediction = classify(test_data, tree)

print("\nPrediction for", test_data, "->", prediction)

# Step 10: Accuracy test
correct = 0
for row in dataset:
    pred = classify(row[:-1], tree)
    if pred == row[-1]:
        correct += 1

accuracy = correct / len(dataset) * 100
print("\nAccuracy on training data:", accuracy, "%")
```

```
=== DECISION TREE STRUCTURE ===

Outlook == Overcast?
-> True:
   Predict: Yes
-> False:
   Humidity == Normal?
   -> True:
      Windy == True?
      -> True:
         Outlook == Sunny?
         -> True:
            Predict: Yes
         -> False:
            Predict: No
      -> False:
         Predict: Yes
   -> False:
      Outlook == Sunny?
      -> True:
         Predict: No
      -> False:
         Windy == True?
         -> True:
            Predict: No
         -> False:
            Predict: Yes

Prediction for ['Rainy', 'Mild', 'Normal', 'False'] -> Yes

Accuracy on training data: 100.0 %
```

12. It's a river-crossing puzzle that involves three missionaries and three cannibals who need to cross a river

from one side to another. The problem comes with certain constraints:

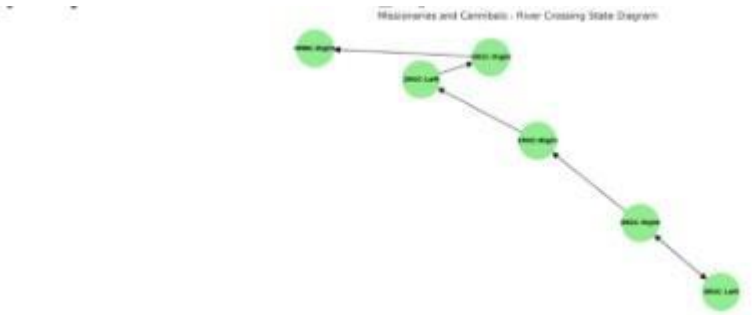
a) There is a boat that can carry at most two people at a time.

b) On either side of the river, if the number of cannibals ever exceeds the number of missionaries, the

cannibals will eat the missionaries, and the mission will fail.

c) The goal is to find a series of moves that allow all the missionaries and cannibals to cross the river

safely. Implement the scenario using Python.




```
from collections import deque
start_state = (3, 3, 'left')
goal_state = (0, 0, 'right')
def is_valid_state(m_left, c_left, boat_pos):
    m_right = 3 - m_left
    c_right = 3 - c_left
    if not (0 <= m_left <= 3 and 0 <= c_left <= 3):
        return False
    if (m_left > 0 and m_left < c_left) or (m_right > 0 and m_right < c_right):
        return False
    return True
def get_successors(state):
    m_left, c_left, boat_pos = state
    successors = []
    moves = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]

    for m_move, c_move in moves:
        if boat_pos == 'left':
            new_state = (m_left - m_move, c_left - c_move, 'right')
        else:
            new_state = (m_left + m_move, c_left + c_move, 'left')

        if is_valid_state(*new_state):
            successors.append(new_state)
```

```
    return successors

# BFS to find the shortest path
def bfs(start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        current_state, path = queue.popleft()

        if current_state == goal:
            return path

        visited.add(current_state)

        for next_state in get_successors(current_state):
            if next_state not in visited:
                queue.append((next_state, path + [next_state]))

    return None

# Run the BFS search
solution = bfs(start_state, goal_state)

# Print the solution
if solution:
    print("Solution found! Sequence of states:")
    for step in solution:
        print(step)
else:
```

```
    print("No solution found.")
```

```
Solution found! Sequence of states:
(3, 3, 'left')
(3, 1, 'right')
(3, 2, 'left')
(3, 0, 'right')
(3, 1, 'left')
(1, 1, 'right')
(2, 2, 'left')
(0, 2, 'right')
(0, 3, 'left')
(0, 1, 'right')
(1, 1, 'left')
(0, 0, 'right')
```

13. Implement a Python program to simulate the Tic Tac Toe game. The program should:

a) Represent the 3×3 game board.

b) Allow a human player to play against an AI player.

c) Print the board after each move.

d) Declare the winner or draw at the end of the game

```
import random


def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board, player):
    # Check rows and columns
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):  # row
            return True
        if all(board[j][i] == player for j in range(3)):  # column
```

```
            return True

    # Check diagonals
    if all(board[i][i] == player for i in range(3)):  # main diagonal
        return True
    if all(board[i][2 - i] == player for i in range(3)):  # anti-diagonal
        return True

    return False

def board_full(board):
    return all(board[i][j] != ' ' for i in range(3) for j in range(3))

def human_move(board):
    while True:
        try:
            move = input("Enter your move as row and column (0 1 2) separated by space: ")
            row, col = map(int, move.split())
            if row in range(3) and col in range(3) and board[row][col] == ' ':
                board[row][col] = 'X'
                break
            else:
                print("Invalid move, cell already occupied or out of bounds.")
        except:
            print("Invalid input. Please enter two numbers separated by space.")

def ai_move(board):
    # Simple AI: random move
    available = [(i, j) for i in range(3) for j in range(3) if board[i][j] == ' ']
    if available:
```

```
        row, col = random.choice(available)
        board[row][col] = 'O'

def tic_tac_toe():
    board = [[' ' for _ in range(3)] for _ in range(3)]
    print_board(board)

    while True:
        human_move(board)
        print_board(board)

        if check_winner(board, 'X'):
            print("Human wins!")
            break

        if board_full(board):
            print("It's a draw!")
            break

        ai_move(board)
        print("AI's move:")
        print_board(board)

        if check_winner(board, 'O'):
            print("AI wins!")
            break

        if board_full(board):
            print("It's a draw!")
            break
```

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

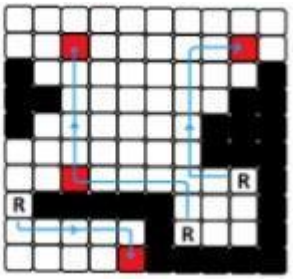
```
  |   |
---------
  |   |
---------
  |   |
---------
Enter your move as row and column (0 1 2) separated by space: 1
Invalid input. Please enter two numbers separated by space.
Enter your move as row and column (0 1 2) separated by space: 1 2
  |   |
---------
  |   | X
---------
  |   |
---------
AI's move:
  |   |
---------
  | O | X
---------
  |   |
---------
Enter your move as row and column (0 1 2) separated by space: 2 1
  |   |
---------
  | O | X
---------
  | X |
---------
AI's move:
  | O |
---------
  | O | X
---------
  | X |
---------
Enter your move as row and column (0 1 2) separated by space: 0 0
X | O |
---------
  | O | X
---------
  | X |
```

14. Imagine you have a simple autonomous vacuum cleaner, which can move around in a grid-based

environment. The grid is divided into cells, some of which are dirty, and some are clean. The vacuum

cleaner can move in four directions: up, down, left, and right. Its primary task is to clean all the dirty cells

while minimizing its movements. Implement the above AI problem using python.



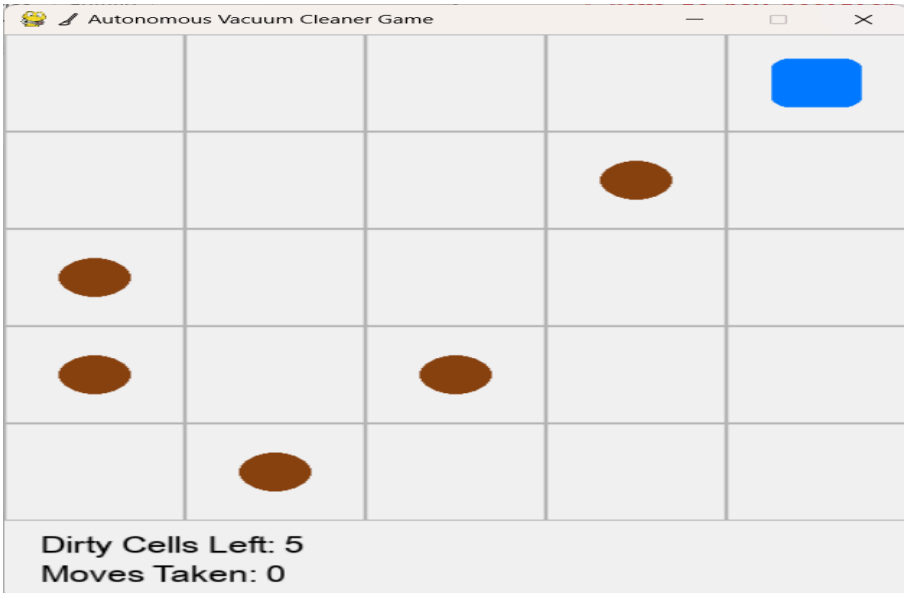

```
import pygame
import random
import sys

# Initialize pygame
pygame.init()

# Grid setup
ROWS, COLS = 5, 5
CELL_SIZE = 100
WIDTH, HEIGHT = COLS * CELL_SIZE, ROWS * CELL_SIZE
FPS = 10

# Colors
WHITE = (240, 240, 240)
BLACK = (0, 0, 0)
GREEN = (0, 200, 0)
BROWN = (139, 69, 19)
BLUE = (0, 120, 255)
```

```
GRAY = (180, 180, 180)

# Initialize window
screen = pygame.display.set_mode((WIDTH, HEIGHT + 80))
pygame.display.set_caption("🧹 Autonomous Vacuum Cleaner Game")

font = pygame.font.SysFont("arial", 24)

# Initialize grid with random dirty cells
grid = [['.' for _ in range(COLS)] for _ in range(ROWS)]
num_dirty = random.randint(5, 8)
dirty_cells = random.sample([(r, c) for r in range(ROWS) for c in range(COLS)], num_dirty)
for r, c in dirty_cells:
    grid[r][c] = 'D'

# Place vacuum at random position
vacuum_r, vacuum_c = random.choice([(r, c) for r in range(ROWS) for c in range(COLS)])
grid[vacuum_r][vacuum_c] = 'V'

moves = 0

def draw_grid():
    """Draws the game grid and elements."""
    for r in range(ROWS):
        for c in range(COLS):
            x, y = c * CELL_SIZE, r * CELL_SIZE
            pygame.draw.rect(screen, GRAY, (x, y, CELL_SIZE, CELL_SIZE), 1)

            if grid[r][c] == 'D':
                pygame.draw.circle(screen, BROWN, (x + CELL_SIZE//2, y + CELL_SIZE//2),
20)
```

```
            elif grid[r][c] == 'V':
                pygame.draw.rect(screen, BLUE, (x + 25, y + 25, 50, 50), border_radius=10)

    # Draw text area
    pygame.draw.rect(screen, WHITE, (0, HEIGHT, WIDTH, 80))
    dirty_count = sum(row.count('D') for row in grid)
    text1 = font.render(f"Dirty Cells Left: {dirty_count}", True, BLACK)
    text2 = font.render(f"Moves Taken: {moves}", True, BLACK)
    screen.blit(text1, (20, HEIGHT + 10))
    screen.blit(text2, (20, HEIGHT + 40))

def move_vacuum(dr, dc):
    """Moves the vacuum cleaner on the grid."""
    global vacuum_r, vacuum_c, moves

    new_r = vacuum_r + dr
    new_c = vacuum_c + dc
    if 0 <= new_r < ROWS and 0 <= new_c < COLS:
        # Clear old position
        grid[vacuum_r][vacuum_c] = '.'
        # Move to new position
        vacuum_r, vacuum_c = new_r, new_c
        # Clean if dirty
        if grid[new_r][new_c] == 'D':
            grid[new_r][new_c] = 'V'
        else:
            grid[new_r][new_c] = 'V'
        moves += 1

# Main game loop
```

```
clock = pygame.time.Clock()
running = True
while running:
    screen.fill(WHITE)
    draw_grid()
    pygame.display.flip()

    dirty_left = sum(row.count('D') for row in grid)
    if dirty_left == 0:
        screen.fill(WHITE)
        win_text = font.render("✅ All cells are clean! Well done!", True, GREEN)
        screen.blit(win_text, (60, HEIGHT // 2 - 30))
        pygame.display.flip()
        pygame.time.wait(2500)
        running = False
        continue

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            if event.key == pygame.K_UP:
                move_vacuum(-1, 0)
            elif event.key == pygame.K_DOWN:
                move_vacuum(1, 0)
            elif event.key == pygame.K_LEFT:
                move_vacuum(0, -1)
            elif event.key == pygame.K_RIGHT:
                move_vacuum(0, 1)
```

clock.tick(FPS)



15. Implement a Python program to solve the N-Queens problem using Hill Climbing. The program should:

a) Represent a state as positions of N queens (one per column).

b) Define a cost function = number of attacking pairs of queens.

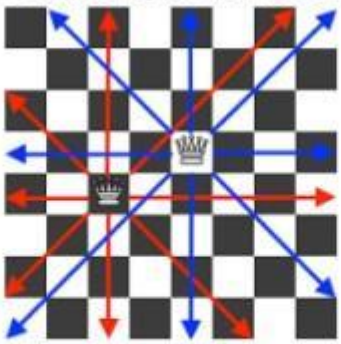
c) Generate neighbors by moving one queen within its column to a different row.

d) Move to the best neighbor if it strictly improves the cost.

e) Stop when no better neighbor exists (local optimum).

f) (Optional) Use random restarts to increase the chance of finding a global optimum.

g) Print the final board, the final cost, and whether a valid solution (cost = 0) was found.



