Wireless Sensor Network

Han Chen

Dec 14, 2015

Table of Contents

1. Introduction and Summary

A wireless sensor network (WSN) are spatially distributed autonomous sensors to monitor physical or environmental conditions and to cooperatively pass their data through the network to a main location [1]. WSNs have been widely used in a variety of areas in the modern society. So, it is of great importance to model and study them.

In this project, several random geometric graphs are used to model the wireless sensor network. Distribution of these graphs includes unit square, disk with unit radius and sphere. They are used to model WSN in different situations. The communication backbones are vital to the connectivity of WSN. To identify a backbone with high dominant percentage, nodes of WSN need to be ordered appropriately [3]. An efficient way is to use the smallest-last ordering algorithm [2]. After ordering and coloring the nodes, backbones could be selected from bipartite subgraphs, which are built on different color sets.

This project creates a precise model for WSN in both local and global areas. They could be used to study WSN and find communication backbones effectively. By employing efficient algorithms, resource required to run these models are significantly saved. The model are then applied on 10 benchmark graphs with different number of nodes and distribution types. A summary of results are demonstrated in the following tables.

Graph Coloring Summary Table 1:

ID	**Number of vertices**	**Radius**	**Distribution type**	**Number of edges**	**Min degree**	**Max degree**	**Average degree**
Test	20	0.4	Square	84	2	13	8
1	1000	0.103	Square	15238	8	52	30
2	4000	0.058	Square	81098	7	63	40
3	4000	0.0715	Square	120798	18	93	60
4	16000	0.0353	Square	485827	13	89	60
5	64000	0.0175	Square	1941591	12	97	60
6	4000	0.123	Disk	120007	14	99	60
7	4000	0.1753	Disk	241618	34	191	120
8	4000	0.484	Sphere	121178	20	110	60
9	16000	0.405	Sphere	982855	40	242	122
10	64000	0.08	Sphere	3758350	33	275	119

Graph Coloring Summary Table 2:

ID	Max min-degree when deleted	Number of colors	Max color class size	Terminal clique size	Number of edges in backbone	Percentage of vertices covered
Test	8	9	4	9	4	80.00%
1	22	22	76	22	143	99.70%
2	26	26	232	26	454	99.65%
3	38	37	164	37	320	99.95%
4	38	37	652	37	1288	99.87%
5	41	40	2627	40	5169	99.80%
6	42	36	171	36	340	100%
7	78	67	93	67	185	100%
8	47	40	173	40	341	99.92%
9	98	78	381	78	754	99.94%
10	111	85	1632	85	3232	99.94%

Backbone Summary Table:

ID	Backbone Colors	Number of Vertices	Number of Edges	Domination Percentage
1	1, 3	144	143	99.70%
	2, 3	138	137	98.60%
2	1, 2	454	453	99.65%
	1, 3	440	439	99.25%
3	1, 3	322	321	99.95%
	1, 4	321	320	99.98%
4	1, 2	1289	1288	99.87%
	1, 3	1285	1284	99.53%
5	1, 2	5170	5169	99.80%
	1, 3	5147	5146	99.76%
6	1, 2	341	340	100.0%
	2, 3	336	335	99.88%
7	2, 3	186	185	100.0%
	1, 2	181	180	99.95%
8	2, 3	342	341	99.92%
	1, 3	340	339	99.82%
9	1, 2	755	754	99.94%
	2, 3	755	754	99.94%
10	1, 2	3233	3232	99.94%
	2, 4	3328	3327	99.94%

2. Programming Environment Description

2.1 Software Description

The model for wireless sensor network is designed and implemented using the Python programming language. Programming environment for this implementation is IPython notebook, which is an interactive computing environment [4]. It is efficient for Python programming, especially for data analysis and figure plotting. The plotting library in NetworkX is also used to display the random geometric graphs [5]. Another plotting library, Matplotlib [6], is also used to refine the figures produced by NetworkX. It also provide a toolkit (mplot3d) for 3-dimension figure display. This toolkit is employed to generate RGGs on the surface of sphere.

The reason that NetworkX and Matplotlib are used, is that they could be called directly from IPython notebook. Thus, it is an efficient way to plot the required figures, without opening extra software or transferring data. Besides, the plotting library in NetworkX is also built on the Matplotlib. By using Matplotlib, I could easily adjust the demonstration of figures generated by NetworkX and add some text description.

2.2 Hardware Description

This project is completed in my PC. It is a Dell XPS 15.

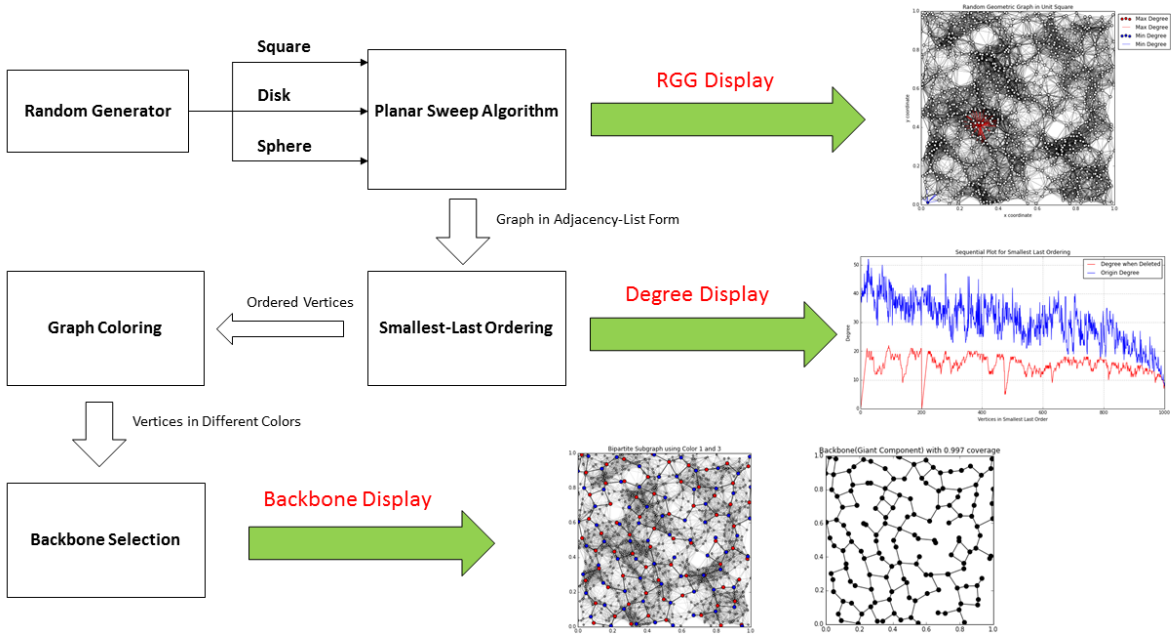
Operating System	Windows 10 64-bit
CPU	Intel I7 2.6GHz
RAM	8GB

3. References

1. Wireless Sensor Network, Wikipedia, https://en.wikipedia.org/wiki/Wireless_sensor_network
2. D.W. Matula, 'Smallest-Last Ordering and Clustering and Graph Coloring Algorithm', *Journal of Association for Computing Machinery*, July 1983
3. D. Mahjoub, D.W. Matula, 'Employing (1 − ε) Dominating Set Partitions as Backbones in Wireless Sensor Networks', *Distributed Computing in Sensor Systems*, 2010
4. IPython Notebook, an interactive Python computational environment, http://ipython.org/notebook.html
5. NetworkX, High-productivity software for complex networks, https://networkx.github.io/
6. Matplotlib, Python plotting library, http://matplotlib.org/

4. Reduction to Practice

4.1 Data Structure Design



First of all, to generate the benchmark graphs, random generators in Python are used.

For RGG in unit square, random numbers between 0 and 1 are generated as x and y coordinates of vertices. For RGG in disk with unit radius, random numbers between -1 and 1 are generated. However, some of them may be out of the disk. In such cases, both x and y coordinates are multiplied with a new random number between 0 and 1, in order to put it into the disk. For RGG in sphere with unit radius, random numbers between -1 and 1 are generated and normalized to put them on the surface of the sphere.

In this step, RGGs are stored in a dictionary. Keys of the dictionary are the numbers assigned to vertices in the order they are generated. Values of the dictionary are coordinates of the vertices.

Secondly, they are used as input for the planar sweep algorithm, to be transformed into adjacency lists.

The planar sweep algorithm sort the vertices by their x coordinates, and then only examine those vertices, whose x coordinates are not larger than the current x

coordinate by the value of a threshold. With this algorithm, only a small fraction of the vertices are examined, saving a lot of time.

After this step, RGGs are stored in adjacency lists. And NetworkX could be used to display them.

Then, vertices are rearranged by smallest-last ordering algorithm. They are colored in the smallest-last order. Several color classes are generated. Vertices are distributed into different color classes. This is implemented using a dictionary, with color values as keys and vertices as values.

Among them, the first 4 color classes are selected. By pairing them, 6 possible bipartite subgraphs are generated. For each bipartite graph, the largest components are selected as the backbone. The two backbone with most edges are displayed.

4.2 Algorithm Descriptions

The smallest last ordering algorithm takes graph in adjacency lists form, and output a new order of vertices:

a. Initialize a list to store the vertices in smallest-last order.
b. Order vertices of the graph according to their degree.
c. Select the vertex with smallest degree, breaking ties alphabetically, and put it to the top of the list.
d. Sequentially, delete it from adjacency lists of its neighbors, and decrement their degree by 1.
e. Repeat step c and d, until all vertices are put into the list.

Finally, vertices are ordered in the list. And the last deleted vertex is in the beginning of the list.

The graph coloring algorithm uses the vertices in smallest-last order as input, and distribute the vertices into different color classes, ensuring that no two adjacent vertices are in the same color class:

a. Initialize a dictionary, with color values as its keys, and a list of vertices as its values.
b. Select the first vertex in the smallest-last vertex list, check its adjacency list to find the smallest available color value.
c. Put this vertex into the dictionary according to its color value.
d. Repeat step b and c, until all vertices are assigned colors.

Finally, vertices of the RGG are distributed into different color classes. Because the smallest available color value is chosen for each vertex, the first several color classes usually contains most of the vertices.

4.3 Algorithm Engineering

In the smallest-last ordering algorithm, all vertices are sorted by their degree. This could be done in O(V) time using bucket sort algorithm, where degrees are used as buckets. The space required for bucket sort is also O(V).

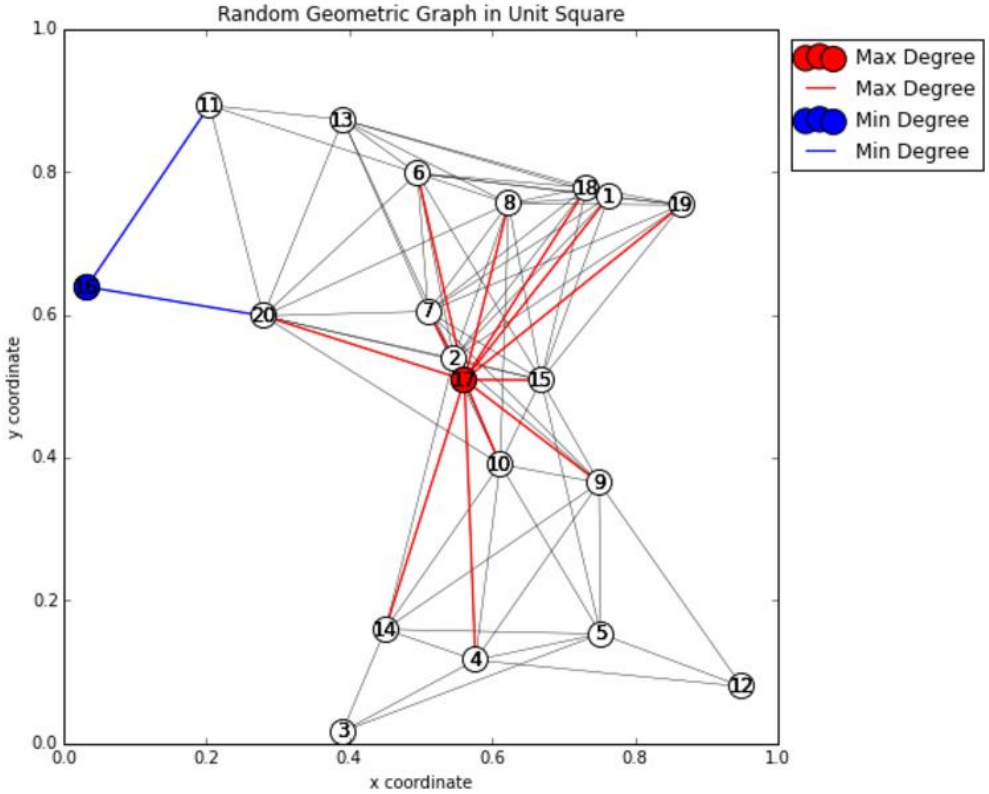
After that, vertices are traversed in this order. At each step, it takes only O(1) time to find the vertex with smallest degree. To update the degree of vertices adjacent to it, its adjacency list is traversed. The total time to traverse adjacency lists at all steps are O(E). So, the total time complexity for smallest-last ordering algorithm is O(E+V).

