Algorithm Engineering – CSE 7350

Wireless Sensor Networks

Term Project

Due 12/15/15

Spencer Kaiser – 28750375

Southern Methodist University – Lyle School of Engineering

Table of Contents

1. Executive Summary

1.1. Introduction

Wireless Sensor Networks (WSN) are present in a plethora of disciplines and their utilization effects the way individuals interact with their environment on a daily basis. The use of Wireless Sensor Networks has a substantial impact upon a wide variety of industries, from health care to military surveillance, to environmental science [10], and their study has lead to great strides in increasing the performance with which data can be gathered and transmitted throughout large, spanning networks.

From a simple and abstract perspective, a Wireless Sensor Network is nothing more than a series of nodes which communicate with other nodes that exist within their range. The efficiency with which non-adjacent nodes can communicate entirely depends on the distribution of nodes in the network and the algorithm which directs traffic. As a result, the overall performance of a network can drastically vary depending on the implementation of the algorithm which identifies connections between nodes and groups of nodes in the greater network.

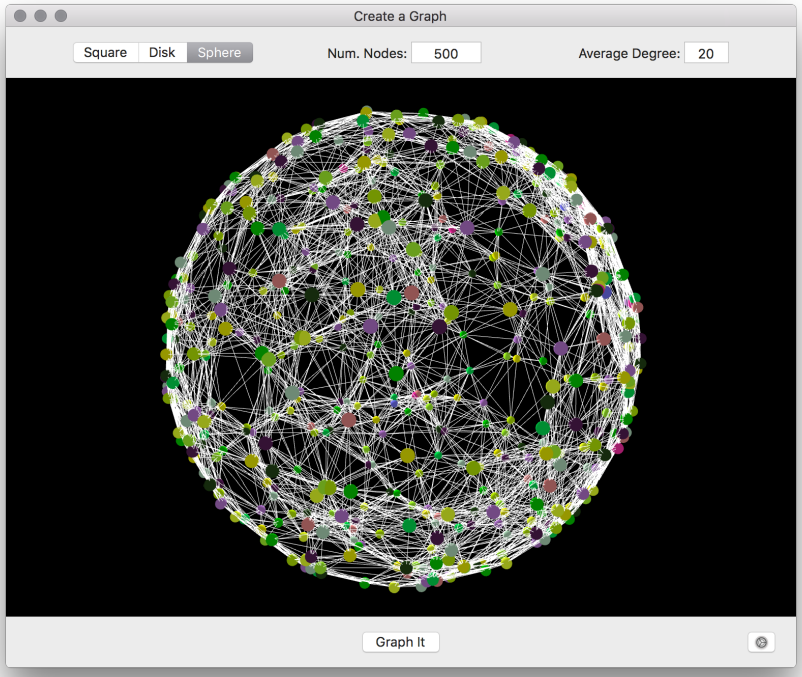
The primary purpose of this report is to discuss the evaluation and results of implementing a series of algorithms to generate, analyze, and test theoretical Wireless Sensor Networks to better understand the implications of their impact upon a network structure and connectivity. The following sections of this report will discuss summarized results, the environment utilized to perform this analysis, intricate details of the implementation, and project outcomes.

1.2. Report Synopsis

The end result of this project is a stand-alone Mac OS X application which can be used to generate simulated Wireless Sensor Networks and visualize a variety of additional characteristics of their structure and composition. The app is composed of a single graphical user interface which allows the user to simulate Wireless Sensor Networks with a given structure of either a planar square, a planar circle, or a sphere. These visualizations can then be used to gather valuable insight such as the composition of the network and connections between nodes and groups of nodes in the network.

The application is a very efficient tool for simple Wireless Sensor Network simulation and visualization, however, as a result of the chosen languages and libraries used in the implementation, there are several limitations that affect the range of the apps usefulness. Regarding visualization, there is an upper bound on the

network size which can be efficiently rendered. This upper bound occurs at approximately 65,000 objects, however, this limitation only applies to the visualization component of the application as the user can still generate additional data about the network at substantially larger object counts.



Overall, the app, pictured right, performs as required for the generation of simulated Wireless Sensor Networks and their analysis. The strengths of the application are derived from the end-user's ability to use the application without training or extensive background knowledge of the subject. Furthermore, because the application is interactive and allows the user to manipulate the network in real time, the overall user experience is excellent.

The table below shows preliminary results for the 10 proposed benchmarks. Additional data can be found in later sections of this report.

Benchmark	Min. Degree	Max. Degree	Max Min-Degree	Num. Edges (M)	Num. Edges in Largest Bipartite Subgraph	% of Verties Covered	Num. Colors	Terminal Clique Size
1	6	48	20	14,348	184	1.3%	22	22
2	13	65	27	77,834	576	0.7%	28	28
3	11	82	33	114,992	439	0.4%	34	33
4	13	95	36	474,538	1,712	0.4%	37	35
5	14	96	39	1,923,520	6,795	0.4%	40	40
6	19	83	33	114,759	426	0.4%	34	31
7	38	152	54	224,753	242	0.1%	58	50
8	36	88	32	120,416	418	0.3%	34	34
9	64	158	60	959,601	989	0.1%	61	52
10	64	165	61	3,841,289	3,931	0.1%	61	55

1.3. Programming Environment

The programming environment utilized during the course of this project consists of an assortment of Mac hardware and software. The machine used to perform all analysis and program execution is a 2013 Retina MacBook Pro running the Mac OS X El Capitan (10.11.1) Operating System. Furthermore, the hardware consists of a 2.0GHz quad-core Intel Core i7 processor, which is utilized by the OS to offer "Turbo Boost" speeds of up to 3.2GHz, a 6MB shared L3 cache, and 8GB of 1600 MHz DDR3 RAM.

The application created during the course of this project was written in the Objective-C language, a superset of the C language which "provides object-oriented capabilities and a dynamic runtime" [7]. Objective-C is one of two languages which can be used to create applications for OS X as well as iOS, Apple's smartphone operating system.

Objective-C was chosen for this project for a few primary reasons. First and foremost, it is the language I am most proficient with and have had substantial experience with. Objective-C is a well structure language which can be used to create scalable and efficient applications. Furthermore, Objective-C was chosen as the primary language for this project in order to enhance the end-user experience as Objective-C. Through the utilization of Objective-C, the created program is an all-in-one application with a simple user interface which allows users to create wireless sensor networks with specific numbers of nodes, average degrees, and structure type. As a result, the application can be used by a wide variety of individuals ranging from those with substantial experience with RGGs and wireless sensor networks to those with absolutely no experience.

One significant drawback of the utilization of Objective-C is the ability to export raw data from the application. Due to the nature of OS X applications and Operating System restrictions, exporting data to the file system requires a substantial amount of effort. As an alternative, there are several libraries which can be utilized to produce data visualization within the app itself, however, these libraries have a steep learning curve and limited documentation. Due to these limitations, several requested data representations have been omitted from this report.

In order to visualize data created by the application, the SceneKit library was used. SceneKit is "an Objective-C framework for building apps and games that use 3D graphics, combining a high-performance rendering engine with a high-level, descriptive AP" [8]. SceneKit seamlessly integrates with Objective-C to create beautiful representations of geometric shapes which, for this particular application, consists of a network of interconnected nodes. Not only does SceneKit represent networks in a very useful way, but it also allows for the

user to manipulate the visualization. The user can use the mouse to manipulate the visualizations orientation, zoom in on components of the graph, and pan in all directions. In addition, the user can generate additional graphs, modify network characteristics, and change structure type all within the app.

While SceneKit is a very powerful library and it's integration with Objective-C is seamless, it was not intended to perform as needed for the scope of this project. After a certain threshold of objects, consisting of both nodes and edges between nodes, the performance of SceneKit begins to degrade. I performed extensive research into the matter and discovered several useful optimizations. Initially, I decided to utilize geometric cylinder objects to link nodes, however, after discussing the matter with a classmate and performing additional research, I implemented edges by using a primitive data type belonging to the SceneKit library [4]. Similarly, after requesting help on the popular site Stack Overflow, another user introduced me to a method which can essentially create a flattened version of the visualization, which did increase overall performance [5]. While there are some useful tips for increasing SceneKit's performance [3], those optimizations only increase the threshold of limitations. Furthermore, after exceeding this threshold, which seems to occur after attempting to render approximately 65,000 objects, SceneKit begins to experience unpredictable behavior and eventually, the entire application crashes [5].

