QUEEN MARY, UNIVERSITY OF LONDON

BIG DATA COURSEWORK 2

Social Network Analysis: Community Detection

Alberto Pacheco, Akshata Shirahatti, Zhanelya Subebayeva, Soham Trivedi

1/12/2015

Table of Contents

Table of Figures

1 Task Overview

In examining complex networks, such as social networks, community structure is characterised as a node of networks that can be grouped into specific sets of nodes in that each set of nodes is connected internally. Within a community network, the vertices are completely connected to one another. This idea comes from the general principle that different pairs of nodes are more likely to be connected if they are members of the same community than nodes that do not share the same communities.

Community detection is the process of finding communities within an arbitrary network through computational algorithms. In general, community detection provides a means of uncovering organizational principles within networks. This is valuable in revealing not only the network structure, but also the relationship between interacting nodes. Higher-level implications of community detection include network infrastructure examination, node interaction, and inference of missing attribute values, prediction of unobserved connections, and more.

In this coursework we were tasked with find communities within a twitter dataset. The data set contained a total of approximately 41 million nodes and over 1 billion edges. The task therefore required a sufficiently fast algorithm to be used to do the community detection.

2 Algorithm

2.1 Algorithm Requirements

For this task the algorithm we chose had to fulfil key requirements. Firstly it had to be fast; as the data set is so large the time required to process all the data could have been huge. It was therefore important that the algorithm we chose was sufficiently fast even in its worst case scenario. Secondly, it needed to be relatively simple; Due to time constraints the algorithm could not be overly complicated. It needed to be understood and implemented in a matter of weeks, so its simplicity was very important.

Taking these requirements into account we looked at a number of different algorithms and decided which would be most suitable.

2.2 Possible Algorithms

When researching community detection algorithms we came across many different methods. Of these there were three that were strongly considered. These were the Girvan-Newman Algorithm, BGLL and a label propagation algorithm.

2.2.1 Girvan-Newman Algorithm

The Grivan-Newman Algorithm (*Girvan and Newman*, 2002) was the first one we examined as it is commonly used for community detection. The algorithm works using edge betweenness centrality measures. It does so by calculating the betweenness measure of all nodes in the network and removing the edge with the highest betweenness. After this any edges affected by this removal have their betweenness recalculated and the previous step is repeated. This is done until there are no edges left and so only the communities themselves are left.

This method of community detection is quite simple and easily implemented. Unfortunately it is quite slow with a time of $O(m^2n)$. This is because the betweenness of the graph is recalculated multiple times. Not doing so can result in large errors so it is unavoidable. The algorithm can run at around 10,000 nodes per hour. With a dataset of our size we required an algorithm that can run much faster. Due to this we looked at the BGLL Algorithm.

2.2.2 BGLL Algorithm

The BGLL Algorithm (*Blondel et Al, 2008*) is quite suitable for large networks because it displays linear complexity. It can run at around 10 million nodes per hour, which makes it quite suitable for our dataset. However, it is a greedy algorithm that is based on an increase in modularity. This is done in two major phases. The first phase starts by initializing n communities where each node characterizes exactly one community and then moving each node to a neighbour's

community. Then the modularity is increased until it can no longer be increased any further by merging. From here, the second phase involves creating a new network with new nodes as communities that were formed in the first phase. The weights between new nodes then become the weights between the original communities. The algorithm then goes back to the first phase to iterate until the modularity cannot be increased any further.

The BGLL Algorithm was considered for this project, but because it uses a sophisticated formula, it proved hard to implement given the complex nature of the algorithm and the time constraints of the coursework.

Due to the complexity of the BGLL and the speed of the Girvan-Newman Algorithm we decided to look for the strongly connected components in the network rather than every possible community. For this we looked towards a Label Propagation Algorithm.