For the graph coloring algorithm, vertices are traversed in smallest-last order. At each step, adjacency list of the current vertex has to be traversed to find the smallest available color value for it. The total time to find the appropriate color value for all vertices is proportional to the number of edges in the graph. That is to say, this could take O(E) time. Besides, traversing all vertices could use O(V) time. As a result, the time complexity for this graph coloring algorithm is also O(V+E). The space for graph coloring is a dictionary to store the color of vertices plus the space used to find the smallest available color value.

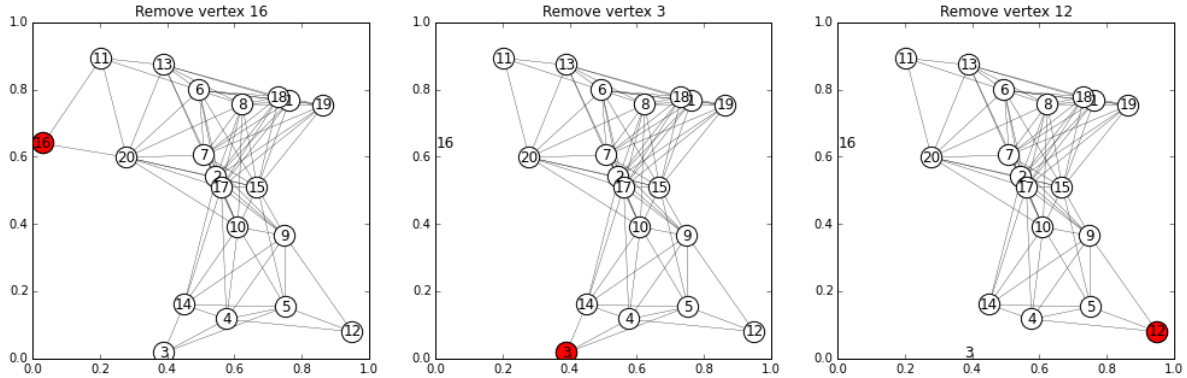
4.4 Verification Walkthrough

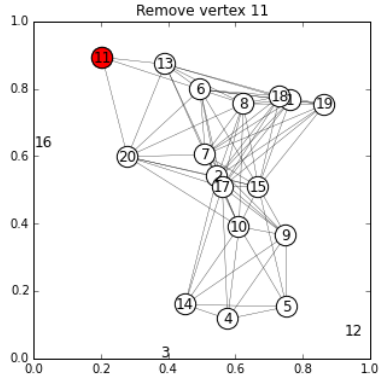
This verification is displayed using a random geometric graph in unit square, with 20 vertices and an adjacent threshold of 0.4. Their coordinates are provided by a random generator in Python. Using the planar sweep method, adjacency lists of these vertices are generated.

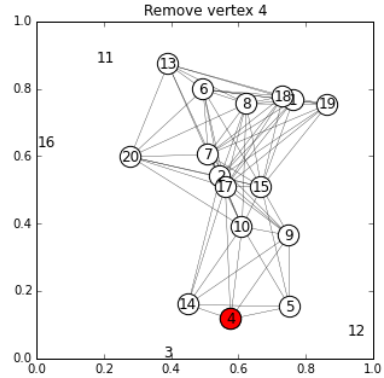
This graph is shown below. Vertex 17 has the maximum degree, and vertex 16 has the minimum degree.

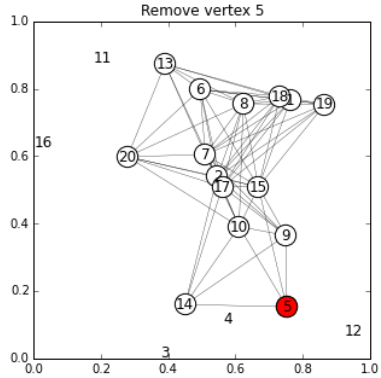


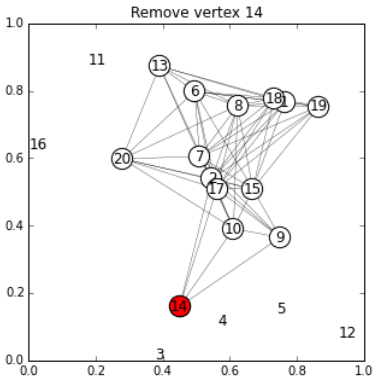
Then, smallest last ordering algorithm is applied on this graph.

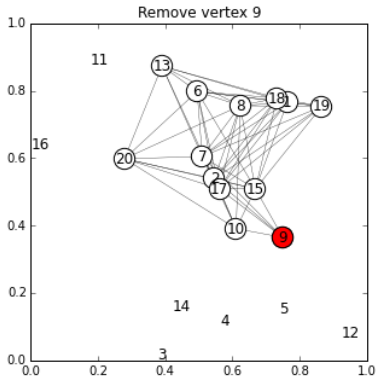


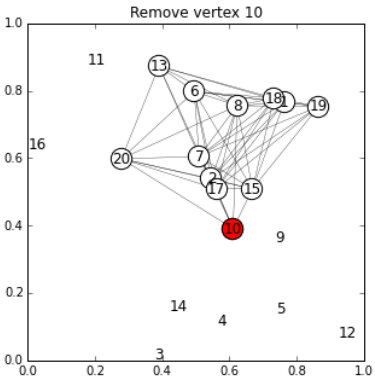

Remove vertex 11

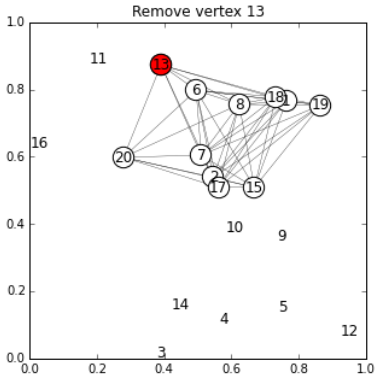

Remove vertex 4

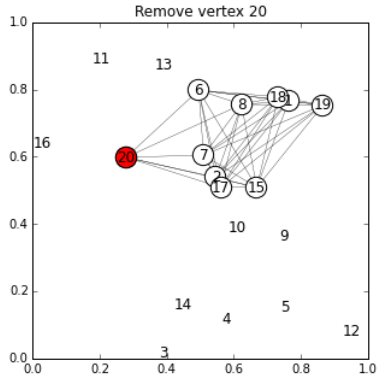

Remove vertex 5

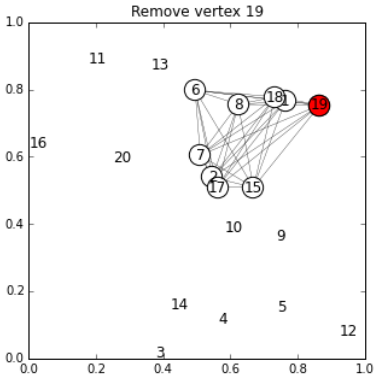

Remove vertex 14

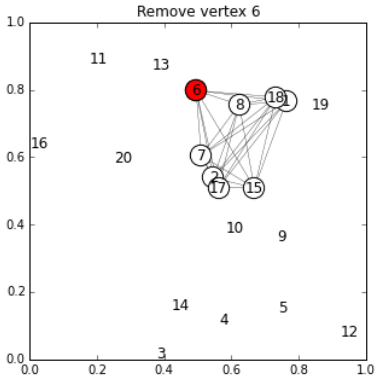

Remove vertex 9

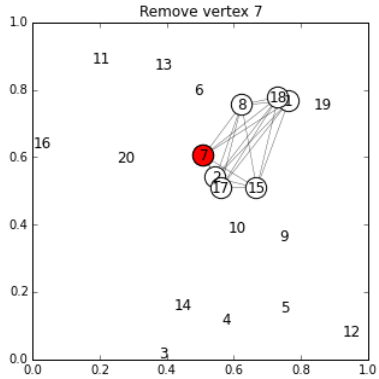

Remove vertex 10

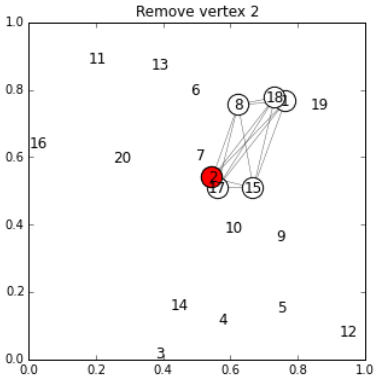

Remove vertex 13

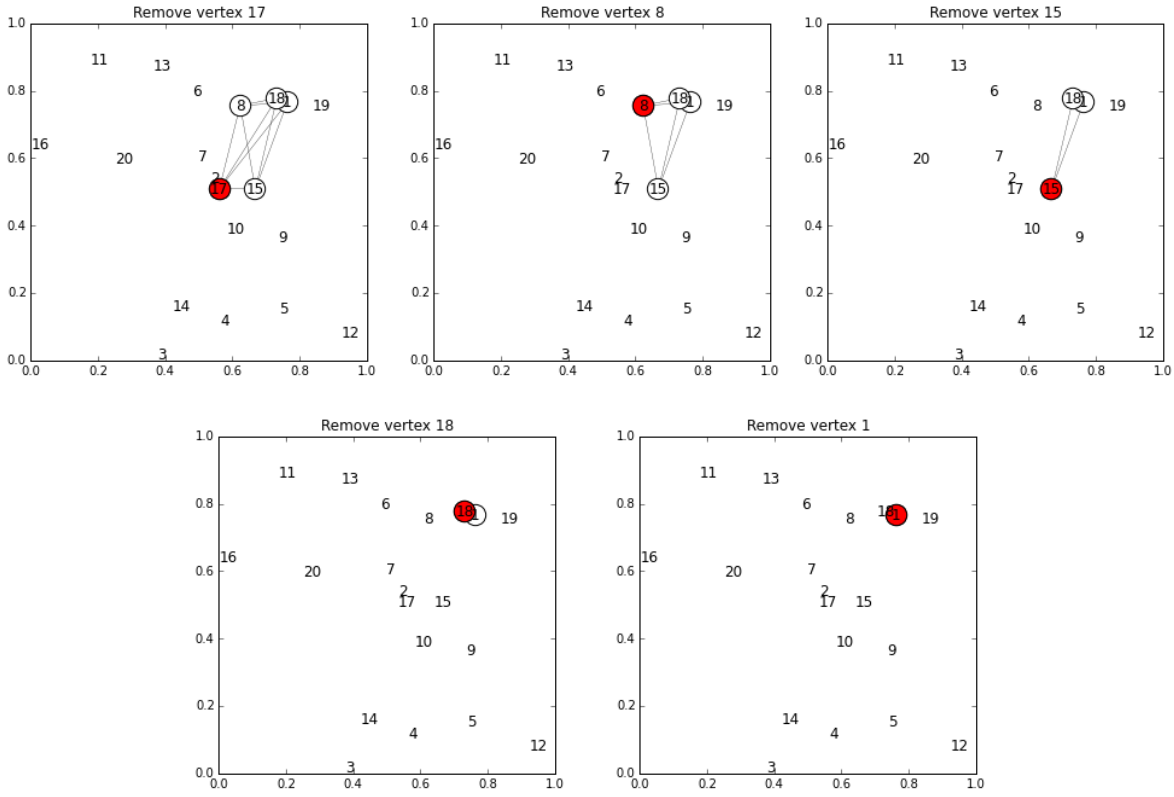

Remove vertex 20


Remove vertex 19


Remove vertex 6


Remove vertex 7

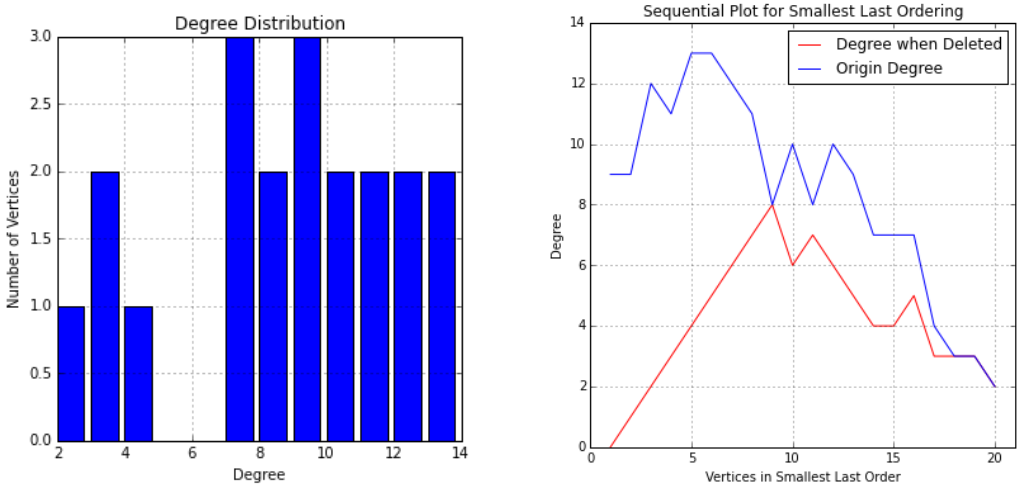

Remove vertex 2



Afterwards, vertices are in smallest last order, as shown in the table below (2nd row).

