

**“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	1
II	Librerías necesarias	2
II-A	random	2
II-B	igraph	2
II-C	time	2
II-D	colorsys	2
II-E	logging	2
II-F	fileinput	2
II-G	pickle	2
II-H	gc (Garbage Collector)	2
II-I	multiprocessing	2
II-J	itertools	2
II-K	folium	2
II-L	math	3
II-M	os	3
II-N	typing	3
II-Ñ	webbrowser	3
II-O	tqdm	3
II-P	matplotlib	3
II-Q	heapq	3
II-R	collections	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 communityLouvain.py	14
V-E	Codigo de bfs.py	19
V-F	Codigo de PrimBim.py	21
VI	Pruebas	23
VI-A	Analisis de Comunidad	23
VI-B	prueba de Dijkstra	24
VI-C	Analisis dfs	24
VI-D	Visualización de BFS en la Red	24
VII	Conclusión	25

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 usuraiois, 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 ìgraph.^{esto} 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 corra 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 características 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 características 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 características principales son: Calcular duración de ejecuciones, crear delays, medir rendimiento de algoritmos y análisis de eficiencia temporal.

II-D. *coloursys*

Es un módulo incluido en la biblioteca estándar que proporciona funciones para convertir colores entre diferentes sistemas de coordenadas de color. Sus usos y características principales son: Ajustar brillo, generar paletas de colores armónicos, conversión 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 características 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 características principales son: Procesar datos línea por línea desde archivos grandes, especialmente útil para procesamiento 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 características 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 características principales son: Optimizar el uso de memoria en procesamiento 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 características 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 procesamiento complejos de datos. Sus usos y características 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 características 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 características principales son: Realizar cálculos trigonométricos, exponenciales, logarítmicos y otras operaciones matemáticas para análisis numérico y científico.

II-M. *os*

Esta librería 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 características principales son: Crear o elimina archivos y carpetas, construye y verifica rutas de forma inteligente y ejecuta comandos del sistema además 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 características 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 librería 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 características principales son: Abrir URLs específicas, 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 características 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 características 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 librería 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 características 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 librería 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 características 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. Código de *createdata.py*

```

1 import random
2
3 def generar_ubicaciones(num_ubicaciones,
4     ↪ nombre_archivo="1_million_location.
5     ↪ txt"):
6     """
7     Genera un archivo de texto con
8     ↪ coordenadas de latitud y longitud
9     ↪ aleatorias.
10
11     Args:

```

```

8     num_ubicaciones (int): El número de
        ↳ ubicaciones a generar.
9     nombre_archivo (str, opcional): El
        ↳ nombre del archivo a crear.
        Por defecto es "Xnumber_location.
        ↳ txt".
10
11     """
12     try:
13         with open(nombre_archivo, 'w') as
            ↳ archivo:
14             for _ in range(num_ubicaciones):
15                 # Genera latitud y longitud
                    ↳ aleatorias dentro de
                    ↳ rangos razonables
16                 latitud = random.uniform(-90,
                    ↳ 90) # Valores de latitud
                    ↳ entre -84 y -82
17                 longitud = random.uniform(-180,
                    ↳ 180) # Valores de
                    ↳ longitud 134 y 136
18                 # Escribe la coordenada en el
                    ↳ archivo con el formato
                    ↳ especificado
19                 archivo.write(f"{latitud},{
                    ↳ longitud}\n")
20             print(f"Se ha generado el archivo '{
                ↳ nombre_archivo}' con {
                ↳ num_ubicaciones} ubicaciones."
                ↳ )
21     except Exception as e:
22         print(f"Ocurrió un error al generar
            ↳ el archivo: {e}")
23
24 def generar_conexiones(num_conexiones,
        ↳ nombre_archivo="1_million_user.txt")
        ↳ :
25     """
26     Genera un archivo de texto con
        ↳ conexiones de aristas aleatorias,
        ↳ simulando usuarios
27     y sus interacciones con ubicaciones.
28
29     Args:
30         num_conexiones (int): El número de
            ↳ conexiones de aristas a
            ↳ generar (filas en el archivo).
31         nombre_archivo (str, opcional): El
            ↳ nombre del archivo a crear.
            Por defecto es "Xnumber_user.txt".
32
33     """
34     try:
35         with open(nombre_archivo, 'w') as
            ↳ archivo:
36             for i in range(num_conexiones):
37                 # Genera un número aleatorio de
                    ↳ ubicaciones visitadas
                    ↳ por el usuario
38                 num_ubicaciones_visitadas =
                    ↳ random.randint(0, 100) #
                    ↳ Cada usuario visita
                    ↳ entre 1 y 20 ubicaciones
39                 # Genera una lista de
                    ↳ ubicaciones visitadas (í
                    ↳ ndices)
40                 ubicaciones_visitadas = [random
                    ↳ .randint(1, 1000700) for
                    ↳ _ in range(
                    ↳ num_ubicaciones_visitadas
                    ↳ )] # Asume que tienes
                    ↳ 10,000 ubicaciones
41                 # Escribe las conexiones en el
                    ↳ archivo con el formato
                    ↳ especificado
42                 archivo.write(f"{i},{','.join(
                    ↳ map(str,
                    ↳ ubicaciones_visitadas))
                    ↳ }\n")

```

```

43     print(f"Se ha generado el archivo '{
        ↳ nombre_archivo}' con {
        ↳ num_conexiones} conexiones.")
44 except Exception as e:
45     print(f"Ocurrió un error al generar
        ↳ el archivo: {e}")
46
47 if __name__ == "__main__":
48     # Genera 10,000 ubicaciones en el
        ↳ archivo "10_thousand_location.txt
        ↳ "
49     generar_ubicaciones(1000000)
50     generar_conexiones(1000000)
51     #generar_ubicaciones(10, "test_location.
        ↳ txt") #para pruebas

```

Listing 1. Código para la generación 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. Código de *grapyparalel.py*

```

1 import igraph as ig
2 import time
3 import logging
4 import fileinput
5 import pickle
6 import gc
7 import multiprocessing as mp
8 from itertools import chain
9
10 NUM_NODOS = 10_000_000

```

```

11
12 # Configuración del logging para registrar
    ↳ información y advertencias en
    ↳ archivo y consola
13 logging.basicConfig(
14     level=logging.INFO,
15     format='%(asctime)s - %(levelname)s -
    ↳ %(message)s',
16     handlers=[
17         logging.FileHandler("
    ↳ grafo_parallelizado.log", mode=
    ↳ 'w', encoding='utf-8'),
18         logging.StreamHandler()
19     ]
20 )
21
22 def cargar_ubicaciones_directo(
    ↳ ubicaciones_path, max_nodos):
23     logging.info(f"Cargando un máximo de {
    ↳ max_nodos} ubicaciones desde el
    ↳ archivo...")
24     latitudes = []
25     longitudes = []
26     with open(ubicaciones_path, 'r') as f:
27         for i, line in enumerate(f):
28             if i >= max_nodos:
29                 logging.info(f"Límite de {
    ↳ max_nodos} nodos
    ↳ alcanzado. Se detiene la
    ↳ lectura de ubicaciones.
    ↳ ")
30                 break
31             try:
32                 # El formato en el archivo
    ↳ original parece ser lat,
    ↳ lon
33                 lat_str, lon_str = line.strip()
    ↳ .split(',')
34                 latitudes.append(float(lat_str)
    ↳ )
35                 longitudes.append(float(lon_str)
    ↳ )
36             except ValueError:
37                 logging.warning(f"Línea
    ↳ malformada en el archivo
    ↳ de ubicaciones: '{line.
    ↳ strip()}'")
38
39     ubicaciones = list(zip(latitudes,
    ↳ longitudes))
40     num_nodos_reales = len(ubicaciones)
41
42     # Advertir si el archivo tiene menos
    ↳ nodos que el esperado
43     if num_nodos_reales < max_nodos:
44         logging.warning(f"El archivo de
    ↳ ubicaciones contiene {
    ↳ num_nodos_reales} nodos, menos
    ↳ que el máximo esperado de {
    ↳ max_nodos}.")
45
46     logging.info(f"Se cargaron {
    ↳ num_nodos_reales} ubicaciones.")
47     return ubicaciones, num_nodos_reales, {'
    ↳ lat': latitudes, 'lon':
    ↳ longitudes}
48
49 def procesar_linea_usuarios(
    ↳ linea_idx_contenido, num_nodos_max):
50     idx, linea = linea_idx_contenido
51     aristas_locales = []
52     conexiones = set()
53     for x in linea.strip().split(','):
54         x_strip = x.strip()
55         if x_strip.isdigit():
56             dst = int(x_strip)

```

```

57         # Validar contra el número máximo
    ↳ de nodos permitidos
58         if 1 <= dst <= num_nodos_max:
59             conexiones.add(dst - 1) #
    ↳ Ajuste a índice base 0
60
61     # El nodo origen (idx-1) también debe
    ↳ ser válido
62     if 0 <= (idx - 1) < num_nodos_max:
63         aristas_locales.extend((idx - 1, dst)
    ↳ for dst in conexiones)
64
65     return aristas_locales
66
67 def procesar_usuarios_parallelizado(
    ↳ usuarios_path, num_nodos,
    ↳ num_procesos=mp.cpu_count()):
68     logging.info(f"Procesando usuarios para
    ↳ {num_nodos} nodos en paralelo con
    ↳ {num_procesos} procesos...")
69     start_time_procesamiento = time.time()
70
71     with fileinput.input(usuarios_path) as f:
72         # Creamos un generador que solo lee
    ↳ hasta la línea 'num_nodos'
73         lineas_a_procesar = ((idx, linea) for
    ↳ idx, linea in enumerate(f,
    ↳ start=1) if idx <= num_nodos)
74
75         with mp.Pool(processes=num_procesos)
    ↳ as pool:
76             # Pasamos num_nodos a cada proceso
    ↳ para la validación
77             resultados = pool.starmap(
    ↳ procesar_linea_usuarios, [(
    ↳ item, num_nodos) for item
    ↳ in lineas_a_procesar])
78
79     # Aplanar la lista de resultados
80     aristas = list(chain.from_iterable(
    ↳ resultados))
81
82     # El filtrado ya se hace dentro de '
    ↳ procesar_linea_usuarios', pero
    ↳ una verificación final no está de
    ↳ más.
83     aristas_filtradas = [(src, dst) for src,
    ↳ dst in aristas if 0 <= src <
    ↳ num_nodos and 0 <= dst <
    ↳ num_nodos]
84
85     logging.info(f"Se procesaron {len(
    ↳ aristas_filtradas)} aristas en {
    ↳ time.time() -
    ↳ start_time_procesamiento:.2f} s."
    ↳ )
86     return aristas_filtradas
87
88 def crear_grafo_igraph_parallelizado(
    ↳ ubicaciones_path, usuarios_path,
    ↳ output_grafo_path, max_nodos):
89     start_time = time.time()
90
91     # 1. Cargar ubicaciones, usando
    ↳ max_nodos como límite.
92     ubicaciones, num_nodos_reales,
    ↳ atributos_ubicacion =
    ↳ cargar_ubicaciones_directo(
    ↳ ubicaciones_path, max_nodos)
93
94     # 2. Procesar conexiones, usando el nú
    ↳ mero real de nodos como límite
    ↳ para la consistencia.
95     aristas = procesar_usuarios_parallelizado
    ↳ (usuarios_path, num_nodos_reales)
96

