“Año del Bicentenario, de la consolidación de nuestra Independencia, y de la conmemoración de las heroicas batallas de Junín y Ayacucho”

CARRERA: INGENIERÍA DE SOFTWARE

PROYECTO FINAL

VIRTUALIZACIÓN DE SISTEMAS OPERATIVOS Y LA NUBE

ESTUDIANTES:

Carpio Guevara Rodrigo Sebastian

Flores Leon Miguel Angel

DOCENTE:

Luque Mamani Edson Fracisco

Arequipa, Perú

22 de junio de 2025

Índice

I. Introducción

En este informe se presenta el desarrollo del proyecto final, el cual tiene el objetivo de analizar y visualizar la estructura de un grafo para una red social, esto con el proposito de descubrir patrones de interez y obtener informacion sobre la conectividad social, las estrcuturas de la comunidad y propiedades de la red.

Esta red social es conformada por dos bases de datos, una formada por diez millones de usuarios y la segunda esta formada por las ubicaciones de estos usuraios, Este conjunto de datos es un subconjunto de todos los usuarios de la red social, junto con sus conexiones y ubicaciones el grafo que crea el programa consiera a los usuarios como nodos y sus conexiones como aristas, por lo que forman un componente conectado del grafo total.

Para el análisis de los datos optamos por utilizar la libreria ïgraph.[es]to debido a la velocidad con la que trabaja y la eficiencia en el uso de los recursos, factores en los que supera a otras librerias como ”pandas”.

La construcción del grafo se logra cargandolo primero y luego almacenandolo en formato pickle, luego el programa correra pruebas en el grafo y en los datos.

El objetivo de este programa es hacer un grafo realista y funcional de la red social con el objetivo de analizar a las distintas comunidades que esta posee y comprender como se relacionan, ademas de eso se evalua el uso de algoritmos para la carga y lectura e datos con el proposito de encontrar la opcion mas eficiente”.

II. Librerias necesarias

II-A. random

Proporciona herramientas para generar números pseudoaleatorios, seleccionar elementos al azar de listas y crear simulaciones estocásticas. Sus usos y caracteristicas principales son: Uso en simulaciones, pruebas, generación de datos de ejemplo y algoritmos que requieran aleatoriedad (muestreos, juegos, pruebas A/B).

II-B. igraph

Librería para crear, manipular y analizar grafos de manera eficaz, soportando análisis de grandes redes. Sus usos y caracteristicas principales son: Construir grafos, añadir vértices y aristas, realizar análisis de centralidad, detectar comunidades, caminos mínimos y otras métricas de análisis de redes.

II-C. time

Proporciona herramientas para medir y administrar el tiempo en programas de Python. Sus usos y caracteristicas principales son: Calcular duración de ejecuciones, crear delays, medir rendimiento de algoritmos y análisis de eficiencia temporal.

II-D. colorsys

Es un módulo incluido en la biblioteca estándar que proporciona funciones para convertir colores entre diferentes sistemas de coordenadas de color. Sus usos y caracteristicas principales son: Ajustar brillo, generar paletas de colores armónicos, convercion entre espacios de color, manipulación de imágenes a nivel de píxeles y conversiones para visualizaciones gráficas .

II-E. logging

Ofrece herramientas para generar registros de ejecución (logs) con diferentes niveles de importancia (INFO, WARNING, ERROR).Sus usos y caracteristicas principales son: Registrar detalles del flujo de ejecución, diagnosticar errores, guardar registros para análisis y seguimiento de actividades en sistemas de software.

II-F. fileinput

Facilita la iteración sobre varias líneas de uno o varios archivos de manera eficaz. Sus usos y caracteristicas principales son: Procesar datos línea por línea desde archivos grandes, especialmente útil para procesamientos en flujo donde no se quiere cargar el documento entero en memoria.

II-G. pickle

Permite serializar (convertir en binario) y deserializar (cargar) objetos de Python para guardarlos en disco. Sus usos y caracteristicas principales son: Guardar modelos de datos, estructuras de grafos, listas, diccionarios u otros objetos para reutilizarlos rápidamente sin necesidad de recalcular.

II-H. gc (Garbage Collector)

Administra la recolección de basura en Python, liberando memoria de objetos no usados. Sus usos y caracteristicas principales son: Optimizar el uso de memoria en procesamientos intensivos, liberar memoria de manera manual para garantizar la eficiencia en análisis de grandes volúmenes de datos.

II-I. multiprocessing

Facilita la ejecución de tareas en paralelo utilizando múltiples núcleos de la CPU. Sus usos y caracteristicas principales son: Paralelizar el procesamiento de datos para reducir tiempos de ejecución, ideal para análisis de grandes volúmenes de datos y para aprovechar al máximo la capacidad del hardware.

II-J. itertools

Proporciona herramientas para crear iteradores eficientes que ayudan a construir bucles y procesamientos complejos de datos. Sus usos y caracteristicas principales son: Ideal para generar combinaciones, permutaciones, secuencias infinitas, productos cartesianos y otros patrones de iteración para análisis y manipulación de datos.

II-K. folium

Librería para crear mapas interactivos a través de datos geoespaciales, basada en Leaflet.js. Sus usos y caracteristicas principales son: Visualizar ubicaciones, representar datos geográficos en mapas interactivos, crear informes y dashboards con elementos de mapeo para análisis espacial.

II-L. *math*

Ofrece funciones matemáticas estándar para cálculos básicos y especializados. Sus usos y caracteristicas principales son: Realizar cálculos trigonométricos, exponenciales, logarítmicos y otras operaciones matemáticas para análisis numérico y científico.

II-M. *os*

Esta libreria es un puente para interactuar directamente con el sistema operativo. Te permite manejar archivos y directorios (crear, borrar, renombrar), manipular rutas de forma segura entre diferentes sistemas (Windows, Linux, macOS), y acceder a comandos o variables de entorno.Sus usos y caracteristicas principales son: Crear o elimina archivos y carpetas, construye y verifica rutas de forma inteligente y ejecuta comandos del sistema ademas de leer variables de entorno.

II-N. *typing*

Proporciona herramientas para declarar anotaciones de tipo en Python, facilitando la legibilidad y ayudando a herramientas de análisis de tipos estáticos. Sus usos y caracteristicas principales son: Mejorar la calidad del código al especificar tipos esperados de variables, parámetros y valores de retorno en proyectos grandes o con equipos multidisciplinarios.

II-Ñ. *webbrowser*

Es una libreria simple para abrir URLs (enlaces web) en el navegador predeterminado del sistema. Es ideal para dirigir a los usuarios a documentación, resultados en línea o cualquier página web externa sin salir del programa. Sus usos y caracteristicas principales son: Abrir URLs especificas, abrir nuevas ventanas o pestañas de manera conveniente para el usuario.

II-O. *tqdm*

Librería para crear barras de progreso en la consola o entornos interactivos, facilitando el seguimiento de la ejecución de bucles y tareas prolongadas. Sus usos y caracteristicas principales son: Visualizar el progreso de iteraciones, procesamientos masivos y análisis de datos para evaluar duración y estimar el tiempo restante en tiempo real.

II-P. *matplotlib*

Biblioteca de visualización de datos para crear gráficos estáticos, animados e interactivos en 2D. Sus usos y caracteristicas principales son: Construir gráficos de línea, dispersión, barras, histogramas, mapas de calor y otros para representar datos, comunicar conclusiones e interpretar resultados en análisis científico y técnico.

II-Q. *heapq*

Esta libreria implementa el algoritmo de montículo (heap) para crear y manipular colas de prioridad eficientes. No crea un tipo de datos de heap separado, sino que trabaja directamente con listas de Python, transformándolas en min-heaps (donde el elemento más pequeño siempre está al principio). Sus usos y caracteristicas principales son: Ideal para procesar elementos en orden de importancia, eficiente para obtener los N elementos más pequeños o grandes de una colección y convertir listas en heaps.

II-R. *collections*

Esta libreria ofrece estructuras de datos especializadas que extienden las capacidades de las listas y diccionarios básicos. Esencialmente, te da herramientas más eficientes y convenientes para manejar situaciones comunes, haciendo el código más limpio y rápido. Usos y caracteristicas principales: Puede Añadir o eliminar elementos eficientemente tanto por el principio como por el final de una doble cola, permite crear tuplas con campos nombrados haciendo el código más legible y auto-documentado.

III. Creación del grafo

Construye un grafo dirigido en igraph a gran escala procesando ubicaciones y conexiones de usuarios en paralelo, asigna atributos de latitud y longitud a cada vértice, y lo guarda en formato pickle para análisis masivos en entornos de big data.

IV. Explicación del Código

IV-A. *Codigo de createdata.py*

```python
import random

def generar_ubicaciones(num_ubicaciones, nombre_archivo="1_million_location.txt"):
    """
    Genera un archivo de texto con coordenadas de latitud y longitud aleatorias.

    Args:
```

```
        num_ubicaciones (int): El número de ubicaciones a generar.
        nombre_archivo (str, opcional): El nombre del archivo a crear.
            Por defecto es "Xnumber_location.txt".
    """
    try:
        with open(nombre_archivo, 'w') as archivo:
            for _ in range(num_ubicaciones):
                # Genera latitud y longitud aleatorias dentro de rangos razonables
                latitud = random.uniform(-90, 90) # Valores de latitud entre -84 y -82
                longitud = random.uniform(-180, 180) # Valores de longitud 134 y 136
                # Escribe la coordenada en el archivo con el formato especificado
                archivo.write(f"{latitud},{longitud}\n")
        print(f"Se ha generado el archivo '{nombre_archivo}' con {num_ubicaciones} ubicaciones.")
    except Exception as e:
        print(f"Ocurrió un error al generar el archivo: {e}")

def generar_conexiones(num_conexiones, nombre_archivo="1_million_user.txt") :
    """
    Genera un archivo de texto con conexiones de aristas aleatorias, simulando usuarios
    y sus interacciones con ubicaciones.

    Args:
        num_conexiones (int): El número de conexiones de aristas a generar (filas en el archivo).
        nombre_archivo (str, opcional): El nombre del archivo a crear.
            Por defecto es "Xnumber_user.txt".
    """
    try:
        with open(nombre_archivo, 'w') as archivo:
            for i in range(num_conexiones):
                # Genera un número aleatorio de ubicaciones visitadas por el usuario
                num_ubicaciones_visitadas = random.randint(0, 100) # Cada usuario visita entre 1 y 20 ubicaciones
                # Genera una lista de ubicaciones visitadas (índices)
                ubicaciones_visitadas = [random.randint(1, 1000700) for _ in range(num_ubicaciones_visitadas)] # Asume que tienes 10,000 ubicaciones
                # Escribe las conexiones en el archivo con el formato especificado
                archivo.write(f"{i},{','.join(map(str, ubicaciones_visitadas))}\n")
        print(f"Se ha generado el archivo '{nombre_archivo}' con {num_conexiones} conexiones.")
    except Exception as e:
        print(f"Ocurrió un error al generar el archivo: {e}")

if __name__ == "__main__":
    # Genera 10,000 ubicaciones en el archivo "10_thousand_location.txt"
    generar_ubicaciones(1000000)
    generar_conexiones(1000000)
    #generar_ubicaciones(10, "test_location.txt") #para pruebas
```

Listing 1. Codigo para la generacion de datos

import random: Importa la librería estándar para generar números pseudoaleatorios, utilizado para crear coordenadas y datos simulados para análisis de grafos.

generar_ubicaciones(): Crea un archivo de texto con coordenadas de latitud y longitud aleatorias para representar nodos en análisis masivos de datos geoespaciales.
random.uniform(-90, 90) / random.uniform(-180, 180): Genera latitudes y longitudes al azar en rangos válidos para representar ubicaciones geográficas en pruebas de análisis de grafos.
generar_conexiones(): Crea un archivo de texto simulando conexiones de usuarios hacia ubicaciones específicas para representar relaciones en análisis de grafos masivos.
random.randint(): Genera un entero aleatorio para asignar visitas de usuario a ubicaciones, utilizado para evaluar escalabilidad y robustez de algoritmos de análisis de grafos.
with open(...): Abre archivos para escritura de manera segura, garantizando su cierre para crear y guardar datos masivos para análisis de grafos y pruebas de rendimiento.
f-string: Utilizado para crear registros de texto estructurados para representar nodos y aristas en análisis de grafos masivos y simulaciones.
__main__: Punto de entrada del programa para invocar la generación de datos masivos de ubicaciones y conexiones, utilizado para pruebas de análisis de grafos.

