# Constructing Bipartite Backbones in Wireless Sensory Networks

Will Spurgin*

6 December 2016

## 1 Executive Summary

### 1.1 Introduction

Imagine a scenario in which you are part of a team monitoring the volcano Mount Pinatubo for surface temperature and seismic activity. Further, you are given a multitude of low cost short-range wireless sensors to place around an area of the volcano to take these measurements. Since Mt. Pinatubo is somewhat hard to reach, you decide to drop these sensors from a helicopter. After dropping your sensors over the area in a uniformly random fashion, you now need to read all the data from these sensors. However, since they are short-range wireless sensors, you would have to be sufficiently close to a sensor to read its data. However, since all the nodes are wireless, they can communicate their data to each other. Ideally, you would rather go to one or relatively few nodes to retrieve all the data. The question becomes: to which node(s) do you go?

This fictional scenario is the problem setting to which this report provides solution discussion, reduction of those solution to practice, and a variety of benchmark measurements. Formally, this report examines the construction of backbones of communication in Wireless Sensor Networks (WSNs) modeled as Random Geometric Graphs (RGGs). The results presented in this report indicate that, generally, RGGs (and thus WSNs) with uniformly distributed nodes can achieve near **100%** coverage from a bipartite backbone using the methods described in this report. The use of the methods presented in this report are strongly recommended for any application involving WSNs or scenarios that can be modeled as RGGs. Study of this work is not new[4, 5], and much of that work

*Masters Candidate, Department of Computer Science, Sourthern Methodist University. Email address: wspurgin@smu.edu.

Table 1: Abbreviated Results

| Number of Nodes | 4000 | 4000 | 1000 | 16000 | 4000 | 64000 | 64000 |
|---|---|---|---|---|---|---|---|
| Desired Avg. Degree | 128 | 64 | 32 | 64 | 64 | 128 | 64 |
| Shape | disk | disk | plane | plane | plane | plane | plane |
| Actual Avg. Degree | 118.08 | 60.20 | 29.02 | 62.10 | 60.11 | 125.25 | 62.99 |
| Radius | 0.179 | 0.126 | 0.101 | 0.036 | 0.071 | 0.025 | 0.018 |
| Number of Edges | 472302 | 240784 | 29018 | 993580 | 240428 | 8016232 | 4031522 |
| Max Degree | 167 | 93 | 46 | 94 | 90 | 169 | 101 |
| Min Degree | 43 | 22 | 7 | 13 | 11 | 23 | 15 |
| Coverage | 99.67% | 99.58% | 99.50% | 99.69% | 99.58% | 99.76% | 99.52% |

corroborates the findings of this report. The process for choosing for the backbones in a WSN are straight forward:

1. For a graph $G$ (our RGG modeling a WSN), produce a Smallest Last Ordering[6] $S_G$.
2. For each vertex $v_j$ in $S_G$ color the vertex with the lowest available color class that is not a color of the neighbors of $v_j$ (i.e. $N(v_j)$).
3. For all 6 of the combination of the largest 4 color classes pick the top 2 by the number of edges of their maximum connected subgraph.
4. $\Delta$ These top two combinations are the two ideal backbones.

The implementation of these steps are further discussed in Section 2.

In the presentation of this report, several tables, and graphics are used to make the seeing the merit of the results effortless. For all benchmarks, the following plots are included: A plot of the RGG (with edges), a plot of the node with lowest degree, a plot of the node with highest degree, a plot of the color classes frequency, a plot of the degree distributions, a plot of the degree when deleted versus original degree of the Smallest Last ordering algorithm, and lastly the plot of the two major backbones of an RGG. Additionally, a table is provided with summary information about the RGG (e.g. number of edges, vertices, desired average degree, actual average degree, etc.). An abbreviated table of results is given below in Table 1.

As will be discussed further in Section 2, the algorithms used in this report are linear in the number of vertices and edges $\mathcal{O}(|V| + |E|)$[6, 4]. The implementations presented in these reports make a few speed gains and memory reductions by using efficient data structures. Figure 1 shows the linear runtime of the two phases of the project: RGG Generation and Bipartite Backbone Construction (and Selection).

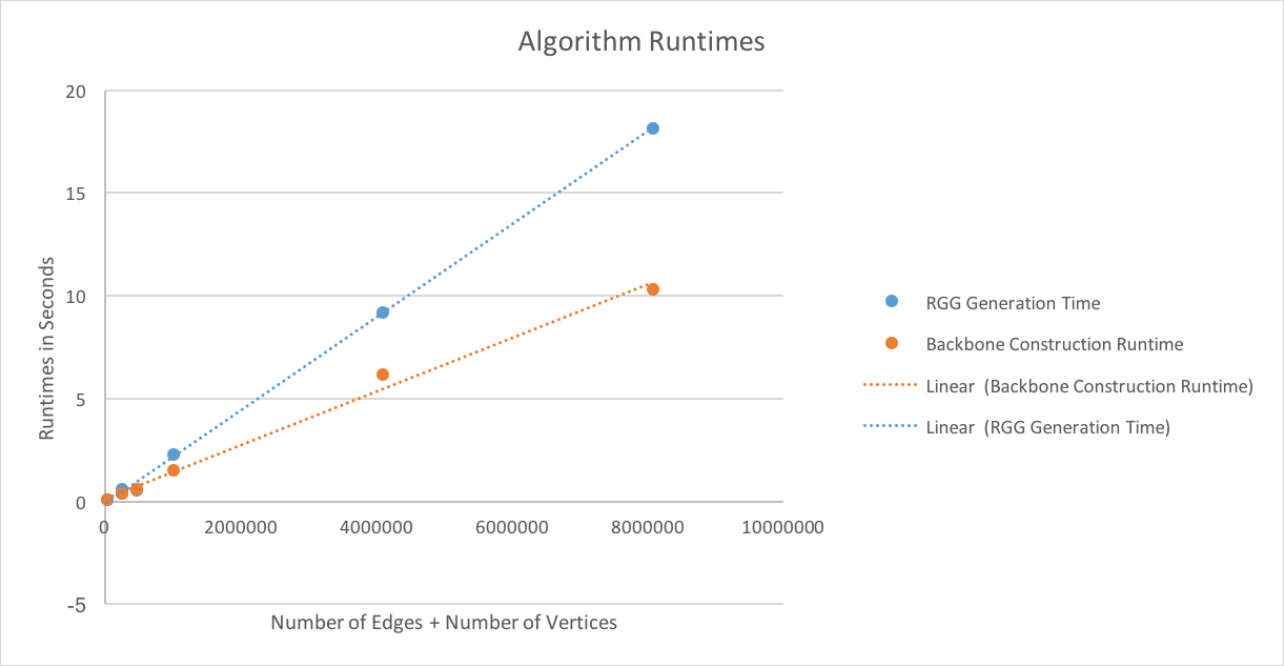



Figure 1: Linear Runtimes

## 1.2 Environment Description

The results of this report were generated with the following hardware and software:

- Hardware:
  - Apple MacBook Pro (Retina, Mid 2012)
  - Processor: 2.6 GHz Intel Core i7 (with turbo boost up to 3.3GHz)
  - Memory: 8 GB 1600 MHz DDR3
  - Intel HD Graphics 4000 1536 MB
  - Storage: 500.28 GB Solid State Drive, SATA Connection.
- Language: C++
  - Compilers:
  - GNU GCC 5.2.0 (Homebrew gcc 5.2.0 build)
  - Clang 8, Apple LLVM version 8.0.0 (clang-800.0.38)
- Graphical Tool:
  - R[7] version 3.3.1 (2016-06-21) – (codename “Bug in Your Hair”)

All algorithms were written in C++ using the C++11 standard. These implementation are less intensive on the resources of the system. The utilization are the measured range of peak average utilization for each benchmark.

- Resource Utilization:
  - CPU Utilization: 5% to 15%

- Memory: 0.625 MB to 40 MB

The tool to generate the graphics for this report was R[7]. R is a statistical coding language, but has the benefit of a large community that has built several packages for developing graphics. For the purposes of this project, the core R libraries were the only graphical tools needed. However, there are several packages for network analysis built for R[3]. All data that is given to R for displaying the various tables and graphs seen in this report are passed via CSV files (as that format is easily read by R).

Further, R is used to generate this report into Latex (and then a PDF thereafter). This document is written in R-Markdown[1] where the R code is embedded to generate plots and tables.

# 2 Reduction to Practice

In this section, the algorithms and implementations of those algorithms used in this report are discussed in detail. To begin, the data structures need to model the simulated data in this report as well as their relationships is presented. Additionally, certain data structures are mentioned that enhance the algorithms real runtime (complexity remains $O(|V| + |E|)$). Thereafter, Section 2.2 describes the smallest last ordering algorithm[6] and greedy graph coloring algorithm implemented in this work. The subsequent section substantiates the linear runtime of the smallest last ordering and coloring algorithm implementations. In Section 2.4 a full walkthrough from generation to bipartite backbone selection is given. Lastly, a discussion of the effectiveness of these algorithms is given in Section 2.5.

## 2.1 Data Structures

A graph is simple a set of **vertices** and a set of relations between those vertices called **edges**. In our scenario, each vertex has geometric data as well ($x$ and $y$ coordinates). This report took the simplest approach when modeling the RGGs. Every vertex was an object (C++ `struct`) with a unique incrementing identifier, the x-y coordinates, and various data points filled in by the application (e.g. color, backbone assignment, etc.). These points are stored in an array (C++ `vector`) at their identifier location (e.g. the node 0 is stored at the 0 index of the array). The **edges** of the graph are then stored in a Hash table (C++ `unordered_map`) with the vertex identifiers as keys pointing to a doubly linked list of pointers to vertices. Most other data structures use a variety of Hash tables and collections of pointers to vertices. When the collection is often searched (e.g. in breadth first searches) a Red-Black Tree [2] is used (C++ `set`). Otherwise, the doubly linked list implementation is used for $\Theta(1)$ inserts and deletions at the beginning or end. The only other notable data structure, is the

one used during the generation of the RGGs. The data begins as a uniformly random distribution of points on either a unit disk or unit square. These are the modeled "sensors" inside the WSN. Given a desired average degree, a radius of adjacency is calculated. This radius of adjacency is used to determine what points are adjacent inside the RGG. A node $v_j$ is adjacent to another node $v_i$ if the Euclidean distance of the difference between the x-Y coordinate vector is less than $R$: the radius of adjacency. Since the process both for generating the RAG and determining its adjacency list is fairly mathematical, a special data structure was used that represents a matrix (in Appendix see `mat.h` and `mat.cpp`). This data structure has been optimized for speed over several years. It represents data as rows and columns, but internally maps the data as one sequential allocation of memory. It makes vector-wise mathematical operations both very fast, and very easy for a programmer[1].

## 2.2 Algorithm Description

The smallest last ordering, as described in [6] is quite simple for a given graph $G$:

1. Initialize: $H \leftarrow G$, $S_G \leftarrow \{\}$
2. Find $v_j$ in $H$ such that the degree of $v_j$ is the minimum degree $\delta$ in $H$.
3. Cut $v_j$ from $H$, $H \leftarrow H - v_j$, $S_G[] \leftarrow v_j$
4. If $H$ is not empty, go to step 2, else reverse $S_G$ and you have the smallest last ordering.

The smallest last ordering is clearly a linear time algorithm in the number of vertices and edges if certain operations can be constant time. Mainly, the discovery of $v_j$ in $H$. If this is constant time, then each node is visited once, and the cutting of $v_j$ from $H$ (given the data structures mentioned above) is at most linear in the number of vertices and edges of $v_j$.

Once a smallest last ordering of $G$ is obtained, we color the nodes by precedence in the smallest last degree ordering. The simplest coloring algorithm is called the Greedy Coloring[4, 2]. Given an ordering of vertices (any ordering, even random), a vertex is assigned the lowest color class possible that is **not** its neighbors' color. This algorithm has a natural tendency to produce a coloring with lower color classes having the highest frequency. Below the algorithm is summarized in step form:

1. Given vertex ordering $O_G$
2. For each $v_i$ in $O_G$
3. Select $c = \min(\text{color})$ not in $\text{colors}(N(v_i))$
4. Assign $\text{color}(v_i) \leftarrow c$

[1]Note that this is my personal opinion, as I, with the help of Dr. Daniel Reynolds of Math department at Southern Methodist University, built this C++ data structure. This work does not provide rigorous proofs of the optimization of this data structure.

This algorithm is also clearly of $O(|V| + |E|)$ complexity. Each node is colored after traversing all of its neighboring edges to learn the colors of its neighbors.