```

```

97 logging.info("Creando grafo de igraph...
    ↪ ")
98 g = ig.Graph(directed=True)
99 # Añadimos la cantidad real de vértices
    ↪ encontrados
100 g.add_vertices(num_nodos_reales)
101 g.add_edges(aristas)
102
103 # Añadir atributos de ubicación al grafo
104 for key, values in atributos_ubicacion.
    ↪ items():
105     g.vs[key] = values
106
107 # Guardar el grafo en un archivo pickle
108 logging.info(f"Guardando grafo con
    ↪ atributos en '{output_grafo_path
    ↪ }'...")
109 with open(output_grafo_path, 'wb') as f:
110     pickle.dump(g, f, protocol=pickle.
    ↪ HIGHEST_PROTOCOL)
111
112 logging.info(f"Grafo guardado. Tiempo
    ↪ total: {time.time() - start_time
    ↪ :.2f} s")
113 del ubicaciones, aristas,
    ↪ atributos_ubicacion, g
114 gc.collect()
115
116 if __name__ == "__main__":
117     ubicaciones_archivo = '10
    ↪ _million_location.txt'
118     usuarios_archivo = '10_million_user.txt'
119     grafo_archivo = '
    ↪ grafo_igraph_paralelizado.pkl'
120
121 # Crear y guardar el grafo, pasando la
    ↪ constante global como el límite m
    ↪ áximo.
122 crear_grafo_igraph_paralelizado(
    ↪ ubicaciones_archivo,
    ↪ usuarios_archivo, grafo_archivo,
    ↪ max_nodos=NUM_NODOS)
123
124 gc.collect()

```

Listing 2. Código 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. Código de dfs.py

```

1 import pickle
2 import time
3 import logging
4 import random
5 import igraph as ig
6 from typing import Optional, List, Tuple
7 import folium
8 from folium.plugins import BeautifyIcon
9 import colorsys
10
11 # --- Constantes y Configuración ---
12 SOURCE_NODE = 1
13 NUM_CAPAS_DESEADAS = 10
14 # Reduce el camino de ~10 millones de nodos
    ↪ a este número antes de crear el
    ↪ mapa.
15 MAX_NODOS_PARA_MAPA = 50
16
17 # --- Configuración del Logging ---
18 logging.basicConfig(level=logging.INFO,
    ↪ format='%(asctime)s - %(levelname)s
    ↪ - %(message)s',
19                     handlers=[logging.
    ↪ FileHandler("
    ↪ analisis_dfs.log",
    ↪ mode='w', encoding='
    ↪ utf-8'),

```



```

20         logging.
21             ↳ StreamHandler
22             ↳ ())
23 # --- Funciones Auxiliares (Sin cambios)
24     ↳ ---
25 def cargar_grafo(grafo_path: str) ->
26     ↳ Optional[ig.Graph]:
27     """Carga un grafo desde un archivo
28     ↳ pickle."""
29     logging.info(f"Cargando el grafo desde
30     ↳ '{grafo_path}'...")
31     start_time = time.time()
32     try:
33         with open(grafo_path, 'rb') as f:
34             g = pickle.load(f)
35             end_time = time.time()
36             logging.info(f"Grafo cargado en {
37             ↳ end_time - start_time:.2f}
38             ↳ segundos. {g.summary()}")
39         return g
40     except Exception as e:
41         logging.error(f"Error crítico al
42         ↳ cargar el grafo: {e}")
43     return None
44
45 def dfs_con_profundidad(graph: ig.Graph,
46     ↳ source: int) -> Tuple[List[int],
47     ↳ float, int, int]:
48     """Realiza un recorrido DFS y devuelve
49     ↳ el camino, el tiempo, y el nodo m
50     ↳ ás profundo."""
51     logging.info(f"Iniciando travesía DFS
52     ↳ desde el nodo {source} y
53     ↳ calculando profundidad.")
54     start_time = time.time()
55     stack = [(source, 0)]
56     visitados = {source}
57     nodos_explorados_en_orden = []
58     max_profundidad, nodo_mas_profundo = 0,
59     ↳ source
60     while stack:
61         current_node, current_depth = stack.
62         ↳ pop()
63         nodos_explorados_en_orden.append(
64         ↳ current_node)
65         if current_depth > max_profundidad:
66             max_profundidad, nodo_mas_profundo =
67             ↳ current_depth,
68             ↳ current_node
69         for neighbor in reversed(graph.
70         ↳ neighbors(current_node, mode='
71         ↳ out')):
72             if neighbor not in visitados:
73                 visitados.add(neighbor)
74                 stack.append((neighbor,
75                 ↳ current_depth + 1))
76     tiempo_total = time.time() - start_time
77     logging.info(f"Travesía DFS completada.
78     ↳ Nodos: {len(
79     ↳ nodos_explorados_en_orden)}. Nodo
80     ↳ más profundo: {nodo_mas_profundo
81     ↳ } (prof: {max_profundidad}).")
82     return nodos_explorados_en_orden,
83     ↳ tiempo_total, nodo_mas_profundo,
84     ↳ max_profundidad
85
86 def generar_color_gradiente_hls(paso_actual
87     ↳ : int, total_pasos: int) -> str:
88     """Genera un color en formato
89     ↳ hexadecimal que va de azul (
90     ↳ inicio) a rojo (final)."""
91     if total_pasos <= 1: return "#FF0000"
92     hue = (240/360) * (1 - (paso_actual / (
93     ↳ total_pasos - 1)))
94     r, g, b = colorsys.hls_to_rgb(hue, 0.5,
95     ↳ 0.8)

```

```

63     return f'#{int(r*255):02x}{int(g*255):02
64     ↳ x}{int(b*255):02x}'
65
66 # --- FUNCIÓN DE VISUALIZACIÓN OPTIMIZADA
67     ↳ CON MARCADORES DE CAPA ---
68 def crear_mapa_dfs_por_capas(
69     graph: ig.Graph,
70     source_node: int,
71     camino_a_dibujar: List[int], # Recibe la
72     ↳ lista ya reducida
73     nodo_mas_profundo: int,
74     ultimo_nodo_real: int
75 ) -> Optional[folium.Map]:
76     """
77     Crea un mapa estático usando un camino
78     ↳ pre-reducido y añade marcadores
79     ↳ de inicio/fin por capa.
80
81     """
82     logging.info(f"Creando mapa estático
83     ↳ para dividirse en {
84     ↳ NUM_CAPAS_DESEADAS} capas.")
85
86     start_coords = (graph.vs[source_node]['
87     ↳ lat'], graph.vs[source_node]['lon
88     ↳ '])
89     m = folium.Map(location=start_coords,
90     ↳ zoom_start=3, tiles="CartoDB
91     ↳ positron")
92
93     total_pasos_visuales = len(
94     ↳ camino_a_dibujar)
95
96     # Capa de Vértices
97     fg_vertices = folium.FeatureGroup(name=f
98     ↳ "Vértices del Camino Visualizado
99     ↳ ({total_pasos_visuales})", show=
100     ↳ True)
101     for node_id in camino_a_dibujar:
102         coords = (graph.vs[node_id]['lat'],
103         ↳ graph.vs[node_id]['lon'])
104         folium.CircleMarker(
105         location=coords, radius=3, color='
106         ↳ #3186cc', fill=True,
107         ↳ fill_color='#3186cc',
108         ↳ fill_opacity=0.6,
109         popup=f"Nodo {node_id}"
110         ).add_to(fg_vertices)
111     m.add_child(fg_vertices)
112
113     # Lógica para dividir el camino en capas
114     num_segmentos = total_pasos_visuales - 1
115     segmentos_por_capa = max(1,
116     ↳ num_segmentos //
117     ↳ NUM_CAPAS_DESEADAS)
118
119     for i in range(0, num_segmentos,
120     ↳ segmentos_por_capa):
121         start_paso, end_paso = i + 1, min(i +
122         ↳ segmentos_por_capa,
123         ↳ num_segmentos)
124
125         fg_camino_segmentado = folium.
126         ↳ FeatureGroup(name=f"Camino (
127         ↳ Pasos {start_paso}-{end_paso})
128         ↳ ", show=(i == 0))
129
130         # Dibuja las líneas de la capa
131         for j in range(i, end_paso):
132             coords = [(graph.vs[
133             ↳ camino_a_dibujar[j]]['lat'
134             ↳ ], graph.vs[
135             ↳ camino_a_dibujar[j]]['lon'
136             ↳ ]), (graph.vs[
137             ↳ camino_a_dibujar[j+1]]['lat
138             ↳ '], graph.vs[
139             ↳ camino_a_dibujar[j+1]]['lon

```



```

106         ↪ ''])
107         color_segmento =
108             ↪ generar_color_gradiente_hls
109             ↪ (j, total_pasos_visuales)
110         folium.PolyLine(locations=coords,
111             ↪ color=color_segmento,
112             ↪ weight=2.5, opacity=1.0).
113             ↪ add_to(fg_camino_segmentado
114             ↪ )
115
116     # --- NUEVO: AÑADIR MARCADORES DE
117     ↪ INICIO Y FIN PARA ESTA CAPA
118     ↪ ---
119     start_node_layer = camino_a_dibujar[i
120     ↪ ]
121     end_node_layer = camino_a_dibujar[
122     ↪ end_paso]
123
124     folium.CircleMarker(
125         location=(graph.vs[
126             ↪ start_node_layer]['lat'],
127             ↪ graph.vs[start_node_layer][
128             ↪ 'lon']),
129         radius=5, color='green', fill=True
130         ↪ , fill_color='lightgreen',
131         ↪ tooltip=f"Inicio Capa: Paso {
132         ↪ start_paso}"
133     ).add_to(fg_camino_segmentado)
134
135     folium.CircleMarker(
136         location=(graph.vs[end_node_layer
137             ↪ ]['lat'], graph.vs[
138             ↪ end_node_layer]['lon']),
139         radius=5, color='red', fill=True,
140         ↪ fill_color='#ff7f7f', # Un
141         ↪ rojo más claro
142         ↪ tooltip=f"Fin Capa: Paso {end_paso
143         ↪ }"
144     ).add_to(fg_camino_segmentado)
145
146     m.add_child(fg_camino_segmentado)
147
148     # Marcadores Especiales (Generales)
149     folium.Marker(location=start_coords,
150         ↪ popup=f"INICIO DFS: Nodo {
151         ↪ source_node}", tooltip="Punto de
152         ↪ Inicio General",
153         icon=BeautifulIcon(icon='play'
154             ↪ , border_color='#2
155             ↪ ca02c', text_color='#2
156             ↪ ca02c')).add_to(m)
157
158     end_real_coords = (graph.vs[
159         ↪ ultimo_nodo_real]['lat'], graph.
160         ↪ vs[ultimo_nodo_real]['lon'])
161     folium.Marker(location=end_real_coords,
162         ↪ popup=f"FIN DEL RECORRIDO: Nodo {
163         ↪ ultimo_nodo_real}", tooltip="Ú
164         ↪ ltimo Nodo Visitado General",
165         icon=BeautifulIcon(icon='stop'
166             ↪ , border_color='#
167             ↪ d62728', text_color='#
168             ↪ d62728')).add_to(m)
169
170     deepest_coords = (graph.vs[
171         ↪ nodo_mas_profundo]['lat'], graph.
172         ↪ vs[nodo_mas_profundo]['lon'])
173     folium.Marker(location=deepest_coords,
174         ↪ popup=f"NODO MÁS PROFUNDO: Nodo {
175         ↪ nodo_mas_profundo}", tooltip="
176         ↪ Nodo Más Profundo General",
177         icon=BeautifulIcon(icon='star'
178             ↪ , border_color='
179             ↪ #800080', text_color='
180             ↪ #800080', spin=True)).
181         ↪ add_to(m)
182
183     folium.LayerControl(collapsed=False).
184     ↪ add_to(m)

