Linear Backbone Determination in a Wireless Sensor Network

Jake Carlson

April 10, 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. 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, finds multiple backbones for the RGG, and displays the results graphically.

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 12. 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	14506	29	44	9	0.089
2	8000	64	Square	0.050	243981	60	89	13	1.204
3	16000	32	Square	0.025	251000	31	56	5	1.300
4	64000	64	Square	0.018	2017430	63	97	16	10.448
5	64000	128	Square	0.025	4002176	125	173	39	18.430
6	128000	64	Square	0.013	4051857	63	104	16	19.670
7	128000	128	Square	0.018	8071395	126	178	40	36.893
8	8000	64	Disk	0.045	247485	61	96	17	1.123
9	64000	64	Disk	0.016	2020273	63	97	19	9.439
10	64000	128	Disk	0.022	4019411	125	170	41	18.206
11	16000	64	Sphere	0.126	512866	64	95	33	19.826
12	32000	128	Sphere	0.126	2046702	127	176	91	80.319
13	64000	128	Sphere	0.089	4099508	128	171	87	147.304

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

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.

Benchmark	Max Deg Deleted	Color Sets	Largest Color Set	Terminal Clique Size
1	20	20	79	19
2	38	36	324	30
3	25	25	1151	24
4	43	41	2538	40
5	73	66	1378	61
6	41	39	5059	36
7	73	67	2735	65
8	43	42	323	41
9	40	38	2534	34
10	73	65	1379	52
11	40	38	632	36
12	91	66	680	57
13	87	68	1349	53

Table 2: Benchmarks for Coloring RGGs

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 3.

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

Table 3: 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 $\Theta(|E| + |V|)$ 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 $O(|V| + |E|)$.

2.3 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:

$$\frac{(2r)^2 - \pi r^2}{(2r)^2} = \frac{4 - \pi}{4} = 0.2146 \tag{6}$$

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:

$$N = \sum_{k=0}^{\infty} n(0.2146)^k = \frac{n}{1 - 0.2146} = 1.273n \tag{7}$$

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 $\Theta(n^2)$ 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 $O(nlg(n))$ time complexity [7]. 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 $O(nlg(n) + 2rn)$ 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:

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

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.

2.4 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 5, Disk are given in Figure 6, and Sphere are given in Figure 7. These histograms clearly follow a normal distribution, so the nodes must be placed uniformly.

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 dgree constant at 16. As we can see in Figure 4, 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 4. 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 8, 9, and 10 which plot the original distribution

