# Linear Time Backbone Determination in a Wireless Sensor Network

Jake Carlson

April 21, 2018

### Abstract

A report on implementing algorithms to partition a random geometric graph into bipartite subgraphs. Three different graph geometries are explored: unit square, unit disk, and unit sphere. Nodes are uniformly distributed in the geometry. Then the edges are determined and the verticies are colored using smallest-last vertex ordering and greedy graph coloring. Once coloring has been used to determine the independent color sets, the combinations of the largest are processed to find the largest backbone. All algorithms used in this report are implemented to run in linear time.

# Contents

# Listings

# 1  Executive Summary

## 1.1  Introduction

Random geometric graphs (RGGs) are useful for simulating wireless sensor networks placed in different topologies. This project examines three different geometries: Square, Disk, and Sphere. The user supplies parameters for how many nodes they want in the network and how many connections they want for each node. Then, the simulation finds the average radius needed for that number of connections, determines the edges in the graph, colors the graph to find independent sets, pairs the four largest independent sets to find the largest bipartite subgraphs, and cleans these bipartites to find the major component, or backbone, of each bipartite. The cleaning ensures that there are no singular points of failure that could cause the network to become disconnected. In other words, each backbone exists so that there are multiple paths between any two nodes in the backbone.

This creates network backbones from the random geometric graphs that are highly reliable and allow the largest number of wireless sensors to connect to it in only one hop. Additionally, the linear time implementation of this simulation ensures efficient running time regardless of the input size. The organization of the code base also makes it easy to implement new topologies by subclassing the main Topology class that implements all of the algorithms needed to determine the backbone.

All of the code used for this project, including the graphical display of the generated graphs at each stage in the backbone determination process, can be found here:

<p align="center">https://github.com/jakecarlson1/sensor-network</p>

## 1.2  Environment Description

The data structures and topologies for this simulation are implemented in Python2.7. The graphics are generated using Processing.py [3]. All development and benchmarking has been done on a 2014 MackBook Pro with a 3 GHz Intel Core i7 processor and 16 GB of DDR3 RAM running macOS High Sierra 10.13.3.

Processing offers an easy to use API for drawing and rendering shapes two- and three-dimensions. The Processing.py implementation allows the entire use of the Python programming languages and libraries.

A separate data generation script was used to generate the summary tables (Tables 1, 2, 3). Because these benchmarks were run in a separate script, the timing does not measure the time required to draw the graphs using Processing. The figures were genetared using the matplotlib library [4]. This library, and a variety of others, could not be imported into Processing.py because the jython interpreter used by Processing only accepts libraries written in raw Python.

The different geometries were implemented in a stand alone Python file and imported into the Processing.py script or the data generation script, depending on what was being run. These classes can then be used directly by Processing or the data generation script. Because there is no intermediary file to hold the generated nodes and edges, there is no additional disk space needed to run the simulation. Everything can be done in system memory managed by Processing.

| Benchmark | Order | A | Topology | r | Size | Realized A | Max Deg | Min Deg | Run Time (s) |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1000 | 32 | Square | 0.101 | 14865 | 29 | 48 | 4 | 0.094 |
| 2 | 8000 | 64 | Square | 0.050 | 245108 | 61 | 93 | 17 | 1.255 |
| 3 | 16000 | 32 | Square | 0.025 | 250658 | 31 | 58 | 6 | 1.593 |
| 4 | 64000 | 64 | Square | 0.018 | 2019116 | 63 | 98 | 16 | 11.124 |
| 5 | 64000 | 128 | Square | 0.025 | 4010430 | 125 | 182 | 35 | 18.915 |
| 6 | 128000 | 64 | Square | 0.013 | 4051390 | 63 | 98 | 17 | 21.506 |
| 7 | 128000 | 128 | Square | 0.018 | 8075034 | 126 | 175 | 29 | 38.348 |
| 8 | 8000 | 64 | Disk | 0.045 | 248036 | 62 | 93 | 16 | 1.209 |
| 9 | 64000 | 64 | Disk | 0.016 | 2023518 | 63 | 104 | 15 | 10.547 |
| 10 | 64000 | 128 | Disk | 0.022 | 4015227 | 125 | 173 | 35 | 18.752 |
| 11 | 16000 | 64 | Sphere | 0.126 | 511920 | 63 | 91 | 35 | 19.625 |
| 12 | 32000 | 128 | Sphere | 0.126 | 2049089 | 128 | 177 | 84 | 79.037 |
| 13 | 64000 | 128 | Sphere | 0.089 | 4094059 | 127 | 173 | 87 | 148.707 |

Table 1: Benchmarks for generating RGGs. A: input average degree, r: node connection radius

| Benchmark | Max Deg Deleted | Color Sets | Largest Color Set | Terminal Clique Size |
|---|---|---|---|---|
| 1 | 22 | 22 | 76 | 21 |
| 2 | 40 | 37 | 320 | 35 |
| 3 | 25 | 24 | 1138 | 22 |
| 4 | 40 | 39 | 2530 | 38 |
| 5 | 73 | 64 | 1373 | 60 |
| 6 | 40 | 39 | 5044 | 36 |
| 7 | 74 | 68 | 2739 | 62 |
| 8 | 39 | 37 | 321 | 31 |
| 9 | 43 | 40 | 2538 | 36 |
| 10 | 72 | 64 | 1371 | 57 |
| 11 | 39 | 38 | 631 | 36 |
| 12 | 87 | 66 | 674 | 61 |
| 13 | 89 | 66 | 1351 | 61 |

Table 2: Benchmarks for coloring RGGs

| Benchmark | B1 Order | B1 Size | B1 Domination | B1 Faces | B2 Order | B2 Size | B2 Domination | B2 Faces |
|---|---|---|---|---|---|---|---|---|
| 1 | 114 | 296 | 0.924 | - | 120 | 290 | 0.961 | - |
| 2 | 546 | 1490 | 0.955875 | - | 558 | 1472 | 0.97175 | - |
| 3 | 1779 | 4458 | 0.9163125 | - | 1726 | 4286 | 0.8928125 | - |
| 4 | 4471 | 11894 | 0.977484375 | - | 4450 | 11854 | 0.9753125 | - |
| 5 | 2559 | 7170 | 0.991296875 | - | 2546 | 7122 | 0.9895 | - |
| 6 | 9106 | 24284 | 0.9815078125 | - | 8954 | 23816 | 0.9807265625 | - |
| 7 | 5169 | 14476 | 0.993421875 | - | 5186 | 14444 | 0.9950546875 | - |
| 8 | 572 | 1504 | 0.984 | - | 558 | 1500 | 0.980375 | - |
| 9 | 4544 | 12124 | 0.9830625 | - | 4522 | 12000 | 0.982625 | - |
| 10 | 2587 | 7280 | 0.99525 | - | 2599 | 7272 | 0.993234375 | - |
| 11 | 1176 | 3166 | 0.992875 | 1992 | 1166 | 3128 | 0.9918125 | 1964 |
| 12 | 1293 | 3616 | 0.99875 | 2325 | 1284 | 3596 | 0.99728125 | 2314 |
| 13 | 2613 | 7390 | 0.99765625 | 4779 | 2603 | 7260 | 0.997703125 | 4659 |

Table 3: Benchmarks for backbone determination

# 2 Reduction to Practice

## 2.1 Data Structure Design

The primary data structure used for this project is an adjacency list. However, to allow for constant time lookup of edges of a node, a Python dictionary is used where the keys are nodes and the values are a list of indicies of adjacent nodes in the original list of nodes. The space needed by the adjacency list is $\Theta(|V| + 2|E|)$. Two entries are used for each edge because they are undirected. This is superior to the adjacency matrix data structure which would require $\Theta(|E|^2)$ space.

In order to make this project maintainable as it is developed along the semester, the object-oriented capabilities of Python are used to design the different geometries. First, a Topology class is defined that creates the interface Processing uses to draw the graphs. This base class implements all of the methods needed for node placement and edge detection in 2D graphs. Then, three subclasses are created: Square, Disk, and Sphere.

The Square and Disk topologies simply need to override the methods for generating nodes and calculating the node radius needed for the desired average degree. The Sphere subclass needs to override a few additional functions because it exists in a 3D space. Other than the methods for generating nodes and calculating the node radius, it also needs to override the function used to draw the graph so that Processing will render the graph properly in 3D.

## 2.2 Algorithm Descriptions

### 2.2.1 Node Placement

A different node placement algorithm is required for each of the geometries. For the Square, the coordinates for each node are generated as two random numbers taken from a unifrom distribution on the range $[0, 1]$. All of these points are guaranteed to be in the unit square.

For the Disk, a similar method is used. The coordinates for nodes are randomly sampled from a uniform distribution; however, if a node has a distance from the center of the Disk greater than the radius of 1, the coordinates for that node are resampled.

For the Sphere a different method must be used so that all of the nodes are placed on the surface of the Sphere and the volume is vacant. For this geometry, the following equations are used:

$$x = \sqrt{1 - u^2} \cos \theta \tag{1}$$
$$y = \sqrt{1 - u^2} \sin \theta \tag{2}$$
$$z = u \tag{3}$$

where $\theta \in [0, 2\pi]$ and $u \in [-1, 1]$. This is guaranteed to uniformly distribute nodes on the surface area of the sphere [5].

All of these algorithms can be solved in $\Theta(|V|)$ where because each node only needs to be assigned a position once.

### 2.2.2 Edge Determination

To calculate the node connection radius needed to acheive the desired average connection, the ratio of node coverage to the total area can be used. This ratio must equal the ratio of the total number of nodes to the average degree, or:

$$\frac{A_{geometry}}{A_{node}} = \frac{Num\,Nodes}{Avg\,Deg} \tag{4}$$

Applying this to each geometry only requires filling in the equation for the area of the geometry and the connection area. This is straight forward for the square and disk. The geometry areas are given by $R^2 = 1$ and $\pi R^2 = \pi$ respectively since these are the unit square and circle. The sphere is slightly more complicated. Since nodes should only be able to connect over the surface of the sphere (following arcs), the connection area is to be taken as the surface area of the spherical cap such that the arc of the cap is twice the length of the connection distance. In other words, a node placed on the surface of the sphere in the center of a spherical cap can connect to any other node that falls in that spherical cap. The equation for the area of the spherical cap is given by

$$S_{cap} = \pi(a^2 + h^2) \tag{5}$$

where $a$ is the distance from the midpoint of the base of the cap to the edge of the base, and $h$ is the distance from the midpoint of the base to the top of the cap (where the node would be) [6]. If we connect these points with a third variable, $x$, such that $x$ is the actual distance from the node to the edge of its connection area, the Pythagorean theorem can be used to substitute in $x^2$ for $a^2 + h^2$. The equation for the node connection radius of the unit sphere then looks identical to that of the unit circle. The final list of equations used to calculate node connection radius for a desired average degree are given in Table 4.

| Geometry | Geometry Area | Node Area | r |
|----------|---------------|-----------|---|
| Square | 1 | $\pi r^2$ | $r = \sqrt{\frac{Average\,Deg}{\pi \times Num\,Nodes}}$ |
| Disk | $\pi$ | $\pi r^2$ | $r = \sqrt{\frac{Average\,Deg}{Num\,Nodes}}$ |
| Sphere | $4\pi$ | $\pi r^2$ | $r = 2 \times \sqrt{\frac{Average\,Deg}{Num\,Nodes}}$ |

Table 4: Equations for node conneciton radius

There are several methods for finding the edges in the graph. The brute force method checks every node, and for each node checks all other nodes to see if they are close enough to form an edge. The brute force method is $\Theta\left(|V|^2\right)$.

The second method to find the edges is the sweep method. This method first sorts the nodes along the x-axis. Then, for any node, we only need to search left and right until the distance along the x-axis is greater than the connection radius for the nodes. This dramatically reduces the search space. The sweep method is $O\left(nlg(n) + 2rn^2\right)$ where $n = |V|$ an $r$ is the connection radius. The $nlg(n)$ portion is for the sorting and the $2rn^2$ portion is for measuring the distance between nodes in a sweep step.

The final method to find edges is the cell method. This method places the nodes into cells of area $r \times r$ based on their position in the topology. When the edge detection runs, each node needs to be visited once, but only the cell the node populates and the neighboring cells need to be searched for connections.

The only method that needs to be adjusted for the Sphere is the cell method. Instead of using a two dimensional grid of cells, a three dimentional mesh is needed to divide the topology. The cells then have volume $r \times r \times r$. Only the current cell and the neighboring cells need to be searched.

### 2.2.3  Graph Coloring

Two algorithms are used for coloring the graphs. The first is smallest-last vertex ordering, which sorts the verticies based on the number of degrees they have. The second is the greedy graph coloring algorithm.

Smallest-last vertex ordering is used to order the nodes for coloring. The steps to this algorithm are as follows [1]:

1. Initialize a representation of your target graph

2. Find the vertex $v_j$ of minimum degree in your representation

3. Update your representation to simulate deleting $v_j$

4. If there are still verticies in the representation, return to step 1, otherwise terminate with the sequence of verticies removed

This algorithm is linear if each of the above steps is linear. Step 1 is linear if we can build a represenation of the graph in linear time. For this, we can use an array of buckets, where each bucket holds the verticies that have the same number of edges as the position of the bucket in the array of buckets. To build this data structure, each node only needs to be visited once, making this linear in both space and time. Next, finding the vertex of minimum degree simply requires finding the lowest index bucket that has a node. This is bounded by the number of buckets, which is bounded by the number of nodes, making Step 2 linear. Next, we have to update the representation of the graph. To do this, we have to look at each node that shares an edge with $v_j$ and move it to the bucket for nodes with one fewer degree. This requires traversing the list of edges for $v_j$ which means Step 3 is linear. Since this is repeated for each node, the runtime of this program is $\Theta\left(|E| + |V|\right)$ and the space needed is $\Theta(|V|)$.

After this, a single traversal of the smallest-last vertex ordering is needed to color the graph. As we traverse this list, we check to see if the nodes before it (that are already colored) share an edge with the current node. The node can then be colored with any color it does not share an edge with or, if it shares an edge with all currently used colors, it is assigned a new color. This algorithm is also linear. Each node needs to be visited once and when a node is visited, all previous nodes are checked to see if they are in the edge list of the current node. Because we used smallest last vertex ordering, as we have to check more and more nodes, we get to check fewer and fewer edges. This makes the greedy coloring algorithm $O(|V| + |E|)$.

### 2.2.4 Backbone Determination

Several algorithms are needed for determining the most suitable backbones for the wireless sensor network. First, the four largest independent sets are paired with each other to generate the largest bipartite subgraphs for the random geometric graph. These bipartites are bound to have minor components that are not connected to the major component, and blocks that are only connected by bridges. These nodes need to be removed in order for the backbone to be considered reliable. Once all of these nodes have been removed from the bipartite, the backbone has been determined. Then, the two backbones with the largest size are selected and their domination (ratio of nodes connected to the backbone) and number of faces (for the sphere topology) are calculated.

The largest independent sets are the largest color sets given by smallest-last vertex coloring. These will be the first four color sets when greedy coloring is used on a sequence of nodes sorted in smallest-last order. The combination of these four independent sets must be taken to find the six largest bipartite subgraphs.

The bipartite subgraphs need to be cleaned up in order to measure the size and coverage area of the backbone. This can be done by first removing all of the tails in the graph, which are sequences of nodes coming off of a component where the end node has degree one, and all nodes in between have degree two. Then, the major component needs to be determined, which is the component with the largest order. Once the largest component is determined, the minor blocks and the bridges connecting them to the major component need to be removed. A bridge is simiar to a tail; it is a chain of edges that, if removed from the graph would increase the number of connected components. These features need to be removed because they do not provide reliability to the wireless sensor network. If a single one of these node were to fail, a portion of the graph would become disconnected from the remaining backbone. This creates a single point of failure that should not occur in a network backbone.

Each of these algorithms can be implemented in linear time. Taking the combinations of the four largest independent color sets can be done by building a bipartite subgraph for each combination where the nodes are copied from the two color sets that make up the bipartite. Each bipartite will then be built in $\Theta(2|V|)$ time and $\Theta(2|V|)$ space where $|V|$ is the number of nodes in each color set. Since there are six ways to choose two items from a set of 4, this runs six times, resulting in $\Theta(12|V|)$ space and time usage for building all of the bipartites.

