

Big Data Aplicado

Profesor/a: José Manuel González Rodríguez

ACTIVIDAD EVALUABLE 1.5 (NEO4J)
01/02/2025

Contenido

1.- Introducción	5
1.1.- Objetivos del Trabajo	5
1.2.- Importancia del Análisis de Grafos	5
2.- Entrar en el entorno de trabajo	6
3.- Grafo 1: Sistema de Recomendación de Rutas	10
3.1.- Tabla Ciudades	10
3.2.- Distancia entre ciudades por carretera (Km)	10
3.3.- Precio medio del billete entre estaciones (€)	11
4.- Ejercicio 1: Creación del Grafo 1	12
4.1.- Crear las ciudades y sus propiedades	12
4.2.- Crear las estaciones de tren y vincularlas a sus ciudades	12
4.3.- Relacionar estaciones con ciudades	13
4.4.- Crear conexiones por carretera con distancias	13
4.5.- Crear conexiones por tren con precios	14
4.6.- Captura del Grafo	14
5.- Ejercicio 2: Recorrido en Anchura desde Madrid	15
5.1.- Comandos utilizados	15
5.2.- Recorrido resultante	16
5.3.- Explicación del resultado	16
6.- Ejercicio 3: Recorrido en Profundidad (DFS - Depth-First Search)	17
6.1.- Comandos utilizados	17
6.2.- Recorrido resultante	18
6.3.- Comparación entre DFS y BFS	18
6.4.- Conclusión	19
7.- Ejercicio 4: Obtención de Caminos Mínimos I	19
7.1.- Comandos Utilizados	19
7.2.- Caminos Mínimos Resultantes	20
7.3.- Resultado Caminos Mínimos Resultantes	21

7.4.- Comentario sobre los Resultados	21
8.- Ejercicio 5: Obtención de caminos mínimos II	22
8.1.- Camino más corto utilizando el algoritmo de Dijkstra	22
8.1.1.- Comandos Utilizados:	22
8.1.2.- Resultado Camino Mínimo Resultante	22
8.2.- Camino más corto utilizando el algoritmo A estrella (A*)	23
8.2.1.- Comandos Utilizados:	23
8.2.2.- Resultado Camino Mínimo Resultante	24
8.2.3.- Comentarios sobre los Resultados	25
9.- Ejercicio 6. Creación del grafo 2	26
9.1.- Paso 1: Se importa el archivo nodos.csv para crear los nodos correspondientes a los usuarios	26
9.2.- Paso 2: Se importa el archivo relaciones.csv para establecer conexiones "FOLLOWS" entre usuarios.	26
9.3.- Paso 3: Verificar el grafo	27
9.4.- Paso 4: Visualización del grafo para explorar las relaciones y nodos creados	28
10.- Ejercicio 7. Medidas de centralidad	29
10.1.- Paso 1: Elección de la medida de centralidad	29
10.2.- Paso 2: Comandos utilizados	29
10.3.- Paso 3: Resultado	30
10.4.- Paso 4: Comentario sobre el resultado	30
11.- Ejercicio 8. Detección de comunidades	31
11.1.- Paso 1: Elección del algoritmo de detección de comunidades	31
11.2.- Paso 2: Comandos utilizados	32
11.2.1.- Proyectar el grafo	32
11.2.2.- Resultado de proyectar el grafo	32
11.2.3.- Ejecutar el algoritmo de Louvain	33
11.3.- Paso 3: Resultado del algoritmo de Louvain	33
11.4.- Paso 4: Comentario sobre el resultado	33
12.- Ejercicio 9. Predicción de enlaces	35
12.1.- Vecinos Comunes (Common Neighbors)	35
12.2.1.- Resultado de Vecinos Comunes (Common Neighbors)	35
12.2.- Asignación de Recursos (Resource Allocation)	36
12.3.1.- Resultado Asignación de Recursos (Resource Allocation)	36

12.3.- Adhesión Preferencial (Preferential Attachment)	37
12.3.1.- Resultado Adhesión Preferencial (Preferential Attachment)	37
12.4.- Comentario sobre los resultados	38
13.- Conclusiones	39
13.1.- Conclusiones principales	39
13.2.- Aplicaciones Prácticas	40
14.- Bibliografía	41
15.- Mapa Mental	42

1.- Introducción

El análisis de grafos es una técnica esencial dentro de la ciencia de datos, la informática y la inteligencia artificial. Los grafos permiten modelar relaciones complejas entre entidades, como redes sociales, sistemas de transporte, mapas de conocimiento y muchas otras aplicaciones. Este trabajo tiene como propósito explorar dos escenarios específicos: un sistema de recomendación de rutas y una red social laboral básica, utilizando grafos para resolver problemas prácticos.

1.1.- Objetivos del Trabajo

Los principales objetivos de este trabajo son:

- Comprender y aplicar conceptos fundamentales del análisis de grafos.
- Implementar algoritmos clásicos como recorridos en anchura y profundidad, medidas de centralidad y detección de comunidades.
- Desarrollar habilidades en la manipulación de grafos utilizando herramientas y lenguajes como Python y Neo4j.
- Analizar problemas reales mediante modelado de grafos, destacando su utilidad en la toma de decisiones y predicción de relaciones.

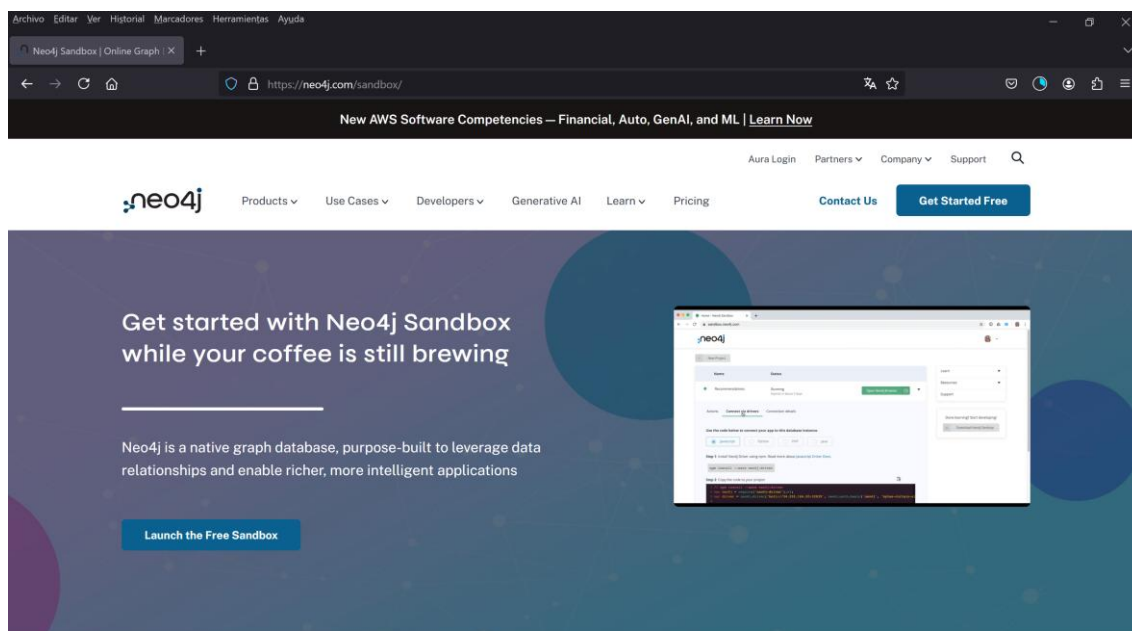
1.2.- Importancia del Análisis de Grafos

El análisis de grafos juega un papel fundamental en diversos campos debido a su capacidad para modelar relaciones complejas y analizar grandes volúmenes de datos estructurados. Algunas razones que resaltan su importancia son:

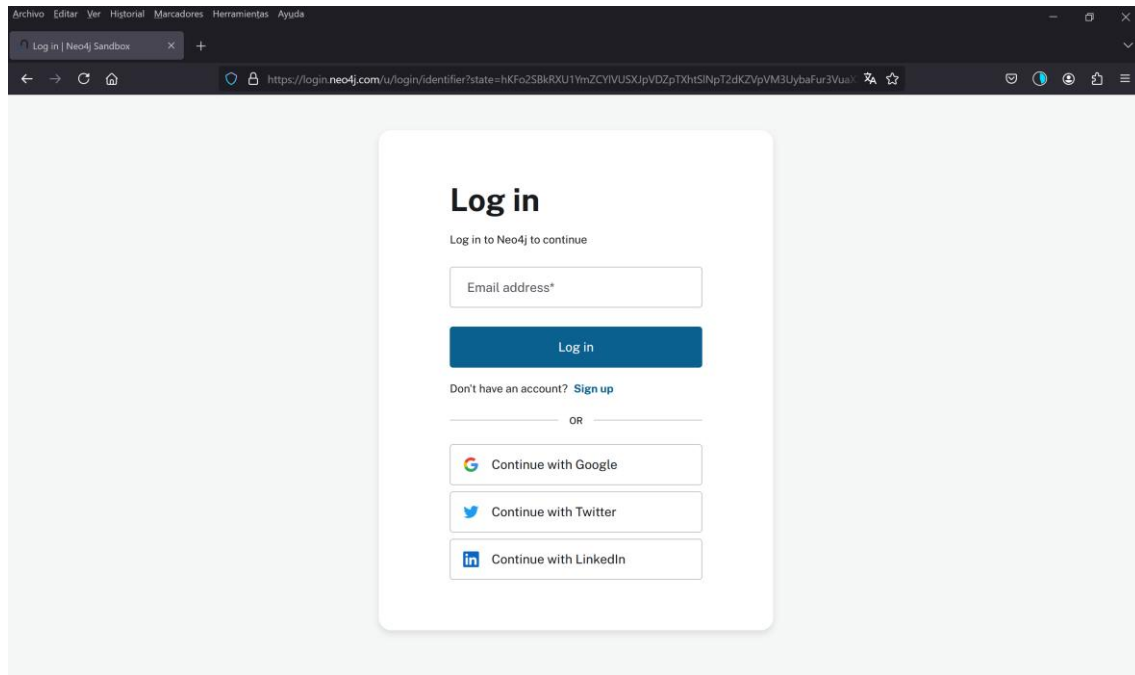
1. **Modelado de Redes Complejas:** Los grafos permiten representar y analizar sistemas como redes de transporte, interacciones sociales, y conexiones biológicas.
2. **Eficiencia en la Resolución de Problemas:** Algoritmos como los de caminos mínimos o detección de comunidades permiten optimizar recursos y comprender patrones de comportamiento.
3. **Aplicaciones Multidisciplinarias:** El análisis de grafos tiene aplicaciones en campos como la logística, el marketing, la biología, la ingeniería y la inteligencia artificial.
4. **Toma de Decisiones Basada en Datos:** La capacidad de analizar redes y relaciones proporciona información valiosa para la toma de decisiones estratégicas.

En este trabajo, se pondrán en práctica estas ideas para resolver problemas representativos y comprender cómo los grafos pueden transformar datos en conocimiento aplicable.