2.2.3 Label Propagation Algorithm

To search for the strongly connected components we decided to look at the Bi-Directional Label Propagation Algorithm (*Lv and Xie, 2012*). This algorithm works in 4 steps. The first is a positive label propagation (PLP) step. Here a node is given a label depending on its value. Then its children are given labels the same way. If a child node has a label greater than that of its parent node, the label assigned is changed to the same as the parent node. This is repeated throughout the network. An example of this can be seen in figure 1.

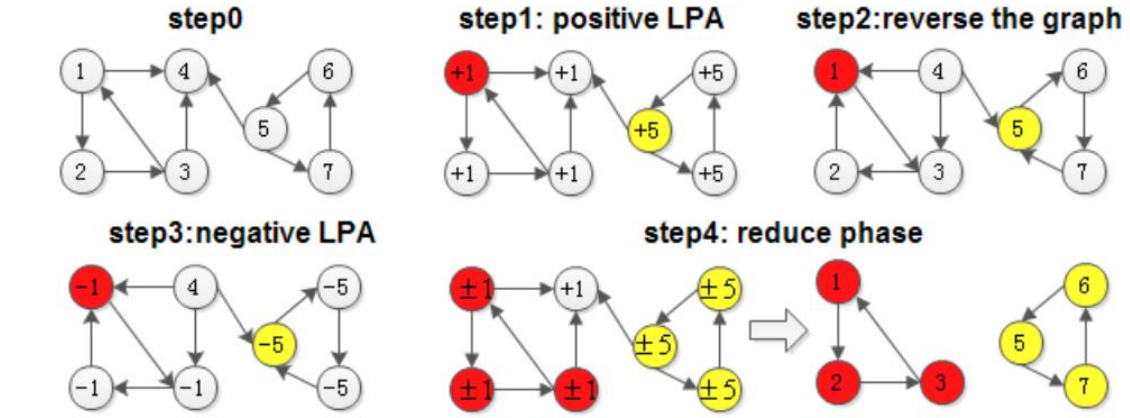

Figure 1 - Label Propagation Algorithm Steps

After this, in step 2, the original network is reversed. This means that all directed edges are turned the other way. On this new reversed network negative label propagation (NLP) is performed. This is the same as the PLP however the labels assigned are given a negative value. Following this, in the final step, the labels of the nodes are checked. The nodes that have the same positive and negative labels make up strongly connected components in the network.

This algorithm seemed to be appropriate for our required task. It is quite simple in terms of our understanding. It did not require a huge understand of the underlying mathematics involved, which was a problem for other algorithms. It also runs at an appropriate speed, with our calculations indicating that it should handle our dataset of 41 million nodes in around two hours. For these reasons we chose this label propagation algorithm for this task.

Implementation wise, we tried to minimise the number of steps that were required to complete all four steps. For this reason some of the algorithm was done out of order. This is further explained in the chapter 3 - implementation.

3 Implementation

Whilst the algorithm we had chosen is split into four clearly defined steps, the implementation of these steps is slightly different. We created a workflow diagram to map out exactly each step in the process. This can be seen in Figure 2 (This flow can be seen with the actual data in Figure 13 located in the appendix). As the diagram indicates, the Data first is sent through the Adjacency List Algorithm, where positive and negative are generated, and then to Label Propagation to Merger to the Positive/Negative Counter Combiner, where the positive and negative labels are put together, and then finally the output is emitted. The Label Propagation code is set so that the Label Propagation is called on itself when the Counter >0. More detailed descriptions of these various steps are described in the sections below. Initially, before even selecting the algorithm, we implemented an edge counter, in order to find out exactly how big the network in the dataset was.

N.B. When running the code on the twitter data we ran into many different problems relating to the Hadoop cluster. Every time we ran the code on the dataset we got a “Error: Java heap space” message. We spoke to the lecturer on many occasions regarding this who found no errors in the Code itself, thinking that the problem is with the amount of memory in the cluster. We tried many different things to rectify this including:

- Extending memory by conf.set("mapreduce.child.java.opts", "Xmx2048m"); and Xmx20480m and Xmx204800m.
- Using SequenceFileoutput format - the job became even slower.
- Using BFS-like structure with ArrayWritables, which made our code crash even on really small piece of data.
- Merging all files into one and trying to run wordcounter and our algorithm on it. This gave the same error.
- Replacing some splits by tokenizers.

