# PROYECTO MEJORADO

June 25, 2025

## 1 Cifrado en Red con Visualización Geográfica y Ataque MITM Simulado

Elaborado por: **Gabriel Rivera**

**QUE HACE EL CODIGO:**

este código simula una red de comunicación segura entre diferentes ciudades de Colombia. Utiliza una combinación de cifrado Vigenère y RSA para proteger los mensajes. Puedes configurar la red con un número específico de nodos (dispositivos), enviar mensajes entre ellos y visualizar el camino que sigue el mensaje en un mapa interactivo. También incluye una simulación de un ataque "Man-in-the-Middle" (MITM) para demostrar cómo se vería una interceptación, aunque el cifrado busca proteger contra ella.

### 1.0.1 1. Instalación de Librería

Esta celda instala la librería `pycryptodome` necesaria para el cifrado RSA.

[ ]:
```
%pip install pycryptodome
```

```
Requirement already satisfied: pycryptodome in /usr/local/lib/python3.11/dist-
packages (3.23.0)
```

### 1.0.2 2. Importaciones y Constantes

Esta celda importa las librerías necesarias y define constantes como el alfabeto para el cifrado y las coordenadas de las ciudades colombianas.

En resumen, esta celda:

Importa las librerías necesarias para crear la red, manejar el cifrado (Vigenère y RSA), simular firmas y visualizar todo en un mapa interactivo.

Define el conjunto de caracteres que se pueden cifrar (ALFABETO).

Define las coordenadas de las ciudades colombianas (CIUDADES_COORDS) para la visualización en el mapa.

Es básicamente la configuración inicial del código.

[ ]:
```
# CIFRADO VIGENÈRE EXTENDIDO CON RED SIMULADA

```

```
import networkx as nx # Importa la librería networkx para trabajar con grafos␣
 ↪(redes)
# import matplotlib.pyplot as plt # Importa matplotlib.pyplot para␣
 ↪visualización (algunas partes pueden estar comentadas o reemplazadas por␣
 ↪plotly)
# import matplotlib.patches as mpatches # Importa parches de matplotlib para␣
 ↪elementos gráficos (posiblemente para leyendas antiguas)
import string # Importa la librería string para manejar cadenas de texto y␣
 ↪alfabetos
import random # Importa la librería random para generar números aleatorios y␣
 ↪selecciones al azar
import hashlib # Importa la librería hashlib para funciones de hashing seguras␣
 ↪como SHA256
from Crypto.PublicKey import RSA # Importa la clave pública de RSA de la␣
 ↪librería PyCryptodome para cifrado asimétrico
from Crypto.Cipher import PKCS1_OAEP # Importa el cifrado PKCS1_OAEP de␣
 ↪PyCryptodome para cifrado RSA seguro
import binascii # Importa la librería binascii para conversiones entre binario␣
 ↪y ASCII binario
# import matplotlib.animation as animation # Importa la librería de animación␣
 ↪de matplotlib (posiblemente no utilizada activamente activamente ahora)
import plotly.graph_objects as go # Importa graph_objects de plotly para crear␣
 ↪gráficos interactivos
# import math # Importa la librería math para cálculos matemáticos como la␣
 ↪distancia euclidiana # Removido math

# Definimos el alfabeto que se utilizará para el cifrado Vigenère
ALFABETO = string.ascii_letters + string.digits + string.punctuation + ' '

# Coordenadas geográficas de las ciudades colombianas para la visualización
CIUDADES_COORDS = {
    "Bogotá": (4.711, -74.0721),
    "Medellín": (6.2442, -75.5812),
    "Cali": (3.4516, -76.5319),
    "Barranquilla": (10.9685, -74.7813),
    "Cartagena": (10.3910, -75.4794),
    "Bucaramanga": (7.1193, -73.1227),
    "Cúcuta": (7.8939, -72.5078),
    "Pereira": (4.8087, -75.6906),
    "Manizales": (5.0689, -75.5174),
    "Ibagué": (4.4389, -75.2322),
    "Pasto": (1.2136, -77.2811),
    "Neiva": (2.9965, -75.2908),
    "Villavicencio": (4.1438, -73.6224),
    "Santa Marta": (11.2408, -74.2048),
    "Montería": (8.7609, -75.8822),
```

```
    "Armenia": (4.5388, -75.6816),
    "Sincelejo": (9.3047, -75.3973),
    "Popayán": (2.4410, -76.6029),
    "Tunja": (5.5350, -73.3677),
    "Riohacha": (11.5444, -72.9078),
    "Valledupar": (10.4634, -73.2934),
    "Apartadó": (7.8834, -76.6117),
    "Florencia": (1.6137, -75.6060),
    "Quibdó": (5.6932, -76.6585),
    "Arauca": (7.0841, -70.7611),
    "San Andrés": (12.5833, -81.7000),
    "Leticia": (-4.2167, -69.9333)
}
```

### 1.0.3 3. Funciones de Cifrado y Firma

Esta celda contiene las funciones para el cifrado y descifrado Vigenère, así como para firmar y verificar mensajes usando hashing.