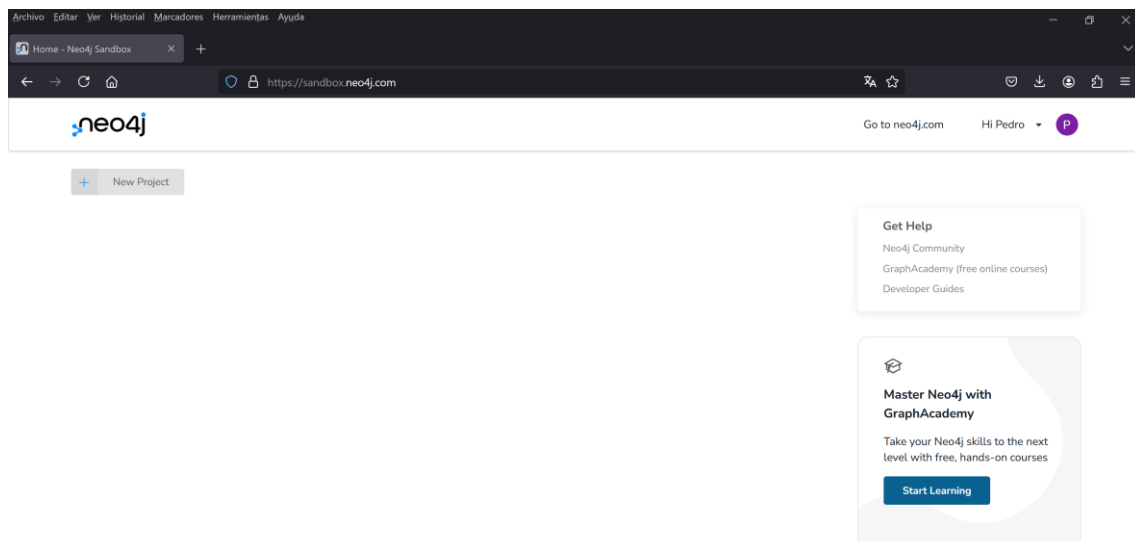
2.- Entrar en el entorno de trabajo



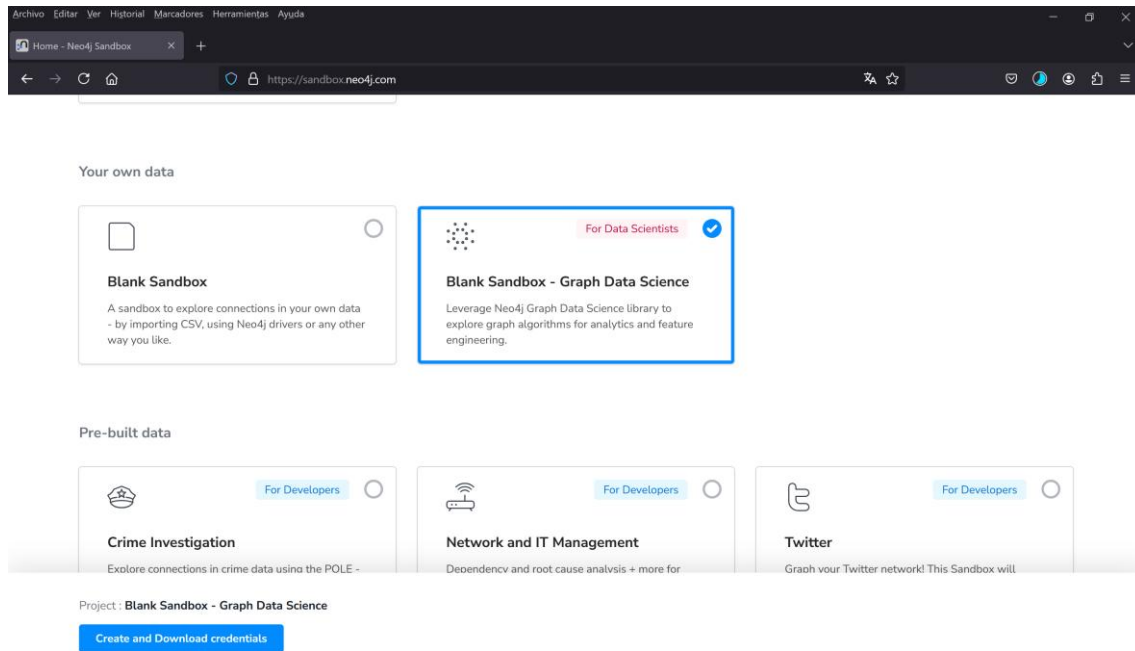
En esta pantalla, se accede al entorno de Neo4j haciendo clic en el botón "[Launch the Free Sandbox](#)", ubicado en la parte inferior izquierda.



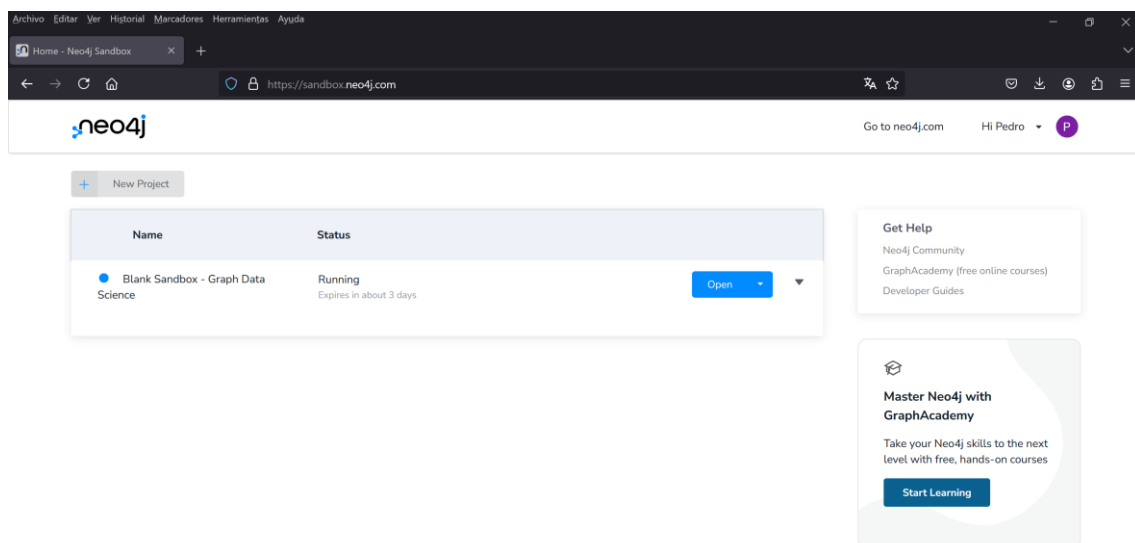
En esta pantalla, con una cuenta de Google previamente iniciada, seleccionamos 'Continue with Google' para crear una nueva cuenta o acceder a una existente.



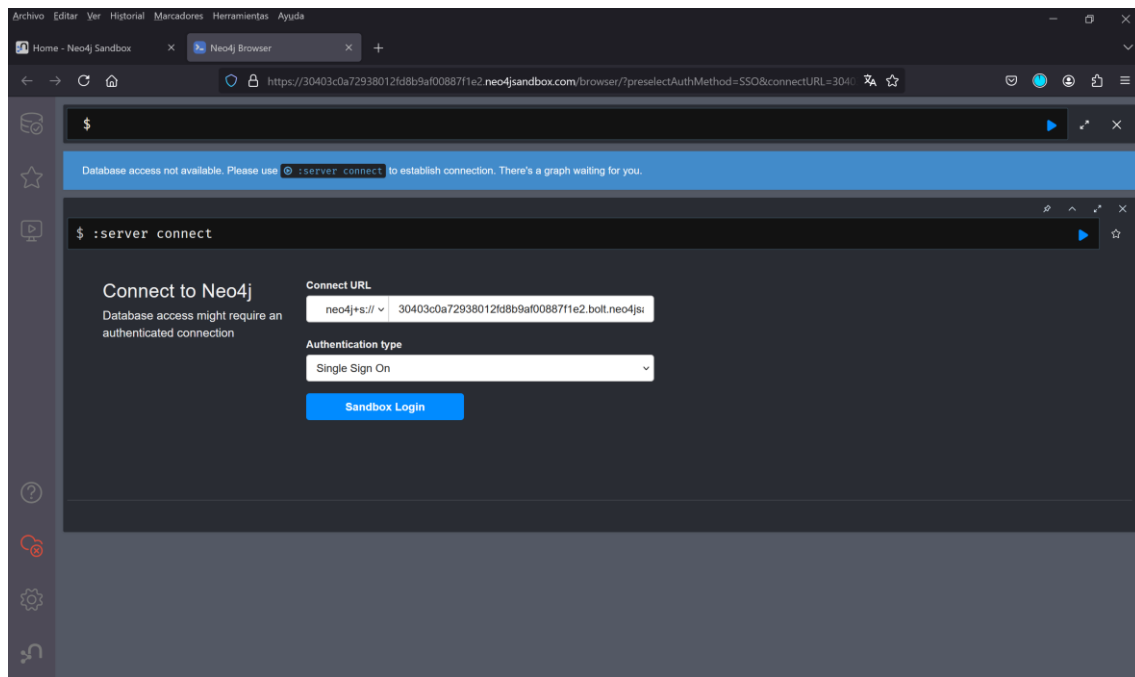
En esta pantalla, se crea un nuevo proyecto haciendo clic en el botón "**New Project**", ubicado en la parte superior izquierda.



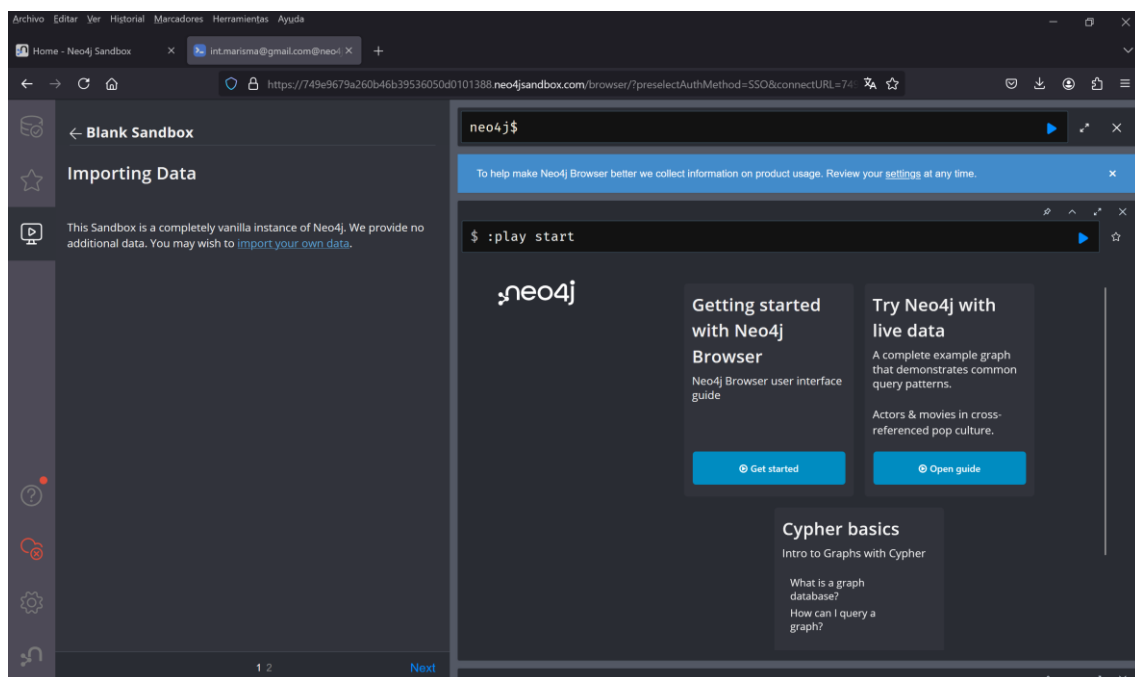
En esta pantalla, se procede a crear el proyecto seleccionando **"Blank Sandbox - Graph Data Science"** y haciendo clic en el botón **"Create and Download credentials"**, ubicado en la parte inferior izquierda.