From a coloring of $G$ we can determine the bipartite backbones. Others have shown that if $\delta = \min(d(v_i|G))$ (that is if $\delta$ is the minimum degree in $G$), the optimal[2] backbone can be selected from the $\binom{2}{\delta}$ combinations of $\delta$ largest independent sets within $G$[4, 5]. For this report, it sufficed to pick only the 4 largest independent sets to find the 2 backbones with the highest vertex coverage. The largest independent sets are simply the largest color classes. Determining the frequency of each color class is a linear time operation in the number of edges. To construct a Bipartite backbone, we take the $\binom{2}{4} = 6$ combinations of the 4 largest color classes and form partitions of the nodes with those color classes. Additionally, we form a subgraph adjacency list of only the nodes within this backbone partition. With this, we can determine the maximal connected subgraph of this bipartite graph (i.e. the largest component) by performing a simple breadth first search with the backbone nodes and backbone adjacency list. If stopped during the breadth first search with nodes still remaining, continue the breadth first search with a node as yet unvisited until all nodes are reached. The nodes visited during the breadth first search where the largest number of edges were reached form the backbone. For clarity, the algorithm is summarized step by step:

1. For bipartite graph $B_{c,k}$ with independent sets $c$ and $k$
2. While nodes unvisited in $B_{c,k}$
    1. $v_i \leftarrow$ popfrom *remaining_nodes*, $\epsilon \leftarrow 0$
    2. $\mathbf{b}_{c,k} \leftarrow \{\}$
    3. Add $v_i$ to queue
    4. While $v_j \leftarrow$ pop from queue
        1. if $v_j$ already visited, continue.
        2. $\epsilon \leftarrow \epsilon + |N(v_j)|$
        3. Push $N(v_J)$ into queue.
        4. $\mathbf{b}_{c,k}[] \leftarrow v_j$
        5. Mark $v_j$ as visitied.
3. Select $\mathbf{b}_{c,k}$ with largest $\epsilon$ is the largest connected subgraph.

In these steps, a few tedium are overlook, for example $\epsilon$, the number of edges in the current component, and $\mathbf{b}_{c,k}$ the current bipartite nodes within the current component are represented as single entities, but, since we want to select only that component which had the most edges, we must obviously store the all (or at least the max) component every time a new component is searched.

This algorithm is linear in the number of edges and vertices of the bipartite backbones. Statistically speaking, in a true RGG there is very little chance that all combinations of bipartite backbones contain

[2] "optimal" is a rather opinionated concept in this case. For the work cited, it was the most redundant (fault tolerant) backbone with the highest vertex coverage (i.e. backbone with highest minimum cut with the highest vertex coverage with that minimum cut).

all vertices of the original graph $G$. Be that as it may, this algorithm can be very expensive if done poorly. As will be discussed in Section 2.3, there are a number of implementation choices that affect the complexity of this algorithm.

## 2.3 Algorithm Engineering

These algorithms presented in the previous section are linear in the number of vertices and edges. However, that does not guarantee that the implementation of those algorithms are. To begin the implementation of the smallest last ordering is discussed.

For the smallest last ordering to be truly linear in the number of vertices and edges of a graph $G$, the complexity of finding the next vertex with the smallest degree must be near constant time. To achieve this complexity, an index can be maintained of degrees to double linked list of points with that degree. Further, given any vertex, its position in its degree list must be reachable in constant time. To accomplish that, a vertex can be either given a pointer to its position or another index identifying vertices to placement in degree list can be created. Regardless, with these data structures, the lowest degree list with any vertices is used to "pop" the next node (i.e $v_j$) off of $G$. With each cutting, the neighbors of $v_j$ are then moved one degree list down (which is a linear operation in the number of edges of $v_j$ since we have the reference to each vertex's placement). In the implementation for this report, each vertex's "current degree" is maintained to quickly discover to which degree list it will be moved. Other implementations are possible, but this report's implementation is clearly linear, as is justified by Figure 1.

Secondly, the greedy coloring algorithm is very simple. To achieve linear time, the complexity of determining the minimum color must be kept linear to the number of edges adjacent to $v_i$. This is easily accomplished in one of two ways. Firstly, since the maximum degree is known, the maximum color cannot be greater than that. A counting sort of the neighborhood colors can thus be used from the base color allowed to the maximum degree. Then a linear pass through the counted set till the first 0 color is reached. This implementation is certainly viable, but has the unnecessary space that plagues counting sorts. With the benchmarks given in this report, such a problem would not pose any significant resource issues. However, another approach, and the approach use in this report's implementation, is to produce a set of colors from the neighborhood colors. A set (in C++) is typically implemented as a Red-Black tree. Thus the size of the Red-Black tree becomes $O(|E_{v_i}|)$ because, at *most* a node will be part of a complete graph or clique in which all edges have different colors. The insert and search time of a Red-Black tree is $\Theta(n)$ however, $n$ in this case is the number of unique colors in the neighbors of $v_i$. Starting at the lowest color to the $i$ th color, $c_i$, the color set is searched for $c_i$ until it is not found within the set. $c_i$ is then assigned to $v_i$. While the set is not constant time lookup or insertion, we have a good estimate with which to bound the complexity. The average number of colors is related to the average degree of $G$. We can say that the algorithm

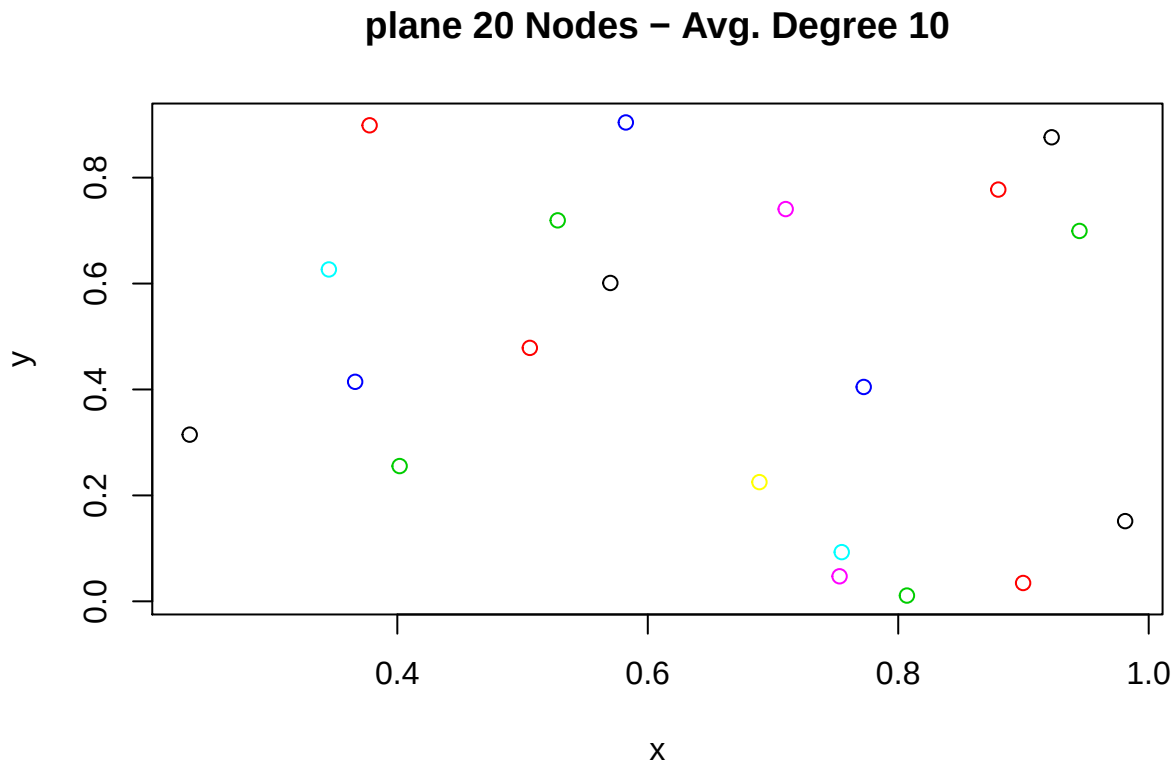



Figure 2: Walkthrough example of RGG

is $\Theta(|V| + |E| \times d(\bar{G}))$ where $d(\bar{G})$ is the average degree. This produces a linear time algorithm in the number of edges and vertices as that dominates the complexity. In reality, the average degree is a conservative estimate, as most vertices' neighborhoods have far fewer colors than the average degree.

## 2.4 Walkthrough and Verification

The algorithms were discussed in depth elsewhere in this report, so this section will focus mostly on visually showing the steps of the algorithm. Figures 2 and 3 show the sample RGG and final Primary Backbone that covers (in this case) 100% of the nodes.

Beginning with our RGG (uncolored), we have what is displayed in Figure **??** . We first want to create our coloring using smallest last order. That algorithm cuts off the node with the minimum degree every iteration. The highlighted node is the first node to cut out (as seen in Figure **??**).

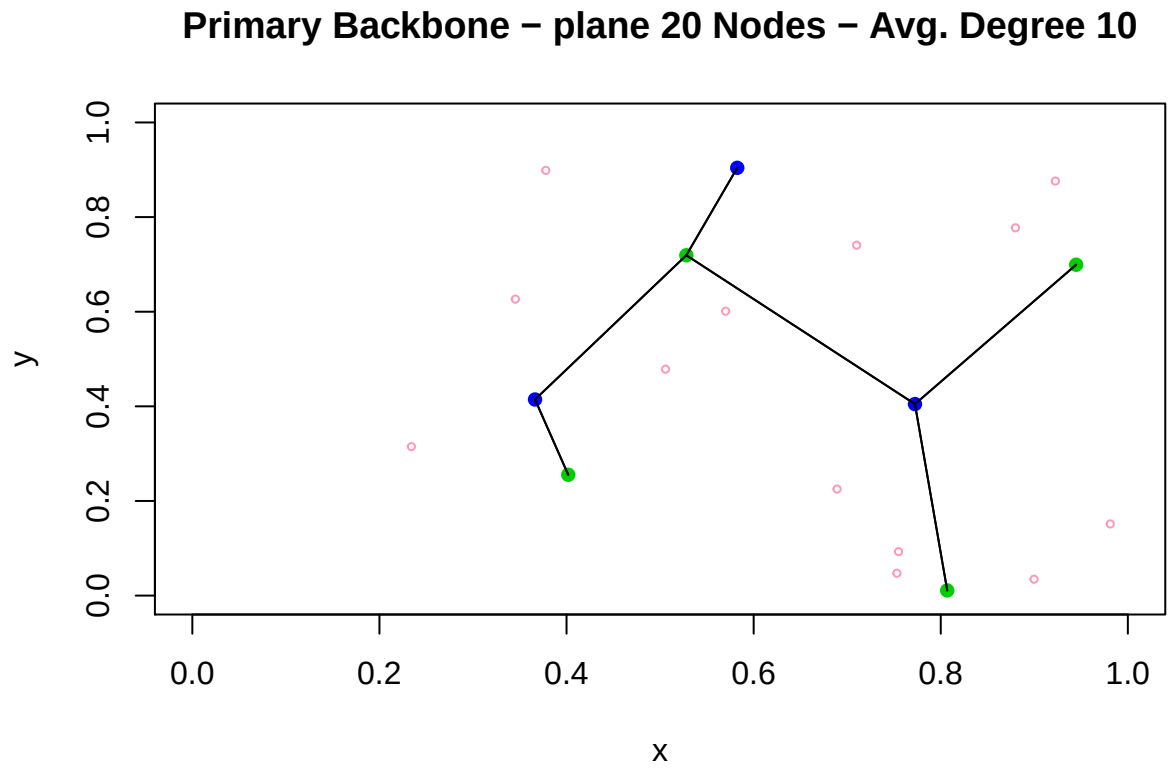
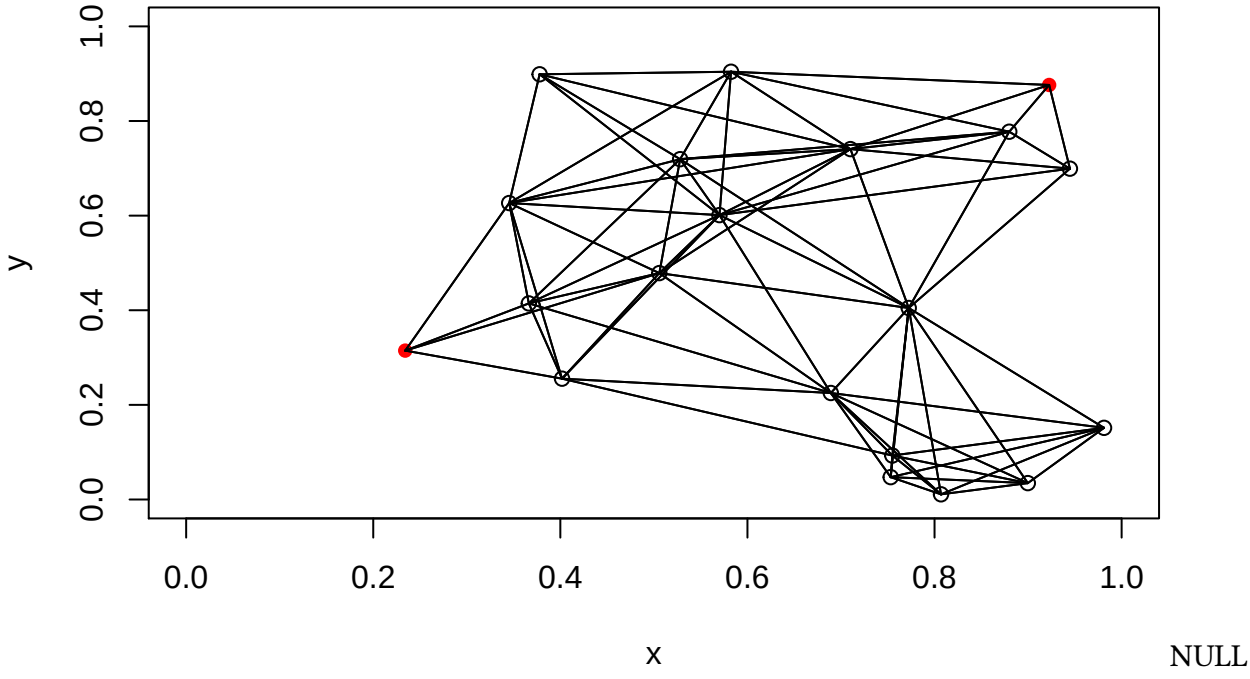
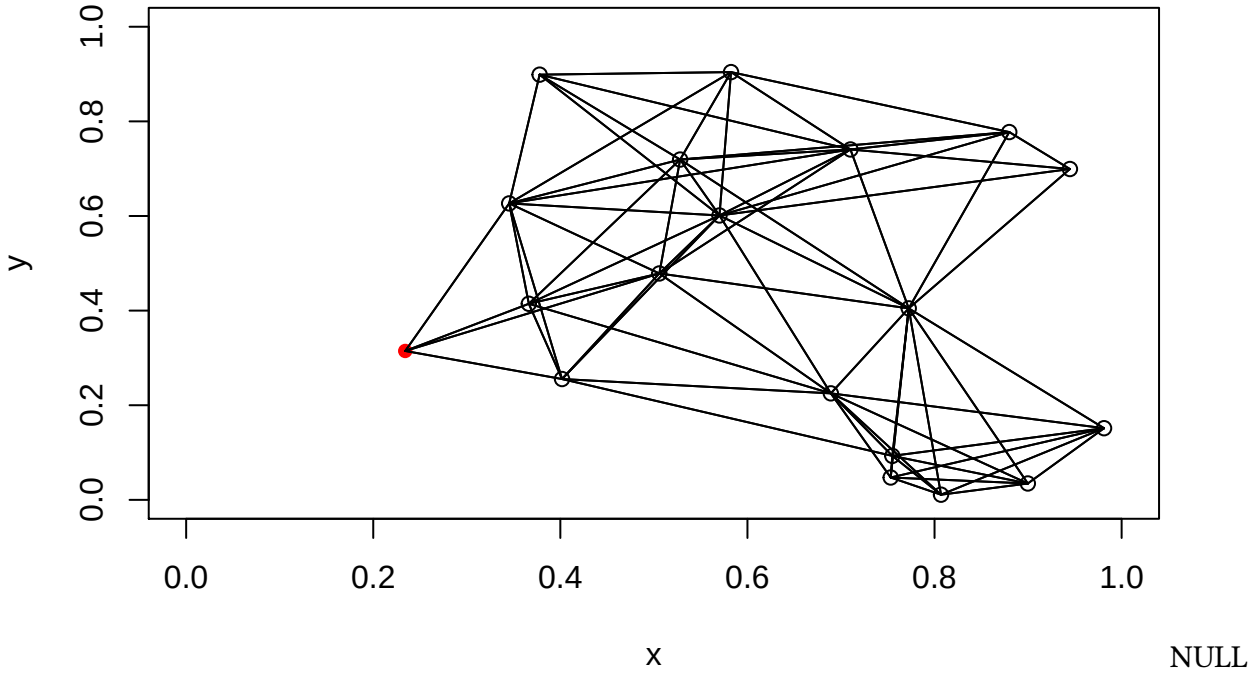