```

```

138     return m
139
140     # --- Bloque Principal de Ejecución ---
141     if __name__ == "__main__":
142         GRAFO_PKL_ENTRADA = '
143             ↪ grafo_igraph_paralelizado.pkl'
144         MAPA_HTML_SALIDA = '
145             ↪ mapa_dfs_con_marcadores_de_capa.
146             ↪ html'
147
148         mi_grafo = cargar_grafo(
149             ↪ GRAFO_PKL_ENTRADA)
150
151         if mi_grafo and mi_grafo.vs:
152             nodos_visitados_dfs, tiempo_dfs,
153             ↪ nodo_profundo, prof_max =
154             ↪ dfs_con_profundidad(mi_grafo,
155             ↪ SOURCE_NODE)
156
157         if nodos_visitados_dfs:
158             logging.info(f"Reduciendo camino
159             ↪ de {len(nodos_visitados_dfs)
160             ↪ }) nodos a un máximo de {
161             ↪ MAX_NODOS_PARA_MAPA} para
162             ↪ visualización.")
163         if len(nodos_visitados_dfs) >
164             ↪ MAX_NODOS_PARA_MAPA:
165             step = len(nodos_visitados_dfs)
166             ↪ // MAX_NODOS_PARA_MAPA
167             camino_para_mapa =
168             ↪ nodos_visitados_dfs[:
169             ↪ step]
170         if camino_para_mapa[-1] !=
171             ↪ nodos_visitados_dfs[-1]:
172             camino_para_mapa.append(
173                 ↪ nodos_visitados_dfs
174                 ↪ [-1])
175         else:
176             camino_para_mapa =
177                 ↪ nodos_visitados_dfs
178
179         logging.info(f"El camino para el
180             ↪ mapa ahora tiene {len(
181             ↪ camino_para_mapa)} nodos.
182             ↪ Creando mapa...")
183
184         mapa = crear_mapa_dfs_por_capas(
185             graph=mi_grafo,
186             source_node=SOURCE_NODE,
187             camino_a_dibujar=
188                 ↪ camino_para_mapa,
189             nodo_mas_profundo=nodo_profundo
190             ↪ ,
191             ultimo_nodo_real=
192                 ↪ nodos_visitados_dfs[-1]
193             ↪ )
194
195         if mapa:
196             mapa.save(MAPA_HTML_SALIDA)
197             print("\n" + "="*60)
198             print(" MAPA OPTIMIZADO CREADO
199                 ↪ CON ÉXITO")
200             print("="*60)
201             print(f"Mapa interactivo
202                 ↪ guardado en: '{
203                 ↪ MAPA_HTML_SALIDA}'")
204             print("\nNovedades en el mapa:")
205             ↪
206             print(f" - El camino visual se
207                 ↪ ha dividido en ~{
208                 ↪ NUM_CAPAS_DESEADAS}
209                 ↪ capas.")
210             print(" - CADA CAPA DE CAMINO
211                 ↪ AHORA TIENE SU PROPIO
212                 ↪ MARCADOR DE INICIO Y FIN
213                 ↪ .")

```

```

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

```

Listing 3. Código 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 'Feature-Groups' 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. Código de dijkstra.py

```

1 # analisis_completo_de_red.py
2 import pickle
3 import time
4 import logging
5 import random
6 from math import radians, sin, cos, sqrt,
   ↳ atan2
7 from itertools import combinations
8 import igraph as ig
9 from typing import Dict, Optional, Tuple,
   ↳ List, Any, Set
10 import folium
11 from folium.plugins import BeautifyIcon
12 from tqdm import tqdm
13 import heapq
14
15 # --- Configuración del Logging ---
16 logging.basicConfig(
17     level=logging.INFO,
18     format='%(asctime)s - %(levelname)s - %(
   ↳ message)s',
19     handlers=[
20         logging.FileHandler("analisis_de_red.
   ↳ log", mode='w', encoding='utf
   ↳ -8'),
21         logging.StreamHandler()
22     ]
23 )
24
25 # --- Funciones de Carga y Cálculo ---
26
27 def cargar_grafo(grafo_path: str) ->
   ↳ Optional[ig.Graph]:
28     """Carga un grafo igraph desde un
   ↳ archivo .pkl."""
29     logging.info(f"Cargando el grafo desde
   ↳ '{grafo_path}'...")
30     start_time = time.time()
31     try:
32         with open(grafo_path, 'rb') as f:
33             g = pickle.load(f)
34         end_time = time.time()
35         logging.info(f"Grafo cargado en {
   ↳ end_time - start_time:.2f}
   ↳ segundos. {g.summary()}")
36         return g
37     except Exception as e:

```

```

38     logging.error(f"Error crítico al
    ↪ cargar el grafo: {e}")
39     return None
40
41 def haversine_distance(lat1: float, lon1:
    ↪ float, lat2: float, lon2: float) ->
    ↪ float:
42     """Calcula la distancia en kilómetros
    ↪ entre dos puntos geográficos."""
43     R = 6371 # Radio de la Tierra en km
44     dlat, dlon = radians(lat2 - lat1),
    ↪ radians(lon2 - lon1)
45     a = sin(dlat / 2)**2 + cos(radians(lat1)
    ↪ ) * cos(radians(lat2)) * sin(dlon
    ↪ / 2)**2
46     return R * 2 * atan2(sqrt(a), sqrt(1 - a
    ↪ ))
47
48 def preparar_pesos_geograficos(graph: ig.
    ↪ Graph, force_recalc: bool = False)
    ↪ -> None:
49     """Asegura que cada arista tenga un
    ↪ atributo 'weight' con su
    ↪ distancia haversine."""
50     if 'weight' in graph.es.attributes() and
    ↪ not force_recalc:
51         logging.info("El atributo 'weight' ya
    ↪ existe en las aristas. No se
    ↪ recalculará.")
52     return
53
54     logging.info("Calculando pesos geográ
    ↪ ficos (distancia Haversine) para
    ↪ cada arista...")
55     pesos = []
56     for edge in tqdm(graph.es, desc="
    ↪ Calculando pesos"):
57         source_v = graph.vs[edge.source]
58         target_v = graph.vs[edge.target]
59         dist = haversine_distance(source_v['
    ↪ lat'], source_v['lon'],
    ↪ target_v['lat'], target_v['lon']
    ↪ )
60         pesos.append(dist)
61     graph.es['weight'] = pesos
62     logging.info("Pesos geográficos
    ↪ calculados y asignados a las
    ↪ aristas.")
63
64
65 def encontrar_nodos_mas_distantes_aprox(
    ↪ graph: ig.Graph) -> Optional[Tuple[
    ↪ int, int]]:
66     """Encuentra un par aproximado de los
    ↪ nodos más distantes usando la
    ↪ caja delimitadora."""
67     logging.info("Buscando el par de nodos
    ↪ geográficamente más distantes (
    ↪ aproximación)...")
68     extremos: Dict[str, Tuple[float, Any]] =
    ↪ {
69         'min_lat': (float('inf'), None), '
    ↪ max_lat': (float('-inf'), None
    ↪ ),
70         'min_lon': (float('inf'), None), '
    ↪ max_lon': (float('-inf'), None
    ↪ )
71     }
72
73     for v in graph.vs:
74         if v['lat'] is None or v['lon'] is
    ↪ None: continue
75         lat, lon = v['lat'], v['lon']
76         if lat < extremos['min_lat'][0]:
    ↪ extremos['min_lat'] = (lat, v.
    ↪ index)
77         if lat > extremos['max_lat'][0]:
    ↪ extremos['max_lat'] = (lat, v.
    ↪ index)
78         if lon < extremos['min_lon'][0]:
    ↪ extremos['min_lon'] = (lon, v.
    ↪ index)
79         if lon > extremos['max_lon'][0]:
    ↪ extremos['max_lon'] = (lon, v.
    ↪ index)
80
81     nodos_candidatos_ids = {node_id for _,
    ↪ node_id in extremos.values() if
    ↪ node_id is not None}
82     if len(nodos_candidatos_ids) < 2:
83         logging.error("No se encontraron
    ↪ suficientes nodos con
    ↪ coordenadas para determinar un
    ↪ rango.")
84     return None
85
86     max_dist, par_mas_distante = -1.0, None
87     for u_id, v_id in combinations(
    ↪ nodos_candidatos_ids, 2):
88         dist = haversine_distance(graph.vs[
    ↪ u_id]['lat'], graph.vs[u_id]['
    ↪ lon'], graph.vs[v_id]['lat'],
    ↪ graph.vs[v_id]['lon'])
89         if dist > max_dist:
90             max_dist, par_mas_distante = dist,
    ↪ (u_id, v_id)
91
92     if par_mas_distante:
93         logging.info(f"Par más distante
    ↪ encontrado: Nodos {
    ↪ par_mas_distante} (Distancia:
    ↪ {max_dist:.2f} km)")
94     return par_mas_distante
95
96 # --- FUNCIÓN DE DIJKSTRA REIMPLEMENTADA
    ↪ ---
97 def encontrar_camino_mas_corto_dijkstra(
    ↪ graph: ig.Graph, source: int, sink:
    ↪ int) -> Tuple[Optional[List[int]],
    ↪ Optional[float], float]:
98     logging.info(f"Iniciando algoritmo de
    ↪ Dijkstra (implementación manual)
    ↪ de nodo {source} a {sink}.")
99     start_time = time.time()
100
101     distances = {v.index: float('inf') for v
    ↪ in graph.vs}
102     previous_nodes = {v.index: None for v in
    ↪ graph.vs}
103     distances[source] = 0
104     priority_queue = [(0, source)]
105     try:
106         while priority_queue:
107             current_distance, current_node_id
    ↪ = heapq.heappop(
    ↪ priority_queue)
108
109             if current_node_id == sink:
110                 logging.info("Destino alcanzado
    ↪ . Reconstruyendo el
    ↪ camino.")
111                 break
112
113             if current_distance > distances[
    ↪ current_node_id]:
114                 continue
115
116             for neighbor_id in graph.neighbors
    ↪ (current_node_id, mode='out
    ↪ '):
117                 edge_id = graph.get_eid(
    ↪ current_node_id,
    ↪ neighbor_id)