After trying all of these we decided to look at other datasets. Initially we wanted to use just part of the twitter dataset. This proved impossible as accessing the data required more memory that we are given and taking the “head” or “tail” of the data did not provide more than one community. We then looked into using the dataset from google+. Unfortunately this data set is also very large and in a very different format to the twitter data. We decided not to use this as it would require too many changes to our code and would most likely give us the same error.

In the end we decided to create a sample dataset. This data was written in the same format as the initial twitter dataset. The code should therefore perform on this data and the twitter data in the exact same way, as long as the Hadoop cluster can cope with the amount of data.

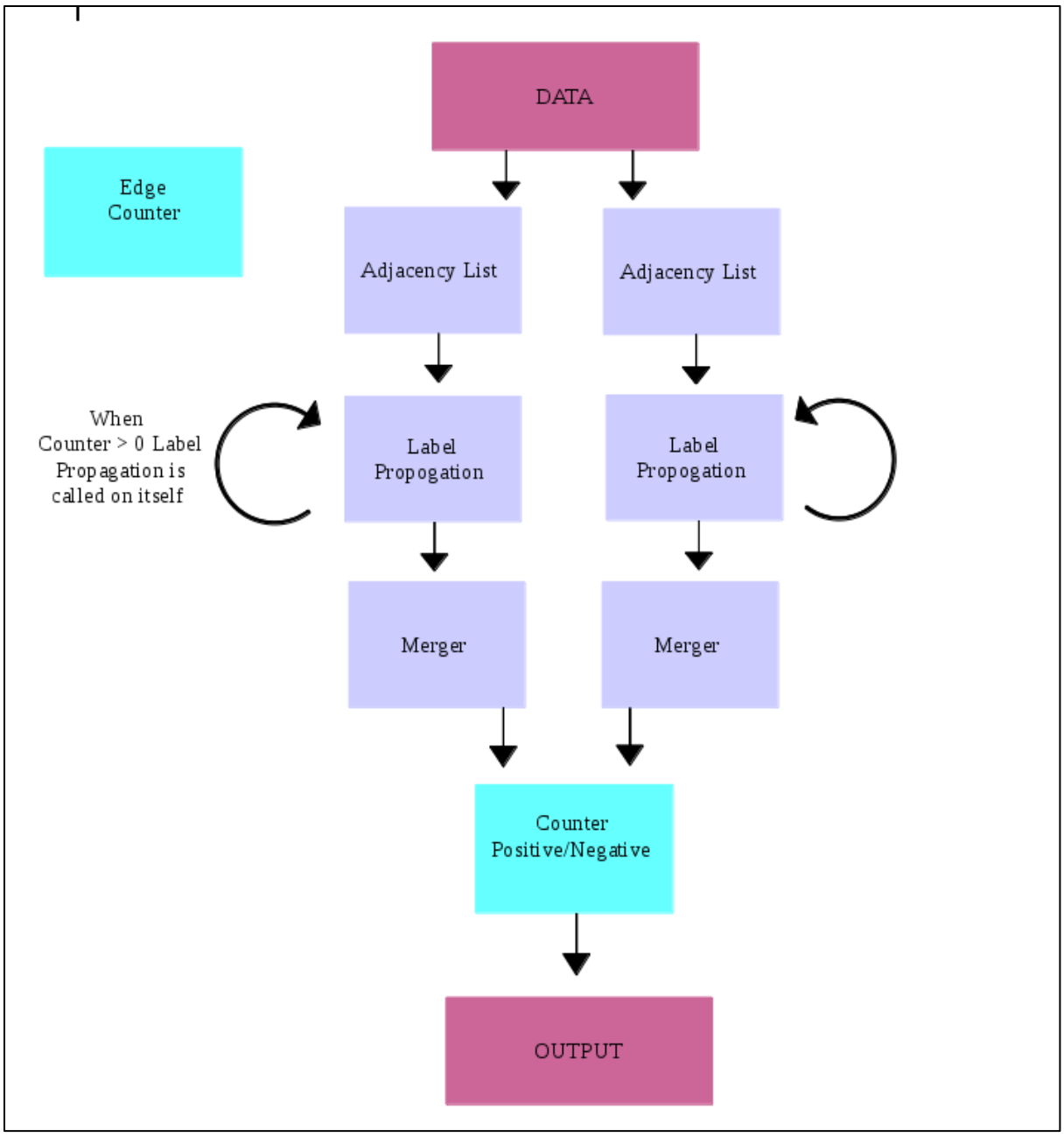

Figure 2 Workflow Schema

3.1 Edge Counter

For the Edge Counter we used code very similar to a word count. This can be seen in Figure 3.

```
public static class Reduce extends MapReduceBase implements Reducer<Text, IntWritable, Text, IntWritable> {
  public void reduce(Text key, Iterator<IntWritable> values, OutputCollector<Text, IntWritable> output, Reporter reporter) throws IOException {
    int sum = 0;
    while (values.hasNext()) {
      sum += values.next().get();
    }
    output.collect(key, new IntWritable(sum));
  }
}
```

Figure 3 Edge Counter code

Here each the total number of lines in the dataset is calculated. As the data contains one edge per line, this is an accurate count of how many edges there are in the data. This gave us a clear indication of what we were working with and helped us find the requirements of the algorithm we chose.

3.2 Adjacency Lists

Adjacency List Mapper

In the Adjacency List Mapper, all of the data is placed into the array named users. The data consists of rows of user-follower pairs. Each row is in a text format with the follower separated from the user by tab. Each row is parsed and the user-follower pair is sent to the Reducer. There are no multiple followers at this stage of the workflow. Instead, each row