PARTITIONING RANDOM GEOMETRIC GRAPHS INTO BIPARTITE SUBGRAPHS

Modeling Backbone Determination in Wireless Sensor Networks

Sahil Johari
SMU ID: 47455841

EXECUTIVE SUMMARY

• Introduction and Summary

Wireless sensor networks (WSN) are spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, pressure, etc. and to cooperatively pass their data through the network to other locations [1]. Wireless Sensor networks have become a hot topic in research communities in recent years as improvements in technology and reductions in manufacturing costs have made their creation and deployment increasingly economically feasible. Technologies such as Wi-Fi, ZigBee and Bluetooth are all commonly used protocols in wireless sensor networks and have been subject to numerous implementation and hardware improvements over their lifetime resulting in sensors that are smaller, cheaper and more energy efficient [7].

In this project, we implement an algorithm that models ad-hoc networks of wireless sensors of varying scale across a variety of geographic areas. The project models these networks by creating random geometric graphs (RGG) of varying size, degree and type and interconnecting vertices if they are within a distance R of one another. In the context of wireless sensors R is the maximum broadcast range of any single sensor and thus determines how far away any two sensors must be to communicate. Determining connections between vertices in the graph is not a trivial task and can be a very expensive process if great care is not taken when doing so. This project makes use of several methods for decreasing the time and number of comparisons required to determine vertex adjacency which will be discussed in detail in a later section. We apply the Smallest Last Ordering algorithm [9] and a graph coloring procedure [10] [11] to identify high quality candidates for communication backbones in the graph [7]. A backbone is a bipartite subgraph of the overall RGG, or sensor network, that adequately covers all vertices that make up the overall network. The backbones must be bipartite because this ensures that no two connected vertices share the same "color" which practically translates to sensor broadcast frequency.

• Programming Environment Description

System information:

OS Name macOS High Sierra
Version 10.13
Processor 2.7 GHz Intel Core i5
Memory 8GB 1867 MHz DDR3
Graphics Intel Iris Graphics 6100 1536 MB

Programming Platform Information:

NetworkX

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.
With NetworkX you can load and store networks in standard and nonstandard data formats, generate many types of random and classic networks, analyze network structure, build network models, design new network algorithms, draw networks, and much more.
I chose NetworkX because of it's easy to use syntax and implementation with python. Furthermore, it provides a flexible visualization of the graphs in a minimal amount of time [2].

Python

Python is a powerful programming language that allows simple and flexible representations of networks, and clear and concise expressions of network algorithms (and other algorithms too). Python has a vibrant

and growing ecosystem of packages that NetworkX uses to provide more features such as numerical linear algebra and drawing. In addition, Python is also an excellent “glue” language for putting together pieces of software from other languages which allows reuse of legacy code and engineering of high-performance algorithms. I prefer python for this project as it allows me to write compact code and has a wide range of modules and libraries for a variety of operations, which makes the work easier and faster. Equally important, Python is free, well-supported, and a joy to use [2][5]. For this project, I am using Python 3.6 and Anaconda 3 as the interpreter.

• References

[1] https://en.wikipedia.org/wiki/Wireless_sensor_network

[2] https://networkx.github.io/

[3] http://pointclouds.org/documentation/tutorials/kdtree_search.php

[4] Zizhen Chen and David W. Matula, Bipartite Grid Partitioning of a Random Geometric Graph, 2017

[5] https://docs.python.org/3/tutorial/datastructures.html

[6] https://stackoverflow.com/a/22161588/8645818

[7] http://tylerspringer.com/finding-reliable-communication-backbones-in-a-wireless-sensor-network/

[8] https://en.wikipedia.org/wiki/K-d_tree

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

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

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

[12] Mahjoub, Dhia, and David W. Matula. "Employing (1− ε) dominating set partitions as backbones in wireless sensor networks." Proceedings of the Meeting on Algorithm Engineering & Experiments. Society for Industrial and Applied Mathematics, 2010.

REDUCTION TO PRACTICE

- ## Data structure Design:

List

The list is a most versatile datatype available in Python which can be written as a list of comma-separated values (items) between square brackets. Important thing about a list is that items in a list need not be of the same type. List has been used throughout the programming, primarily to store the vertices having minimum and maximum degree [5].

Dictionary

A dictionary stores data in the form of key-value pair. Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.
Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples [5].
Dictionary has been used to store the degree of each vertex generated in the RGG.

K-D Tree

A k-d tree, or k-dimensional tree, is a data structure used in computer science for organizing some number of points in a space with k dimensions. It is a binary search tree with other constraints imposed on it. K-d trees are very useful for range and nearest neighbor searches. K-d trees are a special case of binary space partitioning trees [3].
This data structure is the core of the entire algorithm developed for random graph generation, and provides a spectacular efficiency with a complexity of $O\ (log\ n)$ [8].

- ## Algorithm Descriptions:

RGG Generation Algorithm

Let $v1, v2, v3, \ldots\ldots, vn$ be the vertices of a graph. Each vertex v_i has a range represented as a circle of radius r and center v_i. We are required to plot vertices randomly over an area/topology of a unit square, and a unit circle such that they are distributed uniformly across the area.
Furthermore, any two vertices {u, v} must be connected iff their distance D (u, v) ≤ r [4].

Smallest Last Ordering

The Smallest Last Ordering is a way of sequencing the vertices in the graph such that vertex v_i has the minimum degree in the remaining subgraph of vertices $v1, v2, v3, \ldots\ldots, vk$ where k ≤ n. By ordering the vertices in this manner, we can easily color all the vertices in the Random Geometric Graph.

To find the Smallest Last Ordering of a set of vertices, we repeatedly apply the following steps until the graph is left with no vertices [9]:
1. Find the vertex with the minimum degree
2. Delete this vertex from the set in consideration
3. Update all previously adjacent vertices such that their degree is equal to *degree-1*
4. Store the deleted vertex in the last available position in the smallest last ordering stack

Graph Coloring Algorithm

Once we obtain the Smallest Last Ordering of the vertices, we can apply a graph coloring algorithm to it. This is a greedy coloring algorithm, often referred to as the "Grundy coloring algorithm" [11]. This algorithm assigns the lowest available color to a vertex being considered such that no vertex connected to the vertex under consideration is currently using that color.

Bipartite Backbone Selection

Finally, we generate bipartite backbones from the coloring information determined above. As previously mentioned, a backbone is a connect bipartite subgraph that will be the main stem for communication in the wireless sensor network. In the optimal case, every node will have access to a node in the backbone which is equivalent to the backbone having vertex coverage equal to 100%. It then stands to reason that the best metric for determining the quality of a selected backbone is its vertex coverage (also known as domination percentage) [12].

- Algorithm Engineering:

RGG Generation

Random Geometric Graphs can be generated using three approaches [4]:

1. Brute-force
2. Sweep Method
3. Cell Method

In my implementation, I have employed Brute-force method since it delivers the highest efficiency for the way my program is developed.

We begin by generating random Cartesian/ Polar coordinates [6] using the *random* module in python, and store them into a list. The list of coordinates is then fed to the K-d tree, which organizes them into a tree structure.

The K-d tree can search and return the coordinates which are ≤ r (radius) distance apart. This is done by providing *r* as an input to the tree, which then performs searching of coordinates by calculating the distance between each pair of them (brute-force). The output is a list of coordinates arranged in ordered-pairs, which makes it easy to connect them in a straightforward manner.

We also find the vertices with maximum and minimum degree, and highlight them in the output graph. This is achieved by using a dictionary which contains a collection of vertices for each unique degree value. We find the maximum degree and minimum degree values from the dictionary to obtain their respective coordinates and draw them on the graph.

The implemented algorithm basically involves Insertion and Search operations, and therefore the time complexity of this algorithm turns out to be $O(|V|\ log\ (|V|) + |E|)$.

Smallest Last Ordering

In this algorithm, the ordering list is built from the bottom of the list to the top. If we analyze each step of this process, we can see that the Smallest Last Ordering can be implemented in $O(|V| + |E|)$ time. However, this running time is entirely dependent on how efficiently can we find the minimum degree vertex. To do this, we first partition each vertex into buckets by their degree. This can be done in $O(|V|)$ time and does not require an initial sort to be performed. In my implementation, this is achieved with the following logic:

```
for key, val in degree.items():
    degreeBucket.setdefault(val, []).append(key)
```

The “degreeBucket” structure shown in the above code is a dictionary of queues. Queues are used to ensure that the nodes are being processed in the correct.
To begin the Smallest Last Ordering process, we first select and delete the vertex with minimum degree and then popping the first element off that queue. This is always a constant time operation as empty buckets are deleted as they become empty. Once found, the minimum degree vertex is deleted from the bucket and the degrees of its adjacent vertices are updated. We simply iterate through the deleted node’s neighbors in the adjacency list and we reduce their degree by one if they have not yet been deleted. When there is only 1 bucket remaining, its contents will give us the terminal clique as they must all have the same degree.

We perform degree reduction by moving the affected vertex down by 1 to the next lowest bucket. This can be done for each adjacent vertex of the deleted vertex with the following logic:

```
adjacency_list[item].remove(popped_node)
d = degree[item]
degreeBucket[d].remove(item)

if not degreeBucket[d]:
    del degreeBucket[d]

degreeBucket.setdefault(d-1, []).append(item)
degree[item] -= 1
```

We are also pushing the deleted node in the smallest last ordering in a stack so that they can later be accessed from back to front, in constant time.

In this algorithm, we can observe that the partition into buckets takes $O(|V|)$ time. Finding the minimum degree vertex, deleting it and updating its adjacent vertices is done in nearly constant time. This algorithm runs repeatedly until all vertices have been processed, which is done in $O(|E|)$ time. Thus, the entire Smallest Last Ordering process runs in $O(|V| + |E|)$ time as we visit each node and each edge exactly once. This process runs in $O(|V|)$ extra space as we will need to store the resulting order of the vertices in a list.

Graph Coloring Algorithm
In this algorithm, we assign colors to the vertices in the Smallest Last Order beginning from the last vertex that was deleted – which is described above, and continue to iterate over the Smallest Last Order coloring nodes. If at any point all the available colors have been used up, we generate a new color and assign it to the vertex under consideration. Each time a new color is added, it is currently the maximum value color. For each new vertex to be colored, we access the list of its adjacent vertices from the adjacency list dictionary in constant time and get the colors of each of these vertices from the color-map dictionary. If any of its neighbors are not currently colored, that neighbor is not considered. The lowest color that is not currently assigned to one of these neighbors is assigned to the vertex under consideration.

Since we are going through each edge and each vertex exactly once and we are only adding new colors as they are needed, we can see that this coloring algorithm runs in $O\ (|V| + |E|)$ time.
The graph coloring algorithm runs $O(c)$ space complexity where *c* is the maximum number of colors required to color the graph.

Bipartite Backbone Selection

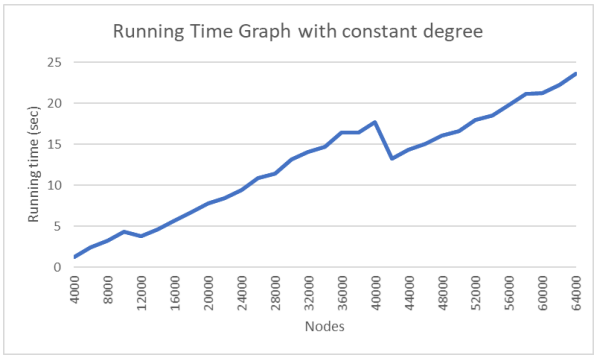
Backbone generation occurs by selecting the largest color classes and joining their vertices according the same rule as for the entire RGG (vertices are connected if they are less than or equal to R from each other) [4]. Because we used the Smallest Last Order to color our graph we can say with a high degree of certainty that the largest color class will likely be one of the lowest colors used.