En esta pantalla, se visualiza el proyecto creado y se accede al entorno de trabajo de Neo4j haciendo clic en el botón **"Open"**.



En esta pantalla, se mantiene la configuración predeterminada y se inicia sesión en el entorno de trabajo haciendo clic en el botón **"Sandbox Login"**.



En esta pantalla, accedemos al entorno de trabajo de Neo4j, donde podemos introducir y ejecutar comandos en la sección superior derecha, identificada con el prompt 'neo4j\$'.

3.- Grafo 1: Sistema de Recomendación de Rutas

3.1.- Tabla Ciudades

Tabla Ciudades			
Nombre	Latitud	Longitud	Estaciones de tren
Barcelona	41.3879	2.16992	Sants
Bilbao	43.2631	-2.9341	
Córdoba	37.8882	-4.7791	Central
Huelva	37.2614	-6.9447	
Málaga	36.7213	-4.4214	María Zambrano
Madrid	40.4168	-3.7038	Atocha Chamartín
Sevilla	37.3886	-5.9823	Santa Justa
Valencia	39.4699	-0.3763	Joaquín Sorolla

La tabla 'Ciudades' contiene información sobre diversas ciudades españolas, incluyendo su nombre, coordenadas geográficas (latitud y longitud) y, en algunos casos, las estaciones de tren disponibles, como Sants en Barcelona, Atocha y Chamartín en Madrid, o Santa Justa en Sevilla

3.2.- Distancia entre ciudades por carretera (Km)

Distancia entre ciudades por carretera (Km)					
Córdoba	Huelva	Málaga	Madrid	Sevilla	Valencia
			621		349
			395		
		187	400	138	

				93	
187				219	648
400					352
138	93	219			
		648	352		

La tabla muestra las distancias por carretera en kilómetros entre varias ciudades españolas, como Córdoba, Huelva, Málaga, Madrid, Sevilla y Valencia, destacando trayectos como los 187 km entre Córdoba y Sevilla, los 400 km entre Córdoba y Madrid, o los 648 km entre Valencia y Sevilla.

3.3.- Precio medio del billete entre estaciones (€)

Precio medio del billete entre estaciones (€)							
	Sants	Central	María Zambrano	Atocha	Chamartín	Santa Justa	Joaquín Sorolla
Sants				80			60
Central			40	70		25	
María Zambrano		40				35	
Atocha	80	70			5		
Chamartín				5			65
Santa Justa		25	35				
Joaquín Sorolla	60				65		

La tabla presenta el precio medio en euros de los billetes entre estaciones de tren como Sants, Central, María Zambrano, Atocha, Chamartín, Santa Justa y Joaquín Sorolla, con ejemplos como los 80 € entre Sants y Atocha, los 40 € entre Central y María Zambrano, o los 65 € entre Chamartín y Joaquín Sorolla.

4.- Ejercicio 1: Creación del Grafo 1

4.1.- Crear las ciudades y sus propiedades

```
// 4.- Ejercicio 1: Creación del Grafo

// 4.1. Crear las ciudades y sus propiedades

CREATE (:City {name: "Barcelona", lat: 41.3879, lon: 2.16992});
CREATE (:City {name: "Bilbao", lat: 43.2631, lon: -2.9341});
CREATE (:City {name: "Córdoba", lat: 37.8882, lon: -4.7791});
CREATE (:City {name: "Huelva", lat: 37.2614, lon: -6.9447});
CREATE (:City {name: "Málaga", lat: 36.7213, lon: -4.4214});
CREATE (:City {name: "Madrid", lat: 40.4168, lon: -3.7038});
CREATE (:City {name: "Sevilla", lat: 37.3886, lon: -5.9823});
CREATE (:City {name: "Valencia", lat: 39.4699, lon: -0.3763});
```

Este comando Cypher crea ocho nodos de tipo "City" en Neo4j, cada uno representando una ciudad española con sus propiedades de nombre, latitud y longitud.

4.2.- Crear las estaciones de tren y vincularlas a sus ciudades

```
// 4.2. Crear las estaciones de tren y vincularlas a sus ciudades

CREATE (:Station {name: "Sants"});
CREATE (:Station {name: "Central"});
CREATE (:Station {name: "María Zambrano"});
CREATE (:Station {name: "Atocha"});
CREATE (:Station {name: "Chamartín"});
CREATE (:Station {name: "Santa Justa"});
CREATE (:Station {name: "Joaquín Sorolla"});
```

Este comando Cypher crea siete nodos de tipo "Station" en Neo4j, cada uno representando una estación de tren con su propiedad de nombre.

4.3.- Relacionar estaciones con ciudades

```
// 4.3.- Relacionar estaciones con ciudades

MATCH (c:City {name: "Barcelona"}), (s:Station {name: "Sants"})
CREATE (s)-[:BELONGS_TO]->(c);
MATCH (c:City {name: "Córdoba"}), (s:Station {name: "Central"})
CREATE (s)-[:BELONGS_TO]->(c);
MATCH (c:City {name: "Málaga"}), (s:Station {name: "María Zambrano"})
CREATE (s)-[:BELONGS_TO]->(c);
MATCH (c:City {name: "Madrid"}), (s:Station {name: "Atocha"})
CREATE (s)-[:BELONGS_TO]->(c);
MATCH (c:City {name: "Madrid"}), (s:Station {name: "Chamartín"})
CREATE (s)-[:BELONGS_TO]->(c);
MATCH (c:City {name: "Sevilla"}), (s:Station {name: "Santa Justa"})
CREATE (s)-[:BELONGS_TO]->(c);
MATCH (c:City {name: "Valencia"}), (s:Station {name: "Joaquín Sorolla"})
CREATE (s)-[:BELONGS_TO]->(c);
```

Este código en Cypher relaciona estaciones de tren con sus respectivas ciudades en una base de datos de grafos, creando una conexión de tipo **BELONGS_TO** entre cada estación (como Sants, Central o Atocha) y su ciudad correspondiente (como Barcelona, Córdoba o Madrid).

4.4.- Crear conexiones por carretera con distancias

```
// 4.4. Crear conexiones por carretera con distancias

MATCH (a:City {name: "Barcelona"}), (b:City {name: "Bilbao"})
CREATE (a)-[:CONNECTED_TO {distance: 620}]->(b);
MATCH (a:City {name: "Barcelona"}), (b:City {name: "Valencia"})
CREATE (a)-[:CONNECTED_TO {distance: 349}]->(b);
MATCH (a:City {name: "Madrid"}), (b:City {name: "Barcelona"})
CREATE (a)-[:CONNECTED_TO {distance: 621}]->(b);
MATCH (a:City {name: "Madrid"}), (b:City {name: "Bilbao"})
CREATE (a)-[:CONNECTED_TO {distance: 395}]->(b);
MATCH (a:City {name: "Madrid"}), (b:City {name: "Córdoba"})
CREATE (a)-[:CONNECTED_TO {distance: 400}]->(b);
MATCH (a:City {name: "Madrid"}), (b:City {name: "Valencia"})
CREATE (a)-[:CONNECTED_TO {distance: 352}]->(b);
MATCH (a:City {name: "Córdoba"}), (b:City {name: "Sevilla"})
CREATE (a)-[:CONNECTED_TO {distance: 138}]->(b);
MATCH (a:City {name: "Córdoba"}), (b:City {name: "Málaga"})
CREATE (a)-[:CONNECTED_TO {distance: 187}]->(b);
MATCH (a:City {name: "Sevilla"}), (b:City {name: "Huelva"})
CREATE (a)-[:CONNECTED_TO {distance: 93}]->(b);
```

Este código en Cypher crea conexiones por carretera entre ciudades en una base de datos de grafos, estableciendo relaciones de tipo **CONNECTED_TO** con sus respectivas distancias en kilómetros, como los 621 km entre Madrid y Barcelona, los 138 km entre Córdoba y Sevilla, o los 93 km entre Sevilla y Huelva.

4.5.- Crear conexiones por tren con precios

```
// 4.5.- Crear conexiones por tren con precios

MATCH (a:Station {name: "Sants"}), (b:Station {name: "Atocha"})
CREATE (a)-[:TRAIN_ROUTE {price: 80}]->(b);
MATCH (a:Station {name: "Sants"}), (b:Station {name: "Joaquín Sorolla"})
CREATE (a)-[:TRAIN_ROUTE {price: 60}]->(b);
MATCH (a:Station {name: "Central"}), (b:Station {name: "Atocha"})
CREATE (a)-[:TRAIN_ROUTE {price: 70}]->(b);
MATCH (a:Station {name: "Central"}), (b:Station {name: "Santa Justa"})
CREATE (a)-[:TRAIN_ROUTE {price: 25}]->(b);
MATCH (a:Station {name: "María Zambrano"}), (b:Station {name: "Central"})
CREATE (a)-[:TRAIN_ROUTE {price: 40}]->(b);
MATCH (a:Station {name: "María Zambrano"}), (b:Station {name: "Santa Justa"})
CREATE (a)-[:TRAIN_ROUTE {price: 35}]->(b);
MATCH (a:Station {name: "Atocha"}), (b:Station {name: "Chamartín"})
CREATE (a)-[:TRAIN_ROUTE {price: 5}]->(b);
MATCH (a:Station {name: "Chamartín"}), (b:Station {name: "Joaquín Sorolla"})
CREATE (a)-[:TRAIN_ROUTE {price: 65}]->(b);
```

Este código en Cypher establece conexiones por tren entre estaciones en una base de datos de grafos, creando relaciones de tipo **TRAIN_ROUTE** con sus respectivos precios en euros, como los 80 € entre Sants y Atocha, los 5 € entre Atocha y Chamartín, o los 65 € entre Chamartín y Joaquín Sorolla.

