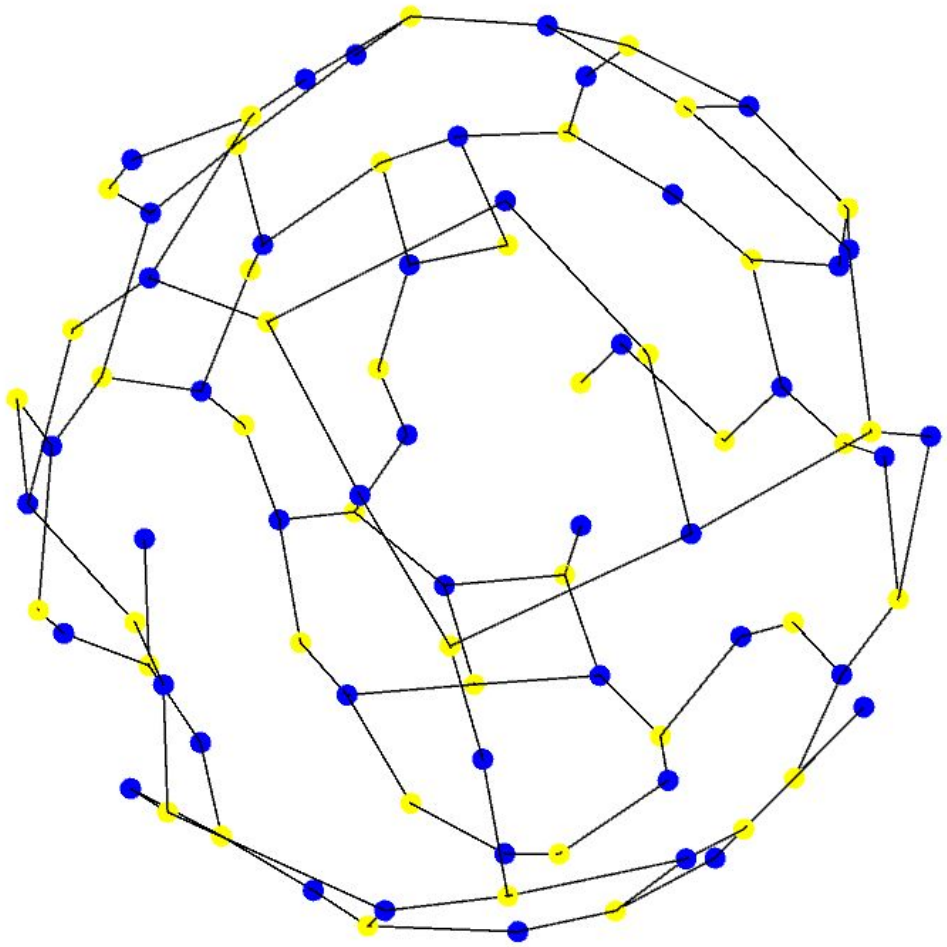
Eric Smith
eric@smu.edu

CSE7343 Algorithm Engineering

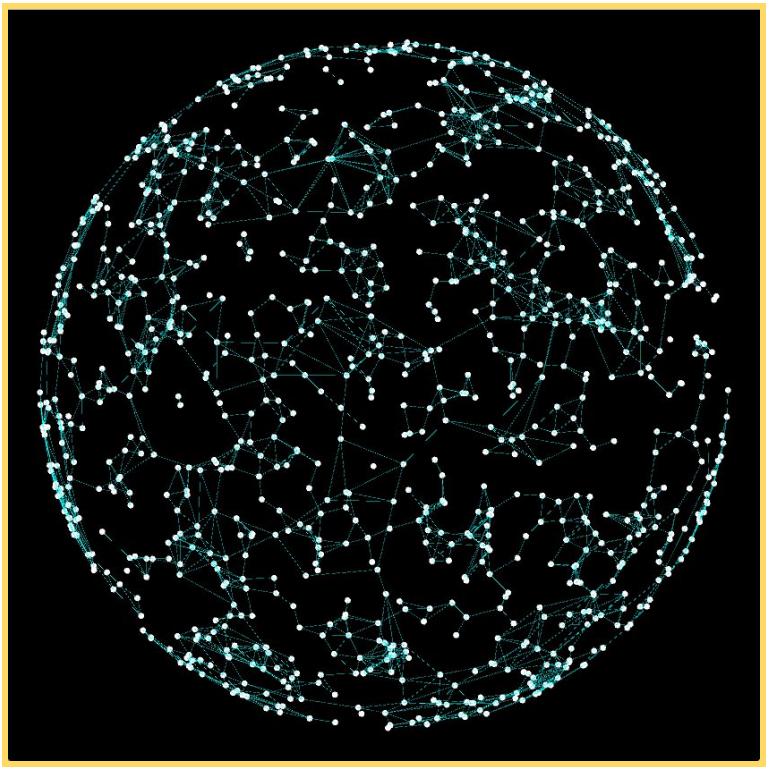
May 8th, 2017



Wireless Sensor Network Project

Table of Contents

EXECUTIVE SUMMARY



Introduction

This paper documents the successful implementation of a *wireless sensor network* program. A program that is able to distribute nodes on the surfaces of multiple shapes - representing different network topologies. It can even connect these nodes given some calculated distance. From there, the program has the ability to "color" each node based off its neighbors. Finally, the program can display the largest *backbone* of the graph, i.e. the structure that connects the largest amount of nodes on the network.

Programming Environment Description

N	*Average Degree*	*Topology*	*Peak Memory*	*Peak CPU*	*Part I*
6400	128	Square	8232 MB	331%	1014 ms
6400	128	Disk	8235 MB	278%	1011 ms
6400	128	Sphere	8231 MB	308%	1921 ms

This project was developed entirely on a 2016 MacBook Pro running MacOS 10.12.4. The computer has an integrated, Intel Iris Graphics 540 on a 2.4 GHz Intel Core i7 with 16 gigabytes of low powered DDR3 memory.

The code itself was implemented in Java 8 using the graphics framework, Processing 3[7]. The table below shows the peak memory and CPU usage as well as the time it took for the program to finish *Part I*. Considering that the CPU usage peaked at 331%, I think using this machine was a perfect fit for developing this project.

(*Note*: I originally started implementing the program in Python, but the performance was terrible.)

Summary

Benchmark	N	R	M	Min Deg	Avg Deg	Real Avg Deg	Max Deg
1 - Square	1000	0.10	14668	11	32	29	27
2 - Square	4000	0.07	120289	20	64	60	83
3 - Square	16000	0.04	496949	12	64	62	67
4 - Square	64000	0.02	2017478	17	64	63	80
5 - Square	64000	0.03	4004023	33	128	125	134
6 - Disk	4000	0.13	121837	15	64	60	32
7 - Disk	4000	0.18	237212	44	128	118	127
8 - Sphere	4000	0.25	127900	40	64	63	68
9 - Sphere	16000	0.18	1024134	89	128	128	119
10 - Sphere	64000	0.09	4095340	89	128	127	127

Benchmark	Max Deg when Deleted	Colors	Largest Color Size	Terminal Clique Size	N of Largest Backbone	M of Largest Backbone	Backbone Domination %
1 - Square	31	23	126	1	126	125	12.60%
2 - Square	63	38	276	5	276	275	6.90%
3 - Square	71	42	985	1	985	984	6.16%
4 - Square	70	41	3807	3	3807	3806	5.95%
5 - Square	135	73	2165	1	2165	2164	3.38%
6 - Disk	65	39	277	3	277	276	6.93%
7 - Disk	120	71	152	1	152	151	3.80%
8 - Sphere	64	39	250	2	250	249	6.25%
9 - Sphere	128	73	540	17	540	539	3.38%
10 - Sphere	131	75	2103	1	2103	2102	3.29%

My program turned out to be a strong, dynamic implementation of a wireless sensor network. Each benchmark listed above is easily met in a reasonable time. The **major result** of this program is a collection of data that can be used when deciding on a wireless network topology, as well as a real time display of the implementation. Furthermore, writing this program was an incredible learning experience of a truly fascinating topic. I enjoyed it very much!

References

[1] D.W. Matula, L.L. Beck: *Smallest-Last Ordering and Clustering and Graph Coloring Algorithms*, 1983.

[2] S.K. Gupta, A. Singh: *On Tree Roots of Graphs*, 2012.

[3] F. Harary: *Graph Theory*, 1969.

[4] Z. Chen: *Wireless Sensor Network,* 2012.

[5] C. Simon: *Generating Uniformly Distributed Numbers on a Sphere* (corysimon.github.io), 2015.

[6] StackOverflow.com: *General programming solutions*

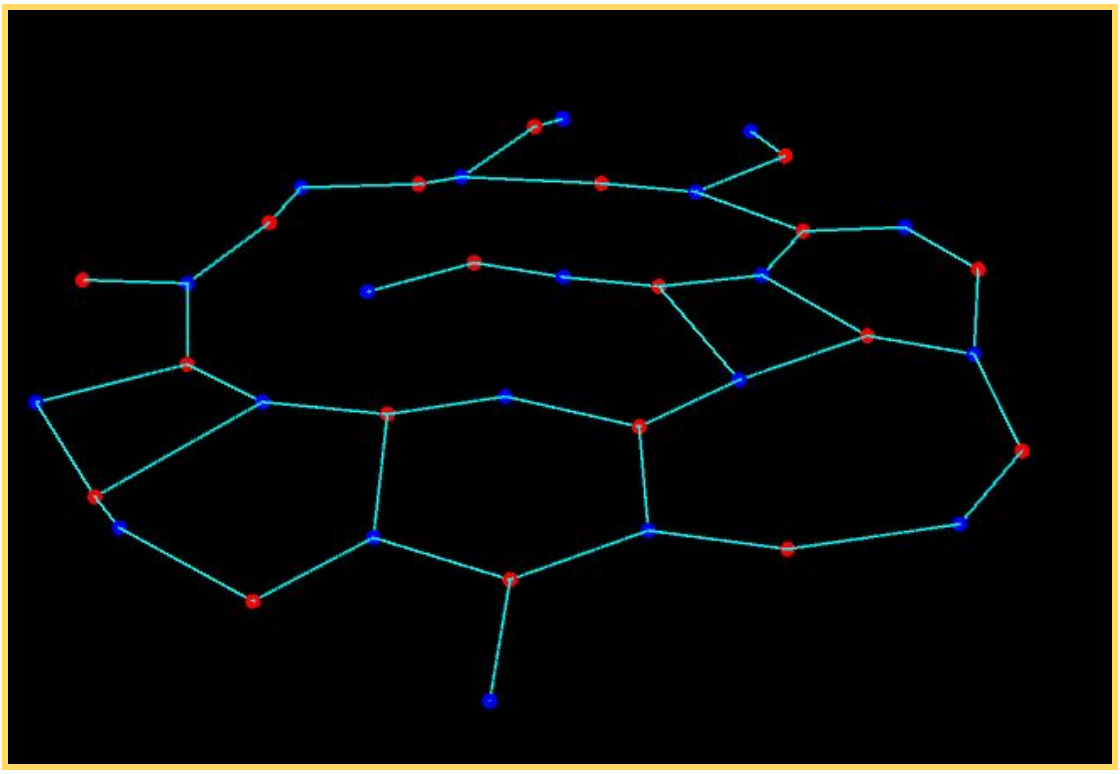
[7] Processing.org: *Processing 3 Graphics Library*

[8] Oracle: *Java 8 programming language*

[9] Google: *Search, Sheets, Docs*

[10] Wikipedia: *Sweep Line Algorithm*

REDUCTION TO PRACTICE



Note: Throughout this section we will use an example where N = 20 and R = 0.40

Part I $O(n * log(n))$

Distributing Nodes

The first part of my implementation utilizes three main data structures: **Vertex, vertexDict, and degreeDict**.

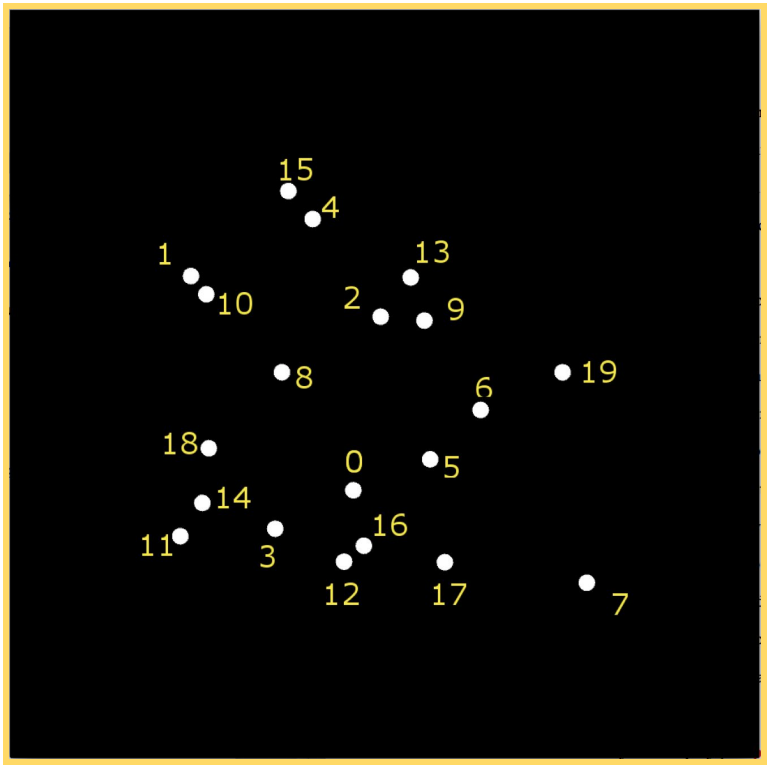
Part I begins in the *setup()* section of the code. To initialize our network, *n* nodes are pseudorandomly distributed on the surface of our graph types using the following equations:

Square: $x = random(0,1)\ y = random(0,1)$

Disk: $a = random(0,1)\ b = random(0,1)$
$x = random(0,1) * cos(2 * \pi * a/b) \quad y = random(0,1) * cos(2 * \pi * a/b)$

Sphere: $\theta = 2 * \pi * random(0,1)\ \phi = acos(\ random(0,1) - 1\ )$
$x = sin(\phi) * cos(\theta) \quad y = sin(\phi) * sin(\theta) \quad z = cos(\phi)$

Let's create an example on a square topology where *N* = 20 and *R* = 0.40. Pseudorandomly distributing our nodes using the above equations gives us the set of nodes below.



As each of these nodes are generated, their positions and ID's are stored in an object of the custom class *Vertex*:

Vertex:

- ***int ID***
- ***int X,Y,Z***
- ***LinkedList neighbors***
- ***int color***

Each vertex is then stored in the *Vertex Dictionary* (henceforth referred to as **vertexDict**). This data structure is the foundation of the program! (An array of integers called **degreeDict** is also initialized in the same loop. More on this later.)

Important: Because Java lacks a public pointer type, the *index* of a vertex in **vertexDict** is the way every vertex is referred to in the program. All data structures concerning vertices utilize these indices to refer back to the **vertexDict**. For example, **degreeDict**, is initialized with the integers *0* through *N*, which will act as a reference to the **Vertex** objects in the **vertexDict**.

The upper bound running time of this process is O(n) as it loops through each node once.

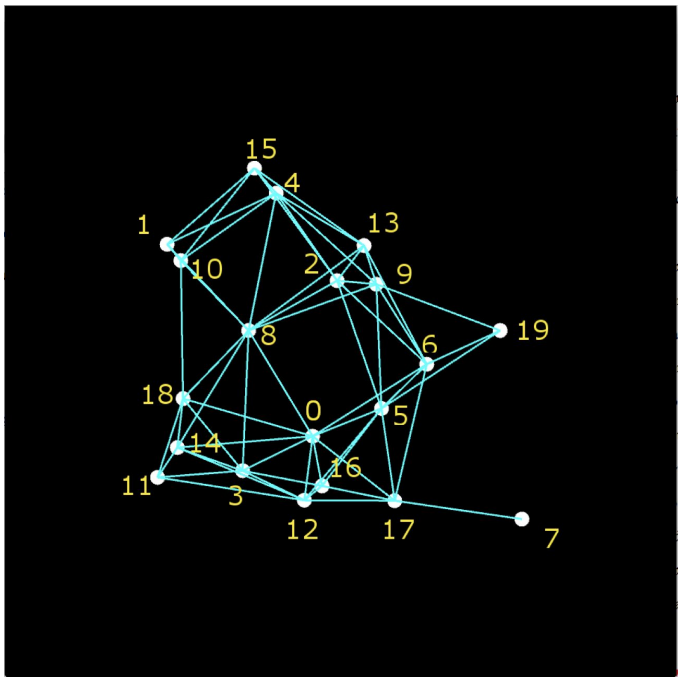
Once we have the **vertexDict** initialized, it's time to determine which vertices are neighbors of each other and store this in our dictionary. Based off the average degree provided as input to the program, we can calculate *R*, the radius for which to connect two points. Based off our topologies, *R* is calculated as:

Square: $\sqrt{AvgDeg / n * \pi}$ *Disk:* $\sqrt{AvgDeg / n}$ *Sphere:* $\sqrt{4 * AvgDeg / n * \pi}$

From there, we use the *Sweep Method* [10] to build our adjacency list, **vertexDict**. First, we use merge sort to sort our vertices based off their *x* coordinates. Because we don't want to change the structure of **vertexDict**, we'll sort the value of **degreeDict** based off the *x* positions of the vertices. (Remember, **degreeDict** only *references* vertices in **vertexDict**, it doesn't hold them.)

We then loop through each index in **degreeDict**, calculating the distances between the vertices. If the distance is less than *R*, we add each vertex to the other's list of neighbors. But because repeatedly calculating distances of points in three dimensional space is an intensive process, we only calculate those distances that fall within a certain X position. (Because we already sorted the vertices by X position, this means looping backwards in the array, creating a "sweeping" motion.) This process happens in $O(n * log(n))$ time. From our example, the following **vertexDict** is created:

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



The graph generated in our example from the vertexDict (left).

The combination of using merge sort and sweep method makes this part run in $O(2 * n * log(n))$ time. Therefore, Part I happens in $O(n + 2 * n * log(n))$ or $O(n * log(n))$ time complexity.

Part II $O(n^2)$

This part of the code is the *smallest last vertex ordering* and *greedy coloring* sections. The most important data structure in Part II is the adjacency list, **colorDict**, which will help us determine the colors of each vertice. Like **vertexDict**, **colorDict** is and adjacency list. However, the latter is implemented as an array of type *LinkedList*, instead of our custom class, **Vertex**. The first value in each linked list will be the ID of a vertex according to the order in **degreeDict**. It is therefore a reduced version of **degreeDict** whose values will be determined based off a "deletion" process.

This process begins by once again sorting **degreeDict** (in reverse order) by the degrees of the vertices in **vertexDict**. Because **degreeDict** holds pointers to vertices in **vertexDict**, the following data structure is created from our example (bottom left).

Note: It is possible to implement smallest last vertex ordering in constant time [1], however, my implementation uses merge sort to implement the process in $O(log(n))$.

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

It's now time to initialize **colorDict** in $O(n)$ *time complexity)* with only the index of our vertices (below):

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

It is now time to color the graph by filling in **colorDict** based off our deletion process. Starting with the smallest degree vertex, each vertex in **degreeDict** is marked "deleted". Then, for each of its neighbors stored in **vertexDict**, if the neighbor hasn't been deleted, the neighbor is added to the linked list at the corresponding spot in **colorDict**. The first step of this process is illustrated from our example below:

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

The smallest degree vertex being deleted from **degreeDict**.

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

The vertex is then marked as deleted in the **vertexDict**.

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

Because (according to **vertexDict**) the vertex *17* is a non-deleted neighbor of *7*, the vertex *17* is added to the linked list at the last index of **colorDict** (left). *(Note: 17 eventually gets deleted by Future Trunks.)*

This process continues with the vertex *19* being deleted next. Once 19 is deleted, *9, 5, and 6* are added to **colorDict** at the second to last index. This part of the algorithm happens in $O(k * n)$ time complexity, where *k* is equal to the maximum degree of a vertex when it's deleted.

While we are deleting nodes in our algorithm, we check to see if the remaining nodes form a *terminal clique*. This happens by determining if the current set of non-deleted vertices is connected to *n-1* distinct, non-deleted vertices (in $O(n^2)$ time complexity).