V. Explicación del Código

V-A. *Codigo de grapy paralel.py*

```
import igraph as ig
import time
import logging
import fileinput
import pickle
import gc
import multiprocessing as mp
from itertools import chain

NUM_NODOS = 10_000_000
```

```python

# Configuración del logging para registrar información y advertencias en archivo y consola
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("grafo_paralelizado.log", mode='w', encoding='utf-8'),
        logging.StreamHandler()
    ]
)

def cargar_ubicaciones_directo(ubicaciones_path, max_nodos):
    logging.info(f"Cargando un máximo de {max_nodos} ubicaciones desde el archivo...")
    latitudes = []
    longitudes = []
    with open(ubicaciones_path, 'r') as f:
        for i, line in enumerate(f):
            if i >= max_nodos:
                logging.info(f"Límite de {max_nodos} nodos alcanzado. Se detiene la lectura de ubicaciones.")
                break
            try:
                # El formato en el archivo original parece ser lat, lon
                lat_str, lon_str = line.strip().split(',')
                latitudes.append(float(lat_str))
                longitudes.append(float(lon_str))
            except ValueError:
                logging.warning(f"Línea malformada en el archivo de ubicaciones: '{line.strip()}'")

    ubicaciones = list(zip(latitudes, longitudes))
    num_nodos_reales = len(ubicaciones)

    # Advertir si el archivo tiene menos nodos que el esperado
    if num_nodos_reales < max_nodos:
        logging.warning(f"El archivo de ubicaciones contiene {num_nodos_reales} nodos, menos que el máximo esperado de {max_nodos}.")

    logging.info(f"Se cargaron {num_nodos_reales} ubicaciones.")
    return ubicaciones, num_nodos_reales, {'lat': latitudes, 'lon': longitudes}

def procesar_linea_usuarios(linea_idx_contenido, num_nodos_max):
    idx, linea = linea_idx_contenido
    aristas_locales = []
    conexiones = set()
    for x in linea.strip().split(','):
        x_strip = x.strip()
        if x_strip.isdigit():
            dst = int(x_strip)
            # Validar contra el número máximo de nodos permitidos
            if 1 <= dst <= num_nodos_max:
                conexiones.add(dst - 1) # Ajuste a índice base 0

    # El nodo origen (idx-1) también debe ser válido
    if 0 <= (idx - 1) < num_nodos_max:
        aristas_locales.extend((idx - 1, dst) for dst in conexiones)

    return aristas_locales

def procesar_usuarios_paralelizado(usuarios_path, num_nodos, num_procesos=mp.cpu_count()):
    logging.info(f"Procesando usuarios para {num_nodos} nodos en paralelo con {num_procesos} procesos...")
    start_time_procesamiento = time.time()

    with fileinput.input(usuarios_path) as f:
        # Creamos un generador que solo lee hasta la línea 'num_nodos'
        lineas_a_procesar = ((idx, linea) for idx, linea in enumerate(f, start=1) if idx <= num_nodos)

        with mp.Pool(processes=num_procesos) as pool:
            # Pasamos num_nodos a cada proceso para la validación
            resultados = pool.starmap(procesar_linea_usuarios, [(item, num_nodos) for item in lineas_a_procesar])

    # Aplanar la lista de resultados
    aristas = list(chain.from_iterable(resultados))

    # El filtrado ya se hace dentro de 'procesar_linea_usuarios', pero una verificación final no está de más.
    aristas_filtradas = [(src, dst) for src, dst in aristas if 0 <= src < num_nodos and 0 <= dst < num_nodos]

    logging.info(f"Se procesaron {len(aristas_filtradas)} aristas en {time.time() - start_time_procesamiento:.2f} s.")
    return aristas_filtradas

def crear_grafo_igraph_paralelizado(ubicaciones_path, usuarios_path, output_grafo_path, max_nodos):
    start_time = time.time()

    # 1. Cargar ubicaciones, usando max_nodos como límite.
    ubicaciones, num_nodos_reales, atributos_ubicacion = cargar_ubicaciones_directo(ubicaciones_path, max_nodos)

    # 2. Procesar conexiones, usando el nú mero real de nodos como límite para la consistencia.
    aristas = procesar_usuarios_paralelizado(usuarios_path, num_nodos_reales)

```

```
    logging.info("Creando grafo de igraph...")
    g = ig.Graph(directed=True)
    # Añadimos la cantidad real de vértices encontrados
    g.add_vertices(num_nodos_reales)
    g.add_edges(aristas)

    # Añadir atributos de ubicación al grafo
    for key, values in atributos_ubicacion.items():
        g.vs[key] = values

    # Guardar el grafo en un archivo pickle
    logging.info(f"Guardando grafo con atributos en '{output_grafo_path}'...")
    with open(output_grafo_path, 'wb') as f:
        pickle.dump(g, f, protocol=pickle.HIGHEST_PROTOCOL)

    logging.info(f"Grafo guardado. Tiempo total: {time.time() - start_time:.2f} s")
    del ubicaciones, aristas, atributos_ubicacion, g
    gc.collect()

if __name__ == "__main__":
    ubicaciones_archivo = '10_million_location.txt'
    usuarios_archivo = '10_million_user.txt'
    grafo_archivo = 'grafo_igraph_paralelizado.pkl'

    # Crear y guardar el grafo, pasando la constante global como el límite máximo.
    crear_grafo_igraph_paralelizado(ubicaciones_archivo, usuarios_archivo, grafo_archivo, max_nodos=NUM_NODOS)

    gc.collect()
```

Listing 2. Codigo para lectura de datos y carga del grafo

import igraph as ig: Importa la librería de análisis de grafos para representar la estructura de nodos y aristas de manera altamente eficiente.
import time: Importa herramientas para medir duración de cálculos y evaluar rendimiento de cada fase del procesamiento.
import logging: Importa herramientas para crear registros de información, advertencias y errores tanto en consola como en archivo de texto.
import fileinput: Importa herramientas para procesar de manera eficaz grandes archivos línea por línea.
import pickle: Importa herramientas para guardar y recuperar objetos de Python (usado para guardar el grafo procesado).
import gc: Importa herramientas para invocar al recolector de basura y liberar memoria no utilizada tras procesar grandes volúmenes de datos.
import multiprocessing as mp: Importa herramientas para realizar procesamiento en paralelo, acelerando la generación de aristas para grafos masivos.
from itertools import chain: Importa herramientas para aplanar listas anidadas (usado para combinar todas las aristas en una única estructura).
NUM_NODOS = 10_000_000: Define el máximo de nodos que serán procesados para crear el grafo, utilizado para filtrar entradas no válidas y controlar memoria.
logging.basicConfig(...): Configura el registro de actividades para guardar mensajes de diagnóstico en un archivo y mostrar información en consola.
cargar_ubicaciones_directo(...): Lee las ubicaciones desde el archivo, limitándolas a un máximo de nodos. Valida cada línea para obtener pares (latitud, longitud) y almacena estos atributos para asignarlos al grafo.
procesar_linea_usuarios(...): Procesa cada línea de usuarios para obtener las conexiones (aristas). Valida que los destinos estén en el rango de nodos existentes y los almacena para crear la estructura de la red.
procesar_usuarios_paralelizado(...): Ejecuta el procesamiento de todas las conexiones en paralelo para maximizar la velocidad en grafos masivos, utilizando todos los núcleos de la máquina para obtener las aristas rápidamente.
crear_grafo_igraph_paralelizado(...): Orquesta la creación final del grafo: carga las ubicaciones, procesa todas las conexiones, arma la estructura de igraph y guarda el resultado como pickle para análisis posterior. Incluye liberación de memoria para evitar saturación.
if __name__ == ¨__main__¨: Bloque de ejecución principal que coordina todas las etapas para crear y guardar un grafo a gran escala, con rutas de archivos específicas para ubicaciones, usuarios y el grafo resultante.