Although the use of SceneKit did introduce several limitations, the overall performance of the application was very respectable. Through the utilization of efficient data structures, efficient memory management, and thorough memory analysis, the overall memory usage of the application is minimal when compared to simpler, more abstract methods. Similarly, through the use of efficient algorithms and due to the fact that Objective-C is a superset of the efficient C language [8], the running time is minimized, even for large values of n. Additional data and discussion of performance can be seen in *Section 3*.

Overall, the programming environment and languages utilized for this project resulted in a user-friendly application which allows for impressive visualization of wireless sensor networks without substantial consumption of memory, however, certain limitations place restrictions of the quantity of data that can be effectively represented. Similarly, environment restrictions make exporting raw data from the app very difficult, which impacts the overall benefit of the app. This application is best suited for users who wish to gain additional insight into Wireless Sensor Networks with the aid of an interactive and user-friendly visualization tool or for those that do not require visualization of extensively large networks.

2. Reduction to Practice

2.1. Data Structure Design

Throughout the duration of this project, all choices regarding utilized data structures revolved around the efficiency with which data could be retrieved. Although this was the most important consideration when initially creating data structures, there was also a large emphasis placed on portability and being able to utilize one source of data throughout the app lifecycle. In order to accomplish this, several classes were implemented.

First and foremost, the Node class was created. In order to allow for the retrieval of core information in constant time, the node class was created and given a large set of instance variables. These variables consist of data such as graph coordinate information, degree when deleted during the Smallest Last First vertex ordering, color, and bipartite subgraph membership. The node class also contains a set of essential arrays and dictionaries which contain data about connected vertices as well as SceneKit objects used for visualization.

In order to accommodate several types of Random Geometric Graphs, the Node class was ‘subclassed’, which is a form of inheritance employed by Objective-C. A Sphere Node and a Circle Node were subclassed, each of which has custom, modified implementations for the method of generating uniformly random coordinates.

Next, a master dictionary, was utilized to contain the adjacency list. Upon initialization, the adjacency list dictionary is filled with node objects, each with random coordinates, and this adjacency list is referenced and modified throughout the remainder of the app lifecycle. This data structure is ideal for this type of utilization as it allow for constant time references to data objects, can be sorted based on key value, and allows for easy insertion and removal.

During the process of generating the Smallest Last First ordering of a given network, the bipartite graph selection process initially required a substantial amount of static code and hard coded variables. In order to implement a more dynamic approach and to increase the efficiency with which these bipartite subgraphs are chosen, a Bipartite Subgraph class was created. This class allows for constant time retrieval of subgraph data and includes an instance variable to maintain the degree count for a given combination of subgraphs.

2.2. Algorithm Descriptions

Perhaps the most intriguing component of this application is it's ability to generate Wireless Sensor Network "Backbones". The first step in the process of creating these backbones, was to implement the Smallest-Last First vertex ordering algorithm as described by Dr. Matula at Southern Methodist University [1].

The algorithm essentially operates by first ordering each vertex according to it's degree. This process, which may sound trivial, is essential for creating the correct ordering. During this step, vertices are placed into buckets based on their degree, however, because the order in which vertices are added and eventually removed must be preserved, the buckets containing these vertices must act as First-In-First-Out (FIFO) queues.

Next, the algorithm removes the first vertex from the smallest degree bucket. Once this vertex is removed, all of it's connected vertices must be updated. This process begins by identifying each connected vertex and decrementing its degree count by placing it in the next smallest degree bucket. As previously stated, the order with which vertices are placed into buckets is important, therefore, if multiple vertices exist in the same bucket, they are moved according to FIFO ordering.

This process continues until every vertex has been removed from the graph, at which point, the Smallest-Last ordering an be obtained by reversing the order in which the vertices were removed. While this step is the last component of the algorithm itself, there exists a previous state in the process which yields invaluable information. When the smallest degree bucket contains the same number of vertices as remain in the graph, a structure known as the "Terminal Clique" has been identified.

After the smallest-last ordering has been identified, a coloring algorithm can then be applied to the graph. This coloring algorithm begins by assigning a numerical color value to the first node in the smallest-last ordering. After assigning this value, the algorithm continues with the next vertex in the order, however, for each subsequent vertex, the color value assigned is equal to the lowest color value which is not present on any adjacent vertices.

The end result of this coloring process is a graph represented with the minimum amount of colors needed where each set of colors contains no edges connecting two vertices of the same color set.

2.3. Algorithm Engineering

In order to maximize the efficiency with which the Smallest-Last ordering algorithm and coloring algorithms, numerous optimizations to data structures and the algorithm itself were made. While many of these optimizations are discussed in the sections above, one substantial optimization is made to the algorithm itself.

After deleting a vertex during the Smallest-Last algorithm, the deleted vertex's connected vertices must be updated. Without optimizations, this process involves either the use of additional instance variables and requires costly searches for the vertices in the degree buckets. In order to significantly improve the efficiency with which this component of the algorithm operates, a hash table is used. Once all nodes have been placed into their respective degree buckets, their locations are then inserted into a hash map. At any point during the algorithm, at the deleted vertex's connected vertices can be identified in constant time and the degree bucket for each of these vertices can be identified by referencing the hash table, which also incurs a constant time operation.

Using these optimizations, a single pass through the vertices, $O(|V|)$, is made where each iteration decrements the degree buckets of the connected vertices. Because each edge is only examined once during the algorithm, just a single pass is made through all edges, incurring an additional time complexity of $O(|E|)$. Overall, the Smallest-Last ordering algorithm implemented by this application is bound by $O(|V| + |E|)$.

As discussed in preceding sections, optimizations were made to the data structures utilized during the execution of the graph coloring algorithm. Through the use of instance variables, which result constant time operations, all nodes in the graph can be colored with only a single pass through the vertices in edges. As a result, the graphing algorithm is also bound by $O(|V| + |E|)$.

2.4. Algorithm Effectiveness

Random Geometric Graph (RGG) creation within the application is very efficient and points within each graph structure are uniformly distributed. Utilizing the Node subclasses discussed above, each type of RGG node has its own specific method of generating coordinates. Each of these implementations utilize an Objective-C function, arc4random_uniform(), which is a uniformly distributed random number generation function [11] to generate points. For the square RGG, any point within the (1,1) plot is acceptable, however, points within the circular RGG and the spherical RGG required further manipulation. For the circle RGG, points must exist within the radius of the circle. In order to maintain uniform distribution, all points which lie outside the circle were discard and new values were calculated. Similarly, for the sphere RGG, all points lying outside the sphere