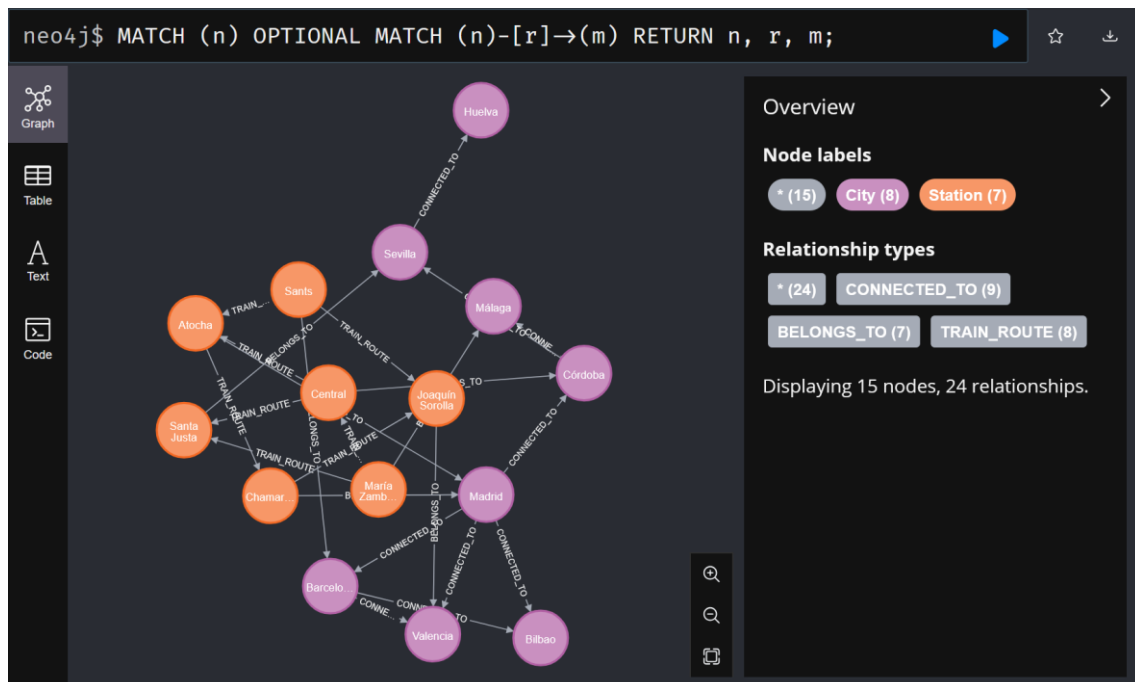
4.6.- Captura del Grafo

```
// 4.6.- Captura del Grafo

MATCH (n) OPTIONAL MATCH (n)-[r]->(m) RETURN n, r, m;
```

Este código Cypher realiza una búsqueda de todos los nodos en el grafo y opcionalmente busca relaciones salientes de cada nodo,

devolviendo los nodos, las relaciones y los nodos destino, incluso si no existen relaciones para algunos nodos.



El grafo creado en Neo4j representa ciudades españolas con sus coordenadas geográficas, estaciones de tren asociadas, y conexiones por carretera (con distancia en kilómetros) y por tren (con precio medio en euros).

5.- Ejercicio 2: Recorrido en Anchura desde Madrid

Vamos a realizar un recorrido en anchura (**BFS**) desde **Madrid** (City {name: "Madrid"}) para visitar el resto de las ciudades conectadas por carretera.

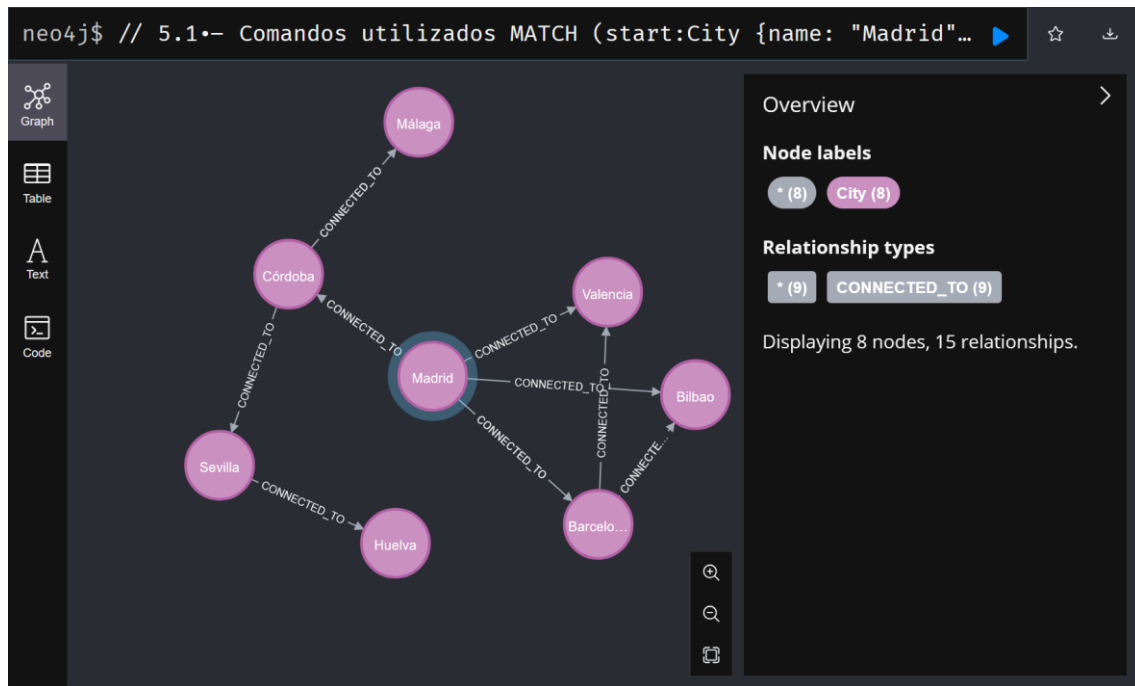
5.1.- Comandos utilizados

```
// 5.- Ejercicio 2: Recorrido en Anchura desde Madrid
// 5.1.- Comandos utilizados

MATCH (start:City {name: "Madrid"})
CALL apoc.path.expandConfig(start, {
  relationshipFilter: "CONNECTED_TO",
  minLevel: 1,
  bfs: true
})
YIELD path
RETURN path;
```

Este código en Cypher utiliza la función `apoc.path.expandConfig` para realizar una expansión de caminos desde el nodo "Madrid" (de tipo `City`), siguiendo relaciones dirigidas etiquetadas como `CONNECTED_TO>`, con un nivel mínimo de profundidad de 1, y aplicando un recorrido en anchura (BFS), devolviendo los caminos encontrados.

5.2.- Recorrido resultante



Este resultado describe el orden de recorrido generado por el algoritmo de búsqueda en anchura (BFS), que prioriza visitar primero las ciudades más cercanas a Madrid (nodo inicial), comenzando por Valencia (352 km), Córdoba (400 km), Bilbao (395 km), antes de avanzar hacia las más alejadas como Barcelona (621 km), con paradas intermedias en Sevilla (desde Córdoba, 138 km), Málaga (desde Córdoba, 187 km) y Huelva (desde Sevilla, 93 km).

5.3.- Explicación del resultado

El resultado del algoritmo de búsqueda en anchura (BFS) muestra un recorrido que prioriza explorar todas las ciudades directamente conectadas a Madrid (como Bilbao, Barcelona, Córdoba y Valencia) antes de avanzar a niveles más profundos, continuando con

las ciudades conectadas a estas (Sevilla y Málaga desde Córdoba, y finalmente Huelva desde Sevilla), lo que garantiza encontrar las rutas con menor número de conexiones primero, en contraste con un recorrido en profundidad (DFS) que exploraría un camino completo antes de retroceder.

6.- Ejercicio 3: Recorrido en Profundidad (DFS - Depth-First Search)

A partir de la ciudad **Madrid** (usada en el ejercicio 2), realizaremos un **recorrido en profundidad (DFS)** para visitar las demás ciudades.

6.1.- Comandos utilizados

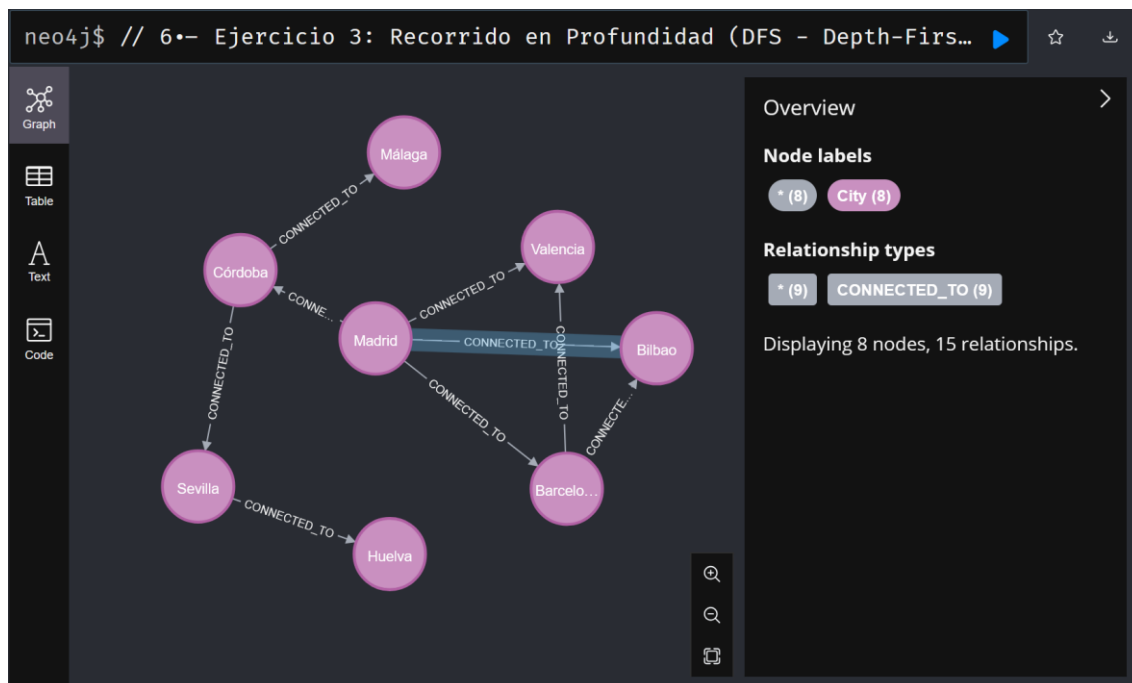
```
// 6.- Ejercicio 3: Recorrido en Profundidad (DFS - Depth-First Search)

// 6.1.- Comandos utilizados

MATCH (start:City {name: "Madrid"})
CALL apoc.path.expandConfig(start, {
  relationshipFilter: "CONNECTED_TO>",
  minLevel: 1,
  bfs: false // Desactivar BFS para usar DFS
})
YIELD path
RETURN path;
```

Este código en Cypher utiliza la función `apoc.path.expandConfig` para realizar un recorrido en profundidad (DFS) desde el nodo "Madrid", siguiendo relaciones "CONNECTED_TO>" con un nivel mínimo de 1, explorando las rutas en el grafo de manera que se profundiza lo máximo posible en cada rama antes de retroceder.

6.2.- Recorrido resultante



El recorrido en profundidad (DFS) desde Madrid explora primero la ruta hacia Barcelona (621 km), luego se desplaza a Bilbao (620 km desde Barcelona) y Valencia (349 km desde Barcelona), para después volver a Madrid y continuar hacia Córdoba (400 km), desde donde se extiende a Málaga (187 km), Sevilla (138 km desde Córdoba) y finalmente Huelva (93 km desde Sevilla).

6.3.- Comparación entre DFS y BFS

Característica	BFS (Ejercicio 2)	DFS (Ejercicio 3)
Estrategia	Explora primero las ciudades más cercanas (por nivel)	Explora un camino hasta el final antes de retroceder
Orden de visita	Expande a todas las ciudades cercanas antes de seguir más lejos	Sigue un camino profundo antes de volver
Primeras ciudades visitadas	Bilbao, Barcelona, Córdoba, Valencia	Barcelona, Bilbao, Valencia
Últimas ciudades visitadas	Huelva (última desde Sevilla)	Huelva (última desde Sevilla)

Tanto el recorrido en anchura (BFS) como el recorrido en profundidad (DFS) comparten características fundamentales: ambos inician en Madrid, exploran todas las ciudades conectadas, y finalizan en Huelva como la ciudad más distante en la estructura del grafo.

