Backbones in Wireless Sensor Networks

An approach to generating communication backbones in wireless sensor networks using the Smallest Last Order.

M. Tyler Springer
CSE 7350 Algorithm Engineering – Semester Project
December 15, 2015

Executive Summary

Introduction and Summary

Wireless Sensor networks have become a hot topic in research communities in recent years as improvements in technology and reductions in manufacturing costs have made their creation and deployment increasingly economically feasible. Technologies such as Wi-Fi, ZigBee and Bluetooth are all commonly used protocols in wireless sensor networks and have been subject to numerous implementation and hardware improvements over their lifetime resulting in sensors that are smaller, cheaper and more energy efficient. This coupled with their unlimited versatility and lack of physical connections have made wireless sensors an increasingly viable choice for interconnected ad-hoc networks [1].

This paper details a project that models ad-hoc networks of wireless sensors of varying scale across a variety of geographic areas. The project models these networks by creating random geometric graphs (RGG) of varying size, degree and type and interconnecting vertices if they are within a distance R of one another [2]. In the context of wireless sensors R is the maximum broadcast range of any single sensor and thus determines how far away any two sensors must be to communicate. Determining connections between vertices in the graph is not a trivial task and can be a very expensive process if great care is not taken when doing so. This project makes use of several methods for decreasing the time and number of comparisons required to determine vertex adjacency which will be discussed in detail in a later section. We apply the Smallest Last Ordering algorithm [3] and a graph coloring procedure [4][10] to identify high quality candidates for communication backbones in the graph. A backbone is a bipartite subgraph of the overall RGG, or sensor network, that sufficiently covers all vertices that make up the overall network. Ideally we would like to find backbones that allow any sensor in the network to communicate with a sensor in the backbone in no more than one hop (100% coverage). The backbones must be bipartite because this ensures that no two connected vertices share the same "color" which practically translates to sensor broadcast frequency. With no overlapping frequencies in the backbone we can ensure that interference is not an issue [11].

Finding high quality candidates for backbones is by no means a trivial task and brute force techniques would quickly prove to be impractical in graphs or networks of sufficient scale. This is particularly true given that backbones need to be generated extremely quickly as in practice nodes may enter or exit the network at any time due to power failure, movement of the sensor or an unlimited number of other reasons. Also wireless sensors often lack significant computational power in an effort to make them more energy efficient or reduce their size so backbone determination must be made as simple as possible. Finally, manual data collection on sufficiently large scale networks is impractical so determining backbones that cover as many sensors as possible is of utmost importance. Consider a network with 100,000 nodes and a backbone that has 95% coverage. While data from 95% of the network can be pulled from a central location, 5,000 nodes are not covered by the backbone manual collection or formation of additional backbones. For these reasons it is extremely important that high quality backbone generation is a scalable process.

Results

This section presents the results of successfully using a combination of the smallest last ordering algorithm, a graph coloring algorithm, and component search to generate communication backbones on wireless sensor networks that are modelled by random geometric graphs. The entire process is done for 2-dimensional graphs on the unit square and unit disk as well as 3-dimensional graphs on the unit sphere. In all cases smallest-last ordering is accomplished in O(|V|+|E|) time which will be demonstrated in the walkthrough. The graph coloring algorithm used, sometimes referred to as the "Grundy coloring algorithm," also runs in O(|V|+|E|) which is quite fast [10]. The component search and backbone selection algorithm is based on breadth-first search and runs in O(|V|*|E|) time due to instances where every node is of extremely low degree. In practice however this algorithm generally runs substantially closer to O(|E|) time. Each step in the overall simulation is quite efficient which allows us to run simulations of very large size in a reasonable amount of time. In order to see that these claims are indeed true we have generated 10 benchmark graphs of varying type, size and average degree that we will apply our entire simulation process to. The abbreviated results of these benchmarks can be seen in the following table:

Graph ID	1	2	3	4	5	6	7	8	9	10
N	1,000	4,000	4,000	16,000	64,000	4,000	4,000	4,000	16,000	64,000
Desired Avg Degree	30	40	60	60	60	60	120	60	120	120
R	0.099	0.057	0.070	0.035	0.017	0.070	0.098	0.070	0.049	0.025
Type	Square	Square	Square	Square	Square	Disk	Disk	Sphere	Sphere	Sphere
Num Edges	14,069	77,969	114,817	473,825	1,921,111	115,513	223,118	121,741	967,081	3,868,443
Min Degree	5	7	17	14	14	18	49	35	86	85
Avg Degree	28.138	38.984	57.408	59.228	60.035	57.756	111.559	60.87	120.885	120.889
Max Degree	48	63	89	88	96	84	163	89	168	168
Num Colors	19	30	35	37	38	39	63	35	62	62
Backbone Coverage	99.60%	99.75%	99.98%	99.81%	99.89%	99.95%	100%	99.95%	99.99%	99.99%
Execution Time	0.98	4.13	4.91	20.27	78.14	5.81	7.29	6.32	27.44	106.71

The strongest features of this implementation lie in its numerous computational and end user experience optimizations. For example, this implementation uses the "cell method" for 2D graphs and the "block method" for 3D graphs to reduce the number of comparisons that need to be made between pairs of points to determine adjacency. Instead of exhaustively testing all pairs, these methods partition the points into cells or blocks of size r and test only the cells or blocks where a given point could have a mathematical possibility of having a connection. This reduces our comparison bound from $O(n^2)$ to the more favorable $O(n^2r^2)$ as r is always less than 1 or in the case of the block method to $O(n^2r^3)$. Another performance optimization is that graphs are visualized using a compressed adjacency list which prevent duplicate edges from being stored thus reducing the number of edges to be drawn by a factor of 2 - meaning that we can successfully visualize larger graphs with less lag. Finally, my implementation generates points on the edge of the sphere using a method described by Archimedes instead of the rejection

method described in class [8]. My tests between both methods indicate that the Archimedian method performs on average about 4x as fast as the rejection method because there are no probabilistic elements to this process and it is based entirely on constant time operations. One notable end user experience optimization that has been made is how vertex colors are generated for visualizations. Instead of prepopulating a list of colors chosen by hand or randomly generating colors, my implementation uses the golden ratio to procedurally generate colors that are as distant from one another on the color spectrum as mathematically possible [9]. One advantage of this approach is that colors are more distinct, making them easier to tell apart from one another on the final visualization of the graph.

Programming Environment

Hardware Description –
Below is a list of pertinent information about the hardware that I used to run my simulation:

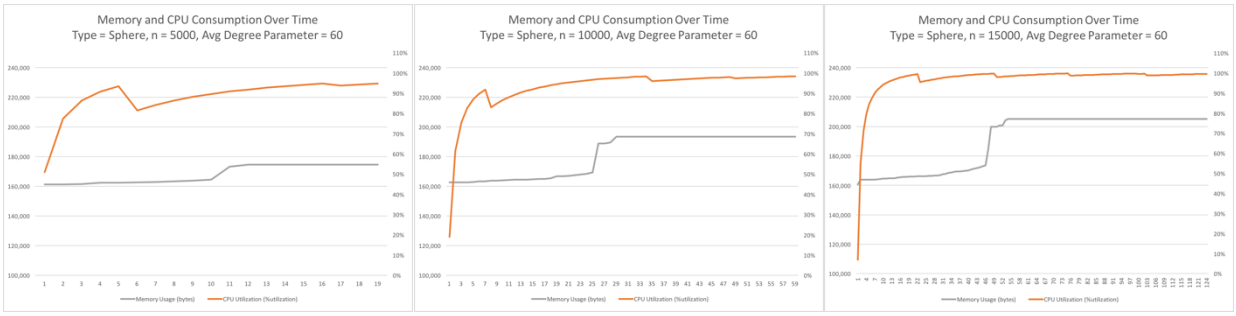
- Computer: Apple MacBook Pro (Mid 2010)
- Processor: Dual Core Intel i7 @ 2.66GHz with Turbo Boost up to 3.3GHz
- Memory: 8GB DDR3 RAM @ 1067MHz
- Graphics Card: NVIDIA GeForce GT 330M with 512MB VRAM
- Secondary Storage: 500GB SATA HDD
- Operating System: Mac OSX 10.10.3 (Yosemite)
- CPython version 2.7.9
- Cython compiler version 0.23.4
- Clang compiler version 6.0 based on LLVM 3.5, build clang-600.0.54

Software Description –
The entire simulation is implemented in Python 2.7. I chose to use python because of its flexibility when dealing with complex data types and structures and the fact that it is a garbage collected language eliminating many memory issues that can occur without careful attention to detail in languages like C++. Python is easy and flexible making it a great choice for any project where extremely high performance computation is not necessary. When dealing with graphs under 4,000 vertices my simulation has execution times comparable to a C based solution, but as the number of vertices increases, the performance rapidly degrades. This is almost entire due to the fact that Python runs on a virtual machine and “Python Objects” are not statically typed so some computation must be done to determine the exact data type. This means that my python based simulation was not scalable and was therefore unacceptable as a final solution for this project. In order to dramatically improve execution time, I ran my Python code through a program called “Cython” which converts the python directly to C code. C of course runs directly on the hardware and generally resulted in execution times that were about 20% of the execution time to run the same simulation directly in Python. Cython is a robust and trusted tool, in fact the extremely popular and well respected “NumPy” library was written entirely using Cython [7].

I chose to use the Processing3 graphics package to visualize the graphs that were output by my simulation. Processing3 is an amazing tool that is based in Java and runs on the JVM that can be used to draw 2D or 3D images and figures of just about anything you want [6]. It provides very easy to use methods for drawing points and lines allowing me to get beautiful visualizations of my graphs in a very small amount of time. Because processing runs on the JVM, it is slower than

some comparable libraries written in C++ (openFramework or QT to name two) but is extremely optimized allowing for very speedy drawing times, even with graphs with many thousand vertices. Processing3, the most recent version of Processing, has added support for a Python "mode" which I mistakenly thought would be a good idea to use [6]. It turns out that this Python code is passed through another tool called Jython that converts and runs python code on the JVM. This is great because you can write python code, making the drawing process near effortless, *but* you cannot use any of the standard python libraries because at the end of the day you are running Java, not python. So in quite an interesting turn, my simulation and visualization code is written entirely in Python, but not a single line is actually executed in the Python environment – though this ended up being a good thing because both conversions resulted in running times not achievable using Python as is. It should be noted that because I use two different programs for my simulation: one to generate the data and another to visualize it, I am using CSV files to pass data from the generation program to the visualization program. This also makes it easy to plot figures such as original degree vs degree when deleted with a high degree of ease.

The simulation code was written to be as memory efficient as possible so that large graphs can be generated without placing excessive load on the machine or worrying about using a significant amount of swap memory which would absolutely tank performance. Below are 3 plots showing the memory and CPU utilization of some sample simulations of size 5,000, 10,000 and 15,000 vertices.



From these graphs we can see that size of the executing portion of the code begins at about 161KB regardless of number of vertices generated which means our program has a very small footprint. Generation of a 5,000 vertex simulation required about 175KB of memory and 2.6MB of disk to complete execution. Generation of a 10,000 vertex simulation required about 193KB of memory to complete execution. Generation of a 15,000 vertex simulation required about 205KB of memory to complete execution. We can see that the amount of memory needed does not grow dramatically with the problem size ensuring that we can safely generate graphs of extremely large size. The first ascending leg of these simulations corresponds to graph generation, the spike occurs during the smallest last and backbone selection process as numerous new pieces of information are generated. The final stable leg of the process corresponds to when the data is being written out to CSV's as no new memory is required. Microsoft Excel was used to plot anything not done in Processing3.

References

[1] M. Hannikainen, "Ultra-Low Energy Wireless Sensor Networks in Practice," Tampere University of Technology, Tampere, Finland, May 2014.

[2] K. Hichem and R. Vlady, "Random geometric graphs as model of wireless sensor networks," vol. 4, IEEE, 2010, pp. 103,107,26–28. [Online]. Available: http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=5451758&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D5451758. Accessed on: Dec. 15, 2015.

[3] D. W. Matula and L. L. Beck, "Smallest-last ordering and clustering and graph coloring algorithms," in *Journal of the ACM (JACM)*, vol. 30, ACM, 1983. [Online]. Available: http://dl.acm.org/citation.cfm?id=322385. Accessed on: Dec. 15, 2015.

[4] R. van Stee, "Vertex Coloring," Max Planck Institut Informatik, Saarbrucken, Germany. [Online]. Available: http://algo2.iti.kit.edu/vanstee/courses/vcolor.pdf.

[5] "Euler's formula," in *Math Is Fun*, 2014. [Online]. Available: https://www.mathsisfun.com/geometry/eulers-formula.html. Accessed on: Dec. 15, 2015.

[6] J. Gilles, A. Parrish, and M. Peyton, "Python mode for processing," Processing. [Online]. Available: http://py.processing.org/. Accessed on: Dec. 15, 2015.

[7] S. Behnel, "Using the Cython compiler to write fast python code," Feb. 6, 2002. [Online]. Available: http://www.behnel.de/cython200910/talk.html. Accessed on: Dec. 15, 2015.

[8] K. Hong, "Uniform distribution of points on the surface of a sphere - 2015," 2015. [Online]. Available: http://www.bogotobogo.com/Algorithms/uniform_distribution_sphere.php. Accessed on: Dec. 15, 2015.

[9] M. Ankerl, "How to generate random colors Programmatically," 2009. [Online]. Available: http://martin.ankerl.com/2009/12/09/how-to-create-random-colors-programmatically/. Accessed on: Dec. 15, 2015.

[10] C. Ayala, "A method of determining the bipartite backbones in a Wireless Sensor Network using the Smallest Last Ordering Algorithm," Dec. 14, 2013

[11] R. Asgarnezhad and J. Torkestani, "A survey on backbone formation algorithms for Wireless Sensor Networks: (A New Classification)," in *Australasian Telecommunication Networks and Applications Conference (ATNAC)*, Nov. 2011.

Verification Walkthrough

Now that we have a thorough understanding of how each algorithm for smallest last ordering, graph coloring and backbone selection works, let's examine the entire simulation on a graph. The following walkthrough is run on a random geometric graph drawn on a unit square with $n = 20$ vertices and $R = .40$ maximum connection distance. This graph has 136 distinct pair-wise edges, a minimum degree of 2, a maximum degree of 11 and average degree of 6.8.

We begin with smallest last ordering as it forms the basis for all subsequent operations performed in the simulation. The first figure in the series, labeled with a white "1" in the upper left corner is the initial state of the graph before any deletions have been performed. The vertex with the minimum current degree is outlined with a red circle and is the next vertex to be deleted in all figures in the series. The vertex outlined in figure 1 is the overall minimum degree vertex in the graph and will be the first vertex that is deleted, in this case that degree is 2. Because this vertex is the first to be deleted, it will be the *last* vertex in the final smallest last ordering.