The tails then need to be removed. This can be done by repeatedly removing all nodes with a degree of one. This will repeatedly remove the last node in the tail until the only remaining node is the node that connected the tail to its component. This will also remove any minor components that consist of a thin chain of nodes with no cycles. This is similar to smallest-last vertex ordering, except the deletion of nodes from the graph stops when there are no more nodes in the bucket for degree one. Since this algorithm is based off of smallest-last vertex ordering, and slvo ran in $\Theta(|E| + |V|)$, this is bounded above by smallest-last vertex ordering, $O(|E| + |V|)$. However, since the bipartite could have no tails in it, the lower bound of the runtime is $\Omega(|V|)$ which is the amount of time needed to place nodes in their respective buckets based on how many edges they have in the bipartite. Regardless, this will require $\Theta(|V|)$ space to create a representation of the bipartite that can be deleted from.

Next, the major component needs to be determined. This can be done with breadth-first seach. BFS will traverse the entire graph, counting the number of nodes that can be reached from some start node. If an entire component has been explored from some start node, and there are still unvisited nodes in the graph, BFS will pick a new start node and begin searching from there. By counting the number of nodes connected to each start node, the size of each component can be determined. The major component can be determined by taking the max of these sizes. BFS works with a queue of nodes to search. At the start of an iteration, the current node is removed from the front of the queue, and all of its neighbors are added to the queue, if they have not already been visited. Since each node is only visited once, the runtime for BFS is $\Theta(|V| + |E|)$. BFS operates in-place on the graph, but a parallel array to the array of nodes is needed to remember if a node has been visited or not. This requires $\Theta(|V|)$ space and time to initialize. All together, this algorithm runs $Theta(2|V| + |E|)$ time.

Next, the bridges need to be removed from the major components. This can be done by modifying depth-first search to check for back-edges to nodes. If some node and its edges are being searched, it is a bridge if and only if none of the decendents of the nodes connected to the current node have a back-edge to the current node or any of its ancestors. Back-edges can be checked by maintaining a list of visit times given by the DFS algorithm (tin), and a list of the minimum entry time of any ancestor (fup). If the current node's neighbors have decendents with an earlier entry time, then they must have a back-edge to that node. If they have a back-edge with the current node, the minimum entry time of the ancestors would be the current time. If the miniumum entry time of the neghbor's ancestors is greater than the

7

current time, it must be a bridge. This is codified in the folowing formula [8]:

$$fup[v] = min \begin{cases} tin[v] \\ tin[p] \text{ for all } p \text{ for which } (v, p) \text{ is a back edge} \\ fup[to] \text{ for all } to \text{ for which } (v, to) \text{ is a tree edge} \end{cases} \tag{6}$$

Given this formula, the current edge $(v, to)$ is a bridge if and only if $fup[to] > tin[v]$ in the DFS tree. DFS runs in $\Theta(|V| + |E|)$ and the book-keeping data structures add a total space requirement of $\Theta(2|V|)$.

Once the bridges have been found, the graph needs to be simulated to have them removed, and the resulting connected components need to be searched again for the major component. BFS can be used again, where if an edge is encountered that is in the set of bridge edges, the neighbors to the current node are not pushed into the queue. Using BFS again has a time and space requirements $Theta(2|V| + |E|)$ time and $\Theta(|V|)$ space.

With the bridges removed, the major component in each graph has been determined and all single points of failure that could result in the disconnection of backbone nodes have been removed. It is then time to determine the two largest backbones for further evaluation. The size of the backbones (the number of edges) can be detemined in linear time by traversing all of the nodes in the backbone and counting the edges that are shared with other nodes in the backbone. This runs in-place on the backbone representation in $\Theta(|V| + |E|)$ time for each backbone that needs to have its size calculated.

The domination of the two largest backbones needs to be calculated. Finding the number of nodes connected directly to the backbone is equivalent to finding the number of nodes that are not connected to the backbone. This can be done by traversing all nodes that are not part of the backbone and, for each of their edges, seeing if the adjacent node is a backbone node. This algorithm requires $\Theta(|V|)$ space and $\Theta(|V| + |E|)$ time to run where $|V|$ is the number of nodes not in the backbone.

Finally, if the topology is a sphere, the number of faces can be determined by using Euler's Polyhedral Formula [7], which is given by:

$$2 = V + F - E \tag{7}$$
$$F = 2 - V + E \tag{8}$$

Where $V$ is the number of verticies, $E$ is the number of edges, and $F$ is the number of faces.

## 2.3   Algorithm Engineering

### 2.3.1   Node Placement

It is easy to implement the algorithms for placing nodes in the different geometries using Python's math library. This library offers functions for sampling points on a uniform distribution. For the Square, sampling on a range $[0, 1]$ is sufficient for all of the nodes. Since each node only needs to be placed once, this runs at $\Theta(|V|)$ where.

For the Disk, the node needs to be resampled if it is too far from the center. To do this, the distance function is used to find the distance between the node and the center. If the node is further than 1 from the center, node generation falls into a while loop which iterates until the node is within the unit circle. Since nodes are taken from a uniform distribution, the number of nodes that will need to be resampled is approximately equal to the ratio of the area of the square that circumscribes the unit circle which falls outside of the unit circle to the total area of the square. This is given by:

$$\frac{(2r)^2 - \pi r^2}{(2r)^2} = \frac{4 - \pi}{4} = 0.2146 \tag{9}$$

Since the placement algorithm for each node of the Disk will iterate until the node falls within the unit circle, the total number of iterations $N$ can be found as the sum of the geometric series:

$$N = \sum_{k=0}^{\infty} n(0.2146)^k = \frac{n}{1 - 0.2146} = 1.273n \tag{10}$$

where $n = |V|$. This shows this implementation is $\Theta(n)$.

For the node placement algorithm of the Sphere, again the math library in Python makes this easy. Each node needs two random values pulled from a uniform distribution, two square root operations, one sine operation, and one cosine operation. Each node only needs to be placed once so the runtime of this algorithm is $\Theta(n)$ where $n = |V|$.

### 2.3.2 Edge Determination

Each method implemented for finding edges has a different time complexity. The brute force method uses an outer loop and an inner loop, which each iterate over every node in the graph. An edge is saved to the adjacency list if the nodes are not the same and the distance between them is less than or equal to the calculated node radius. This is guaranteed to run in $\Theta\left(n^2\right)$ where $n = |V|$. The number of times the distance needs to be calculated is $n \times (n-1)$ because it will not be calculated when the nodes are the same (distance would be zero, but no edge is drawn here). No additional space is needed for the brute force method so the space complexity is $O(1)$.

The implementation of sweep starts by sorting the nodes along the x-axis. Python lists have a built-in sort function that has $O\left(nlg(n)\right)$ time complexity [9]. After this stage, it iterates over every node building a search space which will be scaned for edges. For each node, the list of nodes is searched right $r \times n$ nodes to find those within one radius length of the current node. With the search space built, the search space is iterated over once to find nodes that have a distance less than or equal the node radius. Then, the indicies of the nodes are added to the adjacency list entry for each other. My implementation of this runs in $O\left(nlg(n) + 2rn\right)$ where $n = |V|$ and $r$ is the node connection radius. Because the list sort method sorts inplace, the only additional space needed is for the search space. This saves $O(rn)$ nodes and is reset after every iteration.

The cell method implementation works in linear time. In the first step of the method, the cells are initialized as a list of empty lists. There are $(1/r + 1)^2$ cells. The nodes are then iterated over and assigned a cell by dividing their x and y coordinates by the node radius. At this point, the cells are iterated over and, for each node in the cell, the nodes in the current cell and the four forward adjacent cells and the are checked to see if they fall within the node radius of the current node. All together, this implementation runs at $O\left(n + n + 5nr^2\right) = O\left((2 + 5r^2)n\right)$ where $n = |V|$. The amount of additional space needed is equal to the number of nodes because they are coppied into their respective cells. This places the space complexity at $\Theta(n)$.

### 2.3.3 Graph Coloring

Implementing the smallest-last coloring algorithm involves implementing the smallest-last vertex ordering algorithm and the greedy graph coloring algorithm. For smallest-last vertex ordering, the first thing to do is build the data structure used to represent the graph with deleted nodes. This can be done with a list of sets, where each the index in the list represents the degree of the nodes in that set. The number of sets needed is equal to the maximum degree of the nodes. The index of each node is placed in the set corresponding to the number of edges it has then the RGG. Simultaneously, a dictionary is created that maps each node to the number of degrees it has in the graph with deletions. Each value starts at the number of edges the corresponding node has in the RGG. At this point, we have iterated over all of the nodes once and allocated space for twice the number of nodes by copying them into the sets and using them as the keys for the degrees dictionary.

Because Python dictionaries resize at specific numbers of entries, we can determine the number of additional insertions caused by rehashing while the degrees dictionary is built. Python dictionaries start out with space for 8 entries and quadruple in size until the number of entries is above 50,000, at which point it begins to double in size. Clearly the dictionary grows at a logarithmic rate, but the total number of insertions $I$ for an input size of $n$ is given by:

$$I = \begin{cases} n + 8\sum_{k=1}^{\log_4 \lceil n/8 \rceil} 4^k & n \leq 50,000 \\ n + 8\sum_{k=1}^{6} 4^k + 32768\sum_{k=1}^{\log_2 \lceil n/32768 \rceil} 2^k & n > 50,000 \end{cases} \tag{11}$$

Fortunately, because the entire dictionary is built before it is used by the smallest-last vertex ordering algorithm, it will never again be resized once the algorithm starts. Unfortunately, the sets resize at a similar rate and it is more difficult to predict how large the sets will need to be when performing smallest-last vertex ordering. The degree dictionary will also be used to index into the sets, so we gain a speed up here by not having to iterate over all of the edges for a node and determining if the node it shares an edge with are in the remaining graph each time we want to sift nodes down to lower set.

After setting up the graph representation, the smallest-last vertex ordering algorithm runs until every node has been removed from the representation. To delete a node, the first non-empty set is selected. This set must contain the next node to remove becuase it contains all nodes with smallest degree. Before deleting the node from the graph, and moving all adjacent nodes down a set, the current set is checked to see if it has all remaining nodes. If this is the case, the terminal clique has been found, and the size of the terminal clique must be saved. After this check, a node is popped from the end of the current set, and appended to the smallest-last ordering result. Then, all nodes adjacent to the popped node in the original graph are checked to see if they are in the set with its current degree. If it is, the number of degrees for that node can be decremented and the node can be placed into the correct set for its new degree.

The last step is to reverse the order of the smallest-last ordering result because it was built in the opposite order (smallest-first). All together, excluding the initialization of accessory data structures, this implementation runs in $\Theta(2|V|+2|E|)$ time and $\Theta(2|V|)$ space since nodes are removed from the buckets and added to the result.

After this the graph needs to be colored. For this, initially each node is assigned a color of $-1$ in a node color array that is parallel to the original list of nodes. Then, all of the nodes in the smallest-last vertex ordering are iterated over. At each node, a set of colors that is already used by the neighbors of that node is created by iterating over all of its edge nodes and grabbing their color from the node color array. Then, color just has to be incremented from 0 until it does not exist in the search space set and the color has been determined to assign to the node.

Since the smallest-last odering is used, each time the edges need to be traversed to see if a node is adjacent to the current node, nodes with fewer and fewer edges are being searched. This means that the nodes with the most neighbors are searched first, when the number of other nodes to check is lowest, and the nodes with the fewest neighbors are searched last, when we have the most nodes to check if they share an edge with the current node. All together, this implementation runs in $\Theta(|V| + 2|E|)$ time and $\Theta(|V|)$ space because we need a new array for the colors assigned to each of the nodes.

A setp-by-step walkthough of the smallest-last coloring algorithm is provided to further visualize this algorithm. For this walkthrough, a unit square topology is used with 20 nodes and a node connection radius of 0.4. The smallest-last vertex ordering deletion process is shown in Figure 1. The coloring phase is shown in Figure 2. In the deletion process, the minimum degree node is removed at each step. If there are multiple nodes with the same minimum degree, one is choosen randomly. Once all nodes have been removed, the smallest-last vertex ordering has been detemrined. In the coloring phase, the node that was removed last is assigned a color first. As the smallest-last vertex ordering is traversed, each node's neighbors are checked to see if they have been assigned a color. The first color that has not been used by a neighbor is assigned to the node. To complete this walkthrough, the distribution of the color set sizes and the degrees of nodes when deleted is given in Figure 3.

### 2.3.4    Backbone Determination

Implementing backbone determination requires implementing all of the algorithms needed to create the bipartite subgraphs, remove unwanted nodes, and find the major components. Pairing the independent color sets is the most straightfoward algorithm to implement. First, a list of four sets is created to hold the four largest independent color sets. Since the largest color sets will be the first four colors used in the greedy graph coloring implementation, all of the nodes are iterated over and each one is checked to see if its color is less than four. If that is the case, it is added to the independent set at that index in the initial list. Then, the list of independent sets is iterated over and each set is unioned with each remaining set in the list to get all of the combinations of the independent sets. The Python set union operation iterates over all of the items in each set and adds them to a result set. Since this is called three times on each independent set, and because the nodes needed to be iterated over once to place the nodes in their color sets, the total runtime for this implementation is $O(4|V|)$. The total space used by this algorithm is $O(4|V|)$, because four copies are made of each independent set. However, one of each of these copies is removed when the function returns the combinations.

