CS 549—Fall 2014
Distributed Systems and Cloud Computing
Assignment Four—Social Rank in Hadoop

Implement an algorithm called *SocialRank*, which can be used to find influential persons in a social network. You will then use your implementation of SocialRank to find the ten 'most influential' users in the LiveJournal social network.

The SocialRank algorithm

SocialRank is very similar to PageRank, except that it operates on social network graphs instead of hyperlink graphs. You should review the material on PageRank and its Map/Reduce implementation. We summarize the essentials here.

Let G = (V,E) be a social network graph, where each vertex v ∈ V represents an individual user and each edge, or 'link', $(v_1,v_2) \in E$ represents the fact that user v_1 has listed v_2 as a friend. Note that friendship links are not necessarily reciprocal: if user A is a friend of user B, that does not necessarily mean that user B is a friend of user A. The goal of SocialRank is to assign a *rank* r_i to each vertex v_i; the higher the rank value, the more 'influential' the algorithm considers the person to be.

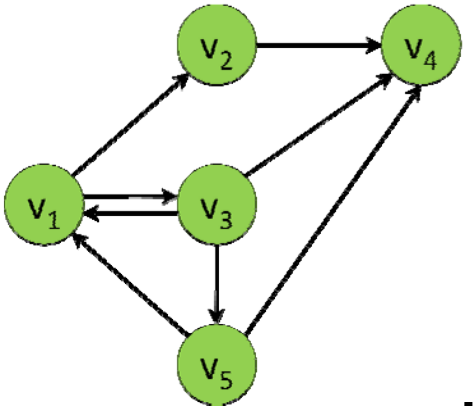
SocialRank is an iterative algorithm, that is, it consists of several rounds. Let r_i^k be the rank of vertex i in round k. Before the first round, all ranks are initialized to one, that is, $r_i^0 = 1$. Given the ranks of round k, the ranks for round k+1 are calculated using the following formula:

$$r_i^{k+1} = d + (1-d) \sum_{j \in B(i)} \frac{1}{|N(j)|} r_j^k$$

Here, d is a constant (we will use d=0.15), N(i):={j | (i,j) ∈ E} is the set of i's friends, and B(i):={j | (j,i) ∈ E} is the set of users/vertices that have i as a friend. The algorithm stops when all the ranks have converged, i.e., when $|r_i^k - r_i^{k-1}|$ is "small" for all i.

A simple example

Your algorithm will eventually run on a data set that contains more than 5 million users. This data set is too large for debugging because (a) each run will take a long time, and (b) we do not know up front what the correct answer should be. Therefore, we begin with the simple example shown below, which consists of just five vertices. In this example, V={1,2,3,4,5} and E={(1,2), (1,3), (2,4), (3,1), (3,4), (3,5), (5,1), (5,4)}.



If we set d=0.15 and run a single round of SocialRank, we get the following values:

i	1	2	3	4	5
r_i^1	0.858	0.575	0.575	1.708	0.433

If we run the algorithm until the ranks for all the vertices change by less than 0.001 compared to the previous round, we arrive at round 10 and the following values:

i	1	2	3	4	5
r_i^{10}	0.332	0.291	0.291	0.580	0.233

Not surprisingly, the user represented by vertex 4 is returned as the most influential.

Data format

Input format: We will assume that the social graph is initially available as a list of edges. That is, we will have a file that contains one line for each link, and each line contains a pair of numbers that represent the vertices that are connected by the link. (We termed this the *edge* representation versus the *adjacency list* representation in the lecture notes.) For example, the graph from above would be encoded as shown on the right. In practice, the data may be spread across multiple files because the data set is very large.

1	2
1	3
2	4
3	1
3	4
3	5
5	1
5	4

Output format: The goal is to produce a file that contains the social rank of each individual in the social network. There should be one line for each vertex, and each line

4	0.580
1	0.332
2	0.291
3	0.291
5	0.233

should contain the vertex identifier and the social rank. The file should be *sorted by rank in reverse order*. For example, the final output from the example above would be encoded as shown above.

Implementation strategy

We provide two implementation hints. First, it is very important to choose a suitable intermediate format (i.e., the format in which the data is output after each round). The input format is not particularly suitable for this because, for instance, the rank is associated with an individual vertex and not with a single link. Also, the intermediate format must clearly preserve all the information in the original input; thus, we cannot simply output a list of vertices and their ranks because the next round would no longer have the links.

