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.

## 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.

## 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.

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 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 eight surrounding test cells for each cell with nodes in it where nodes may be in for possible connections.

[Insert image of cell method]

For displaying the sphere, I use a dynamic programming solution by keeping a table of the already created node vectors required for 3D display using SceneKit. Even with this solution, though, the display has a limited ability to display and move for large vertex and edge counts.

### 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 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|). This is shown in \_\_\_ chart. The deletion of a vertex from the adjacency list causes this |V|2 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.

### Part 3

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 calculate the number of faces on a backbone, I use the formula F + V – E = 2, or rather F = E – V + 2.

## 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 for networks with up to 16000 nodes.

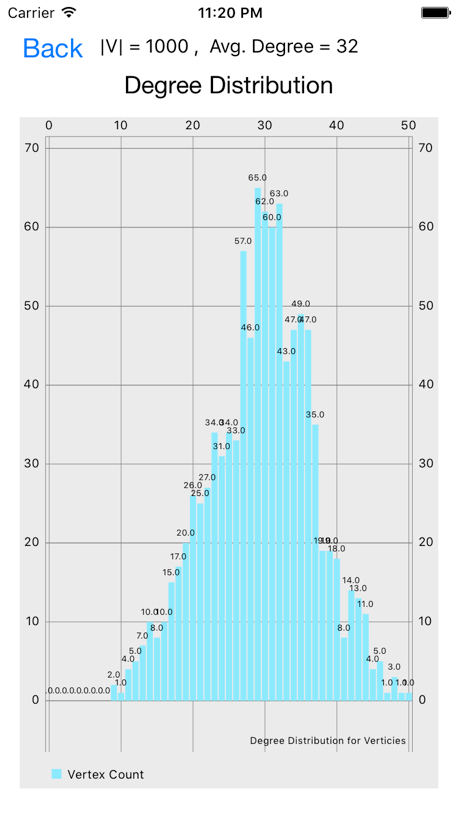
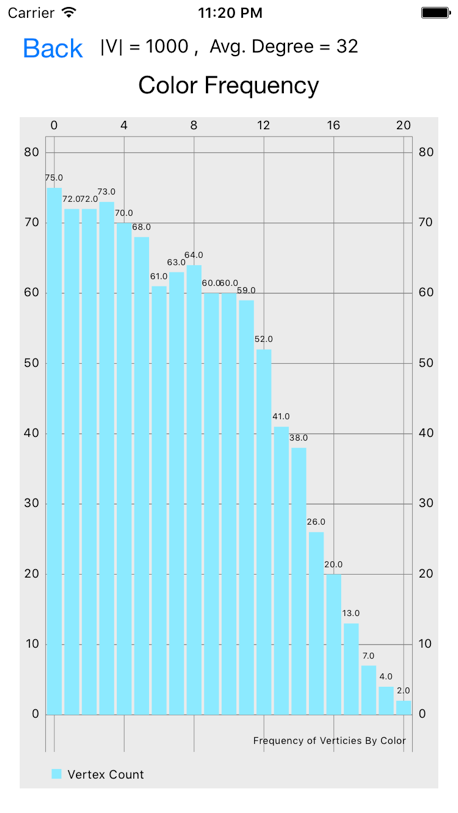
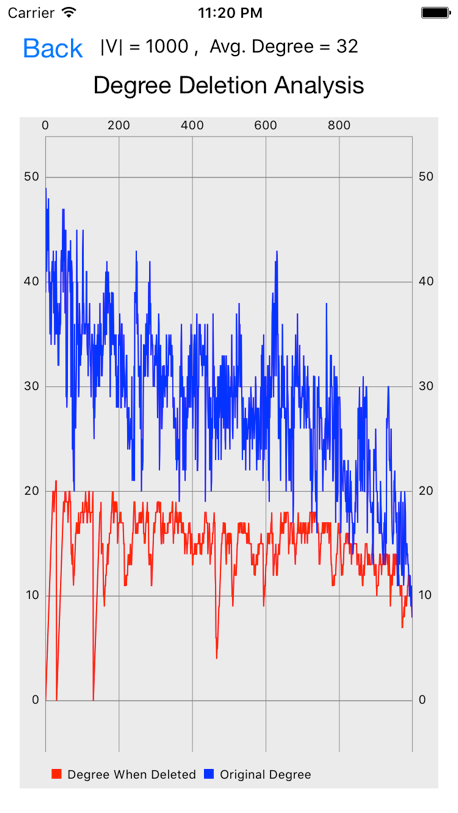
Based on the given benchmarks given for this assignment, the following tables shows the summary values calculated from running the application. The first table shows what the input values are for each benchmark. The benchmark numbers used in the following tables reflect the values in this first table.