only emits one user and its follower; other rows can also contain this userid with other followers. **The if**-statement in the code checks whether the user, or node, has **any** follower. The key and value sent to the Reducer are the user and **its follower**, respectively. Hence, a user with a list of followers will arrive at the Reducer. The code for the Adjacency List Mapper can be seen in Figure 4 presented below.

```
public class ALMapper extends Mapper<Object, Text, LongWritable, LongWritable> {
   private LongWritable user = new LongWritable();
   private LongWritable follower = new LongWritable();
   public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String[] users = value.toString().split("\t");
    if(users.length>1){
        user.set(Long.parseLong(users[0]));
        follower.set(Long.parseLong(users[1]));
        context.write(user, follower);
    }
   }
```

Figure 4 Adjacency List Reducer code

Adjacency List Reducer

The Adjacency List Reducer takes the key (user) and value (number of followers), and puts all the followers into a string. From here, the actual **list** of followers for each user is outputted. The Reducer emits the userid as a key, a positive label in the form +userid, and an adjacency list formed from the list of followers that it receives from the Mapper. The adjacency list is separated from the label by a space, and a comma separates each of the followers within the list. The code for the Adjacency List Reducer can be seen in Figure 5 presented below.

```
public class ALReducer extends Reducer<LongWritable, LongWritable, LongWritable, Text> {
    Text result = new Text();
    public void reduce(LongWritable key, Iterable<LongWritable> values, Context context) throws IOException, InterruptedException {
        StringBuilder concat = new StringBuilder();
        concat.append("+");
        concat.append(key);
        concat.append(" ");
        for (LongWritable value : values) {
          concat.append(value.get());
          concat.append(",");
        }
        concat.setLength(concat.length() - 1);
        result.set(concat.toString());
        context.write(key, result);
    }
```

