Computer Science 1 — CSci 1100
Lab $11\frac{3}{4}$ — Pagerank
Fall Semester 2014

Lab Overview

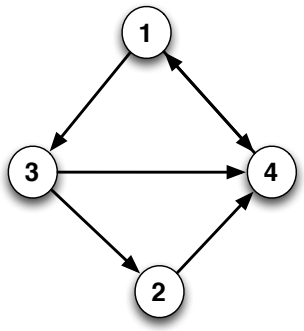
This lab is optional and somewhat longer than other labs. But, it provides a very good practice of many of the concepts that we learnt. You can get up to 4 extra checks by completing all the components of this lab. Try to get as far as you can.

In this lab, we still use dictionaries and sets, and you will implement `pagerank`, the method that made Google the success it is today. In the early days of Internet, it was a big problem to find "high quality" web pages in answer to a keyword search. Google's Sergey Brin and Larry Page, then graduate students in Stanford University, introduced a new algorithm called `pagerank` that can rank pages with respect to how good they were. And, they did in Python! They ran it from a small server in Stanford and they were an overnight success. In today's Google, pagerank is still a part of why some page comes up on top over the others, but many other methods are added on top. In this lab, we will use a graph of friendships to compute pagerank instead of webpages. But, the algorithm is the same.

Pagerank idea

Understanding the intuition may help with solving this lab, but it is not necessary. You can safely skip this section if you wish.

The idea of pagerank comes from social science and is also used very frequently analyzing who is very important in a group of friends. Think of a simple graph, like the one shown below.



In this graph, you have four people. If there is a link from person 1 to person 2, then it means that person 1 likes person 2. It is a directed link because friendship is not always two sided.

Suppose, now we want to find the most liked person. You will start from any person in the graph, and ask her who she likes. She will tell you all the people she likes. You choose one of them in random and go and ask that person. You will continue doing this and travel the graph. This is called **navigation**.

Every now and then you will get tired of asking and just choose a person in random. This is called the **random jump**.

Assuming you navigate some of the time and jump randomly the remaining time, the person you are most likely to be visiting at any point in time will have the highest pagerank. For example, in this example graph, it is clear that more links end up in person 4, so we expect 4 to have high pagerank.

An interesting part of pagerank is that if an important person is your friend, then you are also more important. So, most likely being friends with 4 will give a boost to 1 as well. Well, you will see whether this is true or not when you compute it yourself.

Pagerank assigns a floating point value (score) to each person in the graph. The higher your pagerank, the more important you are. In the next checkpoints, we walk you through how to compute it in detail. You will then complete the lab with a (hopefully fun) exercise.

Checkpoint 1: Reading a graph

To start this lab, download the file `extralab-files.zip` which will contain a number of text files such as `edge1.txt, edge2.txt, edge3.txt`. Each file corresponds to a graph.

For example, the file `edge1.txt` for the picture in page 1 has the following information:

```
1 3 4
2 4
3 2 4
4 1
```

which means that person 1 points to person 3 and 4, person 2 points to person 4, and person 4 points to person 1. Person 3 points to person 2 and 4. So, person 1 and 3 have 2 friends each while all the others have only 1 friend. We will refer to each person as a **node** from now on.

Read this file into a dictionary where each key is a person, and each value is the set of people the person points to. For the above example, you should get the following dictionary:

```
{1: set([3, 4]), 2: set([4]), 3: set([2, 4]), 4: set([1])}
```

To complete checkpoint 1, write a program that reads a file into a dictionary called `graph` of this form and prints the dictionary. Make sure you convert all the key and values to integers. Expected output for `edge1.txt` is shown below.

```
File ==> edge1.txt
4 people total
1: set([3, 4])
2: set([4])
3: set([2, 4])
4: set([1])
```

To complete Checkpoint 1: show your code and your output to a TA or a mentor. You will be graded both on correct output and the good program structure.

Checkpoint 2: Computing the pagerank of scores

Copy your code from check point 1 to a new file for check point 2. In this checkpoint, you will add score computation to your program. First, create a new dictionary called `scores` such that each key is a node (i.e. person) name and the value is a float `1/N` where `N` is the total number of nodes. For the above example, we would get:

```
scores = {1: 0.25, 2: 0.25, 3: 0.25, 4: 0.25}
```

This simply means that we assume each person has exactly the same score.

