Análisis de Rutas en Neo4j

Javier Díaz Machado

ÍNDICE

Ejecución de grafo de ciudades

Objetivo

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

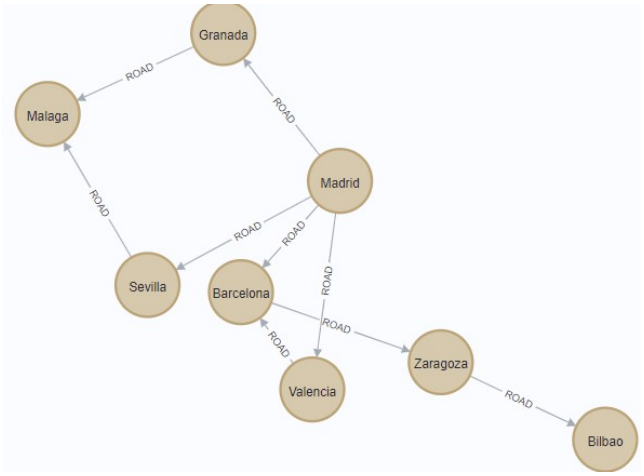
1. Determinar las rutas más cortas entre dos puntos.
2. Identificar las rutas con menor costo (por ejemplo, distancia o tiempo de viaje).
3. Encontrar las rutas más eficientes considerando restricciones específicas (por ejemplo, evitar ciertas áreas, cumplir con límites de tiempo, etc.).
4. Analizar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

Crear el grafo:

```
CREATE (Madrid:City {name: 'Madrid'});
CREATE (Barcelona:City {name: 'Barcelona'});
CREATE (Valencia:City {name: 'Valencia'});
CREATE (Sevilla:City {name: 'Sevilla'});
CREATE (Granada:City {name: 'Granada'});
CREATE (Bilbao:City {name: 'Bilbao'});
CREATE (Zaragoza:City {name: 'Zaragoza'});
CREATE (Malaga:City {name: 'Malaga'});
```

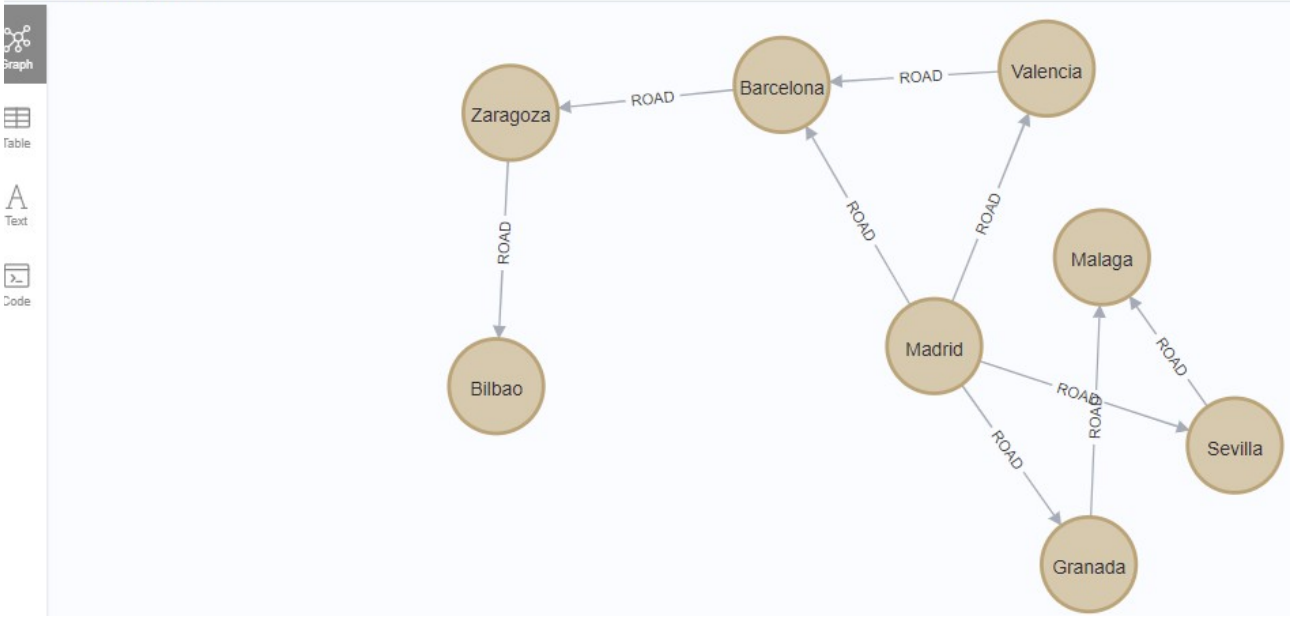
Creación de las relaciones en el caso de que no existan:

```
// 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'})
```



