UNIVERSIDAD DE JAÉN

MINERÍA WEB

CURSO ACADÉMICO 24-25

API para gestión de grafos NO dirigidos

Autores
ALEJANDRO ARROYO LOAISA
JOSÉ PABLO SORIANO TORRES

Profesor
ÁNGEL MIGUEL GARCÍA VICO

25 de abril de 2025

Índice

Índice de figuras

Índice de algoritmos

1. Introducción

La teoría de grafos es un área fundamental en informática y matemáticas, con aplicaciones en la modelización de redes sociales, sistemas de comunicación, rutas de transporte y muchos otros dominios. En este contexto, disponer de herramientas para gestionar y analizar grafos es de gran utilidad para comprender las propiedades estructurales de dichas redes. El presente proyecto consiste en el desarrollo de una **aplicación web para la gestión de grafos no dirigidos** que permite **cargar datos de un grafo, calcular métricas de centralidad** y otras propiedades de la red, **visualizar el grafo** de forma interactiva y **realizar consultas** específicas (como búsqueda de nodos por nombre y cálculo de distancias geodésicas entre nodos).

La aplicación se ha implementado en **Python** utilizando el microframework **Flask** para construir una API web **RESTful**. Esto permite que todas las funcionalidades del sistema estén disponibles a través de endpoints HTTP, facilitando la interacción mediante una interfaz web intuitiva. El usuario puede cargar la definición de un grafo a partir de archivos CSV, tras lo cual el sistema procesa los datos y almacena el grafo en memoria. A continuación, el usuario tiene a su disposición diversas operaciones de análisis: cálculo de métricas de red (centralidad de grado, centralidad de cercanía, centralidad de intermediación, PageRank y HITS), consulta de información detallada de nodos, búsqueda de nodos por nombre y obtención de la distancia geodésica entre dos nodos seleccionados. Asimismo, la aplicación ofrece una visualización gráfica interactiva del grafo cargado, que ayuda a interpretar y explorar la estructura de conexiones de manera visual.

En cuanto a la estructura de este informe: en la **Sección 2** se describe la estructura general del proyecto, incluyendo el diseño de clases y la arquitectura de la aplicación. La **Sección 3** detalla el desarrollo del proyecto, explicando las principales funcionalidades implementadas (carga de datos, cálculo de métricas, endpoints de la API, visualización, etc.) y mostrando ejemplos de la interfaz web. La **Sección 4** evalúa los datos utilizados y profundiza en el análisis de la red y las métricas, proporcionando detalles teóricos y pseudocódigo de los algoritmos empleados. Finalmente, en la **Sección 5** se presentan las conclusiones del trabajo y posibles mejoras a futuro.

2. Estructura del proyecto

La aplicación desarrollada se compone de dos partes principales:

- Por un lado, el **lado del servidor** que contiene la lógica de gestión del grafo (implementado en Flask junto con las clases Python que modelan el grafo).
- Por otro lado, la **interfaz de usuario web** que permite interactuar con dicha lógica a través del navegador.

En la Figura 1 se presenta el diagrama de clases de la solución implementada, donde se ilustran las dos clases fundamentales del modelo de datos: la clase ***Grafo*** y la clase ***Nodo***. La clase Grafo representa un grafo no dirigido y almacena internamente la colección de nodos (objetos de la clase Nodo) junto con sus relaciones (aristas). Cada objeto Nodo modela un vértice del grafo, guardando su identificador único, un nombre o etiqueta asociada y la lista de sus nodos adyacentes (conectados directamente), además de campos para las métricas de centralidad y otras propiedades calculadas. En el diagrama se observa que Grafo contiene a múltiples Nodo y ofrece métodos para manipularlos (agregar/eliminar nodos o aristas) y para calcular las métricas sobre el conjunto de nodos.

Por otra parte, la **arquitectura general** de la aplicación web se esquematiza en la Figura 2. En este diagrama se muestra la interacción entre el usuario, la interfaz web y el servidor Flask. El usuario opera la aplicación a través de un navegador web, donde se carga una página (frontend) que sirve de interfaz gráfica. Cuando el usuario realiza alguna acción (por ejemplo, subir un archivo CSV o solicitar una métrica), la interfaz envía una petición HTTP al servidor Flask utilizando uno de los endpoints RESTful disponibles. El

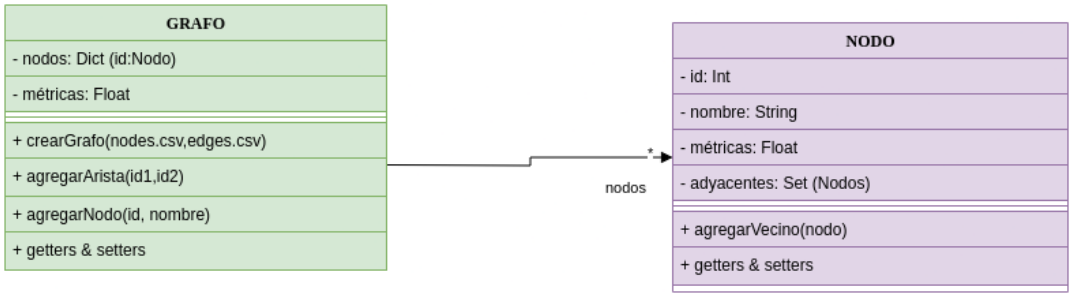

Figura 1: Diagrama de la estructura de clases del proyecto, mostrando la relación entre la clase Grafo y la clase Nodo

servidor recibe la petición en el endpoint correspondiente, la procesa (invocando métodos de la clase Grafo para acceder o modificar los datos del grafo) y responde con un resultado en formato JSON. La interfaz web, al recibir la respuesta, actualiza la información mostrada al usuario (por ejemplo, mostrando un mensaje de éxito, los valores de una métrica o dibujando el grafo). De este modo, se logra una separación clara entre la lógica de negocio (servidor/backend) y la presentación al usuario (cliente/frontend).

Figura 2: Arquitectura general de la aplicación: interacción entre el usuario, interfaz web y servidor Flask.

3. Desarrollo del proyecto

En esta sección se detallan las principales funcionalidades implementadas y cómo estas se integran en la aplicación. El desarrollo abarcó desde la lectura de datos y construcción del grafo en memoria hasta la implementación de los algoritmos de análisis de red y la creación de la interfaz web interactiva que consume la API. A continuación se describen los componentes y operaciones más relevantes del sistema:

3.1. Clase Grafo y construcción del grafo a partir de CSV

La clase central de la aplicación es **Grafo**, encargada de modelar internamente un grafo no dirigido. Durante el desarrollo se implementó la funcionalidad de cargar un grafo desde archivos CSV, lo que permite al usuario suministrar dos ficheros: uno con la lista de nodos y otro con la lista de aristas.

Al recibir estos archivos (por medio del endpoint correspondiente, descrito más adelante), el servidor invoca al constructor de *Grafo*, el cual procesa los CSV para poblar la estructura de datos. En primer lugar, se lee el archivo de nodos (*nodes.csv*), esperando que contenga al menos una columna de identificador único (*id*) y opcionalmente una columna de nombre o etiqueta (*label*) para cada nodo. Por cada entrada en este archivo, se crea un objeto *Nodo* mediante la clase *Nodo*, inicializando sus atributos id y nombre. Todos los nodos se almacenan en un diccionario o estructura similar dentro de *Grafo* para acceder a ellos rápidamente por su identificador. Tras cargar los nodos, se procesa el archivo de aristas (*edges.csv*), que contiene pares de

identificadores de nodos (*source, target*) indicando una conexión no dirigida entre esos nodos. Por cada fila de este segundo CSV, la clase *Grafo* verifica que ambos identificadores de nodo existan en el grafo y, de ser así, añade la relación de adyacencia en ambos sentidos: cada nodo agrega al otro en su lista de adyacentes (ya que el grafo es no dirigido).