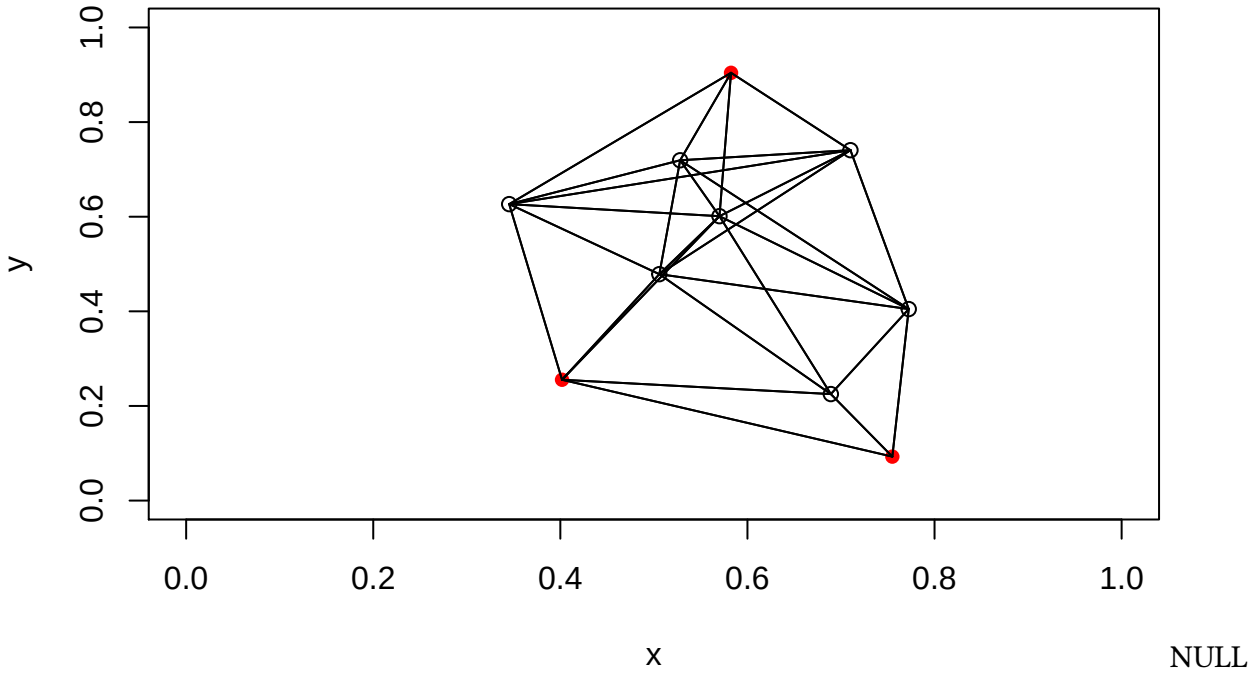

Figure 3: Primary Backbone of RGG




NULL




This process continues as we remove the minimum degree vertex every time. Figure **??** shows the process as it hits the terminal clique. This process continues till there are no more nodes left to cut. Then the we reverse the order in which we took them out to get the smallest last ordering. Figure 4 shows the degree of the nodes when deleted as a function of iterations (additionally the original degree of the node is plotted).




The graph coloring is very simple, as mentioned before. Graphically, it's the reverse of the smallest last ordering (in terms of the degree in question each iteration). Additionally, each node picks its color such that it is the lowest possible color "class" that is not one of its neighbors' colors.

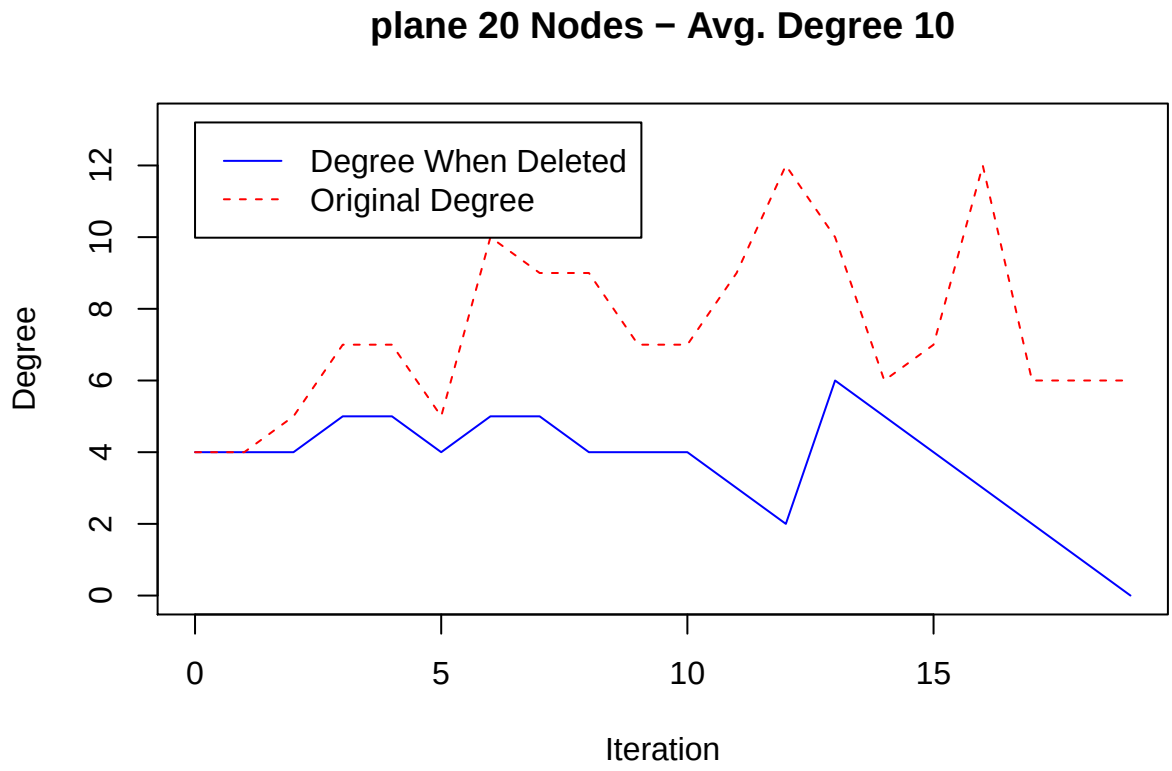



Figure 4: Degree when deleted from Graph, in Smallest Last Order from right to left

Lastly, the bipartite backbone selection is done by selecting those colors classes with the 4 largest frequency (See Figure 2). In this case they are: Black (1), Red (2), Green (3), and Blue (4). All 6 combination of colors (1,2) (1,3) (1,4) (2,3) (2,4) (3,4) are collected. An adjacency list is built of only bipartite nodes. Figure 3 shows the subgraph of the bipartite combination (3,4). In this case, it also happened to be the best choice for backbone with the 100% vertex coverage. However, before we jump ahead to choosing the best backbone, we first have to pick the best **component** of each bipartite graph. A breadth first search is done on the bipartite subgraph to determine the largest connected component of the bipartite subgraph (yes a subgraph of the subgraph). In this instance, the (3,4) combination has only one, large component (the whole bipartite subgraph). Once the each combinations' largest component is found, the vertex coverage is determined for each backbone. This is simply the number of vertices incident to the backbone vertices (i.e. the number of nodes directly connected to the backbone, including the backbone itself) divided by the total number of vertices. Thus we have the ratio of coverage between 0 and 1. In this example, our coverage is 1!

