
Ingeniería en Sistemas Computacionales

Bases de Datos No Relacionales

# Neo4j Lab 3 – Graph Data Science

Marco Ricardo Cordero Hernández

Tlaquepaque, Jal., 10 de mayo de 2023

Como parte del aprendizaje extendido que pretende lograr este curso y específicamente para esta sección del mismo, haciendo uso de Neo4j se probará un poco de las herramientas disponibles para la misma base de grafos, en este caso la extensión ***GDS*** (Graph Data Science), la cual provee múltiples algoritmos aplicables a análisis de grafos, útiles para cosas como la búsqueda del camino más corto entre dos nodos, encontrando (por ejemplo) en el infame algoritmo de Dijkstra.

Para demostrar el uso de esta extensión, se ha dividido este laboratorio en dos partes: manipulación directa desde el cliente gráfico; y manejo del módulo disponible en Python para análisis programáticos.

**GDS desde cliente gráfico**

A manera de demostración práctica, se ha hecho uso de la documentación oficial de Neo4j, de donde se ha extraído el modo de utilización del algoritmo $A^*$, el cual se fundamenta en el análisis de grafos con pesos asignados a sus componentes. El problema que este algoritmo resuelve, así como muchos, es el de encontrar el camino más corto entre dos o más nodos con el menor "costo" posible. Lo que diferencia a este algoritmo de otros similares es el uso de heurística, lo cual, computacionalmente hablando, se traduce a memoria. Precisamente este elemento resulta en la mayor desventaja de su aplicación, puesto que, para grafos más grandes, se requiere más memoria.

El código utilizado es el siguiente (extraído directamente de [https://neo4j.com/docs/graph-data-science/2.3/algorithms/astar/](https://neo4j.com/docs/graph-data-science/2.3/algorithms/astar/)):

```
// Creación de nodos para el ejemplo
CREATE (a:Station {name: 'Kings Cross',         latitude: 51.5308, longitude: -0.1238}),
       (b:Station {name: 'Euston',              latitude: 51.5282, longitude: -0.1337}),
       (c:Station {name: 'Camden Town',         latitude: 51.5392, longitude: -0.1426}),
       (d:Station {name: 'Mornington Crescent', latitude: 51.5342, longitude: -0.1387}),
       (e:Station {name: 'Kentish Town',        latitude: 51.5507, longitude: -0.1402}),
       (a)-[:CONNECTION {distance: 0.7}]->(b),
       (b)-[:CONNECTION {distance: 1.3}]->(c),
       (b)-[:CONNECTION {distance: 0.7}]->(d),
       (d)-[:CONNECTION {distance: 0.6}]->(c),
       (c)-[:CONNECTION {distance: 1.3}]->(e);
```

Added 5 labels, created 5 nodes, set 20 properties, created 5 relationships, completed after 912 ms.

```
// Proyecto local de GDS
CALL gds.graph.project(
    'lab3_demo',
    'Station',
    'CONNECTION',
    {
        nodeProperties: ['latitude', 'longitude'],
        relationshipProperties: 'distance'
    }
);
```

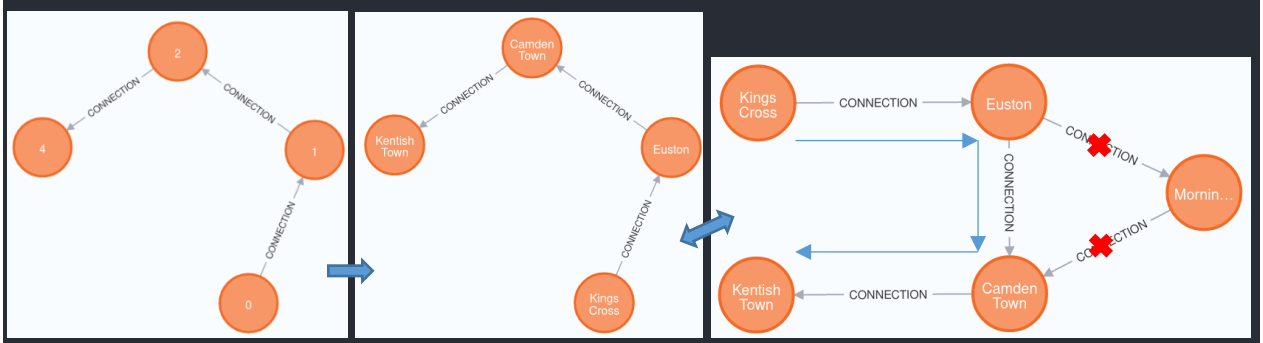
```
CALL gds.graph.project( 'lab3_demo', 'Station', 'CONNECTION', { nodeProperties: ['latitude', 'longitude'], relationshipProperties: 'distance' } );
```

