CS 4320 Spring 2015 – Homework 4
Due: Friday April 17th, 11:59 pm

For this homework, you must work with one partner, as usual. It is out of 78 points and counts for 10% of your final grade. It will introduce you to Hadoop and Neo4j.

This homework, especially the Hadoop part, requires **significant installation and configuration**. For this reason, it is essential that you start early. In particular, if you are not familiar with Linux and/or command line work, you should get started over Spring Break and at least attempt installation following the instructions provided.

There will be a **special tutorial section on the homework on Tuesday, April 7th, from 6-8 pm in Gates Hall G01.** The TAs will discuss the homework and answer your questions. To make the most of the tutorial, you **should attempt to install both Hadoop and Neo4j beforehand** and bring your questions/problems.

Part 1: Page Rank using Map Reduce on Hadoop (38 Points)

In this part you will be using Hadoop to implement PageRank using Map Reduce. Please read all the material below before you jump into coding. It is a lot of information and you may want to go through it more than once before you get started.

In the below instructions, we first explain how to install and run the correct Hadoop environment on Linux, then we explain what you need to implement, and finally we give a tour of the auxiliary classes that we have provided.

Platform

This homework was written and tested in Ubuntu 14.10. We recommend that you implement it on Ubuntu. Windows and/or Mac users are welcome to try running Hadoop on their machines, but **we are not explicitly supporting these platforms**. TAs will provide advice on Mac/Windows if they happen to have expertise and time, but there is no guarantee they will have either. Historically Hadoop has been hard to get working on Windows and more reasonable on Mac OS X, although we experienced difficulties this semester with Hadoop 2.6.0 on OS X when we tried it.

If you do not have Linux, you can run an instance of it on top of your current operating system using free virtualization technology such as:

1) https://www.virtualbox.org/wiki/Downloads. If you use VirtualBox, you can run Linux as if you were running a program--no disk partitioning, wiping, or driver installation is necessary. Detailed instructions on how to use VirtualBox can be found using your favorite search engine.
2) VMware for Windows - You can get this using your Cornell netid
3) VMware for MAC - You can get this using your Cornell netid

If you use VMware, you will have to create a new virtual machine and install Ubuntu on it. The installation is fairly simple and you can use your favorite search engine for help with installation.

Some familiarity with command line interfaces will be required. If you are not familiar with command line interfaces, you may use your favorite search engine to discover how to use a terminal. This link may be of some help:
http://www.pas.rochester.edu/~pavone/particle-www/telescopes/ComputerCommands.htm
In Linux, you can access the manual for a command by typing **man <command name>** into the terminal.

Setting up Java and Hadoop in Ubuntu 14.10:

1) Installing useful tools

Type in the following command on your linux terminal to install build software such as "Make":
sudo apt-get install build-essential
Verify that make is installed by typing into the terminal: ***make -v***
You should get some output related to the version of make and not a 'command not found' error.

2) Installing Java

Keep in mind that we will be testing on Java 7, so do not use Java 8 functionality.

Open a terminal and type in the following command to install the Java runtime and Development kit:
sudo apt-get install openjdk-7-jre openjdk-7-jdk
Verify that Java is installed by running the following commands in a terminal window:

javac -version
java -version
You should get output related to the Java version and not a 'command not found' error.

3) Installing Hadoop 2.6.0

Connect to http://www.apache.org/dyn/closer.cgi/hadoop/common/ . Choose a download link and go inside the hadoop-2.6.0/ folder. Download hadoop-2.6.0.tar.gz and note the location where it was downloaded. Navigate to the location via the terminal by typing the following command on the **terminal:**
cd <absolute path to location>
Extract the contents by typing the following command into the terminal:
tar -xvf hadoop-2.6.0.tar.gz

To keep these instructions simple and consistent, we will install Hadoop to the /opt/ folder. You can, however, put it wherever you want. If you do so, replace all instances of /opt/ in these Instructions to your path. Move the uncompressed directory to /opt/ by running the command:
sudo mv hadoop-2.6.0 /opt/

Verify that you see a directory at **/opt/hadoop-2.6.0/bin/** . You will need to add the whole path to this directory to your PATH environment variable, so you can run Hadoop from any

directory. (The whole path can be obtained by stepping into the directory and typing **pwd**).
To modify PATH, open **~/.bashrc.** with a graphical text editor by typing: **gedit ~/.bashrc.**
Add the following line to the very bottom of the file:
export PATH=$PATH:/opt/hadoop-2.6.0/bin .