Aquí tienes un resumen más corto y sencillo de las funciones en esa celda de código:

**vigenere_cifrar:** Toma un mensaje y una clave, y "revuelve" las letras del mensaje según la clave para hacerlo ilegible.

**vigenere_descifrar:** Toma un mensaje revuelto (cifrado) y la misma clave, y lo "desrevuelve" para obtener el mensaje original.

**firmar_mensaje:** Le pone una "etiqueta" al mensaje (los primeros 8 caracteres de un código único basado en el mensaje) para poder verificar si alguien lo cambia después.

**verificar_firma:** Revisa si la "etiqueta" del mensaje coincide con el mensaje. Si coincide, el mensaje no ha sido alterado; si no, ¡alguien lo cambió!

En esencia, esta celda tiene las herramientas para codificar (Vigenère) y ponerle un sello de seguridad (la firma simulada) a tus mensajes.

[ ]:
```
def vigenere_cifrar(mensaje, clave):
    """
    Cifra un mensaje usando el cifrado Vigenère con una clave dada.

    Toma un mensaje y una clave como entrada. Itera sobre cada carácter del␣
 ↪mensaje y,
    si el carácter se encuentra en el ALFABETO definido, lo cifra utilizando la␣
 ↪clave
    y la posición del carácter en el alfabeto. Los caracteres que no están en el
    alfabeto se dejan sin cifrar.

    Args:
        mensaje (str): El mensaje a cifrar.
        clave (str): La clave para el cifrado Vigenère.
```

```
    Returns:
        str: El mensaje cifrado.
    """
    # Itera sobre cada carácter del mensaje con su índice
    return ''.join(
        # Cifra el carácter si está en el alfabeto
        ALFABETO[(ALFABETO.index(c) + ALFABETO.index(clave[i % len(clave)])) %␣
↪len(ALFABETO)]
        if c in ALFABETO else c # Deja el carácter sin cifrar si no está en el␣
↪alfabeto
        for i, c in enumerate(mensaje)
    )

def vigenere_descifrar(mensaje, clave):
    """
    Descifra un mensaje usando el cifrado Vigenère con una clave dada.

    Toma el mensaje cifrado y la misma clave utilizada para cifrarlo. Itera␣
↪sobre
    cada carácter del mensaje cifrado y, si está en el ALFABETO, lo descifra
    utilizando la clave de forma inversa al cifrado. Los caracteres no presentes
    en el alfabeto se mantienen igual.

    Args:
        mensaje (str): El mensaje a descifrar.
        clave (str): La clave utilizada para el cifrado Vigenère.

    Returns:
        str: El mensaje descifrado.
    """
    # Itera sobre cada carácter del mensaje cifrado con su índice
    return ''.join(
        # Descifra el carácter si está en el alfabeto
        ALFABETO[(ALFABETO.index(c) - ALFABETO.index(clave[i % len(clave)])) %␣
↪len(ALFABETO)]
        if c in ALFABETO else c # Deja el carácter sin descifrar si no está en␣
↪el alfabeto
        for i, c in enumerate(mensaje)
    )

def firmar_mensaje(mensaje):
    """
    Firma digitalmente un mensaje calculando su hash SHA256 y adjuntando los␣
↪primeros 8 caracteres.
```

```
    Simula una firma digital simple. Toma un mensaje, calcula su hash SHA256 y␣
↪luego
    toma los primeros 8 caracteres de ese hash. Devuelve el mensaje original
    concatenado con "||" y estos 8 caracteres del hash como la "firma". Esto␣
↪permite
    verificar si el mensaje fue alterado posteriormente (aunque no proporciona
    autenticación real como una firma digital con claves privadas/públicas).

    Args:
        mensaje (str): El mensaje a firmar.
    Returns:
        str: El mensaje firmado en formato "mensaje||firma".
    """
    # Calcula el hash SHA256 del mensaje
    hash_completo = hashlib.sha256(mensaje.encode()).hexdigest()
    # Toma los primeros 8 caracteres del hash
    hash8 = hash_completo[:8]
    # Concatena el mensaje original con "||" y los 8 caracteres del hash
    return f"{mensaje}||{hash8}"

def verificar_firma(mensaje_firmado):
    """
    Verifica la firma digital de un mensaje firmado.

    Verifica la "firma" de un mensaje que fue firmado previamente con␣
↪firmar_mensaje.
    Toma el mensaje_firmado en formato "mensaje||firma", lo divide en el mensaje
    original y la firma. Recalcula el hash SHA256 del mensaje original extraído␣
↪y
    compara los primeros 8 caracteres con la firma proporcionada.

    Args:
        mensaje_firmado (str): El mensaje firmado en formato "mensaje||firma".

    Returns:
        tuple: Una tupla que contiene un booleano (True si la firma es válida,␣
↪False si no)
               y el mensaje original extraído.
    """
    try:
        # Divide el mensaje firmado en mensaje original y firma
        mensaje, firma = mensaje_firmado.rsplit('||', 1)
        # Recalcula el hash del mensaje original extraído y compara los␣
↪primeros 8 caracteres con la firma
        return firma == hashlib.sha256(mensaje.encode()).hexdigest()[:8],␣
↪mensaje
```

```
    except ValueError:
        # Si no se puede dividir el mensaje (formato incorrecto), la firma es␣
↪inválida
        return False, mensaje_firmado
```

