Instituto Tecnológico de Buenos Aires

Trabajo Práctico Especial

Paradigma NoSQL - Bases de Datos de Grafos (72.92)

Alumno:

Santiago José Hirsch - Legajo N° 62169 (shirsch@itba.edu.ar)

Profesores:

Gómez, Leticia

Vaisman, Alejandro Ariel

Introducción

El presente informe documenta la solución del Trabajo Práctico Especial de la materia Paradigma NoSQL - Bases de Datos de Grafos (72.92). El objetivo principal del trabajo consiste en diseñar, implementar y desplegar una solución capaz de calcular el K-core de un grafo dirigido. La definición del concepto K-core es, *"El subgrafo conexo más grande en el que el grado mínimo es mayor o igual a K".*

El desarrollo requiere implementar el algoritmo que calcule el K-core, sin utilizar librerías externas que brindan esta funcionalidad, utilizando una de las dos opciones vistas en clase. *GraphX* o *GraphFrames*. Además, la aplicación debe ejecutarse en el sistema distribuido provisto por la cátedra con un comando parametrizado mediante *spark-submit*. Luego, la solución debe tanto almacenar en *HDFS* los resultados del algoritmo como imprimir en pantalla.

Finalmente, el trabajo se debe preparar con un proyecto *Maven* en donde se encuentre la implementación realizada y se debe provisionar tanto grafos de prueba como la documentación necesaria para comprender las decisiones de diseño. En este informe se detalla la solución, las distintas versiones utilizadas y los distintos casos de prueba.

Instructivo

Requisitos

Para poder ejecutar la solución se deben tener los siguientes requisitos:

- *mvn*
- *Java 8*
- *Spark + Hadoop*
- *GraphFrames*

Compilación

Luego, se debe compilar el proyecto *Maven*. Para esto se debe ejecutar el siguiente comando desde la terminal en la raíz del proyecto:

```
mvn clean install
```

Archivos necesarios

Previo a la ejecución del proyecto se deben cargar los distintos archivos al nodo. Se deben cargar los siguientes archivos:

- *TPE-GRAFOS-1.jar* (.jar resultante de la compilación del proyecto)
- *graphframes-0.8.0-spark2.4-s_2.11.jar* (.jar de *GraphFrames* instalado manualmente)
- *nodes.csv* (archivo de nodos de prueba)
- *edges.csv* (archivo de aristas de prueba)

Grafos de prueba

Los grafos de prueba se encuentran dentro de la carpeta *TPE-GRAFOS/src/main/resources*. Existen 5 grafos de prueba donde el grafo bajo el nombre de *"example0"* es el provisto por la cátedra. Este mismo listado de grafos se puede encontrar en el nodo bajo la ruta *node1/user/shirsch/examples.* La estructura de la misma es la siguiente:

```
.
├── example0
│   ├── edges.csv
│   └── nodes.csv
├── example1
│   ├── edges.csv
│   └── nodes.csv
├── example2
│   ├── edges.csv
│   └── nodes.csv
├── example3
│   ├── edges.csv
│   └── nodes.csv
└── example4
    ├── edges.csv
    └── nodes.csv

6 directories, 10 files
```

Ejecución

Para ejecutar el proyecto se debe correr el siguiente comando, reemplazando los valores encerrados entre <> por los valores reales:

spark-submit --master yarn --deploy-mode=cluster --class ar.edu.itba.graph.KCore --jars hdfs://node1/<user>/<ruta_graphframes.jar> hdfs://node1/<user>/<ruta_TPE-GRAFOS-1.jar> hdfs:///user/<ruta_nodes.csv> hdfs:///user/<ruta_edges.csv> <K>

Implementación

La implementación de la solución se realizó en *Java* utilizando la librería *GraphFrames* en conjunto con *Spark* y *Hadoop* en un proyecto *Maven*.

Versiones utilizadas

La siguiente lista detalla las versiones utilizadas en el proyecto:

- *mvn 3.9.10*
- *Java 8*
- *GraphFrames 0.8.2-spark2.4-s_2.11*
- *Spark 2.4.8*
- *Hadoop 2.7*

Estructura del proyecto

El desarrollo del proyecto se realizó priorizando la modularización y la limpieza del código. La estructura del mismo es la siguiente:

```
.
├── pom.xml
└── src
    └── main
        └── java
            └── ar
                └── edu
                    └── itba
                        └── graph
                            ├── KCore.java
                            └── utils
                                ├── ErrorUtils.java
                                ├── GraphFrameUtils.java
                                ├── MainUtils.java
                                ├── SparkUtils.java
                                └── VarsUtils.java

9 directories, 7 files
```