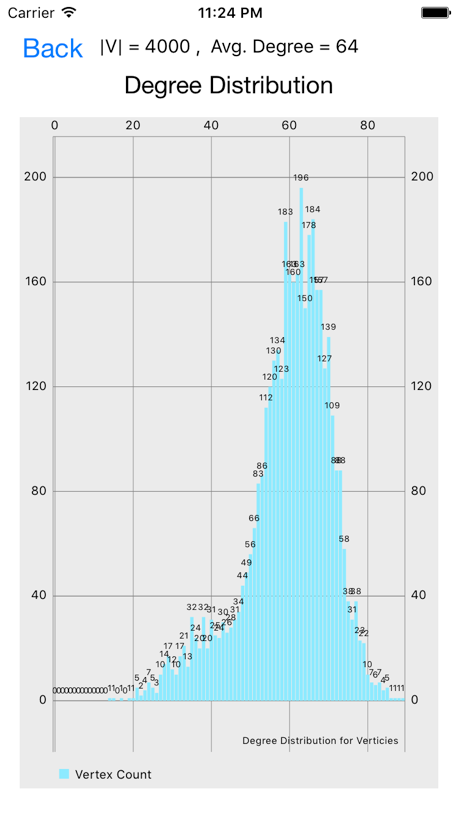
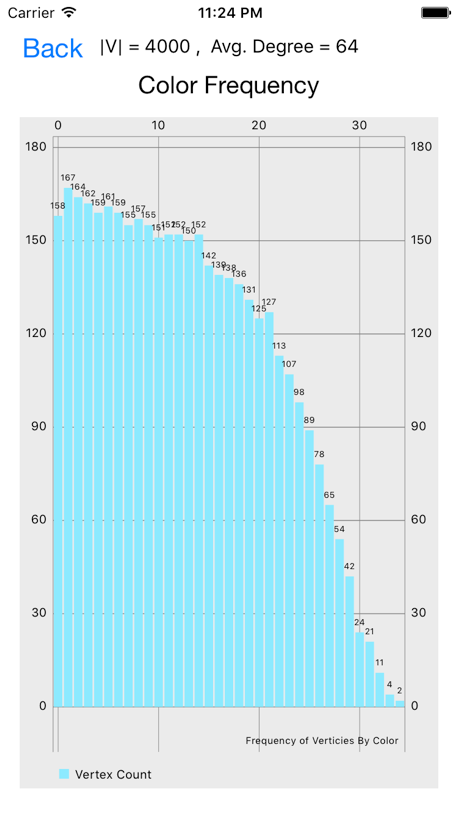
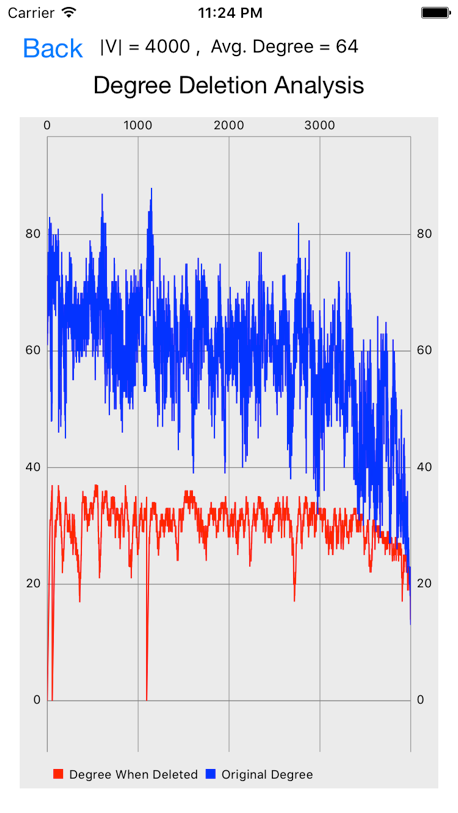
|  |  |  |  |
| --- | --- | --- | --- |
| Benchmark # | N | Avg. Degree | Distribution |
| 1 | 1000 | 32 | Square |
| 2 | 4000 | 64 | Square |
| 3 | 16000 | 64 | Square |
| 4 | 64000 | 64 | Square |
| 5 | 64000 | 128 | Square |
| 6 | 4000 | 64 | Disk |
| 7 | 4000 | 128 | Disk |
| 8 | 4000 | 64 | Sphere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

