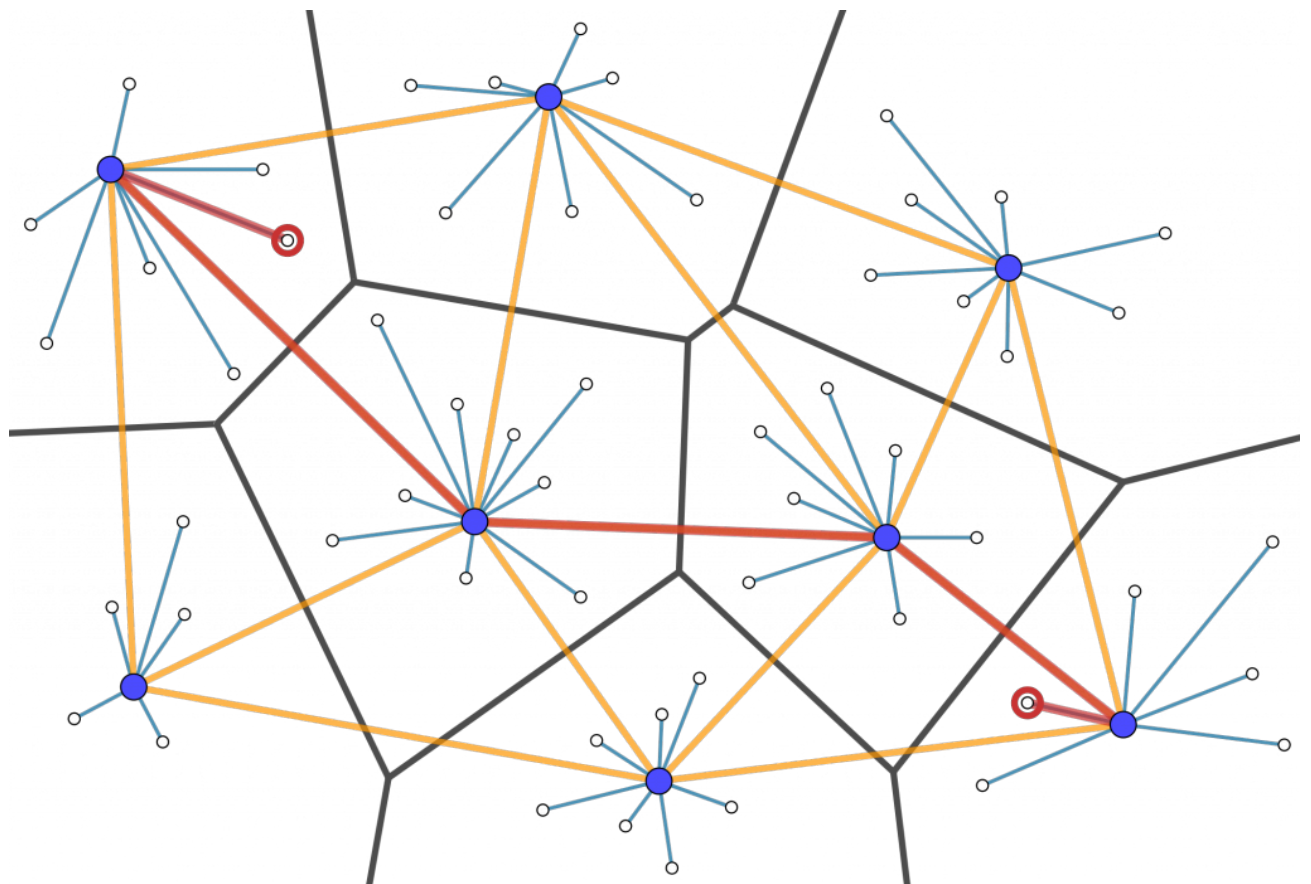


Actividad Grafos - Ciudades y RRSS



Adrián Yared Armas de la Nuez



Contenido

1. Características.....	3
2. Objetivos.....	3
3.2 Relaciones.....	4
3.2.1 Código.....	4
3.2.2 Prueba de ejecución.....	5
3.3 Recorrido en anchura (BFS).....	5
3.3.1 Código.....	5
3.3.2 Prueba de ejecución.....	5
3.3 Recorrido en profundidad (DFS).....	6
3.3.1 Código.....	6
3.3.2 Prueba de ejecución.....	6
4 Enunciado.....	6
4.1 Crear el grafo en memoria.....	7
4.1.1 Código.....	7
4.1.2 Prueba de ejecución.....	7
4.2 Calcular el Camino Mínimo entre Dos Ciudades usando Dijkstra.....	7
4.2.1 Código.....	7
4.2.2 Prueba de ejecución.....	8
4.3 Repetir y modificar las relaciones si es necesario.....	8
4.3.1 Ejemplo Código.....	8
4.3.2 Ejemplo Prueba de ejecución.....	9
4.3.3 Ejemplo Código.....	9
4.3.4 Prueba de ejecución.....	10
4.4 Eliminamos el grafo de la memoria.....	11
4.4.1 Código.....	11
4.4.2 Prueba de ejecución.....	11
5 Análisis de Red Social en Neo4j.....	11
5.1 Objetivo.....	11
5.2.1 Código.....	12
5.2.2 Prueba de ejecución.....	12
3.2.3 Código.....	13
5.2.4 Prueba de ejecución.....	13
5.2.5 Código.....	13
5.2.6 Prueba de ejecución.....	14
5.2.7 Código.....	14
5.2.8 Prueba de ejecución.....	15
5.2.9 Código.....	15