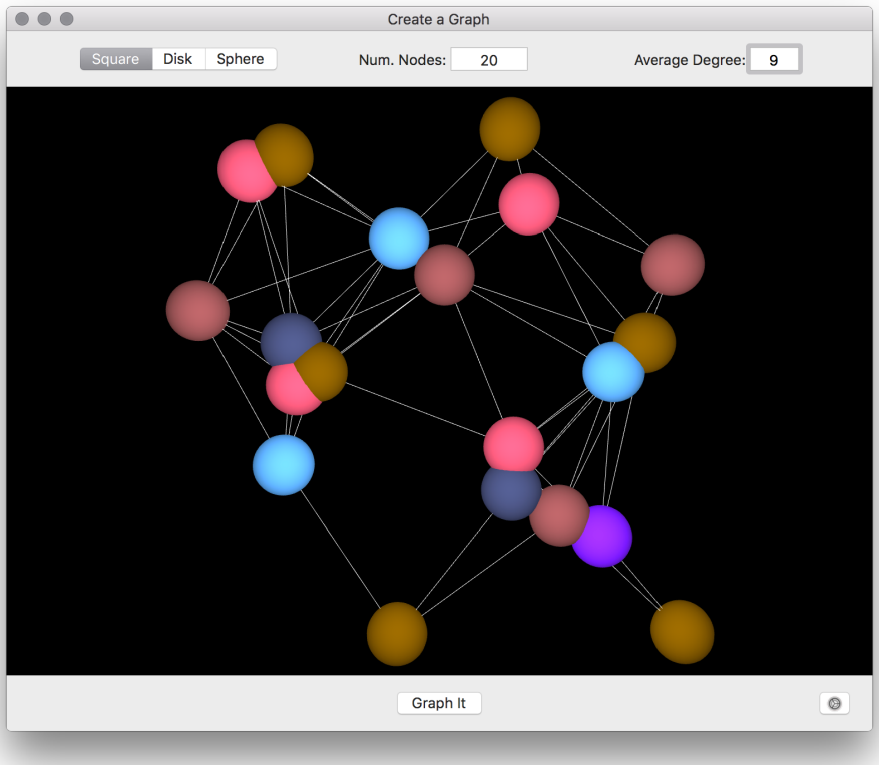
were discarded and recalculated. Points inside the sphere were normalized and migrated to the radius of the sphere. Each of these RGG implementations utilize uniform distribution, making them excellent for analysis.

The coloring algorithm implemented for this application is very thorough, yet efficient. After generating the Smallest-Last ordering, a single pass is made through the vertices and at each vertex, the smallest available color is calculated. In order to perform this calculation, the colors of all adjacent vertices are retrieved and then iterated through, checking for the lowest color which is not present in the list. While this portion of the algorithm could be improved with additional data structure optimizations, it is very efficient as implemented.

Finally, the bipartite backbone selection algorithm is also a well implemented algorithm. As opposed to utilizing hard coded variables for the possible combinations of the top four subgraphs, a specific class was implemented. As a result, the implementation is easy to read, simple, and many needed values during the execution can be retrieved in constant time. That being said, however, there is one component of the algorithm which was not optimized. After determining the total degree of the combination of two subgraphs, forming a bipartite subgraph, the algorithm iterates through all vertices and paints them as belonging to that specific bipartite subgraph. Unfortunately, if a vertex belongs to multiple bipartite subgraphs, it may be a part of multiple iterations. While a solution to this inefficiency was not identified during the course of this project, an optimization may be found to reduce the amount of time needed by this component of the execution.

2.5. Verification Walkthrough

In order to demonstrate the implementation of the Smallest-Last ordering algorithm, this section will walk through the execution of the algorithm upon a sample RGG of square type. The RGG shown on the right has a total number of nodes of 20 and a radius of 0.40.



As discussed in prior sections, the first step in the algorithm is to calculate the degree of each vertex

and place the numerical ID of each vertex in its corresponding degree bucket. After performing this calculation, the resulting degree buckets are shown in the table below, where each bucket contains a list of the nodes with that corresponding degree:

Degree	2	3	4	5	6	7	8	9	10
Node ID	7	4	0,14	1,2,15	5,6,13,17	3,10,16	9,11,12,19	18	8

At the same time, a structure containing degree mapping is created where each key in the structure is a Node ID and the associated value is a pointer to the degree bucket in which the node is currently located:

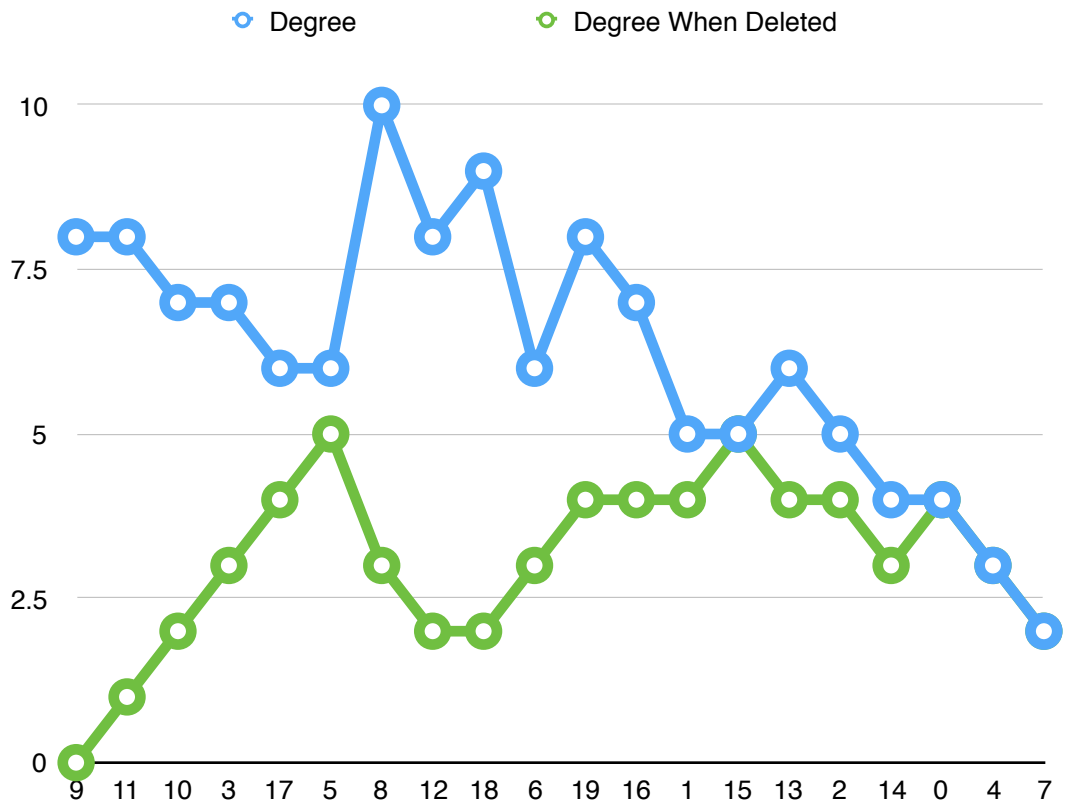
Node ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Degree Bucket	4	5	5	7	3	6	6	2	10	8	7	8	8	6	4	5	7	6	9	8

At this point, the lowest node, Node 1, is deleted from the graph. When this deletion occurs, the edges between it and its adjacent vertices is also deleted. As a result, the adjacent vertices have their degree decremented by one. When this happens, two things must take place; the vertex must move to the next smallest bucket and the degree mapping must be updated. Finally, after deletion has finished, the Node ID for the vertex which was remove is added to the Smallest-Last ordering.

This process continues until all vertices have been removed from the graph, at which point, the smallest-last ordering is equivalent to the revered order in which the vertices were deleted. For the graph above, the smallest-last ordering as as follows:

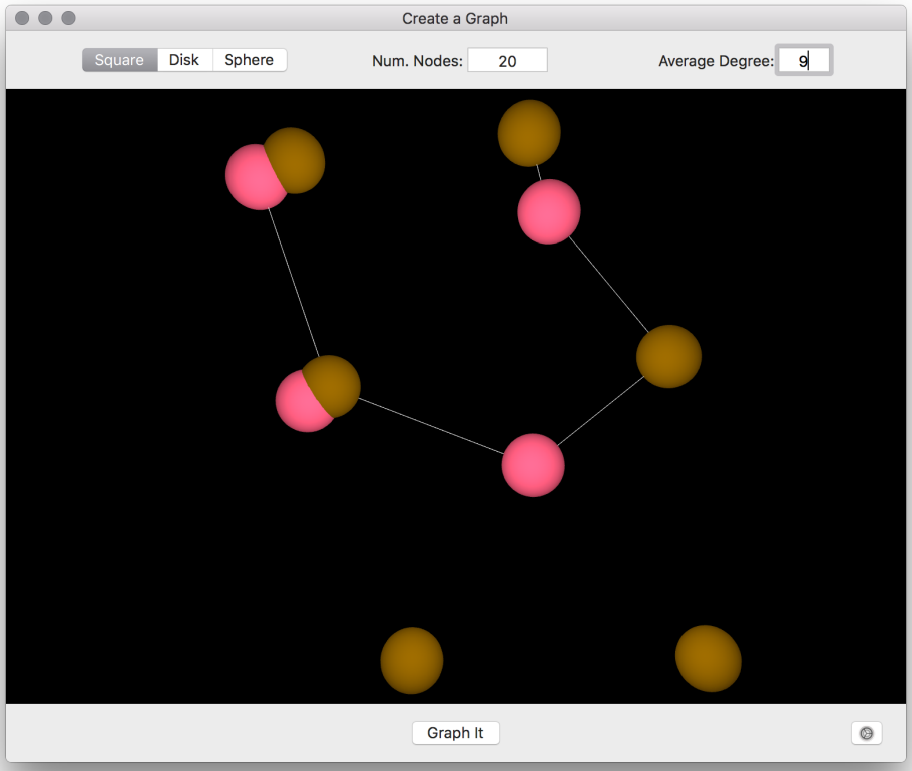
SLF Ordering	9	11	10	3	17	5	8	12	18	6	19	16	1	15	13	2	14	0	4	7

