Tutorial:
Introduction to Giraph

Faraz Faghri
Last update: September 19, 2015

1 Overview

Welcome

Welcome to the Giraph tutorial. This tutorial covers the critical skills needed to develop a Giraph graph processing application. It starts with an already deployed Giraph environment where students will execute a series of hands-on labs.

Objectives

Upon completing this tutorial, students will be able to:

- Perform basic operations on graphs
- Run a simple Giraph application (Connected Components)
- Examine Giraph code for connected component application

Structure

This guide is designed as a set of step-by-step instructions for hands-on exercises. It teaches you how to operate the Giraph service in a functional test environment through a series of examples. You should complete the exercises in the order described in the following steps:

1. Create an input dataset
2. Prepare a HDFS and upload the dataset on it
3. Examine the **Connected Components** application
4. Run the application on Giraph and examine the output
5. (optional) Build the Giraph code base
6. (optional) Use a simple **Shortest Path** application
7. (optional) Run the application on Giraph and examine the output
8. (optional) Use a simple **PageRank** application
9. (optional) Run the application on Giraph and examine the output

2 Requirements

This tutorial is designed to work on the **Hortonworks Sandbox 2.3** virtual machine. You need to have a working HortonWorks Sandbox machine either locally or on the Amazon Web Services.

All assignments are also designed based on **JDK 7** (included in the virtual machine).

Please refer to **Tutorial: Run HortonWorks Sandbox 2.3 Locally** or Tutorial**: Run HortonWorks Sandbox 2.3 on AWS** for more information.

3 Setup Virtual Machine

Step 1: Start the virtual machine; then connect to it through the SSH.

Step 2: After successfully logging in, you should see a prompt similar to the following:

```
[root@sandbox ~]#
```

Step 3: In order to build and package Giraph applications, we need **maven**. In order to install maven, run the following commands:

```
# wget http://repos.fedorapeople.org/repos/dchen/apache-maven/epel-apache-maven.repo -O /etc/yum.repos.d/epel-apache-maven.repo
# yum install apache-maven
```

Step 4: Check the installation using the following command:

```
# mvn –version
```

Step 5: Create some working directories in HDFS where you will store the input data sets and results:

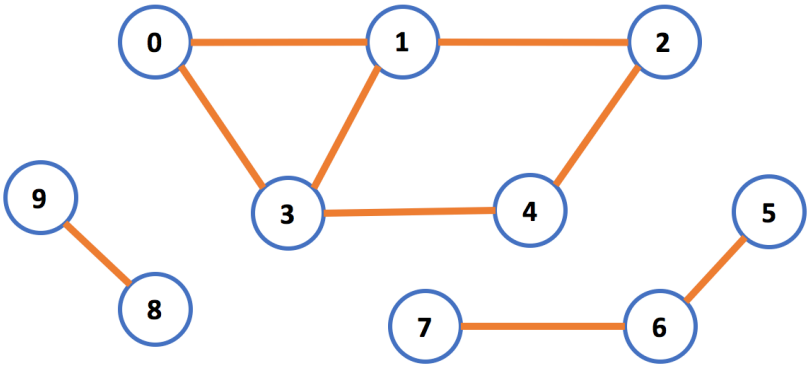
```
# hadoop fs -mkdir -p /giraph-tutorial/input
```

Step 6: Create another directory for results:

```
# hadoop fs -mkdir -p /giraph-tutorial/output
```

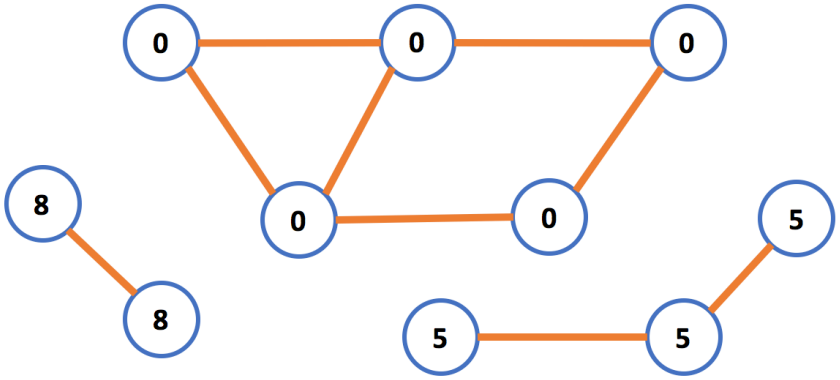
4 Connected Components

In this section, we will look at the Connected Components algorithm and how the code works. The connected component of a graph is a subgraph in which any two vertices are connected to each other by paths and which is connected to no additional vertices in the supergraph. For example, in the following graph there are three connected components:



To find connected components in the graph, each vertex will look at its neighbors and pick the smallest "vertex id" as its own id. If the id has changed, in the next step it will propagate its own id to neighbors. In later steps, the vertex will listen to neighbors, and in case a smaller id was sent, it will update its own id to smallest one.