K-core Decomposition

La implementación del algoritmo toma como parámetros el grafo dirigido y el valor de K. En primer lugar, se transforma el conjunto de bordes del grafo original en un conjunto de bordes no dirigidos. Para ello, cada borde se normaliza de manera que se cumpla que el *Id* del *src* sea menor al *Id* del *dst*, ignorando así la dirección original de los bordes. Esto puede

causar que hayan bordes duplicados por lo que se eliminan en ese mismo paso. Un ejemplo sería el siguiente, se tienen los bordes 10 - 20 y 20 - 10. Como se debe ignorar la dirección, esa conexión debería contar como una conexión y no como dos. Se modifican los bordes para ignorar dirección y quedan como 10 - 20 y 10 - 20. Por lo tanto se elimina el borde duplicado.

Una vez construido este grafo no dirigido, comienza el proceso iterativo. En cada iteración, calcula el grado de cada vértice a partir de sus vecinos y filtra únicamente aquellos cuyo grado es mayor o igual que K. Una vez obtenido este conjunto reducido de vértices, se reconstruye el subgrafo conservando sólo los bordes entre los vértices válidos. El proceso se repite hasta que en una iteración no se elimine ningún vértice adicional, así obteniendo el K-core final.

Pseudocódigo

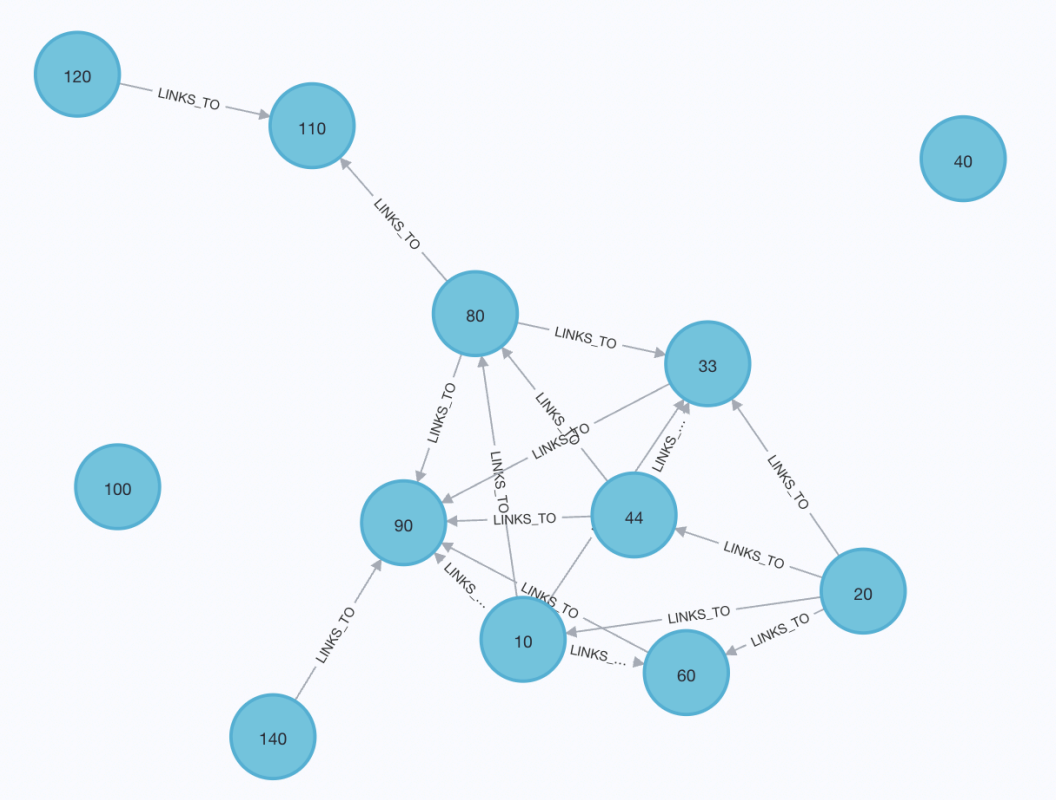
```
FUNCIÓN KCoreDecomposition(grafo, k):
  currentGraph ← grafo

  // 1. Convertir aristas dirigidas en no dirigidas solo una vez
  undirectedEdges ← conjunto vacío
  PARA cada arista (src, dst) en currentGraph.edges:
    si src < dst:
      agregar (src, dst) a undirectedEdges
    si no:
      agregar (dst, src) a undirectedEdges
  undirectedEdges ← eliminar_duplicados(undirectedEdges)

  // Crear grafo inicial no dirigido
  currentGraph ← grafo construido con (currentGraph.vertices, undirectedEdges)

  // 2. Iterar hasta que no haya más cambios
  repetir:
    edges ← currentGraph.edges

    // Construir lista de vecinos de cada vértice
    neighbors ← conjunto vacío
    PARA cada arista (u, v) en edges:
      agregar (id = u, neighbor = v) a neighbors
      agregar (id = v, neighbor = u) a neighbors

    // Calcular grados a partir de vecinos únicos
    degrees ← agrupar neighbors por id y contar vecinos distintos

    // Filtrar vértices cuyo grado ≥ k
    filteredVertices ← conjunto vacío
    PARA cada vértice (id, nombre) en currentGraph.vertices:
      si degrees[id] ≥ k:
        agregar (id, nombre) a filteredVertices

    // Filtrar solo aristas cuyos extremos permanecen en el subgrafo
    filteredEdges ← conjunto vacío
    PARA cada arista (src, dst) en edges:
      si src ∈ filteredVertices Y dst ∈ filteredVertices:
        agregar (src, dst) a filteredEdges

    newGraph ← grafo construido con (filteredVertices, filteredEdges)

    changed ← (cantidad_de_vertices(newGraph) ≠ cantidad_de_vertices(currentGraph))
    currentGraph ← newGraph
  HASTA que changed sea falso
  retornar currentGraph
```