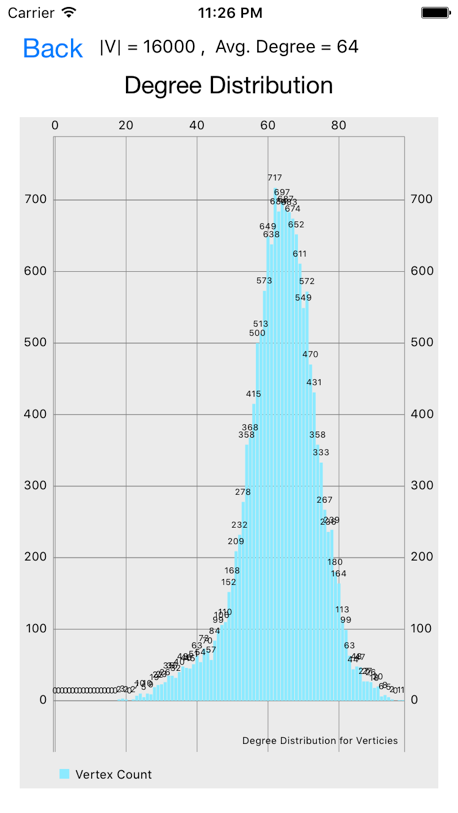
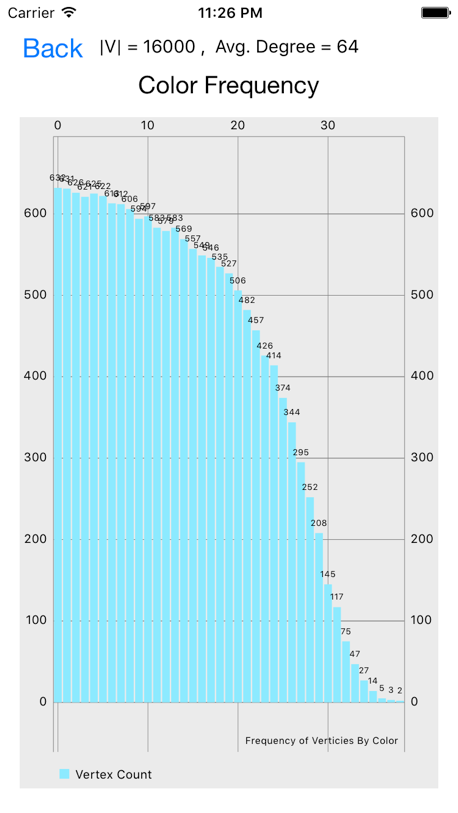
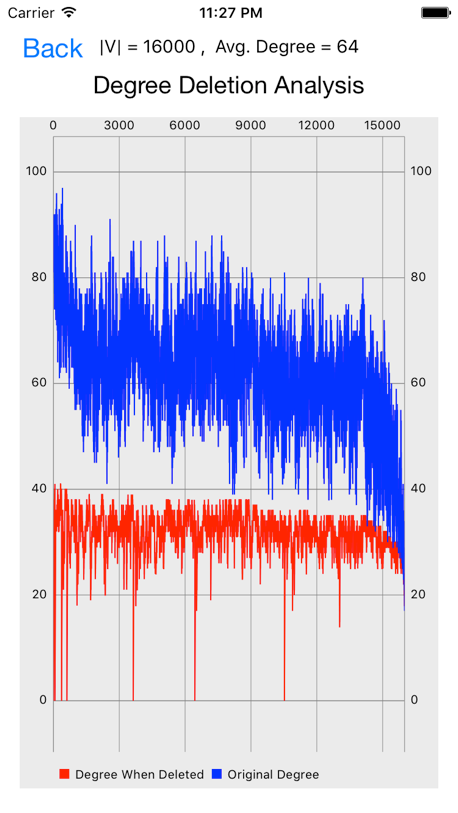
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Benchmark # | Number of Edges (|E|) | Connection Distance (R) | Min Degree | Avg. Degree (realized) | Max Degree | Max Degree when Deleted | Max Edges in a Bipartite Subgraph | Graph Creation 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 |

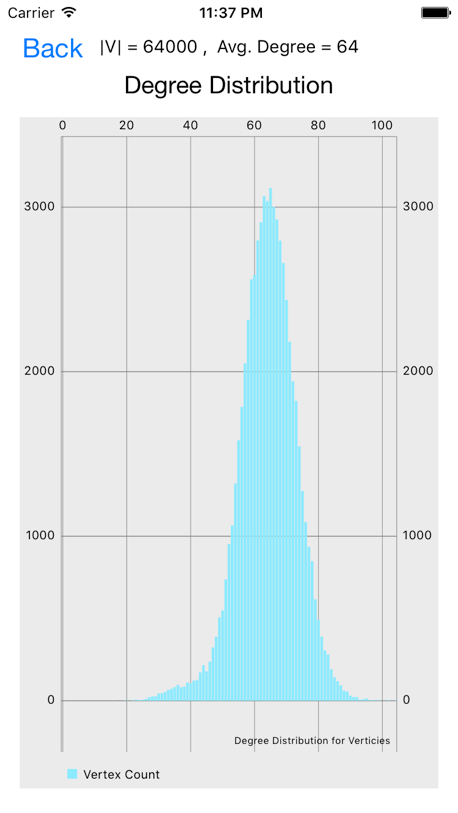
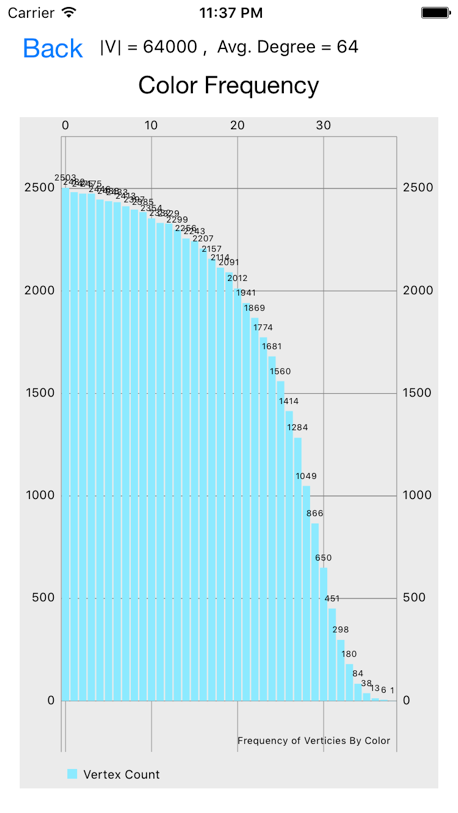
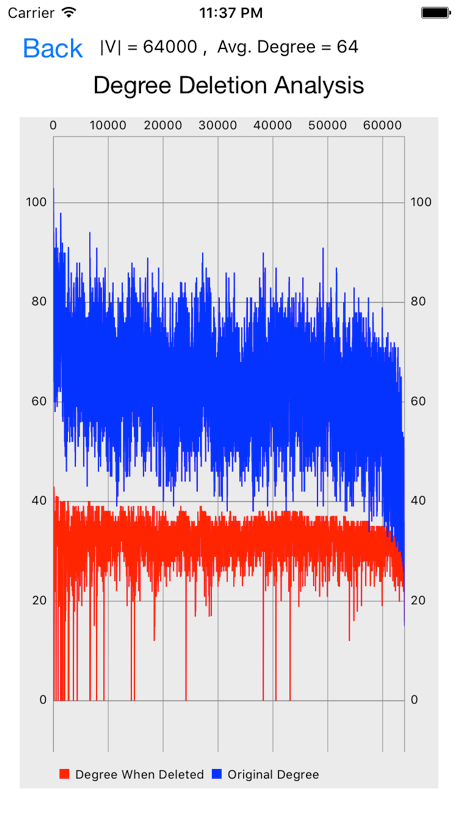
This table shows the output values from running the benchmarks. Each column adds additional information about the network. The “Number of Edges (|E|)” is the number of edges generated, the “Connection Distance (R)” is the calculated max value for two adjacent nodes to connect, the “Min Degree” is the 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.