```
 n queens problem using hill climbing
import random

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

# Cost function: number of attacking pairs
def compute_cost(state):
    attacks = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacks += 1
    return attacks

# Generate all neighbors by moving one queen in its column
def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):
```

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

# Hill Climbing Algorithm
def hill_climbing(n, max_restarts=10):
    best_overall = None
    best_cost = float("inf")

    for restart in range(max_restarts):
        current = [random.randint(0, n - 1) for _ in range(n)]
        current_cost = compute_cost(current)

        while True:
            neighbors = get_neighbors(current)
            neighbor_costs = [compute_cost(neighbor) for neighbor in neighbors]
            min_cost = min(neighbor_costs)
            if min_cost >= current_cost:  # Local optimum
                break
            best_neighbor = neighbors[neighbor_costs.index(min_cost)]
            current, current_cost = best_neighbor, min_cost

        # Track the best solution across restarts
        if current_cost < best_cost:
            best_cost = current_cost
            best_overall = current
```

```
        # If a valid solution is found
        if best_cost == 0:
            break


    print("Final Board:")
    print_board(best_overall)
    print("Final Cost:", best_cost)
    if best_cost == 0:
        print("✅ Solution Found!")
    else:
        print("⚠ Local Optimum Reached.")


# Run the algorithm
n = 8  # You can change N here
hill_climbing(n)
```

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

Final Cost: 0
☑ Solution Found!
```

16. You are given a map with regions (represented as nodes or vertices) that need to be colored such that no

adjacent regions have the same color. The goal is to find a valid coloring for the entire map. Implement in

graph using python.

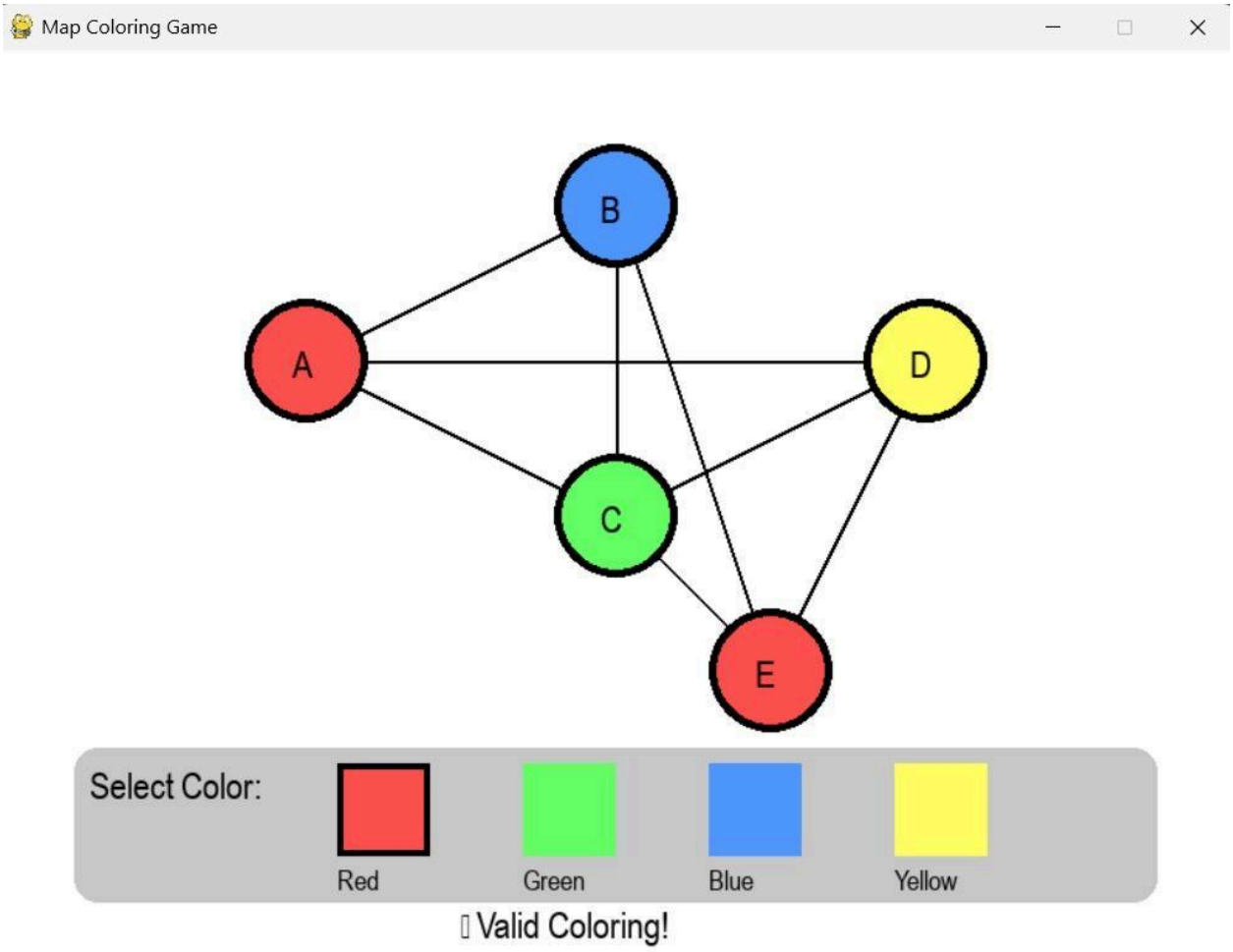

```
import pygame
import sys
import math

# --- Initialize Pygame ---
pygame.init()
WIDTH, HEIGHT = 800, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Map Coloring Game")

FONT = pygame.font.SysFont("arial", 24)
SMALL_FONT = pygame.font.SysFont("arial", 18)

# --- Colors ---
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (200, 200, 200)
RED = (255, 80, 80)
GREEN = (100, 255, 100)
BLUE = (80, 150, 255)
YELLOW = (255, 255, 100)
COLORS = {"Red": RED, "Green": GREEN, "Blue": BLUE, "Yellow": YELLOW}
color_names = list(COLORS.keys())
```

```
# --- Graph (Map) Definition ---
graph = {
    'A': ['B', 'C', 'D'],
    'B': ['A', 'C', 'E'],
    'C': ['A', 'B', 'D', 'E'],
    'D': ['A', 'C', 'E'],
    'E': ['B', 'C', 'D']
}

# --- Node positions (for drawing) ---
positions = {
    'A': (200, 200),
    'B': (400, 100),
    'C': (400, 300),
    'D': (600, 200),
    'E': (500, 400)
}

# --- Node colors ---
node_colors = {node: None for node in graph}
selected_color = "Red"

# --- Function to draw the graph ---
def draw_graph():
    screen.fill(WHITE)

    # Draw edges
    for node, neighbors in graph.items():
        for neighbor in neighbors:
            start = positions[node]
```

```
        end = positions[neighbor]
        pygame.draw.line(screen, BLACK, start, end, 2)

# Draw nodes
for node, (x, y) in positions.items():
    color = node_colors[node]
    pygame.draw.circle(screen, BLACK, (x, y), 40)
    pygame.draw.circle(screen, color if color else GRAY, (x, y), 35)
    text = FONT.render(node, True, BLACK)
    screen.blit(text, (x - 10, y - 12))

# Draw color palette
pygame.draw.rect(screen, GRAY, (50, 450, 700, 100), border_radius=15)
label = FONT.render("Select Color:", True, BLACK)
screen.blit(label, (60, 460))

for i, cname in enumerate(color_names):
    rect = pygame.Rect(220 + i * 120, 460, 60, 60)
    pygame.draw.rect(screen, COLORS[cname], rect)
    if cname == selected_color:
        pygame.draw.rect(screen, BLACK, rect, 4)

    txt = SMALL_FONT.render(cname, True, BLACK)
    screen.blit(txt, (220 + i * 120, 525))

# Display check result
result_text = check_coloring()
result_label = FONT.render(result_text, True, BLACK)
screen.blit(result_label, (300, 550))
```

```
    pygame.display.flip()

# --- Check coloring validity ---
def check_coloring():
    for node, neighbors in graph.items():
        for n in neighbors:
            if node_colors[node] and node_colors[n] == node_colors[node]:
                return "❌ Invalid Coloring!"
    if all(node_colors[n] for n in node_colors):
        return "✅ Valid Coloring!"
    return "🟡 In Progress..."

# --- Detect if click is inside node ---
def get_clicked_node(pos):
    for node, (x, y) in positions.items():
        if math.dist(pos, (x, y)) < 40:
            return node
    return None

# --- Main Loop ---
running = True
while running:
    draw_graph()
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        # Color selection
        if event.type == pygame.MOUSEBUTTONDOWN:
            mx, my = event.pos
```

```
        # Check if clicked on color palette
        for i, cname in enumerate(color_names):
            rect = pygame.Rect(220 + i * 120, 460, 60, 60)
            if rect.collidepoint(mx, my):
                selected_color = cname

        # Check if clicked on node
        node = get_clicked_node((mx, my))
        if node:
            node_colors[node] = COLORS[selected_color]

pygame.quit()
sys.exit()
```



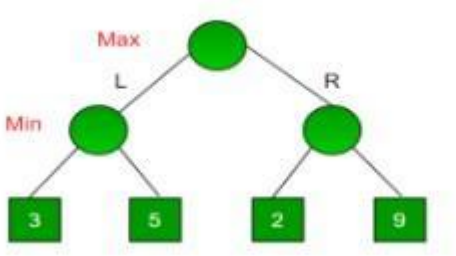
17. Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a

perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance

to move, i.e., you are at the root and your opponent at next level. Using python create the program,

which move you would make as a maximizing player considering that your opponent also plays

optimally?



Minimax Algorithm Example

```
def minimax(depth, node_index, is_maximizing, values):
    """
    Recursive function to apply the minimax algorithm.

    depth: current depth in the game tree
    node_index: index of the current node
    is_maximizing: True if it's the maximizing player's turn, False for minimizing player
    values: list of leaf node values
    """
    # Base case: if we reach a leaf node
    if depth == 2:
        return values[node_index]