Ejemplos

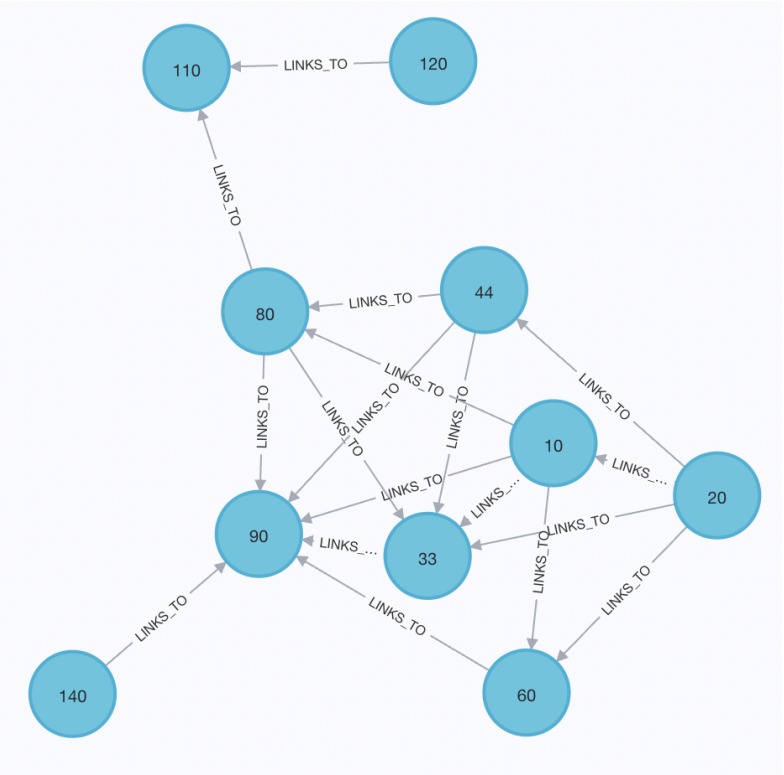
Dentro del directorio *TPE-GRAFOS/src/main/resources* se encuentra un listado de grafos de prueba que pueden ser utilizados. Este mismo listado de grafos se puede encontrar en el nodo bajo la ruta *node1/user/shirsch/examples*. Cada grafo de prueba tiene su característica unica.

