UNIVERSITY OF TRENTO - Italy

Information Engineering
and Computer Science Department

Master Degree in Computer Science

Distributed Algorithms

AA 2015-2016

Community Detection using Giraph

Ardino Pierfrancesco, Natale Maria Pia, Tovo Alessia

Abstract

The content of this work is about the implementation of a Community Detection algorithm using a MapReduce framework. This report is created as course project of the Distributed Algorithms course held by prof. Alberto Montresor. A network is said to have *community structure* if the nodes of the network can be grouped into sub-graphs such that these sub-graphs are densely connected internally. In the case of *non-overlapping* communities, each node belongs to a single sub-graph or it is disconnected from the network if it does not belongs to none of them. *Non-overlapping* communities have strong connections internally and weak connections with the other communities. There can be also *overlapping* communities, here a node could belong to more than one community. It's straightforward to understand that *overlapping* communities have stronger connections that the *non-overlapping* ones. Social networks are paradigmatic examples of graphs with communities. The word community itself refers to a social context. People naturally tend to form groups, within their work environment, family, friends. Detecting communities within a graph can be a difficult task. Decide whether a node has to belong to community instead of another one is not trivial. Also, typically the number of communities in a graph is unknown and so it can be difficult to understand if the found communities are correct or not. In this project we decided to implement the community detection algorithm developed in [1].

Contents

1 Introduction

1.1 Goals

The aim of this work is to show a possible implementation of a community detection algorithm using a MapReduce framework. We made use of the Giraph framework in order to develop and test the proposed solution.

1.2 Content

The report is composed by many sections, firstly we will examine the algorithm described in [1]; then we will describe our implementation of the algorithm and finally we will discuss the results of the algorithm in some graphs. In particular:

- In the first chapter we will discuss the similarity metric used and the stages of the algorithm.
- In the second chapter we will firstly introduce Giraph and then discuss our implementation showing the messages exchanged by the nodes and the nodes structure.
- In the third chapter we will analyze the result obtained from the simulations and explain the different results obtained by varying the simulation parameters.

2 Algorithm Overview

The algorithm, which is called *DEPOLD* (DElayed Processing of Large Degree Nodes),is composed of three Phases:

- *Preprocessing*
- *Coreprocessing*
- *Postprocessing*

A new data structure called *THALS* (Two-Hop Adjacency List with Similarity) is created specifically for the purpose of this MapReduce solution. This data structure is used as input and data format in parallel core-processing algorithms. THALS is composed of the following fields:

- Id of the vertex
- Adjacent list with similarity values which includes neighbor vertex Id and the corresponding similarity value between the vertex and his neighbor.
- Two-Hop adjacent list. The neighbors of a neighbor are stored using an hash table.
- label used to transmit information from Mapper to Reduced.

2.1 Preprocessing

In this phase the nodes with a degree larger than a specific threshold θ are found. This is done because nodes with a large degree can cause a bottleneck during the phase of *Coreprocessing* and can confuse the structure of a community. These nodes are disabled and re-enabled in the Postprocessing phase.

2.2 Coreprocessing

In this phase the similarity value between the linked vertices is iteratively computed. The link will be removed if the similarity value is less than a specific threshold γ. When the change of the topology is less than a threshold φ the algorithm has converged.
The similarity between two nodes is computed as following:

$$S(v_\mathrm{i}, v_\mathrm{j}) = \begin{cases} \frac{1+C(v_\mathrm{i},v_\mathrm{j})+\eta(v_\mathrm{i},v_\mathrm{j})}{F(v_\mathrm{i},v_\mathrm{j})} & \text{if } |E(v_\mathrm{i}, v_\mathrm{j})| = 1 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

where $C(v_\mathrm{i}, v_\mathrm{j})$ is the number of common neighbors between v_i and v_j, $\eta(v_\mathrm{i}, v_\mathrm{j})$ is the number of the edges between the common neighbors of v_i and v_j and $F(v_\mathrm{i}, v_\mathrm{j})$ is the sum of the degree of v_i and v_j.