    # If it's the maximizing player's turn
    if is_maximizing:
```

```
        return max(
            minimax(depth + 1, node_index * 2, False, values),
            minimax(depth + 1, node_index * 2 + 1, False, values)
        )
    # If it's the minimizing player's turn
    else:
        return min(
            minimax(depth + 1, node_index * 2, True, values),
            minimax(depth + 1, node_index * 2 + 1, True, values)
        )

# ------------------------------------------------------------
# Leaf nodes (final states)
values = [3, 5, 2, 9]

# Depth of tree = 2 (Root -> Min -> Leaf)
optimal_value = minimax(0, 0, True, values)
print("The optimal value for the maximizing player is:", optimal_value)

# ------------------------------------------------------------
# Determine which move (L or R) the maximizing player should take
left_subtree = min(values[0], values[1])  # Min node under Left
right_subtree = min(values[2], values[3]) # Min node under Right

best_move = "Left" if max(left_subtree, right_subtree) == left_subtree else "Right"
print("The maximizing player should choose:", best_move)
```

```
----- RESTART: C:/Users/LENOVO/AppData/Local/Program
The optimal value for the maximizing player is: 3
The maximizing player should choose: Left
```

18. Implement a Python program to solve a pathfinding problem using the A* algorithm.

The program should:

a) Represent the environment as a graph or a grid with weighted edges.

b) Take a start node and a goal node as inputs.

c) Use a suitable heuristic function (e.g., Manhattan distance for grids, straight-line distance for

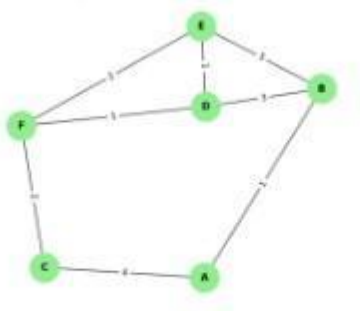
graphs).

d) Explore nodes using the A* evaluation function.

e) Print the order of node expansion and the final optimal path with its total cost.

f) Start Node: A Goal Node: F Nodes: A, B, C, D, E, F Edges with weights: (A, B, 1), (A, C, 4),

(B, D, 3), (B, E, 5), (C, F, 2), (D, F, 1), (D, E, 1), (E, F, 2).



```
import pygame
import sys
import math
import heapq


# --- Initialize Pygame ---
pygame.init()
WIDTH, HEIGHT = 800, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("A* Pathfinding Game")


FONT = pygame.font.SysFont("arial", 24)
```

```
SMALL_FONT = pygame.font.SysFont("arial", 18)

# --- Colors ---
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (200, 200, 200)
BLUE = (80, 150, 255)
GREEN = (100, 255, 100)
YELLOW = (255, 255, 100)
ORANGE = (255, 180, 80)
PURPLE = (180, 100, 255)
RED = (255, 80, 80)

# --- Graph Definition ---
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'D': 3, 'E': 5},
    'C': {'A': 4, 'F': 2},
    'D': {'B': 3, 'E': 1, 'F': 1},
    'E': {'B': 5, 'D': 1, 'F': 2},
    'F': {'C': 2, 'D': 1, 'E': 2}
}

positions = {
    'A': (150, 250),
    'B': (300, 150),
    'C': (300, 350),
    'D': (500, 150),
    'E': (500, 350),
    'F': (650, 250)
```

```
}

start_node = None
goal_node = None

# --- Heuristic ---
def heuristic(n1, n2):
    (x1, y1), (x2, y2) = positions[n1], positions[n2]
    return math.hypot(x2 - x1, y2 - y1)

# --- A* Algorithm ---
def astar(start, goal):
    open_heap = []
    heapq.heappush(open_heap, (0, start))
    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0
    f_score = {node: float('inf') for node in graph}
    f_score[start] = heuristic(start, goal)
    visited_order = []

    while open_heap:
        current = heapq.heappop(open_heap)[1]
        visited_order.append(current)

        if current == goal:
            break

        for neighbor, weight in graph[current].items():
            tentative_g = g_score[current] + weight
```

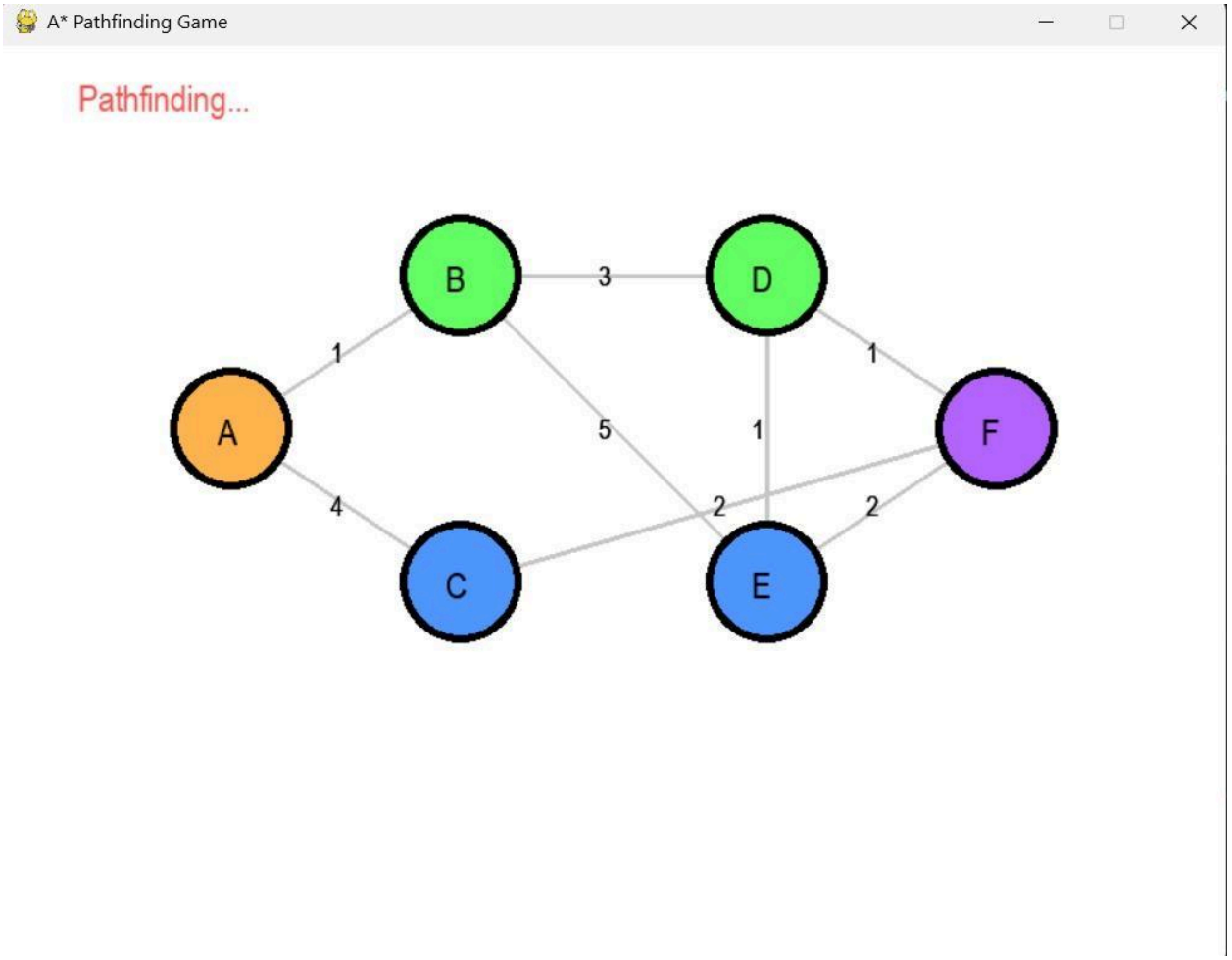
```
            if tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score[neighbor] = tentative_g + heuristic(neighbor, goal)
                heapq.heappush(open_heap, (f_score[neighbor], neighbor))

    path = []
    node = goal
    if node not in came_from and node != start:
        return visited_order, [], float('inf')
    while node in came_from:
        path.insert(0, node)
        node = came_from[node]
    path.insert(0, start)
    return visited_order, path, g_score[goal]

# --- Draw Graph ---
def draw_graph(visited=set(), path=set()):
    screen.fill(WHITE)

    # Draw edges
    for node, neighbors in graph.items():
        for neighbor, weight in neighbors.items():
            start_pos = positions[node]
            end_pos = positions[neighbor]
            pygame.draw.line(screen, GRAY, start_pos, end_pos, 3)
            mid_x = (start_pos[0] + end_pos[0]) // 2
            mid_y = (start_pos[1] + end_pos[1]) // 2
            text = SMALL_FONT.render(str(weight), True, BLACK)
            screen.blit(text, (mid_x - 10, mid_y - 10))
```

```
    # Draw nodes
    for node, (x, y) in positions.items():
        color = BLUE
        if node in visited:
            color = YELLOW
        if node in path:
            color = GREEN
        if node == start_node:
            color = ORANGE
        if node == goal_node:
            color = PURPLE
        pygame.draw.circle(screen, BLACK, (x, y), 40)
        pygame.draw.circle(screen, color, (x, y), 35)
        text = FONT.render(node, True, BLACK)
        screen.blit(text, (x - 10, y - 12))

    # Instructions
    instruction = "Click Start Node" if not start_node else \
                  "Click Goal Node" if not goal_node else "Pathfinding..."
    text_inst = FONT.render(instruction, True, RED)
    screen.blit(text_inst, (50, 20))

    pygame.display.flip()

# --- Detect clicked node ---
def get_clicked_node(pos):
    for node, (x, y) in positions.items():
        if math.dist(pos, (x, y)) < 40:
            return node
```

```
    return None

# --- Main Loop ---
running = True
visited_order, final_path, cost = [], [], 0
step = 0  # animation step

while running:
    draw_graph(set(visited_order[:step]), set(final_path[:step]))
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

        if event.type == pygame.MOUSEBUTTONDOWN:
            clicked = get_clicked_node(event.pos)
            if clicked:
                if not start_node:
                    start_node = clicked
                elif not goal_node:
                    goal_node = clicked
                    visited_order, final_path, cost = astar(start_node, goal_node)
                    print("Node Expansion Order:", " → ".join(visited_order))
                    print("Final Path:", " → ".join(final_path))
                    print(f"Total Cost: {cost}")
                    step = 0  # reset animation

    if start_node and goal_node and step < len(visited_order):
        step += 1
        pygame.time.delay(500)
```

pygame.quit()

sys.exit()



19. Given a list of cities and the distances between each pair of cities, the objective is to find the shortest

possible route that visits each city exactly once and returns to the original city (the salesman's starting

point). Implement the scenario using python.

from itertools import permutations

n = int(input("Enter the number of cities: "))

```
cities = []
for i in range(n):
    city = input(f"Enter city name {i + 1}: ")
    cities.append(city)
graph = {}
for city in cities:
    graph[city] = {}
print("\nEnter the distances between each pair of cities:")
for i in range(n):
    for j in range(i + 1, n):
        dist = int(input(f"Distance between {cities[i]} and {cities[j]}: "))
        graph[cities[i]][cities[j]] = dist
        graph[cities[j]][cities[i]] = dist
start = input("\nEnter the starting city: ")
shortest_distance = float('inf')
best_route = []
for perm in permutations([city for city in cities if city != start]):
    route = [start] + list(perm) + [start]
    distance = 0

    for i in range(len(route) - 1):
        distance += graph[route[i]][route[i + 1]]

    if distance < shortest_distance:
        shortest_distance = distance
        best_route = route
print("\nShortest Route:", " -> ".join(best_route))
print("Total Distance:", shortest_distance)
```

```
Enter the number of cities: 4
Enter city name 1: 1
Enter city name 2: 2
Enter city name 3: 3
Enter city name 4: 4

Enter the distances between each pair of cities:
Distance between 1 and 2: 5
Distance between 1 and 3: 8
Distance between 1 and 4: 3
Distance between 2 and 3: 6
Distance between 2 and 4: 10
Distance between 3 and 4: 9