Next, the idependent color set pairings need to be cleaned. This is a multi-step process that starts with the removal of tails from the bipartites. Like stated earlier, the algorithm to remove tails is similar to the smallest-last vertex ordering algorithm with an early stopping condition for when the bucket for degree 1 is empty. First, some accessory data structures are initialized to save information about the representation of the graph while nodes are deleted. The buckets are initialized as empty sets. The total number of buckets needed is equal to the degree of the node with the max degree. A map is needed to relate each node to its bucket, which is created by iterating over all of the nodes in the bipartite, and
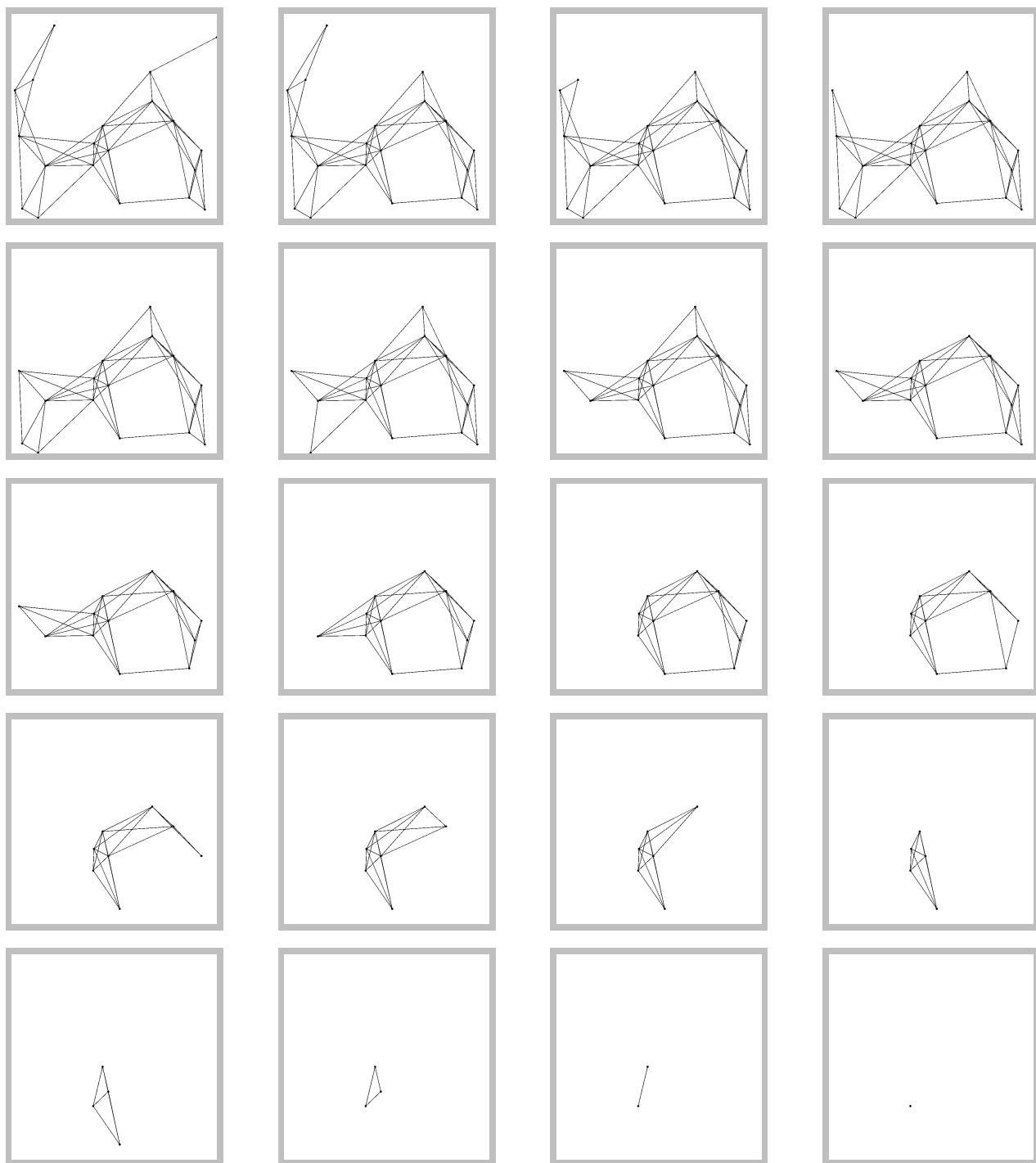
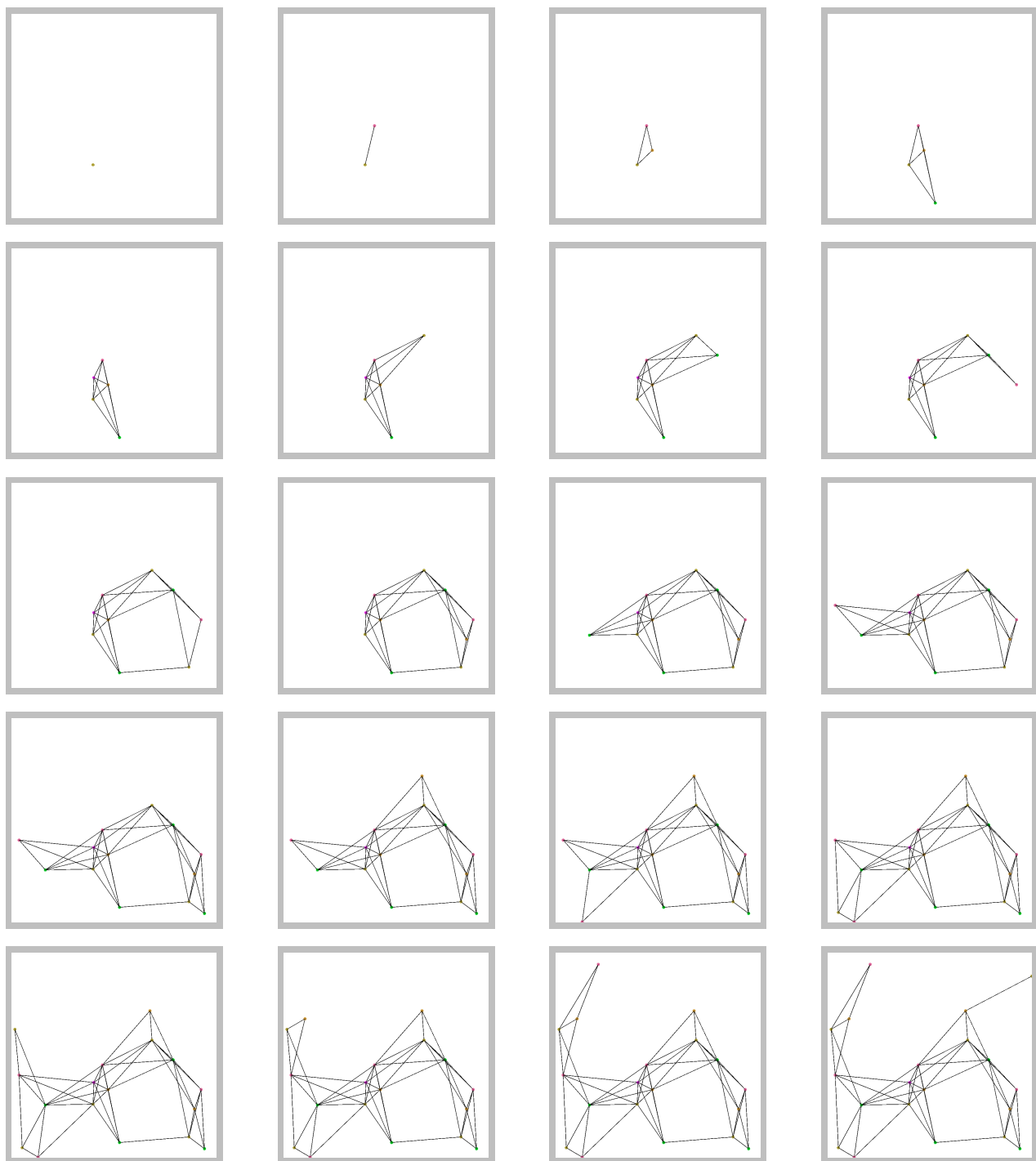Figure 1: Smallest-last vertex ordering deletion process

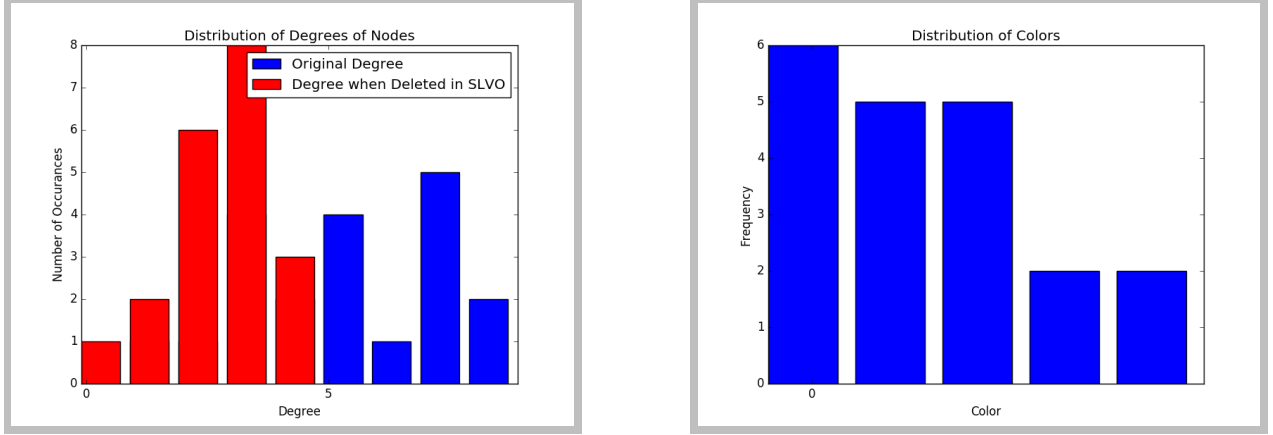Figure 2: Smallest-last vertex ordering coloring process

Figure 3: Distribution of degree when deleted and color set size for the 20 node walkthrough

counting the number of edges it shares with other nodes in the bipartite. Then, the nodes are iterated over again and placed in their buckets. At this point, the total space used is $\Theta(2|V|)$ and the time used is $\Theta(2|V| + 2|E|)$.

At this point, the smallest-last vertex algorithm is run until the sets for degree zero and one are empty. Each iteration of the algorithm, all of the nodes in the degree zero and degree one sets are put in a list of nodes to remove. Both sets are checked so that any nodes in the graph that are not connected to a component are removed. These nodes are then iterated over and each edge it shares with a node in the bipartite is checked to see if the neighbor needs to be moved down a bucket. Once all neighbors have been moved down, the node is removed from the bipartite subgraph. This runs in similar time as smallest-last vertex ordering, $\Theta(2|V| + 2|E|)$. The only additional space needed by the algorithm is the space needed to hold the list of nodes in the first two buckets, however, once the nodes have been copied into the list, the buckets they were in are cleared. Regardless, thin can use $O(|V|)$ in the worst case. All together, tail removal takes $O(3|V|)$ space and $\Theta(4|V| + 4|E|)$ time.

The next part of the cleaning is selecting the major component, which is implemented using breadth-first search. Before starting BFS, some setup is needed. First, the bipartite is copied into a local list for iteration. Then, two dictionaries are created for indexing from the local list of bipartite nodes to the master list of nodes. Next, a list of integers is created for keeping track of which nodes have been visited during BFS. At this point, $O(4|V|)$ space has been used. Then, BFS starts and runs until every node has been visited. While nodes have not been visited, the first unvisited node is selected to be the root of the search tree. This root is put in the queue, added as the first item in a set to a list of sets representing the components in the graph, and the visit time is set to 1. Then, while the queue is not empty, an item is popped and all of its edges are checked to see if they have already been visited. Each one that has not beev visited is pushed into the queue, market as visited, added to the set representing the current component being searched, and the visit time is incremented. Once the queue is empty, the final visit time is saved as the number of nodes in the component. After all nodes have been visited, all that is needed is to return the component with the largest number of visits and the major component has been determined. This implementation of BFS requires $\Theta(|V| + 2|E|)$ time and $O(2|V|)$ space because the nodes are copied into their respective component sets, and the queue could grow to hold all nodes in the graph in the worst case.

The last step in preparing the backbones is to remove all of the bridges and minor blocks. Bridge removal uses depth-first search, however, some other data is needed to keep track of the visit time for nodes (tin) and the visit time of their ancestors (fup) in the DFS tree. First, a local copy of the bipartite is created to iterate over, and, similar to BFS, two dictionaries are created for indexing between the local list of nodes and the master list of nodes. A list is created to keep track of whether nodes have been visited or not, the visit time of the DFS algorithm at the node, and the minimum visit time of a nodes decendents. All of these data structures together require $\Theta(6|V|)$ space and can be created in $\Theta(6|V|)$ time. Now, DFS can run until all of the nodes have been visited. The first node that hasn't been visited is selected as the root of the search tree for DFS. Each edge this node shares with another node in the major component is iterated over. Fup for the current node is calculated for each of the neightbor nodes that has not been visited as the mimimun of fup for the current node and tin of the current edge. If the

13

neighbor hasn't been visited, DFS is called recursively on the edge to search it. Once the search returns, fup for the current node is calculated as the minimum of fup for the current node and fup for the current edge. There is now enough information to determine if the current edge is a bridge. If fup for the current edge is greater than tin for the current node, then the neighbor must not have another path to any of the ancestors of the current node, so it is a bridge and the current nodes are saved to a list of bridges. DFS itself runs in $\Theta(|V| + 2|E|)$ time and uses $O(2|E|)$ space in the worst case which would be that all nodes in the graph are part of a bridge (however, this would never happen because tails have already been removed).

The final step of bridge removal is to use the list of nodes that are part of the bridges to determine the major component with the bridges removed. BFS is suitable for this because it is already implemented to return the major component of a graph. In order to make BFS skip the bridge nodes, each time an edge is visited that has both nodes in the set of bridge nodes, continue is called to skip the rest of the iteration. This will prevent pushing that neighbor to the queue and will disconnect those components. BFS will then proceed and return the major component. All together, bridge removal uses $O(8|V|+2|E|)$ space and runs in $\Theta(8|V| + 4|E|)$ time.

At this point, six potential backbones have been determined from the original six bipartite subgraphs. Now, the two largest backbones need to be determined. These are the backbones with the largest size, or the highest number of edges. To find the two largest backbones, two parallel lists are created that each have two elements. The first list is for the sizes of the backbones, and the second is for the backbones themselves. For each backbone, the size is calcualted by iterating over all the nodes in the backbone and summing the number of edges each node shares with another node in the backbone. Because the backbones are represented as a set, it takes constant time to see if a node is in the backbone. Once the size has been calculated for a backbone, it is checked to see if it is larger than the saved backbone with the minimum size. If this is the case, it repaces that backbone in the list of results and its size is saved. This requires $\Theta(|V| + 2|E|)$ time for each backbone. After the two largest backbones have been determined, some metadata is calculated about them and returned as a parallel array to the list of backbones. This meta data is the order and size of each backbone, which is not dependent on the size of the backbones.

Finally, the domination of the two largest backbones needs to be calculated. This is done by initializing a search space with all of the nodes in the master list of nodes that are not in the backbone. This search space is then iterated over, and each edge is checked to see if the neighboring node is in the backbone. If a node does share an edge with the backbone, it is removed from the search space. Also, once it has been found that the current node shares an edge with a backbone node, the rest of the edges for the current node can be skipped. At the end of this, the search space will have all nodes that do not share an edge with a backbone node. It is then easy to calculate the domination of the backbone by subtracting this number from the total number of nodes and dividing by the total number of nodes. This runs in $\Theta(|V| + |E|)$ time and requires $\Theta(|V|)$ space to initialize the search space.

If the topology is a sphere, the number of faces of the backbone can be calculated using Euler's Polyhedral Formula. This formula operates under the assumtion that a graph is connected and can be represented in planar form. The first is guarunteed because the backbone is the major connected component found in a bipartite subgraph. The second is true because the nodes comprising the backbone can be projected onto a plane and there will be no overlapping edges because the edges do not overlap in the original representation. Therefore, the number of faces can be calculated in constant time using the meta data of the backbone generated earlier.

To illustrate this further, the above walkthrough is extended to include the backbone determination stages based on the two largest color sets. These stages are given in Figure 4. With the selected color sets, removing the tails is sufficient for creating the backbone. The other steps yield the same graph, but are necessary for higher-order graphs.

## 2.4 Verification

### 2.4.1 Node Placement

The nodes can be verified to be distributed uniformly if the degrees follow a normal distribution. To show that the distribution of degrees for each of the geometries are following a normal distribution, the degree histograms are plotted for each of the benchmarks. The histrograms for Square are given in Figure 6, Disk are given in Figure 7, and Sphere are given in Figure 8. These histograms clearly follow a normal distribution, so the nodes must be placed uniformly.
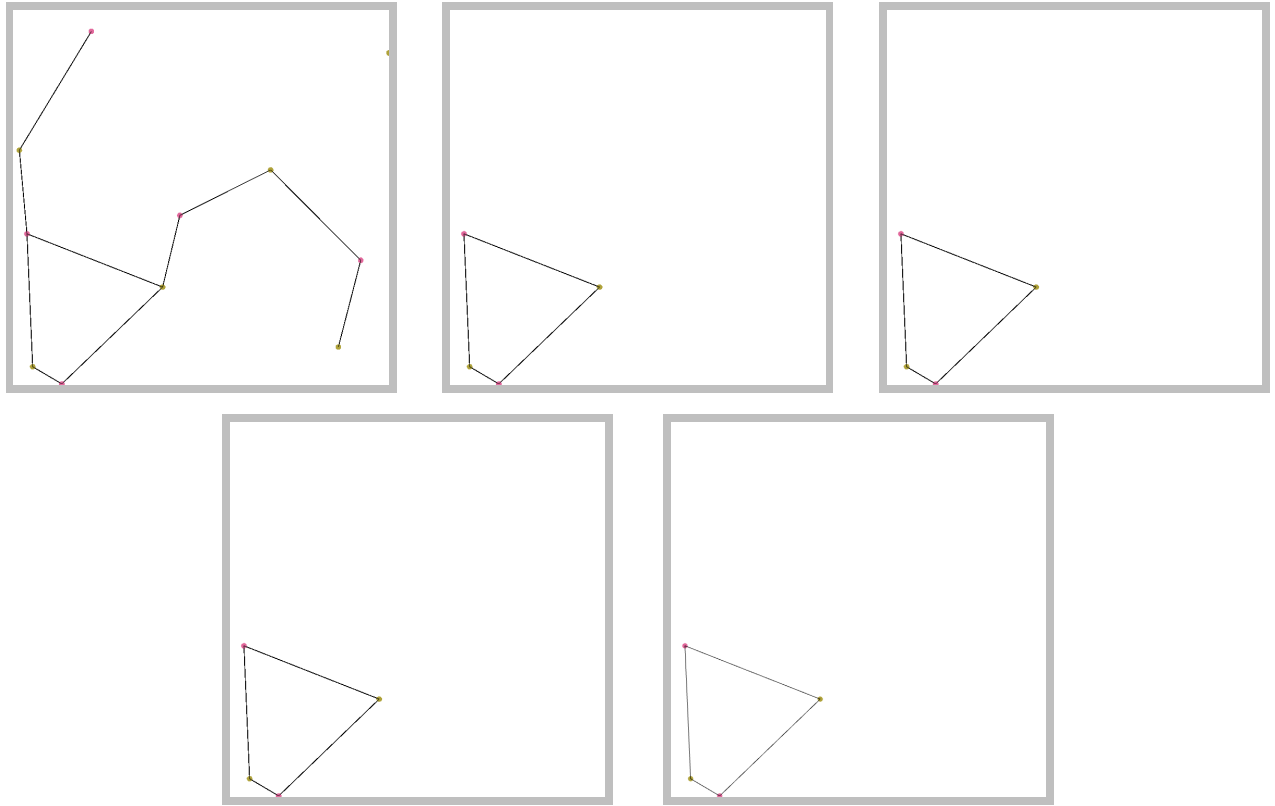
Figure 4: Smallest-last vertex ordering coloring process

### 2.4.2 Edge Determination

The runtime for the edge detection methods can be verified by varying the number of nodes and measuring the runtime of each algorithm. By looking at how the runtime grows, we can calculate the trendline that best fits the growth rate. For the first comparison, the number of nodes is varied from 4,000 to 64,000 in steps of 4,000, while holding the desired average degree constant at 16. As we can see in Figure 5, the growth rates of the brute force and sweep methods are quadratic, while the growth rate of the cell method. The trendline functions are given on the graph.

