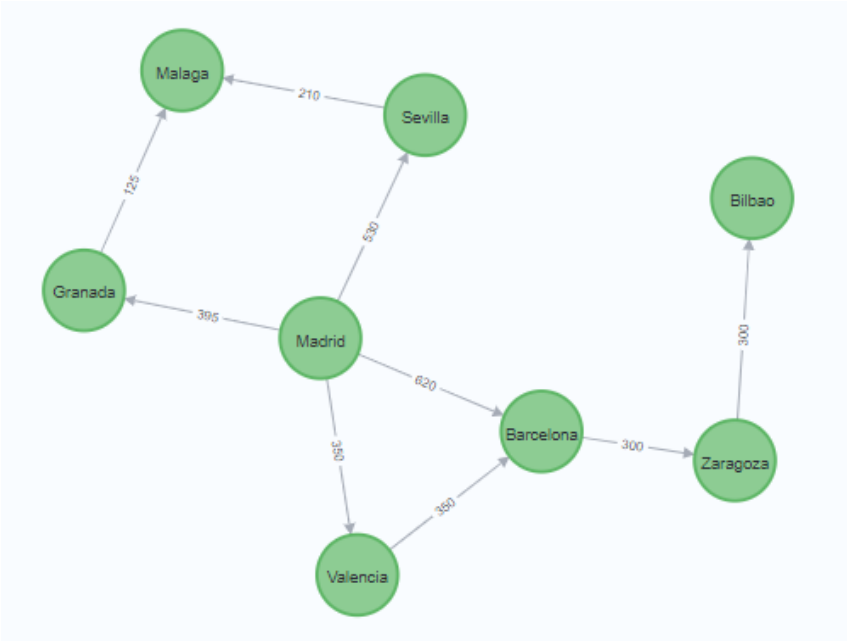
Análisis de Rutas en Neo4j

Características de la base de datos:

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

Objetivo:

- 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.



Crear el Grafo

```
// Creación de los nodos (ciudades)
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'})
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);
```

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)

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

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.

Crear el grafo en memoria:

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

Calcular el Camino Mínimo entre Dos Ciudades usando Dijkstra

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

```
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
```

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

```
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
```

Eliminamos el grafo de la memoria

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

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)

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'}),
    (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'})
```

Crear Relaciones de Amistad con Peso

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.

```
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);
```

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;
```

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),
    (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);
```

Proyectar el Grafo en Memoria

Proyecta el grafo en memoria

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

Aplicar Algoritmos de Análisis

1. 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;
  ```

2. 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
  LIMIT 10;
  ```

3. 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;
```

4. 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;
```

5. 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;
```

Paso 5: Eliminar el Grafo en Memoria (Opcional)

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

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

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**.**