Este procedimiento construye eficientemente la representación interna del grafo en memoria. Se incluyeron además comprobaciones para manejar posibles errores de formato en los CSV (campos faltantes, codificaciones de texto UTF-8/Latin-1, etc.) y asegurar la robustez de la carga de datos.

3.2. Clase Nodo y representación de vértices

La clase **Nodo** fue desarrollada para modelar cada vértice individual del grafo. Sus atributos privados incluyen el identificador *id* (una cadena o número convertido a string para consistencia) y el *nombre* o etiqueta descriptiva, así como una estructura para los adyacentes (los vecinos directos del nodo, almacenados típicamente en un conjunto para evitar duplicados).

Además, como parte del análisis de la red, a cada nodo se le asociaron campos numéricos para almacenar las métricas de centralidad calculadas: centralidad de grado, cercanía, intermediación, PageRank, authority y hub (estas dos últimas provenientes del algoritmo HITS). La clase *Nodo* provee métodos getter para acceder a estos atributos y setter para asignar los valores calculados de las métricas. También se sobreescribieron métodos especiales como *__hash__* y *__eq__* para permitir que los nodos sean utilizados como claves en conjuntos o diccionarios (esto es útil al manejar conjuntos de adyacencia y comparar nodos por su identificador).

La combinación de *Grafo* y *Nodo* proporciona una representación completa: *Grafo* gestiona la colección y las operaciones globales, mientras *Nodo* encapsula los detalles de cada entidad individual en el grafo.

3.3. Cálculo de métricas de red

Uno de los objetivos principales del proyecto fue implementar funcionalidades de **Network Analysis** que permitieran cuantificar la importancia o el papel de cada nodo en el grafo. Para ello, la clase Grafo incluye métodos que calculan diversas métricas de centralidad y conectividad basadas en algoritmos estándar de teoría de grafos. En concreto, se desarrollaron los cálculos para:

- **Centralidad de grado:** se obtiene a partir del número de adyacentes de cada nodo. En la implementación, tras construir el grafo, es trivial derivar esta métrica: equivale al grado del nodo (número de conexiones) normalizado por n-1 (donde n es el total de nodos en el grafo, para expresar la centralidad en un rango de 0 a 1). Cada nodo almacena este valor en su atributo correspondiente de centralidad.
- **Centralidad de cercanía:** mide la inversa de la distancia promedio desde un nodo a todos los demás nodos alcanzables. Para calcularla, la clase Grafo implementa un procedimiento que recorre el grafo desde cada nodo (utilizando un BFS – búsqueda en anchura) para sumar las distancias mínimas a todos los otros nodos. La cercanía de un nodo se define como $(N-1)/\sum_y d(x,y)$, donde $d(x,y)$ es la distancia geodésica entre el nodo x en cuestión y cada otro nodo y, y $N-1$ es el número de nodos alcanzables (excluyendo el propio x). Si un nodo está aislado (no conectado a los demás), su cercanía se define como 0 en la implementación. Este cálculo se realiza de forma iterativa para cada nodo, utilizando la función de BFS para obtener distancias.
- **Centralidad de intermediación (betweenness):** cuantifica cuántos caminos mínimos en la red pasan por un nodo dado. La implementación sigue el algoritmo propuesto por Brandes (adaptado según el pseudocódigo proporcionado en la sección teórica) para contar, para cada par de nodos (s,t) distintos, el número de caminos más cortos que pasan por un nodo intermediario v. En la práctica, se calcularon primero el número total de caminos mínimos entre todos los pares de nodos y luego, para cada nodo v, la fracción de esos caminos que lo atraviesan. Este valor se normaliza por el factor usual

$2/[(N-1)(N-2)]$ (válido para grafos no dirigidos) de manera que la intermediación resulte en un valor entre 0 y 1. Dada su complejidad ($O(N^3)$ en el peor caso), este fue uno de los cálculos más costosos computacionalmente del proyecto.

- **PageRank:** se implementó el algoritmo iterativo de PageRank, originalmente concebido para evaluar la importancia de páginas web, pero aquí aplicado a nodos de grafos genéricos. La función correspondiente en Grafo inicializa a todos los nodos con un valor de PageRank igual (1/N) e itera recalculando los valores hasta converger (o hasta un número máximo de iteraciones). En cada iteración, reparte la contribución de cada nodo a sus vecinos: un nodo transfiere su puntaje de PageRank equitativamente a los nodos a los que está conectado. Para simular el comportamiento de "saltos aleatorios" del algoritmo original, se incluyó un factor de amortiguación (damping factor, típicamente 0.85) que distribuye parte del valor de forma uniforme. Tras la convergencia, se asigna a cada objeto Nodo su valor final de PageRank mediante el setter correspondiente.
- **HITS (Authority y Hub):** finalmente, se incorporó el algoritmo HITS (*Hyperlink-Induced Topic Search*), que computa dos valores por nodo: autoridad (*authority*) y centralidad de concentrador (*hub*). Aunque HITS está pensado para grafos dirigidos (distinguiendo enlaces entrantes y salientes), en este proyecto se aplicó adaptándolo al grafo no dirigido (considerando la red simétrica). La implementación realiza iterativamente el cómputo: inicializa todos los nodos con valores de autoridad y hub en 1, luego en cada iteración recalcula la autoridad de un nodo como la suma de los valores hub de sus vecinos, y el valor de hub de un nodo como la suma de las autoridades de sus vecinos. Tras cada paso se normalizan todos los valores para evitar crecimiento indefinido. Después de un número determinado de iteraciones (o un criterio de convergencia), se obtienen los valores finales, que son almacenados en cada Nodo. De este modo, la aplicación puede identificar qué nodos serían considerados buenos "autoridades" (nodos enlazados por muchos otros nodos de alto hub) y cuáles actúan como buenos "hubs" (nodos que enlazan a muchos nodos de alta autoridad) dentro de la red.

Todas estas métricas de red se calculan en el backend de la aplicación. Debido a que algunas requieren un tiempo de cómputo considerable en grafos grandes (en particular la intermediación y, dependiendo del tamaño, las iteraciones de PageRank y HITS), se optó por realizar el cálculo de métricas de forma asíncrona para mejorar la experiencia de usuario.

En la implementación, cuando el usuario solicita calcular todas las métricas (mediante la interfaz web, al pulsar el botón correspondiente que consume el endpoint /calcular-metricas), el servidor Flask inicia un hilo separado en segundo plano que ejecuta los algoritmos de centralidad. Inmediatamente se devuelve al cliente una respuesta indicando que el cálculo ha comenzado, y en paralelo el backend va procesando cada métrica secuencialmente. Se llevó un control del progreso mediante una estructura compartida (metricas_estado) que almacena un indicador de porcentaje completado y las métricas ya calculadas.

La interfaz puede consultar periódicamente el estado a través de otro endpoint (/estado-metricas) para, por ejemplo, mostrar una barra de progreso o notificar al usuario una vez finalizados todos los cálculos. Esta estrategia permite que la aplicación web siga siendo receptiva (no se bloquea el servidor durante el procesamiento intensivo) y proporciona feedback al usuario en tiempo real mientras se realizan los cálculos.

