Cloud Computing (INGI2145)
Assignment 1: Persons of Interest

Due 6th November 2015 at 11:59pm

For your first homework assignment, you will implement an algorithm to find out Persons of Interest in a social network. A PoI is a person that is reachable through your connections that can introduce you to someone. Such a method is used by LinkedIn to suggest connections that can be used for getting in touch with someone. You will use this algorithm to find the connections that can be used to introduce two arbitrary people using an Orkut social network dataset. **Read this document carefully!**

1. The Persons of Interest algorithm

LinkedIn, a professional social network, aims to connect professionals with each other. One of the key features provided by LinkedIn to allow a **source** to find which of her/his connections can introduce her/him to a specific person, called **destination**.

In this project we will implement a similar algorithm to find a connection that can introduce you to a specific person. Consider the social graph, where each vertex represents an individual user and each edge, or 'link', represents the fact that user v_1 has listed v_2 as a friend. Note that the graph is undirected which means that the friendship links are reciprocal: if user A has listed user B as a friend, then user B has listed user A as a friend as well.

This algorithm will be an iterative one, with several rounds, depending on the degree of separation we specify between source and destination nodes. There are multiple ways to implement this mechanism and you will be free to use whatever approach you can think of to solve this problem.

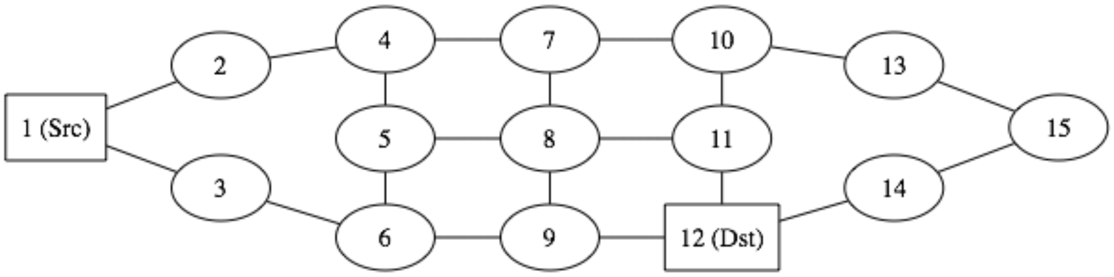
The algorithm should stop when either a maximum specified number of iteration has happened or a connection is found between the source and destination nodes *before* the maximum number of hops.

For our test purposes, due to lack of dataset from LinkedIn, we will use an undirected graph from a social network called Orkut (a discontinued social network owned by Google). The graph contains 3,072,441 nodes/vertices and 117,185,083 edges.

1.1. A simple example

Your algorithm will run on a data set that contains more than 3 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 on the right, which consists of just 15 vertices. Here we would provide three examples and the behavior of the algorithm in those specific cases.

Example 1:
In this example, *Node 1* is our source node (Src) and we want to find a node that can connect us to destination *Node 12* (Dst). We have the constraint that we should be able to connect to our destination node within 4 hops at max. Which means that the intermediary node should be 3 hops away at most.

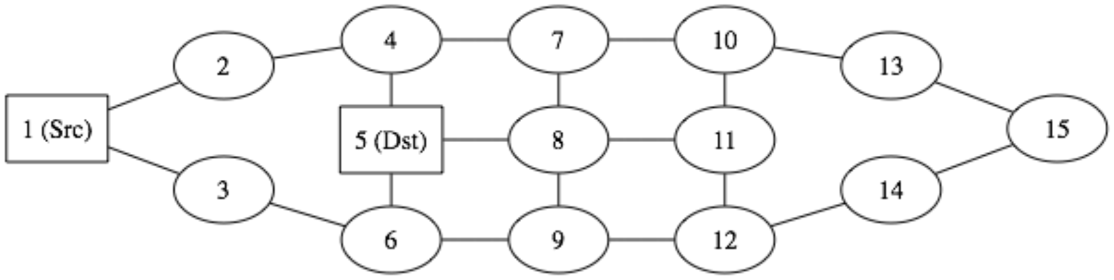


Given this criteria, the following is the possible path to connect to *Node 12* in 4 hops, following the path: 1 -> 3 -> 6 -> 9 -> 12