We require the top two backbones that will be generated from the top 4 color classes. Because there are 4 colors, there are 6 possible bipartite combinations of colors and each must be tested. Getting the vertices that correspond to each color class is extremely fast as we can make use of our coloring map data structure that maps each vertex to its color in near constant time.

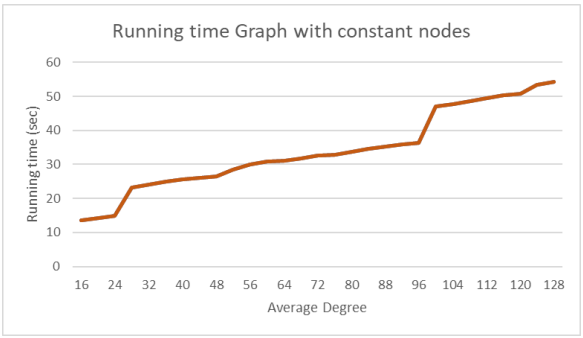
For each of the 6 combinations generated, we must analyze the quality of the resulting backbone. Recall that a backbone must be connected to be considered a backbone, so the first step is to determine the size of the largest connected component in the graph. We do this by performing a component search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

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

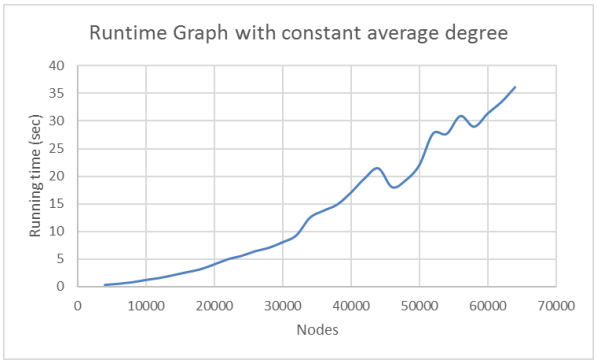
- Running Time Plots:



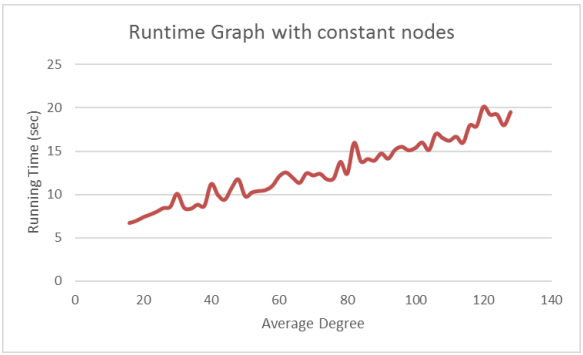
(a) Random Geometric Graph generation



(b) Random Geometric Graph generation



(c) Smallest last ordering and Graph coloring



(d) Smallest last ordering and Graph coloring

Fig 1: Running Time graph when the average degree is kept constant.

Fig 2: Running Time graph when the number of nodes is kept constant.

- Verification Walkthrough:

Smallest Last Ordering

Now that we have a thorough understanding of how each algorithm for smallest last ordering and graph coloring, let's examine the entire simulation on a sample graph. The following walkthrough is implemented on a random geometric graph on unit square with nodes = 20 and R = 0.40 maximum connection distance. This graph has 74 distinct pair-wise edges, a minimum degree of 2, a maximum degree of 11 and average degree of 7.4.

We begin with smallest last ordering as it forms the basis for all subsequent operations performed in the simulation. The first figure in the series, labeled with a "1" in the bottom right corner is the initial state of the graph before any deletions have been performed. The vertex with the minimum current degree is outlined with a blue circle and is the next vertex to be deleted in all figures in the series. The vertex outlined in figure 1 is the overall minimum degree vertex in the graph and will be the first vertex that is deleted, in this case that degree is 2. Because this vertex is the first to be deleted, it will be the last vertex in the final smallest last ordering [7].

We take 16 vertices in a unit square as an example for walkthrough the entire project step by step.

$$Number\ of\ vertices\ =\ 20;\ r\ =\ 0.4$$

The degree list in this case will be:

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

The adjacency list will be:

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

Now, step by step, all the vertices will be deleted and added to a new list in smallest last manner. First, the vertex with lowest degree will be deleted from the degree list and added to the last available position of the smallest last order list. The degree list and adjacency list will be appropriately altered.

We delete our first node. The node with the smallest degree i.e. at the top of our degree list is node 5. We delete vertex 5 from the degree list and update adjacency list accordingly:

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

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

Since vertex 5 had degree 2, the vertices 18 and 15 were updated in the adjacency list.
The new list of smallest last ordering is as follows:

Node																			5
Degree when deleted																			2

After all the vertices have been deleted from the degree list as per the steps discussed, we get a smallest order list that looks like this:

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

This is the result of smallest last ordering algorithm.

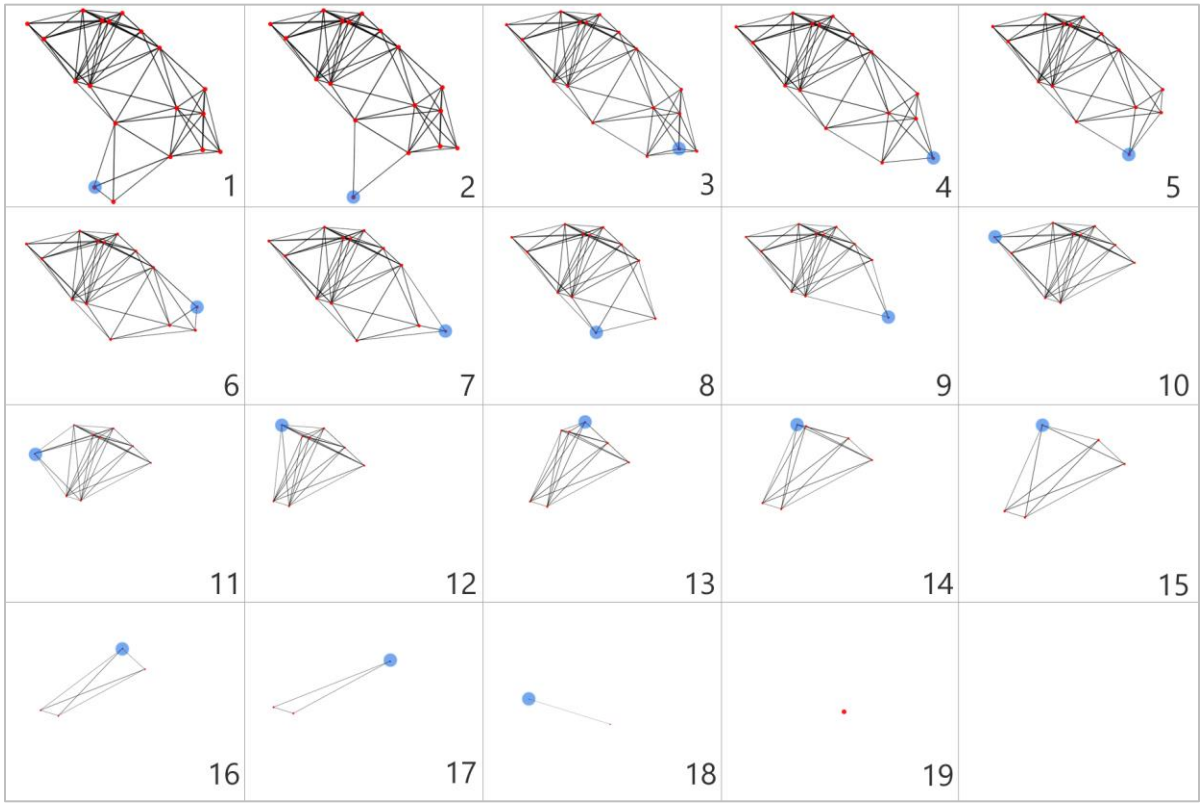


Fig 3: Sequence of Smallest last order on a random geometric graph

Figure 12 in Fig 3 shows the terminal clique, in which the remaining vertices have the same degree. From this point on each remaining vertex will continue to have consistent degrees after each subsequent deletion. By examining the series of figures above, we can see that vertices of minimum degree are being deleted correctly in sequence until no vertices remain. Furthermore, our resulting smallest last ordering is correct and can be found in $O(|V| + |E|)$ time.

Graph Coloring Algorithm

The next step in the simulation is to color the graph by using the smallest last ordering as the input determined in the process above. In the following figures, the next vertex to be colored (the next vertex in the smallest last ordering) is highlighted with a red circle. Note that the first vertex to be colored is the last node we deleted in the process above, and that each following vertex will be colored in the reverse order that we deleted them. Each vertex is assigned the "lowest" available color that is not already assigned to one of its adjacent vertices [7].

Continuing with the example used in the previous section, we will take the output of Smallest last ordering algorithm as the input for Graph coloring. Coloring begins by assigning a color to the first node in the above list, which is 14. We assign C0 as the color to it. For the next vertex in the list, we check whether the adjacent list of that vertex has any vertices with color C0. Since the next vertex 1 is adjacent to vertex 14, we can assign the color C1 to it. Vertex 17 is assigned C2 as it is adjacent to vertex 14. We repeat this procedure for all the vertices in the list and obtain a color mapping as given below:

Node	14	1	17	3	16	19	8	10	4	11	18	6	9	0	7	12	15	2	13	5
Colors assigned	C0	C1	C2	C3	C1	C3	C4	C4	C4	C5	C3	C0	C2	C4	C5	C2	C5	C0	C6	C6

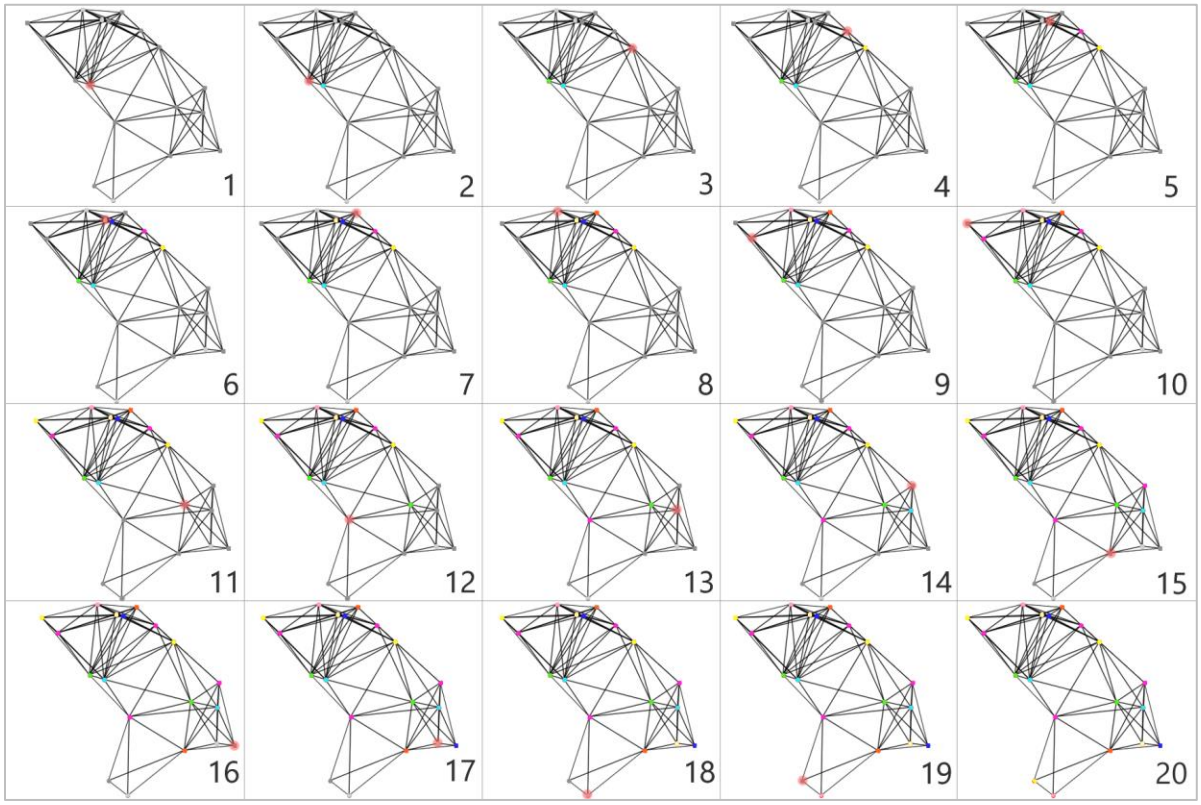


Fig 4: Sequence of Graph coloring on a random geometric graph

From the above sequence, we can see that the graph can be completely colored using 7 distinct colors.