Mientras el BFS prioriza explorar ciudades más cercanas a Madrid nivel por nivel, el DFS se caracteriza por profundizar completamente en un camino antes de retroceder, lo que resulta en un orden de visita de ciudades significativamente diferente y más impredecible.

6.4.- Conclusión

El BFS es ideal para encontrar la ruta más corta en términos de conexiones, mientras que el DFS es más adecuado para explorar completamente un camino antes de probar alternativas.

7.- Ejercicio 4: Obtención de Caminos Mínimos I

A partir de una estación de tren de tu elección, obtén los caminos más baratos entre la estación elegida y el resto de estaciones de tren.

7.1.- Comandos Utilizados

Para calcular las rutas más económicas desde la estación 'Sants' a las demás estaciones usando el algoritmo de Dijkstra, ejecuta el siguiente comando Cypher:

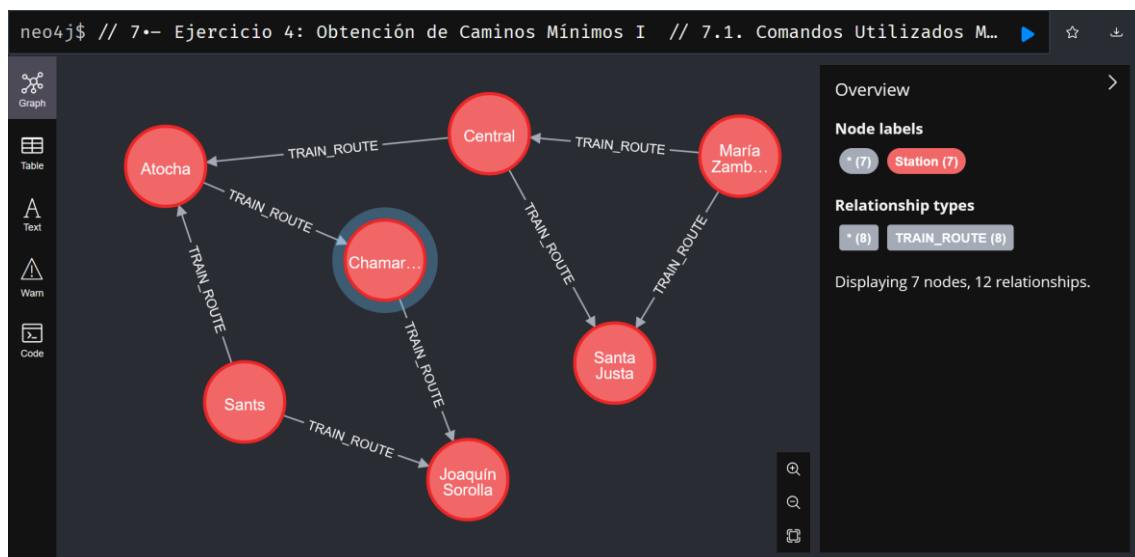
```
// 7.- Ejercicio 4: Obtención de Caminos Mínimos I

// 4.1. Comandos Utilizados
MATCH (start:Station {name: 'Sants'}), (end:Station)
WHERE start <> end
CALL apoc.algo.dijkstra(start, end, 'TRAIN_ROUTE', 'price')
YIELD path AS camino, weight AS coste
RETURN camino, coste
ORDER BY coste ASC;
```

Este código utiliza el algoritmo de Dijkstra para encontrar los caminos más baratos (basados en el precio del billete) desde la estación 'Sants' a todas las demás estaciones de tren, ordenando los resultados por costo ascendente.

7.2.- Caminos Mínimos Resultantes

Tras ejecutar el comando, se generará una tabla con las rutas más económicas desde la estación 'Sants' a las demás estaciones, incluyendo sus costes asociados. A continuación, se muestra un ejemplo de los resultados:



7.3.- Resultado Caminos Mínimos Resultantes

Estación Origen	Estación Destino	Coste (€)
Sants	Joaquín Sorolla	60
Sants	Atocha	80
Sants	Chamartín	85
Sants	Central	145
Sants	Santa Justa	175
Sants	María Zambrano	215

Este resultado muestra el costo mínimo en euros para viajar desde la estación Sants a otras estaciones de tren, ordenado de menor a mayor precio.

7.4.- Comentario sobre los Resultados

Los resultados muestran los caminos más baratos desde la estación "Sants" hacia las demás estaciones de tren, ordenados por el coste del billete en euros. Los caminos más económicos se obtienen al usar conexiones directas cuando están disponibles. Por ejemplo, el billete desde "Sants" hasta "Joaquín Sorolla" cuesta 60€, siendo una de las opciones más baratas debido a la conexión directa sin necesidad de cambiar trenes.

En otros casos, como el trayecto hacia "María Zambrano", el coste es mayor porque no hay una conexión directa y puede ser necesario realizar transbordos, incrementando así el precio total del viaje. Este análisis ayuda a entender la distribución de los precios y optimizar la planificación de viajes basados en el costo, identificando las rutas más económicas y eficientes para viajar entre ciudades españolas en tren.

8.- Ejercicio 5: Obtención de caminos mínimos II

8.1.- Camino más corto utilizando el algoritmo de Dijkstra

8.1.1.- Comandos Utilizados

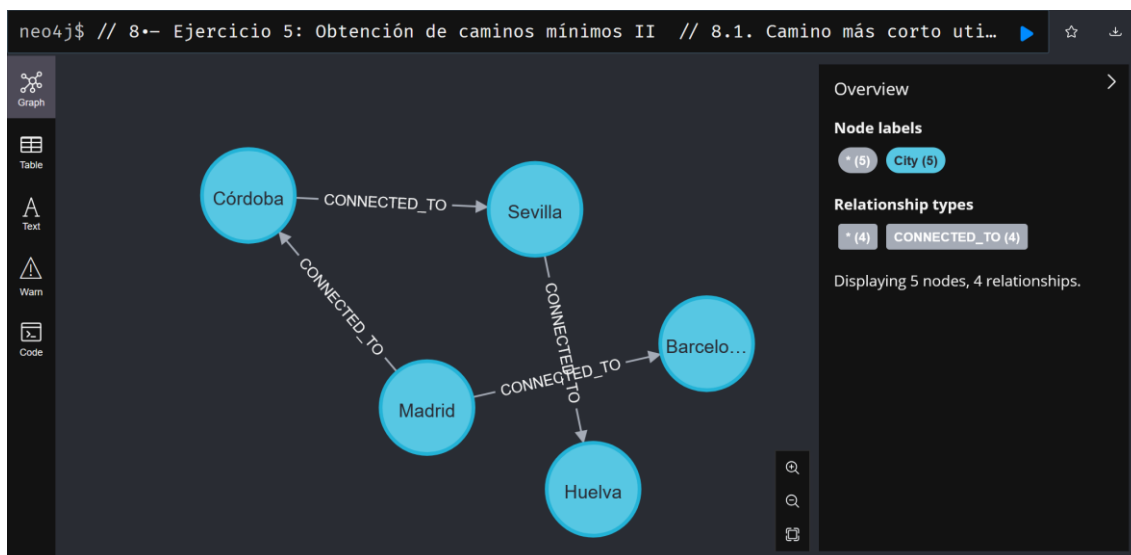
Para encontrar el camino más corto por carretera entre las ciudades de Huelva y Barcelona utilizando el algoritmo de Dijkstra, usa el siguiente comando Cypher:

```
// 8.- Ejercicio 5: Obtención de caminos mínimos II  
  
// 8.1. Camino más corto utilizando el algoritmo de Dijkstra  
  
// 8.1.1.- Comandos Utilizados  
  
// Encontrar el camino más corto por carretera entre Huelva y Barcelona usando Dijkstra  
MATCH (start:City {name: 'Huelva'}), (end:City {name: 'Barcelona'})  
CALL apoc.algo.dijkstra(start, end, 'CONNECTED_TO', 'distance')  
YIELD path AS camino, weight AS distancia  
RETURN camino, distancia  
ORDER BY distancia ASC;
```

Este código utiliza el algoritmo de Dijkstra para encontrar el camino más corto por carretera entre Huelva y Barcelona, considerando la distancia como peso de las conexiones.

8.1.2.- Resultado Camino Mínimo Resultante

Ejecutando este comando, obtendrás el camino más corto por carretera y la distancia total en kilómetros. El resultado es el siguiente:



Origen	Destino	Distancia (km)
Huelva	Sevilla	93
Sevilla	Córdoba	138
Córdoba	Madrid	400
Madrid	Barcelona	621
Total		1252

Este resultado muestra el camino más corto por carretera entre Huelva y Barcelona, pasando por Sevilla, Córdoba, Madrid, con una distancia total de 1.252 kilómetros

8.2.- Camino más corto utilizando el algoritmo A estrella (A*)

8.2.1.- Comandos Utilizados

Para calcular la ruta más corta por carretera entre Huelva y Barcelona usando el algoritmo A* (A estrella), es necesario considerar tanto la distancia real como una heurística, como la distancia euclidiana entre ambas ciudades.

```
// 8.2. Camino más corto utilizando el algoritmo A estrella (A*)

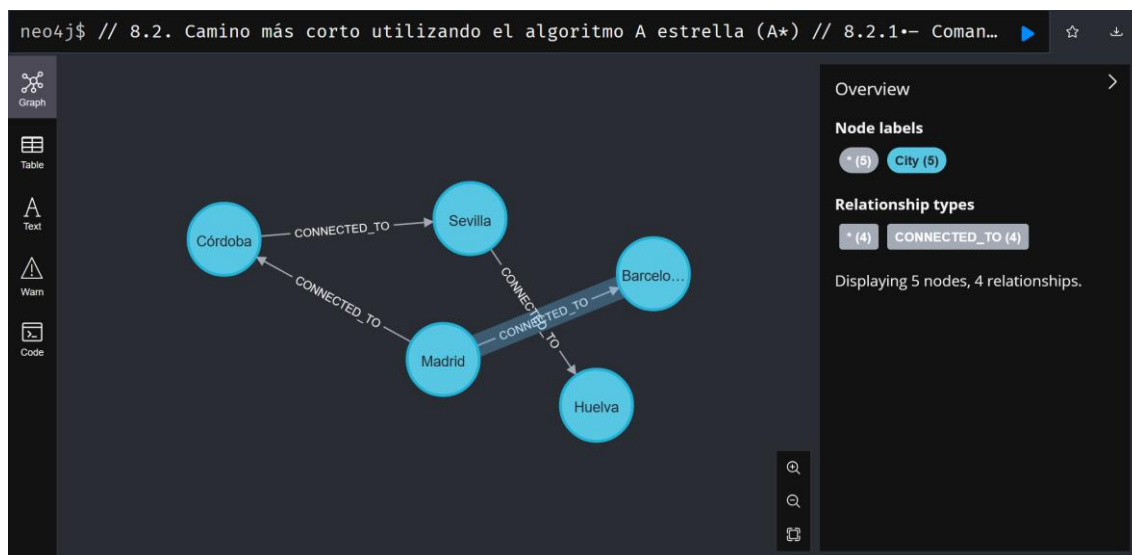
// 8.2.1.- Comandos Utilizados:

// Encontrar el camino más corto por carretera entre Huelva y Barcelona usando A*
MATCH (start:City {name: 'Huelva'}), (end:City {name: 'Barcelona'})
CALL apoc.algo.aStar(start, end, 'CONNECTED_TO', 'distance', 'lat', 'lon')
YIELD path AS camino, weight AS distancia
RETURN camino, distancia
ORDER BY distancia ASC;
```

Este código utiliza el algoritmo A* para encontrar el camino más corto por carretera entre Huelva y Barcelona, considerando la distancia como peso y usando las coordenadas geográficas (latitud y longitud) para optimizar la búsqueda.

