Graph Processing using Map-Reduce

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 (Links to an external site.)**](http://theory.stanford.edu/~aiken/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 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 Project 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

A directed 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 linked to the vertices with IDs 2, 3, 4, 5, 6, and 7. Your task is to write a Map-Reduce program that partitions a graph into K clusters using multi-source BFS (breadth-first search). It selects K random graph vertices, called centroids, and then, at the first iteration, for each centroid, it assigns the centroid id to its unassigned neighbors. Then, at the second iteration. it assigns the centroid id to the unassigned neighbors of the neighbors, etc, in a breadth-first search fashion. After few repetitions, each vertex will be assigned to the centroid that needs the smallest number of hops to reach the vertex (the closest centroid). First you need a class to represent a vertex:

class Vertex {

long id; // the vertex ID

Vector adjacent; // the vertex neighbors

long centroid; // the id of the centroid in which this vertex belongs to

short depth; // the BFS depth

...

}

Vertex has a constructor Vertex( id, adjacent, centroid, depth ).

You need to write 3 Map-Reduce tasks. The first Map-Reduce job is to read the graph:

map ( key, line ) =

parse the line to get the vertex id and the adjacent vector

// take the first 10 vertices of each split to be the centroids

for the first 10 vertices, centroid = id; for all the others, centroid = -1

emit( id, new Vertex(id,adjacent,centroid,0) )

The second Map-Reduce job is to do BFS:

map ( key, vertex ) =

emit( vertex.id, vertex ) // pass the graph topology

if (vertex.centroid > 0)

for n in vertex.adjacent: // send the centroid to the adjacent vertices

emit( n, new Vertex(n,[],vertex.centroid,BFS\_depth) )

reduce ( id, values ) =

min\_depth = 1000

m = new Vertex(id,[],-1,0)

for v in values:

if (v.adjacent is not empty)

m.adjacent = v.adjacent

if (v.centroid > 0 && v.depth < min\_depth)

min\_depth = v.depth

m.centroid = v.centroid

m.depth = min\_depth

emit( id, m )

The final Map-Reduce job is to calculate the cluster sizes:

map ( id, value ) =

emit(value.centroid,1)

reduce ( centroid, values ) =

m = 0

for v in values:

m = m+v

emit(centroid,m)

The second map-reduce job must be repeated multiple times. For your project, repeat it 8 times. The variable BFS\_depth is bound to the iteration number (from 1 to 8). The args vector in your main program has the following 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]+"/i0". The second Map-Reduce job reads from the directory args[1]+"/i"+i and writes in the directory args[1]+"/i"+(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]+"/i8" and writes on args[2]. Note that the intermediate results between Map-Reduce jobs must be stored using SequenceFileOutputFormat.

A skeleton file project3/src/main/java/GraphPartition.java is provided, as well as scripts to build and run this code on Comet. **You should modify GraphPartition.java only**. There is one small graph in small-graph.txt for testing in standalone 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.

You can compile GraphPartition.java using:

run partition.build

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

sbatch partition.local.run

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

sbatch partition.distr.run

This will work on the moderate-sized graph and will write the result in the directory output-distr. Your results should match large-solution.txt.

What to Submit

You need to submit the following files only:

project3/src/main/java/GraphPartition.java

project3/partition.local.out

project3/output/part-r-00000

project3/partition.distr.out

project3/output-distr/part-r-00000 (rename it to part-r-00000\_distr first)