Wireless Sensor Network Project

Travis Siems

# 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 or tables and described in this document.

I have created a wireless network simulator in the form of an iOS application with real-time display and graph statistics. The graphical display helps users understand what is happening with the random geometric graph creation and coloring algorithms.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 0 | 18 | 77 | 183 |
| 2 | 4000 | 0.071 | 119501 | 11 | 59.751 | 87 | 37 | 0 | 31 | 167 | 418 |
| 3 | 16000 | 0.036 | 505902 | 18 | 63.238 | 94 | 39 | 0 | 31 | 630 | 1628 |
| 4 |  |  |  |  |  |  |  |  |  |  |  |
| 5 |  |  |  |  |  |  |  |  |  |  |  |
| 6 | 4000 | 0.063 | 120574 | 20 | 60.287 | 90 | 38 | 0 | 30 | 164 | 416 |
| 7 | 4000 | 0.09 | 235794 | 47 | 117.897 | 164 | 68 | 0 | 52 | 89 | 244 |
| 8 | 4000 | 0.252 | 116325 | 26 | 58.163 | 86 | 36 | 0 | 30 | 169 | 443 |
| 9 |  |  |  |  |  |  |  |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  |  |

## 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++). 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.

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. For the three-dimensional display, I used Apple’s SceneKit which allows a programmer to add objects in a 3D world one-by-one. For displaying the charts, I used a library called Charts, created by Daniel Cohen Gindi. 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. 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 high vertex and edge counts exceeds the amount of RAM allowed for an app on an iPhone. This limitation can be overcome with future iterations to the display method. 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 3 different sections: Basic Stats, Charts, and Bipartite Stats. By keeping all the results in this one view, the user can see as much information as he needs to about the network.

## References

https://www.extremetech.com/computing/183563-apples-new-swift-language-explained-a-clever-move-to-boost-ios-while-holding-android-apps-back

https://developer.apple.com/reference/coregraphics/cgcontext-r9r

https://github.com/danielgindi/Charts

https://support.apple.com/kb/sp705?locale=en\_US

https://developer.apple.com/library/content/documentation/IDEs/Conceptual/iOS\_Simulator\_Guide/Introduction/Introduction.html

http://www.gsmarena.com/apple\_iphone\_7-8064.php

http://editor.method.ac/

http://mathworld.wolfram.com/SpherePointPicking.html

http://stackoverflow.com/questions/14718407/scenekit-on-os-x-with-thousands-of-objects

https://www.weheartswift.com/introduction-scenekit-part-1/

# Wireless Sensor Network Backbone Report

## Reduction to Practice

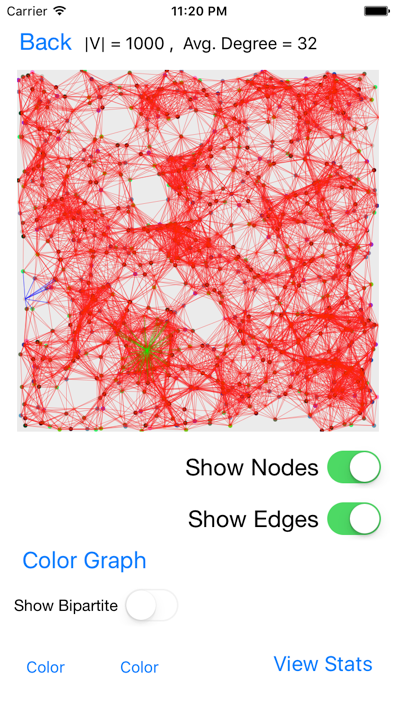
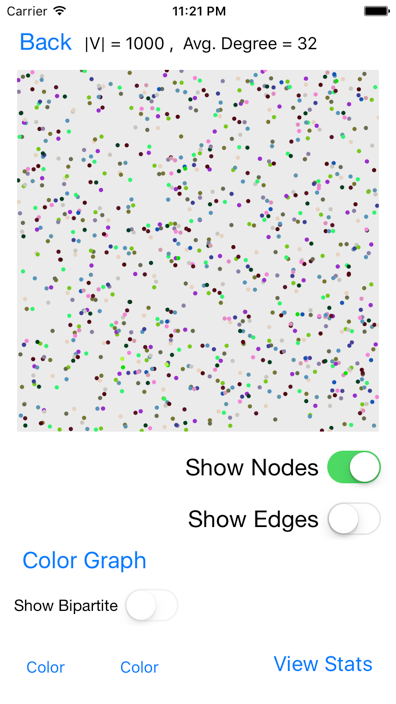
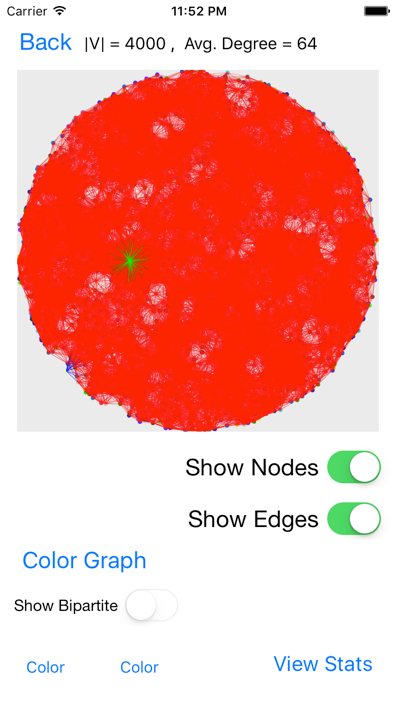
### Part 1