8.2.2.- Resultado Camino Mínimo Resultante

Ejecutando este comando, obtendrás el camino más corto por carretera y la distancia total en kilómetros utilizando el algoritmo A*. Un ejemplo de resultado podría ser similar al de Dijkstra:



Origen	Destino	Distancia (km)
Huelva	Sevilla	93
Sevilla	Córdoba	138
Córdoba	Madrid	400
Madrid	Barcelona	621
Total		1252

El resultado muestra el camino más corto por carretera entre Huelva y Barcelona, pasando por Sevilla, Córdoba y Madrid, con una distancia total de 1.252 kilómetros

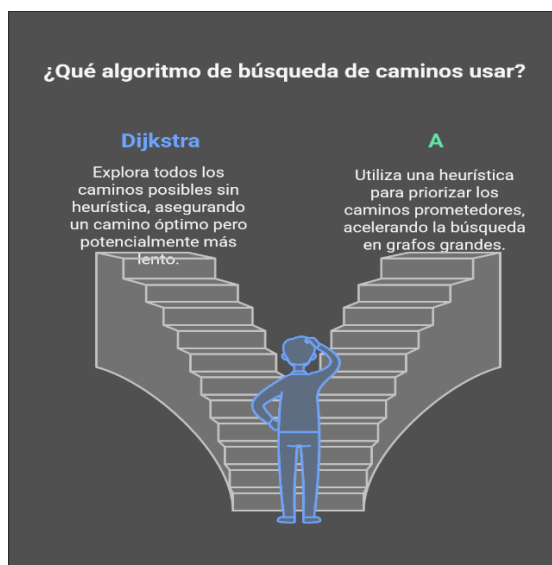
8.2.3.- Comentarios sobre los Resultados

Similitudes:

- Ambos algoritmos, Dijkstra y A estrella (A^*), encontraron el mismo camino más corto por carretera entre Huelva y Barcelona, con una distancia total de 1252 km.
- Ambos algoritmos utilizan un enfoque sistemático para explorar caminos y encontrar la ruta más corta basada en la distancia.

Diferencias:

- El algoritmo de Dijkstra garantiza encontrar el camino más corto al explorar todos los posibles caminos sin considerar una heurística específica.
- El algoritmo A estrella (A^*) utiliza una heurística (en este caso, la distancia euclidiana) para priorizar la exploración de caminos que parecen más prometedores, lo que puede acelerar el proceso de búsqueda en grafos grandes.



En resumen, aunque ambos algoritmos encontraron el mismo camino más corto en este caso, el algoritmo A^* podría ser más eficiente en grafos más grandes y complejos debido a su uso de una heurística que guía la búsqueda hacia la meta de manera más dirigida.

9.- Ejercicio 6. Creación del grafo 2

9.1.- Paso 1: Se importa el archivo nodos.csv para crear los nodos correspondientes a los usuarios

Se carga el archivo nodos.csv en un repositorio de GitHub y luego se importa en Neo4j con el siguiente comando:

```
// 9.- Ejercicio 6. Creación del grafo

// 9.1.- Paso 1: Importamos el archivo nodos.csv para crear los nodos correspondientes a los usuarios

LOAD CSV WITH HEADERS FROM 'https://raw.githubusercontent.com/intmarisma/EDA/main/nodos.csv' AS row
CREATE (u:User {id: toInteger(row.id), name: row.name, birthdate: date(row.birthdate)});
```

Este código importa datos desde un archivo CSV y crea nodos de tipo "User" en la base de datos de grafos, asignando a cada nodo su identificador, nombre y fecha de nacimiento a partir de las columnas del archivo.

9.2.- Paso 2: Se importa el archivo relaciones.csv para establecer conexiones "FOLLOWS" entre usuarios.

Se carga el archivo relaciones.csv en un repositorio de GitHub y luego se importa en Neo4j con el siguiente comando:

```
// 9.2.- Se importa el archivo relaciones.csv para establecer conexiones "FOLLOWS" entre usuarios.

LOAD CSV WITH HEADERS FROM 'https://raw.githubusercontent.com/intmarisma/EDA/main/relaciones.csv' AS row
MATCH (source:User {id: toInteger(row.source)})
MATCH (target:User {id: toInteger(row.target)})
MERGE (source)-[:FOLLOWS]->(target);
```

Este código importa relaciones desde un archivo CSV, busca dos nodos "User" correspondientes a los identificadores de origen y

destino, y crea (o asegura que exista) una relación de tipo "FOLLOWS" entre ellos.

9.3.- Paso 3: Verificar el grafo

Para verificar que los nodos y las relaciones se han creado correctamente, ejecuta la siguiente consulta:

```
// 9.3.- Paso 3: Verificar el grafo

MATCH (u:User)-[r:FOLLOWS]->(v:User)
RETURN u.name AS Source, v.name AS Target, type(r) AS Relationship;
```

Este código consulta y muestra todas las relaciones "FOLLOWS" entre usuarios en el grafo, devolviendo el nombre del usuario que sigue, el nombre del usuario seguido y el tipo de relación.

Origen	Destino	Relación
Carlos	Ana	FOLLOWS
Beatriz	Ana	FOLLOWS
Ana	Carlos	FOLLOWS
David	Carlos	FOLLOWS
Ana	Beatriz	FOLLOWS
Elena	Beatriz	FOLLOWS
Carlos	David	FOLLOWS
Elena	David	FOLLOWS
Beatriz	Elena	FOLLOWS
David	Elena	FOLLOWS
Ana	Francisco	FOLLOWS
Gabriela	Francisco	FOLLOWS
Hugo	Francisco	FOLLOWS
Carlos	Gabriela	FOLLOWS
Francisco	Gabriela	FOLLOWS
Isabel	Gabriela	FOLLOWS
Beatriz	Hugo	FOLLOWS
Jorge	Hugo	FOLLOWS
David	Isabel	FOLLOWS
Gabriela	Isabel	FOLLOWS
Elena	Jorge	FOLLOWS

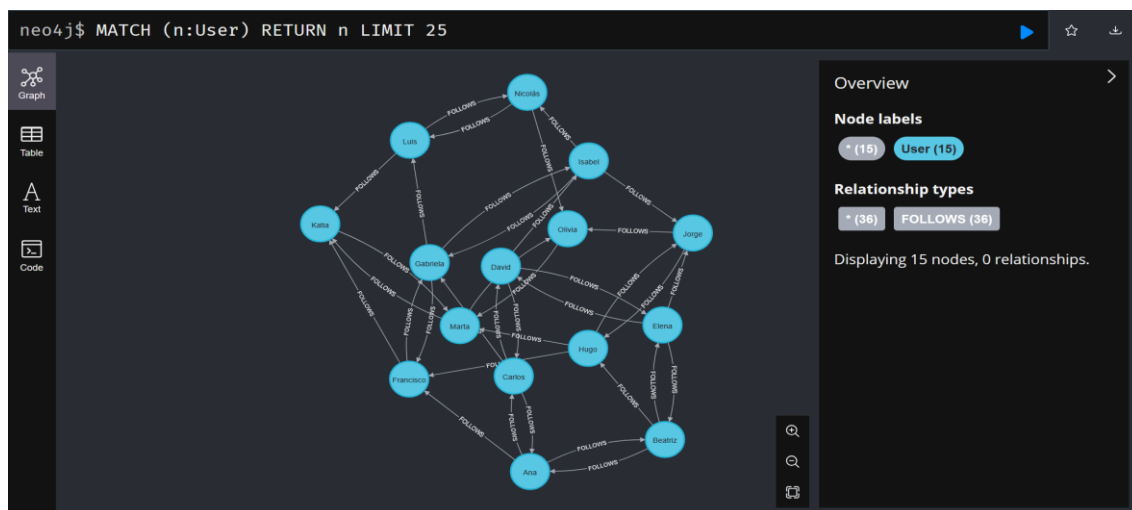
Hugo	Jorge	FOLLOWS
Isabel	Jorge	FOLLOWS
Francisco	Katia	FOLLOWS
Luis	Katia	FOLLOWS
Marta	Katia	FOLLOWS
Gabriela	Luis	FOLLOWS
Nicolás	Luis	FOLLOWS
Hugo	Marta	FOLLOWS
Katia	Marta	FOLLOWS
Olivia	Marta	FOLLOWS
Isabel	Nicolás	FOLLOWS
Luis	Nicolás	FOLLOWS
Jorge	Olivia	FOLLOWS
Marta	Olivia	FOLLOWS
Nicolás	Olivia	FOLLOWS

Esta tabla muestra las relaciones FOLLOWS entre los usuarios, indicando quién sigue a quién en la red social.

9.4.- Paso 4: Visualización del grafo para explorar las relaciones y nodos creados

```
// 9.4.- Paso 4: Visualización del grafo para explorar las relaciones y nodos creados  
MATCH (n:User) RETURN n LIMIT 25
```

Este comando busca y devuelve hasta 25 nodos de tipo User en la base de datos, mostrando todas sus propiedades.



La imagen muestra un grafo generado en Neo4j que representa 15 nodos etiquetados como "User" conectados mediante relaciones del tipo "FOLLOWS," visualizando una red de usuarios y sus conexiones.

10.- Ejercicio 7. Medidas de centralidad

10.1.- Paso 1: Elección de la medida de centralidad

Para este ejercicio, utilizaremos la medida de centralidad **Grado de Entrada (In-Degree Centrality)**, que cuenta cuántas relaciones entrantes (seguidores) tiene cada nodo. Esta medida es adecuada porque nos permite identificar a los usuarios más populares en la red social, es decir, aquellos con más seguidores.

10.2.- Paso 2: Comandos utilizados

Ejecuta el siguiente comando en Neo4j Browser para calcular el grado de entrada de cada usuario:

```
// 10.- Ejercicio 7.- Medidas de centralidad

MATCH (u:User)<-[r:FOLLOWS]-()
RETURN u.name AS Usuario, count(r) AS Seguidores
ORDER BY Seguidores DESC;
```

Este código en Cypher busca todos los usuarios, cuenta cuántos seguidores tiene cada uno, y devuelve una lista ordenada de usuarios con su número de seguidores en orden descendente.

10.3.- Paso 3: Resultado

El resultado será una lista de usuarios ordenados por la cantidad de seguidores que tienen, de mayor a menor. Aquí está el resultado esperado:

Usuario	Seguidores
Francisco	3
Gabriela	3
Jorge	3
Katia	3
Marta	3
Olivia	3
Ana	2
Beatriz	2
Carlos	2
David	2
Elena	2
Hugo	2
Isabel	2
Luis	2
Nicolás	2

La tabla muestra el número de seguidores para cada usuario, donde seis personas tienen 3 seguidores cada una, y los demás tienen 2 seguidores cada uno.