For the second metric, the number of nodes is held constant at 32,000 and varied the desired average degree from 2 to 32 in steps of 2. The graph is given in Figure 5. The cell method clearly grows linearly, but the sweep method is harder to gauge. Since varying the desired average degree should only change the node radius, this should grow linearly as well. However, because each graph is randomly generated, some graphs can have nodes that are closer to sorted order than others. This can effect the measured runtime.

### 2.4.3 Graph Coloring

Smallest-last vertex ordering can be verified by looking at the distribution of the degrees of nodes when deleted. Since this algorithm repeatedly removes the node with the fewest connections, and because the removal of that node will cause the fewest number of nodes to move to the next lowest bucket, we would expect the bulk of the nodes to have a large degree when they are deleted. This would be indicated by a negative skew in the distribution of degrees when deleted. Additionally, since the nodes are only removed when they satisfy the criteria of being the node with the minimum degree, we should see the standard deviation of the distribution of nodes to be much smaller than in the original distribution of degrees. Both of these features can be found in Figures 9, 10, and 11 which plot the original distribution of degrees alongside the distribution of degrees when deleted. We see that the distribution of degrees when deleted follows a normal distrbution with a negative skew and a relatively small standard deviation compared to the original distribution of degrees.

The color sets can be verified by looking at the distribution of colors used to color the graph. The number of items in each color should follow a trend where the first colors used have the most members, and the last colors have the fewest items because they are used to accommodate nodes where the earlier colors are all used by a node's neighbors. This trend is shown in Figures 12, 13, and 14.

To further verify the accuracy of the smallest-last coloring implementation additional code was used to verify that the coloring result was correct while running benchmarks. All of the nodes in the smallest-last vertex ordering are traversed, and for each node, the edges are visited to see if any adjacent nodes have the same color as the node being checked. If any of these neighbors have the same color, the coloring is not correct and our independent sets cannot be used for backbone determination. All of the benchmarks ran and returned valid colorings.

### 2.4.4 Backbone Determination

# References

[1] Matula, David; Beck, Leland, Smallest-Last Ordering and Clustering and Graph Coloring Algorithms, 1983

[2] Johnson, Ian, Linear-Time Computation of High-Converage Backbones for Wireless Sensor Networks, https://github.com/ianjjohnson/SensorNetwork/blob/master/Report/Report.pdf, 2016

[3] Fry, Ben; Reas Casey, Processing, https://processing.org, 2018 v3.3.7

[4] The Matplotlib Development, matplotlib, https://matplotlib.org, 2018

[5] Weisstein, Eric W., Wolfram MathWorld Sphere Point Picking, http://mathworld.wolfram.com/SpherePointPicking.html

[6] Weisstein, Eric W., Wolfram MathWorld Spherical Cap, http://mathworld.wolfram.com/SphericalCap.html

[7] Weisstein, Eric W., Wolfram MathWorld Polyhedral Formula, http://mathworld.wolfram.com/PolyhedralFormula.html

[8] Kogler, Jakob, Finding bridges in a graph in $O(N + M)$, https://e-maxx-eng.appspot.com/graph/bridge-searching.html, 2018

[9] Peters, Tim, Timsort, http://svn.python.org/projects/python/trunk/Objects/listsort.txt

[10] Rees, Gareth, Python's underlying hash data structure for dictionaries, https://stackoverflow.com/questions/4279358/pythons-underlying-hash-data-structure-for-dictionaries, 2010

[11] Thomas, Alec, Why is tuple faster than list?, https://stackoverflow.com/questions/3340539/why-is-tuple-faster-than-list, 2010

[12] Kruse, Lars, Python Speed, Performance Tips, https://wiki.python.org/moin/PythonSpeed/PerformanceTips, 2016

# 3    Appendix A - Figures
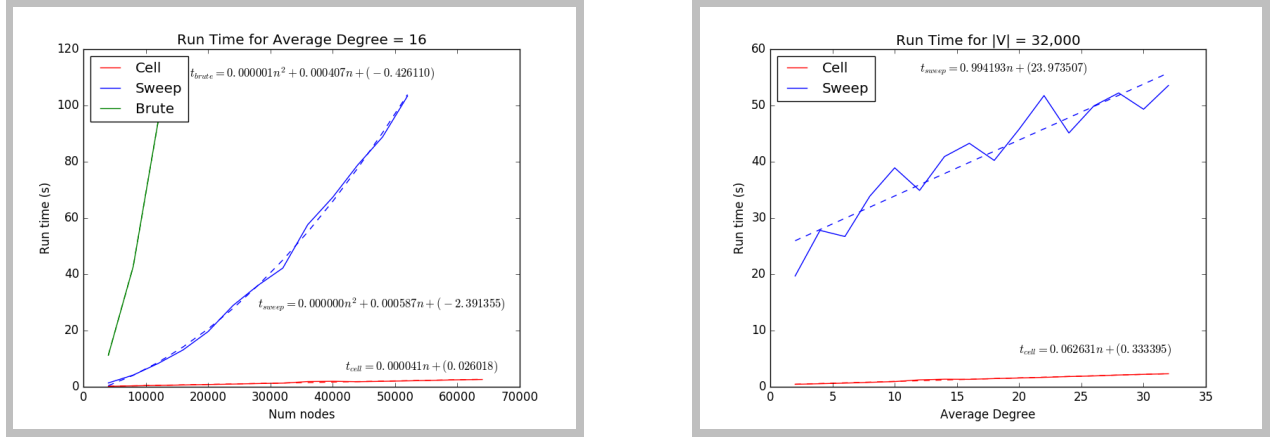


Figure 5: Runtime for edge detection methods. left: constant average degree of 16, right: variable average degree
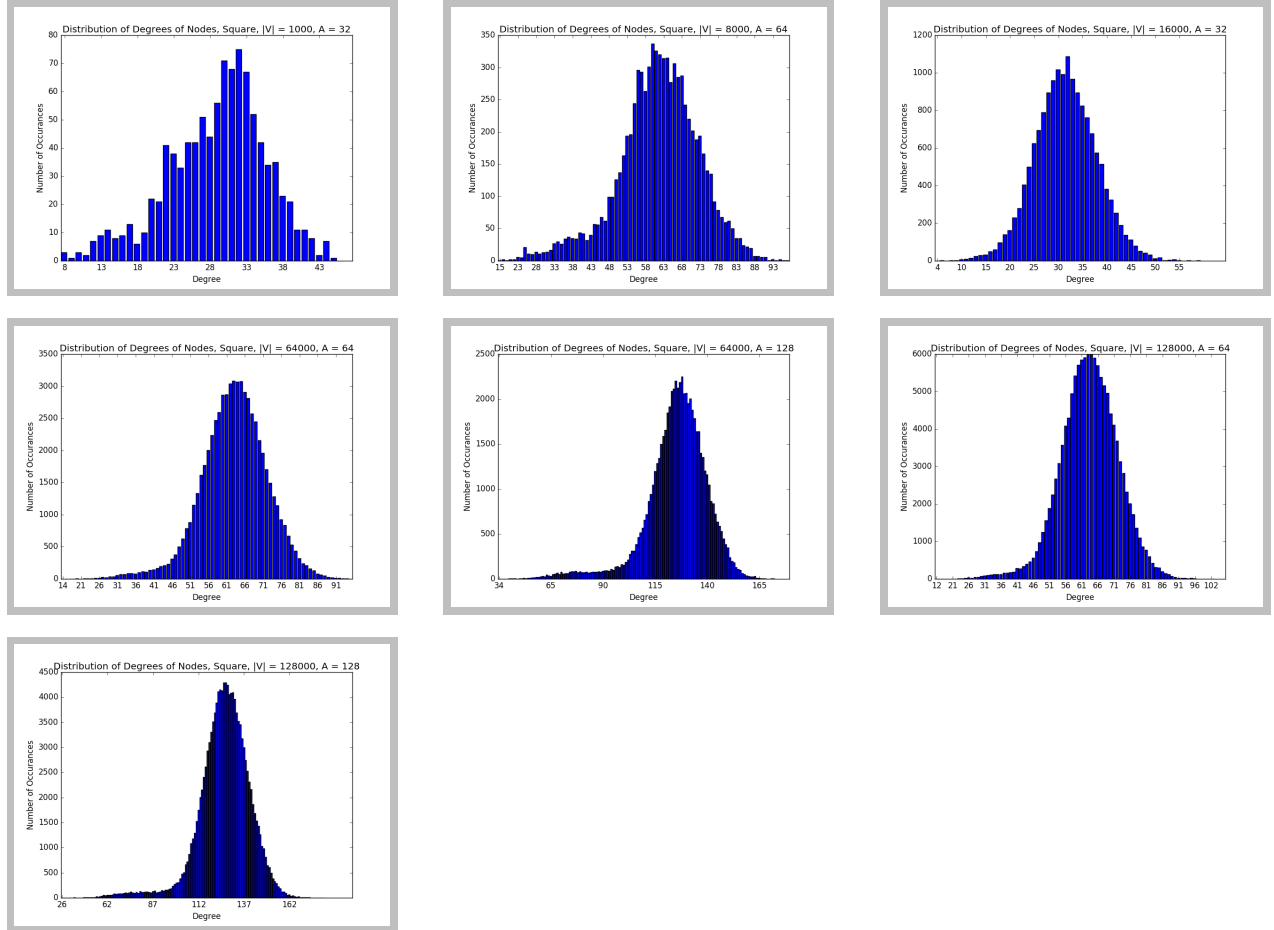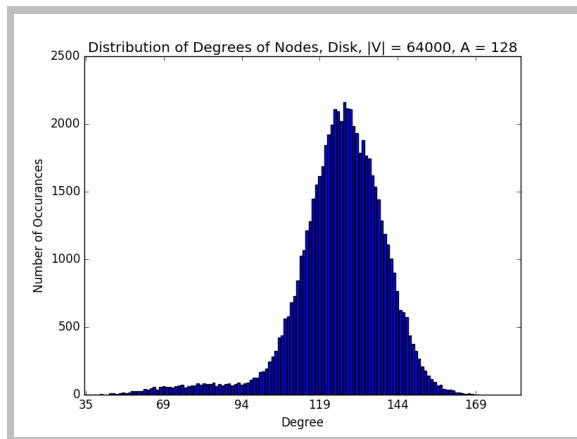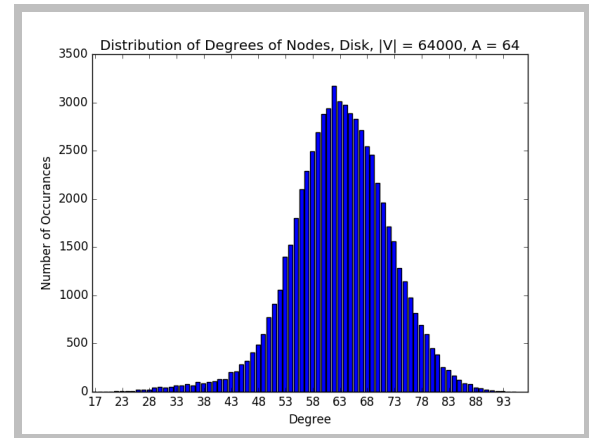
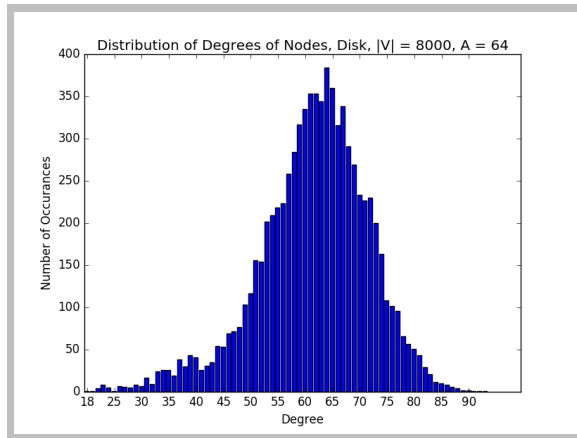Figure 6: Square benchmarks distribution of degree graphs

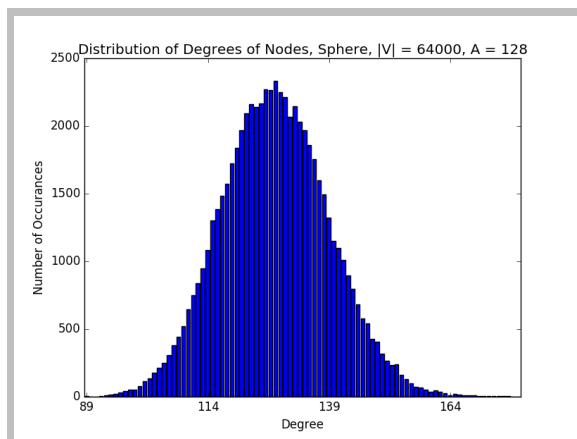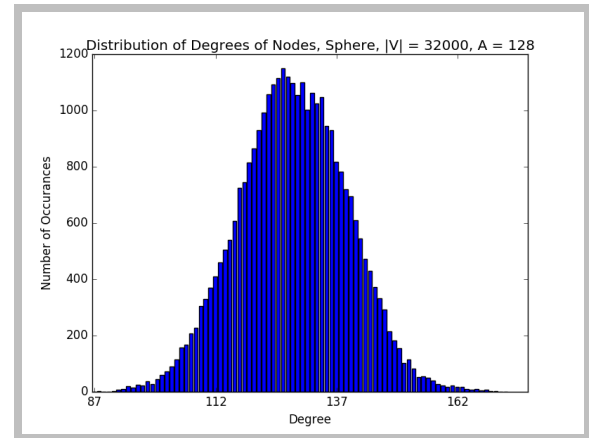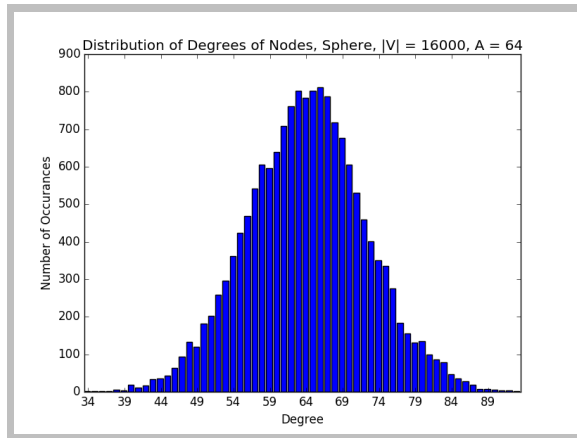Figure 7: Disk benchmarks distribution of degree graphs

