# Programming Assignment 3<br>Graph Processing using Map-Reduce

Due on Tuesday March 24 before midnight

## Description

The purpose of this project is to develop a graph analysis program using Map-Reduce.

This project must be done individually. No copying is permitted. **Note: We will use a system for detecting software plagiarism, called Moss, which is an automatic system for determining the similarity of programs.** That is, your program will be compared with the programs of the other students in class as well as with the programs submitted in previous years. This program will find similarities even if you rename variables, move code, change code structure, etc.

Note that, if you use a Search Engine to find similar programs on the web, we will find these programs too. So don't do it because you will get caught and you will get an F in the course (this is cheating). Don't look for code to use for your project on the web or from other students (current or past). Just do your project alone using the help given in this project description and from your instructor and GTA only.

## Platform

As in other projects, you will develop your program on SDSC Comet. Optionally, you may use your laptop/PC to develop your program first and then, after you make sure that it works there, you transfer it and test it to Comet. You may also use IntelliJ IDEA or Eclipse to help you develop your program, if you have done so in Projects 1 and 2. Note that it is required that you test your programs on Comet before you submit them.

## Setting up your Project

Login into Comet and download and untar project3:

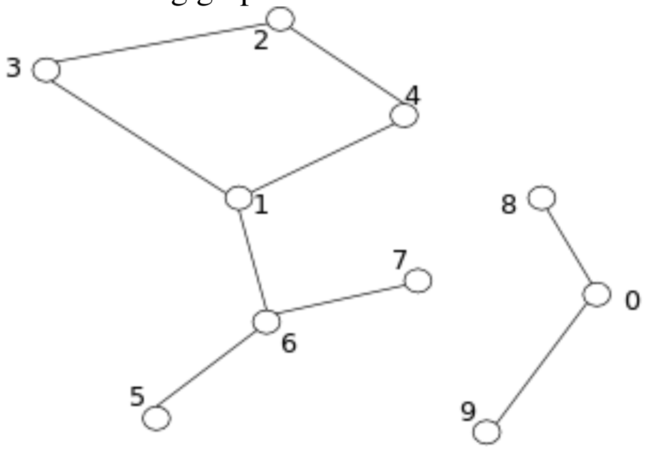
```
wget http://lambda.uta.edu/cse6331/project3.tgz
tar xfz project3.tgz
chmod -R g-wrx,o-wrx project3
```

## Project Description

An undirected graph is represented in the input text file using one line per graph vertex. For example, the line

```
1,2,3,4,5,6,7
```

represents the vertex with ID 1, which is connected to the vertices with IDs 2, 3, 4, 5, 6, and 7. For example, the following graph:



is represented in the input file as follows:

```
3,2,1
2,4,3
1,3,4,6
5,6
6,5,7,1
0,8,9
4,2,1
8,0
9,0
7,6
```

Your task is to write a Map-Reduce program that finds the connected components of any undirected graph and prints the size of these connected components. A connected component of a graph is a subgraph of the graph in which there is a path from any two vertices in the subgraph. For the above graph, there are two connected components: one 0,8,9 and another 1,2,3,4,5,6,7. Your program should print the sizes of these connected components: 3 and 7.

The following pseudo-code finds the connected components. It assigns a unique group number to each vertex (we are using the vertex ID as the group number), and for each graph edge between Vi and Vj, it changes the group number of these vertices to the minimum group number of Vi and Vj. That way, vertices connected together will eventually get the same minimum group number, which is the minimum vertex ID among all vertices in the connected component. First you need a class to represent a vertex:

```
class Vertex extends Writable {
  short tag;                 // 0 for a graph vertex, 1 for a group number
  long group;                // the group where this vertex belongs to
  long VID;                  // the vertex ID
  Vector adjacent;     // the vertex neighbors
  ...
}
```

Class Vertex must have two constructors: Vertex(tag,group,VID,adjacent) and Vertex(tag,group).

First Map-Reduce job:

```
map ( key, line ) =
  parse the line to get the vertex VID and the adjacent vector
  emit( VID, new Vertex(0,VID,VID,adjacent) )
```

Second Map-Reduce job:

```
map ( key, vertex ) =
  emit( vertex.VID, vertex )   // pass the graph topology
  for n in vertex.adjacent:
     emit( n, new Vertex(1,vertex.group) )  // send the group # to the adjacent vertices

reduce ( vid, values ) =
  m = Long.MAX_VALUE;
  for v in values {
     if v.tag == 0
        then adj = v.adjacent.clone()     // found the vertex with vid
     m = min(m,v.group)
  }
  emit( m, new Vertex(0,m,vid,adj) )      // new group #
```

Final Map-Reduce job:

```
map ( group, value ) =
   emit(group,1)

reduce ( group, values ) =
   m = 0
   for v in values
       m = m+v
   emit(group,m)
```

The second map-reduce job must be repeated multiple times. For your project, repeat it 5 times. You can repeat a job, by putting the job in a for-loop. The args vector in your main program has the path names: args[0] is the input graph, args[1] is the intermediate directory, and args[2] is the output. The first Map-Reduce job writes on the directory `args[1]+"/f0"`. The second Map-Reduce job reads from the directory `args[1]+"/f"+i` and writes in the directory `args[1]+"/f"+(i+1)`, where `i` is the for-loop index you use to repeat the second Map-Reduce job. The final Map-Reduce job reads from `args[1]+"/f5"` and writes on `args[2]`. The intermediate results between Map-Reduce jobs must be stored using SequenceFileOutputFormat.

An empty `project3/src/main/java/Graph.java` is provided, as well as scripts to build and run this code on Comet. **You should modify Graph.java only**. There is one small graph in `small-graph.txt` for testing in local mode. It is the graph shown in the figure above. Then, there is a moderate-sized graph `large-graph.txt` for testing in distributed mode. The solution for the large graph is given at `solution-large.txt`.

You can compile Graph.java using:

```
run graph.build
```

and you can run it in standalone mode over the small graph using:

```
sbatch graph.local.run
```

You should modify and run your programs in standalone mode until you get the correct result. After you make sure that your program runs correctly in standalone mode, you run it in distributed mode using: