Graph Gurus 29 demo instructions

This exercise from Graph Gurus 29 demonstrates three community detection algorithms: Connected Components, K-Core, and Louvain Modularity.

These instructions should be distributed along with a GraphStudio solution file (export_GG29_community.tar.gz) and data folder containing 7 files. Optionally, it may include several queries; the queries are included in the solution file.

Datasets

There are three separate unconnected datasets, representing a variety of sizes and connection patterns.

1. Drug Prescribers: 104 vertices, 417 directed edges
    Our data files:
        specialties-subspecialties.csv
        subspecialties-prescriber.csv
        claim.csv
        claim submitted_by prescriber.csv
        claim associated with patient.csv

2. Kangaroo social network 17 vertices, 91 undirected edges
    Original source: http://konect.uni-koblenz.de/networks/moreno_kangaroo
    Our data file: moreno_kangaroo.csv

3. Flickr Images 105,938 vertices 2,316,948 undirected edges
    Original source: http://konect.uni-koblenz.de/networks/flickrEdges
    Our data file: flickrEdges.csv

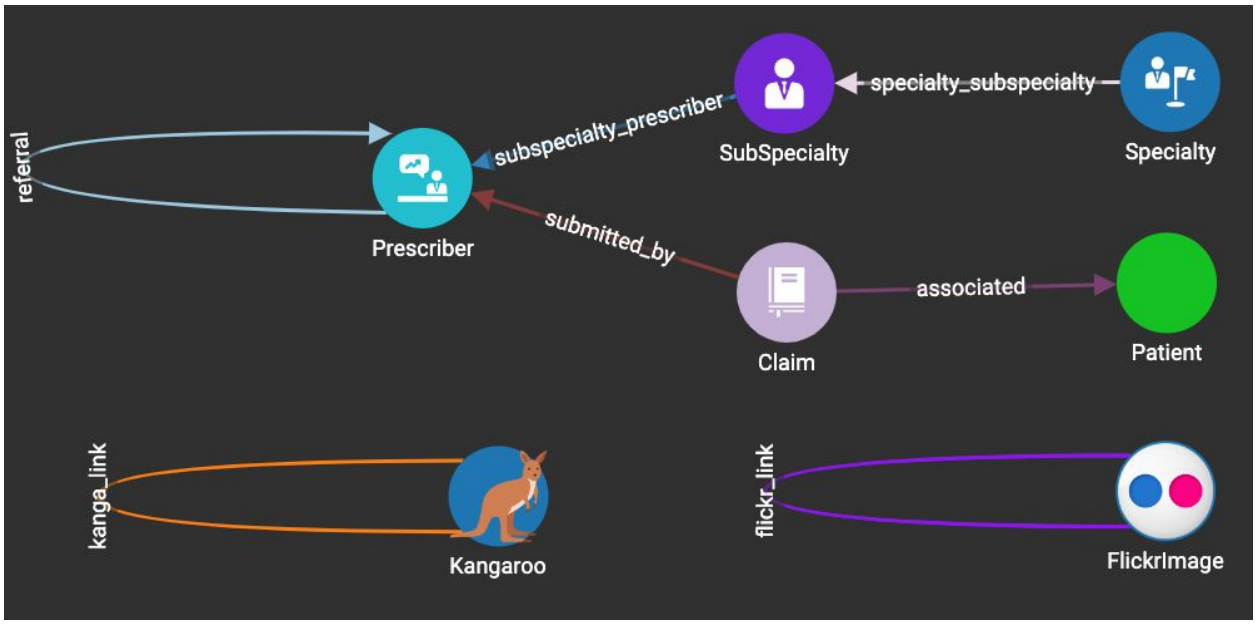
Install the solution file called export_GG29_community.tar.gz, which bundles the schema, loading jobs, and queries used in this exercise. **Follow these steps:**

(1) Start on the main page of GraphStudio. Import the solution file export_GG29_community.tar.gz.
(2) Go to the Map Data to Graph page.
    (a) In the top menu, click on the Add Data File icon. One at a time, click the (+) icon to add each of the 7 data files. Within the Add Data File panel, click each file name, select the Has header checkbox, and click ADD at the bottom right, to load the data file.
    (b) Note that this will place a duplicate file icon for each file on the schema diagram. You can delete these duplicates (they ones which are not linked to a vertex or edge type).
    (c) Finally, click the Publish Data Mapping up-arrow button.

