“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 Rorigo Sebastian

Flores Leon Miguel Angel

DOCENTE:

Luque Mamani Edson Fracisco

Arequipa, Perú

22 de junio de 2025

Índice

I Introducción **1**

II Librerias necesarias **2**
- II-A random . . . 2
- II-B igraph . . . 2
- II-C time . . . 2
- II-D logging . . . 2
- II-E fileinput . . . 2
- II-F pickle . . . 2
- II-G gc (Garbage Collector) . . . 2
- II-H multiprocessing . . . 2
- II-I itertools . . . 2
- II-J folium . . . 2
- II-K math . . . 2
- II-L typing . . . 3
- II-M tqdm . . . 3
- II-N matplotlib . . . 3

III Creación del grafo **3**

IV Explicación del Código **3**
- IV-A Codigo de createdata.py . . . 3

V Explicación del Código **4**
- V-A Codigo de grapy paralel.py . . . 4
- V-B Codigo de dfs.py . . . 6
- V-C Codigo de dijkstra.py . . . 9
- V-D Codigo de community.py . . . 13

VI Pruebas **17**
- VI-A Analisis de Comunidad . . . 17
- VI-B prueba de Dijkstra . . . 17
- VI-C Analisis dfs . . . 17

VIIConclusión **18**

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. 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-E. 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-F. 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-G. 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-H. 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-I. 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-J. 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-K. 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-L. *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-M. *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-N. *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.