Figure 8: Sphere benchmarks distribution of degree graphs
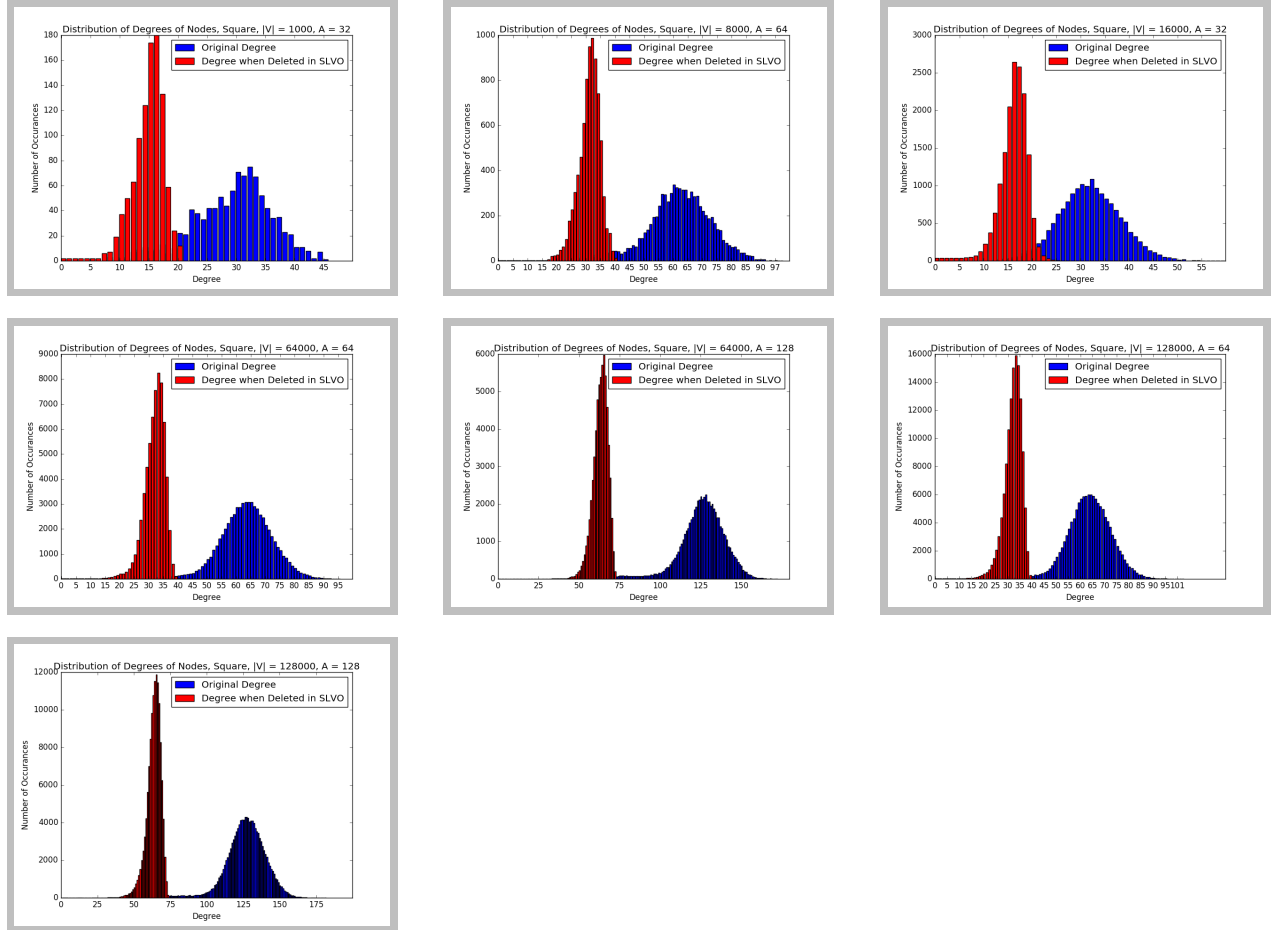
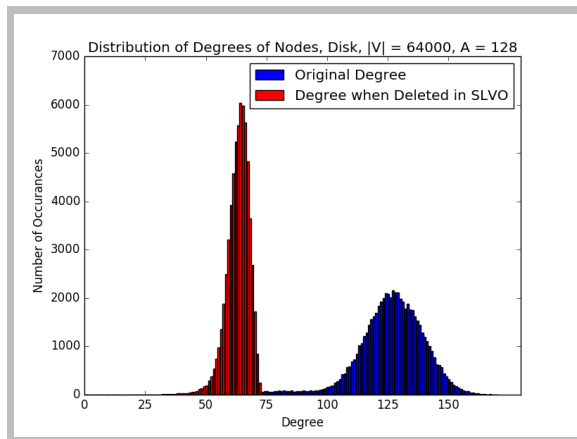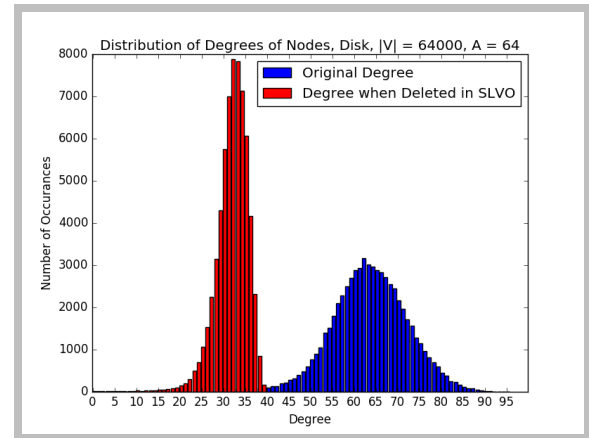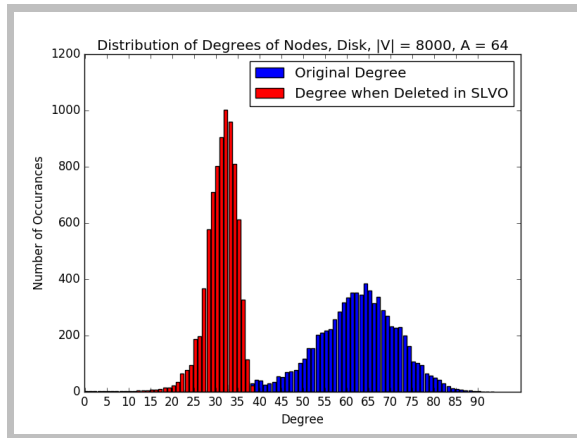Figure 9: Square benchmarks distribution of degree when deleted graphs

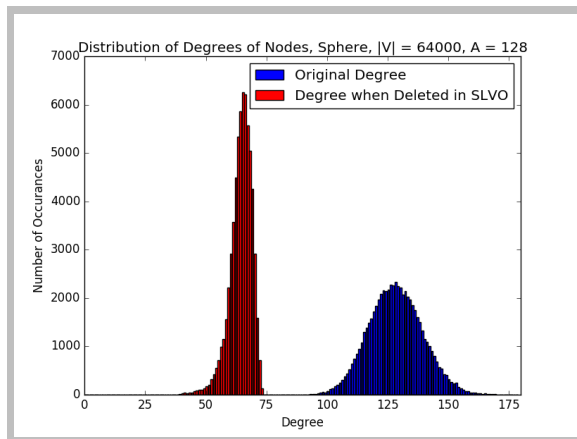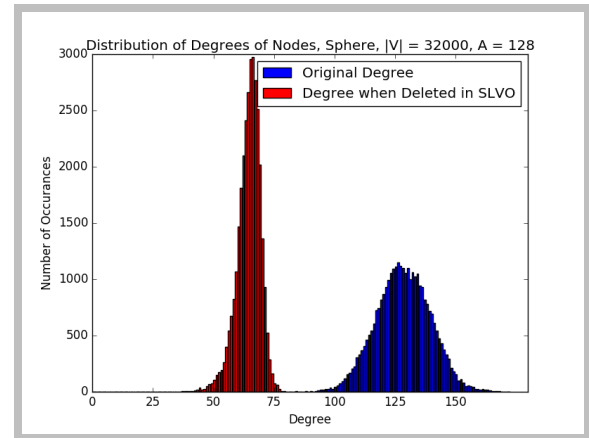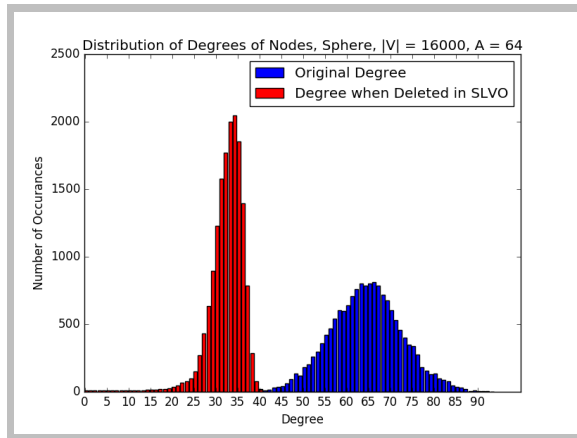Figure 10: Disk benchmarks distribution of degree when deleted graphs

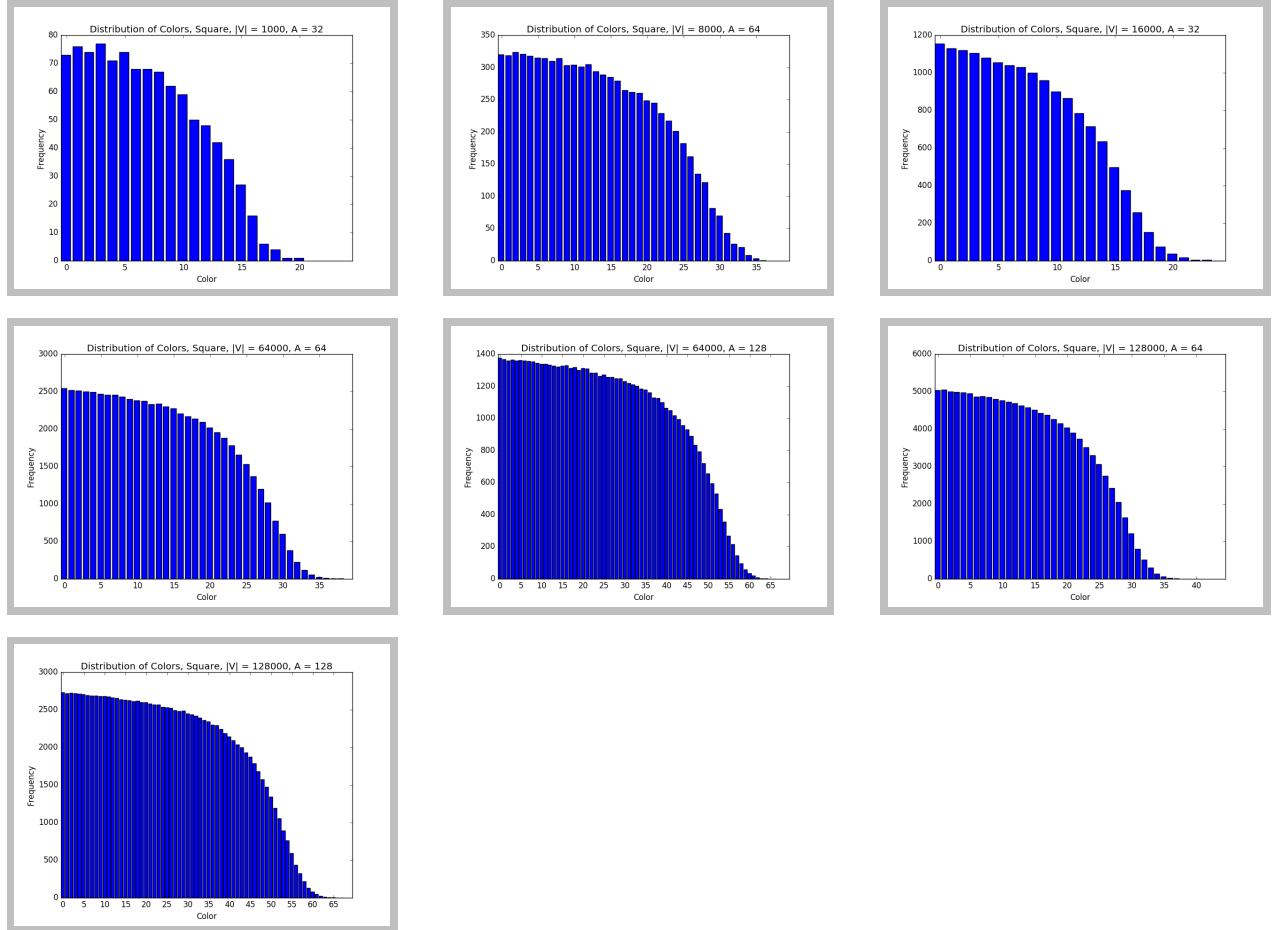Figure 11: Sphere benchmarks distribution of degree when deleted graphs

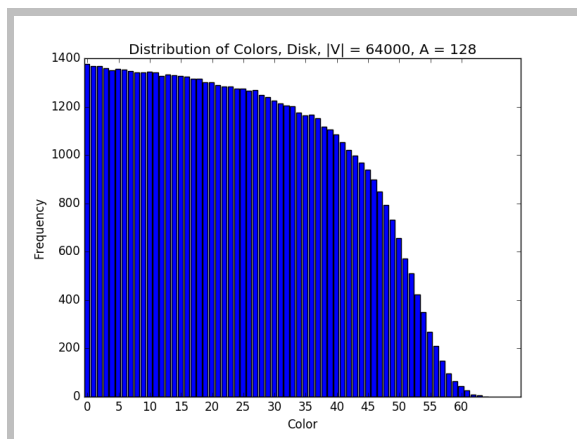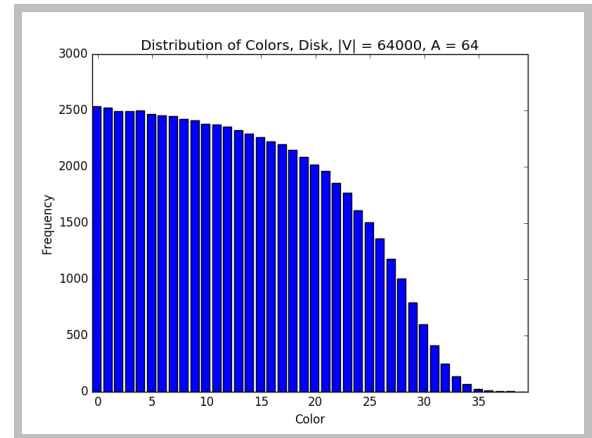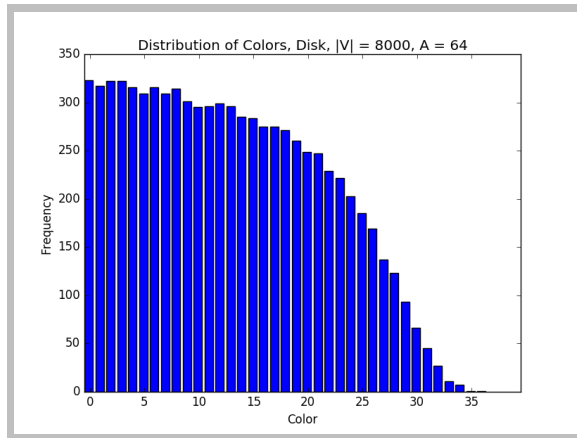Figure 12: Square benchmarks distribution of colors graphs