```

```

118         weight = graph.es[edge_id]['
119             ↳ weight']
120         distance_through_current =
121             ↳ distances[
122                 ↳ current_node_id] +
123                 ↳ weight
124
125         if distance_through_current <
126             ↳ distances[neighbor_id]:
127             distances[neighbor_id] =
128                 ↳ distance_through_current
129                 ↳
130             previous_nodes[neighbor_id]
131                 ↳ = current_node_id
132             heapq.heappush(
133                 ↳ priority_queue, (
134                     ↳ distance_through_current
135                     ↳ , neighbor_id))
136
137     path = []
138     if distances[sink] == float('inf'):
139         tiempo_total = time.time() -
140             ↳ start_time
141         logging.warning(f"No se encontró
142             ↳ un camino entre {source} y
143             ↳ {sink}.")
144         return None, None, tiempo_total
145
146     current = sink
147     while current is not None:
148         path.append(current)
149         current = previous_nodes[current]
150
151     path.reverse()
152     coste_total = distances[sink]
153     tiempo_total = time.time() -
154         ↳ start_time
155     logging.info(f"Dijkstra (manual)
156         ↳ completado en {tiempo_total:.4
157         ↳ f} segundos.")
158
159     if path[0] != source:
160         logging.error("Error en la
161             ↳ reconstrucción del camino.")
162         ↳ )
163         return None, None, tiempo_total
164
165     return path, coste_total,
166         ↳ tiempo_total
167
168 except Exception as e:
169     logging.error(f"Error durante la
170         ↳ ejecución de Dijkstra (manual)
171         ↳ : {e}", exc_info=True)
172     tiempo_total = time.time() -
173         ↳ start_time
174     return None, None, tiempo_total
175
176 # --- Funciones de Visualización con Folium
177 ↳ ---
178
179 def crear_mapa_camino_corto(graph: ig.Graph
180     ↳ , camino: List[int], coste: float,
181     ↳ source: int, sink: int) -> Optional[
182     ↳ folium.Map]:
183     """Crea un mapa base con el camino más
184         ↳ corto encontrado."""
185     logging.info(f"Creando mapa base con el
186         ↳ camino más corto...")
187     try:
188         start_coords = (graph.vs[source]['lat'
189             ↳ , graph.vs[source]['lon'])
190         m = folium.Map(location=start_coords,
191             ↳ zoom_start=6, tiles="CartoDB
192             ↳ positron")
193
194         icon_source = BeautifyIcon(icon='play
195             ↳ , border_color='#2ca02c',
196             ↳ text_color='#2ca02c',
197             ↳ icon_shape='circle')
198         icon_sink = BeautifyIcon(icon='stop',
199             ↳ border_color='#d62728',
200             ↳ text_color='#d62728',
201             ↳ icon_shape='circle')
202         folium.Marker(location=start_coords,
203             ↳ popup=f"SOURCE: Nodo {source}"
204             ↳ , tooltip="Punto de Origen",
205             ↳ icon=icon_source).add_to(m)
206         folium.Marker(location=(graph.vs[sink]
207             ↳ , graph.vs[sink]['lon'
208             ↳ , graph.vs[sink]['lat']
209             ↳ ), popup=f"SINK: Nodo {sink}"
210             ↳ , tooltip="Punto de Destino",
211             ↳ icon=icon_sink).add_to(m)
212
213         path_group = folium.FeatureGroup(name
214             ↳ = 'Camino Más Corto (Dijkstra)'
215             ↳ , show=True)
216         puntos = [(graph.vs[nid]['lat'],
217             ↳ graph.vs[nid]['lon']) for nid
218             ↳ in camino if graph.vs[nid]['
219             ↳ lat'] is not None]
220
221         folium.PolyLine(
222             ↳ puntos,
223             ↳ color='#1f77b4',
224             ↳ weight=5,
225             ↳ opacity=0.9,
226             ↳ tooltip=f"Camino más corto: {coste
227             ↳ :.2f} km"
228             ↳ ).add_to(path_group)
229
230         path_group.add_to(m)
231         return m
232     except Exception as e:
233         logging.error(f"No se pudo crear el
234             ↳ mapa base. Error: {e}")
235         return None
236
237 ### MODIFICADO ###
238 def agregar_caminos_random_al_mapa(m:
239     ↳ folium.Map, graph: ig.Graph,
240     ↳ num_caminos: int) -> List[Dict[str,
241     ↳ Any]]:
242     """Genera caminos aleatorios (usando
243         ↳ Dijkstra de igraph) y los agrega
244         ↳ al mapa para contexto. Retorna
245         ↳ los caminos generados."""
246
247     logging.info(f"Generando {num_caminos}
248         ↳ caminos aleatorios para dar
249         ↳ contexto a la red...")
250     caminos_generados = []
251     max_node_id = graph.vcount() - 1
252     iterator = tqdm(range(num_caminos), desc
253         ↳ = "Generando caminos aleatorios")
254
255     for _ in iterator:
256         for _ in range(10): # Intentos para
257             ↳ encontrar un camino válido
258             u, v = random.randint(0,
259                 ↳ max_node_id), random.
260                 ↳ randint(0, max_node_id)
261             if u == v or graph.vs[u]['lat'] is
262                 ↳ None or graph.vs[v]['lat']
263                 ↳ is None: continue
264
265             camino_nodos = graph.
266                 ↳ get_shortest_paths(u, to=v,
267                 ↳ weights='weight', output='
268                 ↳ vpath')
269
270             if camino_nodos and camino_nodos
271                 ↳ [0]:

```

```

202     camino = camino_nodos[0]
203     if all(graph.vs[nid]['lat'] is
204         ↪ not None for nid in
205         ↪ camino):
206         coste = sum(graph.es[graph.
207             ↪ get_eid(camino[i],
208             ↪ camino[i+1]])['weight
209             ↪ ' for i in range(len
210             ↪ (camino)-1))
211         if coste > 0:
212             caminos_generados.append
213                 ↪ ({'path': camino,
214                 ↪ 'length_km': coste
215                 ↪ })
216             break
217
218     if not caminos_generados:
219         logging.warning("No se pudo generar
220             ↪ ningún camino aleatorio válido
221             ↪ con coordenadas completas.")
222         return []
223
224     camino_mas_corto_muestra = min(
225         ↪ caminos_generados, key=lambda x:
226         ↪ x['length_km'])
227     camino_mas_largo_muestra = max(
228         ↪ caminos_generados, key=lambda x:
229         ↪ x['length_km'])
230
231     random_group = folium.FeatureGroup(name=
232         ↪ f'{num_caminos} Caminos
233         ↪ Aleatorios (Contexto)', show=
234         ↪ False)
235
236     for camino_info in caminos_generados:
237         puntos = [(graph.vs[nid]['lat'],
238             ↪ graph.vs[nid]['lon']) for nid
239             ↪ in camino_info['path']]
240         if camino_info ==
241             ↪ camino_mas_corto_muestra:
242             color, weight, opacity, tooltip =
243                 ↪ '#d62728', 4, 1.0, f"Más
244                 ↪ corto de la muestra ({
245                 ↪ camino_info['length_km']:.2
246                 ↪ f) km)"
247         elif camino_info ==
248             ↪ camino_mas_largo_muestra:
249             color, weight, opacity, tooltip =
250                 ↪ '#2ca02c', 4, 1.0, f"Más
251                 ↪ largo de la muestra ({
252                 ↪ camino_info['length_km']:.2
253                 ↪ f) km)"
254         else:
255             color, weight, opacity, tooltip =
256                 ↪ '#555555', 1, 0.5, f"Camino
257                 ↪ de {camino_info['length_km']
258                 ↪ :.2f} km"
259
260         folium.PolyLine(puntos, color=color,
261             ↪ weight=weight, opacity=opacity
262             ↪ , tooltip=tooltip).add_to(
263             ↪ random_group)
264
265     random_group.add_to(m)
266     return caminos_generados
267
268 def agregar_capa_nodos_de_caminos(m: folium
269     ↪ .Map, graph: ig.Graph,
270     ↪ camino_principal: List[int],
271     ↪ caminos_random: List[Dict[str, Any
272     ↪ ]]):
273     """Agrega una capa al mapa con todos los
274         ↪ nodos que pertenecen a los
275         ↪ caminos visualizados."""
276
277     logging.info("Agregando capa de nodos
278         ↪ que pertenecen a las rutas
279         ↪ calculadas...")
280
281     nodos_en_rutas: Set[int] = set(
282         ↪ camino_principal)
283     for camino_info in caminos_random:
284         nodos_en_rutas.update(camino_info['
285             ↪ path'])
286
287     logging.info(f"Se visualizarán {len(
288         ↪ nodos_en_rutas)} nodos únicos en
289         ↪ la nueva capa.")
290
291     nodos_group = folium.FeatureGroup(name='
292         ↪ Nodos en Rutas', show=False)
293
294     for nodo_id in nodos_en_rutas:
295         v = graph.vs[nodo_id]
296         if v['lat'] is not None and v['lon']
297             ↪ is not None:
298             folium.CircleMarker(
299                 location=[v['lat'], v['lon']],
300                 radius=3,
301                 color='#800080', # Púrpura
302                 fill=True,
303                 fill_color='#800080',
304                 fill_opacity=0.7,
305                 tooltip=f"Nodo {v.index}"
306             ).add_to(nodos_group)
307
308     nodos_group.add_to(m)
309
310 # --- Bloque Principal de Ejecución ---
311 if __name__ == "__main__":
312     GRAFO_PKL_ENTRADA = '
313         ↪ grafo_igraph_paralelizado.pkl'
314     MAPA_HTML_SALIDA = '
315         ↪ analisis_de_red_dijkstra.html'
316     # Nota: El código original calcula 1000
317         ↪ caminos, no 100.
318     NUM_CAMINOS_RANDOM = 100
319
320     mi_grafo = cargar_grafo(
321         ↪ GRAFO_PKL_ENTRADA)
322
323     if mi_grafo:
324         preparar_pesos_geograficos(mi_grafo,
325             ↪ force_recalc=False)
326         par_distante =
327             ↪ encontrar_nodos_mas_distantes_aprox
328             ↪ (mi_grafo)
329
330         if par_distante:
331             SOURCE_NODE, SINK_NODE =
332                 ↪ par_distante
333
334             camino_optimo, coste_total,
335                 ↪ tiempo_dijkstra =
336                 ↪ encontrar_camino_mas_corto_dijkstra
337                 ↪ (mi_grafo, SOURCE_NODE,
338                 ↪ SINK_NODE)
339
340             if camino_optimo and coste_total
341                 ↪ is not None:
342                 mapa_resultado =
343                     ↪ crear_mapa_camino_corto(
344                     ↪ mi_grafo, camino_optimo,
345                     ↪ coste_total,
346                     ↪ SOURCE_NODE, SINK_NODE)
347
348             if mapa_resultado:
349                 caminos_aleatorios_generados
350                     ↪ =
351                     ↪ agregar_caminos_random_al_mapa
352                     ↪ (mapa_resultado,
353                     ↪ mi_grafo,
354                     ↪ NUM_CAMINOS_RANDOM)

```

```

284     promedio_distancia_random =
285         ↪ 0.0
286     num_caminos_generados = len(
287         ↪ caminos_aleatorios_generados
288         ↪ )
289
290     if num_caminos_generados >
291         ↪ 0:
292         distancias_random = [
293             ↪ camino['length_km']
294             ↪ ] for camino in
295             ↪ caminos_aleatorios_generados
296             ↪ ]
297         promedio_distancia_random
298         ↪ = sum(
299         ↪ distancias_random)
300         ↪ /
301         ↪ num_caminos_generados
302
303     agregar_capa_nodos_de_caminos
304         ↪ (mapa_resultado,
305         ↪ mi_grafo,
306         ↪ camino_optimo,
307         ↪ caminos_aleatorios_generados
308         ↪ )
309
310     folium.LayerControl().add_to
311         ↪ (mapa_resultado)
312     mapa_resultado.save(
313         ↪ MAPA_HTML_SALIDA)
314
315     print("\n" + "="*60)
316     print(" ANÁLISIS DE RUTA Ó
317         ↪ PTIMA (DIJKSTRA)
318         ↪ COMPLETADO")
319     print("="*60)
320     print("Algoritmo utilizado:
321         ↪ Implementación manual
322         ↪ de Dijkstra en
323         ↪ Python")
324     print(f"Rango de análisis (
325         ↪ automático): Nodos {
326         ↪ SOURCE_NODE} a {
327         ↪ SINK_NODE}")
328     print(f"Distancia del Camino
329         ↪ Más Corto: {
330         ↪ coste_total:.2f} km")
331     print(f"Número de saltos (
332         ↪ nodos) en el camino:
333         ↪ {len(camino_optimo)}")
334     print(f"Tiempo de cálculo (
335         ↪ Dijkstra): {
336         ↪ tiempo_dijkstra:.4f}
337         ↪ segundos")
338
339     ### AÑADIDO: MOSTRAR EL
340         ↪ RESULTADO DEL
341         ↪ PROMEDIO ###
342     if promedio_distancia_random
343         ↪ > 0:
344         print("-" * 20)
345         print("Análisis de la
346             ↪ muestra de caminos
347             ↪ aleatorios:")
348         print(f"Distancia
349             ↪ promedio de los {
350             ↪ num_caminos_generados
351             ↪ } caminos de
352             ↪ muestra: {
353             ↪ promedio_distancia_random
354             ↪ :.2f} km")
355
356     ### FIN DEL CÓDIGO AÑADIDO
357         ↪ ###
358
359     print(f"\nSe ha generado un
360         ↪ mapa interactivo en:
361         ↪ '{MAPA_HTML_SALIDA}'")
362         ↪ )
363     print("El mapa contiene las
364         ↪ siguientes capas (usa
365         ↪ el control de capas)
366         ↪ :")
367     print("- [Visible] Camino M
368         ↪ ás Corto: La ruta ó
369         ↪ ptima encontrada.")
370     print("- [Oculta] Caminos
371         ↪ Aleatorios: Una
372         ↪ muestra para contexto
373         ↪ de la red.")
374     print("-> El más largo (
375         ↪ distancia) de la
376         ↪ muestra se muestra en
377         ↪ VERDE.")
378     print("-> El más corto (
379         ↪ distancia) de la
380         ↪ muestra se muestra en
381         ↪ ROJO.")
382     print("- [Oculta] Nodos en
383         ↪ Rutas: Muestra todos
384         ↪ los nodos únicos que
385         ↪ forman parte de los
386         ↪ caminos visualizados.
387         ↪ ")
388     print("="*60)
389     else:
390         logging.error("No se pudo
391             ↪ crear el objeto de
392             ↪ mapa base. Abortando
393             ↪ visualización.")
394
395     else:
396         logging.error(f"No se encontró
397             ↪ un camino entre el nodo
398             ↪ {SOURCE_NODE} y el {
399             ↪ SINK_NODE}.")
400
401     else:
402         logging.error("No se pudo
403             ↪ determinar el par de nodos
404             ↪ de origen/destino.
405             ↪ Abortando análisis.")