Enter the starting city: 3

Shortest Route: 3 -> 2 -> 1 -> 4 -> 3
Total Distance: 23
```

20. Implement a Python program to solve a simple scheduling problem using the Simulated Annealing

Algorithm. The program should:

a) Define a set of tasks and available time slots (e.g., 6 tasks, 3 time slots).

b) Represent a schedule as an assignment of tasks to time slots.

c) Define a cost function that penalizes conflicts (e.g., overlapping tasks or exceeding slot capacity).

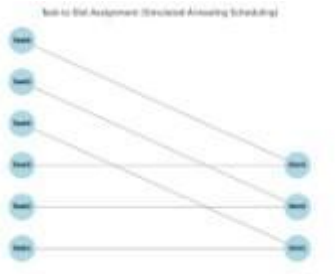
d) Start with a random schedule and iteratively propose small changes (neighbor solutions).

e) Accept better solutions always; accept worse solutions with probability exp(-ΔE / T) where T is the

temperature.

f) Implement a cooling schedule that gradually decreases temperature.

g) Print the best schedule found and its cost.



import random

import math

```
from collections import defaultdict

random.seed(42)

# ----------------------------
# Problem definition
# ----------------------------
tasks = ["A", "B", "C", "D", "E", "F"]
num_slots = 3
slot_capacity = 2  # max allowed tasks per slot (soft constraint)

# Define conflict pairs (these tasks should not be scheduled in the same slot)
# For example: A conflicts with B, C conflicts with D, E conflicts with F
conflicts = {("A","B"), ("B","A"), ("C","D"), ("D","C"), ("E","F"), ("F","E")}

# Weights in cost function
WEIGHT_CONFLICT = 10       # penalty per conflicting pair in same slot
WEIGHT_CAPACITY = 5        # penalty per task exceeding slot capacity
WEIGHT_LOAD_IMBALANCE = 1  # optional: penalty for load imbalance across slots

# ----------------------------
# Helper: cost function
# ----------------------------
def schedule_cost(schedule):
    """
    schedule: dict task -> slot_index
    Returns a scalar cost (lower is better).
    """
    # Build slot -> tasks
    slots = defaultdict(list)
```

```
for t, s in schedule.items():
    slots[s].append(t)

cost = 0

# Conflict penalties
for s_tasks in slots.values():
    n = len(s_tasks)
    if n <= 1:
        continue
    # every unique pair among s_tasks
    for i in range(n):
        for j in range(i+1, n):
            t1, t2 = s_tasks[i], s_tasks[j]
            if (t1, t2) in conflicts:
                cost += WEIGHT_CONFLICT

# Capacity penalties (soft)
for s_idx in range(num_slots):
    count = len(slots.get(s_idx, []))
    if count > slot_capacity:
        cost += (count - slot_capacity) * WEIGHT_CAPACITY

# Load imbalance penalty (encourage balanced schedules)
loads = [len(slots.get(i, [])) for i in range(num_slots)]
avg = sum(loads) / num_slots
variance = sum((l - avg)**2 for l in loads) / num_slots
cost += WEIGHT_LOAD_IMBALANCE * variance

return cost
```

```
# ----------------------------
# Generate initial random schedule
# ----------------------------
def random_schedule():
    sched = {}
    for t in tasks:
        sched[t] = random.randrange(num_slots)
    return sched


# ----------------------------
# Neighbor generation
# ----------------------------
def neighbor(schedule):
    new = schedule.copy()
    if random.random() < 0.5:
        # Move: pick one task and assign to a different slot
        t = random.choice(tasks)
        new_slot = random.randrange(num_slots)
        # ensure actually a change
        if new_slot == schedule[t]:
            new_slot = (new_slot + 1) % num_slots
        new[t] = new_slot
    else:
        # Swap: pick two tasks and swap their slots
        t1, t2 = random.sample(tasks, 2)
        new[t1], new[t2] = new[t2], new[t1]
    return new


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

```
# Simulated Annealing core
# ----------------------------
def simulated_annealing(
    initial_temp=10.0,
    final_temp=1e-3,
    alpha=0.995,
    max_iter=5000
):
    current = random_schedule()
    current_cost = schedule_cost(current)
    best = current.copy()
    best_cost = current_cost

    T = initial_temp
    iter_count = 0

    while T > final_temp and iter_count < max_iter:
        candidate = neighbor(current)
        cand_cost = schedule_cost(candidate)
        delta = cand_cost - current_cost

        accept = False
        if delta <= 0:
            accept = True
        else:
            p = math.exp(-delta / T)
            if random.random() < p:
                accept = True

        if accept:
```

```
            current, current_cost = candidate, cand_cost

            # update best
            if current_cost < best_cost:
                best, best_cost = current.copy(), current_cost

        # Cooling
        T *= alpha
        iter_count += 1

    return best, best_cost

# ----------------------------
# Pretty-print schedule
# ----------------------------
def print_schedule(schedule):
    slots = defaultdict(list)
    for t, s in schedule.items():
        slots[s].append(t)
    print("Schedule (slot -> tasks):")
    for s in range(num_slots):
        print(f" Slot {s}: {slots.get(s, [])}")

# ----------------------------
# Run
# ----------------------------
if __name__ == "__main__":
    print("Initial problem setup:")
    print(f" Tasks: {tasks}")
    print(f" Num slots: {num_slots}, slot capacity: {slot_capacity}")
```

```
print(f" Conflicts: {sorted(set(tuple(sorted(p)) for p in conflicts if p[0]<p[1]))}\n")


best_sched, best_cost = simulated_annealing(
    initial_temp=10.0,
    final_temp=1e-3,
    alpha=0.995,
    max_iter=20000
)


print("\nBest schedule found:")
print_schedule(best_sched)
print(f"Cost = {best_cost:.3f}")
```

```
Initial problem setup:
 Tasks: ['A', 'B', 'C', 'D', 'E', 'F']
 Num slots: 3, slot capacity: 2
 Conflicts: [('A', 'B'), ('C', 'D'), ('E', 'F')]


Best schedule found:
Schedule (slot -> tasks):
 Slot 0: ['C', 'F']
 Slot 1: ['A', 'E']
 Slot 2: ['B', 'D']
Cost = 0.000
```

21. Implement a Python program to solve the Knight's Tour problem using backtracking. The program

should:

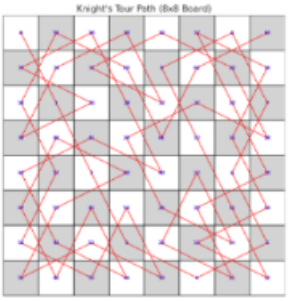
a) Represent the chessboard as an N×N grid.

b) Place the knight on a starting square.

c) Recursively attempt to move the knight to all unvisited squares.

d) Backtrack when no legal moves are available.

e) Print the final path if a full tour is found, or report failure otherwise.



CODE:

```
# Knight's Tour using Backtracking
N = 8  # size of chessboard (8x8)
def print_solution(board):
    for row in board:
        print(' '.join(str(cell).rjust(2, ' ') for cell in row))
    print()
# Check if (x, y) is a valid move
def is_safe(x, y, board):
    return 0 <= x < N and 0 <= y < N and board[x][y] == -1
# Recursive utility to solve Knight's Tour
def solve_knight_tour(x, y, movei, board, x_move, y_move):
    # Base case: if all squares are visited
    if movei == N * N:
        return True

    # Try all possible moves from current coordinate (x, y)
    for k in range(8):
        next_x = x + x_move[k]
        next_y = y + y_move[k]
        if is_safe(next_x, next_y, board):
            board[next_x][next_y] = movei
            if solve_knight_tour(next_x, next_y, movei + 1, board, x_move, y_move):
                return True
            # Backtrack
```

```
            board[next_x][next_y] = -1
    return False

def knight_tour():
    # Initialize board
    board = [[-1 for _ in range(N)] for _ in range(N)]
    # Moves for the knight
    x_move = [2, 1, -1, -2, -2, -1, 1, 2]
    y_move = [1, 2, 2, 1, -1, -2, -2, -1]

    # Start position
    board[0][0] = 0

    # Start solving from (0, 0)
    if not solve_knight_tour(0, 0, 1, board, x_move, y_move):
        print("No solution exists.")
    else:
        print("Knight's Tour Solution Path:")
        print_solution(board)

# Run the program
knight_tour()
```

OUTPUT:

```
Knight's Tour Solution Path:
 0 59 38 33 30 17  8 63
37 34 31 60  9 62 29 16
58  1 36 39 32 27 18  7
35 48 41 26 61 10 15 28
42 57  2 49 40 23  6 19
47 50 45 54 25 20 11 14
56 43 52  3 22 13 24  5
51 46 55 44 53  4 21 12
```

22. Implement a Python program to perform Uniform-Cost Search (UCS) on a weighted graph and validate

its optimality. The program should:

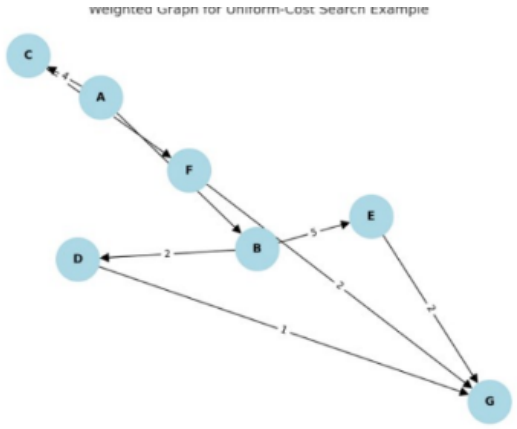
a) Represent the graph as an adjacency list with edge weights.

b) Use a priority queue (min-heap) to always expand the lowest-cost node.

c) Continue expanding nodes until the goal is reached.

d) Print the optimal path and its total cost.

e) Compare the UCS result to a known shortest-path algorithm (e.g., Dijkstra's) to validate optimality.



CODE

```
import heapq
def uniform_cost_search(graph, start, goal):
    queue, visited = [(0, start, [])], set()
    while queue:
        cost, node, path = heapq.heappop(queue)
```

```
        if node == goal:
            return path + [node], cost
        if node not in visited:
            visited.add(node)
            for neighbor, w in graph.get(node, []):
                heapq.heappush(queue, (cost + w, neighbor, path + [node]))
    return None, float('inf')
def dijkstra(graph, start, goal):
    queue, visited = [(0, start, [])], set()
    while queue:
        cost, node, path = heapq.heappop(queue)
        if node == goal:
            return path + [node], cost
        if node not in visited:
            visited.add(node)
            for neighbor, w in graph.get(node, []):
                heapq.heappush(queue, (cost + w, neighbor, path + [node]))
    return None, float('inf')
graph = {
    'C': [('A', 1)],
    'A': [('F', 4), ('B', 3)],
    'F': [('B', 2), ('G', 1)],
    'B': [('D', 2), ('E', 5), ('G', 7)],
    'D': [('G', 1)],
    'E': [('G', 2)],
    'G': []
}

ucs_path, ucs_cost = uniform_cost_search(graph, 'C', 'G')
dij_path, dij_cost = dijkstra(graph, 'C', 'G')
```

print(f"UCS: Path = {ucs_path}, Cost = {ucs_cost}")

print(f"Dijkstra: Path = {dij_path}, Cost = {dij_cost}")

OUTPUT:

```
UCS: Path = ['C', 'A', 'F', 'G'], Cost = 6
Dijkstra: Path = ['C', 'A', 'F', 'G'], Cost = 6
```

23. Implement a Python program to simulate bilateral negotiation with alternating offers and

discounting. The program should:

a) Represent two agents negotiating over 100 units of a resource.

b) Each agent has a discount factor ($0 < \delta \leq 1$) to model time preference.

c) Agents take turns proposing offers:

▪ An offer specifies how much of the 100 units each agent receives.