This algorithm finally comes to an end once vertex *8* has been deleted:

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

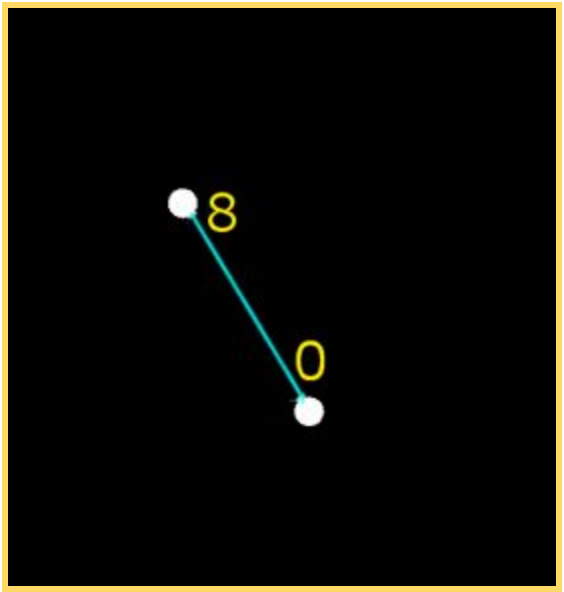
*All vertices have been deleted from **degreeDict**.*

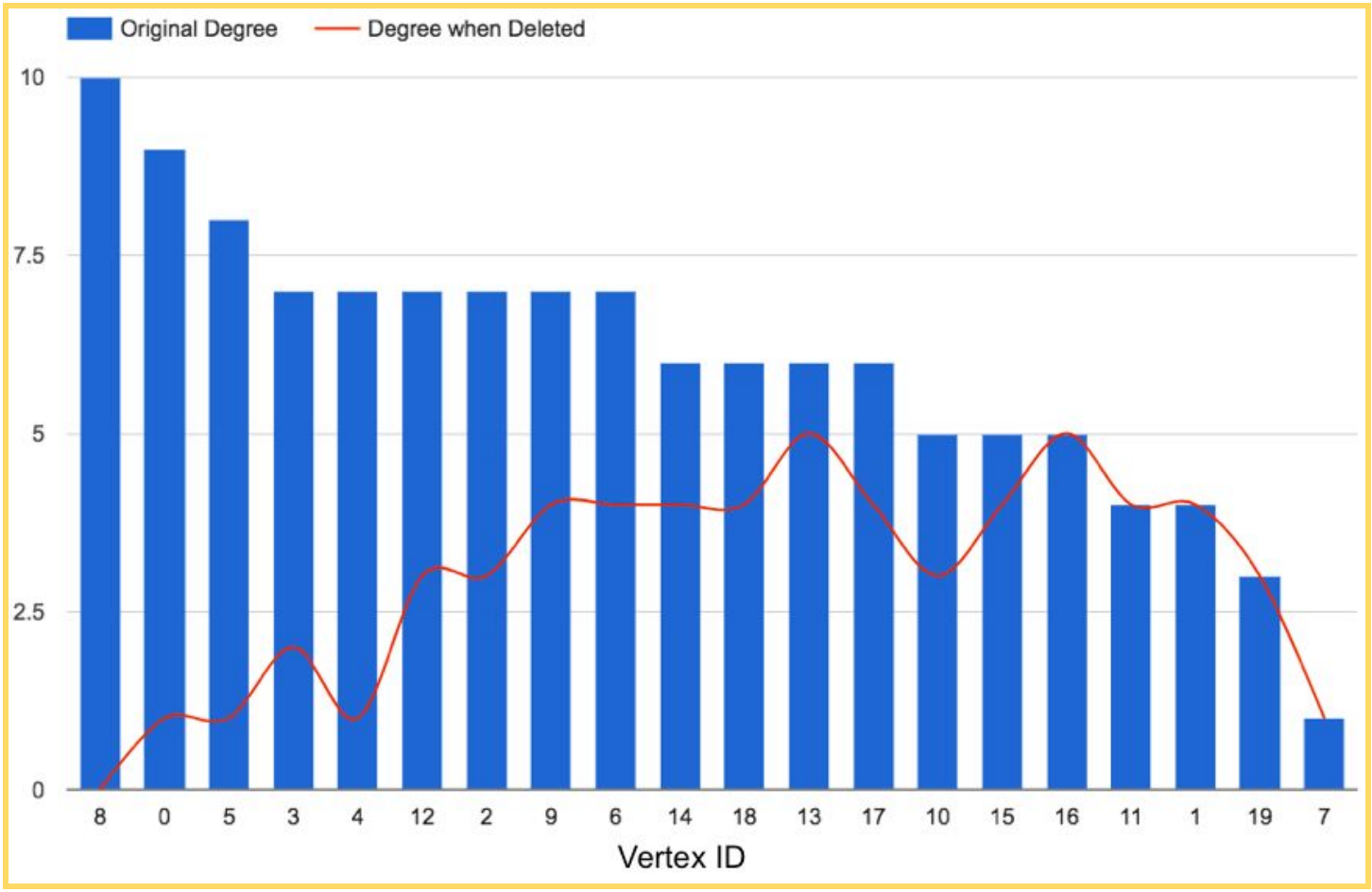
Vertex	Neighbors									

*All vertices have been deleted from **vertexDict**.*

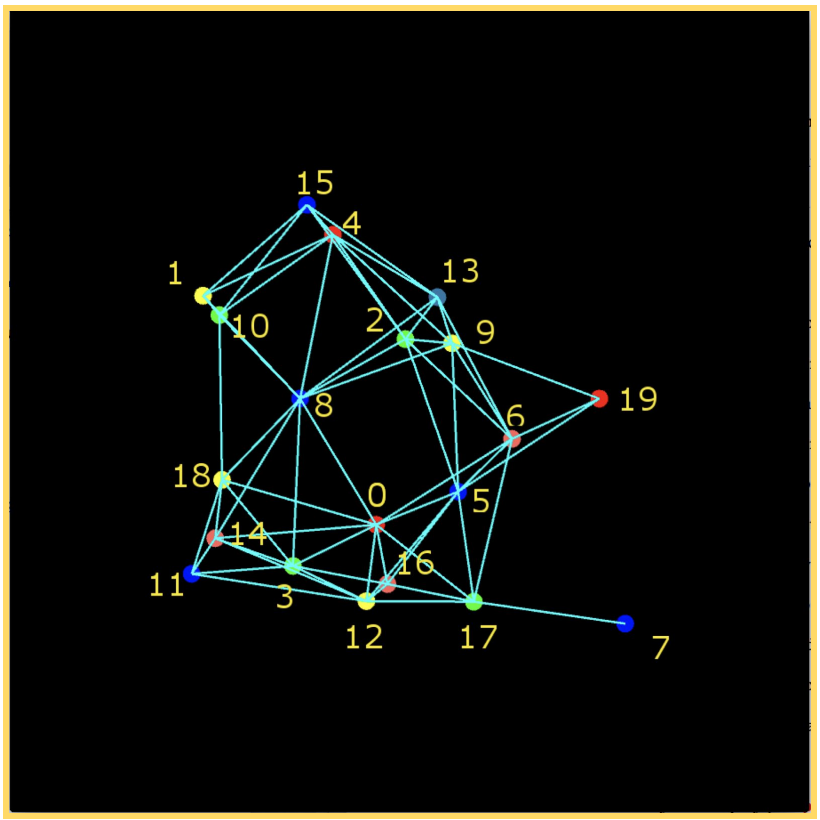
Vertex	Neighbors				
8					
0	8				
5	0				
3	0	8			
4	8				
12	5	0	3		
2	5	8	4		
9	5	8	4	2	
6	0	2	9	5	
14	0	12	8	3	
18	0	8	3	14	
13	6	9	8	4	2
17	6	12	0	5	
10	4	8	18		
15	13	2	4	10	
16	17	5	3	12	0
11	12	3	18	14	
1	4	15	8	10	
19	9	5	6		
7	17				

colorDict is eventually populated with the appropriate values (left), and a terminal clique is also determined with the last two vertices *0* and *8*:





If we keep track of vertices while we're deleting them and their original degrees, we can see the maximum degree when deleted, *five*, comes out to be exactly half of the maximum original degree, *ten*. *(See graph above.)*

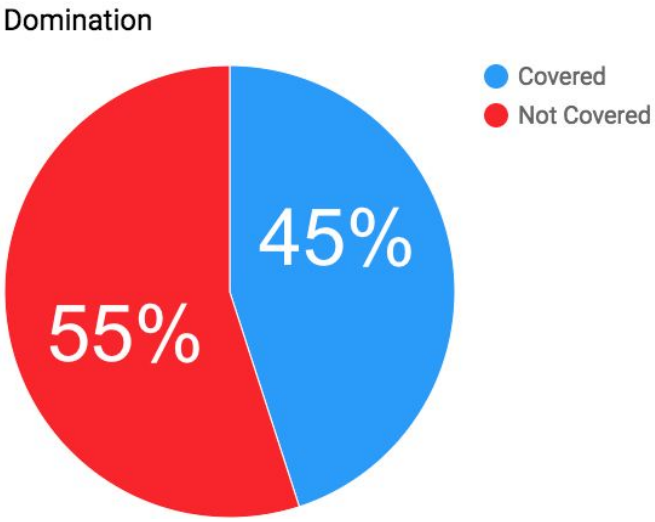


To select colors, we loop through **colorDict** and assign the current color (the first element in the linked list) the "smallest" color that's not in the current linked list. This happens in $O(k^2 * n)$ time complexity where *k* is the size of the largest linked list in **colorDict**. Each time a color is selected, it's added to **colorCount**, a hashmap where key is the color and value is the number of times the color occurs. The vertex is then colored accordingly.

Part III

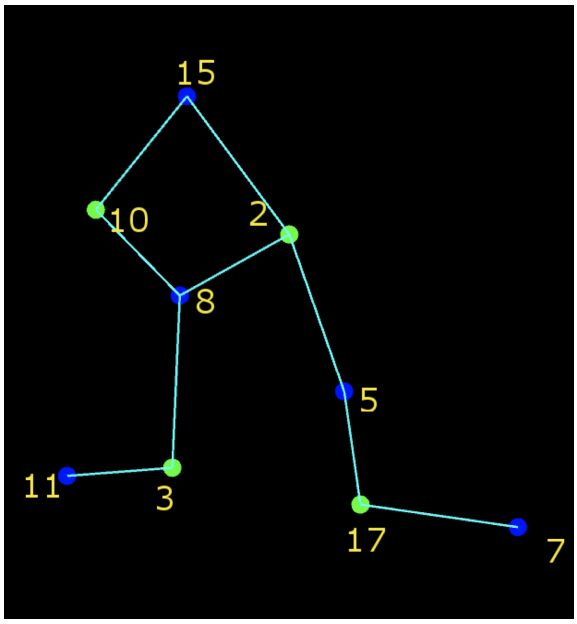
$O(n * log(n))$

Now we will determine the two largest *bipartite subgraphs*. First we sort **colorCount** by value in $O(n * log(n))$. Then, we iterate through colorCount to determine the four largest colors and print the color distribution data for our graphs. Using the *nCr* method, we determine the six different combinations of our four largest colors and store those combinations in a 2D array, **colorCombos**. *(Note: My implementation also works for less than four total colors.)*

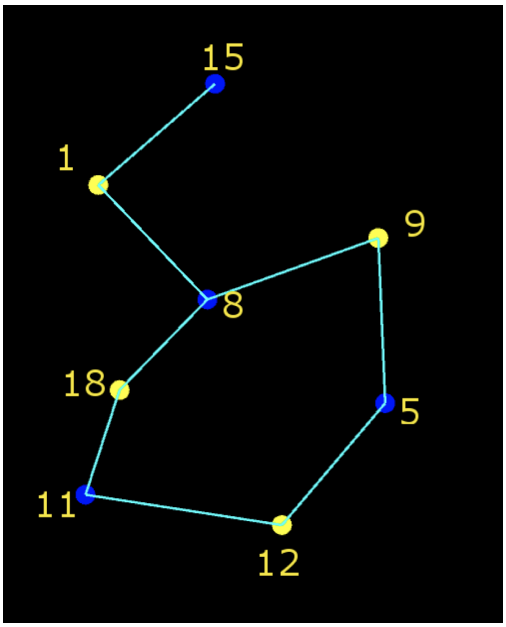


For each color combo, we do a breadth-first search till all nodes of the combination have been visited. Whichever BFS visits the largest number of vertices is considered the largest "backbone" of that color combination. This process is repeated until the two largest backbones of all the combinations are found. These nodes are stored as the starting node of the BFS in **largestStarterNodes**. This process happens in $O(k * j * |V| + |E|)$ where *k* is the number of color combinations from *nCr* and *j* is the number of separate components searched through.

The following bipartite subgraphs were generated from our example:



Largest bipartite subgraph with V = 9

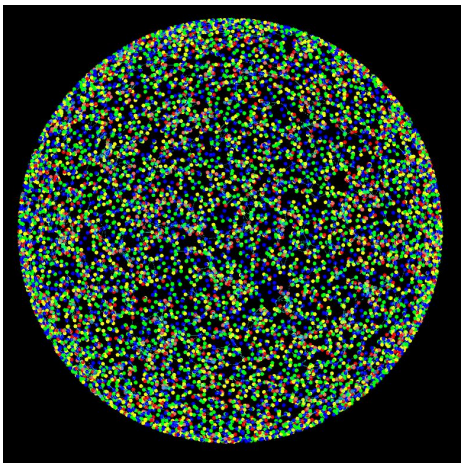


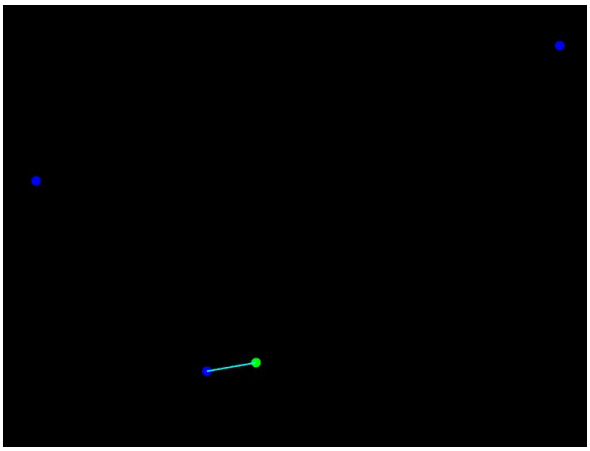
Second bipartite subgraph with V = 8

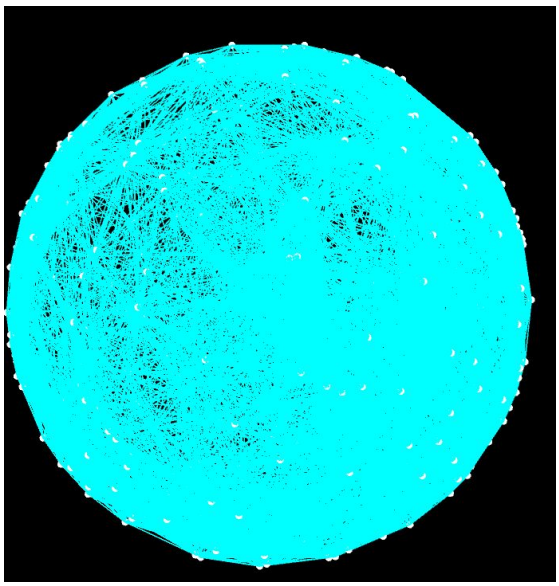
Algorithm Effectiveness

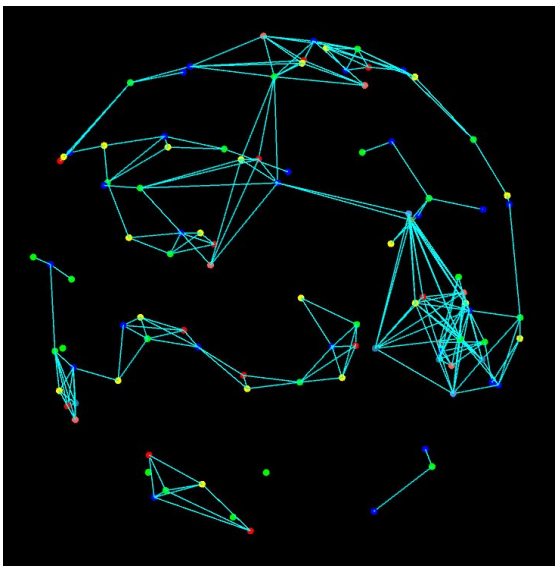
Benchmark				Time to Complete (ms)			
Benchmark #	N	Avg. Degree	Distribution	Adjacency List	Coloring	Backbones	Total
1	1000	32	Square	22	5	8	35
2	4000	64	Square	66	33	49	148
3	16000	64	Square	319	96	149	564
4	64000	64	Square	1768	404	682	2854
5	64000	128	Square	2539	1894	2071	6504
6	4000	64	Disk	56	32	51	139
7	4000	128	Disk	71	62	81	214
8	4000	64	Sphere	83	40	61	184
9	16000	128	Sphere	621	204	275	1100
10	64000	128	Sphere	4344	1892	2063	8299

This table shows the time it took milliseconds for each of our benchmarks (not including outputting to files). As you can see, the biggest determinant of time was the average degree input to the program. My implementation is written to be as dynamic as possible and can take a variety of input sizes from large to small. A weakness of my implementation is the fact that it does not occur in linear time.









Results, Summary, & Display

Benchmark	N	R	M	Min Deg	Avg Deg	Real Avg Deg	Max Deg
1 - Square	1000	0.10	14668	11	32	29	27
2 - Square	4000	0.07	120289	20	64	60	83
3 - Square	16000	0.04	496949	12	64	62	67
4 - Square	64000	0.02	2017478	17	64	63	80
5 - Square	64000	0.03	4004023	33	128	125	134
6 - Disk	4000	0.13	121837	15	64	60	32
7 - Disk	4000	0.18	237212	44	128	118	127
8 - Sphere	4000	0.25	127900	40	64	63	68
9 - Sphere	16000	0.18	1024134	89	128	128	119
10 - Sphere	64000	0.09	4095340	89	128	127	127

Benchmark	Max Deg when Deleted	Colors	Largest Color Size	Terminal Clique Size	N of Largest Backbone	M of Largest Backbone	Backbone Domination %
1 - Square	31	23	126	1	126	125	12.60%
2 - Square	63	38	276	5	276	275	6.90%
3 - Square	71	42	985	1	985	984	6.16%
4 - Square	70	41	3807	3	3807	3806	5.95%
5 - Square	135	73	2165	1	2165	2164	3.38%
6 - Disk	65	39	277	3	277	276	6.93%
7 - Disk	120	71	152	1	152	151	3.80%
8 - Sphere	64	39	250	2	250	249	6.25%
9 - Sphere	128	73	540	17	540	539	3.38%
10 - Sphere	131	75	2103	1	2103	2102	3.29%

The table above shows a standard data set for each randomized benchmark. It includes information on the degrees, color classes, and backbones of the vertices.