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

# 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 número 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

```python
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

# --- 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()])

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 FileNotFoundError:
        logging.error(f"Error crítico: El archivo de grafo '{grafo_path}' no fue encontrado.")
        return None
    except Exception as e:
        logging.error(f"Error crítico al cargar el grafo: {e}")
        return None

# --- ALGORITMO DFS (Sin cambios) ---
def dfs_exploracion_desde_un_punto(graph: ig.Graph, source: int) -> Tuple[List[int], float]:
    logging.info(f"Iniciando travesía DFS completa desde el nodo {source}. ")
    start_time = time.time()
    stack = [source]
    visitados = {source}
    nodos_explorados_en_orden = []
    while stack:
        current_node = stack.pop()
        nodos_explorados_en_orden.append(current_node)
        for neighbor in reversed(graph.neighbors(current_node, mode='out')):
            if neighbor not in visitados:
                visitados.add(neighbor)
                stack.append(neighbor)
    end_time = time.time()
    logging.info(f"Travesía DFS completada en {end_time - start_time:.4f} segundos. Total de nodos visitados: {len(nodos_explorados_en_orden)}.")
    return nodos_explorados_en_orden, end_time - start_time

# --- Función de Visualización (CORREGIDA para una selección de muestra consistente) ---
def crear_mapa_multicapa_dfs(
    graph: ig.Graph,
    source_node: int,
    nodos_visitados: List[int],
    max_puntos_linea: int = 5000,
    num_nodos_muestra: int = 1000
) -> Optional[folium.Map]:
    """
    Crea un mapa con dos capas consistentes:
    1. El camino completo de DFS (submuestreado).
    2. Una muestra aleatoria de nodos tomados DIRECTAMENTE del camino visualizado.
    """
    if not nodos_visitados:
```

```
        logging.warning("No hay nodos visitados para mostrar en el mapa.")
        return None

    logging.info("Creando mapa multicapa con visualización consistente.")
    try:
        start_coords = (graph.vs[source_node]['lat'], graph.vs[source_node]['lon'])
        m = folium.Map(location=start_coords, zoom_start=4, tiles="CartoDB positron")

        # --- 1. Definir el camino visual (submuestreado) ---
        # Esta lista de nodos será la FUENTE DE VERDAD para ambas capas.
        if len(nodos_visitados) > max_puntos_linea:
            step = len(nodos_visitados) // max_puntos_linea
            camino_visualizado = nodos_visitados[::step]
            if camino_visualizado[-1] != nodos_visitados[-1]:
                camino_visualizado.append(nodos_visitados[-1])
        else:
            camino_visualizado = nodos_visitados

        # --- CAPA 1: Camino Completo (Línea Naranja) ---
        fg_camino = folium.FeatureGroup(name="Camino DFS (visual)", show=True)
        puntos_del_camino = [(graph.vs[n]['lat'], graph.vs[n]['lon']) for n in camino_visualizado if graph.vs[n]['lat'] is not None]
        folium.PolyLine(
            puntos_del_camino, color="#ff7f0e", weight=2, opacity=0.8,
            tooltip=f"Camino DFS ({len(camino_visualizado)} de {len(nodos_visitados)} nodos)"
        ).add_to(fg_camino)
        m.add_child(fg_camino)

        # --- CAPA 2: Muestra de Nodos (Puntos Azules) ---
        # CORRECCIÓN CLAVE: La muestra se toma de 'camino_visualizado', no de 'nodos_visitados'.
        fg_puntos = folium.FeatureGroup(name=f"Muestra de Nodos (del camino visual)", show=True)
        end_node_id = nodos_visitados[-1]

        # Candidatos para la muestra son los nodos del camino visual, excluyendo inicio/fin.
        candidatos = [n for n in camino_visualizado if n != source_node and n != end_node_id]

        # Aseguramos no pedir más muestras de las disponibles.
        num_muestras_a_tomar = min(len(candidatos), num_nodos_muestra - 2)
        nodos_a_mostrar_aleatorios = random.sample(candidatos, num_muestras_a_tomar) if num_muestras_a_tomar > 0 else []

        # La muestra final siempre incluye inicio y fin, más los aleatorios.
        nodos_muestra_final = [source_node, end_node_id] + nodos_a_mostrar_aleatorios

        folium.CircleMarker( location=coords, radius=3.5, color=#1f77b4', fill=True, fill_color=#1f77b4', fill_opacity=0.7, popup=f"Nodo de muestra: {node_id}"
            ).add_to(fg_puntos)
        m.add_child(fg_puntos)

        # --- MARCADORES PERMANENTES (Inicio y Fin Reales) ---
folium.Marker( location=start_coords, pop-up=f"INICIO DFS: Nodo {source_node}",
            tooltip="Punto de Inicio", icon=BeautifyIcon(icon='play', border_color='#2ca02c', text_color='#2ca02c', z_index_offset=1000
        ).add_to(m)

        end_real_coords = (graph.vs[end_node_id]['lat'], graph.vs[end_node_id]['lon'])
folium.Marker( location=end_real_coords, pop-up=f"FIN REAL DE LA TRAVESA: Nodo {end_node_id}",
            tooltip="Fin de la Travesía", icon=BeautifyIcon(icon='stop', border_color='#d62728', text_color='#d62728'), z_index_offset=1000
        ).add_to(m)

        # --- CONTROL DE CAPAS ---
        folium.LayerControl().add_to(m)

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

# --- Bloque Principal de Ejecución ---
if __name__ == "__main__":
    GRAFO_PKL_ENTRADA = 'grafo_igraph_paralelizado.pkl'
    MAPA_HTML_SALIDA = 'mapa_dfs_multicapa_corregido.html'

    mi_grafo = cargar_grafo(GRAFO_PKL_ENTRADA)

    if mi_grafo:
        try:
            SOURCE_NODE = random.randrange(mi_grafo.vcount())
            logging.info(f"Nodo de inicio de la travesía DFS seleccionado al azar: {SOURCE_NODE}")
```

```
        except ValueError:
            logging.error("El grafo está vacío y no se puede seleccionar un nodo de inicio.")
            exit()

        nodos_visitados_dfs, tiempo_dfs = dfs_exploracion_desde_un_punto(mi_grafo, SOURCE_NODE)

        if nodos_visitados_dfs:
            mapa = crear_mapa_multicapa_dfs(mi_grafo, SOURCE_NODE, nodos_visitados_dfs)

            if mapa:
                mapa.save(MAPA_HTML_SALIDA)

                print("\n" + "="*55)
                print("      ANÁLISIS DE TRAVESÍA DFS CON MÚLTIPLES CAPAS")
                print("="*55)
                print(f"Travesía iniciada desde el nodo (aleatorio): {SOURCE_NODE}")
                print(f"Tiempo total de la travesía: {tiempo_dfs:.4f} segundos.")
                print(f"Total de nodos visitados en el componente: {len(nodos_visitados_dfs)}")

                print(f"\nMapa interactivo guardado en: '{MAPA_HTML_SALIDA}'")
                print(" -> El mapa contiene DOS capas consistentes que puedes activar/desactivar:")
                print("    1. 'Camino DFS': La línea que muestra la forma de la travesía.")
                print("    2. 'Muestra de Nodos': Puntos aleatorios tomados DEL CAMINO VISIBLE.")
                print("="*55)
            else:
                logging.error("No se pudo crear el objeto de mapa base.")
        else:
            logging.info("La travesía DFS no visitó ningún nodo.")
```