2.3 Postprocessing

In this phase the communities division are found by discovering the *Weakly Connected Components*. Then the high-degree nodes are merged in a community by computing the degree the filtered node contributes to that community and the average contributed degree of the detected community members. It can be that the filtered nodes can belong to more than one communities, in this case the algorithm stores each community to which the node belongs. Finally an hash map containing the Id of the node and the Id of the community to which it belongs is written to the HDFS.

3 Implementation

3.1 Giraph

`Giraph` is a synchronous iterative graph processing framework, originated as the open-source counterpart to `Pregel` and built on top of `Apache Hadoop`. The input to a Giraph computation is a graph composed of vertices and directed edges. Computation proceeds as a sequence of iterations, called *supersteps*. Initially, every vertex is *active*. In each superstep each active vertex invokes the *Compute method* provided by the user. The method implements the graph algorithm that will be executed on the input graph. The Compute method dose the following operations:

- receives messages sent to the vertex in the previous superstep
- computes using the received messages and the vertex value which may result in a modification to the vertex itself
- may send messages to other vertices.

The Compute does not have knowledge of the values of the other vertices. It can retrieve information of the other vertices through Inter-vertex communication by exchanging messages. The computation halts after the vertices have voted to halt and there are no messages in flight.

3.2 Structure of the code

The classes developed for this project are the following:

3.2.1 Node

This class stores information about a node in the Two-Hop adjacent list and in the Similarity map. The class is composed of two fields:

- *Long* ID
- *Boolean* active

The ID field contains the Id of a node while the active field, which is initialized to *true*, contains the status of a node.

3.2.2 Node_Degree

This class stores information about the degree of a node. The class is composed of two fields:

- *Long* ID
- *Long* Degree

The ID field contains the Id of a node while the Degree field, contains the degree of a node. In the case of filtered nodes, the ID field is used to store one of the community of the node.

3.2.3 THALS

This class implements the THALS data structure with some changes. The class is composed of the following fields:

- *Map<Node,Node>* two_hop
- *Boolean* active
- *Long* group_id
- *ArrayList<Node_Degree>* community_members
- *Map<Node, DoubleWritable>* similarity_map
- *ArrayList<Node_Degree>* community_filtered_node

The *two_hop* field contains the two-hop adjacent list of a node, the *active* field contains the status of a node, the *group_id* field contains the community Id of a node, the *community_members* field contains the list of nodes which belong to the same community of the node, the *similarity_map* field is a map containing as key the neighbor of a node and as value the similarity value between the two nodes, the *field community_filtered_node* contains the list of communities to which a filtered node belongs. The two_hop, group_id, community_members, similarity_map fields are used only by the non filtered nodes. We decided to use an ArrayList to store the communities of a filtered node because it can be possible that the node belongs to more than one community.

3.2.4 DepoldInputFormat

This class is used to load the raw data and initialize the vertices and create the edges. Here we specify the type of the Id of the Vertex, *LongWritable* in our case, the type of the data of the Vertex, *THALS* in our case and the type of the data of the Edge, *FloatWritable* in our case.

3.2.5 DepoldOutputFormat

This class is used to write the result of the algorithm to the HDFS, in the case of non filtered nodes it prints the *Id* of the nodes and the *group_id*, while in the case of filtered nodes it prints the *Id* of the nodes and the list of their communities.

3.2.6 DepoldMaster

The *Compute method* of this class is the first thing that gets executed in a superstep, so it useful in combination of *Aggregators* to manage the steps of the algorithm. An *Aggregator* is used to check whether a global condition is satisfied or to keep some statistics. During a superstep, vertices provide values to aggregators. These values get aggregated by the system and the results become available to all vertices in the following superstep. We use five Aggregators to manage the phases of the algorithm, keeping track of how many nodes were filtered, keeping track of how many edges are deleted in the *Coreprocessing* in order to check if the algorithm has converged, check if the *WeaklyConnectedComponents* has converged and check if each node knows each node belonging to its community.

