WIRELESS SENSOR NETWORK PROJECT

Travis Siems

TABLE OF CONTENTS

Executive Summary **2**
- **Introduction and Summary** **2**
- **Programming Environment Description** **3**
- **References** **4**

Wireless Sensor Network Backbone Report **5**
- **Reduction to Practice** **5**
  - Interface 5
  - Part 1 6
  - Part 2 11
  - Part 3 14
  - Verification Walkthrough 18
- **Benchmark Result Summary** **23**
  - Testing Process 23
  - Results 23
  - Analysis 29

Appendixes **30**
- **Appendix 1: MenuViewController.swift** **30**
- **Appendix 2: DisplayViewController.swift** **33**
- **Appendix 3: DisplayView.swift** **49**
- **Appendix 4: SphereScene.swift** **51**
- **Appendix 5: Node.swift** **53**
- **Appendix 6: Edge.swift** **53**
- **Appendix 7: StatsTableViewController.swift** **54**
- **Appendix 8: GraphsViewController.swift** **58**

EXECUTIVE SUMMARY

INTRODUCTION AND SUMMARY

Understanding how a wireless sensor network works is essential for the development and implementation of Internet of Things technology. This project simulates the connections of a wireless sensor network in the form of a graph. The graph can be used to model network algorithms and test implementations. Vertices are placed in a random geometric graph of different shapes and densities. The number of vertices and edges are controlled by various variables to form the graph. Many informative and statistical outputs are created in the form of charts, tables, and graphical representations all described in this document.

I have created a wireless network simulator in the form of an iOS application with real-time display and statistics. The graphical display helps users get feedback and understanding with the random geometric graph creation and coloring algorithms. All the statistics and charts in this document (aside from comparison charts) can be viewed directly from the application. The consolidation of all the information, displays, and charts in one place allows users to see immediate feedback and results so they can actually use and understand what they have at the palm of their hand.

The following table (Table 1) gives an overview of the graph coloring results from running the benchmarks. Each column gives additional information about how the algorithms ran and the results that were achieved.

TABLE 1: GRAPH COLORING SUMMARY

Benchmark #	N (Number of Vertices)	R (Connection Distance)	M (Number of Edges)	Min Degree	Avg. Degree	Max Degree	Max Degree when Deleted	Number of Colors	Terminal Clique Size	Max Color Class Size	Edges in Max Bipartite Subgraph
1	1000	0.101	14529	8	29.058	46	20	19	18	77	183
2	4000	0.071	119501	11	59.751	87	37	34	31	167	418
3	16000	0.036	505902	18	63.238	94	39	36	31	630	1628
4	64000	0.018	2054421	17	64.201	96	40	38	36	2491	6400
5	64000	0.025	3937016	34	123.032	164	72	65	52	1398	3806
6	4000	0.063	120574	20	60.287	90	38	34	30	164	416
7	4000	0.09	235794	47	117.897	164	68	59	52	89	244
8	4000	0.252	116325	26	58.163	86	36	34	30	169	443
9	16000	0.178	924732	47	115.592	158	70	62	62	366	1001
10	64000	0.09	3772681	50	117.896	173	74	66	55	1449	3902

To generate the graph, I first used a randomization function to obtain random coordinates, then I implemented the cell method for linear time edge formation [5]. Converting the coordinates to Cartesian format required formulas from Wolfram Alpha and Matula [9],[5]. I connected nodes a distance formula. To do graph coloring, I first implement the Smallest Last Ordering algorithm based on Dr. Matula's 1983 paper [7]. This allows the coloring to obtain a minimum number of colors. I then use a greedy approach to color each node with the lowest available color.

The next table gives statistics on the largest two backbones based on edge-count. The way these values are generated and calculated are explained in the Reduction to Practice section. Table 2 shows the bipartite subgraph output values from running the benchmarks. The first four value columns of this table are for the backbone of the bipartite graph with the most edges out of the combinations of bipartite graphs from the four colors with the most vertices. The second set of four columns is for the second greatest backbone from those conditions.

TABLE 2: BACKBONE STATISTICS SUMMARY

Benchmark #	Backbone 1 Vertices	Backbone 1 Edges	Backbone 1 Domination Percentage	Backbone 1 Faces	Backbone 2 Vertices	Backbone 2 Edges	Backbone 2 Domination Percentage	Backbone 2 Faces
1	151	182	99.5	–	150	177	99.6	–
2	329	416	99.95	–	324	405	99.9	–
3	1228	1613	99.7	–	1240	1606	99.92	–
4	4902	6359	99.73	–	4875	6329	99.82	–
5	2778	3790	99.98	–	2769	3786	99.97	–
6	324	415	99.95	–	323	414	99.95	–
7	176	236	100	–	173	233	99.9	–
8	333	428	100	97	333	422	99.9	91
9	725	997	99.96	274	722	960	99.93	240
10	2859	3888	99.82	1031	2859	3864	99.96	1007

To find the bipartite graphs, I used my own method where I build components from edges and nodes and combine them in an iterative fashion. I then sorted the components by edge count and chose the first one as the backbone. To calculate the domination percentage, I calculated the size of the graph cover by making a Set of all the vertices connecting to a vertex in the backbone. I then took the Set's size divided by the total number of vertices to see how dominating the backbone is [5]. To calculate the number of faces, I used Euler's Polyhedral formula [6].

PROGRAMMING ENVIRONMENT DESCRIPTION

For my implementation of the wireless sensor network simulation, I used the iOS environment, programming in Swift 3. This language integrates very nicely in the iOS environment and does not require a virtual machine. In other words, the swift code is compiled directly into machine code (similar to C++) [2]. In addition to the language's speed boost, I can also run the program on my iPhone. With the ability to export the application to my phone and interact with it anywhere, I can exhibit the programs functionality to others, increasing the application's worth. This could be taken to the next step by putting the application on the App Store and sharing the functionality with everyone.

To display the two-dimensional networks, I used Apple's Core Graphics library which is able to do two-dimensional drawings with high level commands [3]. For the three-dimensional display, I used Apple's SceneKit which allows a programmer to add objects in a 3D world one-by-one [8]. For displaying the charts, I used a library called Charts, created by Daniel Cohen Gindi [4]. This library can quickly draw many different kinds of charts for data visualization. I integrated all displays, tables, and charts into the application so that all the information could be realized from the same place.

For hardware, I am using a MacBook Pro with 16GB of RAM, running macOS Sierra version 10.12.4. I run the actual application using an iPhone 7 simulator with 2GB of RAM, running on iOS 10.3.1. This simulator is part of Apple's included MacBook applications that allows me to simulate most iOS devices with various operating systems [1]. The reason I choose to run the application using the simulator is to standardize the tests a little bit more. Additionally, the amount of RAM required for displaying high vertex and edge counts exceeds the amount of RAM allowed for an app on an iPhone but not the simulator [1]. This limitation can be overcome with future improvements to the display method, specifically for 3D models. Regardless, the algorithms, rather than the display method, are the focus of this document and the focus of this project.

When running the program without display, the memory usage is very reasonable. However, both displays tend to use far more RAM than the rest of the program. For displaying the results, I made a UITableView with 4 different sections: Coloring Stats, Timing Stats, Charts, and Bipartite Stats. By keeping all the results in this one view, the user can see as much information about the network as he desires. The source code for this project can be found publically on GitHub at *https://github.com/Tsiems/WirelessNetwork*.

REFERENCES

[1]"About Simulator", Developer.apple.com, 2017. [Online]. Available: https://developer.apple.com/library/content/documentation/IDEs/Conceptual/iOS_Simulator_Guide/Introduction/Introduction.html. [Accessed: 20- Apr- 2017].

[2]S. Anthony, "Apple's new Swift language explained: A clever move to boost iOS, while holding Android apps back - ExtremeTech", ExtremeTech, 2017. [Online]. Available: https://www.extremetech.com/computing/183563-apples-new-swift-language-explained-a-clever-move-to-boost-ios-while-holding-android-apps-back. [Accessed: 30- Apr- 2017].

[3]"CGContext - Core Graphics | Apple Developer Documentation", Developer.apple.com, 2017. [Online]. Available: https://developer.apple.com/reference/coregraphics/cgcontext-r9r. [Accessed: 20- Apr- 2017].

[4]"danielgindi/Charts", GitHub, 2017. [Online]. Available: https://github.com/danielgindi/Charts. [Accessed: 10- Apr- 2017].

[5]D. Mahjoub and D. Matula, "Building (1-ε) dominating sets partition as backbones in wireless sensor networks using distributed graph coloring", Proceedings of the 6th IEEE international conference on Distributed Computing in Sensor Systems, p.144-157, June 21-23, 2010, Santa Barbara, CA [doi>10.1007/978-3-642-13651-1_11]

[6]D. Mahjoub and D. Matula, "Constructing efficient rotating backbones in wireless sensor networks using graph coloring", Computer Communications, vol. 35, no. 9, pp. 1086-1097, 2012.

[7]D. Matula and L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms", Journal of the ACM, vol. 30, no. 3, pp. 417-427, 1983.

[8]S. Pop, "Introduction To SceneKit - Part 1 - We ❤ Swift", We ❤ Swift, 2017. [Online]. Available: https://www.weheartswift.com/introduction-scenekit-part-1/. [Accessed: 20- Apr- 2017].

[9]E.W. Weisstein, "Sphere Point Picking", Mathworld.wolfram.com, 2017. [Online]. Available: http://mathworld.wolfram.com/SpherePointPicking.html. [Accessed: 20- Apr- 2017].

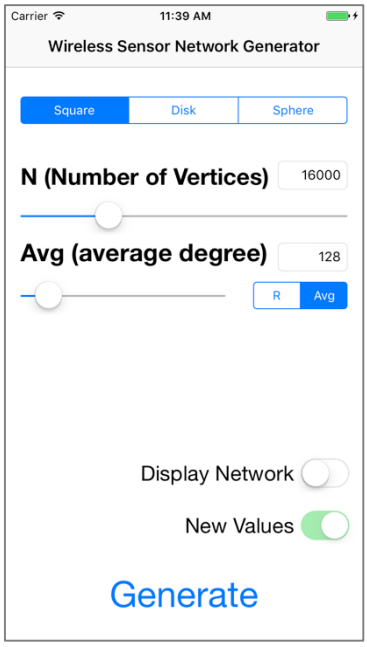
Wireless Sensor Network Backbone Report

Reduction to Practice

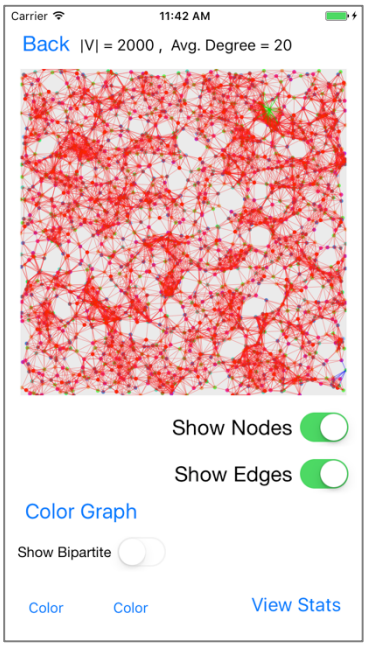
Interface

I implemented all requirements of the project in one iOS application, including graphical displays, statistical reports, and charts. This unifies the whole project and allows someone to get the full experience all in one place. The interface for interacting with this program is described and explained here.

In the first view (Interface View 1), the user determines what input values to use to create the wireless sensor network. The network model can be chosen with the segmented control, the vertex-count can be chosen with the slider or by tapping the number label, the connection type can be chosen using the segmented control, and the connection value can be selected using either the slider or by tapping the number label and typing in a value. To display the network as soon as it is generated, toggle the "Display Network" switch. If you came back to this view for any reason but do not want to generate new values, then make sure the "New Values" switch is toggled off. To generate the graph with all these inputs, tap the "Generate" button to go to the display view.

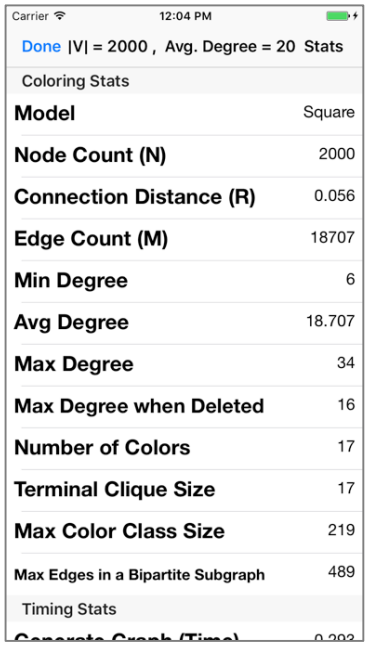


INTERFACE VIEW 1: WSN GENERATOR

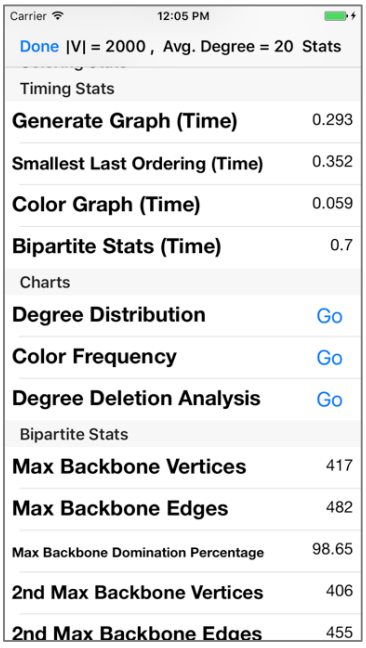


INTERFACE VIEW 2: DISPLAY

In the display view (Interface View 2), the user can control what they see as part of the graphical display. Most simply, they can toggle showing nodes or showing edges using the switches. They can activate the graph coloring algorithms at any time by tapping the "Color Graph" button. This runs all the graph coloring and bipartite backbone algorithms in this project. To show a bipartite graph, first select two different colors by tapping each color button and scrolling to a color. Then tap the show bipartite switch when it is enabled. To go to the statistics view, tap the "View Stats" button.



INTERFACE VIEW 2: STATS TOP



INTERFACE VIEW 3: STATS BOTTOM

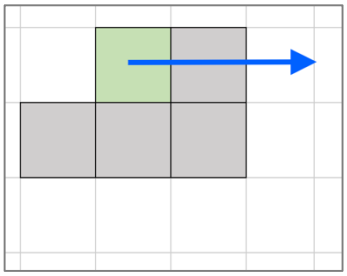
From the statistics view (Interface View 3 and Interface View 4), the user can see the graph coloring statistics, timing data, distribution charts and bipartite backbone statistics. The user can view each chart by tapping the "Go" button next to the desired chart.

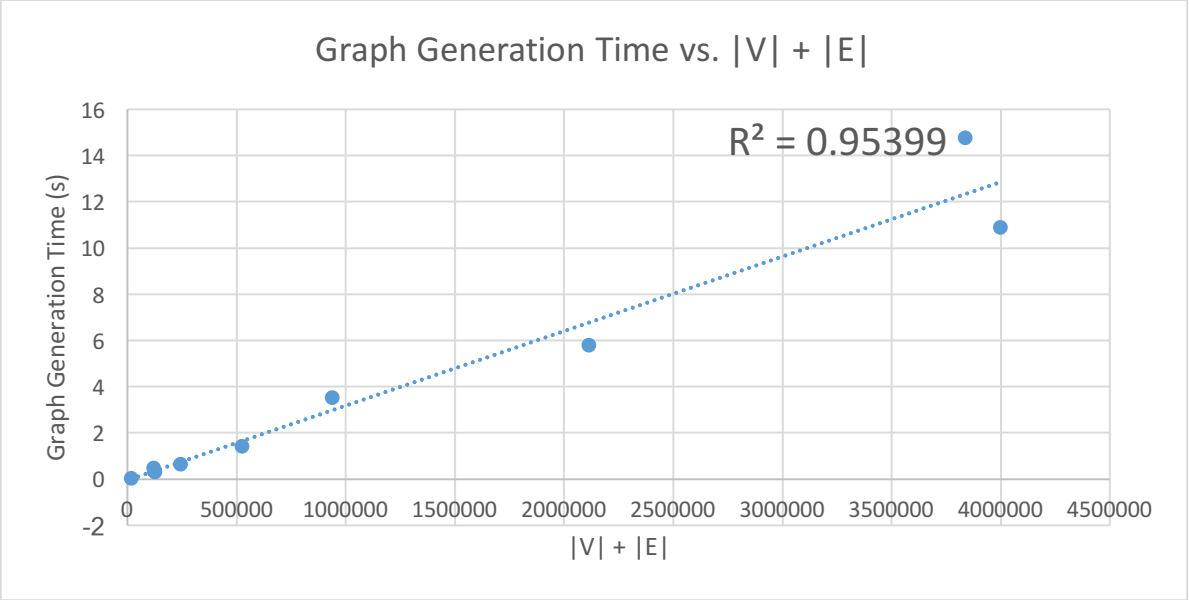
PART 1

The first part of the project (creating and displaying the random geometric graph) is implemented by first generating vertices in the given network model and then determining where edges should be created. After the vertices and edges are in their respective lists, I generate an adjacency list of the vertices where the edges are implicit based on the adjacency list relationship. To create random vertices in a square, I generated a random value [0,1] for x and a random value [0,1] for y as their coordinates. To create random vertices in a disk, I generated a random value [0,360] for θ and a random value [0,1] for the r^2 value. I then converted these polar coordinates to rectangular coordinates using $x = r\cos\theta$ and $y = r\sin\theta$. To create random vertices on the surface of a sphere, I generated a random value [-1,1] for u and a random value [0,360] for θ. I then converted these values to Cartesian coordinates using the following equations: $x = \sqrt{1-u^2}\cos\theta$, $y = \sqrt{1-u^2}\sin\theta$, and $z = u$.

My first implementation to determine the edges was to compare each vertex with every other vertex. The complexity of this implementation was $O(|V|^2+|E|)$ where "|V|" is the number of vertices and "|E|" is the number of edges. This was very inefficient so I decided to attempt other solutions.

I then implemented the cell method to find the edge relationships. To do this, I divided the network into a grid of square "cells" with area R^2 where "R" is the maximum connection distance. I then put each vertex in its respective cell and compared the vertices of each cell to the surrounding cells. The complexity of this algorithm is O(|V|+|E|). The time-complexity is vastly decreased using the cell method because this reduces the number of comparisons needed between vertices. The figure to the right shows which cells are compared around the current cells (as long as they are in the grid).





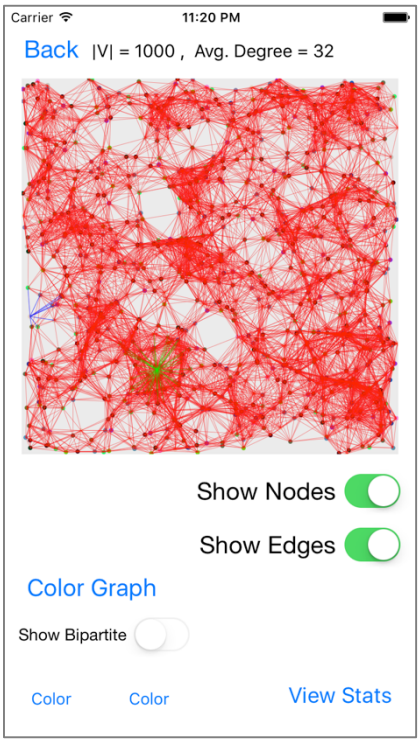
The chart above shows the Graph Generation Time in seconds on the y-axis vs. |V| + |E| on the x-axis. The R^2 correlation constant is 0.95399, which is close to the ideal 1.0 (perfectly correlated). The linear correlation can easily

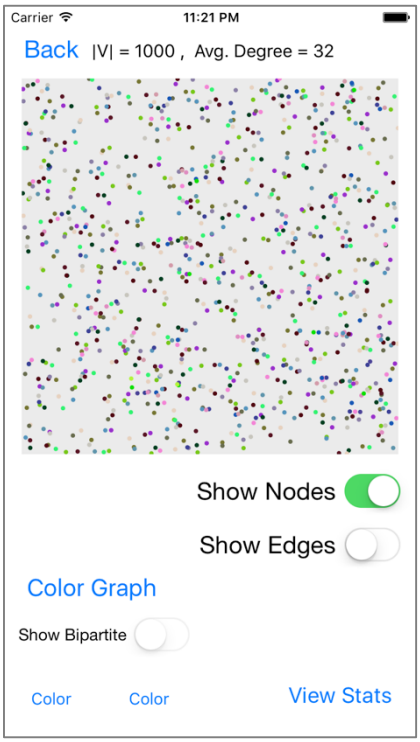
be determined, with slightly more variability as |V|+|E| increases. This linear time complexity was achieved with very little memory overhead. The only thing required was the list of cells that held the nodes. Now, given that I did not use pointers, every node had to be copied completely, but had this been done using pointers, the memory overhead would have been the size of a pointer multiplied by the number of nodes. Since this increases within O(|V|+|E|), I can conclude that this algorithm also has linear memory complexity.

The cell method is also implemented for the sphere, but more test cells are considered with only a few of those cells actually having cells. Since I determined the cells using x/y/z coordinates, many of the center cells and some outside cells are empty. Regardless, I consider a maximum of nine surrounding test cells for each cell with vertices in it where vertices may be in for possible connections.

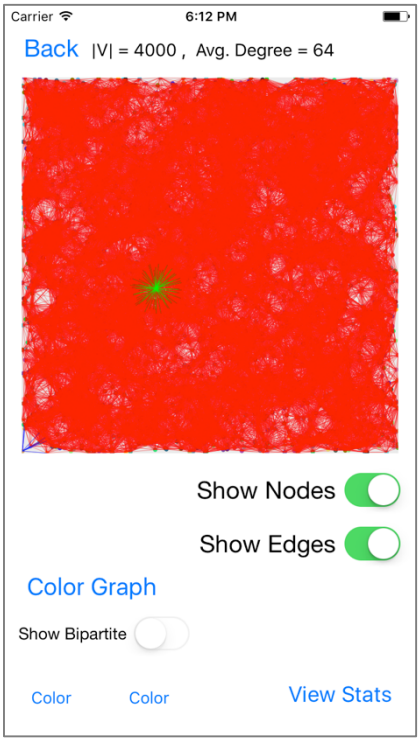
The output from Part 1 is an adjacency list of nodes. Since I cannot use pointers in the same way as other languages, I use Swift's built in mutable List and nested those. With this structure, I could quickly find where a node's connection list was by going to the index of that node's id. This made access very easy and efficient.

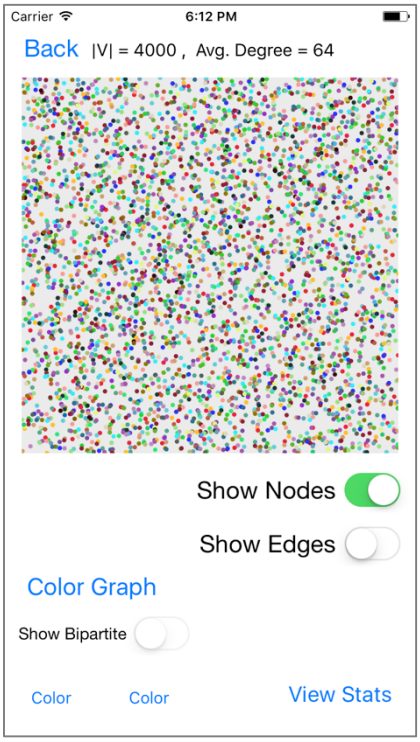
The following images show a graphical representation of the resulting graphs generated from the benchmark tests. The benchmarks are in order by benchmark number and show the vertex-count and average degree as the title of the iOS view. The first image for each pair is shown with edges and the second is shown without edges. If there is only one image, that is because showing so many edges gave no additional information due to clutter.