Ejemplo 0

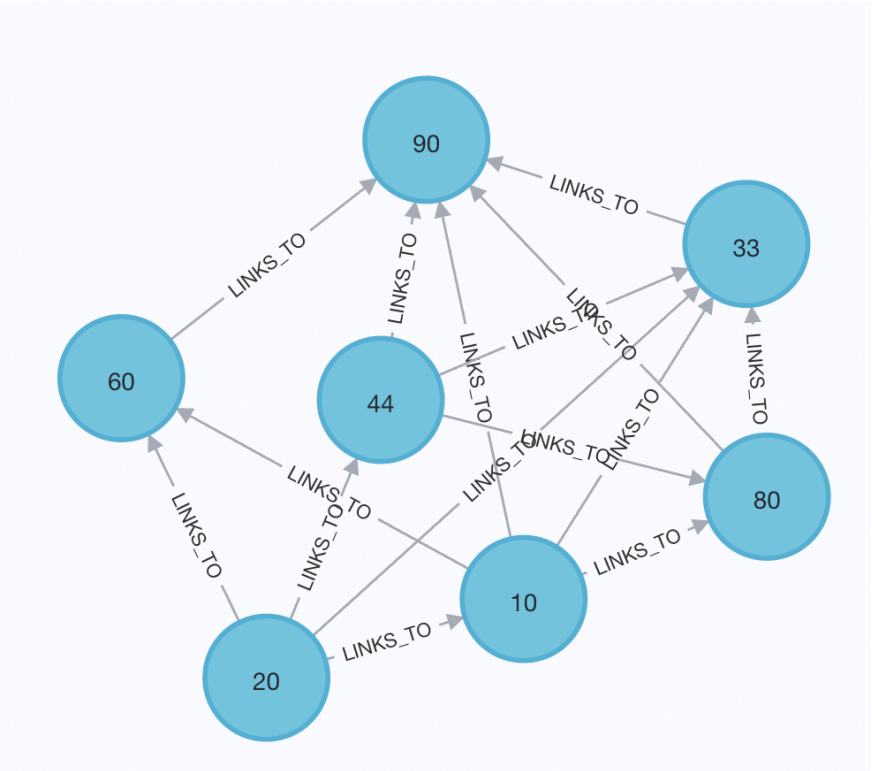
El *ejemplo 0* o *example0* es el provisto por la cátedra. El mismo se visualiza de la siguiente manera (el contenido de los nodos es el Id de cada nodo):