3.4. Endpoints de la API RESTful

Todas las funcionalidades anteriormente descritas se expusieron mediante endpoints HTTP en el servidor Flask, conformando una API RESTful clara. A continuación se enumeran los principales endpoints implementados, junto con su propósito:

POST /cargar-grafo

Carga un nuevo grafo en el sistema. Este endpoint recibe los archivos nodes.csv y edges.csv (en una petición multipart/form-data) enviados desde la interfaz. El servidor guarda temporalmente los archivos y crea un objeto Grafo a partir de ellos. Si la carga es exitosa, retorna un mensaje de confirmación incluyendo el número de nodos y aristas cargados en el grafo.

GET /info-grafo

Retorna información básica del grafo actualmente cargado, como la cantidad de nodos y aristas que contiene. Esto permite verificar rápidamente el tamaño de la red importada.

GET /nodos

Devuelve la lista de todos los nodos del grafo en formato JSON (cada nodo con su id, nombre y número de adyacentes). Es útil para poblar tablas o listas en la interfaz con los nodos existentes.

GET /nodo/$< id\_nodo >$

Proporciona información detallada de un nodo específico identificado por su id. La respuesta incluye todos los datos relevantes: identificador, nombre, lista de adyacentes (vecinos directos) y los valores de cada métrica calculada para ese nodo (centralidad, cercanía, intermediación, PageRank, autoridad y hub). Si el nodo no existe, retorna un error indicando que el id consultado no se encuentra en el grafo.

POST /calcular-metricas

Inicia el cálculo de todas las métricas de centralidad para el grafo cargado. Como se describió, la petición lanza el procesamiento en segundo plano y responde de inmediato confirmando el inicio del cálculo.

GET /estado-metricas

Devuelve el estado actual del proceso de cálculo de métricas. Incluye indicadores como si el cálculo sigue en progreso, el porcentaje completado y una lista de métricas ya calculadas (además de algún mensaje de error en caso de fallo). La interfaz puede usar este endpoint para actualizar la barra de progreso o informar al usuario.

GET /aristas

Lista todas las aristas del grafo. Dado que el grafo es no dirigido, la implementación devuelve cada conexión una sola vez (por ejemplo, para una arista entre A y B, no se duplican A–B y B–A). La respuesta consiste en una colección de pares source, target representando cada arista.

GET /distancia/$< id\_nodo1 >/< id\_nodo2 >$

Calcula la distancia geodésica (longitud del camino más corto) entre dos nodos especificados por sus identificadores. El servidor utiliza el algoritmo BFS sobre la estructura del grafo para encontrar la distancia mínima. Si no existe ningún camino que conecte los dos nodos (es decir, pertenecen a componentes desconectadas), la respuesta indica que la distancia es infinita o que no hay camino. En caso contrario, retorna la distancia como un número entero (cantidad de aristas en el camino más corto) junto con un mensaje descriptivo.

GET /buscar-nodo/< *nombre* >

Permite buscar un nodo por su nombre o etiqueta. Este endpoint realiza una búsqueda lineal en la colección de nodos del grafo para encontrar aquel cuyo nombre coincida con el proporcionado (asumiendo que los nombres de nodo son únicos). Si lo encuentra, retorna la información de ese nodo (similar a /nodo/id); si no, responde con un mensaje indicando que no existe un nodo con ese nombre.

GET /grafo-visualizacion

Genera los datos necesarios para la visualización gráfica del grafo. La respuesta contiene dos listas JSON: una de nodes y otra de links (aristas). Cada entrada en nodes incluye el identificador del nodo, una etiqueta (nombre) y un valor numérico utilizado para la visualización (por ejemplo, se empleó el valor de centralidad de grado de cada nodo para poder reflejar su importancia en el tamaño del nodo dibujado). La lista de links contiene pares de nodos (source, target) que representan cada arista del grafo, evitando duplicados en el caso no dirigido. Este endpoint abstrae la estructura interna del grafo y la convierte a un formato fácilmente consumible por librerías de visualización en el cliente.

3.5. Interfaz web y visualización

Del lado del cliente, se desarrolló una página web estática (servida por Flask) que actúa como interfaz de usuario de la aplicación (figura 3).

Figura 3: Interfaz de Usuario de la Aplicación Web

Esta interfaz contiene formularios y controles para cada una de las operaciones disponibles (figura 4).

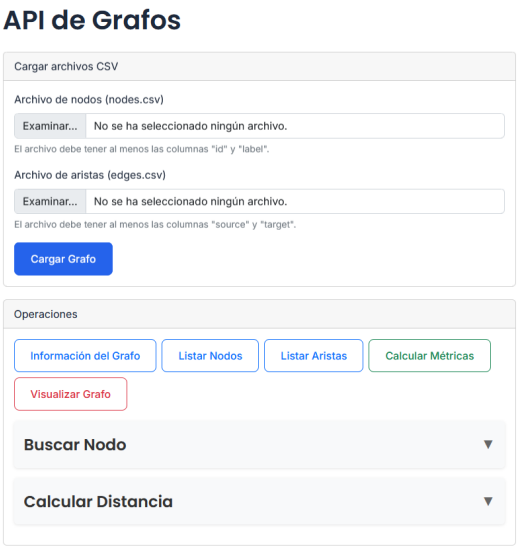

Figura 4: Operaciones Disponibles en la Aplicación Web

Por ejemplo, inicialmente se presenta un formulario para cargar los archivos CSV de nodos y aristas. Una vez el usuario selecciona los ficheros y los envía, la página realiza una petición POST /cargar-grafo y, al obtener la respuesta positiva, muestra un mensaje de éxito con la información del grafo cargado (número de nodos y aristas). La Figura 5 muestra la interfaz web tras cargar un grafo, incluyendo la confirmación de que el grafo se ha cargado correctamente con sus respectivos nodos y conexiones.

Figura 5: Mensaje de Confirmación al introducir un Grafo en la Aplicación Web

Después de la carga, la interfaz ofrece opciones para explorar y analizar el grafo. El usuario puede listar los nodos y aristas (por ejemplo, en tablas desplegables) mediante los endpoints correspondientes, o bien solicitar directamente el cálculo de métricas pulsando un botón. Al hacerlo, se invoca POST /calcular-metricas y la interfaz informa al usuario que el cálculo está en curso. Puede mostrarse una barra de progreso o un indicador textual, gracias a consultas periódicas a GET /estado-metricas que devuelven el porcentaje completado. Una vez finalizado el cálculo, la aplicación puede habilitar la visualización de resultados detallados: por ejemplo, el usuario puede hacer clic en un nodo listado para ver sus métricas individuales (consumiendo GET /nodo/id en segundo plano y mostrando los valores en la página), o solicitar un resumen general de las métricas más relevantes.

La funcionalidad de búsqueda de nodos está integrada en la interfaz mediante un campo de texto donde el usuario introduce el nombre de un nodo. Al enviar la consulta, se llama al endpoint /buscar-nodo/nombre

y, si el nodo existe, la página puede resaltar dicho nodo en la visualización y/o mostrar sus datos en pantalla. De forma similar, para obtener la distancia mínima entre dos nodos, la interfaz ofrece seleccionar dos nodos (por nombre o ID) y luego invoca GET /distancia/id1/id2, mostrando el resultado (ya sea el valor numérico de la distancia o un mensaje de que no hay camino) de forma clara al usuario.