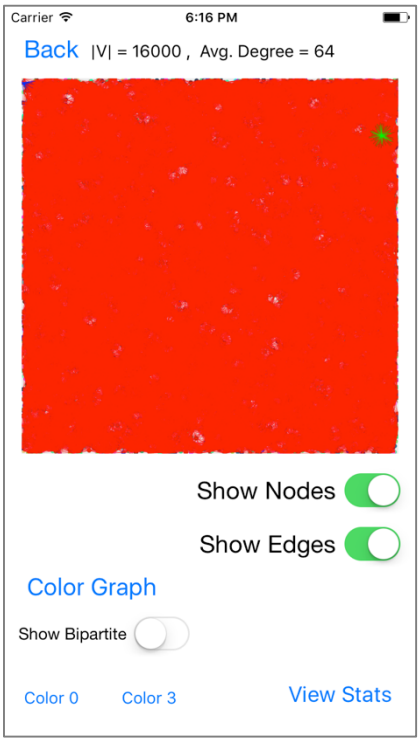


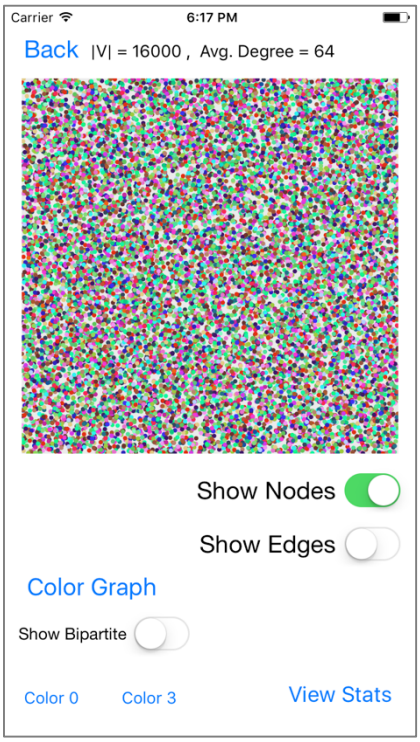
Benchmark 1: The graph is fairly empty. You can clearly see most of the connections, including the minimum degree vertex against the border on the left-center and the maximum degree vertex in the bottom center.



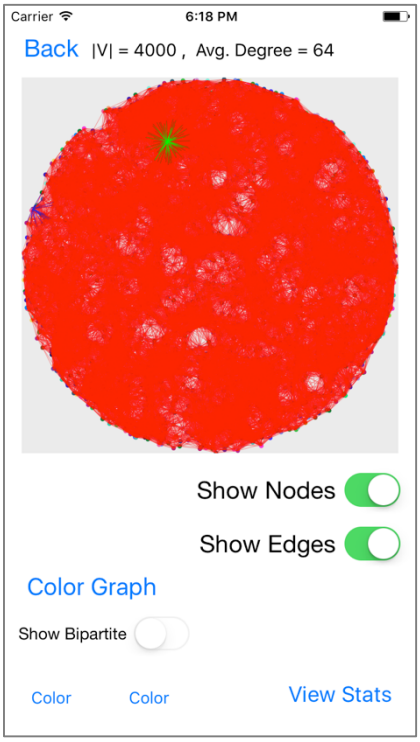


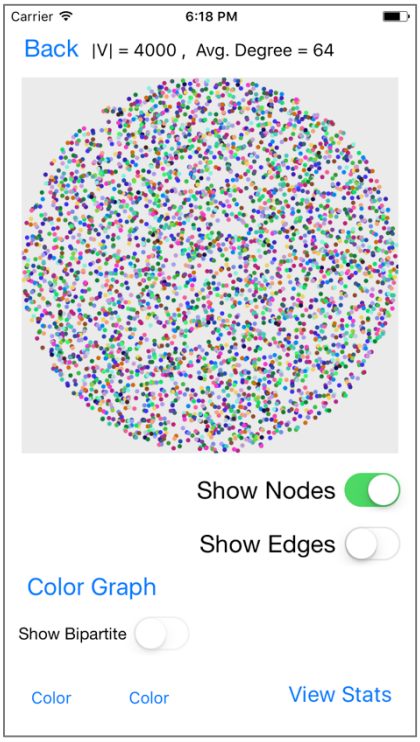
Benchmark 2: The graph is much more crowded with edges and the minimum degree vertex is now in the bottom left-hand corner with the maximum degree vertex towards the center.



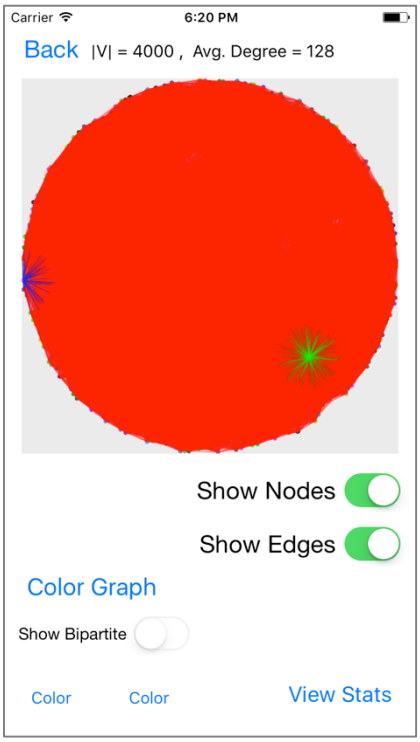


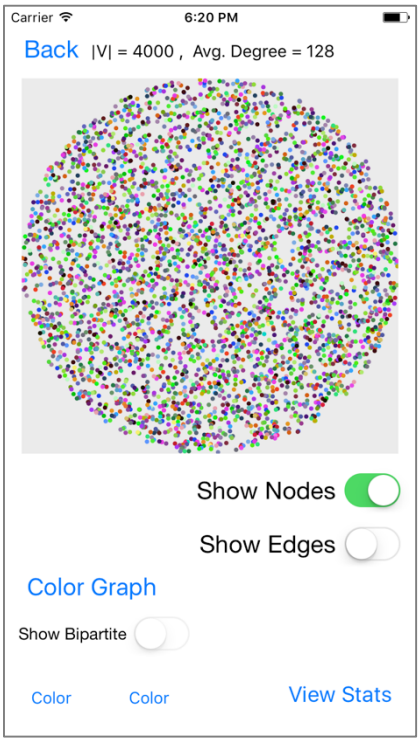
Benchmark 3: There are now so many edges that you can't see the connections. If you look carefully, you can see the minimum degree vertex in the top left-hand corner and the maximum degree vertex towards the top-right.



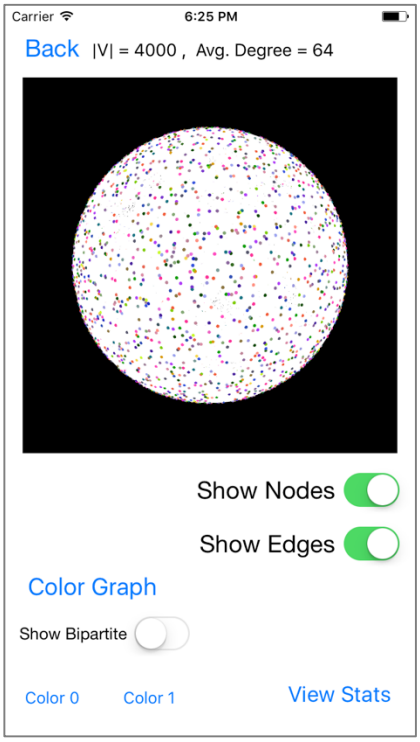


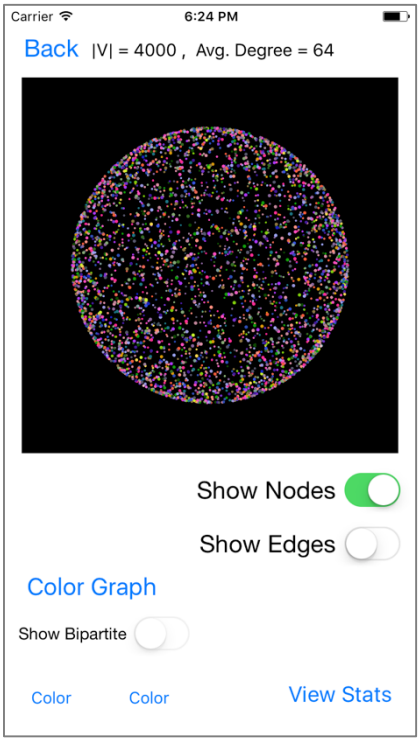
Benchmark 6: There are already so many edges that connections can barely be determined, but the minimum and maximum degree vertices are easily found towards the top-left.



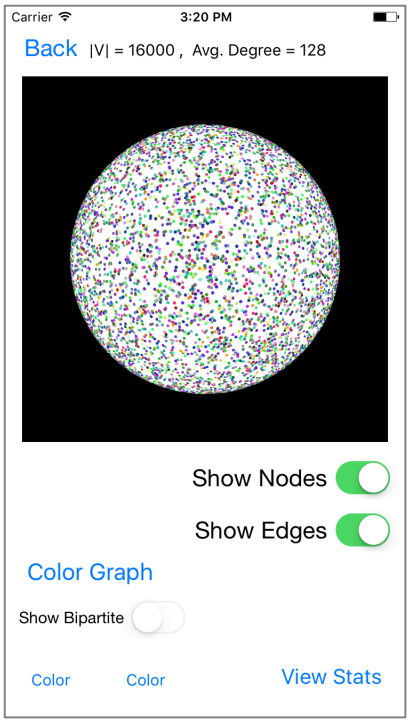


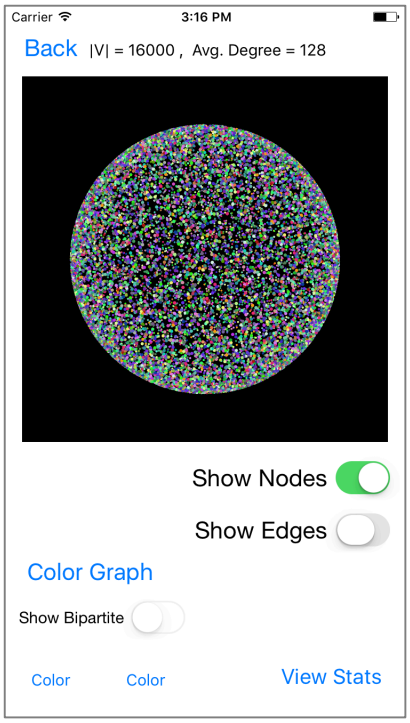
Benchmark 7: There are so many edges that individual connections cannot be seen, but minimum degree and maximum degree vertices can be found very easily.



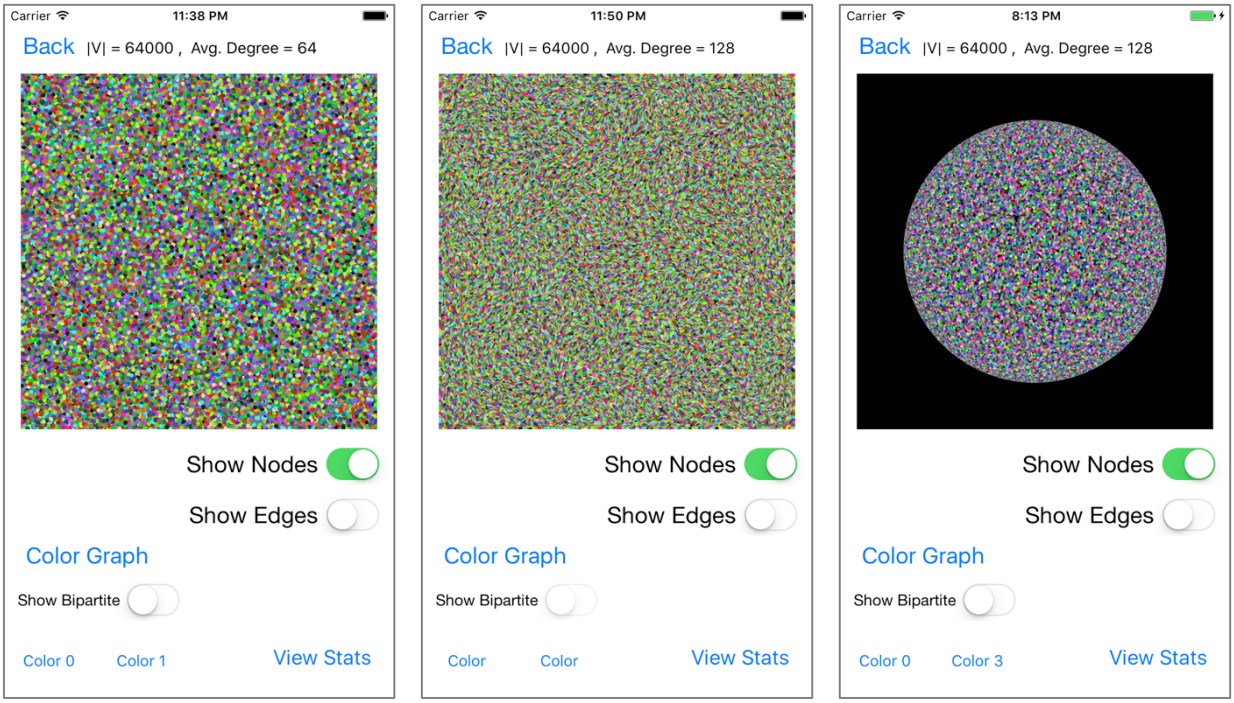


Benchmark 8: There are so many edges that individual connections cannot be seen. The Sphere highlights the min and max degree with red edges, but no red edges can be seen so they are either covered up or on the other side.





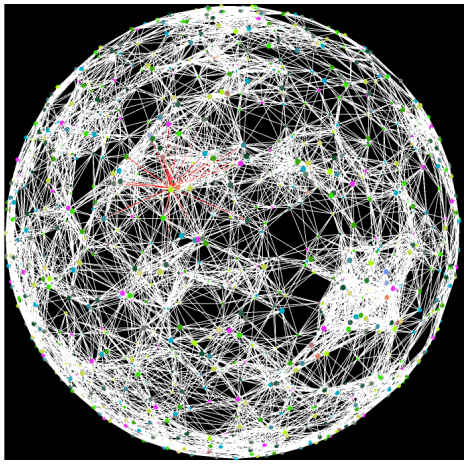
Benchmark 9: There are so many vertices in this projection that even those are hard to pick out individually. With edges displayed, there was no visual difference compared with Benchmark 8.



Benchmarks 4, 5, and 10, respectively: These graphical displays show how crowded the graphs get with so many vertices. There is no empty space, and showing edges would require too much computational time and memory.

For displaying both the square and disk network models, I use Core Graphics to map a vertex's x/y coordinates into the display area, drawing a small circle, and I show edges by drawing a thin line between the two related vertices. The blue edges are all connected to the vertex with the lowest degree. The green edges are all connected to the vertex with the highest degree.

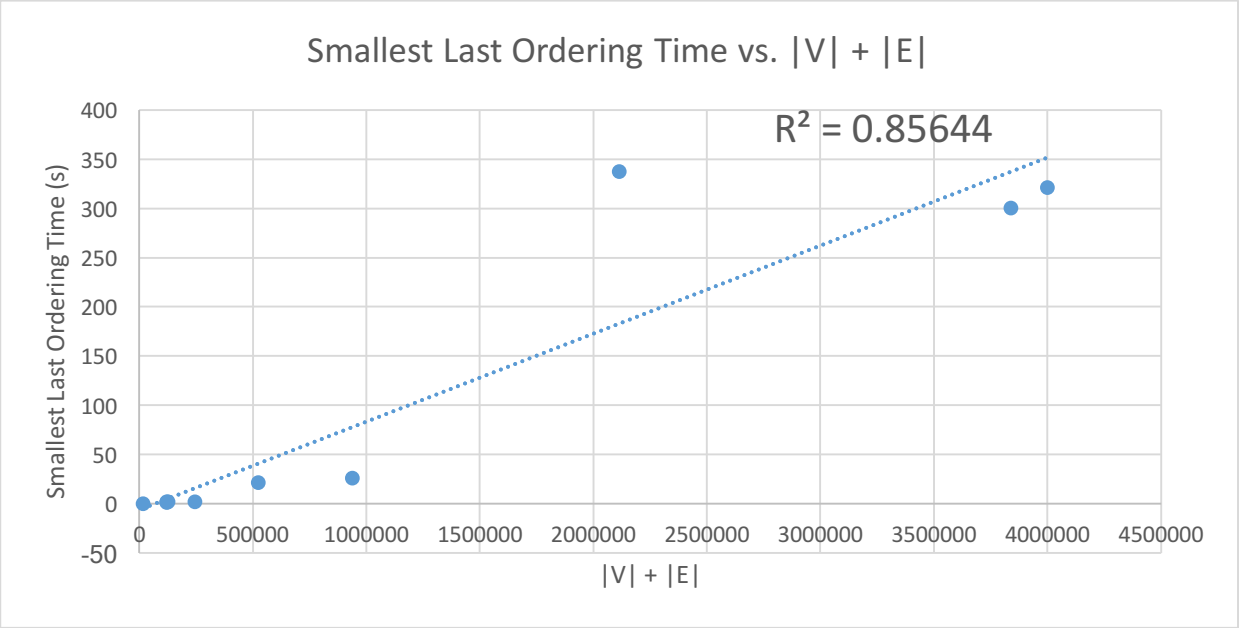
Displaying the sphere using SceneKit is customizable and easy to work with, but it requires far too much RAM to display and interact with when there are many edges. Additionally, the amount of RAM allowed for an app on an actual iPhone is reached after before even benchmark one is displayed. The vertices with the min and max degrees have edges connected to them that are red, and they can be differentiated by the number of edges connected to it. The image to the right shows the maximum degree vertex. Although it may seem difficult to see here, the user is able to pinch to zoom and tap and hold to move around the sphere. This capability significantly helps the interaction and understanding of this graphic.



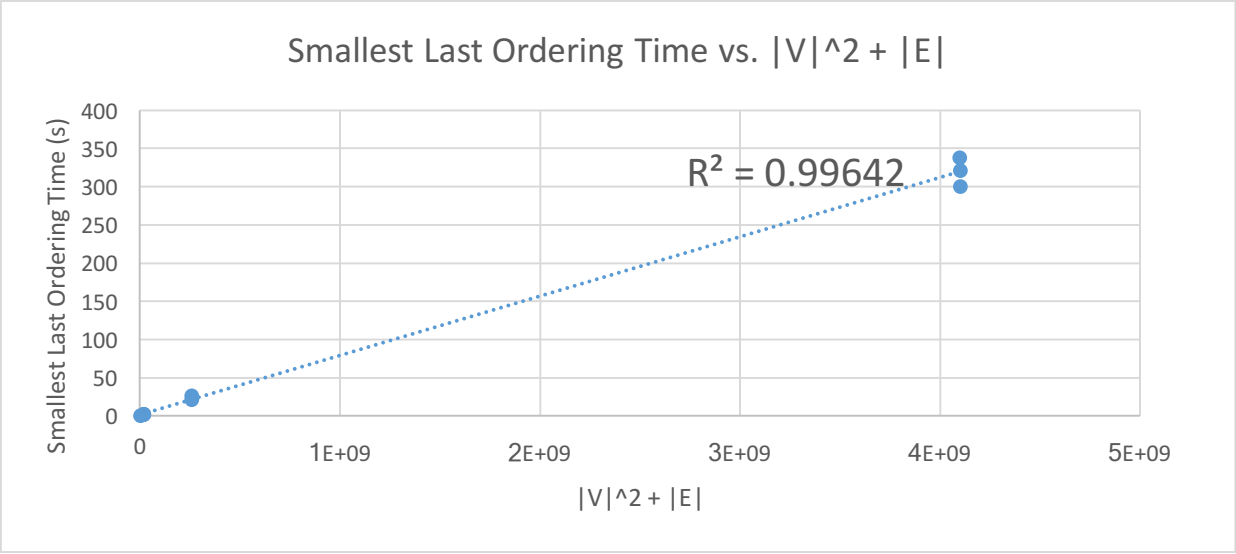
PART 2

I implemented graph coloring by taking in the adjacency list of the network, sorting it using the smallest last ordering algorithm, and greedily assigning the lowest available color to each vertex based on adjacent vertices. Coloring the graph can be done with very little time compared to the smallest last ordering. The complexity of just the graph coloring is $O(|V|+|E|)$. The complexity for my implementation of the smallest last ordering is either $O(|V|^2+|E|)$ or possibly $O(|V|+|E|)$ with some outliers at $O(|V|^2+|E|)$.

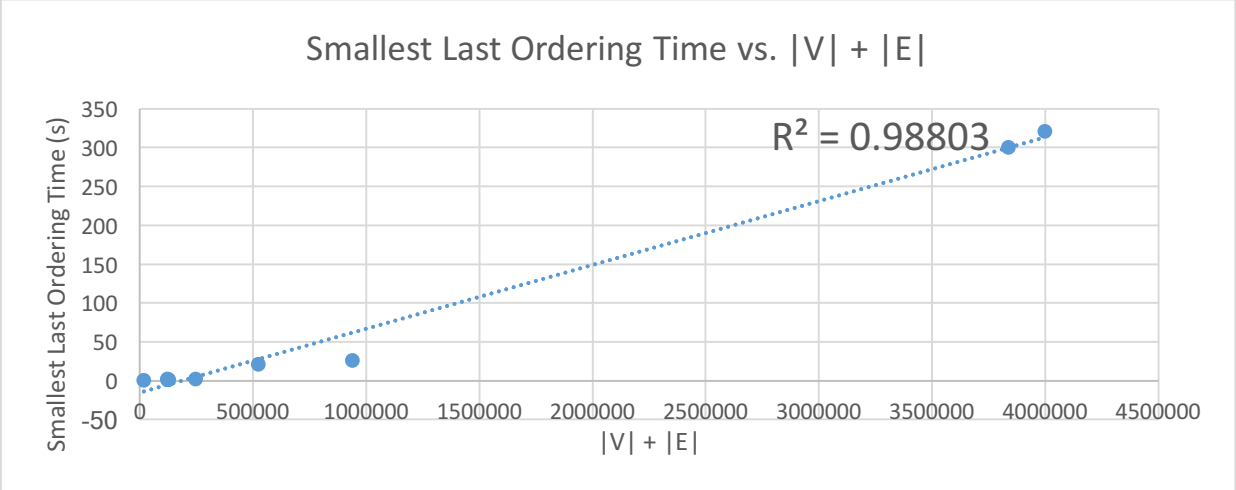
The smallest last ordering algorithm is computed with the following process: first I create a list (called buckets list) that will hold lists of vertices with the same degree. I then assign each vertex to a bucket based on its degree. This is essentially a bucket sort. Then I iterate through the buckets until I find the first one with vertices in it. I then pop the last vertex in that bucket from the list, subtract one from each vertex its connected to, and update my reference list which keeps track of what bucket each vertex is in. I continue this process until all vertices are deleted from the graph. When I delete a node from the adjacency list, I append it to a new one. At the very end, I reverse that list for the correct ordering.