Listing 3. Codigo para el mapa dfs

import pickle: Importa la librería estándar para serializar y deserializar datos, utilizada para cargar el grafo desde disco.
import time: Importa herramientas para medir y evaluar el tiempo de ejecución de partes críticas del algoritmo.
import logging: Importa herramientas para crear registros (logs) de información, advertencias y errores durante la ejecución.
import random: Importa herramientas para generación de números aleatorios, utilizado para seleccionar nodos iniciales y muestras de nodos.
import igraph as ig: Importa la librería de análisis de grafos `igraph`, utilizada para representar y procesar grafos masivos.
from typing import Optional, List, Tuple: Importa herramientas para anotar tipos en el código, promoviendo legibilidad y calidad en entornos de análisis de datos.
import folium: Importa la librería para crear mapas interactivos en formato HTML a partir de datos geoespaciales.
from folium.plugins import BeautifyIcon: Importa herramientas para crear iconos personalizados en mapas de `folium`.
logging.basicConfig(...): Configura la salida de registros en consola y archivo para seguimiento de ejecución, errores y rendimiento.
def cargar_grafo(grafo_path): Define la función para deserializar y cargar un grafo en formato `pickle`, registrando duración y detalles.
def dfs_exploracion_desde_un_punto(...): Implementa la búsqueda en profundidad (DFS) para obtener la secuencia de nodos visitados y evaluar la duración total de la operación.
def crear_mapa_multicapa_dfs(...): Crea un mapa interactivo en `folium` con múltiples capas para representar tanto la ruta de la DFS como una muestra de nodos de la misma ruta.
start_coords = ... : Obtiene las coordenadas iniciales para centrar el mapa en el nodo de origen seleccionado para la búsqueda DFS.
camino_visualizado: Genera una versión subsampleada de la ruta DFS para representar visiblemente una línea en el mapa.
fg_camino: Crea una capa de mapa para representar el camino de la DFS en color naranja para una visualización clara.
fg_puntos: Crea una capa para representar un subconjunto de nodos alcanzados en la ruta de la DFS, marcado con puntos para resaltar detalles de la búsqueda.
folium.Marker(...): Añade marcadores para representar claramente los nodos inicial y final de la búsqueda DFS en el mapa.
folium.LayerControl(): Añade controles para alternar visibilidad de las diferentes capas en el mapa interactivo.
if __name__ == ¨__main__¨: Verifica que el script se ejecuta como programa principal para inicializar la carga de datos y análisis de la DFS.
mi_grafo = cargar_grafo(...): Carga el grafo serializado para análisis de datos masivos y operación de búsqueda.
SOURCE_NODE = random.randrange(...): Elige un nodo inicial al azar para realizar la búsqueda DFS

en el grafo cargado.