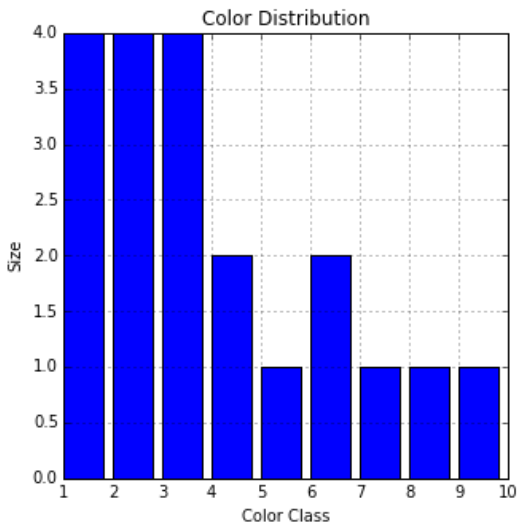
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	18	15	8	17	2	7	6	19	20	13	10	9	14	5	4	11	12	3	16

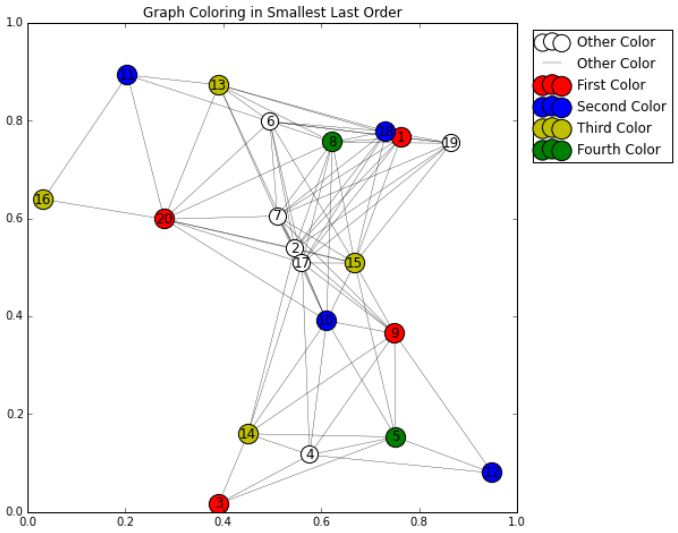
Degree distribution, and the sequential plot of origin degrees and degrees when deleted are also shown.



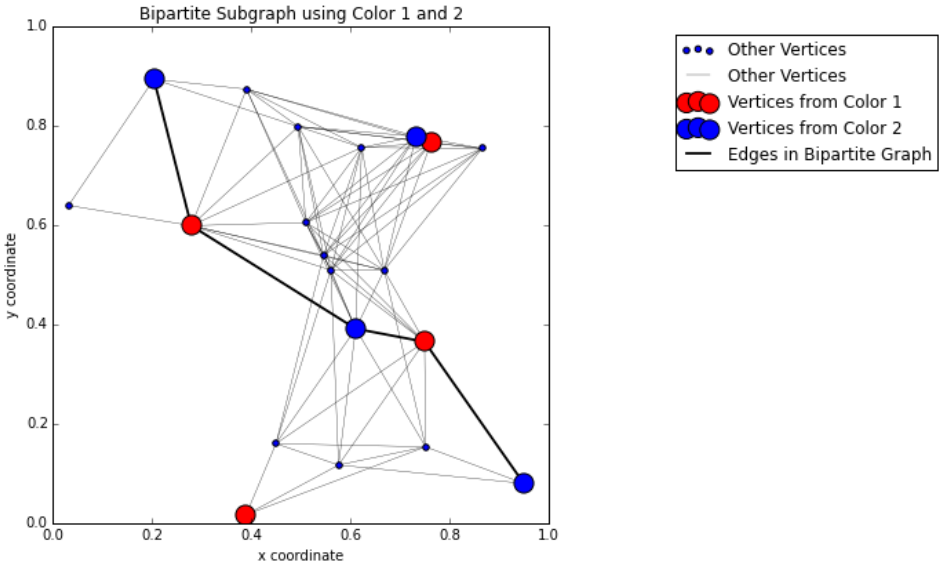
From the degree distribution plot, we could see that minimum degree is 2, maximum degree is 13, and average degree is 8. According to the sequential plot, the last 9 vertices are connected to each other. So the terminal clique size is 9. As a result, 9 color classes are required to color this graph.

Color distribution, and the colored graph with the first 4 colors are shown below.

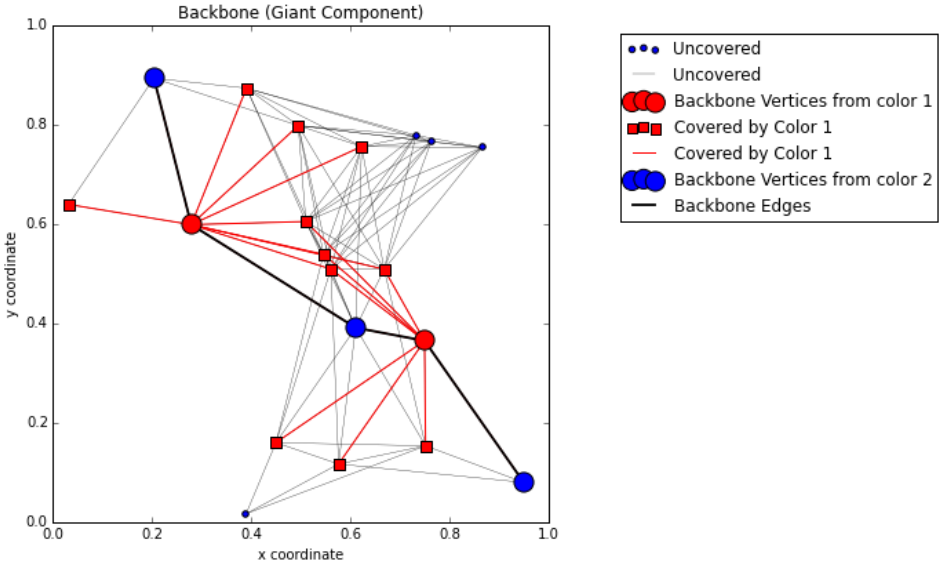




Finally, the bipartite subgraph using the first 2 colors is displayed. Obviously, the largest component of this bipartite subgraph is the one from top left corner to bottom right corner. It contains 2 vertices from color1, 3 vertices from color 2. This giant component could be the backbone of this graph.



Using the giant component as backbone, vertices dominated by color 1 are also selected and plotted. As shown in the plot, 16 out of 20 vertices are connected to vertices in color 1. Domination percentage is 80%. The coverage is not good, because the average degree of this graph is only 8. In the benchmark graphs with degrees more than 30, the percentage could be much higher. Besides, it is obvious that the backbone subgraph is planar.



4.5 Algorithm Effectiveness

For all benchmark graphs with no more than 16000 vertices, total run time is less than 30 seconds. For those two graphs with 64000 vertices, total rum time is less than 5 minutes.

Besides, the most time consuming part in this project is the one to transform RGG into adjacency list form. Other algorithms, including ordering, coloring and backbone identification, take no more than 10 seconds. The reason is that it takes $O(n^2)$ time to generate the adjacency lists. Using improved algorithms, like planar sweep or cell method, could reduce the time by a small fraction. But the asymptotic time bound is still $O(n^2)$. This could be the bottle neck of this project.

Run time for benchmark graphs:

Benchmark Graph	Run time
1, 2, 3, 4, 6, 7, 8, 9	< 30 seconds
5, 10	< 5 minutes

Algorithm Effectiveness Conclusion

Implementation	**Evaluation**
RGG generation: Use random generator for coordinates; transform into adjacency lists with planar sweep method	It could reduce the actual run time. And it could be further improved by using cell method.
Smallest-last ordering	This is implemented in a linear time bound O(V+E). It is an efficient way to order vertices of a RGG for coloring.
Graph Coloring	This is also implemented in a linear time bound. Greedy paradigm is applied to select the smallest available color value to improve efficiency.
Backbone Selection: The first 4 color classes are paired into 6 possible bipartite subgraphs. For each graph, using depth first search to find the largest component, which is selected as backbone.	After graph coloring, all vertices are distributed in different color classes. So it is easy and efficient to combine different color classes into bipartite subgraphs. The DFS is also fast in finding out the largest component by starting from the vertex with largest degree.

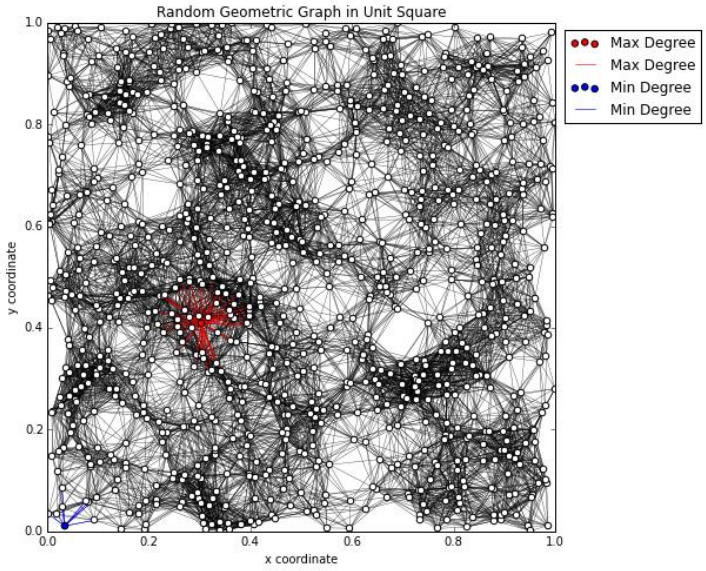
Implementation of these algorithms in Python are included in another file.

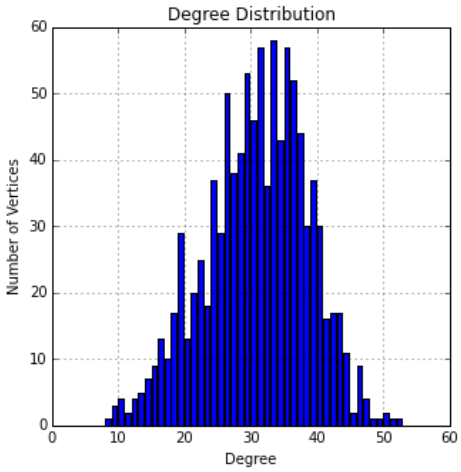
5. Benchmark Result Summary and Display

5.1 Benchmark 1

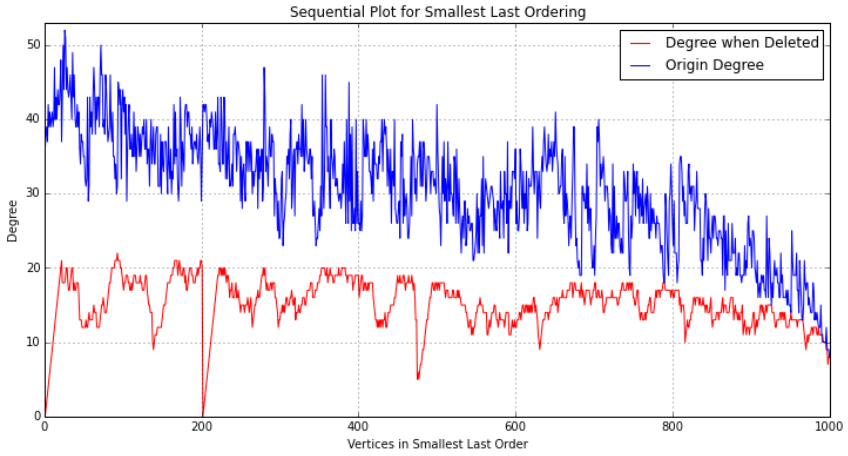
This graph contains 1000 vertices with an estimated average degree of 30.