By the end of all steps, each connected component will have the id of the smallest vertex in that subgraph.



This example is adapted from the Giraph package examples. To learn more, visit:

https://github.com/apache/giraph/blob/release-1.1/giraph-examples/src/main/java/org/apache/giraph/examples/ConnectedComponentsComputation.java

Now clone the “cloudapp-giraph-tutorial” GitHub repository, which includes the code, build it, and try it out:

Step 1: Clone the following GitHub repository:

```
# git clone https://github.com/ffaghri1/cloudapp-giraph-tutorial.git
```

Step 2: Go inside the folder in order to find out how the connected components algorithm is developed in Giraph (you can also find the code in **Appendix A**):

```
# cd cloudapp-giraph-tutorial/
# nano src/ConnectedComponentsComputation.java
```

```
public void compute(
    Vertex<IntWritable, IntWritable, NullWritable> vertex,
    Iterable<IntWritable> messages) throws IOException {
  int currentComponent = vertex.getValue().get();
  // First superstep is special, because we can simply look at the neighbors
  if (getSuperstep() == 0) {
    for (Edge<IntWritable, NullWritable> edge : vertex.getEdges()) {
      int neighbor = edge.getTargetVertexId().get();
      if (neighbor < currentComponent) {
        currentComponent = neighbor;
      }
    }
    // Only need to send value if it is not the own id
    if (currentComponent != vertex.getValue().get()) {
      vertex.setValue(new IntWritable(currentComponent));
      for (Edge<IntWritable, NullWritable> edge : vertex.getEdges()) {
        IntWritable neighbor = edge.getTargetVertexId();
        if (neighbor.get() > currentComponent) {
          sendMessage(neighbor, vertex.getValue());
        }
      }
    }
    vertex.voteToHalt();
    return;
  }
  boolean changed = false;
  // did we get a smaller id ?
  for (IntWritable message : messages) {
    int candidateComponent = message.get();
    if (candidateComponent < currentComponent) {
      currentComponent = candidateComponent;
      changed = true;
    }
  }
  // propagate new component id to the neighbors
  if (changed) {
    vertex.setValue(new IntWritable(currentComponent));
    sendMessageToAllEdges(vertex, vertex.getValue());
  }
  vertex.voteToHalt();
}
```

Step 2: Now, build the code. Make sure you are in the "cloudapp-giraph-tutorial" folder:

```
# mvn clean package
```

Now that the code is built, look at the graph data set you are going to use:

Step 3: Look at the data set you are going to use:

```
# nano data/tiny2_graph.txt
```

We are going to use the earlier unweighted graph. The format of input data is as follow: **"source_id destination_id_1 destination_id_2"**:

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

Step 4: Copy the input file to HDFS:

```
# hadoop fs -put data/tiny2_graph.txt /giraph-tutorial/input
```

Step 5: Run the Connected Components application on the dataset:

```
# hadoop jar target/giraph-mp-1.0-SNAPSHOT-jar-with-dependencies.jar
org.apache.giraph.GiraphRunner ConnectedComponentsComputation -vif
org.apache.giraph.io.formats.IntIntNullTextInputFormat -vip /giraph-
tutorial/input/tiny2_graph.txt  -vof
org.apache.giraph.io.formats.IdWithValueTextOutputFormat -op /giraph-
tutorial/output/connected-components -w 1 -ca
giraph.SplitMasterWorker=false
```

While you are waiting, you should have a closer look at the command and understand what each parameter is responsible for:

Parameter name	Description
-vif	The VertexInputFormat for the job
-vip	The path in HDFS from which the VertexInputFormat loads the data
-vof	The OutputFormat for the job

-op	The path in HDFS to which the OutputFormat writes the data
-w	Number of workers
-ca	A custom argument is defined as a key value pair

Note that since we are not running this example on a multiple server cloud cluster, we are disabling the split master worker. In case you are trying the example on a multi server deployment, remove the "-ca giraph.SplitMasterWorker=false" parameter.