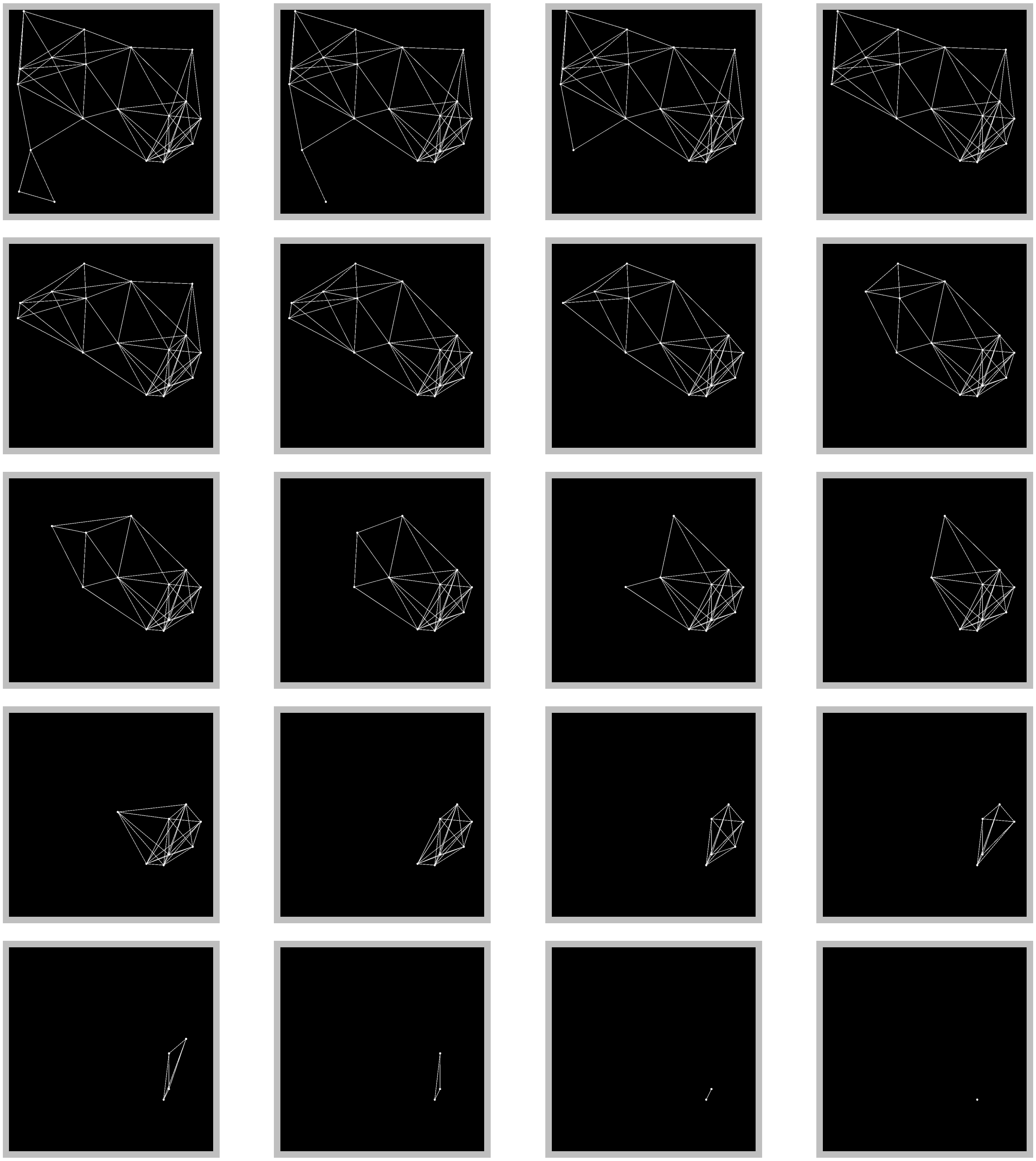


Figure 1: Smallest-last vertex ordering deletion process

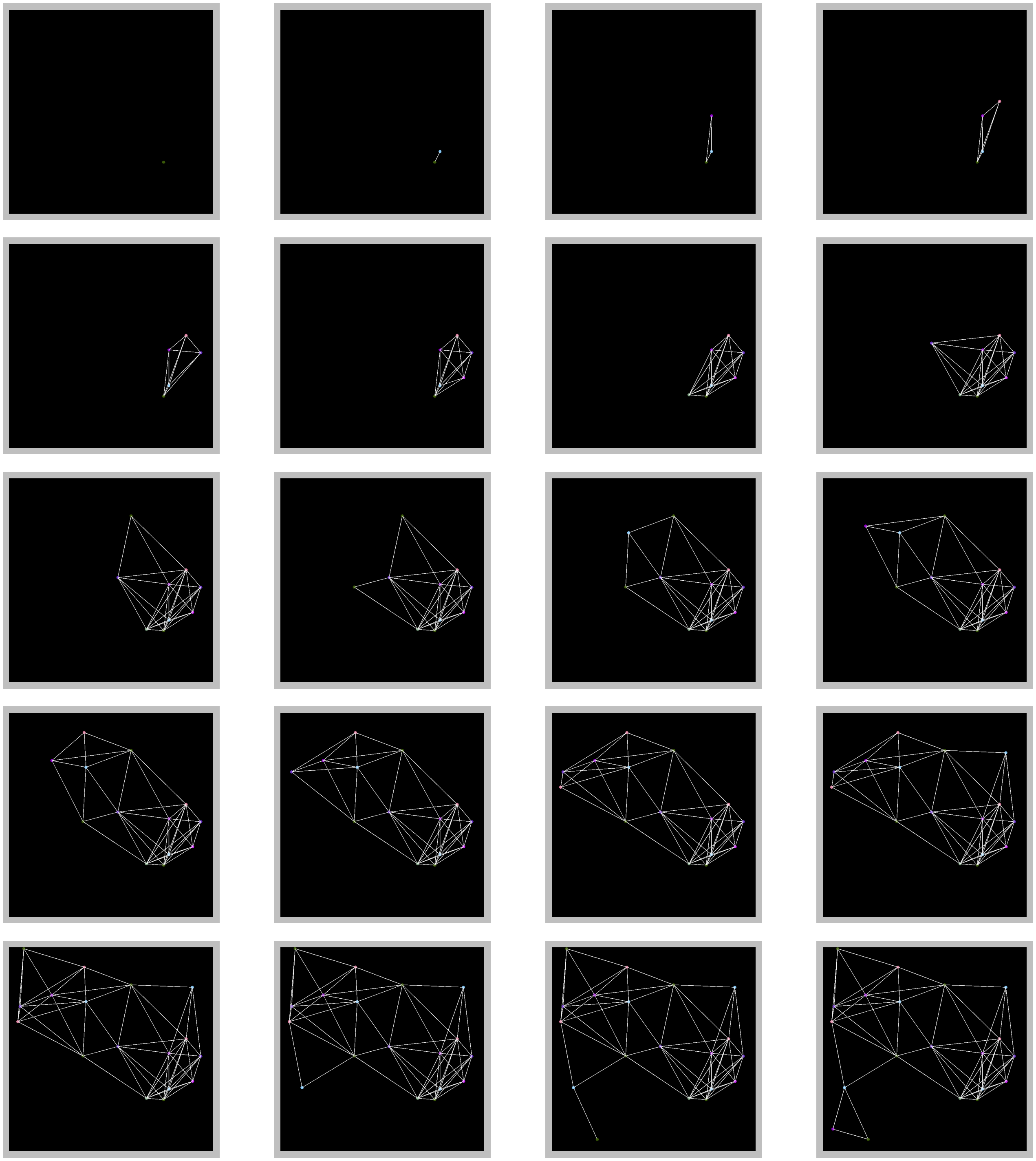


Figure 2: Smallest-last vertex ordering coloring process

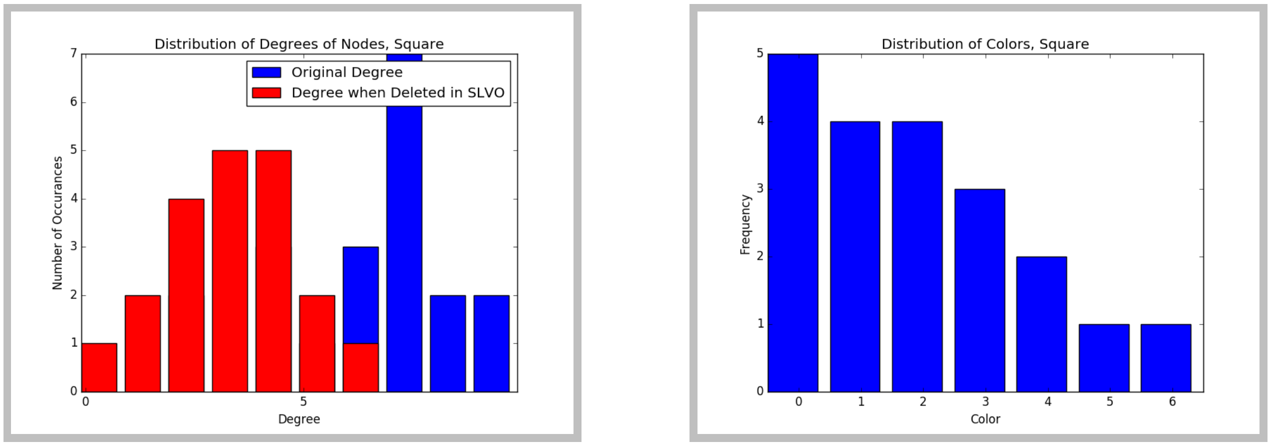


Figure 3: Distribution of degree when deleted and color set size for the 20 node walkthrough

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 11, 12, and 13.