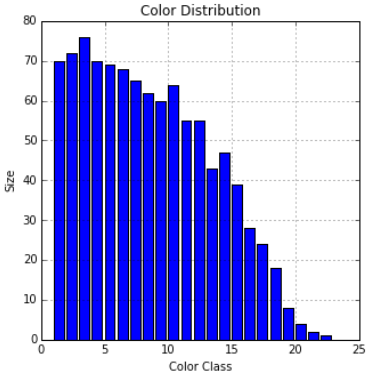
Random geometric graph (vertex with minimum degree is at left bottom corner) and degree distribution:



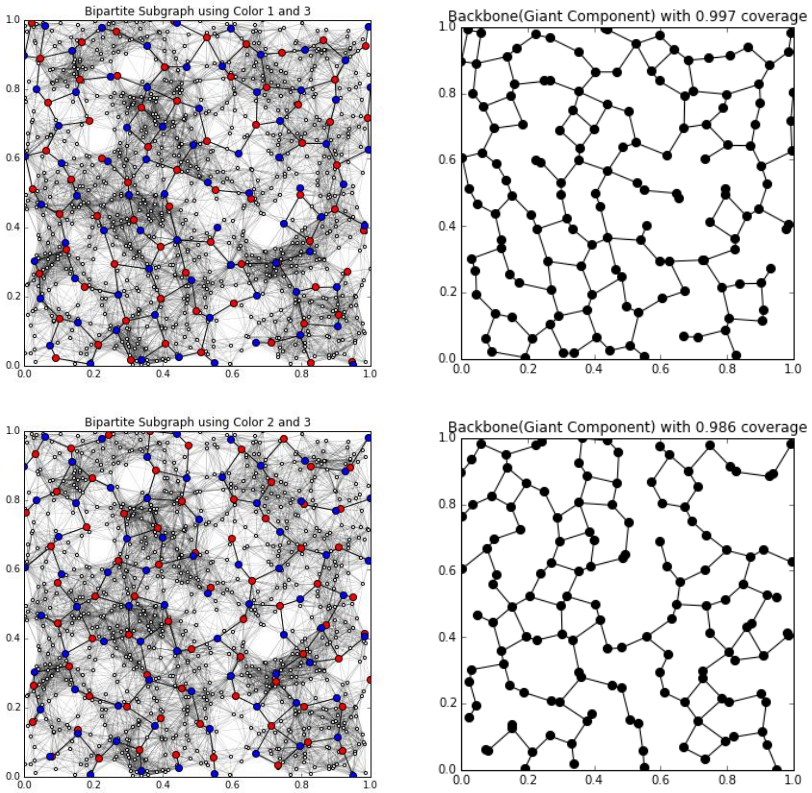


Sequential plot, and color distribution:





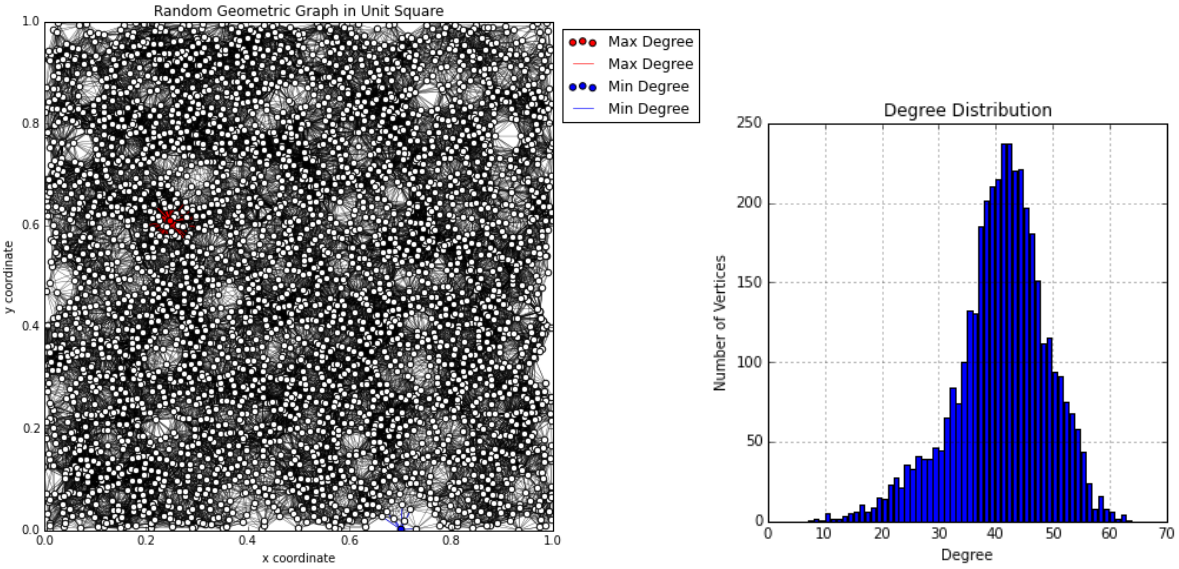
Finally, the bipartite subgraph and backbone are shown. Among all 6 possible bipartite subgraphs, the one using vertices from color 1 and 3 has the most edges. The subgraph with color 2 and 3 has the second most edges. They are selected as backbones and displayed.



5.2 Benchmark 2

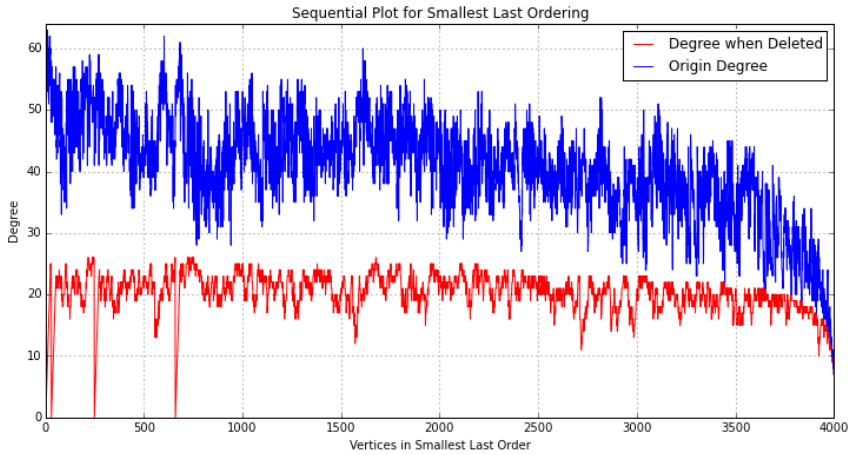
This graph contains 4000 vertices with an estimated average degree of 40.

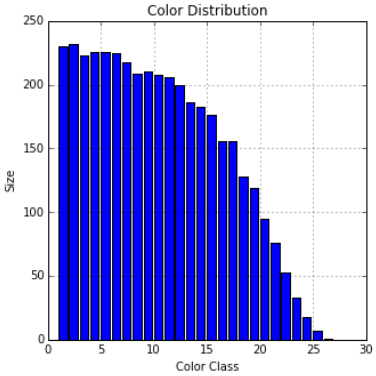
Random geometric graph and degree distribution:



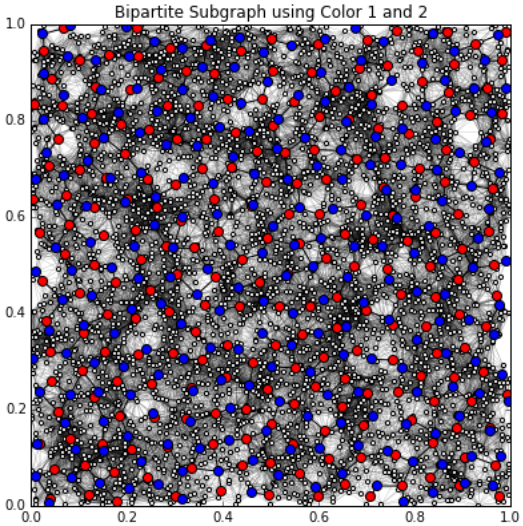
The vertex with maximum degree is at about (0.25, 0.6), and the vertex with minimum degree is at about (0.7, 0.01).

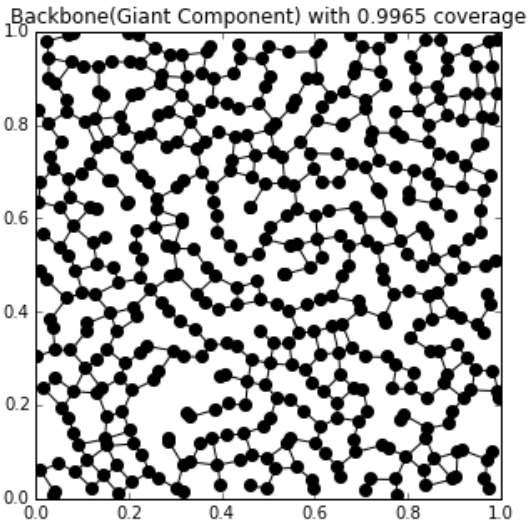
Sequential plot, and color distribution:

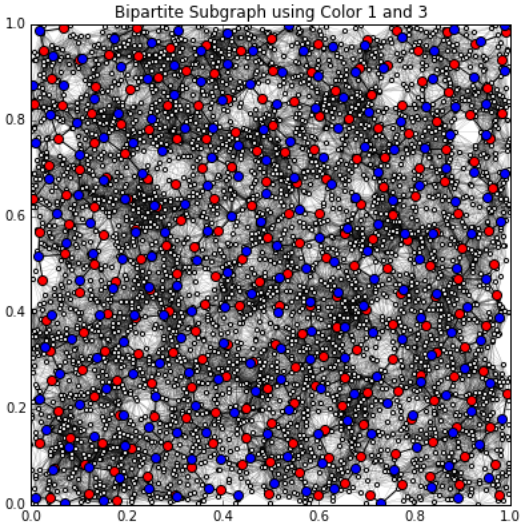


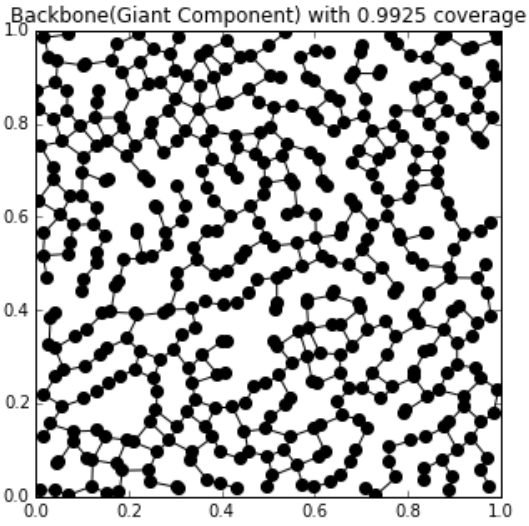


Bipartite Subgraph and backbone:





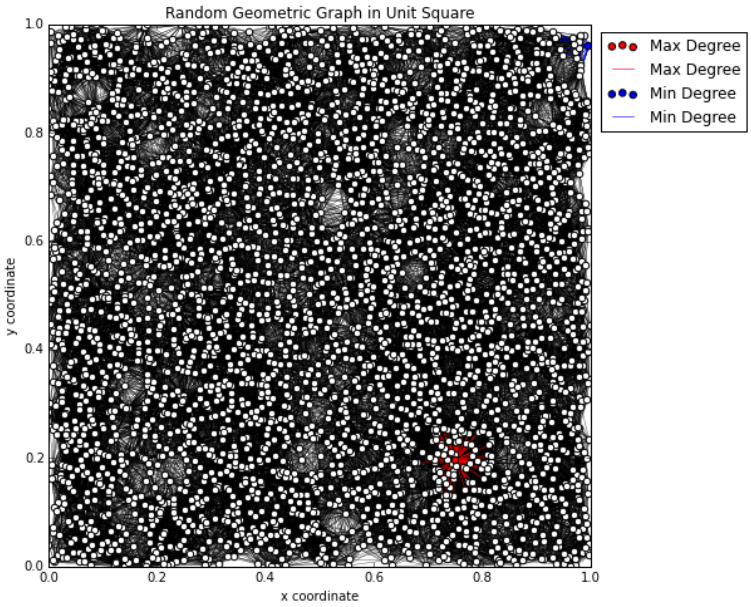


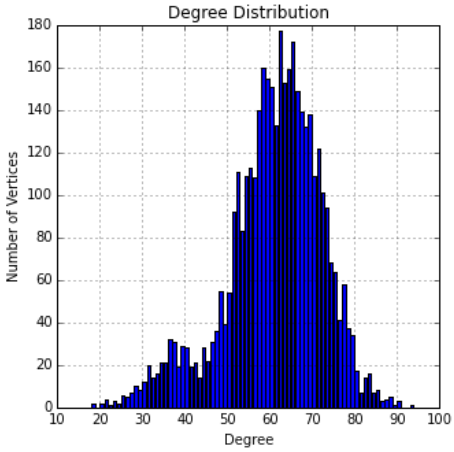


5.3 Benchmark 3

This graph contains 4000 vertices with an estimated average degree of 60.

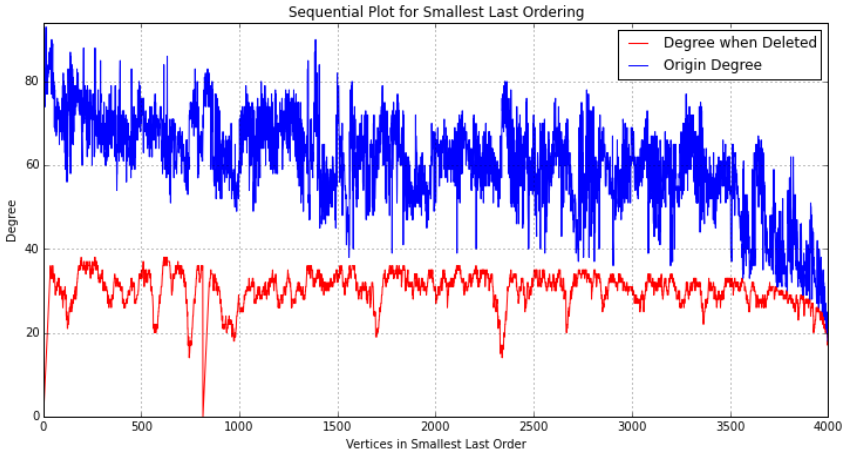
Random geometric graph and degree distribution:

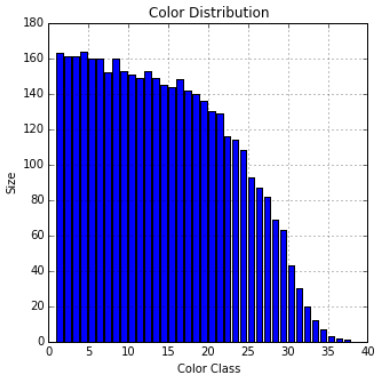




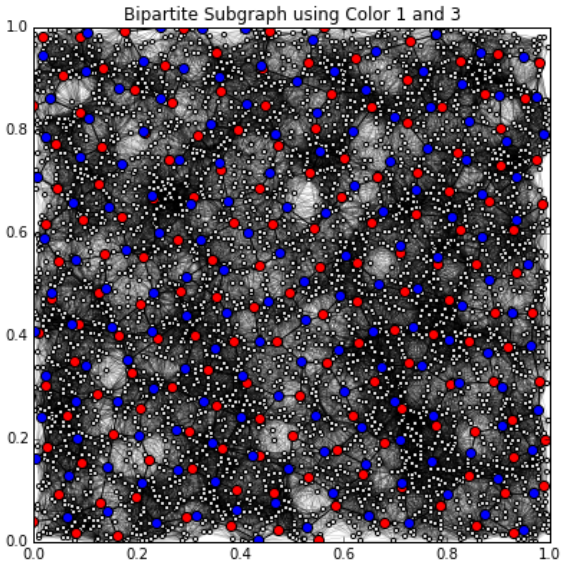
The vertex with maximum degree is at about (0.8, 0.2), and the vertex with minimum degree is at the right top corner.

