A Command Line Tool for Network Construction and Metric Calculation

Esad Akar
Florida Atlantic University
Department of Computer Science, CAP 6315
Boca Raton, FL
eakar2012@fau.edu

Adam Lovett
Florida Atlantic University
Department of Computer Science, CAP 6315
Boca Raton, FL
alovett2014@fau.edu

***Abstract*—This paper outlines our development of a command line tool for network construction from user input or pre-existing text files. Our software, developed in C, provides a simple means of building undirected, unweighted networks of graphs, and allows for easy calculation of shortest paths, edge density, betweenness centrality scores, closeness centrality scores, and network diameter, among other network metrics, without the complicated syntax often involved in existing network metric calculation tools, which allows even novice users to easily create networks with little difficulty.**

We explore our development process, including the development of a vector data structure in C, the inner workings of the software we developed, including the operation of all major functions in the software, performance analysis of the software on a number of different network constructions, including networks of consistent edge density and variable network size and networks of consistent network size with variable edge denstiy, the scalability of our software, both for network size and edge density, the tools we used for testing, including two tools we developed for testing, and future developments we plan to implement to improve software performance and usability, including integration with other existing network metric calculation software and developments we plan to make in multithreading and memory management.

I. Introduction

The inspiration for our project was driven by two factors: one, both of our team members are primarily interested in software development rather than data science, so we felt it to be most valuable for us to work towards a software centric solution to a common problem, and two, a frequent issue encountered when attempting many problems encountered during an introductory course to social networks and their metric analysis — even on small networks, calculating most network metrics is incredibly time consuming, and on large networks, some metric calculations would be virtually impossible for a person in any reasonable amount of time if they could be calculated at all. Despite these complications, and the vast array of tools for both graph analysis and visualization [1], there are limited tools available for the novice data scientists for quickly and easily creating input networks and performing these calculations. Given our motivations and background, and the explosion of the field of data science in the current job market, we set out to design a tool which would allow for relatively easy text entry of networks, so as to be readily usable for beginners, that would then perform several of the tedious metric calculations on the networks, including counting the total number of nodes and edges, calculating the shortest path between any two nodes, including all shortest paths between the two nodes or just the first found, depending on what information the user needs, listing all the neighbors of a node, a check to see if any two nodes are neighbors, calculating the betweenness centrality score for any node, calculating the closeness centrality score for any node, calculating the diameter of the network, and calculating the edge density of the network [2]. The program supports only undirected, unweighted networks for calculation as of now [3]. Additionally, the program provides several other useful features for users after they have entered commands. When the program runs betweenness, our most computationally intensive function, there is a status bar displayed showing the user what percentage of the calculation has been completed. This is particularly useful when a user is performing operations on a very large network, as otherwise it may seem as if the program has hung in an infinite loop. Further, upon completion of any operation the program displays a run time for the operation in seconds, so the user is aware of how long tasks took to be completed, allowing for projection of how long operations would take to be performed on larger networks.

The program is implemented in C, due to its portability across many machines, stability, and time efficiency [4], with all data represented as node structs, stored in a hash table, with adjacent nodes stored in edge lists. Every metric function within the program is performed as a computationally optimized version of the traditional calculations one would learn in an introductory course. For example, diameter is calculated by finding selecting a random node, finding the longest shortest path from that node, then, choosing the terminal node of that shortest path, and finding the longest shortest path from that node, with that value accepted as diameter [2].

II. Related Work

Our software uses many existing network metrics that existed elsewhere; there is nothing technically “original” about calculating shortest paths, betweenness centrality scores, edge densities, etc. As such, there are many other existing network

metric calculation tools in existence today that calculate many of the same statistics as our software, and many that our software does not calculate. Among these is Gephi, an open source tool which allows for text file input or for in software entry for network generation, which then renders the network for viewing utilizing GPU power, minimizing the impact of the render on the CPU [5] and provides utilities for the calculation of metrics including node degrees, network diameters, edge densities, page ranks, network diameter, connected components, clustering coefficients, Eigenvector centrality, average path length, number of nodes and edges in a network, degrees, and clustering coefficients [1].

Another existing tool for network calculations is CFinder, which also uses text file input to find cliques within networks, using the clique percolation method [6], which detects communities by first discovering all cliques of a selected size in a network, then generating a graph of cliques by connecting all cliques which have one less node than the chosen clique size in common, with all cliques connected by an edge in the graph belonging to a community [7]. Although clique detection falls somewhat outside the scope of the simple metrics calculated by our software, it is equally important in social network analysis, and CFinder provides a means to perform these calculations when networks are too complex for manual calculation.

In addition, another tool for network visualization and metric calculation is Pajek. Pajek is designed specifically for use on large scale networks, such as airline networks or internet page links, and works by recursively reducing networks into smaller scale graphs which can be more readily operated upon computationally. It can then be used for calculating several network metrics, including max flow between nodes and shortest paths between nodes, among others [8].

Another, more advanced network analysis tool is StOCNET. StOCNET is only tangentially related to our work, as the calculations it performs go far beyond the scope of our software and types of information to be gleaned from the data. It is only mentioned here because it, uniquely compared to many other network analysis tools, makes use of adjacency matrices for its data input [9].

What separates our software from all the existing solutions is the ease of use for the novice user. Simplistic command line or text file entry of networks with minimal syntactic requirements provide a quick and easy way for beginning data scientists to perform complicated network calculations without having to invest significant time into the learning curve of complicated, GUI based solutions. Furthermore, while our software is portable to any type of system and free for anyone to use, Pajek requires Windows to perform, and CFinder requires a license for use, thus making our software a more accessible solution for the budding data scientist.

III. Body

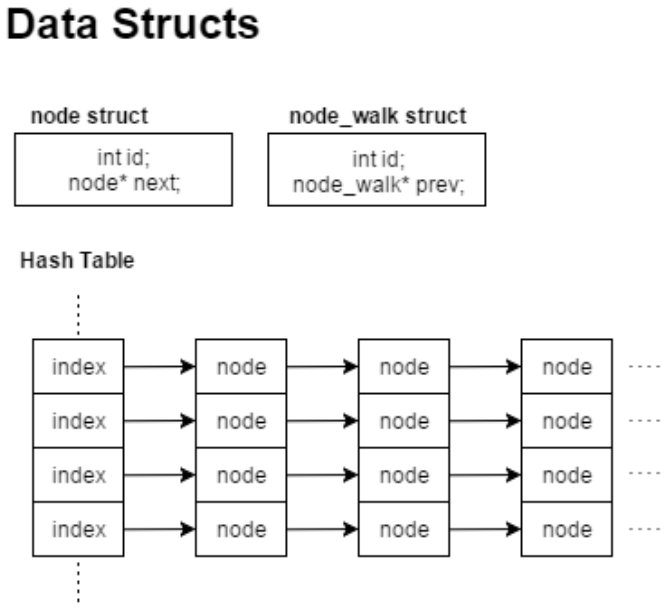
Because the purpose of our program was to provide a variety of network calculations as quickly and with as much scalability as possible, we chose to develop it in C. Though the additional libraries and tools provided in a higher level language such as C++ or Java would have been incredibly useful in development, easing the process of creating a number of our functions, the performance loss from the overhead of those languages would have made computation on larger networks untenable, defeating the purpose of the program.

As we chose to work with C, early on in the development process we did encounter an issue with variability of network size. When the program starts, there is obviously no way of being certain exactly what size network the user intends to create. Using too small a data structure would limit usability, as network size would be capped, however using too large of a data structure would use up too much memory when the user intends to use a small network, which could prove to be particularly problematic when running multiple instances of the program to run tests on many small networks. As such, we obviously needed a dynamically sized data structure; however as C does not contain any kind of variable sized container, such as a vector in C++ or Java, we had to create our own implementation of a vector. Essentially, when the vector container is created, a block of memory is allocated for the data. Once enough elements have been pushed onto the vector to fill up the memory allocated, the program looks for a new, larger block of memory and copies the entire vector into the new block. Though this does cause network input to be more time intensive computationally, the same would be true of a higher level language which had a vector built in as a data structure and this implementation allows us to take advantage of the computational benefits of C.

To compile the program a Makefile is included in the main folder. The compiler used in the Makefile is gcc. If Make is not available, the program can be simply be compiled with this line: *gcc -Wall -g -O3 -std=c11 -DDEBUG=0 main.c -o demo* .
The name of the executable can be changed by replacing “demo” with another name. If compilation is done within an IDE, simply include this definition line at the very top of the main.c file: *#define DEBUG 0* and the program should compile without any issues. Operation of the program is very simple, it is opened using a call from the command line — “demo”. Upon opening the program, the user begins by entering nodes represented by strings of any type (with a maximum character length of 32) for their network one at a time. The program connects consecutively entered nodes with an edge; in order to close a list of connected nodes and begin entering another section of the network not connected to the previous node, the user simply enters a period, and the next node entered will not be connected to the last by an edge. To terminate input, the user enters the end of file command for their operating system (CTRL-Z for Windows). Alternatively, if the user already has a network designed in a text file (which must conform to the same conventions as the text input from the console; one node per

line, periods used as delimiters), the program can load the file in as a network by using the flag "-f" followed by the name of the text file in the command line when making the call to run the program (i.e. "demo –f textFile.txt", a number of example text files suitable for our program are included with this submission). There are several other command line flags that a user can enter when loading a network from a file, including –p to print out all of the connections in the network, –b to print a visual representation of the hash table that stores the network, and –l to print the string representation of the hash table that stores the network.

When the program is reading input from a file or standard input, a simple prime multiplier hash function hashes the string into an index in the hash table and depending on the condition that the node is new or not, it is inserted at the end of a linked list of nodes that fall to the same index value. At the moment of insertion, an id is assigned to the node and the previous node that was inserted into the network are connected via updating both node's edge node lists with each other's node ids. Ids are inserted into edge lists in a way that the edge list always remains fully sorted. In early iterations of the program, search for node ids was performed using linear search, traversing edge lists until the node was found or the node was not found. In later iterations in the interest of efficiency, a binary search was implemented to replace the regular linear search, as a means of reducing the computational footprint of each command, which resulted in significantly faster computation on large, densely connected networks, as each node search runs in time log(n) rather than time n [10]. Once node insertion is finished, the user can metric calculation on the network. Two main vectors are globals: edge lists and node labels, and each is accessed by using the node's id as index values. Our data structure is represented visually as shown below:



Once the network has been loaded into the program, a list of commands available to the user is displayed. Commands which require a parameter are followed by an equals sign, with the parameter to be entered immediately following the command. These commands, parameter requirements, their operations, and internal functionality where nontrivial are as follows:

top=: This command requires a numeric parameter, i.e. "top=10", and returns the top n nodes, by degree, and the degree of those nodes. Because each node has an edge list [3] in the form of a vector of node ids, the top command simply picks the edge lists with the greatest lengths and sorts the values before printing the result.

total=: This command requires either e or n as a parameter, i.e. "total=n", and returns either the total number of edges or nodes in the network, respectively. As the network is built from the read input, at each unique node insertion, a total node counter is incremented. Total command simply prints that value.

shortest path=: This command requires two parameters, each of which must be a node in the network, i.e. "shortest path=the man", and returns the shortest path between those two nodes, including both the length of the path and all nodes on the path. Shortest path proved to be the most difficult function to implement properly. It is also used in commands diameter, shortest path all, betweenness, and closeness centrality. It was essential that not only would this command be able to print the length of the shortest path [3] between two nodes, but also print the nodes that lie in the path. Internally, an implementation of breadth search is used. A stack of node walk structs is allocated and given to the shortest path function. A node walk struct consists of a node id and a pointer to a node walk struct. When breadth search [3] is underway, if a discovered node is new, it is assigned to a node walk struct and its pointer is assigned to the node that discovered the new node. When the node the search function was trying to find is discovered, the function simply chases the pointers from finish to start to print the nodes that lie in the path and print the length of the path.

shortest path all=: This command is entered just as the "shortest path=" command is, with two nodes in the network required as parameters. The difference being that this command returns all possible shortest paths between the two parameter nodes, whereas the previous command only returns the first shortest path in the list. Operationally, it simply runs the shortest path function, but when the final is discovered, all remaining nodes that were inserted into the stack are analyzed to find other shortest paths of the same length.

neighbors=: This command requires a single node in the network as a parameter, i.e. "neighbors=the", and returns all nodes which share an edge with the parameter node. Neighbors prints all the nodes in the edge list of the provided node. The ids of the nodes in the edge list are used as indexes to access the global node labels vector so string labels, not ids, are printed.

count=: This command requires a single node in the network as a parameter, i.e. “count=the”, and returns the degree of the node. Functionally, it only prints the length of the vector of the provided node’s edge list so no excessive computation is done.

is neighbor=: This command requires two parameters, each of which must be a node in the network, i.e. “is neighbor=the man”, checks if those two nodes are connected by an edge, and returns yes or no. The function checks whether the id of the first provided node is in the edge list of the other provided node.

betweenness=: This command requires a single node in the network as a parameter, i.e. “betweenness=the”, and returns the betweenness centrality score for that node. All shortest paths between every pair of nodes has to be discovered first before betweenness scores can be measured. Therefore, shortest path all command is used build up a very large vector of all the shortest path possible between every pair of nodes. Because betweenness is the computationally largest command, a progress counter is printed at 1% intervals. Once the paths are discovered, the command then counts the number of times the id of the provided node takes places in the shortest paths, then the final measurement is printed. Because all shortest paths are stored in memory after the initial betweenness run, if the command is used again, the first part where all shortest paths are found is skipped and the command simply looks to find the node in the already existing vectors of shortest paths.

close cent=: This command requires a single node in the network as a parameter, i.e. “close cent=the”, and returns the closeness centrality score for that node. Closeness centrality is implemented using the shortest path function. The provided node is used to measure the length of the shortest path to all other nodes in the network. Once all the lengths are summed up, the closeness centrality score is calculated and printed.

diameter=: No parameter is required. This returns the diameter of the network. As the command is running, it prints out the longest shortest path that it has found so far and ends with the longest shortest path found in the network. To calculate the diameter of a network, a random node is selected and the shortest path to all other nodes is calculated. Using the longest shortest path, the node at the other end of path is used to calculate the shortest path to all other nodes. The length of the longest path is then the diameter of the network [2]. Essentially only two O(n) runs are needed to calculate the diameter of the network.

print all: No parameter is required. This prints each node in the network and every node which they share an edge with. This is the same functionality provided by the –p flag from the command line when calling the program.

edge density=: No parameter is required. This prints the edge density of the network. Because total node and edge counts are incremented at each unique node insertion at the start of the program, no excessive calculation is done to measure the edge density of the network. It is calculated by dividing the maximum number of edges in the network – $n(n-1)/2$ – by the actual number of edges in the network [2].

commands: No parameter is required. This prints the list of commands for the program.

clear: No parameter is required. This clears the terminal screen.

hash balance: No parameter is required. This prints the visual representation of the hash table storing the network. This is the same functionality provided by the –b flag from the command line when calling the program. It was written to let the programmer analyze the balance of the hash table, and give them feedback on how the prime multiplier and hash table size is performing.

exit: No parameter is required. Exits the program back to the command line.

IV. Experiments, Perfomance Analysis, and Scalability

Scalability of the software was of the utmost concern in our development process, because our tool would be useless if it was only capable of performing using “toy” networks. As such, every effort was made to maximize efficiency.

The following tables shows run times for networks of various sizes, increasing by 100 nodes per network, with network edge density held within 1 percent of 0.25. All computations were performed 10 times (using different parameters where applicable), with the average performance displayed:

Nodes/ metric	top	total	shortest path	shortest path all
100	0	0	0.0001	0.0001
200	0	0	0.0001	0.0001
300	0	0	0.0001	0.0001
400	0	0	0.0001	0.0002
500	0	0	0.0001	0.0002
600	0	0	0.0001	0.0002
700	0	0	0.0001	0.0002
800	0	0	0.0001	0.0003
900	0.0001	0	0.0001	0.0003
1000	0.0001	0	0.0001	0.0003
1100	0.0001	0	0.0001	0.0003

Nodes/ metric	neighbors	count	is neighbor	betweenness
100	0	0	0	0.0177
200	0.0001	0	0	0.1099
300	0.0001	0	0	0.3581
400	0.0002	0	0	0.9531
500	0.0002	0	0	1.8367
600	0.0002	0	0	3.2536
700	0.0003	0	0	5.315
800	0.0003	0	0	7.805
900	0.0003	0	0	11.1449
1000	0.0004	0	0	15.407
1100	0.0006	0	0	20.0343

Nodes/ metric	close cent	diameter	edge density
100	0.0001	0.0003	0
200	0.0001	0.0012	0
300	0.0003	0.0019	0
400	0.0004	0.0033	0
500	0.0005	0.0041	0
600	0.0009	0.0045	0
700	0.0012	0.0063	0
800	0.0014	0.0069	0
900	0.0019	0.0086	0
1000	0.0025	0.0101	0
1100	0.0028	0.0114	0

The following table shows run times for networks of 500 nodes, with varying edge densities, with the edge density of each network increasing by ~ 3 percent per network. Again, all computations were performed 10 times, with average performance displayed.

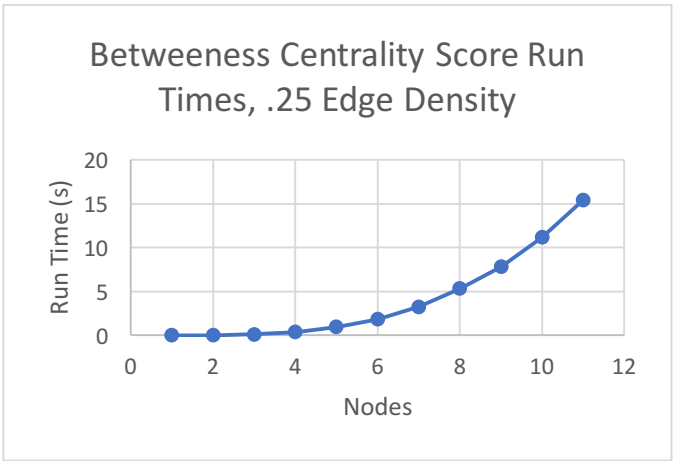
Edge Density/ metric	top	total	shortest path	shortest path all
0.0322	0	0	0.0001	0.0001
0.0605	0	0	0.0001	0.0001
0.0934	0	0	0.0001	0.0001
0.1203	0	0	0.0001	0.0001
0.1549	0	0	0.0001	0.0001
0.1818	0	0	0.0001	0.0001
0.2132	0	0	0.0001	0.0001
0.2424	0	0	0.0001	0.0001
0.2729	0	0	0.0001	0.0001
0.3051	0	0	0.0001	0.0002
0.3343	0	0	0.0001	0.0003

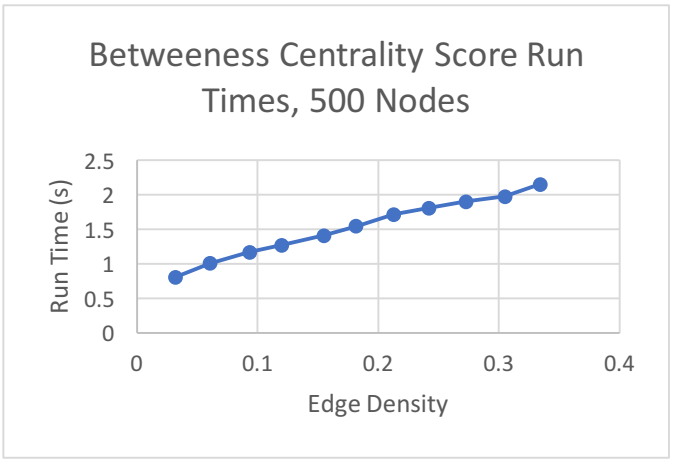
Edge Density/ metric	neighbors	count	is neighbor	betweenness
0.0322	0	0	0	0.8088
0.0605	0.0001	0	0	1.008
0.0934	0.0001	0	0	1.1707
0.1203	0.0001	0	0	1.2716
0.1549	0.0001	0	0	1.4087
0.1818	0.0001	0	0	1.5455
0.2132	0.0002	0	0	1.7156
0.2424	0.0001	0	0	1.8059
0.2729	0.0002	0	0	1.9017
0.3051	0.0002	0	0	1.976
0.3343	0.0003	0	0	2.1492

Edge Density/ metric	close cent	diameter	edge density
0.0322	0.0006	0.0046	0
0.0605	0.0006	0.0033	0
0.0934	0.0006	0.0031	0
0.1203	0.0006	0.0031	0
0.1549	0.0006	0.0036	0
0.1818	0.0006	0.004	0
0.2132	0.0006	0.0033	0
0.2424	0.0006	0.0034	0
0.2729	0.0006	0.0036	0
0.3051	0.0006	0.0036	0
0.3343	0.0006	0.0035	0

As demonstrated by the run times shown above, many of the functions in our program are scalable to networks with virtually any number of nodes and with any edge density with little to no loss of performance. Top, total, shortest path, shortest path all, neighbors, is neighbor, and edge density are impacted very little or not at all by the size or density of the network, with any performance loss from either being due strictly to the volume of data to be reported in some of the larger network cases. These functions are all, essentially, run in constant time, allowing scalability to a network of any size or density the user chooses.

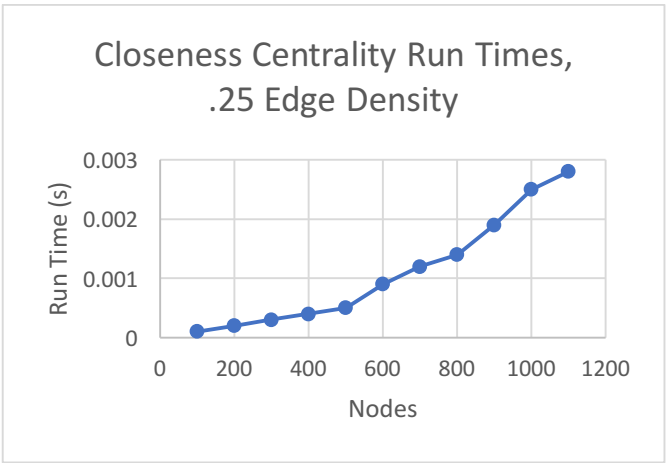
Betweenness centrality, as shown in the data, varies according to both network size and edge density. The following graphs illustrate the growth of run time vs. network size and edge density for betweenness:





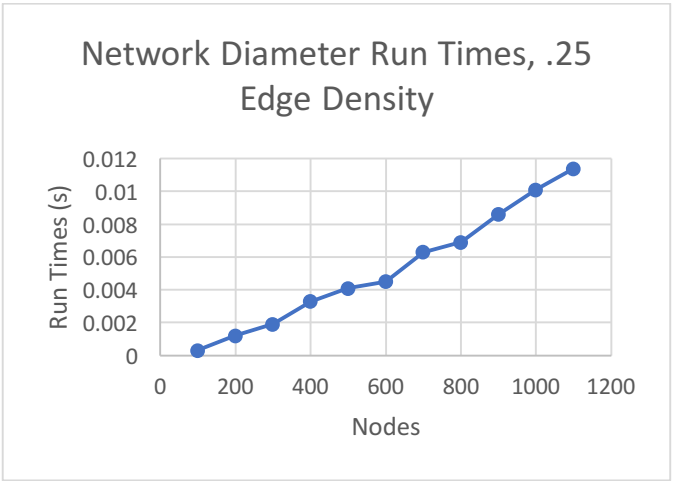
We see here that betweenness centrality score calculation run times grow exponentially versus network size, and linearly versus edge density. As such, our program scales for betweenness centrality score calculation easily for any edge density, so long as the size of the network is not too cumbersome. Although the growth rate for network size calculation is less than optimal, as when working with large scale networks even a factor of n^2 can be a hindrance to usability [8], we have combatted that to a degree by storing all data from the first calculation after it is performed. Thus, even though with a large network the first computation can be very time consuming, subsequent calculations on any other node are almost instantaneous. So, as long as the first calculation can be performed in a reasonable amount of time, there is scalability to almost any sized network.

Interestingly, we see from the data that closeness centrality score computation seems to vary entirely according to network size, with edge density having no impact on run time. The following graph illustrates the growth of run time vs. network size for closeness centrality:



This seems to grow non-linearly, but non-exponentially according to network size. While this is also less than optimal, we see that closeness centrality score run times are consistently less than run times for betweenness centrality scores, thus closeness centrality scales easily for any network where the program is able to reasonably calculate the betweenness centrality scores.

The final metric that shows any significant variation by size of network or edge density is network diameter, which only shows any significant growth in run time with variation in network size. The following graph illustrates the growth of run time vs. network size for network diameter:



Beyond runtime, the other issue with scalability in our program is memory allocation. While every effort was made to make our program as memory efficient as possible in our vector implementation, we ran into an unexpected issue with memory use in our testing phase. When running the betweenness function on a network of size 2000 with .25 edge density, malloc failed in allocating memory blocks at about 20% completion. Further testing with smaller networks of the same edge density failed until the network size was scaled down to only 1100 nodes at .25 edge density. This suggests that without further code optimization there is an upper limit on network

edges of ~ 150,000, as compared to software such as Gephi, which can work with networks of 1,000,000 edges [1]. Unfortunately, this limits the applicability of our program for real world networks, which obviously may contain millions, if not billions, of edges. As we did not discover this issue until late in our testing and development process, we were unable to address the issue beyond adding error handling to prevent system crashes in this iteration of the software.