Bipartite Backbone Selection

The final step in the simulation is to generate some suitable backbones from the coloring determined in the process above. We need the two best backbones that can be generated from the largest 4 color classes. These 4 colors will each be combined, and all 6 possible combinations of colored vertices will be examined to determine the best two backbones with the highest domination percentage. Each pairing of colors is subjected to a component search such that only the largest connected component is submitted for comparison against other pairing. Note that it's not a backbone if it's not connected. The two pairings that result in the highest vertex coverage are chosen as the best backbones.

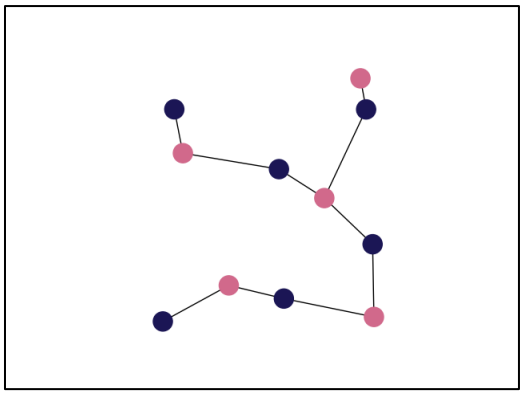


Fig 5(a): Bipartite Backbone 1

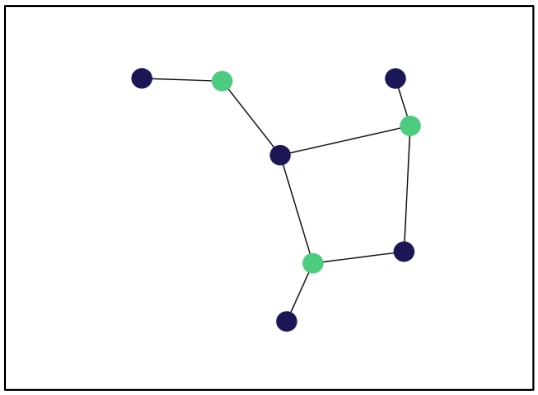


Fig 5(b): Bipartite Backbone 2

Backbone 1 (Fig 5(a)) contains 11 vertices, 10 edges and has 100% vertex coverage. Backbone 2 (Fig 5(b)) contains 8 vertices, 8 edges and has 95% vertex coverage. Coverage is determined by counting the number of vertices that have access to the backbone compared to the complete set of vertices in the graph. Even though both backbones have 100% coverage, backbone 2 is superior when compared to backbone 1 because it uses a lower percentage of the overall vertices. However, it is important to remember that we are trying to find the best possible communication backbone.

Further, we use a set of benchmarks as input to the algorithms and obtain the output. Note that the connected nodes have been shown only for benchmarks for which they are visually clear.

Benchmark 1: Nodes = 4000; Average Degree = 32; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
61192	53	5	30.60	23	294	23	24
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
455	582	92.07%	453	575	90.85%	Backbone 1	

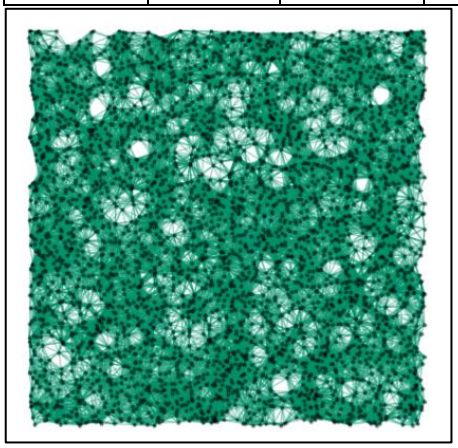
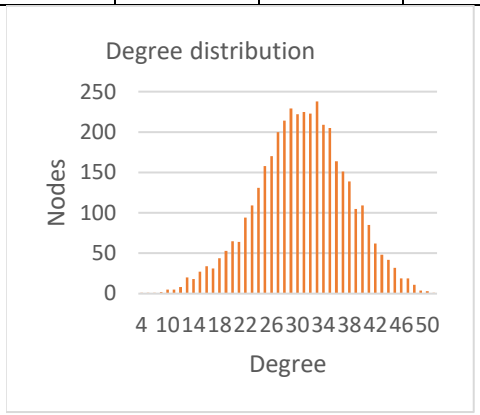
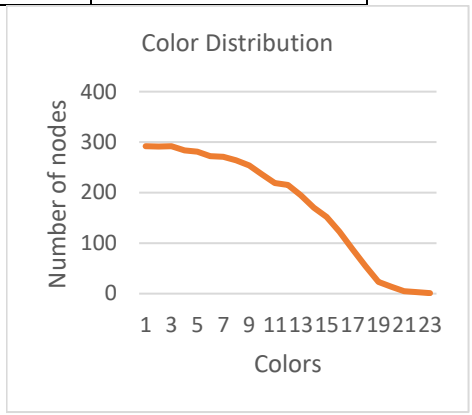


Fig 6: Connected nodes in RGG





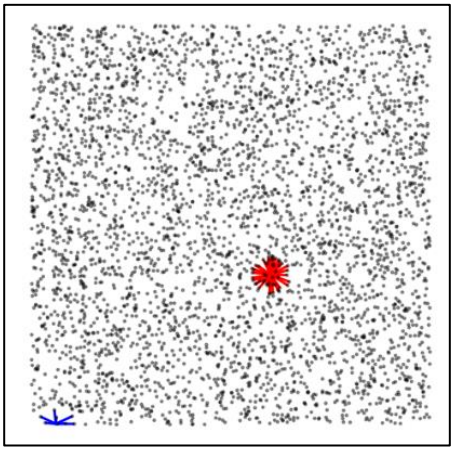
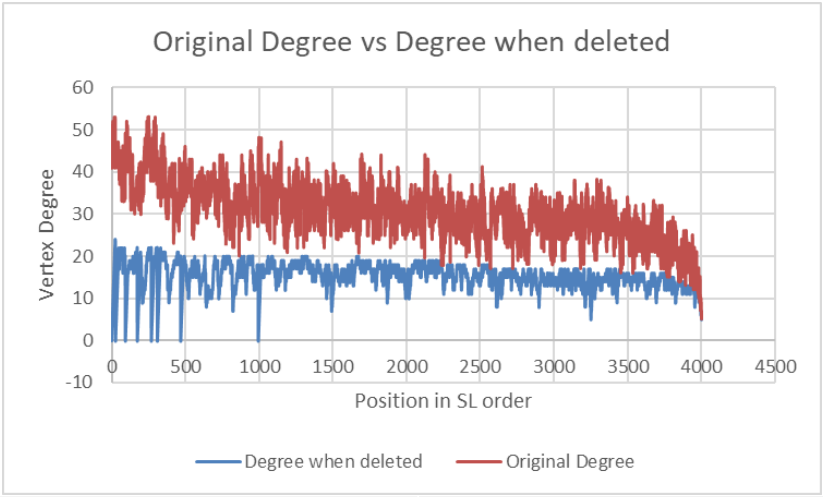


Fig 7: Min (blue) and Max (Red) degree nodes



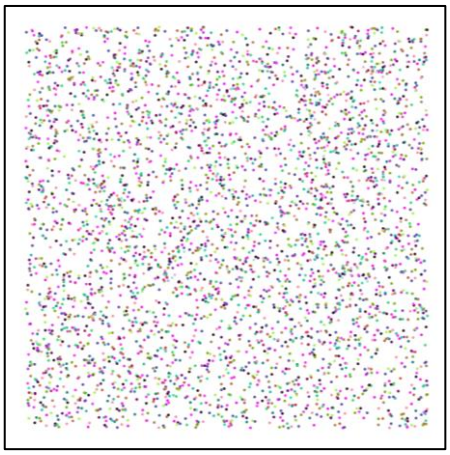


Fig 8: Colored nodes

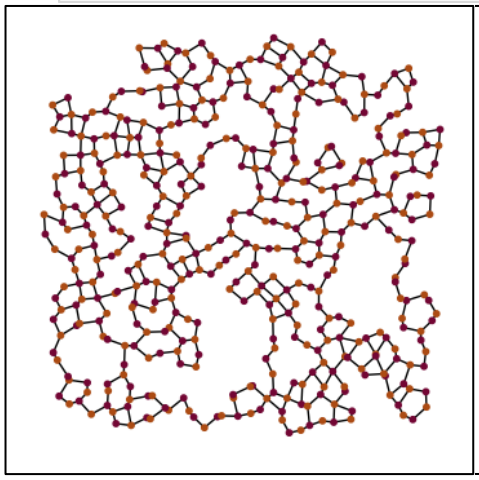


Fig 9: Backbone 1

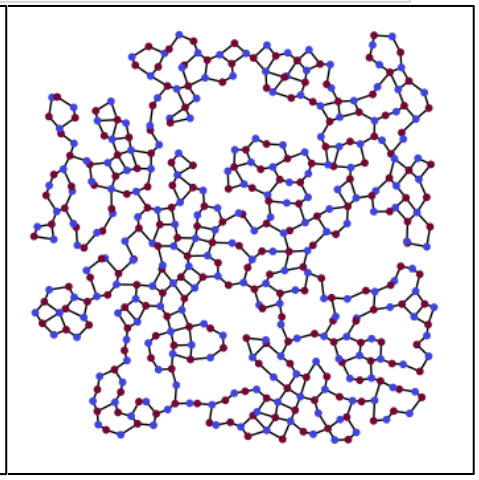


Fig 10: Backbone 2

Benchmark 2: Nodes = 8000; Average Degree = 64; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
245097	96	17	61.27	38	329	37	38
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
561	756	97.02%	574	768	96.80%	Backbone 1	

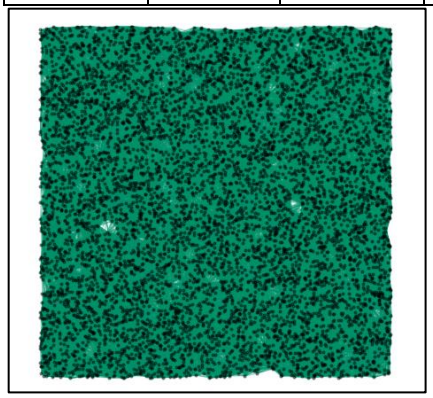
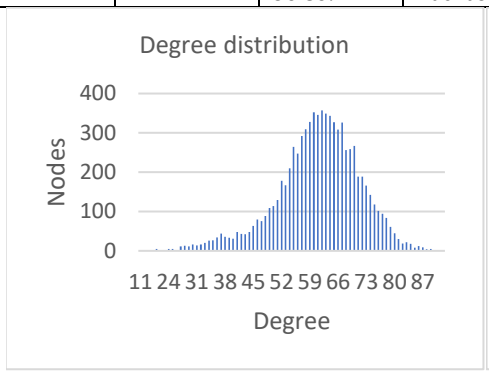
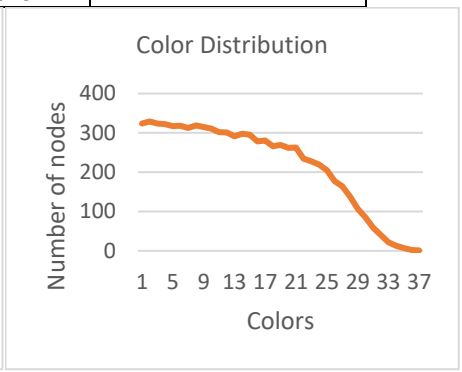


Fig 11: Connected nodes in RGG





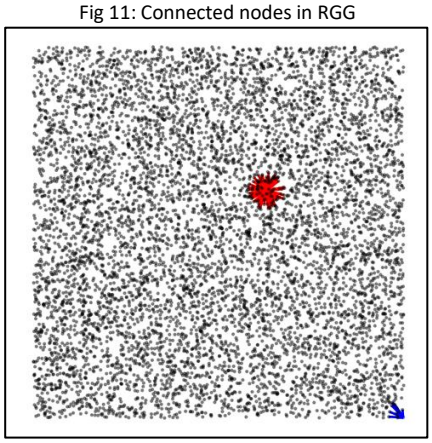
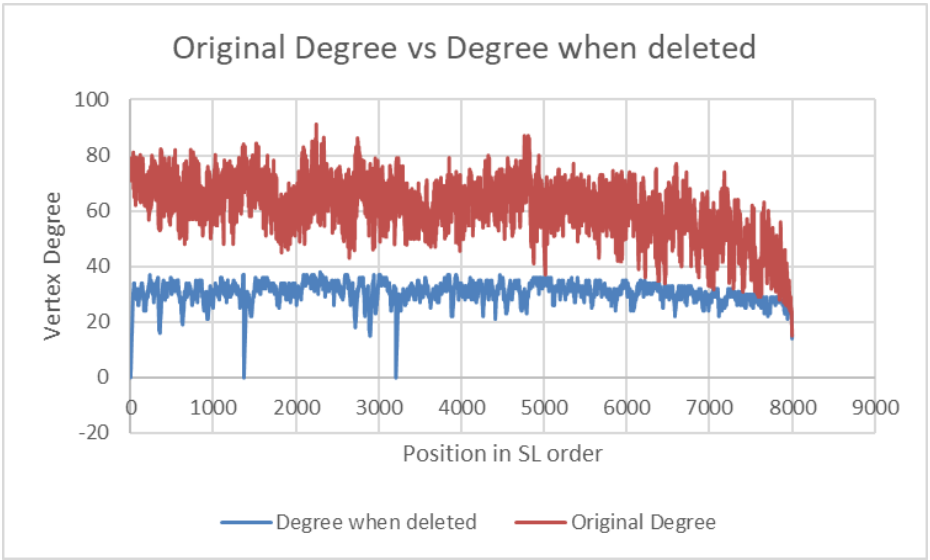


Fig 12: Min (blue) and Max (Red) degree nodes



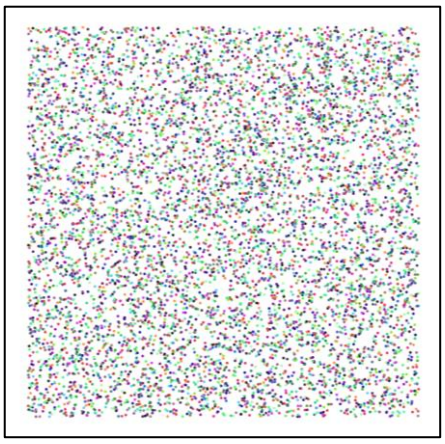


Fig 13: Colored nodes

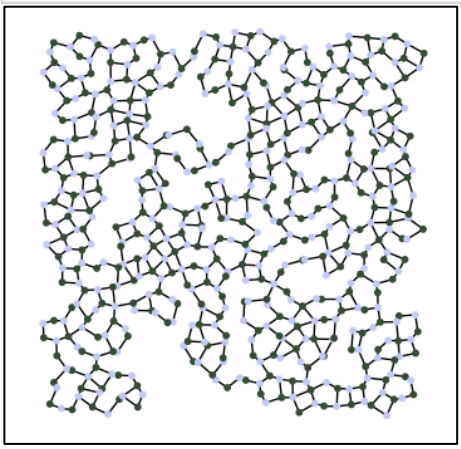


Fig 14: Backbone 1

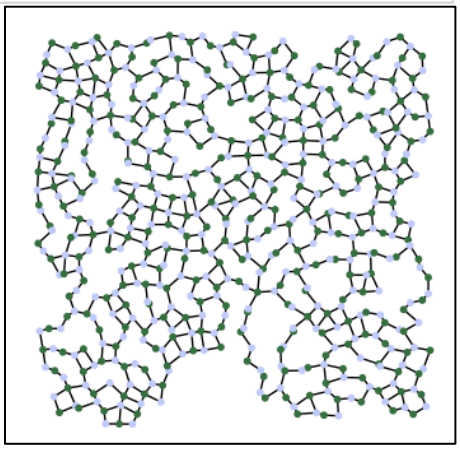


Fig 15: Backbone 2

Benchmark 3: Nodes = 16000; Average Degree = 64; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
498001	92	18	62.25	37	643	37	42
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
1156	1537	98.24%	1113	1449	97.07%	Backbone 1	

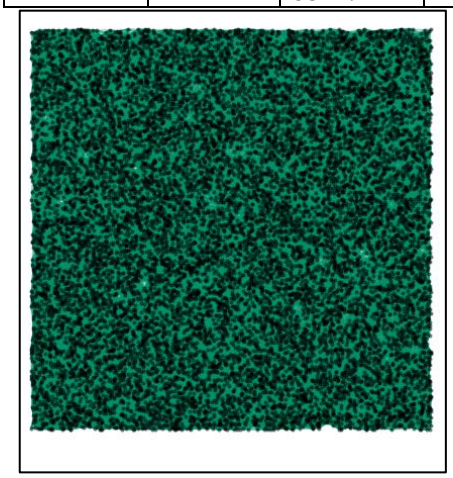
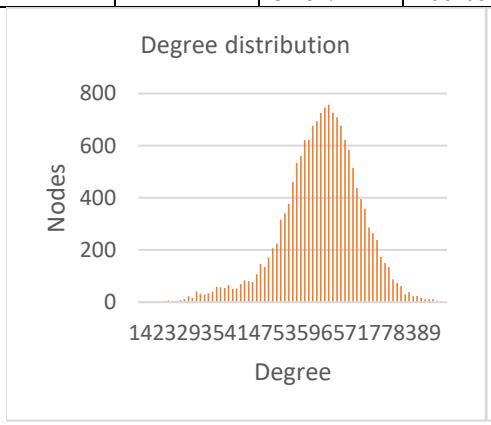
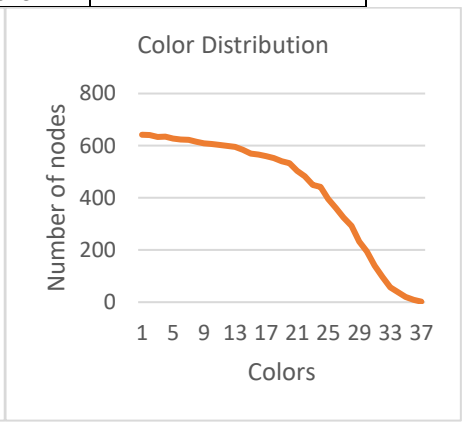


Fig 16: Connected nodes in RGG





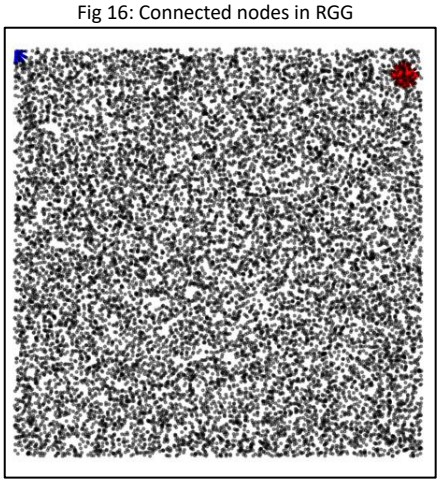
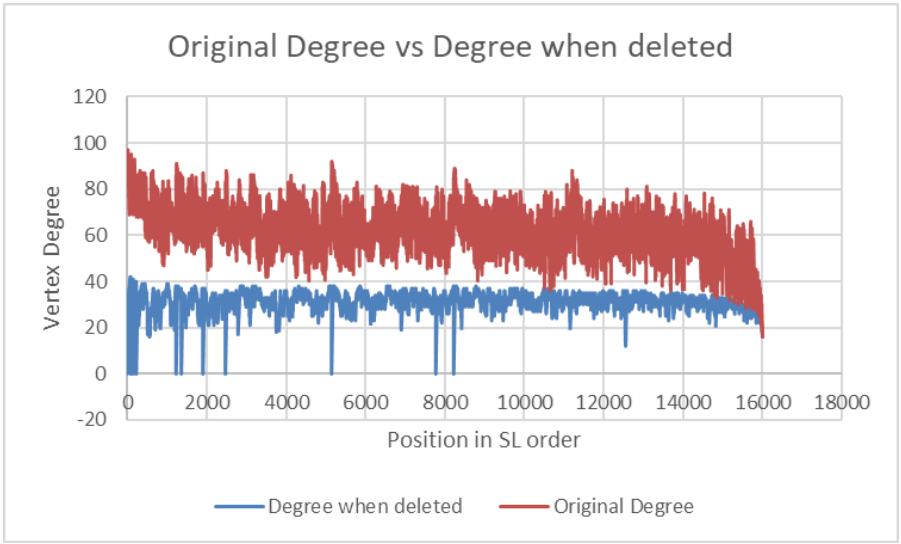


Fig 17: Min (blue) and Max (Red) degree nodes



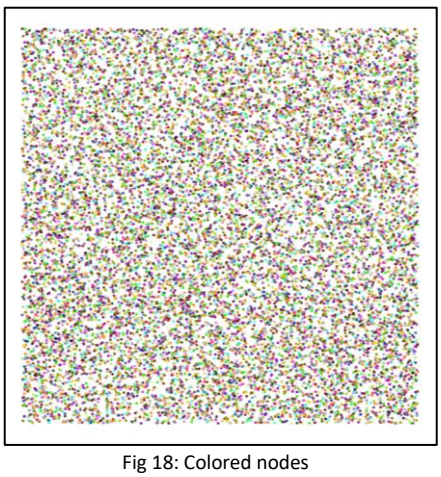


Fig 18: Colored nodes

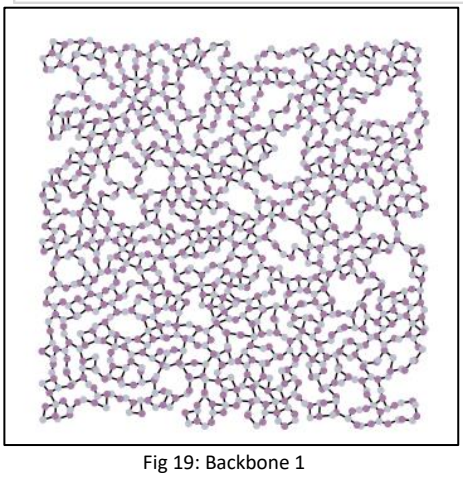


Fig 19: Backbone 1

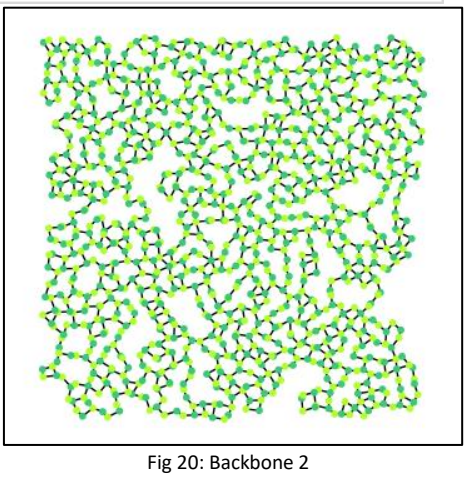


Fig 20: Backbone 2

Benchmark 4: Nodes = 32000; Average Degree = 64; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
1003042	95	21	62.69	38	1274	37	41
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
2230	2941	97.91%	2236	2957	97.77%	Backbone 1	

