Lab 2: Network Structure

February 26th, 2014

Objectives

The goal of this lab is to extract and analyze basic properties of large networks, and to gain some intuition about them. In particular, we will look at

- the distribution of node degrees,
- the appearance and collapse of a giant connected component when the graph density varies,
- strong / weak links and their connection to the size of the giant component.

Download the tar archive `ix-lab02.tar.gz` from Moodle, extract it into your `myfiles` directory, and import the project into Eclipse. Please refer to Lab 1 if you have any difficulties.

Deliverables

There are three deliverables that should be handed in through Moodle, before the start of the next lab (the precise deadline will be communicated on Moodle.)

1. The plot of the degree distribution of links between Wikipedia articles (Sec. 1.)
2. Your implementation of the class `TargetedEdgeRemoval` (Sec. 2.)
3. The plot of the size of the giant component as a function of the number of edges removed, comparing random to targeted removals (Sec. 2.)

1 Degree Distribution

In this part, you will work with the Wikipedia articles graph extracted during the last session. We provide you with two snapshots of the graph:

1. `data/wikigraph-sample.txt` in your project folder. This is a small sample used for testing your jobs locally.
2. `/ix/wikigraph.txt` on HDFS. This is the full graph of links between Wikipedia articles, for which we will compute the node degree distribution on the Hadoop cluster. It is the result of the last step of Lab 1.

Denote N_d the number of nodes with degree d, i.e. with d neighbors. Our objective for this part is to compute the list of pairs (d, N_d) for $d \in \{0, \ldots, d_{\max}\}$, where $d_{\max}$ is the maximum degree among all the nodes of the network.

The file that represents the graph looks as follows:

```
Article1|Article2       1
Article2|Article3       1
```

The value to the right (1) could be thought of as a weight; for our purposes we will simply ignore this value and consider the graph to be unweighted. We further simplify the problem by considering the graph to be *undirected*: edges don't have a direction and we do not distinguish incoming and outgoing links.

Given the structure of the input file (basically, a list of edges), we will divide our problem into two tasks.

1. Associate to each node the list of its neighbors.
2. Count how many times each degree occurs across the network.

1.1 Extracting Neighborhoods

We have created a MapReduce job, `ix.lab02.degdist.NeighborsSet`, that performs the first task. Your job is to implement the corresponding Mapper and Reducer (`NeighborsSetMapper`, respectively `NeighborsSetReducer`.) The output of the Reducer should look as follows:

```
Article1        Article2
Article2        Article1, Article3
Article3        Article2
```

Here we see that the degree of `Article1` is 1 and that of `Article2` is 2. Once you have finished your implementation:

- Use the unit tests provided in the `test` folder to make sure your Mapper and Reducer work correctly.
- Run `NeighborsSet` locally on `data/wikigraph-sample.txt`.

1.2 Computing Degree Occurrences

The second MapReduce job, `ix.lab02.degdist.DegreeDistribution`, takes the output of the first job as input (i.e. the list of neighbors for each node), and outputs for each observed degree the number of times it occurs in the graph.

Just like before, your job is to implement the Mapper `DegreeDistributionMapper` and the Reducer `DegreeDistributionReducer`. The output of the Reducer should look as follows:

```
1       2
2       1
```

The intepretation is that there are two nodes with degree 1 (first line), and one node with degree 2 (second line.) Again, take advantage of the unit tests to check your implementation, then run the job locally on the sample.

1.3 Putting It All Together

Now that we have all the pieces together, we are ready to process the full dataset on the cluster. To avoid having to perform the two tasks manually, we have created a third job, `WikipediaDegreeDistribution`, that combines both operations in one call. You can take a look at its source to see how it works.

As done in Lab 1, export the project as a `jar` archive and transfer it to the cluster access server, for example via `myfiles`. Run the job using the command:

```
hadoop jar ix-lab02.jar ix.lab02.degdist.WikipediaDegreeDistribution /ix/wikigraph.txt
    lab2/degdist
```

Use `hadoop fs -getmerge` to merge the output of the reducers, and copy the resulting file to a local folder. Lastly, pass the file to the Python script[1] `scripts/degree-distrib.py`. This will produce two plots of the data.