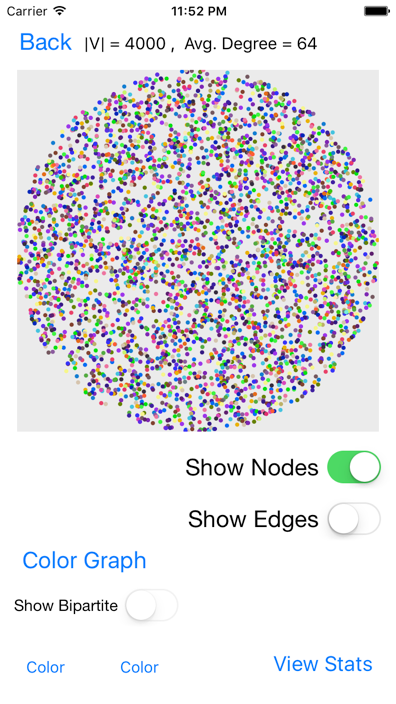
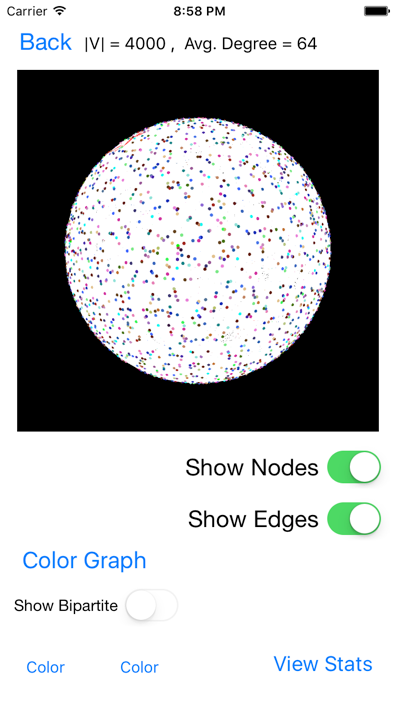
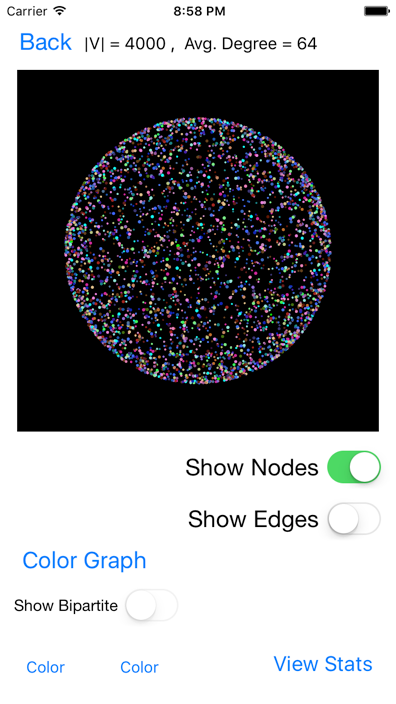
The first part of the project (creating and displaying the random geometric graph) is implemented by first generating vertices in the given network model and then determining where edges should be created. After the vertices and edges are in their respective lists, I generate an adjacency list of the 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 and a random value [0,1] for the r2 value. I then converted these polar coordinates to rectangular coordinates using and . 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 . I then converted these values to Cartesian coordinates using the following equations: , , and .

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.

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 R2 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 nodes. The figure to the right shows which cells are compared around the current cells (as long as they are in the grid).

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 nodes in it where nodes may be in for possible connections.

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 was customizable and easy to work with, but it required far too much RAM to display and interact with when there were many edges. Additionally, the amount of RAM allowed for an app on an actual iPhone is reached after about 1000 vertices. 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.

### 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 assign a valid color to each vertex based on adjacent vertices starting from the lowest color value. 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 O(|V|2+|E|).

The smallest last ordering algorithm is computed with the following process:

The above chart shows how my algorithm runs compared to linear time complexity as defined by O(|V| + |E|). The R2 value is 0.88945 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. Since the R2 value 0.99933 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.

The smallest last ordering should run with a complexity of O(|V|+|E|), but my tests prove that it runs 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.

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 R2 = 0.9811. Since this value is very close to 1.0, I can conclude that this is probably a linear relationship.

### Part 3

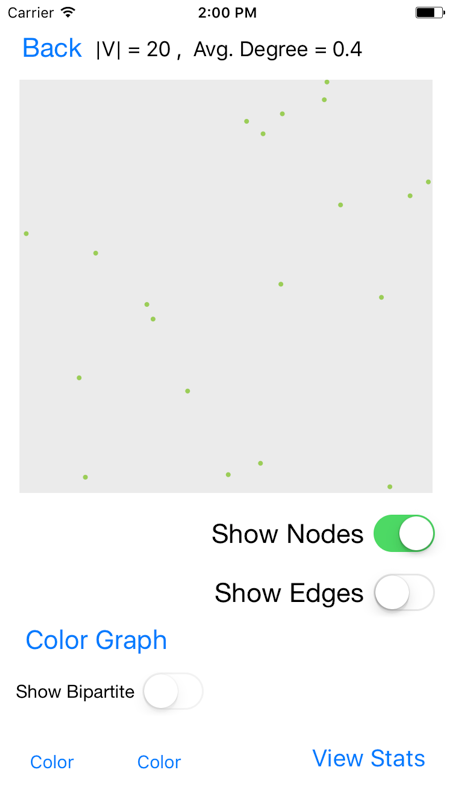
A bipartite graph can be created by selecting two colors and display all the vertices and edges between them. I calculated this by checking which edges have both a vertex with the first color and a vertex with the second color. I then displayed these vertices and edges to see the bipartite subgraph. 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 the formula , or rather .

### 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)**

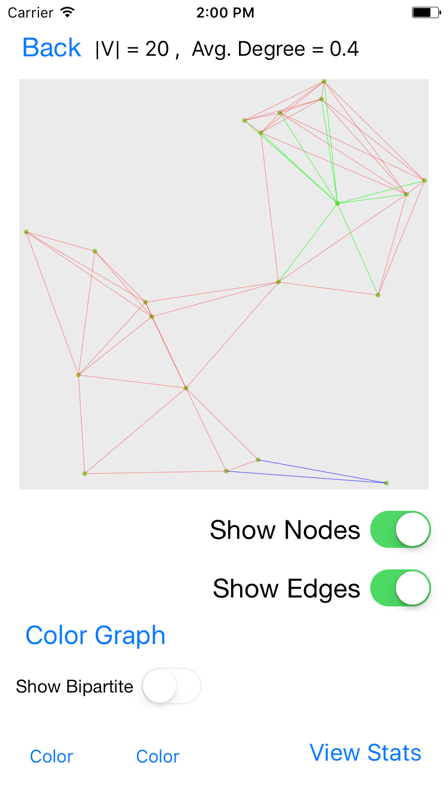
**15|(0.733,0.158)**

**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: . 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.

**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**

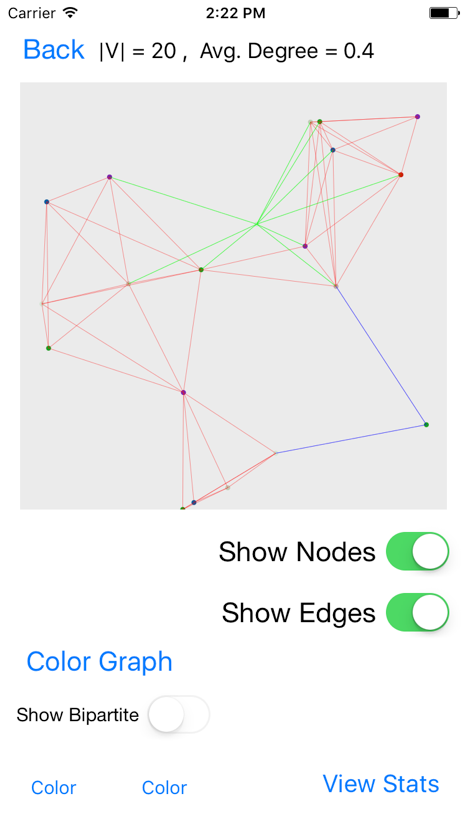
**Color 2 taken by adjacent node 18**

**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**

The display to the right shows the graph after being colored.