nodos_visitados_dfs, tiempo_dfs = dfs_exploracion_desde_un_punto(...): Ejecuta la búsqueda en profundidad para obtener la secuencia de nodos alcanzados junto con el tiempo total requerido para la operación.

mapa = crear_mapa_multicapa_dfs(...): Genera un mapa interactivo para representar los resultados de la búsqueda DFS en formato HTML.

mapa.save(...): Exporta el mapa resultante a un archivo HTML para su análisis y visualización en navegador.

print(...): Emite detalles clave para interpretar los resultados, duración de la operación, cantidad de nodos alcanzados y ruta del mapa generado.

V-C. Codigo de dijkstra.py

```python
# analisis_completo_de_red.py
"""
Script integral para el análisis de una red de gran escala.

Realiza las siguientes operaciones:
1. Carga un grafo pesado desde un archivo .pkl.
2. Encuentra automáticamente el par de nodos geográficamente más distantes.
3. Calcula el CAMINO MÁS CORTO entre ellos usando el algoritmo de Dijkstra.
4. Genera una muestra de 1000 caminos aleatorios para dar contexto a la red.
5. Crea un mapa HTML interactivo y optimizado con Folium que visualiza:
   - La ruta del camino más corto en una capa principal.
   - Los caminos aleatorios en otra capa, destacando el más largo y el más corto de la muestra.
"""

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
import folium
from folium.plugins import BeautifyIcon
from tqdm import tqdm


# --- 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 graph.es:
        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)
    }

    # Itera sobre los vértices para encontrar los 4 puntos extremos
    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

    # Compara las distancias entre los nodos extremos para encontrar el par más distante
    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

def encontrar_camino_mas_corto_dijkstra(graph: ig.Graph, source: int, sink: int) -> Tuple[Optional[List[int]], Optional[float], float]:
    """Calcula el camino más corto usando Dijkstra y devuelve el camino, su coste y el tiempo de ejecución."""
    logging.info(f"Iniciando algoritmo de Dijkstra de nodo {source} a {sink}.")
    start_time = time.time()

    # igraph es extremadamente rápido para esto.
    # shortest_paths devuelve el coste (longitud total del camino)
    # get_shortest_paths devuelve la secuencia de nodos (vértices)
    try:
        coste_total = graph.shortest_paths(source=source, target=sink, weights='weight')[0][0]
        camino_nodos = graph.get_shortest_paths(v=source, to=sink, weights='weight', output='vpath')[0]

        tiempo_total = time.time() - start_time
        logging.info(f"Dijkstra completado en {tiempo_total:.4f} segundos.")

        if not camino_nodos: # Si no hay camino, igraph devuelve lista vacía
            return None, None, tiempo_total

        return camino_nodos, coste_total, tiempo_total
    except Exception as e:
        logging.error(f"Error durante la ejecución de Dijkstra: {e}")
        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'])
        # Usar un mapa base en escala de grises para que los colores resalten
        m = folium.Map(location=start_coords, zoom_start=6, tiles="CartoDB positron")

        # Marcadores mejorados para origen y destino
        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)
```

```
        # Capa para el camino más corto
        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',  # Un azul distintivo
            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

def agregar_caminos_random_al_mapa(m: folium.Map, graph: ig.Graph, num_caminos: int):
    """Genera caminos aleatorios (usando Dijkstra) y los agrega al mapa para contexto."""

    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:
        # Intentar hasta encontrar un par de nodos válidos con un camino entre ellos
        for _ in range(10): # Limitar intentos para evitar bucles infinitos
            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):
                    # Calcular coste sumando los pesos de las aristas del 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 # Camino válido encontrado

    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)

# --- Bloque Principal de Ejecución ---
if __name__ == "__main__":
    # --- Configuración ---
    GRAFO_PKL_ENTRADA = 'grafo_igraph_paralelizado.pkl'
    MAPA_HTML_SALIDA = 'analisis_de_red_dijkstra.html'
    NUM_CAMINOS_RANDOM = 1000

    # 1. Cargar el grafo
    mi_grafo = cargar_grafo(GRAFO_PKL_ENTRADA)
```

```python
    if mi_grafo:
        # 1.5. Asegurar que el grafo tiene pesos para Dijkstra
        preparar_pesos_geograficos(mi_grafo)

        # 2. Encontrar source/sink automáticos
        par_distante = encontrar_nodos_mas_distantes_apr(mi_grafo)

        if par_distante:
            SOURCE_NODE, SINK_NODE = par_distante

            # 3. Calcular el camino más corto con Dijkstra
            camino_optimo, coste_total, tiempo_dijkstra = encontrar_camino_mas_corto_di(mi_grafo, SOURCE_NODE, SINK_NODE)

            if camino_optimo and coste_total is not None:
                # 4. Crear el mapa base con el camino más corto