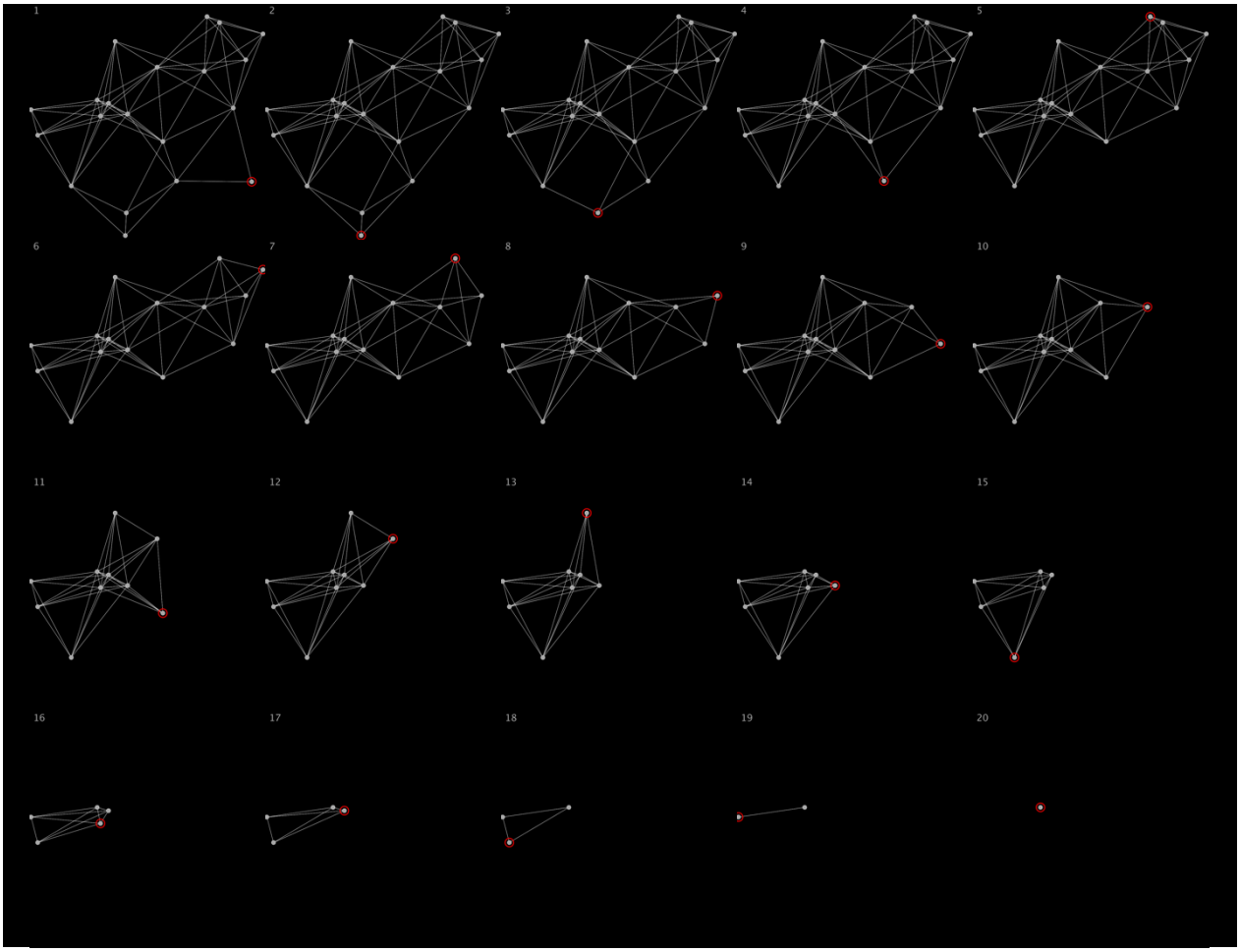
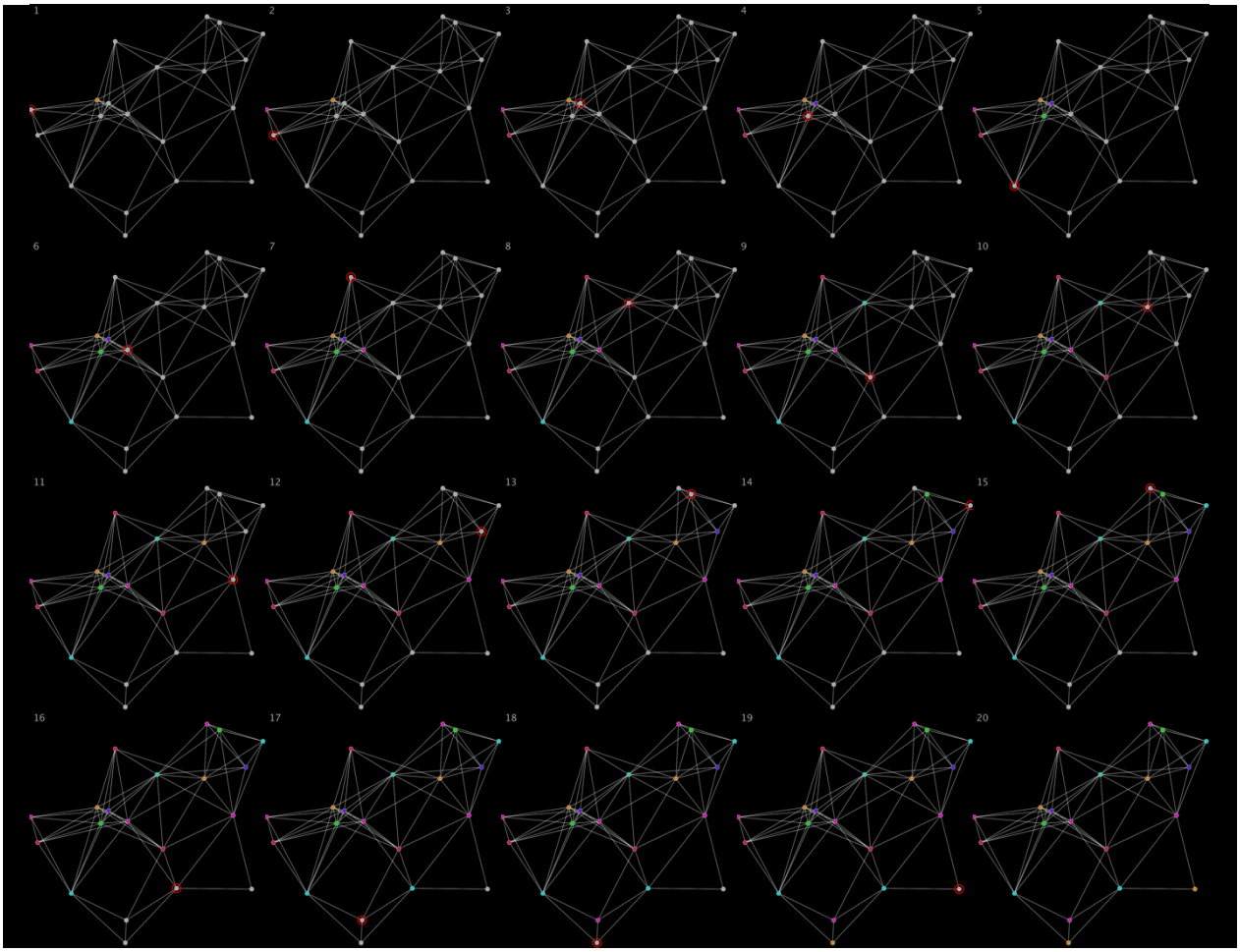


Figure 15 should be noted as the terminal clique, which is found when each remaining vertex has the same degree. From this point on each remaining vertex will continue to have consistent degrees after each subsequent deletion. This behavior can be clearly seen in the degree plots

provided for the benchmarks as the “final plunge” where every vertex approaches degree 0 simultaneously. Figure 20 shows the final vertex to be deleted. This vertex will be the first in the resulting smallest last ordering. By examining the series of figures above we can see that vertices of minimum degree are being deleted correctly in sequence until no vertices remain and that our resulting smallest last ordering is correct and can be found in O(|V|+|E|) time.

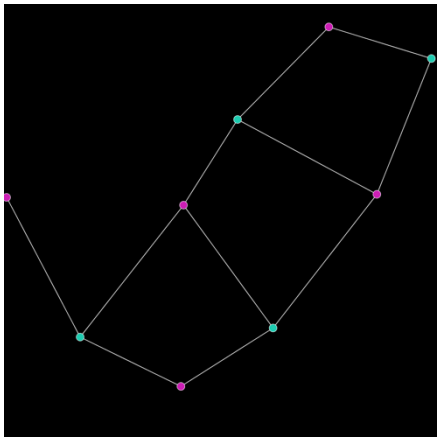
The next step in the simulation is to color the graph by making use of the smallest last ordering determined in the process above. In the following figures, the next vertex to be colored (the next vertex in the smallest last ordering) is outlined with a red circle. Note that the first vertex to be colored is the last node we deleted in the process above, and that each following vertex will be colored in the reverse order that we deleted them. Each vertex is assigned the “lowest” available color that is not already assigned to one of its neighbors in traditional graph coloring fashion.

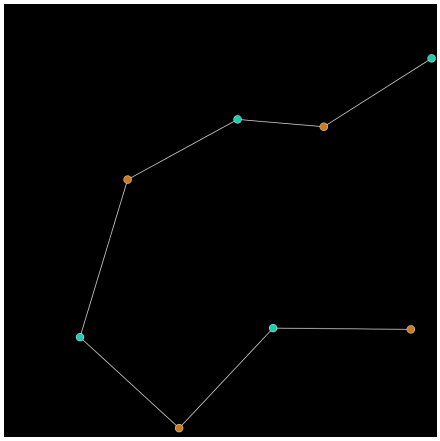


We see that the graph can be completely colored using 6 distinct colors.

The final step in the simulation is to generate some suitable backbones from the coloring determined in the process above. A backbone is a connected bipartite subgraph consisting of no more than 2 colors (hence bipartite) that will be the main trunk for communication in the wireless sensor network. This project specifically asks for the two best backbones that can be

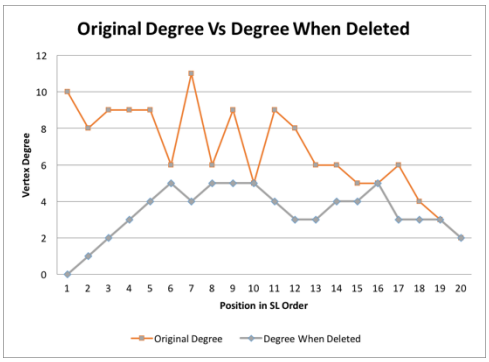
generated from the largest 4 color classes. In this case colors orange, turquoise and hot pink all have 4 vertices in their classes and color salmon red has 3 vertices in its color class making them the 4 largest color classes in the graph. These 4 colors will each be combined and all 6 possible combinations of colored vertices will be examined to determine the best two backbones with the highest domination percentage. Each pairing of colors is subjected to a component search such that only the largest connected component is submitted for comparison against other pairing (its not a backbone if its not connected!). The two pairings that result in the highest vertex coverage are chosen as the best backbones.





Backbone 1 (left) contains 9 vertices, 11 edges and has 100% vertex coverage. Backbone 2 (right) contains 8 vertices, 7 edges and has 100% vertex coverage. Coverage is determined by counting the number of vertices that have access to the backbone compared to the complete set of vertices in the graph. It is tempting to think that despite the fact that both backbones have 100% coverage, backbone 2 is superior when compared to backbone 1 because it uses a lower percentage of the overall vertices. In the context of wireless sensors this may be true when considering things like power consumption. However, it is important to remember that we are trying to find the best possible *communication* backbone. Backbone 1 has several cycles or "faces" meaning that there is more than 1 path to most nodes reducing communication bottlenecks so, in fact, it may be more desirable.

There are several interesting metrics that can in addition to the figures provided above, help us to better understand the features of our graph and if our algorithm is performing correctly. One such metric is original degree compared to degree when deleted.



This plot is in smallest last order, so the last vertex to be deleted is the first point plotted. We can see the "final" plunge to 0 which consists of 6 points, indicating the terminal clique. This is consistent with our figures displayed above and is a good indication that our algorithm is performing correctly. We can also see that degree when deleted is more stable than original degree and that there are no anomalies or random spikes which is another good indication that our simulation is performing as expected.

Reduction To Practice

Data Structures

I chose to implement this project in Python which, unlike many other languages, limits its users to only a handful of "core" data structures. These include lists, dictionaries, tuples and sets. All other data structures are generated as a composition of these containers. Most python containers are really just a hash map at their core, meaning that access time is near constant for almost all operations in Python with the obvious exception of iterating over a collection. With this in mind the first data structure we will consider is that for a single point in any of the RGGs. Each point is represented as a triple for 2D graphs or a 4-tuple for 3D graphs. In python some sample points might look like the following:

```
2D_point = (pointID,x,y)
3D_point = (pointID,x,y,z)
```

In the actual implementation a list of *N* points is generated and is then passed to a method that will determine the adjacency of the set of points. PointIDs are generated sequentially starting at 0 so that fast lookup of point coordinates can be done at any time by simple indexing into the list with whatever vertex you are interested in. Point adjacency information is stored in an auxiliary data structure to simplify output of CSVs and allow us to avoid passing around a single extremely large data structure that contains every piece of information required in the project when only a small subset of that information is required to completed the next step in the simulation (remember that Python does not provide the user with direct access to pointers!). Adjacency information is stored in a dictionary (equivalent to a hash map in most languages) where the key is the vertex in question and the value is a list containing the vertex in question's neighbors. A vertex is considered a neighbor of another vertex if it is within some distance R of the vertex being considered. The dictionary structure allows near constant time lookup of any vertex's neighbors which is key in ensuring that our simulation runs as quickly as possible. Additionally, adjacency lists of this type avoid storing irrelevant information such as which vertices are *not* neighbors of the vertex in question as would happen when using an adjacency matrix. Therefore, this representation of adjacency information is favorable because it provides extremely fast lookup and avoids using any unnecessary memory. An example adjacency list for the points 0,1,2,3 could be declared in Python as follows:

```
adjList = {
    0:[1,2],
    1:[0,3],
    2:[0,3],
    3:[1,2]
}
```

Of course in practice the adjacency list is determined programmatically by evaluating each point and all points within distance R of it. It should be noted that Python will allow us to see if any vertex is a member of the neighbors list in constant time because under the hood it is a hash set. For this reason, neighbors need not be in any particular order. Coloring information is stored in yet another dictionary (again for modularity) where the key is the vertex in question and the

value is an integer corresponding to the color assigned to that node. So given a pointID it is possible to look up the coordinates of a point, the color of a point and the adjacent vertices of point in constant time. This makes processing all vertices in the graph a very efficient and speedy operation. Any remaining data structures used in the project will be described as individual algorithms are introduced.