Now, implement the function shown below in pseudo-code. Your function will take as input a score dictionary like the one you just created above and the graph you read in check point 1, construct and return a totally new set of scores based on the old scores and the graph.

```
def get_newscores(current_scores, graph):
    new_scores = {}

    ## N = number of nodes in the graph

    for key in current_scores: ##initialize zero scores for everyone
         new_scores[key]= 0.15/N

    ## compute new scores for each node in the graph
    ## based on the scores in the dictionary current_scores
    ## and the graph (algorithm is given below)

    for each person p in graph:
        M = the number of people p points to
        for each person q that p points to:
            new_scores[q] += 0.85 * current_scores[p]/M

    return new_scores
```

To complete Checkpoint 2: after reading a graph, create a score dictionary as described earlier, implement the above function and call it for the initial set of scores and print out the scores before and after the call to the above function.

For the test file `edge1.txt`, you should get the following:

```
Filename ==> edge1.txt
4 people total
1: set(['3', '4'])
2: set(['4'])
3: set(['2', '4'])
4: set(['1'])
Initial scores
      1: 0.2500 2: 0.2500 3: 0.2500 4: 0.2500
Computed scores
      1: 0.2500 2: 0.1437 3: 0.1437 4: 0.4625
```

To complete Checkpoint 2: show your program and your output to a TA or a mentor. You will be graded both on correct output and the good program structure.

Checkpoint 3: Will it converge?

Now you have computed the scores for one time. Can we use the new scores to call the function again and again until we stop? This is very similar in essence to the Bunny and Fox problem. First, write a loop to call the function multiple times, each point updating the scores to be the scores returned by the given function.

In the same way as Bunny and Fox problem, we want to compute the scores until they converge. Here is how we define convergence in this case:

First, compute the pairwise difference between the old and new scores for each person.

```
>>>> diff = abs(0.25-0.25) + abs(0.25-0.14375) + \
   abs(0.25-0.14375) + abs(0.25-0.4625)
>>> diff
0.42500000000000004
```

then set a threshold value,

```
threshold = 0.00001
```

At the end of an iteration, compute the difference between the current scores and the scores returned by the function as shown above. If the difference is less than `threshold`, then we break out of the loop. Otherwise, continue repeating the loop with the newly computed scores.

To check the progress of your program, print out the `diff` values at each step. If they are not decreasing, then your program will never stop. This means you have a bug.

Once done, you should print the final scores for each person as before.

To complete Checkpoint 3, implement the loop that is described above for the given threshold and print out the final scores of each person. TAs will test your algorithm with different input files.

File `edge1.txt` converges in 27 steps for the given threshold, and produces the following output:

```
Filename ==> edge1.txt
4 people total
1: set(['3', '4'])
2: set(['4'])
3: set(['2', '4'])
4: set(['1'])
Initial scores
      1: 0.2500 2: 0.2500 3: 0.2500 4: 0.2500
Computing scores with threshold 1e-05
...
Iteration 26 (diff 0.000011):
      1: 0.3426 2: 0.1153 3: 0.1831 4: 0.3590
Iteration 27 (diff 0.000007):
      1: 0.3426 2: 0.1153 3: 0.1831 4: 0.3590
```

Interestingly, person 4 has the highest pagerank value, followed closely by person 1 as expected. Person 1 is really benefiting greatly from person 4's clout. Oh yes, the clout score is also computed very similarly.

To test it further, try setting the threshold even smaller and see that it still converges.

To complete Checkpoint 3: show your code and your output to a TA or a mentor. You will be graded both on correct output and the good program structure.

Checkpoint 4: Now for some fun

We will now do two things. We will first try the algorithm with much larger graphs. You have two graphs: `lesmis` is the graph of all the relationships in the book `Les Miserables` (imagine horrible singing by Russell Crowe) and `hero` is the graph containing all the heros in the Marvel universe. For each graph, you have the edges as before. Plus, you have a key of the names of characters in a separate file.

To complete this checkpoint, you must do the following. I will not give much detail here. You should be able to figure out these steps on your own. Take this as a test of how well your programming skills are progressing:

- Read the graph and compute pagerank as before
- Read the names corresponding to the graph
- Ask the user for a number (k)
- Find the people with the highest top k scores and print their name in the order of their score

To complete Checkpoint 4: show your code that finds and prints the name of the people with the top k pagerank scores in either file.

NOTE: We will not give any help on this last checkpoint. We will also not provide you with correct output. You must convince yourself that you are doing it correctly with proper testing.

NOTE 2: Both files with names are very large, so you better not print the scores in this checkpoint.