Also, while the file is open, if JAVA_HOME does not already exist, add this line to the very bottom:
export JAVA_HOME=/usr/lib/jvm/java-1.7.0-openjdk-i386
The value on the right side of "=" is the input the path to your Java folder.
If you've been following the instructions up until now, the path should be as shown above.
Otherwise, use the path to your Java folder.

Save and exit the file. Quit and reopen the terminal (or type **source ~/.bashrc**) and type **echo $JAVA_HOME**. Make sure the Java path you entered appears.

Verify that Hadoop was installed correctly using the command : **hadoop version.**
You should get a descriptive output, rather than a 'command not found' message.

As a way to get started with Hadoop, you may want to look at this tutorial:
http://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html

Getting started with the code:

Download the zip file that we have provided and place it in a folder of your choice. In that folder run the command : **unzip page_rank.zip** You will see a folder called page_rank. Run the command : **cd page_rank** to step inside the folder. You will see the following contents:
- Makefile
- src folder
- tests folder
- expected folder : This holds the results to the sample tests provided in the tests folder.

Run the Makefile using the command make ***make &>log.txt***
When it's done, if you look at the directory, you will see directories named stage0-9, a file called log.txt, and a jar file PageRank.jar. You have just compiled and run the skeleton code! This is a Hadoop MapReduce job that does nothing.

If you look at the log.txt file (open it using a text editor), you'll see a lot of information about the Hadoop run. This would have normally been printed to the screen, but the &>log.txt puts it in a file named log.txt. For debugging, we recommend that you put System.out.println() calls into your code. However, instead of being displayed on the screen, they will be put in log.txt if you run the above command.

Before you can run the code again, you'll need to run the command **make clean**. This will clean up everything from the previous run. ***However, it won't delete your log file.***
Now, you can start implementing PageRank!

The textbook at **http://lintool.github.io/MapReduceAlgorithms/ed1n/MapReduce-algorithms.pdf** contains a description of Page Rank as well as pseudocode in section 5.3. We recommend that you go through this.

Implementing PageRank
In the skeleton code, there are five files that you should be changing. You will be implementing two mappers and two reducers, which will work together to implement the PageRank algorithm. You will also be adding a little code to the main PageRank class. The five files are:

- TrustMapper.java
- TrustReducer.java
- LeftoverMapper.java
- LeftoverReducer.java
- PageRank.java

The Trust Mapper and Trust Reducer represent the core of the PageRank algorithm: they propagate the PageRank of nodes into their out-neighbors. The LeftoverMapper and LeftoverReducer tie up the loose ends. The PageRank that was "lost" from nodes with no out-links is spread evenly across all nodes, and the random jump factor is taken care of. (See Formula 5.2 in the textbook.)

To implement these, you will need to correctly use the framework of Hadoop. Much of that framework has been built up for you, as you can see below. However, you'll need to use the Hadoop API in your program. You can find that reference at http://hadoop.apache.org/docs/stable/api/.

The PageRank class is mostly written for you. However, you'll find one part that has a comment:
//set up leftover and size

This relates to keeping track of "lost" PageRank from nodes with no outlinks. At this point in the file, you should add code to set up the variables correctly so that they can be passed into the next mapper/reducer pair. We recommend using a Hadoop counter to accomplish this, but you may choose a different implementation.

If you do choose to use a counter for this, you will need to create a new .java file that contains an enum. Then, you will need to include that in the Makefile. If you look in the Makefile, you will see that there is a line:

bases = LeftoverMapper LeftoverReducer NodeInputFormat Node NodeOrDouble NodeOutputFormat NodeRecordReader NodeRecordWriter TrustMapper TrustReducer PageRank

You will need to change this to add the name of your new file (without the .java) to the end of the list. This will automatically add it to the build process that you need to run your project in Hadoop.

Code Tour

The skeleton code that we have provided you with does most of the Hadoop work for you. However, to implement PageRank, you will have to interface with this code. This means that it's going to be important that you know what this code does**. IT IS HIGHLY RECOMMENDED THAT YOU DO NOT EDIT THIS CODE!**

Much of the code is somewhat fragile, and it will be difficult to get working again if you change it. It is very possible to solve the problem with the skeleton code you have been given. Now, we will begin to look at the different Java files you have been given. Feel free to look at the code inside the files and read along.

