Università degli studi di Pisa
Scuola Superiore Sant'Anna

DEPARTMENT OF COMPUTER SCIENCE
DEPARTMENT OF INFORMATION ENGINEERING
AND SCUOLA SUPERIORE SANT'ANNA

Academic Year 2015/2016

PAD final project report

Connected Components algorithm using Hadoop MapReduce framework

Federico Conte
(451530)

1 Introduction

1.1 Description

Given a graph, this algorithm identifies the connected components using Hadoop MapReduce framework. You can find the source code on github.

In the following, a **connected component** will be called as *"Cluster"*, i.e. a set of nodes that compound the connected component.

We have tried to implement the *"The Alternating Algorithm"* proposed in the paper Connected Components in MapReduce and Beyond.
Below, it is reported the pseudo-code of the algorithm.

Algorithm 1 The Alternating Algorithm

INPUT: Edges (u, v) as a set of key-value pairs $\langle$u; v$\rangle$.
INPUT: A unique label ℓ_v for every node v $\in$ V.

```
repeat
    Large-Star
    Small-Star
until Convergence
```

Algorithm 2 The Large-Star operation

```
procedure MAP( u; v )
    Emit ⟨u; v⟩
    Emit ⟨v; u⟩
end procedure
procedure REDUCE( u; Γ(u) )
    m ← arg min_{v∈Γ+(u)} ℓ_v
    Emit ⟨v; m⟩ ∀v where ℓ_v > ℓ_u
end procedure
```

Algorithm 3 The Small-Star operation

```
procedure MAP( u; v )
    if ℓ_v ≤ ℓ_u then
        Emit ⟨u; v⟩
    else
        Emit ⟨v; u⟩
    end if
end procedure
procedure REDUCE( u; N ⊆ Γ(u) )
    m ← arg min_{v∈N∪{u}} ℓ_v
    Emit ⟨v; m⟩ ∀v ∈ N
end procedure
```

1.2 Demonstration

In the Listing 1, it is shown a demonstration of usage of the **ConnectedComponents** class that allows you to run the connected component algorithm on your graph.
This code simply creates a **ConnectedComponents** object, taking as input the graph and the output folder. Invoking the **run** method, the algorithm will produce an hdfs file for each

Reducer task, invoked by MapReduce framework, into the output folder. These files will contain the clusters found in the input graph.

Listing 1: *Usage example of the* ***ConnectedComponents*** *class.*

```
package app;

import pad.ConnectedComponents;

public class App
{
   public static void main( String[] args ) throws Exception
   {
     // Create ConnectedComponents object giving an adjacency/cluster
     // list rappresenting the graph and the output folder as input.
       ConnectedComponents cc = new ConnectedComponents( new
          Path("graph.txt"), new Path("out") );

       // Run all the Jobs necessary to compute the result
       if ( !cc.run() )
          System.exit( 1 );

       System.exit( 0 );
   }
}
```

In this toy application, we don't work on the computed result.
But you can invoke the **TranslatorDriver** as following:

```
HADOOP_HOME=/home/$USER/hadoop-1.2.1
WORKING_DIR=/home/$USER/Exercises-PAD/connectedComponents
JAR_PATH=$WORKING_DIR/target/connectedComponents-1.0-SNAPSHOT.jar
HADOOP=HADOOP_HOME/bin/hadoop
$HADOOP jar $JAR_PATH pad.TranslatorDriver Cluster2Text out outT
```

where :

$HADOOP_HOME: indicates the path to the root directory of hadoop;

$WORKING_DIR: indicates the path to the directory where you have cloned the project;

$JAR_PATH: indicates where it is located the jar file.

This makes it possible to translate the result files in text and look which nodes compound the connected components.

Otherwise, of course, you can code a MapReduce Job to perform the operation that you are looking for taking as input the *"out"* folder.

1.3 Compile

After you have cloned the project, to compile the program you'll need to use the following command lines:

```
cd ConnectedComponents
mvn package
```

1.4 Usage

To run the program, you'll need to use the following command line:

```
$HADOOP jar target/connectedComponents-1.0-SNAPSHOT.jar app.App
```

1.5 Input

In the data folder of the github repository, you can find some graph examples that you can use to try this software. In that folder, there are a lot of files. You have to look up only to the one named as *input_${number}.txt*. The input file can contain:

- the **adjacency list** of the graph, i.e. multiple lines in the following format:
  <NodeID><TAB><Neighbourhood>, where:

  NodeID → is a unique integer ID corresponding to an unique node of the graph.

  Neighbourhood → is a comma separated list of increasing unique IDs corresponding to the nodes of the graph that are linked to <NodeID>.