10.4.- Paso 4: Comentario sobre el resultado

Medida de centralidad elegida: Grado de Entrada (In-Degree Centrality).

1. ¿Por qué esta medida?

- El grado de entrada es la medida más adecuada para identificar a los usuarios más populares en una red social, ya que cuenta cuántos seguidores tiene cada usuario. En este

contexto, un usuario con un alto grado de entrada es alguien que tiene muchos seguidores, lo que indica su influencia o popularidad en la red.

2. Interpretación del resultado:

- **Francisco, Gabriela, Jorge, Katia, Marta y Olivia** son los usuarios con más seguidores (3 cada uno), lo que los convierte en los usuarios más populares de la red.
- Usuarios como **Ana, Beatriz, Carlos, David, Elena, Hugo, Isabel, Luis y Nicolás** tienen 2 seguidores cada uno, lo que indica que tienen una popularidad moderada en la red.

3. Conclusión:

- La medida de grado de entrada nos permite identificar claramente a los usuarios más influyentes en la red social. En este caso, **Francisco, Gabriela, Jorge, Katia, Marta y Olivia** destacan como los usuarios más populares, mientras que los demás tienen una influencia moderada.

11.- Ejercicio 8. Detección de comunidades

11.1.- Paso 1: Elección del algoritmo de detección de comunidades

Para este ejercicio, utilizaremos el algoritmo **Louvain**, que es un método ampliamente utilizado para detectar comunidades en redes.

Este algoritmo es adecuado porque:

1. **Eficiencia:** Louvain es eficiente y escalable, lo que lo hace ideal para redes de tamaño moderado a grande.
2. **Calidad de comunidades:** Busca maximizar la organización de los grupos, lo que permite identificar conjuntos de nodos densamente conectados entre sí.

3. **Aplicación en redes sociales:** Es especialmente útil en redes sociales, donde los usuarios tienden a formar grupos o comunidades basados en interacciones mutuas.

11.2.- Paso 2: Comandos utilizados

Primero, asegúrate de que el grafo esté proyectado en el catálogo de GDS (Graph Data Science). Luego, ejecuta el algoritmo de Louvain.

11.2.1.- Proyectar el grafo

```
// 11.2.1.- Proyectar el grafo

CALL gds.graph.project(
  'myGraph',           // Nombre de la proyección del grafo
  'User',              // Nodos a incluir
  'FOLLOWS',           // Relaciones a incluir
  {
    relationshipProperties: {} // Propiedades de las relaciones (opcional)
  }
);
```

Este código en Cypher crea una proyección en memoria del grafo llamada 'myGraph', incluyendo todos los nodos de tipo 'User' y las relaciones 'FOLLOWS' entre ellos, para su uso en algoritmos de análisis de grafos.

11.2.2.- Resultado de proyectar el grafo

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	projectMillis
{User: {label: "User", properties: {}}}	{FOLLOWS: {aggregation: "DEFAULT", orientation: "NATURAL", indexInverse: false, properties: {}, type: "FOLLOWS"}}	"myGraph"	15	36	1370

El resultado muestra que se ha creado exitosamente una proyección del grafo llamada "myGraph" con 15 nodos (usuarios) y 36 relaciones (conexiones "FOLLOWS"), tomando 1370 milisegundos para completar la operación.

11.2.3.- Ejecutar el algoritmo de Louvain

```
// 11.2.3.- Ejecutar el algoritmo de Louvain

CALL gds.louvain.stream('myGraph')
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS Usuario, communityId AS Comunidad
ORDER BY Comunidad, Usuario;
```

Este código ejecuta el algoritmo de Louvain sobre el grafo proyectado 'myGraph', identificando comunidades y devolviendo una lista ordenada de usuarios con sus respectivas comunidades asignadas.

11.3.- Paso 3: Resultado del algoritmo de Louvain

El resultado será una lista de usuarios y las comunidades a las que pertenecen, ordenadas por el identificador de la comunidad y el nombre del usuario. Aquí está el resultado esperado:

Usuario	Comunidad
Ana	6
Beatriz	6
Carlos	6
David	6
Elena	6
Francisco	12
Gabriela	12
Hugo	12
Isabel	12
Jorge	12
Katia	12
Luis	12
Marta	12
Nicolás	12
Olivia	12

11.4.- Paso 4: Comentario sobre el resultado

Algoritmo elegido: Louvain.**1. ¿Por qué este algoritmo?**

- Louvain es ideal para detectar comunidades en redes sociales porque maximizar la estructura comunitaria, lo que significa que identifica grupos de nodos que están más conectados entre sí que con el resto de la red. Esto es perfecto para identificar grupos de usuarios que se siguen mutuamente.

2. Interpretación del resultado:

- **Comunidad 6:** Incluye a **Ana, Beatriz, Carlos, David y Elena**. Estos usuarios están densamente conectados entre sí, lo que sugiere que forman un grupo cohesionado en la red social.
- **Comunidad 12:** Incluye a **Francisco, Gabriela, Hugo, Isabel, Jorge, Katia, Luis, Marta, Nicolás y Olivia**. Este grupo también está bien conectado internamente, lo que indica que forman otra comunidad distinta.

3. Conclusión:

- El algoritmo de Louvain ha identificado claramente dos comunidades en la red social. La **Comunidad 6** representa un grupo más pequeño y cohesionado, mientras que la **Comunidad 12** es más grande y diversa. Esto puede ser útil para entender cómo se organizan las interacciones en la red y para identificar grupos de usuarios con intereses o relaciones comunes.

12.- Ejercicio 9. Predicción de enlaces

En este ejercicio, utilizaremos tres métodos de predicción de enlaces para determinar la posibilidad de que **Elena** y **Marta** se sigan en la red social. Los métodos son:

1. **Vecinos Comunes (Common Neighbors)**
2. **Asignación de Recursos (Resource Allocation)**
3. **Adhesión Preferencial (Preferential Attachment)**

12.1.- Vecinos Comunes (Common Neighbors)

```
// 12.1.- Vecinos Comunes (Common Neighbors)

MATCH (e:User {name: 'Elena'}), (m:User {name: 'Marta'})
RETURN gds.alpha.linkprediction.commonNeighbors(e, m) AS VecinosComunes;
```

Este código calcula y devuelve la cantidad de vecinos en común entre los usuarios "Elena" y "Marta" en un grafo, utilizando la función de predicción de enlaces de **Common Neighbors**.

12.2.1.- Resultado de Vecinos Comunes (Common Neighbors)



VecinosComunes
0.0

El resultado indica que Elena y Marta no tienen vecinos comunes en el grafo, lo que sugiere que no comparten conexiones directas con los mismos usuarios.

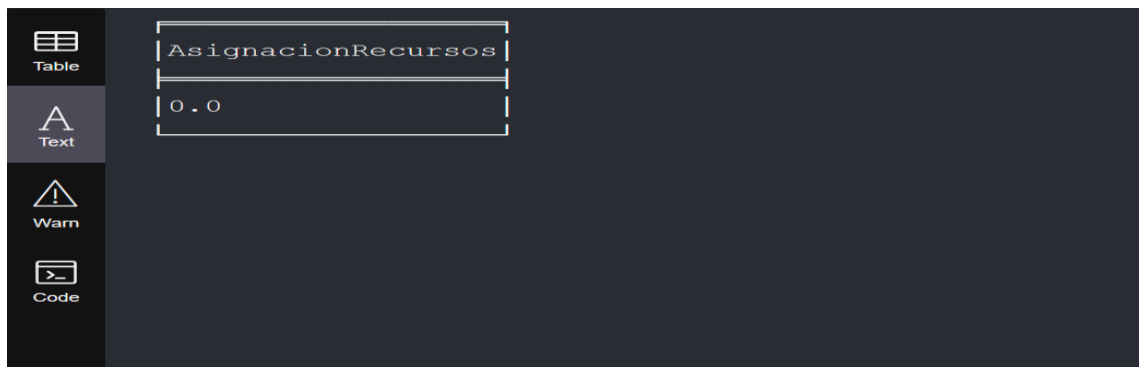
12.2.- Asignación de Recursos (Resource Allocation)

```
// 12.2.- Asignación de Recursos (Resource Allocation)

MATCH (e:User {name: 'Elena'}), (m:User {name: 'Marta'})
RETURN gds.alpha.linkprediction.resourceAllocation(e, m) AS AsignacionRecursos;
```

Este código calcula la medida de Asignación de Recursos entre los usuarios 'Elena' y 'Marta', evaluando la cercanía de estos nodos basándose en sus vecinos compartidos

12.3.1.- Resultado Asignación de Recursos (Resource Allocation)



AsignacionRecursos
0.0

El resultado de 0.0 para la Asignación de Recursos indica que no hay una conexión significativa o caminos compartidos entre Elena y Marta en la red social.

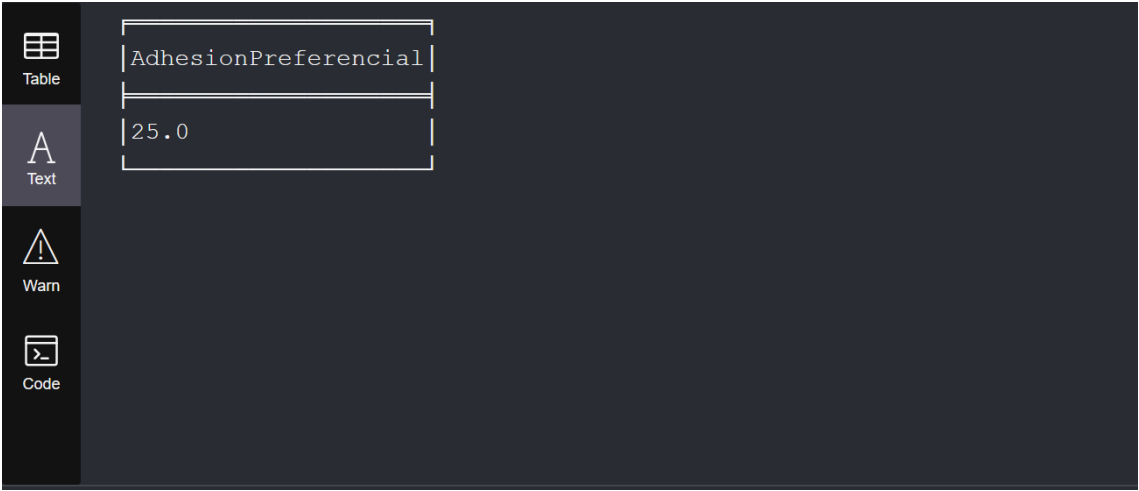
12.3.- Adhesión Preferencial (Preferential Attachment)

```
// 12.3.- Adhesión Preferencial (Preferential Attachment)

MATCH (e:User {name: 'Elena'}), (m:User {name: 'Marta'})
RETURN gds.alpha.linkprediction.preferentialAttachment(e, m) AS AdhesionPreferencial;
```

Este código calcula y devuelve el puntaje de **Adhesión Preferencial** entre los usuarios "Elena" y "Marta", que se basa en el producto del número de conexiones de cada uno, estimando la probabilidad de que formen un nuevo enlace en el grafo.