```

Listing 4. Código para el análisis 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 recalculation, 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. Código de *communityLouvain.py*

```

1 import pickle
2 import time
3 import logging
4 import random
5 from collections import Counter,
    ↳ defaultdict
6 import igraph as ig
7 from typing import Optional, List, Dict,
    ↳ Tuple
8 import folium
9 from folium.plugins import MarkerCluster
10 import matplotlib
11 import matplotlib.colors as colors
12 from tqdm import tqdm
13
14 # --- Configuración del Logging ---
15 logging.basicConfig(level=logging.INFO,
    ↳ format='%(asctime)s - %(levelname)s
    ↳ - %(message)s',
16 handlers=[logging.FileHandler("
    ↳ analisis_comunidades_louvain_manual.
    ↳ log", mode='w', encoding='utf-8'),
17 logging.StreamHandler()])
18
19 # --- UMBRALES DE VISUALIZACIÓN ---
20 UMBRAL_MAX_VISUALIZACION_ARISTAS = 3000
21 UMBRAL_MAX_NODOS_A_DIBUJAR = 3000
22
23 # --- Función de Carga de Grafo ---
24 def cargar_grafo(grafo_path: str) ->
    ↳ Optional[ig.Graph]:
25     logging.info(f"Cargando el grafo desde
    ↳ '{grafo_path}'..."); start_time =
    ↳ time.time()
26     try:
27         with open(grafo_path, 'rb') as f: g =
    ↳ pickle.load(f)
28         end_time = time.time(); logging.info(
    ↳ f"Grafo cargado en {end_time -
    ↳ start_time:.2f} segundos. {g.
    ↳ summary()}")
29         return g
30     except Exception as e:
31         logging.error(f"Error crítico al
    ↳ cargar el grafo: {e}"); return
    ↳ None
32
33 # --- INICIO DE LA IMPLEMENTACIÓN MANUAL DE
    ↳ LOUVAIN ---
34
35 def _calcular_delta_modularity(node: int,
    ↳ target_community: int, node_degree:
    ↳ float,
36                                 links_to_target_community
    ↳ : float,
    ↳ total_community_degree
    ↳ : float,
    ↳ total_edge_weight:
    ↳ float) ->
    ↳ float:
37     if total_edge_weight == 0: return 0
38     term1 = links_to_target_community /
    ↳ total_edge_weight
39     term2 = (total_community_degree *
    ↳ node_degree) / (2 * (
    ↳ total_edge_weight ** 2))
40     return term1 - term2
41
42

```



```

43 def _louvain_phase1(graph: ig.Graph,
    ↳ total_edge_weight: float) -> Tuple[
    ↳ Dict[int, int], bool]:
44     """
45     Fase 1: Optimización de modularidad
    ↳ local.
46     Devuelve la partición y un booleano que
    ↳ indica si hubo cambios.
47     """
48     communities = {v.index: v.index for v in
    ↳ graph.vs}
49
50     # 1. Obtenemos la lista de todas las
    ↳ fuerzas de los nodos.
51     all_strengths = graph.strength(weights='
    ↳ weight' if 'weight' in graph.es.
    ↳ attributes() else None)
52     # 2. Creamos el diccionario mapeando el
    ↳ índice del nodo a su fuerza.
53     node_degrees = {i: strength for i,
    ↳ strength in enumerate(
    ↳ all_strengths)}
54     # -----
55
56     community_degrees = node_degrees.copy()
57
58     improved = False
59     nodes = list(range(graph.vcount()))
60     random.shuffle(nodes)
61
62     for node in tqdm(nodes, desc=" Fase 1:
    ↳ Optimizando modularidad"):
63         current_community = communities[node]
64         best_community = current_community
65         max_gain = 0.0
66
67         links_to_communities = defaultdict(
    ↳ float)
68         for neighbor in graph.neighbors(node,
    ↳ mode='all'):
69             edge_id = graph.get_eid(node,
    ↳ neighbor, directed=False,
    ↳ error=False)
70             if edge_id != -1:
71                 weight = graph.es[edge_id]['
    ↳ weight' if 'weight' in
    ↳ graph.es.attributes()
    ↳ else 1.0
72                 links_to_communities[
    ↳ communities[neighbor]]
    ↳ += weight
73
74         for community, links_weight in
    ↳ links_to_communities.items():
75             if community == current_community:
    ↳ continue
76             gain = _calcular_delta_modularity(
    ↳ node, community, node_degrees[
    ↳ node], links_weight,
    ↳ community_degrees[community],
    ↳ total_edge_weight
77             )
78             if gain > max_gain:
    ↳ max_gain = gain
    ↳ best_community = community
79
80         if max_gain > 0:
81             community_degrees[
    ↳ current_community] -=
    ↳ node_degrees[node]
82             community_degrees[best_community]
    ↳ += node_degrees[node]
83             communities[node] = best_community
84             improved = True
85
86     return communities, improved
91
92 def _louvain_phase2(graph: ig.Graph,
    ↳ communities: Dict[int, int]) -> ig.
    ↳ Graph:
93     """
94     Fase 2: Agregación de comunidades.
    ↳ Construye un nuevo grafo.
95     """
96     new_nodes = list(set(communities.values
    ↳ ()))
97     new_node_map = {old_id: new_id for
    ↳ new_id, old_id in enumerate(
    ↳ new_nodes)}
98
99     new_graph = ig.Graph(directed=False)
100     new_graph.add_vertices(len(new_nodes))
101
102     edge_weights = defaultdict(float)
103
104     for edge in graph.es:
105         source, target = edge.tuple
106         c_source = communities[source]
107         c_target = communities[target]
108         weight = edge['weight' if 'weight'
    ↳ in graph.es.attributes() else
    ↳ 1.0
109
110         if c_source != c_target:
111             new_source = new_node_map[c_source
    ↳ ]
112             new_target = new_node_map[c_target
    ↳ ]
113
114             # Ordenar para evitar duplicados
    ↳ en grafo no dirigido
115             if new_source < new_target:
116                 edge_weights[(new_source,
    ↳ new_target)] += weight
117             else:
118                 edge_weights[(new_target,
    ↳ new_source)] += weight
119
120     new_graph.add_edges(edge_weights.keys())
121     new_graph.es['weight'] = list(
    ↳ edge_weights.values())
122
123     return new_graph
124
125 def detectar_comunidades_louvain_manual(
    ↳ graph: ig.Graph) -> Dict[int, List[
    ↳ int]]:
126     """
127     Implementación manual completa del
    ↳ algoritmo de Louvain.
128     """
129     logging.info("Iniciando detección de
    ↳ comunidades con implementación
    ↳ MANUAL de Louvain...")
130     start_time = time.time()
131
132     # Crucial: El algoritmo de Louvain
    ↳ trabaja sobre grafos no dirigidos
    ↳ .
133     # Esta conversión es necesaria y
    ↳ soluciona el 'ValueError'.
134     if graph.is_directed():
135         logging.info("El grafo es dirigido.
    ↳ Convirtiendo a no dirigido
    ↳ para el análisis de
    ↳ comunidades.")
136         # Usamos 'as_undirected' que es más
    ↳ seguro que modificar el grafo
    ↳ in-place.
137         current_graph = graph.as_undirected(
    ↳ mode='collapse', combine_edges
    ↳ =dict(weight="sum"))
138     else:
139         current_graph = graph.copy()

```

```

140 # La membresía inicial mapea cada nodo
141     ↳ del grafo ORIGINAL a su comunidad
142     ↳ actual.
143 membership = {v.index: v.index for v in
144     ↳ graph.vs}
145
146 pass_num = 1
147 while True:
148     logging.info(f"--- Louvain - Pass {
149     ↳ pass_num} ---")
150
151     total_edge_weight = sum(current_graph
152     ↳ .es['weight']) if 'weight' in
153     ↳ current_graph.es.attributes()
154     ↳ else current_graph.ecount()
155 if total_edge_weight == 0:
156     logging.warning("El grafo no tiene
157     ↳ aristas. Deteniendo el
158     ↳ algoritmo.")
159     break
160
161 communities, improved =
162     ↳ _louvain_phasel(current_graph,
163     ↳ total_edge_weight)
164
165 if not improved:
166     logging.info("No hubo más mejoras
167     ↳ de modularidad.
168     ↳ Convergencia alcanzada.")
169     break
170
171 # Mapear la nueva partición a la
172     ↳ membresía global.
173 # Si el nodo 5 va a la com. 10, y en
174     ↳ la siguiente pasada
175 # la com. 10 va a la super-com. 2, el
176     ↳ nodo 5 debe pertenecer a la
177     ↳ super-com. 2.
178 partition_map = communities
179 for node_idx in range(graph.vcount()):
180     ↳ :
181     membership[node_idx] =
182     ↳ partition_map.get(
183     ↳ membership[node_idx],
184     ↳ membership[node_idx])
185
186 logging.info(" Fase 2: Agregando
187     ↳ comunidades para la siguiente
188     ↳ pasada...")
189 current_graph = _louvain_phase2(
190     ↳ current_graph, communities)
191
192 if current_graph.vcount() == 0 or
193     ↳ current_graph.ecount() == 0:
194     logging.info("El grafo agregado ya
195     ↳ no tiene nodos o aristas.
196     ↳ Deteniendo.")
197     break
198
199 pass_num += 1
200
201 final_comunidades = defaultdict(list)
202 for node, comm in membership.items():
203     final_comunidades[comm].append(node)
204
205 end_time = time.time()
206 logging.info(f"Algoritmo manual de
207     ↳ Louvain completado en {end_time -
208     ↳ start_time:2f} s. Se
209     ↳ encontraron {len(
210     ↳ final_comunidades)} comunidades."
211     ↳ )
212 return dict(final_comunidades)
213
214 # --- Función de Análisis ---
215
216 def analizar_y_seleccionar_comunidades(
217     ↳ comunidades_dict: Dict[int, List[int]
218     ↳ ], num_random: int = 20) -> Dict[
219     ↳ str, List[int]]:
220     logging.info("Analizando y seleccionando
221     ↳ comunidades para visualización
222     ↳ ...");
223
224 if not comunidades_dict or len(
225     ↳ comunidades_dict) < 3: logging.
226     ↳ warning("No hay suficientes
227     ↳ comunidades para un análisis
228     ↳ detallado."); return {}
229
230 comunidades_con_tamaño = [(cid, len(
231     ↳ miembros)) for cid, len(
232     ↳ comunidades_dict.items())
233 id_grande, tamaño_grande = max(
234     ↳ comunidades_con_tamaño, key=
235     ↳ lambda item: item[1])
236 com_mayores_a_uno = [c for c in
237     ↳ comunidades_con_tamaño if c[1] >
238     ↳ 1]
239 id_pequeña, tamaño_pequeña = min(
240     ↳ com_mayores_a_uno, key=lambda
241     ↳ item: item[1]) if
242     ↳ com_mayores_a_uno else min(
243     ↳ comunidades_con_tamaño, key=
244     ↳ lambda item: item[1])
245 ids_extremos = {id_grande, id_pequeña}
246 posibles_ids_random = [cid for cid, size
247     ↳ in comunidades_con_tamaño if cid
248     ↳ not in ids_extremos and 5 < size
249     ↳ <
250     ↳ UMBRAL_MAX_VISUALIZACION_ARISTAS]
251 ids_random = random.sample(
252     ↳ posibles_ids_random, min(
253     ↳ num_random, len(
254     ↳ posibles_ids_random)) if
255     ↳ posibles_ids_random else [])
256 seleccion = {'grande': [id_grande], '
257     ↳ pequena': [id_pequeña], 'random':
258     ↳ ids_random}
259 logging.info(f"Comunidad más grande (ID
260     ↳ {id_grande}): {tamaño_grande}
261     ↳ miembros.")
262 logging.info(f"Comunidad más pequeña (>1
263     ↳ miembro) (ID {id_pequeña}): {
264     ↳ tamaño_pequeña} miembros.")
265 logging.info(f"Se seleccionaron {len(
266     ↳ ids_random)} comunidades
267     ↳ aleatorias (tamaño < {
268     ↳ UMBRAL_MAX_VISUALIZACION_ARISTAS
269     ↳ }).")
270 return seleccion
271
272 # --- Función de Colores ---
273 def crear_mapa_de_colores(tamaño_min: int,
274     ↳ tamaño_max: int):
275     colormap = matplotlib.colormaps.get_cmap(
276     ↳ ('coolwarm'))
277 normalizador = colors.LogNorm(vmin=max
278     ↳ (1, tamaño_min), vmax=tamaño_max)
279 return lambda tamaño: colors.to_hex(
280     ↳ colormap(normalizador(tamaño)))
281
282 # --- Función de Visualización ---
283 def visualizar_comunidades(graph: ig.Graph,
284     ↳ comunidades_dict: Dict[int, List[
285     ↳ int]], seleccion: Dict[str, List[int]
286     ↳ ], output_filename: str):
287     logging.info(f"Creando mapa de
288     ↳ visualización en '{
289     ↳ output_filename}'...")
290 coords_validas = [(v['lat'], v['lon'])
291     ↳ for v in graph.vs if v['lat'] is
292     ↳ not None and v.index != 0]
293 if not coords_validas: logging.error("No
294     ↳ hay nodos con coordenadas para