(3) Go to the Load Data page. Wait a few seconds for the Load (right arrow) button to become active. Click it and wait for the data to be loaded.
(4) Go to the Write Queries page. Click the Install All Queries button, to the left of the "GSQL Queries" heading.

Schema

The schema for the Prescriber dataset has 5 vertex types and 5 edge types. We are to focus on Prescribers. The other two datasets are simple social networks with one vertex type each (Kangaroo or FlickrImage) with undirected relationship edges between vertices.



Data Preparation before Querying

We show a Prescriber-referral-Prescriber edge, but this relationship is not in our source data. Instead, we are going to infer a relationship if the following condition exists:

If Patient X sees Prescriber Y at time T1,
and Patient X also sees Prescriber Z at a later time T2,
then infer a referral edge from Y to Z.

The query ex2_createReferral checks the entire Prescriber graph and creates edges wherever this relationship exists.

Run insert_all_referrals (previously called ex2_main_query, not very descriptive). It is very fast. After running it, you can switch to the Load Data page to look at the graph statistics on the right. You should see 417 referral edges.

Community Detection Queries

We used three algorithms which are in our GSQL Graph Algorithm Library: https://github.com/tigergraph/gsql-graph-algorithms

Some other community detection algorithms not covered in Graph Gurus 29:

- Label Propagation

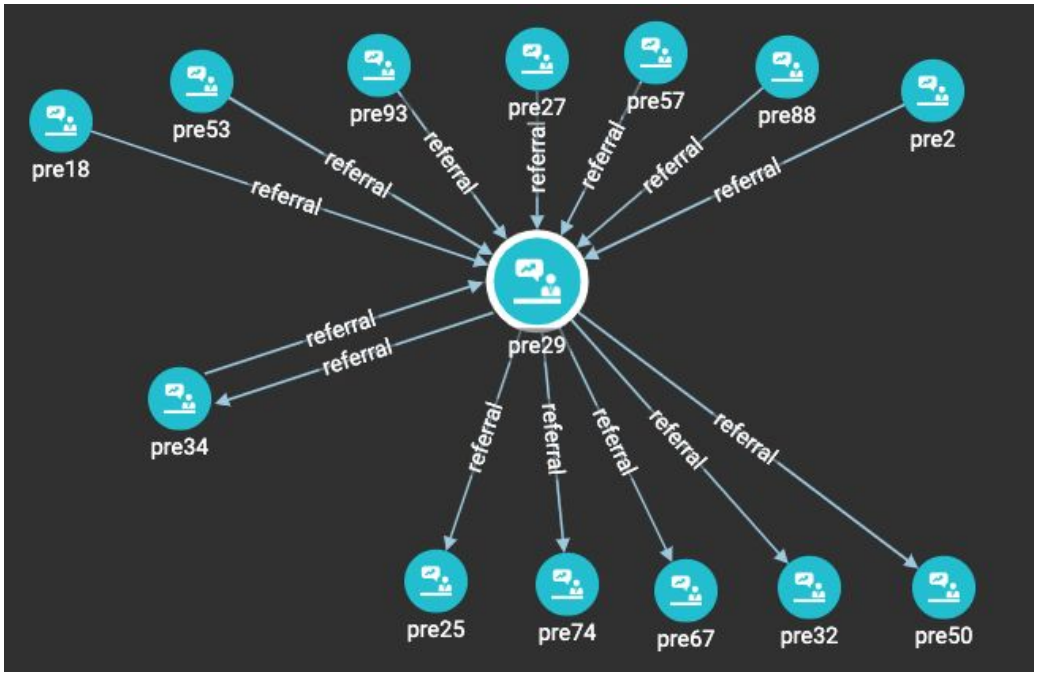
Louvain Modularity

(The following exercise is more detailed and illustrative -- better -- than the webinar.)