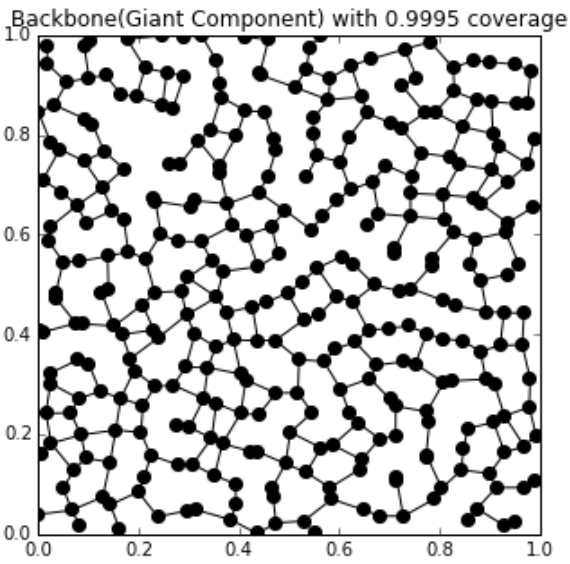
Sequential plot, and color distribution:

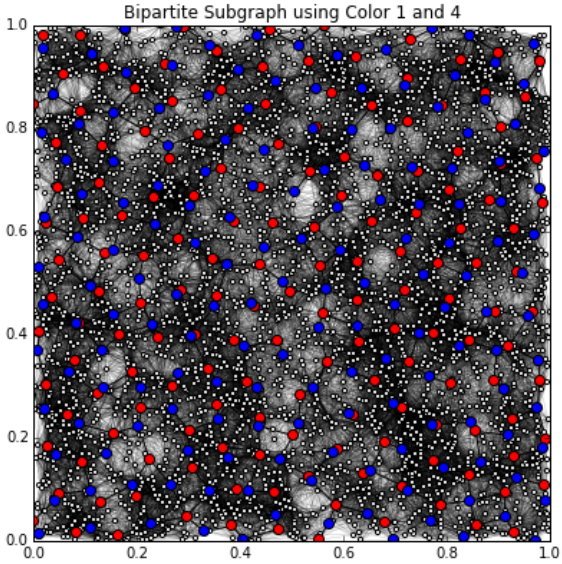


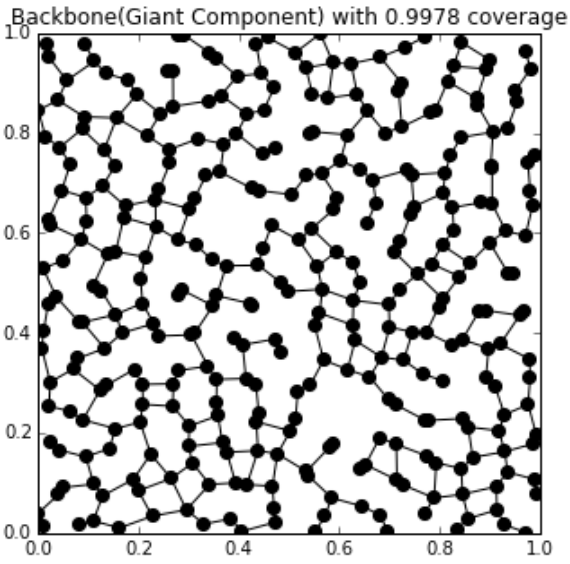


Bipartite Subgraph and backbone:





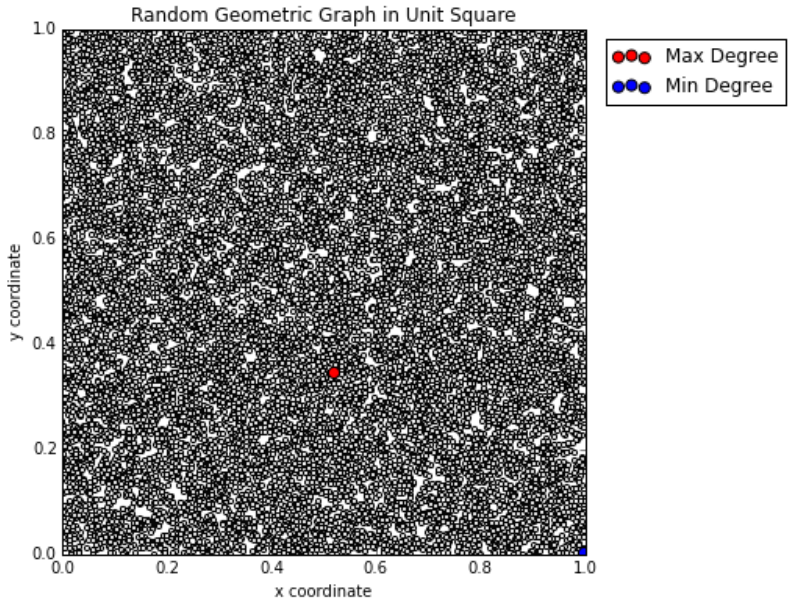


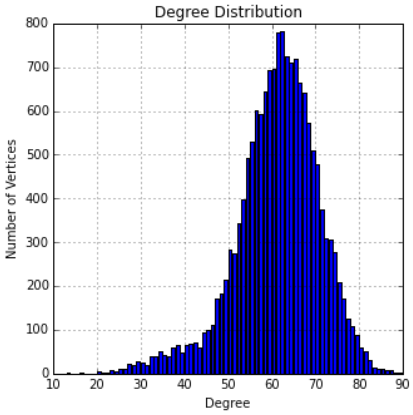


5.4 Benchmark 4

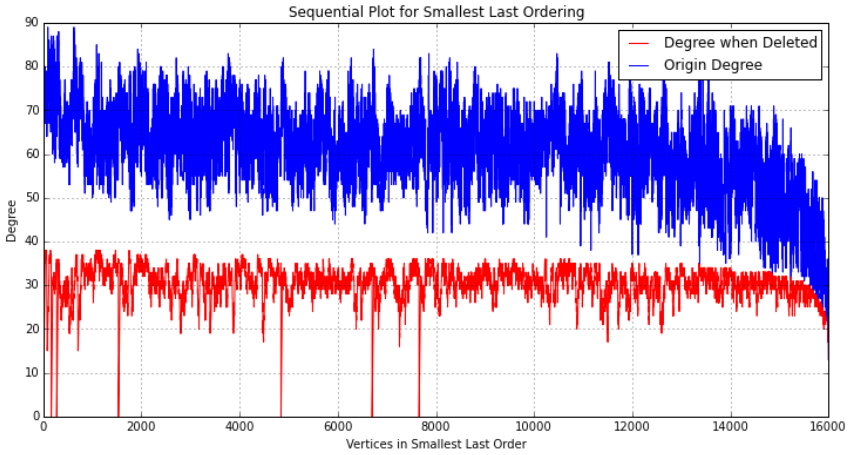
This graph contains 16000 vertices with an estimated average degree of 60.

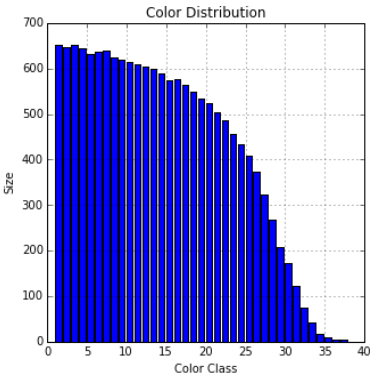
Random geometric graph and degree distribution:



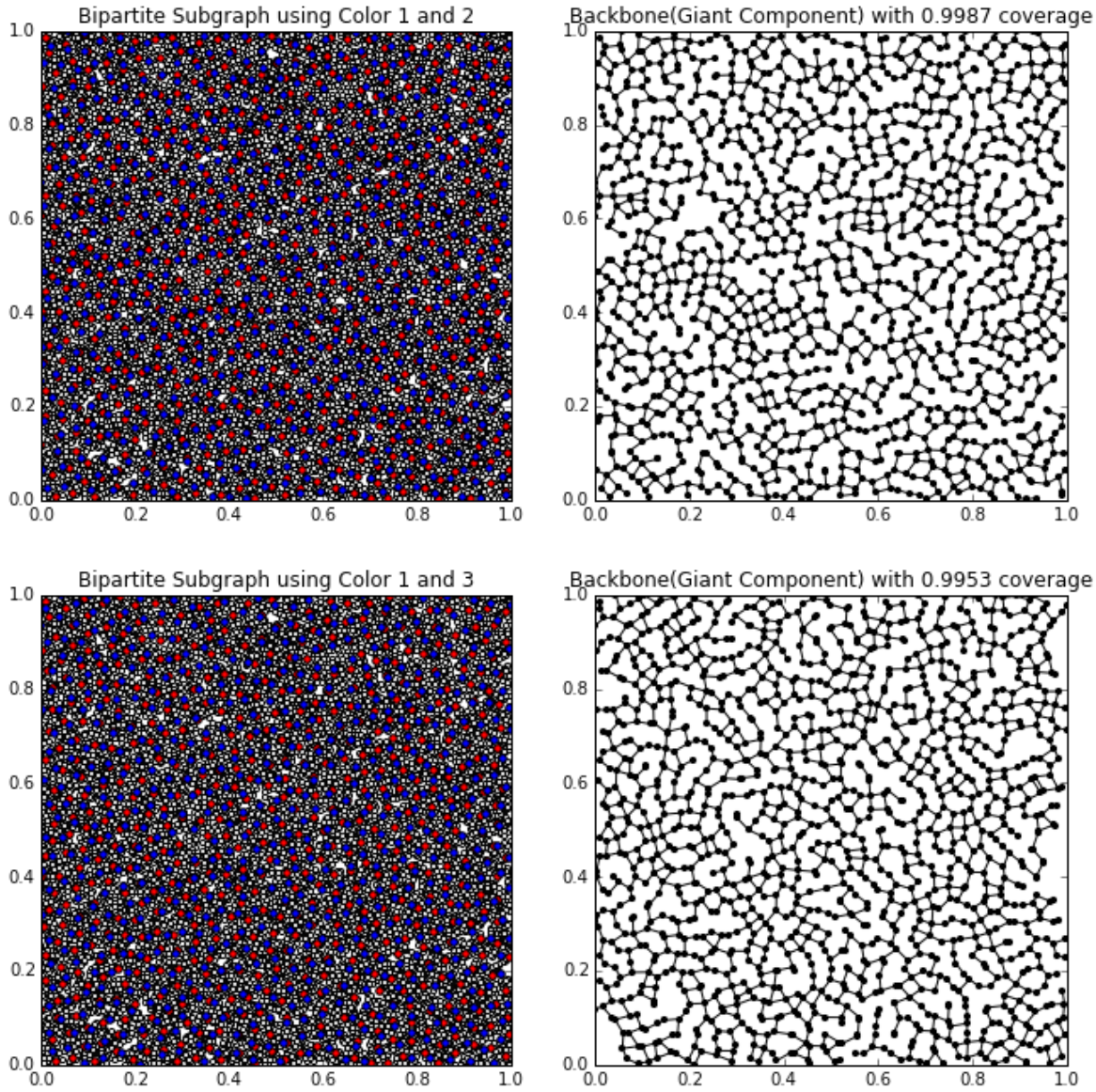


Sequential plot, and color distribution:





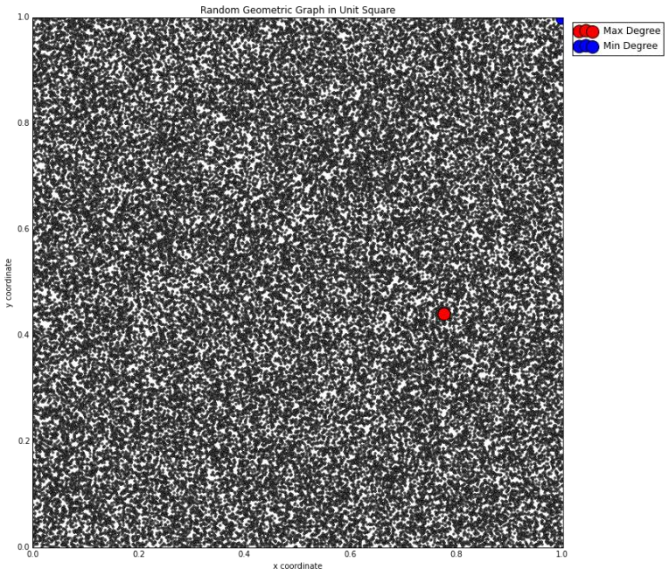
Bipartite Subgraph and backbone:

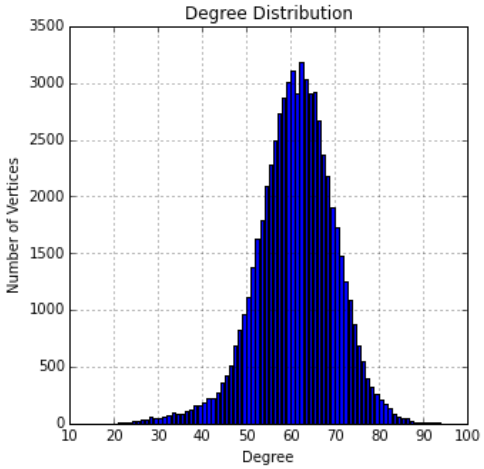


5.5 Benchmark 5

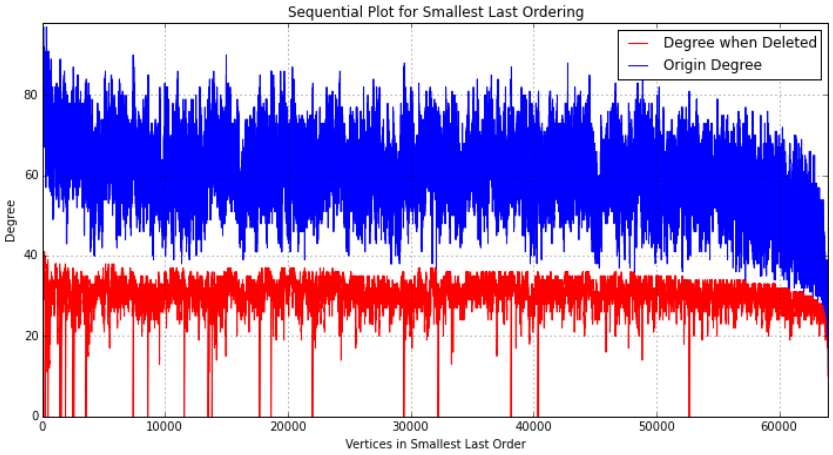
This graph contains 64000 vertices with an estimated average degree of 60.

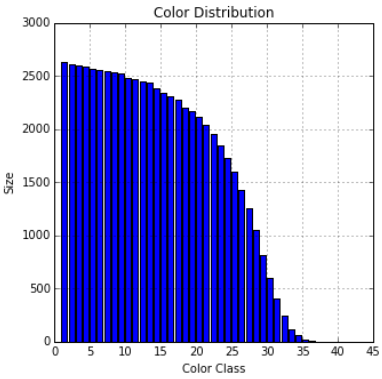
Random geometric graph and degree distribution: (the vertex with minimum degree is at the right top corner)



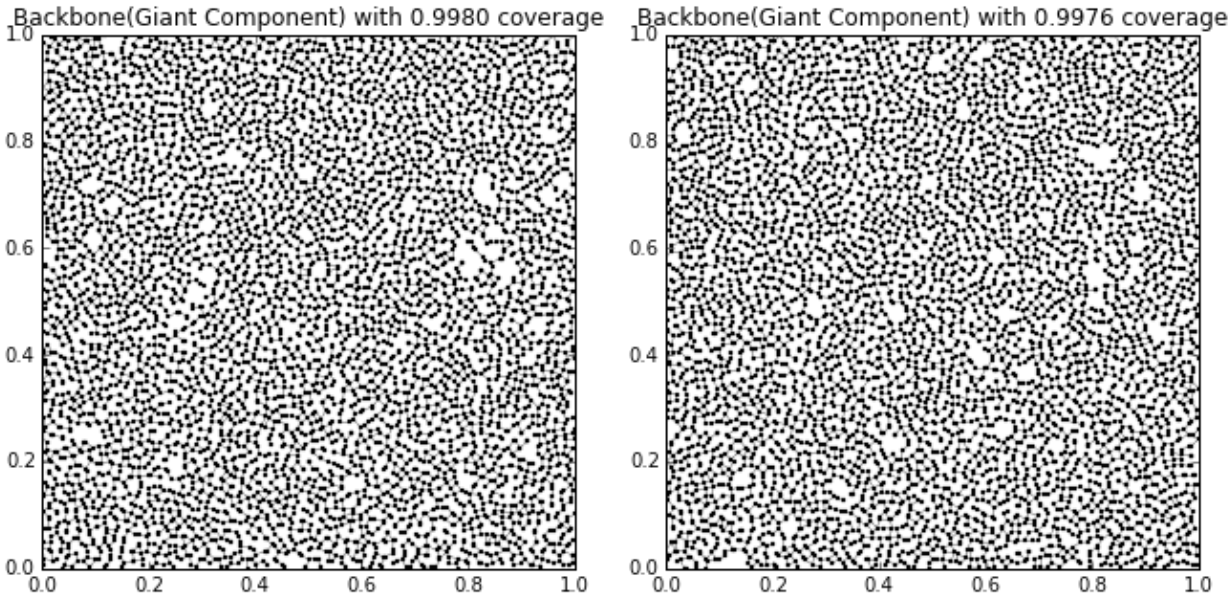


Sequential plot, and color distribution:





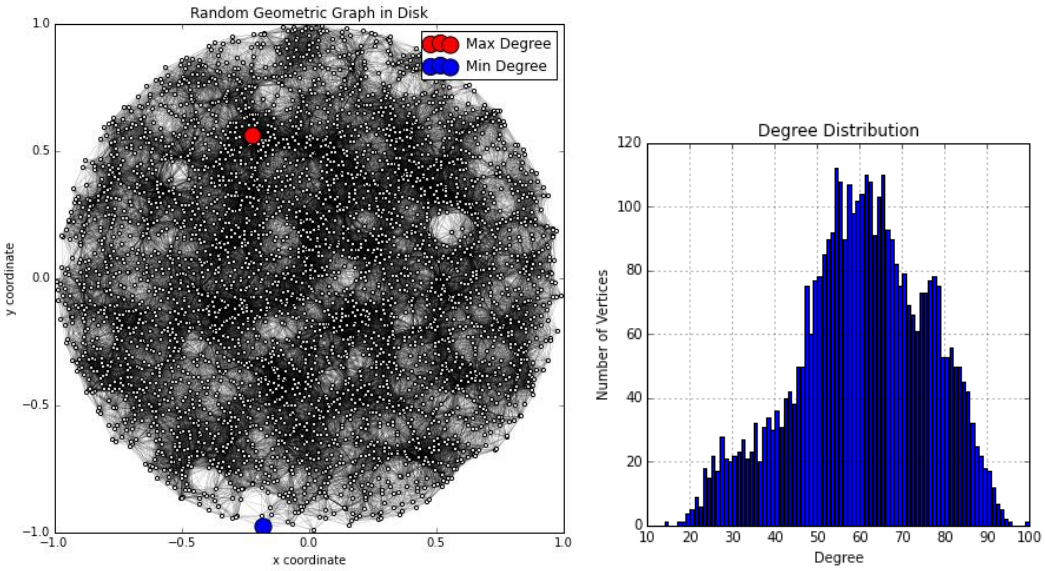
Backbone:



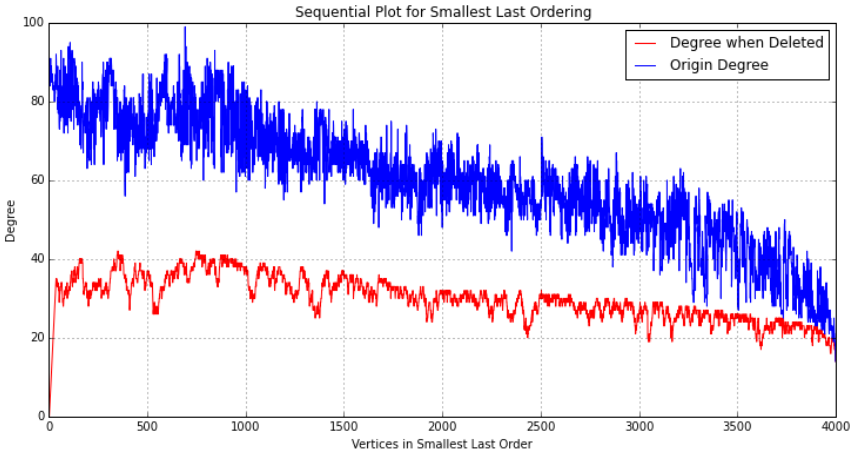
5.6 Benchmark 6

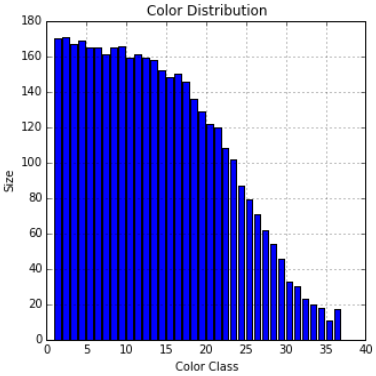
This graph contains 4000 vertices with an estimated average degree of 60.

Random geometric graph and degree distribution:

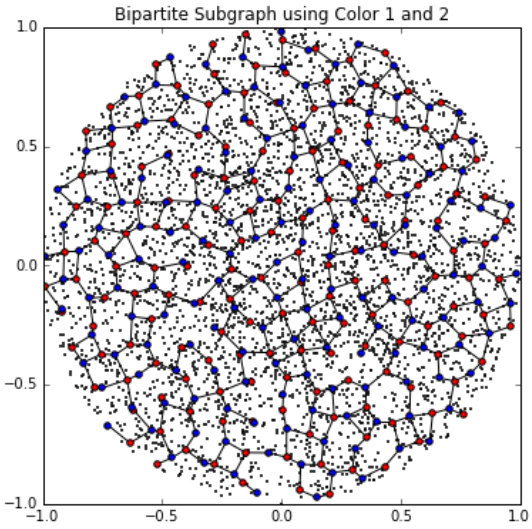


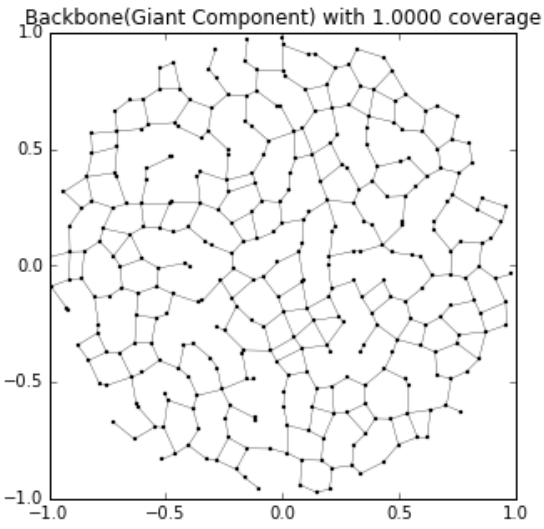
Sequential plot, and color distribution:

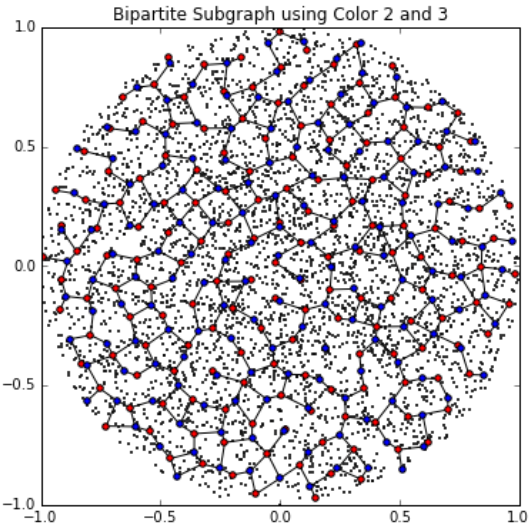


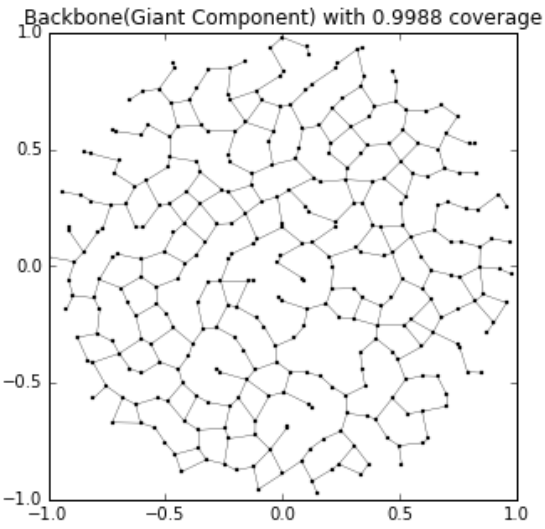


Bipartite Subgraph and backbone:





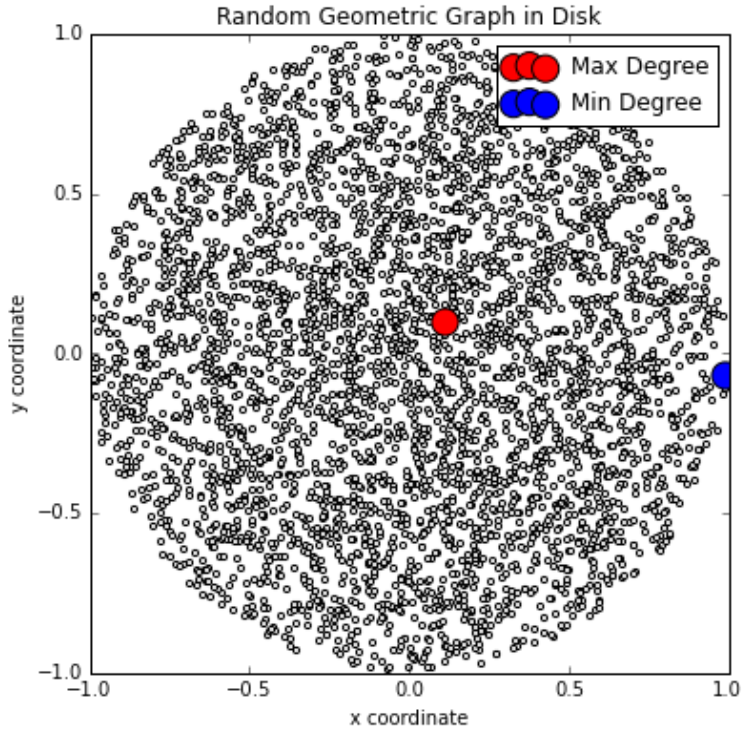


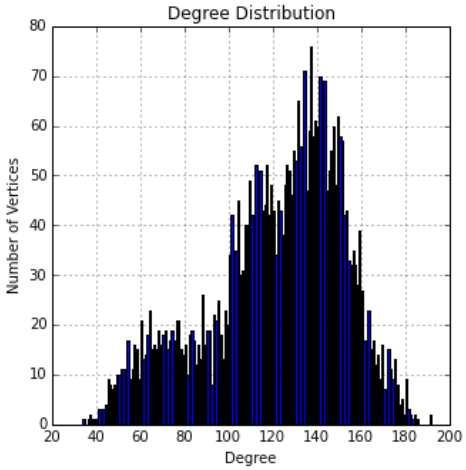


5.7 Benchmark 7

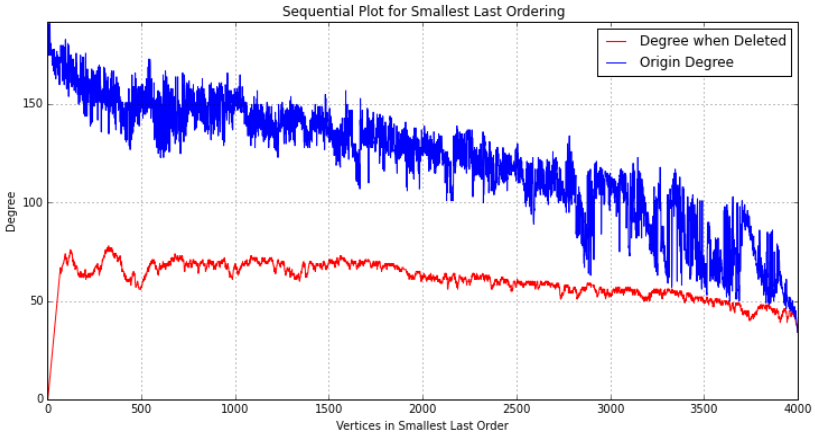
This graph contains 4000 vertices with an estimated average degree of 120.

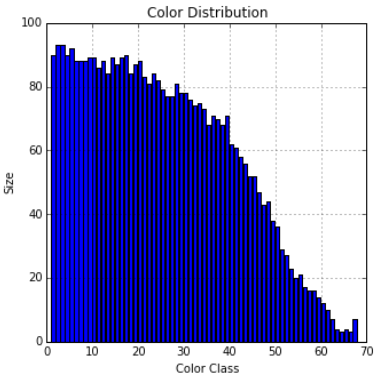
Random geometric graph and degree distribution:



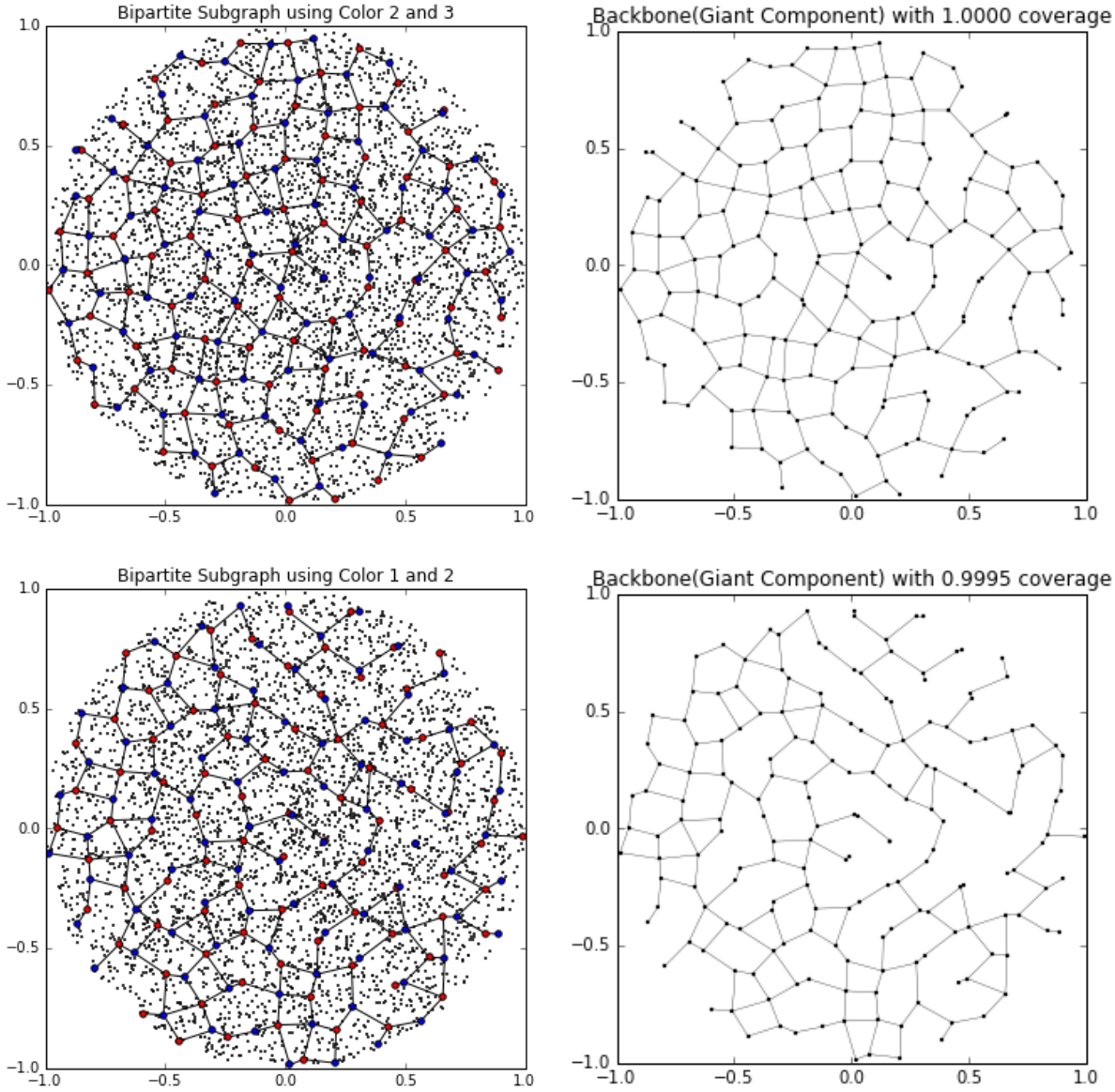


Sequential plot, and color distribution:





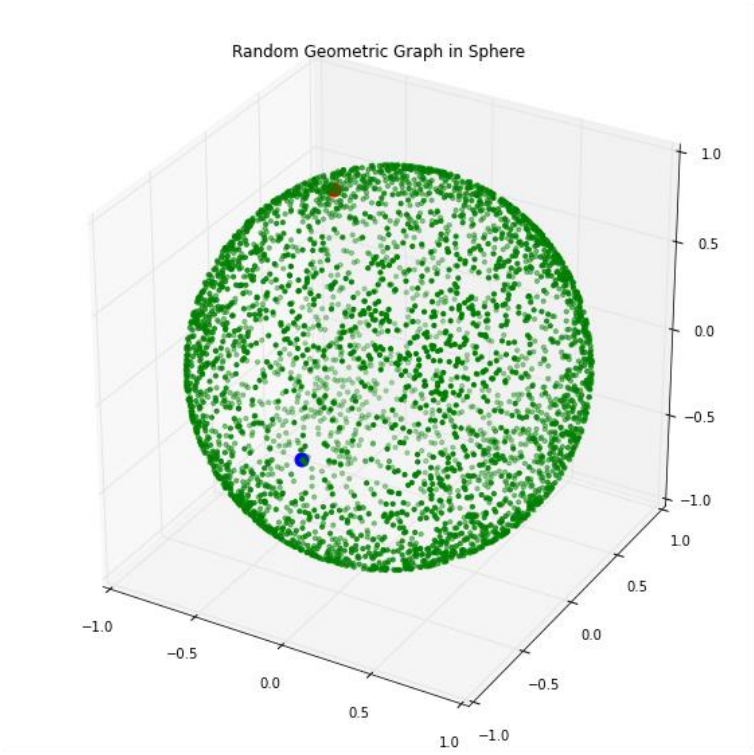
Bipartite Subgraph and backbone:

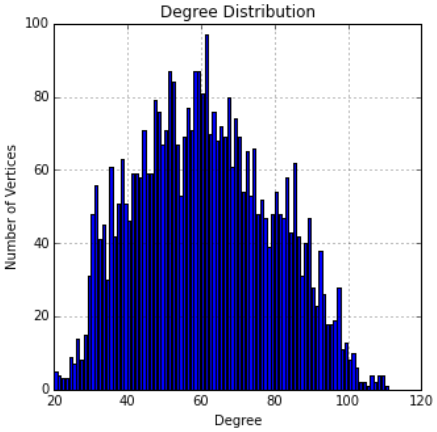


5.8 Benchmark 8

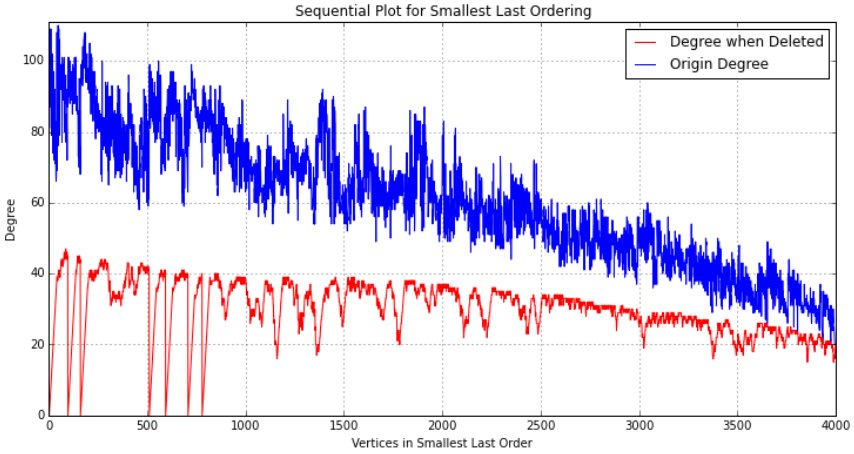
This graph contains 4000 vertices with an estimated average degree of 60.