The above chart shows how my algorithm runs compared to linear time complexity as defined by O(|V| + |E|). The R^2 value is 0.85644, which is not a horrible error rate, but it may be a signal that the actual run time of my implementation is not linear. In the chart below, I plot the Smallest Last Ordering Time data against $|V|^2 + |E|$ which seems to be much better correlated because the R^2 value 0.99642 of this chart is much closer to 1.0 than that of the linear time complexity chart. With this analysis, I can conclude that my implementation of the Smallest Last Ordering algorithm is not linear but likely polynomial or has an anomaly causing this time complexity.



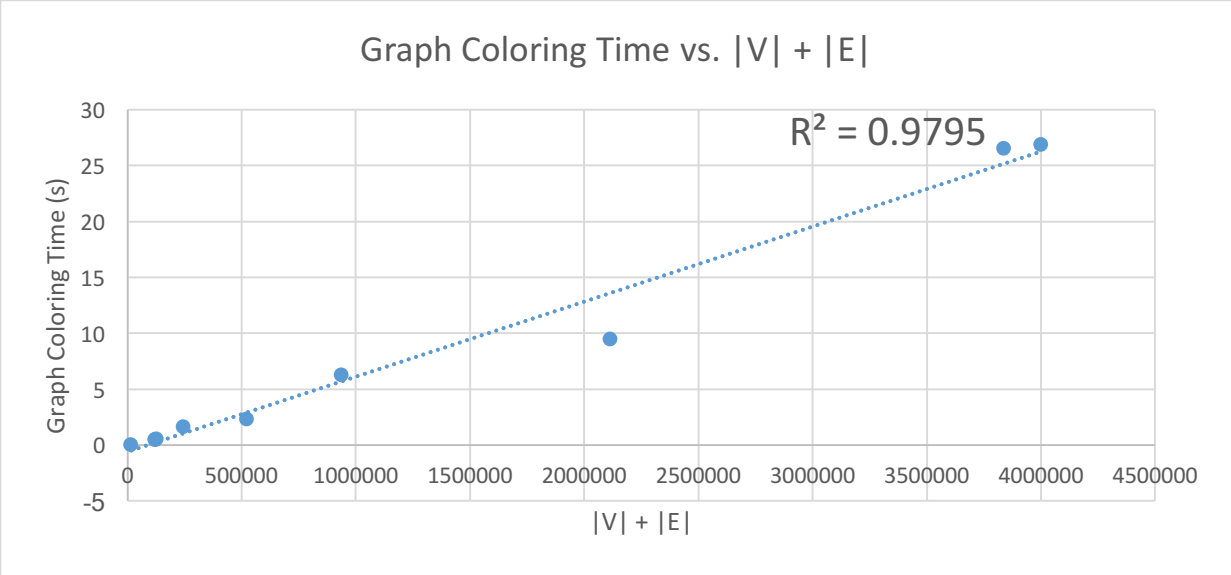
The smallest last ordering should run with a complexity of O(|V|+|E|), but my tests prove that it may likely run with a time complexity of O($|V|^2$+|E|). The deletion of a vertex from the adjacency list causes this nonlinear time complexity because each previously connection node needs to be updated. When implementing the smallest last ordering method, the node with the smallest degree is "deleted" and the connecting nodes are updated. Using Swift, I was unable to use direct pointers to update the nodes as needed in the algorithm, so the running time suffered with the iterations required. Had I been able to use pointers, the updating time would have been much faster. Additionally, the lack of pointers made the space requirement much greater throughout the project. Although the space requirements never grew exponentially, the linear growth could have been much slower with the proper use of pointers. Regardless of this possible time complexity issue, if we remove Benchmark 4, then the linear correlation becomes a real possibility as shown it the chart below. With an R^2 value of 0.98803, may algorithm could be considered to run in linear time. It is possible that the R^2 value alone may not be a good indicator of correlation, but it usually does a good job when determining linear correlations.



A clique is a set of nodes that are completely interconnected. When the nodes are deleted from the graph in the Smallest Last Ordering algorithm, the terminal clique is found the first time that all the remaining nodes are

completely interconnected. The solution to finding the terminal clique of the Smallest Last Ordering algorithm becomes very evident once the buckets are printed throughout the algorithm (as I will do in the verification walkthrough later). Once the minimum bucket is equal to the difference between the new adjacency list size and the old adjacency list size, I can begin to test for the terminal clique. The next main condition that I check for is that the index of the test bucket (which indicates the degree of each vertex inside) is equal to the bucket size minus 1. This makes sense because each node in the terminal clique will have the degree of the number of nodes left minus 1 (itself). Therefore, this method of finding the terminal clique as part of the Smallest Last Ordering algorithm is very efficient, simple, and reliable.

The graph coloring time complexity is O(|V| + |E|). For each vertex, I check to see if it can be colored based on the adjacent vertices' colors. If a color is already taken, then another color must be used. If all colors are taken, then the program adds a new available color. I use a greedy approach by using the lowest color number possible for each vertex after sorting by the Smallest Last Ordering. The chart below shows the benchmark's Graph Coloring Time vs. |V| + |E| as well as a linear correlation with $R^2 = 0.9795$. Since this value is very close to 1.0, I can conclude that this is probably a linear relationship.



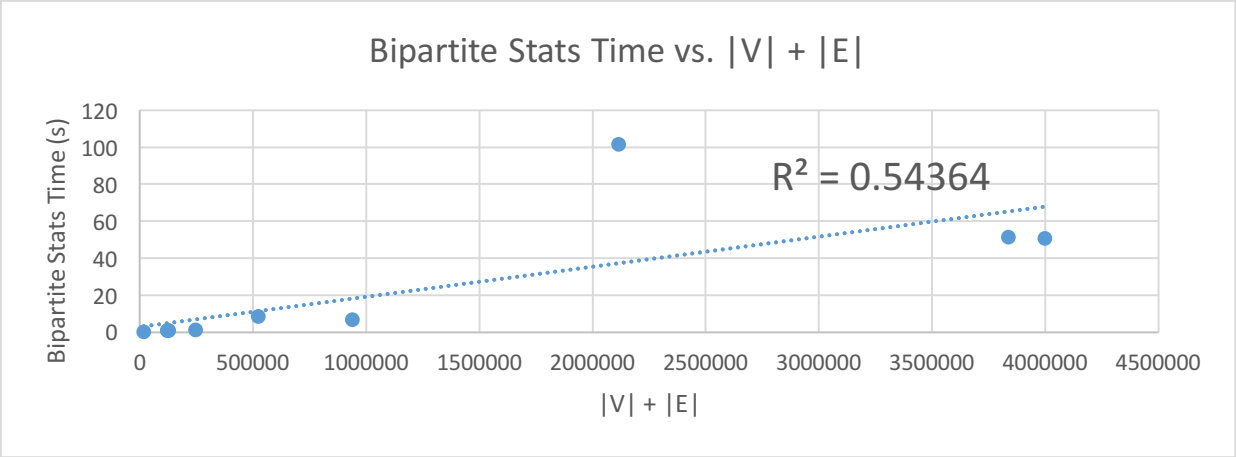
PART 3

A bipartite graph can be created by selecting two colors and display all the vertices and edges between them. I calculated this by checking which edges have both a vertex with the first color and a vertex with the second color. I then displayed these vertices and edges to see the bipartite subgraph. The backbone of this subgraph is the largest component. I broke the subgraph into components, sorted them, and selected the largest one to be the backbone.

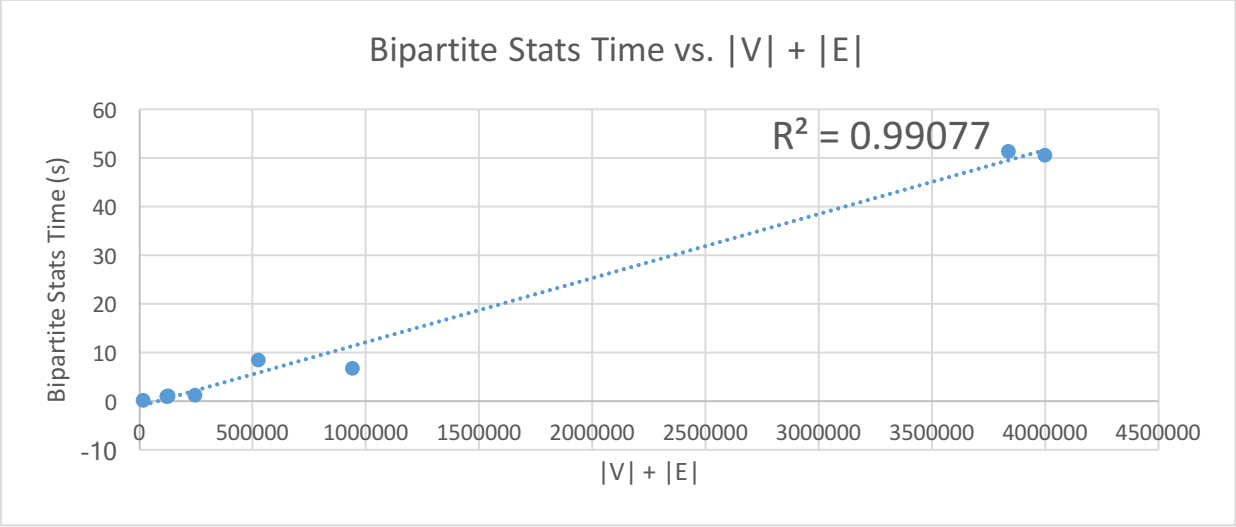
I separate the subgraph into its components by iterating through the edges and assign both related nodes to the same component. If one is already a member of a component, then the other is assigned to it as well. If both have components that are different, then I combine the components. If neither are in a component, I create a new one. If both nodes are already in the same component, then I increment the edge-count for that component. Each of the components is kept in a "components" list, which is later sorted to find the backbone.

The domination percentage is calculated by first determining the backbone of the bipartite graph and then finding how many vertices are in the cover created by that backbone. This value is divided by the total vertex-count to get the final domination percentage of the backbone. To find the cover, I iterated through each node's list of connecting

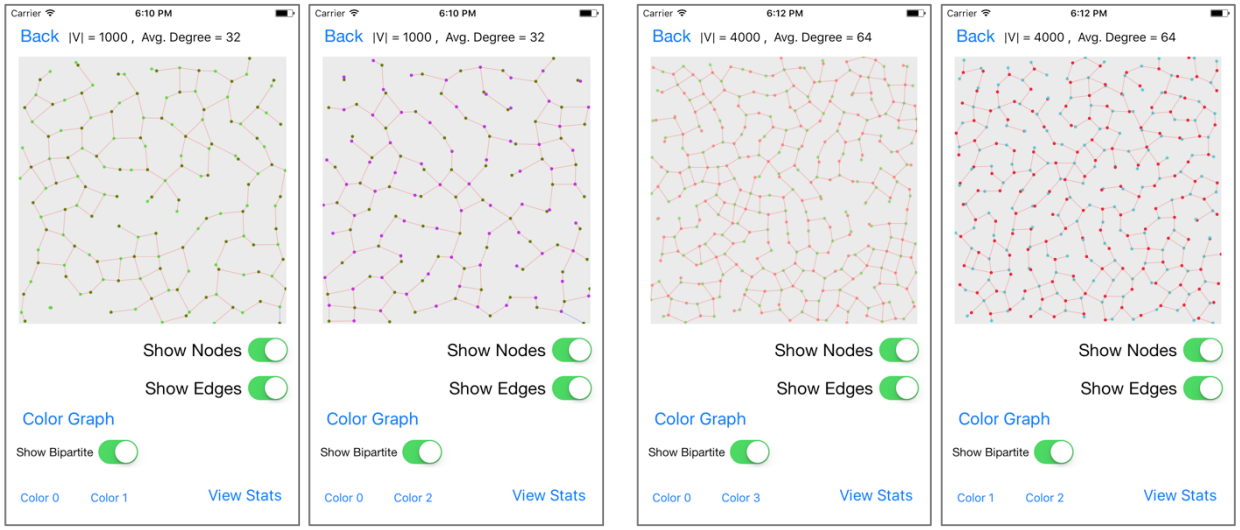
nodes in the adjacency list and added them to a Set. This Set removed duplicates so I could have the nodes in the cover just once. To calculate the number of faces on a backbone of a sphere, I use Euler's Polyhedral formula: $F + V - E = 2$, or rather $F = E - V + 2$.



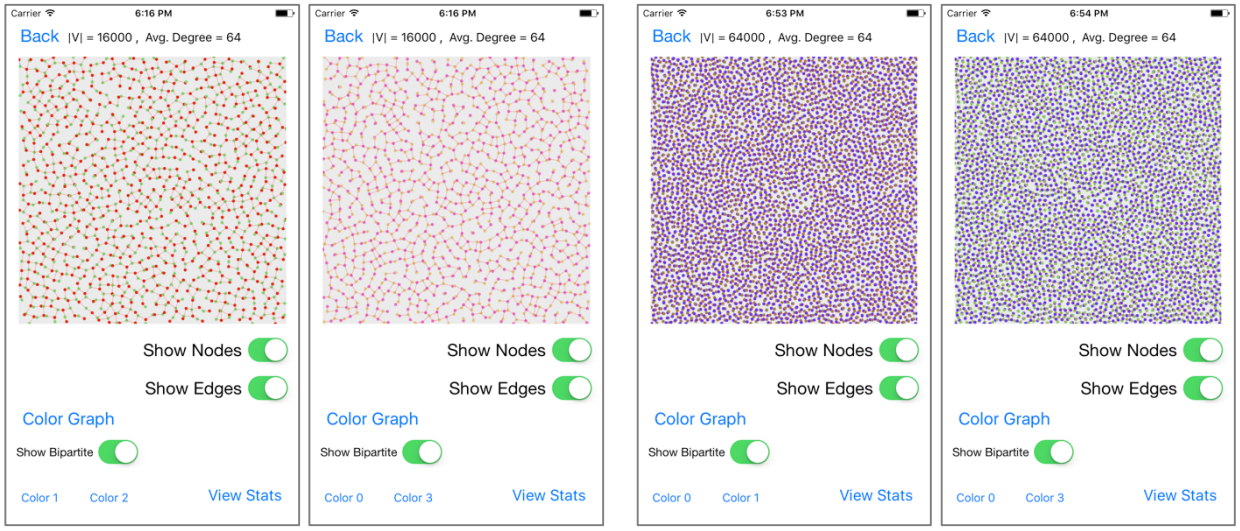
This chart above shows how a linear correlation might not fit for calculating all the bipartite subgraph backbone statistics because it has an R^2 value of 0.54364. Although this value is far from the ideal 1.0, there seems to be an outlier. The next chart shows how removing the outlier can actually make the statistics go from a very nonlinear correlation to a linear with an R^2 value of 0.99077. This outlier, however, cannot simply be ignored. There must be a reason why it is an outlier, and I believe that it is based on the number of vertices in the first few color classes. In order to find the largest backbones, the bipartite subgraphs had to be created, and with so many vertices to work with, this could add to the time. Regardless, most of the time these algorithms fall under a time complexity of O(|V|+|E|), so it is very likely linear time dependent.



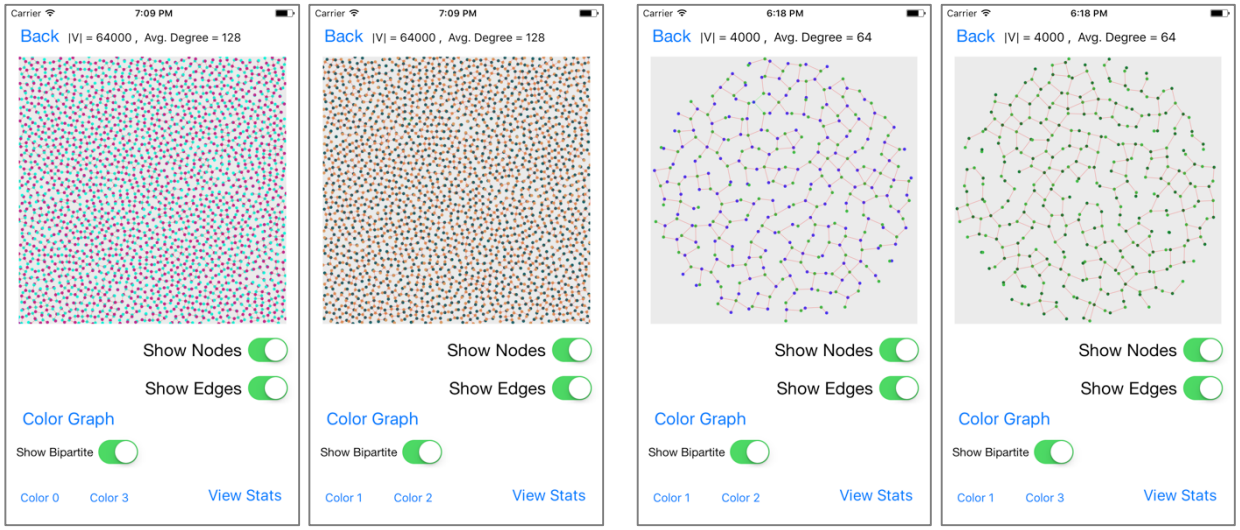
The following images display the two largest bipartite subgraphs for each benchmark. The largest is shown first and the second largest is to the right of the first for each pair. The benchmarks are in order by benchmark number and show the vertex-count and average degree as the title of the iOS view. The colors used in each bipartite subgraph can be seen at the bottom left hand corner of each image. Note that these are the exact same bipartite subgraphs that the summary tables describe, but that minor components are also shown to get a full picture of the subgraph.



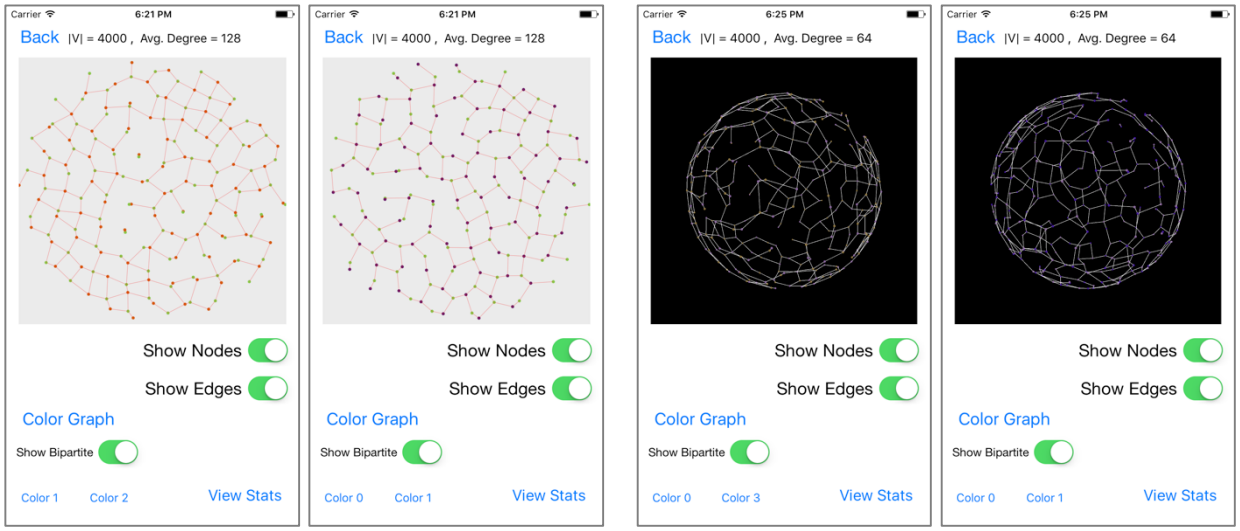
Benchmarks 1 and 2, respectively: These benchmarks are the best for visualizing what is actually happening because they are much more spread out and simple than the remaining benchmarks. Benchmark 1 used colors 0 and 1 for its largest bipartite graph and colors 0 and 2 for its second largest. Benchmark 2 used colors 0 and 3 for its largest bipartite graph and colors 1 and 2 for its second largest.



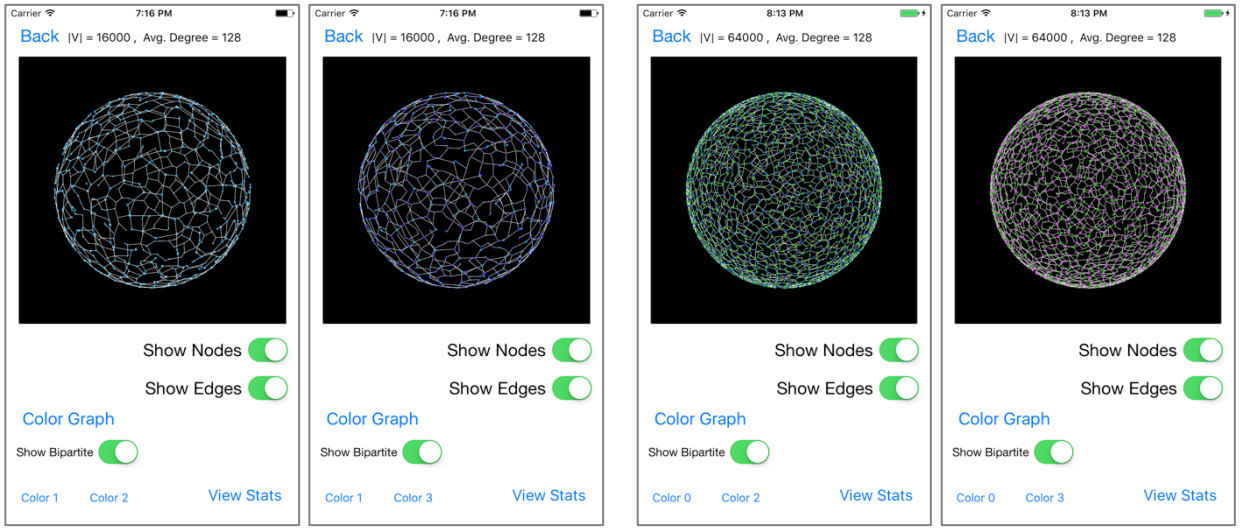
Benchmarks 3 and 4, respectively: The vertices in these benchmarks are much denser than the previous ones. This is due to the smaller R value and the increased vertex-count. Going from benchmark 3 to benchmark 4, the vertex-count increases greatly, but the average degree stays the same. This ensures that the R value will be decreased, and with so many vertices, it ends up being about four times as dense. Benchmark 3 used colors 1 and 2 for its largest bipartite graph and colors 0 and 3 for its second largest. Benchmark 4 used colors 0 and 1 for its largest bipartite graph and colors 0 and 3 for its second largest.



Benchmarks 5 and 6, respectfully: Benchmark 5 is interesting because as the degree increased from benchmark 4, the bipartite graphs are more spread out and less populated. This is because the R value was increased and in order for the nodes to remain independent from one another (keep the same color) they have to be at least R distance apart. Benchmark 6 is the first benchmark with a disk model. The disk model helps show that the network model in a plane does not have to be a square, but rather can fit many different shapes. Benchmark 5 used colors 0 and 3 for its largest bipartite graph and colors 1 and 2 for its second largest. Benchmark 6 used colors 1 and 2 for its largest bipartite graph and colors 1 and 3 for its second largest.