## 2.5 Algorithm Effectiveness

In Table 2 the full results of the benchmarks are given. The overall consensus is positive for the RGG generation, coloring and bipartite backbone selection algorithms. For the RGG generation, the main criteria (in addition to optimized runtime) for evaluating the RGG generation was how close the actual average degree was to the desired average degree. Generally, the plane shaped benchmarks

Table 2: Results

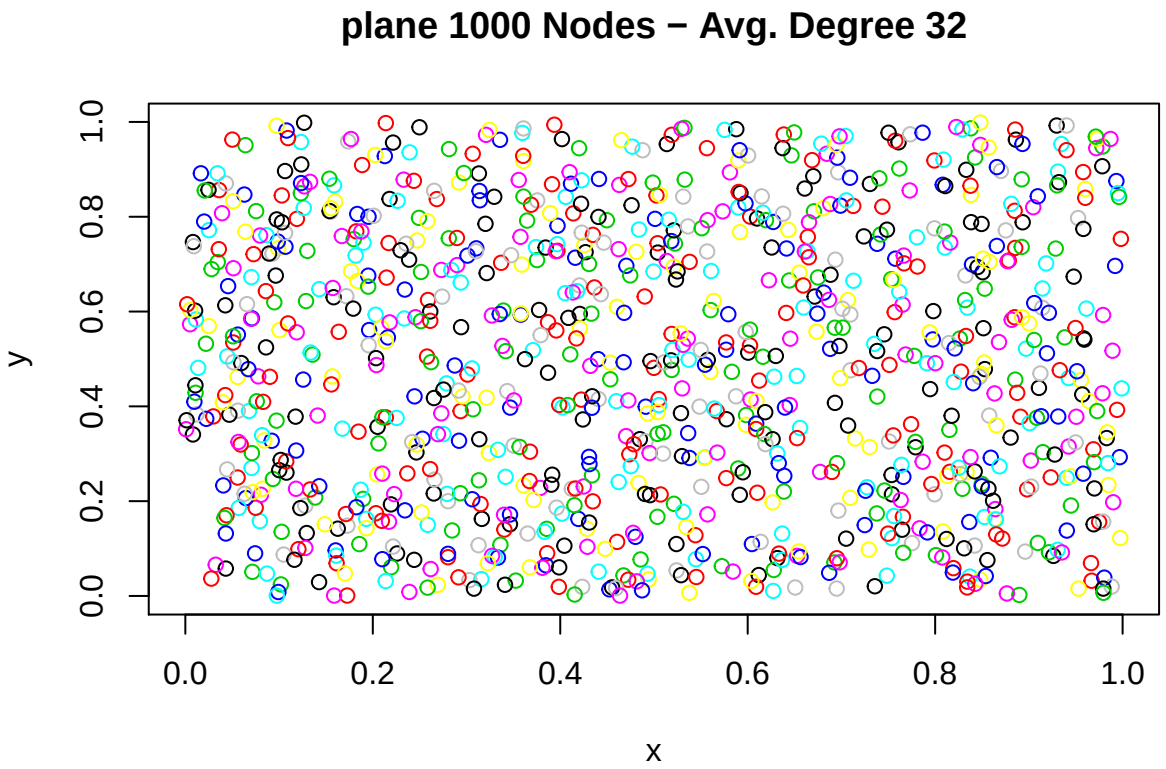
| Number of Nodes | 4000 | 4000 | 1000 | 16000 | 4000 | 64000 | 64000 |
|---|---|---|---|---|---|---|---|
| Desired Avg. Degree | 128 | 64 | 32 | 64 | 64 | 128 | 64 |
| Shape | disk | disk | plane | plane | plane | plane | plane |
| Actual Avg. Degree | 118.08 | 60.20 | 29.02 | 62.10 | 60.11 | 125.25 | 62.99 |
| Radius | 0.179 | 0.126 | 0.101 | 0.036 | 0.071 | 0.025 | 0.018 |
| Number of Edges | 472302 | 240784 | 29018 | 993580 | 240428 | 8016232 | 4031522 |
| Max Degree | 167 | 93 | 46 | 94 | 90 | 169 | 101 |
| Min Degree | 43 | 22 | 7 | 13 | 11 | 23 | 15 |
| Terminal Clique | 70 | 37 | 19 | 39 | 37 | 73 | 41 |
| Max Color Size | 89 | 160 | 78 | 643 | 162 | 1369 | 2534 |
| Primary Backbone Vertices | 3987 | 3983 | 995 | 15951 | 3983 | 63848 | 63692 |
| Primary Backbone Edges | 237 | 386 | 193 | 1641 | 393 | 3717 | 6438 |
| Primary Coverage | 99.67% | 99.58% | 99.50% | 99.69% | 99.58% | 99.76% | 99.52% |
| Secondary Backbone Vertices | 3984 | 3982 | 995 | 15928 | 3982 | 63839 | 63667 |
| Secondary Backbone Edges | 232 | 396 | 178 | 1589 | 395 | 3686 | 6349 |
| Secondary Coverage | 99.60% | 99.55% | 99.50% | 99.55% | 99.55% | 99.75% | 99.48% |

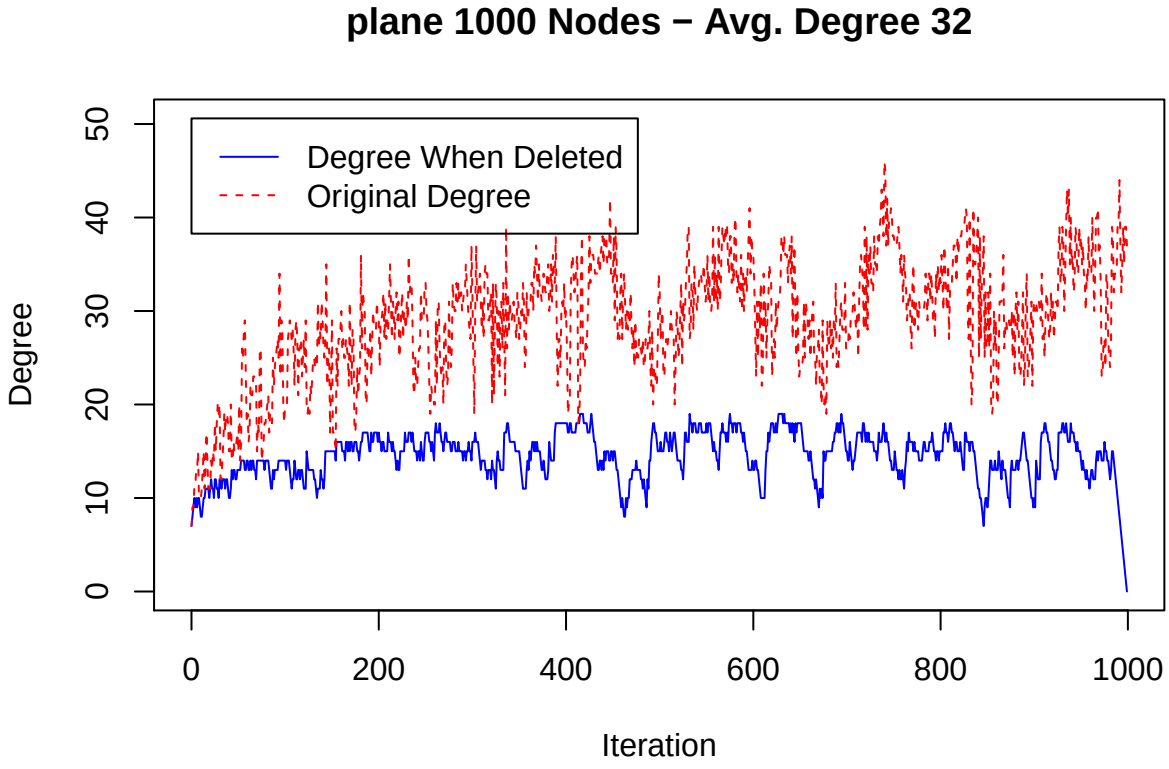
had actual average degrees nearer to the desired. The disk shaped benchmarks were close, but not as close as the plane shaped benchmarks. This likely has to do with the amount of precision given to $\pi$ within the `cmath` header `M_PI` (as it does not have the highest precision at 16 digits). Since the determination of the radius of adjacency $R$ was in relation to $\pi$ for disk shaped areas, it is likely the loss of precision affected the ability to choose a better $R$. For the coloring algorithm, the largest independent sets were almost always the lowest color classes. Further, the degree when deleted versus original degree plots confirm that the smallest last ordering is correct. Lastly, the bipartite selection algorithm performs perfectly to expectation. The smallest vertex coverage is 99.5%.

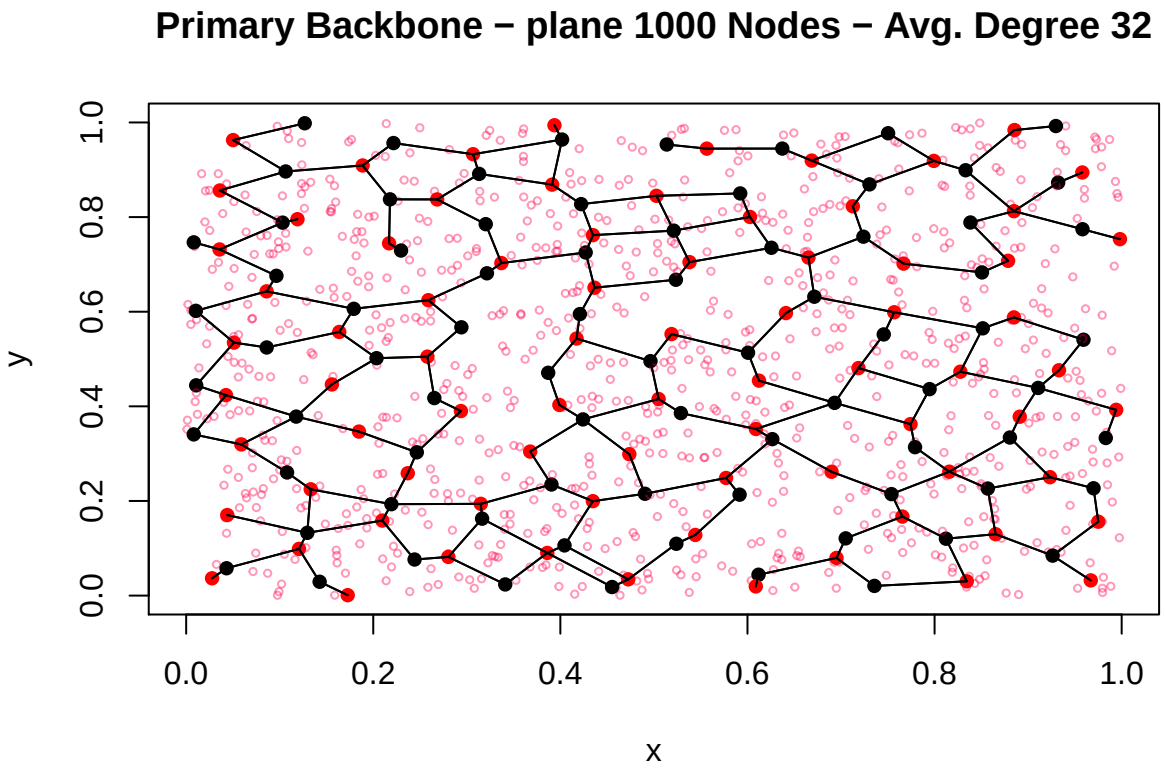
Overall, the only stipulation in the implementation is potentially the RGG generation for disk surfaces for which a refinement has been proposed.