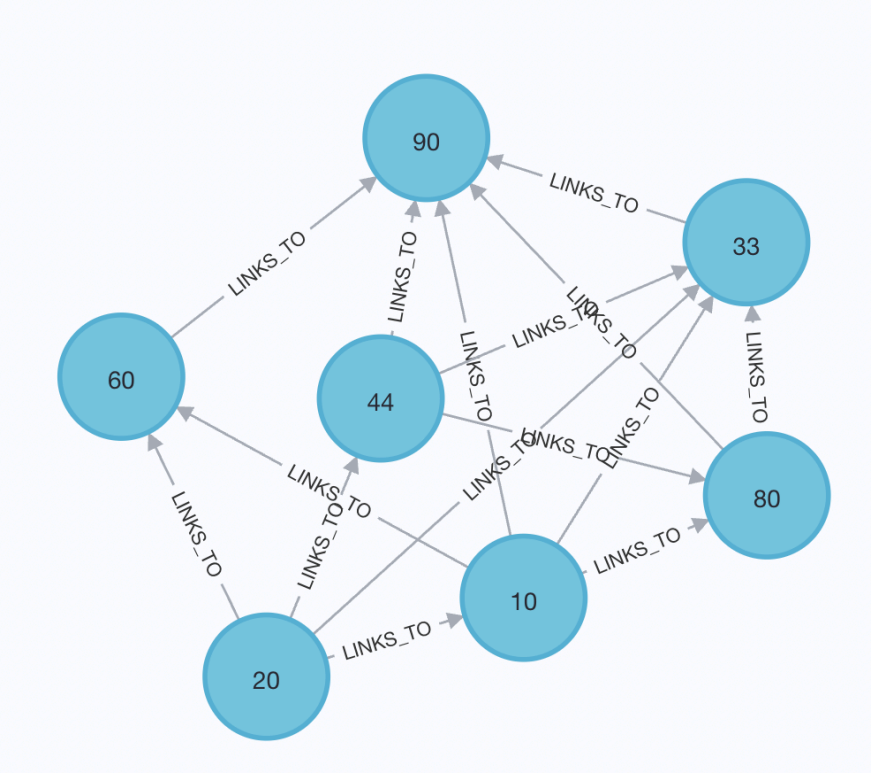
Salida con K = 1



Salida con K = 2



Salida con K = 3

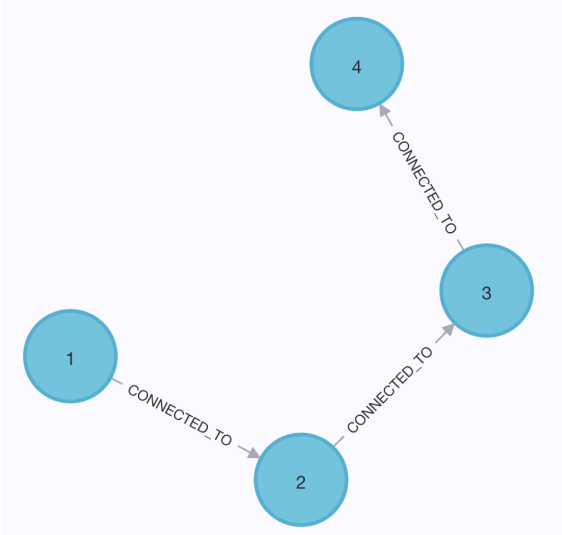


Salida con K >= 4

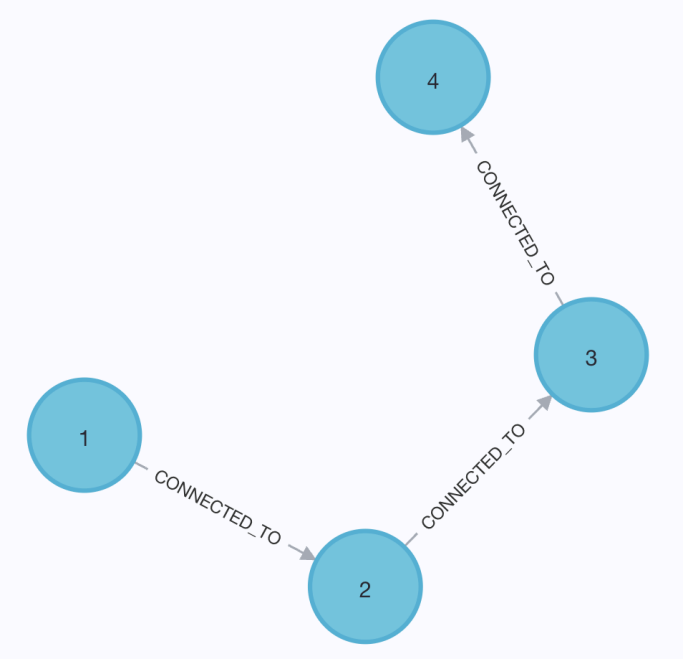
Al ejecutar el algoritmo con K >= 4, la salida es un grafo vacío.

Ejemplo 1

El *ejemplo 1* o *example1* es un grafo muy simple que permite probar fácilmente el algoritmo siendo que sólo tiene nodos de grado 1. El mismo se visualiza de la siguiente manera (el contenido de los nodos es el Id de cada nodo):