Benchmarks 7 and 8, respectfully: Benchmark 7 is interesting in the same way that benchmark 5 was, but it might be easier to see the density change with the disk. Benchmark 8 shows the first Sphere benchmark bipartite graphs. As shown, the entire sphere is shown with the parts further away smaller than the parts closer to provide a helpful representation of the bipartite graphs. Benchmark 7 used colors 1 and 2 for its largest bipartite graph and colors 0 and 1 for its second largest. Benchmark 8 used colors 0 and 3 for its largest bipartite graph and colors 0 and 1 for its second largest.

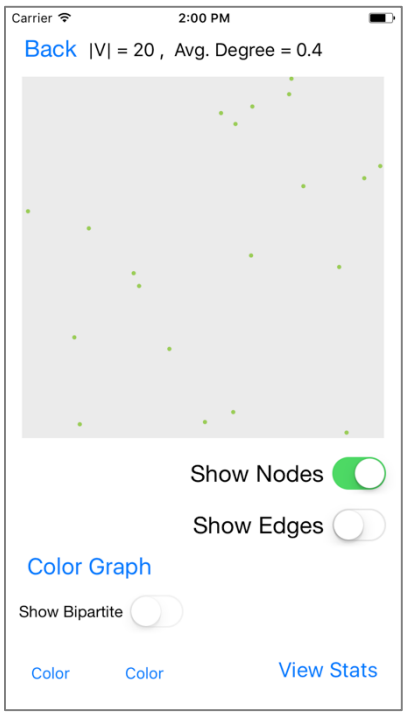


VERIFICATION WALKTHROUGH

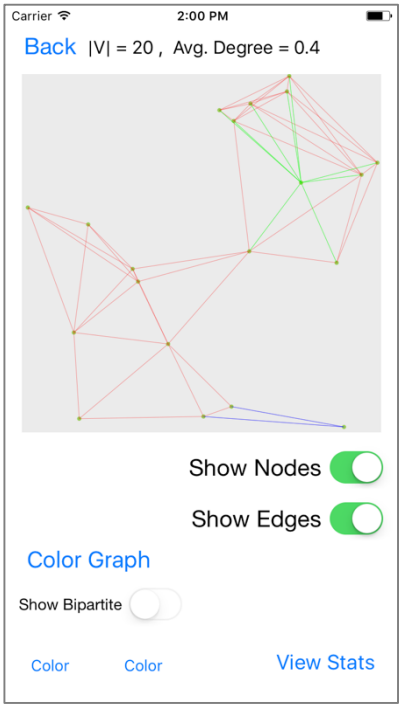
To prove that my algorithms work, I will go through the step-by-step process of calculating the smallest last ordering, coloring, and backbone bipartite subgraph. I printed the values at each step and will use those to show this verification walkthrough. This example will use a random geometric graph in the unit square with N=20 and R=0.40.

First I generated the 20 nodes randomly in the unit square. The node id's and x and y coordinates are printed below as well as a graphical representation of where the nodes are. Note that the x and y values are rounded for display.

```
0|(0.051,0.519)
1|(0.062,0.28)
2|(0.702,0.092)
3|(0.487,0.949)
4|(0.952,0.801)
5|(0.68,0.093)
6|(0.067,0.622)
7|(0.931,0.08)
8|(0.892,0.216)
9|(0.254,0.472)
10|(0.407,0.984)
11|(0.424,0.439)
12|(0.21,0.221)
13|(0.74,0.478)
14|(0.668,0.383)
15|(0.733,0.158)
16|(0.381,1.0)
17|(0.599,0.868)
18|(0.383,0.726)
19|(0.554,0.332)
```



I then connect the vertices if the distance between them is less than R. For this comparison, I use the distance formula: $d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. Displayed below is the resulting adjacency list showing just the id numbers of the vertices and an image of the connected graph.



Adjacency List

```
0 -> 1 -> 12 -> 11 -> 6 -> 9 -> 18 -> X
1 -> 0 -> 6 -> 9 -> 11 -> 12 -> X
2 -> 7 -> 8 -> 13 -> 5 -> 14 -> 15 -> 19 -> X
3 -> 18 -> 16 -> 10 -> 17 -> X
4 -> 13 -> 17 -> X
5 -> 7 -> 8 -> 13 -> 2 -> 14 -> 15 -> 19 -> X
6 -> 1 -> 0 -> 9 -> 18 -> X
7 -> 2 -> 5 -> 15 -> 8 -> X
8 -> 2 -> 5 -> 14 -> 15 -> 19 -> 13 -> 7 -> X
9 -> 1 -> 12 -> 11 -> 19 -> 0 -> 6 -> 18 -> X
10 -> 18 -> 16 -> 3 -> 17 -> X
11 -> 1 -> 12 -> 0 -> 9 -> 18 -> 14 -> 19 -> 13 -> X
12 -> 19 -> 0 -> 9 -> 11 -> 1 -> X
13 -> 2 -> 5 -> 14 -> 15 -> 19 -> 4 -> 8 -> 11 -> X
14 -> 8 -> 11 -> 13 -> 2 -> 5 -> 15 -> 19 -> X
15 -> 7 -> 8 -> 13 -> 2 -> 5 -> 14 -> 19 -> X
16 -> 18 -> 3 -> 10 -> 17 -> X
17 -> 18 -> 16 -> 4 -> 3 -> 10 -> X
18 -> 11 -> 16 -> 3 -> 10 -> 17 -> 0 -> 6 -> 9 -> X
19 -> 12 -> 9 -> 8 -> 11 -> 13 -> 2 -> 5 -> 14 -> 15 -> X
```

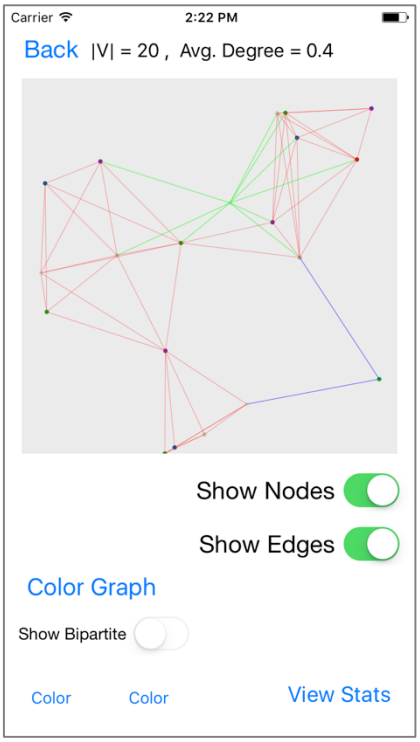
Then I implement the Smallest Last Ordering algorithm by first putting each node in a bucket based on its degree. Then I remove a node from the bucket with the lowest degree and update the bucket that each connecting node is in. For each iteration, I print which node id I remove and the current buckets list. Note that the index of the bucket in the buckets list represents the degree of the nodes in that list. I also print when I find the terminal clique using the conditions described above in PART 2.

```
[[], [], [4], [], [3, 6, 7, 10, 16], [1, 12, 17], [0], [2, 5, 8, 9, 14, 15], [11, 13, 18], [19], [], [], [], [], [],
[], [], [], []]

Removing node:  4
[[], [], [], [], [3, 6, 7, 10, 16, 17], [1, 12], [0], [2, 5, 8, 9, 14, 15, 13], [11, 18], [19], [], [], [], [], [], [],
[], [], []]

Removing node:  17
[[], [], [], [16, 3, 10], [6, 7], [1, 12], [0], [2, 5, 8, 9, 14, 15, 13, 18], [11], [19], [], [], [], [], [], [], [],
[], []]

Removing node:  10
[[], [], [16, 3], [], [6, 7], [1, 12], [0, 18], [2, 5, 8, 9, 14, 15, 13], [11], [19], [], [], [], [], [], [], [], [],
[]]

Removing node:  3
[[], [16], [], [], [6, 7], [1, 12, 18], [0], [2, 5, 8, 9, 14, 15, 13], [11], [19], [], [], [], [], [], [], [], [], []]

Removing node:  16
[[], [], [], [], [6, 7, 18], [1, 12], [0], [2, 5, 8, 9, 14, 15, 13], [11], [19], [], [], [], [], [], [], [], [], []]

Removing node:  18
[[], [], [], [6], [7], [1, 12, 0], [9], [2, 5, 8, 14, 15, 13, 11], [], [19], [], [], [], [], [], [], [], [], []]

Removing node:  6
[[], [], [], [], [7, 1, 0], [12, 9], [], [2, 5, 8, 14, 15, 13, 11], [], [19], [], [], [], [], [], [], [], [], []]

Removing node:  0
[[], [], [], [1], [7, 12, 9], [], [11], [2, 5, 8, 14, 15, 13], [], [19], [], [], [], [], [], [], [], [], []]

Removing node:  1
[[], [], [], [9, 12], [7], [11], [], [2, 5, 8, 14, 15, 13], [], [19], [], [], [], [], [], [], [], [], []]

Removing node:  12
[[], [], [9], [], [7, 11], [], [], [2, 5, 8, 14, 15, 13], [19], [], [], [], [], [], [], [], [], [], []]
```

```
Removing node:  9
[[], [], [], [11], [7], [], [], [2, 5, 8, 14, 15, 13, 19], [], [], [], [], [], [], [], [], [], []]

Removing node:  11
[[], [], [], [], [7], [], [14, 19, 13], [2, 5, 8, 15], [], [], [], [], [], [], [], [], [], [], []]

Removing node:  7
[[], [], [], [], [], [], [14, 19, 13, 2, 5, 15, 8], [], [], [], [], [], [], [], [], [], [], []]
Terminal Clique found: Size 7

Removing node:  8
[[], [], [], [], [], [2, 5, 14, 15, 19, 13], [], [], [], [], [], [], [], [], [], [], [], []]

Removing node:  13
[[], [], [], [], [2, 5, 14, 15, 19], [], [], [], [], [], [], [], [], [], [], [], [], []]

Removing node:  19
[[], [], [], [2, 5, 14, 15], [], [], [], [], [], [], [], [], [], [], [], [], [], []]

Removing node:  15
[[], [], [2, 5, 14], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]

Removing node:  14
[[], [2, 5], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]

Removing node:  5
[[2], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]

Removing node:  2
[[], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], [], []]
```

To color the vertices, I use a greedy approach, assigning the lowest possible color number to each vertex. The following print statements detail the coloring process. The display shows the graph after the coloring process.

```
COLORING NODE  2
Assigning node 2 the color: 0

COLORING NODE  5
Color 0 taken by adjacent node 2
Assigning node 5 the color: 1

COLORING NODE  14
Color 0 taken by adjacent node 2
Color 1 taken by adjacent node 5
Assigning node 14 the color: 2

COLORING NODE  15
Color 0 taken by adjacent node 2
Color 1 taken by adjacent node 5
Color 2 taken by adjacent node 14
Assigning node 15 the color: 3

COLORING NODE  19
Color 0 taken by adjacent node 2
Color 1 taken by adjacent node 5
Color 2 taken by adjacent node 14
Color 3 taken by adjacent node 15
Assigning node 19 the color: 4

COLORING NODE  13
Color 0 taken by adjacent node 2
Color 1 taken by adjacent node 5
Color 2 taken by adjacent node 14
Color 3 taken by adjacent node 15
Color 4 taken by adjacent node 19
Assigning node 13 the color: 5

COLORING NODE  8
Color 0 taken by adjacent node 2
Color 1 taken by adjacent node 5
Color 2 taken by adjacent node 14
Color 3 taken by adjacent node 15
Color 4 taken by adjacent node 19
Color 5 taken by adjacent node 13
Assigning node 8 the color: 6

COLORING NODE  7
Color 0 taken by adjacent node 2
Color 1 taken by adjacent node 5
Color 3 taken by adjacent node 15
Color 6 taken by adjacent node 8
Assigning node 7 the color: 2

COLORING NODE  11
Color 2 taken by adjacent node 14
Color 4 taken by adjacent node 19
Color 5 taken by adjacent node 13
Assigning node 11 the color: 0

COLORING NODE  9
Color 0 taken by adjacent node 11
Color 4 taken by adjacent node 19
Assigning node 9 the color: 1

COLORING NODE  12
Color 4 taken by adjacent node 19
Color 1 taken by adjacent node 9
Color 0 taken by adjacent node 11
Assigning node 12 the color: 2

COLORING NODE  1
Color 1 taken by adjacent node 9
Color 0 taken by adjacent node 11
Color 2 taken by adjacent node 12
Assigning node 1 the color: 3

COLORING NODE  0
Color 3 taken by adjacent node 1
Color 2 taken by adjacent node 12
Color 0 taken by adjacent node 11
Color 1 taken by adjacent node 9
Assigning node 0 the color: 4

COLORING NODE  6
Color 3 taken by adjacent node 1
Color 4 taken by adjacent node 0
Color 1 taken by adjacent node 9
Assigning node 6 the color: 0

COLORING NODE  18
Color 0 taken by adjacent node 11
Color 4 taken by adjacent node 0
Color 0 taken by adjacent node 6
Color 1 taken by adjacent node 9
Assigning node 18 the color: 2

COLORING NODE  16
Color 2 taken by adjacent node 18
Assigning node 16 the color: 0

COLORING NODE  3
Color 2 taken by adjacent node 18
Color 0 taken by adjacent node 16
Assigning node 3 the color: 1

COLORING NODE  10
Color 2 taken by adjacent node 18
Color 0 taken by adjacent node 16
Color 1 taken by adjacent node 3
Assigning node 10 the color: 3

COLORING NODE  17
Color 2 taken by adjacent node 18
Color 0 taken by adjacent node 16
Color 1 taken by adjacent node 3
Color 3 taken by adjacent node 10
Assigning node 17 the color: 4

COLORING NODE  4
Color 5 taken by adjacent node 13
Color 4 taken by adjacent node 17
Assigning node 4 the color: 0
```

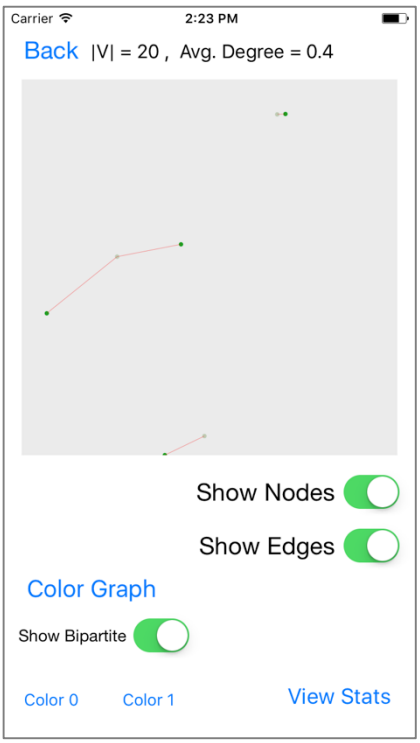


To show the backbone, I first determine the bipartite subgraph. To do this, I select the first two colors (color 0 and color 1) and select the edges that are connected to both colors. The nodes connected to these edges will create the bipartite subgraph. Here are the nodes for the bipartite subgraph.

```
Nodes for Bipartite Graph:
9|(0.254,0.472)
11|(0.424,0.439)
6|(0.067,0.622)
16|(0.381,1.0)
3|(0.487,0.949)
2|(0.702,0.092)
5|(0.68,0.093)
```

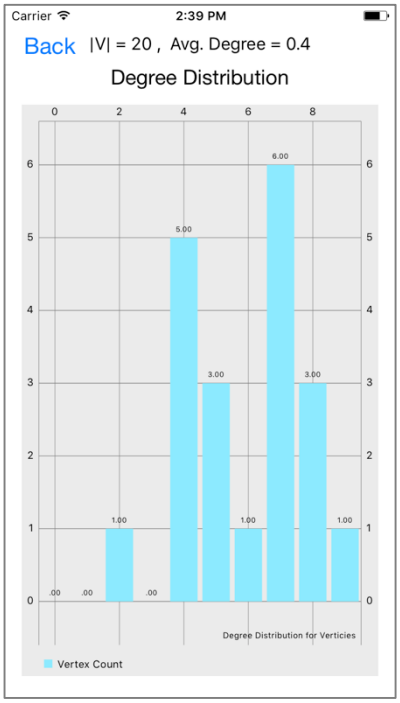
To determine the backbone, I organize the nodes into components. I iterate through the edges and assign both nodes to the same component. If only one of the nodes is in a component, then I assign the other to that component as well. If both nodes are already in a component, then I simply increment the edge-count.

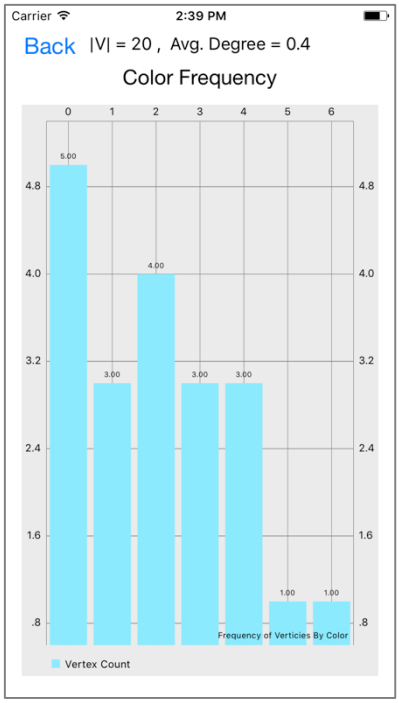
```
Making Components
[]
[([9, 11], 1)]
[([9, 11, 6], 2)]
[([9, 11, 6], 2), ([16, 3], 1)]
[([9, 11, 6], 2), ([16, 3], 1), ([2, 5], 1)]

Backbone:
([9, 11, 6], 2)
```

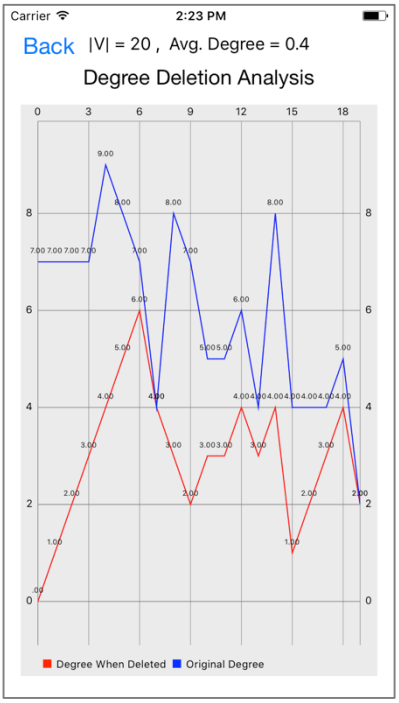


The display to the right shows the bipartite subgraph made by color 0 and color 1. The backbone found above can easily be seen and verified by looking at the display.

The following charts show that the degree distribution, color frequency, and the degree deletion analysis. The degree distribution can help the user understand the frequencies of each degree and how they compare to each other. The color frequency shows how many vertices are in each color. This can also help the use understand the distribution of how the colors ended up being assigned. Finally, the degree deletion analysis shows a node's original degree and the degree it had when it was deleted with respect to the Smallest Last Ordering. This chart can also tell the user how many components the graph was split into during this process.







Overall, the values in these charts and the ones figured out by running the algorithms are consistent with what should be expected. The lowest degree is 2, which can be seen in the graphical representations of the network by observing the blue edges. The highest degree is 9, which can be seen in the graphical representation by observing the green edges. The degree deletion analysis chart indicates that the graph never split into multiple components during the Smallest Last Ordering process. The color frequency chart is consistent with what we might think because the first few colors are used the most and the last few are used much less frequently. The results of this verification walkthrough show that the implementations of my algorithms work to achieve correct and desired results. The graphical displays help the user understand and verify the results. In the next section, I will continue to analyze the data gathered by running the benchmarks for this project.

Benchmark Result Summary

Testing Process

To test each benchmark, I restart the program, set display to "off", choose the number of nodes for the benchmark, choose the average degree (which will calculate the connection distance), and then click generate. The graph will generate fully before moving to the next view. On the display view, I press the "Color Graph" button to run all the algorithms associated with graph coloring and some bipartite graph statistics. Once the button returns to its original color, I know the algorithms are done and I press the "Statistics" button to view the calculated statistics. The Table View includes basic network statistics, statistics about the largest two bipartite graphs, and buttons to the charts for each network. The Xcode console prints the statistics used for the benchmark comparisons so I copy those into an Excel document for later analysis.

This keep this testing procedure the same for every benchmark to ensure as much as possible that only the code affects the timing data. After all the timing data is collected, view each chart and screenshot it for this document. I then display and screenshot the network graph and a bipartite subgraph for networks with up to 16000 vertices.

Results

Based on the given benchmarks given for this assignment, Tables 1, 2, and 4 show the summary values calculated from running the application. Table 3 shows what the input values are for each benchmark. The benchmark numbers used in the other tables reflect the values in this table.

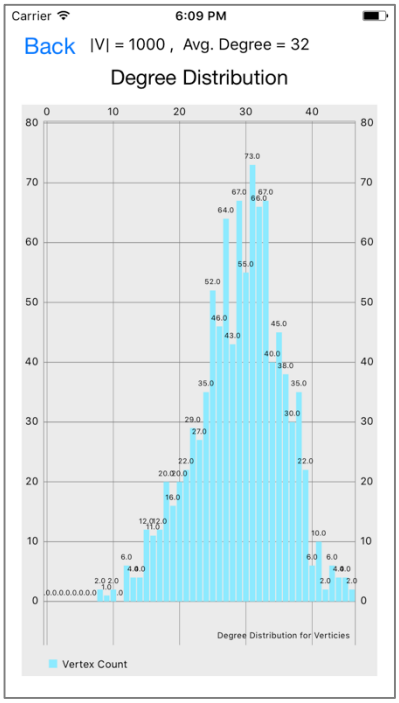
TABLE 3: BENCHMARK INPUT VALUES

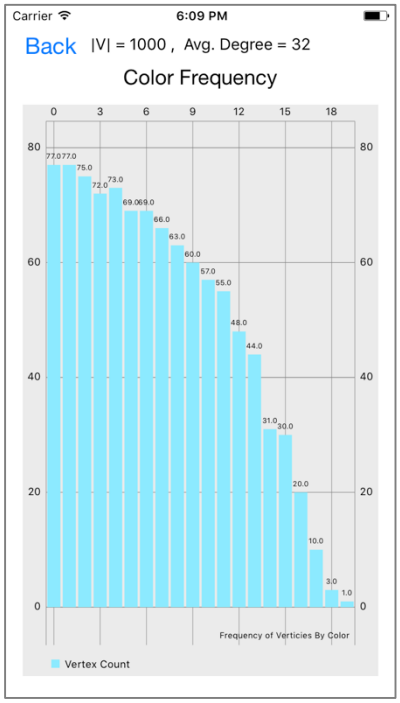
Benchmark #	N	Avg. Degree	Distribution
1	1000	32	Square
2	4000	64	Square
3	16000	64	Square
4	64000	64	Square
5	64000	128	Square
6	4000	64	Disk
7	4000	128	Disk
8	4000	64	Sphere
9	16000	128	Sphere
10	64000	128	Sphere

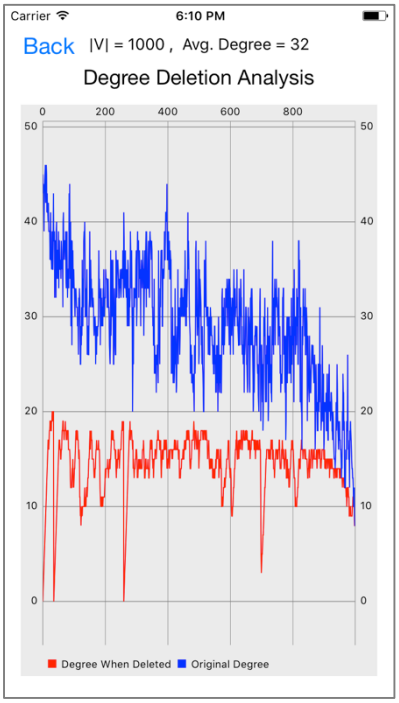
Table 4 shows the timing data for running the benchmarks. The data was analyzed in previous sections using charts and discussion. The "Graph Creation Time" is the number of seconds that it took to generate the vertices and edges of the graph but not display them. The "Smallest Last Ordering Time" is the number of seconds that it took to order the vertices using the smallest last ordering algorithm. The "Coloring Time" is the number of seconds that it took to actually assign colors to each vertex using the ordered adjacency list.