# 3 Simulated Benchmark Results

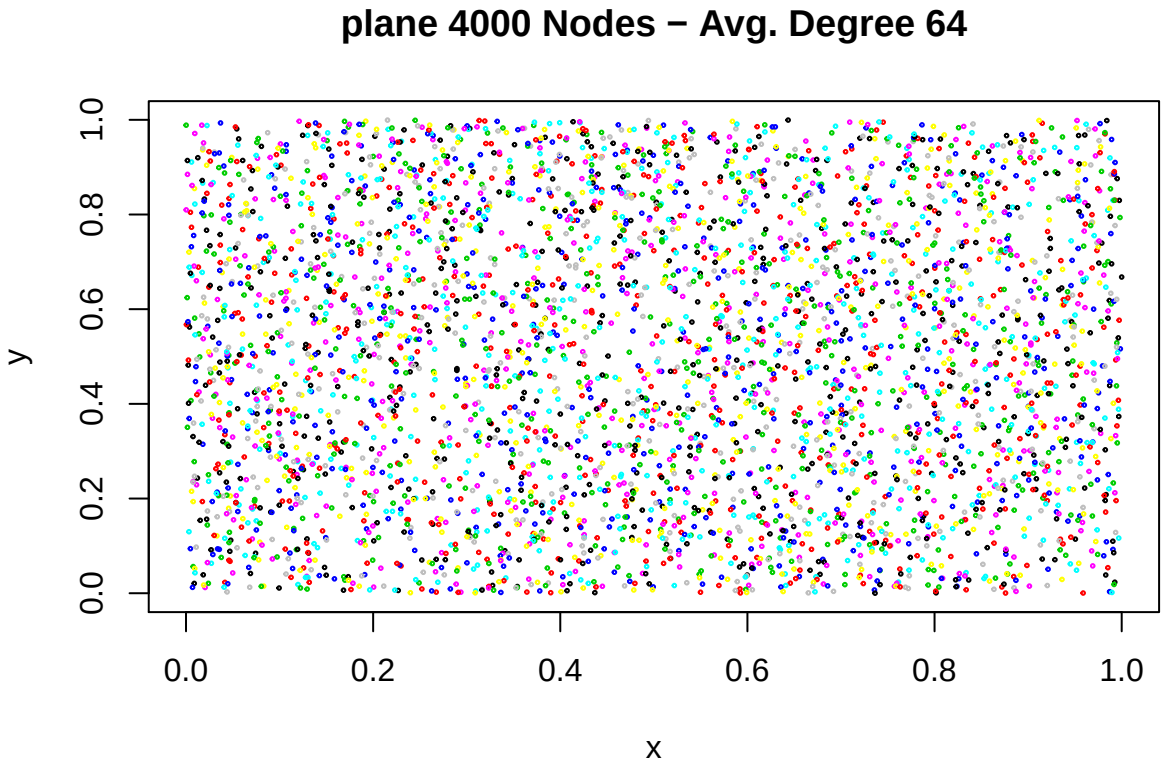
## 3.1 Benchmark 1

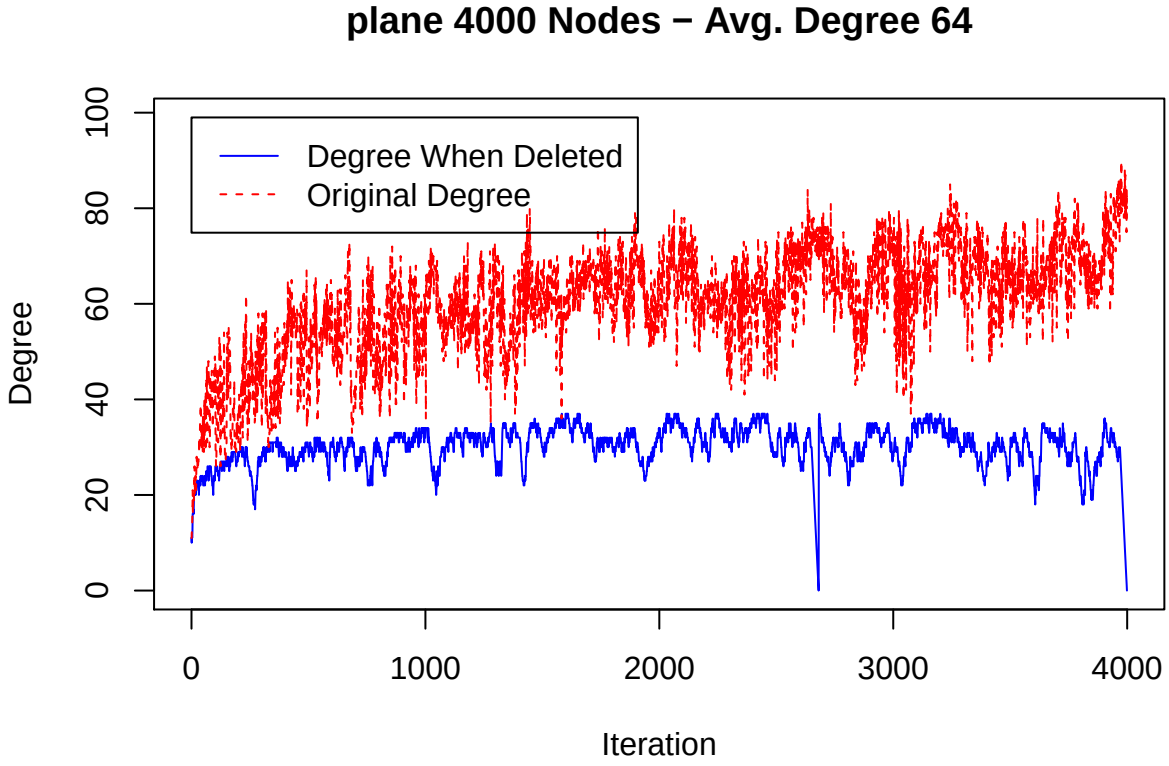


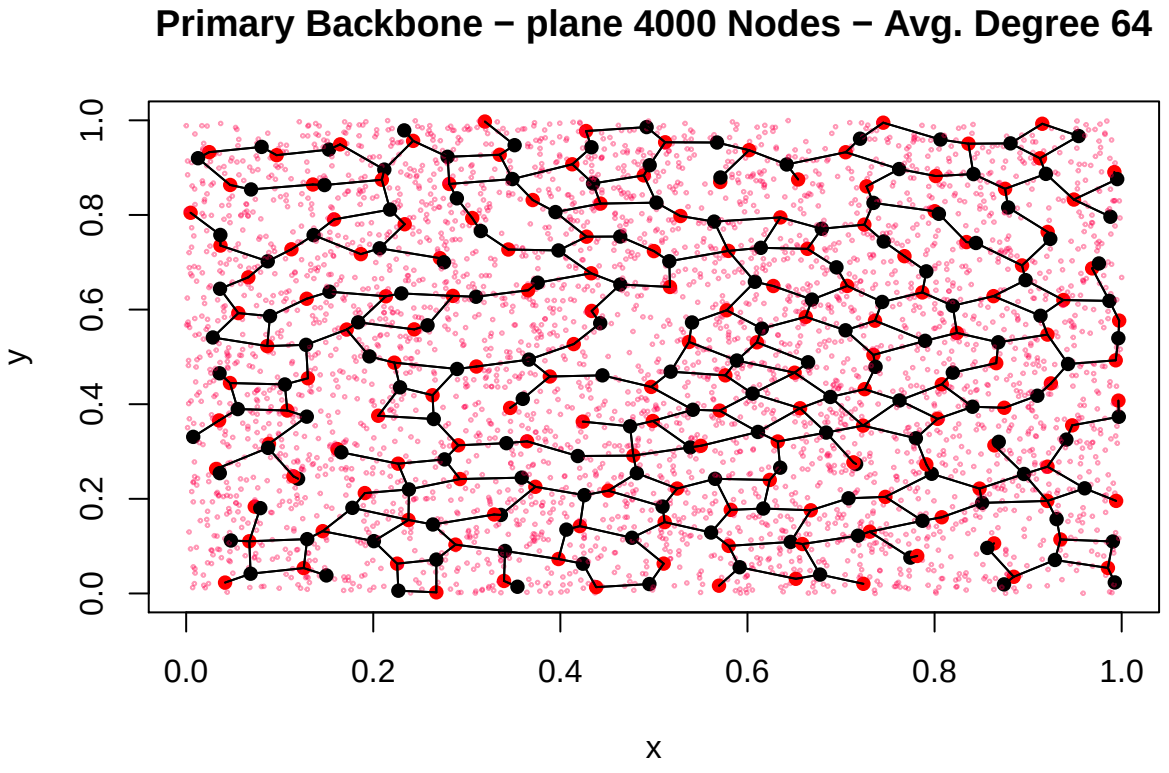





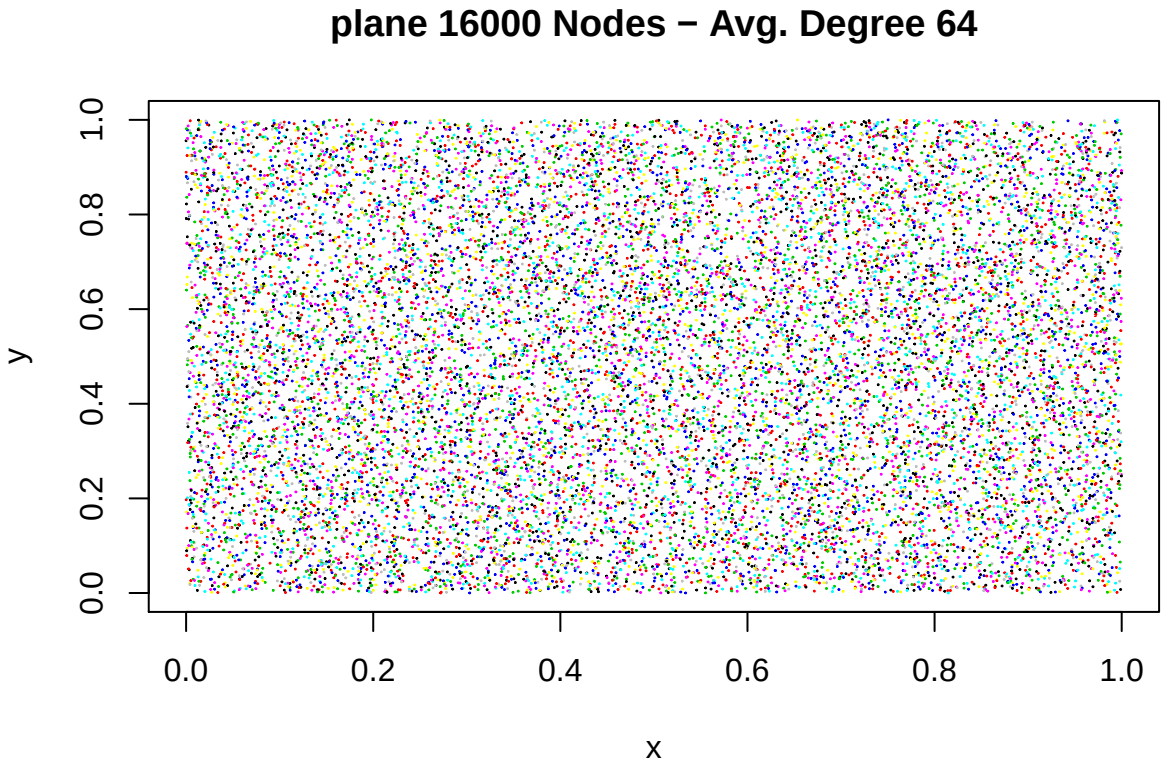

### 3.2 Benchmark 2

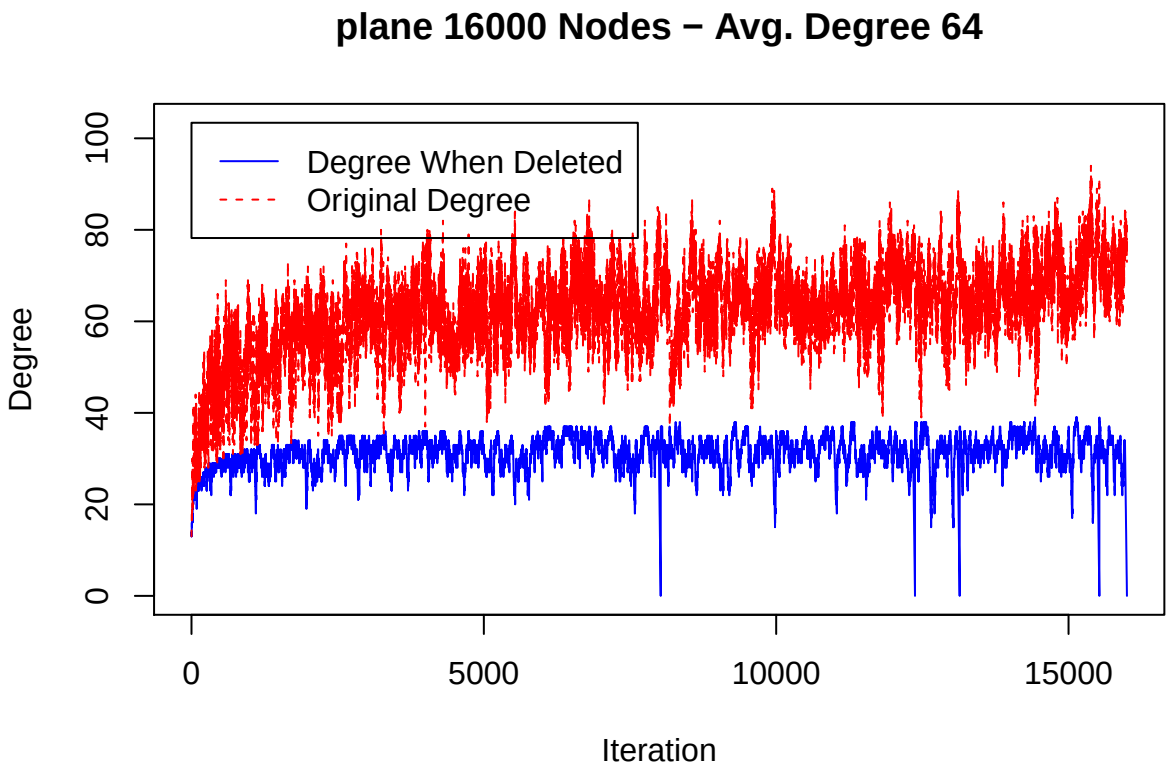


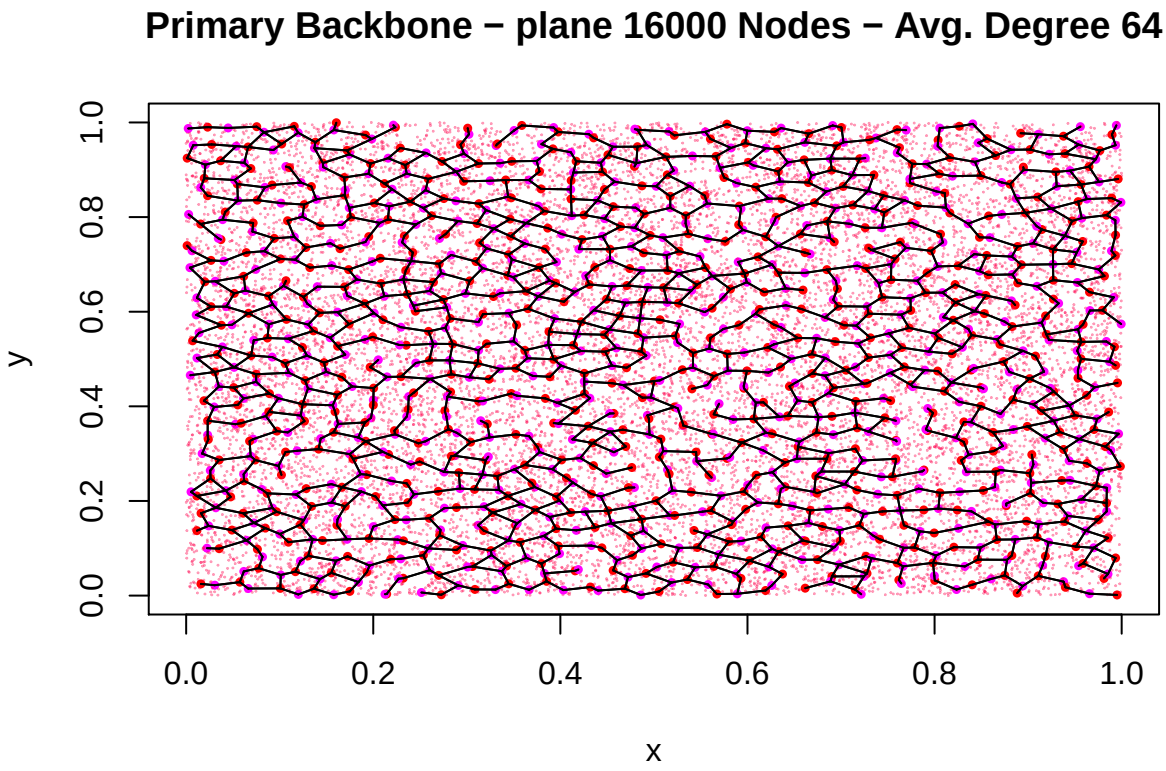

plane 4000 Nodes − Avg. Degree 64
Degree When Deleted
Original Degree
Degree
0
20
40
60
80
100
0
1000
2000
3000
4000
Iteration