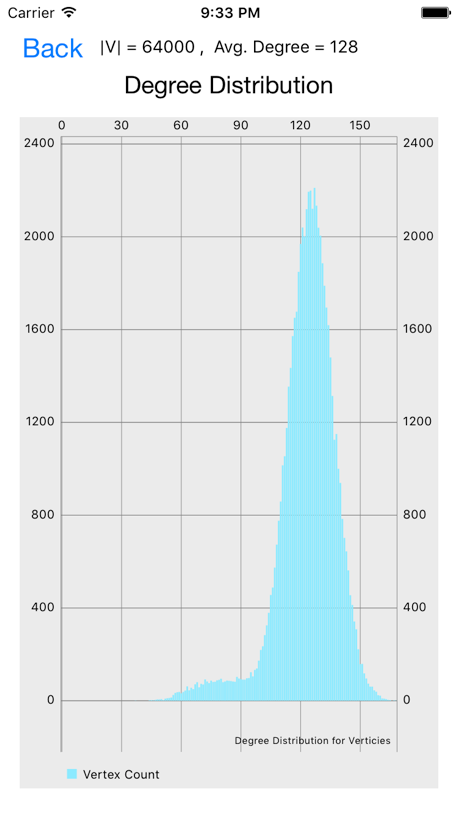
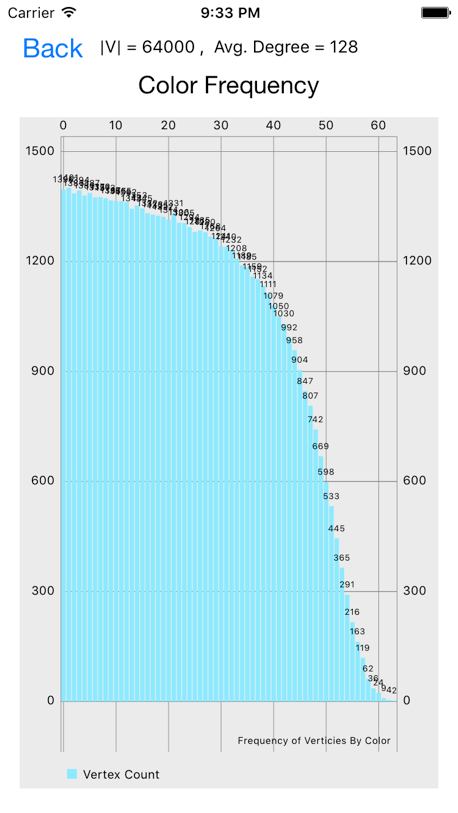
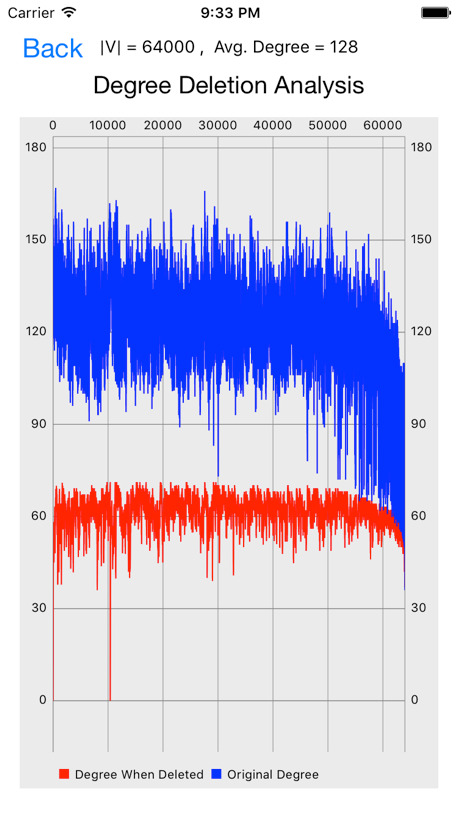
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Benchmark # | Backbone 1 Vertices | Backbone 1 Edges | Backbone 1 Domination Percentage | Backbone 1 Faces | Backbone 2 Vertices | Backbone 2 Edges | Backbone 2 Domination Percentage | Backbone 2 Faces |
| 1 | 142 | 199 | 99.3 | X | 141 | 194 | 99.1 | X |
| 2 | 327 | 495 | 99.83 | X | 325 | 494 | 99.7 | X |
| 3 | 1248 | 1923 | 99.91 | X | 1249 | 1889 | 99.86 | X |
| 4 | 4947 | 7533 | 99.91 | X | 4931 | 7533 | 99.89 | X |
| 5 | 2779 | 4394 | 99.95 | X | 2785 | 4387 | 99.98 | X |
| 6 | 316 | 471 | 99.73 | X | 310 | 464 | 99.33 | X |
| 7 | 180 | 289 | 100 | X | 180 | 288 | 100 | X |
| 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 |