### 1.0.4 4. Creación de la Red y Envío de Mensajes

Esta celda contiene la función para crear la red de dispositivos y la función principal para simular el envío de un mensaje, incluyendo el cifrado RSA de la clave simétrica y la simulación del espía.

Aquí tienes un resumen más conciso de las funciones en esta celda:

**crear_red_dispositivos:** Crea una red aleatoria de dispositivos ubicados en ciudades colombianas, asignando a cada uno claves de seguridad (RSA).

**enviar_mensaje_unico**: Simula el envío de un mensaje entre dos dispositivos en la red. Cifra el mensaje (Vigenère), cifra la clave de ese cifrado (RSA) y, si se activa, muestra cómo un espía vería el mensaje interceptado antes de que llegue al destino.

Básicamente, esta celda te permite construir la red y simular el proceso de envío de mensajes seguros con la posibilidad de ver una simulación de ataque.

[ ]:
```
def crear_red_dispositivos(n):
    """
    Crea una red aleatoria de dispositivos (nodos) distribuidos en ciudades␣
↪colombianas.

    A cada nodo se le asigna un par de claves RSA (pública y privada) y una␣
↪ubicación
    geográfica. Crea un grafo aleatorio usando el modelo de Erdos-Renyi y␣
↪asegura
    que sea conexo (que todos los nodos estén conectados).

    Args:
        n (int): El número de nodos (dispositivos) a crear.

    Returns:
        nx.Graph: El grafo que representa la red de dispositivos.
    """
    # Obtiene la lista de ciudades y la mezcla aleatoriamente
    ciudades = list(CIUDADES_COORDS.keys())
    random.shuffle(ciudades)

    # Crea un grafo aleatorio usando el modelo de Erdos-Renyi
    G = nx.erdos_renyi_graph(n, 0.4)
    # Asegura que el grafo sea conexo (todos los nodos conectados)
    while not nx.is_connected(G):
        G = nx.erdos_renyi_graph(n, 0.4)
```

```
    # Asigna claves RSA y ubicación a cada nodo
    for nodo in G.nodes:
        # Genera un par de claves RSA (pública y privada)
        key = RSA.generate(2048)
        G.nodes[nodo]['rsa_private'] = key
        G.nodes[nodo]['rsa_public'] = key.publickey()
        # Asigna una ciudad al nodo
        ciudad = ciudades[nodo % len(ciudades)]
        G.nodes[nodo]['ciudad'] = ciudad
        G.nodes[nodo]['pos'] = CIUDADES_COORDS[ciudad] # Asigna coordenadas de␣
↪la ciudad
    return G

def enviar_mensaje_unico(G, origen, destino, mensaje, espia_activo=True):
    """
    Simula el envío de un mensaje cifrado y firmado entre dos nodos de la red.

    Utiliza cifrado Vigenère para el mensaje y cifrado RSA para la clave␣
↪simétrica.
    Opcionalmente, simula un ataque Man-in-the-Middle (MITM) donde un nodo espía
    intercepta el mensaje.

    Args:
        G (nx.Graph): El grafo que representa la red de dispositivos.
        origen (int): El índice del nodo de origen.
        destino (int): El índice del nodo de destino.
        mensaje (str): El mensaje a enviar.
        espia_activo (bool, optional): True para activar la simulación del␣
↪espía, False para desactivarla.
                                       Por defecto es True.

    Returns:
        tuple: Una tupla que contiene el camino recorrido por el mensaje y el␣
↪nodo espía (si está activo).
    """
    # Calcula el camino más corto entre origen y destino
    camino = nx.shortest_path(G, origen, destino)
    # Selecciona un nodo espía aleatorio en el camino (excluyendo origen y␣
↪destino) si el espía está activo y el camino tiene al menos 3 nodos
    espia = random.choice(camino[1:-1]) if espia_activo and len(camino) > 2␣
↪else None

    # Genera una clave simétrica aleatoria para el cifrado Vigenère
    clave_simetrica = ''.join(random.choice(ALFABETO) for _ in range(16))
    # Firma digitalmente el mensaje usando SHA256
    mensaje_firmado = firmar_mensaje(mensaje)
```

```
    # Cifra el mensaje firmado usando Vigenère con la clave simétrica
    mensaje_cifrado = vigenere_cifrar(mensaje_firmado, clave_simetrica)

    # Obtiene la clave pública del nodo destino
    public_key = G.nodes[destino]['rsa_public']
    # Crea un objeto de cifrado RSA con PKCS1_OAEP
    cipher_rsa = PKCS1_OAEP.new(public_key)
    # Cifra la clave simétrica usando la clave pública del destino
    clave_cifrada = cipher_rsa.encrypt(clave_simetrica.encode())

    # Imprime información sobre el envío del mensaje
    print(f"\n Enviando mensaje de nodo {origen} a nodo {destino}:")
    print(" - Camino:", [f"{n} ({G.nodes[n]['ciudad']})" for n in camino])
    print(" - Clave cifrada (hex):", binascii.hexlify(clave_cifrada).decode())
    print(" - Mensaje cifrado:", mensaje_cifrado)

    # Simula la intercepción por el espía si está activo
    if espia is not None:
        print(f"\n Nodo espía interceptó el mensaje en nodo {espia} ({G.
↪nodes[espia]['ciudad']}):")
        print(" - Contenido interceptado:", mensaje_cifrado)
        # Simula que el espía intenta descifrar la clave simétrica (para␣
↪demostración, asume que el espía tiene la clave privada del destino)
        try:
            spy_private_key = G.nodes[destino]['rsa_private'] # Espía obtiene␣
↪la clave privada del destino (simulación)
            spy_clave_descifrada = PKCS1_OAEP.new(spy_private_key).
↪decrypt(clave_cifrada).decode()
            print(" - Clave simétrica descifrada (por el espía):",␣
↪spy_clave_descifrada)
        except Exception as e:
            print(f" - El espía no pudo descifrar la clave simétrica. Error:␣
↪{e}")

    # Obtiene la clave privada del nodo destino
    private_key = G.nodes[destino]['rsa_private']
    # Descifra la clave simétrica usando la clave privada del destino
    clave_descifrada = PKCS1_OAEP.new(private_key).decrypt(clave_cifrada).
↪decode()
    # Descifra el mensaje cifrado usando Vigenère con la clave simétrica␣
↪descifrada
    mensaje_descifrado = vigenere_descifrar(mensaje_cifrado, clave_descifrada)
    # Verifica la firma digital del mensaje descifrado
    valido, mensaje_final = verificar_firma(mensaje_descifrado)

    # Imprime información sobre el mensaje recibido
```

```
    print("\n Mensaje recibido:")
    print(" - Descifrado:", mensaje_final)
    print(" - Integridad:", "  Válido" if valido else "  Alterado")

    # Devuelve el camino recorrido y el nodo espía (si existe)
    return camino, espia
```