Todas las métricas pueden ser calculadas desde el botón correspondiente de la aplicación web. Tras el proceso de cálculo (figura 6 y 7. Depende del tamaño del grafo, esto puede llevar un rato largo) podemos consultar los resultados de esta operación en cualquier momento desde "Buscar Nodo" e incluso desde la visualización del grafo (figura 11).

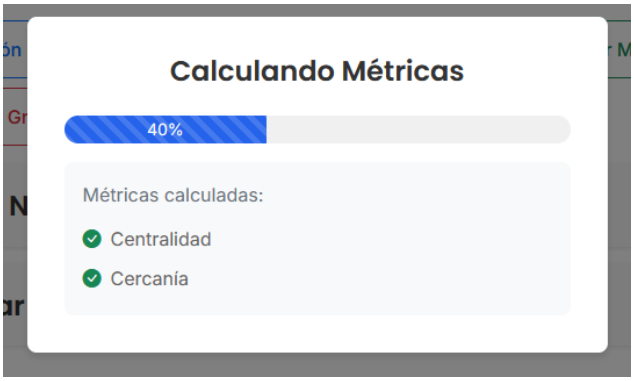

Figura 6: Cálculo de las Métricas en la Aplicación Web

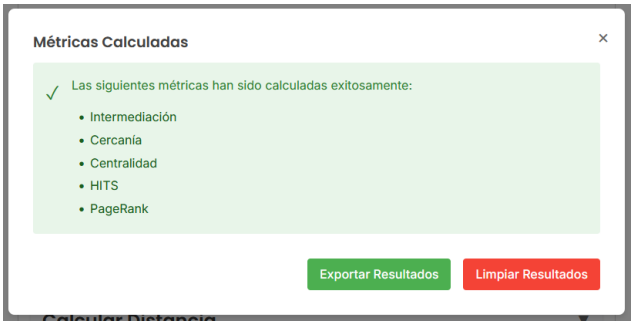

Figura 7: Cálculo de las Métricas Terminado en la Aplicación Web

Uno de los aspectos más importantes del proyecto es la visualización gráfica del grafo. La interfaz web contiene un elemento donde se dibuja el grafo de manera interactiva. Tras haber calculado las métricas, el usuario puede optar por visualizar el grafo completo pulsando el botón correspondiente, lo que detrás de escena hace una llamada a GET /grafo-visualizacion. La respuesta JSON de este endpoint, como se mencionó, provee la lista de nodos con sus labels y la lista de enlaces entre nodos. Mediante una librería de visualización de grafos en JavaScript, la interfaz toma estos datos y genera un grafo visual: cada nodo se representa como un círculo (u otro ícono) etiquetado con su nombre, y cada arista como una línea que conecta dos nodos. La visualización es interactiva, permitiendo al usuario arrastrar nodos, acercar/alejar el zoom y hacer clic en nodos individuales para ver información (según las capacidades de la librería utilizada). Las Figuras 8 y 9 presenta un ejemplo de la visualización resultante dentro de la aplicación web.

Figura 8: Grafo Visualizado sin etiquetas de Nodo en Aplicación Web

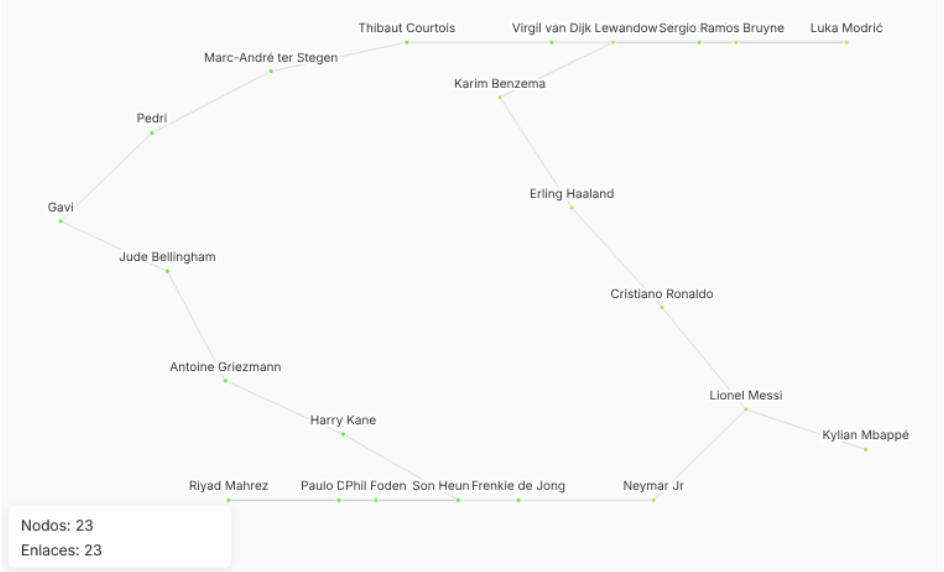

Figura 9: Grafo Visualizado con etiquetas de Nodo en Aplicación Web

Las métricas son accesibles buscando individualmente un nodo desde la opción correspondiente (figura 10) o haciendo clic en el nodo en la visualización del grafo (figura 11).

Además, se puede consultar la Distancia entre dos nodos con la herramienta “Calcular Distancia” e indicando dos nodos (figura 12).

Gracias a este conjunto de funcionalidades, el desarrollo del proyecto logró integrar **carga de datos, análisis algorítmico y visualización interactiva** en una sola herramienta. A lo largo del proceso de