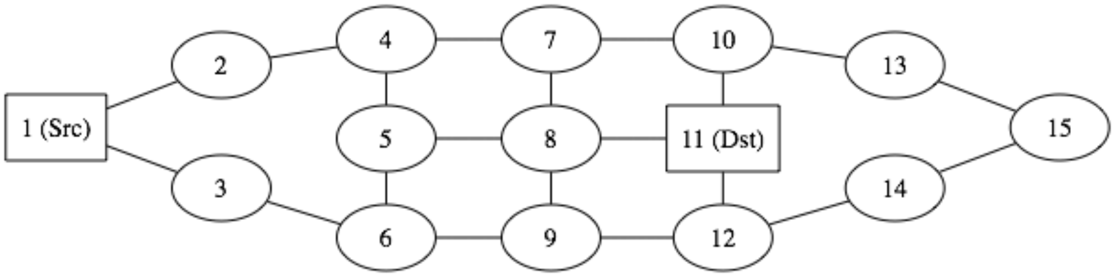
So, the **PoI** is the set of nodes that allow *Node 1* to connect to *Node 12* in 4 hops, which in this case is **[9]**. Note again that the output set will contain the nodes that are the hops prior to our destination node.

Imagine that the constraint in the maximum number of hops had been 6 hops. It might seem that our PoI set becomes [9, 11]. This is however not the desired output.
Instead, the algorithm should terminate as soon as it finds at least one PoI in a smaller number of hops. But keep in mind that there might be multiple PoI at the same small hop count, and not just one. In that case, we want all of them.
Let’s see another example.

Example 2:
In example 2, we have *Node 1* as source and *Node 5* as the destination. Here we have the constraint of finding a connection to *Node 5* in 3 hops. So the output of the algorithm, our **PoI**, will be: **[4,6]**. This is the desired PoI even if the maximum number of hops is greater than 3.



Example 3:
In this final example we want to find a set of nodes that can help *Node 1* connect to *Node 11* in 3 hops. In this case the algorithm should fail, returning an empty **PoI** set: **[]**.



1.2. Data format

Input format: In this assignment, 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. Note that each link is a bidirectional or undirected link. 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 6
4 5
4 7
5 6
5 8
6 9
7 8
7 10
8 9
8 11
9 12
10 11
10 13
11 12
12 14
13 15
14 15
```

Output format: The goal is to produce a file that contains the number of iterations that the algorithm ran and the set of PoI, that is, the connections that are friends of our destination node and our source nodes has a path to them. The data format should be, within a single line:
`<num. of iterations> + "[" + <comma separated PoI set> + "]"`
For the above examples, the output files will be as follows:
Example 1: `2 [9]`
Example 2: `1 [4,6]`
Example 3: `1 []`

1.3. Implementation strategy

We provide two implementation hints. First, it is **very important** to choose a suitable intermediate data format. For example, the data structure that you would create as the output of the initial map reduce job that takes the graph input and the format in which the data is output after each round of the iterative part. Clearly, the input graph needs to be converted into a data structure that will be used for iterative map reduce part. In addition, the intermediate format must preserve the data in the original input.

This type of algorithm is supposed to be implemented as an iterative process with each round as a separate map reduce job. 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 proceeding with the exploration of graph to find the path to the destination node, and one or more jobs for converting the intermediate format into the output format.

NOTE: In an efficient implementation, the number of iterations of the algorithm will be lower than the maximum number of hops given as argument. We will take this into account for grading. For our implementation, *the number of iteration it takes to find the PoI for a destination node that is **n** hops away is **n-2**.*

2. Implementing a MapReduce job for finding PoI

Your task is to implement a MapReduce job that implements the various sub-operations within the Person of Interest algorithm. 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> <srcID> <dstID> <#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> <srcID> <dstID> <iterNo> <#reducers>** This job should perform a single round of Person of Interest algorithm 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. The argument <iterNo> specifies which round of the algorithm is taking place, starting to count from 1.

- **evaluate <inputDir> <outputDir> <srcID> <dstID> <#reducers>** This job should read data in your intermediate format from `<inputDir>` (which is the output directory of the iterative job) and evaluate whether the algorithm should terminate or not. In addition, it should write the output of the evaluation in the output directory (use the final output format specified in section 1.2). **Note that this part could actually be implemented using multiple map reduce jobs depending on your algorithm implementation.**

- **composite <inputDir> <outputDir> <srcID> <dstID> <nHops> <#reducers>** This function is invoked only once and runs the entire Person of Interest algorithm from beginning to end, i.e., until at least a PoI has found or the maximum number of hops <nHops> has exceeded. It should run the init task, followed by the iterative phase (which might include or exclude the evaluate task). Once the algorithm finishes the output should be placed in the `<outputDir>`. During the iterative phase of the algorithm, the intermediate data should be stored in *temporary directories*, one directory per iteration.

Each job must delete the output directory if it already exists. The main class of your program must be `be_uclouvain_ingi2145_p1.PoIDriver`. It must output your full name to `System.out` every time it is invoked.

You should begin by checking out the program skeleton from the course's GitHub repository (https://github.com/mcanini/INGI2145-2015) in the folder `hws/hw1`.
The skeleton provides a class called `Utils` with several helpful functions to facilitate configuring and starting MapReduce jobs, deleting directories and checking whether a PoI has been found. However, note that you might need to adapt the checkResults function.
The **INGI2145-vm** is configured to run your program. Use `vagrant up --provision` to create your VM. You can then find the program skeleton in `/hws/hw1`.

2.1. Running your code on AWS with the Orkut data

We ask that you run your code on AWS on a snapshot of the Orkut social network taken from Stanford Large Network Dataset collection (https://snap.stanford.edu/data/). This data set contains about 117 million links between about 3 million users. It has already been split into multiple files and has been imported into Amazon S3. Specifically, the data is available on the `s3n://ingi2145-project1/` bucket, to which you have all been given read access. If you want to test your code locally, you can download the data (careful: it is over 1.7GB!) from the above bucket by running the following command:

```
aws s3 sync s3://ingi2145-project1/ <target_dir>
```

The following is a step-by-step guide for setting up your solution to run on Amazon's Elastic MapReduce.

- Go to the Amazon Management Console (https://mcanini.signin.aws.amazon.com/console/).

- Using the S3 Management Console (https://console.aws.amazon.com/s3/home), you'll need to create an S3 bucket **named after your AWS login** (`ingi2145-<username>`). This bucket will store input, output, MapReduce logs, and your exported JAR file for use with Elastic MapReduce.

- You will need to upload a JAR file containing your program to this bucket. You can generate the JAR file using the `./gradlew assemble` command in the project directory (the JAR will be created in the `build/libs` directory). You'll be asked to reference the location of this JAR file when creating your MapReduce job.

- 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 cluster". **Name your cluster(s) after your AWS login** (`ingi2145-<username>`). Keep the available default settings for the most part.

- Enable logging (select a suitable directory for the logs, e.g. `s3n://ingi2145-<username>/log`). Click on "Go to advanced options" and disable debugging, as it considerably impacts run time and provides little benefits (it indexes the logs for them to be searchable via the AWS CLI).

- Under the "Hardware Configuration" section, use **m3.xlarge** as the instance type and select the appropriate number of instances for your job. Use 3 core instances for the Orkut data.

- Under the “Steps” section, add a step by selecting “Custom JAR” and clicking on “Configure and add”.

- Under “JAR Location*:” Type in the location of your uploaded JAR file. If you followed the instructions, your JAR file should be inside the S3 bucket named ingi2145-<username>.

- You will need to supply appropriate command line arguments to your MapReduce job.

  For the small test scenario presented in section 1.1, they need to look like this:
  `composite s3n://ingi2145-<AWS username>/in s3n://ingi2145-<username>/out <srcId> <dstId> <nHops> <#reducers>`

  Note that `s3n://ingi2145-<username>/in` refers to the path where you need to store the test input data given in section 1.2, which you can also find in `test_inputs/graph1` in the skeleton we provide.
  Also notice that you don’t need to use a `s3n://` URL for the temporary directories, since we don’t want to save those results on S3.

  For the Orkut data, the parameters need to look something like this:
  `composite s3n://ingi2145-project1/ s3n://ingi2145-<username>/out 1 777 3 3`
  Notice that this time we use a different bucket for the input Orkut data. This will save each of you the pain of uploading a whole GB worth of data. Do also pay attention to the fact that the **source, destination nodes IDs and the maximum number of hops would be different in different test cases.**

While running on AWS, please pay close attention to the following considerations:

- **Watch your costs!** Keep in mind that AWS charges for machine utilization, data transfer and data storage. **You must release any resources when you are no longer using them.** Please be aware that the TAs’ solution took about 21 minutes to find the result for the example shown above. To conserve your AWS credit, only use the full Orkut data and the maximum number of machines after you are sure that your solution works on the sample graph with one machine.

- Due to limitations in AWS’ rights management systems, we can’t prevent you from seeing each other instances (Elastic MapReduce is backed by regular EC2 instances). **Don’t shut down other people’s instances!** We can track user actions on AWS, and abuses will be punished.

- You may of course have to run multiple clusters: they should all have the same name. Make sure that, by the time of submission, you collect results for the right one. To avoid mistakes, we ask that you report your Elastic MapReduce cluster ID in your report.

2.2. Debugging tips

- Try your code locally before trying it on AWS.

- Try the example from section 1.1 before running your code on the Orkut data (both locally and on AWS). The input file (shown in section 1.2) for the simple example is supplied with the project skeleton.

- Try each step individually before you try the composite task. A typical sequence of jobs would look like this (with the input data in a directory called `in`):

