Yifei YANG, Shan LU
CS 161
Instructor Semih SALIHOGLU
Due August 8th at noon
Final Project

Project Part 1. (YANG)

I implemented Tarjan's strongly connected component algorithm in an iterative way. Since it does only one pass of the depth-first search on the graph, it is therefore two times faster compared to Kosaraju's algorithm. I chose to implement this algorithm iteratively because of the large size of the data input. A recursive approach would lead to stack overflow during recursion.

Tarjan's SCC algorithm creates a depth-first search tree of the graph, in which each strongly connected component is a subtree. The algorithm maintains a stack, and adds explored nodes into the stack during search. When it reaches a leaf node that has no successors, it starts to pop the stack and trace the nodes. While tracing the stack, the algorithm determines whether the nodes starting from the top form a single strongly connected component, in this way: define $u.index$ as the depth-first search node counter number for node u, and $u.lowlink$ as the counter number of the earliest node in the stack that u or successors of u can trace back to. By definition, $\forall(u \to v) \in E$,

$$u.lowlink = \begin{cases} \min(u.lowlink, v.lowlink), & \text{if } v \text{ has yet been explored,} \\ \min(u.lowlink, v.index), & \text{if } v \text{ is in the stack.} \end{cases}$$

So when $u.lowlink = u.index$, i.e., u and successors of u cannot trace back to nodes earlier that u, all nodes on the search subtree with u being root form one strongly connected component.

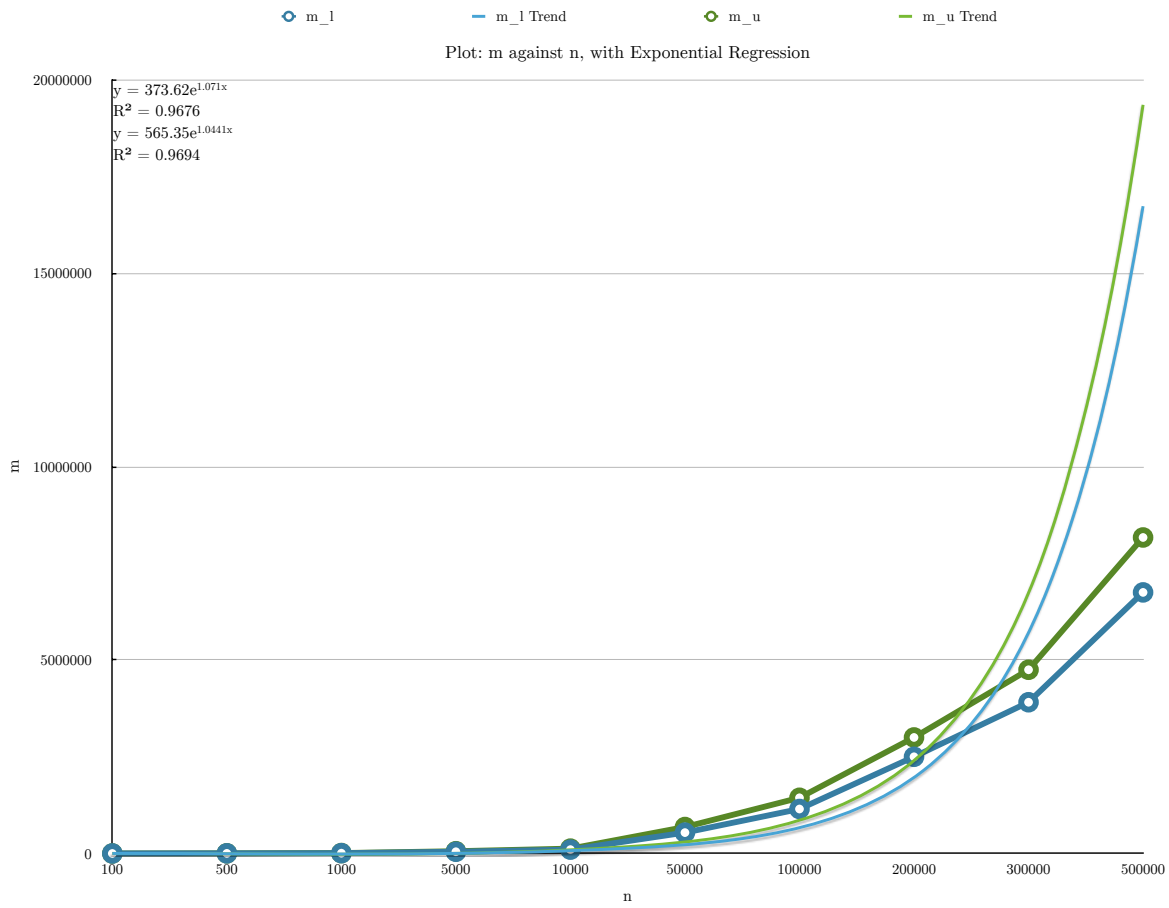
The algorithm explores each node exactly once, and each edge is also visited once, so the time complexity is $O(n + m)$.

In order to perform backtracking in an iterative manner, i.e., without the help of a recursion stack, I need to know the (immediate) parent of each node explored, as well as its visited ancestors that live on the stack. So in my implementation, besides `index` and `lowlink`, a `node` also has `parent` attribute, and a `bool` list called `visited` is maintained to track the nodes on the stack. Attribute `vit` of a node is used to iterate through the children of this node, as my graph is stored in the form of adjacency list. When this iterator reaches the end of its list, all neighbors of a node are considered visited, and the algorithm starts to pop the nodes in the same SCC from the stack. Lastly, the algorithm needs to find the earliest ancestor visited that are not in the same SCC, which is done by tracing the *lowlink* values.

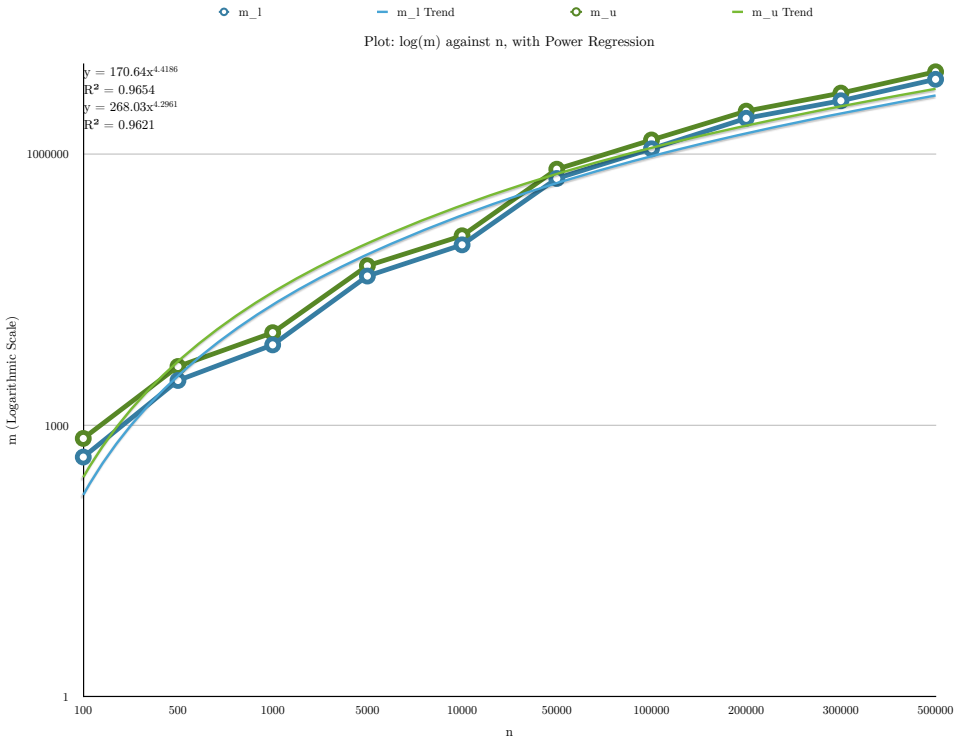
Project Part 2. (YANG)

n	m_l	m_u
100	448	720
500	3150	4500
1000	7800	10640
5000	45000	58500
10000	99650	125550
50000	540000	684000
100000	1150000	1440000
200000	2500000	3000000
300000	3910000	4750000
500000	6750000	8170000

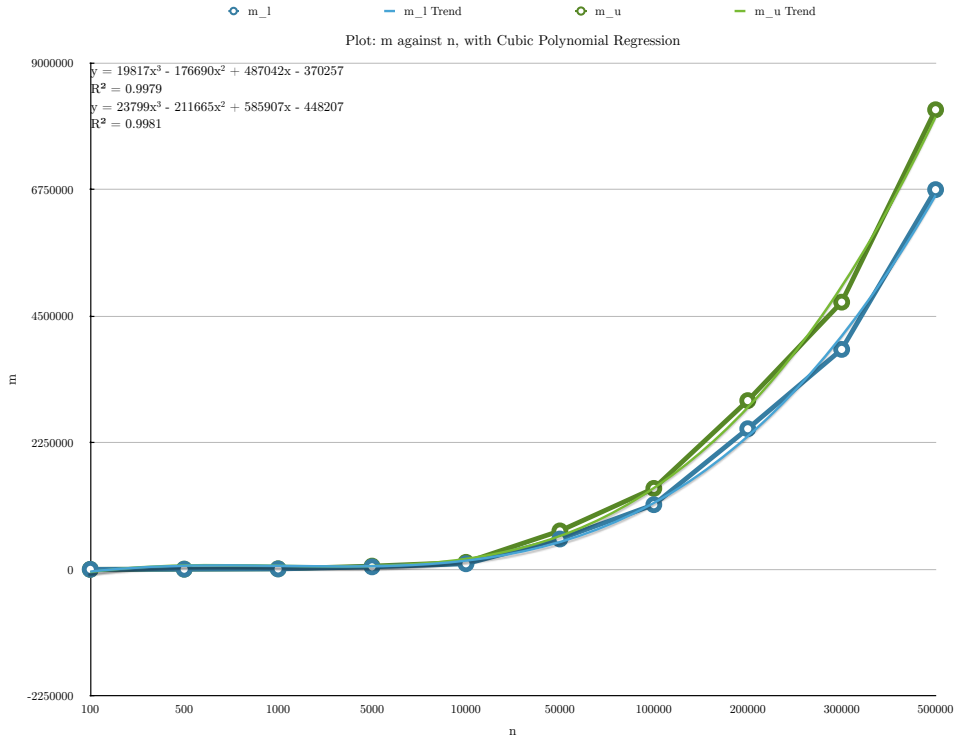
Here are some plots I made from the numbers:



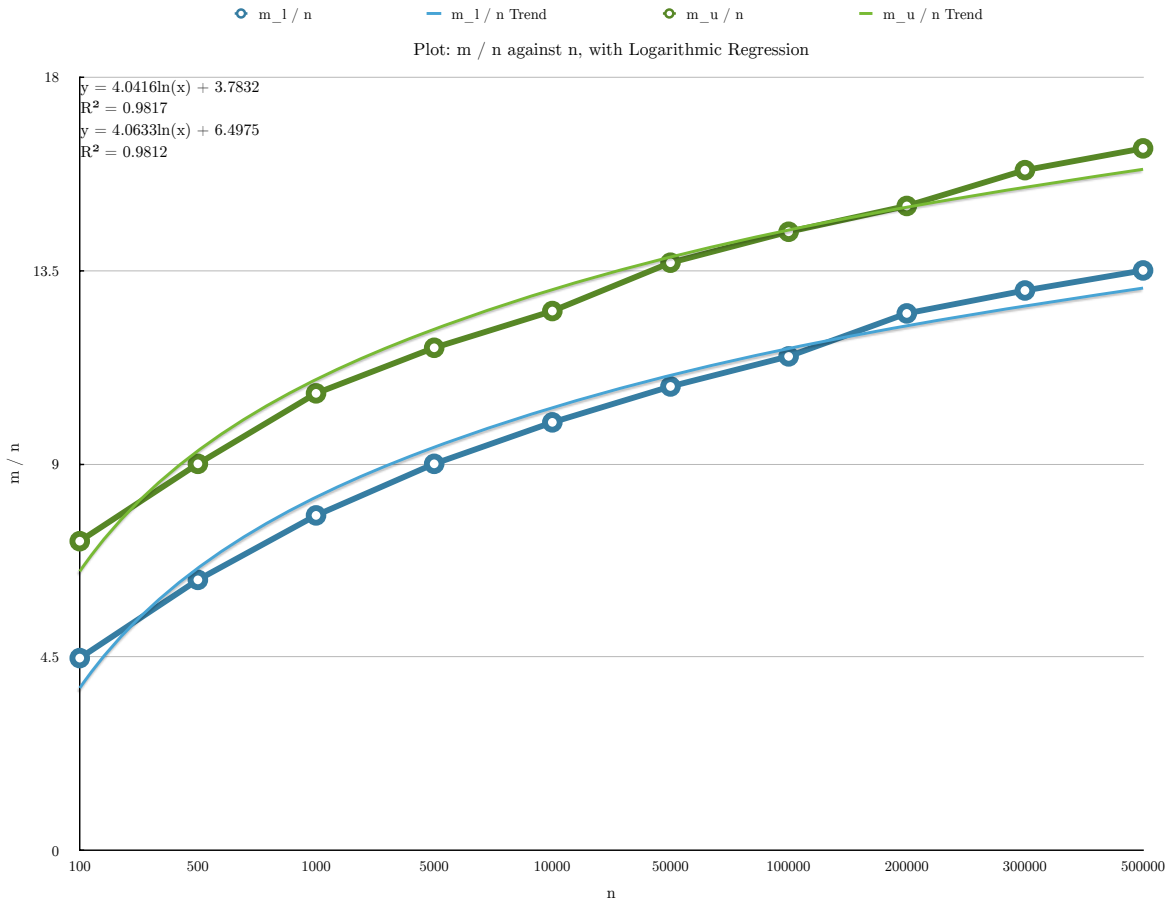
Apparently, an exponential relation between m and n does not fit much.



The R^2 values show that a power relation between $\log m$ and n is not confident enough.



A cubic polynomial relation may fit very closely, but since polynomial regression can be infinitely close as long as the level of the polynomial goes high, it does not depict the real relation between the two variables.



This is my final hypothesis:

$$n = k \ln \frac{m}{n} + c,$$

which yields

$$m = f(n) = \frac{n}{c_1}(e^n + c_2),$$

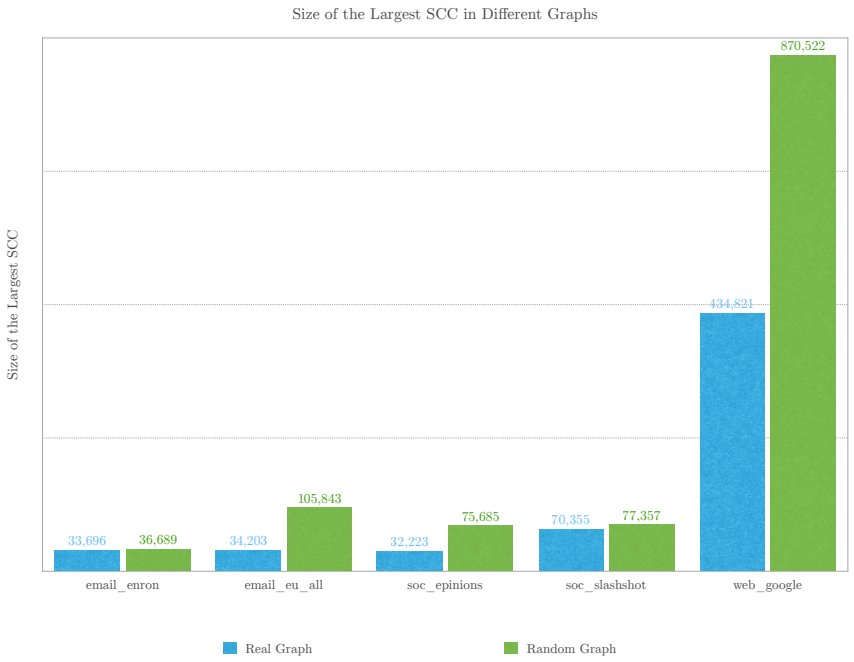
where c_1 and c_2 are constants determined by P you choose to test. This hypothesis is backed by the fact that the logarithmic regression model between $\frac{m}{n}$ and n has the highest R^2 value among all regression models I've tested on.

Project Part 3. (LU)

The chart for some basic statistics of each type of graph is as follows[1]:

	email_enron	email_eu_all	soc_epinons	soc_slashdot	web_google
n	36694	265216	75880	77362	875714
m	36766	418960	508839	828164	5105042
m/n	1.00	1.58	6.71	10.71	5.83
Confidence (Real/Random)	100%/100%	64%/11%	94%/15%	31%/2%	0.69%/0.11%

(1) Size of the largest SCC

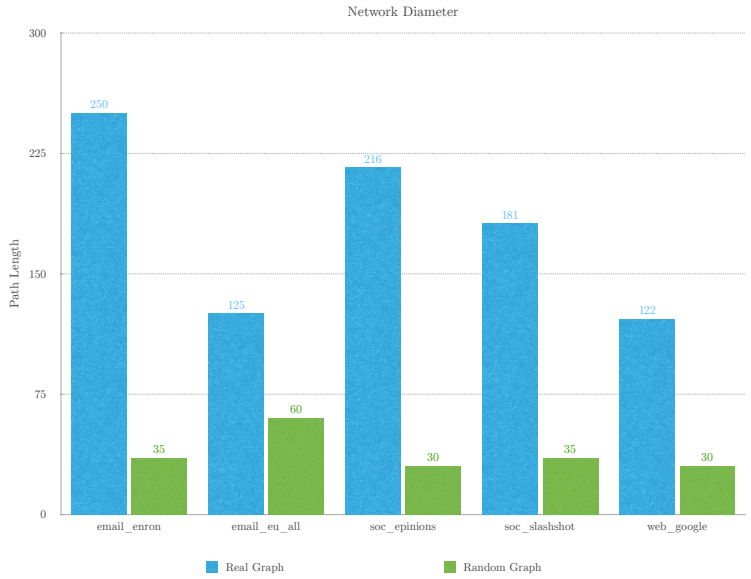


In this figure, each bar represents the size of the largest SCC for a given type of network. In general, this number is positively related to n. Comparing real world network to their randomly generated counterpart at the same scale, we find that the difference in size of largest SCC varies. Specifically, the largest SCC in real world web_google graph is much smaller than that in random_web_google graph. This can be explained by the fact that edges are generated more uniformly between vertices than the real world case. From a real world perspective, it is unlikely to expect bidirectional links between super famous websites like Google or Facebook and those nameless ones. Therefore, the size gap between SCC from web_google and random_web_google is huge.

Algorithm runtime complexity: $O(m+n)$

(2) Network diameter*

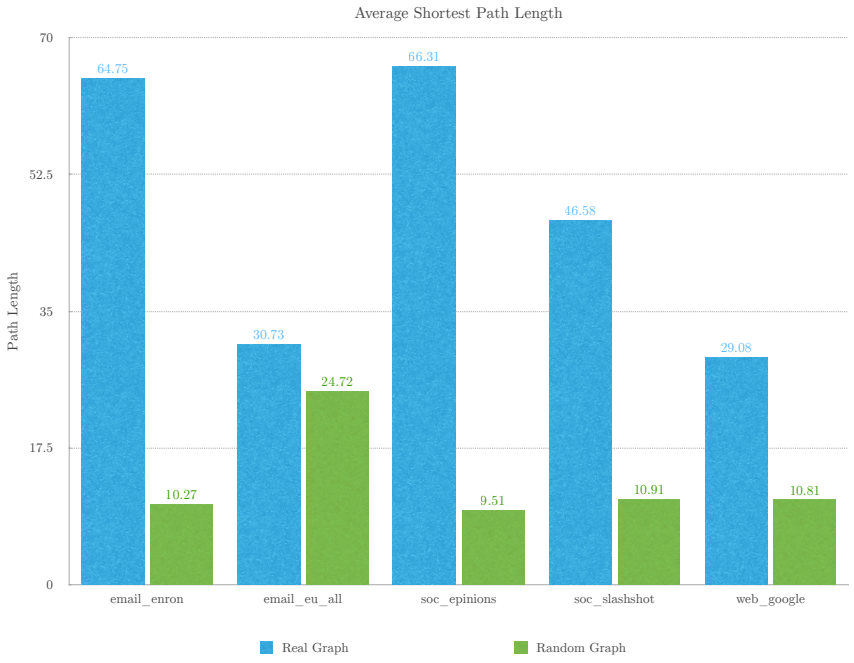
[1]Section titles that followed with a * indicates that metrics may not be globally correct. As the runtime complexity gets large, only partial results are available. In the table above, entry 94%/15% in soc_epinions column indicates that weve checked 94% and 15% vertices as sources from real graph and randomly generated graph, respectively.