Point Generation and Connection Determination

The first step in the program is to generate the points that will make up the RGG. The x, y and z coordinates for each point are generated using simple uniform real distribution random number generators which can be found in the Python standard library. These generators produce uniformly distributed pseudorandom real numbers in the range [0.0,1.0]. In the case of the 2D graphs on the unit square, randomly generating coordinates in the specified range is all that is required as it is impossible for a point to have invalid coordinates. However, graphs on the unit disk and unit sphere require some transformation to ensure that the points generated are indeed valid. For the disk, one possible solution is to randomly generate pairs as with the unit square and reject them if they have a distance greater than the radius away from the origin. However this is a probabilistic method that does not guarantee termination (although it almost certainly will) and could affect how random the points actually are on the disk, so I decided to instead use a method that will always produce a valid point. This involves picking random x,y coordinates as before and then transforming them using some simple trigonometric operations. A random point on the unit disk can be generated using the following logic:

```
a = random.uniform(0,1.0)
b = random.uniform(0,1.0)
if b < a:
      swap(a,b)
x = b * cos(2*pi*a/b)
y = b * sin(2*pi*a/b)
```

In the case of 3D graphs on the unit sphere we also have multiple options for point generation. One such method, was discussed at length in class and involved picking points that fall within the unit sphere and rejecting those that don't and then normalizing the distance vector to unit since the origin is at (0,0,0). This method does indeed generate uniformly distributed random points on the surface of a sphere but is again probabilistic in nature. Over the course of my research for this project I discovered a different method that was developed by Archimedes almost 2,000 years ago. His method is based on the relationship between a cylinder and a sphere in that "if a point is randomly distributed on the cylinder, its inverse axial projection will be uniformly distributed on the sphere" [8]. If we consider that a unit sphere has a radius of one we will quickly realize that the domain of the unit sphere inscribed in a cube is [-1.0,1.0]. So to generate random points about the surface of the unit sphere we simply generate points by first getting a value between -1.0 and 1.0 and then placing them on a cylinder by multiplying it by a value in the domain $[0,2\pi]$ and then projecting it about its inverse axis we will have a point that is uniformly distributed about the surface of the unit sphere. In my implementation this is done with following logic:

```
theta = 2*pi*random.uniform(0,1.0)
```

```
phi = acos(2*random.uniform(0,1.0)-1.0)
x = cos(theta)*sin(phi)
y = sin(theta)*sin(phi)
z = cos(phi)
```

This approach has the added benefit of being *substantially* faster than the rejection method discussed in class. I ran several tests using the Python "TimeIt" library which is specifically designed to provide extremely accurate timing data for functions. I ran the test generating a list of 10,000 points using both methods 10,000 times. On average Archimedes' method was about 3.8x faster than the rejection method. All point generation methods are $O(1)$ for a single point and $O(n)$ when generating a graph of size n.

Generating points is an extremely efficient operation is considered trivial. However determining connection between points is a more complex process. One solution is all-pairs testing where we check the Euclidean distance between each point and record adjacency if this distance in less than R. However this is ridiculous for graphs without an extremely large value for R as you will be comparing points that could not possibly be connected because they are extremely far away. This project makes use of the "cell method" to more intelligently choose which points should be compared. This is accomplished by segmenting the points into cells of size R x R. This cell is then compared to at most 5 other surrounding cells drastically reducing the number of comparisons needed to determine adjacency. In the actual implementation we partition the points into a cell map by first sorting them by their x coordinate. The points are then segmented into "sweeps" by simply checking if their x coordinate is less than a constant multiplied by R. We can determine the total number of sweeps needed by taking the ceiling of R/width of the square (which is unit in this case). Then we sort each sweep's coordinates by their y value and segment them again so now we have cells. One thing that is nice about this approach is that we can reuse the segmenting method for an arbitrary level of dimensions. The "block" method is also used on 3D graphs and simply segments that points an additional time by their z coordinate. Sorting is done using the Python standard sort which is extremely optimized, but still has a relatively expensive time complexity of $O(|V| \lg |V|)$. But by avoiding all pairs-testing which has a time complexity of $O(n^2)$ we reduce our time complexity to $O(n^2 r^2)$ where r is a value equal to or less than 1. In the case of the "block method" the time complexity is $O(n^2 r^3)$. However, in the theoretical worst case where every node is connected to every other node these methods still run in $O(n^2)$. In the average case this method is sufficiently efficient for large graphs.

Smallest Last Ordering

The Smallest Last Ordering is a way of sequencing the vertices in the graph such that vertex v_i has the minimum degree in the remaining subgraph of vertices $v_1, v_2, \ldots, v_k$ where $k \leq n$. By ordering the vertices in this fashion we can easily color all vertices in the RGG to determine sets to be used during the backbone selection process.

Finding the Smallest Last Ordering of a set of vertices is accomplished by repeatedly applying the following four steps until there are no vertices remaining in the graph:

1. Find the vertex with the minimum degree
2. Delete this vertex from the set in consideration

3. Update all previously adjacent vertices such that their degree is equal to degree-1
4. Record the deleted vertex in the last available spot in the smallest last ordering list

The smallest last ordering list is of size *n* because we must record the order of every vertex in the graph. The ordering list is built from the tail of the list to the head. By simply analyzing each step of this process we can see that Smallest Last Ordering can be done in $O(|V| + |E|)$ time. However, this running time is entirely dependent on our ability to find the minimum degree vertex in an efficient manner. To do this we instantiate the ordering process by first partitioning each vertex into buckets by their degree. This can be done in $O(|V|)$ time and does not require an initial sort to be performed. In my actual implementation this is achieved with the following logic:

```
for node in adjList.items():
      degree = len(node[1])
      buckets[degree] = node[0]
```

In Python len() is a constant time operation so this process can clearly be seen to run in $O(|V|)$ time. The "buckets" structure shown in the above code is a dictionary of queues. Queues are used to ensure the nodes are being processed in the correct order and that a node does not unexpectedly drop from extremely high degree to extremely low degree by nature of being randomly chosen from the bucket with minimum degree. This process also makes it trivial to find the largest and smallest degree node in the graph by picking the first and last bucket and the first and last element of these buckets respectively (all constant time operations).

To begin the Smallest Last Ordering process we first select and delete the vertex with minimum degree which as previously described can be done by choosing the first bucket and then popping the first element off of that queue. This is always a constant time operation as empty buckets are deleted as they become empty. Once found, the minimum degree vertex is deleted from the bucket and its neighbors degrees are update. This is trivial as we simply iterate through the deleted node's neighbors in the adjacency list and if they have not yet been deleted we reduce their degree by one. When there is only 1 bucket remaining, its contents will be the members of the terminal clique as they must all have the same degree.

Degree reduction is accomplished by moving the affected vertex down 1 to the next lowest bucket. This can be done for each neighbor of the deleted vertex with the following logic:

```
buckets[minDegree].remove(selectedVertex)
for neighbor in adjList[selectedVertex]:
      if neighbor in buckets:
            buckets[neighbor[bucket]].remove(neighbor)
            buckets[neighbor[bucket]-1].append(neighbor)
      if len(buckets[neighbor[bucket]) == 0:
            del buckets[neighbor[bucket]]
slOrdering[curr] = selectedVertex
curr -= 1
```

Because all structures used above are hash based they all operate in near constant time and add almost no overhead to the process. Additionally, this ensure the lowest bucket always contains the minimum degree node and can be accessed in constant time. So by simply analysis this process will run in $O(|E|)$ time. In the above code we are also recording the deleted node in the smallest last ordering which you will note is from back to front, also in constant time. This means that we are filling the list as we deleted nodes and at termination the list will be in order and complete.

If we carefully examine each step in the algorithm we will see that partition into buckets takes $O(|V|)$ time. Finding the minimum degree vertex, deleting it and updating its neighbors is done in near constant time. When repeatedly run until all vertices have been processed this step can be done in $O(|E|)$ time. Thus the entire Smallest Last Ordering process runs in $O(|V| + |E|)$ time as we visit each node and each edge exactly once. This process runs in $O(|V|)$ extra space as we will need to record the resulting order of the vertices in a list.

Graph Coloring Algorithm

Now that we have gotten the Smallest Last Ordering we can apply a graph coloring algorithm to generate independent sets that we will later use for backbone selection. This is a traditional greedy coloring algorithm that is often referred to as the "Grundy coloring algorithm" [10]. This algorithm assigns the lowest available color to a vertex being considered. In this context "available" simply means no vertex connected to the one being considered is currently using that color.

Color assignment begins with the first element in the Smallest Last Order, which is the vertex that was deleted last in the Smallest Last Ordering process described above and assign it "color 0." We then continue to iterate over the Smallest Last Order coloring nodes as we go. If no color is currently available we add a new color to the pool and assign the vertex under consideration this color. Each time a new color is added, it is currently the maximum value color. When we arrive at each new vertex to be colored we get the list of its neighbors from the adjacency list dictionary in constant time and also get the colors of each of these vertices from the coloring dictionary. If any of its neighbors are not currently colored, that neighbor is not considered. The lowest color that is not currently assigned to one of these neighbors is assigned to the vertex under consideration.

Since we are going through each edge and each vertex exactly once and we are only adding new colors as they are needed we can see that this coloring algorithm runs in $O(|V| + |E|)$ time. We can also efficiently determine the lowest color that is unused with the following logic:

```
for color in itertools.count():
    if color not in neighbor_colors:
        break
```

Where neighbor_colors is a list containing the colors of all neighbors of the vertex being considered. This process counts up from zero and if that number (color) is not in neighbor_colors it is assigned to the vertex under consideration. The graph coloring algorithm runs in $O(m)$ space

complexity where *m* is the maximum number of colors needed to color the graph. An upper bound on the number of colors needed to successfully color the graph is equal to the maximum minimum-degree in the graph + 1. This is another great metric that can be used to ensure our program is working properly.

One advantage (and the primary reason) of using the Smallest Last Order to color the graph is that minimum degree nodes will be colored last ensuring that the vertices colored first will be of large degree. Because of this, the larger degree vertices and thereby the best nodes to be in a backbone will be colored first, thus aggregating the largest color sets towards the lower colors, making backbone selection *much* easier.

Bipartite Backbone Selection

The final step in the project is to generate bipartite backbones from the coloring information determined above. As previously mentioned a backbone is a connect bipartite subgraph that will be the main trunk for communication in the wireless sensor network. In the optimal case, every node will have access to a node in the backbone which is equivalent to the backbone having vertex coverage equal to 100%. It then stands to reason that the best metric for determining the quality of a selected backbone is its vertex coverage (also known as domination percentage). Backbone generation occurs by selecting the largest color classes and joining their vertices according the same rule as for the entire RGG (vertices are connected if they are less than or equal to R from each other). Because we used the Smallest Last Order to color our graph we can say with a high degree of certainty that the largest color class will likely be one of the lowest colors used. It would stand to reason then that the backbone will almost always be comprised of vertices in color set 0 or 1 but this is not always the case. Instead we must test the combinations of the top *k* colors where *k = 2*number of backbones desired*.

This project specifically asks for the top two backbones that be generated from the top 4 color classes. Because there are 4 colors, there a 6 possible bipartite combinations of colors and each must be tested. Getting the vertices that correspond to each color class is extremely fast as we can make use of our coloring hash map data structure that maps each vertex to its color in near (amortized) constant time.

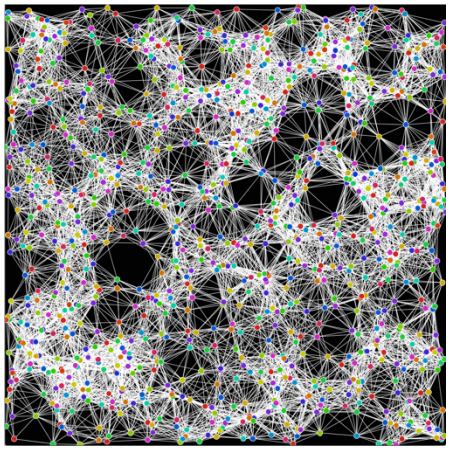
For each of the 6 combinations generated we must analyze the quality of the resulting backbone. Recall that a backbone must be connected to be considered a backbone, so the first step is to determine the size of the largest connected component in the graph. We do this by performing a component search on the bipartite subgraph that we are considering. This is done by a simple breadth first search starting from any vertex in the graph until it has reached all the nodes that it can. If all vertices in the bipartite graph were not reached, the number that were is recorded and this is noted as 1 component. The process is then reinitiated on an arbitrary remaining node until it has reached all the nodes it can. The process is repeated until every vertex has been visited once and thus all components have been determined. We would then return the largest connected component in that particular bipartite subgraph according to the project description. In most cases this “largest” component will have the highest domination percentage. However consider the edge case where there are two components of equal size. It would then be better to return the component with the highest *vertex coverage* instead of size. My implementation does this instead

to help ensure that the best possible backbone is being chosen every time. After component search is run the final step is to choose the two resulting backbones that have the highest vertex coverage. This selection process is trivial and occurs in constant time ultimately returning the best two backbones achievable using this procedure. Vertex coverage is calculated by examining all vertices that can be reached by each vertex in the backbone and dividing that number by the total number of vertices in the graph. This is done in $O(|k|)$ time where k is the number of vertices in the backbone. Technically speaking the running time of this process is bounded by $O(|V| + |E|)$ due to the component search, but in practice it is likely to run in a more favorable time because if a component has more than 50% of the vertices than it is automatically assumed to be the largest connected component in that bipartite subgraph. This process requires $O(|V|)$ extra space to store information about which vertices are members of which backbone.