V-B. Codigo de dfs.py

```
import pickle
import time
import logging
import random
import igraph as ig
from typing import Optional, List, Tuple
import folium
from folium.plugins import BeautifyIcon
import colorsys

# --- Constantes y Configuración ---
SOURCE_NODE = 1
NUM_CAPAS_DESEADAS = 10
# Reduce el camino de ~10 millones de nodos a este número antes de crear el mapa.
MAX_NODOS_PARA_MAPA = 50

# --- Configuración del Logging ---
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s',
                    handlers=[logging.FileHandler("analisis_dfs.log", mode='w', encoding='utf-8'),
```

```
                        logging.StreamHandler()])

# --- Funciones Auxiliares (Sin cambios) ---
def cargar_grafo(grafo_path: str) -> Optional[ig.Graph]:
    """Carga un grafo desde un archivo pickle."""
    logging.info(f"Cargando el grafo desde '{grafo_path}'...")
    start_time = time.time()
    try:
        with open(grafo_path, 'rb') as f:
            g = pickle.load(f)
        end_time = time.time()
        logging.info(f"Grafo cargado en {end_time - start_time:.2f} segundos. {g.summary()}")
        return g
    except Exception as e:
        logging.error(f"Error crítico al cargar el grafo: {e}")
        return None

def dfs_con_profundidad(graph: ig.Graph, source: int) -> Tuple[List[int], float, int, int]:
    """Realiza un recorrido DFS y devuelve el camino, el tiempo, y el nodo más profundo."""
    logging.info(f"Iniciando travesía DFS desde el nodo {source} y calculando profundidad.")
    start_time = time.time()
    stack = [(source, 0)]
    visitados = {source}
    nodos_explorados_en_orden = []
    max_profundidad, nodo_mas_profundo = 0, source
    while stack:
        current_node, current_depth = stack.pop()
        nodos_explorados_en_orden.append(current_node)
        if current_depth > max_profundidad:
            max_profundidad, nodo_mas_profundo = current_depth, current_node
        for neighbor in reversed(graph.neighbors(current_node, mode='out')):
            if neighbor not in visitados:
                visitados.add(neighbor)
                stack.append((neighbor, current_depth + 1))
    tiempo_total = time.time() - start_time
    logging.info(f"Travesía DFS completada. Nodos: {len(nodos_explorados_en_orden)}. Nodo más profundo: {nodo_mas_profundo} (prof: {max_profundidad}).")
    return nodos_explorados_en_orden, tiempo_total, nodo_mas_profundo, max_profundidad

def generar_color_gradiente_hls(paso_actual: int, total_pasos: int) -> str:
    """Genera un color en formato hexadecimal que va de azul (inicio) a rojo (final)."""
    if total_pasos <= 1: return "#FF0000"
    hue = (240/360) * (1 - (paso_actual / (total_pasos - 1)))
    r, g, b = colorsys.hls_to_rgb(hue, 0.5, 0.8)
    return f'#{int(r*255):02x}{int(g*255):02x}{int(b*255):02x}'


# --- FUNCIÓN DE VISUALIZACIÓN OPTIMIZADA CON MARCADORES DE CAPA ---
def crear_mapa_dfs_por_capas(
    graph: ig.Graph,
    source_node: int,
    camino_a_dibujar: List[int], # Recibe la lista ya reducida
    nodo_mas_profundo: int,
    ultimo_nodo_real: int
) -> Optional[folium.Map]:
    """
    Crea un mapa estático usando un camino pre-reducido y añade marcadores de inicio/fin por capa.
    """
    logging.info(f"Creando mapa estático para dividirse en {NUM_CAPAS_DESEADAS} capas.")

    start_coords = (graph.vs[source_node]['lat'], graph.vs[source_node]['lon'])
    m = folium.Map(location=start_coords, zoom_start=3, tiles="CartoDB positron")

    total_pasos_visuales = len(camino_a_dibujar)

    # Capa de Vértices
    fg_vertices = folium.FeatureGroup(name=f"Vértices del Camino Visualizado ({total_pasos_visuales})", show=True)
    for node_id in camino_a_dibujar:
        coords = (graph.vs[node_id]['lat'], graph.vs[node_id]['lon'])
        folium.CircleMarker(
            location=coords, radius=3, color='#3186cc', fill=True, fill_color='#3186cc', fill_opacity=0.6,
            popup=f"Nodo {node_id}"
        ).add_to(fg_vertices)
    m.add_child(fg_vertices)

    # Lógica para dividir el camino en capas
    num_segmentos = total_pasos_visuales - 1
    segmentos_por_capa = max(1, num_segmentos // NUM_CAPAS_DESEADAS)

    for i in range(0, num_segmentos, segmentos_por_capa):
        start_paso, end_paso = i + 1, min(i + segmentos_por_capa, num_segmentos)

        fg_camino_segmentado = folium.FeatureGroup(name=f"Camino (Pasos {start_paso}-{end_paso})", show=(i == 0))

        # Dibuja las líneas de la capa
        for j in range(i, end_paso):
            coords = [(graph.vs[camino_a_dibujar[j]]['lat'], graph.vs[camino_a_dibujar[j]]['lon']), (graph.vs[camino_a_dibujar[j+1]]['lat'], graph.vs[camino_a_dibujar[j+1]]['lon
```

```
                ↪ ’])]
106             color_segmento =
                    ↪ generar_color_gradiente_hls
                    ↪ (j, total_pasos_visuales)
107             folium.PolyLine(locations=coords,
                    ↪ color=color_segmento,
                    ↪ weight=2.5, opacity=1.0).
                    ↪ add_to(fg_camino_segmentado
                    ↪ )
108
109         # --- NUEVO: AÑADIR MARCADORES DE
                ↪ INICIO Y FIN PARA ESTA CAPA
                ↪ ---
110         start_node_layer = camino_a_dibujar[i
                ↪ ]
111         end_node_layer = camino_a_dibujar[
                ↪ end_paso]
112
113         folium.CircleMarker(
114             location=(graph.vs[
                    ↪ start_node_layer][’lat’],
                    ↪ graph.vs[start_node_layer][
                    ↪ ’lon’]),
115             radius=5, color=’green’, fill=True
                    ↪ , fill_color=’lightgreen’,
116             tooltip=f"Inicio Capa: Paso {
                    ↪ start_paso}"
117         ).add_to(fg_camino_segmentado)
118
119         folium.CircleMarker(
120             location=(graph.vs[end_node_layer
                    ↪ ][’lat’], graph.vs[
                    ↪ end_node_layer][’lon’]),
121             radius=5, color=’red’, fill=True,
                    ↪ fill_color=’#ff7f7f’, # Un
                    ↪ rojo más claro
122             tooltip=f"Fin Capa: Paso {end_paso
                    ↪ }"
123         ).add_to(fg_camino_segmentado)
124
125         m.add_child(fg_camino_segmentado)
126
127     # Marcadores Especiales (Generales)
128     folium.Marker(location=start_coords,
            ↪ popup=f"INICIO DFS: Nodo {
            ↪ source_node}", tooltip="Punto de
            ↪ Inicio General",
129                   icon=BeautifyIcon(icon=’play’
                          ↪ , border_color=’#2
                          ↪ ca02c’, text_color=’#2
                          ↪ ca02c’)).add_to(m)
130     end_real_coords = (graph.vs[
            ↪ ultimo_nodo_real][’lat’], graph.
            ↪ vs[ultimo_nodo_real][’lon’])
131     folium.Marker(location=end_real_coords,
            ↪ popup=f"FIN DEL RECORRIDO: Nodo {
            ↪ ultimo_nodo_real}", tooltip="Ú
            ↪ ltimo Nodo Visitado General",
132                   icon=BeautifyIcon(icon=’stop’
                          ↪ , border_color=’#
                          ↪ d62728’, text_color=’#
                          ↪ d62728’)).add_to(m)
133     deepest_coords = (graph.vs[
            ↪ nodo_mas_profundo][’lat’], graph.
            ↪ vs[nodo_mas_profundo][’lon’])
134     folium.Marker(location=deepest_coords,
            ↪ popup=f"NODO MÁS PROFUNDO: Nodo {
            ↪ nodo_mas_profundo}", tooltip="
            ↪ Nodo Más Profundo General",
135                   icon=BeautifyIcon(icon=’star’
                          ↪ , border_color=’
                          ↪ #800080’, text_color=’
                          ↪ #800080’, spin=True)).
                          ↪ add_to(m)
136
137     folium.LayerControl(collapsed=False).
            ↪ add_to(m)
138     return m
139
140 # --- Bloque Principal de Ejecución ---
141 if __name__ == "__main__":
142     GRAFO_PKL_ENTRADA = ’
            ↪ grafo_igraph_paralelizado.pkl’
143     MAPA_HTML_SALIDA = ’
            ↪ mapa_dfs_con_marcadores_de_capa.
            ↪ html’
144
145     mi_grafo = cargar_grafo(
            ↪ GRAFO_PKL_ENTRADA)
146
147     if mi_grafo and mi_grafo.vs:
148         nodos_visitados_dfs, tiempo_dfs,
                ↪ nodo_profundo, prof_max =
                ↪ dfs_con_profundidad(mi_grafo,
                ↪ SOURCE_NODE)
149
150         if nodos_visitados_dfs:
151             logging.info(f"Reduciendo camino
                    ↪ de {len(nodos_visitados_dfs
                    ↪ )} nodos a un máximo de {
                    ↪ MAX_NODOS_PARA_MAPA} para
                    ↪ visualización.")
152             if len(nodos_visitados_dfs) >
                    ↪ MAX_NODOS_PARA_MAPA:
153                 step = len(nodos_visitados_dfs)
                        ↪  // MAX_NODOS_PARA_MAPA
154                 camino_para_mapa =
                        ↪ nodos_visitados_dfs[::
                        ↪ step]
155                 if camino_para_mapa[-1] !=
                        ↪ nodos_visitados_dfs[-1]:
156                     camino_para_mapa.append(
                            ↪ nodos_visitados_dfs
                            ↪ [-1])
157             else:
158                 camino_para_mapa =
                        ↪ nodos_visitados_dfs
159
160             logging.info(f"El camino para el
                    ↪ mapa ahora tiene {len(
                    ↪ camino_para_mapa)} nodos.
                    ↪ Creando mapa...")
161
162             mapa = crear_mapa_dfs_por_capas(
163                 graph=mi_grafo,
164                 source_node=SOURCE_NODE,
165                 camino_a_dibujar=
                        ↪ camino_para_mapa,
166                 nodo_mas_profundo=nodo_profundo
                        ↪ ,
167                 ultimo_nodo_real=
                        ↪ nodos_visitados_dfs[-1]
168             )
169
170             if mapa:
171                 mapa.save(MAPA_HTML_SALIDA)
172                 print("\n" + "="*60)
173                 print(" MAPA OPTIMIZADO CREADO
                        ↪ CON ÉXITO")
174                 print("="*60)
175                 print(f"Mapa interactivo
                        ↪ guardado en: ’{
                        ↪ MAPA_HTML_SALIDA}’")
176                 print("\nNovedades en el mapa:"
                        ↪ )
177                 print(f" - El camino visual se
                        ↪ ha dividido en ~{
                        ↪ NUM_CAPAS_DESEADAS}
                        ↪ capas.")
178                 print(" - CADA CAPA DE CAMINO
                        ↪ AHORA TIENE SU PROPIO
                        ↪ MARCADOR DE INICIO Y FIN
                        ↪ .")
```

```
            print(" - Marcador especial ()
                ↪ para el nodo más
                ↪ profundo.")
        print("="*60)
    else:
        logging.error("El grafo no se pudo
            ↪ cargar o está vacío.")
```

Listing 3. Codigo para el mapa dfs

import pickle, time, logging, random, igraph as ig, folium, from folium.plugins import BeautifyIcon, colorsys: Importa todas las herramientas para procesar, guardar/cargar grafos, realizar recorridos en profundidad (DFS), crear mapas interactivos, representar elementos con iconos y generar colores en gradiente para representar caminos.
SOURCE_NODE, NUM_CAPAS_DESEADAS, MAX_NODOS_PARA_MAPA: Declara las constantes para:

- **SOURCE_NODE**: Nodo de inicio para el DFS.
- **NUM_CAPAS_DESEADAS**: Divide la representación final del camino en este número de capas para organizarlo.
- **MAX_NODOS_PARA_MAPA**: Limita la cantidad de nodos para representar en el mapa, evitando sobrecarga en visualización.

logging.basicConfig(...): Configura el registro para guardar todas las actividades en un archivo de log y mostrarlas en consola, facilitando el seguimiento de la ejecución.
cargar_grafo(grafo_path): Carga un grafo serializado en formato Pickle:

- Abre el archivo.
- Verifica la integridad de los datos cargados.
- Reporta en el log el éxito o falla en la operación.

dfs_con_profundidad(graph, source): Realiza una búsqueda en profundidad (DFS) para obtener:

- El camino visitado (en orden).
- El nodo más profundo alcanzado.
- El tiempo de ejecución.
- El nivel de profundidad alcanzado.

Utiliza una pila para garantizar un orden LIFO en la expansión de nodos, registrando cada nodo alcanzado junto con su profundidad.
generar_color_gradiente_hls(paso_actual, total_pasos): Crea un color en formato hexadecimal para representar cada paso del camino, variando del color azul al rojo según la posición en la secuencia.
crear_mapa_dfs_por_capas(...): Construye un mapa de Folium donde:

- Se representan todos los nodos alcanzados por el DFS.
- Se dividen las aristas en capas para representar visualmente diferentes tramos del camino.
- Se añaden marcadores especiales para:
  - El inicio de la búsqueda.
  - El final del camino alcanzado.
  - El nodo alcanzado a mayor profundidad.
  - El inicio y fin de cada capa de camino.
- Se asigna un color específico para cada arista según la posición en la secuencia para representar el camino de manera gradual.
- Se organiza la visualización mediante 'FeatureGroups' para permitir alternar qué capa mostrar.

if __name__ == ¨__main__¨: Se ejecuta el flujo principal:

- Se carga el grafo.
- Se ejecuta el DFS para obtener la secuencia de nodos alcanzados, junto con la posición del nodo más profundo.
- Se reduce la cantidad de nodos para representar en el mapa, para garantizar la eficiencia al representar caminos largos.
- Se invoca la creación del mapa para representar todas las capas junto a los marcadores de interés.
- Se guarda el mapa en formato HTML para su posterior visualización.