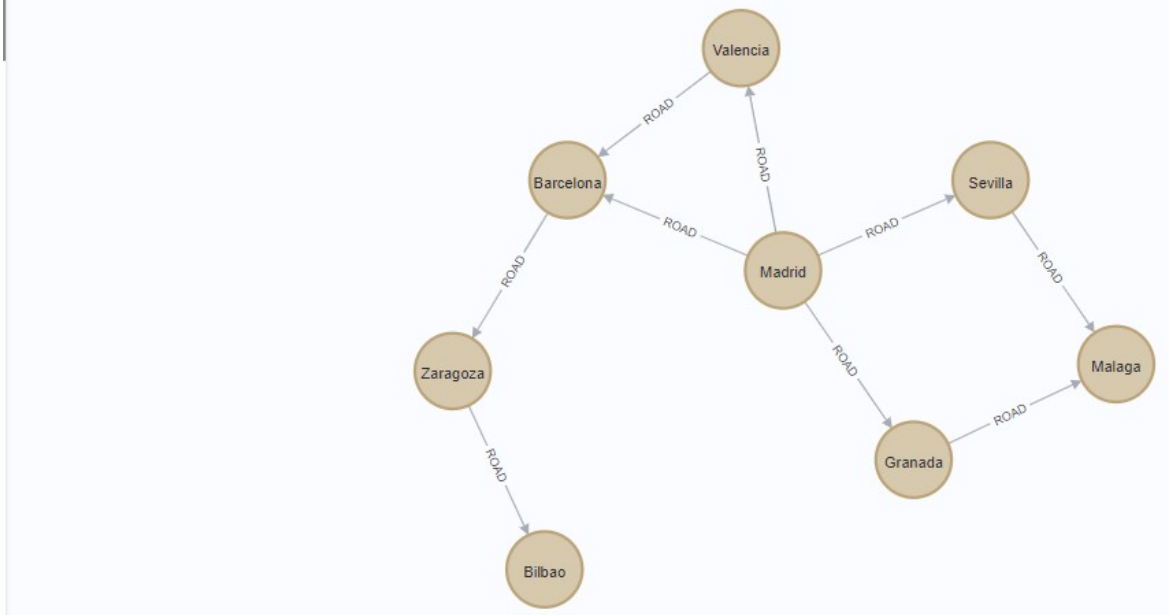
Recorrido en anchura (BFS):

```
// Recorrido en anchura (BFS)
MATCH(start:City {name: 'Madrid'})
CALL apoc.path.expandConfig(start, {maxLevel: 10, bfs: true}) YIELD path
RETURN path
```



Recorrido en profundidad (DFS):

```
// Recorrido en profundidad (DFS)
MATCH(start:City {name: 'Madrid'})
CALL apoc.path.expandConfig(start, {maxLevel: 10, bfs: false}) YIELD path
RETURN path
```



Crear el grafo en memoria:

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

Table | Text | Code

nodeProjection

```
{
  "City": {
    "label": "City",
    "properties": {

    }
  }
}
```

relationshipProjection

```
{
  "ROAD": {
    "orientation": "NATURAL",
    "indexInverse": false,
    "aggregation": "DEFAULT",
    "type": "ROAD",
    "properties": {
      "distance": {
        "defaultValue": null,
        "property": "distance",
        "aggregation": "DEFAULT"
      }
    }
  }
}
```

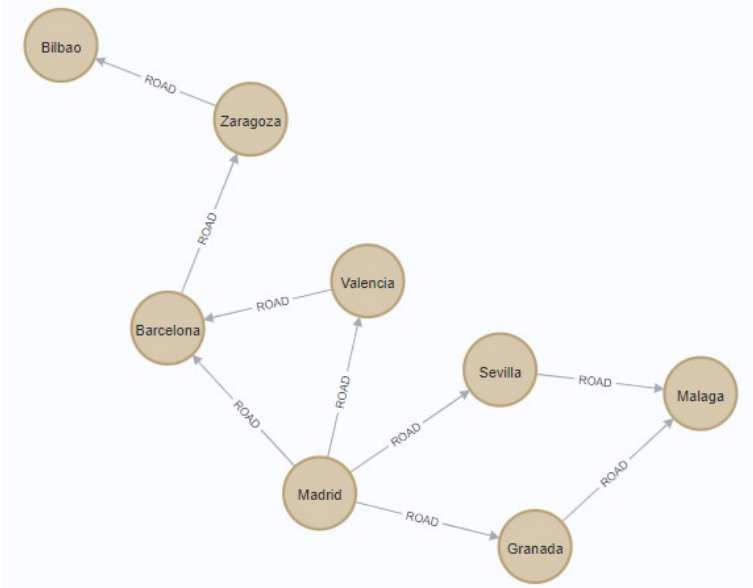
Calcular el camino mínimo entre dos ciudades usando Dijkstra (Málaga y Zaragoza):

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

Table

(no changes, no records)

Esto no nos devuelve ningún resultado, porque como podemos ver en el grafo, es imposible ir de Málaga a Zaragoza y vicebersa.



De modo que creamos una relación para hacer que ir de Zaragoza a Malaga sea posible:

```
MATCH(Zaragoza:City {name: 'Zaragoza'}), (Madrid:City {name: 'Madrid'})
MERGE(Zaragoza)-[:ROAD {distance: 620}]→(Madrid);
```

Y ahora podemos ir desde Zaragoza a Malaga:

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

startCity	endCity	totalCost
"Zaragoza"	"Malaga"	1140.0

También podemos calcular las rutas entre todos los pares de nodos:

```
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
```

startCity	endCity	totalCost
"Granada"	"Malaga"	125.0
"Sevilla"	"Malaga"	210.0
"Barcelona"	"Zaragoza"	300.0
"Zaragoza"	"Bilbao"	300.0
"Madrid"	"Valencia"	350.0
"Valencia"	"Barcelona"	350.0
"Madrid"	"Granada"	395.0
"Madrid"	"Malaga"	520.0
"Madrid"	"Sevilla"	530.0

Ejecución de grafos de relaciones

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.