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.

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

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

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

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

3 Appendix A - Figures

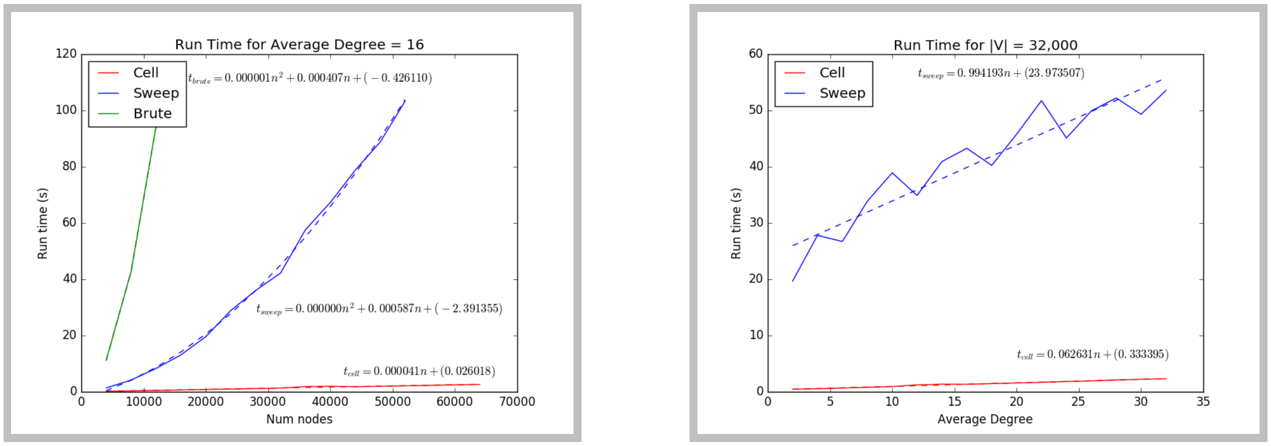


Figure 4: Runtime for edge detection methods. left: constant average degree of 16, right: variable average degree