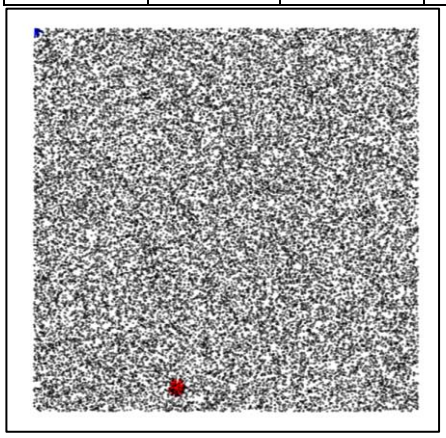
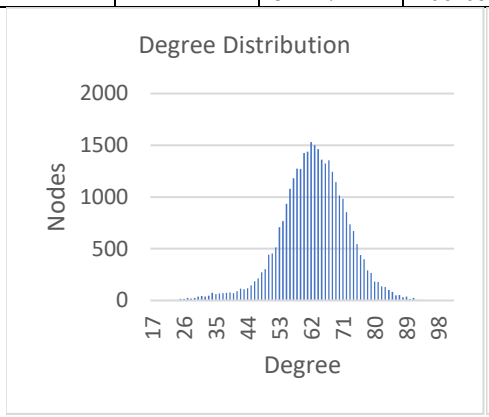
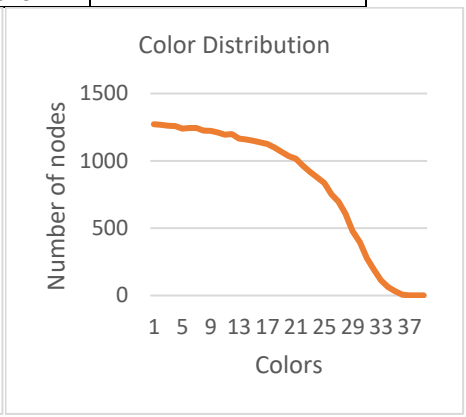


Fig 21: Min (blue) and Max (Red) degree nodes





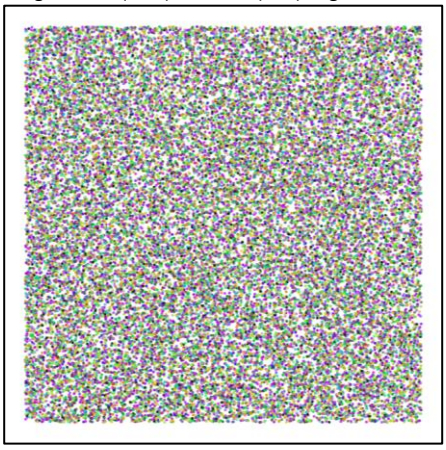
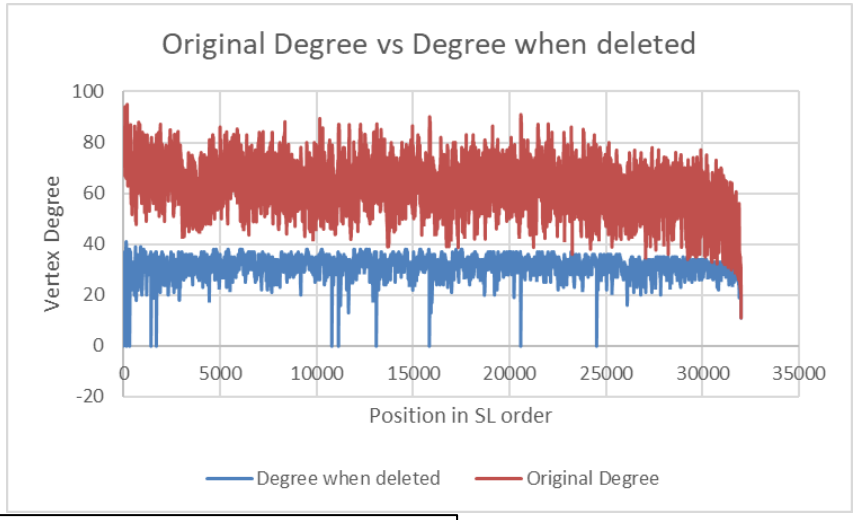


Fig 22: Colored nodes



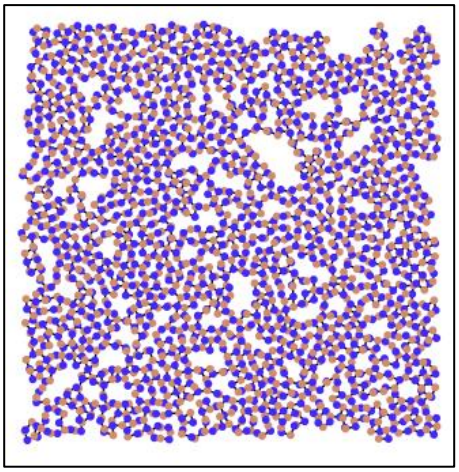


Fig 23: Backbone 1

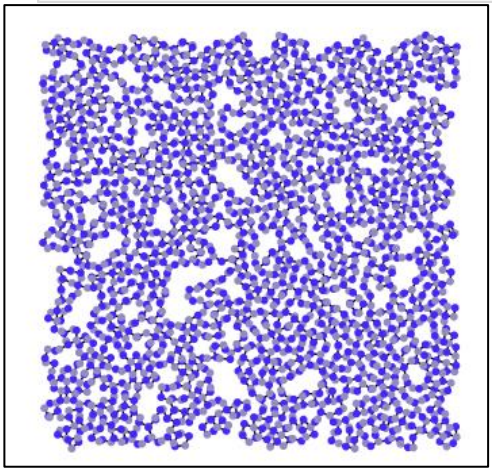
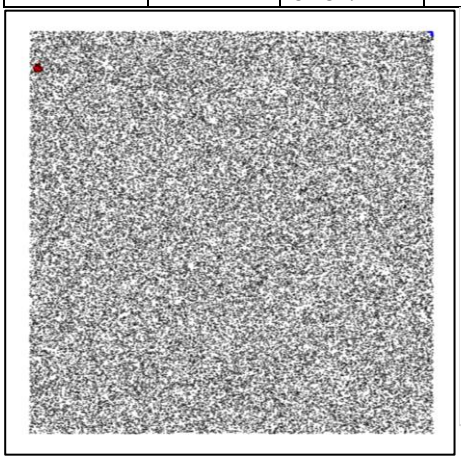
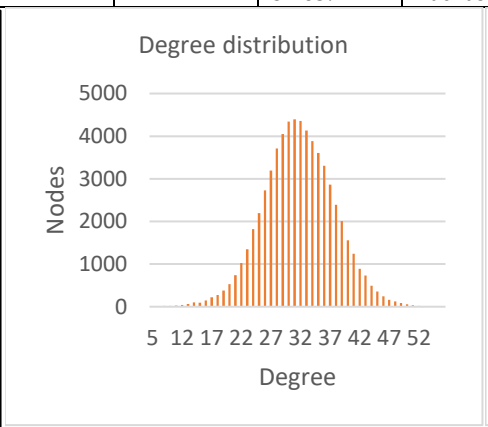


Fig 24: Backbone 2

Benchmark 5: Nodes = 64000; Average Degree = 32; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
1015249	56	6	31.73	27	4562	27	26
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
7409	9298	94.54%	7282	9075	94.05%	Backbone 1	





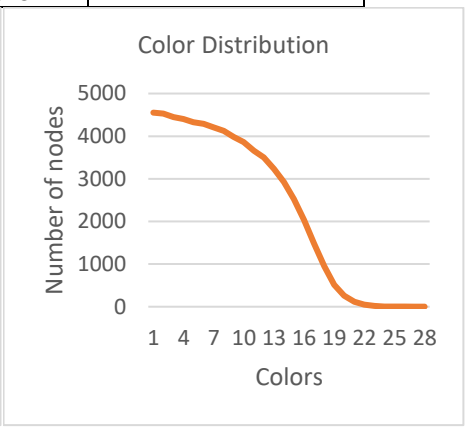


Fig 25: Min (blue) and Max (Red) degree nodes

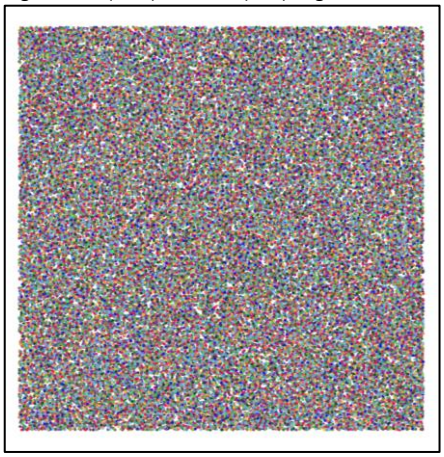
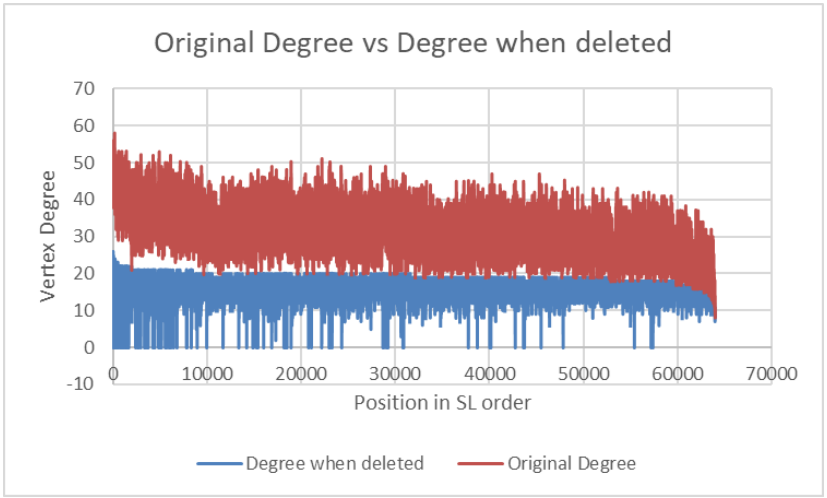


Fig 26: Colored nodes



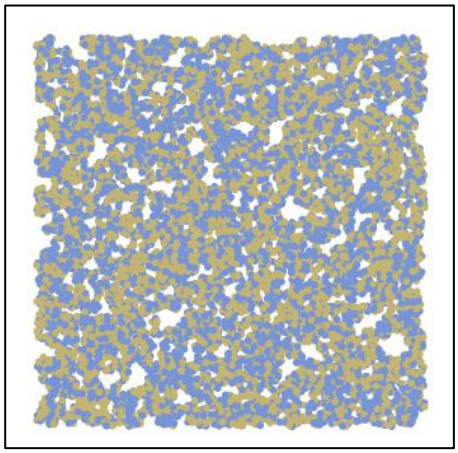


Fig 27: Backbone 1

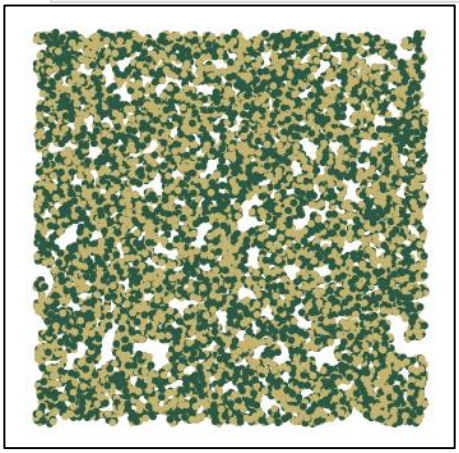
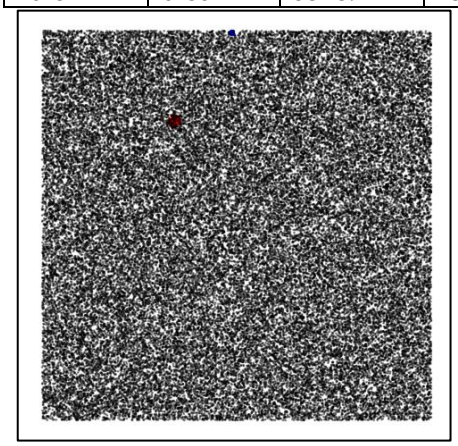
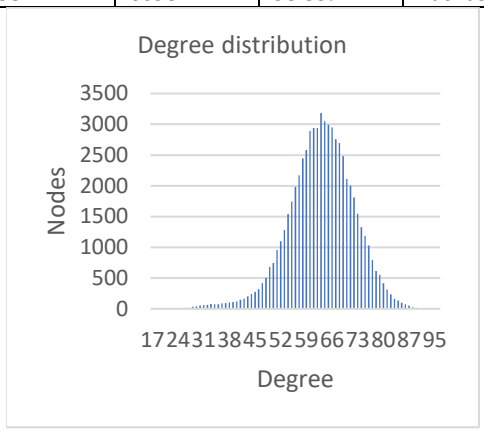


Fig 28: Backbone 2

Benchmark 6: Nodes = 64000; Average Degree = 64; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
2018888	100	13	63.09	41	2541	39	39
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
4613	6150	98.45%	4584	6098	98.33%	Backbone 1	





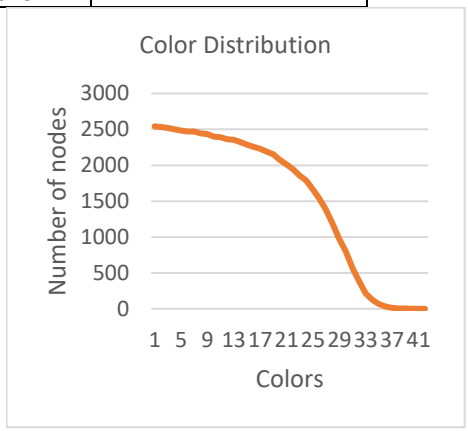


Fig 29: Min (blue) and Max (Red) degree nodes

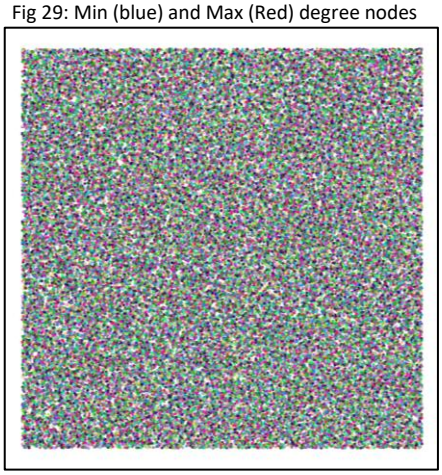
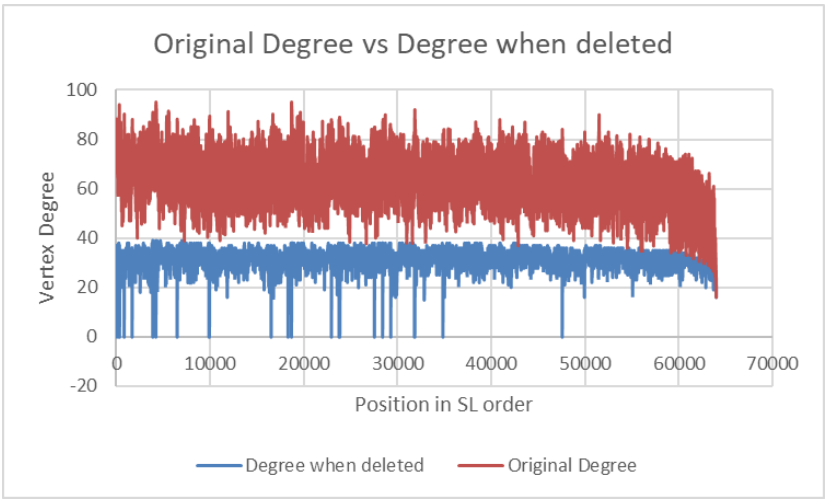


Fig 30: Colored nodes



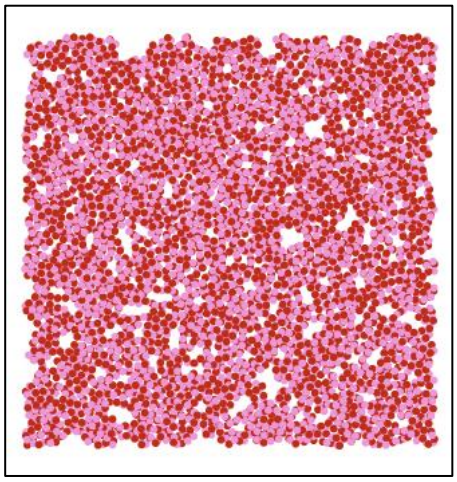


Fig 31: Backbone 1

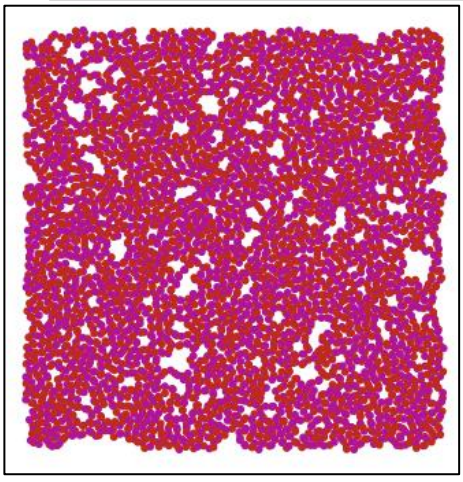


Fig 32: Backbone 2

Benchmark 7: Nodes = 64000; Average Degree = 128; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
4006114	177	25	125.19	65	1374	56	74
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
2581	3630	99.40%	2563	3585	99.28%	Backbone 1	

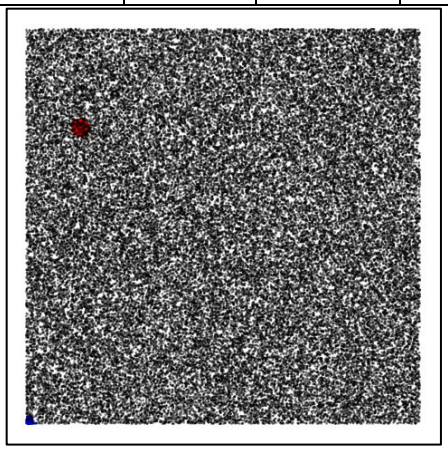
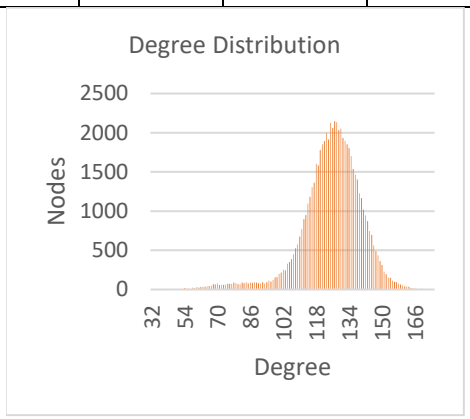
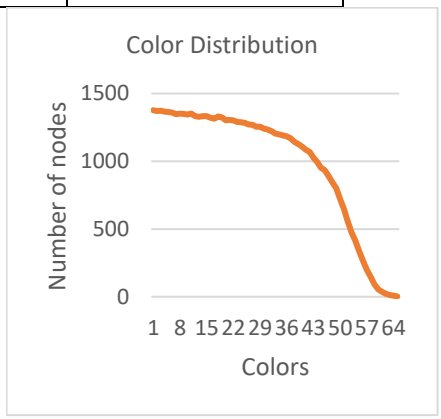


Fig 33: Min (blue) and Max (Red) degree nodes





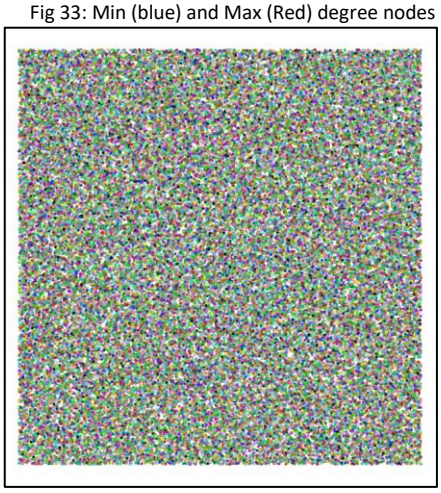
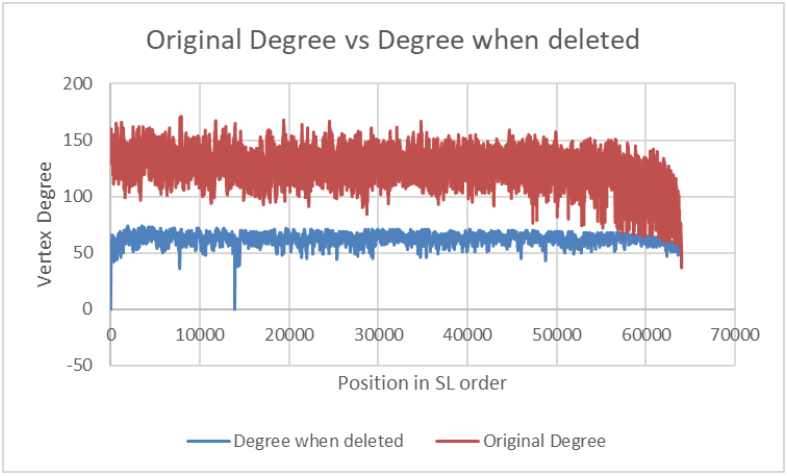