### 1.0.5 5. Visualización Geográfica con Plotly

Esta celda contiene la función para visualizar la red y el camino del mensaje en un mapa interactivo utilizando Plotly.

Aquí tienes un resumen más corto de la función **mostrar_red_en_mapa_plotly:**

Esta función visualiza la red de dispositivos en un mapa interactivo usando Plotly. Muestra todos los nodos (dispositivos) conectados, resalta el nodo de origen, el nodo de destino y el nodo espía (si lo hay), dibuja el camino que siguió el mensaje y reproduce una pequeña animación que simula el movimiento del mensaje a lo largo de ese camino.

Básicamente, convierte los datos de la red y el envío del mensaje en un mapa visual e interactivo.

[ ]:
```
# Nueva función para visualizar la red en un mapa interactivo usando Plotly
def mostrar_red_en_mapa_plotly(G, camino, origen, destino, espia=None):
    """
    Visualiza la red de dispositivos en un mapa interactivo usando Plotly.

    Args:
        G (nx.Graph): El grafo que representa la red de dispositivos.
        camino (list): La lista de nodos que forman el camino del mensaje.
        origen (int): El nodo de origen del mensaje.
        destino (int): El nodo de destino del mensaje.
        espia (int, optional): El nodo espía, si existe.
    """
    # Obtiene las posiciones de los nodos
    pos = nx.get_node_attributes(G, 'pos')
    # Crea etiquetas para los nodos con su número y ciudad
    labels = {n: f"{n} ({G.nodes[n]['ciudad']})" for n in G.nodes}

    # Prepara los datos para las aristas del grafo (sin información de␣
 ↪distancia)
    edge_x = []
    edge_y = []
    for edge in G.edges():
        x0, y0 = pos[edge[0]]
        x1, y1 = pos[edge[1]]
        edge_x.extend([x0, x1, None]) # Use extend for cleaner code
        edge_y.extend([y0, y1, None]) # Use extend for cleaner code
```

```
    # Crea el trazo para las aristas
    edge_trace = go.Scatter(
        x=edge_x, y=edge_y,
        line=dict(width=0.5, color='#888'),
        hoverinfo='none',  # No muestra información al pasar el ratón
        mode='lines')

    # Prepara los datos para los nodos del grafo
    node_x = []
    node_y = []
    node_text = []
    node_color = []
    node_size = []
    for node in G.nodes():
        x, y = pos[node]
        node_x.append(x)
        node_y.append(y)
        node_text.append(labels[node])
        # Asigna colores y tamaños especiales a los nodos de origen, destino y␣
↪espía
        if node == origen:
            node_color.append('orange')
            node_size.append(20)
        elif node == destino:
            node_color.append('red')
            node_size.append(20)
        elif node == espia:
            node_color.append('purple')
            node_size.append(20)
        else:
            node_color.append('lightblue')
            node_size.append(10)

    # Crea el trazo para los nodos
    node_trace = go.Scatter(
        x=node_x, y=node_y, mode='markers+text',
        text=node_text,
        textposition="bottom center",
        hoverinfo='text',
        marker=dict(
            showscale=False,
            colorscale='YlGnBu',
            reversescale=True,
            color=node_color,
            size=node_size,
            colorbar=dict(
                thickness=15,
```

```
                title='Conexiones de Nodos',
                xanchor='left',
                titleside='right'
            ),
            line_width=2))

    # Crear aristas para el camino recorrido por el mensaje (sin información de distancia)
    path_edge_x = []
    path_edge_y = []
    for i in range(len(camino) - 1):
        x0, y0 = pos[camino[i]]
        x1, y1 = pos[camino[i+1]]
        path_edge_x.extend([x0, x1, None]) # Use extend for cleaner code