TABLE 4: TIMING DATA

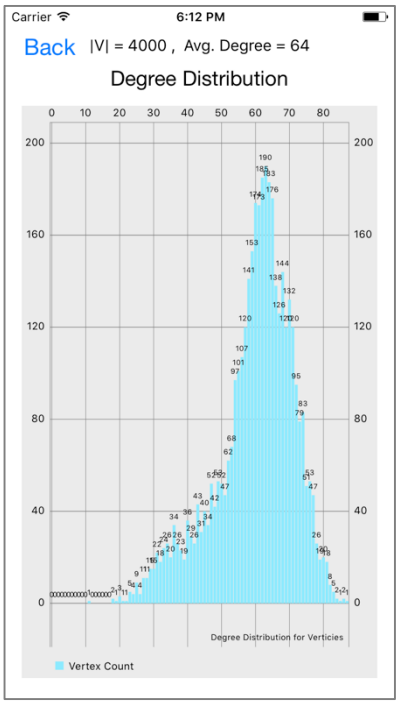
Benchmark #	Graph Creation Time (s)	Smallest Last Ordering Time (s)	Coloring Time (s)	Bipartite Stats Time (s)
1	0.044	0.137	0.05	0.159
2	0.34	1.6	0.532	1.029
3	1.416	21.295	2.362	8.44
4	5.816	337.722	9.514	101.776
5	10.915	320.973	26.921	50.636
6	0.328	1.654	0.539	1.052
7	0.646	1.964	1.66	1.231
8	0.472	1.869	0.51	0.975
9	3.547	26.196	6.272	6.8
10	14.775	300.275	26.571	51.339

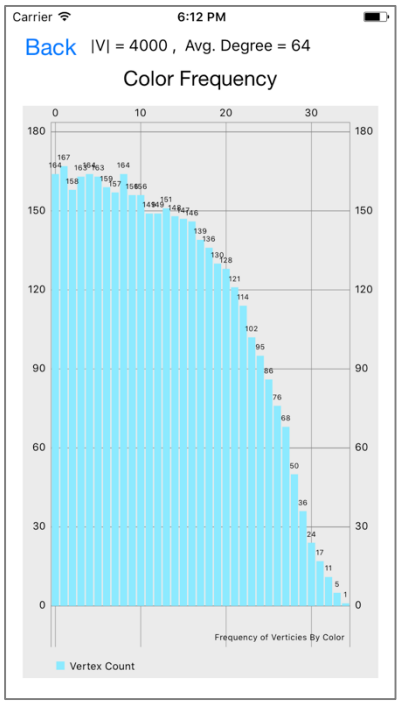


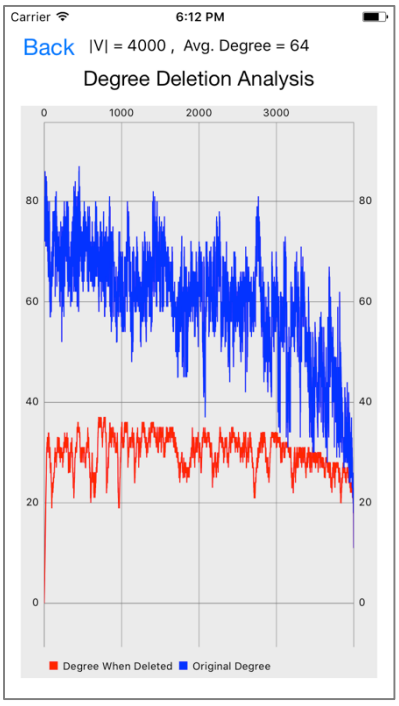




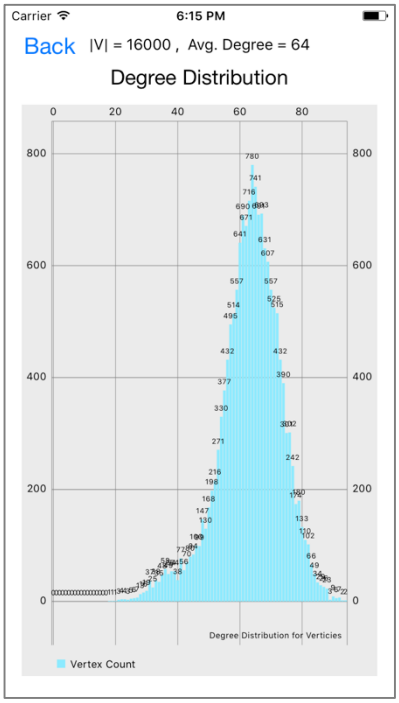
Benchmark 1: The degree distribution is skewed left with a low vertex-count and a decent average degree. It appears the graph split into 3 components for deletion during the Smallest Last Ordering.

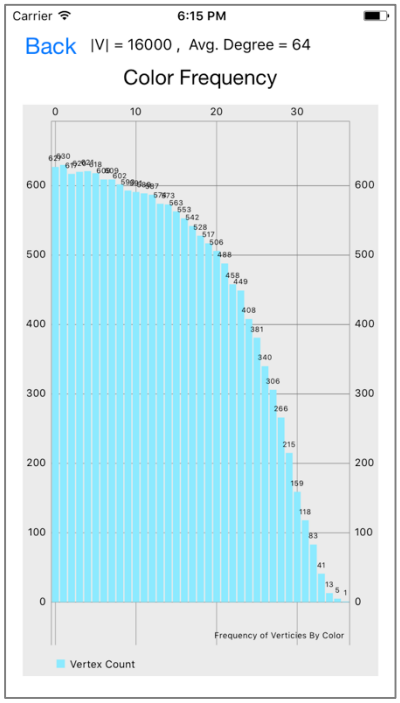


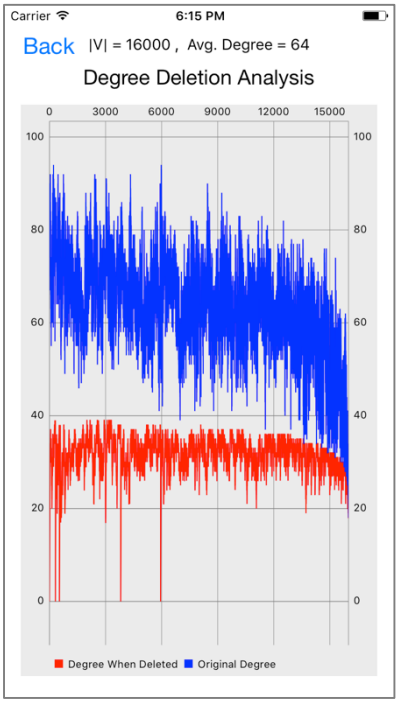




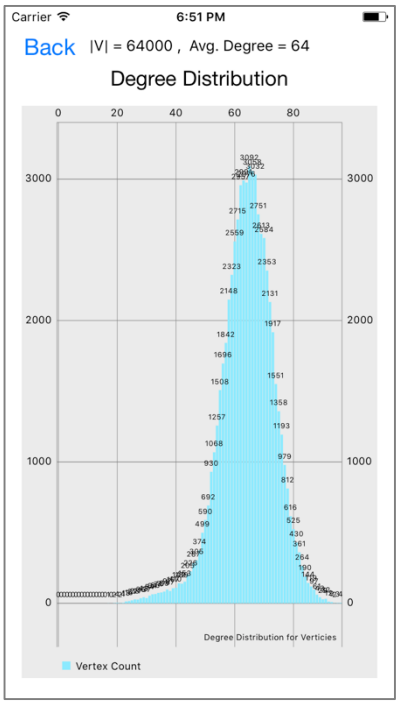
Benchmark 2: The degree distribution is skewed left with a high average degree for a lower vertex-count. Interestingly, due to this, the degree deletion analysis shows that the graph stayed in one component for the deletion process.

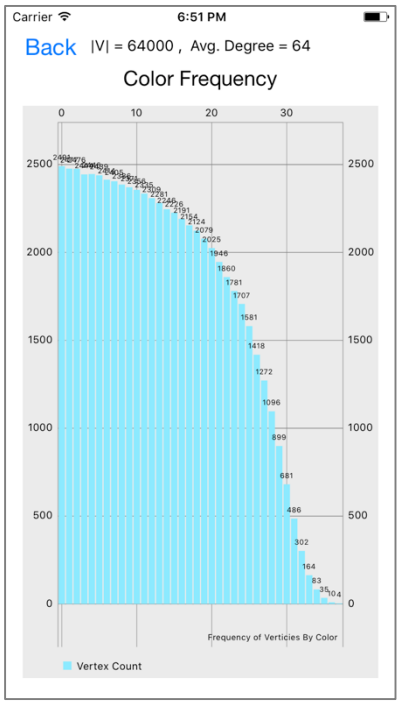


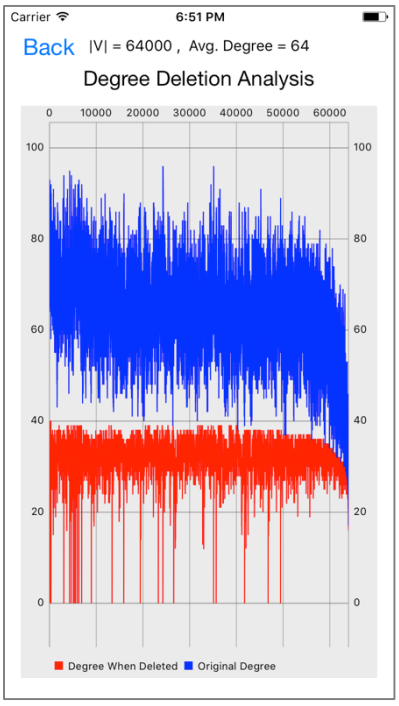




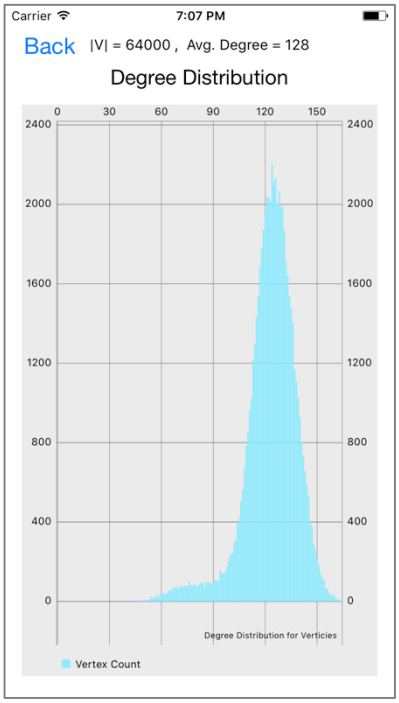
Benchmark 3: The degree distribution is less skewed due to the higher vertex-count. The degree deletion analysis shows that the graph split into 5 components during the Smallest Last Ordering.

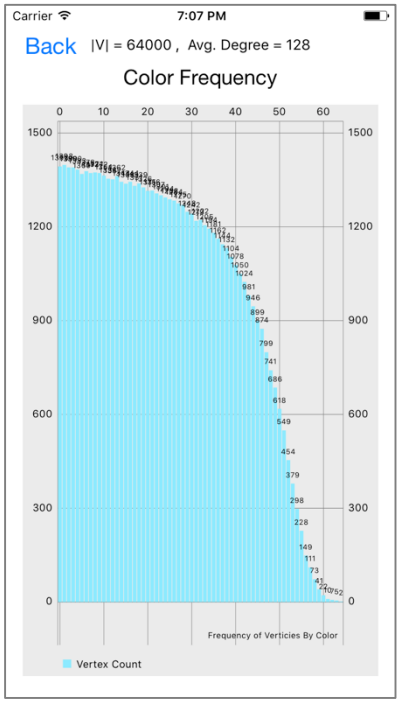


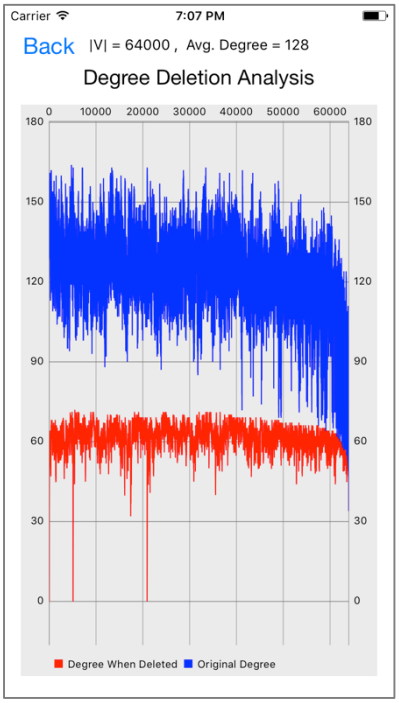




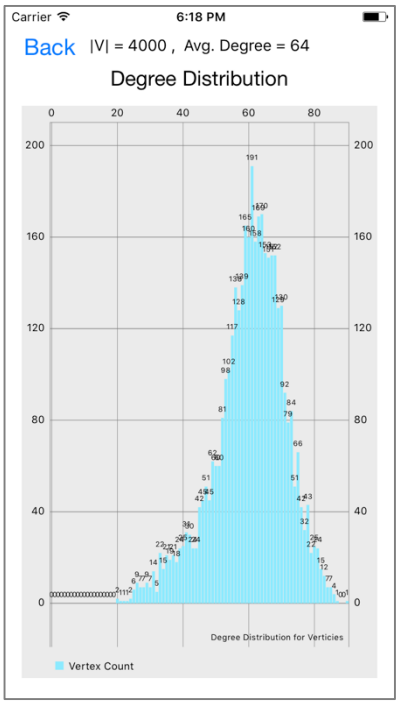
Benchmark 4: The degree distribution is much less skewed with a normal distribution with a median of 65. The degree deletion analysis shows that the graph split into many parts during the Smallest Last Ordering. This seems to be an anomaly compared to the other benchmarks.

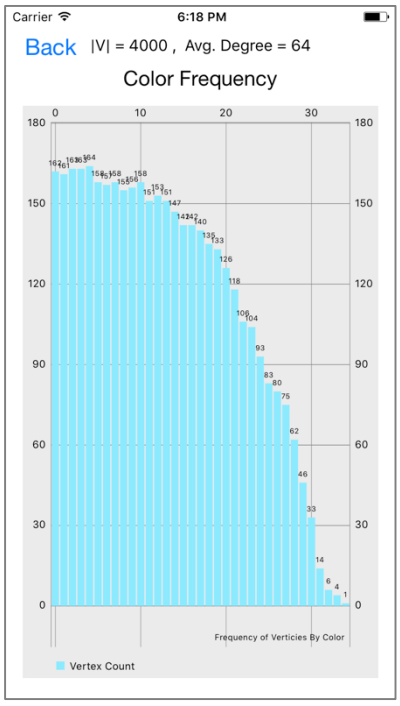


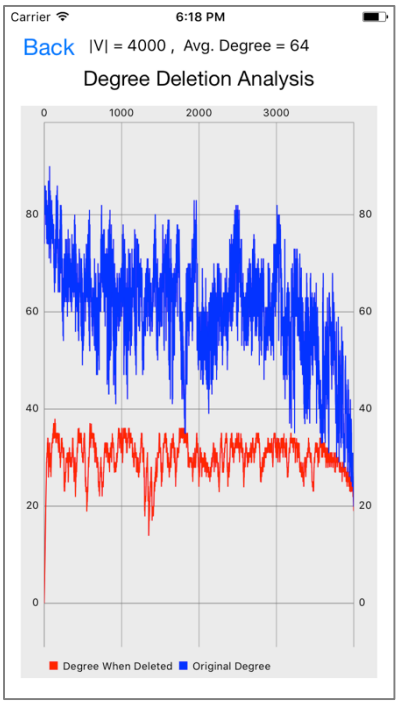




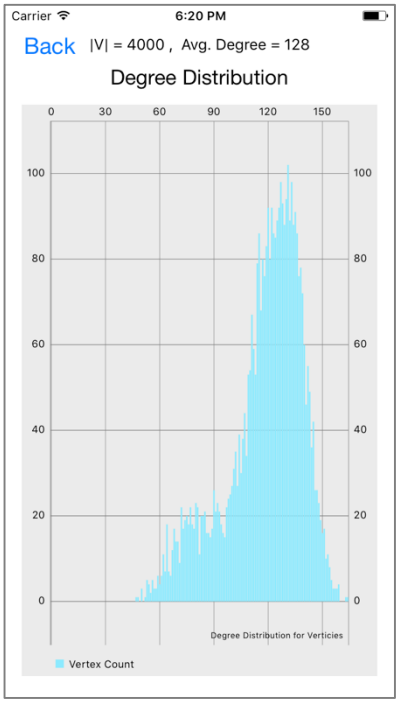
Benchmark 5: The degree distribution is normal with a slight left skew. The degree deletion analysis shows that the graph split into about 3 parts during the Smallest Last Ordering.

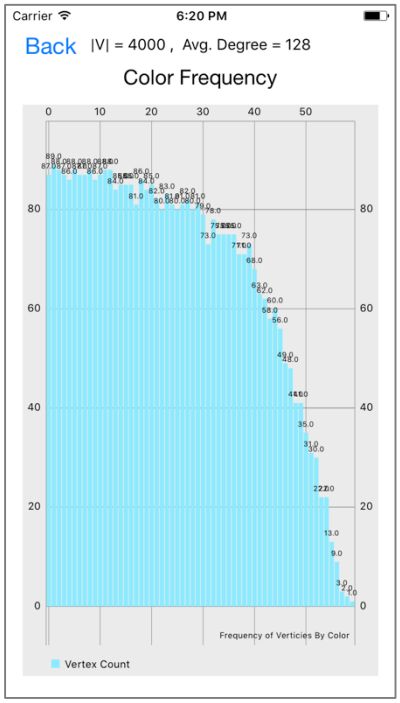


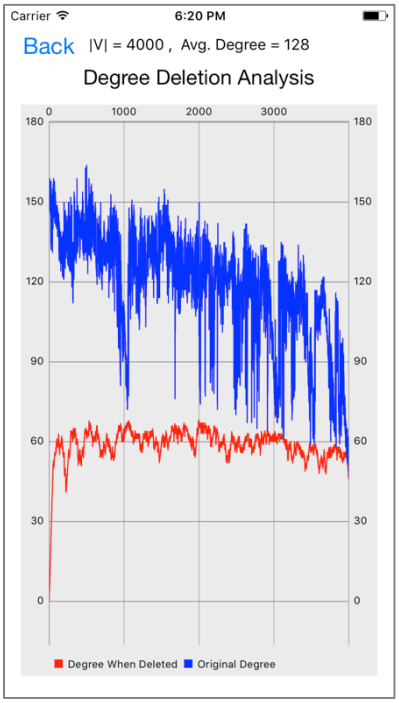




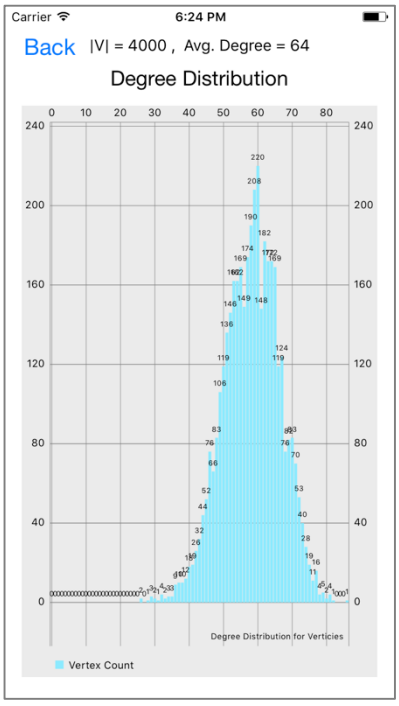
Benchmark 6: The degree distribution is skewed left. The degree deletion analysis shows that the graph was in one component during the entire Smallest Last Ordering process.

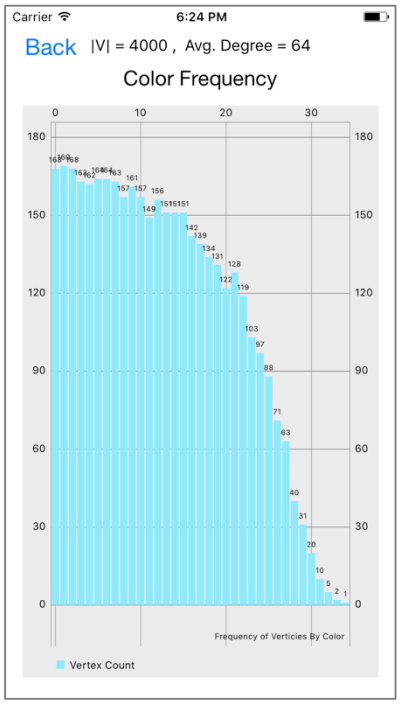


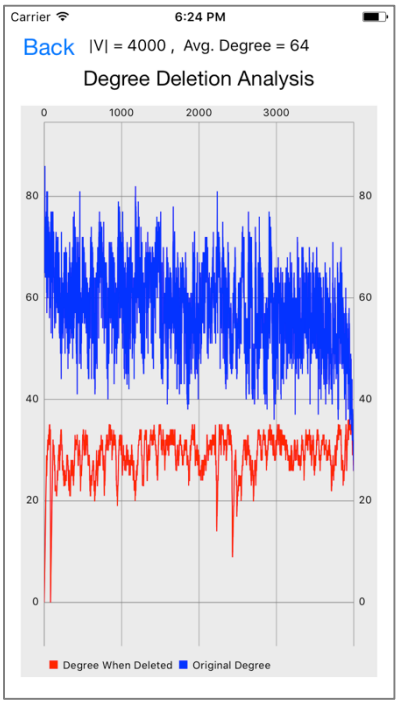




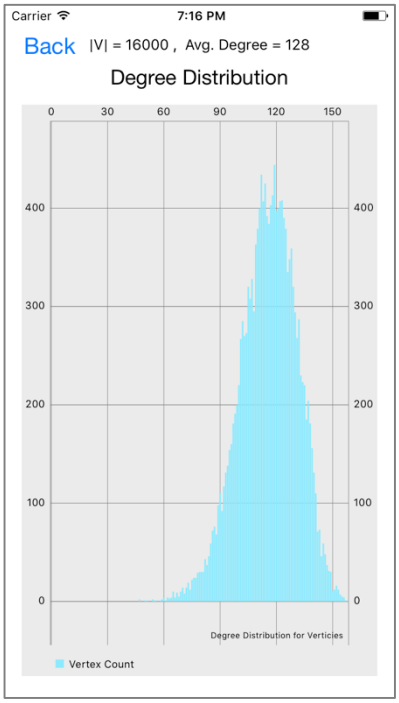
Benchmark 7: The degree distribution is skewed left almost with a small bimodal distribution. This could be due to the many vertices on the edge of the disk with an inevitable smaller degree. The degree deletion analysis shows that the graph was in one component during the entire Smallest Last Ordering process.