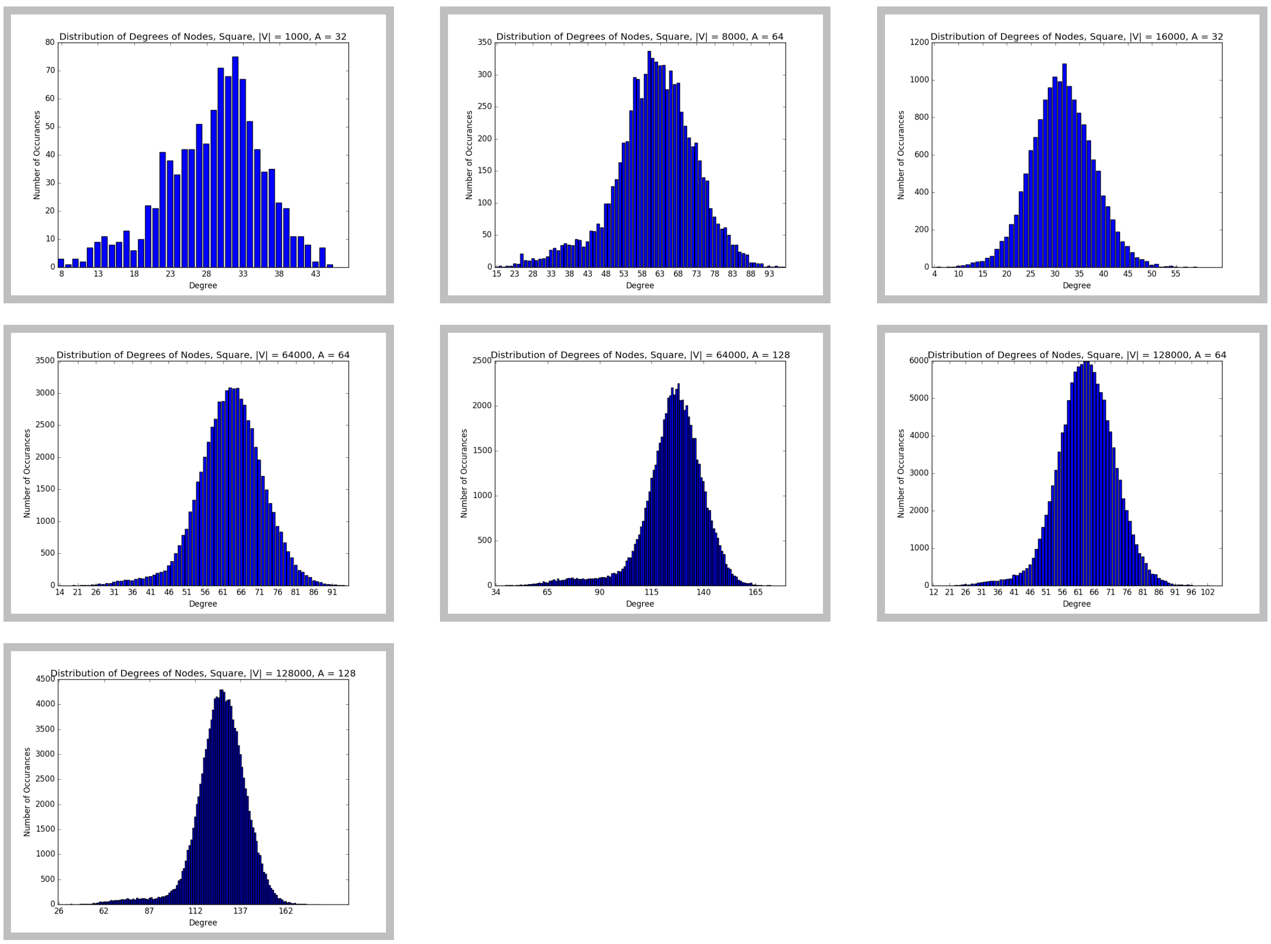


Figure 5: Square benchmarks distribution of degree graphs

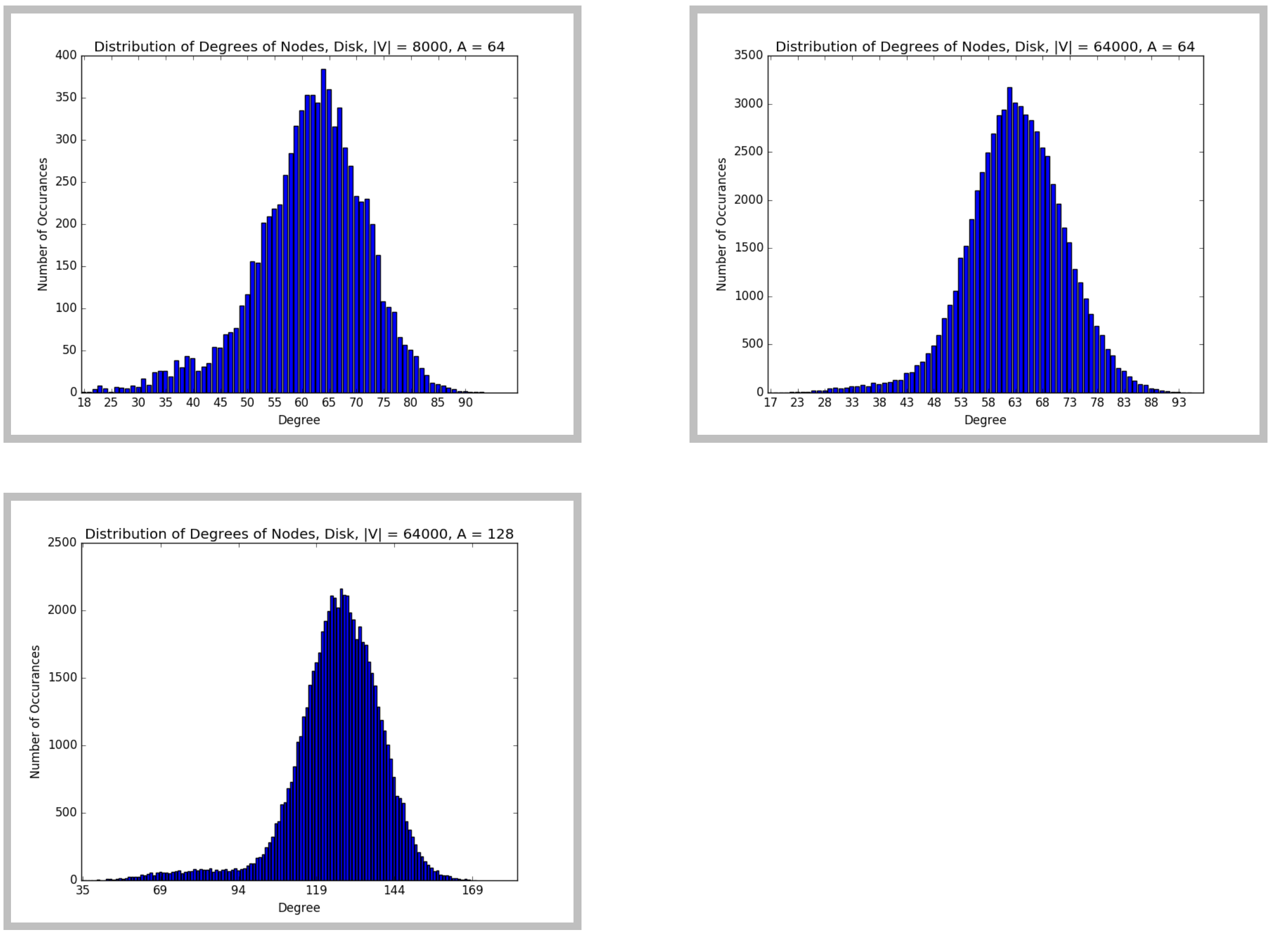