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'}),
```

Table | Code

Server version	Neo4j/5.1.0
Server address	localhost:7687
Query	CREATE (Alice:User {name: 'Alice'}), (Bob:Use {name: 'Grace'}), (Henry:User {name: 'Henry'}) 'Nina'}), (Oscar:User {name: 'Oscar'}), (Paul:Us 'Uma'}), (Victor:User {name: 'Victor'}), (Wendy:

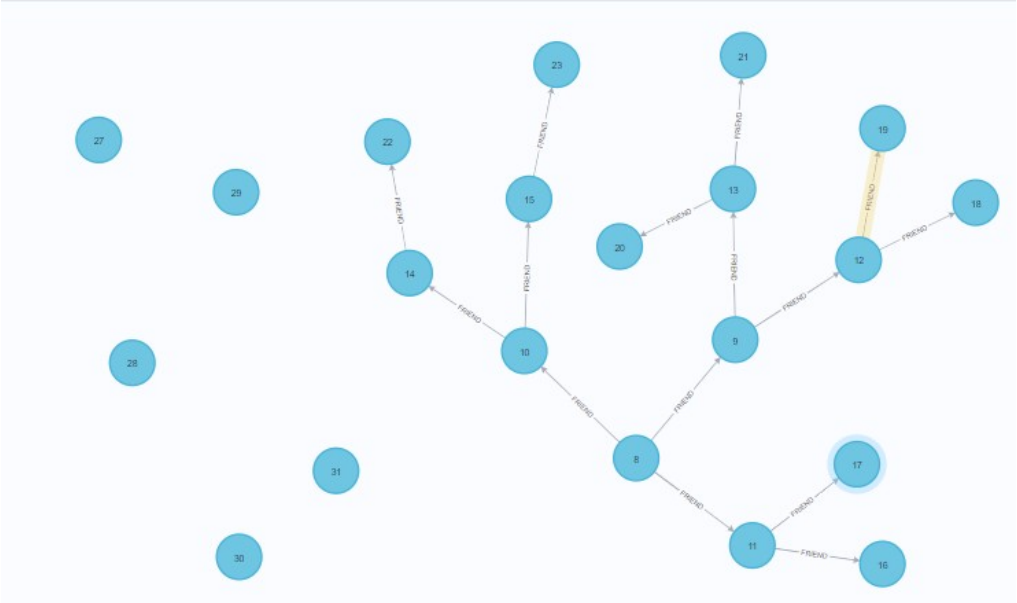
Crear Relaciones de Amistad con Peso:

```
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),
```

Table | Warn | Code

Server version	Neo4j/5.1.0
Server address	localhost:7687
Query	MATCH (a:User {name: 'Alice'}), (b:User {name: 'Bob'}), (c:User {name: 'Charlie'}), (d:User {name: 'David'}), ('Henry'}), (i:User {name: 'Ivy'}), (j:User {name: 'Jack'}), (k:User {name: 'Karen'}), (l:User {name: 'Leo'}), (m:Us (q:User {name: 'Quinn'}), (r:User {name: 'Rose'}), (s:User {name: 'Sam'}), (t:User {name: 'Tina'}), (u:User {nar (y:User {name: 'Yara'}) // Crear relaciones de amistad con peso CREATE (a)-[:FRIEND {weight: 1.0}]->(b), (a (e), (b)-[:FRIEND {weight: 0.8}]->(f), (c)-[:FRIEND {weight: 2.3}]->(g), (c)-[:FRIEND {weight: 1.7}]->(h), (d)-[:F [:FRIEND {weight: 3.1}]->(l), (f)-[:FRIEND {weight: 1.4}]->(m), (f)-[:FRIEND {weight: 2.0}]->(n), (g)-[:FRIEND

```
MATCH (n:User) RETURN n LIMIT 25
```



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;
```