Actividad Grafos - Ciudades y RRSS


5.2.10 Prueba de ejecución.....	16
5.2.11 Aplicar Algoritmos de Análisis.....	16
5.2.11.1 Centralidad de Grado.....	16
5.2.11.1.1 Propósito.....	16
5.2.11.1.2 Código.....	16
5.2.11.1.3 Ejecución.....	17
5.2.11.2 Centralidad de PageRank.....	17
5.2.11.2.1 Propósito.....	17
5.2.11.2.3 Código.....	17
5.2.11.2.4 Ejecución.....	18
5.2.11.3 Detección de Comunidades con Louvain.....	18
5.2.11.3.1 Propósito.....	18
5.2.11.3.3 Código.....	18
5.2.11.3.4 Ejecución.....	19
5.2.11.4 Camino Mínimo entre Dos Usuarios (Dijkstra).....	19
5.2.11.4.1 Propósito.....	19
5.2.11.4.3 Código.....	19
3.2.11.4.4 Ejecución.....	20
5.2.11.5 Existencia de Amistades Comunes.....	20
5.2.11.5.1 Propósito.....	20
5.2.11.5.3 Código.....	20
5.2.11.5.4 Ejecución.....	21
5.2.12 Eliminar el Grafo en Memoria (Opcional).....	21
5.2.12.1 Código.....	21
5.2.12.2 Ejecución.....	21
6 Resumen de los Algoritmos.....	22


1. Características


Características de la base de datos: - Versión 5.1.0 - Librerías instaladas: APOC (5.1.0), Graph Data Science Library (2.4.6)

Versión y nombre de la bd:

 Grafos tema 1 5.1.0 ● ACTIVE

 system

 adrianyaredarmasdelanuez

 neo4j (default)

Plugins:

› APOC ✓ 5.1.0

› Graph Data Science Library ✓ 2.4.6

2. Objetivos

Utilizar diferentes algoritmos de optimización de rutas en Neo4j con el fin de:

- Determinar las rutas más cortas entre dos puntos.
- Identificar las rutas con menor costo (por ejemplo, distancia o tiempo de viaje).
- Encontrar las rutas más eficientes considerando restricciones específicas (por ejemplo, evitar ciertas áreas, cumplir con límites de tiempo, etc.).
- Analizar el tráfico y flujo de movimiento entre diferentes nodos o ubicaciones.
- Visualizar y explorar las redes de rutas de manera interactiva.

```
$ // Creación de los nodos (ciudades) CREATE (Madrid:City {name: 'Mad... ▶ ☆
```

adrianyaredarmasdelanuez\$ CREATE (Madrid:City {name: 'Madrid'})	✓
adrianyaredarmasdelanuez\$ CREATE (Barcelona:City {name: 'Barcelona'})	✓
adrianyaredarmasdelanuez\$ CREATE (Valencia:City {name: 'Valencia'})	✓
adrianyaredarmasdelanuez\$ CREATE (Sevilla:City {name: 'Sevilla'})	✓
adrianyaredarmasdelanuez\$ CREATE (Granada:City {name: 'Granada'})	✓
adrianyaredarmasdelanuez\$ CREATE (Bilbao:City {name: 'Bilbao'})	✓
adrianyaredarmasdelanuez\$ CREATE (Zaragoza:City {name: 'Zaragoza'})	✓
adrianyaredarmasdelanuez\$ CREATE (Malaga:City {name: 'Malaga'})	✓

3.2 Relaciones

3.2.1 Código

```
// Creación de las relaciones (distancias en km) solo si no existen
MATCH (Madrid:City {name: 'Madrid'}), (Barcelona:City {name: 'Barcelona'})
MERGE (Madrid)-[:ROAD {distance: 620}]->(Barcelona);
MATCH (Madrid:City {name: 'Madrid'}), (Valencia:City {name: 'Valencia'})
MERGE (Madrid)-[:ROAD {distance: 350}]->(Valencia);
MATCH (Madrid:City {name: 'Madrid'}), (Sevilla:City {name: 'Sevilla'})
MERGE (Madrid)-[:ROAD {distance: 530}]->(Sevilla);
MATCH (Madrid:City {name: 'Madrid'}), (Granada:City {name: 'Granada'})
MERGE (Madrid)-[:ROAD {distance: 395}]->(Granada);
MATCH (Barcelona:City {name: 'Barcelona'}), (Zaragoza:City {name: 'Zaragoza'})
MERGE (Barcelona)-[:ROAD {distance: 300}]->(Zaragoza);
MATCH (Zaragoza:City {name: 'Zaragoza'}), (Bilbao:City {name: 'Bilbao'})
MERGE (Zaragoza)-[:ROAD {distance: 300}]->(Bilbao);
MATCH (Sevilla:City {name: 'Sevilla'}), (Malaga:City {name: 'Malaga'})
MERGE (Sevilla)-[:ROAD {distance: 210}]->(Malaga);
MATCH (Granada:City {name: 'Granada'}), (Malaga:City {name: 'Malaga'})
MERGE (Granada)-[:ROAD {distance: 125}]->(Malaga);
MATCH (Valencia:City {name: 'Valencia'}), (Barcelona:City {name: 'Barcelona'})
MERGE (Valencia)-[:ROAD {distance: 350}]->(Barcelona);
```