- Which plot looks more useful to you? Why is that the case?
- Can you list some of the properties of the degree distribution?
- Which well-known probability distribution is it close to?

2 Robustness of Giant Component

The second part of this lab is focused on the *giant component* of a network, i.e., the largest connected component. The main objective of this part is to see how the size of the giant component evolves as we reduce the density of edges in the graph. First we will remove edges at random, and in a second step you will be asked to come up with a more clever strategy to reduce the giant component's size faster.

For this part, we will not need any MapReduce job and all the commands will be executed locally. You will work with a third-party library, `JGraphT`[2], which provides facilities to represent and manipulate graphs. The dataset for this part is a sample of the Wikpedia articles network provided in `data/wikigraph-sample-giantcomponent.txt`. Make sure that you use this file and not the ones for the previous part when running your experiments.

2.1 Giant Component Size

Take a look at the class `ix.lab02.giantcomp.GiantComponent`. It is provided as is, and you don't need to fill in anything; it contains a few utility functions that will be useful for later steps.

Run the class on the dataset (`wikigraph-sample-giantcomponent.txt`) in order to compute the size of its largest connected component.

2.2 Random Edge Removal

First, we will look at the evolution of the giant component's size when edges are removed from the graph at random. For this, we ask you to implement the function `apply()` in the class `RandomEdgeRemoval`. Pay attention to the following.

- Remove chunks of 100 edges at a time. Stop once the giant component reaches 20% or less of its initial size.
- After each iteration, record the size of the giant component in the by calling the `add()` method on `results`.

[1] You will need Python and Matplotlib to use the scripts provided in this lab. They should work out of the box on the machines in BC07-08, but might not on your personal computer.

[2] See: `http://jgrapht.org/`.

Run the class on the the dataset, and save the result somewhere. Pass the file as an argument to the python script `scripts/giant-component.py` to visualize how the size of the giant component evolves as edges get removed.

2.3 Breaking the Network Faster

Now, instead of removing edges at random, we ask you to come up with a strategy to remove edges in a more clever way in order to break up the giant component faster.

One way to approach this problem would be to design a *measure of strength* for edges. A value of strength could be associated to each edge, and at each step we would *remove the weakest edges*. Your first task is hence to find such a measure. To help you in this task, consider the networks in Fig. 1. For each graph, which edges are the weakest? That is, which ones would you remove in priority to break (or at least weaken) the connected component?

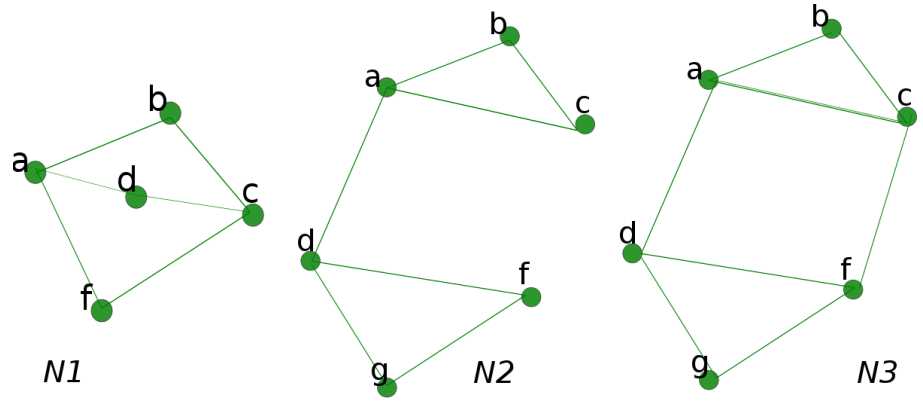

Figure 1: Find the weakest edges in each of the networks.

Hint: The edge (a, b) is called a *local bridge* if removing (a, b) makes the distance between a and b greater than 2, i.e., a and b do not share any common neighbors.

Implement your strategy in the `apply()` method of the class `TargetedEdgeRemoval`. Then, run the class on the dataset and save the results next to the ones you got for the random case. Using the script `giant-components.py` again, this time with the files for both experiments, you can visualize how much improvement you get.

Note that there is no single right answer to this part. In our reference implementation, we manage to shrink the giant component to less than 20% of its initial size after 92,000 edge removals. How does your strategy compare?