                mapa_resultado = crear_mapa_camino_corto(mi_grafo, camino_optimo, coste_total, SOURCE_NODE, SINK_NODE)

                if mapa_resultado:
                    # 5. Agregar la capa de caminos aleatorios para contexto
                    agregar_caminos_random_al_ma(mapa_resultado, mi_grafo, NUM_CAMINOS_RANDOM)

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

                    # 7. Imprimir resumen final en la consola
                    print("\n" + "="*60)
                    print("      ANÁLISIS DE RUTA ÓPTIMA (DIJKSTRA) COMPLETADO")
                    print("="*60)
                    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")
                    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 por Dijkstra.")
                    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("="*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: Importa la biblioteca estándar para serialización y deserialización de datos (usada para guardar y cargar el grafo).

import time: Importa herramientas para medir duración de cálculos y evaluar rendimientos.

import logging: Importa herramientas para crear registros de información, advertencia y error en consola y archivo.

import random: Importa herramientas para generación de números y selecciones al azar, utilizado para seleccionar nodos iniciales y caminos de muestra.

from math import radians, sin, cos, sqrt, atan2: Importa funciones trigonométricas para cálculos

geoespaciales (distancia Haversine).
from itertools import combinations: Importa herramientas para generar todas las combinaciones posibles de elementos, utilizado para encontrar pares extremos de nodos.
import igraph as ig: Importa la librería para representar, procesar y analizar grafos de gran tamaño.
from typing import Dict, Optional, Tuple, List, Any: Importa anotaciones para mejorar la legibilidad y la seguridad de tipos de datos en el código.
import folium: Importa herramientas para crear mapas interactivos en formato HTML para análisis geoespacial.
from folium.plugins import BeautifyIcon: Importa herramientas para representar iconos personalizados en los mapas de `folium`.
from tqdm import tqdm: Importa herramientas para representar barras de progreso al generar múltiples caminos para análisis.
logging.basicConfig(...): Configura la salida de registros para guardar mensajes tanto en consola como en un archivo de registro para análisis posterior.
def cargar_grafo(grafo_path): Abre y deserializa un grafo en formato `pickle`, registrando duración y detalles básicos para evaluar carga de datos.
def haversine_distance(...): Implementa la fórmula de Haversine para determinar la distancia en kilómetros entre dos coordenadas geográficas.
def preparar_pesos_geograficos(...): Asigna pesos a todas las aristas del grafo según la distancia Haversine para habilitar algoritmos de camino mínimo como Dijkstra.
def encontrar_nodos_mas_distantes_aprox(...): Encuentra el par de nodos extremos en la distribución espacial para obtener un punto de partida significativo para análisis de caminos.
def encontrar_camino_mas_corto_dijkstra(...): Ejecuta Dijkstra para obtener el camino de menor coste entre dos nodos extremos, calculando duración total de la operación.
def crear_mapa_camino_corto(...): Crea un mapa base en `folium` para representar visiblemente la ruta óptima calculada, con marcadores para origen y destino.
def agregar_caminos_random_al_mapa(...): Genera una muestra de caminos aleatorios para representar diferentes escalas de distancia en la red, añadiéndolos como una segunda capa en el mapa.
if __name__ == ¨__main__¨: Punto de entrada para la ejecución directa del programa, donde se carga el grafo, calcula la ruta óptima, y construye el mapa de análisis final.
mi_grafo = cargar_grafo(...): Carga en memoria la representación serializada de la red para análisis de caminos y generación de mapas.
preparar_pesos_geograficos(...): Asegura que todas las aristas poseen pesos geográficos para permitir análisis de camino más corto por distancia real.
par_distante = encontrar_nodos_mas_distantes_aprox(...): Identifica un par de nodos extremos para evaluar caminos de largo alcance en la red.
camino_optimo, coste_total, tiempo_dijkstra = encontrar_camino_mas_corto_dijkstra(...): Ejecuta el análisis de camino más corto para obtener la ruta óptima junto con su coste y duración de cálculo.
mapa_resultado = crear_mapa_camino_corto(...): Genera la representación inicial de la ruta óptima para análisis e interacción en navegador.
agregar_caminos_random_al_mapa(...): Añade una segunda capa de caminos de muestra para mostrar la variabilidad y comparación de otras rutas posibles en la misma red.
folium.LayerControl().add_to(...): Añade un control de visibilidad para alternar entre diferentes capas en el mapa resultante.
mapa_resultado.save(...): Exporta el mapa final a un archivo `.html` para facilitar la distribución y análisis.
print(...): Muestra detalles clave del análisis en consola para obtener un resumen claro de la operación ejecutada, junto con la ruta al mapa resultante.

V-D. Codigo de community.py