This example is adapted from the Giraph package tutorials. To learn more, visit:

http://giraph.apache.org/quick_start.html

Step 6: When the application is finished, the output of the application will be on the HDFS. To see the output, you may use the following command:

```
# hadoop fs -cat /giraph-tutorial/output/connected-components/*
```

As you can see, the application has found three connected components with identifiers 0,5,8:

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

Step 7 (Optional): The output files can be downloaded from HDFS using the following command:

```
# hadoop fs -get /giraph-tutorial/output/connected-components/* .
```

Step 8: You should clean the output folder on HDFS before a new run using the following command:

```
# hadoop fs -rm -r -f /giraph-tutorial/output/connected-components/*
```

5 Building Giraph and more examples (optional)

Now that you have successfully run your first Giraph graph processing application, you have the option of building a full Giraph source code and enjoy running some of the pre-built graph algorithms like “shortest path” and “PageRank”.

Please note that this step is optional and building Giraph from source will take between 10-15 minutes, depending on your VM type.

Step 1: Make sure you are in the “giraph-tutorial” directory. You can go to that directory by using the following command:

```
# cd ~/giraph-tutorial/
```

Step 2: Clone the Giraph project from a Github repository. The following command clones the recent stable version 1.1.0:

```
# git clone -b release-1.1 https://github.com/apache/giraph.git
```

Step 3: Build the Giraph by running the following commands:

```
# cd ~/giraph-tutorial/giraph
# mvn -Phadoop_2 -Dhadoop.version=2.6.0 package -e -DskipTests=true
```

The build procedure may take a while. You may have to execute the build command multiple times. However, you should eventually see output similar to the following:

```
[INFO] ------------------------------------------------------------------------
[INFO] Reactor Summary:
[INFO]
[INFO] Apache Giraph Parent ............................ SUCCESS [01:33 min]
[INFO] Apache Giraph Core .............................. SUCCESS [01:51 min]
[INFO] Apache Giraph Examples .......................... SUCCESS [ 37.779 s]
[INFO] Apache Giraph Accumulo I/O ...................... SUCCESS [01:13 min]
[INFO] Apache Giraph HBase I/O ......................... SUCCESS [01:31 min]
[INFO] Apache Giraph HCatalog I/O ...................... SUCCESS [01:50 min]
[INFO] Apache Giraph Hive I/O .......................... SUCCESS [02:23 min]
[INFO] Apache Giraph Gora I/O .......................... SUCCESS [02:13 min]
[INFO] Apache Giraph Rexster I/O ....................... SUCCESS [  1.337 s]
[INFO] Apache Giraph Rexster Kibble .................... SUCCESS [  7.469 s]
[INFO] Apache Giraph Rexster I/O Formats ............... SUCCESS [ 49.688 s]
[INFO] Apache Giraph Distribution ...................... SUCCESS [01:21 min]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
```

```
[INFO] Total time: 15:37 min
[INFO] Finished at: 2015-09-14T21:28:23+00:00
[INFO] Final Memory: 89M/315M
[INFO] ------------------------------------------------------------------------
[root@sandbox ~]#
```

At the end, you will have the distributable Giraph JAR file located at:

```
~/giraph-tutorial/giraph/giraph-core/target/giraph-1.1.0-for-hadoop-2.6.0-jar-with-dependencies.jar
```

You will also have the Giraph examples JAR file, which includes some of most used graph processing algorithms, which is located at:

```
~/giraph-tutorial/giraph/giraph-examples/target/giraph-examples-1.1.0-for-hadoop-2.6.0-jar-with-dependencies.jar
```

Step 4: For convenience, you can create a symbolic link to the JAR files. To do so, you must first go back to the tutorial directory:

```
# cd ~/giraph-tutorial/
```

Next, using the linux "ln" command you can create a symbolic link for the Giraph JAR file:

```
# ln -s ~/giraph-tutorial/giraph/giraph-core/target/giraph-1.1.0-for-hadoop-2.6.0-jar-with-dependencies.jar giraph-core.jar
```