        path_edge_y.extend([y0, y1, None]) # Use extend for cleaner code


    # Crea el trazo para el camino
    path_trace = go.Scatter(
        x=path_edge_x, y=path_edge_y,
        line=dict(width=2.5, color='green'),
        hoverinfo='none', # No muestra información al pasar el ratón
        mode='lines',
        name='Camino')

    # Crear punto animado para el mensaje que se mueve a lo largo del camino
    message_point = go.Scatter(
        x=[pos[camino[0]][0]], y=[pos[camino[0]][1]],
        mode='markers',
        marker=dict(size=10, color='darkgreen'),
        name='Mensaje')

    # Crea la figura de Plotly con los trazos
    fig = go.Figure(data=[edge_trace, node_trace, path_trace, message_point],
                 layout=go.Layout(
                    title='Red de Dispositivos y Camino del Mensaje', # Título actualizado
                    titlefont_size=16,
                    showlegend=True,
                    hovermode='closest',
                    margin=dict(b=20,l=5,r=5,t=40),
                    annotations=[
                     ],
                    xaxis=dict(showgrid=False, zeroline=False, showticklabels=False),
                    yaxis=dict(showgrid=False, zeroline=False, showticklabels=False))
```

```
                )

    # Crear frames para la animación
    frames = []
    frames_per_edge = 10 # Esto debe coincidir con la velocidad de animación␣
↪deseada
    total_frames = len(camino) * frames_per_edge
    for i in range(total_frames):
        edge_index = min(i // frames_per_edge, len(camino) - 2) # Ensure index␣
↪is within bounds
        start_node = camino[edge_index]
        end_node = camino[edge_index + 1]
        start_pos = pos[start_node]
        end_pos = pos[end_node]
        t = (i % frames_per_edge) / frames_per_edge
        x = start_pos[0] + (end_pos[0] - start_pos[0]) * t
        y = start_pos[1] + (end_pos[1] - start_pos[1]) * t
        frames.append(go.Frame(data=[go.Scatter(x=[x], y=[y], mode='markers',␣
↪marker=dict(size=10, color='darkgreen'))]))

    # Asigna los frames a la figura
    fig.frames = frames

    # Agregar botón de reproducción y slider para controlar la animación
    fig.update_layout(
        updatemenus=[
            dict(
                type="buttons",
                buttons=[dict(label="Reproducir",
                              method="animate",
                              args=[None, {"frame": {"duration": 100, "redraw":␣
↪True},
                                           "fromcurrent": True, "transition":␣
↪{"duration": 0}}],
                              args2=[None, {"frame": {"duration": 100, "redraw":
↪ True},
                                            "mode": "immediate", "transition":␣
↪{"duration": 0}}])],
                pad={"r": 10, "t": 87},
                showactive=False,
                x=0.1,
                xanchor="right",
                y=0,
                yanchor="top"
            )
        ],
```

```
        sliders=[dict(steps=[dict(args=[[f.name]], # Corrected args for slider␣
↪steps
                                  label=k,
                                  method="animate") for k, f in enumerate(fig.
↪frames)],
                     transition=dict(duration=0),
                     x=0.1,
                     xanchor="left",
                     y=0,
                     yanchor="top")]
    )

    # Muestra la figura interactiva
    fig.show()
```

### 1.0.6 6. Interacción del Usuario y Ejecución Principal

Esta celda contiene las funciones para interactuar con el usuario (solicitar datos de envío de mensajes y configuración de red) y la lógica principal para iniciar el simulador.