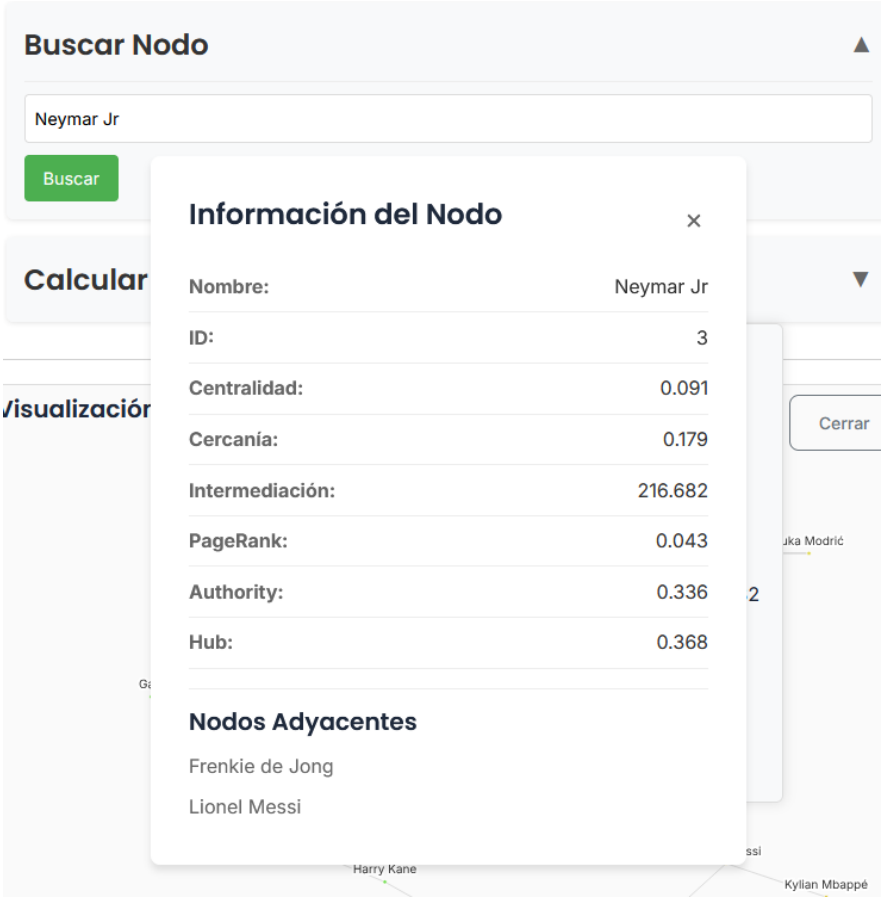

Figura 10: Métricas accedidas desde "Buscar Nodo" en Aplicación Web

implementación se prestó especial atención a la correcta gestión de la información (que los identificadores de nodos sean consistentes en todos los componentes) y a la eficiencia de los algoritmos (usando estructuras adecuadas como diccionarios, conjuntos y evitando cálculos redundantes cuando fue posible). Asimismo, las decisiones de diseño, como la separación cliente-servidor y el uso de múltiples endpoints bien definidos, facilitaron la depuración y pruebas de cada componente por separado durante el desarrollo.

4. Resumen de teoría de grafos

En esta sección, daremos una breve introducción a la teoría de grafos para después describir las métricas que hemos usado en la aplicación. Decir tiene parte de esta sección fue introducida en la práctica primera. Sin embargo, hemos visto necesario dejarla plasmada en este trabajo también. De este modo, la teoría de las métricas, nos ha ayudado a construir el pseudocódigo necesario para sus implementaciones.

4.1. ¿Qué es un grafo?

Un grafo G [BdGP10] consta de un conjunto de vértices o nodos V y un conjunto de arcos A, cada uno de los cuales une un vértice con otro. Su definición matemática se define como un par formado por ambos conjuntos 1. Los arcos que unen los nodos también se llaman **aristas del grafo** y se representan por medio

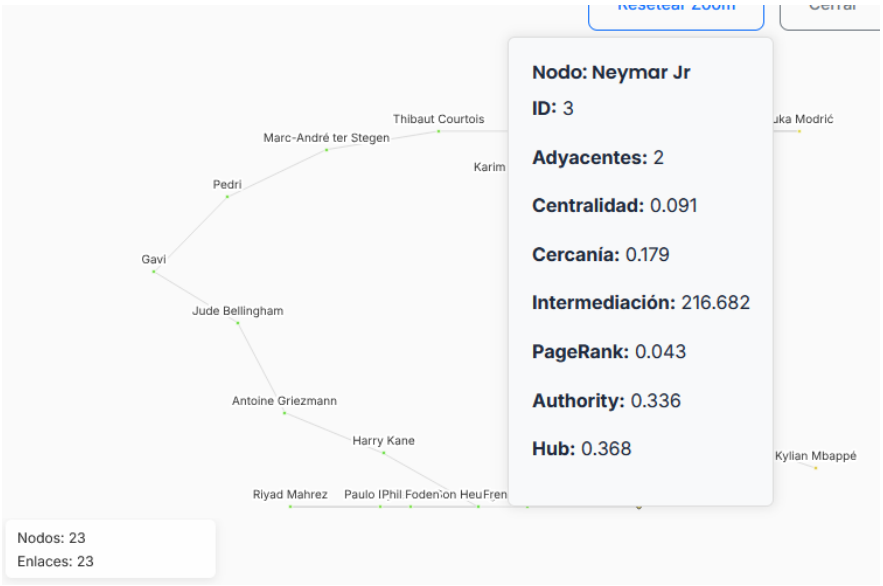

Figura 11: Métricas accedidas desde la Visualización del Grafo en Aplicación Web

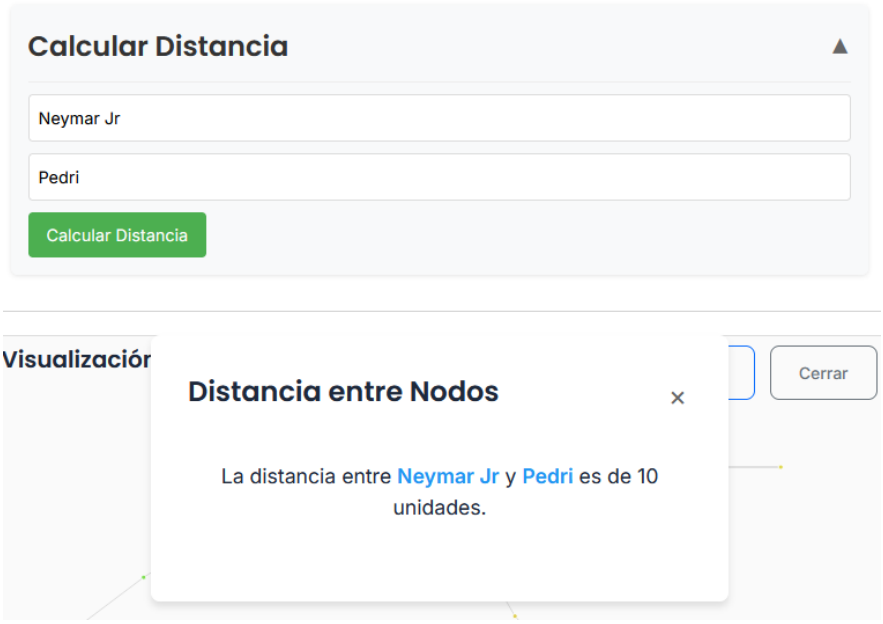

Figura 12: Distancia entre dos nodos del Grafo en Aplicación Web

de un par de elementos, (v_i, v_j), donde los elementos son los nodos que unen el arco.

$$G = (V, A) \tag{1}$$

Una vez vista la definición, podemos hablar de los 2 tipos de grafos que existen:

- **Grafos dirigidos**. Es cuando en un grafo los arcos tienen una dirección. Cada arista viene representada por un **par ordenado**, (v_i, v_j), donde el primer elemento es el nodo origen y el último elemento es el nodo destino.

- **Grafos no dirigidos**. Cuando los arcos del grafo no indican una dirección y las aristas se pueden nombrar sin importar el orden, $(v_i, v_j) = (v_j, v_i)$.

A continuación seguiremos hablando sobre una serie de definiciones de los grafos que nos ayudarán posteriormente en el análisis.

El número de nodos de un grafo se llama **orden del grafo** y se denota como: $ord(g) = |N(G)| = N$. El número de aristas o arcos es $|A(g)| = A = talla$. Así, la definición de un grafo puede darse como: $G(n, m)$, es decir, un grado de orden n y de talla m.

El grado de un vértice x es el número de vértices adyacentes a él, es decir, el número de arcos conectados a él y se denota como $grad(x)$. En nuestro caso ya sabemos que estamos tratando con un grafo **no dirigido**.

Existen muchas formas de representar un grafo, entre algunas de ellas se encuentran la matriz de adyacencias y la lista de adyacencias. En este caso hablaremos de la matriz de adyacencias.

Dado $G = (N, A)$ un grafo de n nodos. La **matriz de adyacencias** M para G es una matriz $M_{n\times n}$ de valores booleanos, donde $M(i, j)$ es verdad si y solo si existe un arco desde el nodo i hasta el nodo j.

4.2. Centralidad

La **centralidad** [Wik25b] se refiere a la importancia de los nodos dentro del grafo. Existen numerosas medidas para calcular esta métrica, aunque todas ellas tienen en común la normalización de sus valores entre 0 y 1.