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)**

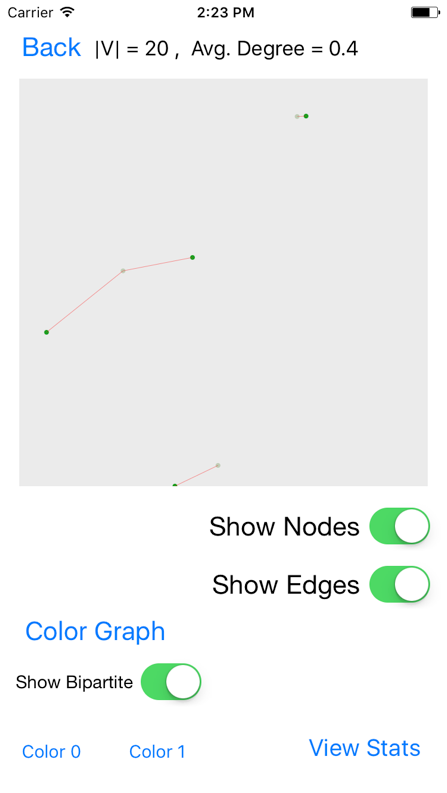
**6|(0.067,0.622)**

**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 increment the edge count.

**Making Components**

**[]**

**[([9, 11], 1)]**

**[([9, 11, 6], 2)]**

**[([9, 11, 6], 2), ([16, 3], 1)]**

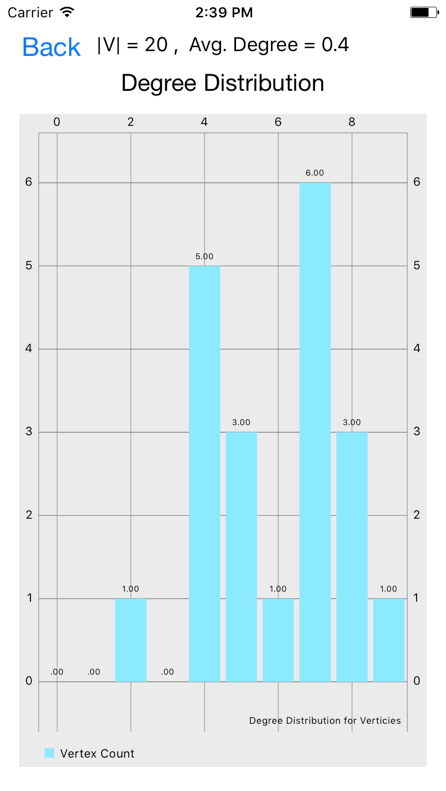
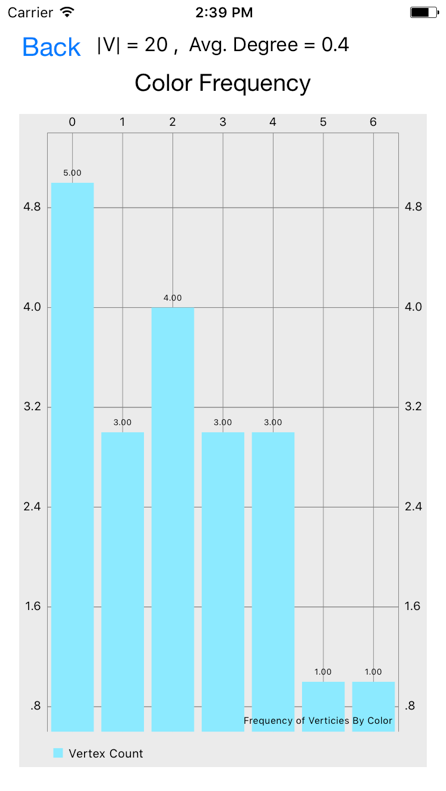
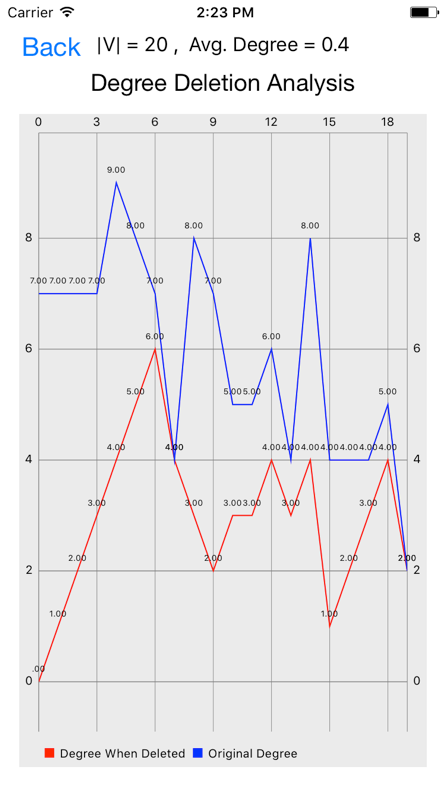
**[([9, 11, 6], 2), ([16, 3], 1), ([2, 5], 1)]**

**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.

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 show and 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, the following tables show the summary values calculated from running the application. The first table shows what the input values are for each benchmark. The benchmark numbers used in the following tables reflect the values in this first table.

Table : 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 |

The next table shows the Graph Coloring Summary Table. This table gives insight as to how

This table shows the output values from running the benchmarks. Each column adds additional information about the network. The “Number of Edges (|E|)” is the number of edges generated, the “Connection Distance (R)” is the calculated max value for two adjacent nodes to connect, the “Min Degree” is the smallest degree for a vertex found in the network, the “Avg. Degree (realized)” is the calculated average degree for the network after generation, and the “Max Degree” is the largest degree for a vertex found in the network a node. The “Max Degree when Deleted” is the largest degree that was deleted when computing the smallest-last ordering. The “Max Edges in a Bipartite Subgraph” is the largest number of edges found out of the bipartite graphs made from the combinations of the top four colors. 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.

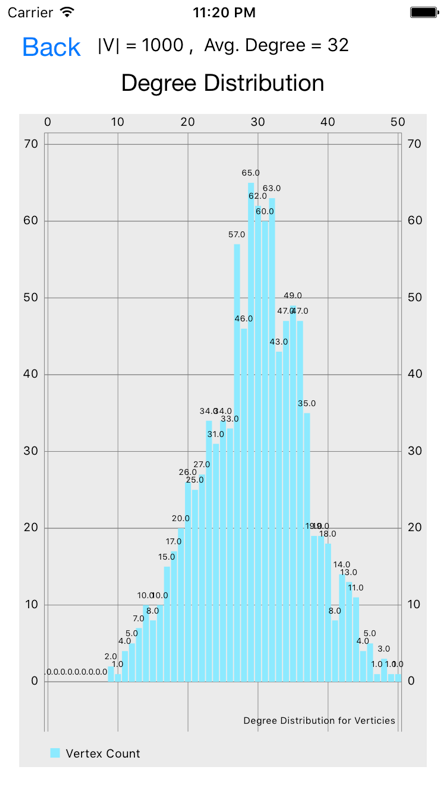
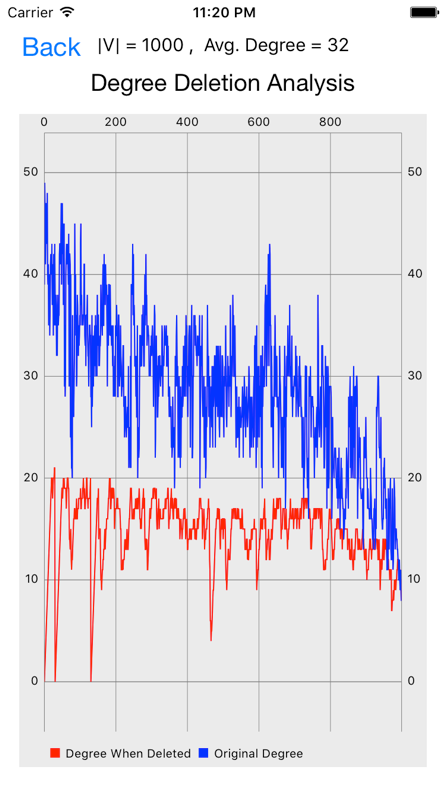
It can be noted that the realized average degree for the square and disk is usually smaller than the benchmark value. This is caused by the bounds of the display area. An average vertex on the edge of the graph has about half of the connection possibilities as an average vertex at least R away from the bounds.