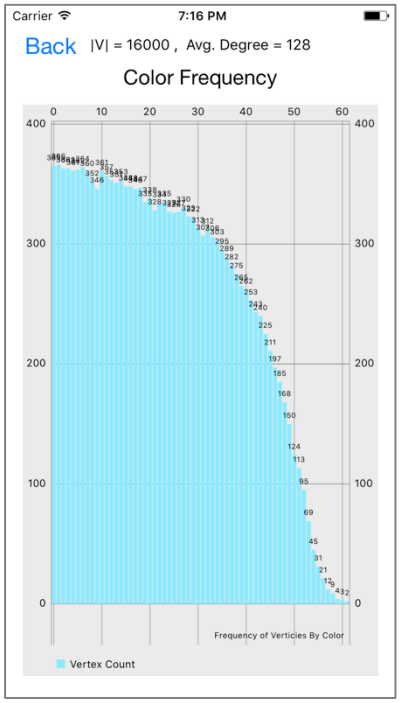


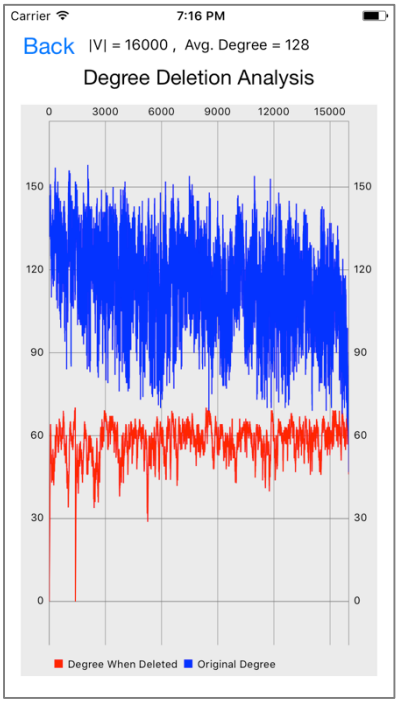




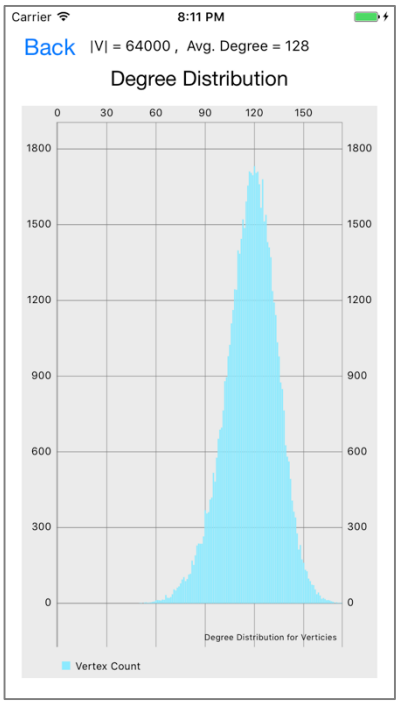
Benchmark 8: The degree distribution is normal with a median of about 60. According to the degree deletion analysis, the graph split into about 2 components during the Smallest Last Ordering.

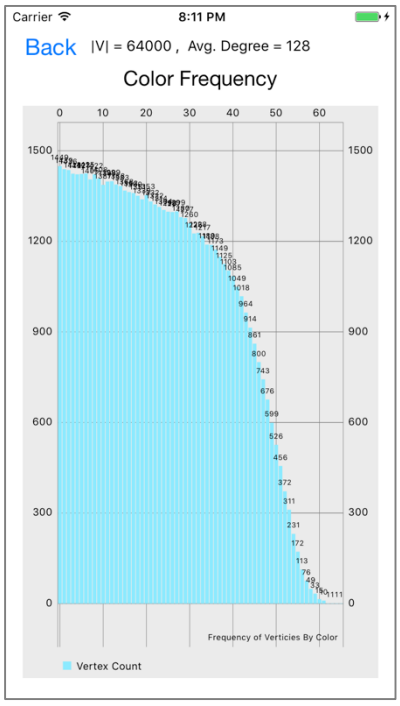


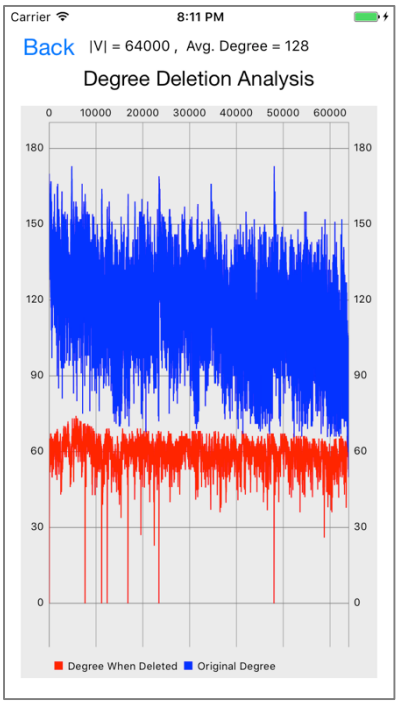




Benchmark 9: The degree distribution is normal with a median of about 120. According to the degree deletion analysis, the graph split into about 2 components during the Smallest Last Ordering.







Benchmark 10: The degree distribution is normal with a median of about 120. According to the degree deletion analysis, the graph split into about 7 components during the Smallest Last Ordering.

ANALYSIS

Overall, these charts show interesting characteristics about the wireless sensor networks. The square and disk network models have degree distributions slightly skewed to the left. The cause of this is probably due to the vertices closer to the border of the unit space that have fewer connection possibilities. This skewedness disappeared in the sphere degree distribution because there is no border on the surface of the sphere. All the color frequency charts followed an expected distribution having most of the first few colors with about the same frequency and then eventually dropping off quickly.

The degree deletion analysis chart can show how many components that the graph split into during the Smallest Last Ordering determination. When nodes are being deleted, it is possible that the graph splits and then eventually one component will finish deleting completely before the others. When this happens, the degree when deleted drops down to zero. By counting how many times the degree drops down to zero, we can see how many components the graph split into during the Smallest Last Ordering process. It's interesting to note that the number of color classes is about half that of the average degree for every benchmark. This could be a property used in evaluating if the graph coloring was done efficiently (like with the Smallest Last Ordering) or simply greedily (random or irrelevant ordering).

Some of the medians are slightly lower than expected based on the average degree. For the disk and square, this can be explained by taking into account the vertices close to the border. These vertices do not have as much area around them to connect to other vertices, so their degree will be smaller. Although this mostly affects the average degree realized, the perceived median degree can also be skewed. For the sphere, however, there are no borders. The lower median is possibly caused by a rounding error when calculating the R value or when determining which cell to put each vertex in. Regardless of this minor difference from the expected result, the rest of the calculations and statistics holds value for analysis.

Benchmark 4 seemed to be an anomaly on many accounts: it was an outlier for the Smallest Last Ordering timing, it was an outlier for the bipartite statistics timing, and it was an outlier in the number of components that the graph split into during the Smallest Last Ordering process. It is unknown what caused these anomalies but it is interesting that they all occurred for this benchmark and that there were no other great anomalies in any other benchmark.

With regards to Table 1 and Table 2, all the values determined are reasonable and about what we might expect. The average degree realized is about the same as the average degree given as input, the number of edges is equal to the average degree realized multiplied by the number of vertices, the number of edges in the max backbone is about two-and-a-half times the max color class size, and the domination percentages of all the backbones are between 99.5% and 100%.

In conclusion, the results calculated from the algorithms seem to be consistent with what they should be with the exception of some minor degree differences. The graph generation was implemented very efficiently using the cell method described in Matula's lecture. The Smallest Last Ordering was implemented effectively, but maybe not in the most efficient way. The issue here is not completely clear, but it is likely the cause of Swift's inability to implement proper pointers, causing memory problems. The graph coloring was implemented in linear time using a greedy algorithm. The bipartite graph statistics were implemented effectively for most of the algorithms, but could probably be fine-tuned in order to eradicate outlier issues as seen from Benchmark 4. By implementing the entirety of the project in an iOS application, I created a way for people to begin understanding some of the algorithmic complexity of some of the characteristics of wireless sensor networks.

The code base for this project can be found at *https://github.com/Tsiems/WirelessNetwork*.

APPENDIXES

APPENDIX 1: MENUVIEWCONTROLLER.SWIFT

```
//
//  MenuViewController.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 2/24/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import UIKit

class MenuViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()

        self.nodeNumberLabel.text = String(Int(self.nodeNumberSlider.value))
        self.adjacencyLabel.text = String((self.adjacencySlider.value))

        //Looks for single or multiple taps.
        let tap: UITapGestureRecognizer = UITapGestureRecognizer(target: self, action:
#selector(MenuViewController.dismissKeyboard))

        view.addGestureRecognizer(tap)

        self.displayNetworkSwitch.isOn = false
        self.adjacencyTypeControl.selectedSegmentIndex = 1
        displayAdjacencyValue()

        self.title = "Wireless Sensor Network Generator"
    }

    //Calls this function when the tap is recognized.
    func dismissKeyboard() {
        //Causes the view (or one of its embedded text fields) to resign the first responder status.
        view.endEditing(true)

        if let amt = Float(self.adjacencyLabel.text!){
            if self.adjacencyTypeControl.selectedSegmentIndex == 0 && amt <
self.adjacencySlider.maximumValue {
                self.adjacencySlider.value = amt
            } else if self.adjacencyTypeControl.selectedSegmentIndex == 1 {

                let nodeCount = Int(self.nodeNumberSlider.value)
                var r:Float = 0.0
                switch networkModelControl.titleForSegment(at:
networkModelControl.selectedSegmentIndex)! {
                case "Square":
                    r = sqrt(amt/Float(nodeCount)/Float.pi)
                case "Disk":
                    r = sqrt(amt/Float(nodeCount))
                case "Sphere":
                    r = sqrt(amt/Float(nodeCount))
                default:
                    r = sqrt(amt/Float(nodeCount)/Float.pi)
                }

                if r < self.adjacencySlider.maximumValue && r != self.adjacencySlider.value {
                    self.adjacencySlider.value = r
                    self.newValuesSwitch.isOn = true
                }
                else {
                    displayAdjacencyValue()
                }
            }
            else {
                displayAdjacencyValue()
            }
```

```
    } else {
        displayAdjacencyValue()
    }


    if let amt = Float(self.nodeNumberLabel.text!) {
        if amt < self.nodeNumberSlider.maximumValue {
            if self.nodeNumberSlider.value != amt {
                self.newValuesSwitch.isOn = true
                self.nodeNumberSlider.value = amt
                displayAdjacencyValue()
            }
        }
        else {
            self.nodeNumberLabel.text = String(Int(self.nodeNumberSlider.value))
        }
    }
    else {
        self.nodeNumberLabel.text = String(Int(self.nodeNumberSlider.value))
    }
}

override func viewDidAppear(_ animated: Bool) {
    self.newValuesSwitch.isEnabled = CURRENT_NODES.count > 0
    self.newValuesSwitch.isOn = CURRENT_NODES.count == 0
}

override func didReceiveMemoryWarning() {
    super.didReceiveMemoryWarning()
    // Dispose of any resources that can be recreated.
}

@IBOutlet weak var networkModelControl: UISegmentedControl!
@IBOutlet weak var nodeNumberSlider: UISlider!

@IBOutlet weak var nodeNumberLabel: UITextField!
@IBOutlet weak var newValuesSwitch: UISwitch!

@IBOutlet weak var adjacencySlider: UISlider!

@IBOutlet weak var adjacencyLabel: UITextField!
@IBOutlet weak var adjacencyTypeControl: UISegmentedControl!

@IBOutlet weak var displayNetworkSwitch: UISwitch!
@IBOutlet weak var adjacencyTypeLabel: UILabel!
@IBAction func newValuesSwitchChanged(_ sender: UISwitch) {

    //reset selected nodes to the number that will actually be generated
    if !self.newValuesSwitch.isOn {
        self.nodeNumberSlider.value = Float(CURRENT_NODES.count)
        self.nodeNumberLabel.text = String(Int(self.nodeNumberSlider.value))
        self.adjacencySlider.value = Float(CURRENT_CONNECTION_DISTANCE)
        displayAdjacencyValue()
    }
}
@IBAction func networkModelChanged(_ sender: UISegmentedControl) {
    //reset selected nodes to the number that will actually be generated
    self.nodeNumberLabel.text = String(Int(self.nodeNumberSlider.value))
    displayAdjacencyValue()
    self.newValuesSwitch.isOn = true
}

@IBAction func nodeNumberSliderValueChanged(_ sender: Any) {
    self.nodeNumberLabel.text = String(Int(self.nodeNumberSlider.value))
    displayAdjacencyValue()
    self.newValuesSwitch.isOn = true

    self.displayNetworkSwitch.isOn = Int(self.nodeNumberSlider.value) <= 16000
}

func displayAdjacencyValue() {
    let value = round(1000*self.adjacencySlider.value)/1000
    if self.adjacencyTypeControl.selectedSegmentIndex == 0 {
        self.adjacencyLabel.text = String(value)
```

```
            self.adjacencyTypeLabel.text = "R (distance for adjacency)"
        } else if self.adjacencyTypeControl.selectedSegmentIndex == 1 {
            let nodeCount = Int(self.nodeNumberSlider.value)
            var avgDegree:Float = 0.0
            switch networkModelControl.titleForSegment(at: networkModelControl.selectedSegmentIndex)! {
            case "Square":
                avgDegree = Float(nodeCount)*Float.pi*value*value
            case "Disk":
                avgDegree = Float(nodeCount)*value*value
            case "Sphere":
                avgDegree = Float(nodeCount)*value*value
            default:
                avgDegree = Float(nodeCount)*Float.pi*value*value
            }
            self.adjacencyLabel.text = String( round(1000*avgDegree)/1000 )
            self.adjacencyTypeLabel.text = "Avg (average degree)"
        }
    }

    @IBAction func adjacencySliderChanged(_ sender: UISlider) {
        displayAdjacencyValue()
        self.newValuesSwitch.isOn = true
    }
    @IBAction func adjacencyTypeChanged(_ sender: UISegmentedControl) {
        displayAdjacencyValue()
    }

    @IBAction func unwindToMenu(segue: UIStoryboardSegue) {
        print("Back at menu")
    }

    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before
navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
        if segue.identifier == "displayNetworkSegue" {
            let vc = segue.destination as! DisplayViewController
            vc.networkModel = networkModelControl.titleForSegment(at:
networkModelControl.selectedSegmentIndex)!
            vc.nodeCount = Int(self.nodeNumberSlider.value)
            vc.connectionDistance = Double(round(1000*self.adjacencySlider.value)/1000)
            if vc.networkModel == "Square" {
                vc.averageDegree =
Double(vc.nodeCount)*Double.pi*vc.connectionDistance*vc.connectionDistance
            } else if vc.networkModel == "Disk" {
                vc.averageDegree = Double(vc.nodeCount)*vc.connectionDistance*vc.connectionDistance
                vc.connectionDistance = vc.connectionDistance/2
            } else if vc.networkModel == "Sphere" {
                vc.averageDegree = Double(vc.nodeCount)*vc.connectionDistance*vc.connectionDistance/4
                vc.connectionDistance = vc.connectionDistance*2
            }

            vc.shouldGenerateNewValues = self.newValuesSwitch.isOn

            vc.shouldShowEdges = self.displayNetworkSwitch.isOn
            vc.shouldShowNodes = self.displayNetworkSwitch.isOn

            vc.title = String(vc.nodeCount)+" vertices, "+String(vc.averageDegree)+" Avg. Degree"
            vc.title = "|V| = "+String(vc.nodeCount)+" ,  Avg. Degree = "+self.adjacencyLabel.text!
        }
    }

}
```