```

```

211     ↪ 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)
212 m = folium.Map(location=[avg_lat,
    ↪ avg_lon], zoom_start=2, tiles="
    ↪ CartoDB positron")
213 tamaños = {cid: len(miembros) for cid,
    ↪ miembros in comunidades_dict.
    ↪ items() if miembros}
214 if not tamaños: logging.error("Las
    ↪ comunidades están vacías, no se
    ↪ puede generar el mapa."); return
215 mapa_color = crear_mapa_de_colores(min(
    ↪ tamaños.values()), max(tamaños.
    ↪ values()))
216 ids_a_visualizar = seleccion['grande'] +
    ↪ seleccion['pequena'] + seleccion
    ↪ ['random']
217 iterator = tqdm(ids_a_visualizar, desc="
    ↪ Creando capas de comunidades")
218
219 for com_id in iterator:
220     miembros_originales =
    ↪ comunidades_dict.get(com_id)
221     if not miembros_originales: continue
222
223     tamaño_original = len(
    ↪ miembros_originales)
224     color = mapa_color(tamaño_original)
225     show_layer = com_id in seleccion['
    ↪ grande'] or com_id in
    ↪ seleccion['pequena']
226     nombre_capa = f"Comunidad {com_id} ({
    ↪ tamaño_original} miembros)"
227     if com_id in seleccion['grande']:
    ↪ nombre_capa = f"Comunidad Más
    ↪ Grande ({tamaño_original})"
228     if com_id in seleccion['pequena']:
    ↪ nombre_capa = f"Comunidad Más
    ↪ Pequeña ({tamaño_original})"
229
230     container = MarkerCluster(name=
    ↪ nombre_capa, show=show_layer)
    ↪ if tamaño_original > 200 else
    ↪ folium.FeatureGroup(name=
    ↪ nombre_capa, show=show_layer)
231     container.add_to(m)
232
233     nodos_a_dibujar = miembros_originales
234     if tamaño_original >
    ↪ UMBRAL_MAX_NODOS_A_DIBUJAR:
235         logging.warning(f"Comunidad {
    ↪ com_id} ({tamaño_original}
    ↪ nodos) excede umbral.
    ↪ Mostrando muestra de {
    ↪ UMBRAL_MAX_NODOS_A_DIBUJAR
    ↪ }.")
236         nodos_a_dibujar = random.sample(
    ↪ miembros_originales,
    ↪ UMBRAL_MAX_NODOS_A_DIBUJAR)
237
238     nodos_visibles_con_coords = []
239     for nodo_id in nodos_a_dibujar:
240         v = graph.vs[nodo_id]
241         if v['lat'] is not None and v['lon']
    ↪ is not None:
242             nodos_visibles_con_coords.
    ↪ append(nodo_id)
243             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)
244
245     if tamaño_original <=
    ↪ UMBRAL_MAX_VISUALIZACION_ARISTAS
    ↪ :
246         if len(nodos_visibles_con_coords)
    ↪ > 1:
247             subgrafo_comunidad = graph.
    ↪ subgraph(
    ↪ nodos_visibles_con_coords
    ↪ )
248             num_aristas_internas = len(
    ↪ subgrafo_comunidad.es)
249             logging.info(f"Comunidad {
    ↪ com_id} (tamaño {tamañ
    ↪ o_original}): Intentando
    ↪ dibujar {
    ↪ num_aristas_internas}
    ↪ aristas internas.")
250
251         for arista in
    ↪ subgrafo_comunidad.es:
252             id_origen =
    ↪ nodos_visibles_con_coords
    ↪ [arista.source];
    ↪ id_destino =
    ↪ nodos_visibles_con_coords
    ↪ [arista.target]
253             v_origen = graph.vs[
    ↪ id_origen]; v_destino
    ↪ = graph.vs[
    ↪ id_destino]
254             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)
255     else:
256         logging.warning(f"Omitiendo dibujo
    ↪ de aristas para comunidad
    ↪ {com_id} (tamaño: {tamañ
    ↪ o_original} > {
    ↪ UMBRAL_MAX_VISUALIZACION_ARISTAS
    ↪ }).")
257
258     folium.LayerControl(collapsed=False).
    ↪ add_to(m)
259     logging.info("Guardando el mapa en el
    ↪ archivo HTML...")
260     m.save(output_filename)
261     logging.info(f"Mapa guardado
    ↪ correctamente en '{
    ↪ output_filename}'.")
262
263 # --- Bloque Principal ---
264 if __name__ == "__main__":
265     GRAFO_PKL_ENTRADA = '
    ↪ grafo_igraph_paralelizado.pkl'
266     MAPA_HTML_SALIDA = '
    ↪ analisis_comunidades_louvain_manual
    ↪ .html'
267     NUM_COMUNIDADES_RANDOM = 50
268
269     mi_grafo = cargar_grafo(
    ↪ GRAFO_PKL_ENTRADA)
270     if mi_grafo:
    ↪ coms_dict =
    ↪ detectar_comunidades_louvain_manual
    ↪ (mi_grafo)
271     if coms_dict:
    ↪ comunidades_seleccionadas =
    ↪ analizar_y_seleccionar_comunidades

```

```

275         ↪ (coms_dict,
276         ↪ NUM_COMUNIDADES_RANDOM)
if comunidades_seleccionadas:
    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. Código 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. Código de bfs.py

```

1
2 import igraph as ig
3 import folium
4 from folium.plugins import BeautifyIcon,
5     ↪ AntPath
6 import pickle
7 import time
8 import logging
9 import os
10 import webbrowser
11 from collections import deque
12
13 # --- Configuración ---
14 GRAFO_PICKLE_PATH = '
15     ↪ grafo_igraph_paralelizado.pkl'
16 MAPA_HTML_OUTPUT = '
17     ↪ mapa_camino_largo_con_arbol.html' #
18     ↪ Nuevo nombre de archivo
19
20 # Nodo de inicio
21 NODO_INICIO = 9999427
22
23 # Define la cantidad máxima de nodos a
24     ↪ explorar.
25 MAX_NODOS_A_EXPLORAR = 5000
26
27 # --- Configuración del Logging ---
28 logging.basicConfig(
29     level=logging.INFO,
30     format='%(asctime)s - %(levelname)s -
31         ↪ %(message)s',
32     handlers=[
33         logging.FileHandler(" analisis_bfs.log
34             ↪ ", mode='w', encoding='utf-8')
35         ↪ ,
36         logging.StreamHandler()
37     ]
38 )
39
40 def visualizar_arbol_bfs(graph,
41     ↪ start_node_id, max_nodes,
42     ↪ output_filename):
43     """
44     Realiza una búsqueda BFS, resalta el
45     ↪ camino más largo con un AntPath
46     ↪ rojo
47     y muestra el árbol de búsqueda completo
48     ↪ con líneas estáticas.
49     """
50     logging.info(f"Iniciando BFS desde el
51         ↪ nodo {start_node_id} con un lí
52         ↪ mite de {max_nodes} nodos.")
53     start_time = time.time()
54
55     # 1. Realizar BFS para recolectar nodos,
56     ↪ niveles y la estructura del á
57     ↪ rbol
58
59     queue = deque([(start_node_id, 0)])
60     visited = {start_node_id}
61     levels = {start_node_id: 0}
62     parent_map = {start_node_id: None}
63     tree_edges = []
64
65     while queue and len(visited) < max_nodes:
66         ↪ :
67         current_node, current_level = queue.
68             ↪ popleft()
69         neighbors = graph.neighbors(
70             ↪ current_node, mode="out")
71
72         for neighbor in neighbors:
73             if neighbor not in visited:
74                 if len(visited) >= max_nodes:
75                     ↪ break
76                 visited.add(neighbor)
77                 levels[neighbor] =
78                     ↪ current_level + 1
79                 parent_map[neighbor] =
80                     ↪ current_node
81                 tree_edges.append((current_node
82                     ↪ , neighbor))
83                 queue.append((neighbor,
84                     ↪ current_level + 1))
85
86     logging.info(f"BFS completado. Se
87         ↪ exploraron {len(visited)} nodos y
88         ↪ {len(tree_edges)} aristas en {
89         ↪ time.time() - start_time:.2f}
90         ↪ segundos.")
91
92     # 2. Encontrar el nodo más lejano en el
93     ↪ árbol BFS
94     farthest_node = max(visited, key=lambda
95         ↪ node: levels[node])
96     max_level = levels[farthest_node]
97     logging.info(f"El camino más largo
98         ↪ encontrado tiene {max_level}
99         ↪ saltos y termina en el nodo {
100         ↪ farthest_node}.")
101
102     # 3. Preparar el mapa con Folium
103     logging.info("Creando mapa con Folium...
104         ↪ ")
105     lats = [graph.vs[n]['lat'] for n in
106         ↪ visited]
107     lons = [graph.vs[n]['lon'] for n in
108         ↪ visited]
109     map_center = [sum(lats) / len(lats), sum
110         ↪ (lons) / len(lons)]
111
112     m = folium.Map(location=map_center,
113         ↪ zoom_start=10, tiles="CartoDB
114         ↪ positron")
115
116     # 4. Crear capas (FeatureGroups) para
117     ↪ cada elemento visual
118     nodos_explorados_group = folium.
119         ↪ FeatureGroup(name=f"Nodos
120         ↪ Explorados ({len(visited)})",
121         ↪ show=True)
122     aristas_estaticas_group = folium.
123         ↪ FeatureGroup(name=f"Árbol BFS Est
124         ↪ ático ({len(tree_edges)} aristas)
125         ↪ ", show=True)
126     camino_animado_group = folium.
127         ↪ FeatureGroup(name="Camino Más
128         ↪ Largo (Animado)", show=True)
129
130     # 5. Añadir los nodos explorados al mapa
131     colors = ['#1f77b4', '#ff7f0e', '#2ca02c
132         ↪ ', '#9467bd', '#8c564b', '#e377c2
133         ↪ ', '#17becf']
134     for node_id in visited:

```