Aquí tienes un resumen más corto de las funciones en esta celda:

**enviar_mensaje_interactivo:** Te permite ingresar por consola quién envía el mensaje, a quién, el contenido del mensaje y si quieres activar el espía, y luego simula el envío y muestra el resultado en el mapa.

**main:** Es la función principal que inicia el programa. Te pregunta si quieres crear la red (definiendo cuántos nodos tendrá) o salir. Una vez creada la red, te indica que uses la siguiente celda para enviar mensajes.

Básicamente, esta celda es el punto de entrada y la interfaz principal para que interactúes con el simulador.

[ ]:
```
# New function to handle sending a single message interactively
def enviar_mensaje_interactivo(G):
    """
    Configures and sends a single message, requesting origin, destination,
    message, and spy preference from the user via console input. This function
    is designed to be called from a separate cell after the network is created.
    Returns True if the user wants to send another message, False to stop.
    """
    # Show available nodes and their cities
    print(f"\nNodos disponibles: {[(n, G.nodes[n]['ciudad']) for n in G.
↪nodes]}")

    # --- Message Configuration ---
    print("\n--- Configuración del Mensaje ---")
    while True:
        try:
```

```
            # Request origin node from the user
            origen = int(input("Nodo origen: "))
            # Check if the origin node is valid
            if origen not in G.nodes:
                print("Nodo de origen no válido. Por favor, ingresa un nodo de␣
↪la lista disponible.")
                continue
            # Request destination node from the user
            destino = int(input("Nodo destino: "))
            # Check if the destination node is valid
            if destino not in G.nodes:
                print("Nodo de destino no válido. Por favor, ingresa un nodo de␣
↪la lista disponible.")
                continue
            # Check if origin and destination nodes are different
            if origen == destino:
                 print("El nodo origen y destino no pueden ser el mismo.")
                 continue
            # Exit the loop if origin and destination nodes are valid and␣
↪different
            break
        except ValueError:
            # Handle error if input is not an integer
            print("Entrada inválida. Por favor, ingresa un número entero.")
        except nx.NetworkXNoPath:
            # Handle error if there is no path between selected nodes
            print(f"No existe un camino entre el nodo {origen} y el nodo␣
↪{destino}.")
        except Exception as e:
            # Handle any other error
            print(f"Ocurrió un error: {e}")

    # Print the selected cities
    print(f"Ciudad de origen seleccionada: {G.nodes[origen]['ciudad']}")
    print(f"Ciudad de destino seleccionada: {G.nodes[destino]['ciudad']}")

    # Request the message to send from the user
    mensaje = input("Mensaje a enviar: ")

    # Request if the user wants to activate the spy
    while True:
        activar_espia_input = input("¿Activar espía (MITM)? (s/n): ").lower()
        # Check if the input is valid ('s' or 'n')
        if activar_espia_input in ['s', 'n']:
            activar_espia = activar_espia_input == 's' # Assign True if input␣
↪is 's', False if 'n'
            break # Exit the loop after valid input
```

```
        else:
            print("Entrada inválida. Por favor, ingresa 's' o 'n'.")

    # Call the function to send the message
    camino, espia = enviar_mensaje_unico(G, origen, destino, mensaje,␣
 ↪activar_espia)

    # Use the new Plotly visualization function
    mostrar_red_en_mapa_plotly(G, camino, origen, destino, espia)


def main():
    """
    Función principal para ejecutar el simulador de red cifrada con menú␣
 ↪interactivo.
    Configura la red inicialmente y luego indica al usuario cómo enviar mensajes
    usando una celda separada.
    Returns the created graph object or None if the user exits.
    """
    print("\n=== Visualización de Comunicación Segura con Cifrado RSA +␣
 ↪Vigenère en una Red Geográfica ===")
    # Obtiene el número de ciudades disponibles
    num_ciudades = len(CIUDADES_COORDS)

    G = None # Initialize G

    while True:
        print("\n--- Menú Principal ---")
        print("1. Introducir cantidad de nodos y configurar red")
        print("2. Salir")
        # Request the user to select a menu option
        opcion_inicial = input("Selecciona una opción: ")

        if opcion_inicial == '1':
            while True:
                try:
                    # Request the number of nodes for the network from the user
                    n = int(input(f"¿Cuántos nodos quieres en la red? (mínimo␣
 ↪3, máximo {num_ciudades}): "))
                    # Check if the number of nodes is within the allowed range
                    if n >= 3 and n <= num_ciudades:
                        G = crear_red_dispositivos(n) # Create the network of␣
 ↪devices

                        print(f"\nRed con {n} nodos creada exitosamente.")
                        print("Ahora puedes ejecutar la siguiente celda de␣
 ↪código para enviar un mensaje.")
```

```
                        return G # Exit the main function after creating the␣
 ↪network
                    else:
                        print(f"Cantidad de nodos no disponible. Por favor,␣
 ↪ingresa un número entre 3 y {num_ciudades}.")
                except ValueError:
                    # Handle error if input is not an integer
                    print("Entrada inválida. Por favor, ingresa un número␣
 ↪entero.")
        elif opcion_inicial == '2':
            print("Saliendo del simulador. ¡Hasta luego!")
            return None # Exit the main function and stop the program
        else:
            print("Opción no válida. Por favor, selecciona 1 o 2.")

# Execute the main function to set up the network
# The returned graph G will be available in the global scope
if __name__ == "__main__":
    # Store the created graph in a global variable to be accessible by the␣
 ↪message sending cell
    created_graph = main()
```

```
=== Visualización de Comunicación Segura con Cifrado RSA + Vigenère en una Red
Geográfica ===

--- Menú Principal ---
1. Introducir cantidad de nodos y configurar red
2. Salir
Selecciona una opción: 1
¿Cuántos nodos quieres en la red? (mínimo 3, máximo 27): 16

Red con 16 nodos creada exitosamente.
Ahora puedes ejecutar la siguiente celda de código para enviar un mensaje.
```

