Big Data Aplicado

UD2 - Procesado y Presentación de Datos Almacenados

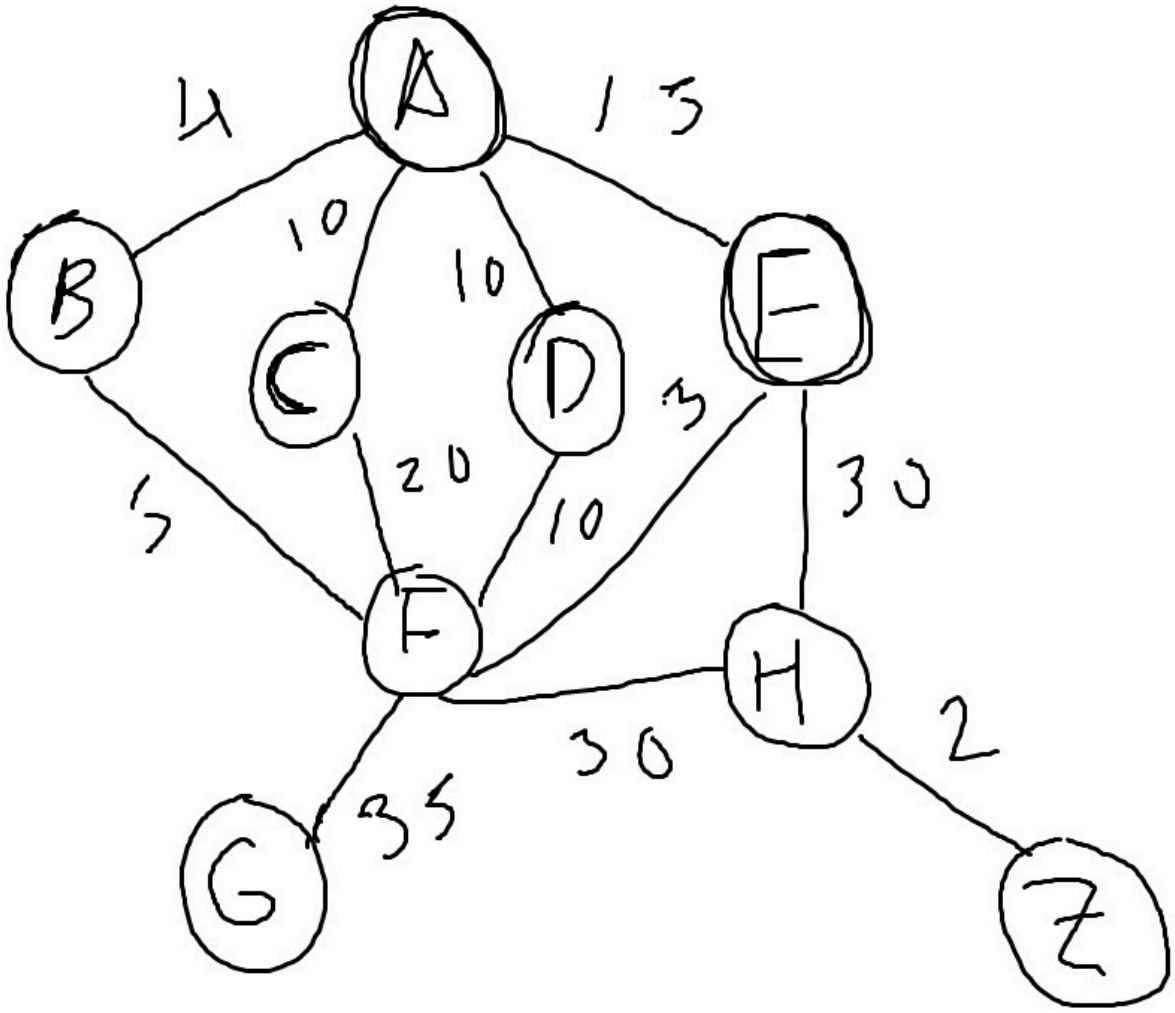
Práctica 2 Neo4j

> Recuerda que para hacer la prácticas puedes optar por cualquiera de la las 3 opciones.
>
> - Instalarla en tu máquina local
> - Usar SandBox Neo4j.com: https://sandbox.neo4j.com/ con Graph Data Science
> - Crear un contenedor docker.

Ejercicio 1:

Diseña un grafo, de temática libre, compuesto por mínimo seis nodos. Para las ponderaciones de las aristas tomaremos valores que vayan en consonancia con el contenido elegido

El grafo elegido será el siguiente:



De los cuales los nodos son ciudades y las aristas son la distancia entre ellas.