Salida con K = 1

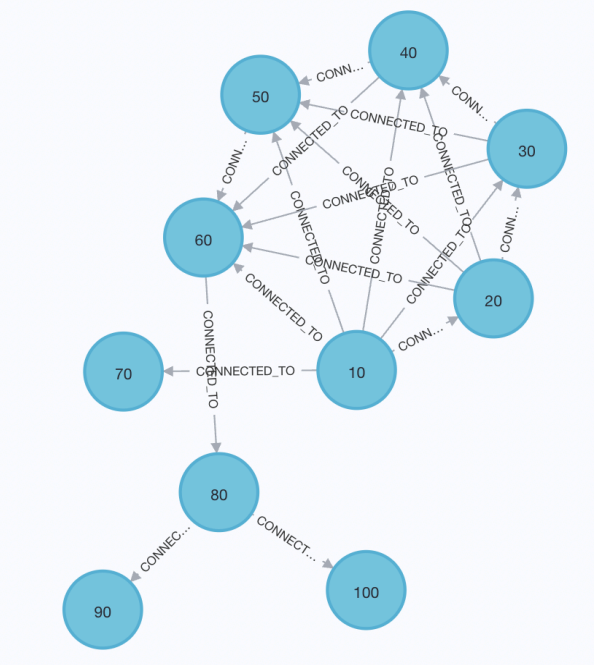


Salida con K >= 2

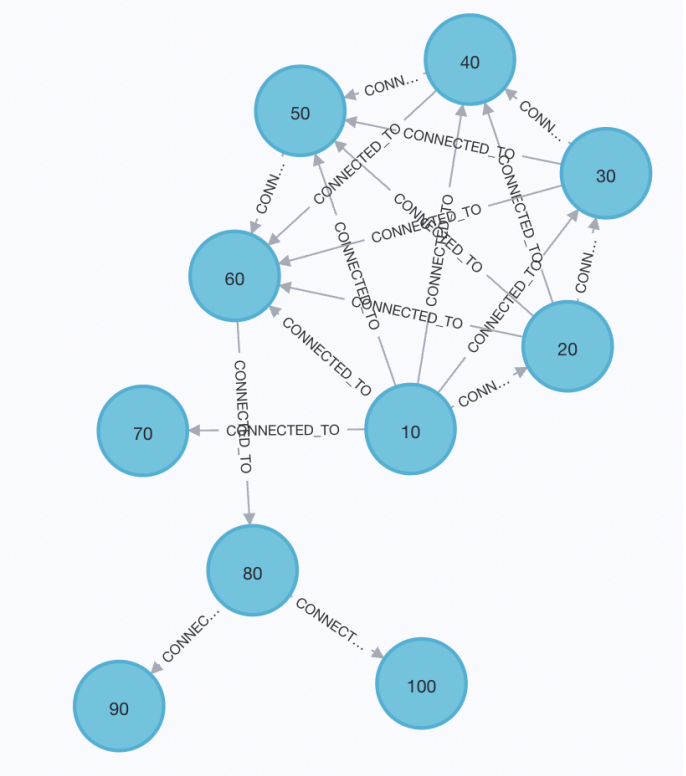
Al ejecutar el algoritmo con K >= 2, la salida es un grafo vacío.

Ejemplo 2

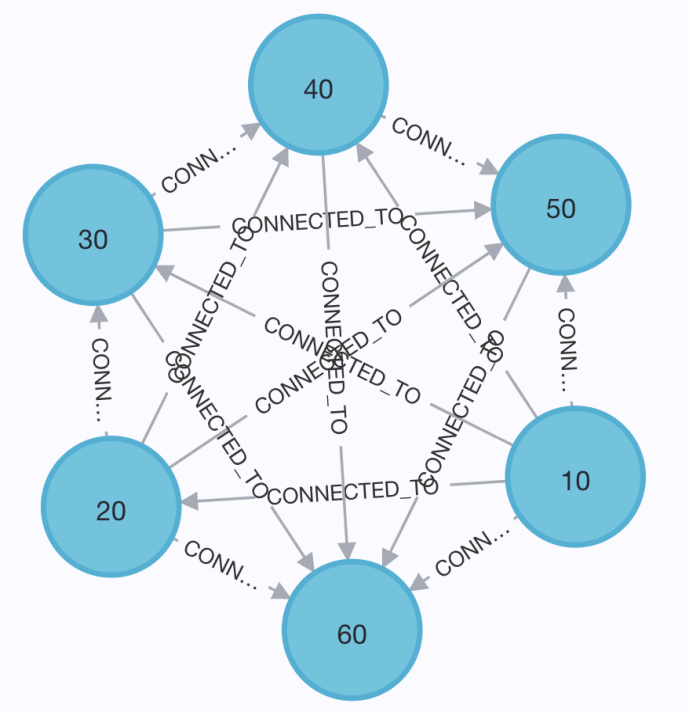
El *ejemplo 2* o *example2* es un grafo complejo que tiene muchos bordes entre los nodos. Este permite probar el algoritmo con valores de K elevados. El mismo se visualiza de la siguiente manera (el contenido de los nodos es el Id de cada nodo):



