CS2043 Homework 4

Due: Friday, February 28, 2014 at 11:59PM EST

Important Note: The configuration of the CSUG Lab machines is our standard environment. Make sure that your scripts execute as intended in one of the CSUG lab machines or in a clone thereof installed on your computer as a Virtual Machine.

General Instrictions:

– You may form groups of **two** students to complete this assignment. Please form a group on CMS and submit only one solution per group.

– For each problem, you will write a script and save it to a file named with the problem label, e.g., **problem1.sh**. Assume that the input is in the same directory as the script. Recall that a `bash` script is a text file whose first line contains:

```
#!  /bin/bash
```

and an `awk` script is a text file whose first line contains:

```
#!  /usr/bin/awk -f
```

The preceding line is followed by the awk program. Here is an example of an `awk` script, called `example.awk`.

```
#!/bin/awk -f
BEGIN { print "example" }
{ print $8, $3}
END { print " - DONE -" }
```

An input file, say `input.txt`, should be passed to the `awk` script via the command line:

```
$ ./example.awk input.txt
```

– Once you complete the assignment, make a compressed tarball using gzip named `submission.tgz`, containing all the scripts you have produced. Submit your compressed tarball to CMS (`http://cms.csuglab.cornell.edu`). Your tarball should contain no directories.

Web Workloads

You are consulting for a bookstore that is planning to launch their online book purchase service. They want to understand the performance of their service when multiple customers are visiting the online store at the same time. They provided us with a log of transactions from a pilot they ran with 1,000 customers. The website designers received feedback from the users, saying that the service can be slow at times, which made them impatient to the point that they decided to leave the service before they could complete their purchases. This is an undesirable outcome as it results in significant sales losses.

We want to make changes to the website and then test the new version by submitting the same workload to it. However, the log is sorted by customer name and transaction type. We do not want to run the transactions in this order, as we want to simulate a more realistic scenario where different customers are performing different actions at the same time (the current order simulates one customer coming to the store at a time).

The log is available at `http://www.cs.cornell.edu/courses/cs2043/2014sp/log.txt.gz`

Each record of the log contains the customer's first name, the type of transaction he or she performed, and the time (in seconds) spent performing that action. The format is as follows:

```
customer transaction time
```

– **problem1.sh** : Write a `bash` script that randomly shuffles the lines of the file so as to produce the effect we want to simulate. Each time you run your script you should get a different order of the lines so that we can produce different transaction mixes to test the service.

(Hint: In `awk` the function `rand()` produces the same sequence of pseudo-random numbers each time we call it. We can change this behavior by seeding the generator with a different seed at each time. We can do this by calling the function `srand()` in the beginning of the program. It uses the current time as the default seed.)

Building Social Networks

We are going to analyze the restaurant data more deeply, as we now master more powerful tools that enable such analysis.

Recall that the restaurant data can be found at

`http://www.cs.cornell.edu/courses/cs2043/2014sp/restaurants.txt.gz`

(we have updated this file to remove parties of one person, so we recommend that you re-download the data).

A social network is a construct to study relationships between individuals, groups, organizations, or even entire societies. Social network models describe a social structure determined by interactions and enable the understanding of social phenomena through the properties of relations between individuals, instead of the properties of the individuals themselves.

In this assignment, we are going to investigate some forms of social networks. The first model is a network built on the concept of *affiliation networks*. The idea behind affiliation networks is that acquaintanceships among people often stem from one or more common or shared affiliations – living on the same street, working at the same place, being fans of the same football club, etc. We define a relationship between two people as their common affiliation to some entity. Therefore, we can model the network with two sets, one corresponding to the population of interest, and the other to the entities that they connect to. The entities do not have any connections among themselves. Moreover, people do not directly connect to each other. We establish the existence of a relationship between two people if both are affiliated to a common entity.

– **problem2.awk**: Write an `awk` script that builds the affiliation network of customers to restaurants. Each line of the output should have the following format:

  ```
  Restaurant:  customer1 customer2 ...  customerN
  ```

  where `Restaurant` is the name of each restaurant that appears in the data, and `customer1 customer2 ...  customerN` is the list of people who have eaten in the restaurant, according to our data, **without repetition**.

Another way to build a social network is by linking people in dyads using some definition of friendship. We define that two people are friends if they ever dined together at a restaurant. For reference, let us call this definition *dyad 1* (the number 1 refers to the fact that the two parties were seen dinning together at least once).

– **problem3.awk**: Write an `awk` script to build all the dyadic relationships between two people using the **dyad 1** definition. Each line of your output should have the following format:

  ```
  name1 name2
  ```

  where `name1` has been to a restaurant with `name2`. The dyads are unordered, so "`name1 name2`" and "`name2 name1`" are the same pairs and should appear only once in the output.

We can think of a stricter definition of friendship where we connect two people if they have dined together at a restaurant at least k times (where k is a positive integer). Let us name this definition **dyad** k. The rationale behind this definition is that people may dine occasionally with acquaintances, but the repeated observation of two people dinning together may establish a stronger relationship.

– **problem4.awk**: Write an `awk` script to build all the dyadic relationships between two people using the **dyad 3** definition. Each line of your output should have the same format as the preceding problem.

The degree of an actor is defined as the number of connections (or friends) it possesses.

– **problem5.awk**: Write an `awk` script to compute the degree of each node using the **dyad 1** definition of friendship. Each line of your output should be formatted as:

```
name degree
```

To check whether our previous solution is correct we can compare the number of dyads produced by **problem3.awk** with the sum of all the degrees produced by **problem5.awk**. Accordingly, each dyad contributes to the sum of degrees twice, once for each actor. Therefore, the sum of degrees should be twice the number of dyads.

– **problem6.sh**: Write a `bash` script to test whether your outputs produced by **problem3.awk** and **problem5.awk** match. The output should contain the number of dyads and the total sum of degrees (the latter should be twice the former):

```
number-of-dyads
sum-of-degrees
```

Finally, in Homework 2, we found who paid the bill in each group that went together to restaurants in Ithaca. Let's see how we can accomplish this more easily using `awk`. Recall that we're looking for the second to last person in the sequence of names of a group.

– **problem7.awk**: Write an `awk` script to list who paid the bill each time "Beula" went out to eat, and how many of these times she was the one who did. The output should be a string like:

```
Beula:  payer1 payer2 ...  payer n
Beula paid k/n times
```

where n is the number of times Beula went out, and k is the number of times she paid the bill. (**Note**: this is an `awk` script, so `grep` and `sed` are not allowed here.)