During the process of deletion, the degree of any given node when it was deleted is recorded. The graphed data for this example is shown below:

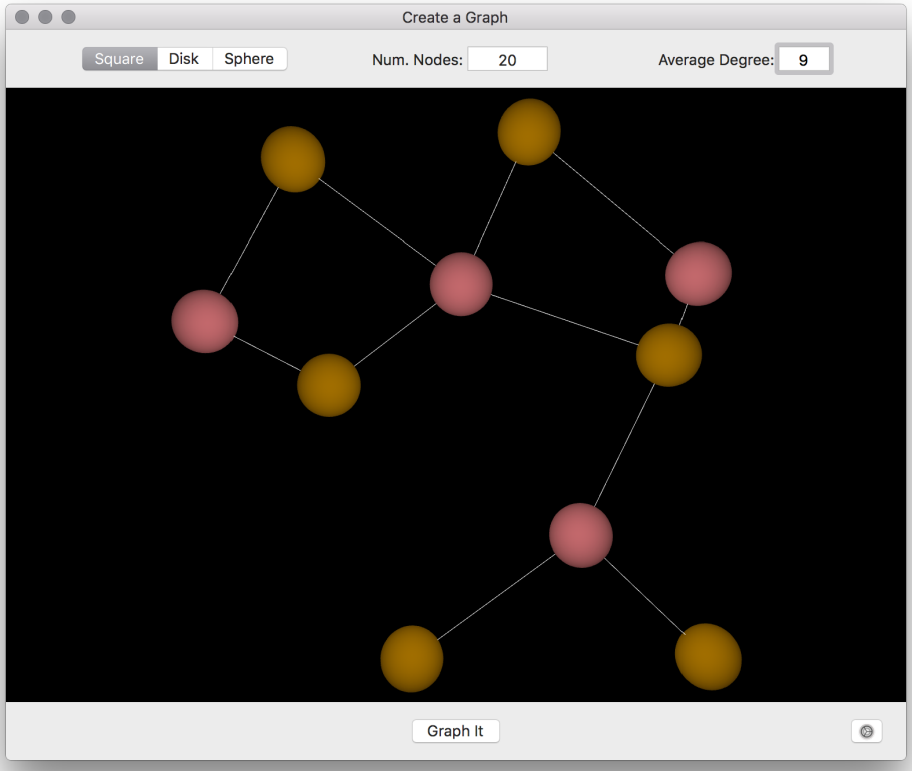


Using the smallest-last ordering, the coloring algorithm is applied to the graph by making a pass through each vertex in the smallest-last ordering and applying the smallest available color. After executing this process, the backbone selection algorithm can begin.

In order to pick the optimal backbone for this graph, the largest four colors must first be identified. After identifying them, a total of 6 combinations can be made to generate bipartite subgraphs. Each of these subgraphs must be analyzed by testing the number of edges between the two sets. After performing this analysis, the top two bipartite subgraphs are chosen as the backbones. The backbones for this example are shown in the figures on the following page.



Walkthrough Figure: Bipartite Subgraphs for the given RGG, which create the backbone of the RGG



3. Benchmark Results

3.1. Random Geometric Graph Generation and Display

Benchmark	N	Expected Avg. Degree	Distribution	Distance Bound (R)	Actual Avg. Degree	Avg, Degree Variance
1	1,000	30	Square	0.0993	28.696	-4.347%
2	4,000	40	Square	0.0571	38.917	-2.708%
3	4,000	60	Square	0.0697	57.496	-4.173%
4	16,000	60	Square	0.0348	59.317	-1.138%
5	64,000	60	Square	0.0174	60.110	0.183%
6	4,000	60	Disk	0.0617	57.380	-4.367%
7	4,000	120	Disk	0.0870	112.377	-6.353%
8	4,000	60	Sphere	0.2449	60.208	0.347%
9	16,000	120	Sphere	0.1732	119.950	-0.042%
10	64,000	120	Sphere	0.0245	120.040	0.034%
11	64,000	120	Square	0.0245	118.462	-1.281%
12	100,000	30	Sphere	0.0099	29.987	-0.044%

3.2. Graph Coloring and Bipartite Backbone Selection

Benchmark	N	Expected Avg. Degree	Distribution	Distance Bound (R)	Actual Avg. Degree	Avg, Degree Variance	Num. Edges (M)
1	1,000	30	Square	0.0993	28.696	-4.347%	14,348
2	4,000	40	Square	0.0571	38.917	-2.708%	77,834
3	4,000	60	Square	0.0697	57.496	-4.173%	114,992
4	16,000	60	Square	0.0348	59.317	-1.138%	474,538
5	64,000	60	Square	0.0174	60.110	0.183%	1,923,520
6	4,000	60	Disk	0.0617	57.380	-4.367%	114,759
7	4,000	120	Disk	0.0870	112.377	-6.353%	224,753
8	4,000	60	Sphere	0.2449	60.208	0.347%	120,416
9	16,000	120	Sphere	0.1732	119.950	-0.042%	959,601
10	64,000	120	Sphere	0.0245	120.040	0.034%	3,841,289
11	64,000	120	Square	0.0245	118.462	-1.281%	3,790,798
12	100,000	30	Sphere	0.0099	29.987	-0.044%	1,499,346

Benchmark	Min. Degree	Max. Degree	Max Min-Degree	Num. Edges (M)	Num. Edges in Largest Bipartite Subgraph	% of Verties Covered	Num. Colors	Terminal Clique Size
1	6	48	20	14,348	184	1.3%	22	22
2	13	65	27	77,834	576	0.7%	28	28
3	11	82	33	114,992	439	0.4%	34	33
4	13	95	36	474,538	1,712	0.4%	37	35
5	14	96	39	1,923,520	6,795	0.4%	40	40
6	19	83	33	114,759	426	0.4%	34	31
7	38	152	54	224,753	242	0.1%	58	50
8	36	88	32	120,416	418	0.3%	34	34
9	64	158	60	959,601	989	0.1%	61	52
10	64	165	61	3,841,289	3,931	0.1%	61	55
11	29	165	62	3,790,798	3,913	0.1%	63	57
12	9	55	26	1,499,346	17,331	1.2%	26	23