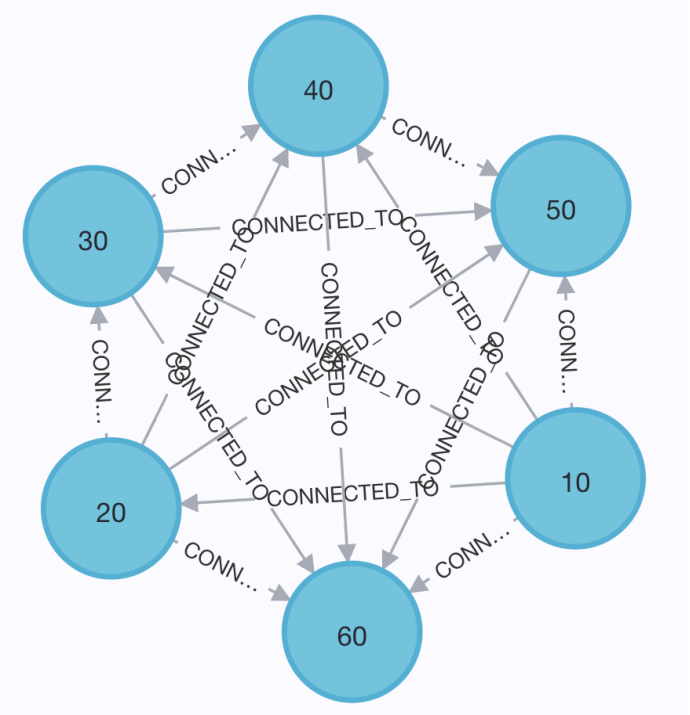
Salida con K = 1



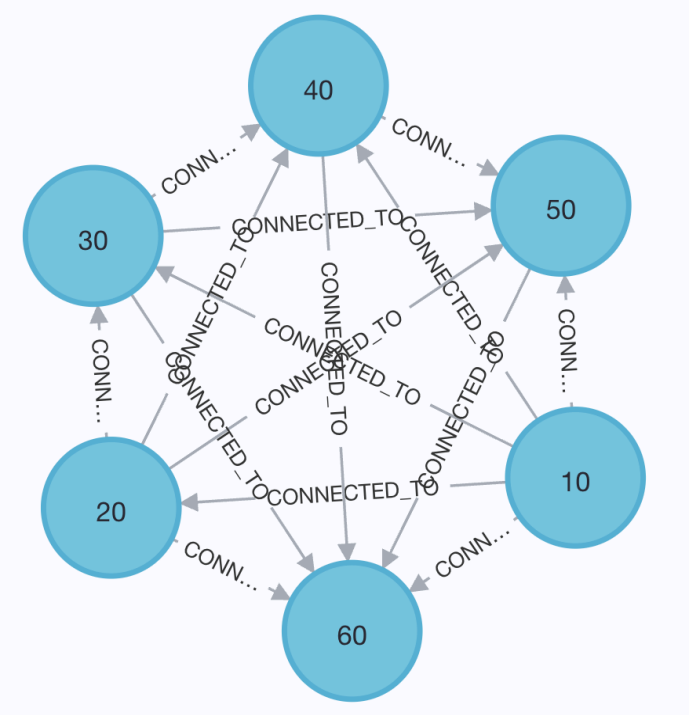
Salida con K = 2



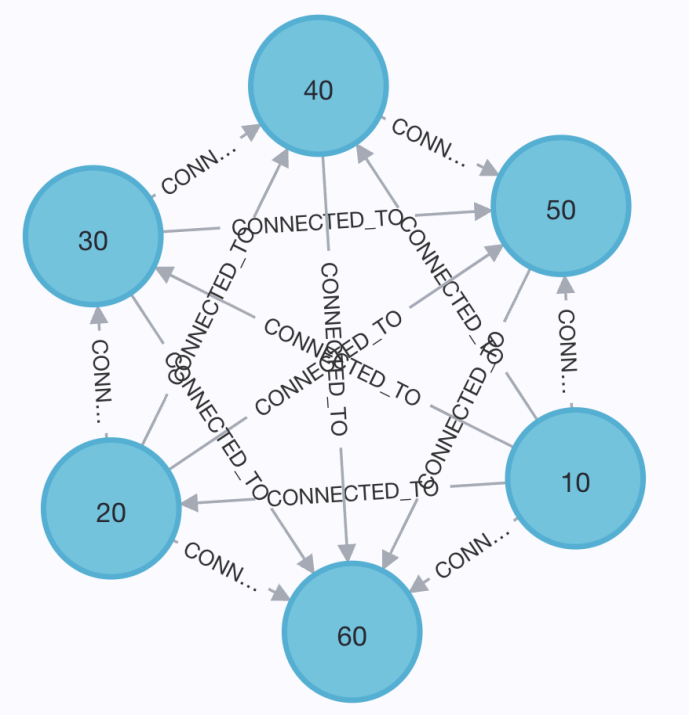
Salida con K = 3



Salida con K = 4



Salida con K = 5

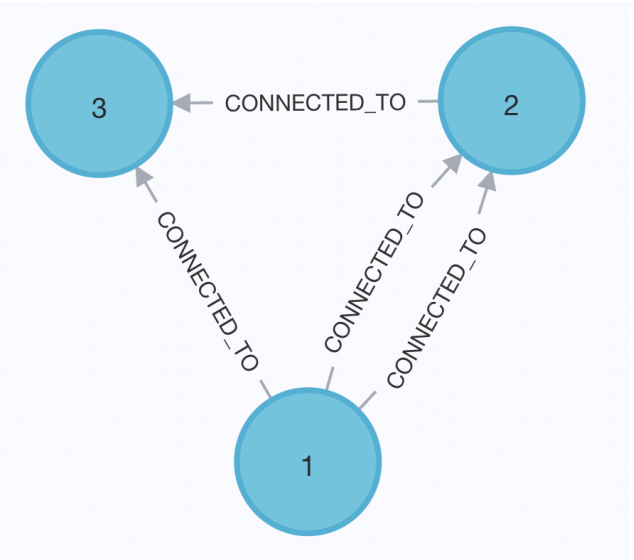


Salida con K >= 6

Al ejecutar el algoritmo con K >= 6, la salida es un grafo vacío.