3.2.7 MessageWritable

This class is used to store the information exchanged between vertices. This class contains every kind of message that can be exchanged. It is composed of the following fields:

- *Long* ID

- *LongArrayList* neighbors
- *Boolean* active
- *LongArrayList* deleted_nodes
- *Long* group_id
- *ArrayList<Node_Degree>* community_members

The *ID* field contains the sender of the message, the *neighbors* field contains the neighbors of the sender, the *active* fields contains the status of the sender, the *deleted_nodes* field contains the node deleted by the sender in the *Coreprocessing*, the *group_id* field contains the community of the sender, the *community_members* field contains the list of nodes belonging to the community of the sender.

3.2.8 Depold

This is the most important class and the core of the system. The algorithm has 3 Stage divided into 14 phases. In the following section we will deeply describe each phase of the algorithm.

3.3 Preprocessing

3.3.1 Preprocessing_Two_Hop_First_Phase

In this phase we compute the adjacency list of the node and send it to its neighbors.

3.3.2 Preprocessing_Two_Hop_Second_Phase

Since we expect that the input is an undirected graphs, in this phase we first check if the *Id* of the senders of the messages are already in the neighbors list of the nodes, if they are not present we add them to the list in order to complete the undirected graph. Then we forward the adjacency list received from its neighbors to all its neighbors so each node will know the neighbors of his neighbors.

3.3.3 Preprocessing_Two_Hop_Third_Phase

In this phase we firstly compute the *Two_hop_map* of the node, then we filter the nodes with a degree larger than the threshold θ setting the *active* field to *false* and finally send the status of the node to its neighbors and to its two-hop neighbors.

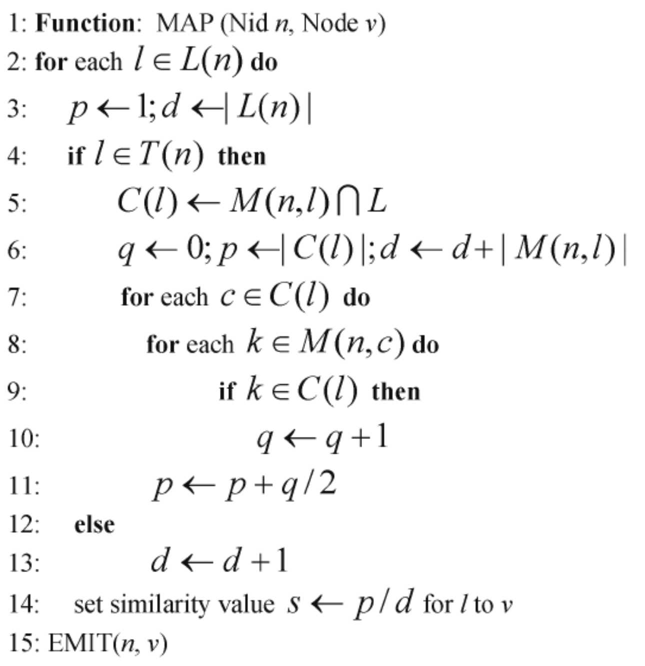
3.3.4 Preprocessing_Second_Phase

In this phase we simply deactivate the edges that connect an unfiltered node to a filtered node by setting the *active* field of the key or of the value of the *Two_hop_map* to *false*.