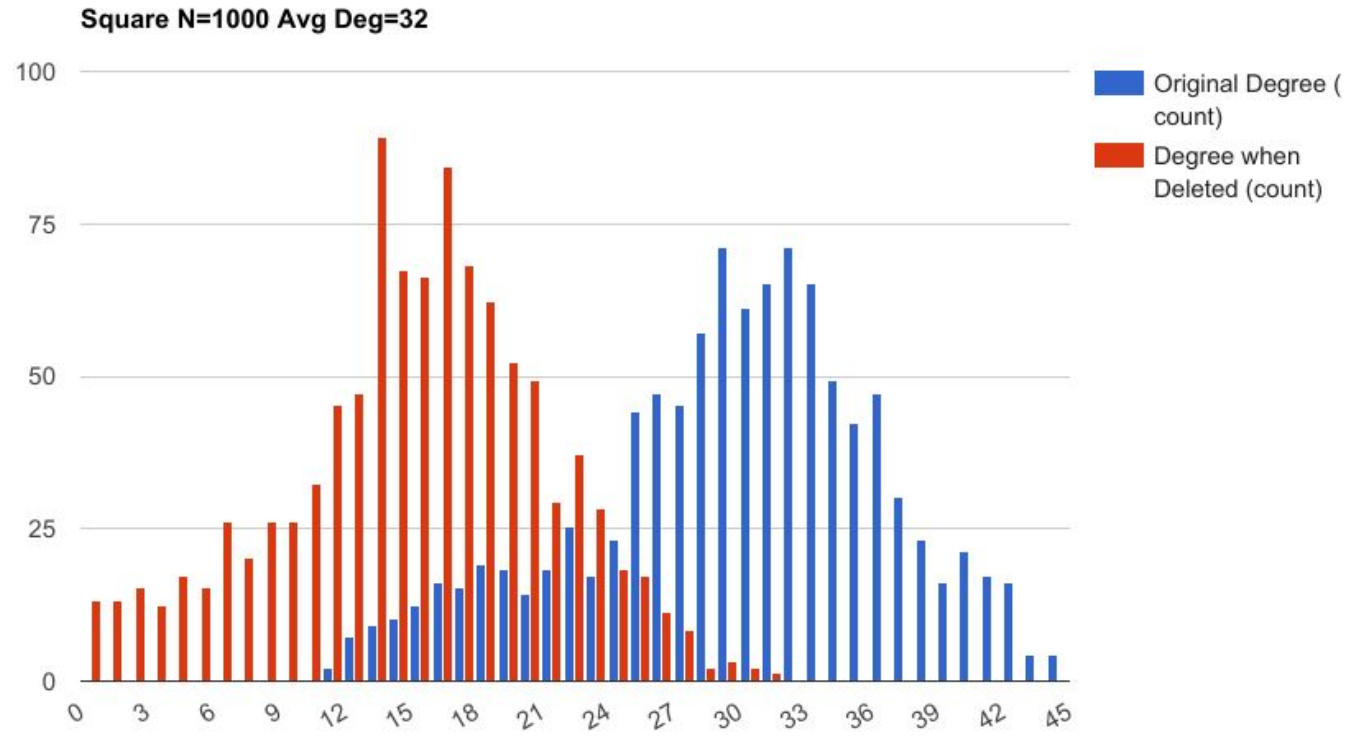
This section will now show a *sequential coloring plot* and *color class size distribution* chart for each benchmark. At the end of this section will be displays of the largest and second largest bipartite subgraphs of each benchmark. This data was calculated by outputting the appropriate values to CSV files. Then, the data files were imported to Google Sheets [9].

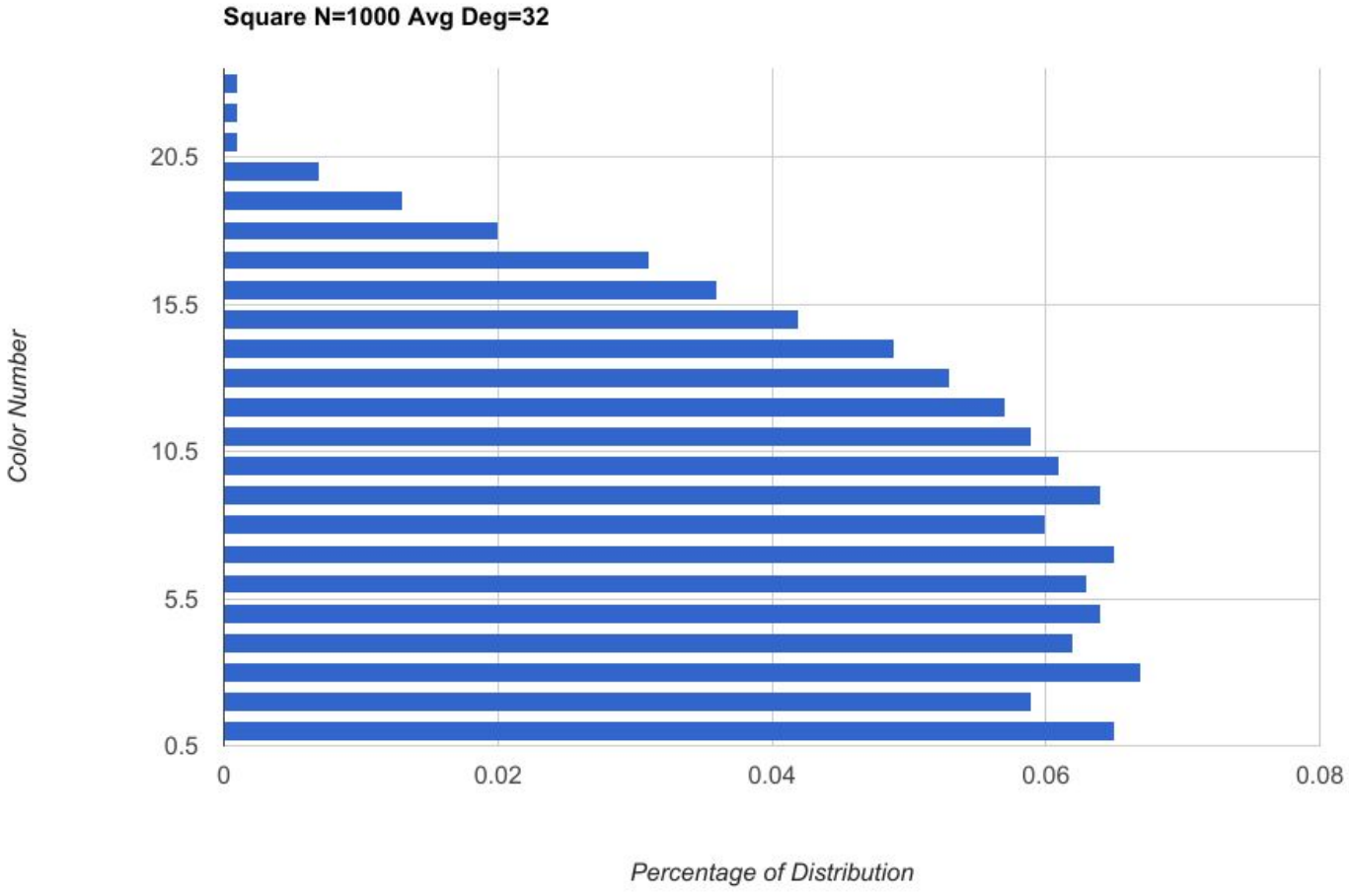
The data below shows a very consistent distribution of degrees and colors despite variations in the number of vertices, the average degree, and the distribution type. Let these charts be a testament to the strength and adaptability of my implementation.

Thanks for reading,

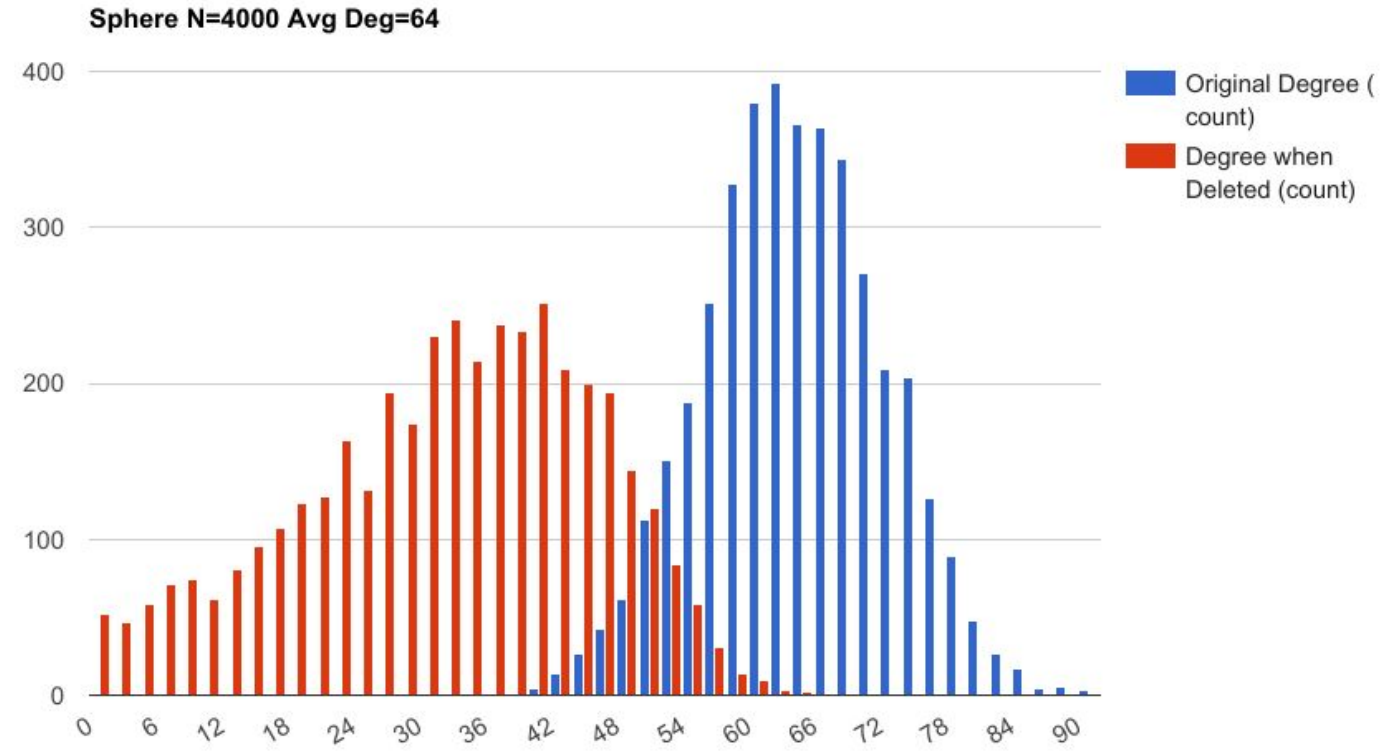
Eric Smith

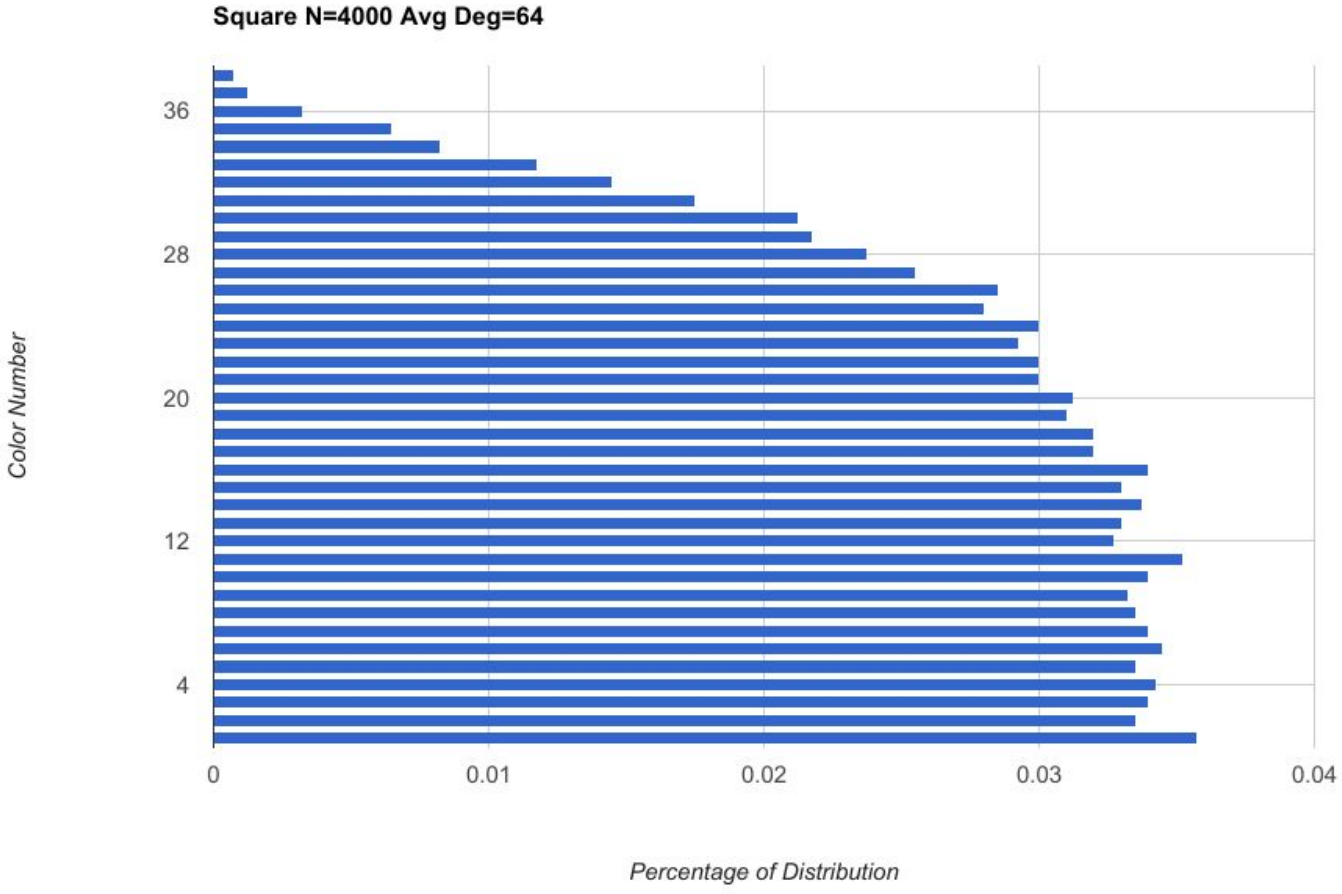
N = 1000 *Avg Deg = 32* *Distribution = Square*



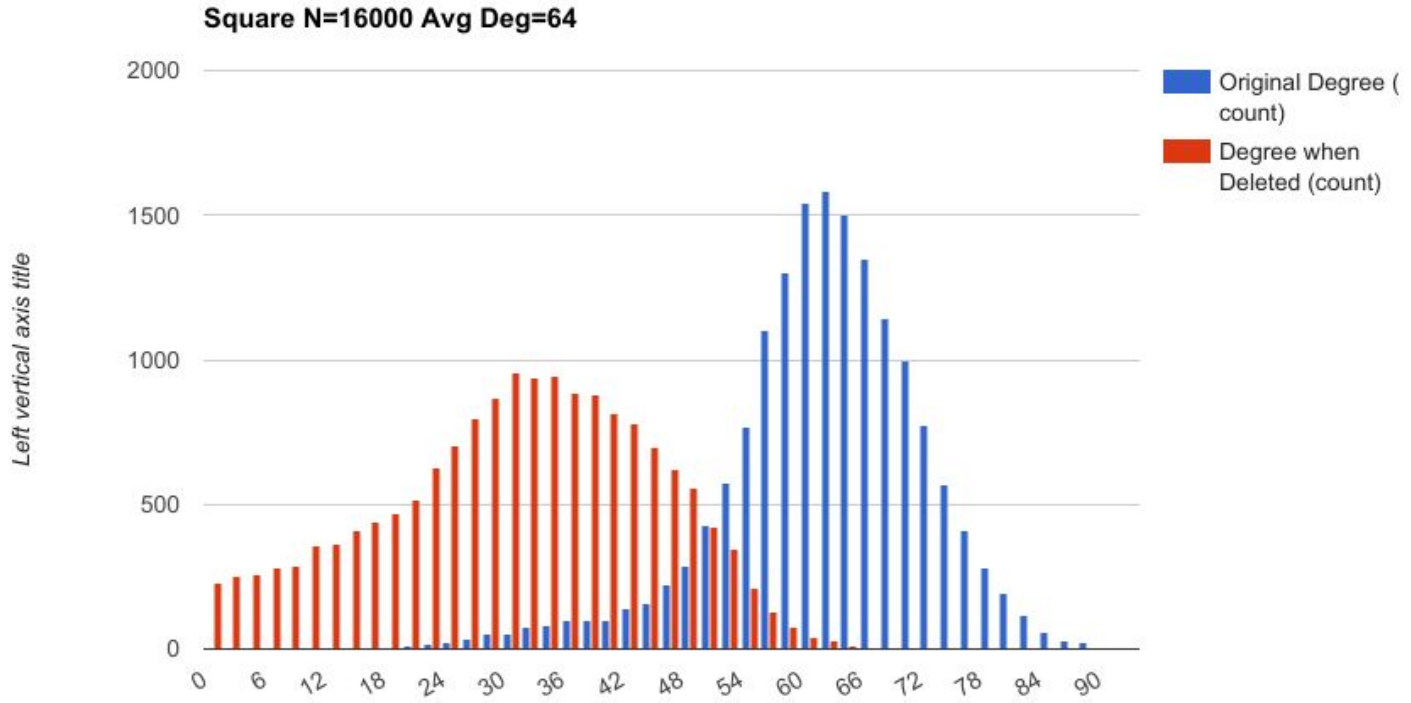


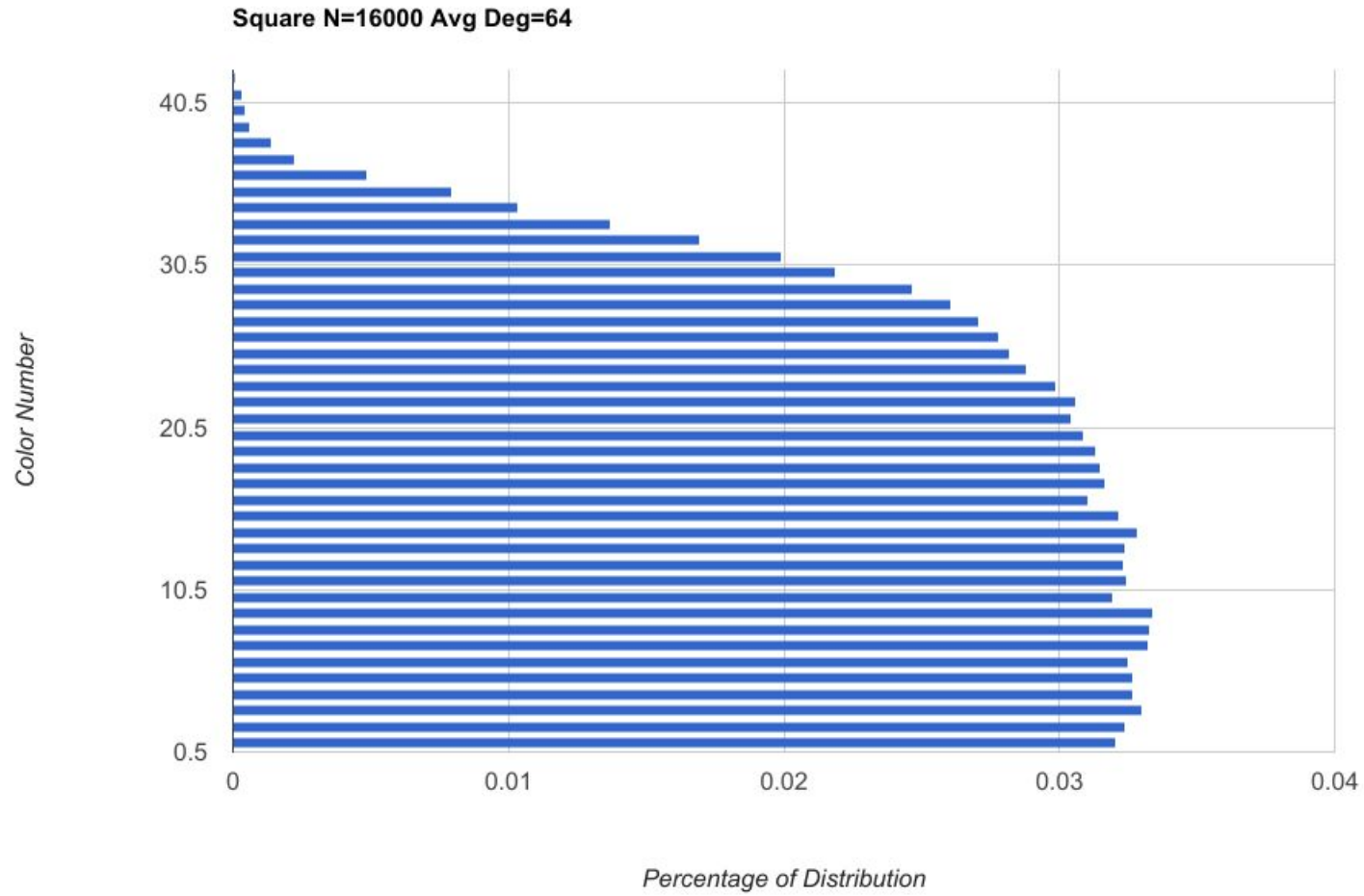
N = 4000 *Avg Deg = 64* *Distribution = Square*