- the **clique list** of the graph, i.e. multiple lines in the following format:
  <NodeID1><SPACE><NodeID2> ... <SPACE><NodeIDN>
  indicating that al the nodes, <NodeID1> ... <NodeIDN>, are linked to each other.

1.6 Output

The program will create a folder where the clusters found are stored in a **star list** format.
In particular, the output files produced by the Reducer tasks are formatted by the **SequenceFileOutputFormat<pad.ClusterWritable, org.apache.hadoop.io.NullWritable>**, where **pad.ClusterWritable** represents an array of integers and is serialized writing on the output file its size and then its elements. Each output file has the following format:

Cluster_1
Cluster_2
...
Cluster_K

where each *Cluster* is a sequence of increasing numbers which they compound a **connected component** of the input graph.

2 Design Choices

When we are talking about huge graph, usually they are represented with an **adjacency list** or a **clique list** (see section 1.5 for more details). In fact, both of these format can be represented in a more compacted way than the **sparse matrix** format.

Since Algorithm 1 expects as input a set of key-values pairs $\langle u; v \rangle$, for what we have said above, we have decided to introduce an **Initialization_Phase** that transforms the adjacency/cliques list into a edges list <NodeID><TAB><NeighbourID>. In this phase, we also capture the nodes and clique number of the input.

As you can see from Algorithm 4, after the **Initialization_Phase**, takes place the cycle suggested by Algorithm 1 where the Large-Star and Small-Star operations are called sequentially until the graph does not reach a convergent configuration.

When we finally reach this point, we still have the edges list <NodeID><TAB><NeighbourID>. So we need to identify the clusters and print them in a usable format (as it is described in section 1.6). This operation is performed during the **Termination_Phase**. The counting of nodes and clusters numbers is performed again in order to compare these values with the initial ones.

Finally, thanks to the **Check_Phase**, we verify that no clusters is malformed, i.e. every node belonging to a cluster cannot be present in another one, but must be unique.

Here it is shown the algorithm pseudo-code we wrote.

Algorithm 4 Our implementation of the Alternating Algorithm

INPUT: G = (V, E) represented with an adjacency/cluster list.
INPUT: A unique and positive label ℓ_v for every node v $\in$ V.

```
1: Initialization_Phase
2: repeat
3:     Large-Star
4:     Small-Star
5: until Convergence
6: Termination_Phase
7: Check_Phase
```

3 Implementation Details

In order to properly follow the descriptions in this section, it is suggested to give a look to the code while reading.

This section is organized with a sub-section for each component of the algorithm:

3.1 Initialization Phase

In the *InitializationDriver.java*, we analyse the first line of the input file to discover its format.

3.1.1 Adjacency List

Thanks to this format, the **nodes number** is easily calculated counting the input file rows, while the **clique number** is clearly zero. We use *InitializationMapperAdjacent.java* as Mapper and zero Reducer. In the Mapper, we read a line and we split it by the <TAB> character and the second part by the <COMMA> character. Then, for each neighbour, we produce the pair <NodeID, NeighbourID> only if $NodeID > NeighbourID$. In fact, with this input format, an undirected link is identified through the information: <u,v> <v,u>; while, in the following operations, we only need the edge direction from the larger to smaller node.

3.1.2 Clique List

Thanks to this format, the **clique number** is equal to the input file rows, while the **nodes number has to be calculated**. We use *InitializationMapperClique.java* as Mapper in which we read a line and we split it by the <SPACE> character and, then, we produce all the combination between two nodes found in the set and we emit only the pairs <NodeID, NeighbourID> where $NodeID > NeighbourID$ for the reason explained above.
We store this result into a **special folder** while, into the regular folder, the Mapper emits all the encountered nodes. We are doing this, since we do not want to create another Job for the nodes number calculation; instead, **we exploit the same Job** forcing the Mapper to compute two different results:

edges list, stored in a special folder and kept unchanged until the Job end;

encountered nodes list, given to *InitializationReducerNumNodes.java* as input which counts all the distinct nodes found so computing the nodes number of the input file. A combiner (*InitializationCombinerNumNodes.java*) is adopted to locally reduce the duplicated nodes number. After we obtain nodes number value, we move the "real" results stored in the special folder into the regular folder.

In both Mapper described above, if a node has **no neighbours**, it is emitted the pair <NodeID, **-1**>. In this way, when in the following phases we recognize this information, we just bring it forward, until the **Termination Phase** where we emit a cluster formed only by that NodeID.