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Benchmark # | Number of Edges (|E|) | Connection Distance (R) | Min Degree | Avg. Degree (realized) | Max Degree | Max Degree when Deleted | Max Edges in a Bipartite Subgraph | Graph Creation Time | Smallest Last Ordering Time | Coloring Time | Bipartite Stats Time |
| 1 | 14733 | 0.101 | 9 | 29.466 | 50 | 21 | 169 | 0.052 | 0.136 | 0.049 | 0.155 |
| 2 | 117482 | 0.071 | 16 | 58.741 | 87 | 36 | 418 | 0.331 | 1.677 | 0.526 | 1.012 |
| 3 | 507230 | 0.036 | 18 | 63.404 | 98 | 41 | 1645 | 1.421 | 20.097 | 2.268 | 8.673 |
| 4 | 2050090 | 0.018 | 16 | 64.065 | 104 | 43 | 6457 | 5.793 | 329.292 | 9.265 | 102.963 |
| 5 | 3935271 | 0.025 | 37 | 122.977 | 168 | 71 | 3833 | 10.901 | 341.375 | 25.458 | 51.681 |
| 6 | 121857 | 0.063 | 16 | 60.929 | 97 | 39 | 411 | 0.335 | 1.733 | 0.571 | 1.072 |
| 7 | 239432 | 0.09 | 42 | 119.716 | 166 | 70 | 250 | 0.633 | 2.151 | 1.745 | 1.338 |
| 8 | 116478 | 0.252 | 27 | 58.239 | 86 | 38 | 442 | 0.462 | 1.838 | 0.504 | 1.038 |
| 9 | 922341 | 0.178 | 56 | 115.293 | 164 | 71 | 1010 | 3.578 | 23.429 | 5.919 | 6.909 |
| 10 | 3773370 | 0.09 | 50 | 117.918 | 175 | 72 | 3876 | 14.654 | 324.988 | 25.274 | 52.089 |

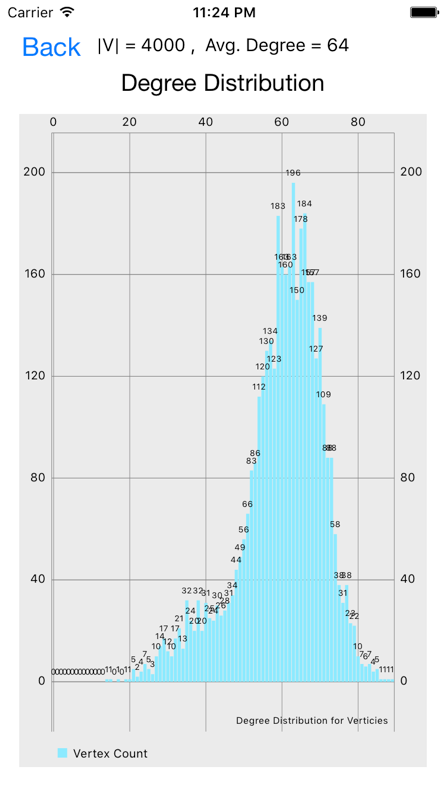
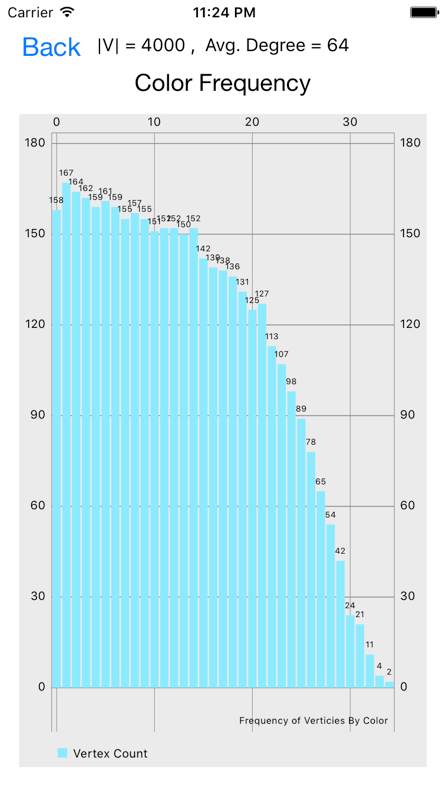
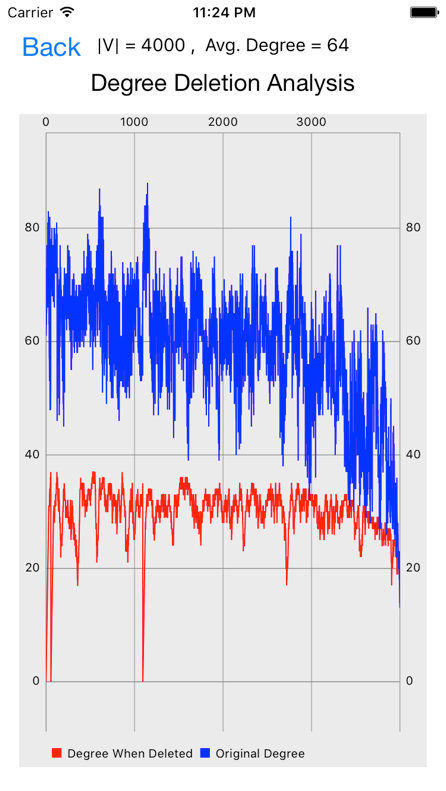
Table : Backbone statistics

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 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 | 142 | 199 | 99.3 | – | 141 | 194 | 99.1 | – |
| 2 | 327 | 495 | 99.83 | – | 325 | 494 | 99.7 | – |
| 3 | 1248 | 1923 | 99.91 | – | 1249 | 1889 | 99.86 | – |
| 4 | 4947 | 7533 | 99.91 | – | 4931 | 7533 | 99.89 | – |
| 5 | 2779 | 4394 | 99.95 | – | 2785 | 4387 | 99.98 | – |
| 6 | 316 | 471 | 99.73 | – | 310 | 464 | 99.33 | – |
| 7 | 180 | 289 | 100 | – | 180 | 288 | 100 | – |
| 8 | 335 | 509 | 100 | 176 | 336 | 498 | 99.55 | 164 |
| 9 | 733 | 1142 | 99.78 | 411 | 733 | 1129 | 99.78 | 398 |
| 10 | 2840 | 4404 | 99.9 | 1566 | 2836 | 4377 | 99.78 | 1543 |

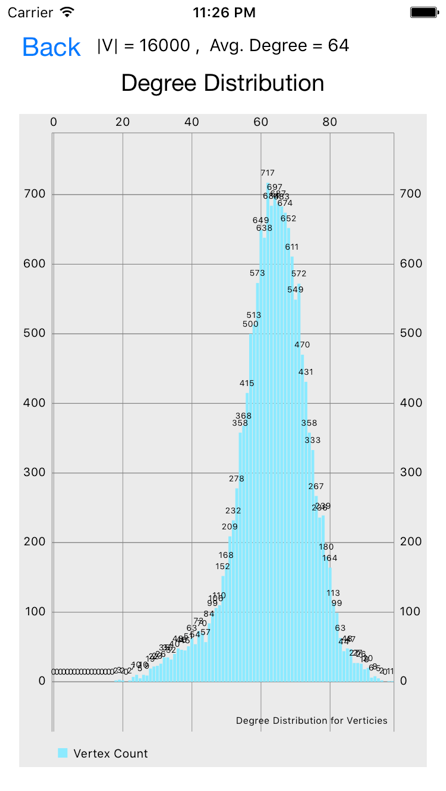
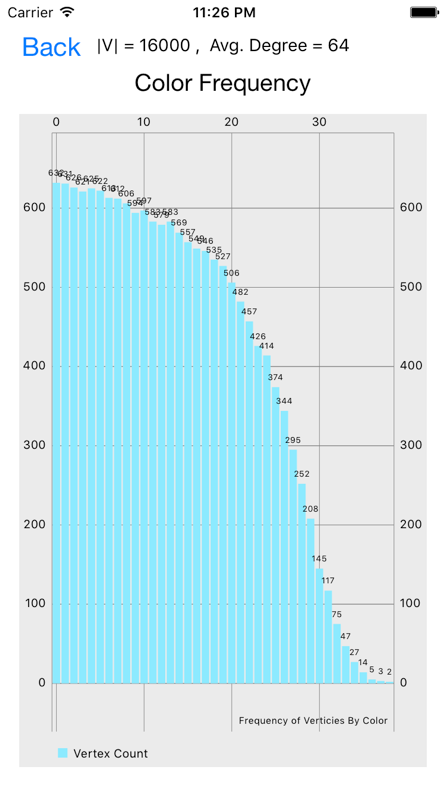
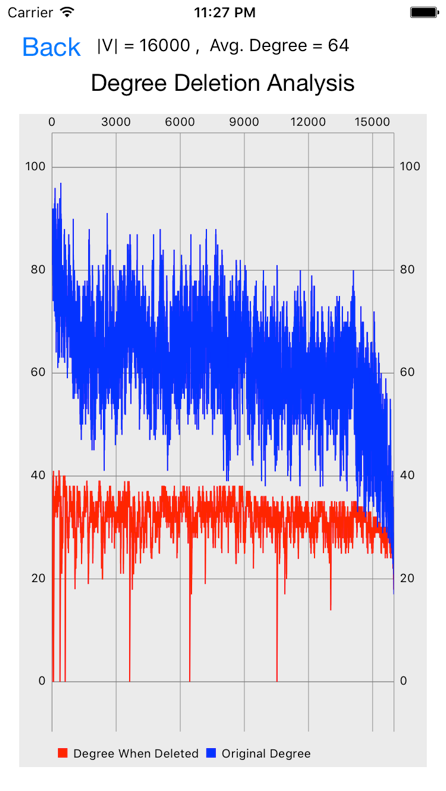
This table shows the bipartite subgraph output values from running the benchmarks. The first four value columns of this table are for the backbone of the bipartite graph with the most edges out of the combinations of bipartite graphs from the four colors with the most vertices. The second set of four columns is for the second greatest backbone from those conditions.

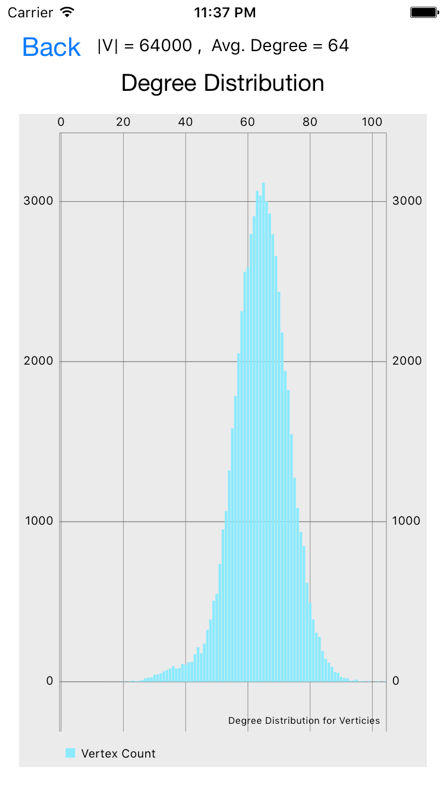
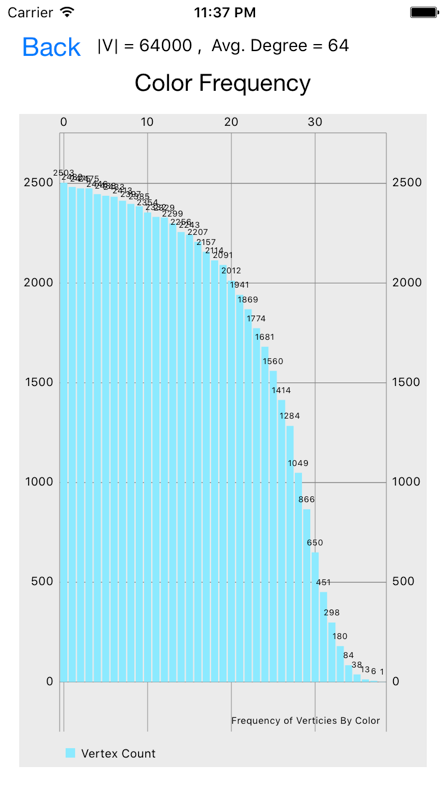
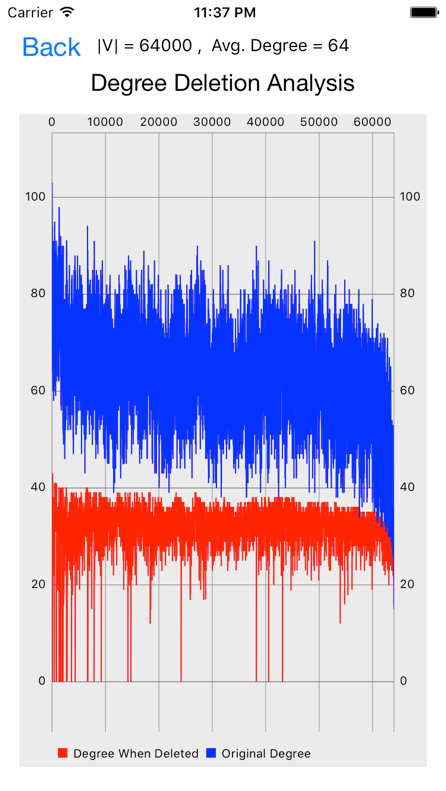
Benchmark 1