4. Appendix

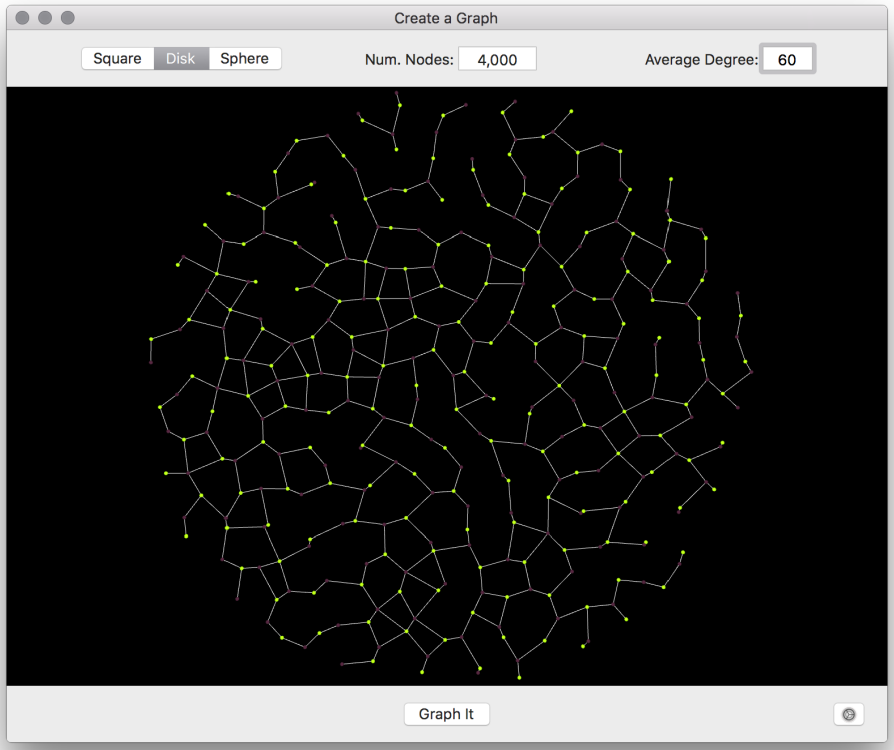
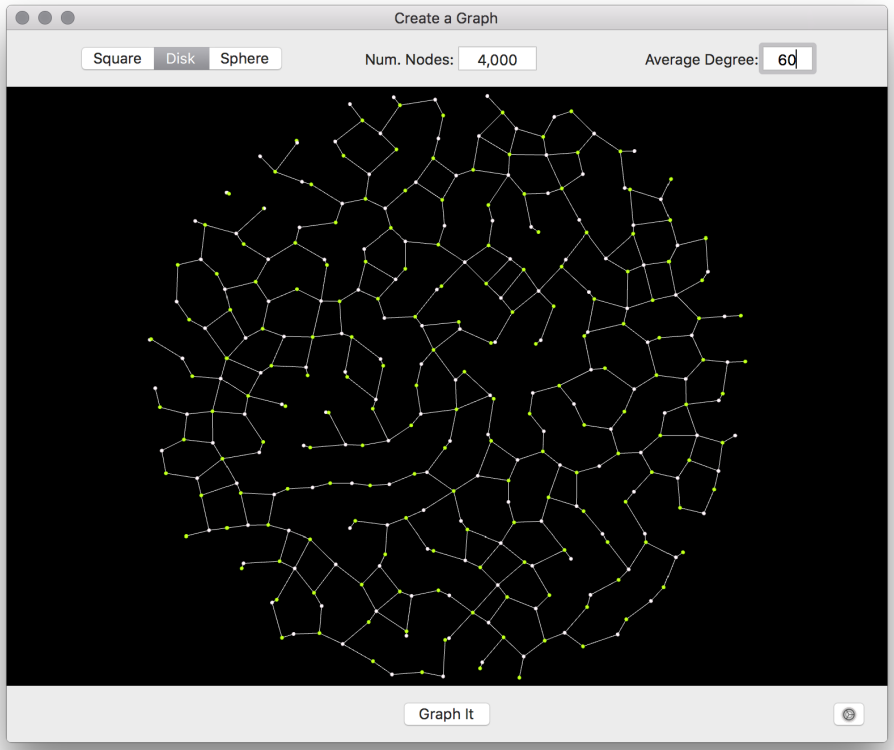


Fig. 1a & Fig. 1b – Bipartite Subgraph, Circle, N: 4,000, Deg.: 60



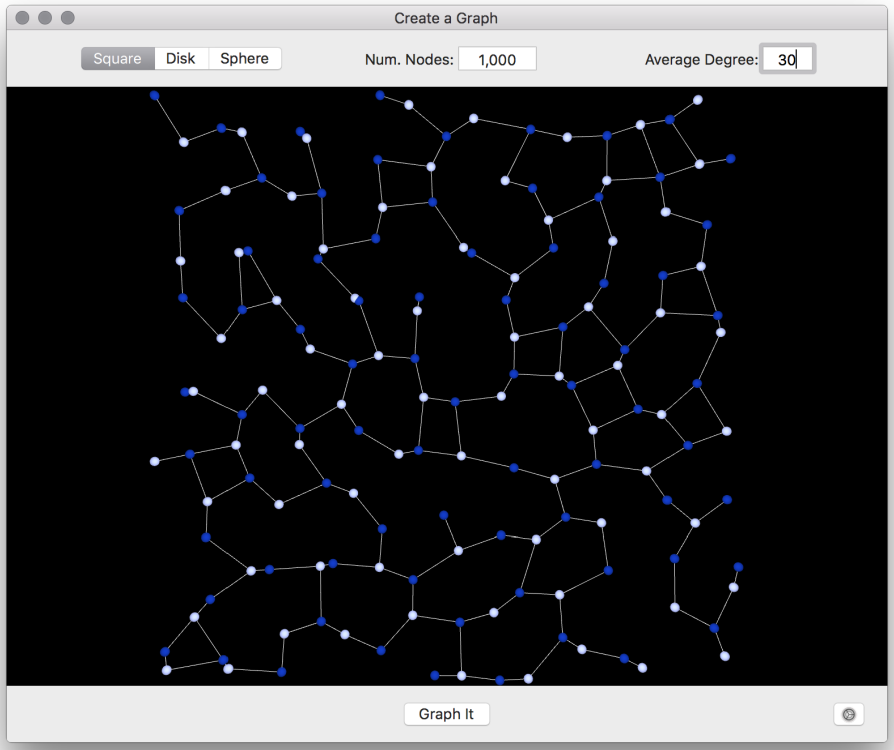
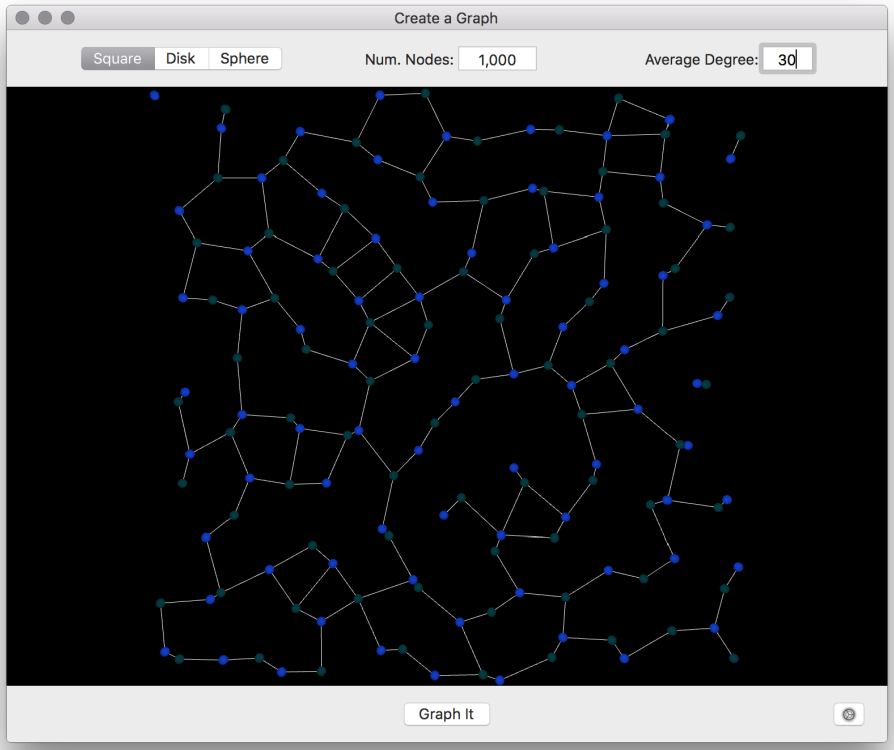