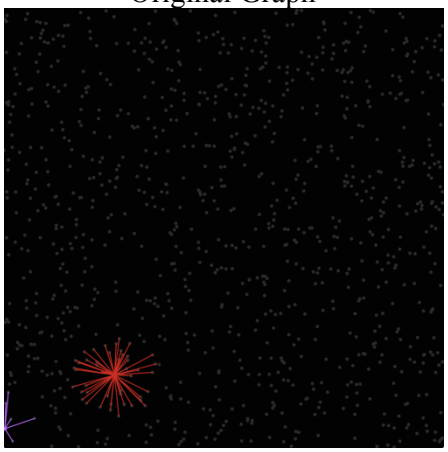
Additional Verification

To ensure that my project generates results correctly and efficiently, I decided to run a few additional benchmarks of my own in addition to those required in the project description. These benchmarks were all substantially larger than any of the 10 benchmarks we were asked to perform. I have selected only one of these, with 100,000 vertices, to include in my project appendix as I am already dangerously near the page limit. I have run my simulation with as many as 128,000 vertices with success. Unfortunately, with graphs over 100,000 nodes Microsoft excel was unable to plot the original vs deleted degree on my computer. It is my hope that the inclusion of this additional benchmark, which are included in the appendix at the end of the paper will convince the reader that the overall process scales incredibly well!

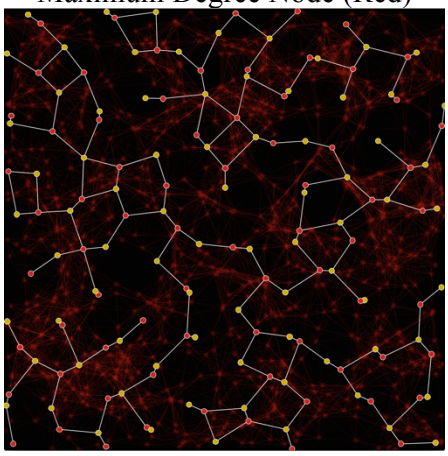
Benchmark 1: Square with n = 1000 vertices and R ≈ .10



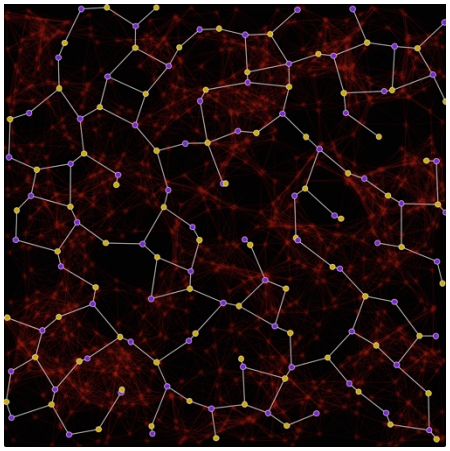
Original Graph



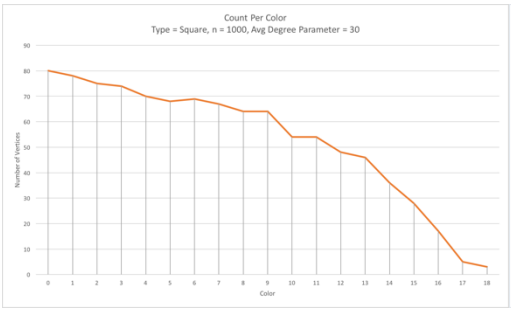
Min Degree Node (Blue) and Maximum Degree Node (Red)

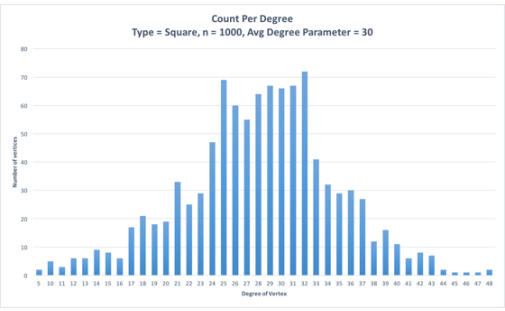


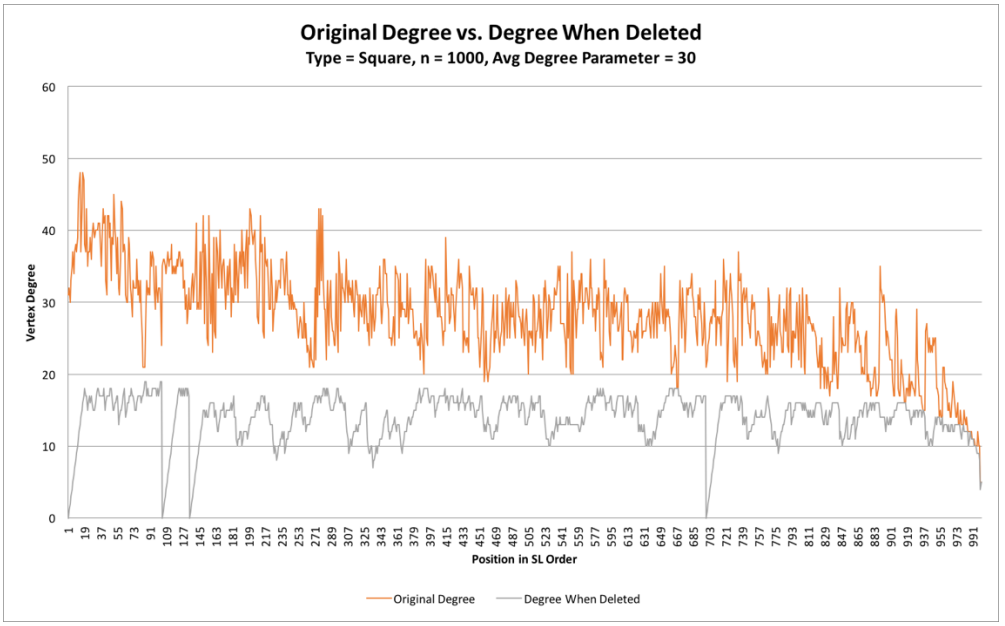
Backbone 1



Backbone 2

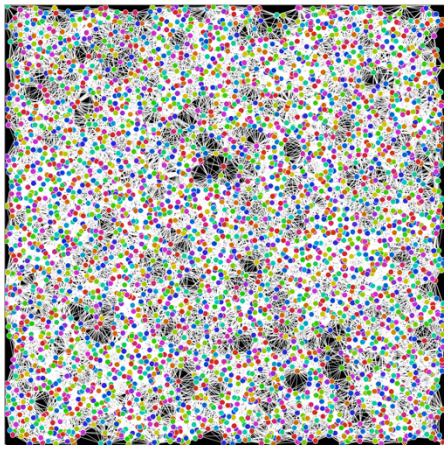




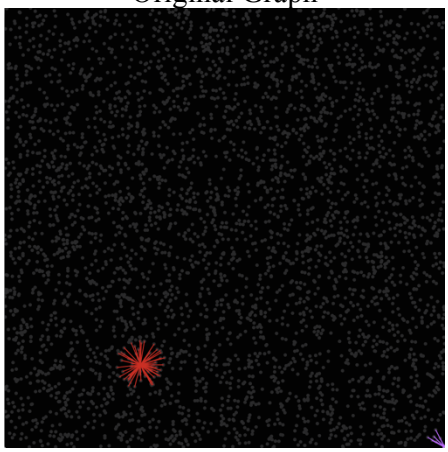


ID	1	**RGG Type**	Square
Num Vertices	1000	**Superior Backbone**	Backbone1
R	0.099	**Backbone1 Vertices**	152
Desired Avg Degree	30	**Backbone1 Edges**	167
Num Edges	14069	**Backbone1 Coverage**	99.60%
Min Degree	5	**Backbone2 Vertices**	154
Avg Degree	28.138	**Backbone2 Edges**	177
Max Degree	48	**Backbone2 Coverage**	99.30%
Max Degree When Deleted	19	**Max Color Size**	80
Number of Colors	19	**Terminal Clique Size**	19

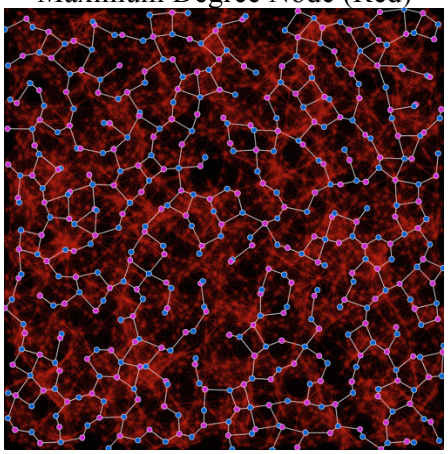
Benchmark 2: Square with n = 4000 vertices and R ≈ .06



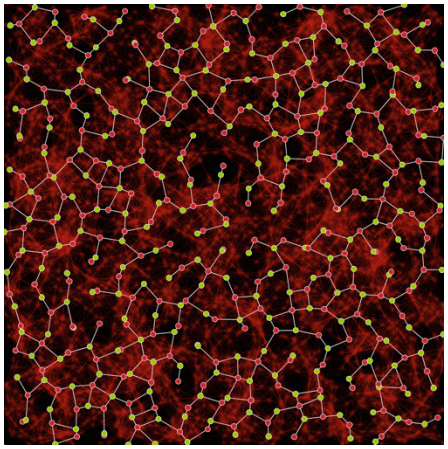
Original Graph



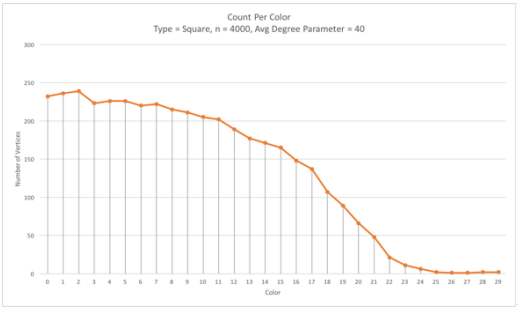
Min Degree Node (Blue) and Maximum Degree Node (Red)

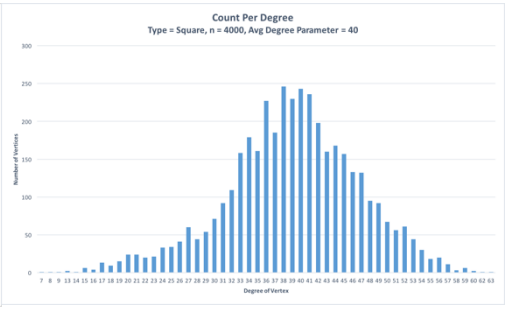


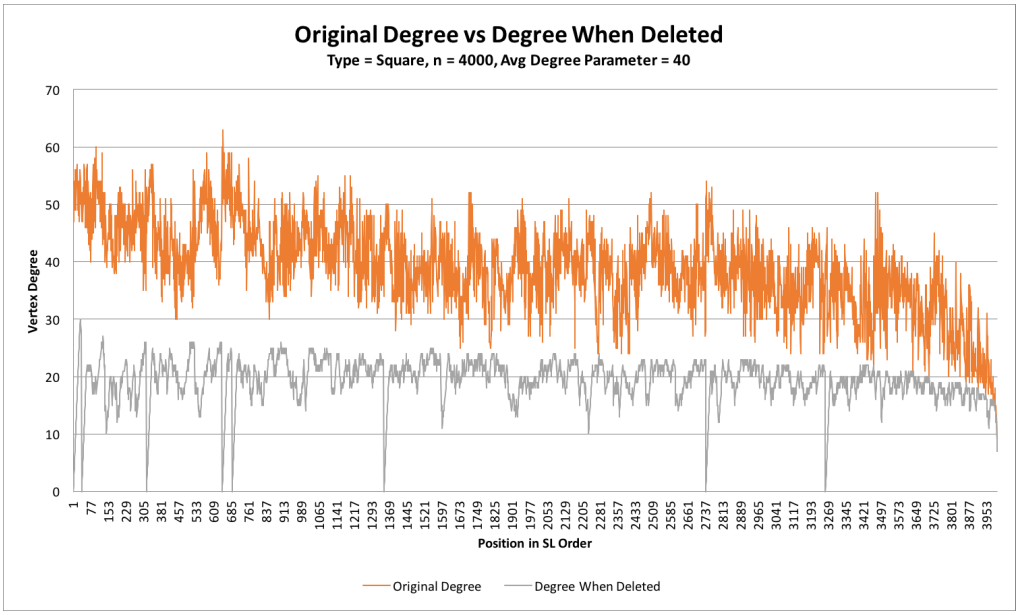
Backbone 1



Backbone 2

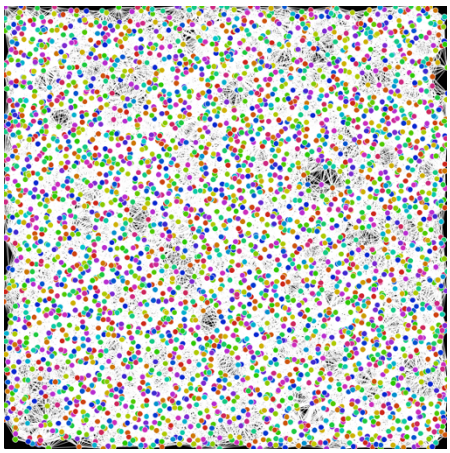




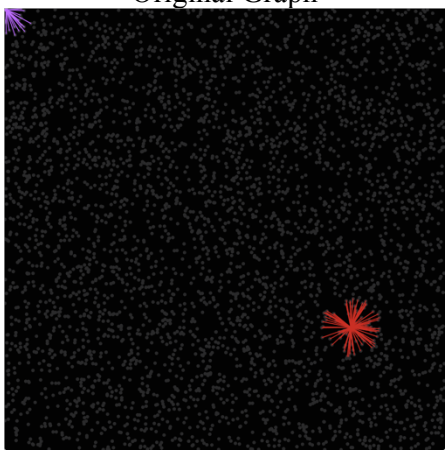


ID	2	**RGG Type**	Square
Num Vertices	4000	**Superior Backbone**	Backbone1
R	0.057	**Backbone1 Vertices**	450
Desired Avg Degree	40	**Backbone1 Edges**	536
Num Edges	77969	**Backbone1 Coverage**	99.75%
Min Degree	7	**Backbone2 Vertices**	463
Avg Degree	38.985	**Backbone2 Edges**	555
Max Degree	63	**Backbone2 Coverage**	99.65%
Max Degree When Deleted	30	**Max Color Size**	239
Number of Colors	30	**Terminal Clique Size**	30

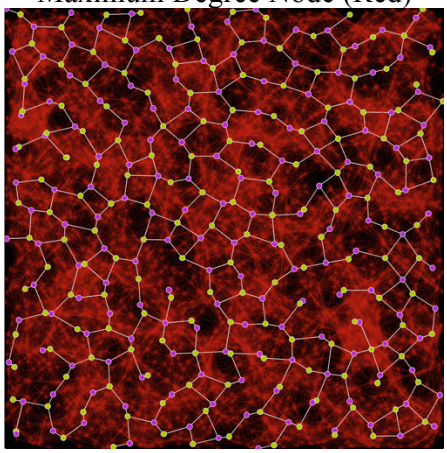
Benchmark 3: Square with n = 4000 vertices and R ≈ .07