Network diameter is defined as the maximum shortest path length between any two vertices in a given graph. In terms of real world graphs, this number is dependent on the extreme case where two edges are connected through a long path. Smaller E-mail communication graph(email_enron) outperforms than all others since real social network may find connection by building a long path between two individuals. However, the random_email_enrons network diameter is relatively small. This can be explained by the famous Six degrees of separation theory. Relationships in the randomly generated social network are more tight. In addition, in random graph cases, this metric is positively related to m/n, which indicates the number of edges each vertex has.

Algorithm runtime complexity: $O(n^3)$

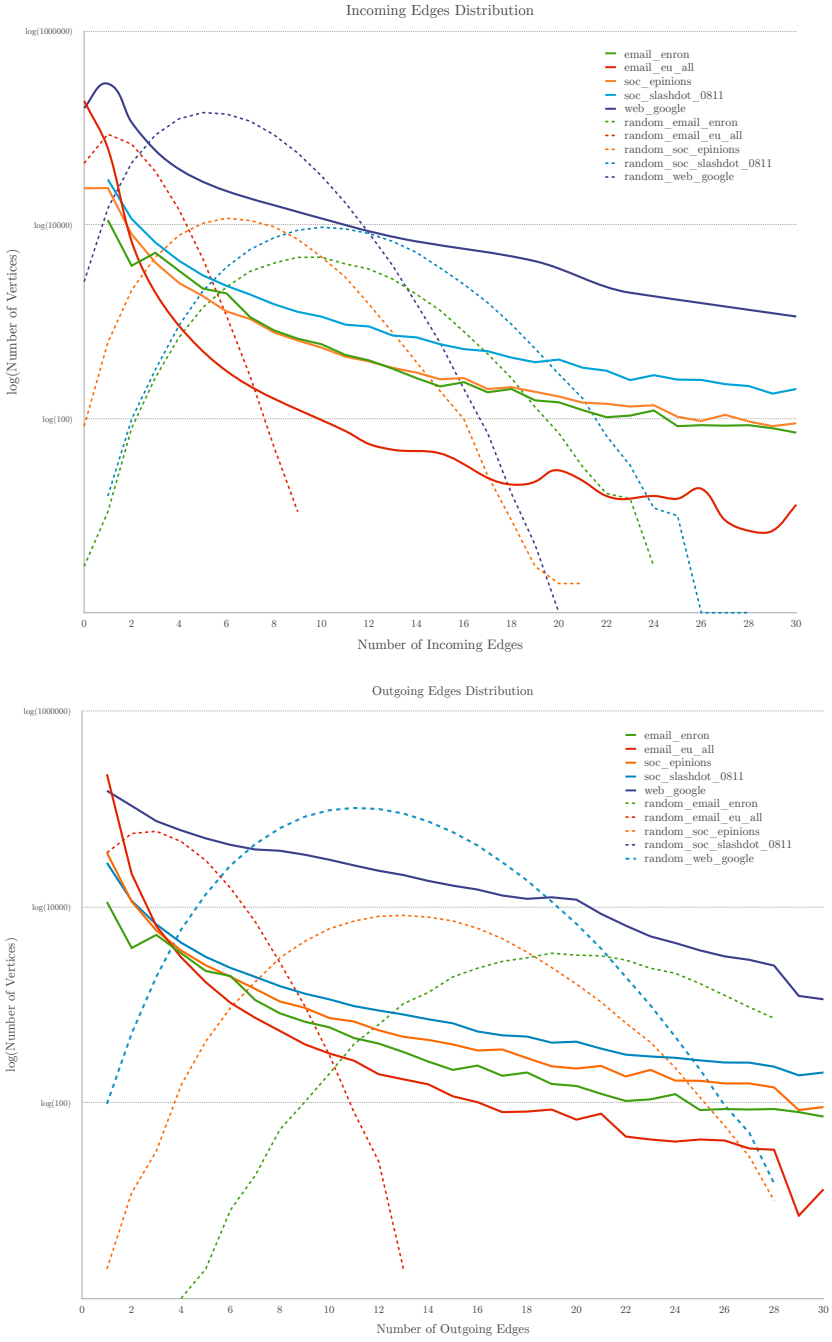
(3) Average shortest path length*



Although average shortest path length cannot be computed exactly due to the high complexity of our algorithm, we find that all 10 graphs average shortest path length have converged to numbers shown above.

Algorithm runtime complexity: $O(n^3)$

(4) Edges distribution (100%)



As shown above, full lines are almost convex while dashed lines are concave. Distributions of both incoming and outgoing edges in random graphs are more "normal" than their real world counterparts, showing that the majority(individuals or webpages) in real world is more lonely than in random world. Tailing phenomena are common in all real world graphs. In

reality, most webpages/individuals network is limited while there exists some "social stars" like rock stars or search engine websites.

Algorithm runtime complexity: $O(n + m)$, when we save the graph the way we want. Specifically, getting outgoing edges distribution is straightforward when adjacent lists is used in saving edges information. However, it is a little bit tricky to get incoming edges information with same data structure. The solution is to calculate the reverse graph of the original directed graph.

Project Part 4. (YANG)

Besides exploring the relationship between n and m, I learned a few other things as I was doing the second part of the project.

The first thing is shell scripting. As I needed to generate a bunch of random data and then call a program to process it, and this process would be run over and over again, I decided to write a shell script to automate the whole procedure. I have written shell scripts before, but only short and straightforward ones. The script I wrote this time is definitely the longest and most complicated shell script I have ever written. It's also uploaded so you could take a look. It basically handles everything: it asks the user for the values of n and m, as well as