| nodeProjection | relationshipProjection | graphName | nodeCount | relationshipCount | projectMillis |
|---|---|---|---|---|---|
| { "Station": { "label": "Station", "properties": { "latitude": { "defaultValue": null, "property": "latitude" }, "longitude": { "defaultValue": null, "property": "longitude" } } } } | { "CONNECTION": { "orientation": "NATURAL", "indexInverse": false, "aggregation": "DEFAULT", "type": "CONNECTION", "properties": { "distance": { "defaultValue": null, "property": "distance", "aggregation": "DEFAULT" } } } } | "lab3_demo" | 5 | 5 | 135 |

```
// Cálculo de costo en memoria para el algoritmo A*
MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.astar.write.estimate('lab3_demo', {
    sourceNode: source,
    targetNode: target,
    latitudeProperty: 'latitude',
    longitudeProperty: 'longitude',
    writeRelationshipType: 'PATH'
})
YIELD nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory
RETURN nodeCount, relationshipCount, bytesMin, bytesMax, requiredMemory;
```

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory |
|---|---|---|---|---|
| 5 | 5 | 1336 | 1336 | "1336 Bytes" |

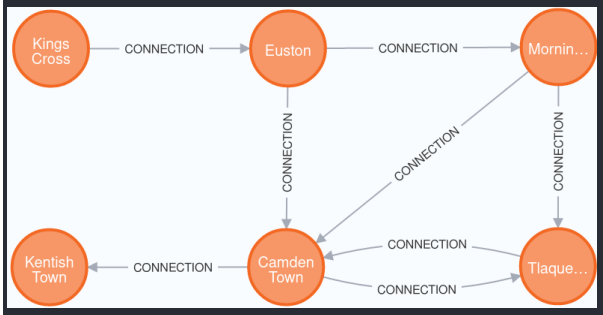
```
// Aplicación del algoritmo A*
MATCH (source:Station {name: 'Kings Cross'}), (target:Station {name: 'Kentish Town'})
CALL gds.shortestPath.astar.stream('lab3_demo', {
    sourceNode: source,
    targetNode: target,
    latitudeProperty: 'latitude',
    longitudeProperty: 'longitude',
    relationshipWeightProperty: '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;
```



Como se puede ver, tan solo para la búsqueda de caminos en un grafo de 5 nodos y 5 relaciones se realizó una proyección de uso de 1336 bytes (≈ 1KB) de memoria. Esto pudiera parecer poco cuando se piensa que los equipos de cómputo actuales cuentan con 6GB de memoria RAM en promedio, sin embargo, cuando se empieza a pensar en que los grafos crecen conforme los datos se van generando, este detalle puede resultar crítico incluso para la operación de la base entera. Se realizó una rápida demostración de lo mencionado.

```
CREATE (f:Station {name: 'Tlaquepaque', latitude: 20.64091, longitude: -103.29327});

MATCH (c:Station {name: 'Camden Town'}),
      (d:Station {name: 'Mornington Crescent'}),
      (f:Station {name: 'Tlaquepaque'})
MERGE (f)-[:CONNECTION {distance: 0.9}]->(c)-[:CONNECTION {distance: 0.9}]->(f)<-[:CONNECTION {distance: 0.8}]-(d);
```




```
// Se regenera el proyecto de GDS con su grafo respectivo para dar los siguientes resultados
```

| nodeCount | relationshipCount | bytesMin | bytesMax | requiredMemory |
|---|---|---|---|---|
| 6 | 8 | 1392 | 1392 | "1392 Bytes" |