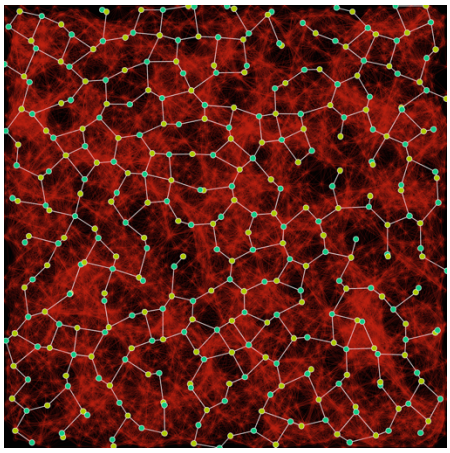
Original Graph



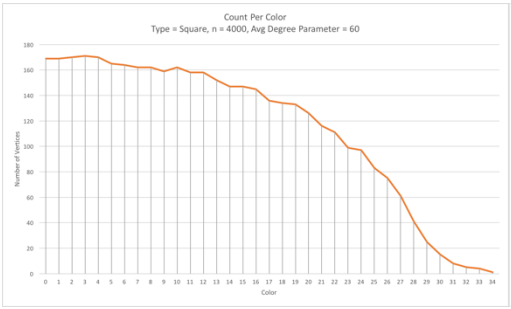
Min Degree Node (Blue) and Maximum Degree Node (Red)

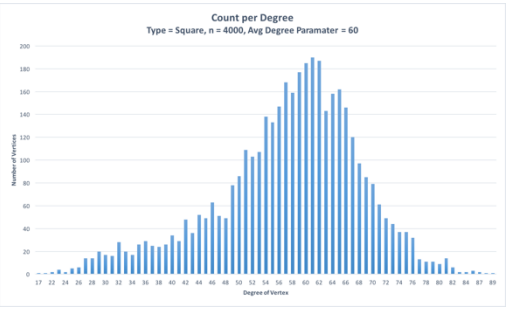


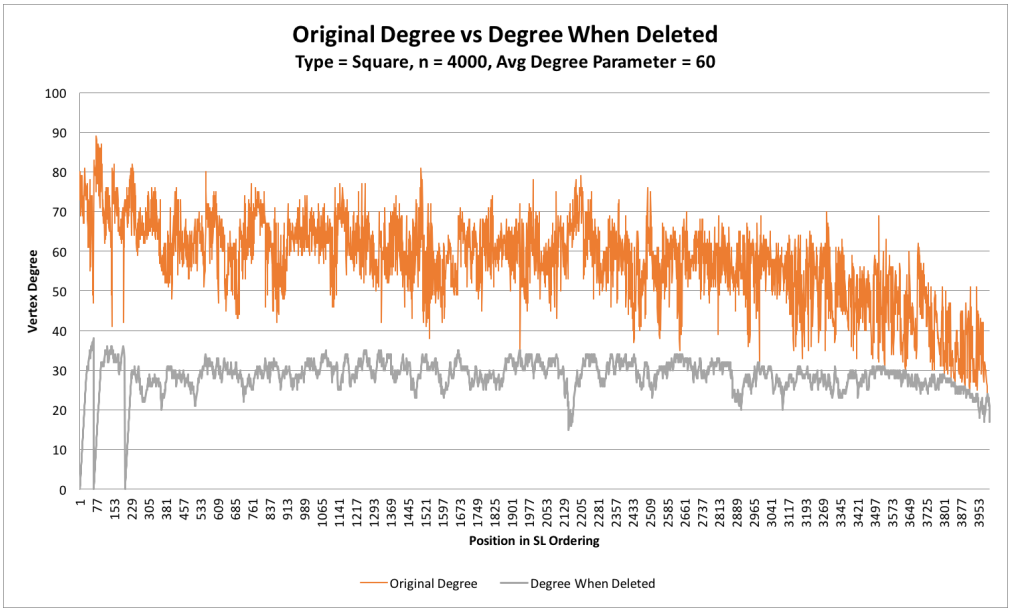
Backbone 1



Backbone 2

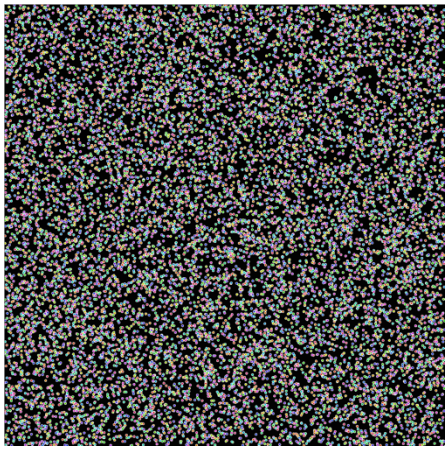




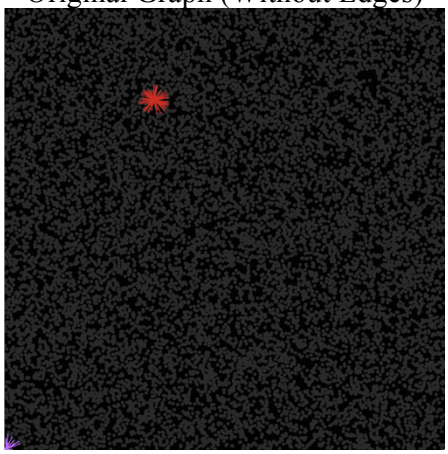


ID	3	**RGG Type**	Square
Num Vertices	4000	**Superior Backbone**	Tie
R	0.070	**Backbone1 Vertices**	337
Desired Avg Degree	60	**Backbone1 Edges**	424
Num Edges	114817	**Backbone1 Coverage**	99.98%
Min Degree	17	**Backbone2 Vertices**	338
Avg Degree	57.409	**Backbone2 Edges**	422
Max Degree	89	**Backbone2 Coverage**	99.98%
Max Degree When Deleted	38	**Max Color Size**	171
Number of Colors	35	**Terminal Clique Size**	30

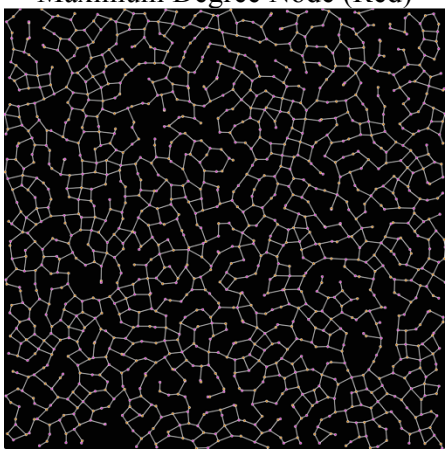
Benchmark 4: Square with n = 16000 vertices and R ≈ .035



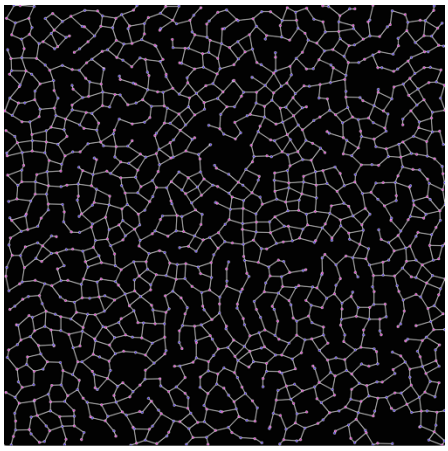
Original Graph (Without Edges)



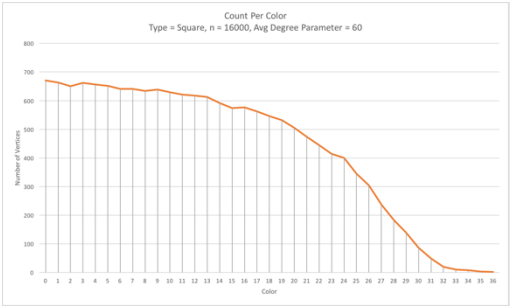
Min Degree Node (Blue) and Maximum Degree Node (Red)

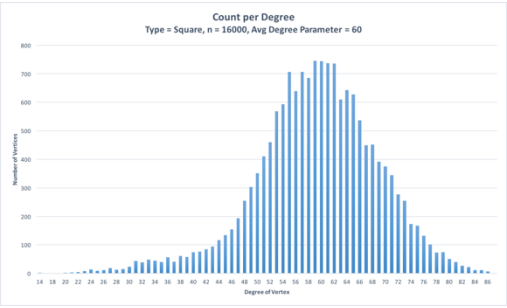


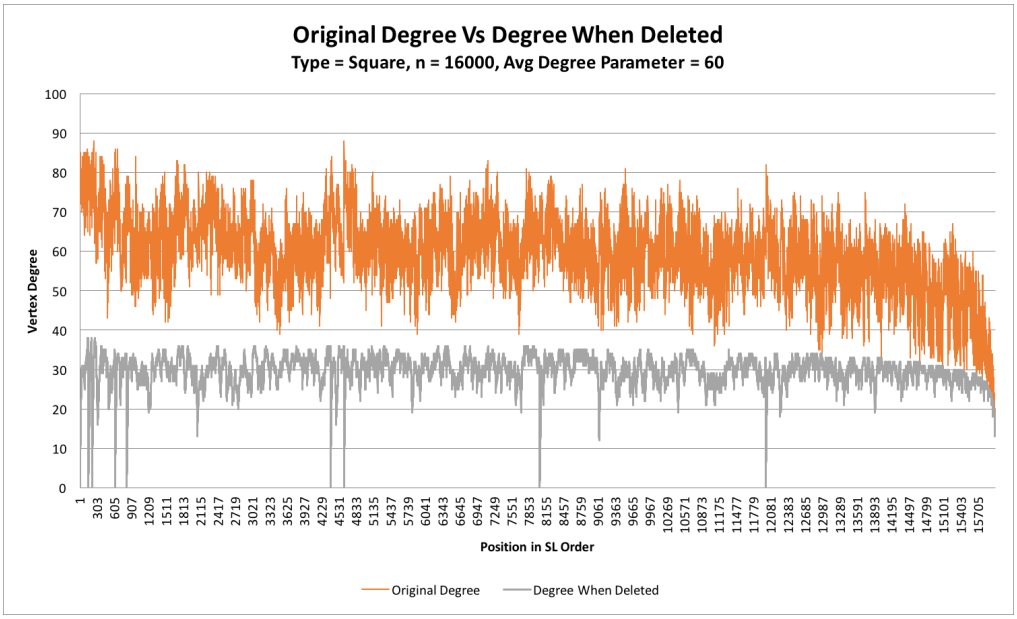
Backbone 1



Backbone 2

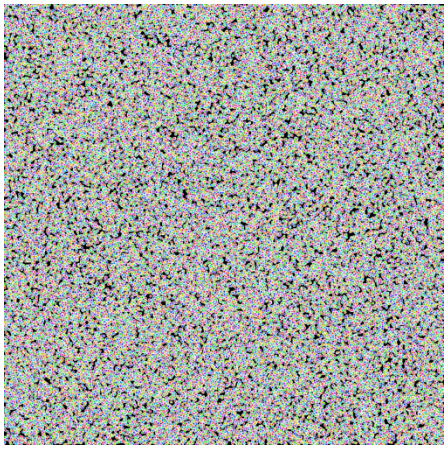




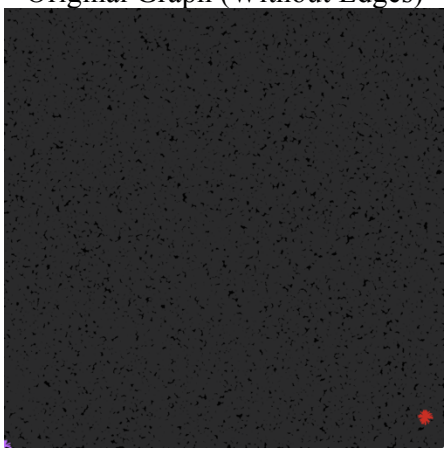


ID	4	**RGG Type**	Square
Num Vertices	16000	**Superior Backbone**	Backbone1
R	0.035	**Backbone1 Vertices**	1308
Desired Avg Degree	60	**Backbone1 Edges**	1671
Num Edges	473825	**Backbone1 Coverage**	99.81%
Min Degree	14	**Backbone2 Vertices**	1315
Avg Degree	59.228	**Backbone2 Edges**	1685
Max Degree	88	**Backbone2 Coverage**	99.79%
Max Degree When Deleted	38	**Max Color Size**	671
Number of Colors	37	**Terminal Clique Size**	31

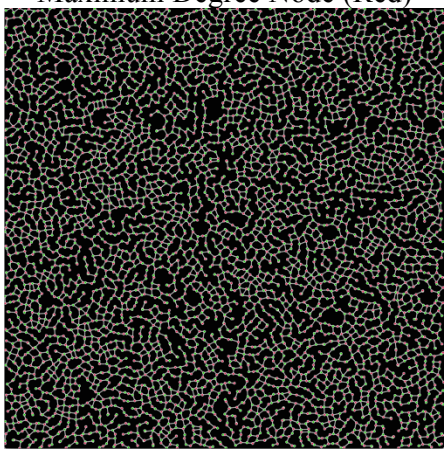
Benchmark 5: Square with n = 64000 vertices and R ≈ .017



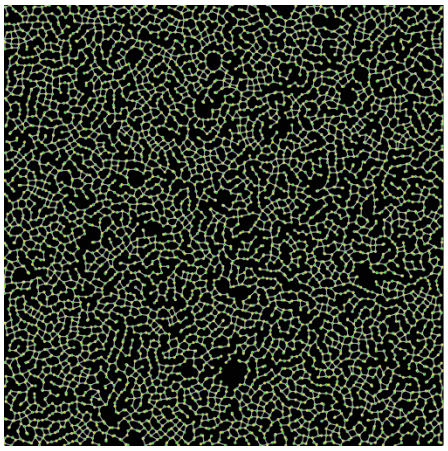
Original Graph (Without Edges)



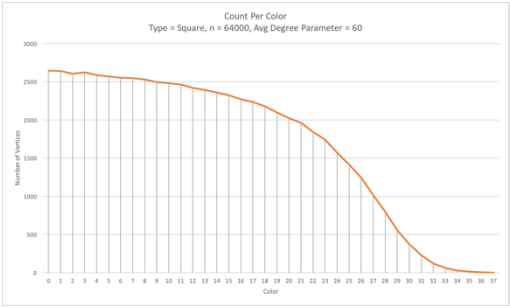
Min Degree Node (Blue) and Maximum Degree Node (Red)