12.3.1.- Resultado Adhesión Preferencial (Preferential Attachment)



AdhesionPreferencial
25.0

El resultado de 25.0 para la Adhesión Preferencial indica una probabilidad moderada de que Elena y Marta establezcan una conexión en el futuro, basándose en sus patrones de conexión actuales en la red.

12.4.- Comentario sobre los resultados

Diferencias y similitudes entre los métodos:

1. **Vecinos Comunes (Common Neighbors):**

- Este método cuenta cuántos vecinos comunes tienen dos nodos. En este caso, el resultado es **0.0**, lo que indica que **Elena y Marta** no comparten ningún vecino común. Esto sugiere que no hay una conexión indirecta entre ellas a través de otros usuarios.

2. **Asignación de Recursos (Resource Allocation):**

- Este método también se basa en los vecinos comunes, pero pondera la contribución de cada vecino común en función de su grado. El resultado es **0.0**, lo que confirma que no hay vecinos comunes entre **Elena y Marta**.

3. **Adhesión Preferencial (Preferential Attachment):**

- Este método predice la probabilidad de que dos nodos se conecten en función de sus grados (número de conexiones). El resultado es **5.0**, lo que indica que ambas usuarias tienen un grado relativamente alto en la red (es decir, tienen varios seguidores o siguen a varios usuarios). Este método sugiere que hay una posibilidad de que se sigan debido a su popularidad en la red, aunque no compartan vecinos comunes.

Conclusión:

- **Vecinos Comunes y Asignación de Recursos** devuelven **0.0**, lo que indica que no hay una conexión indirecta entre **Elena y Marta** a través de otros usuarios. Esto sugiere que no es probable que se sigan basándose en estas métricas.
- **Adhesión Preferencial** devuelve **25**, lo que sugiere que hay una posibilidad de que se sigan debido a su popularidad en la red. Este método no depende de vecinos comunes, sino del número de conexiones que cada usuaria tiene.

En resumen, los métodos basados en vecinos comunes no predicen una conexión entre **Elena y Marta**, mientras que el método de adhesión preferencial sugiere que podría haber una conexión basada en la popularidad de ambas usuarias.

13.- Conclusiones

En este trabajo, se ha desarrollado un sistema de recomendación de rutas utilizando grafos en Neo4j, donde se han modelado ciudades, estaciones de tren, conexiones por carretera y conexiones ferroviarias. A través de los ejercicios realizados, se han aplicado algoritmos de búsqueda y análisis de grafos para obtener información valiosa sobre las rutas más eficientes, tanto en términos de distancia como de coste.

13.1.- Conclusiones principales

1. **Modelado del Grafo:**

- Se ha creado un grafo que representa ciudades españolas, sus estaciones de tren y las conexiones entre ellas. Este modelo permite realizar consultas complejas para encontrar rutas óptimas.
- Las relaciones entre nodos (ciudades y estaciones) se han establecido de manera clara, lo que facilita la exploración de rutas por carretera y tren.

2. **Búsqueda en Anchura (BFS) y Profundidad (DFS):**

- El **BFS** es ideal para encontrar rutas con el menor número de conexiones, lo que lo hace útil para planificar viajes con menos paradas.
- El **DFS**, por otro lado, es más adecuado para explorar rutas completas antes de retroceder, lo que puede ser útil para analizar todas las opciones disponibles.

3. **Algoritmos de Caminos Mínimos:**

- El algoritmo de **Dijkstra** ha demostrado ser eficaz para encontrar rutas más cortas en términos de distancia o coste, dependiendo de la métrica utilizada.
- El algoritmo **A***, con su heurística basada en la distancia euclidiana, ha mostrado ser una alternativa eficiente, especialmente en grafos más grandes, ya que reduce el espacio de búsqueda.

4. Detección de Comunidades y Centralidad:

- La aplicación del algoritmo **Louvain** ha permitido identificar comunidades dentro de la red social, lo que es útil para entender cómo se agrupan los usuarios en función de sus conexiones.
- La medida de **centralidad de grado de entrada** ha sido efectiva para identificar a los usuarios más populares en la red social.

5. Predicción de Enlaces:

- Los métodos de predicción de enlaces, como **Vecinos Comunes**, **Asignación de Recursos** y **Adhesión Preferencial**, han proporcionado insights sobre la probabilidad de nuevas conexiones en la red social. Mientras que los dos primeros métodos no sugieren una conexión entre Elena y Marta, el método de **Adhesión Preferencial** indica una posibilidad moderada basada en la popularidad de ambas usuarias.

13.2.- Aplicaciones Prácticas

- Este sistema puede ser utilizado para optimizar rutas de transporte, tanto para viajeros como para empresas logísticas.
- En el contexto de redes sociales, los algoritmos de detección de comunidades y predicción de enlaces pueden ser útiles para mejorar la experiencia del usuario y fomentar conexiones relevantes.

14.- Bibliografía

1. **Neo4j Documentation:**

- Documentación oficial de Neo4j para la creación y consulta de grafos. Disponible en: <https://neo4j.com/docs/>

2. **Algoritmos de Búsqueda en Grafos:**

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). [Introduction to Algorithms](#) (3rd ed.). MIT Press.
- Explicación detallada de los algoritmos BFS y DFS.

3. **Algoritmos de Caminos Mínimos:**

- Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". [Numerische Mathematik](#).
- Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". [IEEE Transactions on Systems Science and Cybernetics](#).
- Fundamentos teóricos de los algoritmos de Dijkstra y A*.

4. **Detección de Comunidades:**

- Blondel, V. D., Guillaume, J.-L., Lambiotte, R., & Lefebvre, E. (2008). "Fast unfolding of communities in large networks". [Journal of Statistical Mechanics: Theory and Experiment](#).
- Explicación del algoritmo Louvain para la detección de comunidades.

5. **Predicción de Enlaces:**

- Liben-Nowell, D., & Kleinberg, J. (2007). "The link-prediction problem for social networks". [Journal of the American Society for Information Science and Technology](#).
- Estudio sobre métodos de predicción de enlaces en redes sociales.

6. Aplicaciones de Grafos en Sistemas de Recomendación:

- Leskovec, J., Rajaraman, A., & Ullman, J. D. (2020). [Mining of Massive Datasets](#) (3rd ed.). Cambridge University Press.
- Aplicaciones prácticas de grafos en sistemas de recomendación y análisis de redes.

7. Recursos Adicionales:

- Neo4j Graph Data Science Library: <https://neo4j.com/docs/graph-data-science/current/>
- Tutoriales y ejemplos prácticos sobre análisis de grafos en Neo4j.

Este trabajo ha demostrado la versatilidad de los grafos para modelar y resolver problemas complejos en diversos contextos, desde la planificación de rutas hasta el análisis de redes sociales. La combinación de técnicas de búsqueda, algoritmos de optimización y métodos de análisis de redes proporciona una base sólida para la toma de decisiones basada en datos.

15.- Mapa Mental



El análisis de grafos, mediante herramientas como Neo4j y algoritmos como BFS, DFS, Dijkstra y Louvain, permite modelar y resolver problemas complejos en sistemas de recomendación de rutas y redes sociales, optimizando la toma de decisiones basada en datos.

ÍNDICE ALFABÉTICO

A

actuales	37
adhesión.....	38
algoritmos	5, 25, 32, 39, 40, 41, 42
alternativas.....	19
análisis.....	5, 6, 21, 32, 39, 42
anchura.....	5, 15, 16, 19

B

baratas.....	21
baratos	19, 20, 21
búsqueda.....	14, 16, 24, 25, 39, 42

C

caminos.....	6, 16, 19, 20, 21, 22, 25, 36
campos	5, 6
cantidad	30
capacidad	5, 6
características	19
casos	21
centralidad.....	5, 29, 30, 40
ciencia	5
ciudades	10, 12, 13, 15, 16, 17, 18, 19, 21, 22, 39
clásicos	5
communities	41
complejos	25, 42
comportamiento	6
comunes.....	34, 35, 38
comunidades.....	5, 6, 31, 32, 33, 34, 40, 41
conceptos	5
conexión.....	21, 36, 37, 38, 40
conocimiento.....	5, 6
consultas	39
contribución	38
correspondientes	26
costo.....	20, 21
creación	41
csv	26
cuántos	30, 38
cuesta	21

D

data	42
decisiones.....	5, 6, 42
después.....	18
destino	15
detección.....	5, 6, 31, 40, 41
directas	21, 35
dirigidas	16

distancia.....	15, 22, 23, 24, 25, 39
distancias	13
distante.....	19
distinta.....	34
distribución.....	21
diversos	5, 42

E

económicos.....	21
eficientes.....	21, 39
ejercicio	17, 29, 31, 35
ejercicios	39
elección.....	19
empresas	40
enfoque.....	25
enlaces.....	35, 40, 41
entorno	6, 8, 9
entrantes.....	29
escenarios	5
espacio.....	39
específicos	5
estación	13, 19, 20, 21
euros	15, 21
expansión	16
experiencia	40
exploración	25, 39

F

función.....	16, 17, 38, 40
fundamentales	5, 19

G

grafos	5, 6, 25, 32, 39, 41, 42
graph.....	42
grupos	32, 34

H

habilidades	5
herramientas	5, 42
heurística	25, 39

I

ideas.....	6
importancia	5
informática	5
inteligencia	5, 6
interacciones.....	6, 32, 34
intermedias.....	16

K

kilómetros 15, 22, 23, 24, 25

L

latitud 12, 24
 lenguajes 5
 lista 30, 33
 logística 6
 logísticas 40
 longitud 12, 24

M

manipulación 5
 mapas 5
 medidas 5
 métodos 35, 38, 40, 41, 42
 métricas 38
 mínimos 6, 22
 modelo 39
 modularidad 34
 mutuas 32

N

niveles 16
 nodos 12, 13, 14, 26, 27, 28, 29, 32, 34, 36,
 38, 39

O

objetivos 5
 opciones 21, 39
 operación 32
 orden 16, 30

P

pantalla 6, 7, 8, 9
 paradas 16, 39
 patrones 6, 37
 planificación 21, 42
 popularidad 31, 38, 40
 posibilidad 35, 38, 40
 posibles 25
 práctica 6
 prácticas 42
 prácticos 5, 42
 precio 15, 20, 21
 precios 14, 21
 preferencial 38

principales 5, 39
 probabilidad 37, 38, 40
 problemas 5, 6, 42
 profundidad 5, 16, 17, 18, 19
 prometedores 25
 propiedad 13
 proyección 32

R

reales 5
 recomendación 5, 39, 42
 recursos 6
 red 5, 28, 29, 30, 31, 34, 35, 36, 37, 38, 40
 redes 5, 6, 31, 32, 34, 40, 41, 42
 relación 27
 relevantes 40
 representativos 6
 resto 15, 19, 34
 resultante 16, 18
 rutas 5, 17, 21, 39, 40, 42

S

salientes 14
 science 42
 seguidores 29, 30, 31, 38
 sesión 9
 similitudes 38
 sistemas 5, 6, 42
 sistemático 25
 sociales 5, 6, 32, 34, 40, 41, 42

T

tabla 28, 30
 técnica 5
 teóricos 41
 términos 19, 39
 trenes 21

U

usuarios 26, 27, 28, 29, 30, 31, 32, 33, 34, 35,
 36, 38, 40
 útiles 40
 utilidad 5

V

vecinos 35, 36, 38
 viajeros 40
 viajes 21, 39
 volúmenes 5