Nuevamente, el nuevo espacio reclamado no es demasiado, únicamente se requirieron 56 bytes adicionales para 1 nodo y dos relaciones adicionales para casi un total de 2KB. Los ejemplos mostrados en este documento son únicamente ilustrativos, puesto que en un caso real los nodos se verían exponencialmente en aumento. El aprendizaje que esta sección deja es el de estar monitoreando constantemente la memoria disponible para tomar medidas preventivas o en el peor de los casos, correctivas.

**GDS a través de Python**

Como bien se debería de saber, el acceso gráfico a Neo4j es más bien útil para realizar reportes rápidos o como punto de acceso didáctico hacía sus capacidades, sin embargo, cuando realmente se quiere utilizar como base principal en aplicaciones reales, los datos necesitan ser procesados programáticamente, ya sea antes de su manipulación o después. Para ello, como se ha venido realizando desde entregables anteriores, se hará uso de bibliotecas y drivers para la comunicación entre Python y GDS.

Antes de escribir cualquier fragmento de código, es necesario instalar el módulo correspondiente.

```
(venv) marcordero@mrch-ubuntu:~/ITESO/NoSQL/iteso-bdnr-p2023-neo4j$ pip install graphdatascience
Requirement already satisfied: graphdatascience in ./venv/lib/python3.10/site-packages (1.6)
Requirement already satisfied: neo4j<6.0,>=4.4.2 in ./venv/lib/python3.10/site-packages (from graphdatascience) (5.7.0)
Requirement already satisfied: pyarrow<11.0,>=4.0 in ./venv/lib/python3.10/site-packages (from graphdatascience) (10.0.1)
Requirement already satisfied: tqdm<5.0,>=4.0 in ./venv/lib/python3.10/site-packages (from graphdatascience) (4.65.0)
Requirement already satisfied: pandas<2.0,>=1.0 in ./venv/lib/python3.10/site-packages (from graphdatascience) (1.5.3)
Requirement already satisfied: multimethod<2.0,>=1.0 in ./venv/lib/python3.10/site-packages (from graphdatascience) (1.9.1)
Requirement already satisfied: pytz in ./venv/lib/python3.10/site-packages (from neo4j<6.0,>=4.4.2->graphdatascience) (2023.3)
Requirement already satisfied: python-dateutil>=2.8.1 in ./venv/lib/python3.10/site-packages (from pandas<2.0,>=1.0->graphdatascience) (2.8.2)
Requirement already satisfied: numpy>=1.21.0 in ./venv/lib/python3.10/site-packages (from pandas<2.0,>=1.0->graphdatascience) (1.24.3)
Requirement already satisfied: six>=1.5 in ./venv/lib/python3.10/site-packages (from python-dateutil>=2.8.1->pandas<2.0,>=1.0->graphdatascience) (1.16.0)
```