1. **Run algo_pageRank**, with outputLimit = 100. It is currently hardcoded to run on Prescriber vertices and referral edges. Look at the JSON output which lists the most influential prescribers first. Code the first ~5 prescribers to a text file for reference:


```
{
  "@@topScores": [
    {
      "Vertex_ID": "pre29",
      "score": 3.30608
    },
    {
      "Vertex_ID": "pre69",
      "score": 2.56451
    },
    {
      "Vertex_ID": "pre23",
      "score": 2.34658
    },
    {
      "Vertex_ID": "pre53",
      "score": 2.21938
    },
    {
      "Vertex_ID": "pre61",
      "score": 2.04001
    },
```


Prescriber pre29 has the highest PageRank score. If you go to the Explore Graph page, display this vertex (Search vertices by vertex id), then show its referral edges (on the left menu, got to Expand from vertices, select only referral edges, then click Expand), and then if you tidy up the display, it'll look something like this:



2. Now **run algo_louvain_enhanced** with the following inputs:
   vertexType = Prescriber
   edgeType = referral
   revEdgeType = reverse_referral

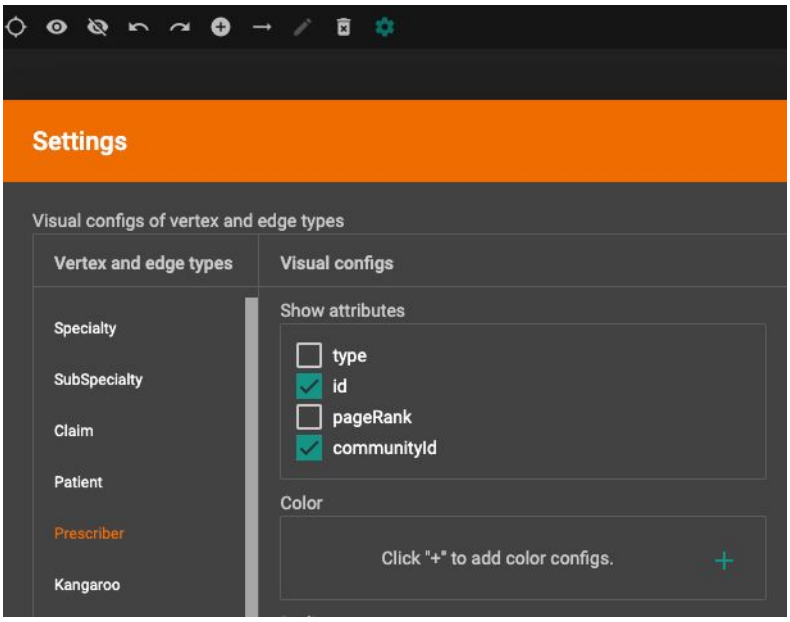
   See the video for an explanation of the output.

3. Your data set is a little different than the data in the webinar. You should probably see this JSON output, around line 75:

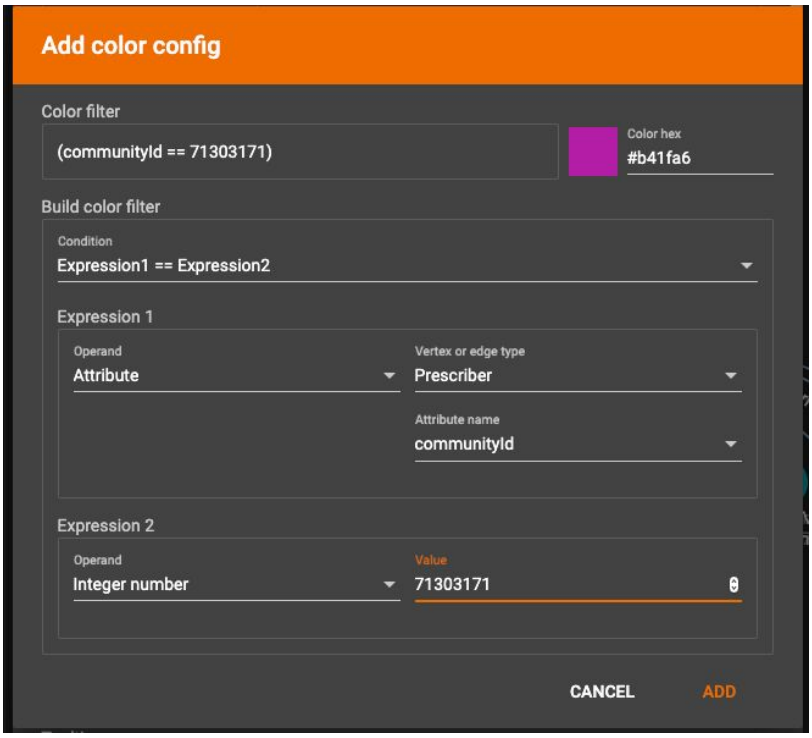

```
{
  "@@clusterMap": {
    "4": 1,
    "5": 2,
    "7": 7,
    "8": 4,
    "9": 1
  }
 },
 {
  "@@clusterListsBySize": {
    "4": [
      81788928
    ],
    "5": [
      73400323,
      71303168
    ],
    "7": [
      77594624,
      79691780,
      75497472,
      84934656,
      80740353,
      75497474,
      71303171
    ],
    "8": [
      73400324,
      71303170,
      71303169,
      72351747
    ],
    "9": [
      72351744
    ]
  }
 },
```


4. So that you maintain these query results, open a new browser window. In that browser, go back to the Explore Graph window, which should still show your pre29+neighbors graph.
   a. Move the cursor over vertex pre29 to see its attribute values. The communityId is 71303171 in my case. Every vertex in the same community should have the same community value.
   b. Click the settings (gear) button in the top menu. Select the Prescriber type and

c. then select the communityId checkbox. Don't click APPLY yet.