Primary Backbone − plane 4000 Nodes − Avg. Degree 64
y
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
x

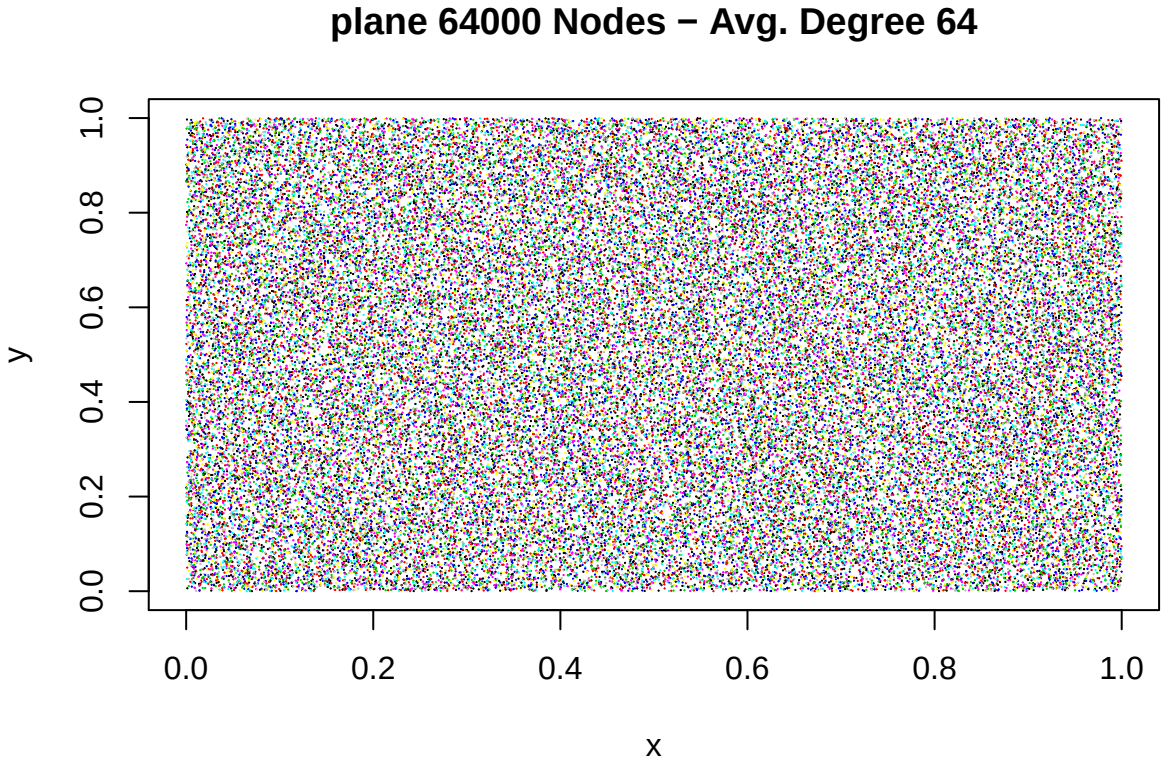
### 3.3 Benchmark 3

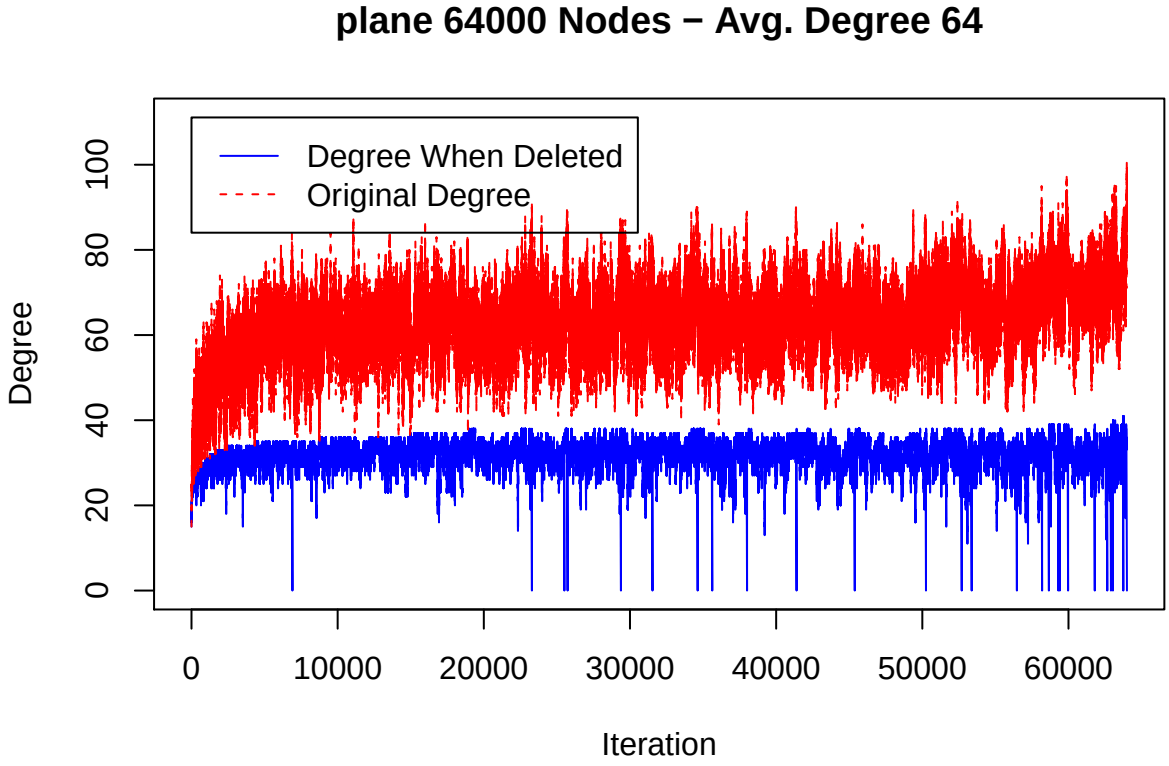


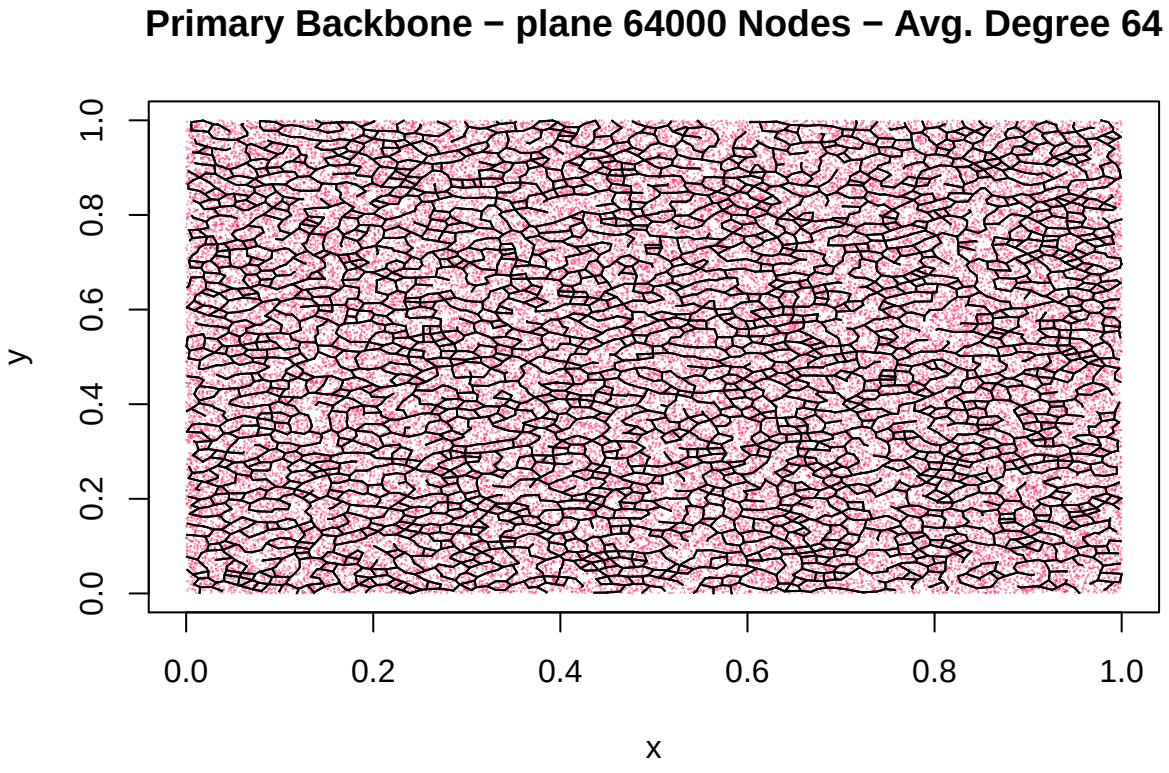





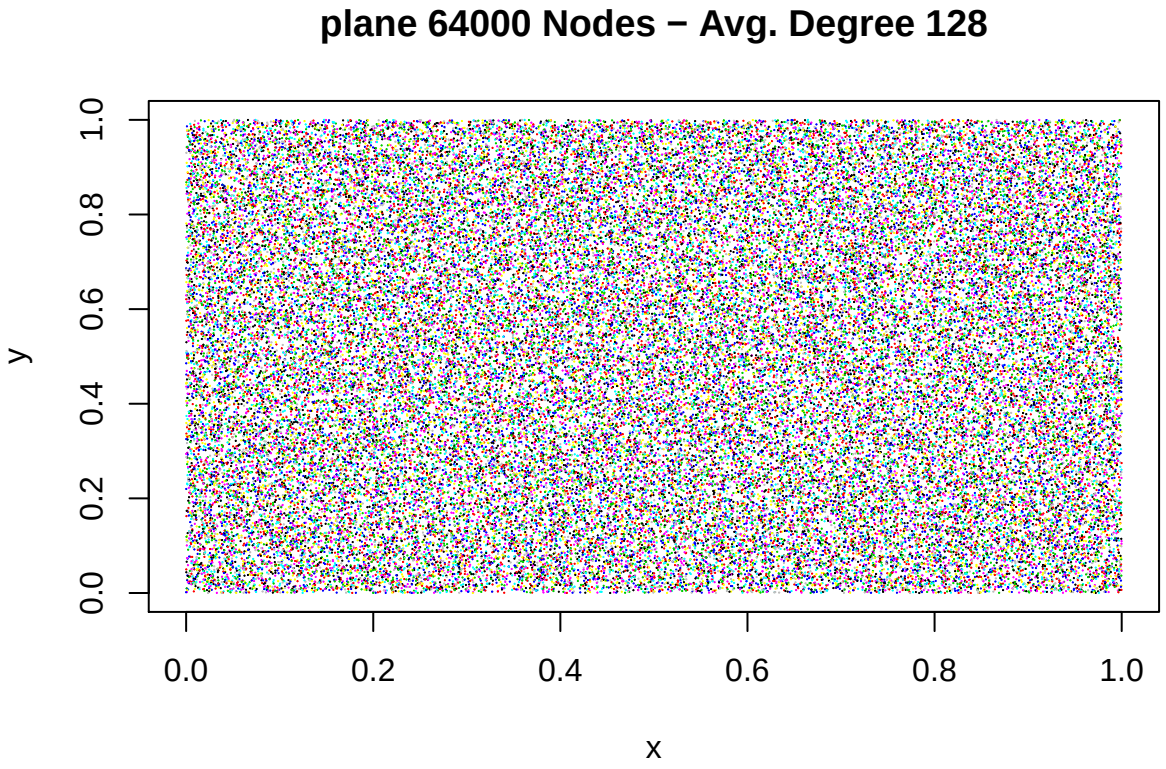

### 3.4 Benchmark 4

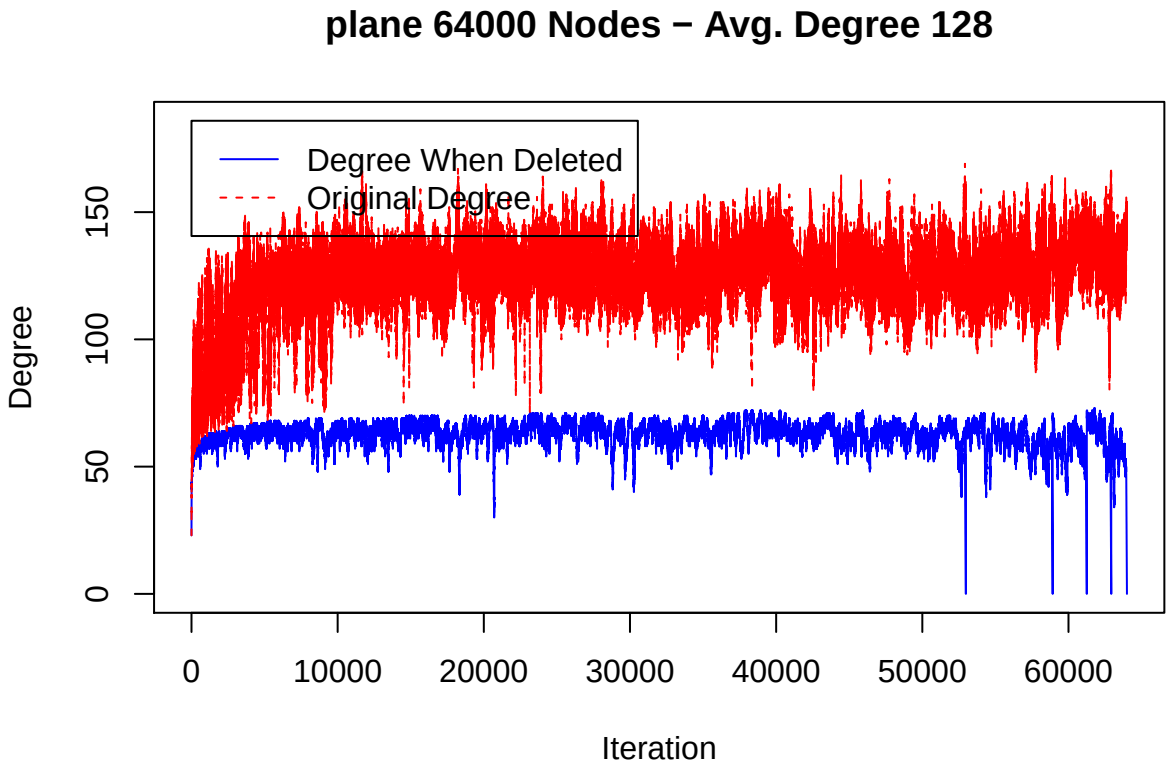


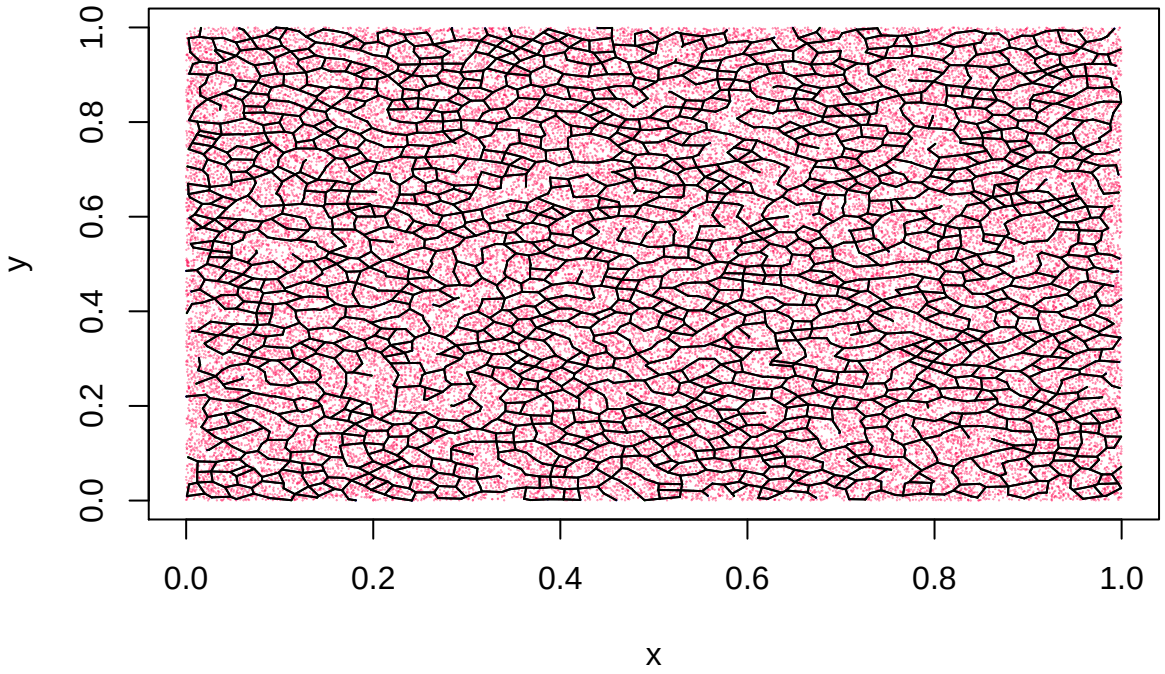

plane 64000 Nodes − Avg. Degree 64
Degree When Deleted
Original Degree
Degree
0
20
40
60
80
100
0
10000
20000
30000
40000
50000
60000
Iteration


Primary Backbone − plane 64000 Nodes − Avg. Degree 64
y
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
x