3.2 Secondary Sort

During the course of the algorithm, we heavily exploit the **secondary sort technique** that allows the programmer to control the order in which the values show up during the Reducer phase. In fact in the Hadoop's implementation of MapReduce this feature is not provided by the framework, unlike the Google one. So we need a "trick" in order to exploit this feature.

First of all, we need a **composite key** that contains the necessary information to perform the sorting and we have to know **how compare** those composite keys during the sorting phase. *NodesPairWritable.java* class is a **writable** and **comparable** data structure used to wrap two nodes into a key:

- **NodeID:** is the natural key used for joining purposes;
- **NeighbourID:** is the value for which we want to sort;

and contains the **compareTo** method that use both nodes during the sorting phase.

Second, we need a **custom partitioner**: *NodePartitioner.java*. Thanks to this component, the composite key will be sent to a Reducer only considering the NodeID, i.e. a given NodeID will be sent to only one Reducer.

Finally, we have to ensure that when the Reducer read the map output records from local disk, those records are combined by NodeID. The *NodeGroupingComparator.java* class specify how to preform this **grouping** process.

The most important usage of this technique is during the **Small-Star** and **Large-Star** operations. In fact, as we can see from Algorithms 2 and 3 respectively at line 6 and 9, we need to obtain the minimum neighbour.
Thanks to the **secondary sort technique**, the Reducer will receive for a given NodeID all its neighbours sorting in ascending order. So, **it is sufficient to compare this NodeID with its first neighbour in order to identify the minimum label**.

Then, we use it during the **Termination Phase**, since we want to store the set of nodes that compose a cluster in ascending order.

3.3 Large-Star & Small-Star

Since the structure of two operations is very similar, we have decided to use the following java class files: *StarDriver.java*, *StarMapper.java*, *StarCombiner.java*, *StarReducer.java* and change their behaviour using an external variable, i.e. the Mapper and Reducer extract this variable during the **setup** and than they act accordingly depending on its value.

Then, we just follow the exactly behaviour described in Algorithms 2 and 3, as you can see from the code we wrote for the Mapper and Reducer.

After a Large-Star or Small-Star operation, it is possible than some links are replicated, as you can see from Figure 1.
For this reason, we added a Combiner phase (*StarCombiner.java*) that is responsible of reducing the amount of data written to local storage, merged, sorted and sent over the network. The Hadoop framework does not ensure how many times a Combiner is called in action, but we know that, if it happens, is after the data sorting:

- after in-memory sort and before writing data to disk;
- after the merging and sorting phases.

Therefore, knowing that the data is sorted, we can exploit the **secondary sort** and delete the duplicates simply storing the last NeighbourId emitted and skipping all the following neighbours with the same identifier.

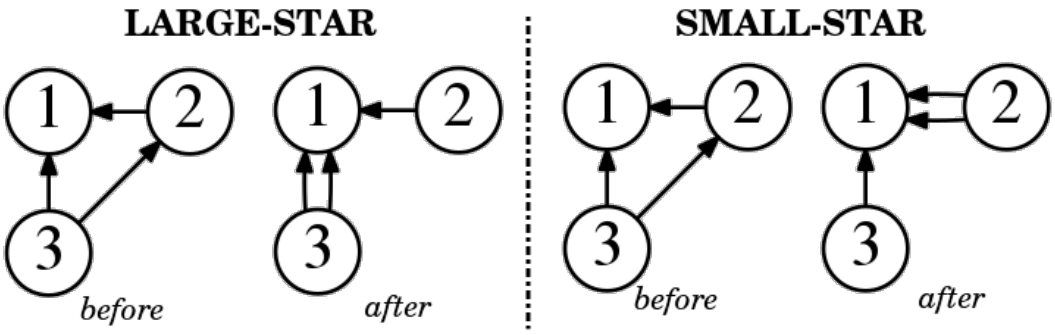

Figure 1: *Example showing the repetition of links formed after a Large/Small-Star operation.*

3.4 Convergence

In order to explain our convergence idea, we would like to use the examples shown in the following figure (Figure 2).
Let us analyse the examples in details.

When we apply the **Large-Star operation** on Node **2** of the first input graph, we have to *"connect all strictly larger neighbours to the minimum neighbour including self"*.
In this case, the minimum neighbour is Node **1**, so we "modify" the blue edges $3 \to 2$, $4 \to 2$ into the red edges $3 \to 1$, $4 \to 1$.