Figure 5 Adjacency List Reducer code

Adjacency List Reversed Mapper and Reducer

The code for Adjacency List Reversed Mapper and Reducer follows a similar structure/purpose in the Adjacency List Mapper and Reducer described above. However, in the Adjacency List Reversed Mapper, the pair being emitted is reversed. The follower is emitted as a key and the value is its corresponding user. The AL Reversed Reducer follows the same logic as the AL Reducer, but instead of positive labels, negative labels are assigned in the form –userid. The adjacency list developed through these steps is characterized by a list of corresponding users for a follower, rather than followers of one user in the output. The code for the Adjacency List Reversed Mapper and Reducer can be seen in Figure 6 presented below.

```
public class ALRMapper extends Mapper<Object, Text, LongWritable, LongWritable> {
   private LongWritable user = new LongWritable();
   private LongWritable follower = new LongWritable();
   public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String[] users = value.toString().split("\t");
    if(users.length>1){
        user.set(Long.parseLong(users[0]));
        follower.set(Long.parseLong(users[1]));
        context.write(follower, user);
    }
   }
```

```
public class ALRReducer extends Reducer<LongWritable, LongWritable, LongWritable, Text> {
    Text result = new Text();
    public void reduce(LongWritable key, Iterable<LongWritable> values, Context context) throws IOException, InterruptedException {
        StringBuilder concat = new StringBuilder();
        concat.append("-");
        concat.append(key);
        concat.append(" ");
        for (LongWritable value : values) {
            concat.append(value.get());
            concat.append(",");
        }
        concat.setLength(concat.length() - 1);
        result.set(concat.toString());
        context.write(key, result);
    }
```

Figure 6 Adjacency List Reverse Mapper and Reducer code

3.3 Label Propagation

After the Label Propagation performs its first iteration, it enters the while loop, which runs only when the counter >0. It only stops running when no nodes are changed, or rather when nothing els e is added to the counter. Each iteration uses the output of the previous one as its input, and generates a new folder as the output. Here, the output of the initial job is run as the input files/depth_0, and the output would be files/depth_1. From here, depth_1 is the input and depth_2 would be the output, etc… In order to make the job faster, we tried to define most of the vars outside of the Mapper or R**educer functions. We also used StringB**uilder instead of the usual concatenation as we found it much more efficient. The Label Propagation runs the initial job with whatever input folder and output folder that a user specifies, however it is important to run it with files/depth_0 output folder for the program to succeed.

The counter is accessible from the Reducer, and it incremented whenever one of the nodes is changed. To detect a change, an extra field is emitted from the Mapper. This is the original label so that the reducer is able to detect the change.

The Mapper emits the userid, positive or negative label, the adjacency list, and the original label for each node. Then, for each node from its adjacency list, the Mapper emits the userid and label as the key and value. The code for the LP Mapper can be seen in Figure 7 below.

```
public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException
{
   //Getting the input line
    input = value.toString().split("\t");
    if(input.length>1)
    {
        user.set(Long.parseLong(input[0]));

        if(input[1].split(" ").length>0){
            label_text.set(input[1].split(" ")[0]);
        }else{
            label_text.set(input[1]);
        }

        StringBuilder concat = new StringBuilder();
        concat.append(input[1]);
        concat.append(" ");
        concat.append(label_text);
        data.set(concat.toString());

        context.write(user, data);
        System.out.println("parent"+" "+user+" "+concat);

        if(input[1].split(" ").length > 1){
            StringTokenizer st = new StringTokenizer((input[1].split(" ")[1]), ",");
            while (st.hasMoreElements()) {
            user.set(Long.parseLong(st.nextToken()));
            context.write(user, label_text);
            System.out.println("child"+" "+user+" "+label_text);
```