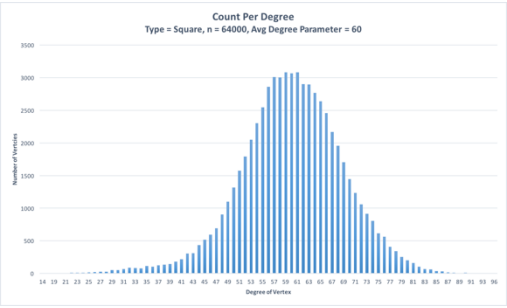


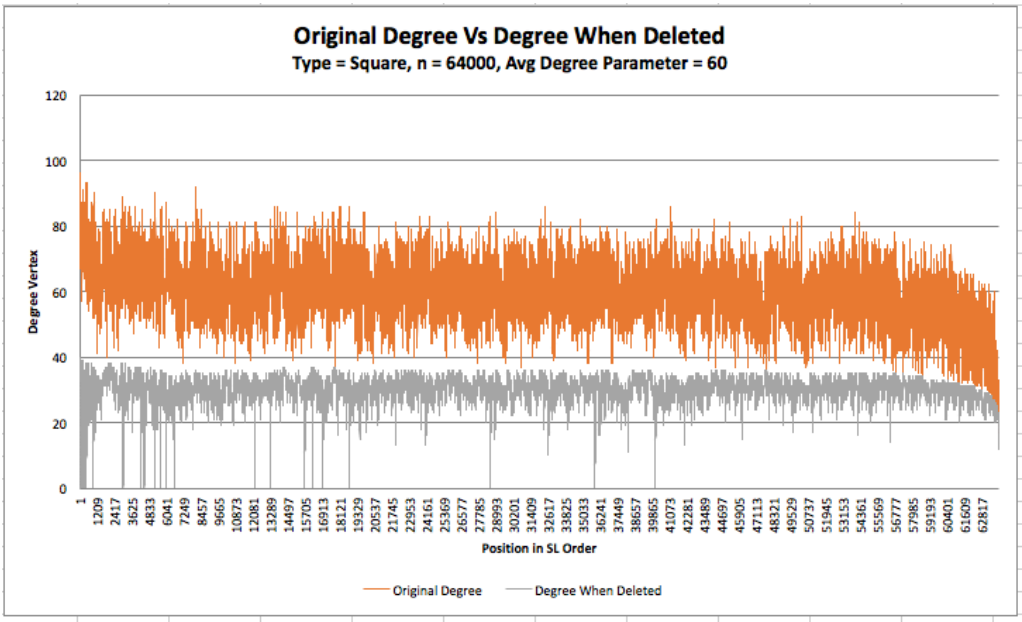
Backbone 1



Backbone 2

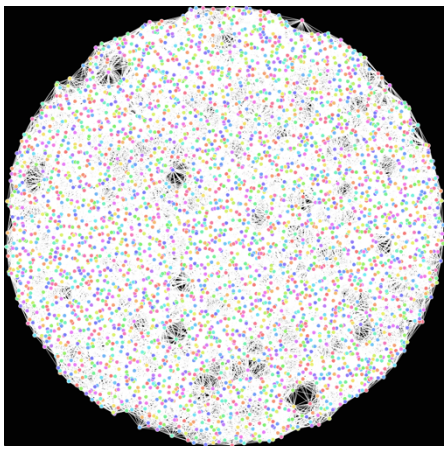




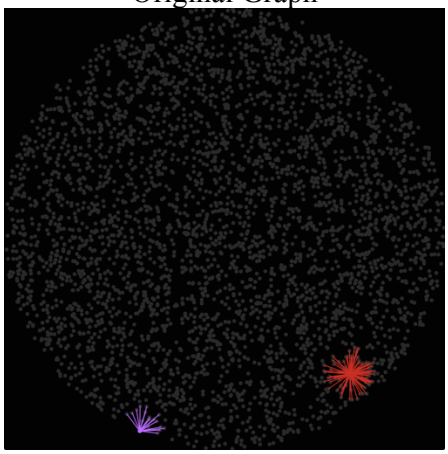


ID	5	**RGG Type**	Square
Num Vertices	64000	**Superior Backbone**	Backbone1
R	0.017	**Backbone1 Vertices**	5225
Desired Avg Degree	60	**Backbone1 Edges**	6781
Num Edges	1921111	**Backbone1 Coverage**	99.89%
Min Degree	14	**Backbone2 Vertices**	5200
Avg Degree	60.035	**Backbone2 Edges**	6676
Max Degree	96	**Backbone2 Coverage**	99.79%
Max Degree When Deleted	40	**Max Color Size**	2645
Number of Colors	38	**Terminal Clique Size**	34

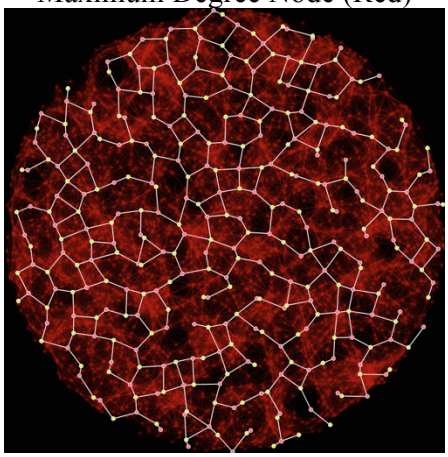
Benchmark 6: Disk with n = 4000 vertices and R ≈ .07



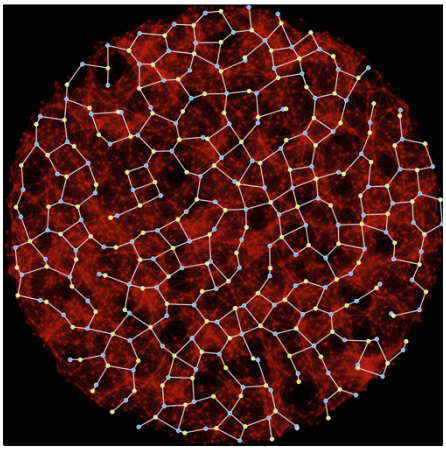
Original Graph



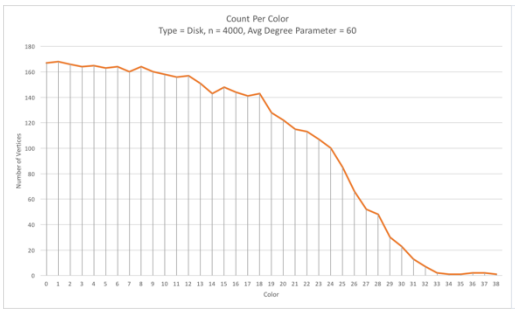
Min Degree Node (Blue) and Maximum Degree Node (Red)

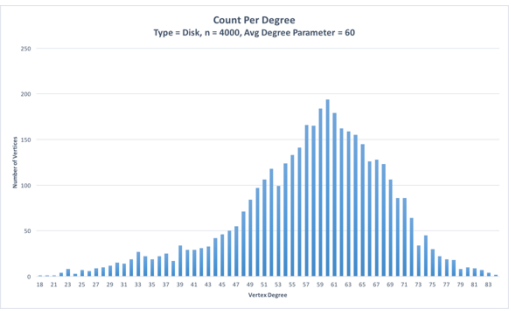


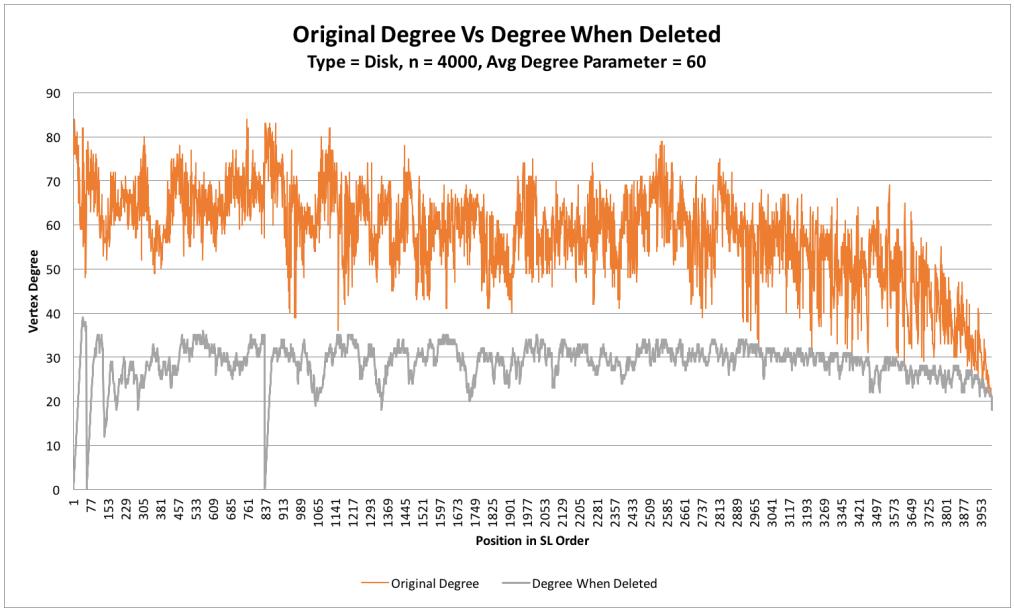
Backbone 1



Backbone 2

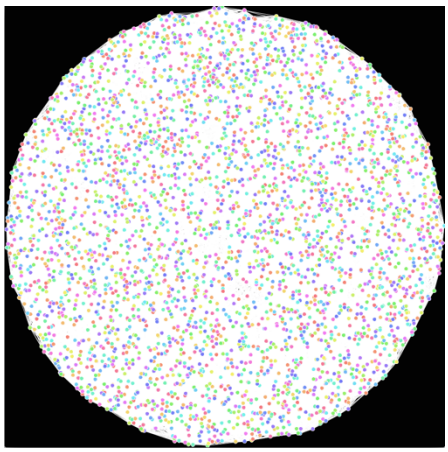




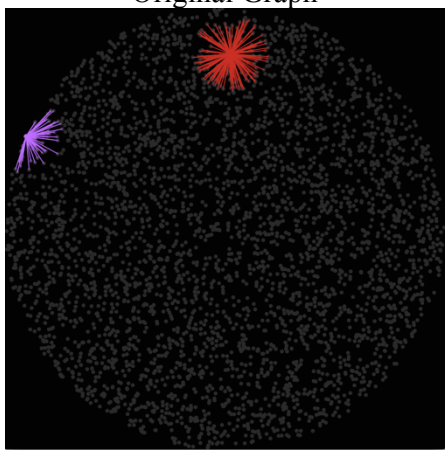


ID	6	**RGG Type**	Disk
Num Vertices	4000	**Superior Backbone**	Backbone1
R	0.070	**Backbone1 Vertices**	331
Desired Avg Degree	60	**Backbone1 Edges**	421
Num Edges	115513	**Backbone1 Coverage**	99.95%
Min Degree	18	**Backbone2 Vertices**	332
Avg Degree	57.757	**Backbone2 Edges**	424
Max Degree	84	**Backbone2 Coverage**	99.80%
Max Degree When Deleted	39	**Max Color Size**	168
Number of Colors	39	**Terminal Clique Size**	38

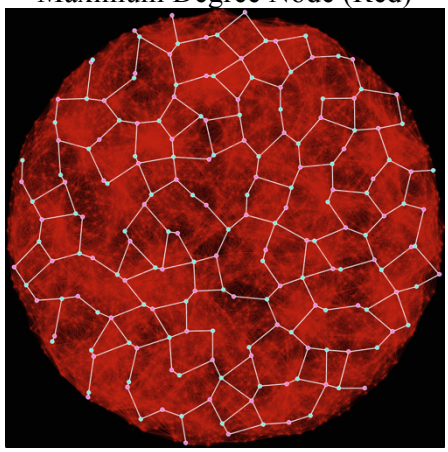
Benchmark 7: Disk with n = 4000 vertices and R ≈ .10



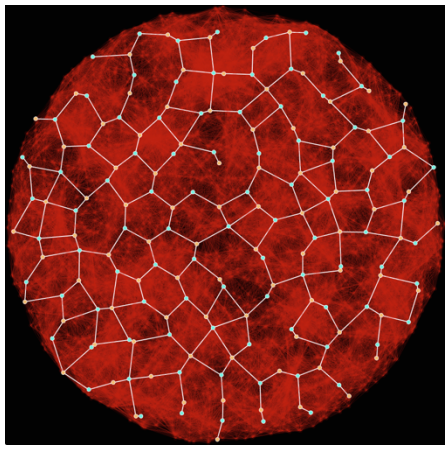
Original Graph



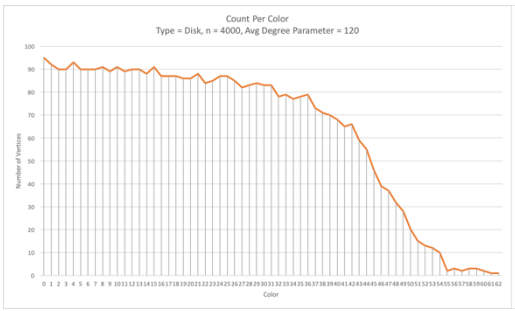
Min Degree Node (Blue) and Maximum Degree Node (Red)