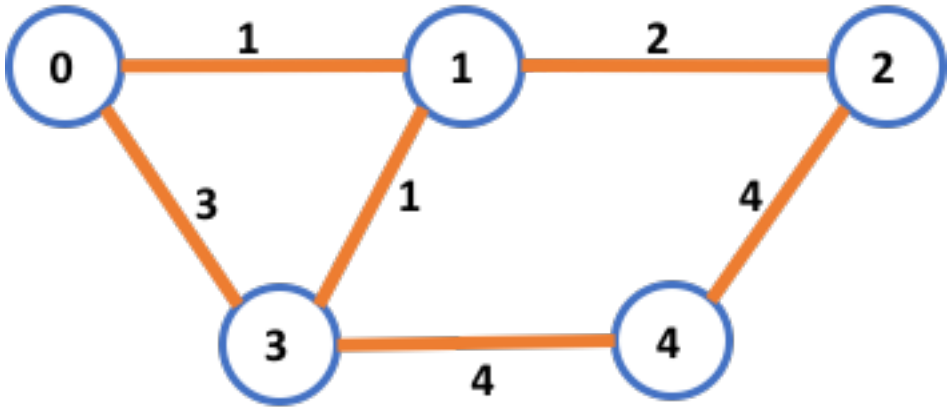
You do the same for the Giraph examples JAR file:

```
# ln -s ~/giraph-tutorial/giraph/giraph-examples/target/giraph-examples-1.1.0-for-hadoop-2.6.0-jar-with-dependencies.jar giraph-examples.jar
```

6 Prepare sample data and HDFS

Giraph needs all the input and output files to be located on HDFS. In this section, you will first prepare some example data sets. Then, you create the necessary folders on HDFS. Lastly, you copy the dataset to the HDFS.

For the input graph dataset, you create a small test network, which represents the following graph:



This graph has 5 vertices and 6 edges. The Id of vertices ranges from 0 to 4, and each edge has a weight.

One way to represent this graph in a file is to create a text file and enter the following information in each line:

```
[source_id, source_value,[[dest_id], [edge_value],...]]
```

This representation is commonly called the **adjacency list**, where each line represents a relationship between a **source_id** object and one or more **dest_id** objects. The **edge_value** (or weight) can be thought of as the strength or distance of the connection between the **source_id** and the **dest_id**.

Step 1: Make sure you are in the tutorial directory you have created:

```
# cd ~/giraph-tutorial/
```

Step 2: Create a new text file for storing the graph data set by using the following command:

```
# nano tiny_graph.txt
```

Step 3: Type (or copy/paste) the following text in the editor; then quit **nano,** and save the file:

```
[0,0,[[1,1],[3,3]]]
[1,0,[[0,1],[2,2],[3,1]]]
[2,0,[[1,2],[4,4]]]
[3,0,[[0,3],[1,1],[4,4]]]
```

```
[4,0,[[3,4],[2,4]]]
```

Step 4: Copy the manually created input file to HDFS:

```
# hadoop fs -put tiny_graph.txt /giraph-tutorial/input
```

7 Shortest path

The shortest path problem is one of the old problems in graph theory. The goal is to find a path between two vertices in a graph such that the number of edges (or if edges are weighted, sum of the weights) is minimized. In this particular example, we will be finding the shortest paths in our small graph data set between each vertex.

Step 1: Make sure you are in the "giraph-tutorial" directory. You can go to that directory by using the following command:

```
# cd ~/giraph-tutorial/
```

Step 2: Using the following command, run Giraph with the already implemented shortest path algorithm on the "tiny_graph" dataset and wait for it to be done:

```
# hadoop jar giraph-examples.jar org.apache.giraph.GiraphRunner
org.apache.giraph.examples.SimpleShortestPathsComputation -vif
org.apache.giraph.io.formats.JsonLongDoubleFloatDoubleVertexInputFormat
-vip /giraph-tutorial/input/tiny_graph.txt  -vof
org.apache.giraph.io.formats.IdWithValueTextOutputFormat -op /giraph-
tutorial/output/shortestpaths -w 1 -ca giraph.SplitMasterWorker=false
```

While you are waiting, take a closer look at the command and understand what each parameter is responsible for:

Parameter name	Description
-vif	The VertexInputFormat for the job
-vip	The path in HDFS from which the VertexInputFormat loads the data
-vof	The OutputFormat for the job
-op	The path in HDFS to which the OutputFormat writes the data
-w	Number of workers
-ca	A custom argument is defined as a key value pair

Note that since you are not running this example on a multiple server cloud cluster, you are disabling the split master worker. In case you are trying the example on a multi server deployment, remove the "-ca giraph.SplitMasterWorker=false" parameter.