Second, SocialRank cannot be implemented efficiently with a single MapReduce job. We can, however, implement each round as a separate job in an iterative process. Thus, the output of round k will be used as the input of round k+1. In addition, we will need three additional types of jobs: One for converting the input data into our intermediate format, one for computing how much the rank values have changed from one round to the next, and one for converting the intermediate format into the output format.

Q1 Implementing a MapReduce task for SocialRank (60%)

Your first task is to implement a MapReduce job that implements the various sub-operations within SocialRank. Your driver should read the command-line arguments and, depending on the first argument, implement the following four functions (you will need to write a mapper/reducer pair for each):

- `init` *`inputDir outputDir #reducers`*: This job should read the file(s) in the input directory, convert them into your intermediate format, and output the data to the output directory, using the specified number of reducers.
- `iter` *`inputDir outputDir #reducers`*: This job should perform a single round of SocialRank by reading data in your intermediate format from the input directory, and writing data in your intermediate format to the output directory, using the specified number of reducers.
- `diff` *`inputDir`*$_1$ *`inputDir`*$_2$ *`outputDir #reducers`*: This job should read data in your intermediate format from *both* input directories and output a single line that contains the maximum difference between any pair of rank values for the same vertex. You must use absolute values for the differences, i.e., the change from 0.98 to 0.97 is 0.01. While diff runs as a single task, you may find it necessary to implement it using two different MapReduce jobs run successively.
- `finish` *`inputDir outputDir #reducers`*: This job should read data in your intermediate format from the input directory, convert the data to the output format, and output it to the output directory, using the specified

number of reducers.

Additionally, you should write a composite function that needs to be submitted only once and runs the entire SocialRank algorithm from beginning to end, i.e., until convergence has occurred.

- `composite` *`inputDir outputDir intermDir`*$_1$ *`intermDir`*$_2$ *`diffDir #reducers`*`:` This function should run the init task, then it should alternate between running the iter and diff tasks until convergence has occurred **(diff <=30 in the case of the LiveJournal data)**, at which point it will run the finish task and place the output into <outputDir>. Note, running the diff task after every iteration could add considerable time to your job, so you may want to run the diff task after every two or three iterations to save computation time.

Each job must delete the output directory if it already exists, and it must also output your name to `System.out` every time it is invoked. The main class of your job must be `edu.stevens.cs549.hadoop.socialrank.SocialRankDriver`. You should document your code sufficiently to enable the grader to fix minor bugs if necessary. A small number of points will be awarded for good documentation, and no partial credit will be given for undocumented code.

You should start with the Eclipse project that is provided, that gives you some code to get you started. This is a Maven project, so be careful to import it as such into your Eclipse workspace. The pom file already imports the requisite Hadoop libraries from global Maven repositories. You should still install Hadoop on your local machine, since you will need its libraries in your classpath when testing your app locally.

Debugging tips

A good way to make sure that your implementation for Question #1 is correct is to test it in pseudo-distributed mode, using the Hadoop installation in your VM and the example graph from above. This will save your AWS credits for the final job with the LiveJournal data. A typical sequence of jobs would look like this (with the input data in a directory called `in/`):

```
init in out1 1
iter out1 out2 1
diff out1 out2 out3
iter out2 out1 1
diff out2 out1 out3
iter out1 out2 1
diff out2 out1 out3
...
finish out1 out3 1
```

You should only implement the composite task after you are sure that the other tasks work properly. **Note:** for your submission, all five tasks (init, iter, diff, finish, composite) are expected to operate from the command line. Please make sure your submission can handle all of the arguments listed above.

Q2 Running SocialRank on real social network data (30%)

Your second task is to use SocialRank to find the ten "most influential" users in a snapshot of the LiveJournal social network from 2006. We will be using data from a paper by Mislove et al., IMC 2007 (paper available at http://www.mpi-sws.mpg.de/~amislove/publications/SocialNetworks-IMC.pdf). This data set contains 77.4 million links between 5.3 million users. It has already been split into multiple files and has been imported into Amazon S3 (http://s3.amazonaws.com/cs549/livejournal-data.tgz).

Step-by-step guide: Running on Amazon's Elastic MapReduce

The following is a step-by-step guide for setting up your solution to run on Amazon's Elastic MapReduce. To conserve your AWS credit, only use the full LiveJournal data and the maximum number of machines after you are sure that your solution works on the sample graph with one or two machines.