V-C. Codigo de dijkstra.py

```
# analisis_completo_de_red.py
import pickle
import time
import logging
import random
from math import radians, sin, cos, sqrt,
    ↪ atan2
from itertools import combinations
import igraph as ig
from typing import Dict, Optional, Tuple,
    ↪ List, Any, Set
import folium
from folium.plugins import BeautifyIcon
from tqdm import tqdm
import heapq

# --- Configuración del Logging ---
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(
        ↪ message)s',
    handlers=[
        logging.FileHandler("analisis_de_red.
            ↪ log", mode='w', encoding='utf
            ↪ -8'),
        logging.StreamHandler()
    ]
)

# --- Funciones de Carga y Cálculo ---

def cargar_grafo(grafo_path: str) ->
    ↪ Optional[ig.Graph]:
    """Carga un grafo igraph desde un
        ↪ archivo .pkl."""
    logging.info(f"Cargando el grafo desde
        ↪ '{grafo_path}'...")
    start_time = time.time()
    try:
        with open(grafo_path, 'rb') as f:
            g = pickle.load(f)
        end_time = time.time()
        logging.info(f"Grafo cargado en {
            ↪ end_time - start_time:.2f}
            ↪ segundos. {g.summary()}")
        return g
    except Exception as e:
```

```
        logging.error(f"Error crítico al cargar el grafo: {e}")
        return None

def haversine_distance(lat1: float, lon1: float, lat2: float, lon2: float) -> float:
    """Calcula la distancia en kilómetros entre dos puntos geográficos."""
    R = 6371 # Radio de la Tierra en km
    dlat, dlon = radians(lat2 - lat1), radians(lon2 - lon1)
    a = sin(dlat / 2)**2 + cos(radians(lat1)) * cos(radians(lat2)) * sin(dlon / 2)**2
    return R * 2 * atan2(sqrt(a), sqrt(1 - a))

def preparar_pesos_geograficos(graph: ig.Graph, force_recalc: bool = False) -> None:
    """Asegura que cada arista tenga un atributo 'weight' con su distancia haversine."""
    if 'weight' in graph.es.attributes() and not force_recalc:
        logging.info("El atributo 'weight' ya existe en las aristas. No se recalculará.")
        return

    logging.info("Calculando pesos geográficos (distancia Haversine) para cada arista...")
    pesos = []
    for edge in tqdm(graph.es, desc="Calculando pesos"):
        source_v = graph.vs[edge.source]
        target_v = graph.vs[edge.target]
        dist = haversine_distance(source_v['lat'], source_v['lon'], target_v['lat'], target_v['lon'])
        pesos.append(dist)
    graph.es['weight'] = pesos
    logging.info("Pesos geográficos calculados y asignados a las aristas.")


def encontrar_nodos_mas_distantes_aprox(graph: ig.Graph) -> Optional[Tuple[int, int]]:
    """Encuentra un par aproximado de los nodos más distantes usando la caja delimitadora."""
    logging.info("Buscando el par de nodos geográficamente más distantes (aproximación)...")
    extremos: Dict[str, Tuple[float, Any]] = {
        'min_lat': (float('inf'), None), 'max_lat': (float('-inf'), None),
        'min_lon': (float('inf'), None), 'max_lon': (float('-inf'), None)
    }

    for v in graph.vs:
        if v['lat'] is None or v['lon'] is None: continue
        lat, lon = v['lat'], v['lon']
        if lat < extremos['min_lat'][0]: extremos['min_lat'] = (lat, v.index)
        if lat > extremos['max_lat'][0]: extremos['max_lat'] = (lat, v.index)
        if lon < extremos['min_lon'][0]: extremos['min_lon'] = (lon, v.index)
        if lon > extremos['max_lon'][0]: extremos['max_lon'] = (lon, v.index)

    nodos_candidatos_ids = {node_id for _, node_id in extremos.values() if node_id is not None}
    if len(nodos_candidatos_ids) < 2:
        logging.error("No se encontraron suficientes nodos con coordenadas para determinar un rango.")
        return None

    max_dist, par_mas_distante = -1.0, None
    for u_id, v_id in combinations(nodos_candidatos_ids, 2):
        dist = haversine_distance(graph.vs[u_id]['lat'], graph.vs[u_id]['lon'], graph.vs[v_id]['lat'], graph.vs[v_id]['lon'])
        if dist > max_dist:
            max_dist, par_mas_distante = dist, (u_id, v_id)

    if par_mas_distante:
        logging.info(f"Par más distante encontrado: Nodos {par_mas_distante} (Distancia: {max_dist:.2f} km)")
    return par_mas_distante

# --- FUNCIÓN DE DIJKSTRA REIMPLEMENTADA ---
def encontrar_camino_mas_corto_dijkstra(graph: ig.Graph, source: int, sink: int) -> Tuple[Optional[List[int]], Optional[float], float]:
    logging.info(f"Iniciando algoritmo de Dijkstra (implementación manual) de nodo {source} a {sink}.")
    start_time = time.time()

    distances = {v.index: float('inf') for v in graph.vs}
    previous_nodes = {v.index: None for v in graph.vs}
    distances[source] = 0
    priority_queue = [(0, source)]
    try:
        while priority_queue:
            current_distance, current_node_id = heapq.heappop(priority_queue)

            if current_node_id == sink:
                logging.info("Destino alcanzado. Reconstruyendo el camino.")
                break

            if current_distance > distances[current_node_id]:
                continue

            for neighbor_id in graph.neighbors(current_node_id, mode='out'):
                edge_id = graph.get_eid(current_node_id, neighbor_id)
```

```
            weight = graph.es[edge_id]['weight']
            distance_through_current = distances[current_node_id] + weight

            if distance_through_current < distances[neighbor_id]:
                distances[neighbor_id] = distance_through_current
                previous_nodes[neighbor_id] = current_node_id
                heapq.heappush(priority_queue, (distance_through_current, neighbor_id))

    path = []
    if distances[sink] == float('inf'):
        tiempo_total = time.time() - start_time
        logging.warning(f"No se encontró un camino entre {source} y {sink}.")
        return None, None, tiempo_total

    current = sink
    while current is not None:
        path.append(current)
        current = previous_nodes[current]

    path.reverse()
    coste_total = distances[sink]
    tiempo_total = time.time() - start_time
    logging.info(f"Dijkstra (manual) completado en {tiempo_total:.4f} segundos.")

    if path[0] != source:
        logging.error("Error en la reconstrucción del camino.")
        return None, None, tiempo_total

    return path, coste_total, tiempo_total

  except Exception as e:
    logging.error(f"Error durante la ejecución de Dijkstra (manual): {e}", exc_info=True)
    tiempo_total = time.time() - start_time
    return None, None, tiempo_total


# --- Funciones de Visualización con Folium ---

def crear_mapa_camino_corto(graph: ig.Graph, camino: List[int], coste: float, source: int, sink: int) -> Optional[folium.Map]:
  """Crea un mapa base con el camino más corto encontrado."""
  logging.info("Creando mapa base con el camino más corto...")
  try:
    start_coords = (graph.vs[source]['lat'], graph.vs[source]['lon'])
    m = folium.Map(location=start_coords, zoom_start=6, tiles="CartoDB positron")

    icon_source = BeautifyIcon(icon='play', border_color='#2ca02c', text_color='#2ca02c', icon_shape='circle')
    icon_sink = BeautifyIcon(icon='stop', border_color='#d62728', text_color='#d62728', icon_shape='circle')
    folium.Marker(location=start_coords, popup=f"SOURCE: Nodo {source}", tooltip="Punto de Origen", icon=icon_source).add_to(m)
    folium.Marker(location=(graph.vs[sink]['lat'], graph.vs[sink]['lon']), popup=f"SINK: Nodo {sink}", tooltip="Punto de Destino", icon=icon_sink).add_to(m)

    path_group = folium.FeatureGroup(name='Camino Más Corto (Dijkstra)', show=True)
    puntos = [(graph.vs[nid]['lat'], graph.vs[nid]['lon']) for nid in camino if graph.vs[nid]['lat'] is not None]

    folium.PolyLine(
      puntos,
      color='#1f77b4',
      weight=5,
      opacity=0.9,
      tooltip=f"Camino más corto: {coste:.2f} km"
    ).add_to(path_group)

    path_group.add_to(m)
    return m
  except Exception as e:
    logging.error(f"No se pudo crear el mapa base. Error: {e}")
    return None

### MODIFICADO ###
def agregar_caminos_random_al_mapa(m: folium.Map, graph: ig.Graph, num_caminos: int) -> List[Dict[str, Any]]:
  """Genera caminos aleatorios (usando Dijkstra de igraph) y los agrega al mapa para contexto. Retorna los caminos generados."""

  logging.info(f"Generando {num_caminos} caminos aleatorios para dar contexto a la red...")
  caminos_generados = []
  max_node_id = graph.vcount() - 1
  iterator = tqdm(range(num_caminos), desc="Generando caminos aleatorios")

  for _ in iterator:
    for _ in range(10): # Intentos para encontrar un camino válido
      u, v = random.randint(0, max_node_id), random.randint(0, max_node_id)
      if u == v or graph.vs[u]['lat'] is None or graph.vs[v]['lat'] is None: continue

      camino_nodos = graph.get_shortest_paths(u, to=v, weights='weight', output='vpath')

      if camino_nodos and camino_nodos[0]:
```

```
            camino = camino_nodos[0]
            if all(graph.vs[nid]['lat'] is not None for nid in camino):
                coste = sum(graph.es[graph.get_eid(camino[i], camino[i+1])]['weight'] for i in range(len(camino)-1))
                if coste > 0:
                    caminos_generados.append({'path': camino, 'length_km': coste})
                    break

    if not caminos_generados:
        logging.warning("No se pudo generar ningún camino aleatorio válido con coordenadas completas.")
        return []

    camino_mas_corto_muestra = min(caminos_generados, key=lambda x: x['length_km'])
    camino_mas_largo_muestra = max(caminos_generados, key=lambda x: x['length_km'])

    random_group = folium.FeatureGroup(name=f'{num_caminos} Caminos Aleatorios (Contexto)', show=False)

    for camino_info in caminos_generados:
        puntos = [(graph.vs[nid]['lat'], graph.vs[nid]['lon']) for nid in camino_info['path']]
        if camino_info == camino_mas_corto_muestra:
            color, weight, opacity, tooltip = '#d62728', 4, 1.0, f"Más corto de la muestra ({camino_info['length_km']:.2f} km)"
        elif camino_info == camino_mas_largo_muestra:
            color, weight, opacity, tooltip = '#2ca02c', 4, 1.0, f"Más largo de la muestra ({camino_info['length_km']:.2f} km)"
        else:
            color, weight, opacity, tooltip = '#555555', 1, 0.5, f"Camino de {camino_info['length_km']:.2f} km"

        folium.PolyLine(puntos, color=color, weight=weight, opacity=opacity, tooltip=tooltip).add_to(random_group)

    random_group.add_to(m)
    return caminos_generados

def agregar_capa_nodos_de_caminos(m: folium.Map, graph: ig.Graph, camino_principal: List[int], caminos_random: List[Dict[str, Any]]):
    """Agrega una capa al mapa con todos los nodos que pertenecen a los caminos visualizados."""

    logging.info("Agregando capa de nodos que pertenecen a las rutas calculadas...")

    nodos_en_rutas: Set[int] = set(camino_principal)
    for camino_info in caminos_random:
        nodos_en_rutas.update(camino_info['path'])

    logging.info(f"Se visualizarán {len(nodos_en_rutas)} nodos únicos en la nueva capa.")

    nodos_group = folium.FeatureGroup(name='Nodos en Rutas', show=False)

    for nodo_id in nodos_en_rutas:
        v = graph.vs[nodo_id]
        if v['lat'] is not None and v['lon'] is not None:
            folium.CircleMarker(
                location=[v['lat'], v['lon']],
                radius=3,
                color='#800080', # Púrpura
                fill=True,
                fill_color='#800080',
                fill_opacity=0.7,
                tooltip=f"Nodo {v.index}"
            ).add_to(nodos_group)

    nodos_group.add_to(m)

# --- Bloque Principal de Ejecución ---
if __name__ == "__main__":
    GRAFO_PKL_ENTRADA = 'grafo_igraph_paralelizado.pkl'
    MAPA_HTML_SALIDA = 'analisis_de_red_dijkstra.html'
    # Nota: El código original calcula 1000 caminos, no 100.
    NUM_CAMINOS_RANDOM = 100

    mi_grafo = cargar_grafo(GRAFO_PKL_ENTRADA)

    if mi_grafo:
        preparar_pesos_geograficos(mi_grafo, force_recalc=False)
        par_distante = encontrar_nodos_mas_distantes_aprox(mi_grafo)

        if par_distante:
            SOURCE_NODE, SINK_NODE = par_distante

            camino_optimo, coste_total, tiempo_dijkstra = encontrar_camino_mas_corto_dijkstra(mi_grafo, SOURCE_NODE, SINK_NODE)

            if camino_optimo and coste_total is not None:
                mapa_resultado = crear_mapa_camino_corto(mi_grafo, camino_optimo, coste_total, SOURCE_NODE, SINK_NODE)

                if mapa_resultado:
                    caminos_aleatorios_generados = agregar_caminos_random_al_mapa(mapa_resultado, mi_grafo, NUM_CAMINOS_RANDOM)
```

```
                promedio_distancia_random = 0.0
                num_caminos_generados = len(caminos_aleatorios_generad)

                if num_caminos_generados > 0:
                    distancias_random = [camino['length_km'] for camino in caminos_aleatorios_gene]
                    promedio_distancia_random = sum(distancias_random) / num_caminos_generados

                agregar_capa_nodos_de_caminos(mapa_resultado, mi_grafo, camino_optimo, caminos_aleatorios_generad)

                folium.LayerControl().add_to(mapa_resultado)
                mapa_resultado.save(MAPA_HTML_SALIDA)

                print("\n" + "="*60)
                print(" ANÁLISIS DE RUTA Ó PTIMA (DIJKSTRA) COMPLETADO")
                print("="*60)
                print("Algoritmo utilizado: Implementación manual de Dijkstra en Python")
                print(f"Rango de análisis (automático): Nodos {SOURCE_NODE} a {SINK_NODE}")
                print(f"Distancia del Camino Más Corto: {coste_total:.2f} km")
                print(f"Número de saltos (nodos) en el camino: {len(camino_optimo)}")
                print(f"Tiempo de cálculo (Dijkstra): {tiempo_dijkstra:.4f} segundos")

                ### AÑADIDO: MOSTRAR EL RESULTADO DEL PROMEDIO ###
                if promedio_distancia_random > 0:
                    print("-" * 20)
                    print("Análisis de la muestra de caminos aleatorios:")
                    print(f"Distancia promedio de los {num_caminos_generados} caminos de muestra: {promedio_distancia_random:.2f} km")
                ### FIN DEL CÓDIGO AÑADIDO ###

                    print(f"\nSe ha generado un mapa interactivo en: '{MAPA_HTML_SALIDA}'")
                    print("El mapa contiene las siguientes capas (usa el control de capas):")
                    print(" - [Visible] Camino Más Corto: La ruta óptima encontrada.")
                    print(" - [Oculta] Caminos Aleatorios: Una muestra para contexto de la red.")
                    print(" -> El más largo (distancia) de la muestra se muestra en VERDE.")
                    print(" -> El más corto (distancia) de la muestra se muestra en ROJO.")
                    print(" - [Oculta] Nodos en Rutas: Muestra todos los nodos únicos que forman parte de los caminos visualizados.")
                    print("="*60)
                else:
                    logging.error("No se pudo crear el objeto de mapa base. Abortando visualización.")
            else:
                logging.error(f"No se encontró un camino entre el nodo {SOURCE_NODE} y el {SINK_NODE}.")
        else:
            logging.error("No se pudo determinar el par de nodos de origen/destino. Abortando análisis.")
```