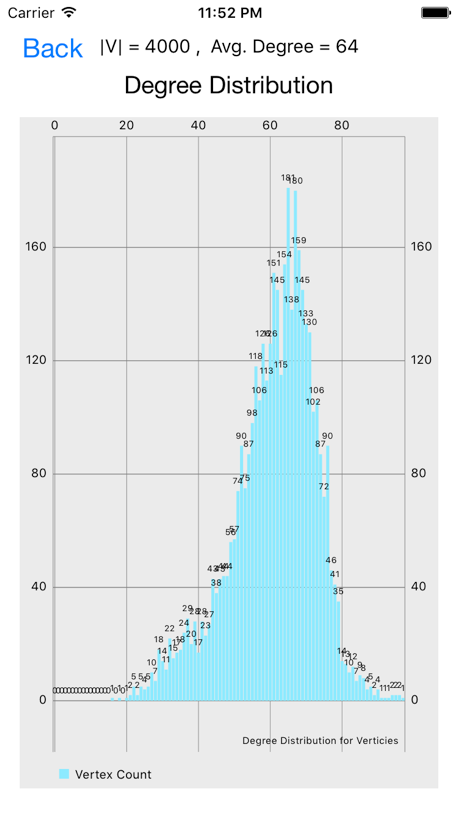
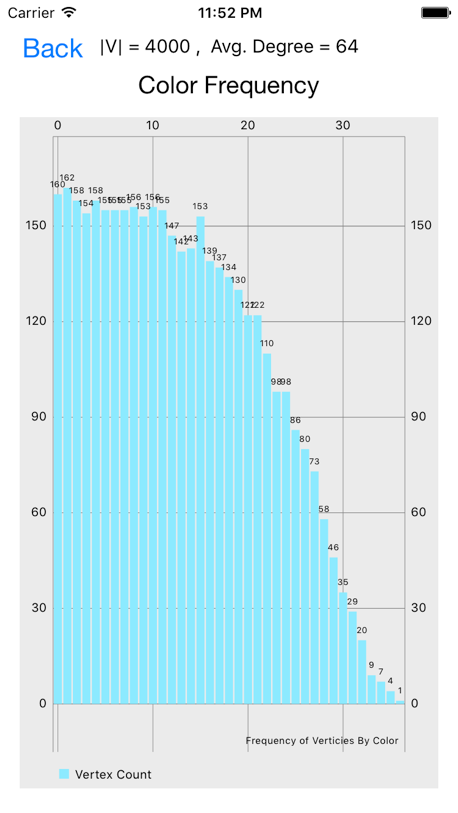
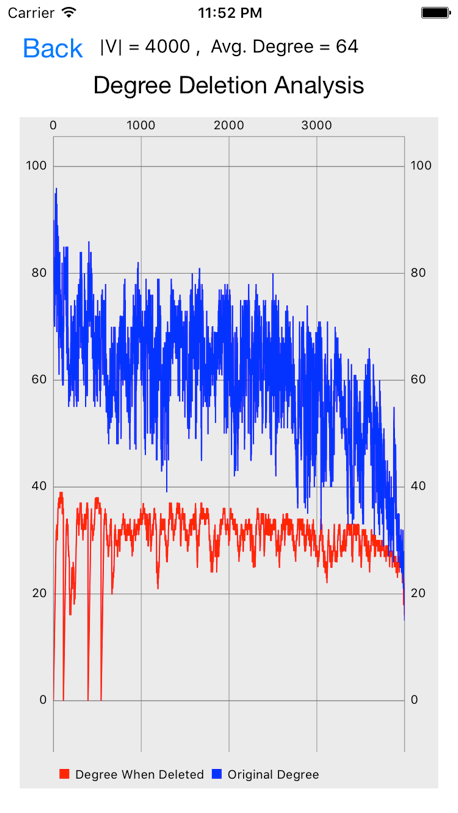
  