This example is adapted from the Giraph package tutorials. To learn more, visit:

http://giraph.apache.org/quick_start.html

Step 3: When the application is finished, the output of the application will be on the HDFS. To see the output, you may use the following command:

```
# hadoop fs -cat /giraph-tutorial/output/shortestpaths/part*
```

Step 4 (Optional): The output files can be downloaded from HDFS using the following command:

```
# hadoop fs -get /giraph-tutorial/output/shortestpaths/part* .
```

Step 5: You should clean the output folder on HDFS before a new run using following command:

```
# hadoop fs -rm -r -f /giraph-tutorial/output/*
```

8 Page Rank

Now you will try another algorithm, this time PageRank. PageRank assigns a weight to each node of a graph; the weight represents the node's relative importance inside the graph. PageRank usually refers to a set of webpages and tries to measure which ones are the most important in comparison with the rest of the set. The importance of a webpage is measured by the number of incoming links, i.e. references, it receives from other webpages. You are going to benefit from the "giraph-examples.jar".

Step 1: Make sure you are in the "giraph-tutorial" directory. You can go to that directory by using the following command:

```
# cd ~/giraph-tutorial/
```

Step 2: Using the following command, run Giraph with the already implemented PageRank algorithm on the "tiny_graph" dataset and wait for it to be done:

```
# hadoop jar giraph-examples.jar org.apache.giraph.GiraphRunner
org.apache.giraph.examples.SimplePageRankComputation -vif
org.apache.giraph.io.formats.JsonLongDoubleFloatDoubleVertexInputFormat -
vip /giraph-tutorial/input/tiny_graph.txt  -vof
```

```
org.apache.giraph.io.formats.IdWithValueTextOutputFormat -op /giraph-
tutorial/output/pagerank -w 1 -ca giraph.SplitMasterWorker=false -mc
org.apache.giraph.examples.SimplePageRankComputation\$SimplePageRankMaster
Compute
```

Step 3: When the application is finished, the output of the application will be on the HDFS. To see the output, you should use the following command:

```
# hadoop fs -cat /giraph-tutorial/output/pagerank/part*
```

Step 4 (Optional): The output files can be downloaded from HDFS using the following command:

```
# hadoop fs -get /giraph-tutorial/output/pagerank/part* .
```

Step 5: You should clean the output folder on HDFS before a new run using following command:

```
# hadoop fs -rm -r -f /giraph-tutorial/output/*
```

Appendix A: “ConnectedComponentsComputations.java” Source Code

```
import org.apache.giraph.graph.BasicComputation;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.graph.Vertex;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.NullWritable;

import java.io.IOException;

/**
 * Implementation of the connected component algorithm that identifies
 * connected components and assigns each vertex its "component
 * identifier" (the smallest vertex id in the component).
 */
public class ConnectedComponentsComputation extends
    BasicComputation<IntWritable, IntWritable, NullWritable, IntWritable> {
  /**
   * Propagates the smallest vertex id to all neighbors. Will always choose to
   * halt and only reactivate if a smaller id has been sent to it.
   *
   * @param vertex Vertex
   * @param messages Iterator of messages from the previous superstep.
   * @throws IOException
   */
  @Override
  public void compute(
      Vertex<IntWritable, IntWritable, NullWritable> vertex,
      Iterable<IntWritable> messages) throws IOException {
    int currentComponent = vertex.getValue().get();

    // First superstep is special, because we can simply look at the neighbors
    if (getSuperstep() == 0) {
      for (Edge<IntWritable, NullWritable> edge : vertex.getEdges()) {
        int neighbor = edge.getTargetVertexId().get();
        if (neighbor < currentComponent) {
          currentComponent = neighbor;
        }
      }
      // Only need to send value if it is not the own id
      if (currentComponent != vertex.getValue().get()) {
        vertex.setValue(new IntWritable(currentComponent));
        for (Edge<IntWritable, NullWritable> edge : vertex.getEdges()) {
```

```
        IntWritable neighbor = edge.getTargetVertexId();
        if (neighbor.get() > currentComponent) {
          sendMessage(neighbor, vertex.getValue());
        }
      }
    }

    vertex.voteToHalt();
    return;
  }

  boolean changed = false;
  // did we get a smaller id ?
  for (IntWritable message : messages) {
    int candidateComponent = message.get();
    if (candidateComponent < currentComponent) {
      currentComponent = candidateComponent;
      changed = true;
    }
  }

  // propagate new component id to the neighbors
  if (changed) {
    vertex.setValue(new IntWritable(currentComponent));
    sendMessageToAllEdges(vertex, vertex.getValue());
  }
  vertex.voteToHalt();
 }
}
```