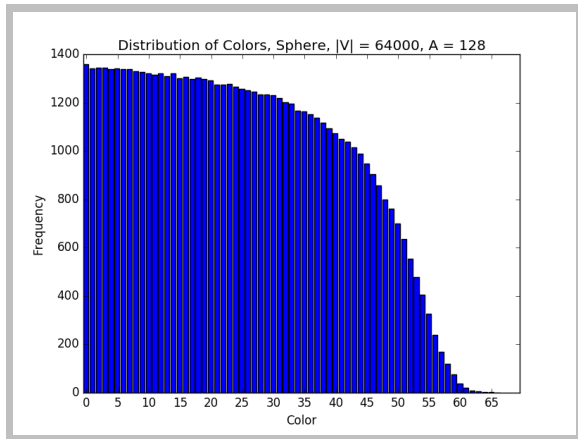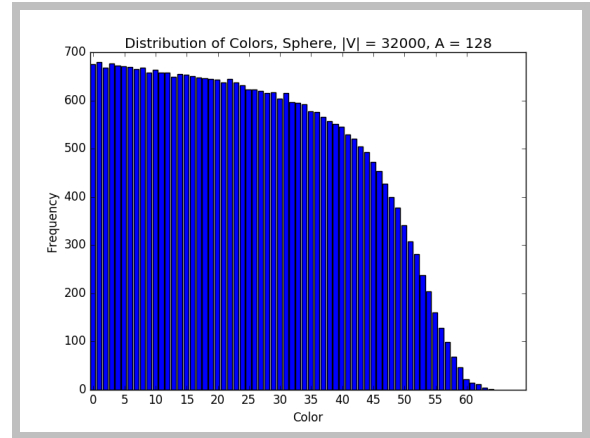Figure 13: Disk benchmarks distribution of colors graphs
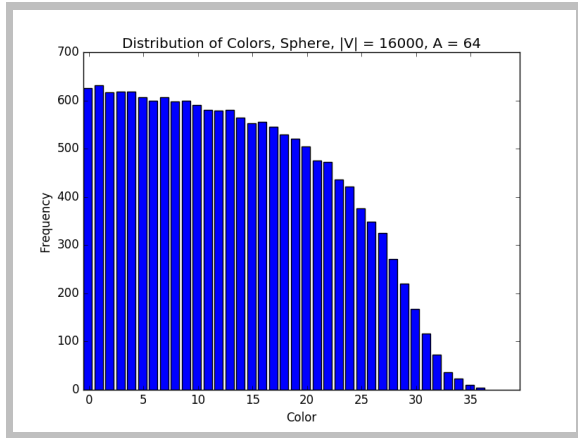
Figure 14: Sphere benchmarks distribution of colors graphs
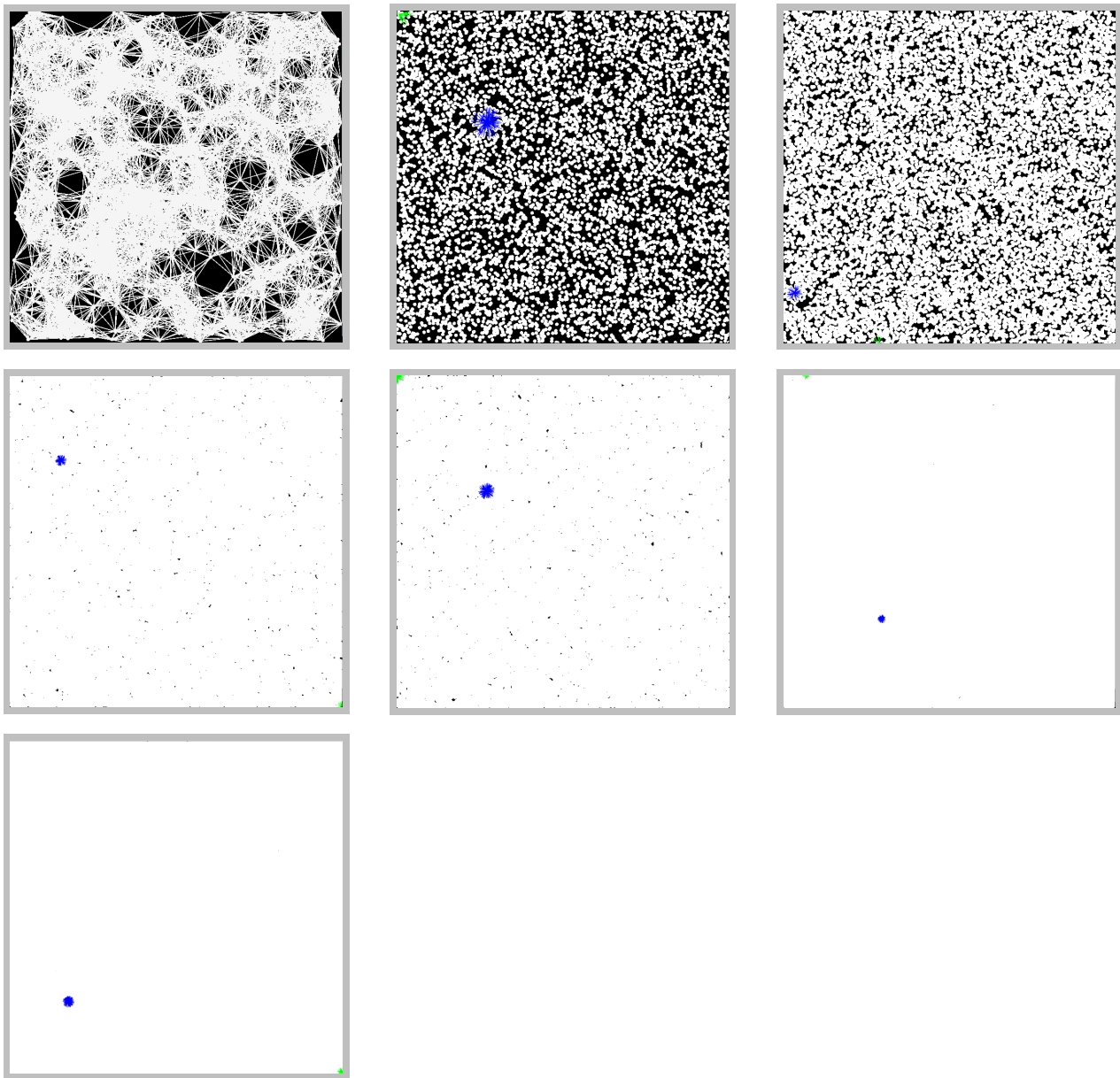
Figure 15: Square benchmark graphs
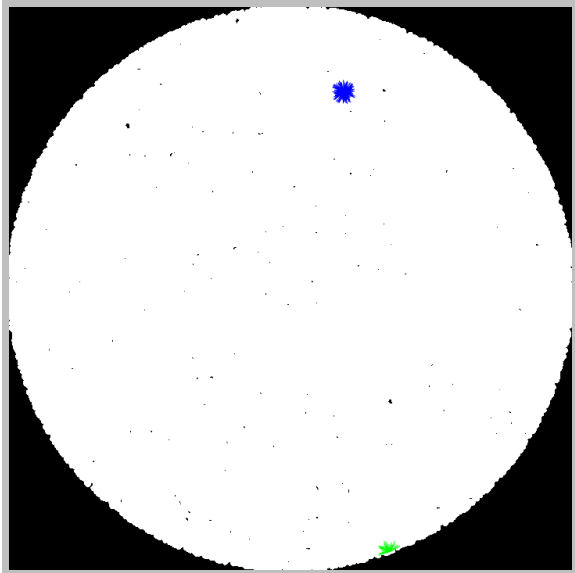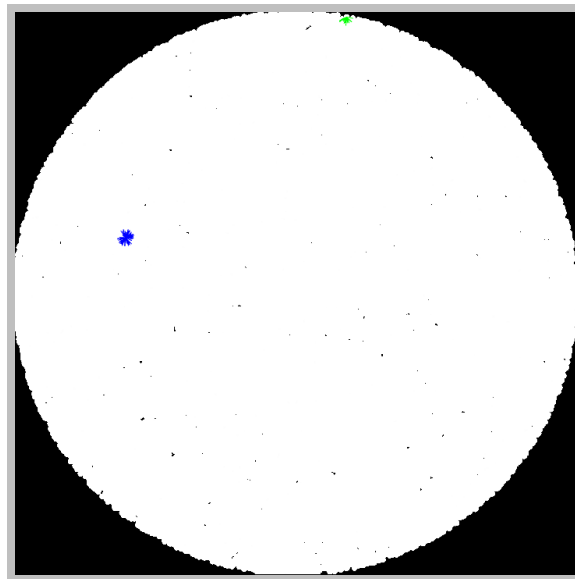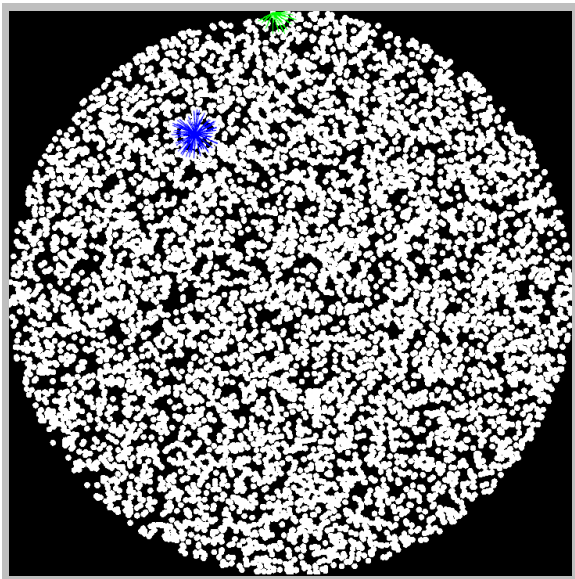
Figure 16: Disk benchmark graphs
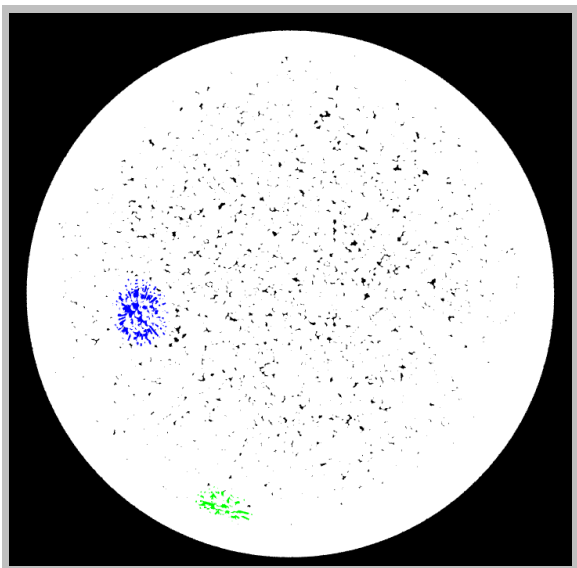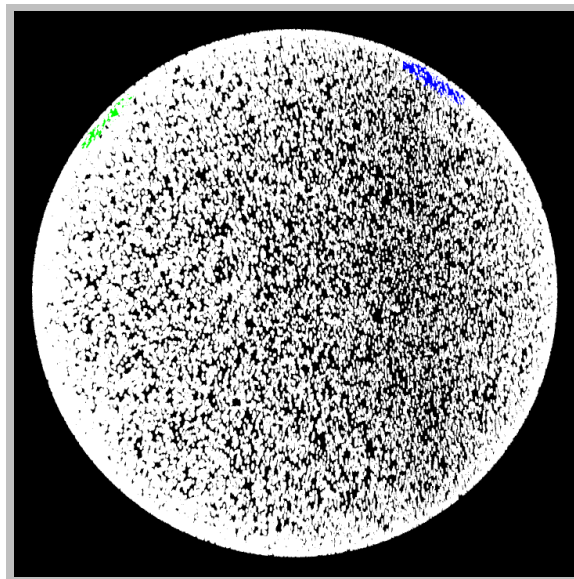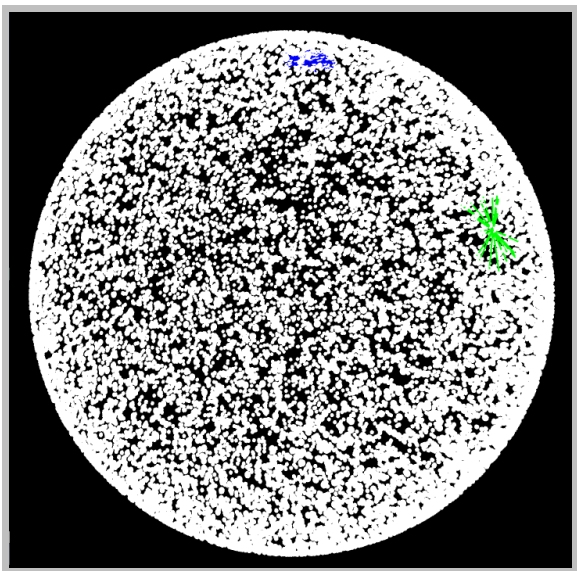
Figure 17: Sphere benchmark graphs

# 4 Appendix B - Code Listings

Listing 1: Processing driver

```python
import random
import sys
import time
import math
from collections import Counter
from objects.topology import Square, Disk, Sphere

CANVAS_HEIGHT = 720
CANVAS_WIDTH = 720

NUM_NODES = 20
AVG_DEG = 10

MAX_NODES_TO_DRAW_EDGES = 8000

RUN_BENCHMARK = False

def setup():
    size(CANVAS_WIDTH, CANVAS_HEIGHT, P3D)
    background(0)

def draw():
    global curr_vis
    global draw_domination

    if curr_vis == 0:
        topology.drawGraph(MAX_NODES_TO_DRAW_EDGES)
    elif curr_vis == 1:
        topology.drawSlvo()
    elif curr_vis == 2:
        topology.drawColoring()
    elif curr_vis == 3:
        topology.drawPairs(0)
    elif curr_vis == 4:
        topology.drawPairs(1)
    elif curr_vis == 5:
        topology.drawPairs(2)
    elif curr_vis == 6:
        topology.drawPairs(3)
    elif curr_vis == 7:
        topology.drawBackbones(draw_domination)

def keyPressed():
    global curr_vis
    global step_size
    global vis_names

    if key == ' ':
        toggleLooping()
    elif key == 'c':
        if curr_vis == 7:
            toggleDrawDomination()
    elif key == 'i':
        topology.switchFgBg()
    elif key == 'l':
        incrementVis()
        topology.mightResetCurrNode()
        print vis_names[curr_vis]
    elif key == 'h':
        decrementVis()
        topology.mightResetCurrNode()
        print vis_names[curr_vis]
    elif key == 'k':
        if curr_vis > 2 and curr_vis < 7:
```

```python
65                  topology.incrementCurrPair()
66              elif curr_vis == 7:
67                  topology.incrementCurrBackbone()
68              else:
69                  topology.incrementCurrNode(step_size)
70          elif key == 'j':
71              if curr_vis > 2 and curr_vis < 7:
72                  topology.decrementCurrPair()
73              elif curr_vis == 7:
74                  topology.decrementCurrBackbone()
75              else:
76                  topology.decrementCurrNode(step_size)
77          elif key == 'y':
78              saveFrame("../report/images/{}-#####.png".format(vis_names[curr_vis]))
79          elif key >= '0' and key <= '9':
80              step_size = 2**int(key)
81              print "New step size:", step_size
82          elif key == ']':
83              step_size = 2*step_size
84              print "New step size:", step_size
85          elif key == '[':
86              step_size = step_size/2
87              print "New step size:", step_size
88          elif key == 'm':
89              print "\n------ Help Menu ------"
90              print "Use 'hjkl' to move between visualizations"
91              print "Press 'i' to invert the color scheme"
92              print "Press 'y' to take a screenshot of the current frame"
93              print "Press 'c' to show the coverage of the backbone"
94              print "Entering a number n between 0 and 9 will set the step size to 2^n
     nodes"
95              print "Using ']' will double the step size, '[' will half it"
96              print "Press space to pause rotation of the sphere"
97
98  # def mouseDragged():
99  #     global topology
100 #     topology.updateRotation(mouseX, mouseY)
101
102 def toggleLooping():
103     global is_looping
104     if is_looping:
105         noLoop()
106         is_looping = False
107     else:
108         loop()
109         is_looping = True
110
111 def toggleDrawDomination():
112     global draw_domination
113     if draw_domination:
114         draw_domination = False
115     else:
116         draw_domination = True
117
118 def incrementVis():
119     global curr_vis
120     global topology
121     if curr_vis < 7:
122         curr_vis += 1
123     background(topology.color_bg)
124
125 def decrementVis():
126     global curr_vis
127     global topology
128     if curr_vis > 0:
129         curr_vis -= 1
130     background(topology.color_bg)
131
```

```python
132  def main():
133      # sys.setrecursionlimit(32000)
134
135      global is_looping
136      global draw_domination
137      global curr_vis
138      global step_size
139      global vis_names
140      is_looping = True
141      draw_domination = False
142      curr_vis = 0
143      step_size = 1
144      vis_names = ["rgg", "slvo", "color", "bipartite", "no-tails", "major-comp", "
         no-bridge", "backbone"]
145
146      global topology
147      topology = Square()
148      # topology = Disk()
149      # topology = Sphere()
150
151      topology.num_nodes = NUM_NODES
152      topology.avg_deg = AVG_DEG
153      topology.canvas_height = CANVAS_HEIGHT
154      topology.canvas_width = CANVAS_WIDTH
155
156      if RUN_BENCHMARK:
157          n_benchmark = 1
158          topology.prepBenchmark(n_benchmark)
159
160      run_time = time.clock()
161
162      topology.generateNodes()
163      topology.findEdges(method="cell")
164      topology.colorGraph()
165      topology.generateBackbones()
166
167      print "Average degree: {}".format(topology.findAvgDegree())
168      print "Min degree: {}".format(topology.getMinDegree())
169      print "Max degree: {}".format(topology.getMaxDegree())
170      print "Num edges: {}".format(topology.findNumEdges())
171      print "Node r: {0:.3f}".format(topology.node_r)
172      print "Terminal clique size: {}".format(topology.term_clique_size)
173      print "Number of colors: {}".format(len(set(topology.node_colors)))
174      print "Max degree when deleted: {}".format(max(topology.deg_when_del.values())
         )
175      color_cnt = Counter(topology.node_colors)
176      print "Max color set size: {}    color: {}".format(color_cnt.most_common(1)
         [0][1],
177                                                          color_cnt.most_common(1)
         [0][0])
178
179      run_time = time.clock() - run_time
180      print "Run time: {0:.3f} s".format(run_time)
181
182      print "\nPress 'm' for the menu"
183
184  main()
```