3.4 Coreprocessing

3.4.1 Coreprocessing_Similarity_Phase

In this phase we compute the similarity between a node and its active neighbors and put the values in the *Similarity_map*. We have noticed that the result of Equation (1) differs from the result of the algorithm used in the *Coreprocessing* of [1] in particular the *CMPI S_Calculator* which is showed below:



```
Function: MAP (Nid n, Node v)
for each $l \in L(n)$ do
    $p \leftarrow 1; d \leftarrow |L(n)|$
    if $l \in T(n)$ then
        $C(l) \leftarrow M(n,l) \cap L$
        $q \leftarrow 0; p \leftarrow |C(l)|; d \leftarrow d + |M(n,l)|$
        for each $c \in C(l)$ do
            for each $k \in M(n,c)$ do
                if $k \in C(l)$ then
                    $q \leftarrow q + 1$
        $p \leftarrow p + q/2$
    else
        $d \leftarrow d + 1$
    set similarity value $s \leftarrow p/d$ for $l$ to $v$
EMIT(n, v)
```

Figure 1: CMPI S_Calculator algorithm

We decided to use this algorithm to compute the similarity with another change. In fact, we have noticed that if v_i and v_j do not have common neighbors $S(v_\mathrm{i}, v_\mathrm{j})$

can differ from $S(v_\text{j}, v_\text{i})$. I.e. if v_i has 3 neighbors and v_j has 2 neighbors but they do not have common neighbors the result using the algorithm would be:

$$S(v_\text{i}, v_\text{j}) = \frac{1}{1+d} = \frac{1}{4} = 0.25 \tag{2}$$

$$S(v_\text{j}, v_\text{i}) = \frac{1}{1+d} = \frac{1}{3} = 0.33 \tag{3}$$

while they should be equal.This can potentially change the graph in an undirected one if $S(v_\text{i}, v_\text{j})$ is under the α threshold and $S(v_\text{j}, v_\text{i})$ is over. We decided to use the maximum between the number of neighbors of the two nodes plus 1. So line 13 of the algorithm becomes

$$d = 1 + man(\pi(i), \pi(j)) \tag{4}$$

where $\pi(i)$ is the number of neighbors of v_i and $\pi(j)$ is the number of neighbors of v_j. This because two nodes that are neighbors but that do not have any common neighbors tend to have a lower similarity.

3.4.2 Coreprocessing_Topology_First_Phase

In the phase we check if the similarity values in the *Similarity_Map* are under the α threshold. If so we delete the edges connecting the nodes, delete the entries in both *Two_Hop_map* and *Similarity_map*, updates the Aggregator that takes track of the deleted nodes and finally send the *Deleted_nodes* list to the neighbors of the nodes.

3.4.3 Coreprocessing_Topology_Second_Phase

In this phase we check if there are nodes in the *Deleted_nodes* list sent by the neighbors of the nodes that have to be deleted in the *Two_Hop_map*. In fact, it can be that some edges representing an entry in the *Two_Hop_map* no longer exists and so we have to delete that entry. If at the end of this phase the number of deleted nodes is over the φ threshold, we return to the beginning of the *Coreprocessing* stage.

3.5 Postprocessing

3.5.1 Postprocessing_WCC_First

In this phase we initialize the algorithm to compute the *Weakly Connected Components*. This algorithm is used to find the communities of the unfiltered nodes.

3.5.2 Postprocessing_WCC_Ssecond

In this phase we compute the WCC and we assign a *group_id* to each unfiltered node. The *Postprocessing_WCC* phase converges if no vertices change community.

3.5.3 Postprocessing_Degree_Calculator_First

In this phase we add to the *Community_members* list all the active neighbors of a node and then we send this list to neighbors of the nodes.

3.5.4 Postprocessing_Degree_Calculator_Second

In this phase we add to the *Community_members* list all nodes received from the neighbors of a node which are not already in the list. Once we add the nodes to the list we send again the list to the neighbors of the nodes. The *Postprocessing_Degree_Calculator* phase converges if no more nodes are added to the *Community_members* list.

3.5.5 Postprocessing_Degree_Calculator_Third

In this phase we send the *Community_members* list to the neighbors which have been filtered in the *Preprocessing* stage.