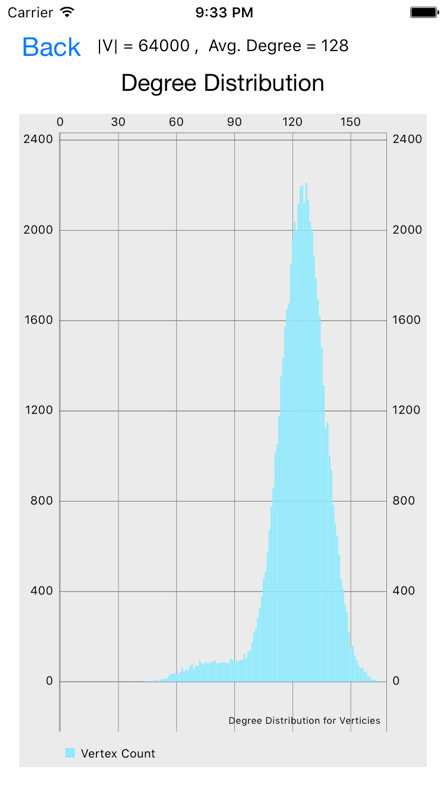
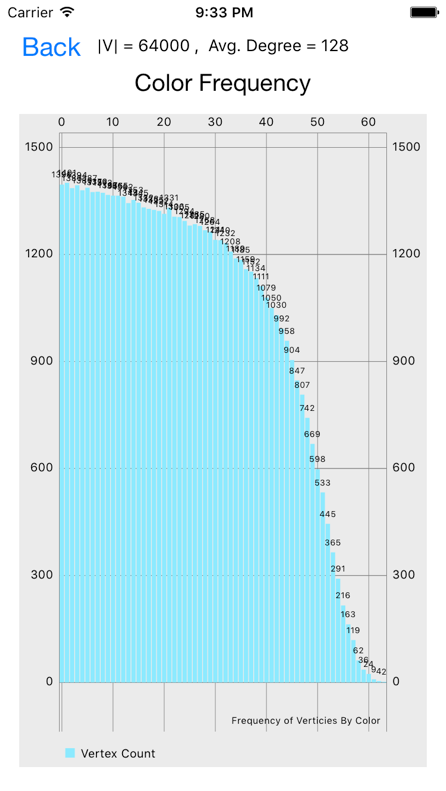
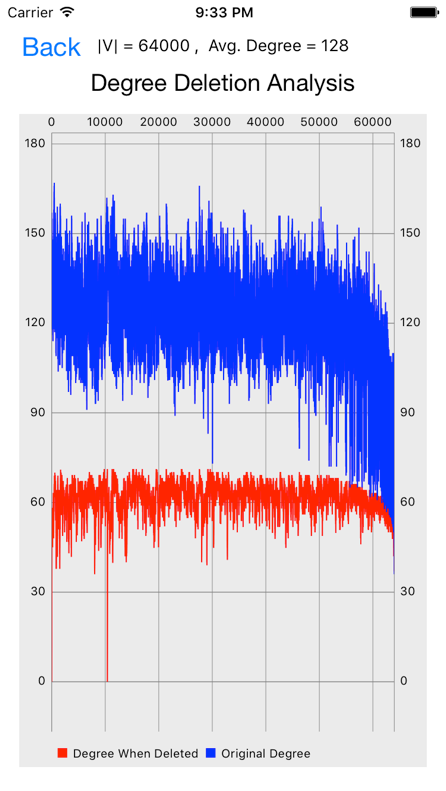
Benchmark 2

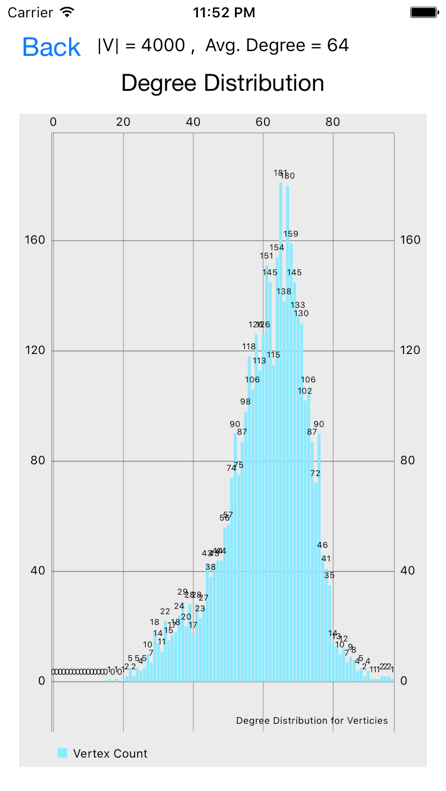
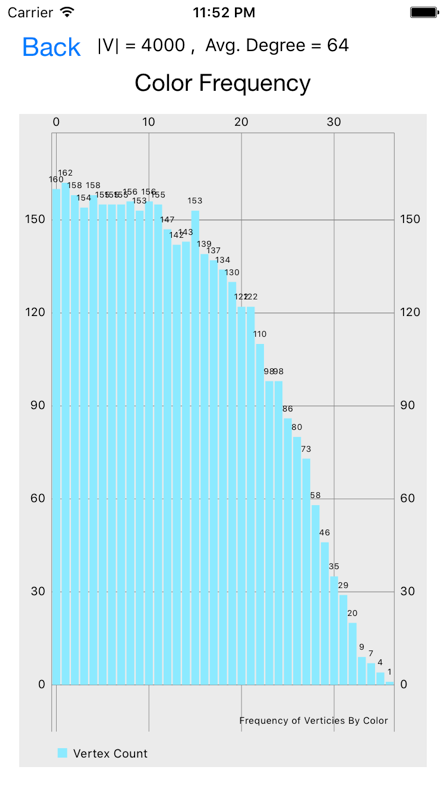
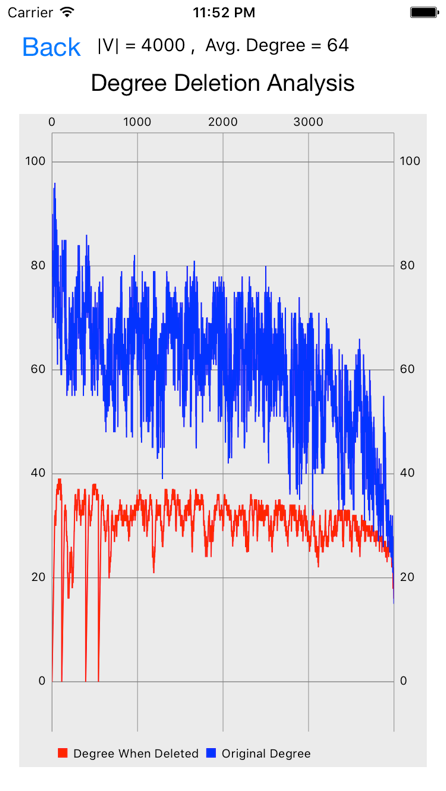
Benchmark 3