3.2.2 Prueba de ejecución

\$ // Creación de las relaciones (distancias en km) solo si no existen...

adrianyaredarmasdelanuez\$ MATCH (Madrid:City {name: 'Madrid'}), (Barc...	✓
adrianyaredarmasdelanuez\$ MATCH (Madrid:City {name: 'Madrid'}), (Vale...	✓
adrianyaredarmasdelanuez\$ MATCH (Madrid:City {name: 'Madrid'}), (Sevi...	✓
adrianyaredarmasdelanuez\$ MATCH (Madrid:City {name: 'Madrid'}), (Gran...	✓
adrianyaredarmasdelanuez\$ MATCH (Barcelona:City {name: 'Barcelona'}), ...	✓
adrianyaredarmasdelanuez\$ MATCH (Zaragoza:City {name: 'Zaragoza'}), (...)	✓
adrianyaredarmasdelanuez\$ MATCH (Sevilla:City {name: 'Sevilla'}), (Ma...	✓
adrianyaredarmasdelanuez\$ MATCH (Granada:City {name: 'Granada'}), (Ma...	✓
adrianyaredarmasdelanuez\$ MATCH (Valencia:City {name: 'Valencia'}), (...)	✓

3.3 Recorrido en anchura (BFS)

3.3.1 Código

//Recorrido en anchura

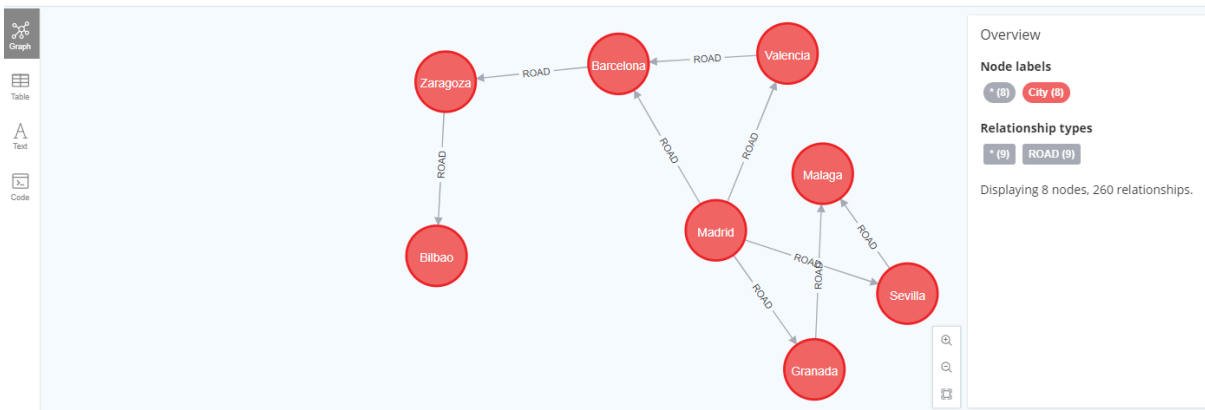
MATCH (start:City {name: 'Madrid'})

CALL apoc.path.expandConfig(start, {maxLevel: 10, bfs: true}) YIELD path

RETURN path

3.3.2 Prueba de ejecución

adrianyaredarmasdelanuez\$ MATCH (start:City {name: 'Madrid'}) CALL apoc.path.expandConfig(start, {maxLevel: 10, bfs: true}) YIELD pat...



Overview

Node labels

- * (8) City (8)

Relationship types

- * (9) ROAD (9)

Displaying 8 nodes, 260 relationships.

