New Methods for Electronic Security

New Mexico

Supercomputing Challenge

Final Report

April 4, 2012

Team 90

Miyamura High School

Team Members:

Alanna Tempest

Joshua Tavares

Sponsoring Teacher:

Andrew Ng

Project Mentor:

John Donahue

Table of Contents:

- ■Executive Summary 3
- ■Introduction 4
  - ●Problem Statement 4
  - ●Significance 4
  - ●Background 5
  - ●Research 6
- ■Description 8
  - ●Methods 8
- ■Computational Model 12
- ■Code Development 14
  - ●Original Algorithm Code 16
- ■Results 17
- ■Conclusion 20
- ■Acknowledgements 22
  - ●Bibliography 23
- ■Visulizations 24

Executive Summary:

Our project's purpose was to look at Internet security in sending and receiving data. We looked into using Dijkstra's algorithm which maps out the shortest route between two points by searching for the shortest distance one step at a time. Unfortunately it requires complete knowledge of all the pathway possibilities and that is not always a possibility with the internet. Dijkstra's algorithm only works on networks set up through a service provider. The internet can be shut down by the service provider company and then our original idea would be obsolete. If there was a shut down then it would not be possible to send information.

Based on our research, ISPs are the safest way to send information, but information can be censored. Mesh networks can circumnavigate ISP shutdowns but are less secure. We did extensive research and have gained a lot of background information and with more time we could build off of what we know and create and test more scenarios and variables. If we did further study we would look into other ways of routing and would hopefully generate more data. There would be more time for trial and error tests and not as much research since we gained so much background knowledge.

Introduction:

Problem Statement:

Our project looks at information sent over the internet. There are many problems that can occur with the internet like having your mail hacked or having the internet shut down so you are unable to communicate with people who live miles, days or even weeks away. Information can be stolen and used, governments can censor what is happening in their country all through the internet. The internet is supposed to give freedom of expression and information that can be used by all people, safe and freely.

Our goal was to look at routing messages through mesh networks in an effort to analyze efficiency and security in order promote internet decentralization (by using mesh networks). Maximizing speed and security is important to any type of data transmission, therefore, we had to minimize the number of places in a mesh network that a message passes through.

Significance:

On the night of January 28th, 2011, after two days of internet-organized protests, the internet was shut down in Egypt. The people were using the internet to spread revolution and the Egyptian government had the internet service providers in the region shut down in an effort to subdue the rebels. The significance of our project is that knowledge and development of mesh networks will help promote internet decentralization and prevent the World Wide Web from being subject to top-down control through government censorship or outright shutdown.

Background:

The World Wide Web was originally created as a framework for giving many people access to internet. In its early stages, its topology was truly that of a web. The World Wide Web was designed to be decentralized as well as self-healing; if a router were to go down or its information was being blocked or censored, another path would be found around it. However, with the commercialization of computers and the internet, internet service itself became centralized for ease of availability. It was simpler to pay a large company to provide internet service than to set up the hardware. (Since then, the hardware has become simpler.) From that arises the current internet model: connection to the internet is obtained through Internet Service Providers, or ISPs.

ISP networks are run through companies. ISPs like Centurylink and Comcast provide and ensure security. Messages are sent from the node and travel through the service provider and then to the recipient. ISPs can though be shut down as seen in Egypt and in other Middle East countries in the past few years. Communication over the internet is almost all shut down and people are unable to communicate using the internet. A solution to an internet shut down is mesh networks. Mesh networks are direct connection between nodes, they do not have to go through any service provider but they are less secure. Since messages and information will not be going through a secure network it is easier for information to be hacked.

Research:

We began our research by determining how to transmit data through a network. We learned about the client-server model for standard data transmission, in which the server sends a message to a client that "listens" for the message. This is all done via one of these protocols: the Transmission Control Protocol (TCP), which establishes a connection between the client and server before the data is sent or the User Datagram Protocol (UDP), which sends information without establishing a connection. Both TCP and UDP sockets are built into Python.

We furthered our research by exploring the security implications of transmitting data from one machine to another. Secure Socket Layer (SSL) is a transport layer protocol that "wraps" ordinary client-server sockets to provide security. SSL utilizes public key cryptography, which provides for encryption protected against all threats except for brute-force or exhaustive key search attacks - and these threats are very rare because few people have the computing power to try every possible "key." SSL is a reasonable way to protect messages sent through mesh networks.