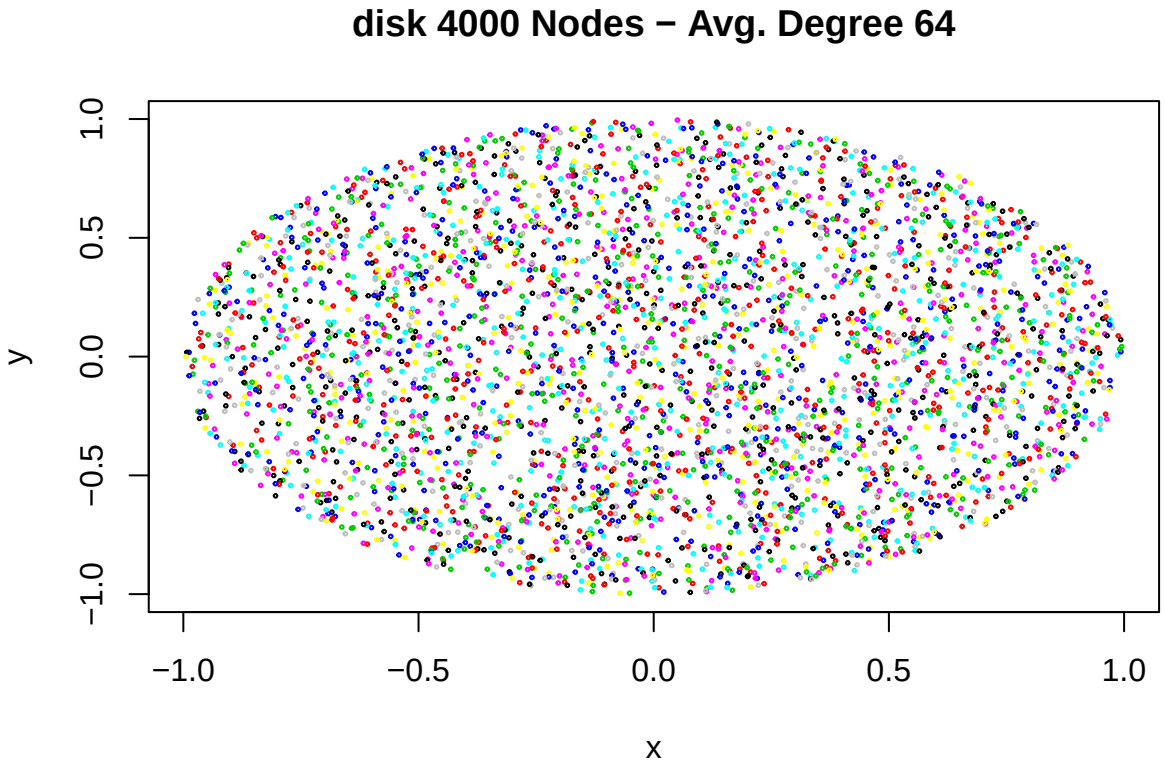
### 3.5 Benchmark 5

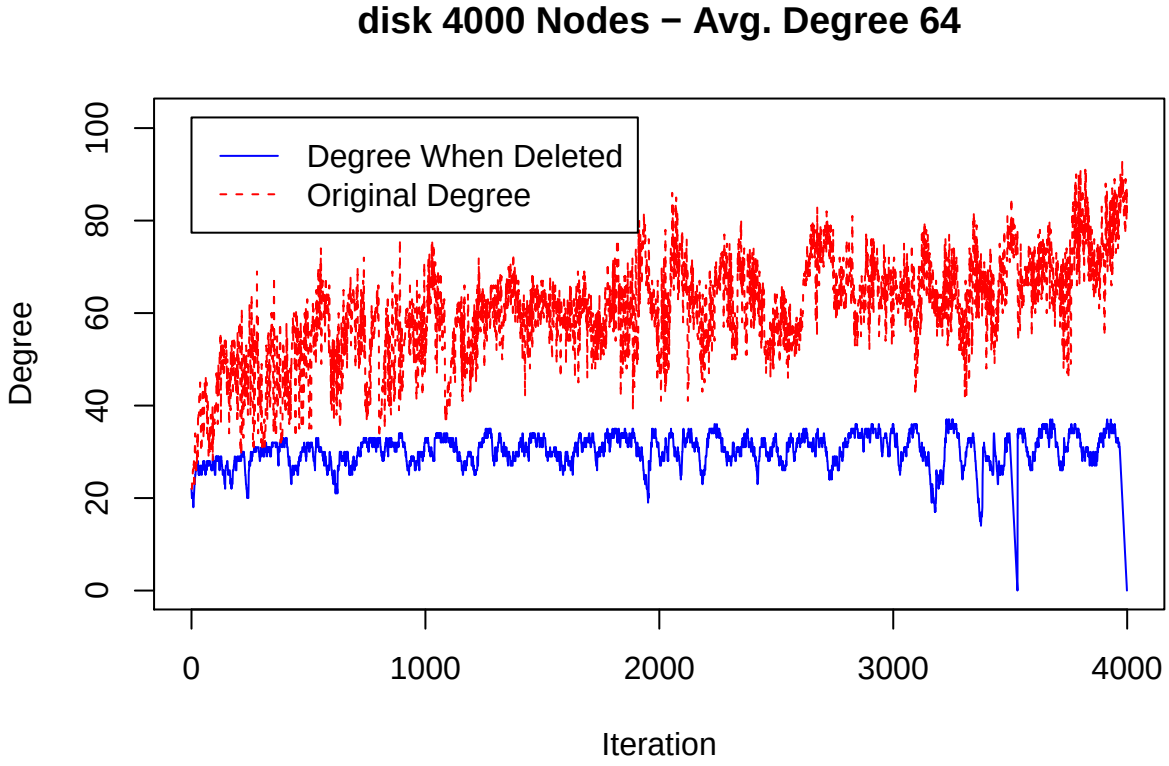


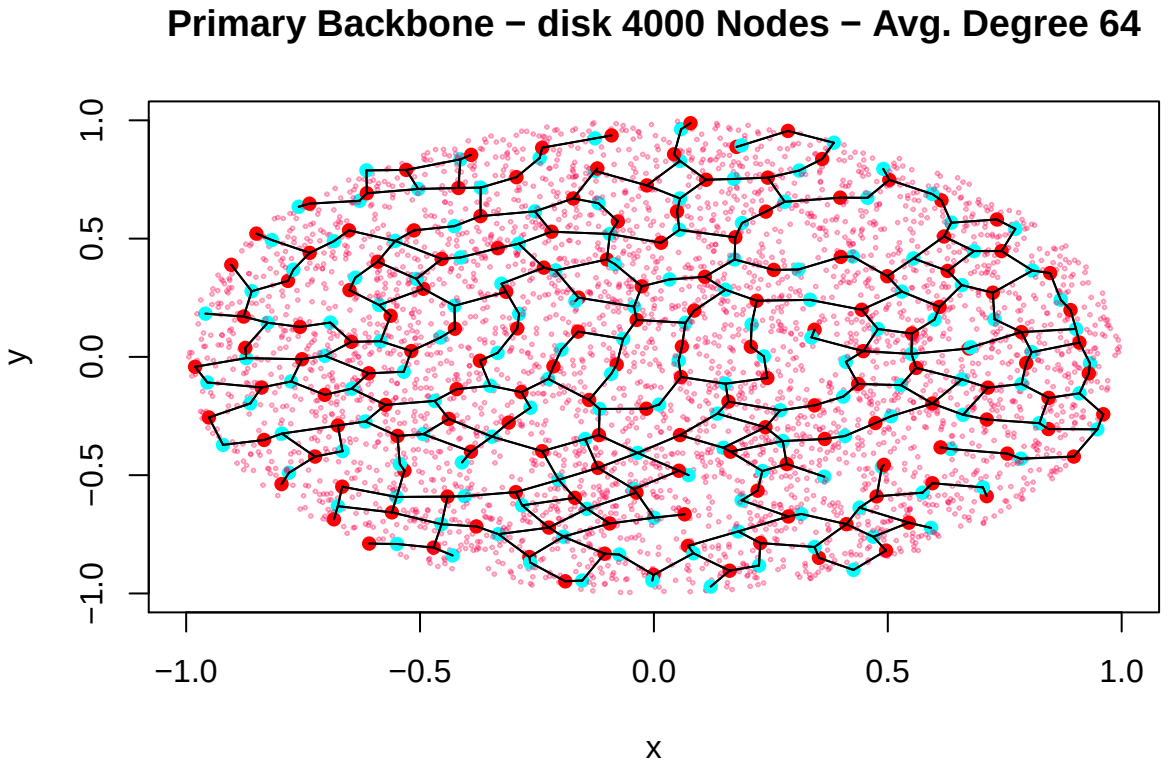





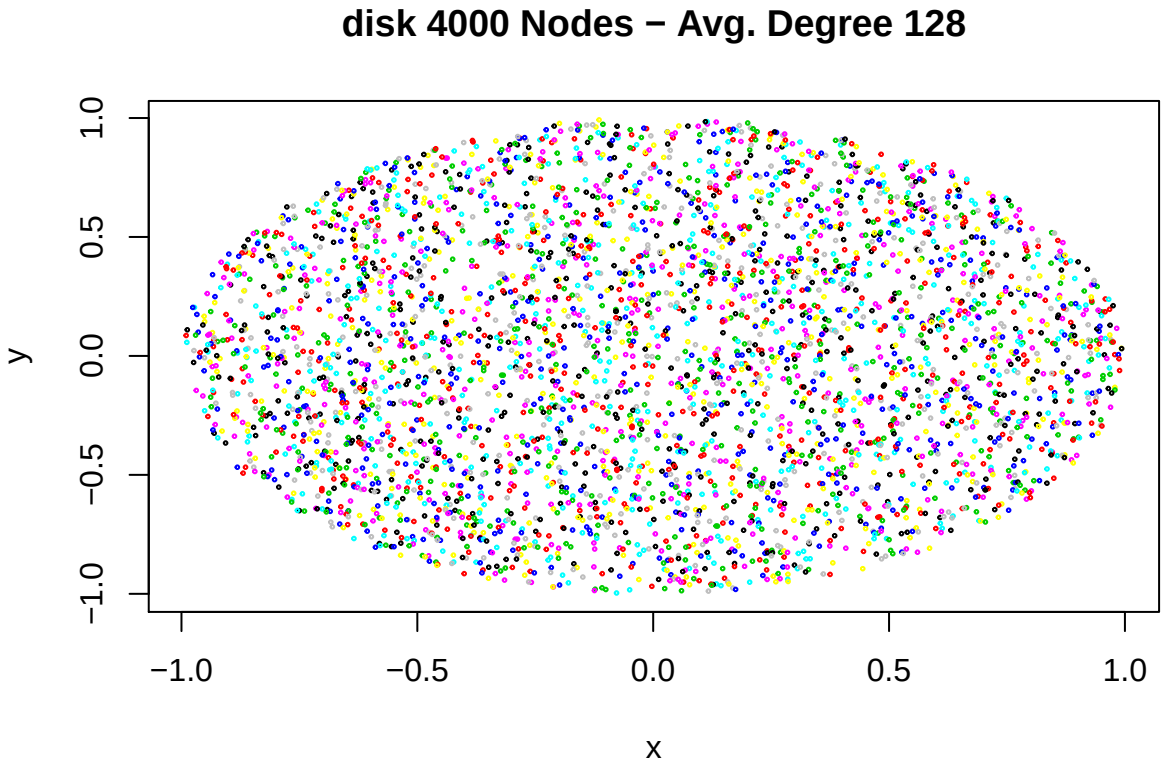

### 3.6 Benchmark 6

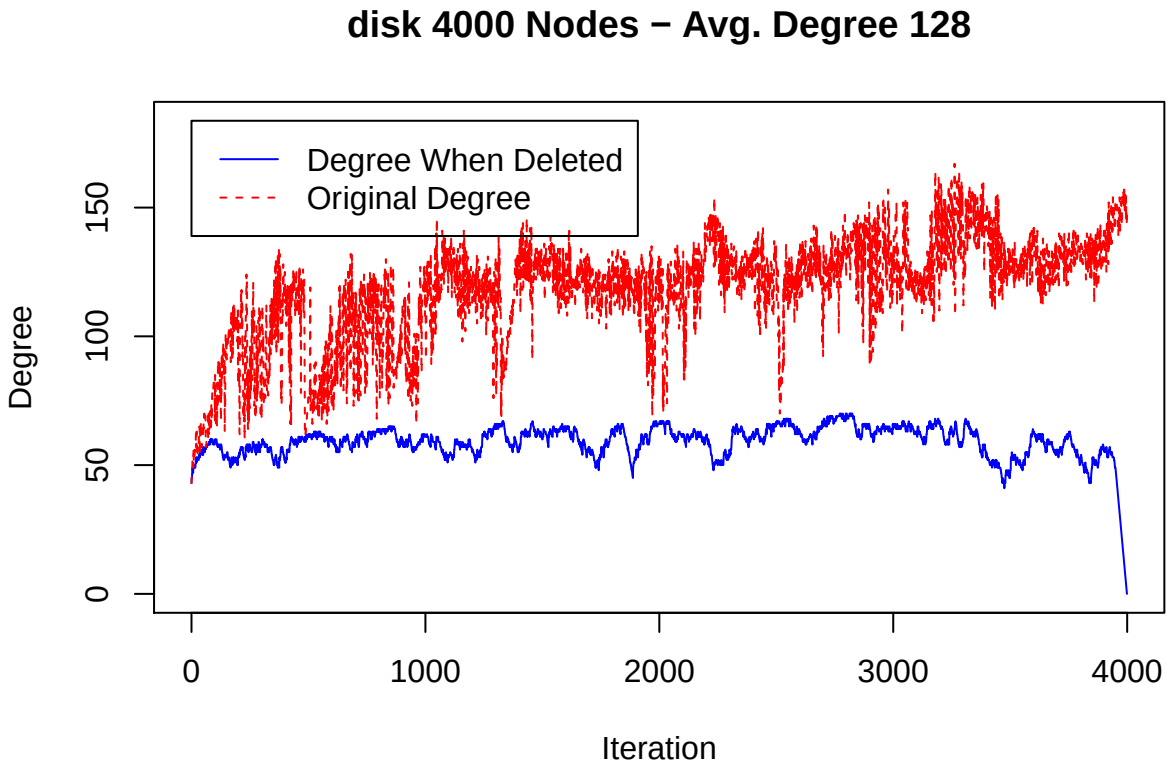


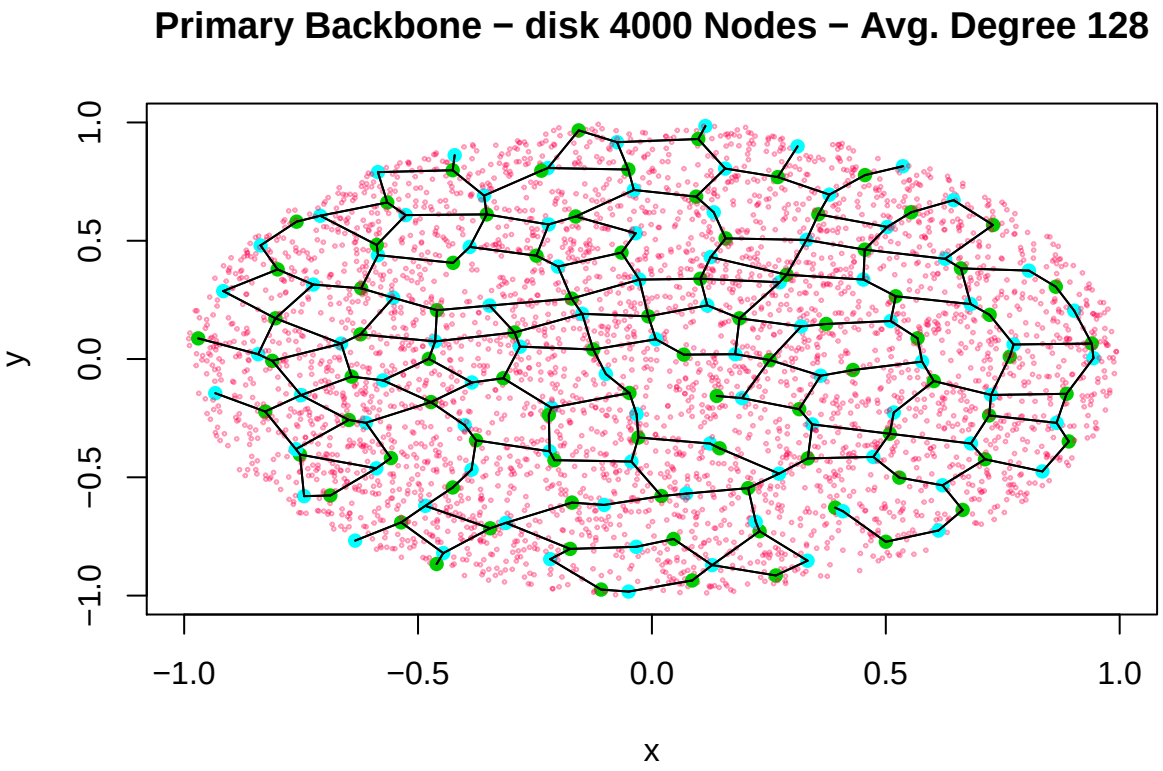

disk 4000 Nodes − Avg. Degree 64
Degree When Deleted
Original Degree
Degree
0
20
40
60
80
100
0
1000
2000
3000
4000
Iteration


Primary Backbone − disk 4000 Nodes − Avg. Degree 64
y
1.0
0.5
0.0
−0.5
−1.0
−1.0
−0.5
0.0
0.5
1.0
x

### 3.7 Benchmark 7






# References


[1] JJ Allaire and et al. *R Markdown*. 2015. URL: http://rmarkdown.rstudio.com.

[2] T.H. Cormen. *Introduction to Algorithms, 3rd Edition:* MIT Press, 2009. ISBN: 9780262033848. URL: https://books.google.com/books?id=VK9hPgAACAAJ.

[3] Mark Handcock et al. "statnet: Software Tools for the Representation, Visualization, Analysis and Simulation of Network Data". In: *Journal of Statistical Software* 24.1 (2008), pp. 1–11. ISSN: 1548-7660. DOI: 10.18637/jss.v024.i01. URL: https://www.jstatsoft.org/index.php/jss/article/view/v024i01.