▪ The opponent can accept or reject the offer.

▪ If rejected, negotiation proceeds to the next round with increased discounting.

d) Stop if an agreement is reached or after a maximum number of rounds.

e) Print the final agreement, the utilities of both agents, and indicate if the result resembles a Nash-like fair outcome

CODE:

"""

Negotiation simulation (alternating offers with discounting) in Pygame

Save as negotiation_pygame.py and run:

python negotiation_pygame.py

Controls (mouse):

- Click the + / - buttons to change delta1/delta2 and max rounds

- Click Start to run the negotiation animation
- Click Reset to reset parameters

The simulation computes the subgame-perfect offers via backward induction (finite horizon) and animates each round showing the proposer, the offer, acceptance, and final outcome.

Requires: pygame (install via pip install pygame)
"""

```
import pygame
import sys
import math
from dataclasses import dataclass

# ---- Model parameters ----
R = 100.0

# ---- Backward induction and simulation (same logic as prior text-based version) ----

def compute_backwards(max_rounds: int, delta1: float, delta2: float):
    S1 = [(0.0, 0.0) for _ in range(max_rounds + 2)]
    S2 = [(0.0, 0.0) for _ in range(max_rounds + 2)]
    for t in range(max_rounds, 0, -1):
        cont_v2 = S2[t + 1][1]
        denom2 = (delta2 ** (t - 1)) if delta2 > 0 else 0.0
        if denom2 > 0:
            a2_min = cont_v2 / denom2
        else:
            a2_min = float('inf')
        if a2_min <= R:
```

```
            v1 = (delta1 ** (t - 1)) * (R - a2_min)
            v2 = cont_v2
            S1[t] = (v1, v2)
        else:
            S1[t] = S2[t + 1]

        cont_v1 = S1[t + 1][0]
        denom1 = (delta1 ** (t - 1)) if delta1 > 0 else 0.0
        if denom1 > 0:
            a1_min = cont_v1 / denom1
        else:
            a1_min = float('inf')
        if a1_min <= R:
            v2 = (delta2 ** (t - 1)) * (R - a1_min)
            v1 = cont_v1
            S2[t] = (v1, v2)
        else:
            S2[t] = S1[t + 1]
    return S1, S2

@dataclass
class RoundState:
    round_no: int
    proposer: str
    offer: tuple | None
    accepted: bool

def simulate_steps(max_rounds, delta1, delta2, starter='player1'):
    S1, S2 = compute_backwards(max_rounds, delta1, delta2)
    history = []
```

```
current = starter
for t in range(1, max_rounds + 1):
    if current == 'player1':
        cont_v2 = S2[t + 1][1]
        denom2 = (delta2 ** (t - 1)) if delta2 > 0 else 0.0
        a2_min = cont_v2 / denom2 if denom2 > 0 else float('inf')
        if a2_min <= R:
            a2 = max(0.0, min(R, a2_min))
            a1 = R - a2
            offer = (a1, a2)
            history.append(RoundState(t, 'player1', offer, True))
            return history
        else:
            history.append(RoundState(t, 'player1', None, False))
            current = 'player2'
    else:
        cont_v1 = S1[t + 1][0]
        denom1 = (delta1 ** (t - 1)) if delta1 > 0 else 0.0
        a1_min = cont_v1 / denom1 if denom1 > 0 else float('inf')
        if a1_min <= R:
            a1 = max(0.0, min(R, a1_min))
            a2 = R - a1
            offer = (a1, a2)
            history.append(RoundState(t, 'player2', offer, True))
            return history
        else:
            history.append(RoundState(t, 'player2', None, False))
            current = 'player1'
return history
```

```
# ---- Pygame UI ----

pygame.init()
WIDTH, HEIGHT = 900, 600
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption('Alternating Offers Negotiation (Discounting)')
FONT = pygame.font.SysFont('Arial', 18)
BIGFONT = pygame.font.SysFont('Arial', 24)
CLOCK = pygame.time.Clock()

# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GRAY = (200, 200, 200)
LIGHTGRAY = (230, 230, 230)
GREEN = (44, 160, 44)
RED = (214, 39, 40)
BLUE = (31, 119, 180)

# UI state
delta1 = 0.95
delta2 = 0.90
max_rounds = 15
starter = 'player1'

running_sim = False
sim_history = []
current_step_index = 0
step_timer = 0.0
STEP_DELAY = 1000  # ms between steps
```

```
# Buttons and rectangles
start_btn_rect = pygame.Rect(700, 480, 160, 40)
reset_btn_rect = pygame.Rect(700, 530, 160, 40)
plus1_rect = pygame.Rect(240, 80, 30, 30)
minus1_rect = pygame.Rect(200, 80, 30, 30)
plus2_rect = pygame.Rect(240, 140, 30, 30)
minus2_rect = pygame.Rect(200, 140, 30, 30)
plusR_rect = pygame.Rect(240, 200, 30, 30)
minusR_rect = pygame.Rect(200, 200, 30, 30)
starter_toggle_rect = pygame.Rect(200, 260, 120, 30)

def draw_text(text, x, y, font=FONT, color=BLACK):
    surf = font.render(text, True, color)
    screen.blit(surf, (x, y))

def draw_ui():
    # background
    screen.fill(WHITE)
    # Title
    draw_text('Alternating-offers Negotiation (Discounted)', 20, 10, BIGFONT)

    # Parameter panel
    pygame.draw.rect(screen, LIGHTGRAY, (180, 60, 360, 260))
    draw_text('Parameters:', 190, 65, BIGFONT)
    # delta1
    draw_text('Player1 discount (delta1):', 200, 85)
    pygame.draw.rect(screen, GRAY, minus1_rect)
    draw_text('-', minus1_rect.x + 9, minus1_rect.y + 5, BIGFONT)
    pygame.draw.rect(screen, GRAY, plus1_rect)
```

```
    draw_text('+', plus1_rect.x + 8, plus1_rect.y + 5, BIGFONT)
    draw_text(f'{delta1:.4f}', 290, 85)

    # delta2
    draw_text('Player2 discount (delta2):', 200, 145)
    pygame.draw.rect(screen, GRAY, minus2_rect)
    draw_text('-', minus2_rect.x + 9, minus2_rect.y + 5, BIGFONT)
    pygame.draw.rect(screen, GRAY, plus2_rect)
    draw_text('+', plus2_rect.x + 8, plus2_rect.y + 5, BIGFONT)
    draw_text(f'{delta2:.4f}', 290, 145)

    # max rounds
    draw_text('Max rounds:', 200, 205)
    pygame.draw.rect(screen, GRAY, minusR_rect)
    draw_text('-', minusR_rect.x + 9, minusR_rect.y + 5, BIGFONT)
    pygame.draw.rect(screen, GRAY, plusR_rect)
    draw_text('+', plusR_rect.x + 8, plusR_rect.y + 5, BIGFONT)
    draw_text(f'{max_rounds}', 290, 205)

    # starter
    pygame.draw.rect(screen, GRAY, starter_toggle_rect)
    draw_text(f'Starter: {starter}', starter_toggle_rect.x + 6, starter_toggle_rect.y + 6)

    # Buttons
    pygame.draw.rect(screen, BLUE if not running_sim else GRAY, start_btn_rect)
    draw_text('Start / Run', start_btn_rect.x + 30, start_btn_rect.y + 12, FONT, WHITE if not
running_sim else BLACK)
    pygame.draw.rect(screen, RED, reset_btn_rect)
    draw_text('Reset', reset_btn_rect.x + 60, reset_btn_rect.y + 12, FONT, WHITE)

    # Display negotiation panel
```

```
pygame.draw.rect(screen, LIGHTGRAY, (20, 340, 660, 240))
draw_text('Negotiation timeline & outcome:', 30, 345, BIGFONT)

# If simulation has steps, draw them
y0 = 380
if not sim_history:
    draw_text('No simulation yet. Adjust parameters and click Start.', 30, y0)
else:
    # show up to first 8 rounds
    for i, rs in enumerate(sim_history[:8]):
        y = y0 + i * 28
        text = f"Round {rs.round_no}: Proposer={rs.proposer}"
        if rs.offer is None:
            text += ' -> Offered nothing (rejected)'
        else:
            a1, a2 = rs.offer
            text += f' -> Offer: P1={a1:.2f}, P2={a2:.2f}'
            text += ' [ACCEPT]' if rs.accepted else ' [REJECT]'
        draw_text(text, 30, y)

# If agreement reached, display final outcome summary to the right
if sim_history and any(r.accepted for r in sim_history):
    # find first accepted
    accepted_rounds = [r for r in sim_history if r.accepted]
    first = accepted_rounds[0]
    a1, a2 = first.offer
    # discounted utilities based on round
    u1 = (delta1 ** (first.round_no - 1)) * a1
    u2 = (delta2 ** (first.round_no - 1)) * a2
    draw_text('Final Agreement:', 700, 60, BIGFONT)
```

```
        draw_text(f'Round: {first.round_no}', 700, 100)
        draw_text(f'Proposer: {first.proposer}', 700, 130)
        draw_text(f'Allocation: P1={a1:.4f}, P2={a2:.4f}', 700, 160)
        draw_text(f'Discounted utilities: U1={u1:.4f}, U2={u2:.4f}', 700, 190)
        fair_tol = 5.0
        if abs(a1 - R/2) <= fair_tol and abs(a2 - R/2) <= fair_tol:
            draw_text('Outcome resembles Nash-like fair (≈50-50)', 700, 220, FONT, GREEN)
        else:
            draw_text('Outcome NOT close to 50-50 (proposer advantage)', 700, 220, FONT,
RED)

def run_simulation():
    global sim_history, running_sim, current_step_index, step_timer
    sim_history = simulate_steps(max_rounds, delta1, delta2, starter)
    running_sim = True
    current_step_index = 0
    step_timer = pygame.time.get_ticks()

def reset_sim():
    global sim_history, running_sim, current_step_index
    sim_history = []
    running_sim = False
    current_step_index = 0

# Helpers for button clicks

def inside(rect, pos):
    return rect.collidepoint(pos)

# Main loop
```

```
def main_loop():
    global delta1, delta2, max_rounds, starter, running_sim, current_step_index, step_timer
    while True:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                pygame.quit()
                sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN and event.button == 1:
                pos = event.pos
                if inside(start_btn_rect, pos):
                    run_simulation()
                elif inside(reset_btn_rect, pos):
                    reset_sim()
                elif inside(plus1_rect, pos):
                    delta1 = min(0.9999, delta1 + 0.01)
                elif inside(minus1_rect, pos):
                    delta1 = max(0.01, delta1 - 0.01)
                elif inside(plus2_rect, pos):
                    delta2 = min(0.9999, delta2 + 0.01)
                elif inside(minus2_rect, pos):
                    delta2 = max(0.01, delta2 - 0.01)
                elif inside(plusR_rect, pos):
                    max_rounds = min(200, max_rounds + 1)
                elif inside(minusR_rect, pos):
                    max_rounds = max(1, max_rounds - 1)
                elif inside(starter_toggle_rect, pos):
                    starter = 'player2' if starter == 'player1' else 'player1'

        # update running simulation animation by stepping through history
        if running_sim and sim_history:
```

```
            now = pygame.time.get_ticks()
            if now - step_timer >= STEP_DELAY:
                # advance to next displayed step if any; here we simply ensure the history is visible
-
                # since simulate_steps returns only until agreement (or empty), we keep
running_sim True
                # for a short while then stop
                current_step_index += 1
                step_timer = now
                # stop if we've shown all steps
                if current_step_index >= len(sim_history):
                    running_sim = False


        draw_ui()
        pygame.display.flip()
        CLOCK.tick(30)


if __name__ == '__main__':
    main_loop()
```

OUTPUT:

24. Implement a Python program to demonstrate the Expectiminimax algorithm for a stochastic game. The

program should:

a) Build a simple game tree with three types of nodes: Max, Min, and Chance.

