Project 3: Apache Giraph

Chalkiopoulos Georgios | $p3352124$

November 16, 2022

An iterative graph processing system built for high scalability

1 Use Apache Giraph SimpleShortestPaths algorithm

The first part of this project requires that we familiarize ourselves with executing Apache Giraph jobs. The first step is to start the *Hadoop Filesystem* and *Map Reduce*, which is done by using the scripts under `/usr/local/hadoop/bin`. This is done as the `hduser`. The output may be found below:

```
hduser@data-science:/usr/local/hadoop/bin$ bash start-dfs.sh
starting namenode, logging to /usr/local/hadoop/bin/../logs/hadoop-
hduser-namenode-data-science.out

localhost: starting datanode, logging to /usr/local/hadoop/bin/../
logs/hadoop-hduser-datanode-data-science.out

localhost: starting secondarynamenode, logging to /usr/local/hadoop
/bin/../logs/hadoop-hduser-secondarynamenode-data-science.out

hduser@data-science:/usr/local/hadoop/bin$ bash start-mapred.sh
starting jobtracker, logging to /usr/local/hadoop/bin/../logs/hadoop-
hduser-jobtracker-data-science.out

localhost: starting tasktracker, logging to /usr/local/hadoop/bin/../
logs/hadoop-hduser-tasktracker-data-science.out
```

We can verify that the service has started by running the jps command:

```
hduser@data-science:/usr/local/hadoop/bin$ jps
3138 NameNode
3619 TaskTracker
3512 JobTracker
```

```
3387 SecondaryNameNode
3660 Jps
3277 DataNode
```

Moreover, the web interface of both the filesystem and the map reduce administration is available under:

- Filesystem: `http://localhost:50070`
- Map Reduce Administration: `http://localhost:50030`

A screenshot of bath Filesystem and Map Reduce Administration is presented below:

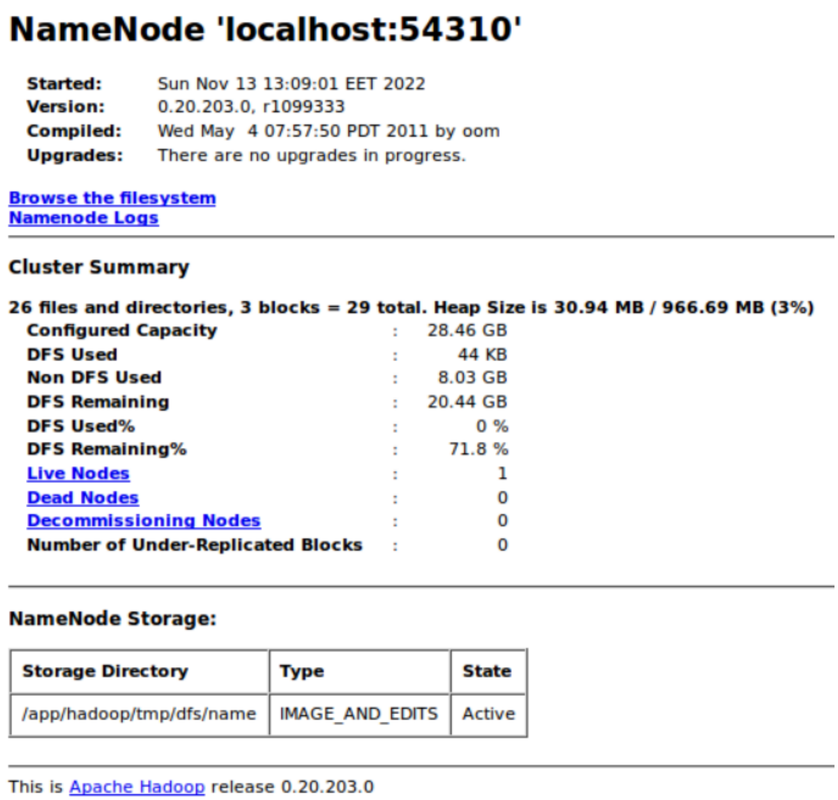


Figure 1: FileSystem Web Interface.