Figure 7 Label Propagation Mapper code

In the Reducer, everything is grouped together on the basis of ID. For each ID, the minimum absolute value of the label is chosen. The Reducer output is in the format of userid, positive or negative label, and the adjacency list. The code for the LP Reducer can be seen in Figure 8 below.

```
public void reduce(LongWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException
{
   for (Text value : values) {
   input = value.toString();
       if(input.split(" ").length > 0)
      {

        sLabel=input.split(" ")[0];
              lLabel=Long.parseLong(sLabel);

        //if exists the adjacent List
        if(Math.abs(lLabel) < Math.abs(lminLabel)){
              lminLabel = lLabel;
           }

        if(input.split(" ").length == 2){
              origLabel = Long.parseLong(input.split(" ")[1]);
              }
        if(input.split(" ").length == 3){
            sNodeList=input.split(" ")[1];
            origLabel = Long.parseLong(input.split(" ")[2]);
        }
      }
  }
```

```
if(origLabel!=lminLabel){
    context.getCounter(LabelPropagation.UpdateCounter.UPDATED)
      .increment(1);
}

if(lminLabel>0){
  result.set("+"+lminLabel+" "+ sNodeList);
  //+" "+origLabel+" "+lminLabel+" "+context.getCounter(LabelPropagation.UpdateCounter.UPDATED).getValue()
}else{
  //if it's -, then it's already having it in front unlike the plus
  result.set(lminLabel +" "+ sNodeList);
}

context.write(key, result);
//result.set(values.toString());
//context.write(key, result);
```

Figure 8 Label Propagation Reducer code

3.3 Merger

The Merger Mapper takes user-label-adjacency list lines from the output of Label Propagation and emits the label and user as key, and a value to group them later in the Reducer through the means of a label. The Merger Reducer essentially groups users by their label and outputs this label as the key and a list of corresponding users with a "," separator as the value. This can be seen in Figure 9.

```
public class MMapper extends Mapper<Object, Text, Text, Text> {
   private Text label = new Text();
   private Text user = new Text();

   public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String[] users = value.toString().split("\t");
    if(users.length>1){
        label.set(users[1].split(" ")[0]);
        user.set(users[0]);
        context.write(label, user);
    }
   }

}


public class MReducer extends Reducer<Text, Text, Text, Text> {
    Text result = new Text();
    Map<Text, Integer> nodes = new HashMap<Text, Integer>();
        public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
           StringBuilder concat = new StringBuilder();
           for (Text value : values) {
              concat.append(value);
              concat.append(",");
           }
           concat.setLength(concat.length() - 1);
           result.set(concat.toString());
           context.write(key, result);
        }

}
```

Figure 9 Merger Mapper and Reducer

3.4 Combiner

Combiner Mapper

The combiner mapper takes the positive and negative output generated from the merger. For each row, there is a list of users separated by "," as a value and positive or negative labels as a key. There is an array that holds these nodes (users). The while loop looks through the array and while there are values present in this array, it looks at each token and sends the tokens to the Combiner Reducer with the corresponding user. For each user in the list, it takes a positive/negative label and emits its absolute value as the key, and the userid as the value. The code for the Combiner Mapper can be seen in Figure 10 presented below.

```
public class CPNMapper extends Mapper<Object, Text, LongWritable, Text> {
   private LongWritable user = new LongWritable();
   private Text data = new Text();
   public void map(Object key, Text value, Context context) throws IOException, InterruptedException {
    String[] users = value.toString().split("\t");
    if(users.length>1){
        user.set(Math.abs(Long.parseLong(users[0])));
        StringTokenizer st = new StringTokenizer(users[1], ",");
        while (st.hasMoreElements()) {
            data.set(st.nextToken());
            context.write(user, data);
        }
    }
   }
```