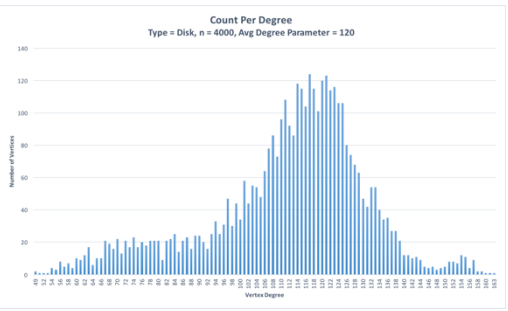


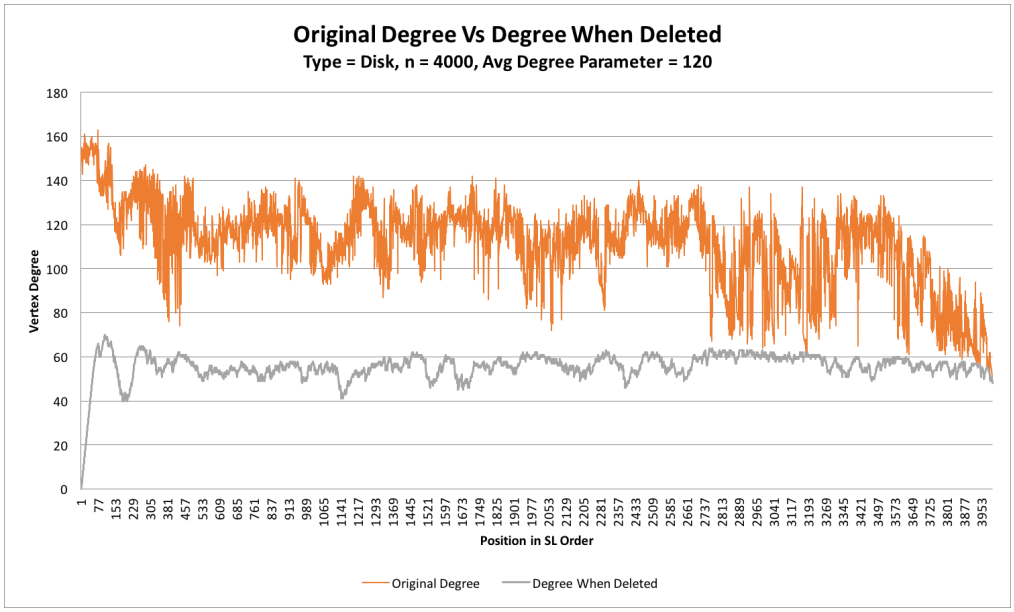
Backbone 1



Backbone 2

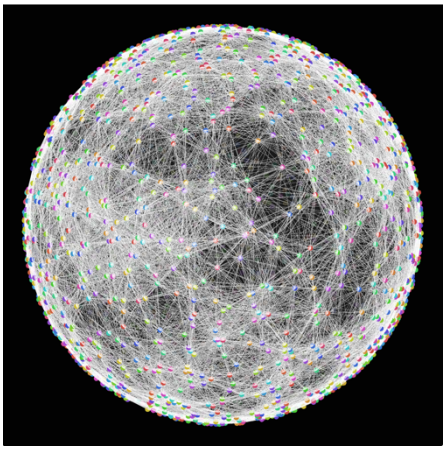




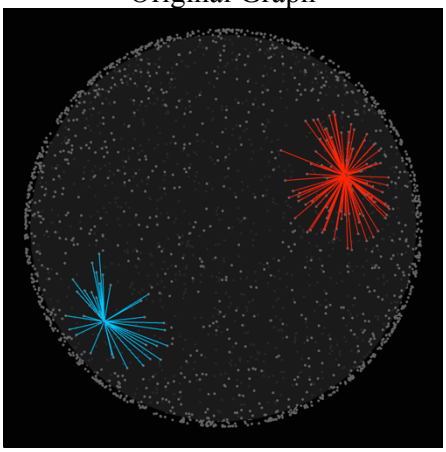


ID	7	**RGG Type**	Disk
Num Vertices	4000	**Superior Backbone**	Tie
R	0.098	**Backbone1 Vertices**	187
Desired Avg Degree	120	**Backbone1 Edges**	246
Num Edges	223118	**Backbone1 Coverage**	100.00%
Min Degree	49	**Backbone2 Vertices**	182
Avg Degree	111.559	**Backbone2 Edges**	240
Max Degree	163	**Backbone2 Coverage**	100.00%
Max Degree When Deleted	70	**Max Color Size**	95
Number of Colors	63	**Terminal Clique Size**	60

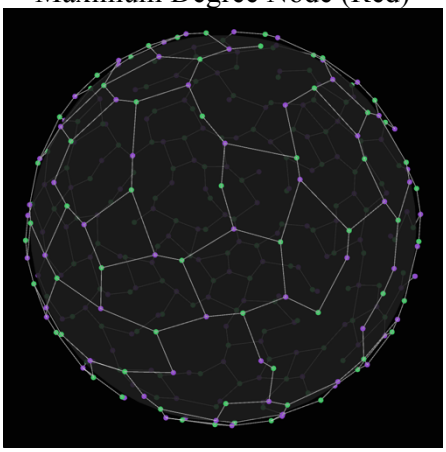
Benchmark 8: Disk with n = 4000 vertices and R ≈ .07



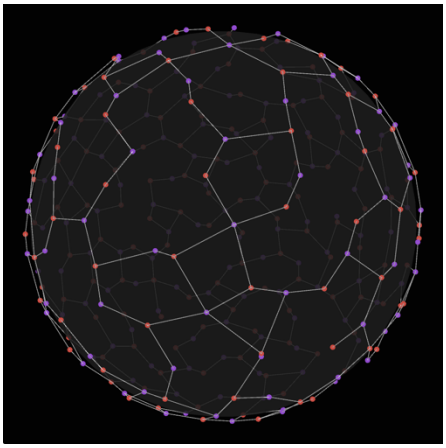
Original Graph



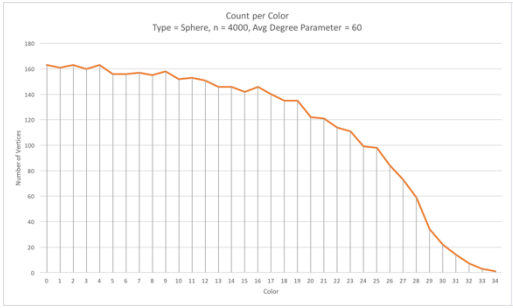
Min Degree Node (Blue) and Maximum Degree Node (Red)

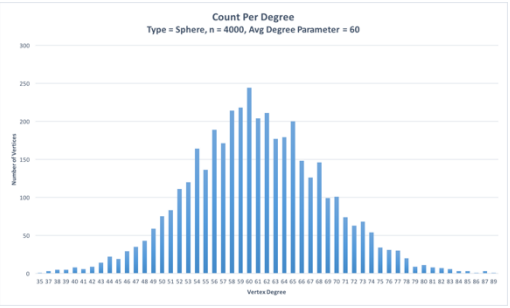


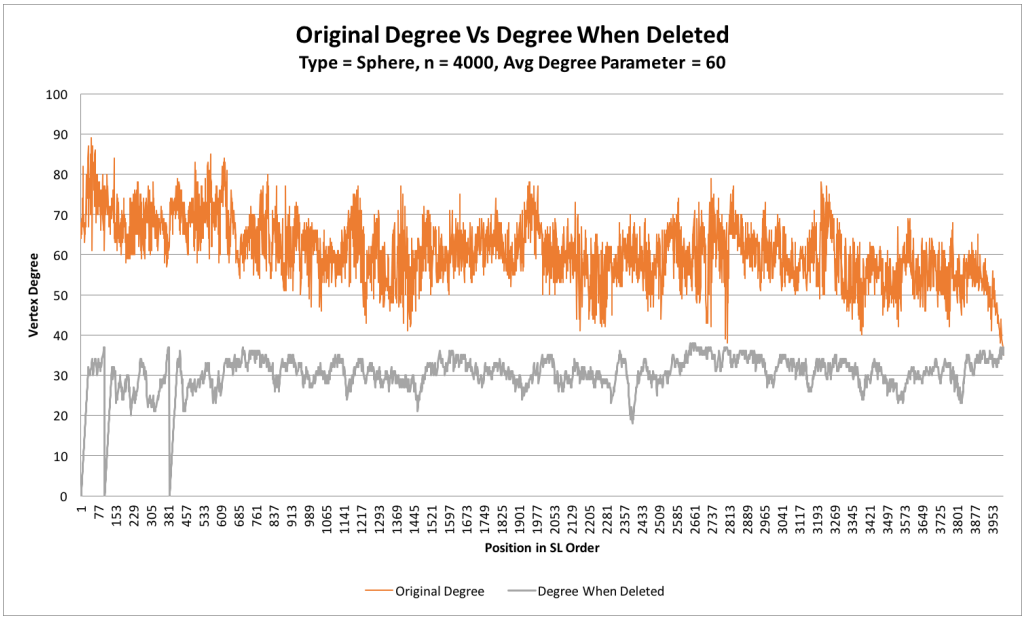
Backbone 1



Backbone 2

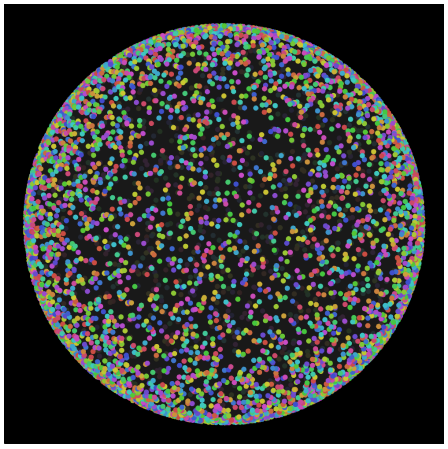




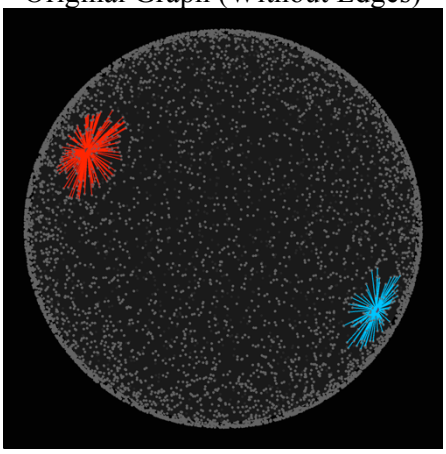


ID	8	**RGG Type**	Sphere
Num Vertices	4000	**Superior Backbone**	Tie
R	0.070	**Backbone1 Vertices**	322
Desired Avg Degree	60	**Backbone1 Edges**	416
Num Edges	121741	**Backbone1 Coverage**	99.95%
Min Degree	35	**Backbone2 Vertices**	319
Avg Degree	60.871	**Backbone2 Edges**	418
Max Degree	89	**Backbone2 Coverage**	99.95%
Max Degree When Deleted	38	**Max Color Size**	163
Number of Colors	35	**Terminal Clique Size**	33
Backbone1 Faces	96	**Backbone2 Faces**	101

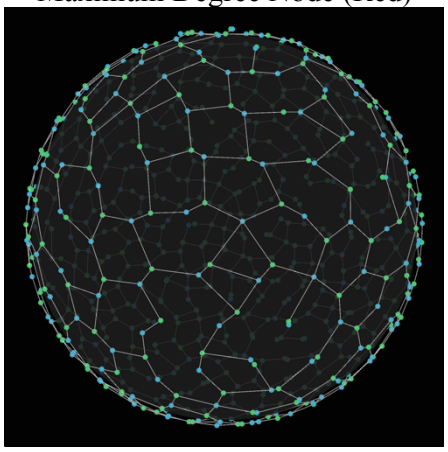
Benchmark 9: Disk with n = 16000 vertices and R ≈ .05



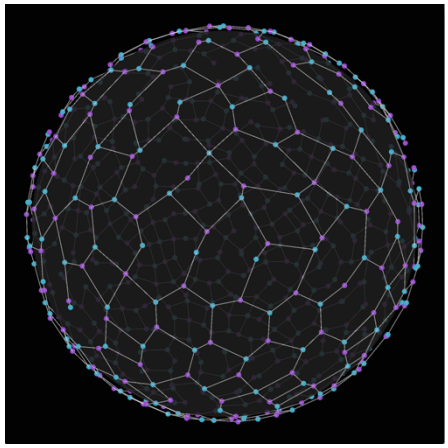
Original Graph (Without Edges)



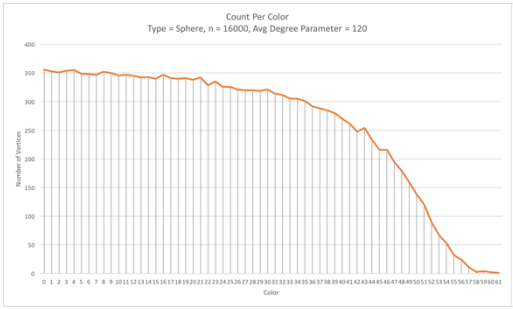
Min Degree Node (Blue) and Maximum Degree Node (Red)