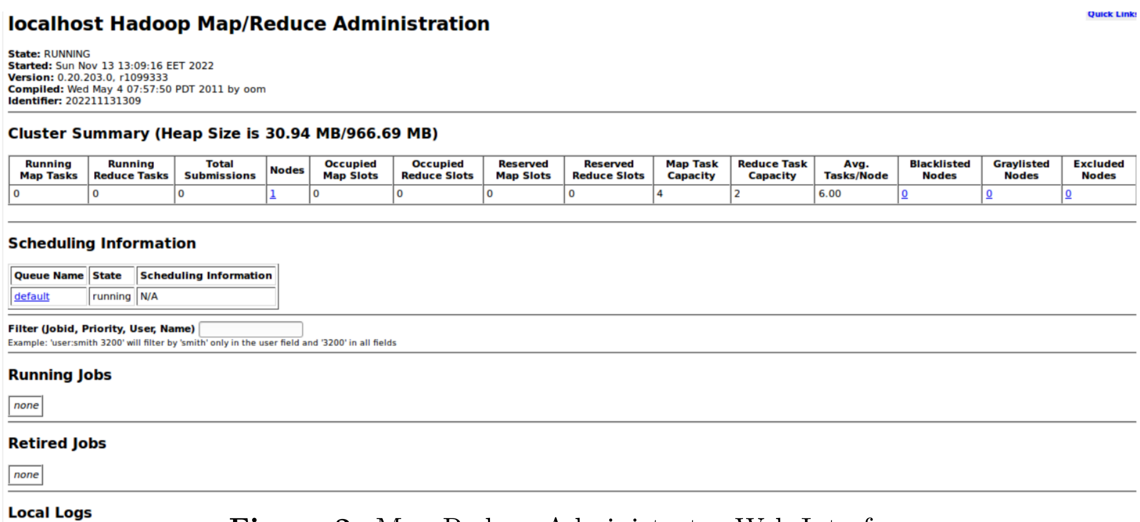


Figure 2: Map Reduce Administrator Web Interface.

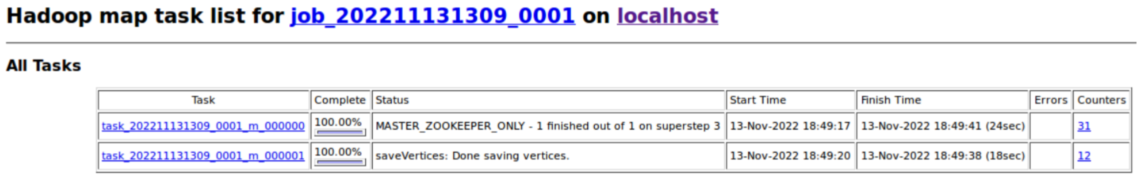
After the service started we executed the following command (text wrapped to fit page):

```
/usr/local/hadoop/bin/hadoop jar /home/sna/eclipse/workspace/giraph/
giraph-examples/target/giraph-examples-1.1.0-for-hadoop-1.2.1-jar-
with-dependencies.jar
org.apache.giraph.GiraphRunner
org.apache.giraph.examples.SimpleShortestPathsComputation
-vif org.apache.giraph.io.formats.JsonLongDoubleFloatDoubleVertex
InputFormat
-vip /user/hduser/input/tiny_graph.txt
-vof org.apache.giraph.io.formats.IdWithValueTextOutputFormat
-op /user/hduser/output/shortestpaths -w 1
```

As indicated by the `-op` cli argument, the output path is located under `/user/hduser/output/shortestpaths`. The output of the file is attached in the report and also presented below.

```
0 1.0
1 0.0
2 2.0
3 1.0
4 5.0
```

The corresponding file is named `task_202211131309_0001_m_000001.txt` which corresponds to the save vertices job, as shown in Figure 3



Hadoop map task list for job_202211131309_0001 on localhost

All Tasks

Task	Complete	Status	Start Time	Finish Time	Errors	Counters
task_202211131309_0001_m_000000	100.00%	MASTER_ZOOKEEPER_ONLY - 1 finished out of 1 on superstep 3	13-Nov-2022 18:49:17	13-Nov-2022 18:49:41 (24sec)		31
task_202211131309_0001_m_000001	100.00%	saveVertices: Done saving vertices.	13-Nov-2022 18:49:20	13-Nov-2022 18:49:38 (18sec)		12

Figure 3: Map Reduce Administration: Save Vertices task id.