Figure 10 Combiner Mapper code

Combiner Reducer

In the Combiner Reducer, the absolute value label is the key, and the user value is checked if it appears twice for the label. If so, it is added to this label's connected nodes graph. In other words, some nodes from the Mapper are sent to the reducer twice, once with the positive label and once with the negative label. For those that do display this trait, it indicates they form a connected map. The first if statement checks if any of the nodes for this label appeared twice. If it does, the Reducer writes this as a result and outputs it. For each user, the label is emitted together with the value as the user. If there is no match, then the Reducer will not output anything. Finally, the map is erased at the end of the algorithm in order to avoid memory overheads. The code for the Combiner Reducer can be seen in Figure 11 presented below.

```
public class CPNReducer extends Reducer<LongWritable, Text, LongWritable, Text> {
    Text result = new Text();
    String[] strarr;
    Map<Text, Integer> nodes = new HashMap<Text, Integer>();
        public void reduce(LongWritable key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
            StringBuilder concat = new StringBuilder();
            for (Text value : values) {
                if(nodes.containsKey(value)){
                    concat.append(value.toString().trim());
                    concat.append(",");
                }else{
                    nodes.put(value, 1);
                }
            }
            if(concat.length()>0) {
                concat.setLength(concat.length() - 1);
                strarr = concat.toString().split(",");
                for(String str:strarr){
                    result.set(str);
                    context.write(key, result);
                }

            }
            nodes.clear();
```

Figure 11 Combiner Reducer code

4 Conclusion

After running the code, the output was run through “RStudio” using the “IGraph” module. Using this allowed us to visualise the output data in a graph. As seen in Figure 12, the nodes have been split into different communities, showing where the strong ties are. If we could use the original dataset, this tool would be extremely useful in identifying where the strongest on twitter are.

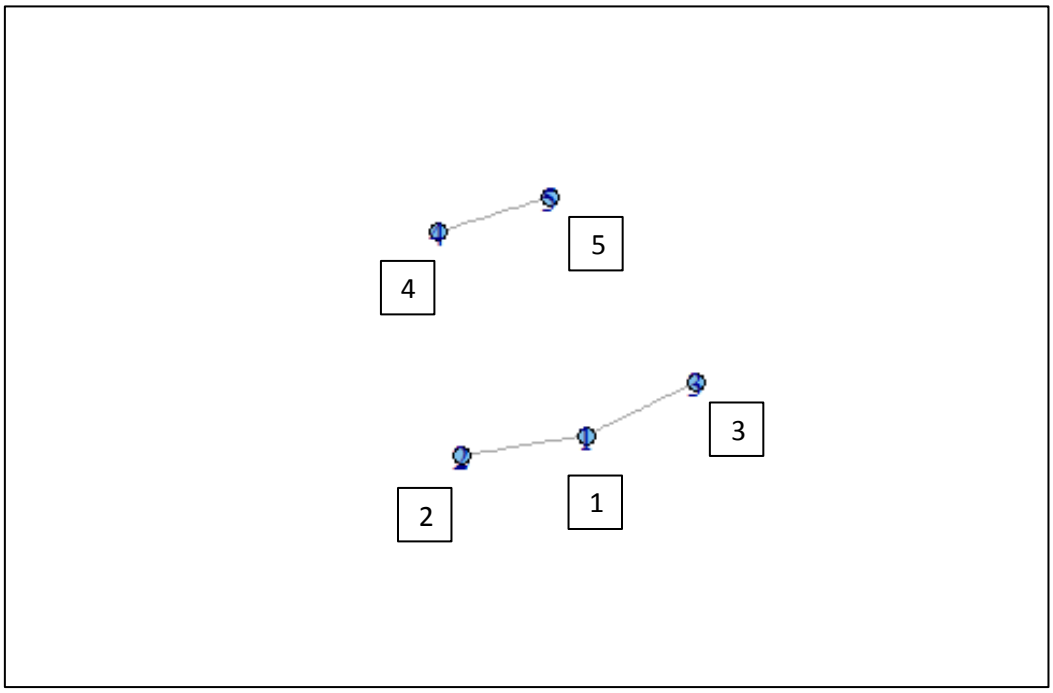

Figure 12 Graph of output

If we were given the opportunity to further study this topic we would focus on two points. The First would be to not only look for strongly connected components but for other connected components. This would include weak ties and communities within communities. Doing this would require the use of other algorithms that would need to be researched and implemented. The second focus would be on using a different, modularity based algorithm that would be more efficient, but went a bit beyond what could be done in this timeframe.