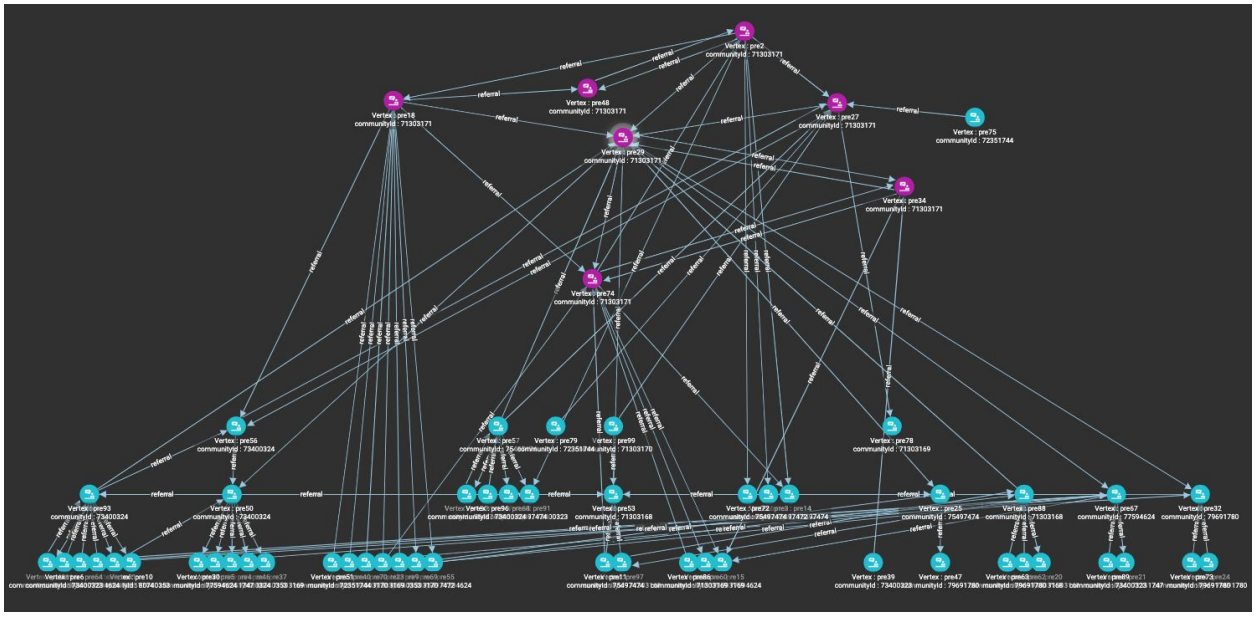
d. Next, add (+) a Color configuration. Pick any distinctive color you like. Now we want to add a condition "Prescriber.communityId = 71303171". GraphStudio supports many types of conditions, so you'll have to go through several steps, but they're logical. When you finish, the condition is expressed like this:



Now you can click ADD and then APPLY.

e. You should now see that pre29 and 5 of its neighbors now have your special color. But not all of its neighbors -- why not? Because we haven't looked at all the graph's connections, and those other vertices probably have even stronger connections to a different community.

f. If you display all the vertices within 2 hops of pre29, both neighbors and neighbors of neighbors, it'll look like this:



The purple vertices are pre29's community. Prescriber pre29 is in the center of the purple cluster, with those that make a referral to pre29 located above it (so that referral is downward).

K-Core

K is a positive integer, and for each value of K, a K-core is the interconnected community where each member is connected to at least K other members. A K-shell are those vertices "on the edge" which have just enough connections to belong to the K-core but not the next denser level, K+1.

1. We started with the Kangaroo data set, with only 17 vertices and 91 edges. Kangaroo 17 has only one connection, and Kangaroo 16 has only 2. The rest have many connections.
    a. The algorithm kcore_decomp tells you the size of each k-core, for each value of k. It can also tell you the membership of each k-core. **Run kcore_decomp, with the following inputs:**
    vertexType = Kangaroo
    edgeType = kanga_link


```
[
  {
    "k": 9,
    "@@kcoreVertices.size()": 13
  },
  {
    "k": 8,
    "@@kcoreVertices.size()": 15
  },
  {
    "k": 2,
    "@@kcoreVertices.size()": 16
  },
```

```
 {
  "k": 1,
  "@@kcoreVertices.size()": 17
 }
]
```


There is a core of 13 vertices which have at least 9 connections to one another, and a slight larger group (15) which have at least 8 connections. The k=2 and k=1 cores apparently add the two low-connectivity vertices that we mentioned before.

b. **Run kcore_decomp again, but this time, set showMembership = true and showShells = true.** This time, it will also list the membership of each shell:


```
 {
  "k": 9,
  "@@coreListMap.get(k)": [
   "2",
   "1",
   "5",
   "6",
   "11",
   "9",
   "8",
   "10",
   "15",
   "7",
   "3",
   "12",
   "4"
  ]
 },
 {
  "k": 8,
  "@@coreListMap.get(k)": [
   "13",
   "14"
  ]
 },
 {
  "k": 2,
  "@@coreListMap.get(k)": [
   "16"
  ]
 },
 {
  "k": 1,
  "@@coreListMap.get(k)": [
   "17"
  ]
 }
]
```


We not only find who is in the densest core, we also confirm that kangaroos 16 and 17 form the two outer shells.

c. The algorithm kcore_max is designed to find the densest core, the k-core for the highest value of k. If you set verbosity=1, then it will also report how it peels away

the outer shells, giving you much the same information as kcore_decomp.