```python
import pickle
import time
import logging
import random
from collections import Counter
import igraph as ig
from typing import Optional, List, Dict
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_lpa_final.log", mode='w', encoding='utf-8'),
logging.StreamHandler()])

# --- UMBRALES DE VISUALIZACIÓN ---
# Si una comunidad tiene más nodos que este umbral, no dibujaremos sus aristas.
UMBRAL_MAX_VISUALIZACION_ARISTAS = 2000
# Si una comunidad tiene más nodos que este umbral, solo dibujaremos una muestra aleatoria de ellos.
UMBRAL_MAX_NODOS_A_DIBUJAR = 5000

# --- 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

# --- Implementación del LPA ---
def detectar_comunidades_lpa(graph: ig.Graph, max_iter: int = 10) -> Dict[int, List[int]]:
    logging.info(f"Iniciando detección de comunidades con algoritmo propio (LPA) con {max_iter} iteraciones...")
    start_time = time.time()

    labels = {v.index: v.index for v in graph.vs}

    for i in range(max_iter):
        logging.info(f"LPA - Iteración {i + 1}/{max_iter}...");
        changes_count = 0

        nodes_to_process = list(range(graph.vcount())); random.shuffle(nodes_to_process)
        iterator = tqdm(nodes_to_process, desc=f"Iteración {i+1}")

        for node_id in iterator:
            neighbors = graph.neighbors(node_id, mode='all')
            if not neighbors: continue

            label_counts = Counter(labels[n] for n in neighbors)
            max_freq = max(label_counts.values())

            most_frequent_labels = [label for label, count in label_counts.items() if count == max_freq]
            new_label = random.choice(most_frequent_labels)

            if labels[node_id] != new_label:
                labels[node_id] = new_label
                changes_count += 1

        logging.info(f"Fin de la iteración {i + 1}. Hubo {changes_count} cambios de etiqueta.")
        if changes_count == 0:
            logging.info("Convergencia alcanzada antes del máximo de iteraciones."); break

    if i == max_iter - 1 and changes_count > 0:
        logging.warning("Se alcanzó el máximo de iteraciones sin convergencia completa.")