Listing 2: Topology class and subclasses

```python
1  import random
2  import math
3  import time
4  from collections import deque
5
6  # benchmarks (num_nodes, avg_deg)
7  SQUARE_BENCHMARKS = [(1000,32), (8000,64), (16000,32), (64000,64), (64000,128),
8                       (128000,64), (128000, 128)]
```

```
9  DISK_BENCHMARKS = [(8000,64), (64000,64), (64000,128)]
10 SPHERE_BENCHMARKS = [(16000,64), (32000,128), (64000,128)]
11
12 """
13 Topology - super class for the shape of the random geometric graph
14 """
15 class Topology(object):
16
17     num_nodes = 100
18     avg_deg = 0
19     canvas_height = 720
20     canvas_width = 720
21
22     def __init__(self):
23         self.nodes = []
24         self.edges = {}
25         self.node_r = 0.0
26         self.minDeg = ()
27         self.maxDeg = ()
28         self.slvo = []
29         self.deg_when_del = {}
30         self.node_colors = []
31         self.num_color_sets = 4
32         self.pairs = []
33         self.no_tails = []
34         self.major_comps = []
35         self.clean_pairs = []
36         self.backbones = []
37         self.backbones_meta = []
38         self.curr_node = 0
39         self.curr_pair = 0
40         self.curr_backbone = 0
41
42         self.rot = (0,0,0)
43         self.color_bg = 0
44         self.color_fg = 255
45         self.color_fill = 180
46
47     # public funciton for generating nodes of the graph, must be subclassed
48     def generateNodes(self):
49         print "Method for generating nodes not subclassed"
50
51     # public function for finding edges
52     def findEdges(self, method="brute"):
53         self._getRadiusForAverageDegree()
54         self._addNodesAsEdgeKeys()
55
56         if method == "brute":
57             self._bruteForceFindEdges()
58         elif method == "sweep":
59             self._sweepFindEdges()
60         elif method == "cell":
61             self._cellFindEdges()
62         else:
63             print "Find edges method not defined: {}".format(method)
64
65         self._findMinAndMaxDegree()
66
67     # brute force edge detection
68     def _bruteForceFindEdges(self):
69         for i, n in enumerate(self.nodes):
70             for j, m in enumerate(self.nodes):
71                 if i != j and self._distance(n, m) <= self.node_r:
72                     self.edges[n].append(j)
73
74     # sweep edge detection
75     def _sweepFindEdges(self):
76         self.nodes.sort(key=lambda x: x[0])
```

```python
77
78          for i, n in enumerate(self.nodes):
79              search_space = []
80              for j in range(1,self.num_nodes-i):
81                  if abs(n[0] - self.nodes[i+j][0]) <= self.node_r:
82                      search_space.append(i+j)
83                  else:
84                      break
85              for j in search_space:
86                  if self._distance(n, self.nodes[j]) <= self.node_r:
87                      self.edges[n].append(j)
88                      self.edges[self.nodes[j]].append(i)
89
90      # cell edge detection
91      def _cellFindEdges(self):
92          num_cells = int(1/self.node_r) + 1
93          cells = []
94          for i in range(num_cells):
95              cells.append([[] for j in range(num_cells)])
96
97          for i, n in enumerate(self.nodes):
98              cells[int(n[0]/self.node_r)][int(n[1]/self.node_r)].append(i)
99
100         for i in range(num_cells):
101             for j in range(num_cells):
102                 for n_i in cells[i][j]:
103                     for c in self._findAdjCells(i, j, num_cells):
104                         for m_i in cells[c[0]][c[1]]:
105                             if self._distance(self.nodes[n_i], self.nodes[m_i]) <=
     self.node_r:
106                                 self.edges[self.nodes[n_i]].append(m_i)
107                                 self.edges[self.nodes[m_i]].append(n_i)
108                     for m_i in cells[i][j]:
109                         if self._distance(self.nodes[n_i], self.nodes[m_i]) <=
     self.node_r and n_i != m_i:
110                             self.edges[self.nodes[n_i]].append(m_i)
111
112     # cell edge detection helper function
113     def _findAdjCells(self, i, j, n):
114         adj_cells = [(1,-1), (0,1), (1,1), (1,0)]
115         return (((i+x[0])%n,(j+x[1])%n) for x in adj_cells)
116
117     # function for finding the radius needed for the desired average degree
118     # must be subclassed
119     def _getRadiusForAverageDegree(self):
120         print "Method for finding necessary radius for average degree not
     subclassed"
121
122     # helper function for findEdges, initializes edges dict
123     def _addNodesAsEdgeKeys(self):
124         self.edges = {n:[] for n in self.nodes}
125
126     # claculates the distance between two nodes (2D)
127     def _distance(self, n, m):
128         return math.sqrt((n[0] - m[0])**2+(n[1] - m[1])**2)
129
130     # public function for finding the number of edges
131     def findNumEdges(self):
132         sigma_edges = 0
133         for k in self.edges.keys():
134             sigma_edges += len(self.edges[k])
135
136         return sigma_edges/2
137
138     # public function for finding the average degree of nodes
139     def findAvgDegree(self):
140         return 2*self.findNumEdges()/self.num_nodes
141
```

```python
        # helper funciton for finding nodes with min and max degree
        def _findMinAndMaxDegree(self):
            self.minDeg = self.edges.keys()[0]
            self.maxDeg = self.edges.keys()[0]

            for k in self.edges.keys():
                if len(self.edges[k]) < len(self.edges[self.minDeg]):
                    self.minDeg = k
                if len(self.edges[k]) > len(self.edges[self.maxDeg]):
                    self.maxDeg = k

        # public function for getting the minimum degree
        def getMinDegree(self):
            return len(self.edges[self.minDeg])

        # public functino for getting the maximum degree
        def getMaxDegree(self):
            return len(self.edges[self.maxDeg])

        # public function for setting up the benchmark to run, must be subclassed
        def prepBenchmark(self, n):
            print "Method for preparing benchmark not subclassed"

        # public function for drawing the graph
        def drawGraph(self, n_limit):
            self._drawNodes(self.nodes)
            if self.num_nodes <= n_limit:
                self._drawEdges(self.nodes)
            else:
                self._drawMinMaxDegNodes()

        # responsible for drawing the nodes in the canvas
        def _drawNodes(self, node_list):
            strokeWeight(2)
            stroke(self.color_fg)
            fill(self.color_fg)

            for n in node_list:
                ellipse(n[0]*self.canvas_width, n[1]*self.canvas_height, 5, 5)

        # responsible for drawing the edges in the canavas
        def _drawEdges(self, node_list):
            strokeWeight(1)
            stroke(self.color_fg)
            fill(self.color_fg)

            s = set(node_list)

            for n in node_list:
                for m_i in self.edges[n]:
                    if self.nodes[m_i] in s:
                        line(n[0]*self.canvas_width, n[1]*self.canvas_height, self.
nodes[m_i][0]*self.canvas_width, self.nodes[m_i][1]*self.canvas_height)

        # responsible for drawing the edges of the min and max degree nodes
        def _drawMinMaxDegNodes(self):
            strokeWeight(1)
            stroke(0, self.color_fg, 0)
            fill(self.color_fg)
            for n_i in self.edges[self.minDeg]:
                line(self.minDeg[0]*self.canvas_width, self.minDeg[1]*self.
canvas_height, self.nodes[n_i][0]*self.canvas_width, self.nodes[n_i][1]*self.
canvas_height)

            stroke(0, 0, self.color_fg)
            for n_i in self.edges[self.maxDeg]:
                line(self.maxDeg[0]*self.canvas_width, self.maxDeg[1]*self.
canvas_height, self.nodes[n_i][0]*self.canvas_width, self.nodes[n_i][1]*self.
```

```python
            canvas_height)

        # uses smallest last vertex ordering to color the graph
        def colorGraph(self):
            self.slvo, self.deg_when_del = self._smallestLastVertexOrdering()
            self.node_colors = self._assignNodeColors(self.slvo)
            self.color_map = self._mapColorsToRGB(self.node_colors)

        # constructs a degree structure and determines the smallest last vertex
        ordering
        def _smallestLastVertexOrdering(self):
            deg_sets = {l:set() for l in range(len(self.edges[self.maxDeg])+1)}
            deg_when_del = {n:len(self.edges[n]) for n in self.nodes}

            for i, n in enumerate(self.nodes):
                deg_sets[deg_when_del[n]].add(i)

            smallest_last_ordering = []

            clique_found = False
            j = len(self.nodes)
            while j > 0:
                # get the current smallest bucket
                curr_bucket = 0
                while len(deg_sets[curr_bucket]) == 0:
                    curr_bucket += 1

                # if all the remaining nodes are connected we have the terminal clique
                if not clique_found and len(deg_sets[curr_bucket]) == j:
                    clique_found = True
                    self.term_clique_size = curr_bucket

                # get node with smallest degree
                v_i = deg_sets[curr_bucket].pop()
                smallest_last_ordering.append(v_i)

                # decrement position of nodes that shared an edge with v
                for n_i in (n_i for n_i in self.edges[self.nodes[v_i]] if n_i in
        deg_sets[deg_when_del[self.nodes[n_i]]]):
                    deg_sets[deg_when_del[self.nodes[n_i]]].remove(n_i)
                    deg_when_del[self.nodes[n_i]] -= 1
                    deg_sets[deg_when_del[self.nodes[n_i]]].add(n_i)

                j -= 1

            # reverse list since it was built shortest-first
            return smallest_last_ordering[::-1], deg_when_del

        # assigns the colors to nodes given in a smallest-last vertex ordering as a
        parallel array
        def _assignNodeColors(self, slvo):
            colors = [-1 for _ in range(len(slvo))]
            for i in slvo:
                adj_colors = set([colors[j] for j in self.edges[self.nodes[i]]])
                color = 0
                while color in adj_colors:
                    color += 1
                colors[i] = color

            return colors

        # generates random color codes for each color set and returns them in a
        dictionary
        def _mapColorsToRGB(self, color_list):
            s = set(color_list)
            color_map = {}
            while len(s) > 0:
                c = s.pop()
```

```
269              color_map[c] = (random.randint(0,255), random.randint(0,255), random.
        randint(0,255))
270
271          return color_map
272
273      # draw nodes as they are removed in smallest-last vertex ordering
274      def drawSlvo(self):
275          l = [self.nodes[i] for i in self.slvo[0:self.num_nodes - self.curr_node]]
276          self._drawNodes(l)
277          self._drawEdges(l)
278
279      # increments curr_node, used to limit the number of nodes drawn
280      def incrementCurrNode(self, s):
281          if self.curr_node + s <= self.num_nodes:
282              self.curr_node += s
283              background(self.color_bg)
284          elif self.curr_node != self.num_nodes:
285              self.curr_node = self.num_nodes
286              background(self.color_bg)
287
288      # decrements curr_node, used to limit the number of nodes drawn
289      def decrementCurrNode(self, s):
290          if self.curr_node - s >= 0:
291              self.curr_node -= s
292              background(self.color_bg)
293          elif self.curr_node != 0:
294              self.curr_node = 0
295              background(self.color_bg)
296
297      # used to reset curr node if all nodes have been drawn and the method changes
298      def mightResetCurrNode(self):
299          if self.curr_node == self.num_nodes:
300              curr_node = 0
301              background(self.color_bg)
302
303      # increments curr_backbone, used to draw different backbones
304      def incrementCurrPair(self):
305          if self.curr_pair < len(self.pairs) - 1:
306              self.curr_pair += 1
307              background(self.color_bg)
308
309      # decrements curr_backbone, used to draw different backbones
310      def decrementCurrPair(self):
311          if self.curr_pair > 0:
312              self.curr_pair -= 1
313              background(self.color_bg)
314
315      # increments curr_backbone, used to draw different backbones
316      def incrementCurrBackbone(self):
317          if self.curr_backbone < len(self.backbones) - 1:
318              self.curr_backbone += 1
319              background(self.color_bg)
320
321      # decrements curr_backbone, used to draw different backbones
322      def decrementCurrBackbone(self):
323          if self.curr_backbone > 0:
324              self.curr_backbone -= 1
325              background(self.color_bg)
326
327      # switch foreground and background colors
328      def switchFgBg(self):
329          self.color_fg, self.color_bg = self.color_bg, self.color_fg
330          background(self.color_bg)
331
332      # # update the rotation of the drawing
333      # def updateRotation(self, x, y):
334      #     # self.rot = (self.rot[0], self.rot[1]-math.pi/100, self.rot[2])
335      #     # self.rot = (x*math.cos(self.rot[0])*math.pi/500, self.rot[1], self.rot
```

```
                [2])
336      #         self.rot = (self.rot[0], x*math.cos(self.rot[1])*math.pi/1000, self.rot
                [2])
337      #        # rotateX(self.rot[0])
338      #        # rotateZ(self.rot[2])
339      #        # rotateY(-1*self.rot[1])

341      # used to draw the graph with the nodes colored
342      def drawColoring(self):
343          l = [self.nodes[i] for i in self.slvo[0:self.curr_node]]
344          self._drawNodes(l)
345          self._applyColors(self.slvo[0:self.curr_node])
346          self._drawEdges(l)

348      # places colors on the nodes
349      def _applyColors(self, node_i_list):
350          strokeWeight(5)

352          num_colors = max(self.node_colors)

354          for n_i in node_i_list:
355              c = self.color_map[self.node_colors[n_i]]
356              stroke(c[0], c[1], c[2])
357              fill(c[0], c[1], c[2])
358              ellipse(self.nodes[n_i][0]*self.canvas_width, self.nodes[n_i][1]*self.
         canvas_height, 5, 5)

360      # public function for pairing the independent sets and picking the largest
         backbones
361      def generateBackbones(self):
362          # pair four largest independent sets
363          self.pairs = self._pairIndependentSets(self.node_colors)

365          # delete minor components and tails
366          self.no_tails, self.major_comps, self.clean_pairs = self._cleanPairs(self.
         pairs)

368          # pick two backbones of largest size
369          self.backbones, self.backbones_meta = self._getLargestBackbones(self.
         clean_pairs)

371          # calculate domination
372          self.backbones_meta = self._getDonimations(self.backbones, self.
         backbones_meta)

374      # pairs the four largest independent color sets
375      def _pairIndependentSets(self, color_list):
376          # the first four color sets should be the largest (slvo)
377          indep_sets = [set() for _ in range(self.num_color_sets)]

379          for i, n in enumerate(self.nodes):
380              if self.node_colors[i] < self.num_color_sets:
381                  indep_sets[self.node_colors[i]].add(i)

383          # return combinations of sets (union)
384          return [s1 | s2 for i, s1 in enumerate(indep_sets) for s2 in indep_sets[i
         +1:]]

386      # removes the minor components and tails from the bipartite subgraphs
387      def _cleanPairs(self, bipartites):
388          no_tails = []
389          major_comps = []
390          results = []
391          for b in bipartites:
392              # remove the tails and save the graph for visualization
393              b = self._removeTails(b)
394              no_tails.append(b)
395
```

```python
                # use BFS to get the major component
                major_comp = self._bfs(b)
                major_comps.append(major_comp)

                # use DFS to remove bridges
                backbone = self._removeBridges(major_comp)
                results.append(backbone)

        return no_tails, major_comps, results

    # remove tails from bipartite, very similar to smallest-last vertex ordering
    def _removeTails(self, bipartite):
        bipartite = bipartite.copy()
        # build graph representation
        points = list(bipartite)