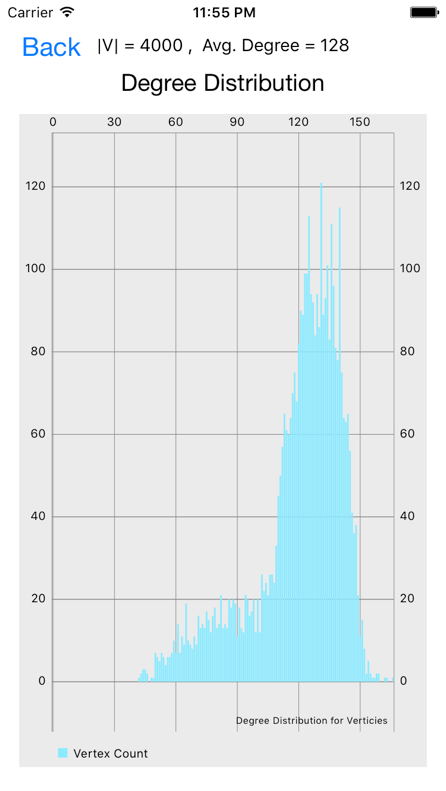
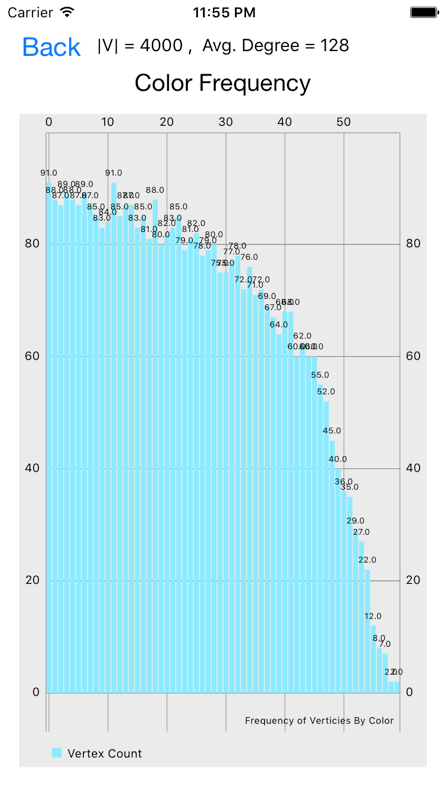
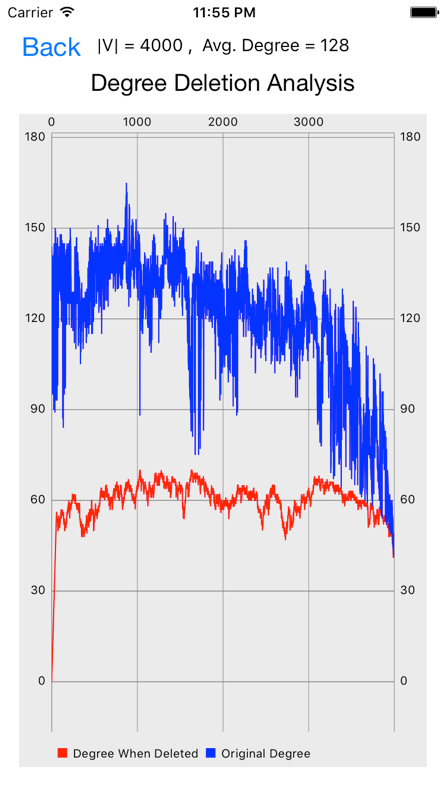
Benchmark 4

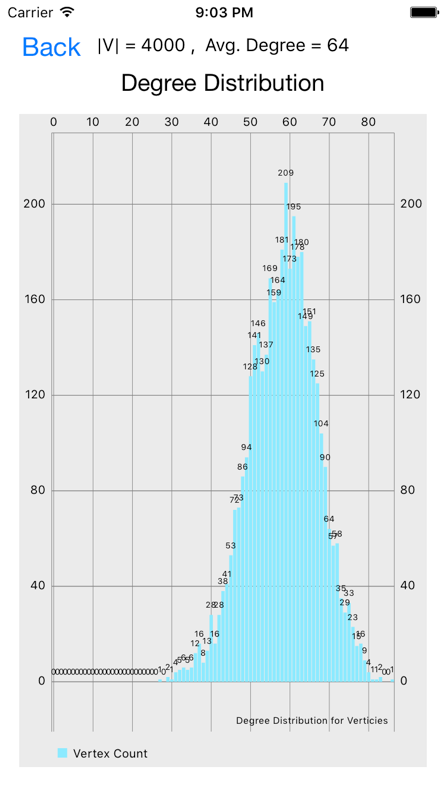
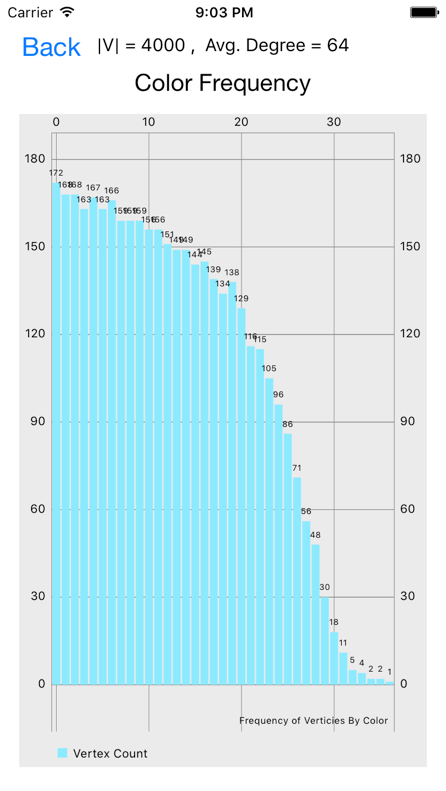
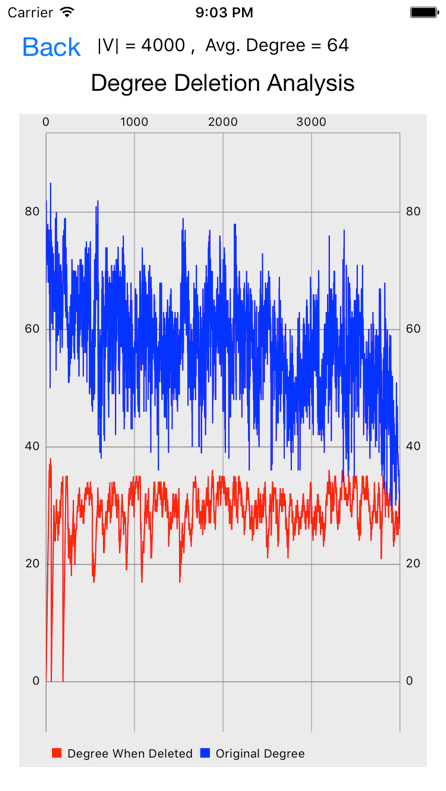
Benchmark 5