Fig. 2a & Fig. 2b – Bipartite Subgraph, Square, N: 1,000, Deg.: 30



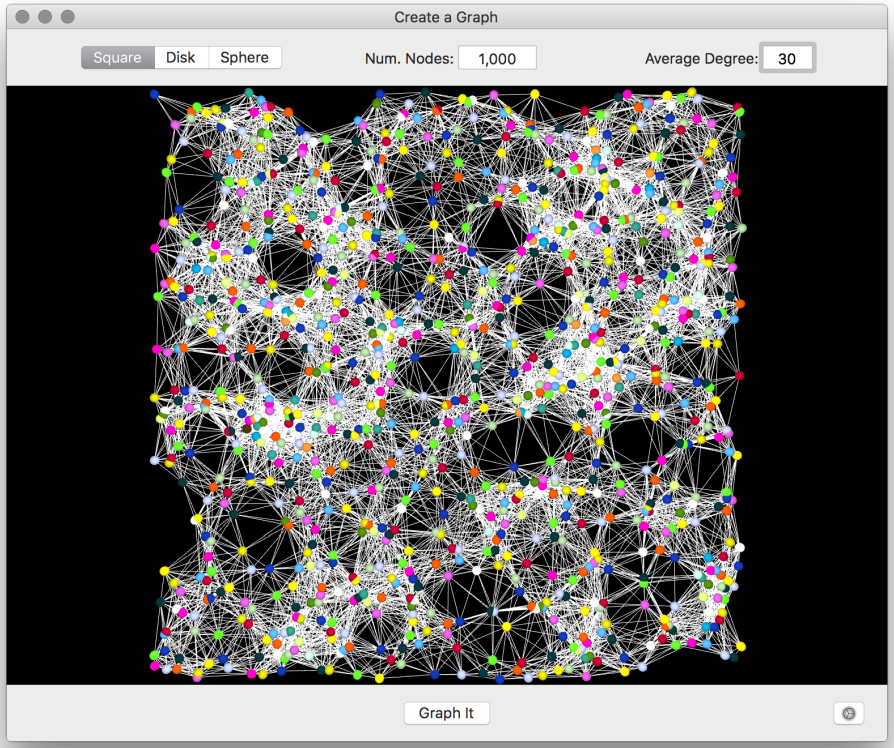
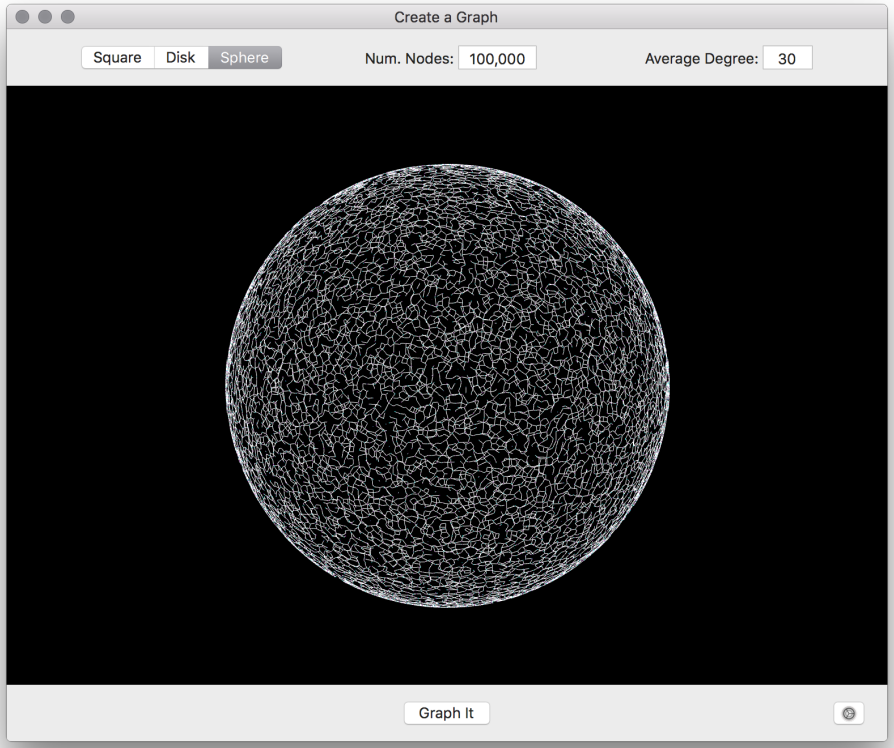


Fig. 3 – Colored RGG, Square, N: 1,000, Deg.: 30

Fig. 4 – Bipartite Subgraph, Sphere, N: 100,000, Deg.: 30



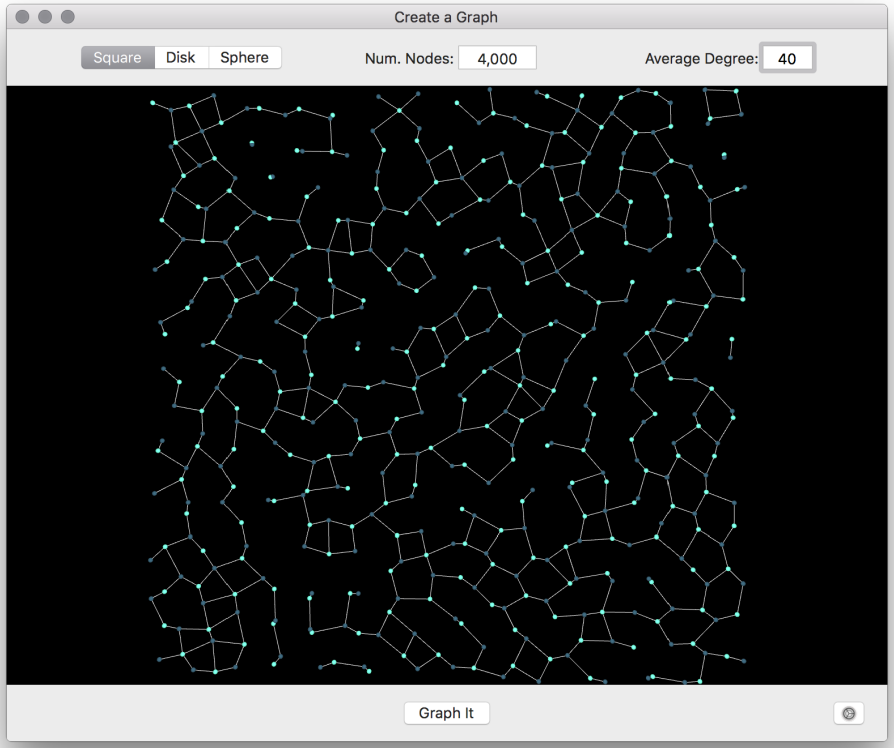
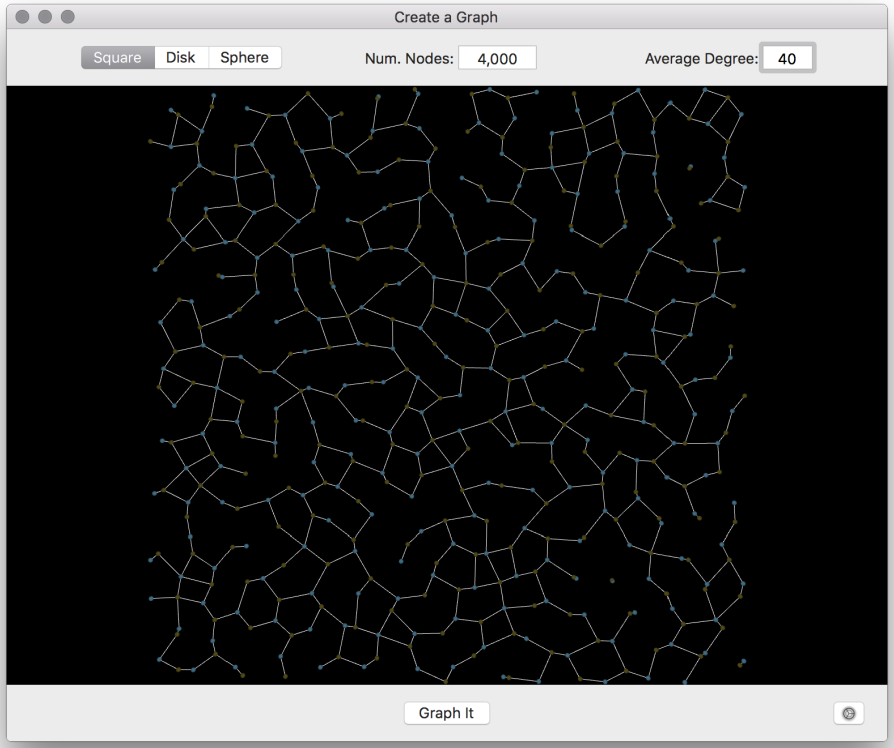