3.3 Recorrido en profundidad (DFS)

3.3.1 Código

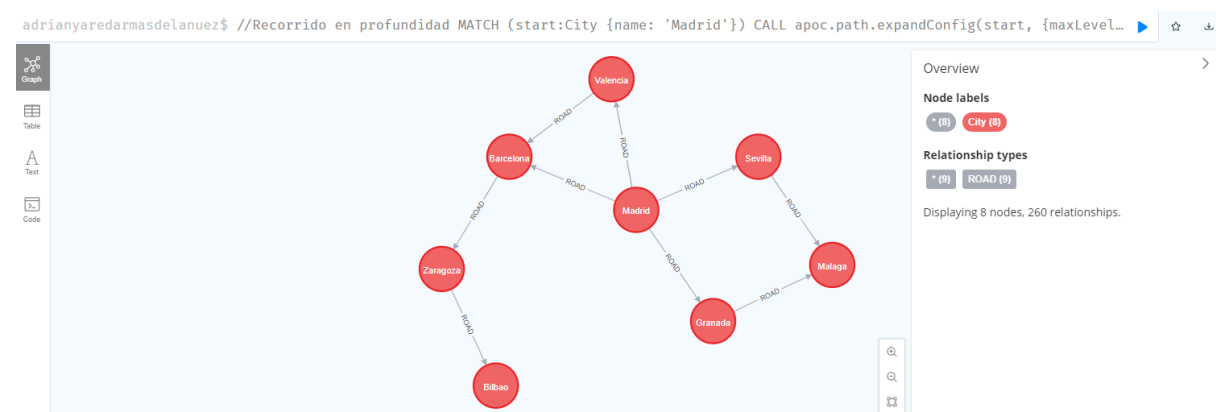
//Recorrido en profundidad

MATCH (start:City {name: 'Madrid'})

CALL apoc.path.expandConfig(start, {maxLevel: 10, bfs: false}) YIELD path

RETURN path

3.3.2 Prueba de ejecución



4 Enunciado

Para los siguientes algoritmos será necesario cargar el grafo en memoria. Esta es una

práctica común al trabajar con el plugin Graph Data Science (GDS) en Neo4j porque permite ejecutar algoritmos de manera más eficiente y rápida. Estos son algunos motivos principales para crear el grafo en memoria: en memoria ya que nos permite:

1. Rendimiento Mejorado: Los algoritmos de GDS, como Dijkstra, PageRank, o Louvain, están optimizados para funcionar en grafos en memoria. Al proyectar el grafo en memoria, Neo4j utiliza estructuras de datos altamente optimizadas que permiten realizar cálculos complejos a gran velocidad en comparación con operar directamente sobre los nodos y relaciones de la base de datos.

2. Separación de la Estructura del Grafo: Al crear un grafo en memoria, puedes definir qué nodos, relaciones y propiedades incluir. Esto es útil cuando solo necesitas trabajar con una parte específica del grafo, como una subestructura (por ejemplo, solo las ciudades y las carreteras en un grafo grande con muchos tipos de nodos y relaciones). De esta forma, no es necesario alterar la estructura de la base de datos original.

3. Control sobre Propiedades y Configuraciones: Proyectar un grafo en memoria permite definir propiedades específicas, como `relationshipWeightProperty`, para

usarlas en algoritmos. Esto facilita la configuración de los pesos o atributos específicos que el algoritmo necesita sin modificar los datos originales.

4. Almacenamiento Temporal y Flexibilidad: Los grafos en memoria son temporales y pueden ser eliminados fácilmente con `CALL gds.graph.drop()`. Esto permite probar diferentes configuraciones de grafos o algoritmos sin comprometer la integridad de los datos originales.

5. Soporte para Algoritmos Complejos: Muchos algoritmos avanzados, como Dijkstra y A*, solo están disponibles a través de GDS en grafos proyectados en memoria. Si intentas ejecutar estos algoritmos directamente sobre los nodos y relaciones en la base de datos, no tendrías acceso a las versiones optimizadas de GDS.

4.1 Crear el grafo en memoria

4.1.1 Código

```
CALL gds.graph.project( 'cityGraph', 'City', { ROAD: { type: 'ROAD', properties: 'distance' } } )
```

4.1.2 Prueba de ejecución

```
adrianyaredarmasdelanuez$ CALL gds.graph.project( 'cityGraph', 'City', { ROAD: { type: 'ROAD', properties: 'distance' } } )
```

	nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount
1	<pre>{ "City": { "label": "City", "properties": { } } }</pre>	<pre>{ "ROAD": { "orientation": "NATURAL", "indexInverse": false, "aggregation": "DEFAULT", "type": "ROAD", "properties": { "distance": { "defaultValue": null, "property": "distance", "aggregation": "DEFAULT" } } } }</pre>	"cityGraph"	8	9

4.2 Calcular el Camino Mínimo entre Dos Ciudades usando Dijkstra

Con el grafo `cityGraph` proyectado, calcularemos el camino mínimo entre dos ciudades específicas, como Madrid y Bilbao, usando el algoritmo de Dijkstra

4.2.1 Código

```
MATCH (start:City {name: 'Madrid'}), (end:City {name: 'Bilbao'})
CALL gds.shortestPath.dijkstra.stream('cityGraph', {
  sourceNode: id(start),
```



```
targetNode: id(end),
relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs
RETURN gds.util.asNode(sourceNode).name AS startCity,
gds.util.asNode(targetNode).name AS endCity,
totalCost
```

4.2.2 Prueba de ejecución

adrianyaredarmasdelanuez\$ MATCH (start:City {name: 'Madrid'}), (end:City {name: 'Bilbao'}) CALL gds.shortestPath.dijk

	startCity	endCity	totalCost
1	"Madrid"	"Bilbao"	1220.0

4.3 Repetir y modificar las relaciones si es necesario

Realizar la misma operación entre las ciudades de Málaga y Zaragoza. Modifique las relaciones si fuese necesario.