La centralidad es un **atributo estructural** pues es un valor que depende de las relaciones que tenga un nodo con el resto.

Esta medida corresponde literalmente al **grado del nodo**, es decir, al número de aristas que posee dicho nodo.

Para el caso de nuestro grafo, si para cada nodo $n \in N$,$\delta(n)$ denota el grado de dicho nodo, entonces su centralidad de grado $C_D(n)$ se define como:

$$C_D(n) = \delta(n)$$

Si se tiene la matriz de adyacencia del grafo, la centralidad de un nodo i se puede definir como:

$$C_D(i) = \sum_j a_{i,j} = \sum_j a_{j,i}$$

Ahora bien, para normalizar esta medida, lo normal es dividir el grado de dicho nodo por el número de nodos del grafo menos el que estamos calculando.

$$C_D(n) = \frac{\delta(n)}{N-1}$$

Nos mide si un nodo es más activo, es decir, el que tiene más relaciones con los demás.

4.2.1. Pseudocódigo

Para calcular la **centralidad** necesitamos saber el grado de un nodo. Para saber el grado, calcularemos su **matriz de adyacencia** como vemos en el algoritmo 1.

Algorithm 1 Cálculo de la matriz de adyacencia

```
matriz[n° nodos][n° nodos]
indice ← -1
for cada clave, nodo en grafo do
    id2pos[clave] ← indice++
    for cada ady en nodo.ady do
        if ady.id no está en id2pos then
            id2pos[clave] ← indice++
        end if
        matriz[id2pos[clave]][id2pos[ady.id]] ← 1
        matriz[id2pos[ady.id]][id2pos[clave]] ← 1
    end for
end for
```

Una vez tengamos la **matriz de adyacencia** solo tendremos que sumar la fila o la columna (ya que estamos ante un grafo no dirigido) del nodo al que queramos calcular su centralidad, como podemos observar en el código 2. Nótese que se divide por $N-1$ para normalizar el valor.

Algorithm 2 Cálculo de la centralidad para cada nodo

```
for cada nodo en grafo do
    nodo.centralidad ← matriz[id2pos[nodo.id]].sum()/(N-1)
end for
```

4.3. Índice de cercanía

En esta métrica [Wik25c] es necesario definir el término de **distancia geodésica**. La distancia geodésica entre el nodo i y el nodo j, $d(i,j)$ es la longitud o el número de aristas del camino más corto que conecta a ambos nodos. Si estos nodos no están conectados, la distancia sería infinito[1].

Dicho esto, el **índice de cercanía** de un nodo i se calcula como:

$$C_c(i) = \frac{1}{\sum_{j \in N} d(i,j)}$$

Por tanto sería dividir uno por la sumatoria de las distancias geodésicas de un nodo con todos los demás. Para normalizar esta medida, no es más que multiplicarla por el número de nodos del grafo menos aquel en donde calculamos la medida.

$$C_c(i) = \frac{N-1}{\sum_{j \in N} d(i,j)}$$

Nos mide si un nodo puede interaccionar rápidamente con todos los demás.

4.3.1. Pseudocódigo

La centralidad de cercanía mide lo cerca que está un nodo de todos los demás nodos del grafo. Para ello, calculamos la suma de las **distancias geodésicas** (caminos más cortos) desde un nodo a todos los demás. Un valor alto indica que el nodo tiene acceso rápido a otros nodos en la red. A continuación, en el Algoritmo 3 se muestra el pseudocódigo para el cálculo de esta medida.

[1] Si un nodo no tiene cómo llegar a otro, está medida valdría 0, pues estamos dividiendo por infinito.

Algorithm 3 Cálculo de la centralidad de cercanía

```
Función cercania_calculo(nodo, grafo)
suma ← 0
N ← número_nodos(grafo)
for cada nodo_g en grafo do
    if nodo no es nodo_g then
        d ← distancia_geodesica_BFS(nodo, nodo_g, grafo)
        suma ← suma + d
    end if
end for
if suma mayor que 0 then
    return (N-1)/suma
else
    return 0 {Por si el nodo está en modo "forever alone"}
end if
```

Para calcular las distancias geodésicas entre nodos, utilizamos el algoritmo de búsqueda en anchura (BFS), que permite encontrar el camino más corto en grafos no ponderados. El Algoritmo 4 muestra la implementación de este procedimiento.

Algorithm 4 Cálculo de la distancia geodésica mediante BFS

```
Función distancia_geodesica_BFS(ini, fin, grafo)
{Setup inicial - necesitamos estas estructuras de datos}
cola ← nueva_cola() {¡Cola, no pila! Aquí está el truco}
visitado ← nuevo_mapa()
distancia ← nuevo_mapa()
{Configuración inicial del nodo de partida}
encolar(cola, ini)
visitado[ini] ← verdadero
distancia[ini] ← 0
{El loop principal - la sala de máquinas del algoritmo}
while cola no esté vacía do
    nodo_actual ← desencolar(cola)
    if nodo_actual es fin then
        return distancia[nodo_actual]
    end if
    {Exploración nivel por nivel (por eso es "breadth-first")}
    for cada nodo_ady en vecinos(nodo_actual, grafo) do
        if no visitado[nodo_ady] then
            encolar(cola, nodo_ady)
            visitado[nodo_ady] ← verdadero
            distancia[nodo_ady] ← distancia[nodo_actual] + 1
        end if
    end for
end while
return infinito
```

Como podemos observar, la centralidad de cercanía nos proporciona información valiosa sobre la eficiencia con la que un nodo puede comunicarse con el resto de la red. Los nodos con alta cercanía están estratégicamente posicionados para difundir información rápidamente a través del grafo.

4.4. Índice de intermediación

El **índice de intermediación** [Wik25d] cuantifica el número de veces que un nodo se encuentra entre las geodésicas[2] de otros nodos.

Formalmente, la intermediación $C_B(i)$ de un nodo i en un grafo se define como:

$$C_B(i) = \sum_{j \neq k \in N} \frac{b_{jik}}{b_{jk}}$$

donde b_{jk} es el número de caminos más cortos desde el nodo j hasta el nodo k, y b_{jik} el número de caminos más cortos desde j hasta k que pasan a través del nodo i

Para normalizar se puede dividir por el mayor número posible de pares de nodos, quitando al nodo al cual calculamos la medida.

$$C'_B(i) = \frac{2}{(n-1)(n-2)} \sum_{j \neq k \in N} \frac{b_{jik}}{b_{jk}}$$

Un nodo tendrá una alta intermediación si es un **vértice de corte** para muchas geodésicas entre actores. La idea es cuantificar el control de un humano en la comunicación existente con otros humanos en una red social. La idea intuitiva es: si elegimos dos nodos aleatoriamente y a su vez elegimos aleatoriamente uno de los posibles caminos más cortos entre ellos, entonces los nodos con mayor valor de intermediación, tendrán más probabilidad de aparecer en este camino.

4.4.1. Pseudocódigo

La centralidad de intermediación mide la frecuencia con la que un nodo se encuentra en los caminos más cortos entre otros nodos. A continuación, en el Algoritmo 5 se muestra el pseudocódigo para el cálculo de esta medida.