2 Implement Label Propagation Community Detection with Apache Giraph

The Label Propagation algorithm for community detection [1] builds on the following idea: Suppose that a node x has neighbors $x_1; x_2; \dots; x_k$ and that each neighbor carries a label denoting the community to which they belong to. Then x determines its community based on the labels of its neighbors.

For the second part of this homework, we will have to implement an iterative Pregel-like algorithm that assigns to all nodes of the graph a community id, according to the previous algorithm. We decided to proceed with the Eclipse development, running the code directly through the eclipse IDE.

To support code execution, several changes had to be implemented in the *GiraphAppRunner.java* file, as well as the *SimpleLabelPropagationComputation.java* file.

All changes implemented are provided in the zip submission file. The application uses the `input_graph.txt` file which contains the graph illustrated in Figure 4

Regarding the `GiraphAppRunner.java` file, we added a command which deletes the current output directory.

```
    // delete
    FileUtils.deleteDirectory(new File("labelPropagation"));
```

The main changes were done in the *SimpleLabelPropagationComputation.java* file, which is where the computations on the nodes are performed. The main changes are highlighted below, following the algorithm provided. In order to better follow the development life cycle, we will node the code, according to each functionality. Please keep in mind that, for the sake of simplicity, some parts of the code,for example loggers, are not presented in the report code blocks. The entire code is submitted in the zip file

For this specific implementation the number of Max Supersters was set to 60 since

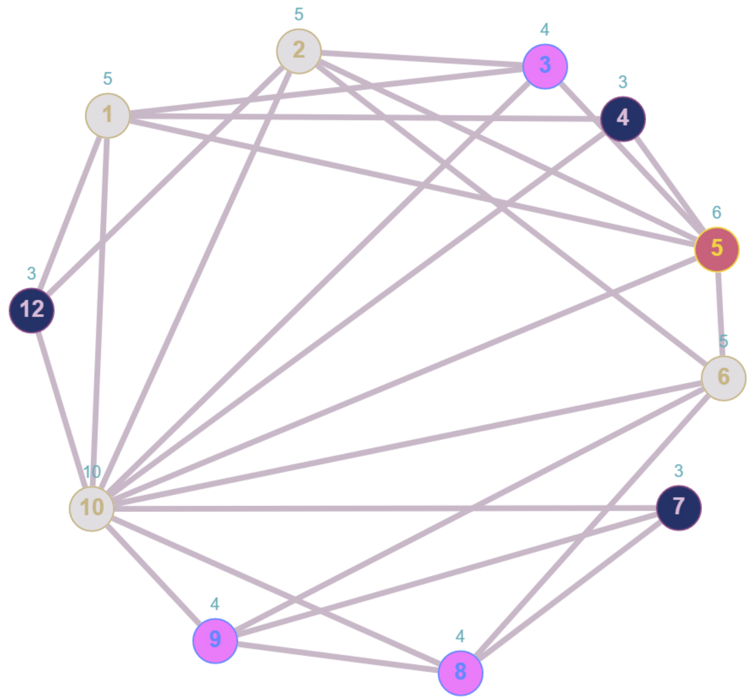


Figure 4: Graph generated using input$_graph.txtfile(node12isactually0)$.

this proved to have a nice balance, which will be explained later on. .

```
    /** Number of supersteps for this test */
    public static final int MAX_SUPERSTEPS = 30;
```

- Every vertex should send an initial message to all its neighbors with the label of its community. Initially, every vertex forms a community on its own, so the vertex id can be used as the community label.

