CMS/CS/EE 144 **Guru: Yujie/Xander**

Homework 2: Starting with a crawl

Assigned: 01/13/16 Due: 01/22/16 9:00am

We encourage you to discuss these problems with others, but you need to write up the actual solutions alone. At the top of your homework sheet, please list all the people with whom you discussed. Crediting help from other classmates will not take away any credit from you.

You may turn in a hard copy of your assignment at the beginning of class or email your assignment to cms144.caltech@gmail.com. If you turn in a hard copy, ***remember to keep a copy of your homework to use in your self-evaluation****. If you turn in an electronic copy,* ***be sure to submit it as a single file****.*

Start early and come to office hours with your questions! We also encourage you to post your questions on Piazza, as well as answer the questions asked by others on Piazza.

1 Getting to know *Erdős–Rényi* [35 points]

In class, we've talked a bit about the *Erdős–Rényi* random graph model and begun to explore how well it models the four "universal" properties that we've been focusing on. In this problem, you will revisit three of these four properties and prove some additional results.

Recall, the *Erdős–Rényi* random graph model creates an undirected random graph, denoted by $G(n,p)$, by considering n vertices and letting every possible edge between vertices occur independently with probability p.

(a) **Degree distribution:** Let D denote the degree of a random vertex. You saw in class that D is binomially distributed, i.e.,

$$P\left(D=k\right)=\binom{n-1}{k}p^k(1-p)^{n-1-k}.$$

Suppose that we let $n\to\infty$, keeping $(n-1)p$ constant, prove that the distribution of D converges to a Poisson distribution with parameter $\lambda := (n-1)p$, i.e.,

$$\lim_{n\to\infty} P\left(D=k\right)=\frac{e^{-\lambda}\lambda^k}{k!}.$$

(b) **Clustering:** Calculate the expected number of triangles, denoted $E[T]$, that $G(n,p)$ contains.

Prove that $E[T]$ has a threshold. Specifically, characterize the $p=p(n)$ such that $\lim_{n\to\infty}\mathbb{E}\left[T\right]=\infty$? and $\lim_{n\to\infty}\mathbb{E}\left[T\right]=0$?

(c) **Diameter:** Suppose $p\in(0,1)$ is held constant. Prove that the (maximal) diameter of $G(n,p)$ equals two with a probability that approaches 1 as n becomes large, i.e., prove that

$$\lim_{n\to\infty} P\left(diameter(G(n,p))=2\right)=1.$$

Note: This part is significantly harder than the previous two!

2 Working with real data [25 points]

We'll now look for first hand evidence of the 'universal properties' by studying a real world dataset. We'll work with a collaboration network between researchers working in the area of network science. This dataset was prepared by Mark Newman (http://www-personal.umich.edu/ mejn/netdata/), who is himself a prominent figure in this area.

In this network, nodes correspond to researchers, and there is an undirected edge between two researchers who have been co-authors on scientific paper. The original dataset contains 1589 nodes, but we'll work with the largest connected component of this network, consisting of 379 nodes. You can download a text file at the following URL, which contains the node adjacencies in JSON format, with the key as a node and the value as a list of its neighbors:

http://courses.cms.caltech.edu/cs144/homeworks/net_sci_coauthorships.txt

Your task: Plot the histogram, as well as a complementary cumulative distribution function (ccdf) of the node degrees. Also, compute the average clustering coefficient, the overall clustering coefficient, the maximal diameter, and the average diameter.

Note: It is okay to use existing network libraries (e.g., networkx for python) to do the calculations. If you do this, please specify the library and the functions you are using.

Bonus points: While it is fine to use a network library for the calculations, we do hope that (if you have time) you consider implementing some/all of the functions yourself (especially if you are not familiar with the algorithms underneath). To encourage this, we will award a few extra credit points for those who implement their own algorithms. If you do this, be sure to describe the algorithm you used, and its efficiency.

3 Before you can walk you must... [40 points]

A web crawler is a program that automatically traverses and collects the web pages on the Internet. Crawlers are widely used by search engines to find and retrieve what's on the web so that they can build an index of the pages for searching. However, they're also useful for many other reasons, e.g., for gathering data, for validating the links on a web page, or for gathering everything from a particular site. You are going to implement a very primitive web crawler to explore the web pages in the Caltech domain (URLs containing ".caltech.edu") and count the number of hyperlinks each page contains.

The basic idea of crawling: A web crawler starts by retrieving a web page, parsing the HTML source and extracting all hyperlinks in it. Then, the crawler uses these hyperlinks to retrieve those pages and extracts the hyperlinks in them. This process repeats until the crawler has followed all links and downloaded all pages. Therefore, a crawler requests web pages just as a web browser does, but it browses automatically, following all the the hyperlinks in each page. Although the idea is simple, there are a few issues that come up:

1. Many web pages contain links to multimedia/data files, the crawler should not waste bandwidth downloading these files.

2. Many web sites deliver dynamic content; the web page is generated dynamically based on a query string specified in the URL. A crawler can get trapped on such a site, since there are potentially 'infinitely many' pages. For example, on an online calendar, the crawler can keep following the links of dates/months and get trapped in an endless loop.

3. Given that the crawler has extracted a long queue of links to visit, which one should be selected to visit next?

4. Given that the crawler has seen a page already, when should it go back to revisit the page and check for new and changed links? (You can ignore this for this assignment.)

There are many other issues too, such as the robot exclusion standard, Javascript in web pages, ill-formed HTML and so on.

Your task: Your task is to write a crawler that, given a URL in the Caltech domain, retrieves the web page, extracts all hyperlinks in it, counts the number of hyperlinks, follows the extracted hyperlinks to retrieve more pages in Caltech domain, and then repeats the process for each successive page. During this process you must keep updating the number of hyperlinks on a web page and the number of hyperlinks which point to that page.

You may write your crawler in any programming language you prefer, but you need to turn in a program that the TAs can test on the Caltech CS cluster. You should feel free to use third party libraries or the code we provide (see below) to aid in retrieving and parsing the web pages. If you find you are having trouble with these parts of the problem, contact the TAs, since this is not meant to be the focus of the problem.

Instead, your main task is to code the traversal of the web graph. Specifically, you should design your own selection policy. (You can ignore the revisit policy for this assignment and visit each page only once.) You could do this using something similar to a breadth-first-search or depth-first-search, or something more sophisticated. A key component in this will be to ensure that you do not visit pages more than once and that you do not get trapped. (To prevent your web crawler from getting trapped you will have to deal with issues related to "Non-HTML pages" and "Dynamic pages.")

Important notes:

- If you search the web, you will likely be able to find source code for a crawler. We expect that you will *not* look at external source code when doing this assignment and that you will write your own crawler.
- Ensure that your crawler stays within the Caltech domain. Do not crawl library linked services (e.g. JSTOR, IEEE Xplore, Wiley Online Library, etc.) or systematically download academic journal articles. These are violations of the terms of service.
- An efficient crawler can easily be mistaken for a malicious denial of service (DoS) attack. Do not do DoS attacks on the Caltech servers! You might want to limit the number of parallel requests and the time between requests. For a discussion on web crawling etiquette, see http://en.wikipedia.org/wiki/Web_crawler#Politeness_policy.
- You should work on this problem individually. Feel free to discuss ideas with your classmates, but you should not look at any other students' code and you should write your code by yourself.

What you turn in: You should submit the following after crawling at least 2000 HTML pages in the Caltech domain starting from www.caltech.edu.

1. Your code.
2. An explanation of the selection policy you chose and its strengths and weaknesses.
3. Two histograms. One for the number of hyperlinks per page. One for the number of hyperlinks which point to each page.
4. Two complementary cumulative distribution functions (ccdf). One for the number of hyperlinks per page. One for the number of hyperlinks which point to each page.

5. The average clustering coefficient and the overall clustering coefficient of the graph. Treat the edges as undirected for these calculations.

6. The average and maximal diameter of the graph. Treat the edges as undirected for these calculations.

7. A comparison of the degree distribution, the clustering coefficients, and the diameters with those in Problem 2. Can you describe the similarities and differences regarding the "universal" properties between collaboration network and the web graph?

   If you calculated the maximal and average diameter using a different algorithm from that used in Problem 2, be sure to describe the algorithm that you used here.

Submit (1) via email to **cms144.caltech@gmail.com** and the other sections with the rest of your homework.

Helpful tips: A Python script to extract all hyperlinks in a web page specified by an URL is provided on the course page (http://www.cs.caltech.edu/courses/cs144/homeworks/fetcher.py). It is a very simple program based only on the Python standard libraries, so you just can install Python (http://www.python.org) and run it. However, the script was tested with Python 2.7.3 and might not be compatible with newer versions of Python. The code occupies about 100 lines, including lots of comments. Take a look at it even you are not familiar with Python. The script performs the following tasks.

1. Use the URL to make a connection and fetch the http header. If the Content-Type is not “text/html”, return “None”.

2. Retrieve the web page content.

3. Use an html parser to parse the page and extract all hyperlinks.

4. Refine the hyperlinks, e.g., convert relative URLs to absolute URLs, and ignore the parameters of dynamic pages.

If you use Python, you can import this file and simply call the function “fetch_links(URL)” to get a list (may be empty) which contains all hyperlinks in the web page specified by the URL. It returns “None” if the URL does not point to a valid HTML page or if some error occurred. If you use another programming language, you can still use this script by making a system call to execute it and piping the output to your program. The output is a list (may be empty "[]") of hyperlinks which looks like: "[’http://today.caltech.edu’, ’http://www.caltech.edu/copyright/’]". The output is “None” if the URL does not point to a valid HTML page or if some error occurred. Take C for example:

```
#include <stdio.h>
FILE* output = popen("python fetcher.py http://www.caltech.edu", "r");
// Read and process the output of the script ...
pclose(output);
```

An alternative to using the provided Python code is to use a library for the language you are using. It is very likely that you can find third party libraries written for your programming language to retrieve and parse web pages, for example:

- Libcurl in C to retrieve the web pages.

- Libxml in C to parse the webpages and extract hyperlinks.

- Mechanize in Perl/Python/Ruby to retrieve and parse the web pages.

For Java/Perl/Python/Ruby, you can even use the standard libraries to retrieve and parse pages instead of downloading more sophisticated/complicated libraries.