Figure 6: Disk benchmarks distribution of degree graphs

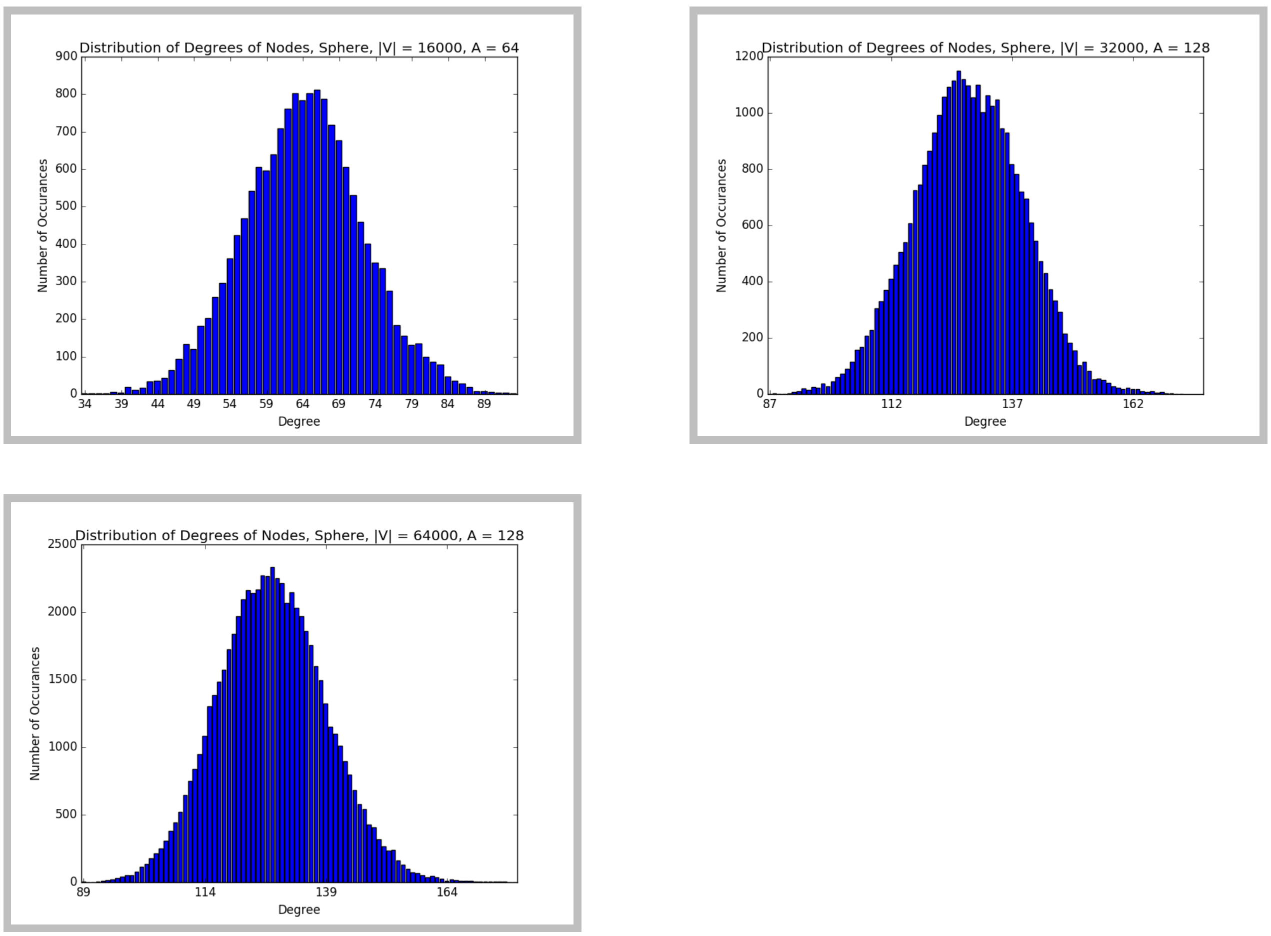