The received messages are turned into an ArrayList named `results`:

```
// create list from messages
List<Long> result = new ArrayList<>();
for (LongWritable message : messages) {
    result.add(message.get());
  }
```

- While a vertex receives messages, it adopts the community of the majority of its neighbors. If there is a tie, the vertex adopts the smallest value.

```java
// Create frequency map
Long current_min  = (long) 999999;
Integer max_freq = 0;
Set<Long> distinct = new HashSet<>(result);

for (Long s: distinct) {
    Integer freq = Collections.frequency(result, s);

    // If the number of neihgbors having a label is bigger than
       the current one
    // assign the max_freq and current min to the one found
    if (freq > max_freq) {
        max_freq = freq;
        current_min = s;
    }

    if ((freq == max_freq) && (s < current_min)) {
        current_min = s;
    }
}
```

- If there is only one neighbor, the vertex adopts its value if it's smaller than its current value.

```java
if (result.size() == 1) {

    if (result.get(0) < vertex.getValue().get()){
        vertex.setValue(new LongWritable(result.get(0)));
        sendMessageToAllEdges(vertex, vertex.getId());
    }
}
```

- If a vertex adopts a new value, it will propagate the new value to its neighbors.

```java
// if vertex value changes propagate
if (!vertex.getValue().equals(new LongWritable(current_min))) {
    vertex.setValue(new LongWritable(current_min));
    sendMessageToAllEdges(vertex, vertex.getValue());
    }
```

- The computation will terminate if no new values are assigned or the maximum number of iterations is reached. You can consider any number between 30 and 80 as the number of maximum iterations.

```
    // if superstep is not the first and less than the
       MAX_SUPERSTEPS proceed
    else if ((getSuperstep() != 0)  && (getSuperstep() <
       MAX_SUPERSTEPS)) {
```

The execution of the algorithm was successful, and full details can the found in the corresponding log file named `log_input_graph.log`. What we observed is that as the values were being propagated into the network, the lower values were dominating, which is expected as mentioned in the corresponding paper. The output of the algorithm oscillated between 1 and 0 (for most of the nodes) for many supersteps. Stopping at superstep 59 (the 60$^{\text{th}}$ superstep) the output is:

```
0   1
1   0
2   0
3   1
4   1
5   0
6   1
7   0
8   0
9   0
10  0
```

As mentioned previously the logs provide a detailed calculation of each node's value at every superstep.

3 Bonus Question: Illustrate the output of your algorithm using Gephi, given a graph of your choice of more than 100 vertices as input.

For this part of the project, the following steps had to be followed:

- Create a graph of 100 vertices using Snap and save them in a file following the `input_graph.txt` format
- Use *GiraphAppRunner* to run *SimpleLabelPropagationComputation*. Make any modifications needed
- Create a `.gdf` file that can be imported to Gephi
- Visualize the results

The above points are presented step by step below. First of all, using snap we generated a graph using the snap.TUNGraph Class of 100 nodes. The nodes were written into a file named `huge_graph.txt` with the correct format.

```python
"""
File: SNA_generate.py
Code used to generate huge_graph.txt
"""
import snap

f = open("huge_graph.txt", "w")
UGraph1 = snap.GenRndGnm(snap.TUNGraph, 100, 350)
for NI in UGraph1.Nodes():
    f.write(str(NI.GetId()))
    f.write(" ")
    out = NI.GetOutDeg()
    for i in range(out):
        f.write(str(NI.GetNbrNId(i)))
        f.write(" ")
    f.write("\n")
```

The first lines of `huge_graph.txt` are also presented below:

```
0 10 17 20 26 44 56 58 89
1 24 29 51 53 56 71 79 97 98
2 63
3 4 24 30 31 32 40 52 57 69 70 73 76 80 87 95 98
4 3 5 11 48 60 77 81 94
```

The file was used to run the *SimpleLabelPropagationComputation* using *GiraphAppRunner*. Some minor changes were made in the `GiraphAppRunner.java` as shown below:

```
// delete
FileUtils.deleteDirectory(new File("labelPropagation_huge"));

setInputPath("huge_graph.txt");
setOutputPath("labelPropagation_huge");
```