Cálculo de Caminos Mínimos entre Todos los Pares de Nodos

4.3.1 Ejemplo Código

```
CALL gds.alpha.allShortestPaths.stream('cityGraph', {
relationshipWeightProperty: 'distance'
})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE gds.util.isFinite(distance) = true
WITH gds.util.asNode(sourceNodeId) AS source, gds.util.asNode(targetNodeId) AS
target, distance
WHERE source <> target
RETURN source.name AS startCity, target.name AS endCity, distance AS totalCost
ORDER BY totalCost ASC, startCity ASC, endCity ASC
LIMIT 10
```

4.3.2 Ejemplo Prueba de ejecución

adrianyaredarmasdelanuez\$ CALL gds.alpha.allShortestPaths.stream('cityGraph', { relationshipWeightProperty: 'distance' }) YIELD source...

	startCity	endCity	totalCost
1	"Granada"	"Malaga"	125.0
2	"Sevilla"	"Malaga"	210.0
3	"Barcelona"	"Zaragoza"	300.0
4	"Zaragoza"	"Bilbao"	300.0
5	"Madrid"	"Valencia"	350.0
6	"Valencia"	"Barcelona"	350.0
7	"Madrid"	"Granada"	395.0
8	"Madrid"	"Malaga"	520.0
9	"Madrid"	"Sevilla"	530.0
10	"Barcelona"	"Bilbao"	600.0

4.3.3 Ejemplo Código

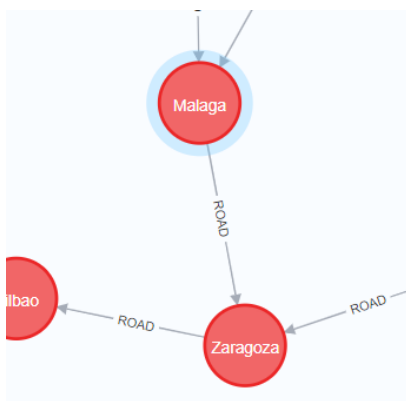
//Creo la relacion entre Málaga y Zaragoza

MATCH (Malaga:City {name: 'Malaga'}), (Zaragoza:City {name: 'Zaragoza'})

MERGE (Malaga)-[:ROAD {distance: 125}]->(Zaragoza);

```
MATCH (Malaga:City {name: 'Malaga'}), (Zaragoza:City {name: 'Zaragoza'})
MERGE (Malaga)-[:ROAD {distance: 125}]->(Zaragoza);
```

Set 1 property, created 1 relationship, completed after 8 ms.



Elimino el graph actual:

```
CALL gds.graph.drop('cityGraph');
```

	graphName	database	memoryUsage	sizeInBytes	nodeCount	relationshipCount	configuration	density
1	"cityGraph"	"adrianyaredarmasdelanuez"	""	-1	8	9	{ "relationshipProjection": { "ROAD": { "orientation": "NATURAL", "indexInverse": false, "aggregation": "DEFAULT", "type": "ROAD", "properties": { "distance": { "defaultValue": null, "property": "distance", "aggregation": "DEFAULT" } } } } }	0.1607142857142

4.3.4 Prueba de ejecución

Muestro la relación:

*Este código realiza una consulta para obtener los caminos más cortos entre nodos en un grafo llamado **cityGraph**, utilizando el peso de la relación 'distance'. Filtra los resultados donde la distancia es finita y excluye los caminos donde el nodo de origen es igual al de destino, y para terminar, devuelve las ciudades de inicio y fin, junto con el costo total del camino más corto filtrado por costo y limitado a los 10 caminos más cortos.*

//Código:

```
CALL gds.alpha.allShortestPaths.stream('cityGraph', {
  relationshipWeightProperty: 'distance'
})
YIELD sourceNodeId, targetNodeId, distance
WITH sourceNodeId, targetNodeId, distance
WHERE gds.util.isFinite(distance) = true
WITH gds.util.asNode(sourceNodeId) AS source, gds.util.asNode(targetNodeId) AS
target, distance
WHERE source <> target
RETURN source.name AS startCity, target.name AS endCity, distance AS totalCost
ORDER BY totalCost ASC, startCity ASC, endCity ASC
LIMIT 10
```

Ejecución:

```
adrianyaredarmasdelanuez$ CALL gds.alpha.allShortestPaths.stream('cityGraph', { relationshipWeightProperty: 'distance' }) YIELD
```

	startCity	endCity	totalCost
1	"Granada"	"Málaga"	125.0
2	"Málaga"	"Zaragoza"	125.0

Como he explicado anteriormente he añadido la relación entre Málaga y Zaragoza y con el posterior código la he mostrado, tiene una distancia finita y en el orden de costo se encuentra el segundo empatado por el primero.

4.4 Eliminamos el grafo de la memoria

4.4.1 Código

```
CALL gds.graph.drop('socialGraph');
```

4.4.2 Prueba de ejecución

```
adrianyaredarmasdelanuez$ CALL gds.graph.drop('cityGraph');
```

	graphName	database	memoryUsage	sizeInBytes	nodeCount	relationshipCount	configuration
1	"cityGraph"	"adrianyaredarmasdelanuez"	""	-1	8	10	{ "relationshipProjection": { "ROAD": { "orientation": "NATURAL", "indexInverse": false, "aggregation": "DEFAULT", "type": "ROAD", "properties": { "distance": { "defaultValue": null, "property": "distance", "aggregation": "DEFAULT" } } } } }

arted streaming 1 records after 1 ms and completed after 2 ms.