Fig. 5a & Fig. 5b – Bipartite Subgraph, Square, N: 4,000, Deg.: 40



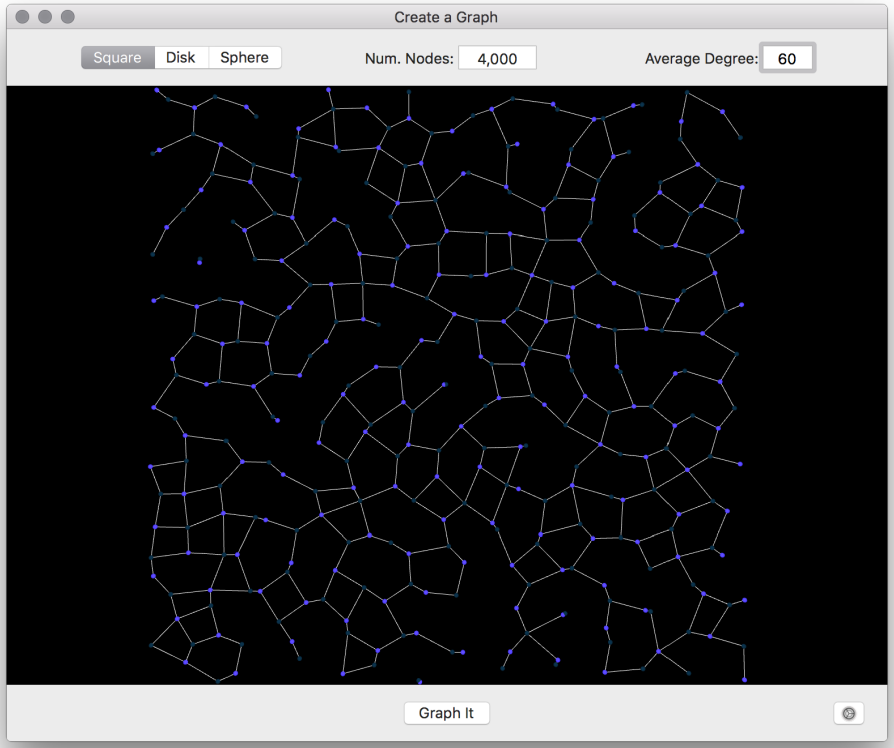
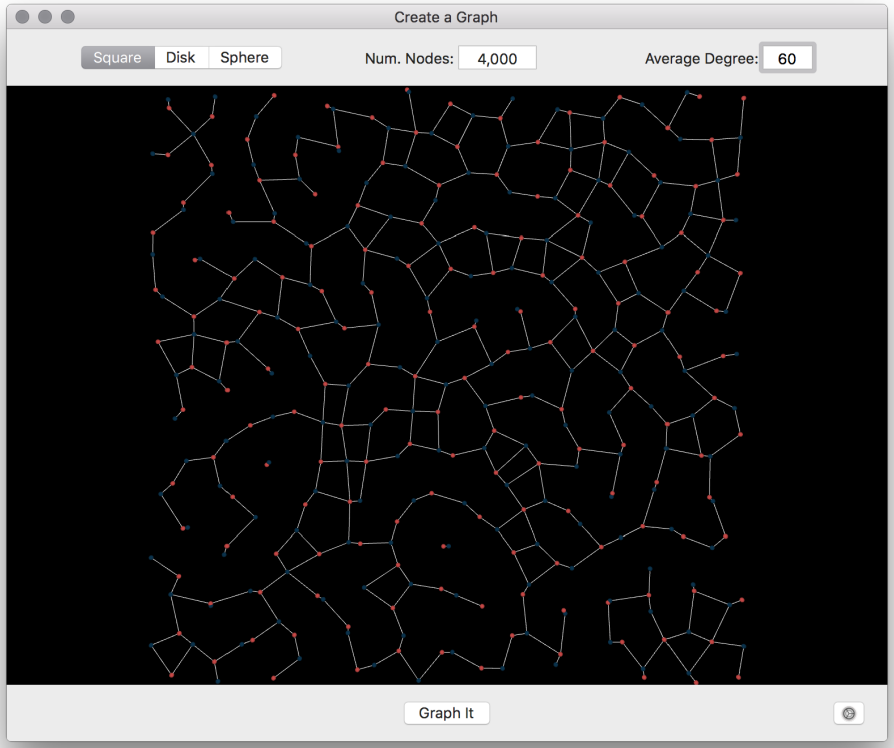


Fig. 6a & Fig. 6b – Bipartite Subgraph, Square, N: 4,000, Deg.: 60



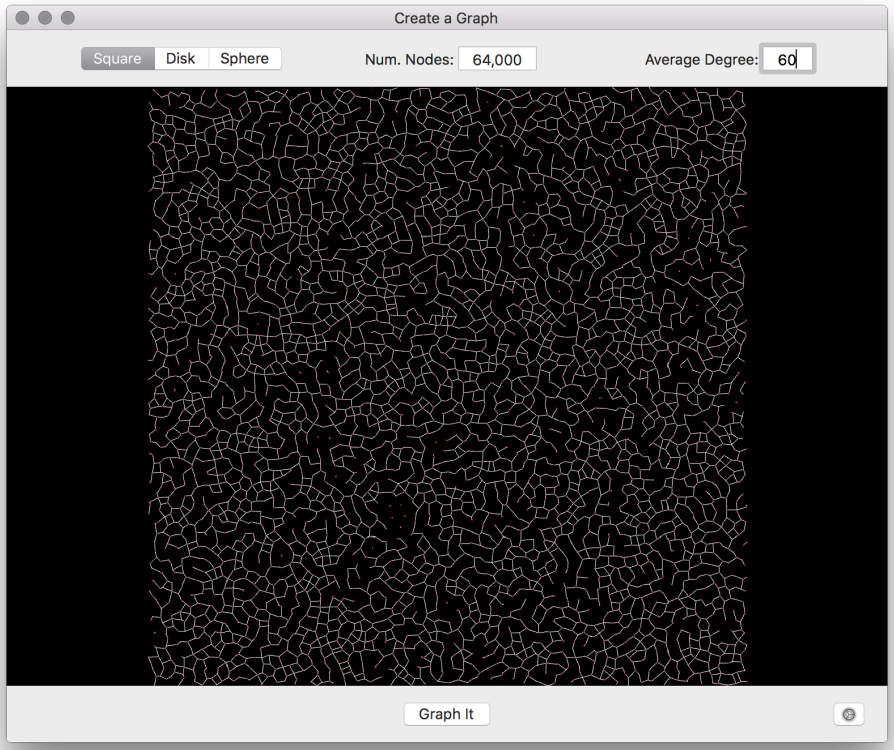
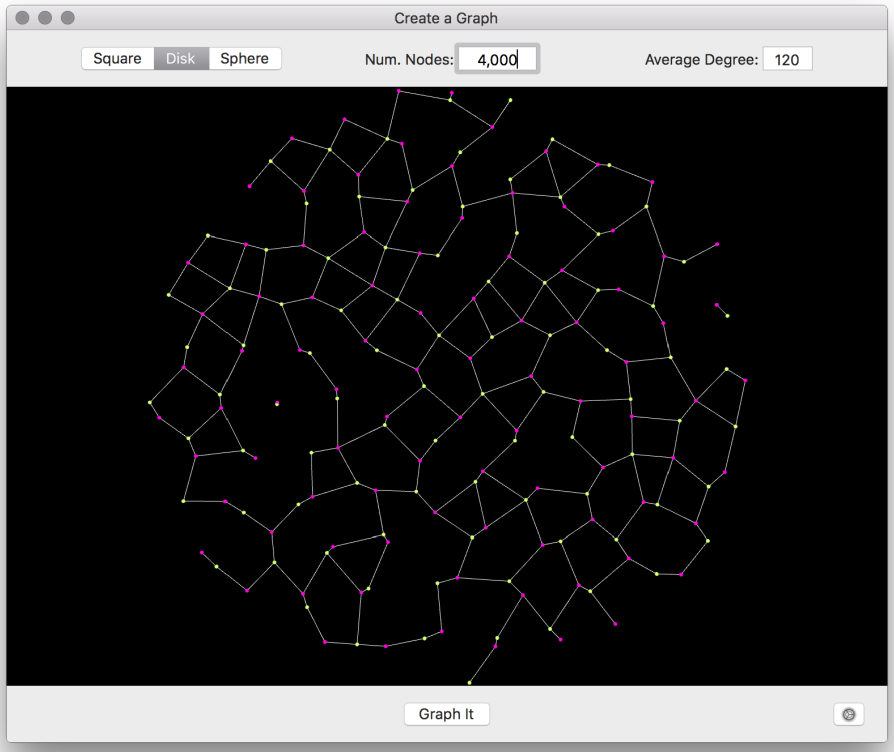