PageRank.java
PageRank is the main class of your program. It sets up the Hadoop job and tells Hadoop where to read from and write to. It consists of 4 methods: the main method and 3 methods that return Job objects. The Job class represents a MapReduce run in Hadoop. The three methods that return Job set up a properly-formatted run. The main method is much more interesting. Here, we actually run the correct job. The idea is that, every other run, we will run the TrustMapper/ TrustReducer job, and on the odd runs we run the LeftoverMapper /LeftoverReducer job. For the first job, the input comes from the input directory. From then on, the input comes from the output of the last run.

Node.java
Node is probably the class you will deal with the most. It represents a node in the internet graph (i.e. a web page). It contains an identification number, a PageRank value, and a collection of outgoing links. It implements the iterable interface, which allows you to use a for-each loop to go through the outgoing links. The write and readFields methods are from the Writable interface. Writable allows for objects of class Node to be used as output and input from Mappers and Reducers.

NodeOrDouble.java
NodeOrDouble is a class that may seem a bit strange to you on first glance. You may construct it with either a Node object or a Double object. Afterwards, you cannot change which your NodeOrDouble object holds. You can, however, tell which it holds (isNode) and get out the object it holds (getNode and getDouble). Be careful, trying to retrieve the wrong type will result in a Null object! Again, it implements the Writable interface.

NodeInputFormat.java and NodeRecordReader.java
These classes provide the ability to read from files and get a Node out. This means that you don't have to parse lines from the text file yourself. The NodeInputFormat class tells Hadoop how to create an appropriate RecordReader, in our case the NodeRecordReader. It doesn't do anything else besides return a constructor for a NodeRecordReader. The NodeRecordReader class parses a file that contains Nodes for you.
A line in a node file looks like:
nodeid pagerank out1,out2,...
where nodeid is an integer, pagerank is a decimal number, and out1, out2, . . . are all integers.
 Here, nodeid is the id *of the node, pagerank is the node's current PageRank value, and the outs are the outgoing links of the node.*

NodeOutputFormat.java and NodeRecordWriter.java
These classes are the inverse of NodeInputFormat and NodeRecordReader. They print out Nodes to a file in the same format as above. They allow Hadoop to simply output nodes, without you having to do any pretty printing in your mapper and reducer code.

Makefile
This is not a Java file, as you can probably guess. Makefiles are designed to automate the compiling, running, and cleanup of large projects. Since Hadoop requires quite a bit to get set up, we've automated this with a Makefile. As described earlier, the command make will compile your Java and run Hadoop for you. The command make clean will then clean up everything from that process for you. For your purposes :
- If you run **make**, your code should compile and run and generate several intermediate directories
- If you run **make clean** , your intermediate directories will be cleaned up
 If you're interested in make, please see the official documentation at
http://www.gnu.org/software/make/manual/html_node/index.html

Tests
To test a file, place **one(!)** of the tests in the input directory and type **make clean**. If you run make clean when no clean up is needed (no intermediate directories are present), you will get errors which you may ignore. In order to get your code running, type : **make &>log.txt**.

Upon running this, your final result will be captured in stage9/output.txt. View this file before you run make clean again, so that you know what your final page rank values are. We have given you a limited number of tests to play with. However, you should expect that your code will have to pass more tests. The goal is to match our reference implementation. Note that, because we are dealing with floating-point numbers, your answers may be close but not match ours exactly. In this case, you can be considered to be passing the test. You can find our example output in the 'expected' folder that is provided to you inside the zipped archive.

There are four sample tests we have provided. The text *center* has one central node that all others are connected to, and other structure among the nodes. The test *singleton* is just a single node that connects to itself. *Triangle cycle* is three nodes arranged in a cycle. In the line test, the first node connects to all the others, the second to all the others except the first one, and so on (the last node has no out-edges).

Part 2: Graph Queries using Neo4j (40 Points)

And now for something completely different!