5 Análisis de Red Social en Neo4j

Características de la base de datos:

- Versión 5.1.0
- Librerías instaladas: APOC (5.1.0), Graph Data Science Library (2.4.6)

5.1 Objetivo

Crear un grafo de 25 usuarios en una red social con relaciones de amistad ponderadas

(weight). Usaremos varios algoritmos de análisis de grafos para estudiar la estructura de la red y la influencia de los usuarios.

5.2.1 Código

```
//Creación de los nodos de usuarios
CREATE (Alice:User {name: 'Alice'}),
(Bob:User {name: 'Bob'}),
(Charlie:User {name: 'Charlie'}),
(David:User {name: 'David'}),
(Emma:User {name: 'Emma'}),
(Frank:User {name: 'Frank'}),
(Grace:User {name: 'Grace'}),
(Henry:User {name: 'Henry'}),
(Ivy:User {name: 'Ivy'}),
(Jack:User {name: 'Jack'}),
(Karen:User {name: 'Karen'}),
(Leo:User {name: 'Leo'}),
(Mona:User {name: 'Mona'}),
(Nina:User {name: 'Nina'}),
(Oscar:User {name: 'Oscar'}),
(Paul:User {name: 'Paul'}),
(Quinn:User {name: 'Quinn'}),
(Rose:User {name: 'Rose'}),
(Sam:User {name: 'Sam'}),
(Tina:User {name: 'Tina'}),
(Uma:User {name: 'Uma'}),
(Victor:User {name: 'Victor'}),
(Wendy:User {name: 'Wendy'}),
(Xander:User {name: 'Xander'}),
(Yara:User {name: 'Yara'})
```

5.2.2 Prueba de ejecución

```
adrianyaredarmasdelanuez$ //Creación de los nodos de usuarios CREATE
```



Added 25 labels, created 25 nodes, set 25 properties, completed after 14 ms.

3.2.3 Código

Creamos relaciones FRIEND entre los usuarios y les asignamos un peso (weight). Este peso podría representar, por ejemplo, la cercanía o frecuencia de interacción entre los usuarios.

```
//Crear Relaciones de Amistad con Peso
```

```
MATCH (a:User {name: 'Alice'}), (b:User {name: 'Bob'}), (c:User {name: 'Charlie'}),  
(d:User {name: 'David'}), (e:User {name: 'Emma'}), (f:User {name: 'Frank'}),
```

```
(g:User {name: 'Grace'}), (h:User {name: 'Henry'}), (i:User {name: 'Ivy'}),  
(j:User {name: 'Jack'}), (k:User {name: 'Karen'}), (l:User {name: 'Leo'}),  
(m:User {name: 'Mona'}), (n:User {name: 'Nina'}), (o:User {name: 'Oscar'}),  
(p:User {name: 'Paul'}), (q:User {name: 'Quinn'}), (r:User {name: 'Rose'}),  
(s:User {name: 'Sam'}), (t:User {name: 'Tina'}), (u:User {name: 'Uma'}),  
(v:User {name: 'Victor'}), (w:User {name: 'Wendy'}), (x:User {name: 'Xander'}),  
(y:User {name: 'Yara'})
```

```
// Crear relaciones de amistad con peso
```

```
CREATE (a)-[:FRIEND {weight: 1.0}]->(b), (a)-[:FRIEND {weight: 2.5}]->(c),  
(a)-[:FRIEND {weight: 1.2}]->(d),  
(b)-[:FRIEND {weight: 3.0}]->(e), (b)-[:FRIEND {weight: 0.8}]->(f), (c)-[:FRIEND  
{weight: 2.3}]->(g),  
(c)-[:FRIEND {weight: 1.7}]->(h), (d)-[:FRIEND {weight: 1.5}]->(i), (d)-[:FRIEND  
{weight: 2.1}]->(j),  
(e)-[:FRIEND {weight: 2.0}]->(k), (e)-[:FRIEND {weight: 3.1}]->(l), (f)-[:FRIEND  
{weight: 1.4}]->(m),  
(f)-[:FRIEND {weight: 2.0}]->(n), (g)-[:FRIEND {weight: 1.9}]->(o), (h)-[:FRIEND  
{weight: 0.9}]->(p);
```

5.2.4 Prueba de ejecución

```
adrianyaredarmasdelanuez$ MATCH (a:User {name: 'Alice'}), (b:User {name: 'Bob'}),
```



Set 15 properties, created 15 relationships, completed after 41 ms.