Listing 4. Codigo para el analisis Dijkstra

import pickle, time, logging, random, math, itertools, igraph, folium, heapq, tqdm: Importa todas las librerías requeridas para: carga de grafos, cálculo de distancias, algoritmos de caminos mínimos, manejo de datos geoespaciales, generación de mapas interactivos y registro de actividades.

logging.basicConfig(...): Inicializa la configuración de logging para guardar todas las actividades tanto en consola como en el archivo de registro `analisis_de_red.log`.

cargar_grafo(grafo_path): Carga un grafo de igraph desde un archivo `.pkl`. Mide y registra el tiempo de carga. Si falla, captura la excepción y la registra.

haversine_distance(...): Calcula la distancia en kilómetros entre dos pares de coordenadas (lat/lon) utilizando la fórmula de Haversine para obtener una distancia geográfica realista en la superficie terrestre.

preparar_pesos_geograficos(graph): Asegura que todas las aristas tengan un peso correspondiente a la distancia Haversine calculada. Si los pesos ya existen y no se fuerza su recalculo, no realiza ninguna operación. Si no, recorre todas las aristas

para asignarles la distancia correspondiente.
encontrar_nodos_mas_distantes_aprox(graph): Identifica un par de nodos cuya posición geográfica (usando coordenadas lat/lon) representa una de las distancias máximas aproximadas en el grafo. Utiliza para ello:

- Identificación de extremos mínimos y máximos para latitudes y longitudes.
- Cálculo de la distancia Haversine para estos extremos.

Retorna el par de nodos más distantes junto con la distancia correspondiente.
encontrar_camino_mas_corto_dijkstra(...): Implementa manualmente el algoritmo de Dijkstra para obtener:

- El camino más corto entre dos nodos.
- El costo total asociado al camino.
- El tiempo de ejecución.

Utiliza una cola de prioridad (`heapq`) para garantizar eficiencia en la búsqueda. Si falla, captura y registra la excepción correspondiente.
crear_mapa_camino_corto(...): Construye un mapa de `folium` para representar:

- El camino óptimo encontrado (usando una *FeatureGroup* para organizarlo).
- Marcadores para los nodos de inicio y fin, con íconos distintivos.

Si falla, registra error en el *logging*.
agregar_caminos_random_al_mapa(...): Crea una muestra de caminos aleatorios para obtener una referencia de comparación. Utiliza Dijkstra para obtener caminos al azar y asigna:

- Color rojo para el camino más corto de la muestra.
- Color verde para el camino más largo de la muestra.
- Color gris para caminos intermedios.

Si falla al generar caminos, lo registra en el *logging*. Retorna la colección de caminos generados.
agregar_capa_nodos_de_caminos(...): Recopila todos los nodos pertenecientes al camino óptimo y a los caminos aleatorios para crear una *capa de nodos*. Esta capa permite evaluar la cobertura de la red considerando todas las rutas calculadas.
if __name__ == ¨__main__¨: Orquesta la ejecución de todas las partes:

1. Carga el grafo.
2. Prepara los pesos de las aristas.
3. Identifica los nodos extremos para obtener el camino más largo aproximado.
4. Ejecuta Dijkstra para obtener la ruta óptima.
5. Crea un mapa para representar la ruta óptima.
6. Genera caminos aleatorios para comparación.
7. Añade una capa de nodos para representar todas las rutas calculadas.
8. Exporta el mapa final como HTML junto con todos los elementos requeridos.
9. Imprime detalles de ejecución para análisis posterior.

Si falla en cualquier paso crítico, detiene la ejecución e informa mediante registros en consola y en el *log*.

V-D. Codigo de communityLouvain.py

```python
import pickle
import time
import logging
import random
from collections import Counter, defaultdict
import igraph as ig
from typing import Optional, List, Dict, Tuple
import folium
from folium.plugins import MarkerCluster
import matplotlib
import matplotlib.colors as colors
from tqdm import tqdm

# --- Configuración del Logging ---
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s',
handlers=[logging.FileHandler("analisis_comunidades_louvain_manual.log", mode='w', encoding='utf-8'),
logging.StreamHandler()])

# --- UMBRALES DE VISUALIZACIÓN ---
UMBRAL_MAX_VISUALIZACION_ARISTAS = 3000
UMBRAL_MAX_NODOS_A_DIBUJAR = 3000

# --- Función de Carga de Grafo ---
def cargar_grafo(grafo_path: str) -> Optional[ig.Graph]:
    logging.info(f"Cargando el grafo desde '{grafo_path}'..."); start_time = time.time()
    try:
        with open(grafo_path, 'rb') as f: g = pickle.load(f)
        end_time = time.time(); logging.info(f"Grafo cargado en {end_time - start_time:.2f} segundos. {g.summary()}")
        return g
    except Exception as e:
        logging.error(f"Error crítico al cargar el grafo: {e}"); return None

# --- INICIO DE LA IMPLEMENTACIÓN MANUAL DE LOUVAIN ---

def _calcular_delta_modularity(node: int, target_community: int, node_degree: float,
                               links_to_target_community: float,
                               total_community_degree: float,
                               total_edge_weight: float) -> float:
    if total_edge_weight == 0: return 0
    term1 = links_to_target_community / total_edge_weight
    term2 = (total_community_degree * node_degree) / (2 * (total_edge_weight ** 2))
    return term1 - term2