    comunidades = {};
    start_node = 1 if graph.vs[0].attributes().get('name') == 'dummy_root' else 0
    for node, label in labels.items():
        if node < start_node: continue
        if label not in comunidades: comunidades[label] = []
        comunidades[label].append(node)

    end_time = time.time()
    logging.info(f"LPA completado en {end_time - start_time:.2f} s. Se encontraron {len(comunidades)} comunidades.")
    return 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 (con todas las optimizaciones) ---
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)

        # Muestreo de nodos para comunidades gigantes
        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)

        # Dibujo de aristas solo para comunidades pequeñas
        if tamaño_original <= UMBRAL_MAX_VISUALIZACION_ARISTAS:
            if len(nodos_visibles_con_coords) > 1:
                subgrafo_comunidad = graph.subgraph(nodos_visibles_con_coords)

                # Log de diagnóstico
                num_aristas_internas = len(subgrafo_comunidad.es)
                logging.info(f"Comunidad {com_id} (tamaño {tamaño_original}): "
                             f"Intentando dibujar conexiones. Se encontraron {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: {tamaño_original} > {UMBRAL_MAX_VISUALIZACION_ARISTAS}).")

    folium.LayerControl(collapsed=False).add_to(m)
    logging.info("Guardando el mapa en el archivo HTML. Este paso puede tardar...")
    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_lpa_final.html'
    NUM_COMUNIDADES_RANDOM = 20
    MAX_ITER_LPA = 10

    mi_grafo = cargar_grafo(GRAFO_PKL_ENTRADA)
    if mi_grafo:
        coms_dict = detectar_comunidades_lpa(mi_grafo, max_iter=MAX_ITER_LPA)
        if coms_dict:
            comunidades_seleccionadas = analizar_y_seleccionar_comunidades(coms_dict, NUM_COMUNIDADES_RANDOM)
            if comunidades_seleccionadas:
                visualizar_comunidades(mi_grafo, coms_dict, comunidades_seleccionadas, MAPA_HTML_SALIDA)