Figure 7: Sphere benchmarks distribution of degree graphs

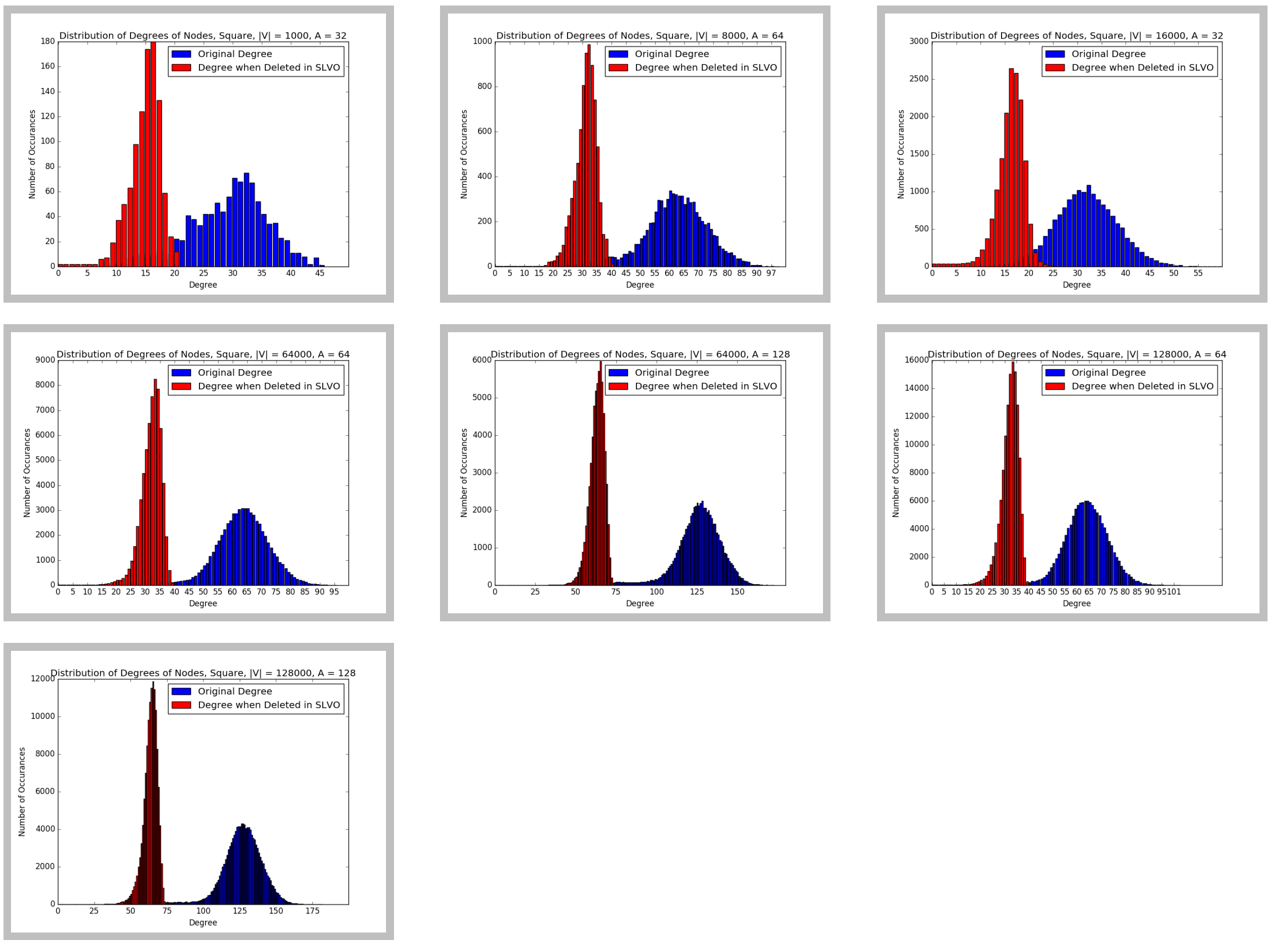


Figure 8: Square benchmarks distribution