Fig 34: Colored nodes



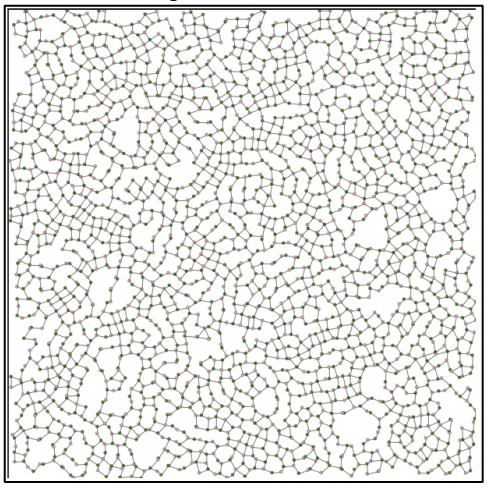


Fig 35: Backbone 1

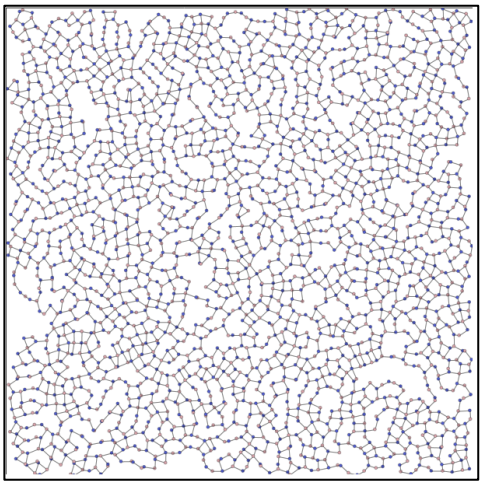


Fig 36: Backbone 2

Benchmark 8: Nodes = 128000; Average Degree = 64; Topology = Square

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
4054162	99	16	63.35	41	5066	41	43
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
8985	11878	98.31%	8987	11930	98.11%	Backbone 1	

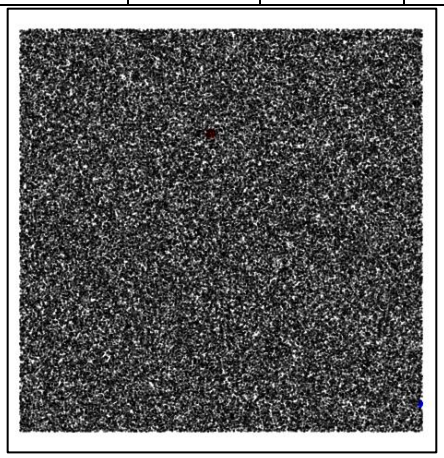
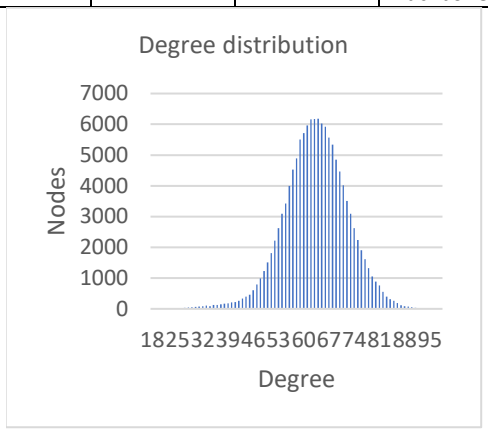
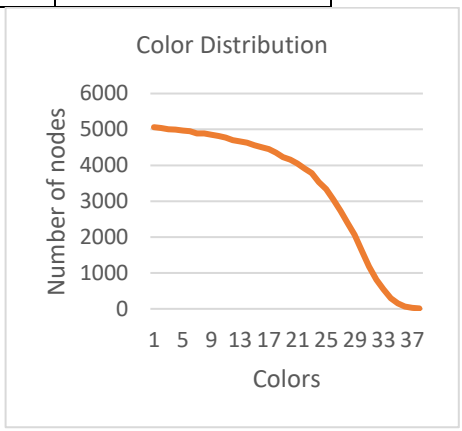


Fig 37: Min (blue) and Max (Red) degree nodes





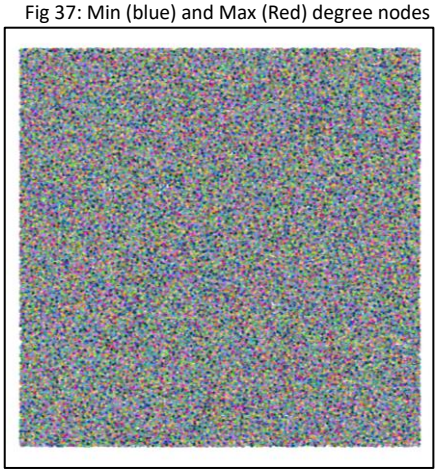
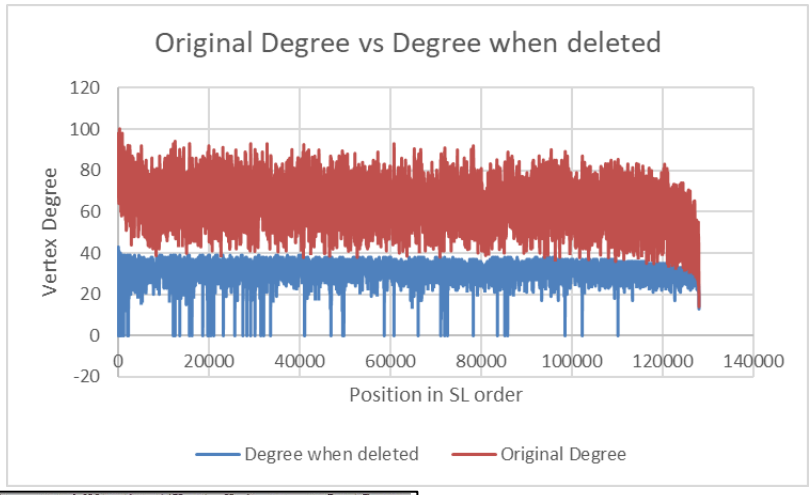


Fig 38: Colored nodes



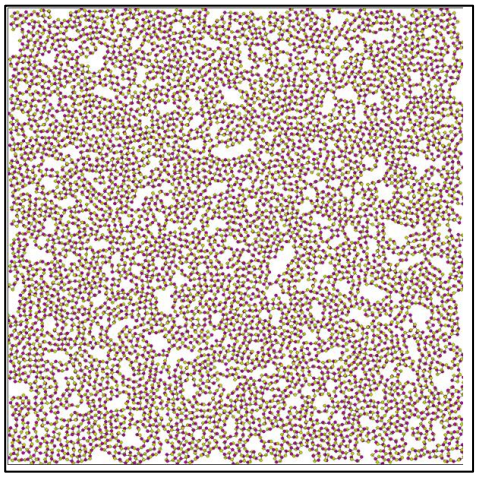


Fig 39: Backbone 1

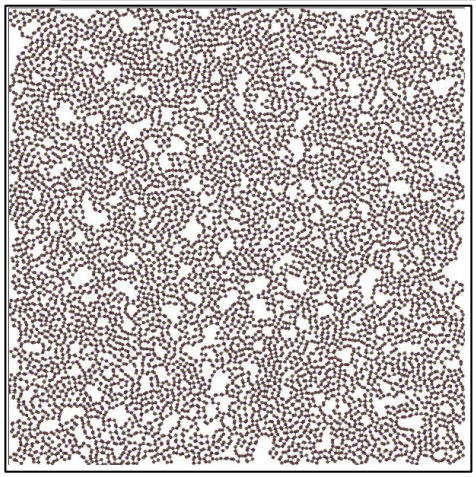


Fig 40: Backbone 2

Benchmark 9: Nodes = 8000; Average Degree = 64; Topology = Disk

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
245208	93	24	61.3	35	36	37	39
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
576	789	97.80%	570	775	97.12%	Backbone 1	

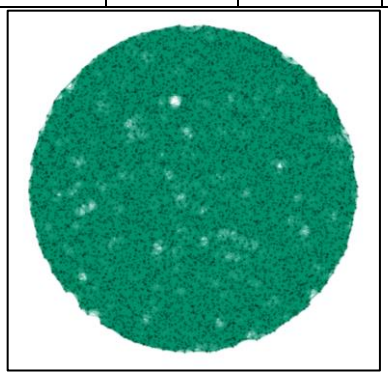
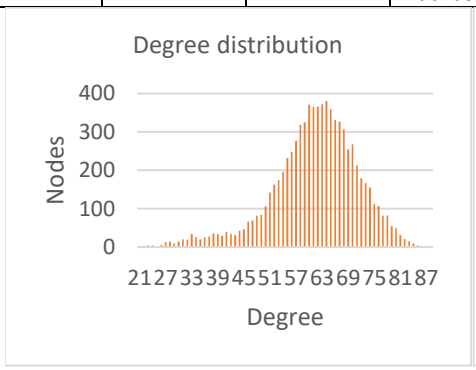
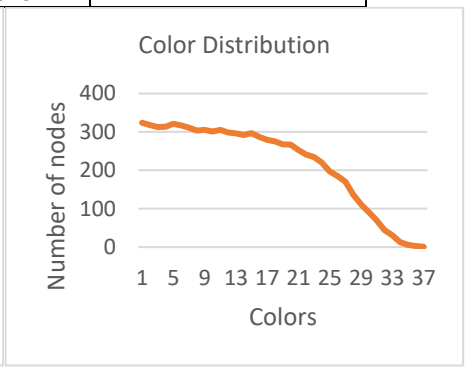


Fig 41: Connected nodes in RGG





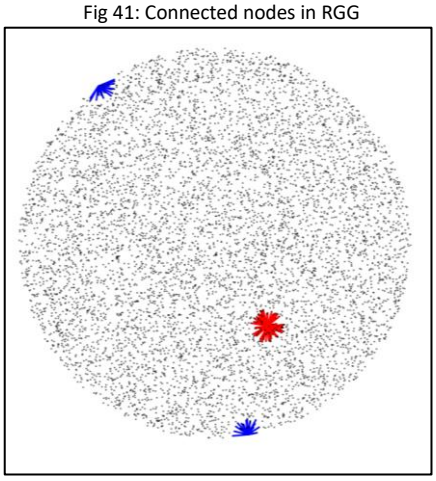
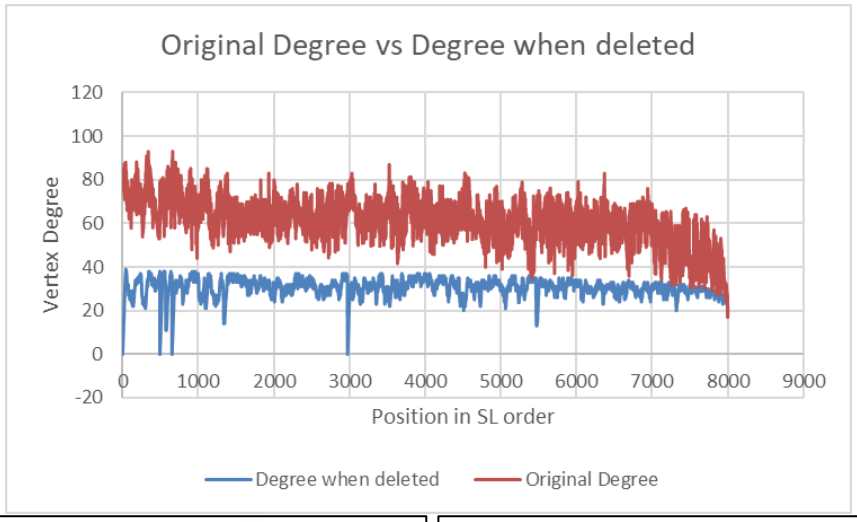