We employ graph theory to mesh networks. Graph theory emcompasses the properties and applications of data structures called graphs. Graphs are sets of nodes (or vertices) and edges, where edges are pathways that connect one node to another. One of the most popular types of graphs are the tree structures. Graph theory is applied to problems such as the "Chinese Postman Problem," which finds the shortest path that bypasses each of the required mailboxes in a city. We can also apply graph theory to mesh networks. Each device connecting to the wireless mesh network is easily called a node, and each direct wireless connection between two nodes is an edge in graph theory.

A type of graph generator to which our research led us was the Barabási-Albert model. This model is an algorithm for generating stochastic (random) graphs which are fairly representative of mesh networks. The algorithm utilizes preferential attachment, where edges attach to nodes based on how many edges are already attached to that node. Nodes of higher degree, or nodes that have more edges attached to them, are more likely to have more edges added to them. The result is a scale-free network, in which the degree distribution of nodes is according to a power law. Some nodes have many connections while others have few. This is representative of networks in general because the likelihood that a user will connect to a common site is greater than the likelihood that they will connect to an uncommon site. For example, someone navigating the web will probably navigate to Google or Facebook multiple times and will navigate to other, more obscure sites once or not at all.

We believe preferential attachment and the scale-free Barabási-Albert-generated graphs are representative of mesh networks as well. It makes sense: the chance that a node will be connected to a node with a great deal of existing connections should be high.

Connected to graph theory are common algorithms for searching graphs. Since our project can be represented by graphs, we can use these algorithms for finding nearest neighbors, minimum spanning trees, and shortest paths. Since we want to find the shortest path from one node to another, Dijkstra's algorithm seemed to be most appealing: it finds shortest paths. However, it finds the shortest path to every other node in the graph, and it cannot stop halfway through and always have determined the shortest path to the specified destination node. Dijkstra's

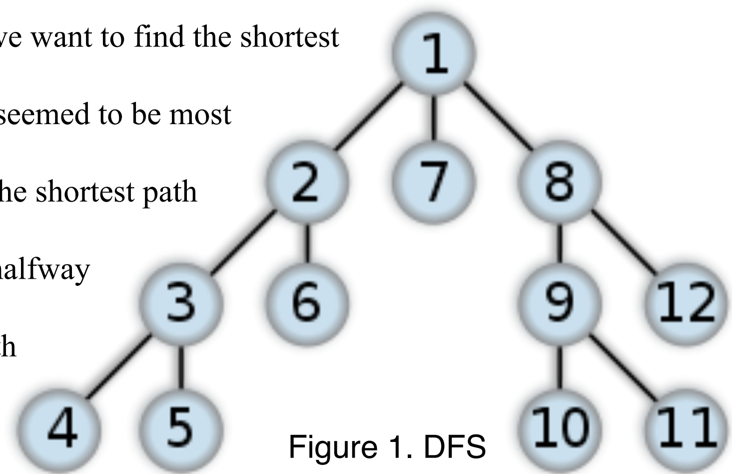

Figure 1. DFS

algorithm requires complete knowledge of the graph it is searching.

Depth-first search (DFS) is an algorithm for searching a tree or graph. It starts at an initial node and searches each path that extends from it until there is nowhere else to look. It then will back-track, returning to the initial node and then it will travel the best route. In the figure to the right, the nodes are numbered in order of how the depth-first search goes through them.

Breadth-first search (BFS) is similar to DFS in that it will explore every node directly or indirectly connected to the starting node. However, it searches differently. If a message started in the node labeled 1, in figure 2, it would start by look at all adjacent nodes (2, 3, and 4). Then it would search the children of (or nodes immediately below) node 2, which are 5 and 6. The numerical pattern of the tree is the order that the nodes are searched. Figure 3 shows how the message will start with the initial node at depth 0 and search each depth alphabetical order until it finds the destination or it has searched all nodes.