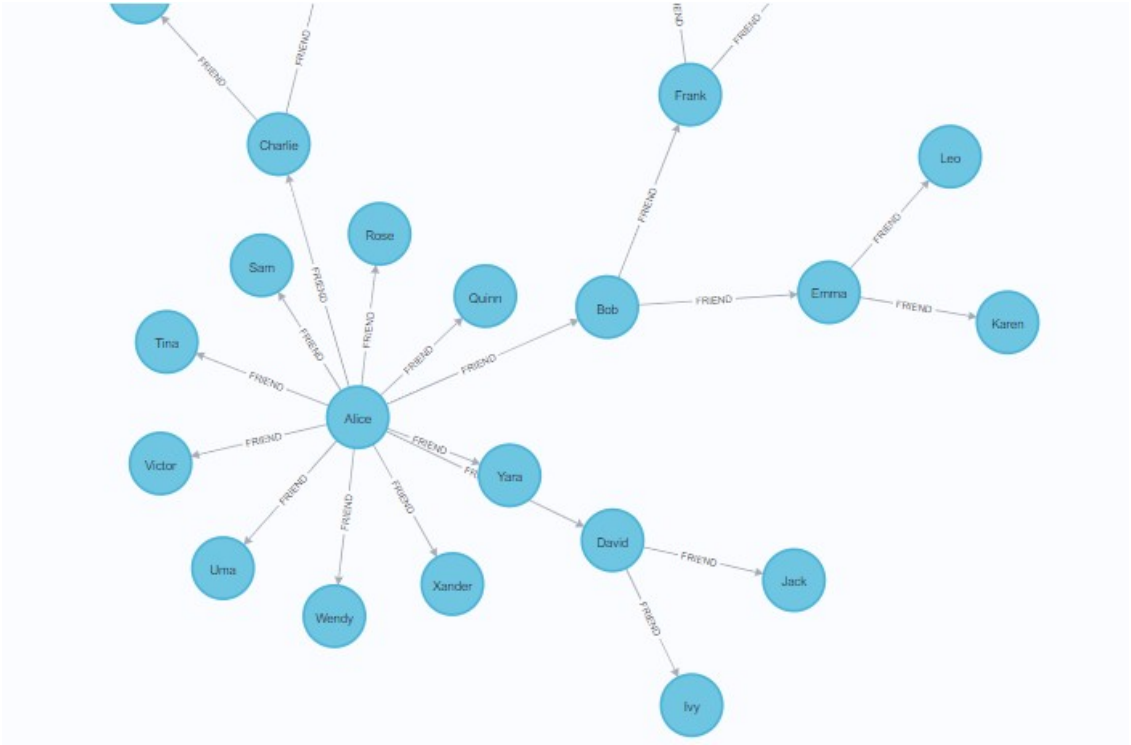
Table | Text | Code

	isolatedNode
1	"Quinn"
2	"Rose"
3	"Sam"
4	"Tina"
5	"Uma"
6	"Victor"

Conectar los nodos aislados con ‘Alice’:

```
// 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),
```

Set 9 properties, created 9 relationships, completed after 21 ms.



Proyectar el Grafo en Memoria

Proyectar el Grafo en Memoria:

```
CALL gds.graph.project(
    'socialGraph',
    'User',
    {
        FRIEND: {
            type: 'FRIEND',
            properties: 'weight'
        }
    }
);
```

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	projectMillis
{User: {label: "User", properties: {}}}	{FRIEND: {orientation: "NATURAL", indexInverse: false, aggregation: "DEFAULT", type: "FRIEND", properties: {weight: {defaultValue: null, property: "weight", aggregation: "DEFAULT"}}}}	"socialGraph"	25	24	8324

Aplicar Algoritmos de Análisis:

- Centralidad de Grado

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

```
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;
```

user	degree
"Alice"	12.0
"Frank"	2.0
"Bob"	2.0
"Emma"	2.0

- Centralidad de PageRank

Propósito: Medir la importancia de cada nodo en función de sus conexiones y la importancia de los nodos que lo conectan.

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

	user	pageRank
1	"Oscar"	0.33552578125000004
2	"Paul"	0.33552578125000004
3	"Karen"	0.24276289062500003

- Detección de Comunidades con Louvain

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

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

	user	communityId
1	"Jack"	9
2	"David"	9
3	"Ivy"	9
4	"Leo"	13

- Camino Mínimo entre Dos Usuarios (Dijkstra)

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

```
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;
```

startUser	endUser	totalCost
"Alice"	"Karen"	6.0

- Existencia de Amistades Comunes

Propósito: Mostrar pares de usuarios que tienen amigos en común.

```
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;
```

user1	user2	commonFriends
"Ivy"	"Alice"	1
"Henry"	"Alice"	1
"Grace"	"Alice"	1
"Frank"	"Alice"	1
"Karen"	"Alice"	1

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 (Dijkstra): 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.