5.2.5 Código

//Como se podrá comprobar, existen varios nodos aislados. Para verificar cuáles son

//debemos utilizar este código:

```
MATCH (n:User)  
WHERE NOT (n)--()  
RETURN n.name AS isolatedNode;
```

5.2.6 Prueba de ejecución

```
adrianyaredarmasdelanuez$ MATCH (n:User) WHERE NOT (n)--() RETU
```

Table	isolatedNode
1	"Quinn"
2	"Rose"
3	"Sam"
4	"Tina"
5	"Uma"
6	"Victor"
7	
7	"Wendy"
8	"Xander"
9	"Yara"

5.2.7 Código

```
// Conectar los nodos aislados con `Alice`
MATCH (a:User {name: 'Alice'}),
(q:User {name: 'Quinn'}),
(r:User {name: 'Rose'}),
(s:User {name: 'Sam'}),
(t:User {name: 'Tina'}),
```

```
(u:User {name: 'Uma'}),
(v:User {name: 'Victor'}),
(w:User {name: 'Wendy'}),
(x:User {name: 'Xander'}),
(y:User {name: 'Yara'})
CREATE (a)-[:FRIEND {weight: 1.0}]->(q),
(a)-[:FRIEND {weight: 1.0}]->(r),
(a)-[:FRIEND {weight: 1.0}]->(s),
(a)-[:FRIEND {weight: 1.0}]->(t),
(a)-[:FRIEND {weight: 1.0}]->(u),
(a)-[:FRIEND {weight: 1.0}]->(v),
(a)-[:FRIEND {weight: 1.0}]->(w),

(a)-[:FRIEND {weight: 1.0}]->(x),
(a)-[:FRIEND {weight: 1.0}]->(y);
```

5.2.8 Prueba de ejecución

Conecta los nodos anteriormente aislados que como se puede ver en el apartado anterior son 9, al igual que los que se unen aquí.

```
adrianyaredarmasdelanuez$ // Conectar los nodos aislados con `Alice` MATCH (a:User
```



Set 9 properties, created 9 relationships, completed after 18 ms.

5.2.9 Código

```
//Proyecta el grafo en memoria
CALL gds.graph.project(
'socialGraph',
'User',
{
    FRIEND: {
        type: 'FRIEND',
        properties: 'weight'
    }
});
```


5.2.10 Prueba de ejecución

```
adrianyaredarmasdelanuez$ //Proyecta el grafo en memoria CALL gds.graph.project( 'so
```

	nodeProjection	relationshipProjection	graphName	nodeCount
1	<pre>{ "User": { "label": "User", "properties": { // ... } } }</pre>	<pre>{ "FRIEND": { "orientation": "NATURAL", "indexInverse": false, "aggregation": "DEFAULT", "type": "FRIEND", "properties": { "weight": { "defaultValue": null, "property": "weight", "aggregation": "DEFAULT" } } } }</pre>	"socialGraph"	25

Started streaming 1 records after 7 ms and completed after 13 ms.

5.2.11 Aplicar Algoritmos de Análisis

5.2.11.1 Centralidad de Grado

5.2.11.1.1 Propósito

Identificar los nodos con el mayor número de conexiones directas, lo que representa su popularidad o nivel de actividad.

5.2.11.1.2 Código

Este código consulta un grafo llamado "socialGraph" usando el algoritmo de grado de nodos (degree) de Neo4j. Devuelve los 10 nodos con el mayor grado de conexión, mostrando el nombre del nodo (usuario) y su puntuación de grado. Los resultados se ordenan por el grado en orden descendente.

```
CALL gds.degree.stream('socialGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS degree
ORDER BY degree DESC
LIMIT 10;
```

5.2.11.1.3 Ejecución

```
adrianyaredarmasdelanuez$ CALL gds.degree.stream('socialGraph') YIELD r
```

	user	degree
1	"Alice"	12.0
2	"Frank"	2.0
3	"Bob"	2.0
4	"Emma"	2.0
5	"David"	2.0
6	"Charlie"	2.0
7		

Como podemos observar devuelve los nodos por orden de conexión, siendo el primero el mayor con 12 y precedido por múltiples resultados con 2 de puntuación.

5.2.11.2 Centralidad de PageRank

5.2.11.2.1 Propósito

Identificar los nodos con el mayor número de conexiones directas, lo que representa su popularidad o nivel de actividad.

5.2.11.2.3 Código

Este código ejecuta el algoritmo de PageRank sobre un grafo llamado 'socialGraph' que devuelve el identificador de los nodos junto con su puntuación de PageRank, ordenados de mayor a menor. Finalmente, limita el resultado a los 10 nodos con las mayores puntuaciones, mostrando el nombre de los usuarios y su puntuación.

```
CALL gds.pageRank.stream('socialGraph')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS user, score AS pageRank
ORDER BY pageRank DESC
LIMIT 10;
```

5.2.11.2.4 Ejecución

```
adrianyaredarmasdelanuez$ CALL gds.pageRank.stream('socialGraph
```

	user	pageRank
1	"Paul"	0.33552578125000004
2	"Oscar"	0.33552578125000004
3	"Mona"	0.24276289062500003
4	"Karen"	0.24276289062500003
5	"Nina"	0.24276289062500003
6	"Leo"	0.24276289062500003
7		

Como podemos ver muestra la puntuación PageRank de mayor a menor tal como indica el código.

5.2.11.3 Detección de Comunidades con Louvain

5.2.11.3.1 Propósito

Agrupar los nodos en comunidades basadas en la modularidad de sus conexiones, identificando subgrupos en la red.