In addition to our main program, we also developed two additional utilities to assist in the testing process. These utilities were both developed in Java, as they are less computationally intensive than our network program and the overhead loss from using Java was worth the additional utility provided by the libraries available to us. One is a simple text in/text out utility, which allows a user to input any text file; after the text file is inputted, the program scrubs the text file of all punctuation and changes all words to lower case, then outputs each word onto a new line in a new text file, which allows users to use any text file available to them as input for the program, making it much easier to use the software on a large variety of networks, with many different edge densities and diameters. This also allows for useful real world analysis of writing, by allowing for comparisons between the linguistic usages of various authors.

The second utility we developed was integral to our testing and optimization process. It allows the user to select a network size (in multiples of 100) and an edge density, and the program then generates a network of that size with an edge density within 3% of the target edge density (up to about .65, densities larger than this are unreliable due to the nature of the random number generator used to create networks), all while maintaining consistent network diameter. This tool works by creating a network of strings made of simple integers, from 1 to n, assigning each integer in the network a random number of neighbors as the network is being created, such that the average node degree of the network achieves a total edge density close to the density the user requested. The only drawback to our network generation tool is that it does not follow a power law distribution. Due to the nature of the random number generators available in Java, we were only able to generate networks which follow a normal distribution, which led to relatively small network diameters given the sizes of the networks; a tool to generate scale-free networks may have been more useful in assessing real world network applicability [2]. However, despite its limitations, the network generation tool allowed us to see exactly how computation time grew with network size growth, while holding other network variables relatively constant. As such, we were able to more thoroughly optimize our main program throughout development and see exactly how scaling works. Without this consistency in our test networks, it would have been virtually impossible to provide accurate data on scalability.

V. Conclusions

Our tool is has proven to be very useful for testing and development of small networks and network metric calculations. The ease of use for network creation and input compared to other tools available for similar calculations make it a potentially invaluable tool, particularly for students first learning how to work with networks such as these.

Because we feel that this project has the potential to be a utility which provides functionality that does not seem to exist elsewhere, and because we have learned so much in the process of developing it, we plan to move forward with more upgrades beyond what we have submitted for class. First and foremost, we will address the memory allocation issue that limits network size. As of this time, we are unsure if the issue is with the amount of memory available or address space, but it must be addressed in order to make our software functional for real world applications.

Next we plan to implement multithreading for the more computationally intensive functions. This will allow for faster calculation on networks with many nodes, making the program more practically useful on larger, real-world type networks (assuming the memory allocation issue has been resolved).

References

[1] I.-A. Apostolato, “An overview of Software Applications for Social Network Analysis,” *International Review of Social Research*, vol. 3, no. 3, Jan. 2013.

[2] X. Zhu, “Social Network Modeling and Analysis Part II: Social Network Measures”, Florida Atlantic University, 2017.

[3] X. Zhu, “Social Network Modeling and Analysis Part I: Graph Theory and Basics”, Florida Atlantic University, 2017.

[4] D. Ritchie, "The development of the C language", *ACM SIGPLAN Notices*, vol. 28, no. 3, pp. 201-208, 1993.

[5] M. Bastian, S. Heymann, M. Jacomy, “Gephi: an open source software for exploring and manipulating networks,” presented at the 3rd Int'l AAAI Conference on Weblogs and Social Media, San Jose, California, 2009.

[6] [CFinder] Clusters and Communities: Overlapping dense groups in networks", *Cfinder.org*, 2017. [Online]. Available: http://www.cfinder.org. [Accessed: 01- May- 2017].

[7] X. Zhu, “Social Network Modeling and Analysis Part IV: Community Detection”, Florida Atlantic University, 2017.

[8] V. Batagelj and A. Mrvar, "Pajek-program for large network analysis", *Connections*, vol. 21, no. 2, pp. 47-57, 1998.

[9] P. J. Carrington, J. Scott, and S. Wasserman, *Models and methods in social network analysis*. New York: Cambridge University Press, 2005.

[10] I. Flores and G. Madpis, "Average binary search length for dense ordered lists", *Communications of the ACM*, vol. 14, no. 9, pp. 602-603, 1971.