One fine day you wake up and find yourself on a strange planet called Gliese. You meet the Queen of the planet; she tells you that long ago, humans came to this planet, settled down and started interacting and interbreeding with the planet's original residents, who are also called Gliese. Not all Gliese welcomed this change; in particular, since the Queen is an ally of the humans, some of the Gliese have become the Queen's enemies and are seeking to kill her and her friends. You have to help the Queen save herself and the kingdom from this mayhem. You love CS4320 and are excited to apply CS4320 concepts in real life. You decide to use a Neo4j database to model genealogical and demographic information about the residents of Gliese and solve the Queen's problems.

To install Neo4j, go to: http://www:neo4j.org/download and follow the appropriate link for your operating system. Neo4j's documentation is well-written and complete: http://docs.neo4j.org/chunked/stable/index.html

Data Model

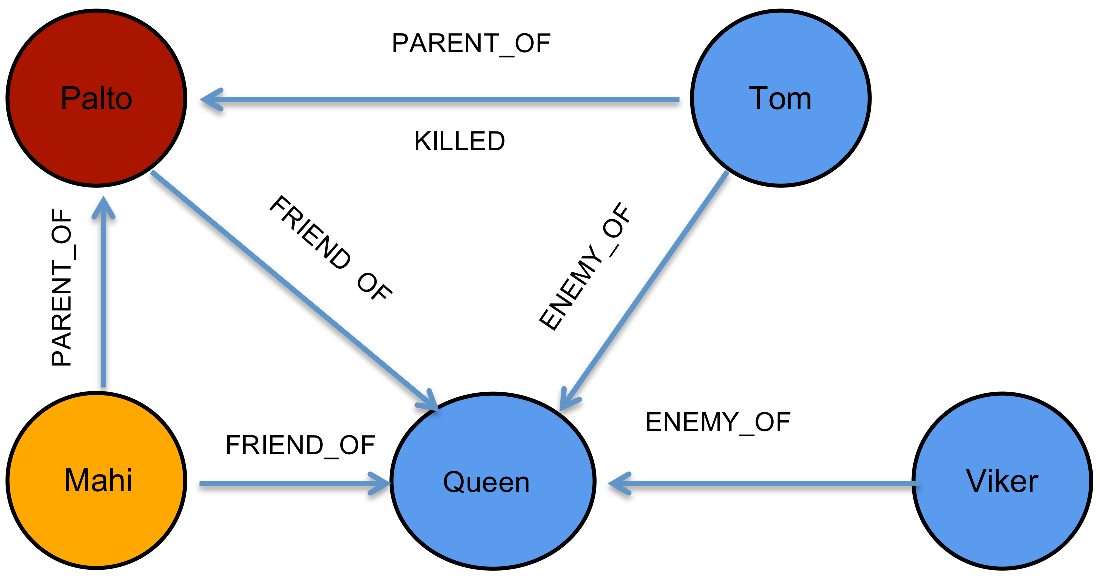
There are three types of beings on Gliese: humans, Gliese and "mixed" (beings with mixed human and Gliese ancestry) Thus, your data model will have three kinds of components:

1. Gliese with property *name*
2. Human with property *name*
3. Mixed with property *name*

Here is an example of a data set represented using this model:

```
CREATE (queen:Gliese { name : 'Queen'}),
(viker:Gliese { name : 'Viker'}),
(tom:Gliese { name : 'Tom'}),
(mahi:Human { name : 'Mahi'}),
(palto:Mixed { name : 'Palto'}),
(tom)-[:PARENT_OF]->(palto),
(mahi)-[:PARENT_OF]->(palto),
(palto)-[:FRIEND_OF]->(queen),
(mahi)-[:FRIEND_OF]->(queen),
(viker)-[:ENEMY_OF]->(queen),
(tom)-[:ENEMY_OF]->(queen),
(tom)-[:KILLED]->(palto);

match n return n;
```



Additionally, you may assume the following data constraints hold:

```
CREATE CONSTRAINT ON (a:Gliese) ASSERT a.name IS UNIQUE;
CREATE CONSTRAINT ON (b:Mixed) ASSERT b.name IS UNIQUE;
CREATE CONSTRAINT ON (h:Human) ASSERT h.name IS UNIQUE;
```

Here is some more useful information about the data model:

There are 0 or more humans, Gliese and mixed beings; one special being is the Queen, who is a Gliese. There is a PARENT_OF relation on nodes; every node has either two parents or zero parents (if it was a being whose ancestry is unknown/not recorded). Any human who has parents has two human parents, any Gliese who has parents has two Gliese parents. A mixed being has one human parent and one Gliese parent, or one mixed parent and the other parent of any type. Every mixed being has two parents (we assume we have full genealogical data from the time that the interbreeding began). No being can be its own parent, grandparent or ancestor, as time travel is not part of this universe!