                print("\n" + "="*60)
                print(" ANÁLISIS DE COMUNIDADES (LPA - VERSIÓN FINAL OPTIMIZADA)")
                print("="*60)
                print(f"Se encontraron un total de {len(coms_dict)} comunidades.")
                print("\nSe ha generado un mapa interactivo en: ", f"'{MAPA_HTML_SALIDA}'")
                print("\nOPTIMIZACIONES DE VISUALIZACIÓN:")
                print(f"  - Muestreo de Nodos: Para comunidades con > {UMBRAL_MAX_NODOS_A_DIBUJAR} miembros,")
                print("    solo se muestra una muestra aleatoria para no colapsar el navegador.")
                print(f"  - Omisión de Aristas: Los caminos solo se dibujan para comunidades")
                print(f"    con <= {UMBRAL_MAX_VISUALIZACION_ARISTAS} miembros.")
                print("\nEl color de cada comunidad indica su tamaño original:")
                print("  - Colores fríos (azul): Comunidades pequeñas.")
                print("  - Colores cálidos (rojo): Comunidades grandes.")
                print("\nPara más detalles, revisa el archivo de log: 'analisis_comunidades_lpa_final.log'")
                print("="*60)
```

Listing 5. Codigo para el mapa de comunidades

import pickle: Importa herramientas para serialización y deserialización de datos en disco (usadas para guardar y cargar grafos).
import time: Importa herramientas para medir duración de cálculos e informar rendimientos.
import logging: Importa herramientas para crear registros de log (información, advertencias, errores).
import random: Importa herramientas para generación de números y selección de elementos al azar (usadas para inicialización de etiquetas y muestreos).
from collections import Counter: Importa una estructura de datos para contabilizar elementos frecuentes, utilizada para evaluar la frecuencia de etiquetas en vecinos durante LPA.
import igraph as ig: Importa la librería de análisis de grafos de alto rendimiento para representar y procesar la estructura de la red.
from typing import Optional, List, Dict: Importa anotaciones para asegurar la consistencia de tipos en la declaración de variables y retorno de las funciones.
import folium: Importa herramientas para crear mapas interactivos para representar comunidades georreferenciadas.
from folium.plugins import MarkerCluster: Importa herramientas para representar grandes cantidades de marcadores en mapa sin saturarlo, mediante clusters de marcadores.
import matplotlib, matplotlib.colors as colors:

Importa herramientas para crear mapas de color en la representación gráfica de comunidades.
from tqdm import tqdm: Importa herramientas para representar barras de progreso durante cálculos intensivos.
logging.basicConfig(...): Configura registro de actividad para guardar en archivo y mostrar en consola.
UMBRAL_MAX_VISUALIZACION_ARISTAS: Umbral utilizado para decidir si representar todas las aristas de una comunidad en el mapa, para garantizar legibilidad.
UMBRAL_MAX_NODOS_A_DIBUJAR: Umbral utilizado para representar solo una muestra de nodos para comunidades excesivamente grandes y garantizar rendimientos óptimos en el navegador.
cargar_grafo(grafo_path): Carga la estructura de datos de un grafo igraph desde disco en formato pickle, registrando duración y detalles básicos.
detectar_comunidades_lpa(graph, max_iter): Ejecuta el algoritmo de propagación de etiquetas (LPA) para detectar comunidades en la red, con un número máximo de iteraciones configurables, incluyendo criterios de convergencia anticipada.
analizar_y_seleccionar_comunidades(comunidades_dict, num_random): Identifica comunidades clave para representar (más grande, más pequeña, y otras al azar), considerando criterios de tamaño para una visualización informada y variada.
crear_mapa_de_colores(tamaño_min, tamaño_max): Crea un mapa de colores para representar comunidades según su tamaño, escalado en una paleta de colores definida para distinguir comunidades grandes de pequeñas.
visualizar_comunidades(graph, comunidades_dict, seleccion, output_filename): Genera un mapa interactivo de todas las comunidades seleccionadas, utilizando `folium`. Incluye representación de nodos, muestreo para comunidades grandes, y representación de aristas para comunidades pequeñas para garantizar legibilidad y rendimiento.
if __name__ == ¨__main__¨: Bloque principal para la ejecución autónoma del análisis: carga el grafo, detecta comunidades, selecciona aquellas a representar y guarda el mapa resultante en formato HTML. Incluye mensajes para guiar al usuario y detalles de configuración y limitaciones en la representación final.

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.

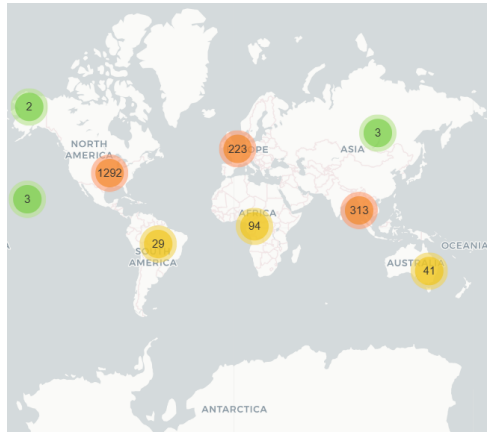

Fig 1. Muestra de la prueba de comunidades

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*

EEn 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.

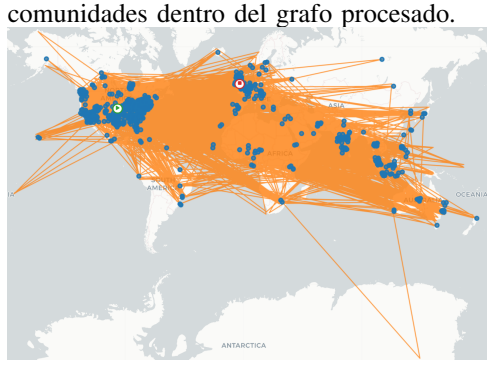

Fig 3. Mapa dfs

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.