number of rounds and number of graphs per round; it compiles the program as needed; it drives the generation, maintenance, and renewal of random graphs; it calls my program to determine if a graph is singly-strongly-connected and collects the results; and it calculate the frequency of singly-strongly-connected graph. Later I updated the script so that it became able to tell the user about its current progress in every stage of processing. I think this script improved my productivity a lot, as it eliminated a lot of tedious work that I would have to do without this script. Although I spent a good amount of time to teach myself about scripting, I think this is definitely a worthy learning experience, as this is also a useful skill in computer science.

The use of a computing cluster. As the data size gets large, the computing power of the corn machine turns out to be insufficient, and I needed to exploit the power of the barley machine. I have used a time-sharing research computing cluster before, but I only ran scripts written by others, and have neither written a qsub script nor managed the resource allocation before. This time I consulted the online farmshare wiki a lot, and learned how to manage the queue, etc. Also, since I have written an automating script for SCC checking for problem 2, deploying such tests became really easy.

Use Unix low-level system call to expedite file input. As the size of data gets huge, the file-reading process became the speed bottleneck of my programs. So I decided to optimize the number-reading subroutine. I did some research and found using Unix system call would help. The basic idea is to open the input file as a whole, in the form of raw data. Next I use `mmap()` call to map this file (which now lives in memory as a block of data) to huge C-style character array. Then I go through every character in the string, and finally I convert meaningful digits into numbers. I did some testing on super-large data sets. With $n = 500000$, m around 10^7, one single random graph file is about 150 MB large. Traditional file reading using `scanf()` takes around 4 seconds to read in every meaningful number, and my new method takes about 0.97 seconds to read the file, scan the memory, and store every meaningful number into a container. Since I am always running my second program on large data inputs, I deployed this improvement, and my performance improved substantially. However, on smaller data sets, the time advantage gained from the new method turns out to be marginal, and the given input test cases have trailing whitespace characters at the end of each line, which requires extra attention from my new method. So I did not incorporate this code into my program for part 1.