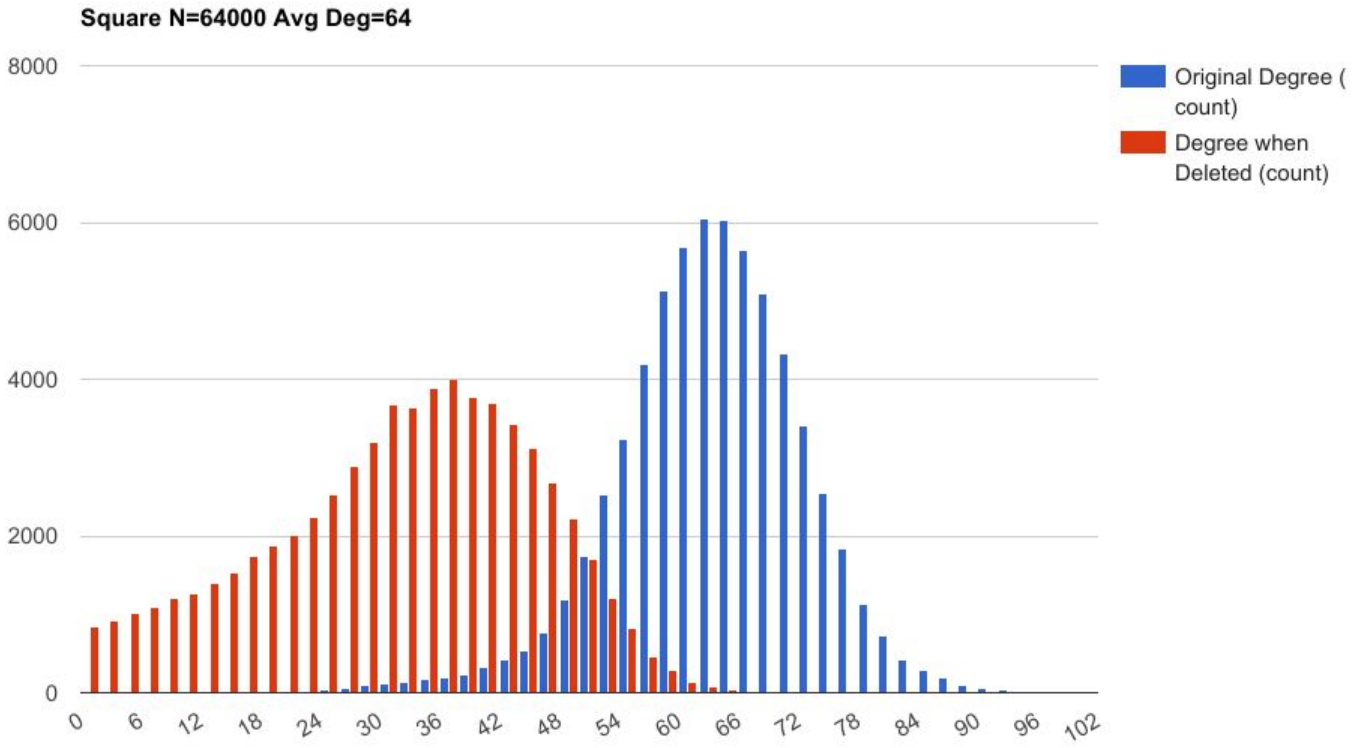


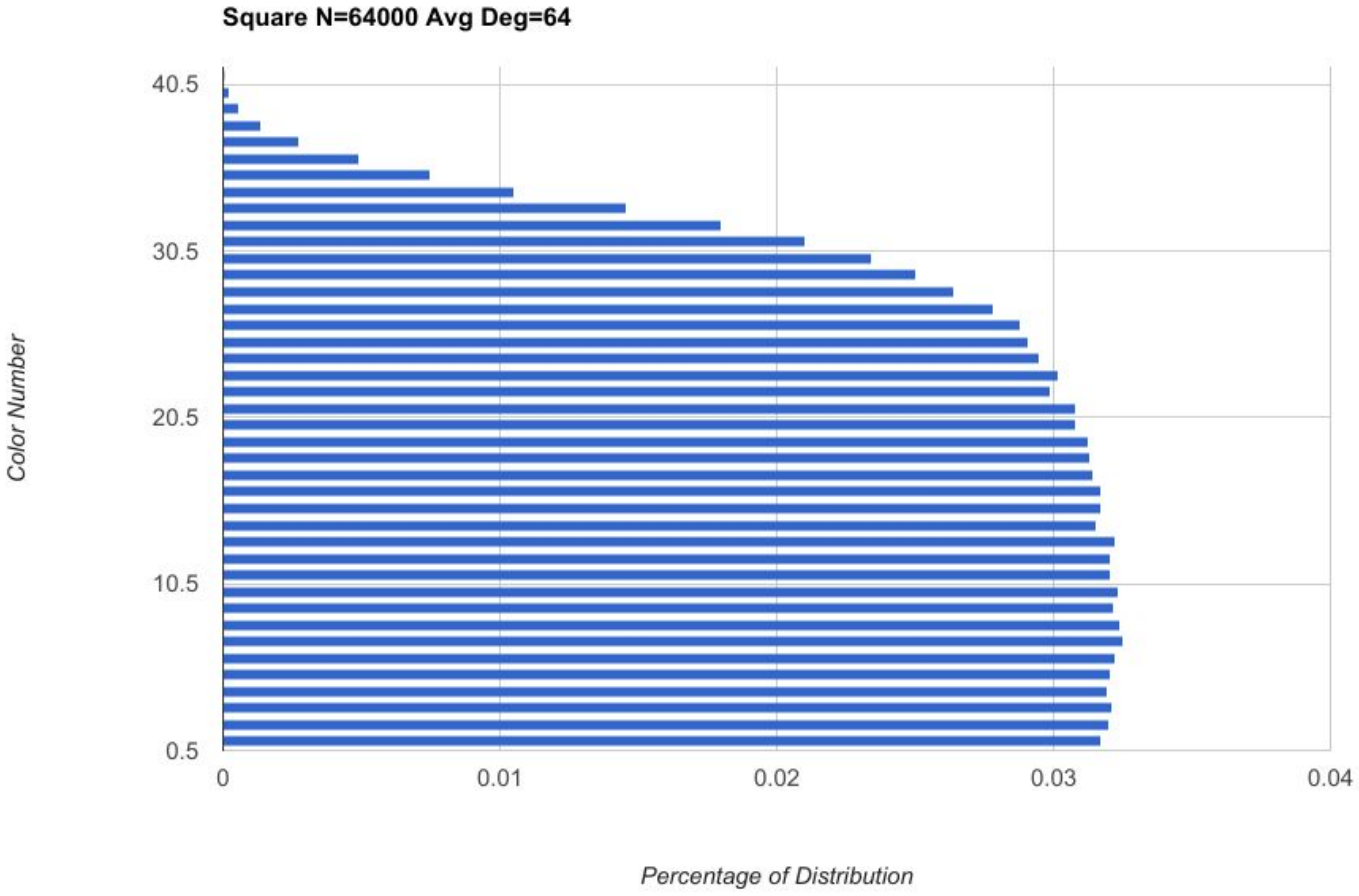
N = 16000 *Avg Deg = 64* *Distribution = Square*



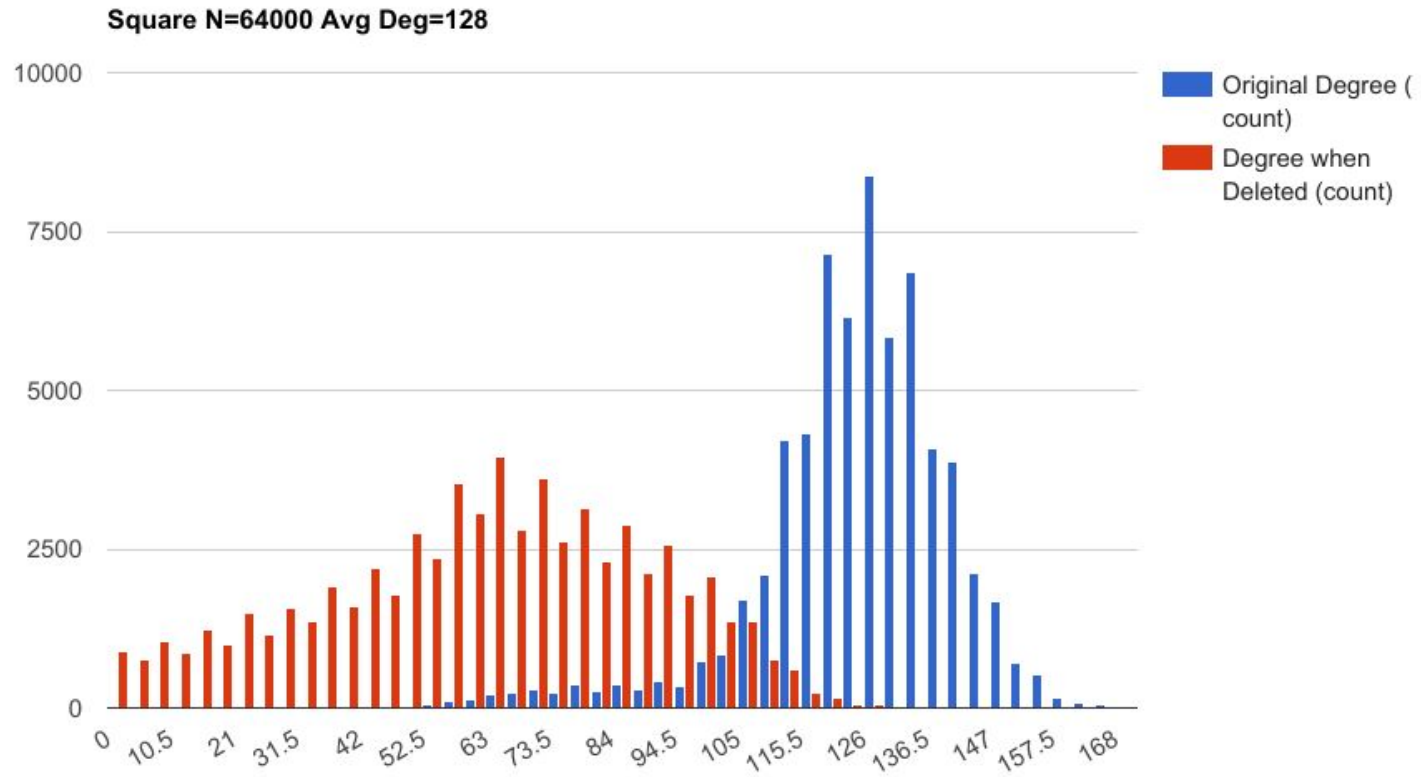


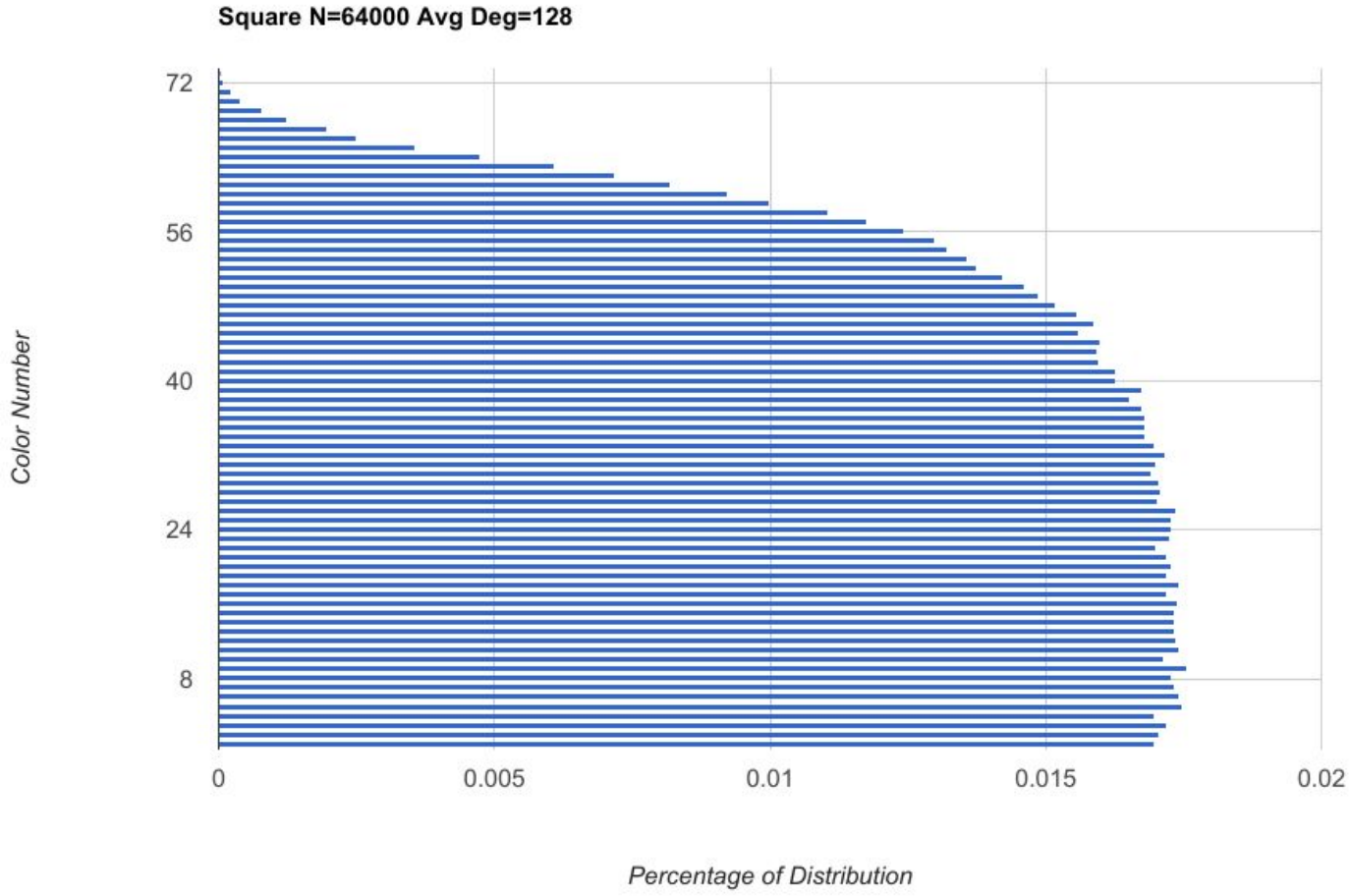
N = 64000 *Avg Deg = 64* *Distribution = Square*



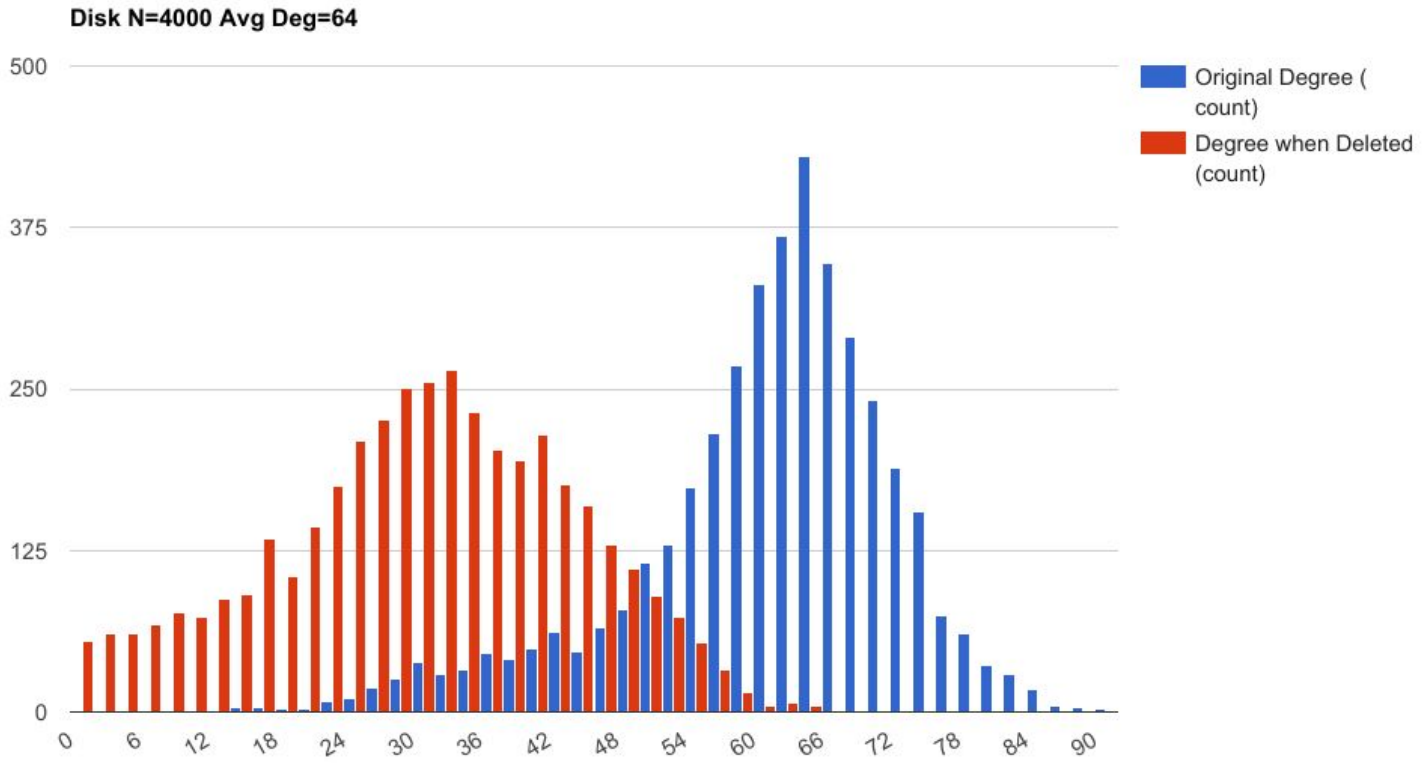


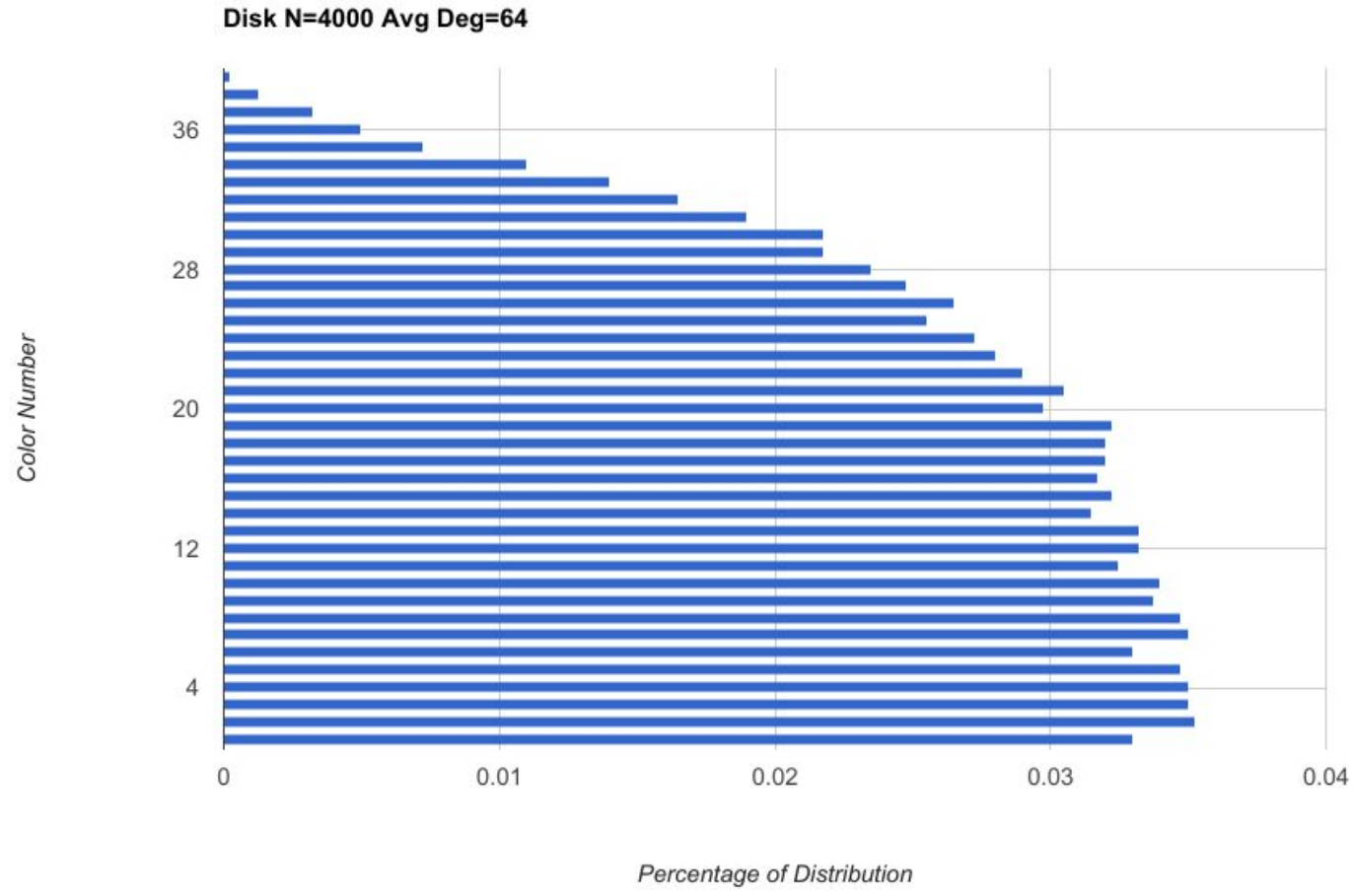
N = 64000 *Avg Deg = 128* *Distribution = Square*