5.2.11.3.3 Código

Este código utiliza el algoritmo Louvain para detectar comunidades en un grafo llamado 'socialGraph'. Devuelve los nodos junto con sus respectivas comunidades, ordenados por el ID de comunidad y limitados a los primeros 20 resultados.

```
CALL gds.louvain.stream('socialGraph')
YIELD nodeId, communityId
RETURN gds.util.asNode(nodeId).name AS user, communityId
ORDER BY communityId
LIMIT 20;
```

5.2.11.3.4 Ejecución

```
adrianyaredarmasdelanuez$ CALL gds.louvain.stream('socialGraph')
```

	user	communityId
1	"Jack"	9
2	"David"	9
3	"Ivy"	9
4	"Leo"	13
5	"Frank"	13
6	"Mona"	13
7		

Started streaming 20 records after 7 ms and completed after 198 ms.

Como dice el código, devuelve los usuarios filtrados por puntuación de comunidad de menor a mayor.

5.2.11.4 Camino Mínimo entre Dos Usuarios (Dijkstra)

5.2.11.4.1 Propósito

Calcular la ruta más corta entre dos usuarios, tomando en cuenta la propiedad weight de las relaciones.

5.2.11.4.3 Código

Este código encuentra el camino más corto entre los usuarios Alice y Karen, en "socialGraph" utilizando el algoritmo de Dijkstra, considerando el peso de las relaciones entre ellos. Luego, devuelve los nombres de los usuarios y el costo total del camino más corto entre ellos.

```
MATCH (start:User {name: 'Alice'}), (end:User {name: 'Karen'})
CALL gds.shortestPath.dijkstra.stream('socialGraph', {
  sourceNode: id(start),
  targetNode: id(end),
```

relationshipWeightProperty: 'weight'

}}

YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs

RETURN gds.util.asNode(sourceNode).name AS startUser,

gds.util.asNode(targetNode).name AS endUser,

totalCost;

3.2.11.4.4 Ejecución

```
adrianyaredarmasdelanuez$ MATCH (start:User {name: 'Alice'}), (end:User {name: 'Karen'}) CALL g
```

	startUser	endUser	totalCost
1	"Alice"	"Karen"	6.0

Aquí muestra el resultado del costo total mínimo entre las relaciones de "Alice" y "karen", cual es 6.0.

5.2.11.5 Existencia de Amistades Comunes

5.2.11.5.1 Propósito

Mostrar pares de usuarios que tienen amigos en común

5.2.11.5.3 Código

En este apartado, consulta una base de datos de grafos para encontrar los pares de usuarios con más amigos en común, excluyendo a los usuarios que son ellos mismos. Devuelve los 10 pares con más amigos comunes, ordenados de mayor a menor.

```
MATCH (a:User)-[:FRIEND]-(commonFriend)-[:FRIEND]-(b:User)
```

```
WHERE a <> b
```

```
RETURN a.name AS user1, b.name AS user2, count(commonFriend) AS  
commonFriends
```

```
ORDER BY commonFriends DESC
```

```
LIMIT 10;
```

5.2.11.5.4 Ejecución

```
adrianyaredarmasdelanuez$ MATCH (a:User)-[:FRIEND]-(commonFriend)-[:FRIEND]-(b:User)
```

	user1	user2	commonFriends
1	"Ivy"	"Alice"	1
2	"Henry"	"Alice"	1
3	"Grace"	"Alice"	1
4	"Frank"	"Alice"	1
5	"Emma"	"Alice"	1
6	"Nina"	"Bob"	1
7			

Started streaming 10 records after 8 ms and completed after 9 ms.

En esta lista muestra los que tienen más amigos en común, en este caso todos tienen 1.

5.2.12 Eliminar el Grafo en Memoria (Opcional)

Una vez completado el análisis, eliminamos el grafo en memoria para liberar recursos.

5.2.12.1 Código

El código `CALL gds.graph.drop('socialGraph');` elimina un grafo llamado "socialGraph" en la base de datos de Neo4j, liberando los recursos asociados a él.

```
CALL gds.graph.drop('socialGraph');
```

5.2.12.2 Ejecución

```
adrianyaredarmasdelanuez$ CALL gds.graph.drop('socialGraph');
```

	graphName	database	memoryUsage	sizeInBytes	nodeCount	relationshipCount	configuration
1	"socialGraph"	"adrianyaredarmasdelanuez"	""	-1	25	24	{ "relationshipProjection": { "FRIEND": { "orientation": "NATURAL", "indexInverse": false, "aggregation": "DEFAULT", "type": "FRIEND", "properties": { "weight": { "defaultValue": null, "property": "weight", "aggregation": "DEFAU } } } } }

6 Resumen de los Algoritmos

- Centralidad de Grado: Mide la popularidad o actividad de los usuarios en la red.
- Centralidad de PageRank: Determina la importancia de los usuarios en función de la estructura de sus conexiones.
- Detección de Comunidades: Identifica subgrupos o comunidades en la red.
- Camino Mínimo: Encuentra la ruta más corta entre dos usuarios, considerando el peso de las relaciones.
- Existencia de Amistades Comunes: Propósito: Mostrar pares de usuarios que tienen amigos en común.