### 1.0.7 7. Celda menu principal

En resumen, el resultado de esta celda muestra la interacción que tuviste con el menú principal del simulador:

1.Ves el menú principal.

2.Seleccionas la opción 1 o la opcion 2 para salir.

3.Ingresas la cantidad de nodos que deceas.

4.El programa configura la red que deceas.

5.Te indica que ya puedes usar la siguiente celda para

enviar mensajes.

Básicamente, el output confirma que la configuración inicial de la red se completó correctamente según tus especificaciones.

[ ]:
```
if __name__ == "__main__":
    # Store the created graph in a global variable to be accessible by the␣
 ↪message sending cell
    created_graph = main()
```

```
=== Visualización de Comunicación Segura con Cifrado RSA + Vigenère en una Red
Geográfica ===

--- Menú Principal ---
1. Introducir cantidad de nodos y configurar red
2. Salir
Selecciona una opción: 16
Opción no válida. Por favor, selecciona 1 o 2.

--- Menú Principal ---
1. Introducir cantidad de nodos y configurar red
2. Salir
Selecciona una opción: 1
¿Cuántos nodos quieres en la red? (mínimo 3, máximo 27): 16

Red con 16 nodos creada exitosamente.
Ahora puedes ejecutar la siguiente celda de código para enviar un mensaje.
```

### 1.0.8 8. Celda para Enviar Mensajes

Ejecuta esta celda cada vez que quieras enviar un nuevo mensaje después de configurar la red.

Aquí tienes un resumen muy corto de esta celda:

Esta celda es simplemente un activador que, si ya configuraste la red ejecutando la celda anterior (celda 6), te permite enviar un mensaje llamando a la función interactiva de envío. Si no has configurado la red, te avisa que lo hagas primero.

Puedes ejecutar esta celda cada vez que quieras enviar un nuevo mensaje.

[ ]:
```
# Celda para enviar un mensaje después de configurar la red
# Ejecuta esta celda cada vez que quieras enviar un nuevo mensaje

# Verifica si el grafo se creó exitosamente en la primera celda
if 'created_graph' in globals() and created_graph is not None:
    # Llama a la función para configurar y enviar un mensaje usando el grafo␣
 ↪creado
    enviar_mensaje_interactivo(created_graph)
else:
```

```
    print("La red no ha sido configurada. Por favor, ejecuta la celda 6 de␣
↪código y configura la red primero.")
```

```
Nodos disponibles: [(0, 'Popayán'), (1, 'Montería'), (2, 'Medellín'), (3,
'Valledupar'), (4, 'Ibagué'), (5, 'Leticia'), (6, 'Quibdó'), (7,
'Villavicencio'), (8, 'Cartagena'), (9, 'Barranquilla'), (10, 'Florencia'), (11,
'Sincelejo'), (12, 'Pereira'), (13, 'Arauca'), (14, 'Cúcuta'), (15,
'Manizales')]

--- Configuración del Mensaje ---
Nodo origen: 3
Nodo destino: 9
Ciudad de origen seleccionada: Valledupar
Ciudad de destino seleccionada: Barranquilla
Mensaje a enviar: ytf
¿Activar espía (MITM)? (s/n): s

 Enviando mensaje de nodo 3 a nodo 9:
 - Camino: ['3 (Valledupar)', '8 (Cartagena)', '9 (Barranquilla)']
 - Clave cifrada (hex): 7c3f6489cdc25cc5e11fa6d45c4634af30e10ec2b41997c6a87c8a7a
3dbd4af5956b8cc94bfec5967b48b55b7de510610f3dfa8732853ea46864d37296678e7ceca0c22a
08a912470f22239227e34feac5899c2a66b88fbbceb74a4516c12ece67e7ce52d7a5ddea456d745d
d22afe4ab1662fbd3d4d240306d8b86b3bfd902f6473fe943b48e5ccb01ce5fbfbec5c25a8aedfe6
be72fb4e9adae0c7bdbd72514491a9711682725ebae83d7a7b383cc383b37570826520d88e1e74a1
f9fd95f240465dba64a7999ad9f719b3805ff99fd9224ad15187b71d6ca2c52601bd79dae2b61cc1
2cace7f839fbb4c68aaab7aa74a1b41f32f6dee74740070cfe2f9018
 - Mensaje cifrado: Qo=i;\r7:u>MX

 Nodo espía interceptó el mensaje en nodo 8 (Cartagena):
 - Contenido interceptado: Qo=i;\r7:u>MX
 - Clave simétrica descifrada (por el espía): s{.m?\44s6>-@/;w

 Mensaje recibido:
 - Descifrado: ytf
 - Integridad:  Válido
```