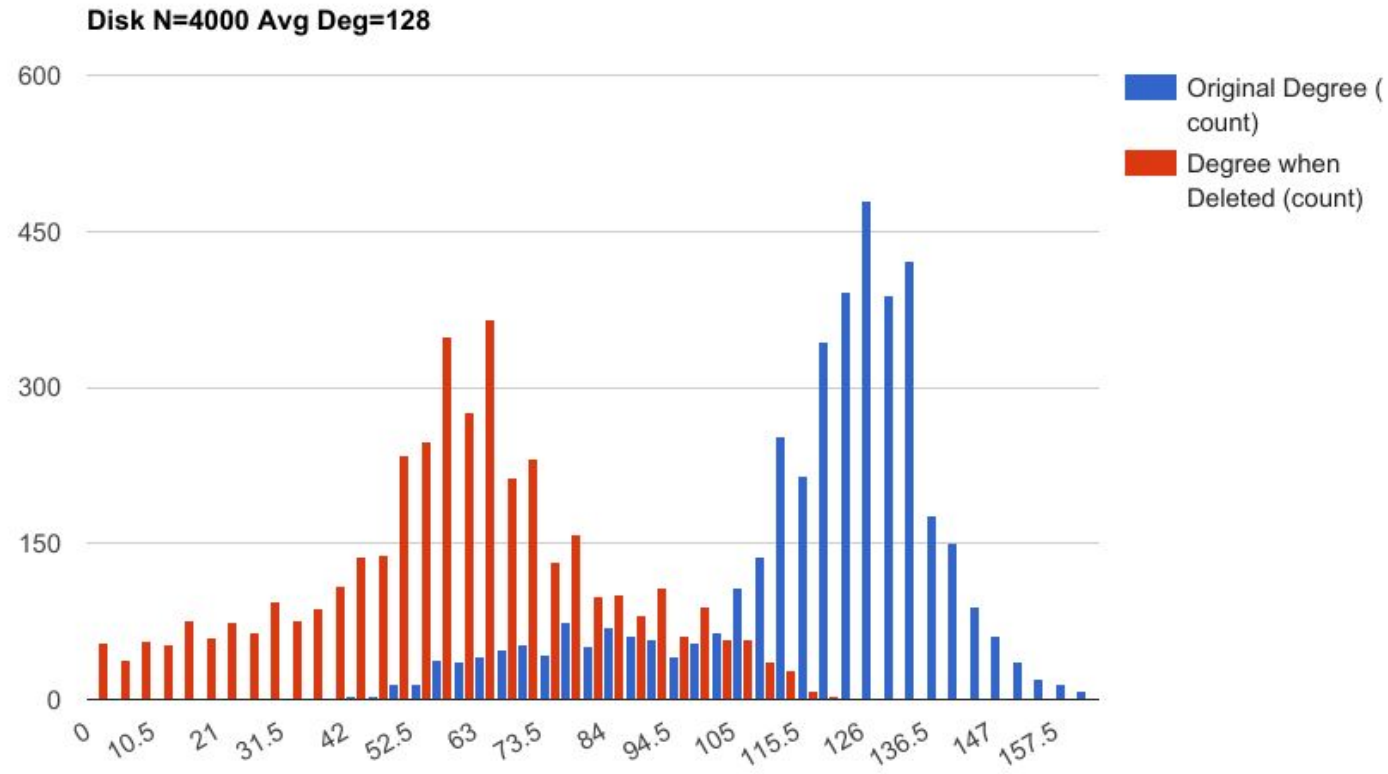


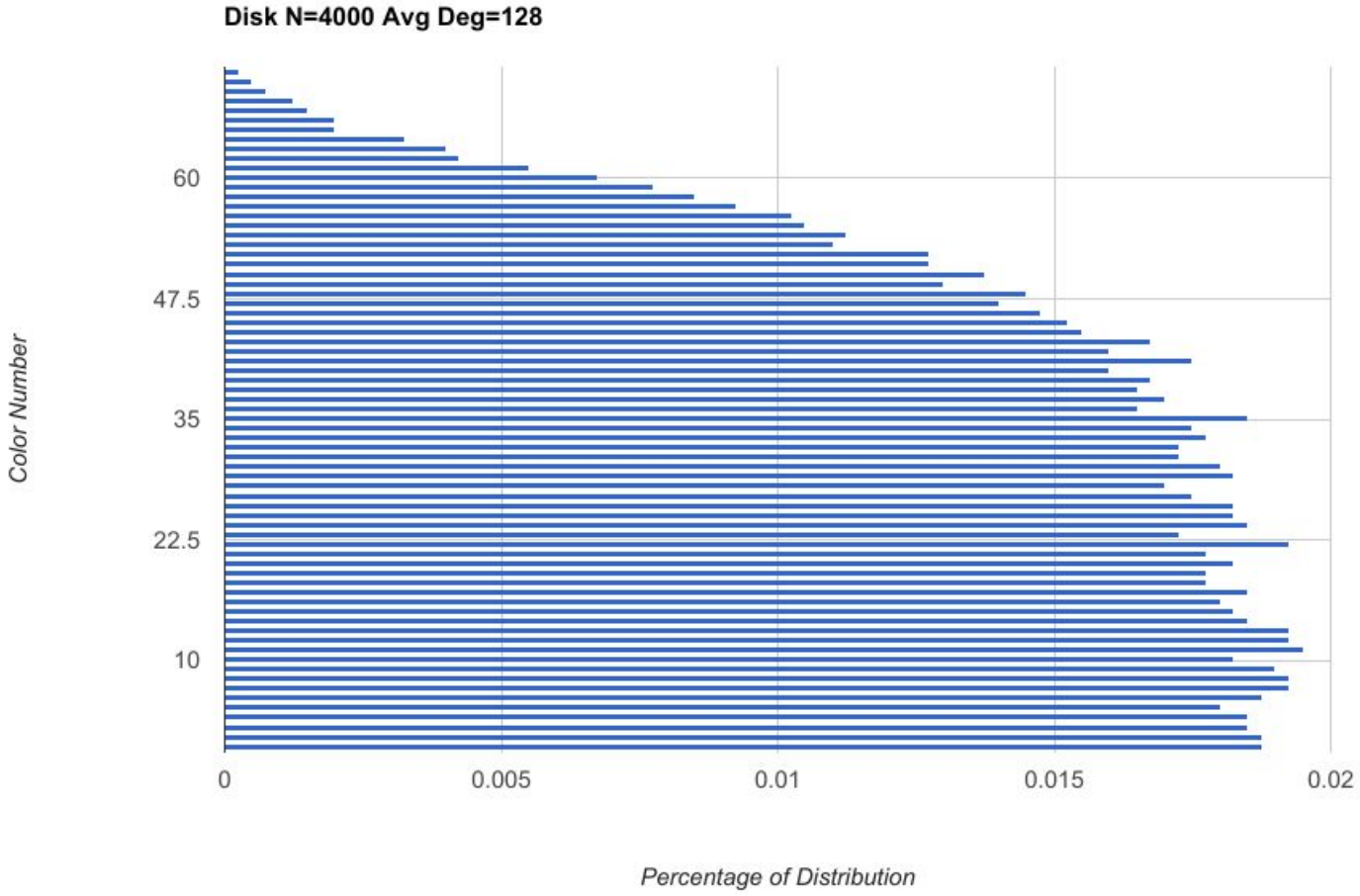
N = 4000 *Avg Deg = 64* *Distribution = Disk*



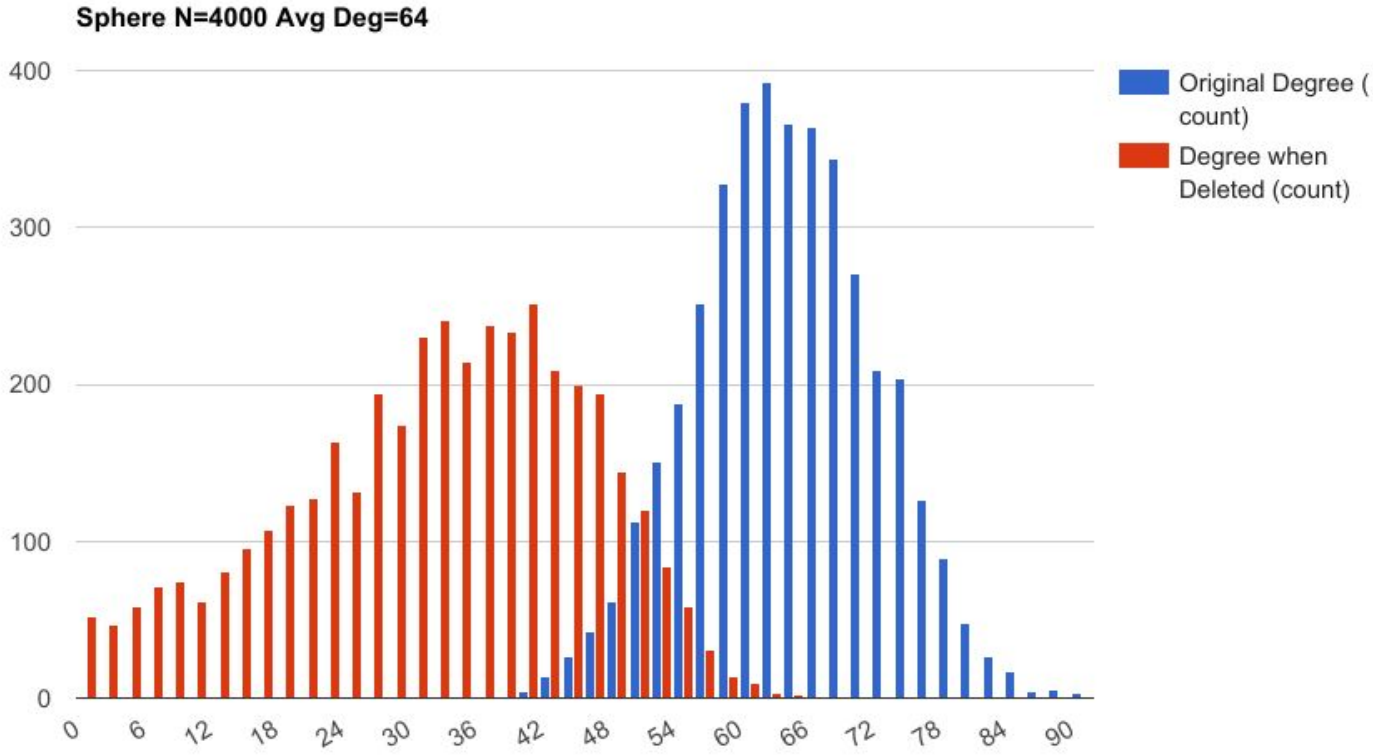


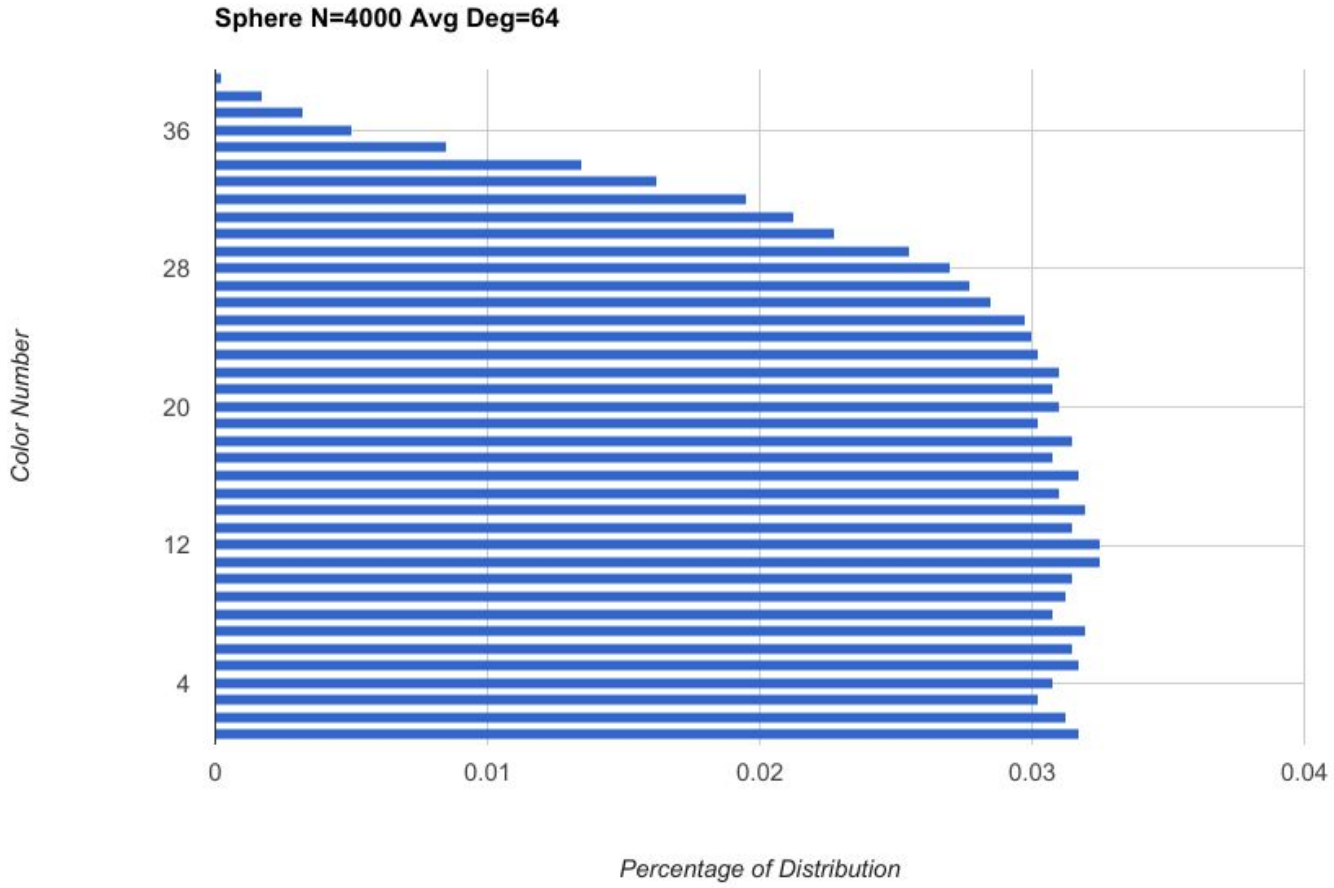
N = 4000 *Avg Deg = 128* *Distribution = Disk*



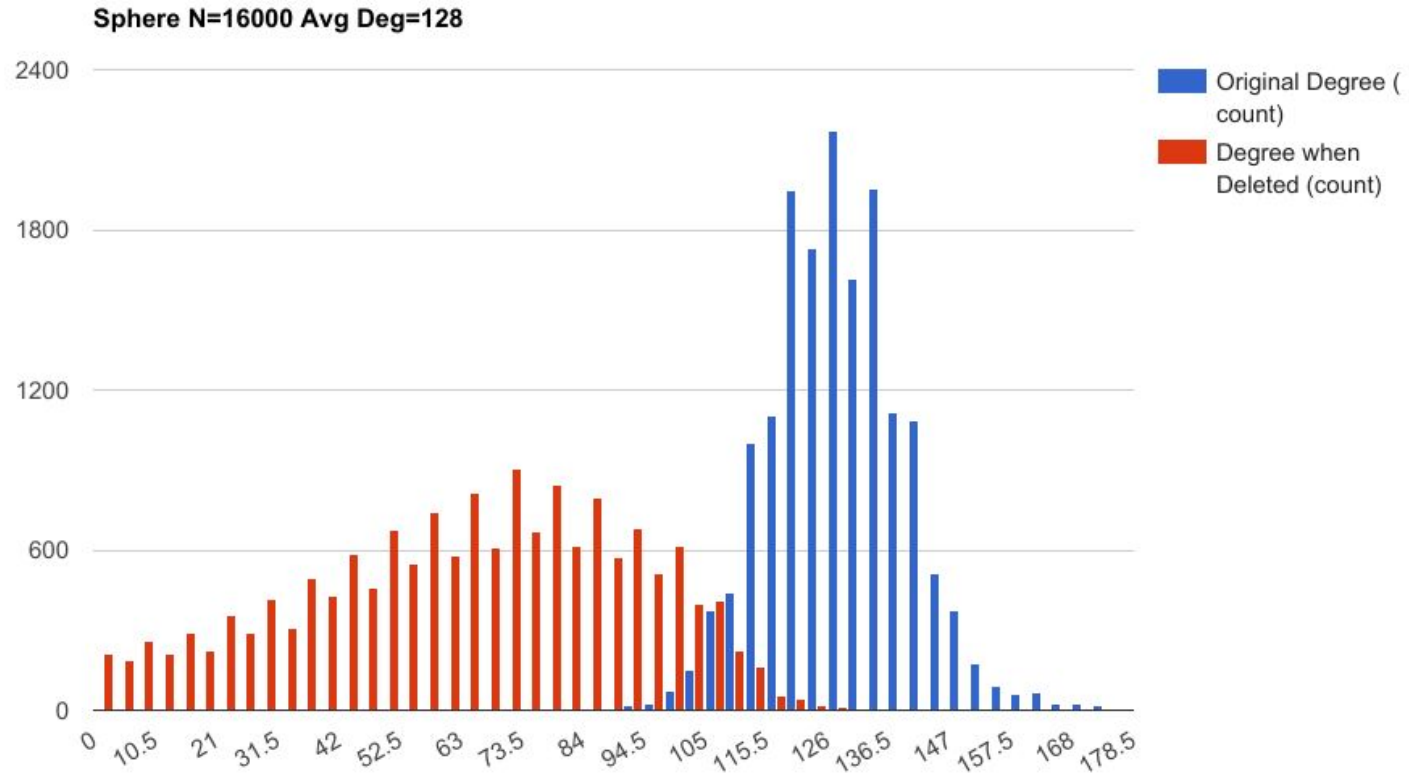


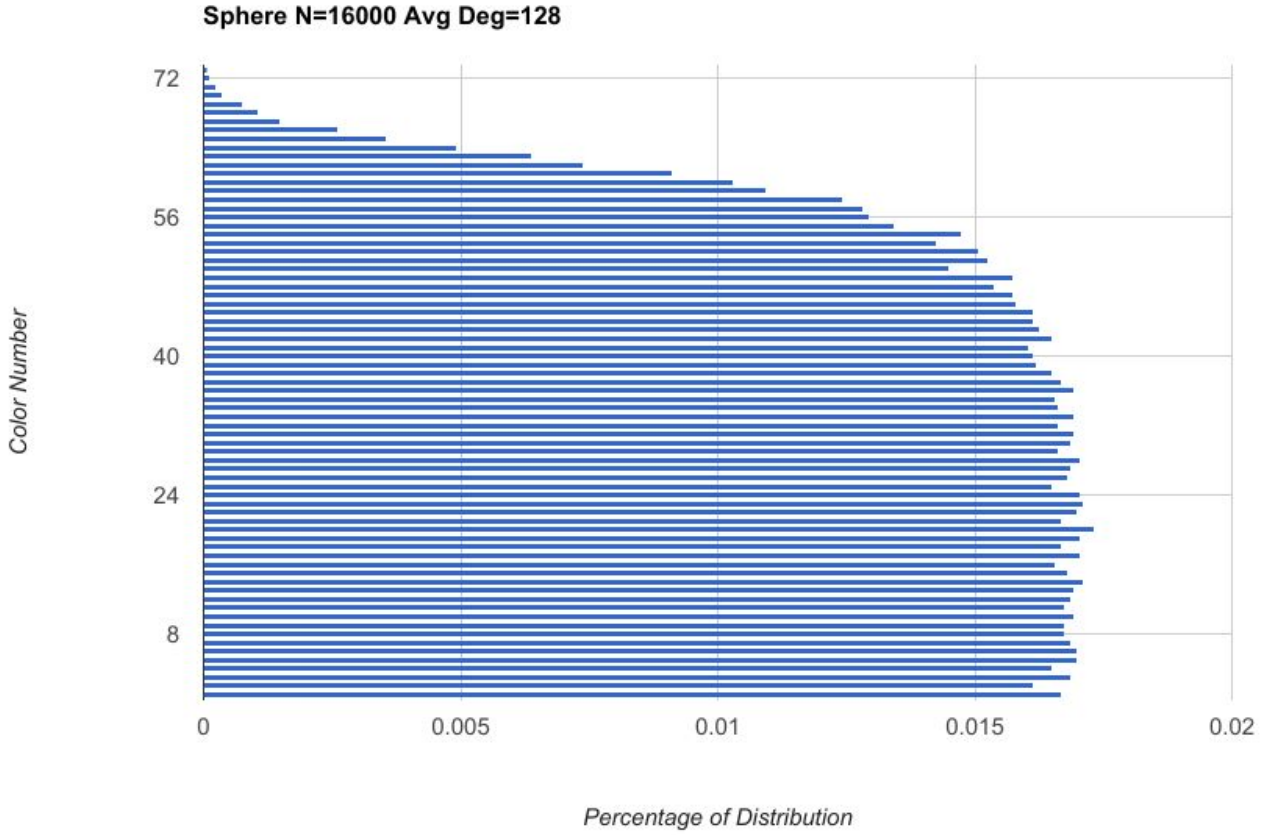
N = 4000 *Avg Deg = 64* *Distribution = Sphere*