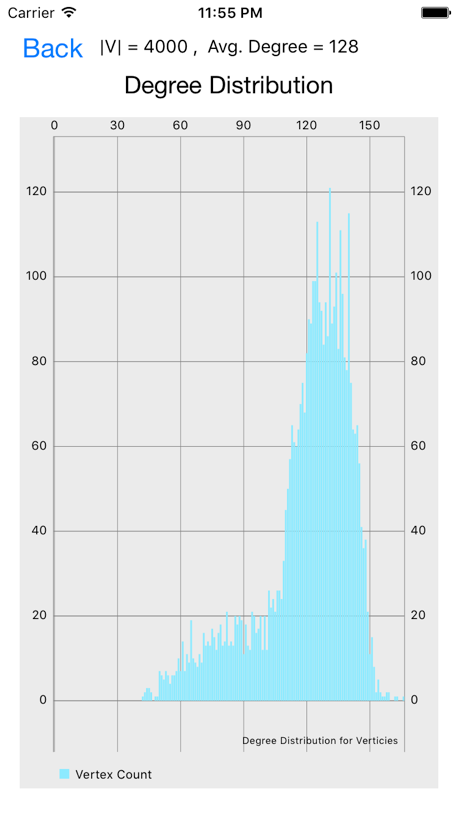
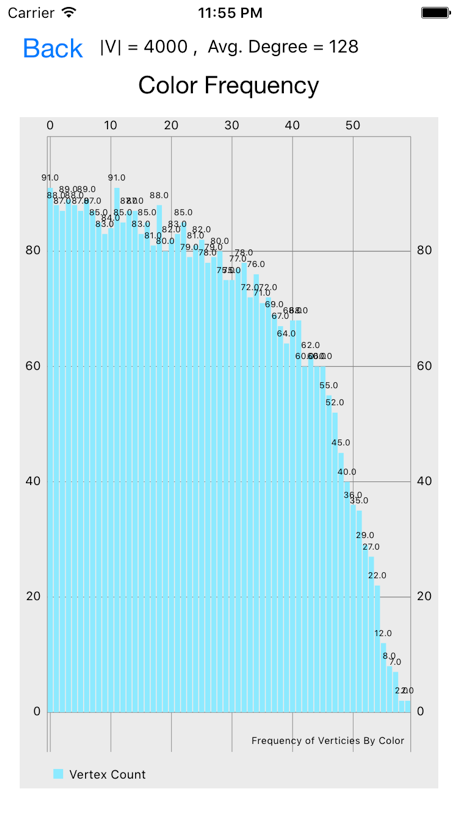
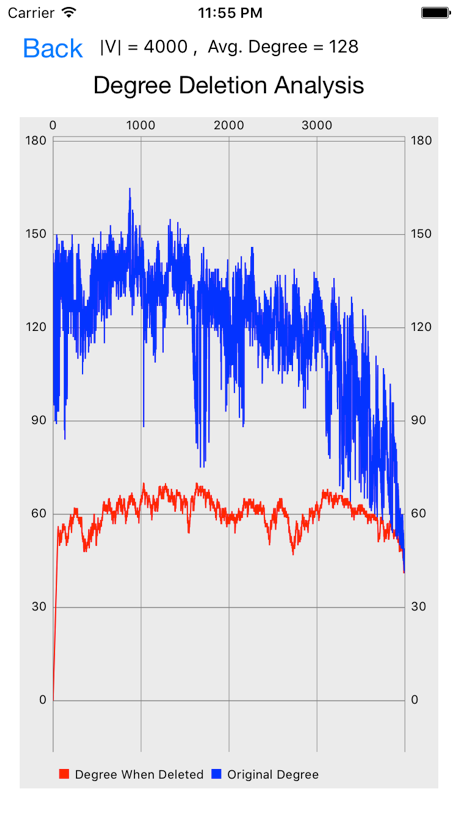
  