```
init in tmp1 1 12 1
iter tmp1 tmp2 1 12 1 1
evaluate tmp2 out 1 12 1
iter tmp2 tmp3 1 12 2 1
evaluate tmp3 out 1 12 1
```

- The skeleton project configures Hadoop to log both to the console and to a text file. Hadoop uses the Log4J framework for logging, which supports different logging levels. By default, we log messages with the `FATAL` level to the console, and messages with `INFO` level or higher to a file name `p1.log`. In any case, do not write to `System.out` in your final submission, except to write out your name as instructed above.

- On AWS, you can find log files in `s3://ingi2145-<username>/<log-dir-you-chose>/<job_id>/`. You especially want to consider the the log files under `steps/<step_id>`, where `<step_id>` is the ID of the MapReduce step. The file `syslog` includes the full MapReduce log, while `stdout` can be used to retrieve a subset of the log (there is code to configure this in the skeleton code we supply you).

- For reference, the solution of the TAs takes about 21 minutes to run on the Orkut data case (with 2 core instances and the test case for Orkut data given in section 2.1). Your solution may run faster or slower, but make sure to investigate if you encounter differences higher than a factor of 2.

3. Report

We ask that you submit a short report in addition to your code. This report should be **no longer than 2 pages** and should answer the following questions:

- A high-level description of your solution, and of the implementation choices you made.
- In particular, you should explain the intermediate representation that you used.

- Describe the challenges you encountered, if any (it's fine not to have anything to say, don't make up something).
- Output PoIs (according to the data format of section 1.2) that your solution found for *every* test case in the file `tests/orkut.txt`
- Your Elastic MapReduce cluster ID

4. Extra Credit

The following extra credit item will be available for this assignment:

- [0.5pt] Write an additional job (or jobs) to output the set of immediate neighbors of the source node that have been used to PoIs found by your algorithm. In your submission, include a file called friends.txt that describes how to run this job. For example, in the example 1 of section 1.1 the output of this additional job should be `[3]`. For example 2, it should be `[2,3]`. For example 3, it is the empty set: `[]`.

These points will *only* be awarded if the main portions of the assignment work correctly.

5. Grading

This assignment carries 3pt of credit out of 20. The following base policy will be applied to determine your grade:

- [1.5pt] if the solution runs correctly on the small example provided in section 1.1
- [0.5pt] if the solution works for the Orkut data on the test cases provided in `tests/orkut.txt`
- [0.5pt] if the solution works for the Orkut data using an additional set of test cases (unknown to you)
- [0.5pt] if the report includes all the requested information

If you only submit a partial solution, then we will award up to [1pt] for the effort depending on the quality of the code, documentation and design (as described in the report).
You should document your code properly. A small credit will be awarded for good documentation, and **no credit** will be given for undocumented code.
There will be **no extensions** to the assignment deadline. An (automatic) lateness penalty applies for homework turned in late: 10% is deducted on the assignment grade for each day late or fraction thereof. If an assignment is 1 minute late, it is one day late.

6. Submission

You should submit your solution as a single .tar.gz file via INGInious (https://inginious.info.ucl.ac.be/course/INGI2145) before the deadline. Access to INGInious requires a valid INGI or UCL account.
Note: You should submit only your source code and the report. Please don't submit data or JAR files.

Submission checklist

- ❏ In the file PoIDriver.java, be sure you wrote your full name and you output your name on System.out as instructed.
- ❏ Your code contains a reasonable amount of documentation.
- ❏ You have written a report with all the information requested at section 3.
- ❏ Your .tar.gz file is smaller than 150 KB. Please don't submit data or JAR files.