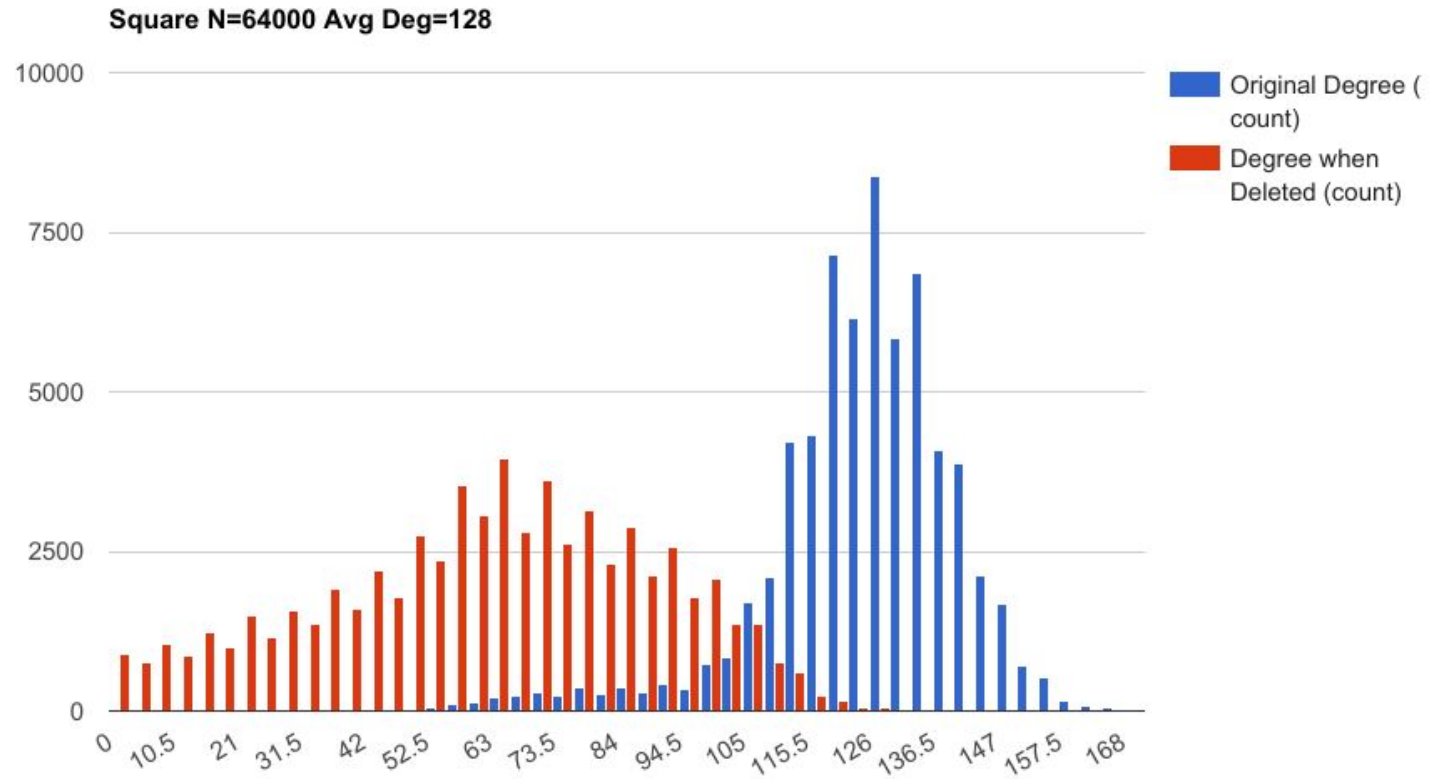


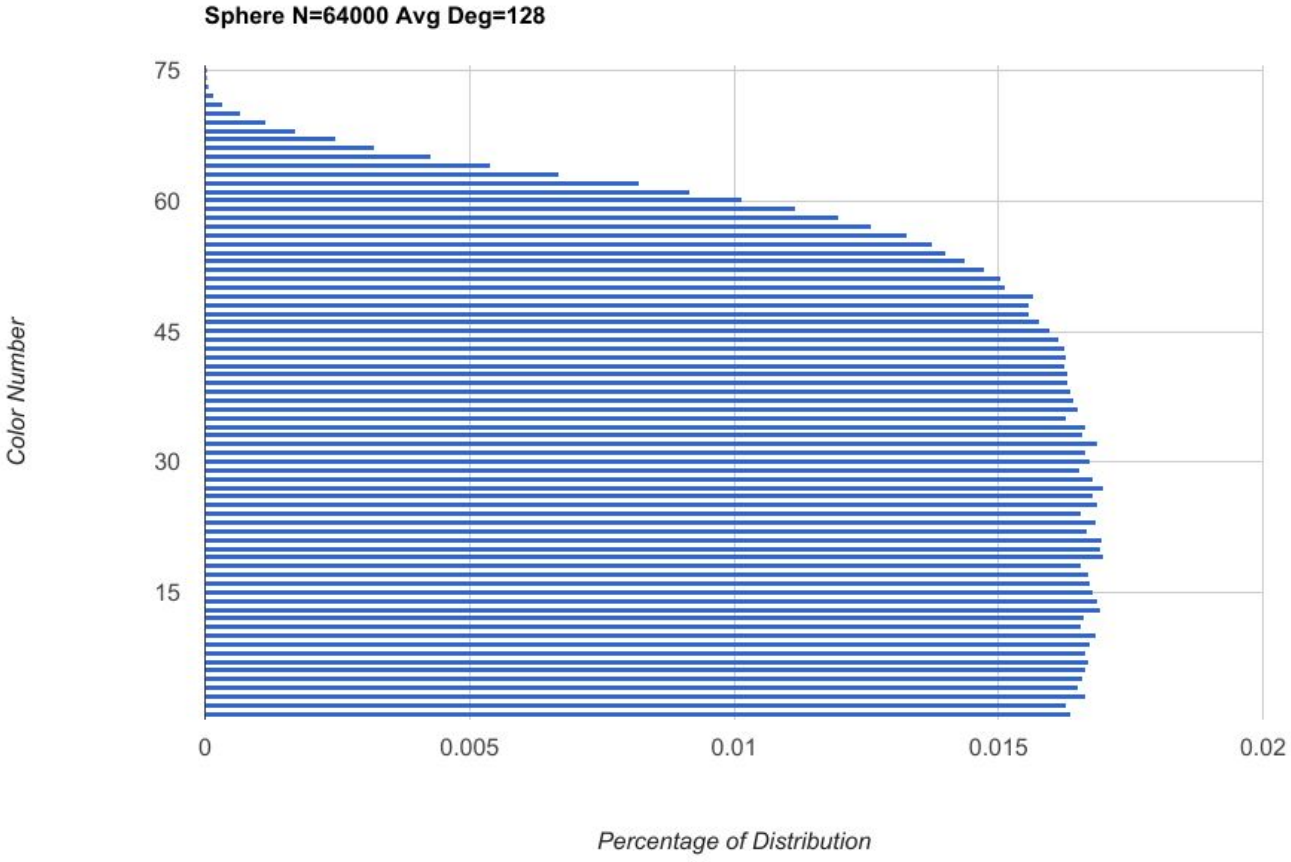
N = 16000 *Avg Deg = 128* *Distribution = Sphere*





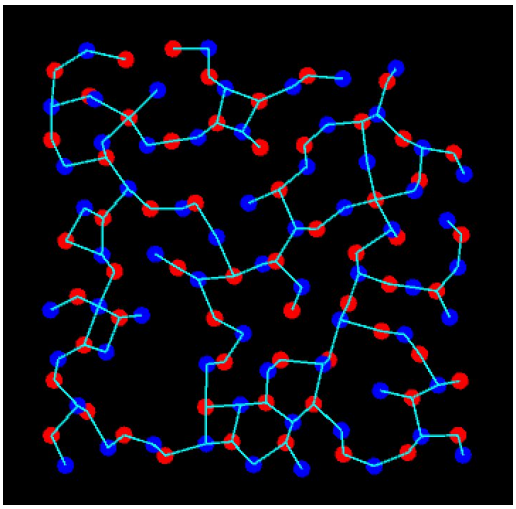
N = 64000 *Avg Deg = 128* *Distribution = Sphere*