The log and output file for this run is named `labelPropagation_huge.log` and the result `part-m-00000.txt`. Another python script was used to generate the `huge_graph.gdf` file.

```
"""
File: SNA_generate.py
Code used to generate huge_graph.gdf
"""
f = open("huge_graph.gdf", "w")
f.write("nodedef>name VARCHAR, community INT\n")

with open("part-m-00000.txt", "r") as fr:
    for line in fr.readlines():
        f.write(",".join(line.split()))
        f.write("\n")

f.write("edgedef>node1 VARCHAR,node2 VARCHAR\n")
with open("huge_graph.txt", "r") as fr:
    for line in fr.readlines():
        root = line.split()[0]
        for node in line.split()[1:]:
            f.write(f"{root},{node}")
            f.write("\n")
```

Finally the file was imported to Gephi. Figure 5 represents the graph with the nodes painted according to the community they below. Only two communities existed at the end of the algorithm, with values 0 and 3.

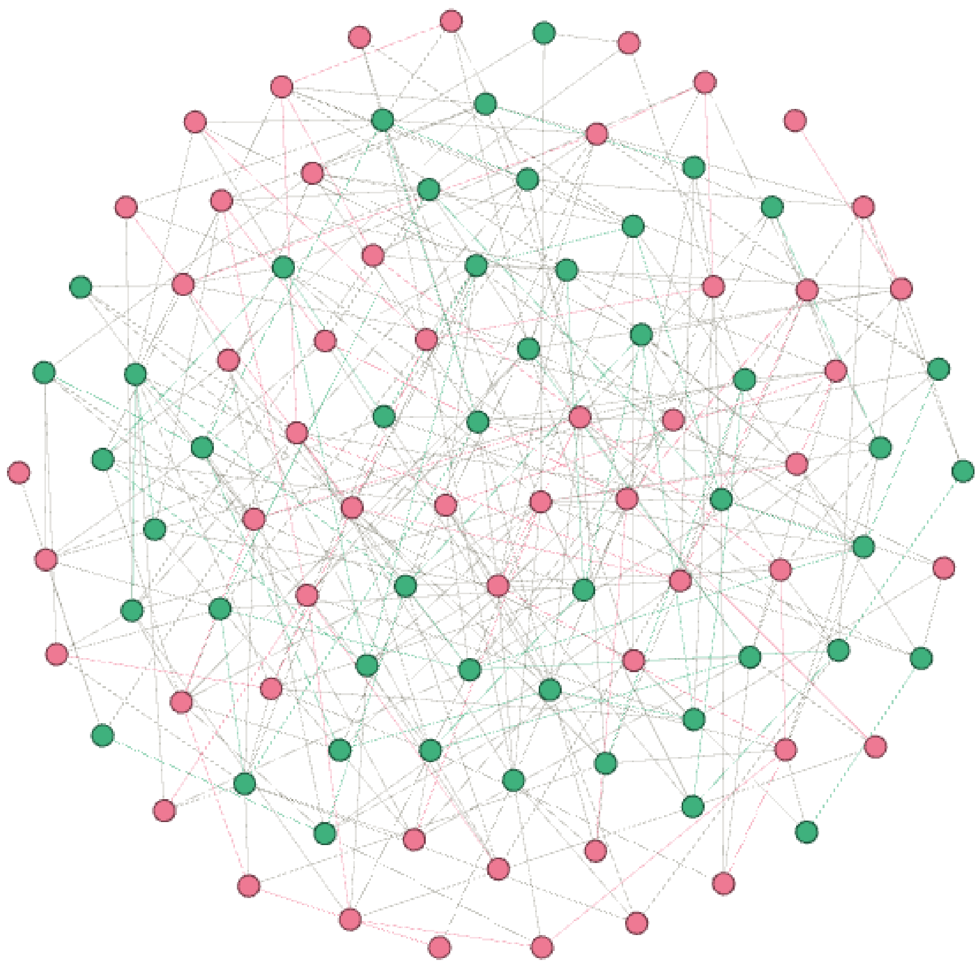


Figure 5: Graph generated using `huge_graph.gdf` (100 nodes). Colors correspond to the community assigned.