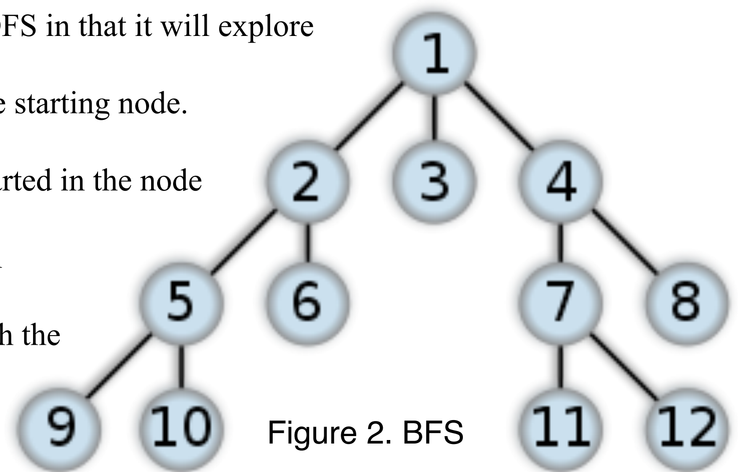

Figure 2. BFS

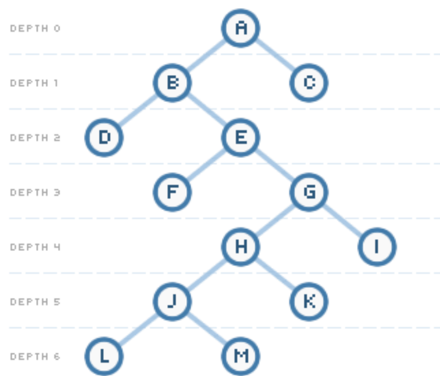

Figure 3. BFS Depth

Description:

Methods:

In order to minimize the number of places in a mesh network that a message passes through (in other words, find the shortest path through a mesh network), we used a breadth-first search algorithm on random unweighted graphs generated by the Barabási-Albert model. To simulate this and get a general picture of how the breadth-first search algorithm works on graphs of varying size and topology, we did many trials on many randomly generated graphs.

We elected to use unweighted graphs. Unweighted graphs are reasonably representative of mesh networks. A weighted graph is a graph that has values upon its edges. Weighting commonly represents distance or cost of travel. For example, an edge weight between nodes representing Los Angeles and New York City might be 2500, because that is the number of miles between the cities. Edge weighting is useful when considering that a route through one extra node (or city) may be much faster than another route. It is not useful in considering the weights of edges in a mesh network, however. This is because wireless mesh signals are sent via radio waves, which travel at the speed of light. Edge weights are therefore not useful. The most time consuming part lies in the network node itself, in receiving and sending the message, hence, the goal in routing messages should simply be to minimize the number of nodes a message passes through.

The Barabási-Albert algorithm for generating stochastic graphs requires inputs for the total number of nodes and the number of edges added with each node. We chose to have three edges constantly added with each node. While it is unlikely that each node joining a mesh

network is to establish exactly three connections, we believe that it is a good estimate. As mentioned earlier, this does not mean that every node is of degree three (or has three edges).

To portray different size networks we used graphs ranging in size from 20 nodes to 700 nodes. Since the graphs generated are random, we did five graphs per network size in an effort to eliminate possible bias. With each of the five graphs, we found the shortest path five different times, each time from random initial and destination nodes. For our trials, we used a random number generator to pick the initial and destination nodes.

It must be recognized that to send messages through a decentralized mesh network, an algorithm cannot rely on complete knowledge of the network. This is because a singular node in a mesh network only knows the existence of the nodes immediately connected to it. Also, nodes and connections constantly join and leave the network, so any supposed "complete knowledge" of a network can easily become obsolete. Therefore, we cannot use Dijkstra's algorithm.

We turn to the breadth-first search algorithm. We want an algorithm to quickly find the shortest path from one node to another in the mesh network. For unweighted graphs, the breadth-first search gives the shortest paths (or one of the shortest, if there are multiple paths with the same shortest length). The breadth-first search algorithm may not be the fastest algorithm for searching graphs because it tries all nodes in its path until it finds the destination node, but it is thorough and it is guaranteed to work. This algorithm was chosen over the depth-first search algorithm because it prioritizes nodes of low degree. It does this by searching closer destination nodes (or nodes of lesser depth) before farther destination nodes (or nodes of greater depth). The depth-first search algorithm searches first for nodes of the greatest depth, and can take much longer to find a destination node that is one edge away from the start node. Furthermore, the

depth-first search algorithm finds the shortest path through a tree but not a complex network graph. Therefore, we chose to use the breadth-first search algorithm.

Computational Model:

The breadth-first search algorithm is a search algorithm that requires an initial node, or root, and a graph to search. It can also accept a destination node. If only an initial node and a graph are given, the breadth-first search can return a spanning tree, or a tree connecting all the vertices of the graph. This is not what we want, however. If a destination node is also given, the shortest path from the initial node to the destination node can be found. This is what we want, therefore, our function requires an initial node, a destination node, and a graph. We began by programming the algorithm (see Fig. 4). After that, we used the Barabási-Albert-model algorithm that came with the NetworkX library, which generates stochastic graphs. Finally, we generated a variety of graphs and ran the breadth-first search with a variety of initial and destination nodes.

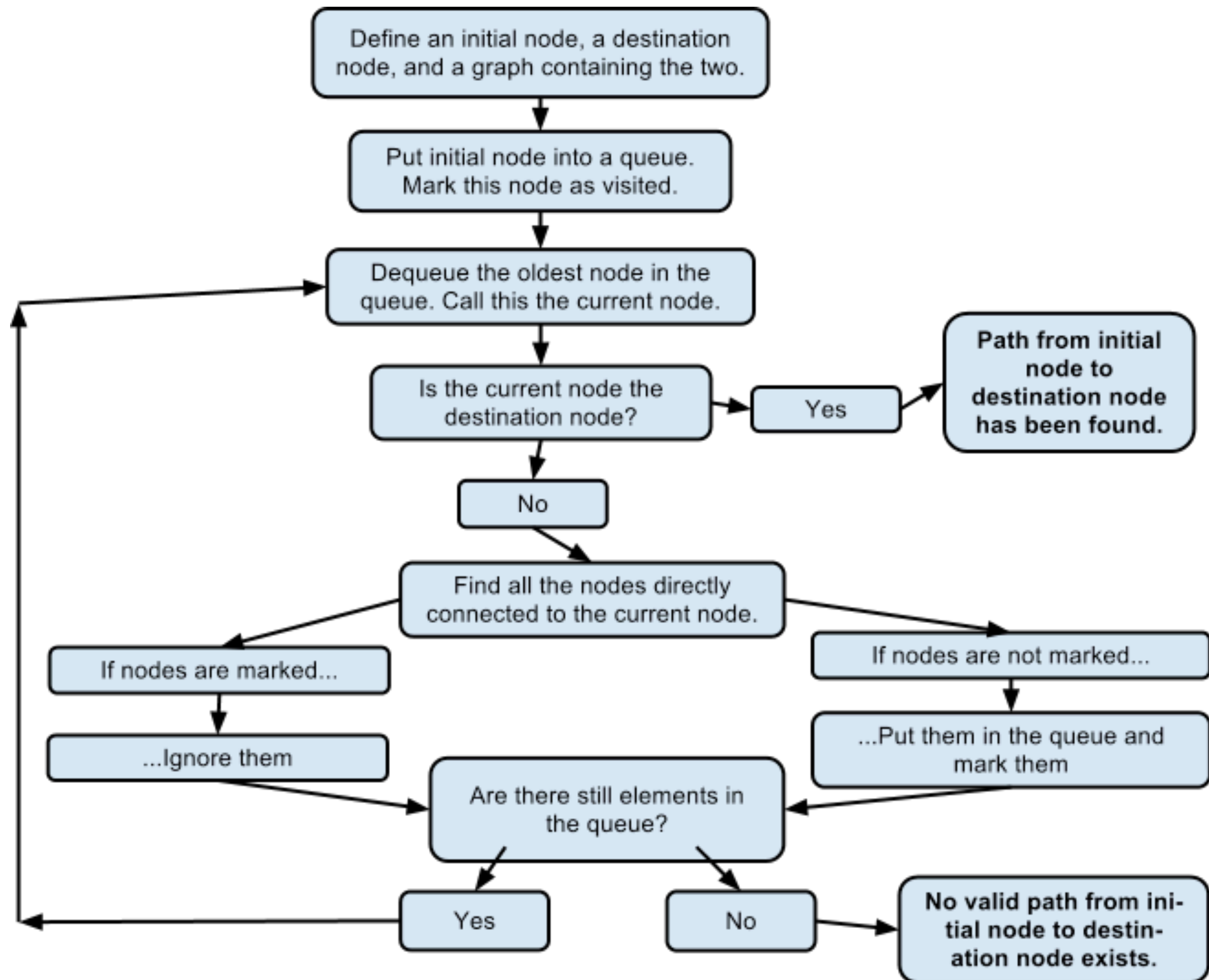

Figure 4. Breadth-First Search Algorithm.

Code Development:

We programmed our breadth-first search algorithm in Python. Python was chosen primarily because it is a good beginner programming language and because part of our team was already familiar with its syntax. As our project developed, Python revealed itself as an even better choice because of the amount of online support. Overall, the best part about using Python is the available graph packages. We began with a graph package called pygraph, which, as we experimented with it, revealed itself to be functional but not entirely efficient. For example, in order to make a graph, we had to add edges one by one, which was toilsome, especially with larger graphs. We did more research and found another graphing package for python, called NetworkX. NetworkX provides classes, or pre-programmed sets of commands, for graphs and digraphs (directed graphs). Most useful to us were NetworkX's graph generators and its ability to draw graphs. Drawing graphs in NetworkX requires another Python package called Matplotlib, a 2D plotting library, which we installed as well. Visualizations of graphs are extremely beneficial, especially in graphs with many nodes; the confusing computer representation of graphs consists of organized but long lists of edges and node names.

The original code for our algorithm is on page 16.

The initial programming of our algorithm involved the following:

Python contains a class called collections.deque(). "Deque" is short for "double-ended queue." This class is a data structure in Python that is similar to a list. Lists in Python are simply lists. Common operations on lists are pops and appends. "Pop" pulls an item in the list out of the list. "Append" adds an item to the end of the list. Pops and appends are efficient on the right side (or at the end) of the list, but are not as efficient on the left side. The deque class is designed for

pops and appends on both sides. This is important because we appended new elements to the right side of our queue and popped from the oldest elements on the left side. The functions we used for this (which are part of the deque class) were append() and popleft().

This code is a clear, simple coding of the breadth-first search algorithm. However, it does not provide very useful output. The function needs to print the shortest path from the initial node to the destination node. In order to do this, a way to store nodes as the graph is traversed must be established. This is done through dictionaries. Dictionaries are a built-in way to store data in Python. Dictionaries are lists of key:value pairs, where the command “dictionary_name[key]” returns “value”. Essentially, data used to return the shortest path needs to be stored with more information than in a simple list. The solution to this is to use a dictionary. As the graph is traversed, all nodes must be stored in a way that notes the path the algorithm took to get to the node. Therefore, the dictionary must have any node as a key and the node’s parent (or where it immediately came from) as a value. Then, upon reaching the destination node, the program can search the dictionary for a node, return its parent, which is the key’s value, and then search the dictionary for the parent and return its parent... and so on until the path is traced backwards.

Initial Algorithm Code:

```
# Breadth First Search
# Alanna Tempest, alannayt@gmail.com
# 2012-03-29

def bfs(graph, initial_node, goal_node):
    import networkx as nx
    from collections import deque

    # create queue, "visited" list, and general list for use in
    # algorithm
    queue = deque()
    visited = []
    list = []

    queue.append(initial_node)
    visited.append(initial_node)

    while len(queue) > 0:
        select = queue.popleft()  # 'select' is the node we're
                                  # working with at a point in time.
                                  # '.popleft()' ejects the oldest
                                  # value in the queue.
        if select == goal_node:
            return goal_node + " reached!"
        else:
            list = graph.neighbors(select)
            if len(list) == 0:
               break
            else:
                for i in range(len(list)):
                    if list[i] not in mark:
                        visited.append(list[i])
                        queue.append(list[i])
    return goal_node, "not found."
```

Results:

Shortest Path*	Path length
[2, 3, 16]	2
[45, 3, 26]	2
[47, 16, 23, 13]	3
[13, 3, 12]	2
[17, 3, 0]	2
	Average: 2.2
[15, 28, 22]	2
[41, 4, 22]	2
[6, 3]	3
[49, 6, 16]	4
[29, 48]	3
	Average: 1.6
[17, 7, 26]	2
[1, 3, 6]	2
[19, 0, 3, 2]	3
[35, 26, 8, 22]	3
[33, 35, 1, 36]	3
	Average: 2.6
[40, 10, 9]	2
[26, 3, 1, 42]	3
[25, 0, 22]	2
[1, 33, 11, 12]	3
[0, 3, 1]	2
	Average: 2.4
[5, 16, 33]	2
[37, 12, 40]	2
[35, 13, 30]	2
[6, 3, 0]	2
[35, 4, 6]	2
	Average: 2.0

Table 1 (left). Data for Network Size 50

Table 2 (below): All Averages

Network Size (Number of Nodes)	Five Averages of Each Set of Five Paths**	Average of "Five Averages…" Column
20	1.2, 1.8, 2.2, 1.2, 2.0	1.68
50	2.2, 1.6, 2.6, 2.4, 2.0	2.16
100	2.6, 2.8, 1.8, 1.8, 2.6	2.32
150	2.6, 2.4, 3.0, 3.0, 2.6	2.72
200	2.4, 2.2, 3.0, 2.8, 2.2	2.52
250	3.0, 3.2, 3.0, 2.8, 2.4	2.88
300	3.0, 3.0, 3.0, 3.6, 3.4	3.20
350	3.0, 3.6, 3.4, 2.8, 2.6	3.08
400	3.6, 3.4, 3.4, 3.4, 4.0	3.56
450	3.2, 2.8, 3.6, 3.4, 3.2	3.24
500	3.4, 3.4, 3.6, 3.0, 3.4	3.36
550	3.2, 3.2, 3.6, 3.0, 3.2	3.24
600	3.4, 3.4, 3.4, 3.6, 3.4	3.44
650	3.2, 3.8, 3.6, 3.4, 3.6	3.52
700	3.2, 3.6, 3.2, 3.6, 3.4	3.40

*As generated by our program. The first number listed is the initial node and the final is the destination node. The length of the path is the number of edges it traverses.

**As described in Methods, we ran a total of twenty-five trials per network size.

Table 3 (right). Averages Of Averages

Network Size (number of nodes)	Average Path Length
20	1.68
50	2.16
100	2.32
150	2.72
200	2.52
250	2.88
300	3.2
350	3.08
400	3.56
450	3.24
500	3.36
550	3.24
600	3.44
650	3.52
700	3.4

Graph 1 (below). Graph Of Table 3

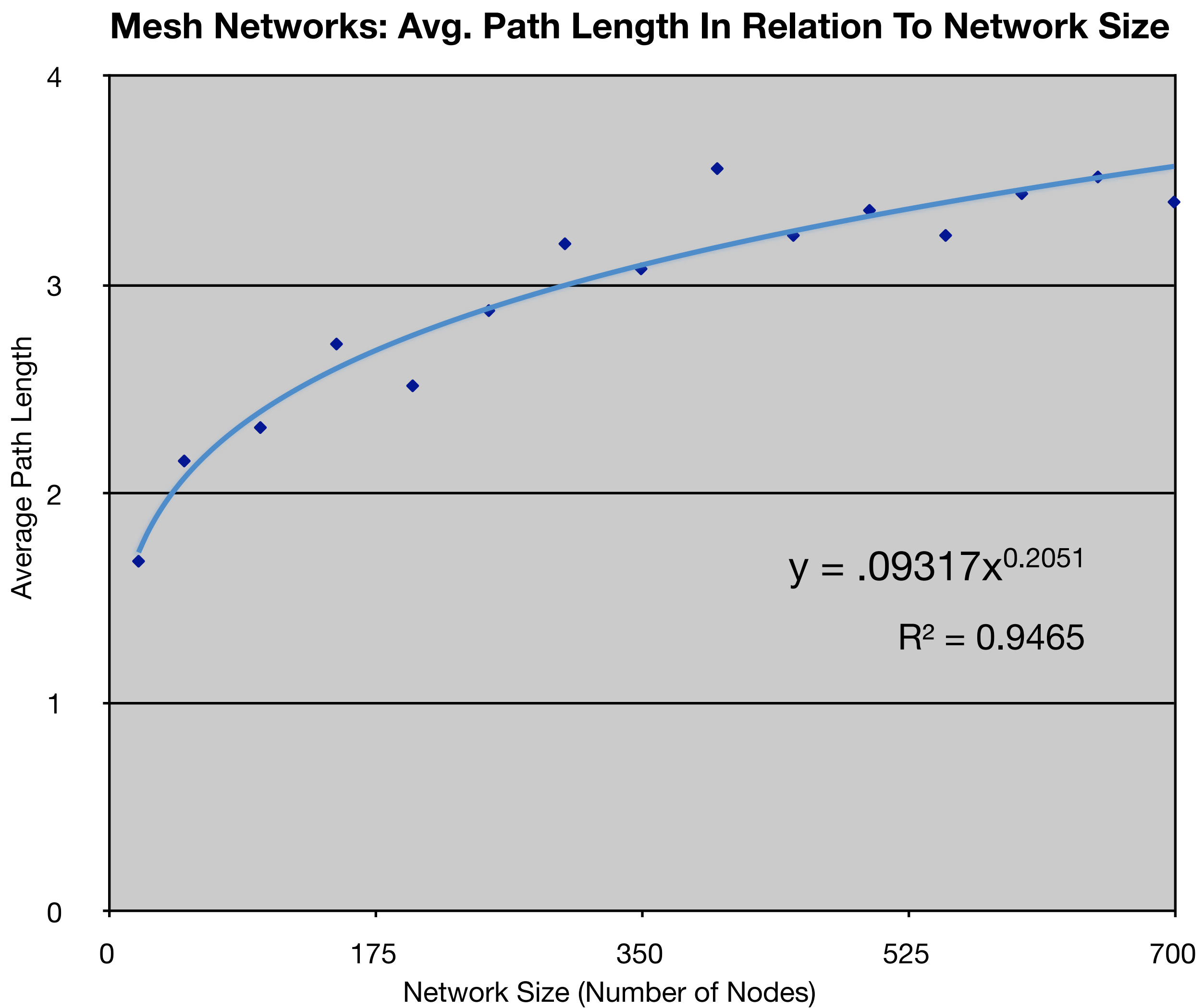

Table 1 shows the output for a network with size 50 nodes. This data is provided to show where the averages in the middle column of Table 2 come from. Each group of five results is generated on a different random graph with the same specifications. Therefore, the center column of Table 2 shows each of the averages of different randomly generated graph.

For the most part, the average path length increased with the size of the network (See Table 3). However, as the size of the network continues to increase, the average path length does not appear to increase proportionally. To see the relationship, we plotted the points from Table 3 in Numbers, the Mac equivalent of Excel. We then used the line of best fit. The "power law" regression had the highest coefficient of determination, and the shape of the data fit the line generated by the computer. In Graph 1, $y = .09317x^{(0.2051)}$ is the equation for the line of best fit, and R^2 is the coefficient of determination.

In general, the shortest path was usually three edges long. Rarely was there a shortest path that was five edges long.

Conclusion:

The breadth-first search algorithm was successful in traversing and finding shortest paths through graphs of small and great size.

For the different network sizes we used, the shortest path was rarely 5 paths away and was never 6 paths. Though the shortest path increased with the number of nodes, in larger graphs, the shortest path was not proportionally longer. A graph of seven hundred nodes is traversed with an average path length of 3.5, while a graph of fifty nodes is traversed with an average path length of 2.1. This is notable because the time it takes to traverse a graph is less than we expected for large graphs. It is remarkable that a graph fourteen times larger than the fifty-node graph can be traversed in less than twice the time.

However, there are to be expected delays in running a breadth-first search on large networks because the more nodes there are, the more edges there are connecting different nodes. Therefore, searching all the nodes immediately connected can take a long time because a singular node can be connected to hundreds of other nodes. Should we have the opportunity to work more on this project, we would analyze the amount of time this takes. We would also like to analyze scaled graphs, or ones that do not use preferential attachment and modified versions of the preferential attachment mechanism we used. We would hope to evaluate how to best represent mesh networks, then, as mentioned earlier, we would analyze the amount of time the breadth-first search algorithm takes in real time.

While the simulation was fast, sending a signal from one node to another is expected to take longer than the simulation. We do not have the information to accurately estimate the time it

takes to run a breadth-first search on a mesh network. However, our research supports that, of common search algorithms, the BFS is the most efficient algorithm for mesh network routing.

Acknowledgements:

First and foremost we would like to Mr. Ng, our project sponsor, for his encouragement and enthusiasm. We would also like to thank our school, and specifically our principal, Mr. Chiapetti, for helping us with the logistics of travelling to Challenge events. Many times the support from our sponsoring teacher and school urged us to proceed with our project.

We would also like to thank the supercomputing staff for their constant availability and wonderful support. One such support we received was the help of a project mentor. Our project mentor, John Donahue, provided amazing help over the last six months. We were often confused by our research and programming, but John patiently and consistently elucidated many issues for us. We owe our biggest thanks to John.

Finally, our positive team dynamic and our ability to work together was essential to our overall success. Neither of us could have done this alone. Alanna would like to thank Josh for his unwavering positive attitude. Josh would like to thank Alanna for giving him the opportunity to work with her and for not giving up. It was truly a great experience working on this project.

Bibliography:

Dijkstra's Algorithm. (n.d.). Retrieved from http://en.wikipedia.org/wiki/Dijkstra's_algorithm

Python Patterns - Implementing Graphs. (2000). Retrieved from

http://www.python.org/doc/essays/graphs.html

distanceedjohn. *Graph Traversals*. (2008, April 2). Retrieved from

http://www.youtube.com/watch?v=or9xlA3YYzo

What is FreedomBox? (n.d.). Retrieved from http://freedomboxfoundation.org/learn/

LastArtOfTheProblem. *Public Key Cryptography: Diffie-Hellman Key Exchange.* (2012,

February

24). Retrieved from http://www.youtube.com/watch?v=3QnD2c4Xovk

What's the difference between DFS and BFS? (n.d.). Retrieved from

http://www.programmerinterview.com/index.php/data-structures/dfs-vs-bfs/

Breadth-first search. (n.d.). Retrieved from http://en.wikipedia.org/wiki/Breadth-first_search

Connectionless Transfer (UDP). (n.d.). Retrieved from http://pcnineoneone.com/howto/

dataxfer3/

Dibbell, J. (2012, February). The Shadow Web. *Scientific American*, *306*(3), 60-645.