Como se puede observar, la biblioteca ya se había instalado. De cualquier forma, el requerimiento previo ya ha sido satisfecho en este punto. Ya con esto preparado, se ha realizado el siguiente programa, el cual, por el carácter de esta actividad, no cuenta con el mayor grado de calidad ni de complejidad.

```python
from neo4j import GraphDatabase
from neo4j.exceptions import ClientError
from graphdatascience import GraphDataScience

URI = 'bolt://localhost:7687'
AUTH = ('neo4j', "iteso@123")
driver = GraphDatabase.driver(URI, auth=AUTH)

with driver as drv:
    drv.verify_connectivity()

with driver.session() as session:
    gds = GraphDataScience(URI, auth=AUTH)

    # Borrado inicial de proyecto y datos previos
    gds.run_cypher('''
        MATCH (n)
        DETACH DELETE n
    ''')

    try:
        gds.run_cypher('''
            CALL gds.graph.drop('lab3_demo_py')
            YIELD graphName
        ''')
    except ClientError:
        pass

    # Creación de nodos
    gds.run_cypher(
    """
        CREATE (a:Station {name: 'Kings Cross',         latitude: 51.5308,  longitude: -0.1238}),
               (b:Station {name: 'Euston',              latitude: 51.5282,  longitude: -0.1337}),
               (c:Station {name: 'Camden Town',         latitude: 51.5392,  longitude: -0.1426}),
               (d:Station {name: 'Mornington Crescent', latitude: 51.5342,  longitude: -0.1387}),
               (e:Station {name: 'Kentish Town',        latitude: 51.5507,  longitude: -0.1402}),
               (f:Station {name: 'Tlaquepaque',         latitude: 20.64091, longitude: -103.29327}),
               (a)-[:CONNECTION {distance: 0.7}]->(b),
               (b)-[:CONNECTION {distance: 1.3}]->(c),
               (b)-[:CONNECTION {distance: 0.7}]->(d),
               (d)-[:CONNECTION {distance: 0.6}]->(c),
               (c)-[:CONNECTION {distance: 1.3}]->(e),
               (f)-[:CONNECTION {distance: 0.9}]->(c),
               (c)-[:CONNECTION {distance: 0.9}]->(f),
               (d)-[:CONNECTION {distance: 0.8}]->(f)
    """
    )

    # Creación del proyecto
    G_stations, project_result = gds.graph.project(
        'lab3_demo_py',
        {'Station': {'properties': ['latitude', 'longitude']}},
        {'CONNECTION': {'properties': ['distance']}}
```

```
    )

    source_id = gds.find_node_id(['Station'], {'name': 'Kings Cross'})
    target_id = gds.find_node_id(['Station'], {'name': 'Tlaquepaque'})

    print(gds.shortestPath.astar.stream.estimate(
            G = G_stations,
            sourceNode = source_id,
            targetNode = target_id,
            latitudeProperty = 'latitude',
            longitudeProperty = 'longitude',
            relationshipWeightProperty = 'distance'
        )
    )

# Resultado
requiredMemory                                          1392 Bytes
treeView            Memory Estimation: 1392 Bytes\n|-- algorithm: ...
mapView             {'components': [{'components': [{'memoryUsage'...
bytesMin                                                       1392
bytesMax                                                       1392
nodeCount                                                         6
relationshipCount                                                 8
heapPercentageMin                                               0.1
heapPercentageMax                                               0.1
Name: 0, dtype: object

    print(gds.shortestPath.astar.stream(
            G = G_stations,
            sourceNode = source_id,
            targetNode = target_id,
            latitudeProperty = 'latitude',
            longitudeProperty = 'longitude',
            relationshipWeightProperty = 'distance'
        )
    )

# Resultado
   index  sourceNode  targetNode  totalCost           nodeIds                costs                      path
0      0          58          63        2.2  [58, 59, 61, 63]  [0.0, 0.7, 1.4, 2.2]  ((cost), (cost), (cost))
```

El resultado obtenido de la ejecución anterior resulta mucho menos intuitivo que aquel que pudiera proporcionar el navegador de Neo4j, sin embargo, sí existe la presencia de un dato interesante para este algoritmo: *totalCost* o costo total de la ruta elegida por el algoritmo. Traducido en información relevante, este dato podría interpretarse como distancias reales entre puntos geográficos en una ciudad para optimizar rutas de entrega para aplicaciones que requieran de envíos.

En esta ocasión, la única conclusión que habrá que externar es que la documentación de Neo4j es tan oscura y confusa como la de Oracle, pero al final, la información está ahí. La habilidad de comprensión lectora y nivel de conocimiento técnico han salido beneficiados de esta experiencia con GDS.

También se puede mencionar que, aunque los ejemplos mostrados son algo burdos, las posibilidades de soporte para grandes volúmenes de datos que necesitan ser analizados son muchas, y todo ello sin la necesidad de desarrollar los algoritmos desde cero o replicarlos fidedignamente, puesto que estos ya se encuentran disponibles tan solo con instalar la librería utilizada.