Ejercicio 2: Dado el grafo anterior:

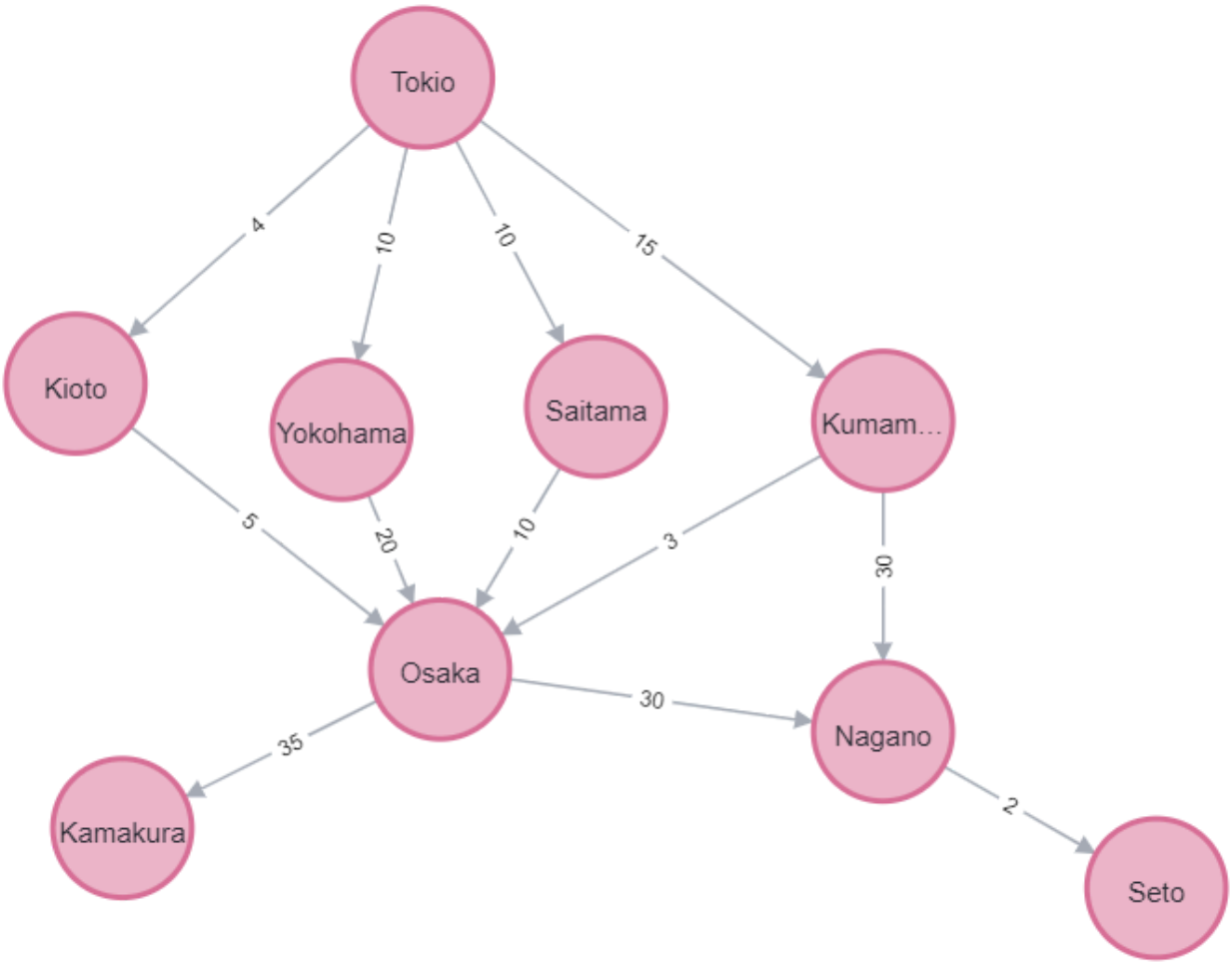
1. Crea el grafo en Neo4j.

El grafo anterior se haría de la siguiente forma:

```
CREATE (a:City {name: 'Tokio',         latitude: 65.4632, longitude: 0.7931
       (b:City {name: 'Kioto',         latitude: 63.5345, longitude: 1.3453
       (c:City {name: 'Yokohama',         latitude: 64.6567, longitude: 1.3
       (d:City {name: 'Saitama',        latitude: 66.2528, longitude: 1.34
       (e:City {name: 'Kumamoto',         latitude: 68.4790, longitude: 1.3
       (f:City {name: 'Osaka',         latitude: 65.4632, longitude: 2.6830
       (g:City {name: 'Kamakura',         latitude: 64.6567, longitude: 3.1
       (h:City {name: 'Nagano',        latitude: 66.2934, longitude: 2.787
       (z:City {name: 'Seto',        latitude: 70.7934, longitude: 3.3759}
       (a)-[:CONNECTION {distance: 4}]->(b),
       (a)-[:CONNECTION {distance: 10}]->(c),
       (a)-[:CONNECTION {distance: 10}]->(d),
       (a)-[:CONNECTION {distance: 15}]->(e),
       (b)-[:CONNECTION {distance: 5}]->(f),
       (c)-[:CONNECTION {distance: 20}]->(f),
```

```
(d)-[:CONNECTION {distance: 10}]->(f),
(e)-[:CONNECTION {distance: 3}]->(f),
(e)-[:CONNECTION {distance: 30}]->(h),
(f)-[:CONNECTION {distance: 35}]->(g),
(f)-[:CONNECTION {distance: 30}]->(h),
(h)-[:CONNECTION {distance: 2}]->(z)
```

Lo que nos crea nuestro grafo:



2. Haz un recorrido en anchura y profundidad.

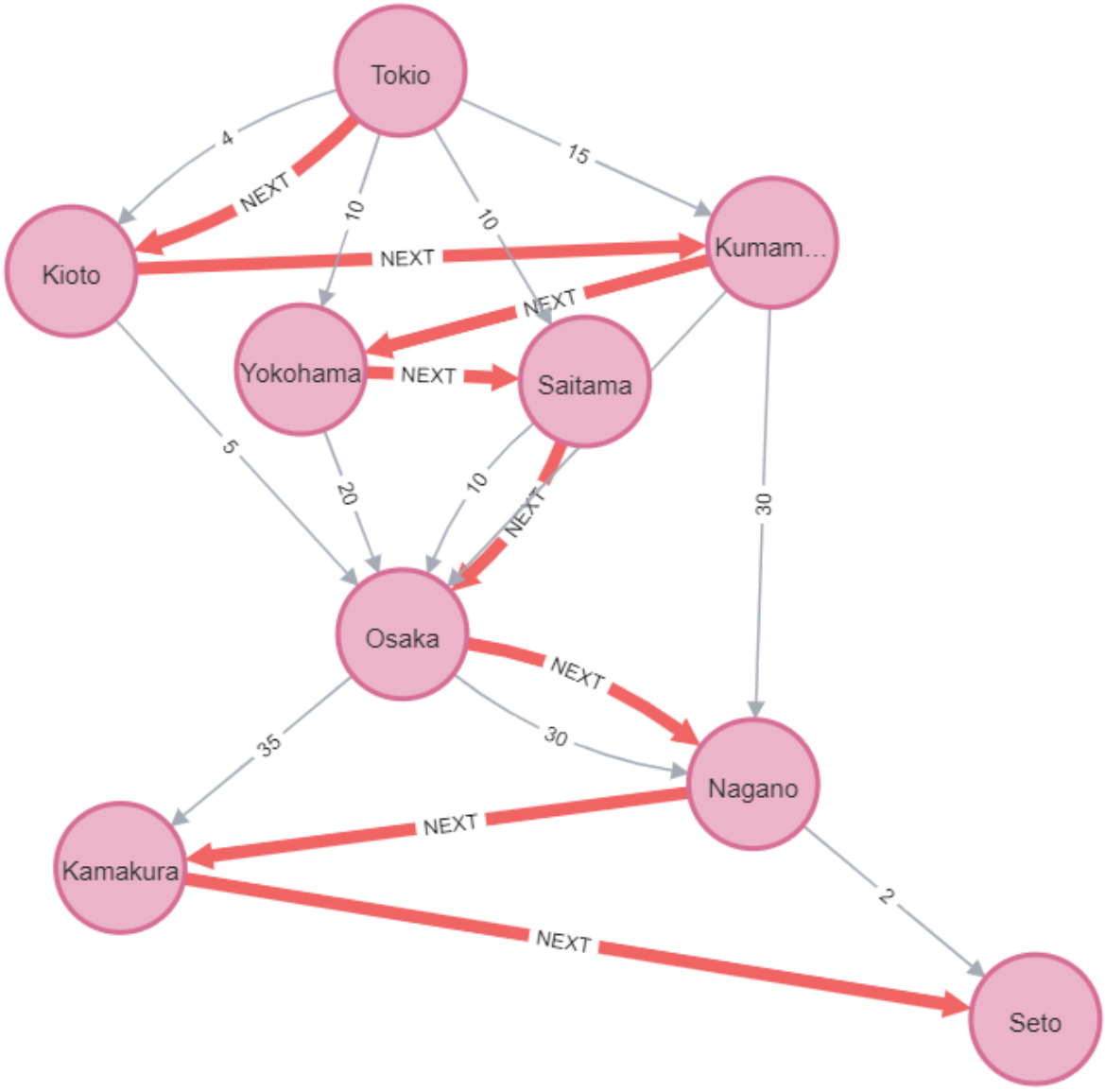
- Búsqueda en anchra

```
// Primero hacemos la proyección
MATCH (source:City)-[r:CONNECTION]->(target:City)
RETURN gds.graph.project(
   'myGraph_BFS',
   source,
   target
)
```

```
// Luego hacemos la búsqueda BFS
MATCH (source:City {name:'Tokio'})
```

```
CALL gds.bfs.stream('myGraph_BFS', {
  sourceNode: source
})
YIELD path
RETURN path
```

El resultado final se ve en el siguiente grafo:

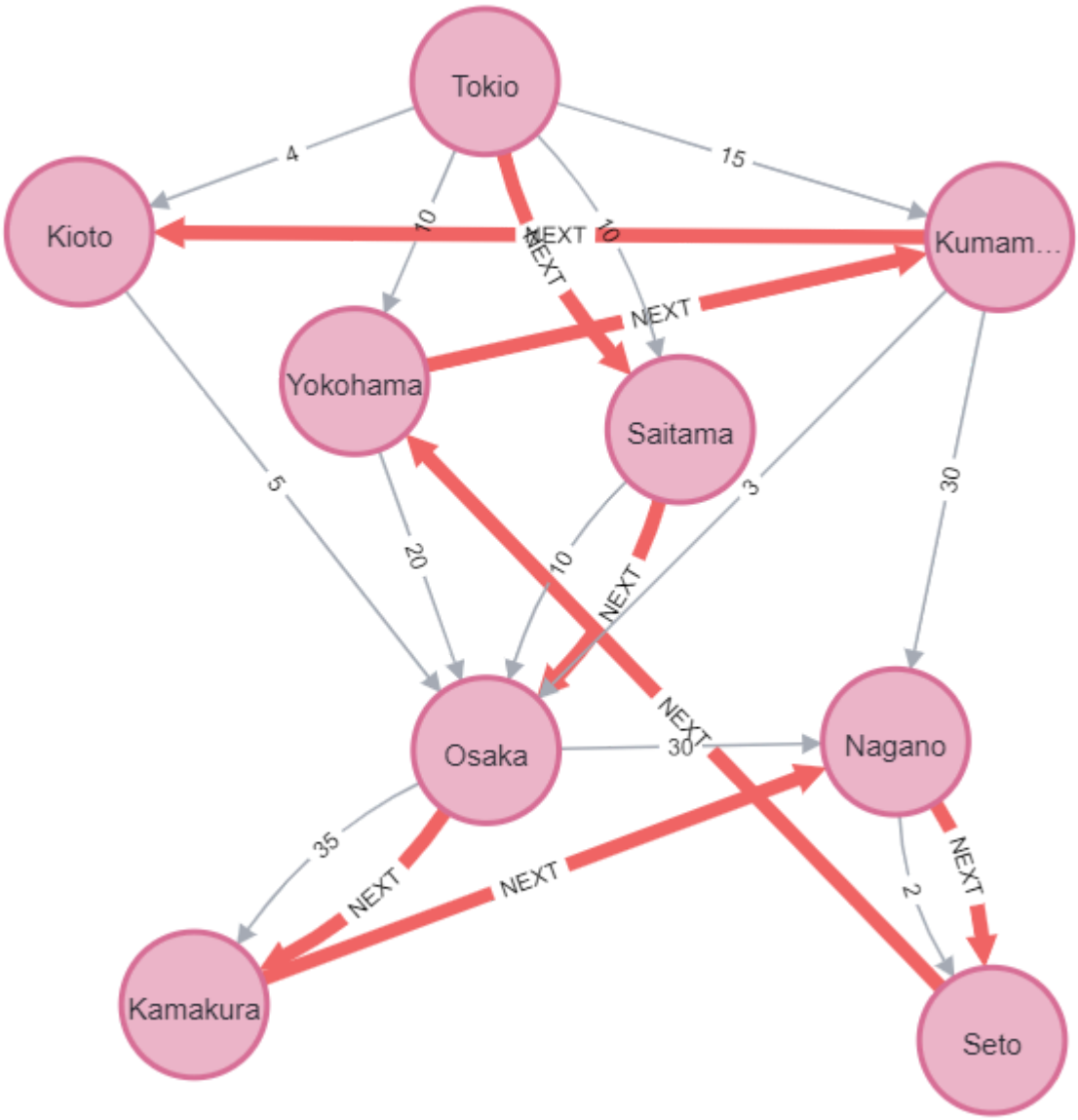


- Búsqueda en profundidad

```
// Primero hacemos la proyección
MATCH (source:City)-[r:CONNECTION]->(target:City)
RETURN gds.graph.project(
   'myGraph_DFS',
   source,
   target
)
```

```
MATCH (source:City{name:'Tokio'})
CALL gds.dfs.stream('myGraph_DFS', {
  sourceNode: source
})
YIELD path
RETURN path
```

El grafo resultante es el siguiente:



3. Obtén el camino mínimo, utilizando el algoritmo de Dijkstra, entre un nodo y los restantes.

Lo primero que hay que hacer es de nuevo la proyección:

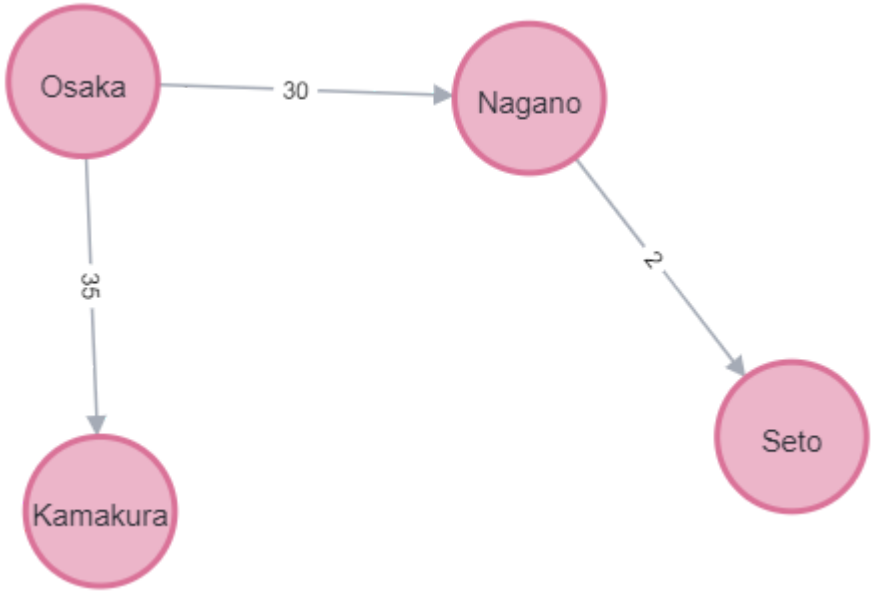
```
// Creamos la Proyección del grafo
MATCH (source:City)-[r:CONNECTION]->(target:City)
RETURN gds.graph.project(
  'myGraph_Dijkstra',
  source,
  target,
```

```
  { relationshipProperties: r { .distance } }
)
```