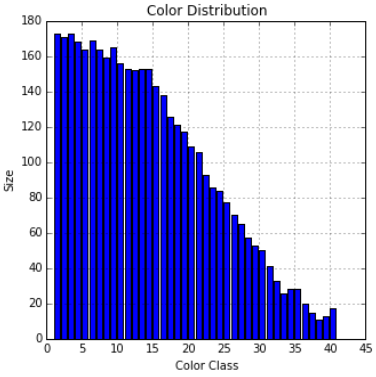
Random geometric graph and degree distribution:



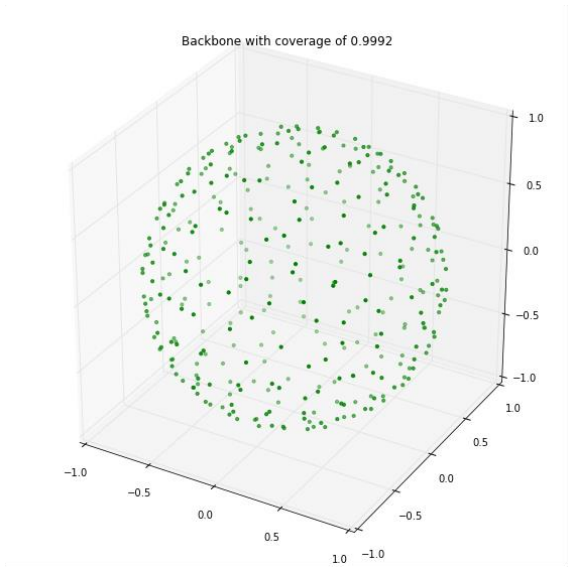


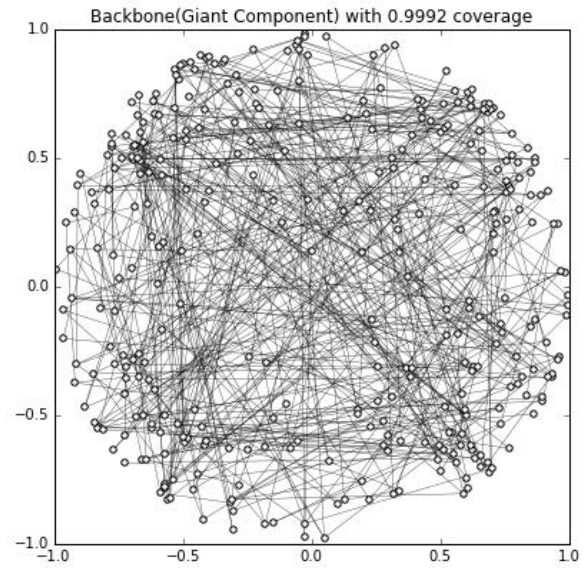
Sequential plot, and color distribution:





Backbone of this sphere graph and its projection on z>0 hemisphere:

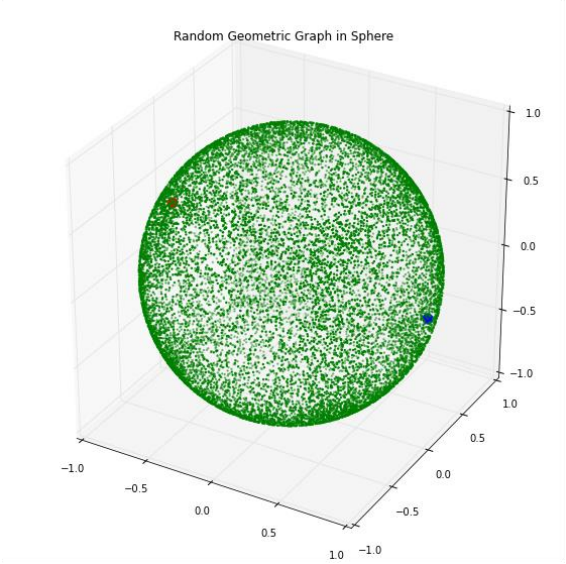


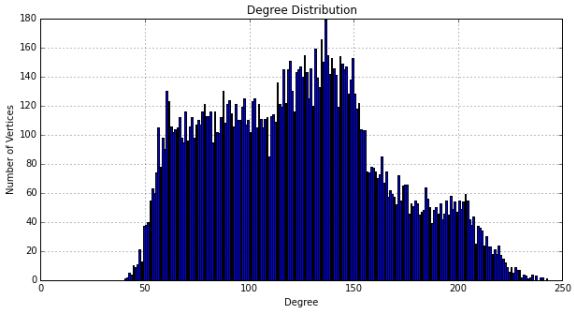


5.9 Benchmark 9

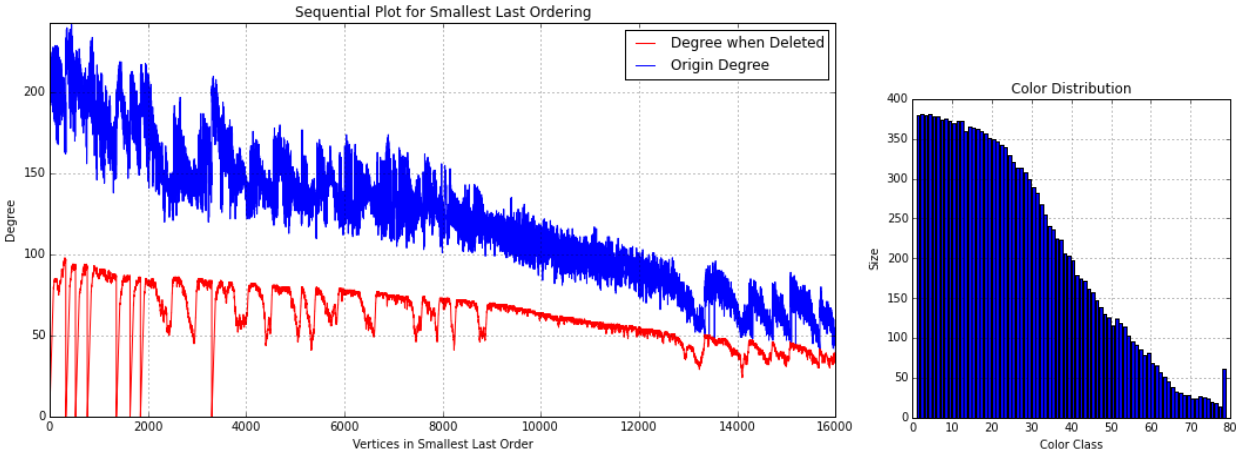
This graph contains 16000 vertices with an estimated average degree of 120.

Random geometric graph and degree distribution:

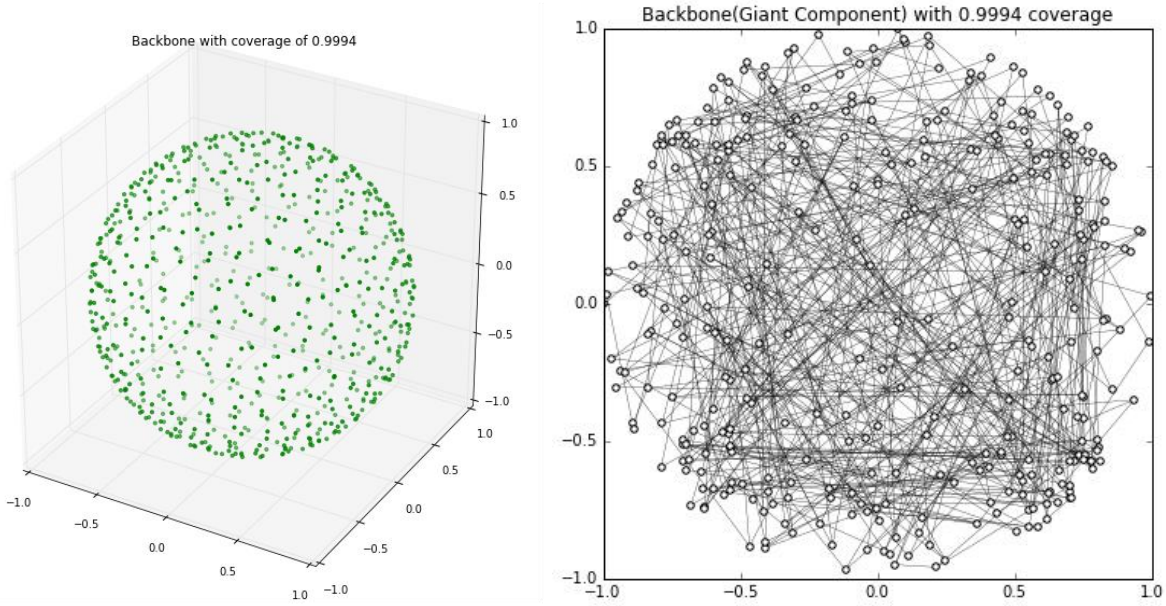




Sequential plot, and color distribution:



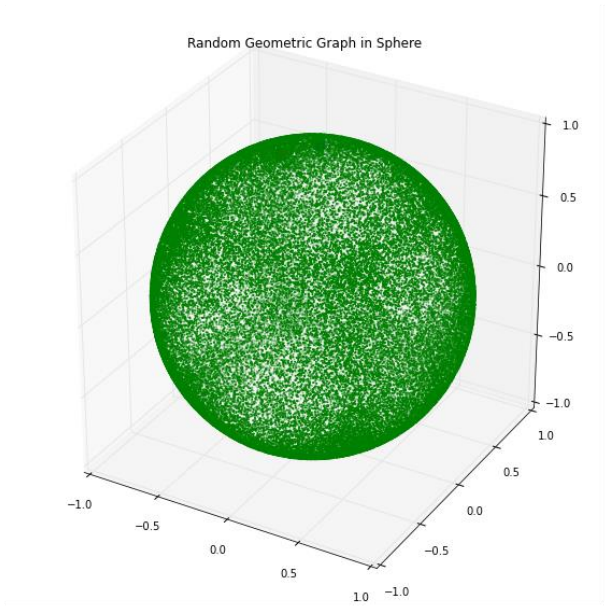
Backbone of this sphere graph and its projection on z>0 hemisphere:

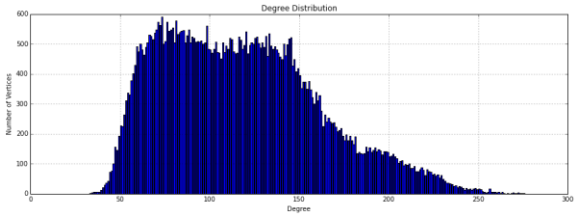


5.10 Benchmark 10

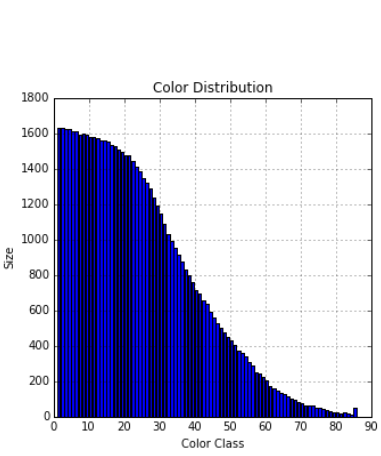
This graph contains 64000 vertices with an estimated average degree of 120.

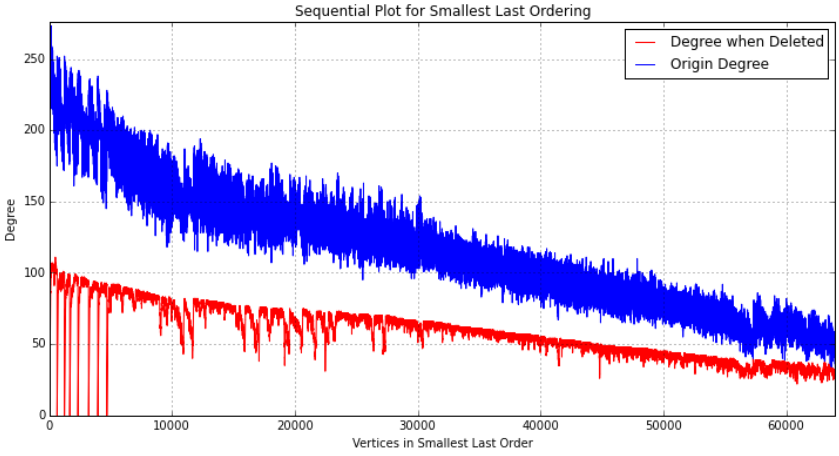
Random geometric graph and degree distribution:



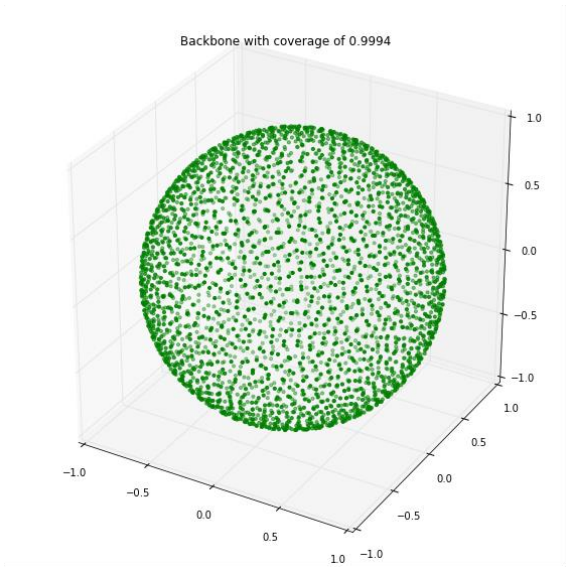


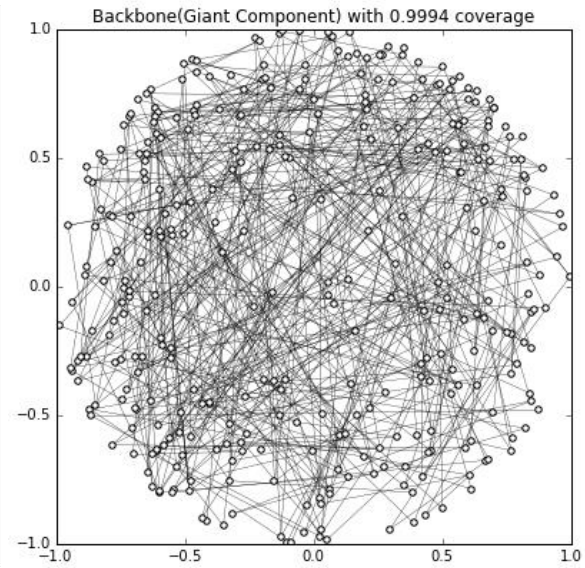
Sequential plot, and color distribution:





Backbone of this sphere graph and its projection on z>0 hemisphere:


Backbone with coverage of 0.9994
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0


Backbone(Giant Component) with 0.9994 coverage
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0