Algorithm 5 Cálculo de la centralidad de intermediación

```
1:  Función intermediacion(nodo, grafo)
2:  suma ← 0
3:  N ← número_nodos(grafo)
4:  for cada n1 en grafo do
5:      for cada n2 en grafo do
6:          if n1 ≠ n2 y n1 ≠ nodo y n2 ≠ nodo then
7:              caminos_n1_n2 ← caminos_mas_cortos(n1, n2)
8:              if caminos_n1_n2 mayor que 0 then
9:                  caminos_n1_nodo_n2 ← caminos_mas_cortos_entre(n1, nodo, n2)
10:                 suma ← suma + (caminos_n1_nodo_n2 / caminos_n1_n2)
11:             end if
12:         end if
13:     end for
14: end for
15: return 2/((N−1)(N−2))*suma
```

Este algoritmo itera sobre todos los pares de nodos distintos del nodo al que estamos calculando la métrica. Para cada par, determina cuántos de los caminos más cortos entre ellos pasan por el nodo de interés.

Para contar los caminos más cortos entre dos nodos, utilizamos el Algoritmo 6, que se basa en una modificación del algoritmo BFS.

[2] Ya sabemos que es el camino más corto entre dos nodos.

Algorithm 6 Conteo de caminos más cortos entre dos nodos

```
Función caminos_mas_cortos(n1, n2, grafo)
if n1 = n2 then
    return 1 {Hay 1 camino trivial a sí mismo}
end if
Cola ← [(n1, 0)] {(nodo_actual, distancia)}
distancias ← Diccionario con ∞ para cada nodo
distancias[n1] ← 0
contador_caminos ← Diccionario con 0 para cada nodo
contador_caminos[n1] ← 1
while Cola no esté vacía do
    (nodo_actual, distancia) ← extraer primer elemento de Cola
    for cada vecino en grafo[nodo_actual] do
        if distancias[vecino] = ∞ then
            distancias[vecino] ← distancia + 1
            contador_caminos[vecino] ← contador_caminos[nodo_actual]
            Agregar (vecino, distancia + 1) a la Cola
        else if distancias[vecino] = distancia + 1 then
            contador_caminos[vecino] ← contador_caminos[vecino] + contador_caminos[nodo_actual]
        end if
    end for
end while
return contador_caminos[n2]
```

Este algoritmo no solo encuentra los caminos más cortos entre dos nodos, sino que también cuenta cuántos caminos distintos de longitud mínima existen. Para ello, usa un contador que se incrementa cada vez que se descubre un nuevo camino de la misma longitud mínima.

Finalmente, el Algoritmo 7 determina cuántos caminos más cortos entre dos nodos pasan por un nodo intermedio específico.

Algorithm 7 Conteo de caminos más cortos que pasan por un nodo específico

```
Función caminos_mas_cortos_entre(n1, nodo, n2, grafo)
caminos_n1_nodo ← caminos_mas_cortos(n1, nodo, grafo)
caminos_nodo_n2 ← caminos_mas_cortos(nodo, n2, grafo)
if caminos_n1_nodo = 0 o caminos_nodo_n2 = 0 then
    return 0 {No hay conexión a través de nodo}
end if
caminos_totales ← caminos_mas_cortos(n1, n2, grafo)
return (caminos_n1_nodo * caminos_nodo_n2) / caminos_totales
```

En definitiva, se calcula la proporción de caminos más cortos entre n1 y n2 que pasan por el nodo intermedio. Primero cuenta los caminos más cortos desde n1 hasta el nodo intermedio y desde el nodo intermedio hasta n2. Luego multiplica estos valores y divide por el total de caminos más cortos entre n1 y n2. Al multiplicarlos (caminos_n1_nodo * caminos_nodo_n2), obtenemos el número total de posibles combinaciones de caminos que pasan por el nodo intermedio. Y al dividir por caminos_totales (número total de caminos más cortos entre n1 y n2), calculamos qué fracción o proporción de todos los caminos más cortos pasan por el nodo intermedio específico.

En conjunto, estos tres algoritmos nos permiten determinar la centralidad de intermediación de un nodo, que es una medida crucial para identificar los nodos que actúan como "puentes.en una red, controlando el flujo de información entre distintas comunidades.

4.5. Valores de PageRank

El Algoritmo *PageRank* [Bro24] fue propuesto por los co-fundadores de Google con el fin de determinar el orden de las páginas devueltas tras una determinada consulta.

Luego, ¿qué es el valor de *pagerank* de una página?

- Este valor representa la importancia de una página.
- Se encuentra entre 0 y 1.
- Cada página empieza con una cantidad determinada.

El algoritmo de PageRank se muestra en la ecuación 2.

$$PR(p_i) = \frac{1-d}{N} + d * \sum_{p_j \in in(p_i)} \frac{PR(p_j)}{|out(p_j|} \tag{2}$$

En donde:

- d es un factor de amortiguación.
- $in(p_i)$ es el conjunto de página que apuntan a la página p_i.
- $out(p_i)$ es el número de enlaces salientes de la página p_i.
- N es el número total de páginas.

¿Qué pasaría si una página no tiene enlaces que salen de ella, es decir, que $out(p_i) = 0$? Esto es lo que se llama un **sumidero** y puede crear problemas en este algoritmo a querer dividir entre 0.

Una solución posible es redistribuir el PageRank de los nodos sumidero de manera uniforme en todas las páginas del grafo para evitar que estas páginas nunca distribuyan su valor a otras.

En conclusión, un valor alto de **pagerank** en un nodo quiere decir que es un nodo importante y que además está relacionado con otros nodos que son considerados importantes también.

4.5.1. Pseudocódigo

Este enfoque recursivo permite identificar nodos influyentes incluso cuando no tienen un alto grado de centralidad. A continuación, en el Algoritmo 8 se muestra el pseudocódigo para calcular el PageRank de forma recursiva.

Algorithm 8 Cálculo recursivo del PageRank

```
1:  Función page_rank_recursivo(nodo, grafo, visited, d, N)
2:  if nodo está en visited then
3:     return  visited[nodo] {Evitamos recalcular}
4:  end if
5:  sum ← 0
6:  for cada ady en nodo.adyacentes do
7:     enlaces_salientes ← grado(ady) {Número de enlaces que salen de ady}
8:     if enlaces_salientes > 0 then
9:        PR ← page_rank_recursivo(ady, grafo, visited, d, N)
10:       sum ← sum + PR / enlaces_salientes
11:    end if
12: end for
13: PR_nodo ← (1 - d) / N + d * sum
14: visited[nodo] ← PR_nodo {Guardamos el resultado}
15: return  PR_nodo
```

El algoritmo PageRank implementado aquí utiliza un enfoque recursivo con memorización para evitar cálculos repetidos.

El algoritmo realiza un recorrido recursivo por los nodos que apuntan al nodo actual, calculando su contribución al PageRank. La memorización mediante el diccionario `visited` evita que se recalculen valores ya obtenidos, mejorando el rendimiento en grafos grandes.

4.6. Análisis HITS

¿Quiénes son identificados como las principales autoridades y hubs en la red? Explique cómo estos roles contribuyen a la estructura y cohesión global de la red.

Búsqueda de Temas Inducidos por Hipervínculos (HITS, *Hyperlink Induced Topic Search* [Wik25a]) también conocido como ***hubs* y autoridades**. Este concepto viene de la mano con la creación de páginas web cuando Internet se estaba formando allá por el s.XX, en donde, un buen hub representaba una página que apunta a muchas otras, mientras que una buena autoridad representa una página que está vinculada por muchos hubs diferentes.

Por lo tanto, este algoritmo asigna dos notas a cada página: su **autoridad**, que estima el valor del contenido de la página, y su valor **hub**, que estima el valor de sus enlaces a otras páginas.