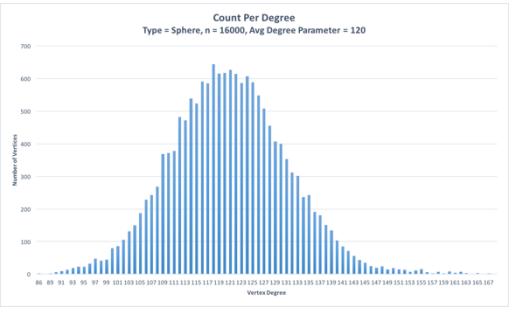


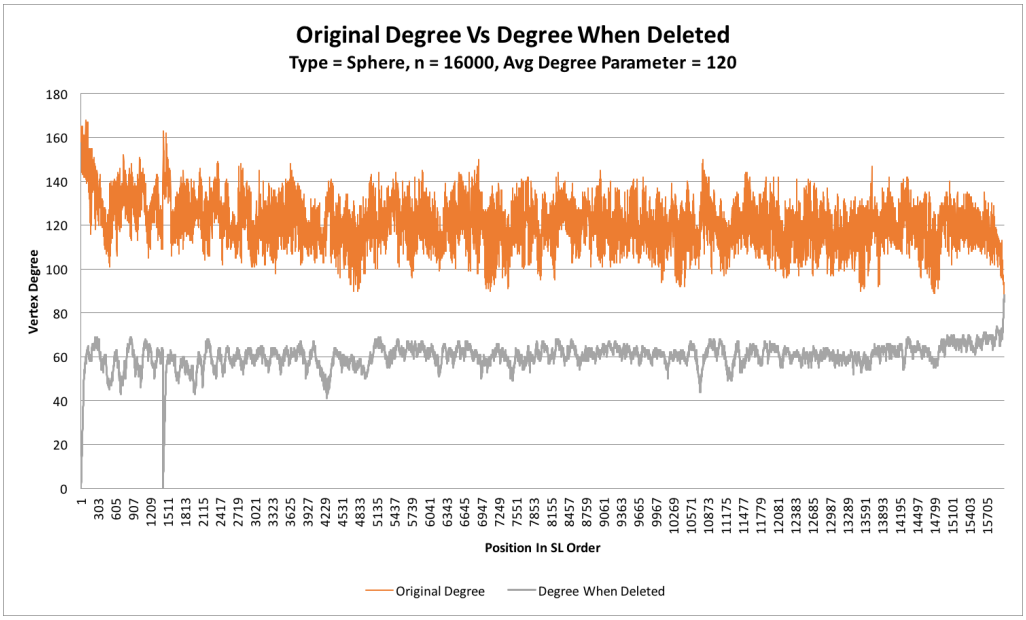
Backbone 1



Backbone 2

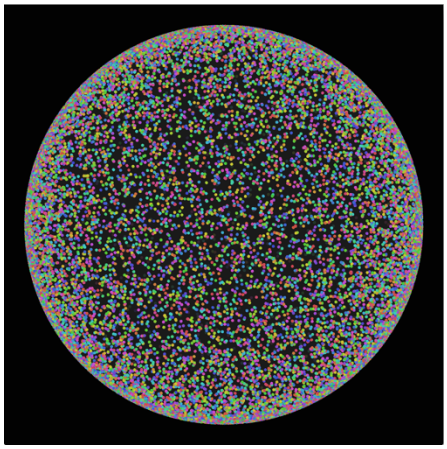




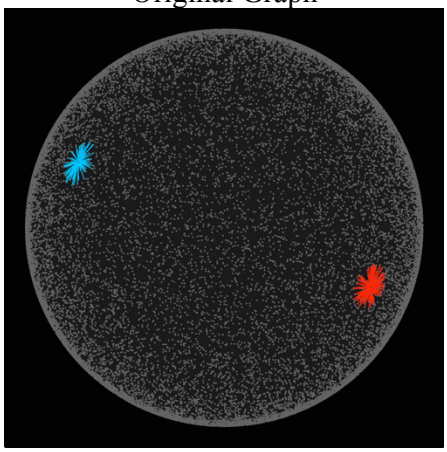


ID	9	**RGG Type**	Sphere
Num Vertices	16000	**Superior Backbone**	Tie
R	0.049	**Backbone1 Vertices**	708
Desired Avg Degree	120	**Backbone1 Edges**	979
Num Edges	967081	**Backbone1 Coverage**	99.99%
Min Degree	86	**Backbone2 Vertices**	705
Avg Degree	120.885	**Backbone2 Edges**	989
Max Degree	168	**Backbone2 Coverage**	99.99%
Max Degree When Deleted	88	**Max Color Size**	356
Number of Colors	62	**Terminal Clique Size**	52
Backbone1 Faces	273	**Backbone2 Faces**	286

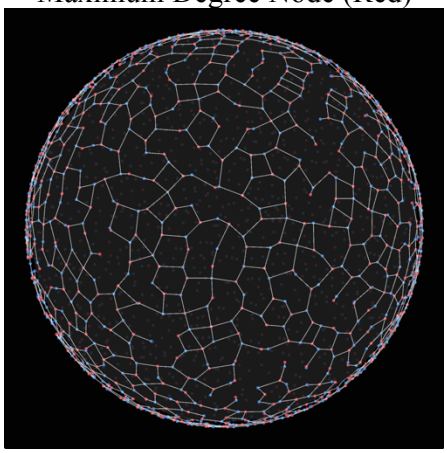
Benchmark 10: Disk with n = 64000 vertices and R ≈ .025



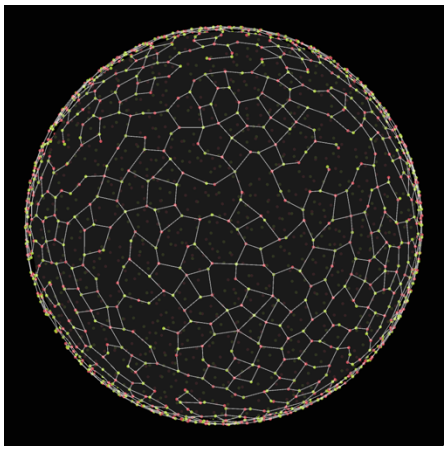
Original Graph



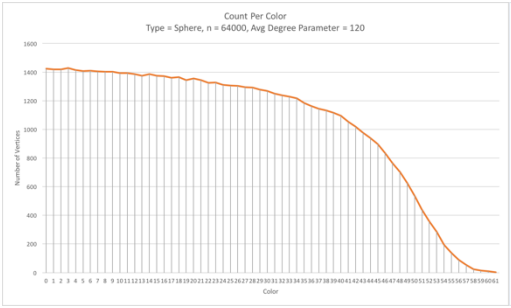
Min Degree Node (Blue) and Maximum Degree Node (Red)

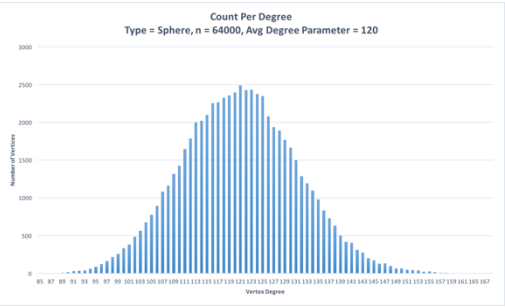


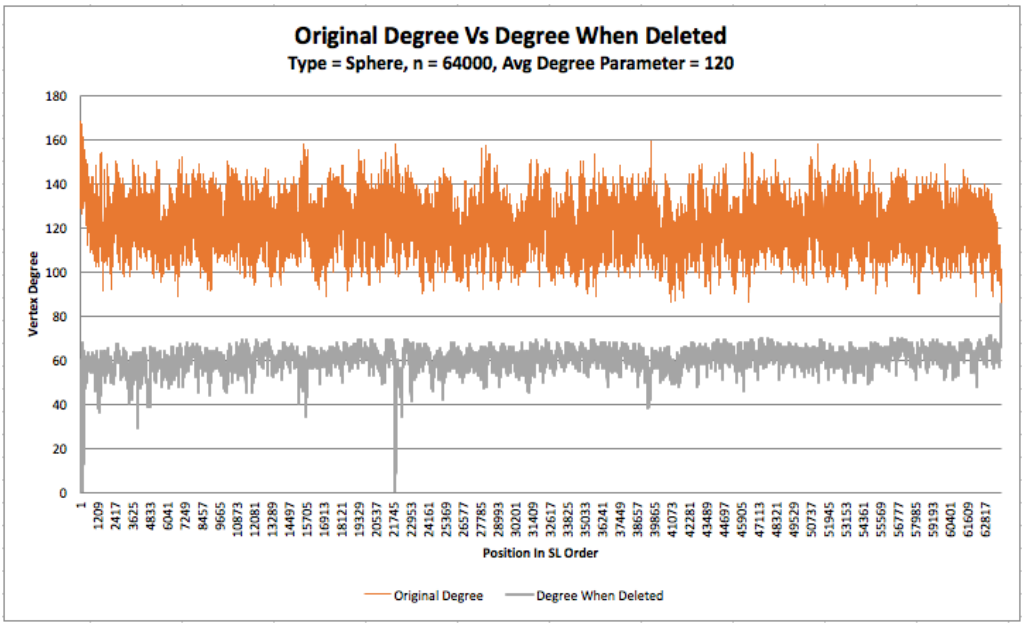
Backbone 1



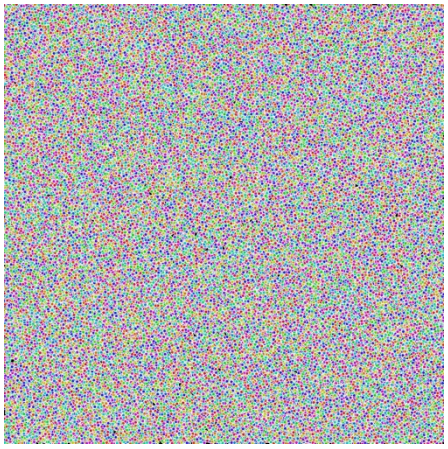
Backbone 2



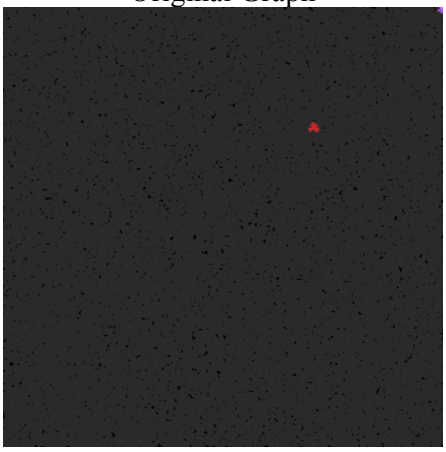




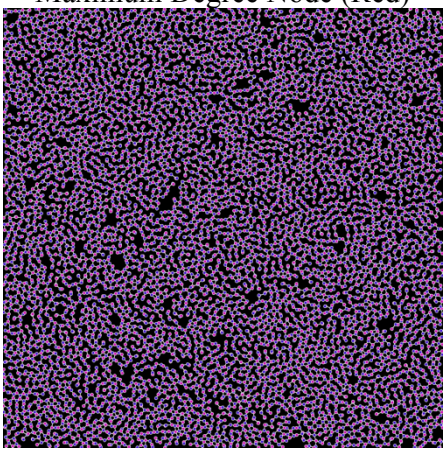
ID	10	**RGG Type**	Sphere
Num Vertices	64000	**Superior Backbone**	Backbone 1
R	0.025	**Backbone1 Vertices**	2836
Desired Avg Degree	120	**Backbone1 Edges**	3931
Num Edges	3868443	**Backbone1 Coverage**	99.988%
Min Degree	85	**Backbone2 Vertices**	2835
Avg Degree	120.889	**Backbone2 Edges**	3928
Max Degree	168	**Backbone2 Coverage**	99.986%
Max Degree When Deleted	85	**Max Color Size**	1428
Number of Colors	62	**Terminal Clique Size**	56
Backbone1 Faces	1097	**Backbone2 Faces**	1095



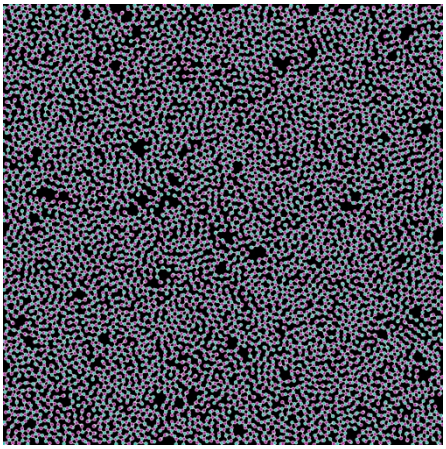
Original Graph



Min Degree Node (Blue) and Maximum Degree Node (Red)



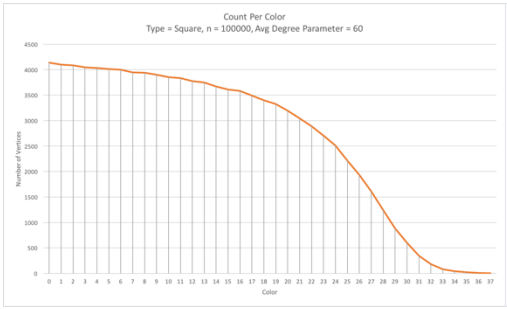
Backbone 1

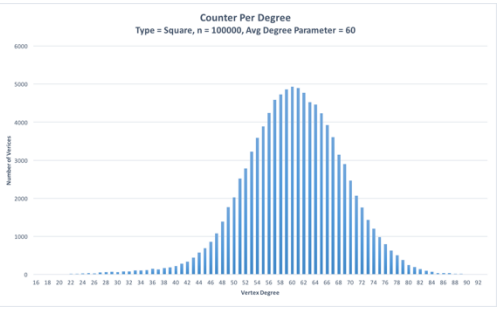


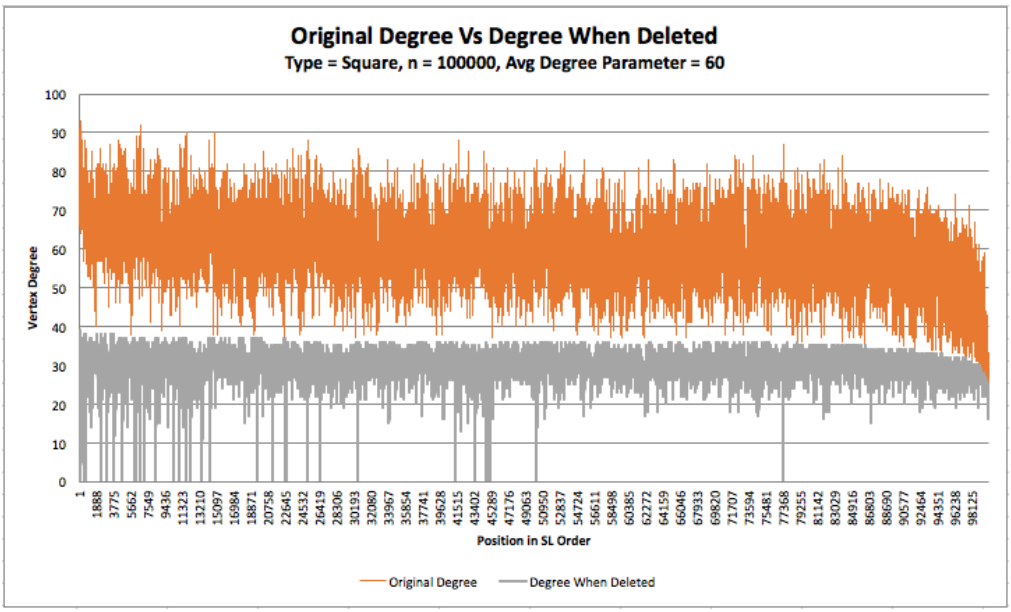
Backbone 2

APPENDIX

Benchmark 11: Disk with n = 100000 vertices and R ≈ .014







ID	7	**RGG Type**	Square
Num Vertices	100000	**Superior Backbone**	Backbone1
R	0.014	**Backbone1 Vertices**	8073
Desired Avg Degree	60	**Backbone1 Edges**	10402
Num Edges	3014338	**Backbone1 Coverage**	99.80%
Min Degree	16	**Backbone2 Vertices**	8106
Avg Degree	60.287	**Backbone2 Edges**	10446
Max Degree	93	**Backbone2 Coverage**	99.78%
Max Degree When Deleted	39	**Max Color Size**	4137
Number of Colors	38	**Terminal Clique Size**	32

The purpose of this additional benchmark is to show that my program can handle generating graphs that are substantially larger than what was requested in the project description. My program had no problems what so ever generating or visualizing a graph of this size. In quite a humorous twist, it was Microsoft Excel that had the problem as it could barely handle drawing the original vs deleted degree plot. Perhaps I should use R or MatPlotLib for large graphs in the future!

Look at how smooth the color and degree distributions are!