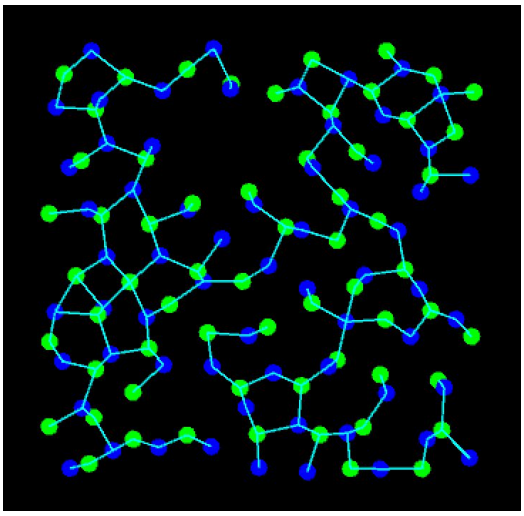


Backbones

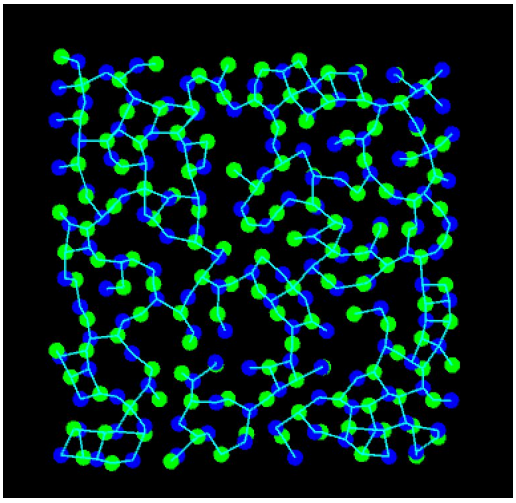
Benchmark 1 - Largest Subgraph



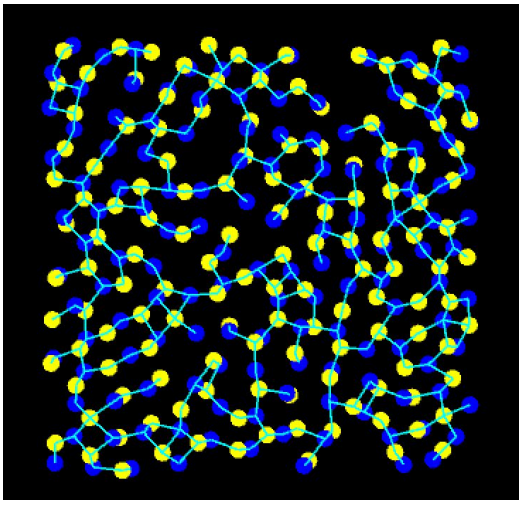
Benchmark 1 - 2nd Largest Subgraph



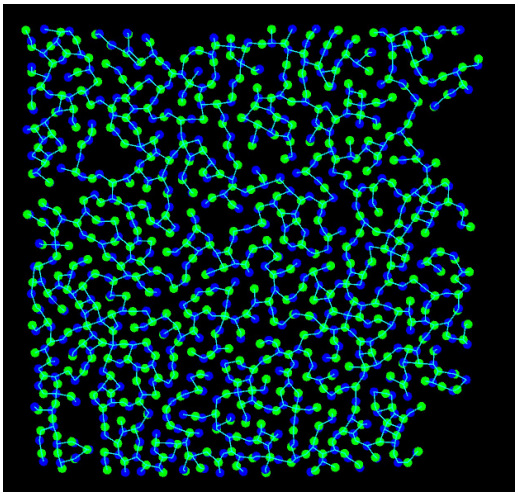
Benchmark 2 - Largest Subgraph



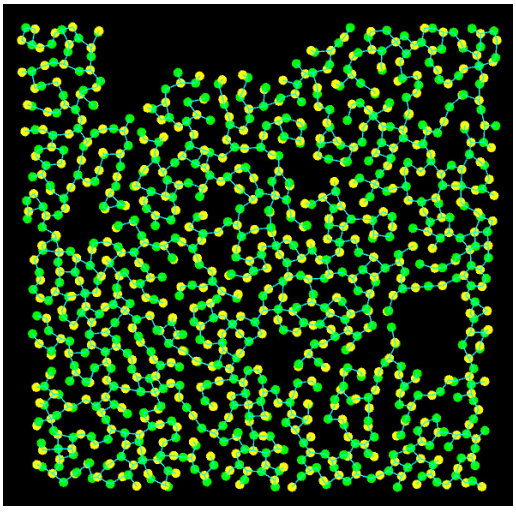
Benchmark 2 - 2nd Largest Subgraph



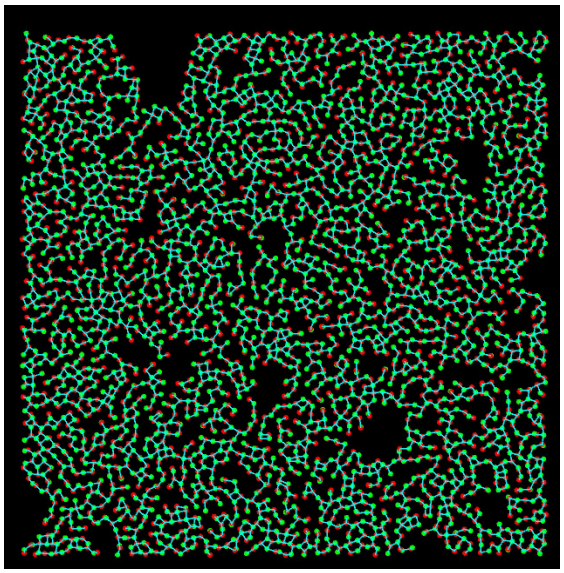
Benchmark 3 - Largest Subgraph



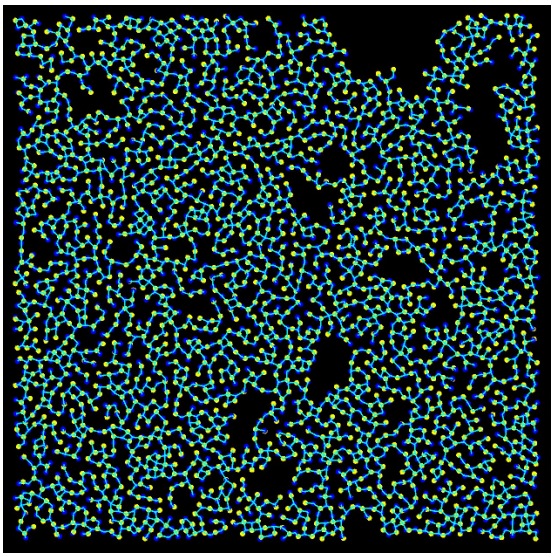
Benchmark 3 - 2nd Largest Subgraph



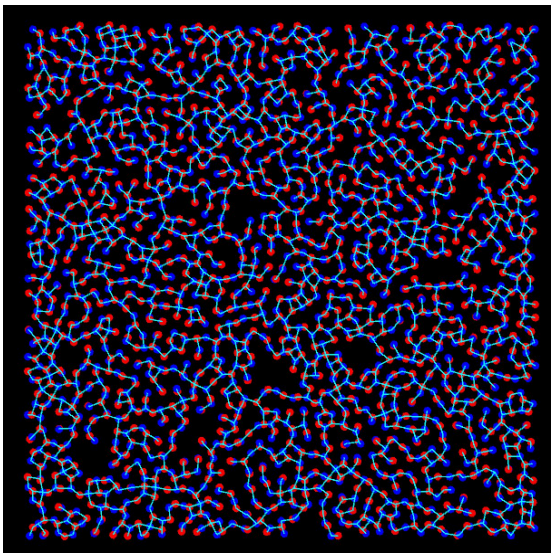
Benchmark 4 - Largest Subgraph



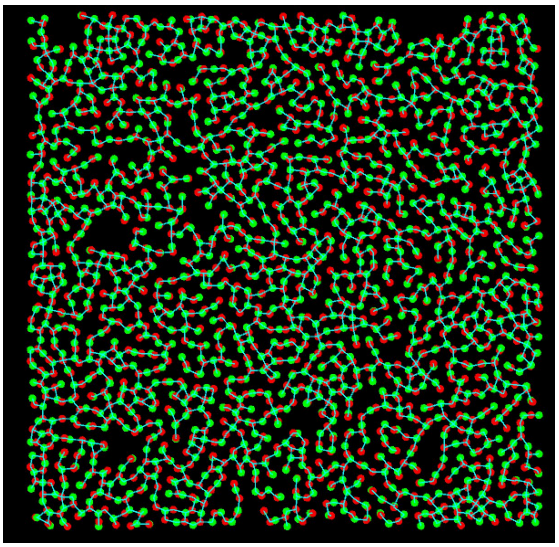
Benchmark 4 - 2nd Largest Subgraph



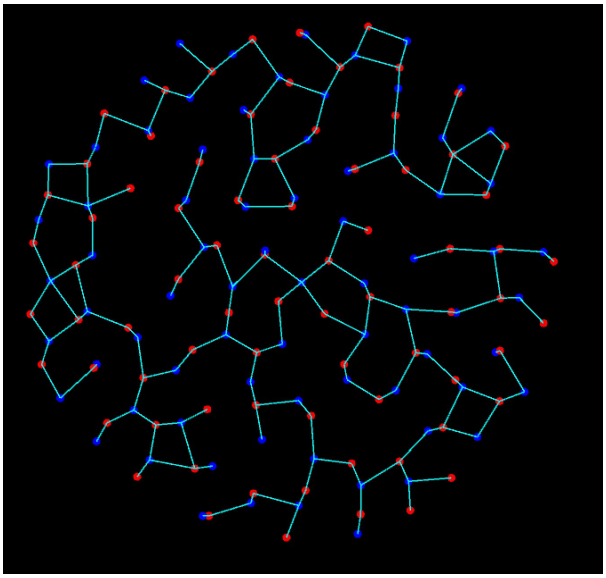
Benchmark 5 - Largest Subgraph



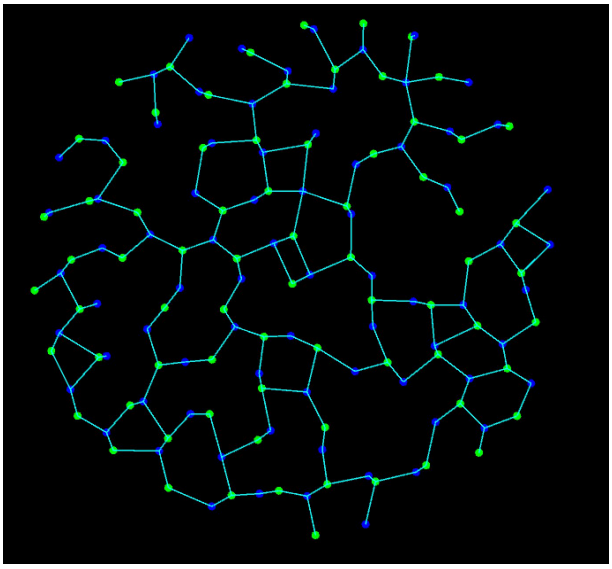
Benchmark 5 - 2nd Largest Subgraph



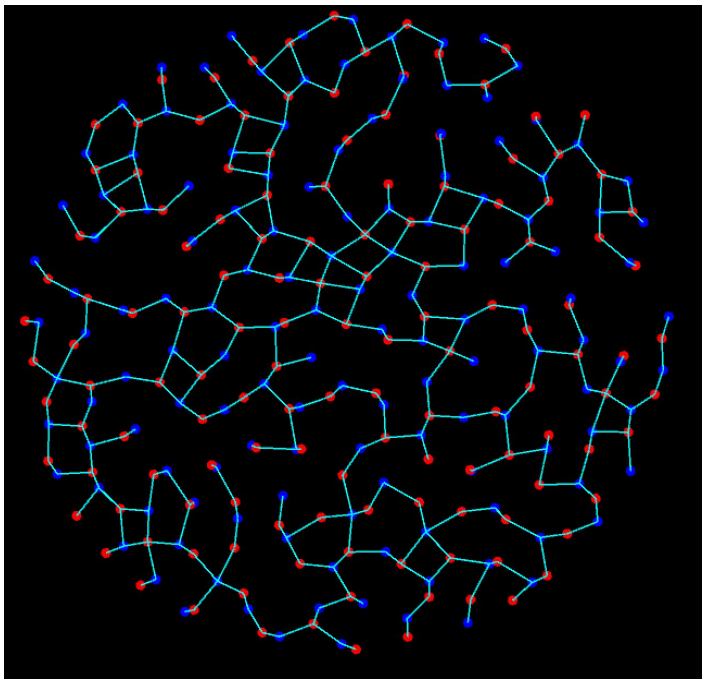
Benchmark 6 - Largest Subgraph



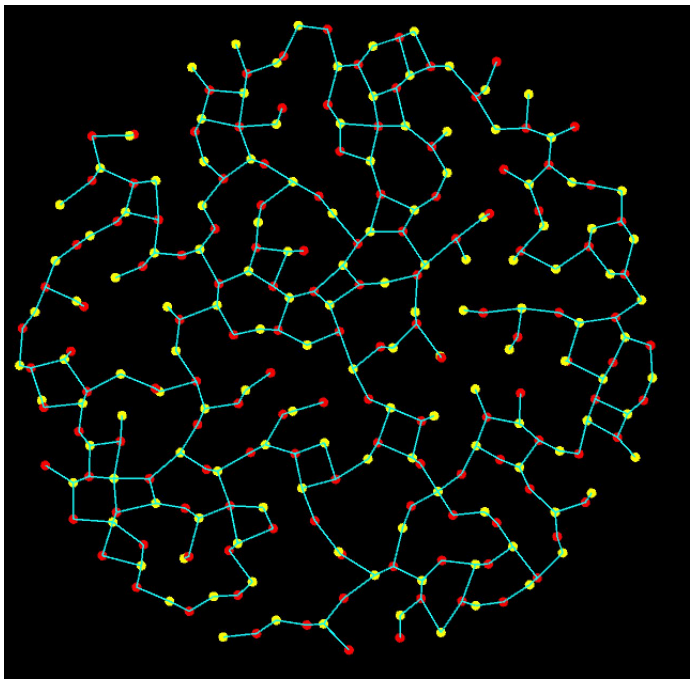
Benchmark 6 - 2nd Largest Subgraph



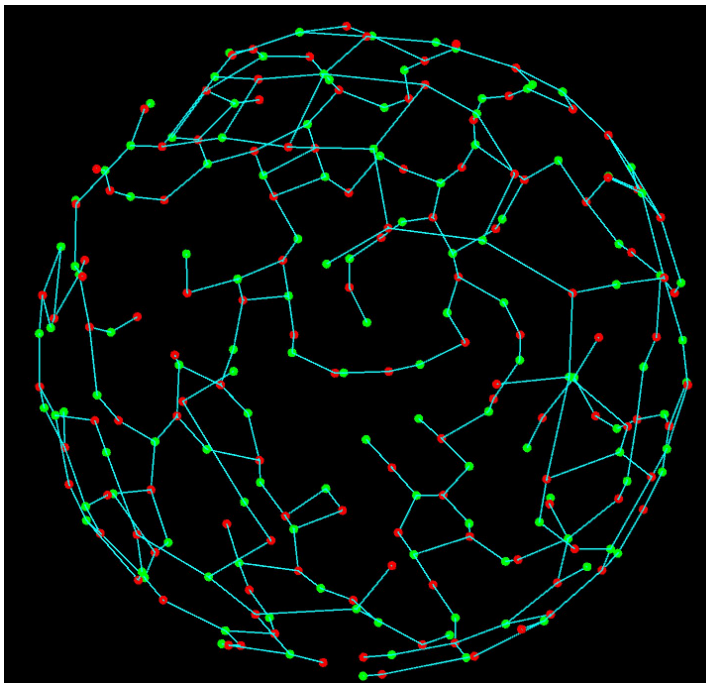
Benchmark 7 - Largest Subgraph



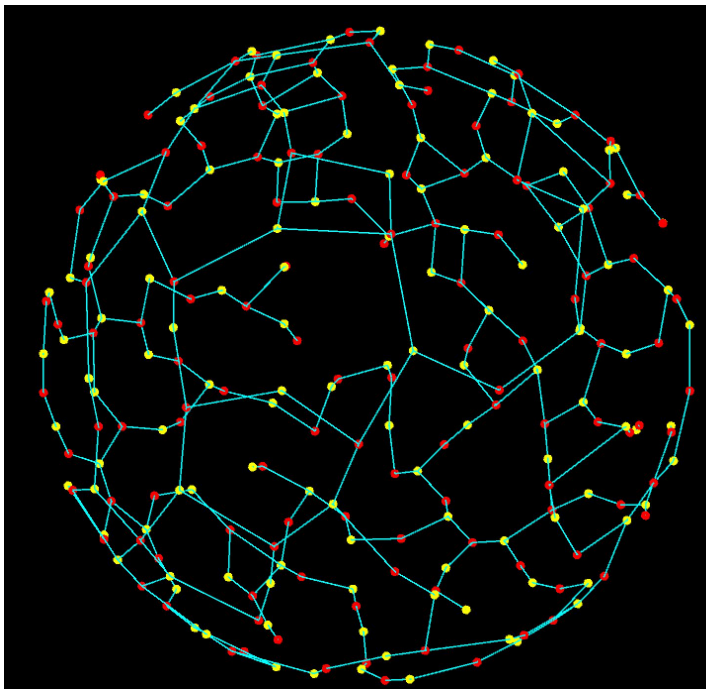
Benchmark 7 - 2nd Largest Subgraph



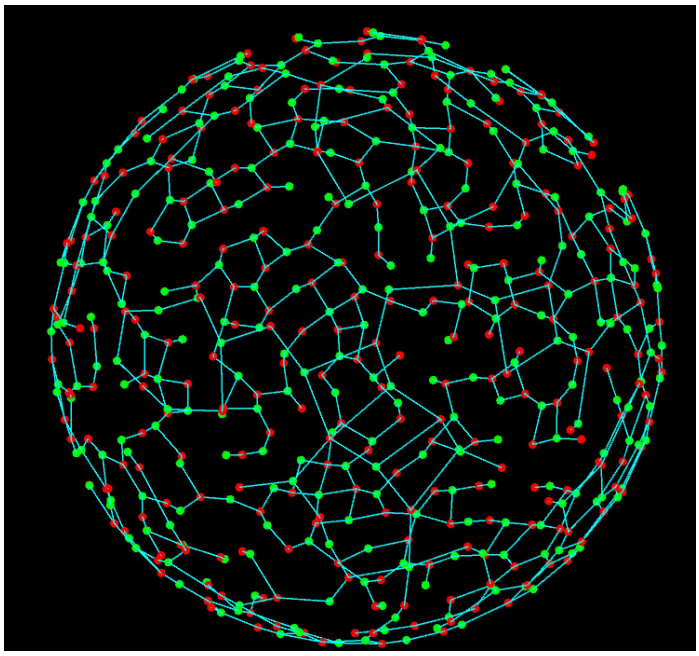
Benchmark 8 - Largest Subgraph



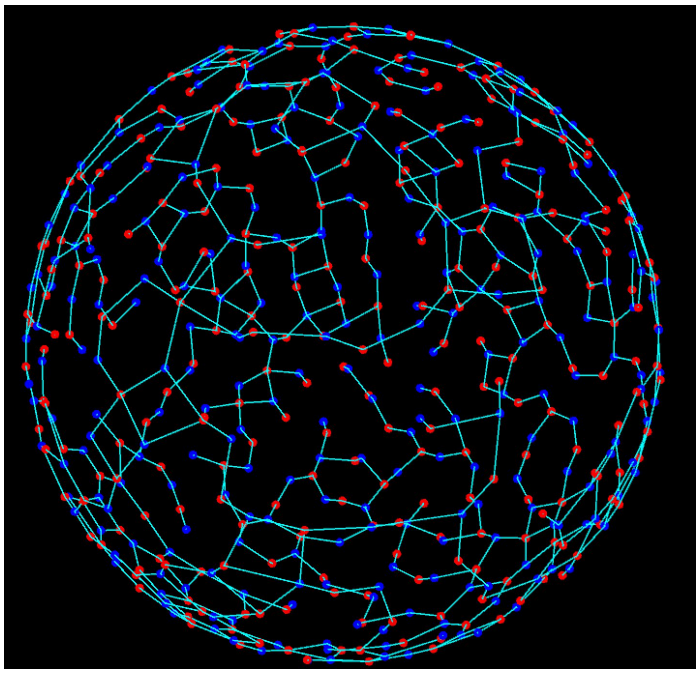
Benchmark 8 - 2nd Largest Subgraph



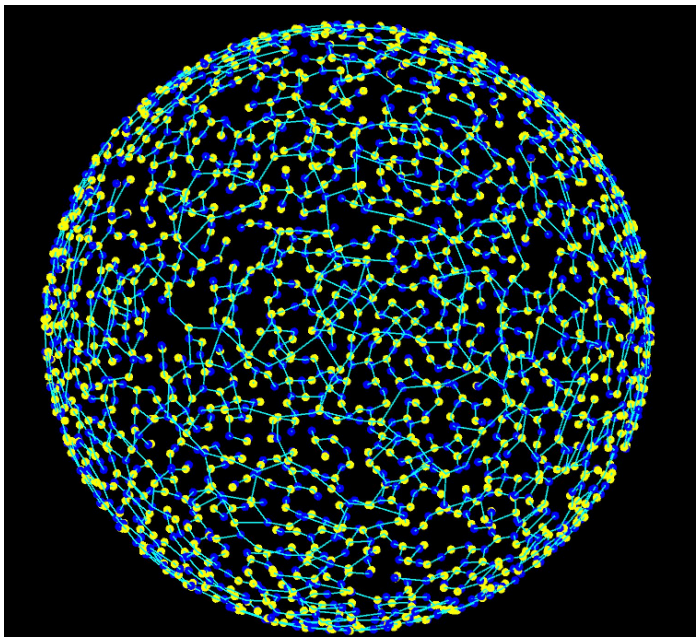
Benchmark 9 - Largest Subgraph



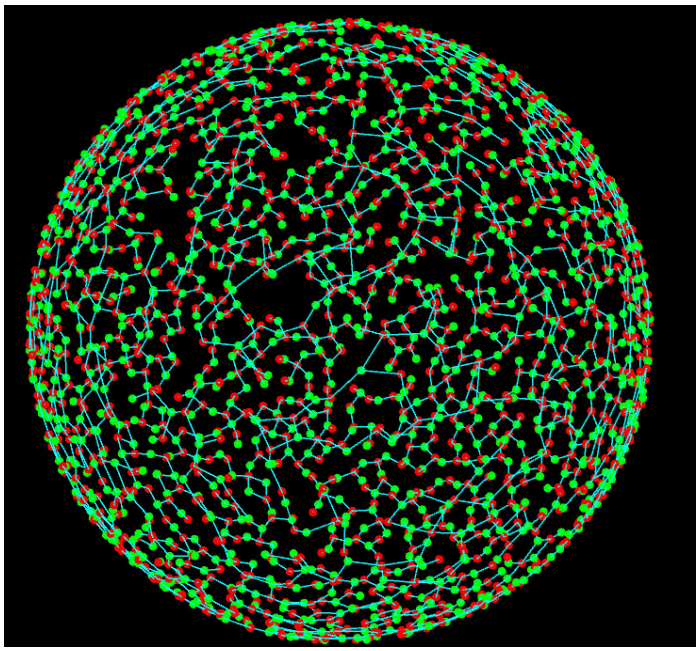
Benchmark 9 - 2nd Largest Subgraph



Benchmark 10 - Largest Subgraph



Benchmark 10 - 2nd Largest Subgraph



Appendices

After this page is the source code of my project.

Because of the way Processing formats PDFs, I highly recommend downloading and running the code from my github:

https://github.com/smitheric95/Java_WirelessSensorNetwork

Java_WirelessSensorNetwork

```
import java.util.*;

/******* INPUT ********/
int avgDegree = 10;
String mode = "sphere"; // square, disk, sphere
int n = 1001; // number of vertices (nodes)
/**********************/

/* Globals */
double R = 0; // calculated in calculateRadius
int graphSize = 500;
int totalDeg = 0; // for real avg degree
int maxDegDeleted = -1;
int numEdges = 0;
float rotX = 0; // rotation
float rotY = 0;
float zoom = 300;
float angle = 0; // rotation with keyboard
Vertex[] vertexDict = new Vertex[n]; // adjacency list of vertices and their neighbors
Integer[] degreeDict = new Integer[n]; // ordered by smallest degree last, array of indices in
vertexDict
int numNotDeleted = n, terminalCliqueSize = 0; // calculating terminal clique
float nodeStrokeWeight = 0.0, edgeStrokeWeight = 0.0;

// output files for creating graphs as needed
PrintWriter outputSequential, outputDistribution;

// first node is the vertex to color
LinkedList[] colorDict = new LinkedList[n];

// calculating four largest colors
HashMap<Integer, Integer> colorCount = new HashMap<Integer, Integer>(); // color : number of
times it occurs
int[] largestColors;
int[][] colorCombos; // all possible combinations of the n most popular colors

// NOTE: not always "color i"
color[] largestColorRGBs = {
  // blue, green, yellow, red
  color(0, 0, 255), color(0, 255, 0), color(255,255,0), color(255, 0, 0),
};

// logic for real time display
int nodeDrawCount = 0;
boolean nodesDrawn = false;
int lineDrawCount = 0;
int colorDrawCount = 0;
```

```
int time = 0;
boolean userDrawLines = false,
        userColorNodes = false,
        userDrawFirstComponent = false,
        userDrawSecondComponent = false,
        firstComponentDrawn = false,
        cliqueDetermined = false;

void setup() {
   long startTime = System.nanoTime();
   smooth();
   size(840, 840, P3D); // set size of window
   surface.setTitle("Drawing Vertices...");

   // create output file
   outputSequential = createWriter("output/outputSequential_" + n + "_" + avgDegree + "_" + mode
+ ".csv");
   outputDistribution = createWriter("output/outputDistribution_" + n + "_" + avgDegree + "_" +
mode + ".csv");

   /***************************** PART I ******************************/

   // build map of nodes
   for(int i = 0; i < n; i++) {
      Vertex v = new Vertex(i);
      Random random = new Random();

      if (mode == "square") {
         v.positionX = random.nextFloat() - 0.5;
         v.positionY = random.nextFloat() - 0.5;
      }
      else if (mode == "disk") {
         // generate random points on a disk
         // http://stackoverflow.com/a/5838991
         float a = random.nextFloat();
         float b = random.nextFloat();

         // ensure b is greater by swapping
         if (b < a) { float temp = b; b = a; a = temp; }

         fill(204, 102, 0);

         v.positionX = (float)(b*Math.cos(2*Math.PI*a/b));
         v.positionY = (float)(b*Math.sin(2*Math.PI*a/b));
      }
      else { // sphere
         // generate random points on the surface of a sphere
         // http://corysimon.github.io/articles/uniformdistn-on-sphere/
```

```
            float theta = (float)(2 * Math.PI * random.nextFloat());
            float phi = (float)(Math.acos(2 * random.nextFloat() - 1));
            v.positionX = sin(phi) * cos(theta);
            v.positionY = sin(phi) * sin(theta);
            v.positionZ = cos(phi);
        }

        vertexDict[i] = v;
        degreeDict[i] = i;
    }// end build map

    R = calculateRadius(); // calculate radius based off avgDegree

    // build vertexDict using sweep method
    // sort degreeDict, which currently is an array of IDs in vertexDict, based on X positions
    // to be sorted by another comparison later on
    Arrays.sort(degreeDict, new Comparator<Integer>() {
        public int compare(Integer v1, Integer v2) {
            return Float.compare(vertexDict[v1].positionX, vertexDict[v2].positionX);
        }
    });

    // go through each vertex
    for (int i = 0; i < n; i++) {
        int j = i-1;

        // if the vertex to left is within range, calculate distance
        while ((j >= 0) && (vertexDict[degreeDict[i]].positionX - vertexDict[degreeDict[j]].positionX
<= R)) {
            // calculate distance based off topology
            if (dist(vertexDict[degreeDict[i]].positionX, vertexDict[degreeDict[i]].positionY,
vertexDict[degreeDict[i]].positionZ,
                    vertexDict[degreeDict[j]].positionX, vertexDict[degreeDict[j]].positionY,
vertexDict[degreeDict[j]].positionZ) <= R) {

                // add both to each other&amp;apos;s linked lists
                vertexDict[degreeDict[i]].neighbors.add(vertexDict[degreeDict[j]].ID);
                vertexDict[degreeDict[j]].neighbors.add(vertexDict[degreeDict[i]].ID);

                numEdges++;
            }

            j -= 1;

        } // end while
    } // end for
    /* end sweep method */
```

```
  // calculate time part 2 took
  long endTime = System.nanoTime();
  println(((endTime - startTime)/1000000) + " ms to build adj list");

  /************************** END PART I ****************************/


  /************************** PART II ****************************/
  startTime = System.nanoTime(); // reset our time counter

  // smallest last vertex ordering
  Arrays.sort(degreeDict, new Comparator<Integer>() {
      public int compare(Integer v1, Integer v2) {
          return -1 * Float.compare(vertexDict[v1].neighbors.getSize(),
vertexDict[v2].neighbors.getSize());
      }
  });

  // initialize colorDict with sorted indices in degreeDict
  for (int i = 0; i < n; i++) {
      colorDict[i] = new LinkedList();
      colorDict[i].add(degreeDict[i]); // first node will be base of colorDict
  }


  outputSequential.println("Original Degree, Degree when Deleted");

  /***** generate colorDict *****/
  // start at the lowest degree
  int degreeIndex = degreeDict.length - 1;
  while (degreeIndex > -1) {
      Vertex curVertex = vertexDict[degreeDict[degreeIndex]];
      totalDeg += curVertex.neighbors.getSize();
      int curDegree = 0;

      // loop through each neighbor
      ListNode curNeighbor = curVertex.neighbors.front;
      while (curNeighbor != null) {
          int j = curNeighbor.ID; // index in vertexDict
          //if hasn&apos;t been deleted from vertexDict
          if (!vertexDict[j].deleted) {
              colorDict[degreeIndex].append(curNeighbor.ID);
              curDegree++;
          }
          curNeighbor = curNeighbor.next;
      }

      //delete from vertexDict
```

```
        vertexDict[degreeDict[degreeIndex]].deleted = true;
        numNotDeleted--;

        // determine terminal clique
        // source: http://stackoverflow.com/a/30106072
        if (!cliqueDetermined) {
            int cliqueCount = 0;
            // for each node in the adjacency list (that hasn&apos;t been deleted)
            for (int j = 0; j < vertexDict.length; j++) {
                if (!vertexDict[j].deleted) {
                    // loop through each neighbor
                    int remainingNeighbors = 0;
                    ListNode curNode = vertexDict[j].neighbors.front;
                    while (curNode != null) {
                        // count the ones that haven&apos;t been deleted
                        if (!vertexDict[curNode.ID].deleted)
                            remainingNeighbors++;
                        curNode = curNode.next;
                    }
                    // if the number of remaining neighbors == numNotDeleted-1 distict vertices
(candidate for clique)
                    if (remainingNeighbors == numNotDeleted - 1)
                        cliqueCount++;
                    else break;
                }
            }
            if (cliqueCount == numNotDeleted && numNotDeleted != 0) {
                // we have a clique, ladies and gentlemen!
                terminalCliqueSize = numNotDeleted;
                cliqueDetermined = true;
            }
        }

        // print original degree vs degree when deleted
        outputSequential.println(curVertex.neighbors.size + "," + curDegree);
        if (curDegree > maxDegDeleted)
            maxDegDeleted = curDegree;

        degreeIndex--;
    }
    /*** colorDict generated ***/

    // set first vertex.color = 1
    vertexDict[colorDict[0].front.ID].nodeColor = 1;
    colorCount.put(1, 1);

    // starting at 1, color all other nodes in colorDict
    for (int i = 1; i < colorDict.length; i++) {
```

```
    // make an array of the nodes&amp;apos; linkedlist size-1
    int[] colorList = new int[ colorDict[i].size - 1 ];

    // initialize array
    int count = 0;
    ListNode curNode = colorDict[i].front.next;
    while (curNode != null) {
        colorList[count] = vertexDict[curNode.ID].nodeColor;
        curNode = curNode.next;
        count++;
    }

    // find the smallest positive color
    int curColor = firstMissingPositive(colorList);

    // color the vertex
    vertexDict[colorDict[i].front.ID].nodeColor = curColor;

    // increment the count of the corresponding color
    Integer freq = colorCount.get(curColor);
    colorCount.put(curColor, (freq == null) ? 1 : freq +1);
}
// calculate time part 2 took
endTime = System.nanoTime();
println(((endTime - startTime)/1000000) + " ms to color nodes");

/***************************** PART III ********************************/
/***** Bipartite backbone selection *****/
// sort the occurences of color
colorCount = sortByValues(colorCount);

// determine the number of colors in colorCount (at most 4)
int numLargestColors = colorCount.size();
if (numLargestColors > 4)
    numLargestColors = 4;
largestColors = new int[numLargestColors];

// find the four (at most) largest colors - store in largestColors
// print color distribution to output file
outputDistribution.println("Color Number, Percentage of Distribution");
Set set = colorCount.entrySet();
Iterator iterator = set.iterator();
int itCount = 0;
while (iterator.hasNext()) {
    Map.Entry c = (Map.Entry)iterator.next();
    if (itCount < numLargestColors)
        largestColors[itCount] = (int)c.getKey();
```

```
    // ouput color and the number of times it occurs
    outputDistribution.println(c.getKey() + "," + ((int)c.getValue() * 1.0 / n));
    itCount++;
}

// initialize colorCombos to have all possible cominations of the (at most)
// four most common colors: AB, AC, AD, BC, BD, CD
int numCombos = (int)choose(numLargestColors, 2);
colorCombos = new int[numCombos][2]; // r = itCount nCr 2

// calculate the different combinations (nCr)
int r = 0, c1 = 0;
while (c1 < numLargestColors-1) {
    int c2 = c1+1;
    while (c2 < numLargestColors) {
        colorCombos[r][0] = largestColors[c1];
        colorCombos[r][1] = largestColors[c2];
        c2++;
        r++;
    }
    c1++;
}

// try all combinations to find two largest backbones
// first and second largest sizes and their starting nodes
int[] largestStarterNodes = new int[2], largestSizes = new int[2];
int[][] largestColorCombos = new int[2][2]; // the color comination of the two largest backbones


// for each color combination, calculate the backbone
// (largest connected component of each bipartite subgraph)
for (int j = 0; j < numCombos; j++) {
    int curColor1 = colorCombos[j][0];
    int curColor2 = colorCombos[j][1];
     // size of the bipartite subgraph = sizes of the two current colors
    int bipartiteSize = colorCount.get(curColor1) + colorCount.get(curColor2);
    int numNodesVisited = 0;

    while (numNodesVisited < bipartiteSize) {
        // pick the node that will be the starting point of the BFS
        // (first node in vertexDict that&apos;s of color1 or 2 that hasn&apos;t been visited
        int curStarterNode = 0;
        while (curStarterNode < vertexDict.length && (vertexDict[curStarterNode].visited[j] ||
                vertexDict[curStarterNode].nodeColor != curColor1 &&
                vertexDict[curStarterNode].nodeColor != curColor2))
                    curStarterNode++;

        // nodes visited in the traversal
```

```
        int curSize = BFS(curStarterNode, j, curColor1, curColor2);

        // if curSize is the largest so far, remember starting node and largest size
        if (curSize > largestSizes[0]) {
            // store the previous largest as the second largest
            largestSizes[1] = largestSizes[0];
            largestStarterNodes[1] = largestStarterNodes[0];
            largestColorCombos[1][0] = largestColorCombos[0][0];
            largestColorCombos[1][1] = largestColorCombos[0][1];

            largestSizes[0] = curSize;
            largestStarterNodes[0] = curStarterNode;
            largestColorCombos[0][0] = curColor1;
            largestColorCombos[0][1] = curColor2;
        }
        else if(curSize > largestSizes[1]) {
            largestSizes[1] = curSize;
            largestStarterNodes[1] = curStarterNode;
            largestColorCombos[1][0] = curColor1;
            largestColorCombos[1][1] = curColor2;
        }

        // reduce the remaining nodes to visit
        numNodesVisited += curSize;
    }
  }
  // calculate time part 3 took
  endTime = System.nanoTime();
  println(((endTime - startTime)/1000000) + " ms to find backbones");

  // exist for drawing only
  BFS(largestStarterNodes[0], -1, largestColorCombos[0][0], largestColorCombos[0][1]);
  BFS(largestStarterNodes[1], -2, largestColorCombos[1][0], largestColorCombos[1][1]);

  /********** Logic for Summary Table ***********/
  int minDeg = vertexDict[degreeDict[degreeDict.length-1]].neighbors.getSize();
  int maxDeg = vertexDict[0].neighbors.getSize();

  // println("----------------- Summary Table ----------------");
  // N, R, M (numEdges), min degree, avg degree, real avg degree, max degree,
  // max degree when deleted, number of colors, size of largest color class
  // terminal clique size, n of largest backbone, m of largest backbone, domination percentage

  //println(n, R, numEdges, minDeg, avgDegree, totalDeg/n, maxDeg);
  //println(maxDegDeleted, colorCount.size(), largestSizes[0], terminalCliqueSize, largestSizes[0],
largestSizes[0] - 1, (largestSizes[0]*1.0)/n);
  //println("-----------------------------------------------");
```

```
  println();
  println("1st Largest subgraph starts at: " + largestStarterNodes[0] + " with a size of: " +
largestSizes[0] + " and of color combo of " + largestColorCombos[0][0] + ", " +
largestColorCombos[0][1]);
  println("2nd Largest subgraph starts at: " + largestStarterNodes[1] + " with a size of: " +
largestSizes[1] + " and of color combo of " + largestColorCombos[1][0] + ", " +
largestColorCombos[1][1]);

  // close output files
  outputSequential.flush();
  outputSequential.close();
  outputDistribution.flush();
  outputDistribution.close();
}

void draw() {
  // put matrix in center
  pushMatrix();
  translate(width/2, height/2);
  scale(zoom);
  rotate(angle);
  noFill();
  background(0);

  // rotate matrix based off mouse movement
  rotateX(rotX);
  rotateY(rotY);

  // delay drawing
  // source:
https://forum.processing.org/one/topic/how-do-you-make-a-program-wait-for-one-or-two-seconds.ht
ml
  if (millis() > time){
      if (!firstComponentDrawn)
          time = millis() + 1;
      if (nodeDrawCount < n) // replace with press space!
          nodeDrawCount++;
      else if (userDrawLines){
          nodesDrawn = true;
          if (n > 20)
              lineDrawCount += n/20;
          else lineDrawCount = 20;
          if (userColorNodes)
              if (n > 20)
                  colorDrawCount += n/20;
              else colorDrawCount = 20;
          if (userDrawFirstComponent && !firstComponentDrawn) {
              firstComponentDrawn = true;
```

```
            nodeDrawCount = n;
        }
    }
}

// count what&apos;s been drawn
int linesDrawn = lineDrawCount;
int colorsDrawn = colorDrawCount;

// calculate stroke weight depending on graph type and size
nodeStrokeWeight = 0.03;
edgeStrokeWeight = 0.005;
if ((!userDrawFirstComponent && !userDrawSecondComponent)) {
    if (n > 1000) {
        nodeStrokeWeight = 0.02;
        edgeStrokeWeight = 0.001;
    }
    else if (n > 10000) {
        nodeStrokeWeight = 0.0001;
        edgeStrokeWeight = 0.00001;
    }
}
if (mode == "square") {
    nodeStrokeWeight /= 2;
    edgeStrokeWeight /= 2;
}

// draw nodes
for (int i = 0; i < nodeDrawCount; i++) {
    stroke(255);
    strokeWeight(nodeStrokeWeight);

    Vertex curVertex = vertexDict[i];

    if ((!userDrawFirstComponent && !userDrawSecondComponent) || (userDrawFirstComponent
&& curVertex.toDraw[0]) || (userDrawSecondComponent && curVertex.toDraw[1])) {
        // find appropriate color
        int j;
        for (j = 0; j < largestColors.length; j++)
            if (largestColors[j] == curVertex.nodeColor) break;

        if (j < largestColors.length && (colorsDrawn > 0 || userDrawFirstComponent ||
userDrawSecondComponent)) {
            // set color based off... well, color
            stroke(largestColorRGBs[j]);
        }
        else if (userColorNodes) stroke((curVertex.nodeColor*50)%255,
(curVertex.nodeColor*20)%255,(curVertex.nodeColor*70)%255);
```

```
            colorsDrawn--;
            curVertex.drawVertex(); // draw!
        }

        stroke(0, 255, 255);
        strokeWeight(edgeStrokeWeight);
        // draw line between vertex and its neighbors
        if (userDrawFirstComponent || userDrawSecondComponent || nodesDrawn && (linesDrawn >
0)) {
            ListNode curNeighbor = curVertex.neighbors.front;

            while (curNeighbor != null) {
                int index = curNeighbor.ID;
                if ((!userDrawFirstComponent && !userDrawSecondComponent) ||
(userDrawFirstComponent && curVertex.toDraw[0] && vertexDict[curNeighbor.ID].toDraw[0]) ||
(userDrawSecondComponent && curVertex.toDraw[1] && vertexDict[curNeighbor.ID].toDraw[1]))
                    line(curVertex.positionX, curVertex.positionY, curVertex.positionZ,
vertexDict[index].positionX, vertexDict[index].positionY, vertexDict[index].positionZ);
                curNeighbor = curNeighbor.getNext();
            }
            linesDrawn--;
        }
    }
    //println("userDrawFirstComponent: " + userDrawFirstComponent + ", " +
"userDrawSecondComponent: " + userDrawSecondComponent);

    popMatrix();
} // end draw()

// returns radius of a point based average degree
// check video to see if accurate
double calculateRadius() {
    if (mode == "square") {
        return Math.sqrt( (avgDegree*1.0/(n*Math.PI)) );
    }
    else if (mode == "disk") {
        return Math.sqrt( avgDegree*1.0/n );
    }
    else {
        return Math.sqrt( 4*avgDegree*1.0/n );
    }
}

// prints BFS traversal on an adjacency list
// edited from source: http://www.geeksforgeeks.org/breadth-first-traversal-for-a-graph/
// colorCombo == -1 if the node is to be drawn (part of the 1st largest componenent)
// colorCombo == -2 if the node is to be drawn (part of the 2nd largest componenent)
int BFS(int v, int colorCombo, int c1, int c2) {
```

```
    java.util.LinkedList<Integer> queue = new java.util.LinkedList<Integer>();
    int count = 0; // number of nodes visited

    // mark the current node as visited and enqueue it
    if (colorCombo > -1)
        vertexDict[v].visited[colorCombo] = true;
    queue.add(v);

    while (queue.size() != 0) {
        // Dequeue a vertex from queue and print it
        v = queue.poll();

        /* Get all adjacent vertices of the dequeued vertex s
        If a adjacent has not been visited, then mark it
        visited and enqueue it */
        ListNode curNode = vertexDict[v].neighbors.front;
        while (curNode != null) {
            // if the node hasn&apos;t been visited (or it needs to be drawn)
            // and it&apos;s the right color, mark it visited
            if (((colorCombo > -1 && !vertexDict[curNode.ID].visited[colorCombo]) || ((colorCombo
== -1 && !vertexDict[curNode.ID].visitedWhileDrawn[0]) || (colorCombo == -2 &&
!vertexDict[curNode.ID].visitedWhileDrawn[1])))
                && (vertexDict[curNode.ID].nodeColor == c1 || vertexDict[curNode.ID].nodeColor ==
c2)) {
                // mark the node as visited
                if (colorCombo > -1)
                    vertexDict[curNode.ID].visited[colorCombo] = true;

                // draw the node if necessary
                else {
                    // mark
                    if (colorCombo == -1) {
                        vertexDict[curNode.ID].visitedWhileDrawn[0] = true;
                        vertexDict[curNode.ID].toDraw[0] = true;
                    }
                    else {
                        vertexDict[curNode.ID].visitedWhileDrawn[1] = true;
                        vertexDict[curNode.ID].toDraw[1] = true;
                    }
                }
                queue.add(curNode.ID);
                count++;
            }
            curNode = curNode.next;
        }
    }
    return count + 1;
}
```

```java
// find the smallest missing element in a sorted array
// http://www.programcreek.com/2014/05/leetcode-first-missing-positive-java/
// this function was copied directly from its source
public int firstMissingPositive(int[] A) {
  int n = A.length;

  for (int i = 0; i < n; i++) {
    while (A[i] != i + 1) {
      if (A[i] <= 0 || A[i] >= n)
        break;

        if(A[i]==A[A[i]-1])
            break;

      int temp = A[i];
      A[i] = A[temp - 1];
      A[temp - 1] = temp;
    }
  }

  for (int i = 0; i < n; i++){
    if (A[i] != i + 1){
      return i + 1;
    }
  }

  return n + 1;
}

// sort HashMap by value
// source: http://beginnersbook.com/2013/12/how-to-sort-hashmap-in-java-by-keys-and-values/
// this function was copied directly from its source
private static HashMap sortByValues(HashMap map) {
    List list = new java.util.LinkedList(map.entrySet());
    // Defined Custom Comparator here
    Collections.sort(list, new Comparator() {
      public int compare(Object o1, Object o2) {
        return -1 * ((Comparable) ((Map.Entry) (o1)).getValue())
          .compareTo(((Map.Entry) (o2)).getValue());
      }
    });

    // Here I am copying the sorted list in HashMap
    // using LinkedHashMap to preserve the insertion order
    HashMap sortedHashMap = new LinkedHashMap();
    for (Iterator it = list.iterator(); it.hasNext();) {
      Map.Entry entry = (Map.Entry) it.next();
```

```
            sortedHashMap.put(entry.getKey(), entry.getValue());
        }
        return sortedHashMap;
    }

// x choose y
// source: http://stackoverflow.com/a/1678715
// this function was copied directly from its source
public static double choose(int x, int y) {
    if (y < 0 || y > x) return 0;
    if (y > x/2) {
        // choose(n,k) == choose(n,n-k),
        // so this could save a little effort
        y = x - y;
    }

    double denominator = 1.0, numerator = 1.0;
    for (int i = 1; i <= y; i++) {
        denominator *= i;
        numerator *= (x + 1 - i);
    }
    return numerator / denominator;
}

void mouseDragged() {
    rotX += (pmouseY-mouseY) * 0.1;
    rotY += -1 * (pmouseX-mouseX) * 0.1;
}

void keyPressed() {
    // https://forum.processing.org/two/discussion/2151/zoom-in-and-out
    if (keyCode == UP) {
        zoom += 20;
    }
    else if (keyCode == DOWN) {
        zoom -= 20;
    }
    else if (keyCode == RIGHT) {
        angle += .03;
    }
    else if (keyCode == LEFT) {
        angle -= .03;
    }
    if (key == 32) { // space
        if (userDrawSecondComponent) {
            userDrawSecondComponent = false;
            colorDrawCount = n;
            surface.setTitle("All Vertices and Edges");
```

```
        }
        else if (userDrawFirstComponent) {
            userDrawSecondComponent = true;
            userDrawFirstComponent = false;
            surface.setTitle("2nd Largest Component");
        }
        else if (userColorNodes) {
            userDrawFirstComponent = true;
            surface.setTitle("1st Largest Component");
        }
        else if (userDrawLines) {
            userColorNodes = true;
            surface.setTitle("Coloring");
        }
        else if (nodeDrawCount < n) {
            nodeDrawCount = n;
            surface.setTitle("All Vertices");
        }
        else {
            userDrawLines = true;
            surface.setTitle("All Vertices and Edges");
        }
    }
}
```

LinkedList

```
public class LinkedList {
    ListNode front;
    ListNode back;

    private int size;

    public LinkedList() {
        this.size = 0;
        this.front = null;
        this.back = null;
    }

    // add to front
    public void add(int ID) {
        ListNode node = new ListNode(ID);
        node.next = this.front;
        this.front = node;
        if (this.back == null)
            this.back = this.front;

        this.size++;
```

```
}

// add directly to back
public void append(int ID) {
    ListNode node = new ListNode(ID);
    if (this.back != null)
        this.back.next = node;

    this.back = node;

    if (this.front == null)
        this.front = node;

    this.size++;
}

public void printList() {
    ListNode cur = this.front;

    System.out.print(" ");
    while (cur != null) {
        System.out.print("[" + cur.ID + "]->");
        cur = cur.getNext();
    }
    System.out.println("X\n");
}

// delete from list and return whether it got delted
public boolean delete(int ID) {
    // empty list
    if (this.size == 0)
        return false;

    ListNode cur = this.front;

    // delete head
    if (cur.ID == ID) {
        this.front = cur.next;
    }

    // loop till we find node with the right ID
    while (cur.next != null) {
        if (cur.next.ID == ID) {
            cur.next = cur.next.next; // delete
            return true;
        }
        cur = cur.next;
    }
```

```
        // node not found
        return false;
    }

    public ListNode getFront() {
        return this.front;
    }

    public int getSize() {
        return this.size;
    }
}
```

ListNode

```
public class ListNode {
    int ID; // index in vertexDict
    ListNode next;

    public ListNode(int ID) {
        this.ID = ID;
        this.next = null;
    }

    public ListNode getNext() {
        return this.next;
    }

}
```

Vertex

```
public class Vertex {
    int ID;
    float positionX;
    float positionY;
    float positionZ;
    LinkedList neighbors;
    private boolean sortByDegree;
    boolean deleted; // pseudo deleted for coloring
    int nodeColor;
    boolean[] visited; // for calculating largest backbone (index for each color combo)
    boolean[] visitedWhileDrawn;
    boolean[] toDraw; // whether or not to draw the vertex

    public Vertex(int ID) {
        this.ID = ID;
        this.positionX = 0;
        this.positionY = 0;
```

```
        this.positionZ = 0;
        this.sortByDegree = false;
        this.neighbors = new LinkedList();
        this.deleted = false;
        this.nodeColor = 0;
        this.visited = new boolean[6];
        this.toDraw = new boolean[2];
        this.visitedWhileDrawn = new boolean[2];
    }

    public int getNumNeighbors() {
        return this.neighbors.size;
    }

    public void drawVertex() {
        //strokeWeight(0.05);
        point(this.positionX, this.positionY, this.positionZ);
    }

    public void printVertex() {
        System.out.println("[" + this.ID + " (" + this.toString().substring(33, this.toString().length()) +
")]: " + this.positionX + ", " + this.positionY + ", " + this.positionZ + " Color: " + this.nodeColor);
        this.neighbors.printList();
    }
}
```