```

```python
def _louvain_phase1(graph: ig.Graph, total_edge_weight: float) -> Tuple[Dict[int, int], bool]:
    """
    Fase 1: Optimización de modularidad local.
    Devuelve la partición y un booleano que indica si hubo cambios.
    """
    communities = {v.index: v.index for v in graph.vs}

    # 1. Obtenemos la lista de todas las fuerzas de los nodos.
    all_strengths = graph.strength(weights='weight' if 'weight' in graph.es.attributes() else None)
    # 2. Creamos el diccionario mapeando el índice del nodo a su fuerza.
    node_degrees = {i: strength for i, strength in enumerate(all_strengths)}
    # -----------------------

    community_degrees = node_degrees.copy()

    improved = False
    nodes = list(range(graph.vcount()))
    random.shuffle(nodes)

    for node in tqdm(nodes, desc=" Fase 1: Optimizando modularidad"):
        current_community = communities[node]
        best_community = current_community
        max_gain = 0.0

        links_to_communities = defaultdict(float)
        for neighbor in graph.neighbors(node, mode='all'):
            edge_id = graph.get_eid(node, neighbor, directed=False, error=False)
            if edge_id != -1:
                weight = graph.es[edge_id]['weight'] if 'weight' in graph.es.attributes() else 1.0
                links_to_communities[communities[neighbor]] += weight

        for community, links_weight in links_to_communities.items():
            if community == current_community: continue
            gain = _calcular_delta_modularity(node, community, node_degrees[node], links_weight, community_degrees[community], total_edge_weight)
            if gain > max_gain:
                max_gain = gain
                best_community = community

        if max_gain > 0:
            community_degrees[current_community] -= node_degrees[node]
            community_degrees[best_community] += node_degrees[node]
            communities[node] = best_community
            improved = True

    return communities, improved
```

```python
def _louvain_phase2(graph: ig.Graph, communities: Dict[int, int]) -> ig.Graph:
    """
    Fase 2: Agregación de comunidades. Construye un nuevo grafo.
    """
    new_nodes = list(set(communities.values()))
    new_node_map = {old_id: new_id for new_id, old_id in enumerate(new_nodes)}

    new_graph = ig.Graph(directed=False)
    new_graph.add_vertices(len(new_nodes))

    edge_weights = defaultdict(float)

    for edge in graph.es:
        source, target = edge.tuple
        c_source = communities[source]
        c_target = communities[target]
        weight = edge['weight'] if 'weight' in graph.es.attributes() else 1.0

        if c_source != c_target:
            new_source = new_node_map[c_source]
            new_target = new_node_map[c_target]

            # Ordenar para evitar duplicados en grafo no dirigido
            if new_source < new_target:
                edge_weights[(new_source, new_target)] += weight
            else:
                edge_weights[(new_target, new_source)] += weight

    new_graph.add_edges(edge_weights.keys())
    new_graph.es['weight'] = list(edge_weights.values())

    return new_graph

def detectar_comunidades_louvain_manual(graph: ig.Graph) -> Dict[int, List[int]]:
    """
    Implementación manual completa del algoritmo de Louvain.
    """
    logging.info("Iniciando detección de comunidades con implementación MANUAL de Louvain...")
    start_time = time.time()

    # Crucial: El algoritmo de Louvain trabaja sobre grafos no dirigidos.
    # Esta conversión es necesaria y soluciona el `ValueError`.
    if graph.is_directed():
        logging.info("El grafo es dirigido. Convirtiendo a no dirigido para el análisis de comunidades.")
        # Usamos `as_undirected` que es más seguro que modificar el grafo in-place.
        current_graph = graph.as_undirected(mode='collapse', combine_edges=dict(weight="sum"))
    else:
        current_graph = graph.copy()
```

```
    # La membresía inicial mapea cada nodo del grafo ORIGINAL a su comunidad actual.
    membership = {v.index: v.index for v in graph.vs}

    pass_num = 1
    while True:
        logging.info(f"--- Louvain - Pass {pass_num} ---")

        total_edge_weight = sum(current_graph.es['weight']) if 'weight' in current_graph.es.attributes() else current_graph.ecount()
        if total_edge_weight == 0:
            logging.warning("El grafo no tiene aristas. Deteniendo el algoritmo.")
            break

        communities, improved = _louvain_phase1(current_graph, total_edge_weight)

        if not improved:
            logging.info("No hubo más mejoras de modularidad. Convergencia alcanzada.")
            break

        # Mapear la nueva partición a la membresía global.
        # Si el nodo 5 va a la com. 10, y en la siguiente pasada
        # la com. 10 va a la super-com. 2, el nodo 5 debe pertenecer a la super-com. 2.
        partition_map = communities
        for node_idx in range(graph.vcount()):
            membership[node_idx] = partition_map.get(membership[node_idx], membership[node_idx])

        logging.info(" Fase 2: Agregando comunidades para la siguiente pasada...")
        current_graph = _louvain_phase2(current_graph, communities)

        if current_graph.vcount() == 0 or current_graph.ecount() == 0:
            logging.info("El grafo agregado ya no tiene nodos o aristas. Deteniendo.")
            break

        pass_num += 1

    final_comunidades = defaultdict(list)
    for node, comm in membership.items():
        final_comunidades[comm].append(node)

    end_time = time.time()
    logging.info(f"Algoritmo manual de Louvain completado en {end_time - start_time:.2f} s. Se encontraron {len(final_comunidades)} comunidades.")
    return dict(final_comunidades)

# --- Función de Análisis ---
def analizar_y_seleccionar_comunidades(comunidades_dict: Dict[int, List[int]], num_random: int = 20) -> Dict[str, List[int]]:
    logging.info("Analizando y seleccionando comunidades para visualización ...");
    if not comunidades_dict or len(comunidades_dict) < 3: logging.warning("No hay suficientes comunidades para un análisis detallado."); return {}
    comunidades_con_tamaño = [(cid, len(miembros)) for cid, miembros in comunidades_dict.items()]
    id_grande, tamaño_grande = max(comunidades_con_tamaño, key=lambda item: item[1])
    com_mayores_a_uno = [c for c in comunidades_con_tamaño if c[1] > 1]
    id_pequeña, tamaño_pequeña = min(com_mayores_a_uno, key=lambda item: item[1]) if com_mayores_a_uno else min(comunidades_con_tamaño, key=lambda item: item[1])
    ids_extremos = {id_grande, id_pequeña}
    posibles_ids_random = [cid for cid, size in comunidades_con_tamaño if cid not in ids_extremos and 5 < size < UMBRAL_MAX_VISUALIZACION_ARISTAS]
    ids_random = random.sample(posibles_ids_random, min(num_random, len(posibles_ids_random))) if posibles_ids_random else []
    seleccion = {'grande': [id_grande], 'pequena': [id_pequeña], 'random': ids_random}
    logging.info(f"Comunidad más grande (ID {id_grande}): {tamaño_grande} miembros.")
    logging.info(f"Comunidad más pequeña (>1 miembro) (ID {id_pequeña}): {tamaño_pequeña} miembros.")
    logging.info(f"Se seleccionaron {len(ids_random)} comunidades aleatorias (tamaño < {UMBRAL_MAX_VISUALIZACION_ARISTAS}).")
    return seleccion

# --- Función de Colores ---
def crear_mapa_de_colores(tamaño_min: int, tamaño_max: int):
    colormap = matplotlib.colormaps.get_cmap('coolwarm')
    normalizador = colors.LogNorm(vmin=max(1, tamaño_min), vmax=tamaño_max)
    return lambda tamaño: colors.to_hex(colormap(normalizador(tamaño)))

# --- Función de Visualización ---
def visualizar_comunidades(graph: ig.Graph, comunidades_dict: Dict[int, List[int]], seleccion: Dict[str, List[int]], output_filename: str):
    logging.info(f"Creando mapa de visualización en '{output_filename}'...")
    coords_validas = [(v['lat'], v['lon']) for v in graph.vs if v['lat'] is not None and v.index != 0]
    if not coords_validas: logging.error("No hay nodos con coordenadas para
```

```
        ↪ visualizar."); return
    avg_lat = sum(c[0] for c in
        ↪ coords_validas) / len(
        ↪ coords_validas); avg_lon = sum(c
        ↪ [1] for c in coords_validas) /
        ↪ len(coords_validas)
    m = folium.Map(location=[avg_lat,
        ↪ avg_lon], zoom_start=2, tiles="
        ↪ CartoDB positron")
    tamaños = {cid: len(miembros) for cid,
        ↪ miembros in comunidades_dict.
        ↪ items() if miembros}
    if not tamaños: logging.error("Las
        ↪ comunidades están vacías, no se
        ↪ puede generar el mapa."); return
    mapa_color = crear_mapa_de_colores(min(
        ↪ tamaños.values()), max(tamaños.
        ↪ values()))
    ids_a_visualizar = seleccion['grande'] +
        ↪  seleccion['pequena'] + seleccion
        ↪ ['random']
    iterator = tqdm(ids_a_visualizar, desc="
        ↪ Creando capas de comunidades")

    for com_id in iterator:
        miembros_originales =
            ↪ comunidades_dict.get(com_id)
        if not miembros_originales: continue

        tamaño_original = len(
            ↪ miembros_originales)
        color = mapa_color(tamaño_original)
        show_layer = com_id in seleccion['
            ↪ grande'] or com_id in
            ↪ seleccion['pequena']
        nombre_capa = f"Comunidad {com_id} ({
            ↪ tamaño_original} miembros)"
        if com_id in seleccion['grande']:
            ↪ nombre_capa = f"Comunidad Más
            ↪ Grande ({tamaño_original})"
        if com_id in seleccion['pequena']:
            ↪ nombre_capa = f"Comunidad Más
            ↪ Pequeña ({tamaño_original})"

        container = MarkerCluster(name=
            ↪ nombre_capa, show=show_layer)
            ↪ if tamaño_original > 200 else
            ↪ folium.FeatureGroup(name=
            ↪ nombre_capa, show=show_layer)
        container.add_to(m)

        nodos_a_dibujar = miembros_originales
        if tamaño_original >
            ↪ UMBRAL_MAX_NODOS_A_DIBUJAR:
            logging.warning(f"Comunidad {
                ↪ com_id} ({tamaño_original}
                ↪ nodos) excede umbral.
                ↪ Mostrando muestra de {
                ↪ UMBRAL_MAX_NODOS_A_DIBUJAR
                ↪ }.")
            nodos_a_dibujar = random.sample(
                ↪ miembros_originales,
                ↪ UMBRAL_MAX_NODOS_A_DIBUJAR)

        nodos_visibles_con_coords = []
        for nodo_id in nodos_a_dibujar:
            v = graph.vs[nodo_id]
            if v['lat'] is not None and v['lon
                ↪ '] is not None:
                nodos_visibles_con_coords.
                    ↪ append(nodo_id)
                folium.CircleMarker(location=(v
                    ↪ ['lat'], v['lon']),
                    ↪ radius=4, color=color,
                    ↪ fill=True, fill_color=
                    ↪ color, fill_opacity=0.7,
                    ↪  tooltip=f"Nodo {v.index
                    ↪ } (Com. {com_id})").
                    ↪ add_to(container)

        if tamaño_original <=
            ↪ UMBRAL_MAX_VISUALIZACION_ARISTAS
            ↪ :
            if len(nodos_visibles_con_coords)
                ↪ > 1:
                subgrafo_comunidad = graph.
                    ↪ subgraph(
                    ↪ nodos_visibles_con_coords
                    ↪ )
                num_aristas_internas = len(
                    ↪ subgrafo_comunidad.es)
                logging.info(f"Comunidad {
                    ↪ com_id} (tamaño {tamañ
                    ↪ o_original}): Intentando
                    ↪  dibujar {
                    ↪ num_aristas_internas}
                    ↪ aristas internas.")

                for arista in
                    ↪ subgrafo_comunidad.es:
                    id_origen =
                        ↪ nodos_visibles_con_coords
                        ↪ [arista.source];
                        ↪ id_destino =
                        ↪ nodos_visibles_con_coords
                        ↪ [arista.target]
                    v_origen = graph.vs[
                        ↪ id_origen]; v_destino
                        ↪  = graph.vs[
                        ↪ id_destino]
                    folium.PolyLine(locations=[(
                        ↪ v_origen['lat'],
                        ↪ v_origen['lon']), (
                        ↪ v_destino['lat'],
                        ↪ v_destino['lon'])],
                        ↪ color=color, weight
                        ↪ =1.5, opacity=0.5).
                        ↪ add_to(container)
        else:
            logging.warning(f"Omitiendo dibujo
                ↪  de aristas para comunidad
                ↪ {com_id} (tamaño: {tamañ
                ↪ o_original} > {
                ↪ UMBRAL_MAX_VISUALIZACION_ARISTAS
                ↪ }).")

    folium.LayerControl(collapsed=False).
        ↪ add_to(m)
    logging.info("Guardando el mapa en el
        ↪ archivo HTML...")
    m.save(output_filename)
    logging.info(f"Mapa guardado
        ↪ correctamente en '{
        ↪ output_filename}'.")


# --- Bloque Principal ---
if __name__ == "__main__":
    GRAFO_PKL_ENTRADA = '
        ↪ grafo_igraph_paralelizado.pkl'
    MAPA_HTML_SALIDA = '
        ↪ analisis_comunidades_louvain_manual
        ↪ .html'
    NUM_COMUNIDADES_RANDOM = 50

    mi_grafo = cargar_grafo(
        ↪ GRAFO_PKL_ENTRADA)
    if mi_grafo:
        coms_dict =
            ↪ detectar_comunidades_louvain_manual
            ↪ (mi_grafo)
        if coms_dict:
            comunidades_seleccionadas =
                ↪ analizar_y_seleccionar_comunidades
```

```
                ↪ (coms_dict,
                ↪ NUM_COMUNIDADES_RANDOM)
275         if comunidades_seleccionadas:
276             visualizar_comunidades(mi_grafo
                    ↪ , coms_dict,
                    ↪ comunidades_seleccionadas
                    ↪ , MAPA_HTML_SALIDA)
277
278             print("\n" + "="*60)
279             print(" ANÁLISIS DE COMUNIDADES
                    ↪  (LOUVAIN - IMPLEMENTACI
                    ↪ ÓN MANUAL)")
280             print("="*60)
281             print(f"Se encontraron un total
                    ↪  de {len(coms_dict)}
                    ↪ comunidades.")
282             print("\nSe ha generado un mapa
                    ↪  interactivo en:", f"'{
                    ↪ MAPA_HTML_SALIDA}'")
283             print("\nOPTIMIZACIONES DE
                    ↪ VISUALIZACIÓN:")
284             print(f" - Muestreo de Nodos:
                    ↪ Para comunidades con > {
                    ↪ UMBRAL_MAX_NODOS_A_DIBUJAR
                    ↪ } miembros,")
285             print(" solo se muestra una
                    ↪ muestra aleatoria para
                    ↪ no colapsar el navegador
                    ↪ .")
286             print(f" - Omisión de Aristas:
                    ↪ Los caminos solo se
                    ↪ dibujan para comunidades
                    ↪ ")
287             print(f" con <= {
                    ↪ UMBRAL_MAX_VISUALIZACION_ARISTAS
                    ↪ } miembros.")
288             print("\nEl color de cada
                    ↪ comunidad indica su tama
                    ↪ ño original:")
289             print(" - Colores fríos (azul):
                    ↪  Comunidades pequeñas.")
290             print(" - Colores cálidos (rojo
                    ↪ ): Comunidades grandes."
                    ↪ )
291             print("\nPara más detalles,
                    ↪ revisa el archivo de log
                    ↪ : '
                    ↪ analisis_comunidades_louvain_manual
                    ↪ .log'")
292             print("="*60)
```