Fig. 7 – Bipartite Subgraph, Square, N: 64,000, Deg.: 60

Fig. 8 – Bipartite Subgraph, Square, N: 4,000, Deg.: 120



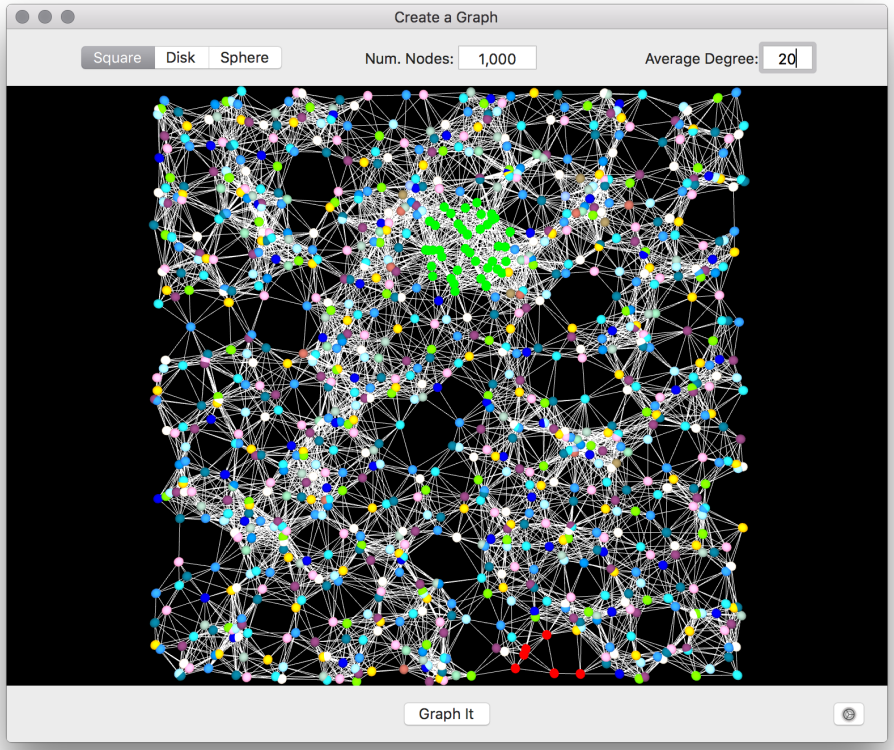
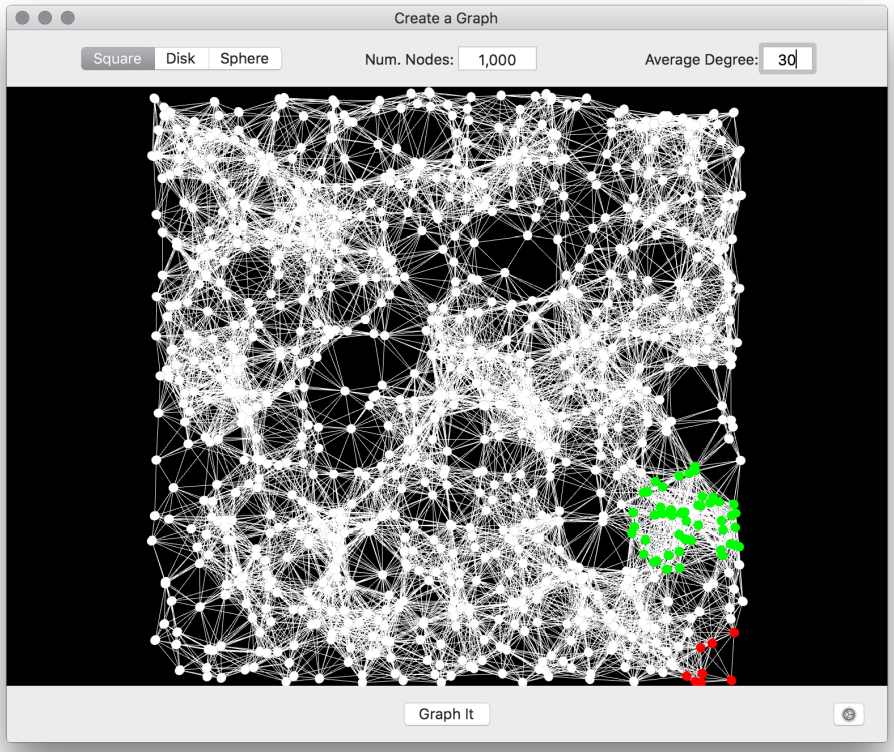


Fig. 9a & Fig. 9b – RGGs with highlighted Max and Min Degree nodes



5. References

[1] Matula, David W., *Smallest-last ordering and clustering and graph coloring algorithms*, http://dl.acm.org/citation.cfm?id=322385, 1983

[2] Hong, K., *Uniform Distribution of Points on the Surface of a Sphere*, http://www.bogotobogo.com/Algorithms/uniform_distribution_sphere.php, 2015

[3] Anonymous, *SceneKit on OS X with thousands of objects*, http://www.bogotobogo.com/Algorithms/uniform_distribution_sphere.php, 2013

[4] Anonymous, *Drawing a line between two points using SceneKit*, http://stackoverflow.com/questions/21886224/drawing-a-line-between-two-points-using-scenekit/21891749#21891749, 2014

[5] Kaiser, Spencer, *SceneKit crashes without verbose crash info*, http://stackoverflow.com/questions/34257251/scenekit-crashes-without-verbose-crash-info/34259854?noredirect=1#comment56263543_34259854, 2015

[6] Ayala, Chris, *Wireless Sensor Networks Project*, Southern Methodist University, 2014

[7] Uppalapati, Chandana Datta , *Wireless Sensor Networks*, Southern Methodist University, 2014

[8] Apple Inc, *About Objective-C*, https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, 2014

[9] Apple Inc, *SceneKit Framework Reference*, https://developer.apple.com/library/ios/documentation/SceneKit/Reference/SceneKit_Framework/

[10] John A. Stankovic, Anthony D. Wood, Tian He, *Realistic Applications for Wireless Sensor Networks*, http://www-users.cs.umn.edu/~tianhe/Research/Publications/Papers/Jack-RealisticApplication.pdf

[11] Apple Inc., *arc4random_uniform Man Page*, https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/arc4random_uniform.3.html, 1997