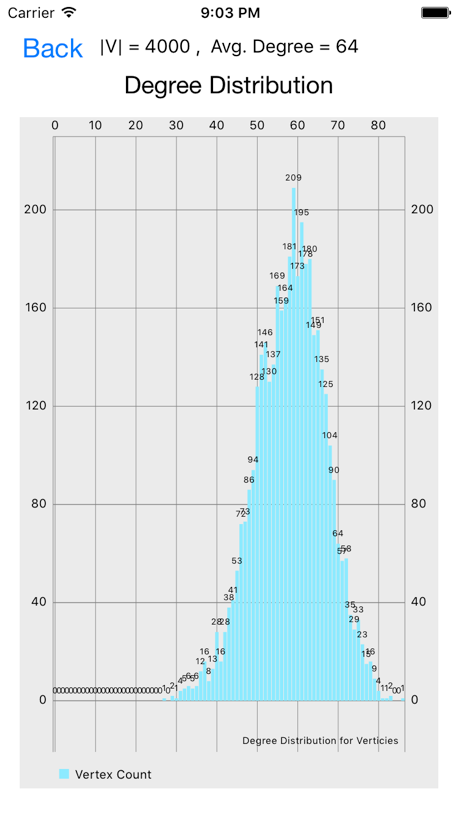
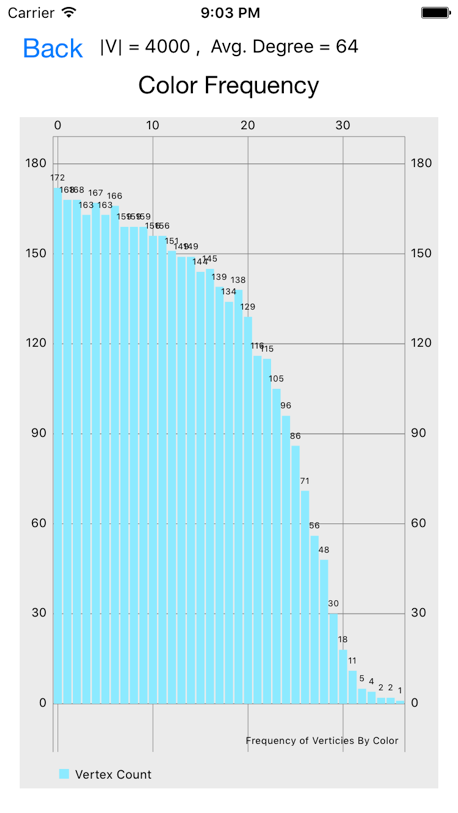
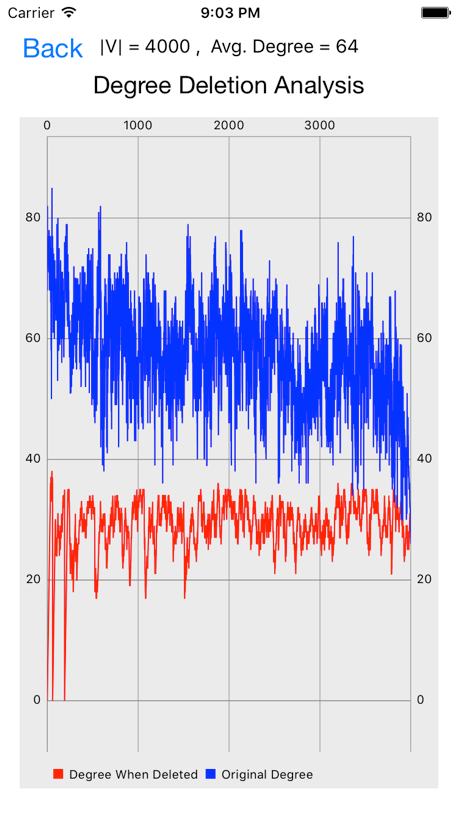
  

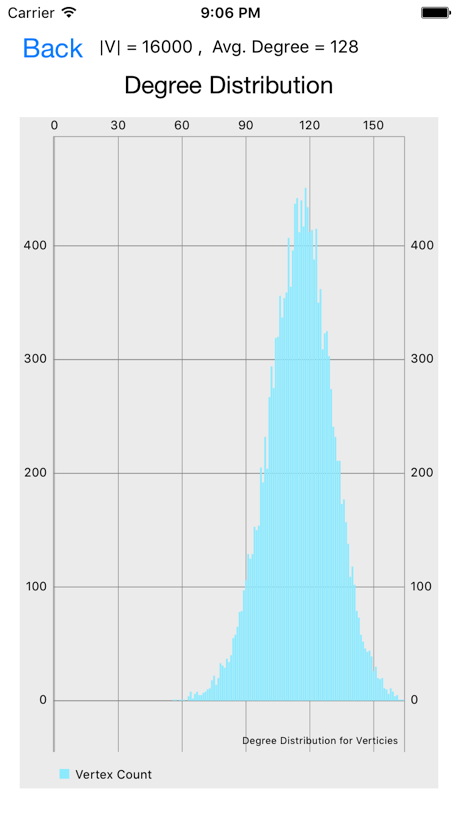
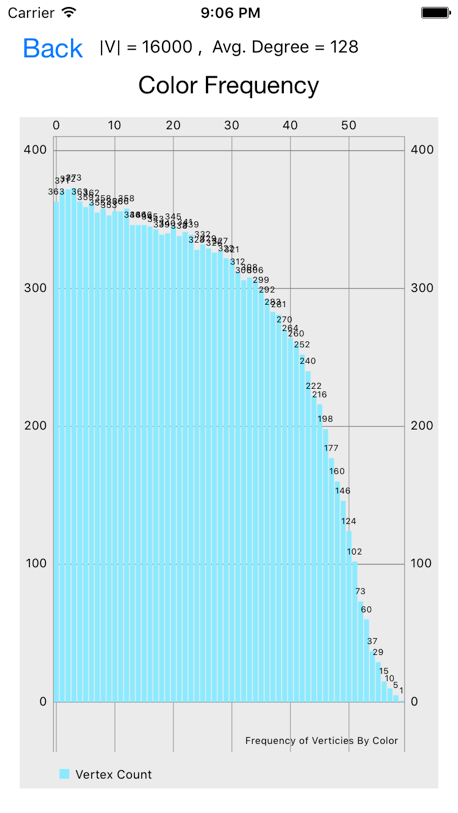
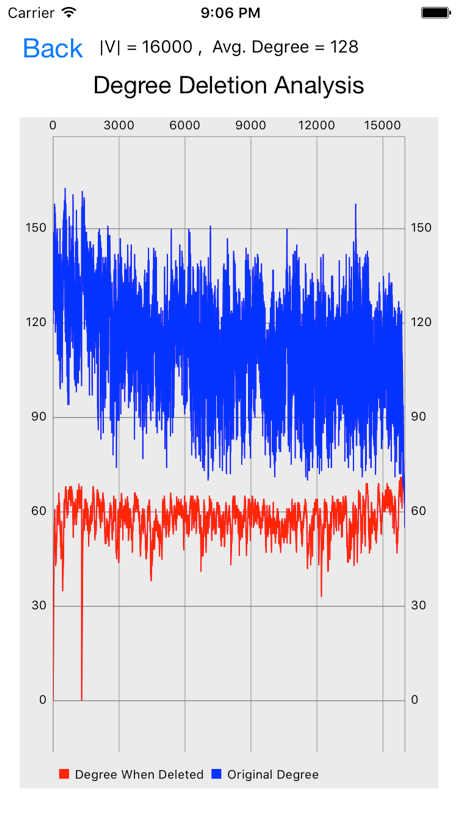
  