of degree when deleted graphs

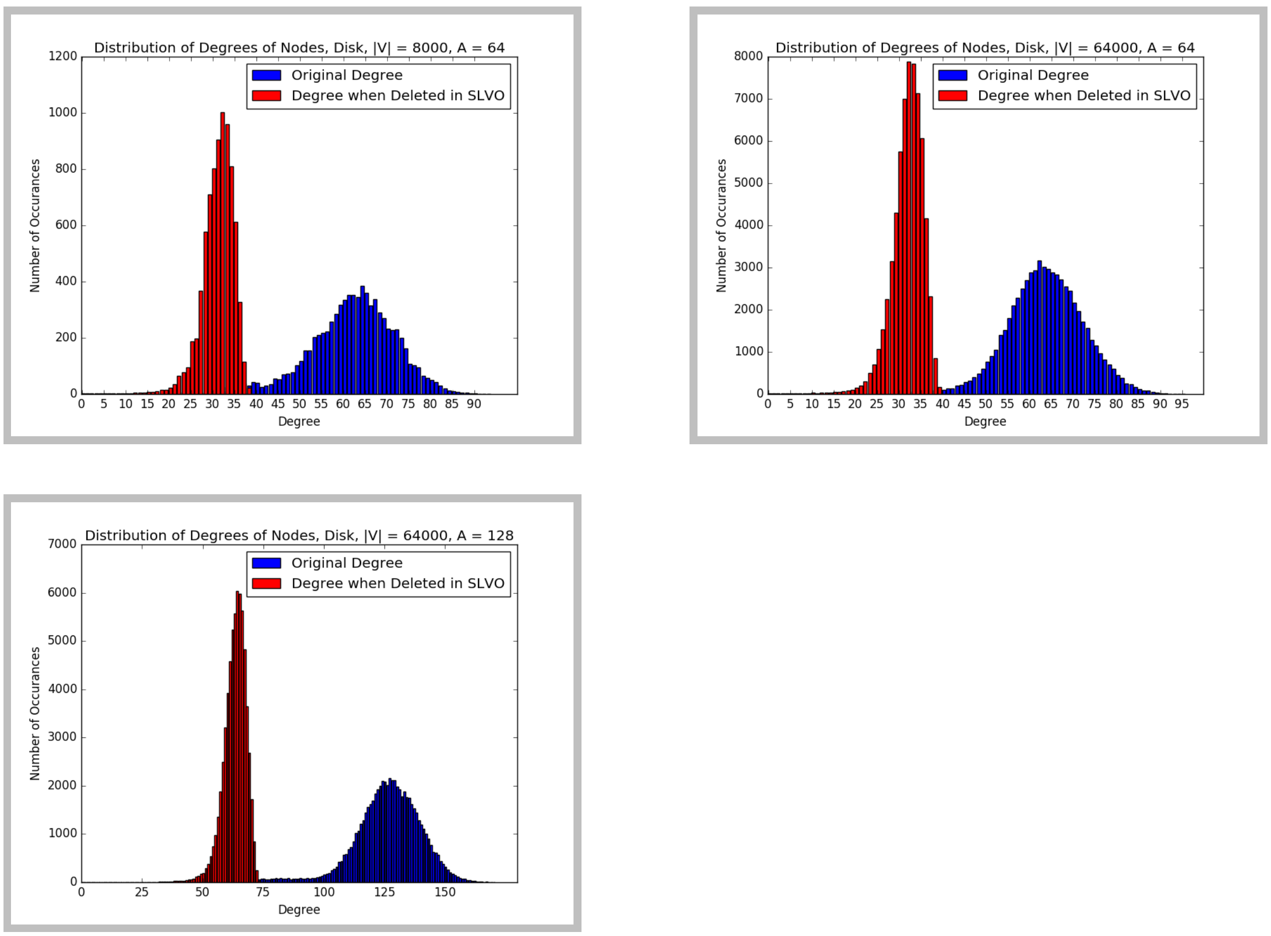


Figure 9: Disk benchmarks distribution of degree when deleted graphs