```

84     lat, lon = graph.vs[node_id]['lat'],
      ↪ graph.vs[node_id]['lon']
85     level = levels[node_id]
86     popup_text = f"<b>Nodo:</b> {node_id}
      ↪ }<br><b>Nivel:</b> {level}"
87
88     folium.CircleMarker(
89         location=[lat, lon], radius=3,
      ↪ color=colors[level % len(
      ↪ colors)],
90         fill=True, fill_color=colors[level
      ↪ % len(colors)],
      ↪ fill_opacity=0.6, popup=
      ↪ popup_text
91     ).add_to(nodos_explorados_group)
92
93     # --- NUEVO: 6. Añadir las aristas está
      ↪ ticas del árbol BFS completo ---
94     for parent, child in tree_edges:
95         parent_coords = (graph.vs[parent]['
      ↪ lat'], graph.vs[parent]['lon'
      ↪ ])
96         child_coords = (graph.vs[child]['lat'
      ↪ ], graph.vs[child]['lon'])
97
98         folium.PolyLine(
99             locations=[parent_coords,
      ↪ child_coords],
100             color='#AAAAAA', # Un color gris
      ↪ claro para que no domine
101             weight=1, # Línea delgada
      ↪ opacity=0.7
102         ).add_to(aristas_estaticas_group)
103
104     # 7. Reconstruir y añadir el camino más
      ↪ largo (rojo y animado)
105     longest_path_coords = []
106     curr = farthest_node
107     while curr is not None:
108         coords = (graph.vs[curr]['lat'],
      ↪ graph.vs[curr]['lon'])
109         longest_path_coords.append(coords)
110         curr = parent_map.get(curr)
111         longest_path_coords.reverse()
112
113     AntPath(
114         locations=longest_path_coords,
115         delay=1000, weight=5, color='#d62728'
      ↪ , # Rojo y grueso
116         pulse_color='#FFFFFF', dash_array
      ↪ =[25, 40]
117     ).add_to(camino_animado_group)
118
119     # 8. Añadir marcadores de INICIO y FIN
      ↪ sobre todas las capas
120     start_lat, start_lon = graph.vs[
      ↪ start_node_id]['lat'], graph.vs[
      ↪ start_node_id]['lon']
121     folium.Marker(
122         location=[start_lat, start_lon],
      ↪ popup=f"<b>INICIO:</b> {
      ↪ start_node_id}",
123         tooltip=f"INICIO: Nodo {start_node_id}
      ↪ }",
124         icon=BeautifulIcon(icon='play',
      ↪ icon_shape='circle',
      ↪ border_color='#2ca02c',
      ↪ text_color='#2ca02c',
      ↪ background_color='#FFF')
125     ).add_to(m)
126
127     end_lat, end_lon = graph.vs[
      ↪ farthest_node]['lat'], graph.vs[
      ↪ farthest_node]['lon']
128     folium.Marker(
129         location=[end_lat, end_lon], popup=f"
      ↪ <b>FIN:</b> {farthest_node}<br>

```

```

      ↪ <b>Nivel:</b> {max_level}",
131     tooltip=f"FIN: Nodo {farthest_node}",
132     icon=BeautifulIcon(icon='stop',
      ↪ icon_shape='circle',
      ↪ border_color='#d62728',
      ↪ text_color='#d62728',
      ↪ background_color='#FFF')
133 ).add_to(m)
134
135     # 9. Añadir todas las capas y el control
      ↪ al mapa
136     nodos_explorados_group.add_to(m)
137     aristas_estaticas_group.add_to(m) # Añ
      ↪ adimos la capa de aristas está
      ↪ ticas
138     camino_animado_group.add_to(m)
139     folium.LayerControl().add_to(m)
140
141     # 10. Guardar y abrir el mapa
142     m.save(output_filename)
143     logging.info(f"Mapa guardado en '{
      ↪ output_filename}'")
144     webbrowser.open('file://' + os.path.
      ↪ realpath(output_filename))
145
146 if __name__ == "__main__":
147     if not os.path.exists(GRAFO_PICKLE_PATH)
      ↪ :
148         logging.error(f"El archivo del grafo
      ↪ '{GRAFO_PICKLE_PATH}' no fue
      ↪ encontrado.")
149     else:
150         logging.info(f"Cargando el grafo
      ↪ desde '{GRAFO_PICKLE_PATH}'...
      ↪ ")
151         start_load_time = time.time()
152         with open(GRAFO_PICKLE_PATH, 'rb') as
      ↪ f:
153             g = pickle.load(f)
154             logging.info(f"Grafo cargado en {time
      ↪ .time() - start_load_time:.2f}
      ↪ segundos.")
155
156         if not (0 <= NODO_INICIO < g.vcount()
      ↪ ):
157             logging.error(f"El NODO_INICIO ({
      ↪ NODO_INICIO}) está fuera
      ↪ del rango.")
158             NODO_INICIO = random.randint(0, g.
      ↪ vcount() - 1)
159             logging.warning(f"Se ha
      ↪ seleccionado un nuevo nodo
      ↪ de inicio aleatorio: {
      ↪ NODO_INICIO}")
160
161         visualizar_arbol_bfs(g, NODO_INICIO,
      ↪ MAX_NODOS_A_EXPLORAR,
      ↪ MAPA_HTML_OUTPUT)

```

Listing 6. Código 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 eje-

cución, 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.

V-F. Código de PrimBim.py

```

1         import pickle
2     import logging
3     import igraph as ig
4     from typing import Optional, Tuple, List
5     import polars as pl
6     import graphistry
7     import random
8     import time
9     import webbrowser
10    from tqdm import tqdm
11    from math import radians, sin, cos, sqrt,
12        ↪ atan2
13
14    # --- PARÁMETROS DE CONFIGURACIÓN ---
15    MAX_NODOS_PARA_ANALISIS = 200000
16    MAX_ARISTAS_MST_PARA_GRAPHISTORY = 75000
17    GRAPHISTORY_USERNAME = "MiquelAngFl"

```

```

17    GRAPHISTORY_PASSWORD = "sxtXy3ZG.DAhdsG"
18
19    # --- Configuración del Logging ---
20    logging.basicConfig(
21        level=logging.INFO, format='%(asctime)s
22        ↪ - %(levelname)s - %(message)s',
23        handlers=[logging.FileHandler("
24        ↪ analisis_prim_graphistry.log",
25        ↪ mode='w', encoding='utf-8'),
26        ↪ logging.StreamHandler()]
27    )
28
29    # --- Implementación Manual del Heap
30    ↪ Binario (Min-Heap) ---
31
32    class MinHeap:
33    def __init__(self): self.heap, self.
34    ↪ position_map = [], {}
35
36    def is_empty(self) -> bool: return len(
37    ↪ self.heap) == 0
38
39    def _parent(self, i: int) -> int: return
40    ↪ (i - 1) // 2
41
42    def _left_child(self, i: int) -> int:
43    ↪ return 2 * i + 1
44
45    def _right_child(self, i: int) -> int:
46    ↪ return 2 * i + 2
47
48    def _swap(self, i: int, j: int):
49    self.position_map[self.heap[i][1]],
50    ↪ self.position_map[self.heap[j]
51    ↪ ][1]] = j, i
52    self.heap[i], self.heap[j] = self.
53    ↪ heap[j], self.heap[i]
54
55    def _bubble_up(self, i: int):
56    parent_idx = self._parent(i)
57    while i > 0 and self.heap[i][0] <
58    ↪ self.heap[parent_idx][0]:
59    self._swap(i, parent_idx); i =
60    ↪ parent_idx; parent_idx =
61    ↪ self._parent(i)
62
63    def _bubble_down(self, i: int):
64    min_index = i
65    while True:
66    left_idx, right_idx = self.
67    ↪ _left_child(i), self.
68    ↪ _right_child(i)
69
70    if left_idx < len(self.heap) and
71    ↪ self.heap[left_idx][0] <
72    ↪ self.heap[min_index][0]:
73    min_index = left_idx
74
75    if right_idx < len(self.heap) and
76    ↪ self.heap[right_idx][0] <
77    ↪ self.heap[min_index][0]:
78    min_index = right_idx
79
80    if i == min_index: break
81    self._swap(i, min_index); i =
82    ↪ min_index
83
84    def push(self, weight: float, node_id:
85    ↪ int):
86    self.heap.append((weight, node_id));
87    ↪ self.position_map[node_id] =
88    ↪ len(self.heap) - 1
89    self._bubble_up(len(self.heap) - 1)
90
91    def pop(self) -> Tuple[float, int]:
92    if self.is_empty(): raise IndexError(
93    ↪ "pop from an empty heap")
94    min_element = self.heap[0];
95    ↪ last_element = self.heap.pop()
96    ↪ ; del self.position_map[
97    ↪ min_element[1]]
98
99    if not self.is_empty():
100    self.heap[0] = last_element; self.
101    ↪ position_map[last_element
102    ↪ ][1]] = 0
103    self._bubble_down(0)
104
105    return min_element
106
107    def update(self, node_id: int,
108    ↪ new_weight: float):

```



```

58     if node_id not in self.position_map:
59         ↪ return
60     idx = self.position_map[node_id]
61     if new_weight < self.heap[idx][0]:
62         self.heap[idx] = (new_weight,
63             ↪ node_id); self._bubble_up(
64             ↪ idx)
65
66 # --- Funciones de Carga y Preparación ---
67 def cargar_grafo(grafo_path: str) ->
68     ↪ Optional[ig.Graph]:
69     logging.info(f"Cargando el grafo desde
70         ↪ '{grafo_path}'...")
71     start_time = time.time();
72     try:
73         with open(grafo_path, 'rb') as f: g =
74             ↪ pickle.load(f)
75         logging.info(f"Grafo cargado en {time
76             ↪ .time() - start_time:.2f}
77             ↪ segundos."); return g
78     except Exception as e:
79         logging.error(f"Error al cargar el
80             ↪ grafo: {e}"); return None
81
82 def haversine_distance(lat1: float, lon1:
83     ↪ float, lat2: float, lon2: float) ->
84     ↪ float:
85     R = 6371; dlat, dlon = radians(lat2 -
86         ↪ lat1), radians(lon2 - lon1)
87     a = sin(dlat / 2)**2 + cos(radians(lat1)
88         ↪ ) * cos(radians(lat2)) * sin(dlon
89         ↪ / 2)**2
90     a = min(1.0, a); c = 2 * atan2(sqrt(a),
91         ↪ sqrt(1 - a)); return R * c
92
93 def preparar_pesos_geograficos(graph: ig.
94     ↪ Graph):
95     if 'weight' in graph.es.attributes():
96         ↪ return
97     logging.info("Calculando pesos geográficos
98         ↪ (Haversine)...")
99     pesos = []
100     for edge in tqdm(graph.es, desc="
101         ↪ Calculando pesos"):
102         source_v = graph.vs[edge.source];
103         ↪ target_v = graph.vs[edge.
104         ↪ target]
105         if source_v['lat'] is not None and
106             ↪ target_v['lat'] is not None:
107             pesos.append(haversine_distance(
108                 ↪ source_v['lat'], source_v['
109                 ↪ lon'], target_v['lat'],
110                 ↪ target_v['lon']))
111         else:
112             pesos.append(float('inf'))
113     graph.es['weight'] = pesos
114     logging.info("Pesos geográficos
115         ↪ asignados.")
116
117 def encontrar_nodo_inicio_valido(graph: ig.
118     ↪ Graph, nodes_in_component: List[int
119     ↪ ]) -> Optional[int]:
120     random.shuffle(nodes_in_component)
121     for node_id in nodes_in_component:
122         v = graph.vs[node_id]
123         if v['lat'] is not None and graph.
124             ↪ degree(node_id) > 0:
125             return node_id
126     return None
127
128 def prim_mst_forest_manual(graph: ig.Graph)
129     ↪ -> Tuple[List[Tuple[int, int]],
130     ↪ float]:
131     logging.info("Calculando el Bosque de
132         ↪ Expansión Mínima con Prim manual
133         ↪ ...")
134     start_time = time.time()

```