Consideramos dos tipos de actualizaciones: reglas de actualización de **autoridad** y de **concentrador**. Una aplicación en k-pasos del algoritmo HITS implica solicitar k veces primero la regla de actualización de autoridades y después la de concentrador.

La regla de actualización de autoridades. En ella, $\forall p$ páginas, actualizamos $auth(p)$ para que sea: $auth(p) = \sum_{i=1}^{n} hub(i)$ donde n es el número total de páginas conectadas a p e i es una página conectada a p. Es decir, la puntuación de autoridad de una página, es la suma de todas las puntuaciones de concentradores de las páginas que apuntan a dicha página.

La regla de actualización de concentradores. En ella, $\forall p$ páginas, actualizamos $hub(p)$ para que sea: $hub(p) = \sum_{i=1}^{n} auth(i)$ donde n es el número total de páginas enlazadas desde p e i es una página enlazada desde p, es decir, que sale desde p. Así, la puntuación de una página en el concentrador es la suma de las puntuaciones de la autoridad de todas las página que salen de dicha página.

4.6.1. Pseudocódigo

A diferencia de PageRank, que asigna un único valor de importancia a cada nodo, HITS asigna dos valores: autoridad (authority) y concentrador (hub). Las autoridades son nodos que contienen información valiosa, mientras que los concentradores son nodos que apuntan a buenas autoridades. A continuación, en el Algoritmo 9 se presenta el pseudocódigo para implementar HITS.

Algorithm 9 Algoritmo HITS (Hyperlink-Induced Topic Search)

```
{Inicialización de valores de autoridad y hub}
for cada nodo en grafo do
    nodo.auth ← 1
    nodo.hub ← 1
end for
for i hasta num_iters do
    norm ← 0
    {Actualización de valores de autoridad}
    for cada nodo en grafo do
        nodo.auth ← 0
        for cada ady en nodo do
            nodo.auth ← nodo.auth + ady.hub
        end for
        norm ← norm + square(nodo.auth)
    end for
    norm ← sqrt(norm)
    for cada nodo en grafo do
        nodo.auth ← nodo.auth/norm
    end for
    {Actualización de valores de hub}
    for cada nodo en grafo do
        nodo.hub ← 0
        for cada ady en nodo do
            nodo.hub ← nodo.hub + ady.auth
        end for
        norm ← norm + square(nodo.hub)
    end for
    norm ← sqrt(norm)
    for cada nodo en grafo do
        nodo.hub ← nodo.hub/norm
    end for
end for
```

El algoritmo HITS funciona siguiendo un proceso iterativo que actualiza los valores de autoridad y hub de manera alternada. El proceso se puede resumir en los siguientes pasos:

1. Inicialmente, se asigna a todos los nodos un valor de autoridad y hub igual a 1.
2. En cada iteración:
   - Los valores de autoridad se actualizan sumando los valores de hub de los nodos que apuntan al nodo actual.
   - Los valores de hub se actualizan sumando los valores de autoridad de los nodos a los que apunta el nodo actual.
   - Después de cada actualización, se normalizan los valores dividiendo por la norma euclidiana (raíz cuadrada de la suma de los cuadrados) para evitar el crecimiento exponencial.
3. El proceso continúa durante un número predefinido de iteraciones o hasta alcanzar la convergencia.

La convergencia del algoritmo HITS suele ser bastante rápida, generalmente necesita entre 5 y 20 iteraciones para obtener resultados estables. Una vez finalizado, los nodos con altos valores de autoridad representan

fuentes valiosas de información, mientras que los nodos con altos valores de hub son buenos puntos de partida para explorar el grafo, ya que enlazan a múltiples fuentes de calidad.

5. Conclusión

En conclusión, se ha desarrollado una aplicación web completa para la gestión y análisis de grafos no dirigidos, cumpliendo con los objetivos planteados al inicio. La herramienta permite a usuarios cargar sus propios conjuntos de datos de grafos y obtener de forma automática diversas métricas de centralidad y conectividad, apoyando el análisis estructural de la red. Además, la integración de una interfaz web con visualización interactiva proporciona una experiencia intuitiva, combinando resultados cuantitativos (valores de métricas) con representaciones gráficas que facilitan la interpretación de dichos resultados en el contexto del grafo.

El proyecto ha logrado demostrar la viabilidad de combinar tecnologías de desarrollo web (*Flask, APIs REST*) con algoritmos de minería de grafos, dando lugar a una aplicación educativa y funcional. Mediante la API RESTful diseñada, las funcionalidades están modularizadas y podrían ser reutilizadas o ampliadas en el futuro. Por ejemplo, se podrían integrar fácilmente nuevos endpoints para métricas adicionales o exportar los resultados en otros formatos. La arquitectura cliente-servidor adoptada también permitiría, de ser necesario, escalar o reemplazar partes del sistema (cambiar la biblioteca de visualización front-end sin alterar la lógica backend, o viceversa).

Entre las posibles **líneas de mejora** y trabajos futuros, cabe mencionar la extensión de la herramienta para **grafos dirigidos o ponderados**, lo cual implicaría ajustar los algoritmos de métricas y la estructura de datos para considerar direcciones de arista o pesos. También sería interesante optimizar aún más el rendimiento para manejar grafos de mayor escala, quizás incorporando técnicas avanzadas o paralelismo en los cálculos de métricas más complejas. Otra mejora posible sería enriquecer la interfaz con funcionalidades adicionales, como filtrado de nodos por propiedades, resaltado de rutas específicas en la visualización, o incluso la edición manual del grafo en la propia interfaz.

En resumen, el trabajo realizado proporciona una base sólida para la gestión de grafos en entornos web, combinando **análisis teórico-práctico** de redes con una implementación tecnológica concreta. Esta aplicación no solo cumple un propósito académico (en el contexto de la asignatura de Minería Web), sino que también puede servir de punto de partida para herramientas más especializadas en análisis de redes, demostrando el valor de integrar visualización y cálculo de métricas en una plataforma unificada. El resultado es una herramienta útil para estudiantes e investigadores que deseen explorar las propiedades de un grafo de manera interactiva y obtener información valiosa sobre la estructura de sus datos.

6. Acceso y Descarga de la Aplicación

La aplicación web desarrollada para este proyecto ha sido levantada completamente funcional en el siguiente enlace:

API de Grafos - Enlace a la aplicación web

El código completo del proyecto para descargar y levantar la aplicación web se encuentra para descargar en el siguiente repositorio de Github:

Github - Enlace al repositorio

Referencias

[BdGP10] Alfredo Caicedo Barrero, Graciela Wagner de García, and Rosa María Méndez Parra. *Introducción a la Teoría de Grafos*. Elizcom sas, 2010.

[Bro24] Brown University. Graph Help Session, 2024. Último acceso: 3 marzo 2025.

[Wik25a] Wikipedia contributors. Algoritmo hits — wikipedia, la enciclopedia libre, 2025. [En línea; consultado el 3 de marzo de 2025].

[Wik25b] Wikipedia contributors. Centralidad — wikipedia, la enciclopedia libre, 2025. [En línea; consultado el 2 de marzo de 2025].

[Wik25c] Wikipedia contributors. Centralidad de cercanía — wikipedia, la enciclopedia libre, 2025. [En línea; consultado el 2 de marzo de 2025].

[Wik25d] Wikipedia contributors. Centralidad de intermediación — wikipedia, la enciclopedia libre, 2025. [En línea; consultado el 2 de marzo de 2025].