Some beings have killed other beings, and this is recorded via a KILLED relationship. Any being (human, Gliese or mixed) can kill any other being, with the restriction that there are no suicides (no being can kill itself). Also, a being can be killed by one or more other beings.

There are two special relations FRIEND_OF and ENEMY_OF to keep track of allegiance to the Queen. The queen is neither her own friend nor her own enemy; any other being may be a friend of the Queen, or an enemy of the Queen, or both.

Finally, you can make the following assumptions:

1. The names of all objects are case-sensitive unless stated otherwise.
2. Any node representing a Gliese has the label "Gliese", representing a human has the label "Human" and representing a mixed being has the label "Mixed".
3. All Relationships are unique. That is, if a particular being killed another, then the relationship between them will only appear once.

What you need to do

You need to provide Neo4j code to answer some queries, as described below.

As you create data and test your queries, you may find the following command useful:
MATCH (n) OPTIONAL MATCH (n)-[r]-() DELETE n,r
This command deletes all the nodes and relationships and clears the database.

The queries you write must satisfy these constraints:

1. Your queries may only contain a single RETURN statement.
2. You may not modify the database. This includes, but is not limited to, using the keywords SET, UPDATE and DELETE.

Any queries that don't satisfy the above constraints will receive 0 points.

Query 1 (4 Points)

The Queen knows who her friends and enemies are but she wants to know if there is anybody who is both her friend and her enemy. List the names of all such beings. Order the names in ascending lexicographic order.

Query 2 (5 Points)

Sadly, the Queen's enemies have already started killing her friends. Your task is to find all enemies of the Queen who have killed at least one of her friends. For each such enemy, list the friends who were killed by them. Order the names of the enemies in ascending lexicographic order.

Your query output should have the following format:
{(Harry,(Alen)),(Tom,(Lina,Aish)),(Viker,(Eve,Seena,Akon))}, where e.g. Viker is an enemy of the Queen and (Eve, Seena, Akon) are the three friends of the Queen who were killed by Viker.

Query 3 (5 Points)

In retaliation to the events from the previous query, some of the Queen's friends are killing her enemies. Display the names of all friends of the Queen who have killed at least 3 of her enemies; for each such friend, give the count of enemies killed. The result should be lexicographically sorted in ascending order on the name and then in descending order of the number of enemies killed.

Query 4 (5 Points)

Since the mixed-ancestry beings are the ones that are in most danger, the Queen wants to get some more information about them. Write a query to display the list of mixed beings *who have grandparents* along with a list of their corresponding grandparents. Your query list should be lexicographically sorted in ascending order by name of the being. The list of grandparents for each being should also be lexicographically sorted in ascending order.

Query 5 (8 Points)

You now decide to investigate the full ancestry of all mixed beings. Write a query to display, for each mixed being, a list of the being's parents, and a list of all their *other* ancestors (excluding their parents). Sort the output in ascending lexicographic order by the mixed being's name.

Query 6 (5 Points)

Find the names of all the beings that have no parents and thus are at the top of the ancestral hierarchy. Sort the output lexicographically in ascending order.

Query 7 (8 Points)

The Queen wants to keep families together. Help her by writing a query to find the names of every being's full siblings. Two beings are full siblings if they have the same parents (both parents must be the same). Your output should have the format {(tom),(dick,harry)}, where Dick and Harry are Tom's full siblings. Do not output anything for beings that have no full siblings. Again, your output should be lexicographically sorted in ascending order by name.

Submission Guidelines

Submit two files, as follows:

Part 1 (Hadoop)

Submit a .zip file that contains:
src/ directory with all of your .java files ;
Makefile - the makefile that handles any new .java files that you may have added
README - optional if there is anything the grader needs to know about your submission

Part 2 (Neo4j)

Submit a single text file called queries.txt that contains all your queries. Make sure the queries are listed in the right order in your file, and don't forget to add semicolons to the end of every query. If you want to make notes to yourself, please do so using COMMENTS ("//"), to avoid breaking our grading script.