Appendix 2: DisplayViewController.swift

```
//
//  DisplayViewController.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 2/24/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import UIKit
import SceneKit

var CURRENT_NODES:[Node] = []
var CURRENT_EDGES:[Edge] = []
var CURRENT_CONNECTION_DISTANCE = 0.08
var TIME_TO_CREATE_GRAPH = 0.0
var TIME_TO_COLOR_GRAPH = 0.0
var TIME_FOR_SMALLEST_LAST_ORDERING = 0.0
var TIME_FOR_BIPARTITE = 0.0
var CURRENT_ADJACENCY_LIST:[[Node]] = []
var CURRENT_COLORS_ASSIGNED:[Int] = []
var CURRENT_COLOR_FREQUENCIES:[Int:Int] = [:]
var CURRENT_COLOR_FREQUENCIES_PAIRED:[(Int,Int)] = []
var COLORS:[Int] = []
var DEGREE_WHEN_DELETED:[Int] = []
var ORIGINAL_DEGREE:[Int] = []
var TERMINAL_CLIQUE_SIZE = 1

let VERIFICATION_WALKTHROUGH = false


func getRandomDouble() -> Double {
    return Double(Float(arc4random()) / Float(UINT32_MAX))
}

func getRandomFloat() -> Float {
    return Float(arc4random()) / Float(UINT32_MAX)
}

class DisplayViewController: UIViewController,UIPickerViewDelegate, UIPickerViewDataSource  {

    @IBOutlet weak var titleLabel: UILabel!

    var networkModel = "Square"
    var nodeCount = 128 // will be changed by the slider
    var connectionDistance = 0.075
    var averageDegree = 6.0

    var sphereNodes:[Node] = []
    var sphereEdges:[Edge] = []
    var extremeDegreeEdges:[Edge] = []

    var shouldShowNodes = true
    var shouldShowEdges = true
    var shouldGenerateNewValues = true
    var node_size = 2.0
    var edge_width = 0.2

    var colorStats:[(String,String)] = []
    var bipartiteStats:[(String,String)] = []

    var graphAdjList:[[Node]] = []

    @IBOutlet weak var sceneKitView: SCNView!
    @IBOutlet weak var drawView: DisplayView!
    @IBOutlet weak var showNodesSwitch: UISwitch!
    @IBOutlet weak var showEdgesSwitch: UISwitch!

    @IBOutlet weak var showBipartiteSwitch: UISwitch!

    override func viewDidLoad() {
        super.viewDidLoad()
```

```
        self.titleLabel.text = self.title
        showNodesSwitch.isOn = shouldShowNodes
        showEdgesSwitch.isOn = shouldShowEdges

        self.colorPicker.dataSource = self
        self.colorPicker.delegate = self

        self.showBipartiteSwitch.isEnabled = color1 != color2 && color1 != -1 && color2 != -1


        CURRENT_CONNECTION_DISTANCE = connectionDistance
        TERMINAL_CLIQUE_SIZE = 1

        let startTime = Date()
        if shouldGenerateNewValues {
            if networkModel == "Disk" {
                let nodes = generateRandomNodesInDisk(num: nodeCount)
                CURRENT_NODES = nodes

                let edges:[Edge] = generateEdgesUsingCellMethod(nodes: CURRENT_NODES, r:
connectionDistance)
                CURRENT_EDGES = edges

                self.drawView.isHidden = false
                self.sceneKitView.isHidden = true

            } else if networkModel == "Sphere" {
                let nodes = generateRandomNodesInSphere(num: nodeCount)
                CURRENT_NODES = nodes

                let edges = generateEdgesUsingCellMethod3d(nodes: CURRENT_NODES, r: connectionDistance)

                CURRENT_EDGES = edges

                sphereEdges = edges
                sphereNodes = nodes
            }
            else {
                let nodes = generateRandomNodesInSquare(num: nodeCount)
                CURRENT_NODES = nodes

                let edges:[Edge] = generateEdgesUsingCellMethod(nodes: CURRENT_NODES, r:
connectionDistance)
                CURRENT_EDGES = edges


                self.drawView.isHidden = false
                self.sceneKitView.isHidden = true
            }


        }

        let graphCreatedTime = Date()

        TIME_TO_CREATE_GRAPH = graphCreatedTime.timeIntervalSince(startTime)

        self.graphAdjList = getAdjacencyList(nodes: CURRENT_NODES, edges: CURRENT_EDGES)
        CURRENT_ADJACENCY_LIST = self.graphAdjList

        if VERIFICATION_WALKTHROUGH {
            print("\nAdjacency List")
            for row in CURRENT_ADJACENCY_LIST {
                for item in row {
                    print(item.id,terminator:" -> ")
                }
                print("X")
            }
            print()
        }


        let minMaxDegreeIds = findMinAndMaxDegreeNodes()
```

```
        extremeDegreeEdges = []
        for e in CURRENT_EDGES {
            if e.node1.id == minMaxDegreeIds.0 || e.node2.id == minMaxDegreeIds.0 {
                let edge = e
                edge.color = UIColor.blue
                extremeDegreeEdges.append(edge)
            }
            else if e.node1.id == minMaxDegreeIds.1 || e.node2.id == minMaxDegreeIds.1 {
                let edge = e
                edge.color = UIColor.green
                extremeDegreeEdges.append(edge)
            }
        }



        if networkModel == "Sphere" {
            displaySphereNetwork()

        } else {
            drawView.nodes = CURRENT_NODES
            drawView.edges = CURRENT_EDGES
            drawView.NODE_SIZE = self.node_size
            drawView.EDGE_WIDTH = self.edge_width
            drawView.shouldShowNodes = self.shouldShowNodes
            drawView.shouldShowEdges = self.shouldShowEdges
            drawView.extremeDegreeEdges = extremeDegreeEdges
            drawView.model = networkModel
        }




    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    @IBAction func backButtonPressed(_ sender: Any) {
        print("Going to menu")
        self.performSegue(withIdentifier: "unwindToMenu", sender: self)
    }

    func resetSphereNetwork() {
        self.sceneKitView.scene = SphereScene()
    }

    func displaySphereNetwork() {
        self.drawView.isHidden = true
        self.sceneKitView.isHidden = false

        self.sceneKitView.scene =
SphereScene(nodes:sphereNodes,edges:sphereEdges,extremeEdges:extremeDegreeEdges,shouldShowNodes:shouldS
howNodes,shouldShowEdges:shouldShowEdges,shouldShowExtremeEdges: !self.showBipartiteSwitch.isOn )

        self.sceneKitView.backgroundColor = UIColor.black
        self.sceneKitView.allowsCameraControl = CURRENT_NODES.count < 7000
    }


    func generateRandomNodesInSquare(num:Int) -> [Node]{
        var nodes:[Node] = []
        var current_id = 0
        for _ in 0 ..< num {
            let x = getRandomDouble()
            let y = getRandomDouble()
            let node = Node(x:x,y:y, id: current_id, color: 0 )

            nodes.append( node )
            current_id += 1
        }
        if VERIFICATION_WALKTHROUGH {
```

```
            for node in nodes {
                print(node)
            }
        }
        return nodes
    }

    func generateRandomNodesInDisk(num:Int) -> [Node]{
        var nodes:[Node] = []
        var current_id = 0
        for _ in 0 ..< num {
            let r = sqrt(getRandomDouble())
            let degree = getRandomDouble()*360

            let x = (r * cos(degree) + 1.0)/2.0
            let y = (r * sin(degree) + 1.0)/2.0
            let node = Node(x:x,y:y, id: current_id, color: 0 )
            nodes.append( node )
            current_id += 1
        }
        return nodes
    }

    func generateRandomNodesInSphere(num:Int) -> [Node]{
        var nodes:[Node] = []
        var current_id = 0
        for _ in 0 ..< num {
            let u = getRandomDouble()*2.0 - 1.0
            let degree = getRandomDouble()*360
            let x = sqrt(1 - u*u)*cos(degree)
            let y = sqrt(1 - u*u)*sin(degree)
            let z = u
            nodes.append( Node(x:x,y:y,z:z, id: current_id, color: 0 ) )
            current_id += 1
        }
        return nodes
    }


    func generateEdgesBruteForce(nodes:[Node], r:Double) -> [Edge] {
        // WARNING: this is very slow and is O(n^2)
        var edges:[Edge] = []

        for i in 0 ..< nodes.count-1 {
            for j in i+1 ..< nodes.count {
                if sqrt( pow(nodes[i].x-nodes[j].x,2) + pow(nodes[i].y-nodes[j].y,2) + pow(nodes[i].z-
nodes[j].z,2)) <= r {
                    edges.append( Edge(node1: nodes[i],node2: nodes[j]))
                }
            }
        }
        print("Brute Force Edges: ",edges.count)

        return edges
    }

    func generateEdgesUsingCellMethod(nodes:[Node], r:Double) -> [Edge] {
        var edges:[Edge] = []

        let rowCount = Int(ceil(1.0/r))

        var cells:[[[Node]]] = [[[Node]]](repeating: [[Node]](repeating: [], count: rowCount ), count:
rowCount)

        for node in nodes {
            cells[ Int(node.x/r) ][ Int(node.y/r) ].append(node)
        }

        //for each cell
        var i = 0
        while (i < rowCount) {
            var j = 0
            while (j < rowCount) {
                var testNodes:[Node] = []
```

```
                //find adjacent cells
                if i+1 < rowCount && j+1 < rowCount {
                    let firstHalf = cells[i+1][j]
                    let secondHalf = cells[i][j+1] + cells[i+1][j+1]

                    testNodes.append(contentsOf:firstHalf + secondHalf)

                    if j > 0 {
                        testNodes.append(contentsOf:cells[i+1][j-1])
                    }
                } else if i+1 < rowCount {
                    testNodes.append(contentsOf: cells[i+1][j])

                    if j > 0 {
                        testNodes.append(contentsOf:cells[i+1][j-1])
                    }
                } else if j+1 < rowCount {
                    testNodes.append(contentsOf: cells[i][j+1])
                }


                // test nodes in adjacent cells
                for node in cells[i][j] {
                    var l = 0
                    while (l < testNodes.count) {
                        if sqrt( pow(node.x-testNodes[l].x,2) + pow(node.y-testNodes[l].y,2)) <= r {
                            edges.append( Edge(node1: node,node2: testNodes[l]))
                        }
                        l += 1
                    }
                }

                // test nodes within
                var k = 0
                while k < cells[i][j].count-1 {
                    var l = k+1
                    while l < cells[i][j].count {
                        if sqrt( pow(cells[i][j][k].x-cells[i][j][l].x,2) + pow(cells[i][j][k].y-
cells[i][j][l].y,2)) <= r {
                            edges.append( Edge(node1: cells[i][j][k],node2: cells[i][j][l]))
                        }
                        l += 1
                    }
                    k += 1
                }
                j += 1
            }
            i += 1
        }

        print(edges.count)

        return edges
    }

    func generateEdgesUsingCellMethod3d(nodes:[Node], r:Double) -> [Edge] {
        var edges:[Edge] = []

        let rowCount = Int(ceil(2.0/r))

        var cells:[[[[Node]]]] = [[[[Node]]]](repeating: [[[Node]]](repeating: [[Node]](repeating: [],
count: rowCount ), count: rowCount ), count: rowCount)

        for node in nodes {
            let i = Int((node.x+1.0)/r)
            let j = Int((node.y+1.0)/r)
            let k = Int((node.z+1.0)/r)
            cells[ i ][ j ][ k ].append(node)
        }

        let time2 = Date()

        //for each cell
```

```
var i = 0
while (i < rowCount) {
    var j = 0
    while (j < rowCount) {
        var k = 0
        while (k < rowCount) {
            if cells[i][j][k].count > 0 {
                var testNodes:[Node] = []

                //find adjacent cells
                if i+1 < rowCount && j+1 < rowCount && k+1 < rowCount {
                    let one = cells[i][j][k+1]
                    let two = cells[i][j+1][k]
                    let three = cells[i][j+1][k+1]
                    let four = cells[i+1][j][k]
                    let five = cells[i+1][j][k+1]
                    let six = cells[i+1][j+1][k]
                    let seven = cells[i+1][j+1][k+1]

                    testNodes.append(contentsOf: ((one+two)+(three+four))+((five+six)+seven))

                    if j > 0 {
                        let eight = cells[i+1][j-1][k]
                        let nine = cells[i+1][j-1][k+1]
                        testNodes.append(contentsOf: eight+nine)
                    }
                } else if i+1 < rowCount && j+1 < rowCount {
                    let two = cells[i][j+1][k]
                    let four = cells[i+1][j][k]
                    let six = cells[i+1][j+1][k]

                    testNodes.append(contentsOf: (two+four)+six )

                    if j > 0 {
                        let eight = cells[i+1][j-1][k]
                        testNodes.append(contentsOf: eight)
                    }

                } else if i+1 < rowCount && k+1 < rowCount {
                    let one = cells[i][j][k+1]
                    let four = cells[i+1][j][k]
                    let five = cells[i+1][j][k+1]

                    testNodes.append(contentsOf: (one+four)+five )

                    if j > 0 {
                        let eight = cells[i+1][j-1][k]
                        let nine = cells[i+1][j-1][k+1]
                        testNodes.append(contentsOf: eight+nine)
                    }
                } else if i+1 < rowCount {
                    testNodes.append(contentsOf: cells[i+1][j][k])

                    if j > 0 {
                        testNodes.append(contentsOf:cells[i+1][j-1][k])
                    }
                } else if j+1 < rowCount && k+1 < rowCount {
                    let one = cells[i][j][k+1]
                    let two = cells[i][j+1][k]
                    let three = cells[i][j+1][k+1]

                    testNodes.append(contentsOf: (one+two)+three )

                } else if j+1 < rowCount {
                    testNodes.append(contentsOf: cells[i][j+1][k])
                } else if k+1 < rowCount {
                    testNodes.append(contentsOf: cells[i][j][k+1])
                }


                // test nodes in adjacent cells
                for node in cells[i][j][k] {
                    var l = 0
                    while (l < testNodes.count) {
```

```
                                if sqrt( pow(node.x-testNodes[l].x,2) + pow(node.y-testNodes[l].y,2) +
pow(node.z-testNodes[l].z,2)) <= r {
                                    edges.append( Edge(node1: node,node2: testNodes[l]))
                                }
                                l += 1
                            }
                        }

                        // test nodes within
                        var m = 0
                        while m < cells[i][j][k].count-1 {
                            var l = m+1
                            while l < cells[i][j][k].count {
                                if sqrt( pow(cells[i][j][k][m].x-cells[i][j][k][l].x,2) +
pow(cells[i][j][k][m].y-cells[i][j][k][l].y,2) + pow(cells[i][j][k][m].z-cells[i][j][k][l].z,2)) <= r {
                                    edges.append( Edge(node1: cells[i][j][k][m],node2:
cells[i][j][k][l]))
                                }
                                l += 1
                            }
                            m += 1
                        }
                    }
                    k+=1
                }
                j += 1
            }
            i += 1
        }

        let time3 = Date()

        print("Time new: ",time3.timeIntervalSince(time2))

        print(edges.count)

        return edges
    }


    func getAdjacencyList(nodes:[Node],edges:[Edge]) -> [[Node]]{
        var adjList:[[Node]] = [[Node]](repeating: [], count: nodes.count)

        for node in nodes {
            adjList[node.id] = [node]
        }

        for edge in edges {
            adjList[edge.node1.id].append(edge.node2)
            adjList[edge.node2.id].append(edge.node1)
        }

        return adjList
    }


    func updateValueLocation(id:Int, sortedList: inout [(Int,Int)],  referenceList: inout
[([Node],Int)]) {
        let testValue = sortedList[referenceList[id].1].1

        var i = referenceList[id].1

        let value = sortedList.remove(at: i)
        while i < sortedList.count {
            if sortedList[i].1 <= testValue {
                break
            }
            else {
                //update reference list
                referenceList[sortedList[i].0].1 = i
            }
            i += 1
        }
        sortedList.insert(value, at: i)
```

```
        referenceList[sortedList[i].0].1 = i
    }

    func subtractBucket(value:Int,buckets: inout [[Int]], referenceList: inout [([Int],Int)]) {

        //iterate through current value's bucket
        for (index,nodeId) in buckets[ referenceList[value].1 ].enumerated() {
            //find node
            if nodeId == value {
                //move node to lower bucket
                _ = buckets[ referenceList[value].1 ].remove(at: index)
                referenceList[value].1 -= 1
                buckets[ referenceList[value].1 ].append(value)
                break
            }
        }
    }

    func sortBySmallestLastDegreeFast(adjList: [[Node]], shouldPrint:Bool=false) -> [[Node]] {
        colorStats = []
        DEGREE_WHEN_DELETED = []
        ORIGINAL_DEGREE = []

        let startTimeFaster = Date()

        var newList:[[Node]] = []

        var buckets:[[Int]] = [[Int]](repeating: [], count: adjList.count-1)

        var referenceList:[([Int],Int)] = [([Int],Int)](repeating: ([],-1), count: adjList.count)

        var maxDegree = 0
        var totalDegree = 0
        TERMINAL_CLIQUE_SIZE = 1

        //put values in list
        for list in adjList {
            buckets[list.count-1].append(list[0].id)

            var newList = list.map({$0.id})
            newList.remove(at: 0)
            referenceList[list[0].id] = (newList,list.count-1)

            if list.count-1 > maxDegree {
                maxDegree = list.count-1
            }
            totalDegree += list.count-1
        }

        if shouldPrint {
            print(buckets)
        }

        let midTimeFaster = Date()
        var minIndex = 0
        while newList.count < adjList.count {
            while( buckets[minIndex].count == 0  ) {
                minIndex += 1
            }
            let min_bucket_size = buckets[minIndex].count
            let minVal = buckets[minIndex].popLast()!

            let number_left = adjList.count-newList.count
            if min_bucket_size == number_left && TERMINAL_CLIQUE_SIZE == 1 && minIndex+1 ==
min_bucket_size{

                TERMINAL_CLIQUE_SIZE = min_bucket_size
                if shouldPrint {
                    print(min_bucket_size,minIndex,newList.count,adjList.count)
                    print("Terminal Clique found: Size ",TERMINAL_CLIQUE_SIZE)
                }
            }
```

```
            newList.append(adjList[minVal])
            DEGREE_WHEN_DELETED.append(referenceList[minVal].1)

            referenceList[minVal].1 = -1

            let nodeList = referenceList[minVal].0
            ORIGINAL_DEGREE.append(nodeList.count)


            for id in nodeList {
                if referenceList[id].1 >= 0 {
                    subtractBucket(value: id, buckets: &buckets, referenceList: &referenceList)
                }
            }

            if minIndex > 0 {
                minIndex -= 1
            }
            if shouldPrint {
                print("\nRemoving node: ",minVal)
                print(buckets)
            }
        }


        colorStats.append( ("Min Degree",String(DEGREE_WHEN_DELETED[0])) )
        colorStats.append( ("Avg Degree",String( round( Double( totalDegree ) / Double( newList.count )
* 1000 ) / 1000 ) ) )
        colorStats.append( ("Max Degree",String(maxDegree)) )
        colorStats.append( ( "Max Degree when Deleted",String( DEGREE_WHEN_DELETED.max()!  ) ) )


        print("STATS",colorStats)
        newList.reverse()
        ORIGINAL_DEGREE.reverse()
        DEGREE_WHEN_DELETED.reverse()


        print("SETUP TIME: ",midTimeFaster.timeIntervalSince(startTimeFaster))

        return newList
    }

    func colorGraph(adjList: [[Node]],shouldPrint:Bool=false) -> [Int] {

        COLORS = []

        var colorsAssigned:[Int] = [Int](repeating: -1, count: adjList.count)

        for list in adjList {
            if shouldPrint {
                print("\nCOLORING NODE ",list[0].id)
            }
            var colorsTaken:[Int] = []
            for node in list {
                if shouldPrint && colorsAssigned[node.id] != -1 {
                    print("Color",colorsAssigned[node.id],"taken by adjacent node",node.id)
                }
                colorsTaken.append(colorsAssigned[node.id])
            }
            var k = 0
            while k < COLORS.count {
                if colorsTaken.contains(COLORS[k]) {
                    k += 1
                } else {
                    colorsAssigned[list[0].id] = COLORS[k]
                    list[0].color = COLORS[k]
                    if shouldPrint {
                        print("Assigning node",list[0].id,"the color:",list[0].color)
                    }
                    break
                }
            }
            if k == COLORS.count {
```

```
                colorsAssigned[list[0].id] = COLORS.count
                list[0].color = COLORS.count
                if shouldPrint {
                    print("Assigning node",list[0].id,"the color:",list[0].color)
                }
                COLORS.append(COLORS.count)
            }
        }

        var newNodes:[Node] = []
        for k in adjList {
            newNodes.append((k[0]))
        }
        drawView.nodes = newNodes

        return colorsAssigned
    }

    func sortColors() {
        CURRENT_COLOR_FREQUENCIES = [:]
        for num in CURRENT_COLORS_ASSIGNED {
            if CURRENT_COLOR_FREQUENCIES[num] != nil {
                CURRENT_COLOR_FREQUENCIES[num] = CURRENT_COLOR_FREQUENCIES[num]! + 1
            } else {
                CURRENT_COLOR_FREQUENCIES[num] = 1
            }
        }
        CURRENT_COLOR_FREQUENCIES_PAIRED = []
        for (k,v) in CURRENT_COLOR_FREQUENCIES {
            CURRENT_COLOR_FREQUENCIES_PAIRED.append((k,v))
        }
        CURRENT_COLOR_FREQUENCIES_PAIRED.sort { ($0.1) > ($1.1) }
    }

    @IBAction func pressedColorGraphButton(_ sender: Any) {

        resetSphereNetwork()

        let startTimeFaster = Date()
        let sorted = sortBySmallestLastDegreeFast(adjList: self.graphAdjList,shouldPrint:
VERIFICATION_WALKTHROUGH)
        let midTimeFaster = Date()


        let midTime = Date()
        CURRENT_COLORS_ASSIGNED = colorGraph(adjList: sorted,shouldPrint:VERIFICATION_WALKTHROUGH )
        let endTime = Date()



        print("TIME TO SORT FAST: ",midTimeFaster.timeIntervalSince(startTimeFaster))

        TIME_TO_COLOR_GRAPH = endTime.timeIntervalSince(midTime)
        TIME_FOR_SMALLEST_LAST_ORDERING = midTimeFaster.timeIntervalSince(startTimeFaster)

        sortColors()

        colorStats.append( ("Number of Colors",String(COLORS.count)) )
        colorStats.append( ("Terminal Clique Size",String(TERMINAL_CLIQUE_SIZE)) )
        colorStats.append( ("Max Color Class Size",String(CURRENT_COLOR_FREQUENCIES_PAIRED[0].1)) )

        if VERIFICATION_WALKTHROUGH {
            let elements = getNodesAndEdgesFromBipartiteGraph(with: COLORS[0], secondColor:
COLORS[1],shouldPrint: true)
            let parts = getComponents(edges: elements.1,shouldPrint: VERIFICATION_WALKTHROUGH)

            print("\nBackbone:")
            print(parts[0])
        }

        let beforeBipartite = Date()
        let maxEdges = getMaxBipartiteEdges()
        colorStats.append(("Max Edges in a Bipartite Subgraph",String(maxEdges)))
        bipartiteStats = []
```

```
        let maxMajorComponent = getMaxBipartiteGraphElements()

        let afterBipartite = Date()

        bipartiteStats.append(("Max Backbone Vertices",String(maxMajorComponent.1)))
        bipartiteStats.append(("Max Backbone Edges",String(maxMajorComponent.0)))
        bipartiteStats.append(("Max Backbone Domination
Percentage",String(round(maxMajorComponent.2*10000)/100)))

        if networkModel == "Sphere" {
            bipartiteStats.append(("Max Backbone Faces",String(maxMajorComponent.0-
maxMajorComponent.1+2)))
        }

        bipartiteStats.append(("2nd Max Backbone Vertices",String(maxMajorComponent.5)))
        bipartiteStats.append(("2nd Max Backbone Edges",String(maxMajorComponent.4)))
        bipartiteStats.append(("2nd Max Backbone Domination
Percentage",String(round(maxMajorComponent.6*10000)/100)))

        if networkModel == "Sphere" {
            bipartiteStats.append(("2nd Max Backbone Faces",String(maxMajorComponent.4-
maxMajorComponent.5+2)))
        }

        bipartiteStats.append(("Max Backbone Colors",String(maxMajorComponent.3.0)+" and
"+String(maxMajorComponent.3.1)))

        bipartiteStats.append(("2nd Max Backbone Colors",String(maxMajorComponent.7.0)+" and
"+String(maxMajorComponent.7.1)))


        TIME_FOR_BIPARTITE = afterBipartite.timeIntervalSince(beforeBipartite)

        generateColorValues()

        if networkModel == "Sphere" {
            displaySphereNetwork()
        } else {
            drawView.setNeedsDisplay()
        }
    }
    @IBAction func showValueChanged(_ sender: Any) {
        self.shouldShowNodes = self.showNodesSwitch.isOn
        self.shouldShowEdges = self.showEdgesSwitch.isOn
        drawView.shouldShowNodes = self.shouldShowNodes
        drawView.shouldShowEdges = self.shouldShowEdges

        if networkModel == "Sphere" {
            displaySphereNetwork()
        } else {
            drawView.setNeedsDisplay()
        }
    }

    @IBAction func unwindToDisplay(segue: UIStoryboardSegue) {
        print("Back at display")
    }


    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before
navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        // Get the new view controller using segue.destinationViewController.
        // Pass the selected object to the new view controller.
        if segue.identifier == "showStatsSegue" {
            let vc = (segue.destination as! UINavigationController).topViewController as!
StatsTableViewController

            var statistics:[(String,String)] = []

            statistics.append(("Generate Graph (Time)",String(round(1000*TIME_TO_CREATE_GRAPH)/1000)))
            statistics.append(("Smallest Last Ordering
```

```
(Time)",String(round(1000*TIME_FOR_SMALLEST_LAST_ORDERING)/1000)))
            statistics.append(("Color Graph (Time)",String(round(1000*TIME_TO_COLOR_GRAPH)/1000)))
            statistics.append(("Bipartite Stats (Time)",String(round(1000*TIME_FOR_BIPARTITE)/1000)))



            var colorStatistics:[(String,String)] = []

            colorStatistics.append(("Model",networkModel))

            colorStatistics.append(("Node Count (N)",String(CURRENT_NODES.count)))
            colorStatistics.append(("Connection Distance
(R)",String(round(1000*CURRENT_CONNECTION_DISTANCE)/1000)))
            colorStatistics.append(("Edge Count (M)",String(CURRENT_EDGES.count)))

            for val in colorStats {
                colorStatistics.append(val)
            }


            vc.bipartiteStats = self.bipartiteStats
            vc.statistics = statistics
            vc.colorStats = colorStatistics

            vc.title = self.title! + "  Stats"
        }
    }

    @IBOutlet weak var color2Button: UIButton!
    @IBOutlet weak var color1Button: UIButton!
    var color1:Int = -1
    var color2:Int = -1
    @IBOutlet weak var colorPicker: UIPickerView!
    var selecting1 = false

    // returns the number of 'columns' to display.
    func numberOfComponents(in pickerView: UIPickerView) -> Int{
        return 1
    }

    // returns the # of rows in each component..
    func pickerView(_ pickerView: UIPickerView, numberOfRowsInComponent component: Int) -> Int{
        return COLORS.count
    }

    func pickerView(_ pickerView: UIPickerView, titleForRow row: Int, forComponent component: Int) ->
String? {
        return String(COLORS[row])
    }

    func pickerView(_ pickerView: UIPickerView, didSelectRow row: Int, inComponent component: Int)
    {
        if selecting1 {
            color1 = COLORS[row]
            self.color1Button.setTitle("Color "+String(COLORS[row]), for: .normal)
        } else {
            color2 = COLORS[row]
            self.color2Button.setTitle("Color "+String(COLORS[row]), for: .normal)
        }
        self.showBipartiteSwitch.isEnabled = color1 != color2 && color1 != -1 && color2 != -1
        if self.showBipartiteSwitch.isOn && !self.showBipartiteSwitch.isEnabled {
            self.showBipartiteSwitch.isOn = false
        }
        colorPicker.isHidden = true
        displayBipartiteGraph()
    }

    func getNodesAndEdgesFromBipartiteGraph(with firstColor:Int,secondColor:Int,shouldPrint:Bool=false)
-> ([Node],[Edge]) {
        var bipartiteNodes:[Node] = []
        var bipartiteEdges:[Edge] = []

        var nodeIds:[Int] = []
```

```
        if shouldPrint {
            print("Nodes for Bipartite Graph, in order:")
        }

        for edge in CURRENT_EDGES {
            if (edge.node1.color == firstColor && edge.node2.color == secondColor) || (edge.node1.color
== secondColor && edge.node2.color == firstColor) {
                bipartiteEdges.append(edge)
                if !nodeIds.contains(edge.node1.id) {
                    nodeIds.append(edge.node1.id)
                    bipartiteNodes.append(edge.node1)

                    if shouldPrint {
                        print(edge.node1)
                    }
                }
                if !nodeIds.contains(edge.node2.id) {
                    nodeIds.append(edge.node2.id)
                    bipartiteNodes.append(edge.node2)
                    if shouldPrint {
                        print(edge.node2)
                    }
                }
            }
        }

        if shouldPrint {
            print()
        }

        return (bipartiteNodes,bipartiteEdges)
    }



    @IBAction func showBipartiteSwitchChanged(_ sender: Any) {
        displayBipartiteGraph()
    }

    func displayBipartiteGraph() {
        if self.showBipartiteSwitch.isOn && color1 != color2 && color1 != -1 && color2 != -1 {
            resetSphereNetwork()

            let bipartiteGraph = getNodesAndEdgesFromBipartiteGraph(with: color1, secondColor: color2)


            if networkModel == "Sphere" {
                sphereNodes = bipartiteGraph.0
                sphereEdges = bipartiteGraph.1
                displaySphereNetwork()
            } else {
                self.drawView.nodes = bipartiteGraph.0
                self.drawView.edges = bipartiteGraph.1
                self.drawView.shouldShowExtremeEdges = false
                drawView.setNeedsDisplay()

            }
        } else {

            if networkModel == "Sphere" {
                sphereNodes = CURRENT_NODES
                sphereEdges = CURRENT_EDGES
                displaySphereNetwork()
            } else {
                self.drawView.nodes = CURRENT_NODES
                self.drawView.edges = CURRENT_EDGES
                self.drawView.shouldShowExtremeEdges = true
                drawView.setNeedsDisplay()
            }
        }
    }

    func getDominationPercentage(nodeIds:[Int]) -> Double {
        //calculate the domination percentage of the backbones
```

```
        var ids:Set<Int> = Set.init()

        //add all ids to the set
        for id in nodeIds {
            for connected in graphAdjList[id] {
                ids.insert(connected.id)
            }
        }

        return Double(ids.count)/Double(CURRENT_NODES.count)
    }

    func getMaxBipartiteGraphElements() -> (Int,Int,Double,(Int,Int),Int,Int,Double,(Int,Int)) {

        var maxIndex = (-1,-1)
        var maxMajorCompSize = -1
        var numberOfNodes = -1
        var maxDominationPercentage:Double = 0.0

        var secondMaxIndex = (-1,-1)
        var secondMaxSize = -1
        var secondNumberOfNodes = -1
        var secondDominationPercentage:Double = 0.0

        //find the top bipartite graphs
        for i in 0..<min(COLORS.count-1,3) {
            for j in (i+1)..<min(COLORS.count,4) {
                let elements = getNodesAndEdgesFromBipartiteGraph(with:
CURRENT_COLOR_FREQUENCIES_PAIRED[i].0, secondColor: CURRENT_COLOR_FREQUENCIES_PAIRED[j].0)
                let parts = getComponents(edges: elements.1)
                if parts[0].1 > maxMajorCompSize {
                    secondMaxSize = maxMajorCompSize
                    secondMaxIndex = maxIndex
                    secondNumberOfNodes = numberOfNodes
                    secondDominationPercentage = maxDominationPercentage

                    maxMajorCompSize = parts[0].1
                    maxIndex = (i,j)
                    numberOfNodes = parts[0].0.count
                    maxDominationPercentage = getDominationPercentage(nodeIds: parts[0].0)
                } else if parts[0].1 > secondMaxSize {
                    secondMaxSize = parts[0].1
                    secondMaxIndex = (i,j)
                    secondNumberOfNodes = parts[0].0.count
                    secondDominationPercentage = getDominationPercentage(nodeIds: parts[0].0)
                }
            }
        }


        return
(maxMajorCompSize,numberOfNodes,maxDominationPercentage,maxIndex,secondMaxSize,secondNumberOfNodes,seco
ndDominationPercentage,secondMaxIndex)
    }

    func getMaxBipartiteEdges() -> Int {
        var bipartiteEdges:[[Int]] = [[Int]](repeating: [Int](repeating: 0, count: COLORS.count
),count: COLORS.count)


        for edge in CURRENT_EDGES {
            bipartiteEdges[edge.node1.color][edge.node2.color] += 1
            bipartiteEdges[edge.node2.color][edge.node1.color] += 1
        }

        var maxEdgeCount = 0

        for i in 0..<COLORS.count {
            for j in i..<COLORS.count {
                if bipartiteEdges[i][j] > maxEdgeCount {
                    maxEdgeCount = bipartiteEdges[i][j]
                }
            }
        }
```

```
    return maxEdgeCount
}

func getComponents(edges:[Edge],shouldPrint:Bool=false) -> [([Int],Int)] {
    var components:[([Int],Int)] = []

    if shouldPrint {
        print("Making Components")
        print(components)
    }

    //find the components using the edges
    for edge in edges {
        //used for node1 and node2
        var component1 = -1
        var component2 = -1
        for i in 0..<components.count {
            if components[i].0.contains(edge.node1.id) {
                component1 = i
            }
            if components[i].0.contains(edge.node2.id) {
                component2 = i
            }
        }

        //check which of the nodes are in a component already
        if component1 != -1 && component2 != -1 {
            if component1 != component2 {
                let smallIndex = min(component1,component2)
                let bigIndex = max(component1,component2)

                let removed = components.remove(at: bigIndex)

                //combine the two components
                components[smallIndex].0.append(contentsOf: removed.0)
                components[smallIndex].1 += removed.1 + 1
            } else {
                components[component1].1 += 1
            }
        } else if component1 != -1 {
            components[component1].0.append(edge.node2.id)
            components[component1].1 += 1
        } else if component2 != -1 {
            components[component2].0.append(edge.node1.id)
            components[component2].1 += 1
        } else {
            //make new component
            components.append( ([edge.node1.id,edge.node2.id],1) )
        }

        if shouldPrint {
            print(components)
        }
    }

    //sort by largest component first
    return components.sorted { ($0.1) > ($1.1) }
}

func findMinAndMaxDegreeNodes() -> (Int,Int) {
    var ids = (0,0)
    var min = 100000
    var max = 0

    for list in graphAdjList {
        if list.count > max {
            max = list.count
            ids.1 = list[0].id
        }
        else if list.count < min {
            min = list.count
            ids.0 = list[0].id
        }
```

```
        }

        return ids
    }



    @IBAction func color1ButtonPressed(_ sender: Any) {
        selecting1 = true
        self.colorPicker.dataSource = self
        self.colorPicker.delegate = self
        colorPicker.isHidden = COLORS.count == 0
    }

    @IBAction func color2ButtonPressed(_ sender: Any) {
        selecting1 = false
        self.colorPicker.dataSource = self
        self.colorPicker.delegate = self
        colorPicker.isHidden = COLORS.count == 0
    }



}
```