Luego obtenemos el camino mínimo, nuestro origen será Osaka.

```
MATCH (source:City {name: 'Osaka'})
CALL gds.allShortestPaths.dijkstra.stream('myGraph_Dijkstra', {
    sourceNode: source,
    relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
    index,
    gds.util.asNode(sourceNode).name AS sourceNodeName,
    gds.util.asNode(targetNode).name AS targetNodeName,
    totalCost,
    [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
    costs,
    nodes(path) as path
ORDER BY index
```

Lo que nos devuelve este grafo:



Y la siguiente tabla:

```
╒═════╤══════════════╤══════════════╤═════════╤═══════════════════════════════
│index│sourceNodeName│targetNodeName│totalCost│nodeNames
│costs            │path
│
╞═════╪══════════════╪══════════════╪═════════╪═══════════════════════════════
│0    │"Osaka"       │"Osaka"       │0.0      │["Osaka"]
│[0.0]            │[(:City {name: "Osaka",latitude: 65.4632,longitude:
2.683})]         │
```

```
├─────┼──────────┼──────────┼────────┼──────────────────────────────┤
│1    │"Osaka"   │"Nagano"  │30.0    │["Osaka", "Nagano"]
│[0.0, 30.0]   │[(:City {name: "Osaka",latitude: 65.4632,longitude:
2.683}), (:City {n│
│     │          │          │        │
│             │ame: "Nagano",latitude: 66.2934,longitude: 2.7879})]
│
├─────┼──────────┼──────────┼────────┼──────────────────────────────┤
│2    │"Osaka"   │"Seto"    │32.0    │["Osaka", "Nagano",
"Seto"]│[0.0, 30.0, 32.0]│[(:City {name: "Osaka",latitude:
65.4632,longitude: 2.683}), (:City {n│
│     │          │          │        │
│             │ame: "Nagano",latitude: 66.2934,longitude: 2.7879}),
(:City {name: "Se│
│     │          │          │        │
│             │to",latitude: 70.7934,longitude: 3.3759})]
│
├─────┼──────────┼──────────┼────────┼──────────────────────────────┤
│3    │"Osaka"   │"Kamakura"│35.0    │["Osaka", "Kamakura"]
│[0.0, 35.0]   │[(:City {name: "Osaka",latitude: 65.4632,longitude:
2.683}), (:City {n│
│     │          │          │        │
│             │ame: "Kamakura",latitude: 64.6567,longitude:
3.1194})]          │
└─────┴──────────┴──────────┴────────┴──────────────────────────────┘
```

4. Calcula el camino mínimo, utilizando el algoritmo de Dijkstra y el algoritmo A*, entre dos nodos elegidos. *Si tu grafo no tiene latitud y longitud, utiliza otro algoritmo de camino mínimo entre los disponibles en [path finding GDS]*