Listing 5. Codigo para el mapa de comunidades

import pickle, time, logging, random, igraph, folium, tqdm, ...: Importa todas las librerías requeridas para: carga de grafos, algoritmos de detección de comunidades, representación de mapas geográficos, asignación de colores, muestreo de datos, y análisis y registro de actividades.

logging.basicConfig(...): Inicializa la configuración de registro para guardar todas las actividades tanto en consola como en el archivo de registro `analisis_comunidades_louvain_manual.log`.

UMBRAL_MAX_VISUALIZACION_ARISTAS, UMBRAL_MAX_NODOS_A_DIBUJAR: Constantes para controlar la cantidad de aristas y nodos a representar, evitando sobrecarga gráfica en comunidades de gran tamaño.

cargar_grafo(grafo_path): Carga un grafo de igraph desde un archivo `.pkl`. Mide y registra el tiempo requerido para la operación. Si falla, captura la excepción y la registra en el *log*.

_calcular_delta_modularity(...): Calcula la variación de modularidad al cambiar un nodo de comunidad. Utilizado en la fase 1 de Louvain para evaluar si un movimiento aumenta la modularidad.

_louvain_phase1(...): Implementa la fase de optimización local de Louvain:

- Asigna comunidades iniciales (cada nodo en su comunidad).
- Itera sobre cada nodo para evaluar cambios a comunidades vecinas.
- Utiliza `_calcular_delta_modularity()` para evaluar mejoras.
- Retorna comunidades actuales y bandera de si hubo mejoras.

_louvain_phase2(...): Construye un nuevo grafo donde cada comunidad detectada en la fase 1 pasa a representar un nodo. Esto permite continuar la optimización de modularidad de manera jerárquica.

detectar_comunidades_louvain_manual(...): Implementa el algoritmo de Louvain en su totalidad:

- Carga el grafo y lo asegura no dirigido.
- Ejecuta las Fases 1 y 2 de manera iterativa.
- Finaliza cuando no hay mejoras significativas en la modularidad.
- Genera y retorna un diccionario con todas las comunidades detectadas.

analizar_y_seleccionar_comunidades(...): Analiza todas las comunidades para:

- Identificar la comunidad más grande y la más pequeña.
- Elegir un conjunto de comunidades al azar para representar variedad.
- Retornar un diccionario con la selección final para visualización.

crear_mapa_de_colores(...): Construye una escala de color basada en el tamaño de las comunidades:

- Utiliza un mapa de color *coolwarm*.
- Normaliza los tamaños para representar comunidades grandes en colores cálidos y comunidades pequeñas en fríos.

visualizar_comunidades(...): Crea un mapa interactivo de todas las comunidades seleccionadas:

- Centra el mapa en las coordenadas promedio.
- Añade cada comunidad como una capa, utilizando diferentes colores según su tamaño.
- Añade una muestra de nodos para comunidades grandes para mantener la legibilidad.
- Dibuja aristas solo para comunidades por debajo del umbral de tamaño, para no saturar la visualización.
- Exporta el mapa como un archivo HTML.

if __name__ == ¨__main__¨: Orquesta todas las etapas:

1. Carga el grafo.
2. Ejecuta Louvain para detectar comunidades.

3. Analiza comunidades para seleccionar las principales y una muestra aleatoria.
4. Genera un mapa interactivo donde:
   - Se destacan comunidades grandes, pequeñas y al azar.
   - Se adapta la cantidad de nodos y aristas para evitar saturación.
5. Imprime detalles para el usuario al finalizar, junto con la ubicación del mapa generado.

Si falla en cualquier punto crítico, lo informa mediante registros en consola y en el archivo de *log*.

V-E. Codigo de bfs.py

```python
import igraph as ig
import folium
from folium.plugins import BeautifyIcon, AntPath
import pickle
import time
import logging
import os
import webbrowser
import random
from collections import deque

# --- Configuración ---
GRAFO_PICKLE_PATH = 'grafo_igraph_paralelizado.pkl'
MAPA_HTML_OUTPUT = 'mapa_camino_largo_con_arbol.html' # Nuevo nombre de archivo

# Nodo de inicio
NODO_INICIO = 9999427

# Define la cantidad máxima de nodos a explorar.
MAX_NODOS_A_EXPLORAR = 5000

# --- Configuración del Logging ---
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("analisis_bfs.log", mode='w', encoding='utf-8'),
        logging.StreamHandler()
    ]
)

def visualizar_arbol_bfs(graph, start_node_id, max_nodes, output_filename):
    """
    Realiza una búsqueda BFS, resalta el camino más largo con un AntPath rojo
    y muestra el árbol de búsqueda completo con líneas estáticas.
    """
    logging.info(f"Iniciando BFS desde el nodo {start_node_id} con un límite de {max_nodes} nodos.")
    start_time = time.time()

    # 1. Realizar BFS para recolectar nodos, niveles y la estructura del árbol
    queue = deque([(start_node_id, 0)])
    visited = {start_node_id}
    levels = {start_node_id: 0}
    parent_map = {start_node_id: None}
    tree_edges = []

    while queue and len(visited) < max_nodes:
        current_node, current_level = queue.popleft()
        neighbors = graph.neighbors(current_node, mode="out")

        for neighbor in neighbors:
            if neighbor not in visited:
                if len(visited) >= max_nodes:
                    break
                visited.add(neighbor)
                levels[neighbor] = current_level + 1
                parent_map[neighbor] = current_node
                tree_edges.append((current_node, neighbor))
                queue.append((neighbor, current_level + 1))

    logging.info(f"BFS completado. Se exploraron {len(visited)} nodos y {len(tree_edges)} aristas en {time.time() - start_time:.2f} segundos.")

    # 2. Encontrar el nodo más lejano en el árbol BFS
    farthest_node = max(visited, key=lambda node: levels[node])
    max_level = levels[farthest_node]
    logging.info(f"El camino más largo encontrado tiene {max_level} saltos y termina en el nodo {farthest_node}.")

    # 3. Preparar el mapa con Folium
    logging.info("Creando mapa con Folium...")
    lats = [graph.vs[n]['lat'] for n in visited]
    lons = [graph.vs[n]['lon'] for n in visited]
    map_center = [sum(lats) / len(lats), sum(lons) / len(lons)]

    m = folium.Map(location=map_center, zoom_start=10, tiles="CartoDB positron")

    # 4. Crear capas (FeatureGroups) para cada elemento visual
    nodos_explorados_group = folium.FeatureGroup(name=f"Nodos Explorados ({len(visited)})", show=True)
    aristas_estaticas_group = folium.FeatureGroup(name=f"Árbol BFS Estático ({len(tree_edges)} aristas)", show=True)
    camino_animado_group = folium.FeatureGroup(name="Camino Más Largo (Animado)", show=True)

    # 5. Añadir los nodos explorados al mapa
    colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#9467bd', '#8c564b', '#e377c2']
    for node_id in visited:
        lat, lon = graph.vs[node_id]['lat'], graph.vs[node_id]['lon']
```

```
        level = levels[node_id]
        popup_text = f"<b>Nodo:</b> {node_id}<br><b>Nivel:</b> {level}"

        folium.CircleMarker(
            location=[lat, lon], radius=3,
            color=colors[level % len(colors)],
            fill=True, fill_color=colors[level % len(colors)],
            fill_opacity=0.6, popup=popup_text
        ).add_to(nodos_explorados_group)

    # --- NUEVO: 6. Añadir las aristas estáticas del árbol BFS completo ---
    for parent, child in tree_edges:
        parent_coords = (graph.vs[parent]['lat'], graph.vs[parent]['lon'])
        child_coords = (graph.vs[child]['lat'], graph.vs[child]['lon'])

        folium.PolyLine(
            locations=[parent_coords, child_coords],
            color='#AAAAAA', # Un color gris claro para que no domine
            weight=1, # Línea delgada
            opacity=0.7
        ).add_to(aristas_estaticas_group)

    # 7. Reconstruir y añadir el camino más largo (rojo y animado)
    longest_path_coords = []
    curr = farthest_node
    while curr is not None:
        coords = (graph.vs[curr]['lat'], graph.vs[curr]['lon'])
        longest_path_coords.append(coords)
        curr = parent_map.get(curr)
    longest_path_coords.reverse()

    AntPath(
        locations=longest_path_coords,
        delay=1000, weight=5, color='#d62728', # Rojo y grueso
        pulse_color='#FFFFFF', dash_array=[25, 40]
    ).add_to(camino_animado_group)

    # 8. Añadir marcadores de INICIO y FIN sobre todas las capas
    start_lat, start_lon = graph.vs[start_node_id]['lat'], graph.vs[start_node_id]['lon']
    folium.Marker(
        location=[start_lat, start_lon],
        popup=f"<b>INICIO:</b> {start_node_id}",
        tooltip=f"INICIO: Nodo {start_node_id}",
        icon=BeautifyIcon(icon='play', icon_shape='circle', border_color='#2ca02c', text_color='#2ca02c', background_color='#FFF')
    ).add_to(m)

    end_lat, end_lon = graph.vs[farthest_node]['lat'], graph.vs[farthest_node]['lon']
    folium.Marker(
        location=[end_lat, end_lon], popup=f"<b>FIN:</b> {farthest_node}<br><b>Nivel:</b> {max_level}",
        tooltip=f"FIN: Nodo {farthest_node}",
        icon=BeautifyIcon(icon='stop', icon_shape='circle', border_color='#d62728', text_color='#d62728', background_color='#FFF')
    ).add_to(m)

    # 9. Añadir todas las capas y el control al mapa
    nodos_explorados_group.add_to(m)
    aristas_estaticas_group.add_to(m) # Añadimos la capa de aristas estáticas
    camino_animado_group.add_to(m)
    folium.LayerControl().add_to(m)

    # 10. Guardar y abrir el mapa
    m.save(output_filename)
    logging.info(f"Mapa guardado en '{output_filename}'.")
    webbrowser.open('file://' + os.path.realpath(output_filename))

if __name__ == "__main__":
    if not os.path.exists(GRAFO_PICKLE_PATH):
        logging.error(f"El archivo del grafo '{GRAFO_PICKLE_PATH}' no fue encontrado.")
    else:
        logging.info(f"Cargando el grafo desde '{GRAFO_PICKLE_PATH}'...")
        start_load_time = time.time()
        with open(GRAFO_PICKLE_PATH, 'rb') as f:
            g = pickle.load(f)
        logging.info(f"Grafo cargado en {time.time() - start_load_time:.2f} segundos.")

        if not (0 <= NODO_INICIO < g.vcount()):
            logging.error(f"El NODO_INICIO ({NODO_INICIO}) está fuera del rango.")
            NODO_INICIO = random.randint(0, g.vcount() - 1)
            logging.warning(f"Se ha seleccionado un nuevo nodo de inicio aleatorio: {NODO_INICIO}")

        visualizar_arbol_bfs(g, NODO_INICIO, MAX_NODOS_A_EXPLORAR, MAPA_HTML_OUTPUT)
```