Fig 42: Min (blue) and Max (Red) degree nodes



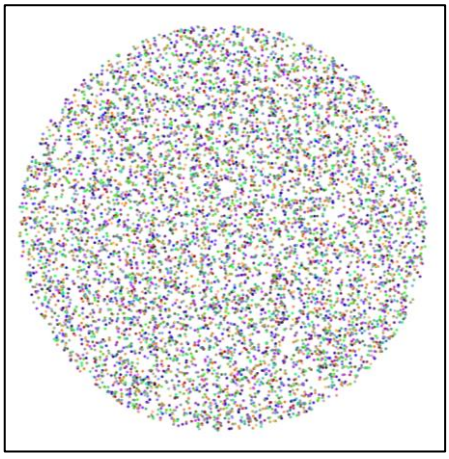


Fig 43: Colored nodes

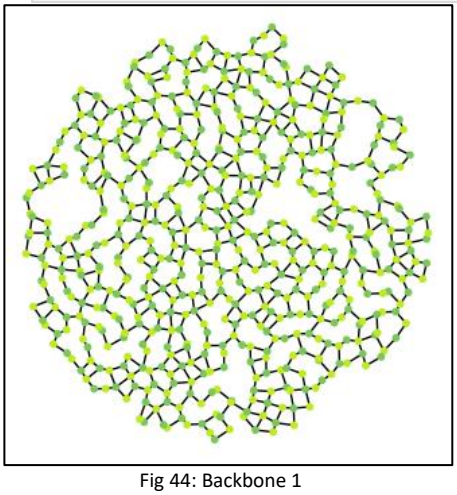


Fig 44: Backbone 1

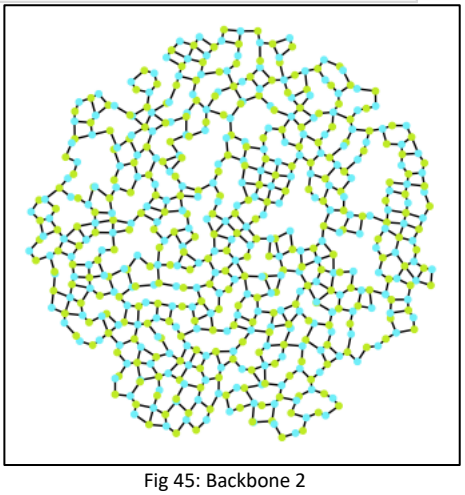


Fig 45: Backbone 2

Benchmark 10: Nodes = 8000; Average Degree = 128; Topology = Disk

No. of Edges	Max. Degree	Min. Degree	Avg. Degree	No. of Colors	Max color	Terminal size	Max Degree when deleted
482858	163	52	120.71	56	61	58	71
Backbone 1 vertices	Backbone 1 edges	Backbone 1 Coverage	Backbone 2 vertices	Backbone 2 edges	Backbone 2 Coverage	Superior Backbone	
323	448	99.24%	318	435	99.11%	Backbone 1	

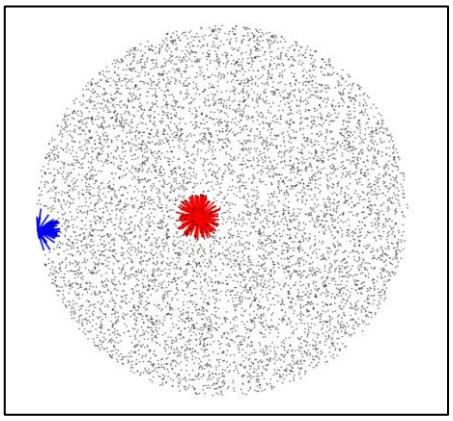
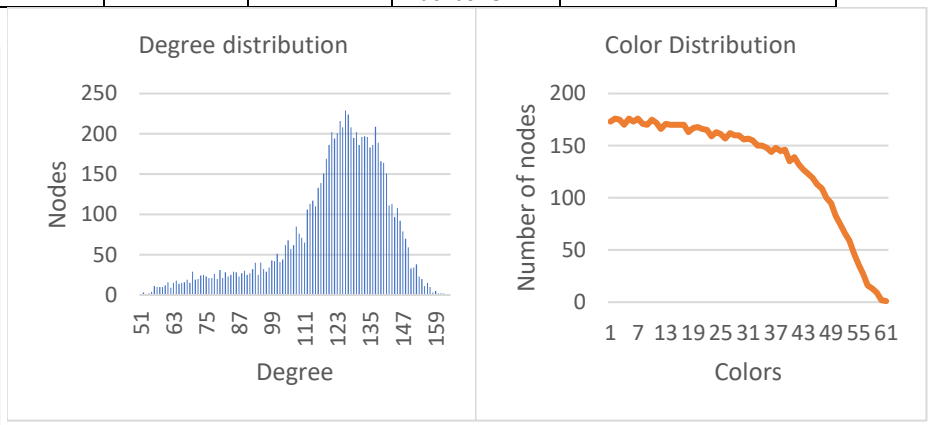


Fig 46: Min (blue) and Max (Red) degree nodes



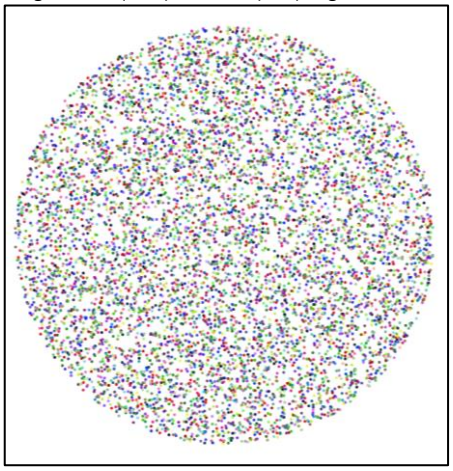
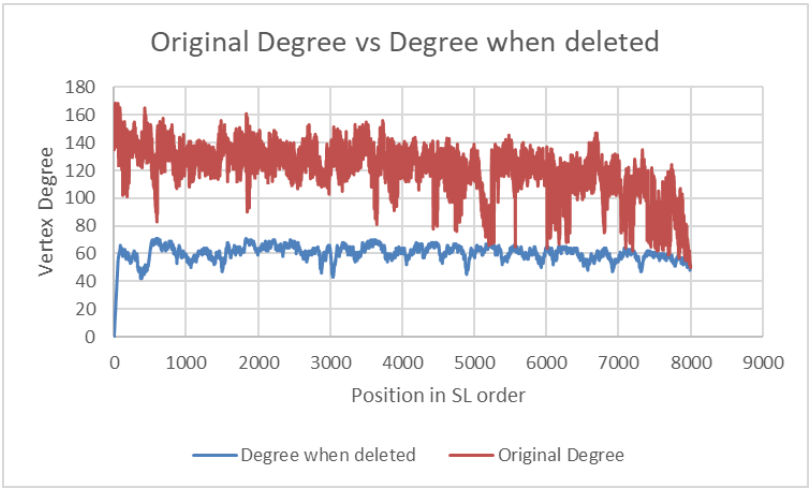


Fig 47: Colored nodes



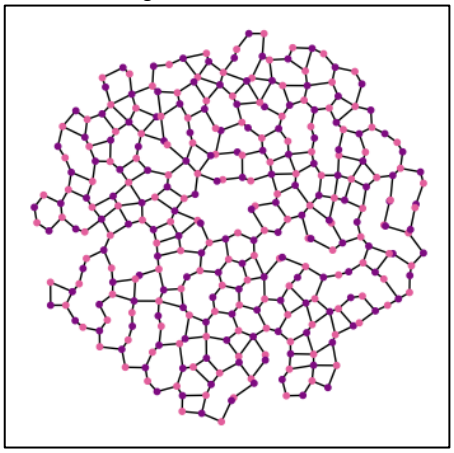


Fig 48: Backbone 1

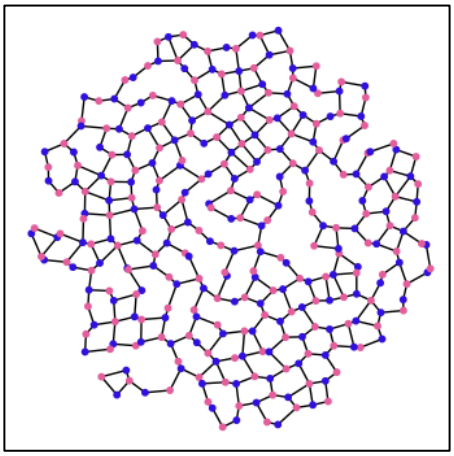


Fig 49: Backbone 2

SUMMARY

In this report, we saw how we can generate a Random geometric graph (RGG) which represents a set of wireless sensors placed at random points in a topology, which was Part 1 of this project. We found the sensors with maximum and minimum sensor coverage. In Part 2, we implemented the two algorithms – Smallest last order and Graph coloring - on the random geometric graph to obtain high quality communication backbones. We also demonstrated the implementation of each of these algorithms on a small-sized graph and applied them on a set of input benchmarks. Finally, we used the output from Part 2 as an input to Part 3, which is to generate Bipartite backbones with maximum coverage. We obtained a set of two backbones and determined the superior one based on the domination percentage of each of them on the wireless network area.

Given below is the overview of the output generated for each of the benchmarks we used in our algorithm:

Benchmark #	1	2	3	4	5	6	7	8	9	10
Nodes	4000	8000	16000	32000	64000	64000	64000	128000	8000	8000
Average Degree	32	64	64	64	32	64	128	64	64	128
R	0.05	0.05	0.035	0.025	0.012	0.017	0.025	0.012	0.089	0.126
Topology	Square	Square	Square	Square	Square	Square	Square	Square	Disk	Disk
No. of Edges	61192	245097	498001	1003042	1015249	2018888	4006114	4054162	245208	482858
Execution Time (I)	1.62	4.94	12.66	21.38	29.5	52.31	69.67	89.06	5.28	8.76
Max. Degree	53	96	92	95	56	100	177	99	93	163
Min. Degree	5	17	18	21	6	13	25	16	24	52
Output Avg. Degree	30.6	61.27	62.25	62.69	31.73	63.09	125.19	63.35	61.3	120.71
Execution Time (II)	0.31	1.32	3.87	13.98	43.69	49.74	68.58	251.45	1.23	2.49
No. of Colors	23	38	37	38	27	41	65	41	35	56
Max Degree when deleted	24	38	42	41	26	39	74	43	39	71
Max. color count	294	329	643	1274	4562	2541	1374	5066	36	61
Terminal Clique size	23	37	37	37	27	39	56	41	37	58
Execution Time (III)	7.34	23.48	61.29	111.44	180.5	261.24	354.66	463.76	23.28	37.32
Backbone 1 vertices	455	561	1156	2230	7409	4613	2581	8985	576	323
Backbone 1 edges	582	756	1537	2941	9298	6150	3630	11878	789	448
Backbone 1 Coverage	92.07%	97.02%	98.24%	97.91%	94.54%	98.45%	99.40%	98.31%	97.80%	99.24%
Backbone 2 vertices	453	574	1113	2236	7282	4584	2563	8987	570	318
Backbone 2 edges	575	768	1449	2957	9075	6098	3585	11930	775	435
Backbone 2 Coverage	90.85%	96.80%	97.07%	97.77%	94.05%	98.33%	99.28%	98.11%	97.12%	99.11%

- Additional Information

Based on the above outcome, I tried to predict the Execution time for some additional benchmarks to see how far can my program go with increasing input size.

Below is a summarized result of the additional benchmarks:

Benchmark	1	2	3	4
Topology	Square	Square	Disk	Disk
Nodes	256000	1000000000	256000	1000000000
Average Degree	64	64	64	64
Execution time (I)	175	750	47	90
Execution time (II)	280	4400	28	71
Execution time (III)	567	940	160	250

As observed, the running times are the highest for Part II of the project, but are fair considering the input size. Note that the above predicted times are approximate values and are calculated in seconds.