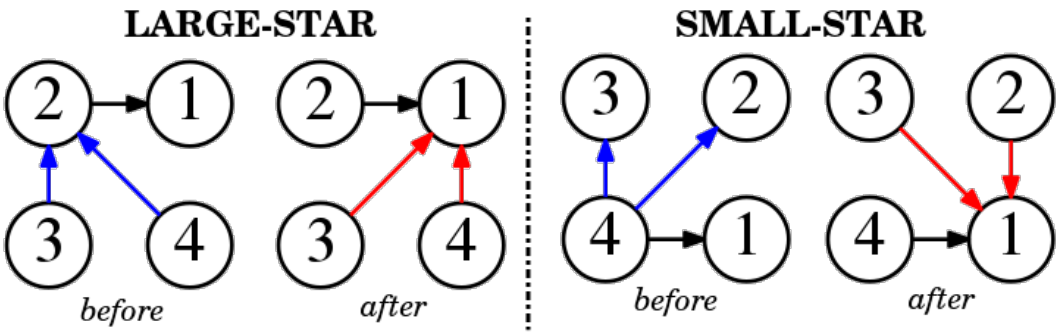

Figure 2: *Example showing the convergence idea. The input graphs are different for the two operations. Arcs that are modified after the operation are highlighted with blue and red colours.*

The number of modified edges can be counted looking if the minimum neighbour coincides with the node that we are analysing. **If it does not coincide, the number of modified edges is equal to the number of emitted edges**.

When we apply the **Large-Star operation** on Node **1**, since Node **1** is the minimum node, we do not have any modification: edge $2 \rightarrow 1$ is not changed.

When we apply the **Small-Star operation** on Node **4** of the second input graph, we have to *"connect all smaller neighbours and self to the minimum neighbour"*.
In this case, the minimum neighbour is Node **1**, so we "modify" the blue edges $4 \rightarrow 3$, $4 \rightarrow 2$ into the red edges $3 \rightarrow 1$, $2 \rightarrow 1$. For the Small-Star operation, we do not add the connection to the smallest node ($4 \rightarrow 1$) in the **modified edges counting**, in fact the nodes are obviously already connected, as you can see in Figure 2.

The **Convergence condition** is reached when:

$$LargeStar.NumChanges + SmallStar.NumChanges = 0 \tag{1}$$

This means that only when both the operations: Large-Star and Small-Star do not modify the graph, then we can conclude that the graph has reached its convergence point.

3.5 Termination Phase

In order to realize this phase, we used three java class files: *TerminationDriver.java*, *TerminationMapper.java*, and *TerminationReducer.java*. The first one is used to configure the **Termination Job** that has the other two class files respectively as Mapper and Reducer. This Job expects a pairs list of integers nodes as input, and produces an output formatted as described in section 1.6.

For each pair <u, v>, this **Mapper** emits the pair <u, v> if $\ell_u < \ell_v$, otherwise <v, u>.
In this way, the **Reducer** will receive a cluster for each key. In fact, if a **connected component** has reached its convergence point, all nodes belonging to this sub-graph are connected to the minimum node.

Therefore, in the **Reducer** phase, we receive the minimum NodeID of a **cluster** and all the nodes belonging to it (sorted in ascending order thanks to the **secondary sorting technique**). What remains to be done is just adding these nodes to a **ClusterWritable** object and emitting it, so producing a **star list**. As we said, during this phase, the counting of nodes and clusters number is performed again in order to compare these values with the initial ones.

3.6 Check Phase

In the **Termination Phase**, we have made sure to not store duplicate nodes within a cluster, exploiting the secondary sorting property.

As we said, a cluster is malformed if one of its node is present in another cluster. So in order to verify if all the resulting clusters are well-formed, it **is sufficient to check that all nodes are unique**. If a node is found twice, for what said above, it is surely present in two different clusters.

As usual, we need a class to configure the Job: *CheckDriver.java*. The Mapper (*CheckMapper.java*) emits all nodes of the clusters. The Reducer (*CheckReducer.java*) increments an **error counter** if receives a given node identifier more that one time.

To be precise, we should check if **all the nodes of input graph are present in the final result**. During the steps of the algorithm, a node is never added. Besides, as we said, we ensure that there are no duplicated nodes.
Therefore, in order to verify the property, it is necessary to check that **the nodes number of the input graph is equal to the one of the final results**.

These two counting are calculated respectively in the **Initialization and Termination Phases**. We have chosen to simply report this two values to the final user that will judge if to keep or not the obtained result. This can be useful if the number of node lost are small enough to be an acceptable error for the final user.

4 Testing Procedures

In order to test this software, we have prepared some verification outputs in the data folder on github, with the purpose to compare these handmade expected outputs with the software outputs.