In addition to the libraries that came with Python 2.7.2, we installed and used these python packages:

- Distribute 0.6.25
- Numpy 1.6.1
- python-graph
- Network X
- Matplotlib

Visualizations:

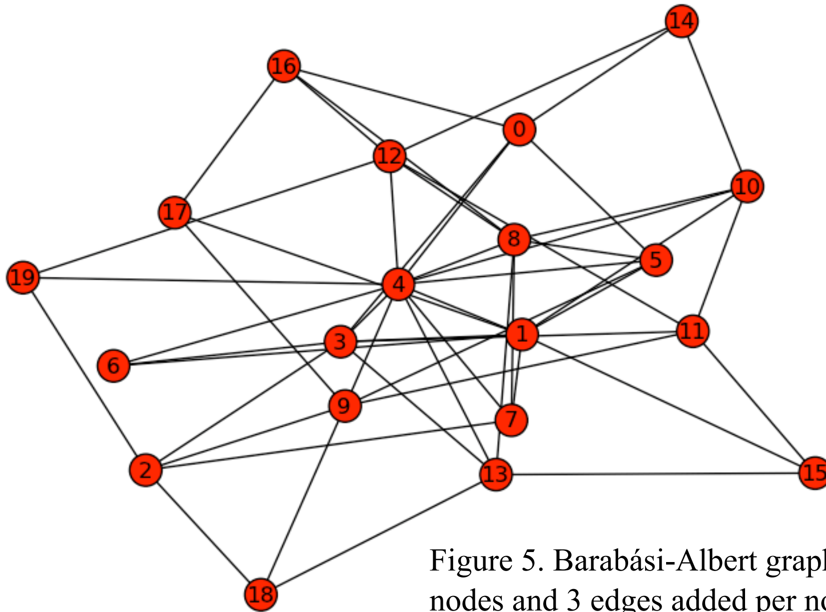

Figure 5. Barabási-Albert graph-generation with 20 nodes and 3 edges added per node.

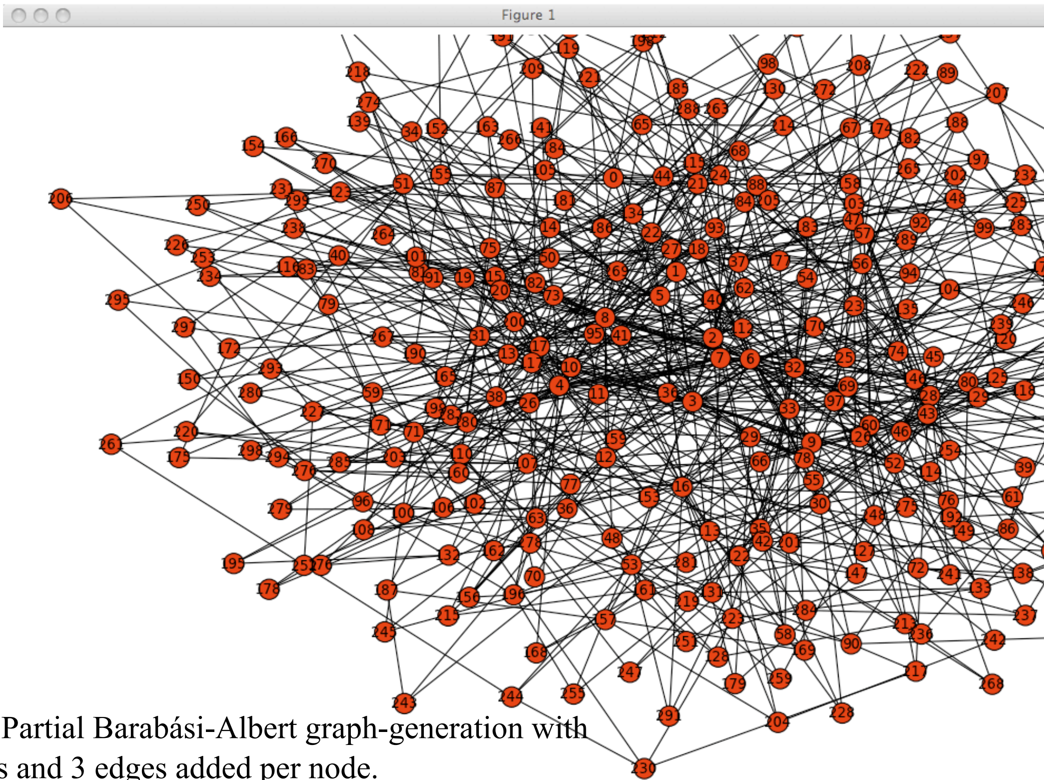

Figure 6. Partial Barabási-Albert graph-generation with 300 nodes and 3 edges added per node.

zoom rect, x=0.337714 y=-0.0448695

Figure 7. Partial Map of the Internet generated by The Opte Project