Este output muestra todo el proceso de envío de un mensaje que simulaste:

Te muestra los nodos disponibles y te pide elegir el origen, destino, el mensaje y si activar el espía.

Luego, simula el envío: te muestra el camino, la clave y el mensaje cifrados.

Si activaste el espía y hay nodos intermedios, te muestra qué ve el espía (el mensaje y, simuladamente, la clave).

Finalmente, te muestra el mensaje tal como lo recibió el destino (descifrado y verificando si la firma es válida).

Básicamente, es el log completo de un envío de mensaje simulado.

En el contexto de este código, el "grafo" (representado por la variable G, creada con la librería networkx) es básicamente un modelo de la **red de comunicaciones simulada.**

Piensa en él como un mapa de conexiones:

Cada **nodo** en el grafo representa un dispositivo (como un computador o un teléfono) ubicado en una ciudad colombiana. A cada nodo se le asignan claves de seguridad (RSA) y su ubicación geográfica.

Cada **arista** (la línea que conecta dos nodos) representa una conexión o un enlace de comunicación entre dos dispositivos.

Entonces, cuando hablamos del grafo, nos referimos a esa estructura que muestra cuántos dispositivos hay, dónde están ubicados y cómo están conectados entre sí. Es la base sobre la cual se simula el envío de mensajes.

EN CASO DE QUE NO TE MUESTRE EL (MITM) DEBES TENER EN CUNETA LO SIGUIENTE:

La función **enviar_mensaje_unico,** que simula el envío del mensaje, es la encargada de determinar si habrá un espía y dónde estará ubicado. Aquí te detallo los pasos relevantes:

**Cálculo del Camino:** Primero, la función calcula el camino más corto entre el nodo de origen y el nodo de destino utilizando **nx.shortest_path(G, origen, destino)**. Este camino es una lista de nodos por los que pasará el mensaje. **Verificación de la Longitud del Camino:** La simulación del espía solo tiene sentido si hay nodos intermedios por los que el mensaje debe pasar entre el origen y el destino. Si el camino es directo, es decir, solo involucra al nodo de origen y al nodo de destino **(len(camino) == 2)**, no hay un punto intermedio para que un espía intercepte el mensaje en ruta. **Selección del Espía (si es posible):**

La línea clave es:

**espia = random.choice(camino[1:-1]) if espia_activo and len(camino) > 2 else None**

Esta línea hace lo siguiente:

Verifica si **espia_activo** es **True** (si seleccionaste "s" para activar el espía).

Verifica si la longitud del camino es mayor que 2 **(len(camino) > 2)**. Esto asegura que haya al menos un nodo entre el origen y el destino.

Si ambas condiciones son verdaderas, selecciona aleatoriamente **(random.choice**) un nodo de la lista **camino[1:-1]**. Esta porción de la lista camino incluye todos los nodos del camino excepto el nodo de origen (índice 0) y el nodo de destino (último índice)

Si alguna de las condiciones no se cumple (espía no activo o camino con solo 2 nodos), la variable espia se establece en None.

**Visualización:** La función **mostrar_red_en_mapa_plotly** recibe la variable espia. Dentro de esta función, se asigna un color especial ('purple') y un tamaño mayor (20) al nodo si su índice coincide con el valor de espia. Si espia es None, ningún nodo cumplirá esta condición y, por lo tanto, ningún nodo se visualizará como espía.

En resumen, aunque el código te pregunta si quieres activar el espía, la simulación real y la visualización del espía solo ocurren si hay nodos intermedios en el camino del mensaje donde la

interceptación puede tener lugar. Si el camino más corto es directo, la variable espia se mantiene como None, y por eso no lo ves en el mapa.