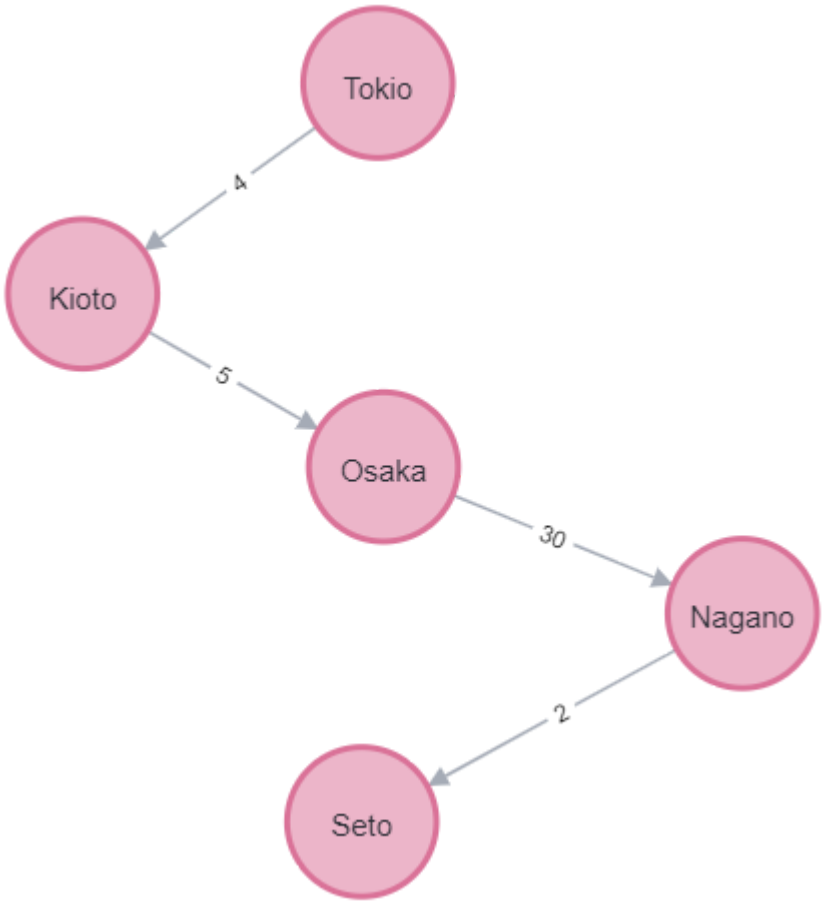
- Algoritmo de Dijkstra

  La proyección ya la tenemos, por lo tanto solo hay que indicar la búsqueda. Nuestro origen será Tokio y nuestro destino será Seto.

```
MATCH (source:City {name: 'Tokio'}), (target:City {name: 'Seto'})
CALL gds.shortestPath.dijkstra.stream('myGraph_Dijkstra', {
    sourceNode: source,
    targetNode: target,
    relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
    index,
    gds.util.asNode(sourceNode).name AS sourceNodeName,
    gds.util.asNode(targetNode).name AS targetNodeName,
    totalCost,
    [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
    costs,
```

```
    nodes(path) as path
ORDER BY index
```

Lo que devuelve es siguiente grafo:



Y esta tabla:

```
╒═════╤══════════════╤══════════════╤═══════════╤═══════════════════════════
│index│sourceNodeName│targetNodeName│totalCost│nodeNames
│costs                    │path
│
╞═════╪══════════════╪══════════════╪═══════════╪═══════════════════════════
│0    │"Tokio"       │"Seto"        │41.0     │["Tokio", "Kioto",
"Osaka", "Nagano", "Seto"]│[0.0, 4.0, 9.0, 39.0, 41.0]│[(:City
{name: "Tokio",latitude: 65.4632,longitude: 0.7931}), (:City {│
│     │              │              │         │
│                         │name: "Kioto",latitude:
63.5345,longitude: 1.3453}), (:City {name: "Os│
│     │              │              │         │
│                         │aka",latitude: 65.4632,longitude:
2.683}), (:City {name: "Nagano",lati│
│     │              │              │         │
│                         │tude: 66.2934,longitude: 2.7879}),
(:City {name: "Seto",latitude: 70.7│
│     │              │              │         │
│                         │934,longitude: 3.3759})]
```

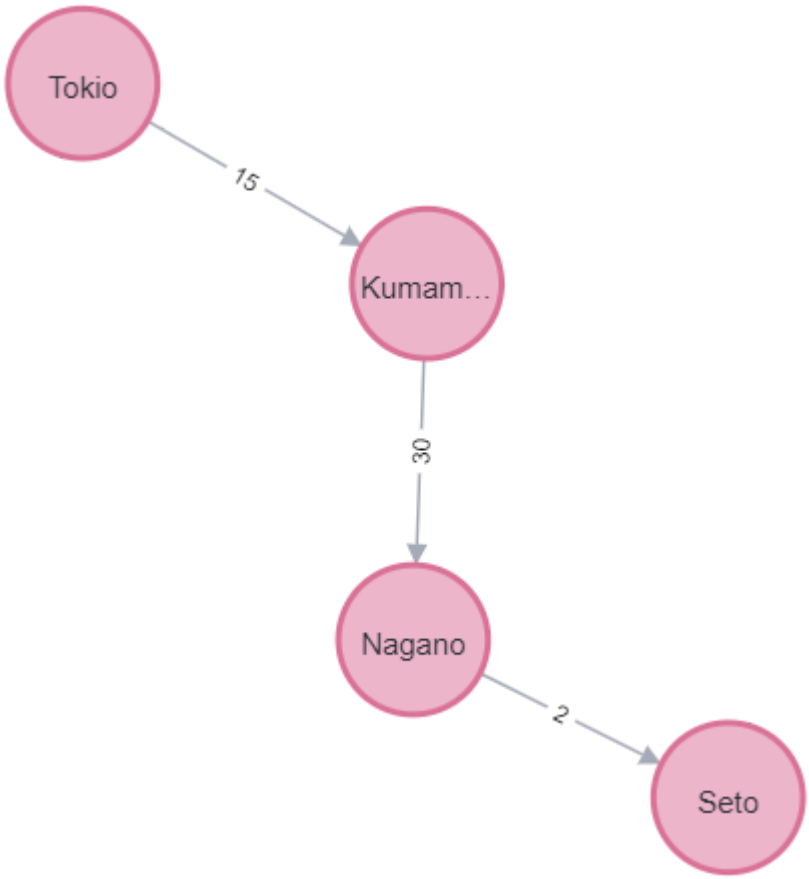
```
|
└────┴──────────────┴──────────────┴──────────┴──────────────────────┘
```

- Algoritmo A*

```
MATCH (source:City)-[r:CONNECTION]->(target:City)
RETURN gds.graph.project(
  'myGraph_A',
  source,
  target,
  {
    sourceNodeProperties: source { .latitude, .longitude },
    targetNodeProperties: target { .latitude, .longitude },
    relationshipProperties: r { .distance }
  }
)
```

```
MATCH (source:City {name: 'Tokio'}), (target:City {name: 'Seto'})
CALL gds.shortestPath.astar.stream('myGraph_A', {
    sourceNode: source,
    targetNode: target,
    latitudeProperty: 'latitude',
    longitudeProperty: 'longitude',
    relationshipWeightProperty: 'distance'
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
    index,
    gds.util.asNode(sourceNode).name AS sourceNodeName,
    gds.util.asNode(targetNode).name AS targetNodeName,
    totalCost,
    [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS nodeNames,
    costs,
    nodes(path) as path
ORDER BY index
```