For example, you can test the following graph:

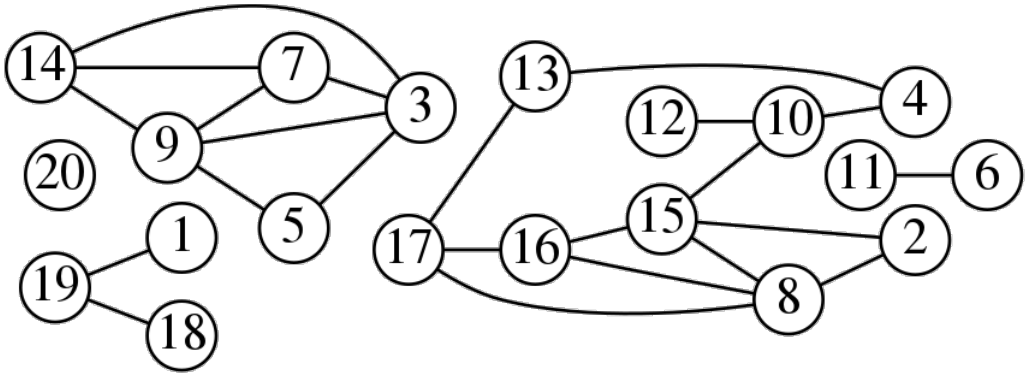

Figure 3: *One of the testing input graph.*

input_1.txt: The adjacency list of the graph shown in Figure 3.

init_1.txt: The expected result for the **Initialization Phase** applied to *input_1.txt*.

large-star_1.txt: The expected result for the **Large-Star** operation applied to *input_1.txt*.

small-star_1.txt: The expected result for the **Small-Star** operation applied to *input_1.txt*.

term_1.txt: It is the handmade input for the **Termination Phase**, in this way we can check the last phase without running all the algorithm.

cluster_1.txt: Contains a line for each **Cluster** found, and the set of nodes that compose each cluster are stored in ascending order (it is a **star list** as explained in section 1.6). It is the final expected result of the **Termination Phase** applied to *term_1.txt* and of the all algorithm as well. In this case, the content of the file is the following:

```
1 18 19
20
2 4 8 10 12 13 15 16 17
3 5 7 9 14
6 11
```

In the bin folder on github, you can find a *bash script* that tests each phase for every appropriate input found in the data folder. Pay attention that for the *StarTest.sh* script, you need to specify the *type* of operation as argument: "small" or "large".

5 Experimental Evaluation

In this section, we want to analyse the results observed by running the algorithm on a large input graph. The input, that we have used, is a **73.7 MB** text file containing **5,869,938** nodes and **three millions** of clique. Unfortunately, we could not use a bigger dataset for lack of storage resource on our machine.

The platform used has the following characteristics:

Hardware: x86 64 architecture with Intel Pentium Dual-Core B960 @ 2.0Ghz, 2MB L3 cache, 4GB of RAM and 500GB ATA Disk Western Digital with only 46.3GB available;

Operating System: Linux 4.2.0-23-generic, Ubuntu 15.10;

Framework: Hadoop 1.2.1 MapReduce.

The Initialization Phase has produced **348,528,515** edges, taking **442** seconds of CPU time.

The loop reached its contingency point after **10** iterations, executing Large-Star or Small-Star operations: Table 1 shows some information on those Jobs.

Type Operation	# Iteration	Launched Map Tasks	Map Input Records	CPU Time Spent (s)	Number of edges changed
Large-Star	0	84	348,528,515	18,024	200,808,008
Small-Star	1	50	204,617,318	4,847	159,810
Large-Star	2	1	3,969,061	177	32,485
Small-Star	3	1	3,830,717	90	139
Large-Star	4	1	3,830,717	170	2,928
Small-Star	5	1	3,830,640	92	2
Large-Star	6	1	3,830,635	174	214
Small-Star	7	1	3,830,634	92	0
Large-Star	8	1	3,830,634	177	0
Small-Star	9	1	3,830,634	92	0

Table 1: *Shows some information on the Large-Star and Small-Star Jobs executed by the Algorithm 4 on the input described above.*

Thanks to the Termination Phase, we have discovered **2,039,304** clusters. Looking to the recalculated nodes number, we can say that no node has been lost. This phase took **83** seconds of CPU time.

Finally, the Check Phase was performed in **26** seconds of CPU time and reports that the clusters obtained are well-formed.

In conclusion, we could see that:

- the number of iteration was comparable with the logarithm of the number of nodes;
- the number of edges never increased as it is demonstrated in the paper, and, actually, it decreases really rapidly, as we can see from Table 1.