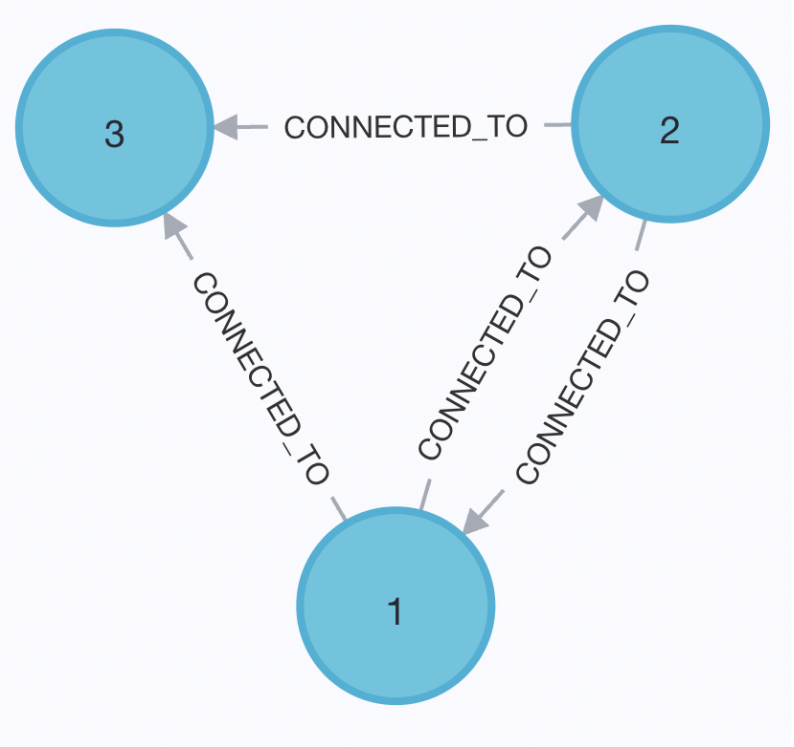
Ejemplo 3

El *ejemplo 3* o *example3* es un grafo muy simplificado con un borde duplicado, causando que sea un multigrafo. Este permite probar la funcionalidad del proyecto para detectar multigrafos y mostrar el error en pantalla. El mismo se visualiza de la siguiente manera (el contenido de los nodos es el Id de cada nodo):



Ejemplo 4

El *ejemplo 4* o *example4* es un grafo muy simplificado con un borde dirigido en ambas direcciones, causando que sea un multigrafo. Este permite probar la funcionalidad del proyecto para detectar multigrafos y mostrar el error en pantalla. El mismo se visualiza de la siguiente manera (el contenido de los nodos es el Id de cada nodo):



Mensajes de error

A la hora de ejecutar el proyecto, pueden haber errores. A continuación se detallan los mensajes de error esperados con sus respectivos causantes:

Número inválido de argumentos

Este error es causado por invocar el programa *KCore* con una cantidad de argumentos distinta a la esperada, que es 3.

Mensaje

Invalid number of arguments.
Usage: java -jar <program>.jar <path_to_nodes_csv> <path_to_edges_csv> <k_value>

Argumento nulo o vacío

Este error es causado por invocar el programa *KCore* con alguno de los tres argumentos esperados como un string vacío o nulo.

Mensaje

Arguments must not be null or empty.

K <= 0

Este error es causado por invocar el programa *KCore* con un valor de K negativo.

Mensaje

The third argument (k value) must be a non-negative integer.

K no es un número entero

Este error es causado por invocar el programa *KCore* con un valor de K diferente a un número entero.

Mensaje

The third argument (k value) must be a valid integer.

Grafo de entrada inválido

Este error es causado por invocar el programa *KCore* con un grafo de entrada invalido. Que el grafo sea invalido quiere decir que el mismo es un multigrafo.

Mensaje

The underlying structure is not valid, it is a multigraph.

Ruta inexistente

Este error es causado por invocar el programa *KCore* con uno de los parámetros de los archivos de nodos o bordes como una ruta inexistente.

Mensaje

Path does not exist: <ruta_inexistente>

La ruta no es un archivo

Este error es causado por invocar el programa *KCore* con uno de los parámetros de los archivos de nodos o bordes como una ruta a un objeto que no es un archivo.

Mensaje

Path is not a file: <ruta>