El grafo resultante es el siguiente:



Junto con esta tabla:

```
┌─────┬──────────────┬──────────────┬─────────┬──────────────────────────────
│index│sourceNodeName│targetNodeName│totalCost│nodeNames
│costs                  │path
│
└─────┴──────────────┴──────────────┴─────────┴──────────────────────────────
│0    │"Tokio"       │"Seto"        │47.0     │["Tokio", "Kumamoto",
"Nagano", "Seto"]│[0.0, 15.0, 45.0, 47.0]│[(:City {name:
"Tokio",latitude: 65.4632,longitude: 0.7931}), (:City {│
│     │              │              │         │
│                       │name: "Kumamoto",latitude:
68.479,longitude: 1.3453}), (:City {name: "│
│     │              │              │         │
│                       │Nagano",latitude: 66.2934,longitude:
2.7879}), (:City {name: "Seto",la│
│     │              │              │         │
│                       │titude: 70.7934,longitude: 3.3759})]
│
└─────┴──────────────┴──────────────┴─────────┴──────────────────────────────
```

5. Borra el grafo completo.

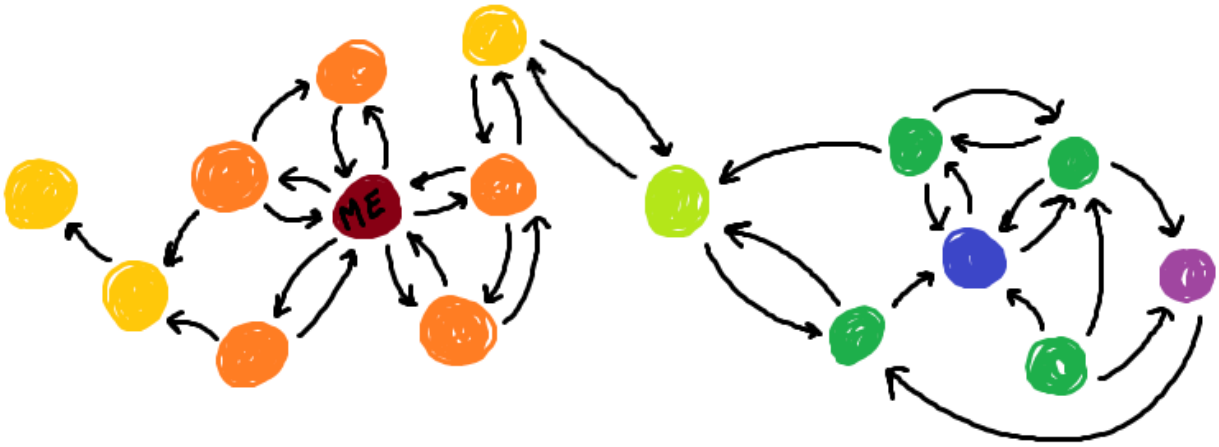
Para borrar un grafo por completo se usa el siguiente comando:

```
match (n) detach delete(n)
```

Ejercicio 3:

Diseña e implementa en Neo4j un grafo que represente una red social conformada por ti y, al menos, cuatro amigos más. Hecho esto, obtén las medidas de centralidad de grado, cercanía e intermediación para cada uno de los nodos.

El grafo planteado será el siguiente, donde **ME** seré yo y el resto serán amigos y otros usuarios que les siguen:



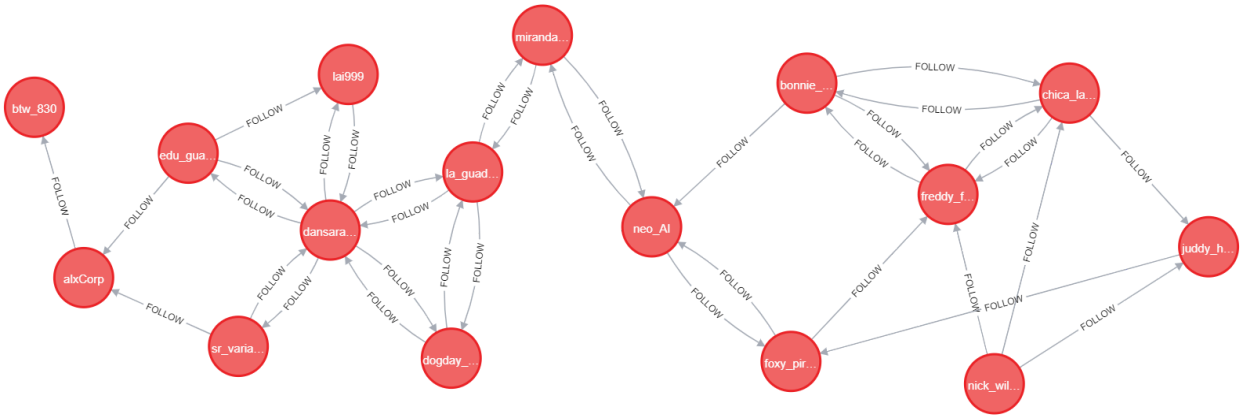
El código en cypher para representar este grafo sería el siguiente:

```
// Crear los nodos
CREATE
(a:User {name: 'dansarasix_doremi'}),
(b:User {name: 'lai999'}),
(c:User {name: 'edu_guapo'}),
(d:User {name: 'sr_variable'}),
(e:User {name: 'la_guadaLupe_1043'}),
(f:User {name: 'dogday_22'}),
(g:User {name: 'alxCorp'}),
(h:User {name: 'btw_830'}),
(i:User {name: 'miranda_japan'}),
(j:User {name: 'neo_AI'}),
(k:User {name: 'freddy_faz555'}),
(l:User {name: 'chica_laca211'}),
(m:User {name: 'bonnie_crazy_:3'}),
(n:User {name: 'foxy_pirate_1001'}),
(o:User {name: 'nick_wilde_XP'}),
(p:User {name: 'juddy_hopps_cop'}),

  (a)-[:FOLLOW]->(b),
  (b)-[:FOLLOW]->(a),
  (a)-[:FOLLOW]->(c),
  (c)-[:FOLLOW]->(a),
  (a)-[:FOLLOW]->(d),
  (d)-[:FOLLOW]->(a),
```

```
(a)-[:FOLLOW]->(e),
(e)-[:FOLLOW]->(a),
(e)-[:FOLLOW]->(f),
(a)-[:FOLLOW]->(f),
(f)-[:FOLLOW]->(a),
(f)-[:FOLLOW]->(e),
(c)-[:FOLLOW]->(b),
(c)-[:FOLLOW]->(g),
(d)-[:FOLLOW]->(g),
(g)-[:FOLLOW]->(h),
(e)-[:FOLLOW]->(i),
(i)-[:FOLLOW]->(e),
(i)-[:FOLLOW]->(j),
(j)-[:FOLLOW]->(i),
(k)-[:FOLLOW]->(l),
(l)-[:FOLLOW]->(k),
(l)-[:FOLLOW]->(m),
(l)-[:FOLLOW]->(p),
(k)-[:FOLLOW]->(m),
(m)-[:FOLLOW]->(k),
(m)-[:FOLLOW]->(j),
(m)-[:FOLLOW]->(l),
(n)-[:FOLLOW]->(k),
(n)-[:FOLLOW]->(j),
(j)-[:FOLLOW]->(n),
(o)-[:FOLLOW]->(k),
(o)-[:FOLLOW]->(l),
(o)-[:FOLLOW]->(p),
(p)-[:FOLLOW]->(n)
```

Y generando nuestra versión en neo4j:



Medidas de centralidad

- Centralidad de grado

Como siempre, hacemos primero la proyección y luego la centralidad.

```
MATCH (source:User)-[r:FOLLOW]->(target:User)
RETURN gds.graph.project(
  'myGraph_centralidad_grado',
  target,
  source
)
```

```
CALL gds.degree.stream('myGraph_centralidad_grado')
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name AS name, score AS followers
ORDER BY followers DESC, name DESC
```

Esto nos devolverá la siguiente tabla:

Usuario	Seguidores
dansarasix_doremi	5
freddy_faz555	4
neoAI	3
la_guadaLupe_1043	3
chica_laca211	3
miranda_japan	2
lai999	2
juddy_hopps_cop	2
foxy_pirate_1001	2
dogday_22	2
bonnie_crazy_:3	2
alxCorp	2
sr_variable	1
edu_guapo	1
btw_830	1
nick_wilde_XP	0

- Centralidad de cercanía

```
MATCH (source:User)-[r:FOLLOW]->(target:User)
RETURN gds.graph.project(
  'myGraph_cercania',
  source,
  target
)
```

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

La tabla resultante es la siguiente:

Usuario	Puntaje
miranda_japan	0.40625
neo_AI	0.40625
la_guadaLupe_1043	0.38235294117647056
dansarasix_doremi	0.3
foxy_pirate_1001	0.3
dogday_22	0.30952380952380953
freddy_faz555	0.3023255813953488
lai999	0.26
sr_variable	0.2549019607843137
edu_guapo	0.2549019607843137
chica_laca211	0.24528301886792453
bonnie_crazy_:3	0.2407
alxCorp	0.22
juddy_hopps_cop	0.20967741935483872
btw_830	0.19230769230769232
nick_wilde_XP	0

- Centralidad de intermediación

```
MATCH (source:User)-[r:FOLLOW]->(target:User)
RETURN gds.graph.project(
  'myGraph_intermediacion',
  source,
  target
)
```

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

Y obtenemos esta tabla:

Usuario	Puntaje (%)
miranda_japan	92
la_guadaLupe_1043	91
neo_AI	90.5
dansarasix_doremi	84
foxy_pirate_1001	48.33
freddy_faz555	32.83
bonnie_crazy_:3	27.16
chica_laca211	15.33
alxCorp	14
sr_variable	12
edu_guapo	12
juddy_hopps_cop	5.83
dogday_22	0
lai999	0
btw_830	0
nick_wilde_XP	0

Las conclusiones que podemos sacar son las siguientes:

- El usuario con más seguidores en el grafo es **dansarasix_doremi** (yo) con 5 seguidores.
- **miranda_japan** es el usuario en el grafo que más expande rápida y eficientemente la información a través del grafo (40.62% de eficiencia), y también el nodo más influyente en el grafo (92% de influencia).