[4] Dhia Mahjoub and David W. Matula. "Building (1 - $\epsilon$) Dominating Sets Partition as Backbones in Wireless Sensor Networks Using Distributed Graph Coloring". In: *Distributed Computing in Sensor Systems: 6th IEEE International Conference, DCOSS 2010, Santa Barbara, CA, USA, June 21-23, 2010. Proceedings*. Ed. by Rajmohan Rajaraman et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 144–157. ISBN: 978-3-642-13651-1. DOI: 10.1007/978-3-642-13651-1_11. URL: http://dx.doi.org/10.1007/978-3-642-13651-1_11.

[5] Dhia Mahjoub and David W. Matula. "Constructing efficient rotating backbones in wireless sensor networks using graph coloring". In: *Computer Communications* 35.9 (2012). Special Issue: Wireless Sensor and Robot Networks: Algorithms and Experiments, pp. 1086–1097. ISSN:

0140-3664. DOI: http://dx.doi.org/10.1016/j.comcom.2012.02.013. URL: http://www.sciencedirect.com/science/article/pii/S0140366412000722.

[6] David W. Matula and Leland L. Beck. "Smallest-last Ordering and Clustering and Graph Coloring Algorithms". In: *J. ACM* 30.3 (July 1983), pp. 417–427. ISSN: 0004-5411. DOI: 10.1145/2402.322385. URL: http://doi.acm.org.proxy.libraries.smu.edu/10.1145/2402.322385.

[7] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria, 2015. URL: https://www.R-project.org/.