APPENDIX 3: DISPLAYVIEW.SWIFT

```
//
//  DisplayView.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 2/28/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import UIKit

var COLOR_VALUES:[UIColor] = [UIColor(colorLiteralRed: getRandomFloat(), green: getRandomFloat(), blue:
getRandomFloat(), alpha: 1.0)]

func generateColorValues() {
    COLOR_VALUES = []
    for _ in 0 ..< COLORS.count {
        COLOR_VALUES.append(UIColor(colorLiteralRed: getRandomFloat(), green: getRandomFloat(), blue:
getRandomFloat(), alpha: 1.0))
    }
}

class DisplayView: UIView {

    var NODE_SIZE:Double = 5.0
    var EDGE_WIDTH:Double = 0.3

    var nodes:[Node] = []
    var edges:[Edge] = []
    var extremeDegreeEdges:[Edge] = []

    var shouldShowNodes = true
    var shouldShowEdges = true
    var shouldShowExtremeEdges = true

    var xMin:Double = 50
    var yMin:Double = 100
    var xMax:Double = 250
    var yMax:Double = 300

    var model = "Square"

    override func draw(_ rect: CGRect)
    {
        xMax = Double(rect.size.width)
        yMax = Double(rect.size.height)

        //draw nodes
        if shouldShowNodes {
            if model == "Square" || model == "Disk"{
                for node in nodes {
                    drawNodeInSquare(obj: node)
                }
            }
        }

        //draw edges
        if shouldShowEdges {
            if model == "Square" || model == "Disk" {
                for edge in edges {
                    drawEdgeInSquare(obj: edge)
                }

                if shouldShowExtremeEdges {
                    for edge in extremeDegreeEdges {
                        drawEdgeInSquare(obj: edge)
                    }
                }
            }
        }
    }
```

```
    func drawNodeInSquare(obj:Node) {
        let ctx = UIGraphicsGetCurrentContext()
        ctx?.setFillColor(COLOR_VALUES[obj.color].cgColor)
        ctx?.setStrokeColor(COLOR_VALUES[obj.color].cgColor)
        ctx?.setLineWidth(CGFloat(NODE_SIZE))

        let rectangle = CGRect(x: xMax*obj.x-NODE_SIZE/2, y: yMax*obj.y-NODE_SIZE/2, width: NODE_SIZE,
height: NODE_SIZE)
        ctx?.addEllipse(in: rectangle)
        ctx?.drawPath(using: .fillStroke)
    }

    func drawEdgeInSquare(obj:Edge) {
        let context = UIGraphicsGetCurrentContext()
        context?.setLineWidth(CGFloat(EDGE_WIDTH*2))
        context?.setLineWidth(CGFloat(EDGE_WIDTH))
        context?.setStrokeColor(obj.color.cgColor)
        context?.move(to: CGPoint(x: xMax*obj.node1.x, y: yMax*obj.node1.y))
        context?.addLine(to: CGPoint(x: xMax*obj.node2.x, y: yMax*obj.node2.y))
        context?.strokePath()
    }
}
```

APPENDIX 4: SPHERESCENE.SWIFT

```
//
//  SphereScene.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 4/13/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import UIKit
import SceneKit

extension SCNGeometry {
    class func lineFrom(vector vector1: SCNVector3, toVector vector2: SCNVector3) -> SCNGeometry {
        let indices: [Int32] = [0, 1]

        let source = SCNGeometrySource(vertices: [vector1, vector2], count: 2)
        let element = SCNGeometryElement(indices: indices, primitiveType: .line)

        return SCNGeometry(sources: [source], elements: [element])
    }
}

class SphereScene: SCNScene {
    var nodes:[Node] = []
    var edges:[Edge] = []
    var extremeDegreeEdges:[Edge] = []
    var vectors:[Int:SCNVector3] = [:]

    override init() {
        super.init()
        basicSphere()
    }

    init(nodes:[Node]) {
        super.init()
        self.nodes = nodes
        displayNodesOnSphere(nodes: self.nodes)
    }


init(nodes:[Node],edges:[Edge],extremeEdges:[Edge],shouldShowNodes:Bool=true,shouldShowEdges:Bool=false
,shouldShowExtremeEdges:Bool=false) {
        super.init()

        self.nodes = nodes
        self.edges = edges
        self.extremeDegreeEdges = extremeEdges
        if shouldShowNodes {
            displayNodesOnSphere(nodes: self.nodes)
        }
        if shouldShowEdges {
            displayEdgesOnSphere(edges: self.edges)
            if shouldShowExtremeEdges {
                displayEdgesOnSphereWithColor(edges: extremeDegreeEdges)
            }
        }
    }

    required init(coder aDecoder: NSCoder) {
        fatalError("init(coder:) has not been implemented")
    }

    func displayEdgesOnSphere(edges:[Edge]) {
        for e in edges {
            if vectors[e.node1.id] == nil{
                vectors[e.node1.id] = SCNVector3(x: Float(e.node1.x), y: Float(e.node1.y), z:
Float(e.node1.z))
            }
            if vectors[e.node2.id] == nil {
                vectors[e.node2.id] = SCNVector3(x: Float(e.node2.x), y: Float(e.node2.y), z:
Float(e.node2.z))
```

```
            }
            let geometry = SCNGeometry.lineFrom(vector: vectors[e.node1.id]!, toVector:
vectors[e.node2.id]!)
            let node = SCNNode(geometry: geometry)
            self.rootNode.addChildNode(node)
        }
    }

    func displayEdgesOnSphereWithColor(edges:[Edge]) {
        for e in edges {
            if vectors[e.node1.id] == nil{
                vectors[e.node1.id] = SCNVector3(x: Float(e.node1.x), y: Float(e.node1.y), z:
Float(e.node1.z))
            }
            if vectors[e.node2.id] == nil {
                vectors[e.node2.id] = SCNVector3(x: Float(e.node2.x), y: Float(e.node2.y), z:
Float(e.node2.z))
            }
            let geometry = SCNGeometry.lineFrom(vector: vectors[e.node1.id]!, toVector:
vectors[e.node2.id]!)
            geometry.firstMaterial?.diffuse.contents = UIColor.red
            let node = SCNNode(geometry: geometry)
            self.rootNode.addChildNode(node)
        }
    }

    func displayNodesOnSphere(nodes:[Node]) {
        let radius:CGFloat = 0.01
        for n in nodes {
            let sphereGeometry = SCNSphere(radius: radius)
            sphereGeometry.segmentCount = 4
            sphereGeometry.firstMaterial?.diffuse.contents = COLOR_VALUES[n.color]
            let sphereNode = SCNNode(geometry: sphereGeometry)

            vectors[n.id] = SCNVector3(x: Float(n.x), y: Float(n.y), z: Float(n.z))

            sphereNode.position = vectors[n.id]!

            self.rootNode.addChildNode(sphereNode.flattenedClone())
        }
    }

    func basicSphere() {
        let sphereGeometry = SCNSphere(radius: 1.0)
        let sphereNode = SCNNode(geometry: sphereGeometry)
        self.rootNode.addChildNode(sphereNode)
    }


}
```

Appendix 5: Node.swift

```
//
//  Node.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 2/28/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import UIKit

class Node: NSObject {
    var x: Double = 0.0
    var y: Double = 0.0
    var z: Double = 0.0
    var color:Int = 0
    var id: Int = -1

    init(x:Double,y:Double,z:Double=0.0,id:Int,color:Int) {
        self.x = x
        self.y = y
        self.z = z
        self.id = id
        self.color = color
    }

    override var description: String {
        return "\(id)|(\(round(1000*x)/1000),\(round(1000*y)/1000))"
    }
}
```

Appendix 6: Edge.swift

```
//
//  Edge.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 2/28/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import UIKit

class Edge: NSObject {
    var node1:Node
    var node2:Node
    var color: UIColor = UIColor.red

    init(node1:Node,node2:Node,color:UIColor = UIColor.red) {
        self.node1 = node1
        self.node2 = node2
        self.color = color
    }
}
```

APPENDIX 7: STATSTABLEVIEWCONTROLLER.SWIFT

```
//
//  StatsTableViewController.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 4/2/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import UIKit

class StatsTableViewController: UITableViewController {

    var statistics:[(String,String)] = []
    var bipartiteStats:[(String,String)] = []
    var colorStats:[(String,String)] = []

    var graphs:[String] = ["Degree Distribution","Color Frequency","Degree Deletion Analysis"]

    var graphData:[Double] = []
    var graphData2:[Double] = []
    var graphTitle:String = ""
    var graphDescription:String = ""
    var graphAxisLabel:String = ""
    var graphAxisLabel2:String = ""

    override func viewDidLoad() {
        super.viewDidLoad()

        print("\nColoring Stats")
        for stat in colorStats {
            print(stat.1,terminator:",")
        }

        print("\nTiming Data")
        for stat in statistics {
            print(stat.1,terminator:",")
        }
        print("\nBipartite stats: ")

        var counter = 1
        for stat in bipartiteStats {
            if counter % 4 == 0 {
                // place an empty spot for copy pasting the faces data
                if stat.0 != "Max Backbone Faces" && stat.0 != "2nd Max Backbone Faces" {
                    print("--",terminator:",")
                }
            }
            print(stat.1,terminator:",")
            counter += 1
        }
        print("\nDone")
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        // Dispose of any resources that can be recreated.
    }

    // MARK: - Table view data source

    override func numberOfSections(in tableView: UITableView) -> Int {
        // #warning Incomplete implementation, return the number of sections
        return 4
    }

    override func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
        // #warning Incomplete implementation, return the number of rows
        switch section {
        case 0:
            return colorStats.count
        case 1:
```

```
            return statistics.count
        case 2:
            return graphs.count
        case 3:
            return bipartiteStats.count
        default:
            return statistics.count
        }
    }

    override func tableView(_ tableView: UITableView, titleForHeaderInSection section: Int) -> String?
{
        switch section {
        case 0:
            return "Coloring Stats"
        case 1:
            return "Timing Stats"
        case 2:
            return "Charts"
        case 3:
            return "Bipartite Stats"
        default:
            return "Other Stats"
        }
    }

    override func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
        switch indexPath.section {
        case 0:
            let cell = tableView.dequeueReusableCell(withIdentifier: "statsCell", for: indexPath) as!
StatsTableViewCell
            cell.typeLabel.text = colorStats[indexPath.row].0
            cell.valueLabel.text = colorStats[indexPath.row].1
            return cell
        case 1:
            let cell = tableView.dequeueReusableCell(withIdentifier: "statsCell", for: indexPath) as!
StatsTableViewCell
            cell.typeLabel.text = statistics[indexPath.row].0
            cell.valueLabel.text = statistics[indexPath.row].1
            return cell
        case 2:
            let cell = tableView.dequeueReusableCell(withIdentifier: "generateGraphCell", for:
indexPath) as! StatsTableViewCell
            cell.typeLabel.text = graphs[indexPath.row]
            return cell
        case 3:
            let cell = tableView.dequeueReusableCell(withIdentifier: "statsCell", for: indexPath) as!
StatsTableViewCell
            cell.typeLabel.text = bipartiteStats[indexPath.row].0
            cell.valueLabel.text = bipartiteStats[indexPath.row].1
            return cell
        default:
            let cell = tableView.dequeueReusableCell(withIdentifier: "statsCell", for: indexPath) as!
StatsTableViewCell
            cell.typeLabel.text = statistics[indexPath.row].0
            cell.valueLabel.text = statistics[indexPath.row].1
            return cell
        }
    }

    @IBAction func unwindToStats(segue: UIStoryboardSegue) {
        print("Back at stats")
    }


    // MARK: - Navigation

    // In a storyboard-based application, you will often want to do a little preparation before
navigation
    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "graphSegue" {
            let vc = segue.destination as! GraphsViewController
```

```
            vc.input_data = self.graphData
            vc.input_data2 = self.graphData2
            if self.graphData2.count > 0 {
                vc.chartType = "Line"
            } else {
                vc.chartType = "Bar"
            }
            vc.chartTitle = self.graphTitle
            vc.chartDataLabel = self.graphAxisLabel
            vc.chartDataLabel2 = self.graphAxisLabel2
            vc.chartDescription = self.graphDescription

            //remove "Stats" from title
            if let titleString = self.title {
                let index = titleString.index(titleString.endIndex, offsetBy:-6)
                vc.title = titleString.substring(to: index)
            }
        }
    }
    @IBAction func goButtonPressed(_ sender: Any) {
        var indexPath: NSIndexPath!

        if let button = sender as? UIButton {
            if let superview = button.superview {
                if let cell = superview.superview as? StatsTableViewCell {
                    indexPath = self.tableView.indexPath(for: cell)! as NSIndexPath
                    print(indexPath)

                    self.graphTitle = graphs[indexPath.row]
                    self.graphDescription = ""
                    self.graphAxisLabel = ""
                    self.graphData = []
                    switch self.graphTitle {
                    case "Degree Distribution":
                        self.graphAxisLabel = "Vertex Count"
                        self.graphDescription = "Degree Distribution for Verticies"

                        self.graphData = [Double](repeating: 0.0, count: CURRENT_ADJACENCY_LIST.count)
                        for list in CURRENT_ADJACENCY_LIST {
                            self.graphData[list.count-1] += 1.0
                        }
                        var i = self.graphData.count-1

                        while i > -1 {
                            if self.graphData[i] == 0.0 {
                                _ = self.graphData.popLast()
                            }
                            else {
                                break
                            }
                            i -= 1
                        }
                        self.graphData2 = []
                    case "Color Frequency":
                        self.graphAxisLabel = "Vertex Count"
                        self.graphDescription = "Frequency of Verticies By Color"
                        self.graphData = [Double](repeating: 0.0, count: CURRENT_COLORS_ASSIGNED.count)
                        for num in CURRENT_COLORS_ASSIGNED {
                            self.graphData[num] += 1.0
                        }
                        var i = self.graphData.count-1

                        while i > -1 {
                            if self.graphData[i] == 0.0 {
                                _ = self.graphData.popLast()
                            }
                            else {
                                break
                            }
                            i -= 1
                        }
                        self.graphData2 = []
                    case "Degree Deletion Analysis":
                        self.graphAxisLabel = "Degree When Deleted"
```

```
                        self.graphAxisLabel2 = "Original Degree"
                        self.graphDescription = ""

                        self.graphData = [Double](repeating: 0.0, count: DEGREE_WHEN_DELETED.count)
                        var i = 0
                        for val in DEGREE_WHEN_DELETED {
                            self.graphData[i] = Double(val)
                            i += 1
                        }

                        i = 0
                        self.graphData2 = [Double](repeating: 0.0, count: ORIGINAL_DEGREE.count)
                        for val in ORIGINAL_DEGREE {
                            self.graphData2[i] = Double(val)
                            i += 1
                        }


                    default:
                        print("Graph data not created")
                    }

                    print("Generating graph")
                    performSegue(withIdentifier: "graphSegue", sender: nil)
                }
            }
        }
    }


}
```

APPENDIX 8: GRAPHSVIEWCONTROLLER.SWIFT

```
//
//  GraphsViewController.swift
//  NetworkSimulation
//
//  Created by Travis Siems on 4/2/17.
//  Copyright © 2017 Travis Siems. All rights reserved.
//

import Charts
import UIKit

class GraphsViewController: UIViewController {

    @IBOutlet weak var titleLabel: UILabel!
    @IBOutlet var barChartView: BarChartView!

    @IBOutlet weak var lineChartView: LineChartView!
    @IBOutlet weak var chartTitleLabel: UILabel!

    var chartTitle = ""
    var input_data:[Double] = []
    var input_data2:[Double] = []
    var chartDataLabel = ""
    var chartDataLabel2 = ""
    var chartDescription = ""
    var chartType = "Bar"

    override func viewDidLoad() {
        super.viewDidLoad()
        self.titleLabel.text = self.title

        if chartType == "Bar" {
            barChartView.isHidden = false
            lineChartView.isHidden = true

            barChartView?.noDataText = "UH OH! No Data!"
            barChartView.setNeedsDisplay()


            setChart(values: input_data, label:chartDataLabel, description:chartDescription)
            chartTitleLabel.text = chartTitle
        } else if chartType == "Line" {
            barChartView.isHidden = true
            lineChartView.isHidden = false

            lineChartView?.noDataText = "UH OH! No Data!"
            lineChartView.setNeedsDisplay()


            setLineChart(values1: input_data, values2: input_data2, label1:chartDataLabel,
label2:chartDataLabel2, description:chartDescription)
            chartTitleLabel.text = chartTitle

        }

    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
    }


    func setChart(values: [Double],label:String,description:String) {
        var dataEntries: [BarChartDataEntry] = []

        for i in 0..<values.count {
            let dataEntry = BarChartDataEntry(x: Double(i), y: Double(values[i]),data:values[i] as
AnyObject)
            dataEntries.append(dataEntry)
        }
```

```
        let chartDataSet = BarChartDataSet(values: dataEntries, label: label)
        barChartView.data = BarChartData(dataSet: chartDataSet as IChartDataSet)

        barChartView.chartDescription?.text = description
    }

    func setLineChart(values1: [Double], values2:
[Double],label1:String,label2:String,description:String) {

        var dataEntries1: [ChartDataEntry] = []
        var dataEntries2: [ChartDataEntry] = []

        for i in 0..<values1.count {
            let dataEntry = ChartDataEntry(x: Double(i), y: values1[i])
            dataEntries1.append(dataEntry)
        }

        for i in 0..<values2.count {
            let dataEntry = ChartDataEntry(x: Double(i), y: values2[i])
            dataEntries2.append(dataEntry)
        }


        let lineChartDataSet1 = LineChartDataSet(values: dataEntries1, label: label1)
        lineChartDataSet1.drawCirclesEnabled = false
        lineChartDataSet1.setColor(NSUIColor.red, alpha: 1.0)
        let lineChartDataSet2 = LineChartDataSet(values: dataEntries2, label: label2)
        lineChartDataSet2.drawCirclesEnabled = false
        lineChartDataSet2.setColor(NSUIColor.blue, alpha: 1.0)

        let lineChartData = LineChartData(dataSets: [lineChartDataSet1,lineChartDataSet2])
        lineChartView.data = lineChartData

        lineChartView.chartDescription?.text = description
    }


}
```