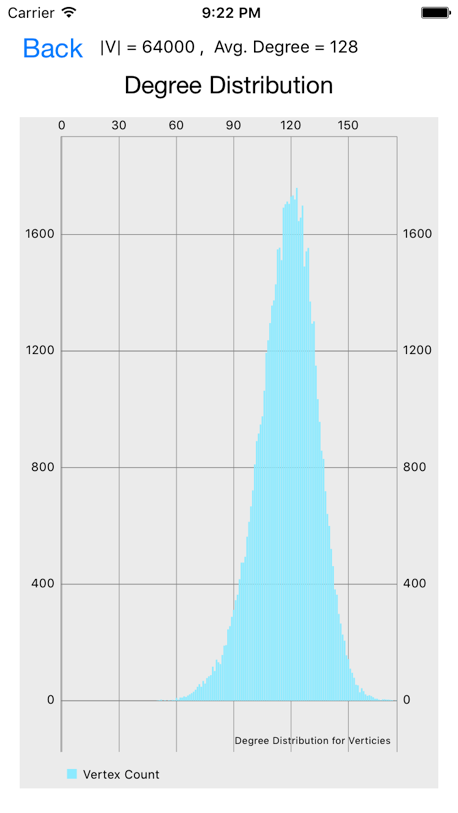
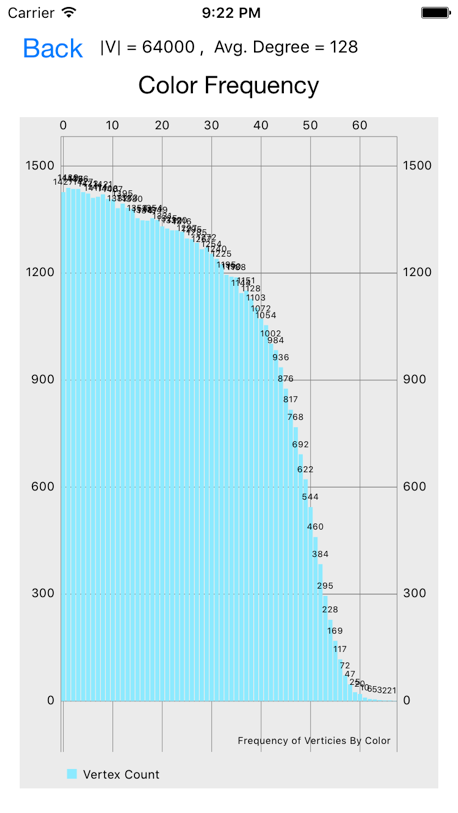
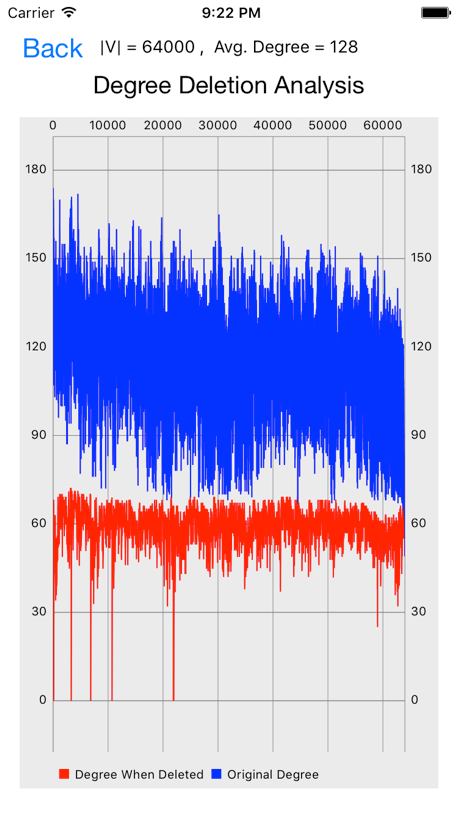
  

Generating graph is O( |V| \* average\_degree )

~~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