1. Go to the Amazon S3 Management Console (https://console.aws.amazon.com/s3/home) and create a new bucket with a unique name. This bucket will store input, output, MapReduce logs, and your exported jar file for use with Elastic MapReduce.

2. Within your created bucket, create two new folders named "jars" and "in". Upload your exported jar file into the "jars" folder, and upload the test graph data into the "in" folder. When you are ready to test with the LiveJournal data, download the .tgz file, extract it, and upload the files to your "in" folder in S3. Make sure to delete the old test graph data.

1. Now we are ready to run an Elastic MapReduce job. Go to the EMR Management Console (https://console.aws.amazon.com/elasticmapreduce/home) and click "Create New Job Flow".

2. Give your job flow any name, select "Run your own application -> Custom JAR", and then Continue.

3. Under "Jar Location*:" Type in the location of your uploaded jar file (containing your compiled code). If you followed the instructions from step 2, your path should be "*yourbucketname*`/jars/SocialRank.jar`"

4. "Jar Arguments*:" will have the arguments to run your composite task. Input

and output folders should be referred to as S3 paths. Here is a sample set of arguments:

```
edu.stevens.cs549.hadoop.socialrank.SocialRankDriver composite
s3n://bucketname/in s3n://bucketname/out intermediate1
intermediate2 diffdir 20
```

Note that only the input and output need to be S3 paths. Intermediate data can be stored in HDFS folders (This will happen automatically when no s3n:// is specified).

5. On the next page, specify the number of instances that will run your MapReduce job. To start, set the instance count for "Core Instance Group" to 1, and leave the instance count for "Task Instance Group" at 0. This will run your job with two machines – one master and one slave node. When you are ready to run your job on the full LiveJournal dataset, you can set the Core Instance count to 19 for a total of 20 machines.

6. On the next page, make sure to enable logging and set the log path to "`s3n://yourbucketname/log`". This will be very useful for debugging your job if anything goes wrong.

7. On the next page, select "Proceed with no bootstrap actions"

8. On the final page, review your settings to make sure they are correct. When you are ready, click "Create Job Flow" to start your job. This can take a long time on the LiveJournal data, but it should complete in less than 3 hours using the full 20 machines. On the sample graph, expect your job flow to take a maximum of 30 minutes.

What if things go wrong?

You can look at the Hadoop logs in the S3 Management Console under `yourbucketname/log/jobflowid/`. The most valuable log information will likely be under steps /2/, since these are the logs generated by your jar file.

Please watch your resource consumption while working on this assignment; if you cannot get to the convergence threshold within a reasonable budget (say, $15-$20), just submit the top-10 you have, and document the number of rounds in your README file for partial credit.

Q3 Experimenting with different numbers of reducers (10%)

Experiment with different numbers of reducers and track how this choice affects the overall speed of your MapReduce job flow. In your report, include the details about what combinations of machines and reducers you used and your results. Also

include your thoughts on why you saw these results (e.g. after a point, adding reducers may slow overall time, or running twice with the exact same arguments may show a notable difference).

Submitting

Once you have your code working, please follow these instructions for submitting your assignment:

- Export your Eclipse project to the file system.
- Create a zip archive file, named after you, containing a directory with your name. E.g. if your name is Humphrey Bogart, then name the directory Humphrey_Bogart.
- In that directory you should provide the zip archive that contains your sources, and your Hadoop app jar file, SocialRank.jar.
- Also include in the directory a report of your submission. This report should be in PDF format. **Do not provide a Word document.**

The content of the report is very important. A missing or sloppy report will hurt your final grade, even if you get everything working. In this report, describe how you implemented SocialRank in M/R, with a particular focus on the representations that you used for inputs to and outputs from the map and reduce steps, and the overall structure of your solution. You should also describe how you tested the code. Again, it is very important for your grade that you do adequate testing.

You should also provide a video of your local testing, on the small test graph. Remember that your name should be displayed on each iteration of the job.

You should also provide a video showing your use of EMR to set up and start a run of your MR task. Show that you have set up a bucket in S3 as prescribed (no need to show the upload), and show the steps in the EMR console. Make sure your name is visible in the video, but don't provide your AWS credentials in the video! There is obviously no point in showing the entire run in EMR, since it will take some time, the video should just show you starting the program running.

Remember the format of the submission: A zip archive file, named after you, with a directory named after you. In this directory, provide a zip archive of your source files and resources, a jar file of your submission, a report for your submission, and videos (unless you have uploaded the videos to Google Drive).

Do **not** upload the LiveJournal data as part of your submission.