2. If you run Louvain on the Kangaroo dataset (algo_louvain_enhanced, vertexType = Kangaroo, edgeType = kanga_link), you'll see that the largest community it finds has only 3 members, and it finds 10 communities having only a single member each. Not every algorithm gives helpful results for every graph.

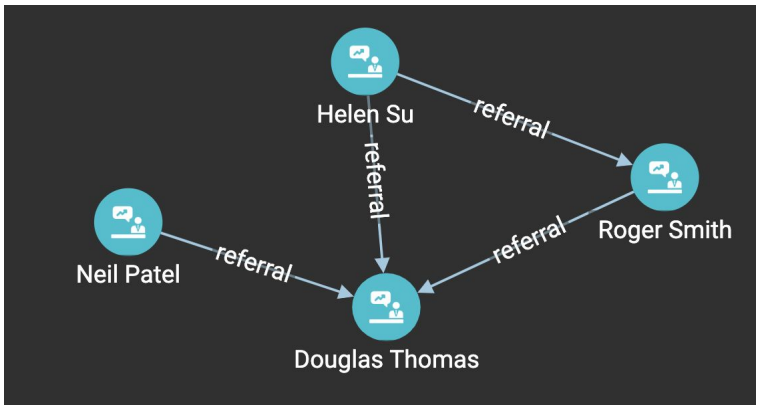
3. Try **running the kcore_decomp algorithm on the Flickr dataset** (vertexType = FlickrImage, edgeType = flickr_link). There are over 100,000 vertices and 2 million edges in this dataset. There are two interesting things about the results here.
    a. The max K-core has 574 members with K=573. Since k is just 1 less than the next of members, this means every possible connection actually exists. When you have a group where every vertex connects to every other one, this is called a **Complete Graph** or a **Clique**.
    b. If you scroll to the end of the results (K=1) and move backward, pay attention to changes in size. There are only small changes from K = 1 to 5, then a big jump at 6. Why? This is hard to say without understanding something about the real-world data that this is modeling, but it is a clue for what to look for. An even bigger size change is between 6 and 7. There are gradual changes all the way until 94/95.

Strongly Connected Component (SCC)

An SCC is a set of vertices for which there is a direct path from each vertex to every other vertex. That is, there is a way to go from anywhere to anywhere.

The scc_enhanced algorithm gives a bit more informative output than the standard query in the algorithm library. **Run the scc_enhanced query.** It has been hardcoded for the Prescriber graph (the only one with directed edges), so you don't need to set any input parameters.

In the JSON output, you will see the 99 of the 104 vertices all belong to the one main component. That means this component is highly interconnected. They remaining 5 vertices are singletons. One of them, pre72, is part of the regular graph. The other 4 have human names like "Helen Su" instead of id codes. Those 4 form a connected component, but not a strongly connected component, because the paths are one way without any return loop.



Connected Component

A Connected Component is a set of vertices in an undirected graph for which there is a direct path from each vertex to every other vertex. One can apply the same criteria to a directed graph if you ignore the direction of an edge. This special ignore-the-direction case is called the **weakly connected component** criterion.

Run the conn_comp query on the Flickr dataset (vertexType = FlickrImage, edgeType = flickr_link).

The output begins with a list of component IDs and the component sizes. Component 105906196 has the vast majority of vertices -- 105722. This is a common phenomenon. In network theory, this one big component is called the **giant component.** Then there are numerous small components, having only 2 or 3 … at most 11 members.

Finding connected components or the giant component is an important preparatory step for many algorithms. Consider PageRank. PageRank assumes that you are working with a connected graph. First find the giant component. Then run PageRank on it.

###