b) Assign probabilities to random events at chance nodes.

c) Use recursive Expectiminimax search to compute the expected utility of each node.

d) Print the optimal decision at the root node for the AI player.

e) Show how randomness affects the final choice

CODE:

```
def expectiminimax(node, node_type, tree, probabilities=None):
    # Leaf node
    if isinstance(tree[node], int):
        return tree[node]
    # MAX node
    if node_type == 'max':
        return max(expectiminimax(child, tree['types'][child], tree, probabilities) for child in tree[node])
    # MIN node
    if node_type == 'min':
        return min(expectiminimax(child, tree['types'][child], tree, probabilities) for child in tree[node])
    # CHANCE node
    if node_type == 'chance':
        prob_list = probabilities[node]
```

```
        return sum(
            prob * expectiminimax(child, tree['types'][child], tree, probabilities)
            for child, prob in zip(tree[node], prob_list)
        )

# Example tree structure from the image:
# Root (MAX) --> Chance Node --> Leaf1, Leaf2
#                Hit Node    --> Leaf3, Leaf4

tree = {
    'Root': ['Chance', 'Hit'],
    'types': {'Root': 'max', 'Chance': 'chance', 'Hit': 'min', 'Leaf1': 'leaf', 'Leaf2': 'leaf', 'Leaf3': 'leaf', 'Leaf4': 'leaf'},
    'Chance': ['Leaf1', 'Leaf2'],
    'Hit': ['Leaf3', 'Leaf4'],
    'Leaf1': 5,
    'Leaf2': 12,
    'Leaf3': 15,
    'Leaf4': 8
}

# Probabilities for the chance node from the edge labels in the image
probabilities = {
    'Chance': [0.5, 0.5]  # Probability for Leaf1 and Leaf2
}

# Compute expected utility for both children of the root node
chance_utility = expectiminimax('Chance', tree['types']['Chance'], tree, probabilities)
hit_utility = expectiminimax('Hit', tree['types']['Hit'], tree, probabilities)

print(f"Expected utility of 'Chance' node: {chance_utility}")
```

```
print(f"Utility of 'Hit' node (MIN): {hit_utility}")


# Optimal decision at the root (MAX node)
if chance_utility > hit_utility:
    print("AI should choose the Chance node.")
else:
    print("AI should choose the Hit node.")


print("Randomness affects the choice because the Chance node utility is an average, not a fixed value.")
```

OUTPUT:

```
----- RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python313/8.py -----
Expected utility of 'Chance' node: 8.5
Utility of 'Hit' node (MIN): 8
AI should choose the Chance node.
Randomness affects the choice because the Chance node utility is an average, not
 a fixed value.
```

25. Implement a Python program to solve a pathfinding problem using Beam Search. The program should:

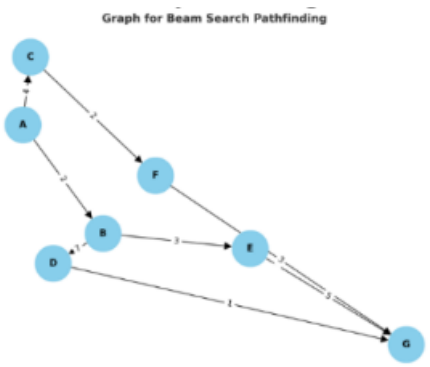
a) Represent the search space as a graph (nodes and weighted edges).

b) Use a heuristic function (e.g., straight-line distance to the goal) to guide the search.

c) Keep only the k most promising nodes (beam width) at each level.

d) Print the path found from the start node to the goal node.

e) Compare results with BFS to show efficiency vs completeness trade-off.



CODE:

```
from queue import Queue
import heapq

# ----------- Graph Representation -----------
graph = {
    'A': {'B': 2, 'C': 4},
    'B': {'D': 7, 'E': 3},
    'C': {'F': 4, 'G': 5},
    'D': {'H': 1},
    'E': {'H': 6},
    'F': {'G': 2, 'H': 5},
    'G': {'H': 3},
    'H': {}
}

# ----------- Heuristic Function -----------
# (Straight-line distance or estimated cost to goal)
heuristic = {
    'A': 10,
    'B': 8,
    'C': 7,
    'D': 4,
    'E': 5,
    'F': 3,
    'G': 2,
    'H': 0
}

# ----------- Beam Search Implementation -----------
def beam_search(start, goal, beam_width):
```

```
    print(f"\n--- Beam Search (Beam width = {beam_width}) ---")
    beam = [(heuristic[start], [start])]
    level = 0

    while beam:
        print(f"\nLevel {level}: Candidates -> {[path[-1] for (_, path) in beam]}")
        new_beam = []
        for _, path in beam:
            node = path[-1]
            if node == goal:
                print("Goal reached!")
                return path
            for neighbor, cost in graph[node].items():
                new_path = path + [neighbor]
                score = heuristic[neighbor]
                heapq.heappush(new_beam, (score, new_path))

        # Keep only k best nodes
        beam = heapq.nsmallest(beam_width, new_beam)
        level += 1

    print("Goal not found (beam too narrow).")
    return None

# ----------- Breadth-First Search (for comparison) -----------
def bfs(start, goal):
    print("\n--- Breadth-First Search ---")
    visited = set()
    queue = Queue()
    queue.put([start])
```

```
    while not queue.empty():
        path = queue.get()
        node = path[-1]

        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph[node]:
                new_path = path + [neighbor]
                queue.put(new_path)

    return None

# ----------- Main Function -----------
if __name__ == "__main__":
    start = 'A'
    goal = 'H'
    beam_width = 2  # you can change this to 1, 2, 3, etc.

    beam_result = beam_search(start, goal, beam_width)
    bfs_result = bfs(start, goal)

    print("\nBeam Search Path:", beam_result)
    print("BFS Path:", bfs_result)

    # ----------- Efficiency vs Completeness Discussion -----------
    print("\n--- Comparison ---")
```

```
print("Beam Search:")
print("- Faster, memory-efficient, but may miss optimal path if beam width is too small.")
print("BFS:")
print("- Guarantees the shortest path but explores all nodes (less efficient).")
```

OUTPUT:

```
--- Beam Search (Beam width = 2) ---

Level 0: Candidates -> ['A']

Level 1: Candidates -> ['C', 'B']

Level 2: Candidates -> ['G', 'F']

Level 3: Candidates -> ['H', 'H']
Goal reached!

--- Breadth-First Search ---

Beam Search Path: ['A', 'C', 'F', 'H']
BFS Path: ['A', 'B', 'D', 'H']

--- Comparison ---
Beam Search:
- Faster, memory-efficient, but may miss optimal path if beam width is too small
.
BFS:
- Guarantees the shortest path but explores all nodes (less efficient).
```

26. Implement a Python program to solve Sudoku as a Constraint Satisfaction Problem using

backtracking search with constraint checking. The program should:

a) Accept a Sudoku puzzle as input (use a 9×9 matrix with 0s for empty cells).

b) Use backtracking to assign values to empty cells.

c) Apply constraints to check row, column, and 3×3 subgrid validity

d) Print the solved Sudoku grid.

CODE:

```
# Sudoku Solver using Backtracking and Constraint Checking
N = 9  # Sudoku grid size (9x9)


# Function to print the Sudoku grid
```

```
def print_grid(grid):
    for i in range(N):
        for j in range(N):
            print(grid[i][j], end=" ")
        print()

# Check if placing num at grid[row][col] is valid
def is_safe(grid, row, col, num):
    # Row constraint
    if num in grid[row]:
        return False

    # Column constraint
    for i in range(N):
        if grid[i][col] == num:
            return False

    # 3x3 Subgrid constraint
    start_row = row - row % 3
    start_col = col - col % 3
    for i in range(3):
        for j in range(3):
            if grid[start_row + i][start_col + j] == num:
                return False

    return True

# Find the next empty cell (returns tuple of row, col)
def find_empty_location(grid):
    for i in range(N):
```

```
        for j in range(N):
            if grid[i][j] == 0:
                return (i, j)
    return None

# Backtracking function to solve Sudoku
def solve_sudoku(grid):
    empty = find_empty_location(grid)
    if not empty:
        return True  # Puzzle solved

    row, col = empty

    for num in range(1, 10):  # Try numbers 1–9
        if is_safe(grid, row, col, num):
            grid[row][col] = num  # Assign value

            if solve_sudoku(grid):  # Recursive step
                return True

            # Backtrack
            grid[row][col] = 0

    return False

# -------- Main Program --------
if __name__ == "__main__":
    # Example Sudoku puzzle (0 = empty)
    sudoku_grid = [
        [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]
]

print("Original Sudoku Puzzle:")
print_grid(sudoku_grid)
print("\nSolving...\n")

if solve_sudoku(sudoku_grid):
    print("Solved Sudoku Grid:")
    print_grid(sudoku_grid)
else:
    print("No solution exists.")
```

OUTPUT:

```
Original Sudoku Puzzle:
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

Solving...

Solved Sudoku Grid:
5 3 4 6 7 8 9 1 2
6 7 2 1 9 5 3 4 8
1 9 8 3 4 2 5 6 7
8 5 9 7 6 1 4 2 3
4 2 6 8 5 3 7 9 1
7 1 3 9 2 4 8 5 6
9 6 1 5 3 7 2 8 4
2 8 7 4 1 9 6 3 5
3 4 5 2 8 6 1 7 9
```

27. Implement a Python program to solve a Grid-World navigation problem using Q-Learning. The

program should:

a) Define a grid environment (e.g., 5×5 grid) with:

- **A start state.**
- **A goal state with positive reward.**
- **One or more obstacle states with negative reward.**
- **Step penalty for each move to encourage shorter paths.**

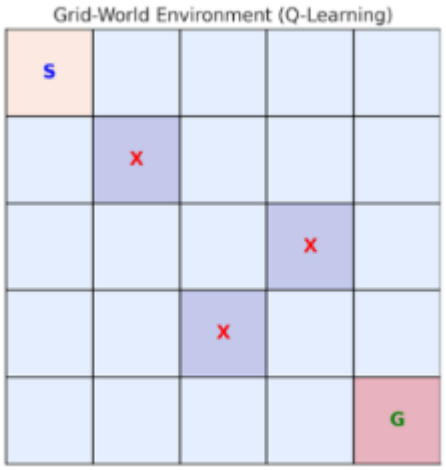
b) Implement Q-Learning with:

- **Learning rate (α).**
- **Discount factor (γ).**
- **Exploration rate (ε-greedy strategy).**

c) Train the agent for a number of episodes.

d) Print the learned Q-table.

e) Show the optimal path found from start to goal.