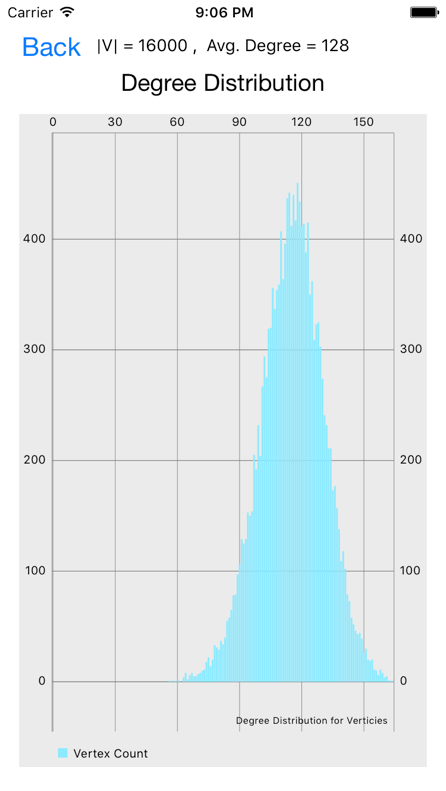
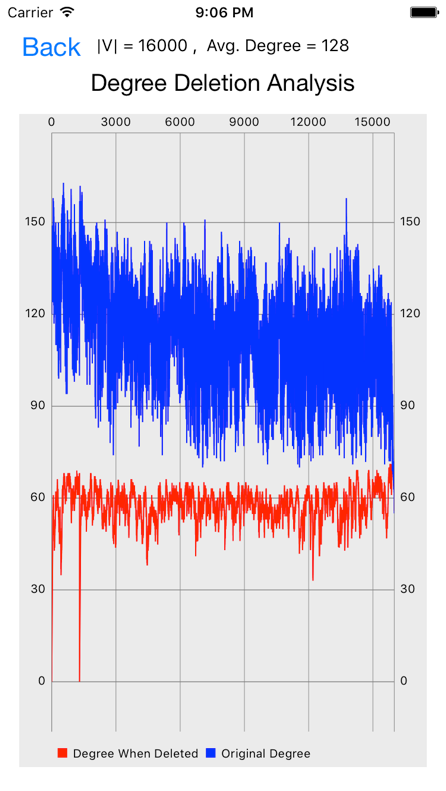
Benchmark 6

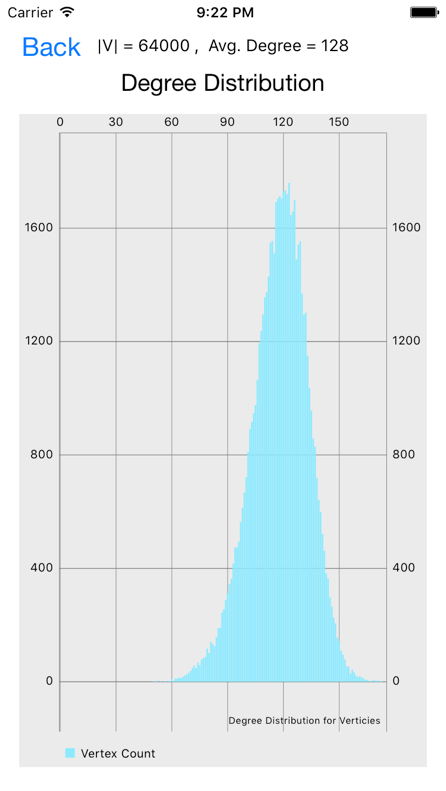
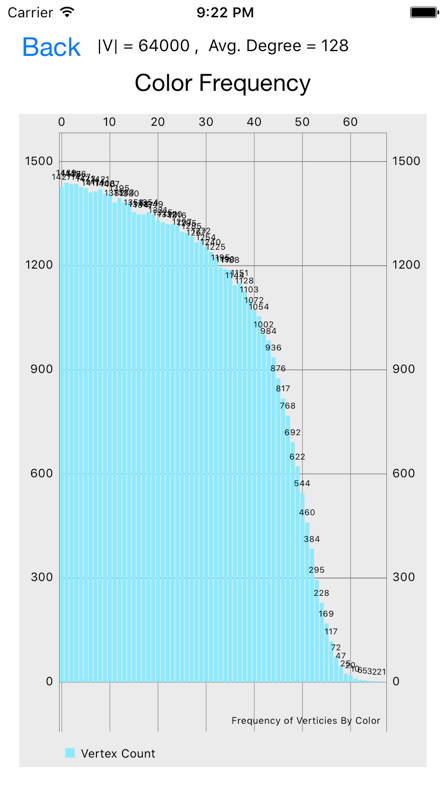
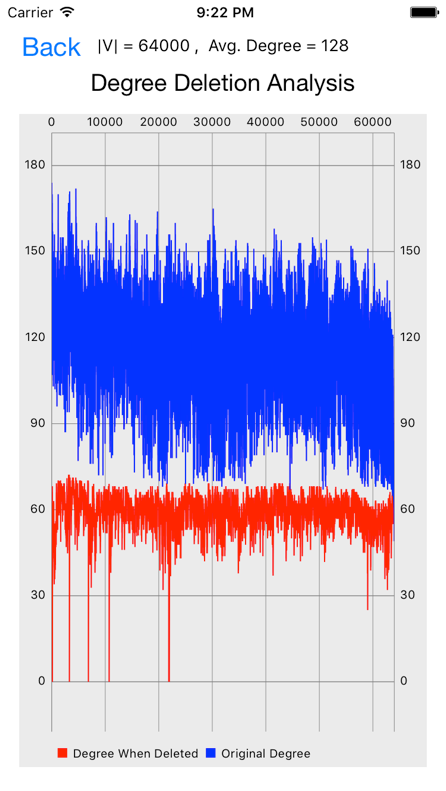
Benchmark 7

Benchmark 8

Benchmark 9

Benchmark 10

Generating graph is O( |V| \* average\_degree )

~~Coloring graph by smallest last algorithm is O( |V|^2 + lg(average\_degree) )~~

Smallest Last Ordering is O( |V|^2 )

Coloring graph is O( |V| + average\_degree )

https://arxiv.org/pdf/1207.1875.pdf

http://sc2xx8ju8d.search.serialssolutions.com/?ctx\_ver=Z39.88-2004&ctx\_enc=info%3Aofi%2Fenc%3AUTF-8&rfr\_id=info%3Asid%2Fsummon.serialssolutions.com&rft\_val\_fmt=info%3Aofi%2Ffmt%3Akev%3Amtx%3Ajournal&rft.genre=article&rft.atitle=Smallest-last+ordering+and+clustering+and+graph+coloring+algorithms&rft.jtitle=Journal+of+the+ACM+%28JACM%29&rft.au=Matula%2C+David&rft.au=Beck%2C+Leland&rft.date=1983-07-01&rft.pub=ACM&rft.issn=0004-5411&rft.eissn=1557-735X&rft.volume=30&rft.issue=3&rft.spage=417&rft.epage=427&rft\_id=info:doi/10.1145%2F2402.322385&rft.externalDocID=322385&paramdict=en-US