3.5.6 Postprocessing_Group_Detector

In this final phase the filtered nodes are merged into the existing communities. As said before, a filtered node can belong to more than one community. A filtered node belongs to a community if the number of its neighbors which belong to the community is larger than the average degree of that community. At the end of this stage, each unfiltered node has the *group_id* of the community to which it belongs, while each filtered node has the *community_filtered_node* list of the communities to which it belongs.

4 Simulations and Results

4.1 Assumption

The assumption made for the project are the following:

- the *Id* of a node is an Integer value
- the *group_id* of a node is the smallest *Id* of a **unfiltered** node in that community
- if all node are filtered than all node belong to the same community

4.2 Input graph

The input graph file should be in the following JSON format:
JSONArray(<vertex id>, <vertex value>,JSONArray(JSONArray(<dest vertex id>, <edge value>), ...))
Here is an example with vertex id 1, vertex value 4.3, and two edges. First edge has a destination vertex 2, edge value 2.1. Second edge has a destination vertex 3, edge value 0.7.
$[1, 4.3, [[2, 2.1], [3, 0.7]]]$

4.3 Graph Generation

The graphs used in the simulations have been generated using the *NetworkX* library of *Python* using the *random_partition_graph* function. I.e. If we want to generate a graph with 4 partitions each of 50 elements, a probability of connection between nodes of the same partition equal to 0.5 and a probability of connection between nodes of different partitions equal to 0.01 we can run the following script with these parameters:

python create_graph.py 50 50 50 50 0.5 0.01

which will write in a file the generated graph.

4.4 Running a simulation

To run a simulation is enough to launch the following command:

hadoop jar Depold.jar org.apache.giraph.GiraphRunner depold.Depold -vif depold.DepoldInputFormat -vip <input_path_in_HDFS> -vof depold.DepoldOutputFormat -op <output_path_in_HDFS> -w 4 -mc depold.DepoldMaster -ca Depold.threshold=< θ threshold> -ca Depold.similarity_limit=< γ threshold> -ca Depold.converge=< φ threshold>

Since Giraph accepts only *Long* values as parameter, insert the γ threshold as a *Long*, so 0.3 is equal to 3 and 0.35 is equal to 35.
Make sure that the *giraph-core* jar is in the Hadoop classpath.

4.5 Results

The first set of simulations has been done using the following graph with 15 nodes taken from [1] and showed in Figure 2. Note that after some tests, we have noticed that in [1] the similarity between nodes has been processed using the formula in Equation (1), so our results differ from the ones in the Paper.

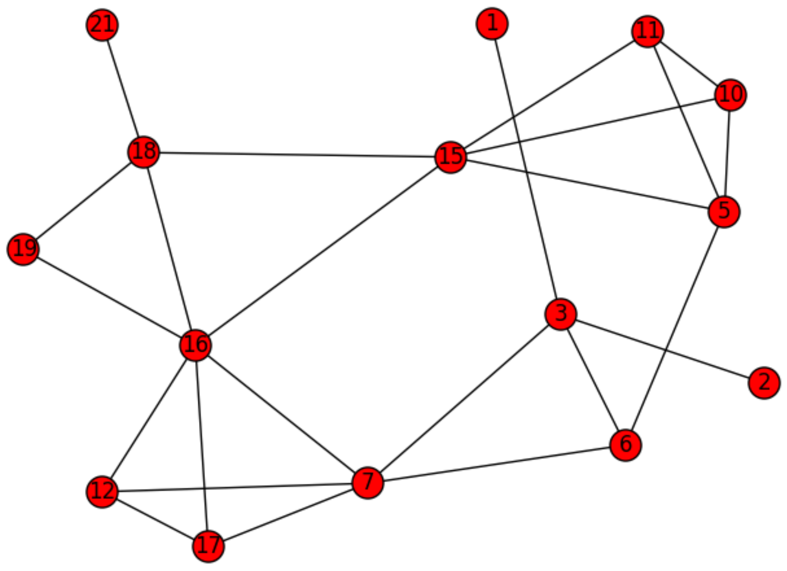


Figure 2: Graph used in the first set of simulations

The parameters used during the simulations are the following:

Simulation number	θ	γ	φ
1	5	0.2	1
2	5	0.3	1
3	4	0.3	1

Table 1: Parameters of the simulations

Simulation number	Duration	Filtered nodes	Communities
1	75s	1	3
2	74s	1	1
3	70s	3	2

Table 2: Results of the simulations with 16 nodes

The results of the simulations are shown in Table 2. As can be seen the first simulation detected three communities $C_1 = (v_1, v_2, v_3)$, $C_{12} = (v_{12}, v_{16}, v_{17})$ and $C_5 = (v_5, v_6, v_{10}, v_{11}, v_{19}, v_{15}, v_{21}, v_{18}, v_{16})$ and finally v_7 does not belong to any community.
The second simulation detected one community $C_5 = (v_5, v_{10}, v_{11}, v_{15})$ while the remaining nodes do not belong to any communities.
As can be noticed, an higher γ helps to detect very strong communities. Finally, the third simulation detected two communities $C_{12} = (v_7, v_{12}, v_{17}, v_{16})$, $C_{18} = (v_{16}, v_{21}, v_{18}, v_{19})$ with the remaining nodes that do not belong to any communities. As can be noticed, lowering the θ threshold helps to detect the community C_{18} which has not been detect in the previous two simulation.

The second set of simulations has been done using a graph generated with the script showed in Section 4.3 with 5 partitions each of 100 nodes, a probability of connection between nodes of the same partition equal to 0.5 and a probability of connection between nodes of different partitions equal to 0.01. The generated graph is showed in Figure 3

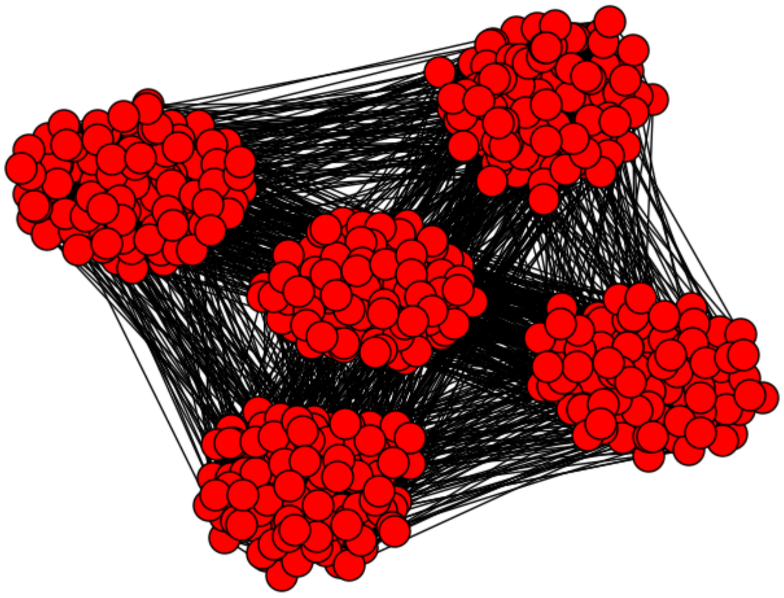


Figure 3: Graph used in the second set of simulations

The parameters used during the simulations are the following:

Simulation number	θ	γ	φ
1	90	0.3	2
2	60	0.3	2
3	50	0.3	2

Table 3: Parameters of the simulations

Simulation number	Duration	Filtered nodes	Communities
1	120s	0	4
2	100s	46	4
3	180s	361	11

Table 4: Results of the simulations with 500 nodes

The results of the simulations are shown in Table 4. The first two simulations detect the exact number of communities, in fact there are 4 communities with 100 nodes each. The second simulation takes less time to complete since a higher number of nodes is filtered and so the algorithm converge in less time. The third simulation detects 11 communities and nearly 50 nodes without a community, this is due the small θ threshold that confuse the communities and, moreover, it takes more time to finish since the algorithm takes more steps and so more time to converge in the *WCC* phase.