CODE:

```
import numpy as np
import random

# Define grid world parameters
rows, cols = 5, 5
obstacles = [(1, 1), (2, 3), (3, 2)]
start = (0, 0)
goal = (4, 4)
actions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up
action_names = ['R', 'D', 'L', 'U']
step_penalty = -1
obstacle_penalty = -10
goal_reward = 20

# Hyperparams
alpha = 0.1          # Learning rate
gamma = 0.9          # Discount factor
epsilon = 0.2        # Exploration rate
episodes = 500       # Training episodes
```

```
# Initialize Q-table
Q = np.zeros((rows, cols, len(actions)))

def valid_state(state):
    r, c = state
    if 0 <= r < rows and 0 <= c < cols and state not in obstacles:
        return True
    return False

def get_next_state(current, action):
    next_r, next_c = current[0] + action[0], current[1] + action[1]
    next_state = (next_r, next_c)
    if not valid_state(next_state):
        return current  # Invalid move, stay in place
    return next_state

# Training loop
for episode in range(episodes):
    state = start
    while state != goal:
        r, c = state
        # Explore vs exploit
        if random.uniform(0, 1) < epsilon:
            a = random.randint(0, 3)
        else:
            a = np.argmax(Q[r, c])

        action = actions[a]
        next_state = get_next_state(state, action)
        nr, nc = next_state
```

```
        # Rewards
        if next_state == goal:
            reward = goal_reward
        elif next_state in obstacles:
            reward = obstacle_penalty
        else:
            reward = step_penalty

        # Q-Learning update
        Q[r, c, a] += alpha * (reward + gamma * np.max(Q[nr, nc]) - Q[r, c, a])

        state = next_state

# Print Q-table
print("Learned Q-table (shape:", Q.shape, "):")
print(Q)

# Find optimal path from start to goal
state = start
path = [state]
while state != goal:
    r, c = state
    a = np.argmax(Q[r, c])
    next_state = get_next_state(state, actions[a])
    if next_state == state or next_state in path:
        # Trapped or loop detected
        break
    path.append(next_state)
    state = next_state
```

print("Optimal path from start to goal:")

print(path)

OUTPUT:

```
Learned Q-table (shape: (5, 5, 4) ):
[[[ 4.348907   -2.75953111  2.09835875  1.96910907]
  [ 5.94323     4.01827089  2.20882642  3.90880124]
  [ 7.7147      5.96826693  3.18282471  5.05469777]
  [ 8.95120972  9.683       4.25919091  6.49937442]
  [ 1.98665279 11.85935093  3.11458553  0.19627475]]

 [[-2.52175466 -1.21051596 -2.49943089 -2.59019997]
  [ 0.          0.          0.          0.        ]
  [ 9.61796703 -1.43205649 -0.04548171  0.13878856]
  [11.87        9.28487952  6.74810871  6.87647269]
  [11.21890612 14.3         9.13781531  9.30391368]]

 [[-1.79566399  1.26495416 -1.50235243 -2.15069292]
  [-1.42121964  0.35185929 -1.42844403 -1.38699896]
  [-1.41889045 -1.39151876 -1.39136115  0.28914141]
  [ 0.          0.          0.          0.        ]
  [13.17226295 17.         13.83285401 11.23697686]]

 [[ 4.67081252 -1.24608255 -1.22478977 -1.37981625]
  [-0.44763157  8.77134479 -0.8115668  -1.35983147]
  [ 0.          0.          0.          0.        ]
  [ 1.49370053 15.8389198  -0.1         0.61304829]
  [15.27399923 20.         10.42051921 13.3075339 ]]

 [[-0.52800853 -0.85019376 -0.86349718 -0.95940088]
  [12.58228535 -0.14622934 -0.50445431  1.6573281 ]
  [16.56665323 -0.1         0.79998162  3.06592463]
  [19.98984942  7.54288379  2.92785137  0.33817855]
  [ 0.          0.          0.          0.        ]]]
Optimal path from start to goal:
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 3), (1, 4), (2, 4), (3, 4), (4, 4)]
```

28. Implement a Python program to solve the Travelling Salesman Problem using Ant Colony

Optimization (ACO). The program should:

a) Define a set of cities with pairwise distances (matrix).

b) Simulate ants building tours probabilistically based on pheromone trails and heuristic visibility

(1/distance).

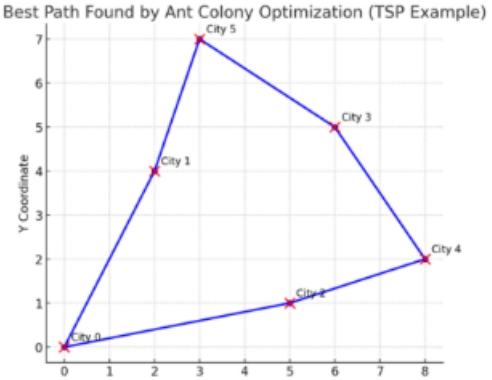
c) Update pheromones each iteration:

▪ Evaporate pheromones globally.

▪ Reinforce pheromones on paths used by ants (especially best tours).

d) Repeat for multiple iterations to converge toward the optimal/near-optimal route.

e) Print the best path and its total cost.



CODE:

```
import random


# ---------------- Parameters ----------------
NUM_CITIES = 5
NUM_ANTS = 10
ALPHA = 1.0        # pheromone influence
BETA = 5.0         # distance influence
EVAPORATION = 0.5  # pheromone evaporation rate
Q = 100            # pheromone deposit factor
ITERATIONS = 50


# ---------------- Distance Matrix ----------------
distance_matrix = [
    [0, 2, 9, 10, 7],
    [1, 0, 6, 4, 3],
    [15, 7, 0, 8, 3],
    [6, 3, 12, 0, 11],
    [9, 7, 5, 6, 0]
]


# ---------------- Initialization ----------------
num_cities = len(distance_matrix)
```

```
pheromone = [[1 for _ in range(num_cities)] for _ in range(num_cities)]

# Function to compute total tour length
def tour_length(tour):
    length = 0
    for i in range(len(tour) - 1):
        length += distance_matrix[tour[i]][tour[i + 1]]
    length += distance_matrix[tour[-1]][tour[0]]  # return to start
    return length

# Choose next city based on probability
def select_next_city(current_city, unvisited):
    total = 0.0
    probabilities = []
    for city in unvisited:
        pher = pheromone[current_city][city] ** ALPHA
        visibility = (1.0 / (distance_matrix[current_city][city] + 1e-10)) ** BETA
        score = pher * visibility
        probabilities.append((city, score))
        total += score

    # Normalize probabilities
    if total == 0:
        return random.choice(list(unvisited))
    probabilities = [(city, score / total) for city, score in probabilities]

    # Roulette wheel selection
    r = random.random()
    cumulative = 0.0
    for city, prob in probabilities:
```

```
        cumulative += prob
        if r <= cumulative:
            return city
    return probabilities[-1][0]

# ---------------- Ant Colony Optimization ----------------
def ant_colony_optimization():
    global pheromone
    best_tour = None
    best_length = float('inf')

    for iteration in range(ITERATIONS):
        all_tours = []

        for _ in range(NUM_ANTS):
            start = random.randint(0, num_cities - 1)
            tour = [start]
            unvisited = set(range(num_cities)) - {start}

            while unvisited:
                current_city = tour[-1]
                next_city = select_next_city(current_city, unvisited)
                tour.append(next_city)
                unvisited.remove(next_city)

            all_tours.append(tour)

        # Find best tour of this iteration
        for tour in all_tours:
            length = tour_length(tour)
```

```
            if length < best_length:
                best_length = length
                best_tour = tour

        # Evaporate pheromone
        for i in range(num_cities):
            for j in range(num_cities):
                pheromone[i][j] *= (1 - EVAPORATION)

        # Deposit pheromone
        for tour in all_tours:
            length = tour_length(tour)
            deposit = Q / length
            for i in range(len(tour) - 1):
                a, b = tour[i], tour[i + 1]
                pheromone[a][b] += deposit
                pheromone[b][a] += deposit
            a, b = tour[-1], tour[0]
            pheromone[a][b] += deposit
            pheromone[b][a] += deposit

        print(f"Iteration {iteration+1:02d} | Best Length So Far = {best_length}")

    return best_tour, best_length

# ---------- Main ----------
if __name__ == "__main__":
    best_path, best_cost = ant_colony_optimization()
    print("\nOptimal / Best Path Found:")
    print(" -> ".join(str(city) for city in best_path) + f" -> {best_path[0]}")
```

```
print(f"Total Cost: {best_cost}")
```

OUTPUT:

```
Iteration 09 | Best Length So Far = 22
Iteration 10 | Best Length So Far = 22
Iteration 11 | Best Length So Far = 22
Iteration 12 | Best Length So Far = 22
Iteration 13 | Best Length So Far = 22
Iteration 14 | Best Length So Far = 22
Iteration 15 | Best Length So Far = 22
Iteration 16 | Best Length So Far = 22
Iteration 17 | Best Length So Far = 22
Iteration 18 | Best Length So Far = 22
Iteration 19 | Best Length So Far = 22
Iteration 20 | Best Length So Far = 22
Iteration 21 | Best Length So Far = 22
Iteration 22 | Best Length So Far = 22
Iteration 23 | Best Length So Far = 22
Iteration 24 | Best Length So Far = 22
Iteration 25 | Best Length So Far = 22
Iteration 26 | Best Length So Far = 22
Iteration 27 | Best Length So Far = 22
Iteration 28 | Best Length So Far = 22
Iteration 29 | Best Length So Far = 22
Iteration 30 | Best Length So Far = 22
Iteration 31 | Best Length So Far = 22
Iteration 32 | Best Length So Far = 22
Iteration 33 | Best Length So Far = 22
Iteration 34 | Best Length So Far = 22
Iteration 35 | Best Length So Far = 22
Iteration 36 | Best Length So Far = 22
Iteration 37 | Best Length So Far = 22
Iteration 38 | Best Length So Far = 22
Iteration 39 | Best Length So Far = 22
Iteration 40 | Best Length So Far = 22
Iteration 41 | Best Length So Far = 22
Iteration 42 | Best Length So Far = 22
Iteration 43 | Best Length So Far = 22
Iteration 44 | Best Length So Far = 22
Iteration 45 | Best Length So Far = 22
Iteration 46 | Best Length So Far = 22
Iteration 47 | Best Length So Far = 22
Iteration 48 | Best Length So Far = 22
Iteration 49 | Best Length So Far = 22
Iteration 50 | Best Length So Far = 22

Optimal / Best Path Found:
3 -> 1 -> 0 -> 2 -> 4 -> 3
Total Cost: 22
```

29. Write a Python program to implement a simple Trust and Reputation System for service providers. The

program should:

a) Represent multiple service providers offering services with varying quality.

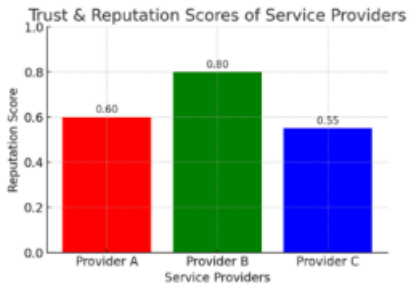
b) Allow multiple clients/agents to interact with providers and give feedback (positive or negative).

c) Maintain a trust score for each provider based on feedback.

d) Compute reputation as an aggregate of all client trust scores.

e) Allow clients to select providers based on the highest reputation score.

f) Print provider trust scores, reputation values, and the selected best provider.