Listing 6. Codigo para el bfs

import igraph as ig: Importa la biblioteca de análisis de grafos para representar y explorar redes grandes de nodos y aristas.
import folium: Importa herramientas para crear mapas interactivos donde representar nodos y caminos.
from folium.plugins import BeautifyIcon, AntPath: Importa herramientas para crear iconos estéticamente distintivos y representar caminos animados en mapas de Folium.
import pickle, time, logging, os, webbrowser, random: Importa herramientas para guardar/cargar grafos, medir duración de cálculos, crear registros de ejecución, verificar existencia de archivos, abrir mapas en el navegador y seleccionar elementos al azar.

from collections import deque: Importa una cola de dos extremos para realizar recorridos BFS de manera eficaz.
GRAFO_PICKLE_PATH, MAPA_HTML_OUTPUT, NODO_INICIO, MAX_NODOS_A_EXPLORAR: Definen la ruta al grafo serializado, la ruta de salida para guardar el mapa, el nodo de inicio para la búsqueda BFS y la cantidad máxima de nodos a visitar.
logging.basicConfig(...): Configura el registro para guardar mensajes tanto en consola como en un archivo de texto, garantizando seguimiento y diagnóstico del análisis BFS.
visualizar_arbol_bfs(...): Realiza una búsqueda BFS en un grafo para:

- Explorar nodos y construir un mapa de parent_map para representar el camino alcanzado.
- Identificar el nodo más lejano alcanzado para representar el camino más largo en la búsqueda.
- Crear un mapa interactivo donde representar:
  - Nodos alcanzados (con color según nivel BFS).
  - Aristas del árbol BFS para representar toda la expansión.
  - El camino más largo alcanzado marcado con una línea animada (AntPath).
- Añadir marcadores de inicio y fin para facilitar la comprensión de la visualización.
- Exportar y abrir el mapa en el navegador por defecto.

if __name__ == ¨__main__¨: Carga el grafo desde un archivo pickle y ejecuta la visualización BFS para un nodo de inicio específico:

- Verifica si el nodo de inicio es válido, de lo contrario escoge uno al azar.
- Llama a `visualizar_arbol_bfs` para realizar la búsqueda, crear el mapa y guardar el resultado en disco.
- Abre el mapa en el navegador para inspección.

VI. Pruebas

VI-A. *Analisis de Comunidad*

En las pruebas realizadas se mostro que la comuniad mas grane esta conformada por 8997594 usuarios y la mas pequeña es de 2 usuarios.

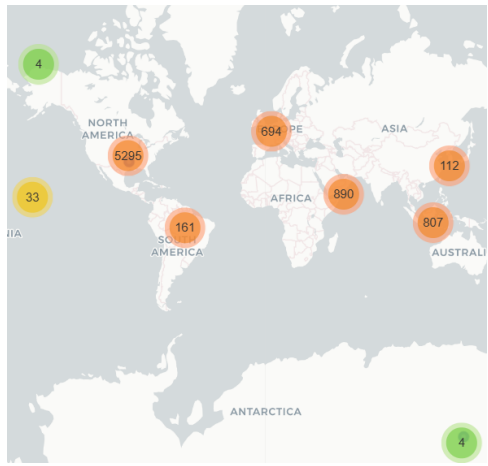

Fig 1. Muestra de la prueba de comunidades

VI-B. *prueba de Dijkstra*

En la prueba de Dijkstra se muestra que el camino mas corto es entre el nodo 674 como origen hasta el nodo 9999427 como llegada.

Fig 2. Prueba de dijsktra

VI-C. *Analisis dfs*

En el contexto de este programa, un algoritmo de búsqueda en profundidad (*Depth-First Search*, DFS) podría aplicarse para explorar la estructura interna del grafo construido, permitiendo identificar componentes conexos o evaluar la alcanzabilidad entre nodos. Aunque no implementado de manera explícita en este flujo de trabajo, el DFS es una técnica clave para análisis topológicos en redes masivas, complementando otras estrategias de análisis y facilitando la comprensión de la conectividad y la distribución de comunidades dentro del grafo procesado.

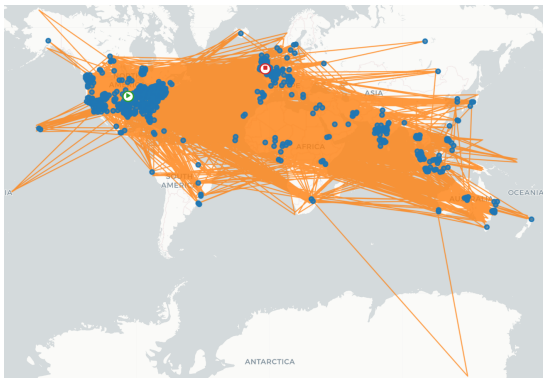

Fig 3. Mapa dfs multicapa.

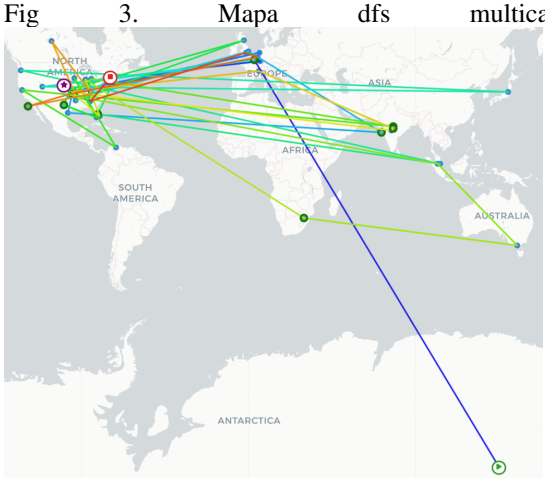

Fig 4. Mapa dfs con marcadores de capa.

VI-D. Visualización de BFS en la Red

La Fig 5 presenta el resultado de una búsqueda en anchura (BFS, *Breadth-First Search*) ejecutada sobre el grafo de comunidades analizado. El nodo de origen, marcado con un ícono de *play* en color verde, representa el punto de partida de la búsqueda, mientras que el nodo de destino, marcado con un ícono de *stop* en color rojo, señala un nodo alcanzado en la frontera de la expansión.

Los enlaces en color gris corresponden a las aristas recorridas durante la búsqueda, alcanzando todos los nodos alcanzables desde el origen inicial. El algoritmo BFS garantiza que cada nodo alcanzado lo hace con el menor número de saltos posible, revelando la conectividad interna de la red.

Esta representación gráfica permite evaluar la dispersión geográfica de comunidades detectadas por el algoritmo de Louvain, facilitando el análisis de alcanzabilidad de cada componente y brindando una visión clara de la estructura de la red.

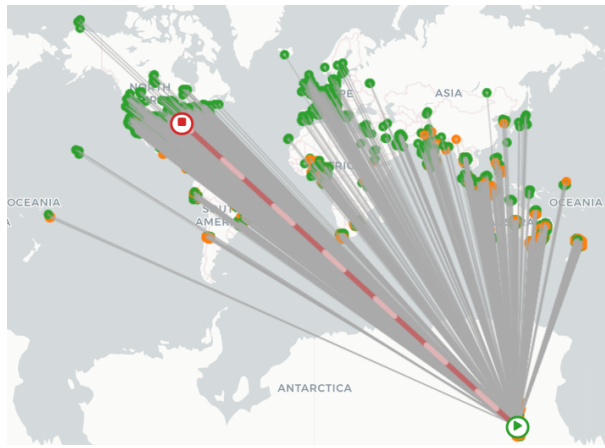

Fig 5. Mapa bfs.

VII. Conclusión

En este trabajo se implementó un flujo de procesamiento y análisis de grafos masivos a través de *igraph*, abordando de manera efectiva la creación de una estructura de datos para representar una red de hasta diez millones de nodos y sus respectivas conexiones. Se implementaron estrategias de paralelización para procesar las relaciones de usuarios de manera escalable y eficaz, garantizando que el tiempo de ejecución sea práctico para datos de gran magnitud.

El sistema desarrollado carga primero las ubicaciones geográficas de cada nodo, garantizando la consistencia de datos al verificar que las coordenadas sean válidas. Luego, procesa de manera paralela las conexiones para crear las aristas del grafo, alcanzando una representación compacta y eficaz en memoria. El resultado final, guardado en formato `pickle`, ofrece una base sólida para análisis de redes a gran escala, facilitando futuras etapas de investigación como detección de comunidades, análisis de caminos críticos o simulación de propagación de información.

Los principios implementados en esta solución —validación de datos de entrada, procesamiento concurrente y liberación explícita de memoria para evitar saturación de recursos— representan un modelo práctico para la creación de grafos masivos en entornos de investigación e industria, donde la eficiencia y la escalabilidad son criterios clave para garantizar la integridad y relevancia de los análisis realizados.