Network Analytics, Homework 0

Due: Sunday, November 3, 2019 at 9:00 PM, Shanghai Time

General Instrictions:

- For each problem, you will write the output and save it to a file named with the problem label. For example, for Problem 1, your filename should be **problem1.out**. If your file name deviates from this convention, our automated grader will skip your file without grading it.
- In your code, assume that the input file is in the same directory as your script.
- Once you complete the assignment, make a compressed zip or compressed tarball (if you are familiar with unix shell) named `submission.zip or submission.tgz`, containing all the scripts, answers, and output you have produced. Submit your compressed tarball to Gradescope. The entry code for this course is 97B277. Your compressed package should contain no directories.

Python Scripting: Six degrees of Separation

Six degrees of separation is the theory that everyone is on average six steps away, by way of introduction, from any other person in the world, so that a chain of "a friend of a friend" statements can be made to connect any two people in a maximum of six steps.

This theory was made popular by the game "Six Degrees of Kevin Bacon" where the goal is to link any actor to the famous actor Kevin Bacon through no more than six connections, where two actors are directly connected if they have appeared in a movie or commercial together.

In this exercise we are going to build an engine that returns someone's "Kevin Bacon Number" in Python.

0.1 Dataset

We observed 100 college students over a period of 2 years. We recorded 500 occasions in which students form parties to go eat in a restaurant in Ithaca, NY. Download the file `restaurants.txt` from the course website, which lists these events, one per line, where each line lists the names in the party and the associated restaurant. Each line has the following format:

```
name1,name2,...,nameN;restaurant
```

Each party size varies from 1 to 9.

0.2 Building social networks

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 using our restaurant dataset.

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.

- **problem1.out** (10 pts): Write python code 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 start by defining 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).

- **problem2.out** (10 pts): Write python code 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 (in networks language, we say that the relationship is undirected), 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.

- **problem3.out** (10 pts): Write python code 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.

- **problem4.out** (10 pts): Write python code 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 **problem2.out** with the sum of all the degrees produced by **problem4.out**.

- **question1.txt** (10 pts): Write in the first line of this file how many times each dyad contributes to the sum of degrees. How many times the number of dyads should the sum of degrees equals to? To test whether your outputs produced in **problem2.out** and **problem4.out** match your answer, write the total number of dyads and the sum of degrees in the second and third lines of this file, respectively.

0.3 Network Algorithms

We will break the task of find the network distance between two people into two simpler steps.

Step 1: An adjacency list representation of a network is a collection of unordered lists, one for each vertex in the graph. Each list describes the set of neighbors of its vertex. The main operation performed by the adjacency list data structure is to report a list of the friends of a given actor. In other words, the total time to report all of the neighbors of an actor is proportional to its degree.

– **problem5.out** (20 pts) : write Python code to build and output an adjacency list from the `restaurants.txt` file in Python using the `dyad 1` definition.

Here is an example. If your input, e.g., `restaurant.txt` file has the following lines:

```
A,B;R1
B,C;R2
C,A;R1
```

your script should produce the following "`Actor: Adjacent to`" lines:

```
A: B C
B: A C
C: A B
```

(Hint1: use nested dictionaries to build the adjacency list. For example, if "Alice" and "Hussam" are friends, then you would have the entries `{"Alice":{"Hussam": 1}, "Hussam":{"Alice": 1}}` in your dictionary. If your dictionary is called `friends`, then `friends["Alice"]["Hussam"]` returns the value 1 if Alice and Hussam are friends. Otherwise, either `"Alice"` is not a key of `friends` or `"Hussam"` is not a key of `friends["Alice"]`, or vice-versa, and it returns a key error.)

(Hint2: If you want to access `friends["Alice"]["Hussam"]`, remember to initialize both `friends` and `friends["Alice"]` as dictionaries before its first use so as to avoid a key error. It is easy to initialize `friends`, but it can be challenging to initialize `friends["Alice"]` as you don't know whether `"Alice"` will be in the dataset. You avoid this problem by calling the function `friends.setdefault("Alice", {})` before using the dictionary `friends["Alice"]`. This function does nothing if `friends["Alice"]` is already a dictionary, but initializes `friends["Alice"]` as a dictionary if it is not.) Another way to accomplish this is to use the `defaultdict` library of the package `collections`.

Step 2: Now, we are interested in computing the distance between a given actor and all other actors. To this end, we will use a network search algorithm called Breadth First Search (BFS).

BFS begins with a given actor, whom we call the root, and then inspects all of his or her friends. These friends have distance 1 from the root. Then for each of those friends in turn, it inspects their unvisited friends. These "friends of friends" have distance 2 from the root, and so on.

BFS can be implemented using an adjacency list and a queue. Here are the steps the algorithm performs:

1. Enqueue the root node, set the distance of root to zero, and mark root as visited.
2. Dequeue a node (let's call this node `n`).
   - enqueue all friends of `n` that have not yet been visited
   - set the friends distance to the distance of `n` plus one
   - mark the friends of `n` as visited
3. If the queue is empty, then the algorithm exits. Otherwise repeat from Step 2.

– **problem6.out** (30 pts) : write a Python function to compute the distance of all foodies in restaurants.txt who are reachable from a given root. The first argument of your function should be the name of the root, e.g., "Beula", and the second, the input file name that encodes de network.

Here is an example. If your `restaurants.txt file` looked like

```
Alice,Hussam;CTB
Harsh,Hussam;Subway
Harsh,Atheendra;CTB
Harsh,Sangha;Subway
```

Then, if you run your function with the root "Alice", your output should be (not necessarily sorted by distance):

```
Alice 0
Hussam 1
Harsh 2
Atheendra 3
Sangha 3
```

In this exercise you will output, for every person in the dataset, their average distance between themselves and all other actors. The format of your output should be:

```
actor1 avg1
actor avg2
...
actorN avgN
```

(Hint1: The restaurants.txt dataset contains a group of students who attend the same college in a small town area. Therefore, their distance to one another is expected to be small.)

(Hint2: Use a list to implement the queue. You might find these two functions useful.

- `list.append(x)` : inserts element x at the end of a list.
- `list.pop(i)` : returns and deletes element at index i from list.)

(Hint3: Use a dictionary to keep track of the actors that have been visited by BFS and their distance from the root, i.e., `dist= {"Alice": 0, "Hussam" : 1}`)

Important: After you finish this assignment, please fill out the course survey. The link is in the calendar section of the course website.