CODE:

```
"""
Simple Trust & Reputation System

- Providers have hidden true_quality in [0,1].
- Clients interact with providers and provide binary feedback (1=positive, 0=negative).
- Each client keeps per-provider feedback counts.
- Per-client trust score for provider p: (pos + 1) / (pos + neg + 2)  (Beta(1,1) prior -> Laplace smoothing)
- Provider reputation: mean of all clients' trust scores for that provider.
- Clients choose provider with highest reputation when selecting.
"""

import random
from collections import defaultdict
from typing import Dict, List, Tuple

random.seed(123)  # reproducible demo

class Provider:
```

```
    def __init__(self, pid: int, true_quality: float):
        self.id = pid
        self.true_quality = float(true_quality)  # probability of delivering positive experience
        self.global_pos = 0
        self.global_neg = 0

    def deliver_service(self) -> int:
        """Return 1 for positive experience, 0 for negative (stochastic using true_quality)."""
        return 1 if random.random() < self.true_quality else 0

class Client:
    def __init__(self, cid: int):
        self.id = cid
        self.feedback_counts: Dict[int, List[int]] = defaultdict(lambda: [0, 0])

    def give_feedback(self, provider: Provider, outcome: int):
        """Record client feedback for provider (outcome=1 or 0)"""
        if outcome == 1:
            self.feedback_counts[provider.id][0] += 1
            provider.global_pos += 1
        else:
            self.feedback_counts[provider.id][1] += 1
            provider.global_neg += 1

    def trust_score_for(self, provider_id: int) -> float:
        """Compute per-client trust estimate for a provider with Laplace smoothing."""
        pos, neg = self.feedback_counts.get(provider_id, [0, 0])
        return (pos + 1) / (pos + neg + 2)  # Beta(1,1) posterior mean

class TrustReputationSystem:
```

```
def __init__(self):
    self.providers: Dict[int, Provider] = {}
    self.clients: Dict[int, Client] = {}

def add_provider(self, pid: int, true_quality: float):
    self.providers[pid] = Provider(pid, true_quality)

def add_client(self, cid: int):
    self.clients[cid] = Client(cid)

def run_interaction(self, client: Client, provider: Provider):
    """One interaction: service delivered, client records feedback."""
    outcome = provider.deliver_service()
    client.give_feedback(provider, outcome)

def compute_reputation(self, provider_id: int) -> float:
    """Aggregate reputation as mean of all clients' trust scores for this provider."""
    if not self.clients:
        return 0.0
    scores = [client.trust_score_for(provider_id) for client in self.clients.values()]
    return sum(scores) / len(scores)

def select_best_provider(self) -> Tuple[int, float]:
    """Select provider with highest reputation; return (pid, reputation)."""
    best_pid, best_rep = None, -1.0
    for pid in self.providers:
        rep = self.compute_reputation(pid)
        if rep > best_rep:
            best_pid, best_rep = pid, rep
    return best_pid, best_rep
```

```
    def simulate(self, rounds: int, interactions_per_round: int = 1):
        """Each round: each client selects a provider (by current reputation) and interacts."""
        for r in range(1, rounds + 1):
            client_ids = list(self.clients.keys())
            random.shuffle(client_ids)
            for cid in client_ids:
                client = self.clients[cid]
                best_pid, _ = self.select_best_provider()
                if best_pid is None:
                    best_pid = random.choice(list(self.providers.keys()))
                provider = self.providers[best_pid]
                for _ in range(interactions_per_round):
                    self.run_interaction(client, provider)


    def print_status(self):
        print("=== Providers summary ===")
        for pid, prov in self.providers.items():
            rep = self.compute_reputation(pid)
            print(f"Provider {pid}: true_quality={prov.true_quality:.3f},
global_pos={prov.global_pos}, global_neg={prov.global_neg}, reputation={rep:.4f}")
            print("   Per-client trust scores:")
            for cid, client in self.clients.items():
                t = client.trust_score_for(pid)
                print(f"      Client {cid}: trust={t:.4f}  (pos,neg)={client.feedback_counts.get(pid,
[0,0])}")
        best_pid, best_rep = self.select_best_provider()
        if best_pid is not None:
            print(f"\nSelected best provider by reputation: Provider {best_pid} with reputation
{best_rep:.4f}")
        else:
```

```
            print("\nNo providers available.")


def demo():
    # create system
    sys = TrustReputationSystem()

    # Add providers (id, true_quality)
    sys.add_provider(1, 0.85)   # high-quality provider
    sys.add_provider(2, 0.60)   # medium-quality
    sys.add_provider(3, 0.35)   # low-quality

    # Add clients
    num_clients = 8
    for cid in range(1, num_clients + 1):
        sys.add_client(cid)

    # Initial state print (no interactions)
    print("Initial state (no interactions):")
    sys.print_status()
    print("\n--- Running simulation: clients pick best provider by current reputation and
interact ---\n")

    # Run simulation: many rounds so reputations converge
    sys.simulate(rounds=30, interactions_per_round=1)

    # Final status
    print("\nFinal status after simulation:")
    sys.print_status()

    # Show some extra summary
    print("\nDetailed provider reputations (rounded):")
```

```
    for pid in sys.providers:
        print(f" Provider {pid}: Reputation = {sys.compute_reputation(pid):.3f}")


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

```
Initial state (no interactions):
=== Providers summary ===
Provider 1: true_quality=0.850, global_pos=0, global_neg=0, reputation=0.5000
   Per-client trust scores:
      Client 1: trust=0.5000  (pos,neg)=[0, 0]
      Client 2: trust=0.5000  (pos,neg)=[0, 0]
      Client 3: trust=0.5000  (pos,neg)=[0, 0]
      Client 4: trust=0.5000  (pos,neg)=[0, 0]
      Client 5: trust=0.5000  (pos,neg)=[0, 0]
      Client 6: trust=0.5000  (pos,neg)=[0, 0]
      Client 7: trust=0.5000  (pos,neg)=[0, 0]
      Client 8: trust=0.5000  (pos,neg)=[0, 0]
Provider 2: true_quality=0.600, global_pos=0, global_neg=0, reputation=0.5000
   Per-client trust scores:
      Client 1: trust=0.5000  (pos,neg)=[0, 0]
      Client 2: trust=0.5000  (pos,neg)=[0, 0]
      Client 3: trust=0.5000  (pos,neg)=[0, 0]
      Client 4: trust=0.5000  (pos,neg)=[0, 0]
      Client 5: trust=0.5000  (pos,neg)=[0, 0]
      Client 6: trust=0.5000  (pos,neg)=[0, 0]
      Client 7: trust=0.5000  (pos,neg)=[0, 0]
      Client 8: trust=0.5000  (pos,neg)=[0, 0]
Provider 3: true_quality=0.350, global_pos=0, global_neg=0, reputation=0.5000
   Per-client trust scores:
      Client 1: trust=0.5000  (pos,neg)=[0, 0]
      Client 2: trust=0.5000  (pos,neg)=[0, 0]
      Client 3: trust=0.5000  (pos,neg)=[0, 0]
      Client 4: trust=0.5000  (pos,neg)=[0, 0]
      Client 5: trust=0.5000  (pos,neg)=[0, 0]
      Client 6: trust=0.5000  (pos,neg)=[0, 0]
      Client 7: trust=0.5000  (pos,neg)=[0, 0]
      Client 8: trust=0.5000  (pos,neg)=[0, 0]

Selected best provider by reputation: Provider 1 with reputation 0.5000

--- Running simulation: clients pick best provider by current reputation and interact ---
```

```
Final status after simulation:
=== Providers summary ===
Provider 1: true_quality=0.850, global_pos=206, global_neg=34, reputation=0.8359
   Per-client trust scores:
      Client 1: trust=0.8438  (pos,neg)=[26, 4]
      Client 2: trust=0.8438  (pos,neg)=[26, 4]
      Client 3: trust=0.8125  (pos,neg)=[25, 5]
      Client 4: trust=0.8438  (pos,neg)=[26, 4]
      Client 5: trust=0.8750  (pos,neg)=[27, 3]
      Client 6: trust=0.8438  (pos,neg)=[26, 4]
      Client 7: trust=0.8438  (pos,neg)=[26, 4]
      Client 8: trust=0.7812  (pos,neg)=[24, 6]
Provider 2: true_quality=0.600, global_pos=0, global_neg=0, reputation=0.5000
   Per-client trust scores:
      Client 1: trust=0.5000  (pos,neg)=[0, 0]
      Client 2: trust=0.5000  (pos,neg)=[0, 0]
      Client 3: trust=0.5000  (pos,neg)=[0, 0]
      Client 4: trust=0.5000  (pos,neg)=[0, 0]
      Client 5: trust=0.5000  (pos,neg)=[0, 0]
      Client 6: trust=0.5000  (pos,neg)=[0, 0]
      Client 7: trust=0.5000  (pos,neg)=[0, 0]
      Client 8: trust=0.5000  (pos,neg)=[0, 0]
Provider 3: true_quality=0.350, global_pos=0, global_neg=0, reputation=0.5000
   Per-client trust scores:
      Client 1: trust=0.5000  (pos,neg)=[0, 0]
      Client 2: trust=0.5000  (pos,neg)=[0, 0]
      Client 3: trust=0.5000  (pos,neg)=[0, 0]
      Client 4: trust=0.5000  (pos,neg)=[0, 0]
      Client 5: trust=0.5000  (pos,neg)=[0, 0]
      Client 6: trust=0.5000  (pos,neg)=[0, 0]
      Client 7: trust=0.5000  (pos,neg)=[0, 0]
      Client 8: trust=0.5000  (pos,neg)=[0, 0]

Selected best provider by reputation: Provider 1 with reputation 0.8359

Detailed provider reputations (rounded):
 Provider 1: Reputation = 0.836
 Provider 2: Reputation = 0.500
 Provider 3: Reputation = 0.500
```

30. Write a Python program to:

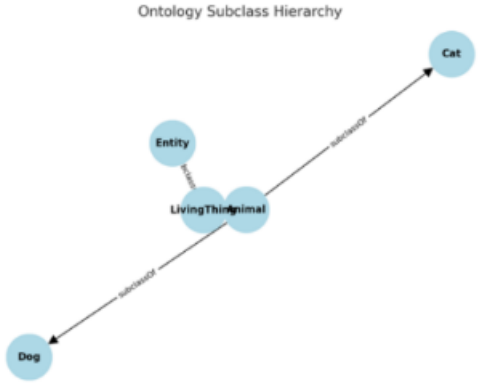
a) Parse ontology triples representing subclass relationships.

b) Build a knowledge graph (dictionary-based).

c) Support queries like:

- **Find all subclasses of a given class.**
- **Check if one class is a subclass of another (directly or indirectly).**

d) Print the results clearly.



CODE:

```
# ---------------- Ontology Graph Builder ----------------

# Example ontology triples (subject, predicate, object)
triples = [
    ("Dog", "rdfs:subClassOf", "Mammal"),
    ("Cat", "rdfs:subClassOf", "Mammal"),
    ("Mammal", "rdfs:subClassOf", "Animal"),
    ("Animal", "rdfs:subClassOf", "LivingBeing"),
    ("Fish", "rdfs:subClassOf", "Animal"),
    ("Shark", "rdfs:subClassOf", "Fish"),
]

# ---------------- Build Knowledge Graph ----------------
def build_knowledge_graph(triples):
    graph = {}
    for subj, pred, obj in triples:
        if pred == "rdfs:subClassOf":
            if obj not in graph:
                graph[obj] = set()
            graph[obj].add(subj)
    return graph
```

```
# ---------------- Query Functions ----------------

# Find all subclasses (direct + indirect)
def find_all_subclasses(graph, superclass):
    subclasses = set()
    direct = graph.get(superclass, set())

    for sub in direct:
        subclasses.add(sub)
        subclasses |= find_all_subclasses(graph, sub)  # recursive call

    return subclasses

# Check if one class is subclass of another
def is_subclass(graph, subclass, superclass):
    # Direct subclass?
    if subclass in graph.get(superclass, set()):
        return True

    # Check indirect subclasses recursively
    for sub in graph.get(superclass, set()):
        if is_subclass(graph, subclass, sub):
            return True
    return False

# ---------------- Main ----------------
if __name__ == "__main__":
    graph = build_knowledge_graph(triples)
    print("Knowledge Graph (Superclass → Subclasses):")
    for key, value in graph.items():
```

```
    print(f"  {key} → {', '.join(value)}")


print("\n--- Queries ---")


# Query 1: Find all subclasses of 'Animal'
target_class = "Animal"
result = find_all_subclasses(graph, target_class)
print(f"All subclasses of '{target_class}': {', '.join(sorted(result))}")


# Query 2: Check subclass relationship
subclass, superclass = "Dog", "LivingBeing"
print(f"\nIs '{subclass}' a subclass of '{superclass}'? → {is_subclass(graph, subclass, superclass)}")


subclass, superclass = "Shark", "Mammal"
print(f"Is '{subclass}' a subclass of '{superclass}'? → {is_subclass(graph, subclass, superclass)}")
```

OUTPUT:

```
Knowledge Graph (Superclass → Subclasses):
  Mammal → Dog, Cat
  Animal → Fish, Mammal
  LivingBeing → Animal
  Fish → Shark

--- Queries ---
All subclasses of 'Animal': Cat, Dog, Fish, Mammal, Shark

Is 'Dog' a subclass of 'LivingBeing'? → True
Is 'Shark' a subclass of 'Mammal'? → False
```