        deg_sets = {l:set() for l in range(len(self.edges[self.maxDeg])+1)}
        deg_map = {n_i:len([e_i for e_i in self.edges[self.nodes[n_i]] if e_i in
bipartite]) for n_i in points}

        for i in points:
            deg_sets[deg_map[i]].add(i)

        # remove nodes with zero or one edge until there are no tails
        while len(deg_sets[0]) > 0 or len(deg_sets[1]) > 0:
            to_remove = deg_sets[0] | deg_sets[1]
            deg_sets[0] = set()
            deg_sets[1] = set()

            for n_i in list(to_remove):
                for e_i in [e_i for e_i in self.edges[self.nodes[n_i]] if e_i in
bipartite]:
                    if e_i in deg_sets[deg_map[e_i]]:
                        deg_sets[deg_map[e_i]].remove(e_i)
                        deg_map[e_i] -= 1
                        deg_sets[deg_map[e_i]].add(e_i)

                bipartite.remove(n_i)

        return bipartite

    # use BFS to find the major component
    def _bfs(self, bipartite, rm_edges=None):
        points = list(bipartite)
        # used to index into the points array
        index_to_local = {n_i:i for i, n_i in enumerate(points)}
        # used to index into the nodes array
        index_to_global = {i:n_i for i, n_i in enumerate(points)}
        visited = [0 for _ in points]
        visits = []
        components = []

        while 0 in visited:
            visit = 1

            queue = deque()
            root = visited.index(0)
            queue.append(root)
            visited[root] = 1
            # builds a set for the points in each component
            components.append(set([index_to_global[root]]))

            while len(queue) > 0:
                curr = queue.pop()

                for e in [index_to_local[e] for e in self.edges[self.nodes[points[
curr]]] if e in bipartite]:
                    if rm_edges != None and (e in rm_edges and curr in rm_edges):
                        continue
```

```python
461                         if visited[e] == 0:
462                             visit += 1
463                             queue.append(e)
464                             components[-1].add(index_to_global[e])
465                             visited[e] = 1
466
467                 visits.append(visit)
468
469             if len(components) > 0:
470                 return components[visits.index(max(visits))]
471             else:
472                 return set()
473
474         # removes all bridges and minor blocks from major component
475         # algorithm: https://e-maxx-eng.appspot.com/graph/bridge-searching.html
476         def _removeBridges(self, major_comp):
477             points = list(major_comp)
478             # used to index into the points array
479             index_to_local = {n_i:i for i, n_i in enumerate(points)}
480             # used to index into the nodes array
481             index_to_global = {i:n_i for i, n_i in enumerate(points)}
482             visited = [0 for _ in points]
483             bridge_nodes = set()
484             tin = [-1 for _ in points]
485             fup = [-1 for _ in points]
486             visit = 0
487
488             for i, p in enumerate(points):
489                 if visited[i] == 0:
490                     self._dfs(major_comp, points, i, p, index_to_local, visited,
       bridge_nodes, tin, fup, visit)
491
492             return self._bfs(major_comp, bridge_nodes)
493
494         # use DFS to find bridges
495         def _dfs(self, comp, points, i, p, index_to_local, visited, bridge_nodes, tin,
        fup, visit, to=-1):
496             visited[i] = 1
497             tin[i] = visit
498             fup[i] = visit
499             visit += 1
500             for e in [index_to_local[e] for e in self.edges[self.nodes[p]] if e in
       comp]:
501                 if e == to:
502                     continue
503                 if visited[e] == 1:
504                     fup[i] = min(fup[i], tin[e])
505                 else:
506                     self._dfs(comp, points, e, points[e], index_to_local, visited,
       bridge_nodes, tin, fup, visit, to=i)
507                     fup[i] = min(fup[i], fup[e])
508                     if fup[e] > tin[i]:
509                         if i not in bridge_nodes:
510                             bridge_nodes.add(i)
511                         if e not in bridge_nodes:
512                             bridge_nodes.add(e)
513
514         # public function for drawing the color set pairs
515         def drawPairs(self, mode=0):
516             l_i = []
517             if mode == 0:
518                 l_i = list(self.pairs[self.curr_pair])
519             elif mode == 1:
520                 l_i = list(self.no_tails[self.curr_pair])
521             elif mode == 2:
522                 l_i = list(self.major_comps[self.curr_pair])
523             elif mode == 3:
524                 l_i = list(self.clean_pairs[self.curr_pair])
```

```python
        l_n = [self.nodes[i] for i in l_i]
        self._drawNodes(l_n)
        self._applyColors(l_i)
        self._drawEdges(l_n)


# returns the two major components with the largest size
def _getLargestBackbones(self, c_pairs):
    sizes = [-1]
    result = [None]
    # sizes = [-1, -1]
    # result = [None, None]
    for p in c_pairs:
        size = self._calcSize(p)

        if size > min(sizes):
            min_i = sizes.index(min(sizes))
            sizes[min_i] = size
            result[min_i] = p

    # saves backbone meta data (order, size)
    meta = [(len(result[i]), sizes[i]) for i in range(len(result))]
    if len(result) > 1 and sizes[1] > sizes[0]:
        return result[::-1], meta[::-1]

    return result, meta

# calculates the size of a graph
def _calcSize(self, graph):
    size = 0
    for n_i in list(graph):
        size += len([e for e in self.edges[self.nodes[n_i]] if e in graph])

    return size

# calculates the percentage of nodes covered by each backbone
def _getDonimations(self, b_bones, meta):
    for i, b in enumerate(b_bones):
        # find the number of nodes that do not share an edge with a backbone
node
        # search all nodes not in backbone
        search_space = set(range(self.num_nodes)) - b
        for n_i in list(search_space):
            for e in self.edges[self.nodes[n_i]]:
                if e in b:
                    search_space.remove(n_i)
                    break

        meta[i] = (meta[i][0], meta[i][1], (self.num_nodes - len(search_space)
    + 0.0)/self.num_nodes)

    return meta

# public function for drawing the backbones
def drawBackbones(self, draw_domination=False):
    l_i = list(self.backbones[self.curr_backbone])
    l_n = [self.nodes[i] for i in l_i]
    if draw_domination:
        self._drawDomination(l_i)
    else:
        background(self.color_bg)
    self._drawNodes(l_n)
    self._applyColors(l_i)
    self._drawEdges(l_n)

# draws connection radius around backbone nodes
def _drawDomination(self, node_i_list):
    strokeWeight(5)
```

```
591            stroke ( self . color_fill )
592            fill ( self . color_fill )
593
594            for n_i in node_i_list :
595                ellipse ( self . nodes [ n_i ] [ 0 ] * self . canvas_width , self . nodes [ n_i ] [ 1 ] * self .
      canvas_height , 2* self . node_r * self . canvas_width , 2* self . node_r * self .
      canvas_height )
596
597 """
598 Square − inherits from Topology , overloads generateNodes and
      _getRadiusForAverageDegree
599 for a unit square topology
600 """
601 class Square ( Topology ) :
602
603     def __init__ ( self ) :
604         super ( Square , self ) . __init__ ( )
605
606     # places nodes uniformly in a unit square
607     def generateNodes ( self ) :
608         for i in range ( self . num_nodes ) :
609             self . nodes . append ( ( random . uniform ( 0 , 1 ) , random . uniform ( 0 , 1 ) ) )
610
611     # calculates the radius needed for the requested average degree in a unit
      square
612     def _getRadiusForAverageDegree ( self ) :
613         self . node_r = math . sqrt ( self . avg_deg / ( self . num_nodes * math . pi ) )
614
615     # gets benchmark setting for square
616     def prepBenchmark ( self , n ) :
617         self . num_nodes = SQUARE_BENCHMARKS [ n ] [ 0 ]
618         self . avg_deg = SQUARE_BENCHMARKS [ n ] [ 1 ]
619
620 """
621 Disk − inherits from Topology , overloads generateNodes and
      _getRadiusForAverageDegree
622 for a unit circle topology
623 """
624 class Disk ( Topology ) :
625
626     def __init__ ( self ) :
627         super ( Disk , self ) . __init__ ( )
628
629     # places nodes uniformly in a unit square and regenerates the node if it falls
630     # outside of the circle
631     def generateNodes ( self ) :
632         for i in range ( self . num_nodes ) :
633             p = ( random . uniform ( 0 , 1 ) , random . uniform ( 0 , 1 ) )
634             while self . _distance ( p , ( 0.5 , 0.5 ) ) > 0.5 :
635                 p = ( random . uniform ( 0 , 1 ) , random . uniform ( 0 , 1 ) )
636             self . nodes . append ( p )
637
638     # calculates the radius needed for the requested average degree in a unit
      circle
639     def _getRadiusForAverageDegree ( self ) :
640         self . node_r = math . sqrt ( ( self . avg_deg + 0.0 ) / self . num_nodes ) / 2
641
642     # gets benchmark setting for disk
643     def prepBenchmark ( self , n ) :
644         self . num_nodes = DISK_BENCHMARKS [ n ] [ 0 ]
645         self . avg_deg = DISK_BENCHMARKS [ n ] [ 1 ]
646
647 """
648 Sphere − inherits from Topology , overloads generateNodes ,
      _getRadiusForAverageDegree ,
649 and _distance for a unit sphere topology . Also updates the drawGraph function for
650 a 3D canvas
651 """
```

```python
class Sphere(Topology):

    # adds rotation and node limit variables
    def __init__(self):
        super(Sphere, self).__init__()
        self.rot = (0,math.pi/4,0) # this may move to Topology if rotation is
    given to the 2D shapes
        # used to control _drawNodes functionality
        self.n_limit = 8000
        self.num_faces = []

    # places nodes in a unit cube and projects them onto the surface of the sphere
    def generateNodes(self):
        for i in range(self.num_nodes):
            # equations for uniformly distributing nodes on the surface area of
            # a sphere: http://mathworld.wolfram.com/SpherePointPicking.html
            u = random.uniform(-1,1)
            theta = random.uniform(0, 2*math.pi)
            p = (
                math.sqrt(1 - u**2) * math.cos(theta),
                math.sqrt(1 - u**2) * math.sin(theta),
                u
            )
            self.nodes.append(p)

    # calculates the radius needed for the requested average degree in a unit
    sphere
    def _getRadiusForAverageDegree(self):
        self.node_r = math.sqrt((self.avg_deg + 0.0)/self.num_nodes)*2

    # calculates the distance between two nodes (3D)
    def _distance(self, n, m):
        return math.sqrt((n[0] - m[0])**2+(n[1] - m[1])**2+(n[2] - m[2])**2)

    # gets benchmark setting for sphere
    def prepBenchmark(self, n):
        self.num_nodes = SPHERE_BENCHMARKS[n][0]
        self.avg_deg = SPHERE_BENCHMARKS[n][1]

    # public function for drawing graph, updates node limit if necessary
    def drawGraph(self, n_limit):
        self.n_limit = n_limit
        self._drawNodesAndEdges(self.nodes)

    # responsible for drawing nodes and edges in 3D space
    def _drawNodesAndEdges(self, node_list):
        # positions camera
        camera(self.canvas_width/2, self.canvas_height/2, self.canvas_width*-2,
    0.5,0.5,0, 0,1,0)

        # updates rotation
        self.rot = (self.rot[0], self.rot[1]-math.pi/100, self.rot[2])

        background(self.color_bg)
        strokeWeight(2)
        stroke(self.color_fg)
        fill(self.color_fg)

        s = set(node_list)

        for n in node_list:
            pushMatrix()

            # sets new rotation
            rotateZ(self.rot[2])
            rotateY(-1*self.rot[1])

            # sets drawing origin to current node
```

```python
                translate(n[0]*self.canvas_width, n[1]*self.canvas_height, n[2]*self.
        canvas_width)

                # places ellipse at origin
                ellipse(0, 0, 10, 10)

                # draw all edges
                if len(node_list) <= self.n_limit:
                    for e_i in self.edges[n]:
                        if self.nodes[e_i] in s:
                            e = self.nodes[e_i]
                            # draws line from origin to neighboring node
                            line(0,0,0, (e[0] - n[0])*self.canvas_width, (e[1] - n[1])
        *self.canvas_height, (e[2] - n[2])*self.canvas_width)
                # draw edges for min degree node
                elif n == self.minDeg:
                    stroke(0,self.color_fg,0)
                    for e_i in self.edges[n]:
                        e = self.nodes[e_i]
                        # draws line from origin to neighboring node
                        line(0,0,0, (e[0] - n[0])*self.canvas_width, (e[1] - n[1])*
        self.canvas_height, (e[2] - n[2])*self.canvas_width)
                    stroke(self.color_fg)
                # draw edges for max degree node
                elif n == self.maxDeg:
                    stroke(0,0,self.color_fg)
                    for e_i in self.edges[n]:
                        e = self.nodes[e_i]
                        # draws line from origin to neighboring node
                        line(0,0,0, (e[0] - n[0])*self.canvas_width, (e[1] - n[1])*
        self.canvas_height, (e[2] - n[2])*self.canvas_width)
                    stroke(self.color_fg)

                popMatrix()

    # draw nodes as they are removed in smallest-last vertex ordering
    def drawSlvo(self):
        l = [self.nodes[i] for i in self.slvo[0:self.num_nodes - self.curr_node]]
        self._drawNodesAndEdges(l)

    # used to draw the graph with the nodes colored
    def drawColoring(self):
        l = [self.nodes[i] for i in self.slvo[0:self.curr_node]]
        self._drawNodesAndEdges(l)
        self._applyColors(self.slvo[0:self.curr_node])

    # places colors on the nodes
    def _applyColors(self, node_i_list, draw_domination=False):
        strokeWeight(2)

        num_colors = max(self.node_colors)

        for n_i in node_i_list:
            c = self.color_map[self.node_colors[n_i]]
            stroke(c[0], c[1], c[2])
            fill(c[0], c[1], c[2])

            pushMatrix()

            # sets new rotation
            rotateZ(self.rot[2])
            rotateY(-1*self.rot[1])

            # sets drawing origin to current node
            translate(self.nodes[n_i][0]*self.canvas_width, self.nodes[n_i][1]*
        self.canvas_height, self.nodes[n_i][2]*self.canvas_width)

            # places ellipse at origin
```

```python
780                 ellipse(0, 0, 10, 10)

781
782                 if draw_domination:
783                     stroke(self.color_fill)
784                     fill(self.color_fill, 0.2)
785                     # places sphere at origin
786                     sphere(self.node_r*self.canvas_width)

787
788                 popMatrix()

789
790         # public function for pairing the independent sets and picking the largest
            backbones
791         def generateBackbones(self):
792             # uses base class method for generating backbones and meta data
793             super(Sphere, self).generateBackbones()

794
795             # calculate faces
796             self.num_faces = self._countFaces(self.backbones_meta)

797
798         # calcualtes the number of faces in the backbones of sphere topology
799         def _countFaces(self, b_meta):
800             # Euler's polyhedral formula
801             # http://mathworld.wolfram.com/PolyhedralFormula.html
802             return [2 - m[0] + m[1] for m in b_meta]

803
804         # public function for drawing the color set pairs
805         def drawPairs(self, mode=0):
806             l_i = []
807             if mode == 0:
808                 l_i = list(self.pairs[self.curr_pair])
809             elif mode == 1:
810                 l_i = list(self.no_tails[self.curr_pair])
811             elif mode == 2:
812                 l_i = list(self.major_comps[self.curr_pair])
813             elif mode == 3:
814                 l_i = list(self.clean_pairs[self.curr_pair])

815
816             l_n = [self.nodes[i] for i in l_i]
817             self._drawNodesAndEdges(l_n)
818             self._applyColors(l_i)

819
820         # public function for drawing the backbones
821         def drawBackbones(self, draw_domination=False):
822             l_i = list(self.backbones[self.curr_backbone])
823             l_n = [self.nodes[i] for i in l_i]
824             self._drawNodesAndEdges(l_n)
825             self._applyColors(l_i, draw_domination)
```