The third and final set of simulations has been done using a generated graph with 4 partitions each of 50 nodes, a probability of connection between nodes of the same partition equal to 0.5 and a probability of connection between nodes of different partitions equal to 0.01. The generated graph is showed in Figure 4

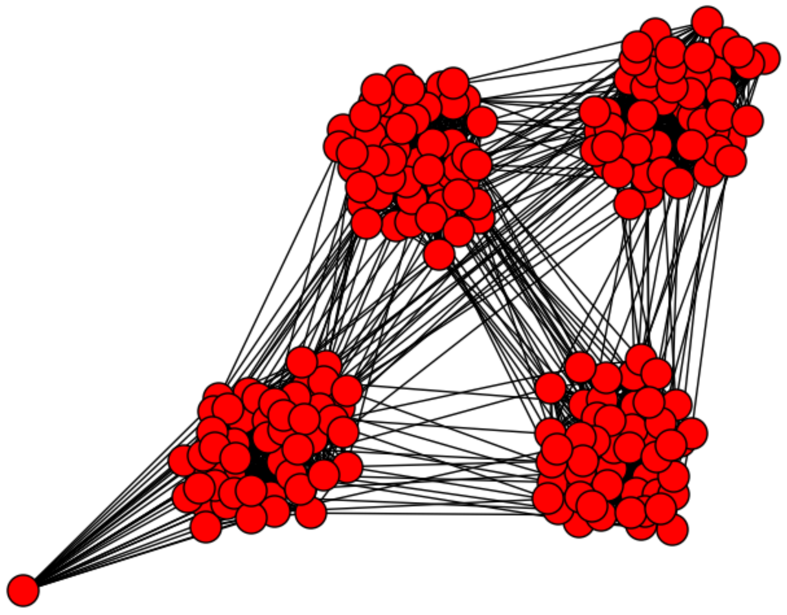


Figure 4: Graph used in the third set of simulations

The parameters used during the simulations are the following:

Simulation number	θ	γ	φ
1	40	0.3	2
2	30	0.3	2
3	20	0.3	2
4	20	0.2	2

Table 5: Parameters of the simulations

Simulation number	Duration	Filtered nodes	Communities
1	90s	0	4
2	95s	16	4
3	76s	187	4
4	75s	187	4

Table 6: Results of the simulations with 200 nodes

As shown in Table 6 each simulation gives the expected number of communities, but only the first two are correct. Indeed, they give as output 4 communities of 50 nodes. As can be seen, even if the second simulation has filtered 16 nodes, it is slower than the first, this can be explained by the fact that the *WCC* phase takes more steps to converge and so more time to complete. The last two simulations give the exact number of communities but they are wrong. Indeed, there is a community with more than 100 nodes, which is not correct. Moreover, in both simulations, there are 7 nodes that do not belong to any communities.

5 Conclusion

During this report we showed that Community Detection is not trivial, find a good similarity metric can be difficult and different metrics even if with small changes can lead to a completely different results.
Find good thresholds for large graphs is crucial. In fact, as we showed, small changes in the thresholds lead to different results in detected communities.
Changing the θ threshold can decrease the simulation time since high degree nodes can act as a bottleneck but it can also confuse the communities and sometimes also increase the simulations time due to the more steps taken by the algorithm.
Changes in the γ threshold help to detect strong communities, high threshold, or weakly communities, low threshold. Finally, a good compromise in the φ threshold is crucial in graphs with a large number of nodes. A small threshold leads to more accurate topology but it can drastically increase the simulation time, while a high threshold leads to an opposite results.

References

[1] J. Shi, W. Xue, W. Wang, Y. Zhang, B. Yang, and J. Li, “Scalable community detection in massive social networks using MapReduce,” *IBM Journal of*

Research and Development, vol. 57, pp. 12:1–12:14, May 2013.