5 References

Blondel, V. D.; Guillaume, J.-L.; Lambiotte, R.; and Lefebvre,E. 2008. Fast Unfolding of Communities in Large Networks.Journal of Statistical Mechanics: Theory and Experiment 2008:P10008.

Girvan, M., and M. E. J. Newman. “Community Structure in Social and Biological Networks.” *Proceedings of the National Academy of Sciences of the United States of America* 99.12 (2002): 7821–7826. *PMC*.

Lv, Lu, and Lei Xie. "Enumerate Strongly Connected Components of Large-scale Graph with MapReduce." 2012.

6 Appendix

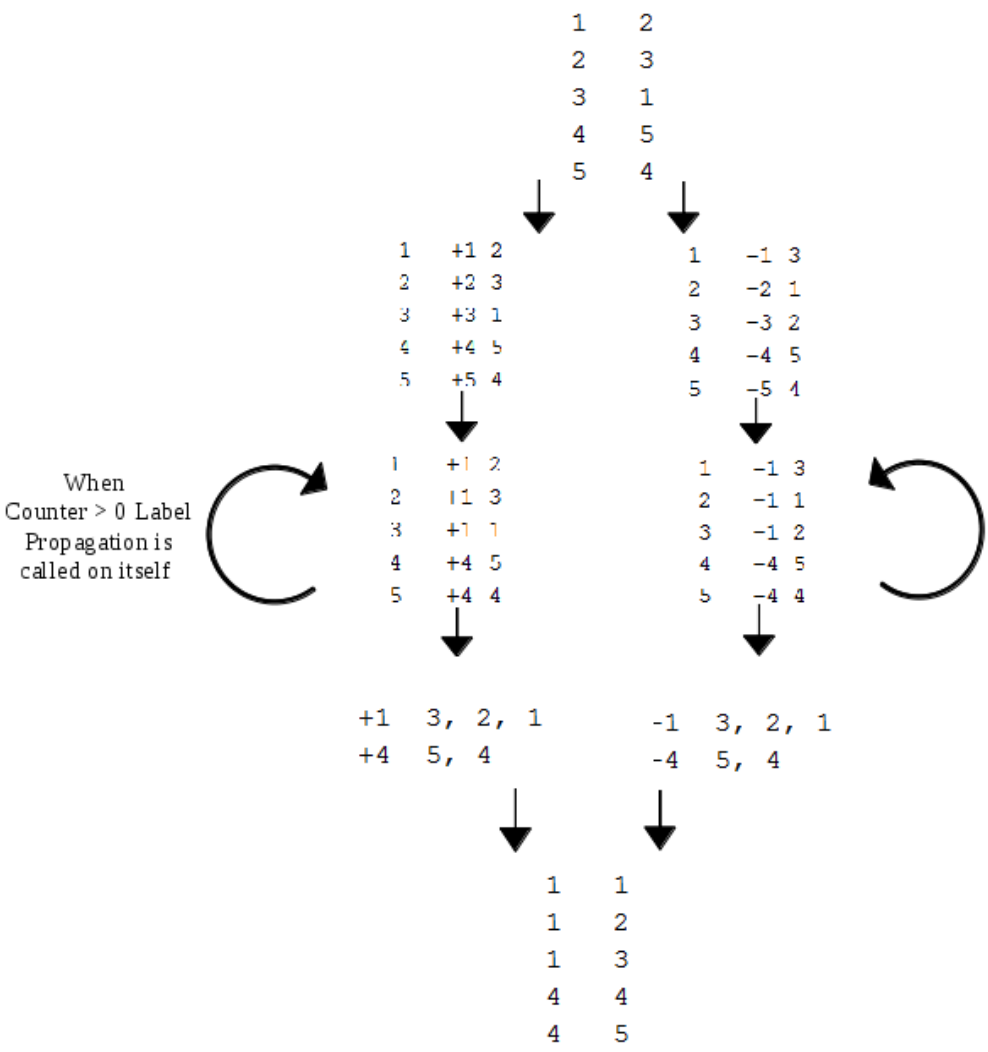

Figure 13 Data Flow