```

102     parent, in_mst = {v.index: None for v in
103         ↪ graph.vs}, {v.index: False for v
104         ↪ in graph.vs}
105     forest_edges, total_forest_weight = [],
106         ↪ 0
107     components = graph.components(mode='weak
108         ↪ ')
109     logging.info(f"Grafo tiene {len(
110         ↪ components)} componentes.
111         ↪ Calculando un MST para cada uno."
112         ↪ )
113     for component_nodes in tqdm(components,
114         ↪ desc="Procesando Componentes"):
115         if len(component_nodes) <= 1:
116             ↪ continue
117         start_node =
118             ↪ encontrar_nodo_inicio_valido(
119             ↪ graph, component_nodes)
120         if start_node is None: continue
121         key = {node: float('inf') for node in
122             ↪ component_nodes}
123         key[start_node] = 0
124         pq = MinHeap()
125         for node in component_nodes: pq.push(
126             ↪ key[node], node)
127         while not pq.is_empty():
128             weight, u = pq.pop()
129             if in_mst[u]: continue
130             in_mst[u] = True
131             if weight != float('inf'):
132                 ↪ total_forest_weight +=
133                 ↪ weight
134             if parent[u] is not None:
135                 ↪ forest_edges.append((parent
136                 ↪ [u], u))
137             for neighbor_v in graph.neighbors(
138                 ↪ u, mode='all'):
139                 if neighbor_v in key:
140                     edge = graph.es[graph.
141                         ↪ get_eid(u, neighbor_v
142                         ↪ , directed=False)]
143                     edge_weight = edge['weight']
144                     if not in_mst[neighbor_v]
145                         ↪ and edge_weight < key
146                         ↪ [neighbor_v]:
147                         key[neighbor_v] =
148                             ↪ edge_weight
149                         parent[neighbor_v] = u
150                         pq.update(neighbor_v,
151                             ↪ edge_weight)
152         logging.info(f"Algoritmo de Prim
153             ↪ completado en {time.time() -
154             ↪ start_time:.4f} s.")
155     return forest_edges, total_forest_weight
156
157 # --- Función de Visualización del MST con
158     ↪ Graphistry ---
159 def visualizar_mst_con_graphistry(graph: ig
160     ↪ .Graph, mst_edges: List[Tuple[int,
161     ↪ int]]):
162     if not mst_edges:
163         logging.error("La lista de aristas
164             ↪ del MST está vacía."); return
165
166     logging.info("Preparando el MST para la
167         ↪ visualización en Graphistry...")
168     aristas_a_dibujar = mst_edges
169     if len(mst_edges) >
170         ↪ MAX_ARISTAS_MST_PARA_GRAPHISTRY:
171         logging.warning(f"El MST ({len(
172             ↪ mst_edges)} aristas) supera el
173             ↪ límite de {
174             ↪ MAX_ARISTAS_MST_PARA_GRAPHISTRY
175             ↪ }. Tomando muestra.")
176     aristas_a_dibujar = random.sample(
177         ↪ mst_edges,
178         ↪ MAX_ARISTAS_MST_PARA_GRAPHISTRY

```

```

141     ↪ )
142 df_edges_pl = pl.DataFrame(
143     ↪ aristas_a_dibujar, schema={'
144     ↪ src_orig': pl.Int64, 'dst_orig':
145     ↪ pl.Int64, orient="row")
146
147 nodos_unicos = set(df_edges_pl['src_orig
148     ↪ '].to_list() + df_edges_pl['
149     ↪ dst_orig'].to_list())
150 logging.info(f"La muestra del MST
151     ↪ contiene {len(nodos_unicos)}
152     ↪ nodos únicos.")
153
154 nodos_data = {
155     'node_orig': list(nodos_unicos),
156     'lat': [graph.vs[i]['lat'] for i in
157     ↪ nodos_unicos],
158     'lon': [graph.vs[i]['lon'] for i in
159     ↪ nodos_unicos],
160     'degree_original': [graph.degree(i)
161     ↪ for i in nodos_unicos]
162 }
163 df_nodes_pl = pl.DataFrame(nodos_data)
164 id_map = {original_id: new_id for new_id
165     ↪ , original_id in enumerate(
166     ↪ nodos_unicos)}
167
168 # Usar .replace() en lugar de .map_dict
169     ↪ ()
170 src_remapped = df_edges_pl['src_orig'].
171     ↪ replace(id_map)
172 dst_remapped = df_edges_pl['dst_orig'].
173     ↪ replace(id_map)
174
175 df_edges_pl = df_edges_pl.with_columns([
176     src_remapped.alias('src'),
177     dst_remapped.alias('dst')
178 ]).drop(['src_orig', 'dst_orig'])
179
180 node_remapped = df_nodes_pl['node_orig']
181     ↪ ].replace(id_map)
182 df_nodes_pl = df_nodes_pl.with_columns(
183     node_remapped.alias('node')
184 )
185
186 try:
187     logging.info("Convirtiendo a Pandas y
188     ↪ generando la visualización...
189     ↪ ")
190     df_nodes_pd = df_nodes_pl.to_pandas()
191     df_edges_pd = df_edges_pl.to_pandas()
192
193     g_viz = graphistry.bind(source='src',
194     ↪ destination='dst', node='node
195     ↪ ').plot(df_edges_pd,
196     ↪ df_nodes_pd)
197     if isinstance(g_viz, str): url =
198     ↪ g_viz
199     else: url = g_viz.url
200
201     logging.info(f"Visualización del MST
202     ↪ creada. Ábrela en tu navegador
203     ↪ : {url}")
204     print(f"\n-> URL de la visualización
205     ↪ del MST: {url}\n")
206     webbrowser.open(url)
207 except Exception as e:
208     logging.error(f"Ocurrió un error al
209     ↪ generar la visualización de
210     ↪ Graphistry: {e}")
211
212 # --- Bloque Principal de Ejecución ---
213 if __name__ == "__main__":
214     GRAFO_PKL_ENTRADA = '
215     ↪ grafo_igraph_paralelizado.pkl'

```

```

216 try:
217     graphistry.register(api=3, username=
218     ↪ GRAPHISTRY_USERNAME, password=
219     ↪ GRAPHISTRY_PASSWORD)
220 except Exception as e:
221     logging.error(f"No se pudo registrar
222     ↪ en Graphistry: {e}"); exit()
223
224 grafo_completo = cargar_grafo(
225     ↪ GRAFO_PKL_ENTRADA)
226 if not grafo_completo: exit()
227
228 if grafo_completo.vcount() >
229     ↪ MAX_NODOS_PARA_ANALISIS:
230     logging.warning(f"El grafo completo
231     ↪ ({grafo_completo.vcount()})
232     ↪ nodos) es demasiado grande.
233     ↪ Creando subgrafo de muestra...
234     ↪ ")
235     nodos_muestra_ids = random.sample(
236     ↪ range(grafo_completo.vcount())
237     ↪ , MAX_NODOS_PARA_ANALISIS)
238     grafo_para_analisis = grafo_completo.
239     ↪ subgraph(nodos_muestra_ids)
240     logging.info(f"Subgrafo creado: {
241     ↪ grafo_para_analisis.summary()})
242     ↪ ")
243 else:
244     grafo_para_analisis = grafo_completo
245
246 preparar_pesos_geograficos(
247     ↪ grafo_para_analisis)
248
249 if grafo_para_analisis.is_directed():
250     grafo_no_dirigido =
251     ↪ grafo_para_analisis.
252     ↪ as_undirected(mode='collapse',
253     ↪ combine_edges=dict(weight="
254     ↪ min"))
255 else:
256     grafo_no_dirigido =
257     ↪ grafo_para_analisis
258
259 mst_aristas, _ = prim_mst_forest_manual(
260     ↪ grafo_no_dirigido)
261
262 if mst_aristas:
263     print("\n" + "="*60)
264     print(" VISUALIZACIÓN DEL MST DEL
265     ↪ SUBGRAFO CON GRAPHISTRY")
266     print("="*60)
267     # Se pasa el grafo completo para
268     ↪ poder buscar los atributos
269     ↪ originales de los nodos
270     visualizar_mst_con_graphistry(
271     ↪ grafo_completo, mst_aristas)
272     print("="*60)
273 else:
274     logging.error("No se pudo calcular el
275     ↪ MST para el subgrafo.")

```

Listing 7. Código para el PrimBim

VI. PRUEBAS

VI-A. Analisis de Comunidad

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

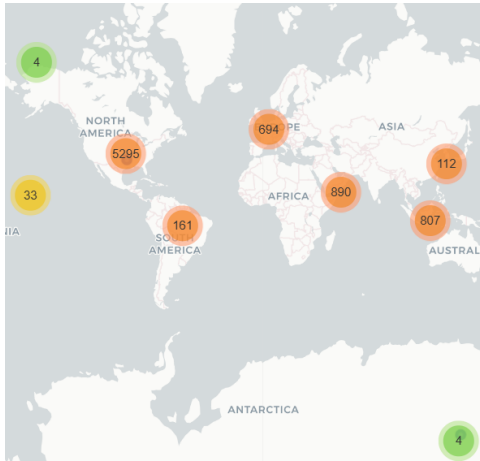


Fig 1. Muestra de la prueba de comunidades

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.



Prueba de dijkstra

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.

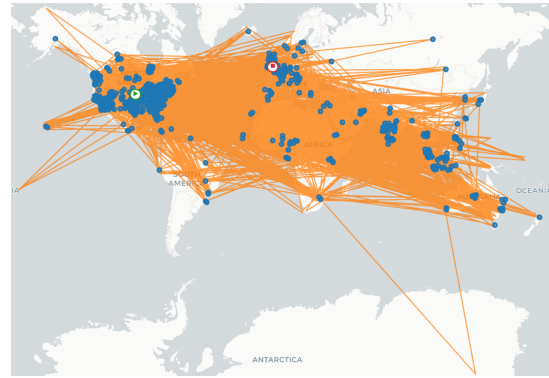


Fig 3. Mapa dfs multicapa.

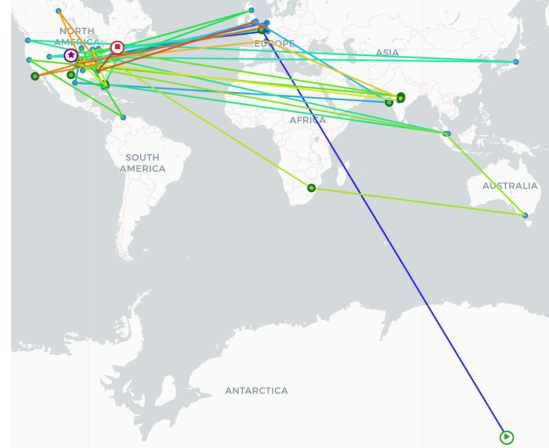


Fig 4. Mapa dfs con marcadores de capa.

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.

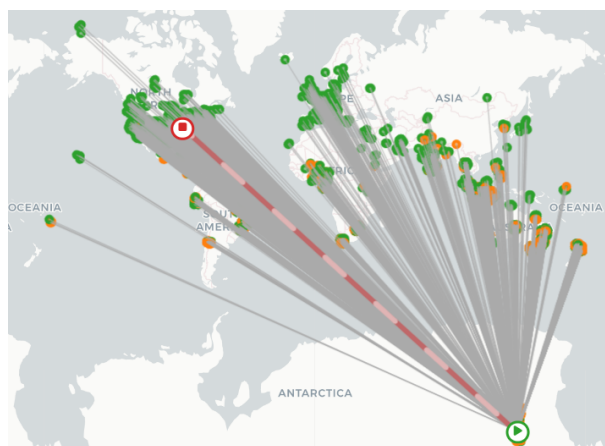


Fig 5. Mapa bfs.

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.