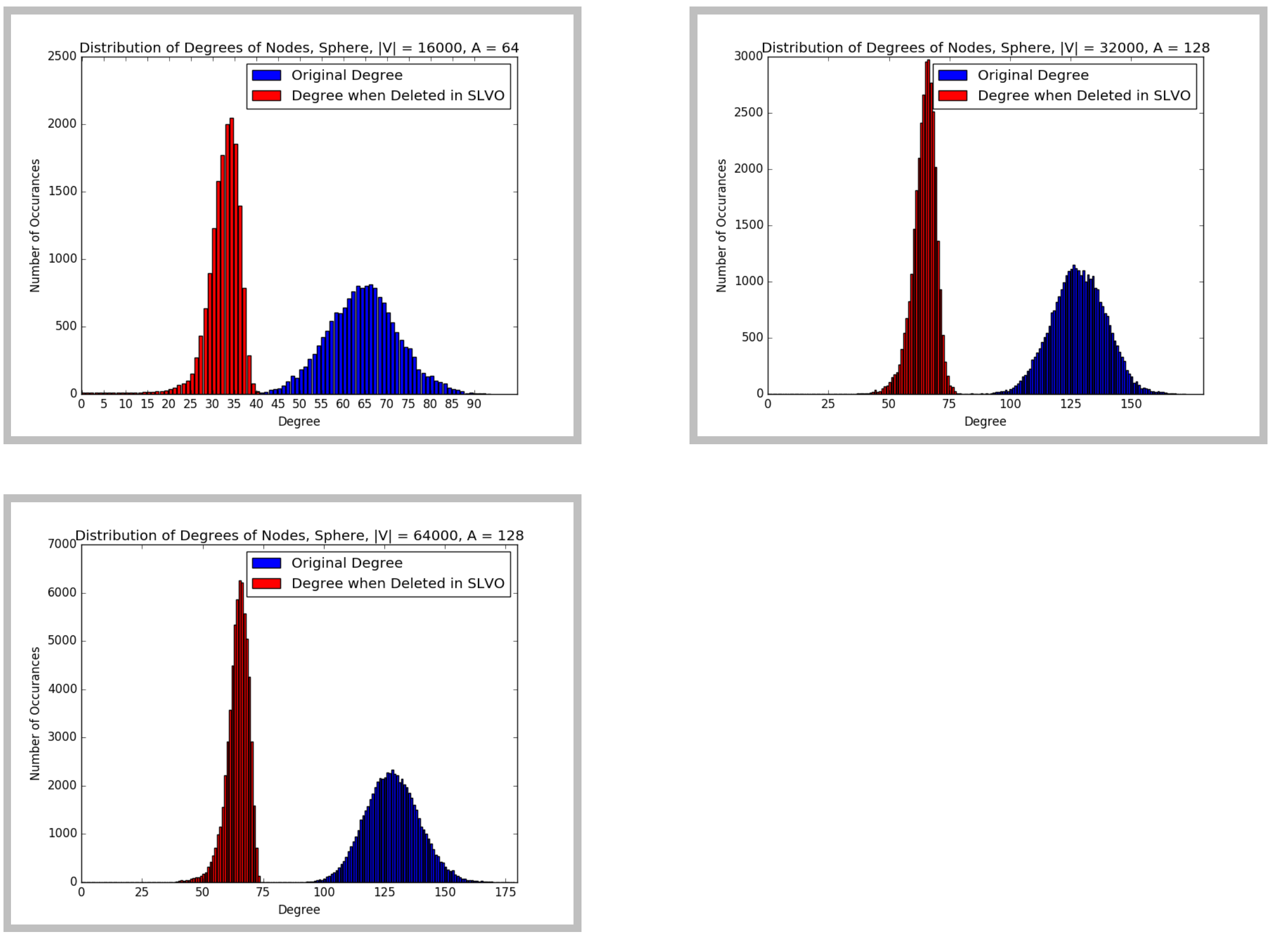


Figure 10: Sphere benchmarks distribution of degree when deleted graphs

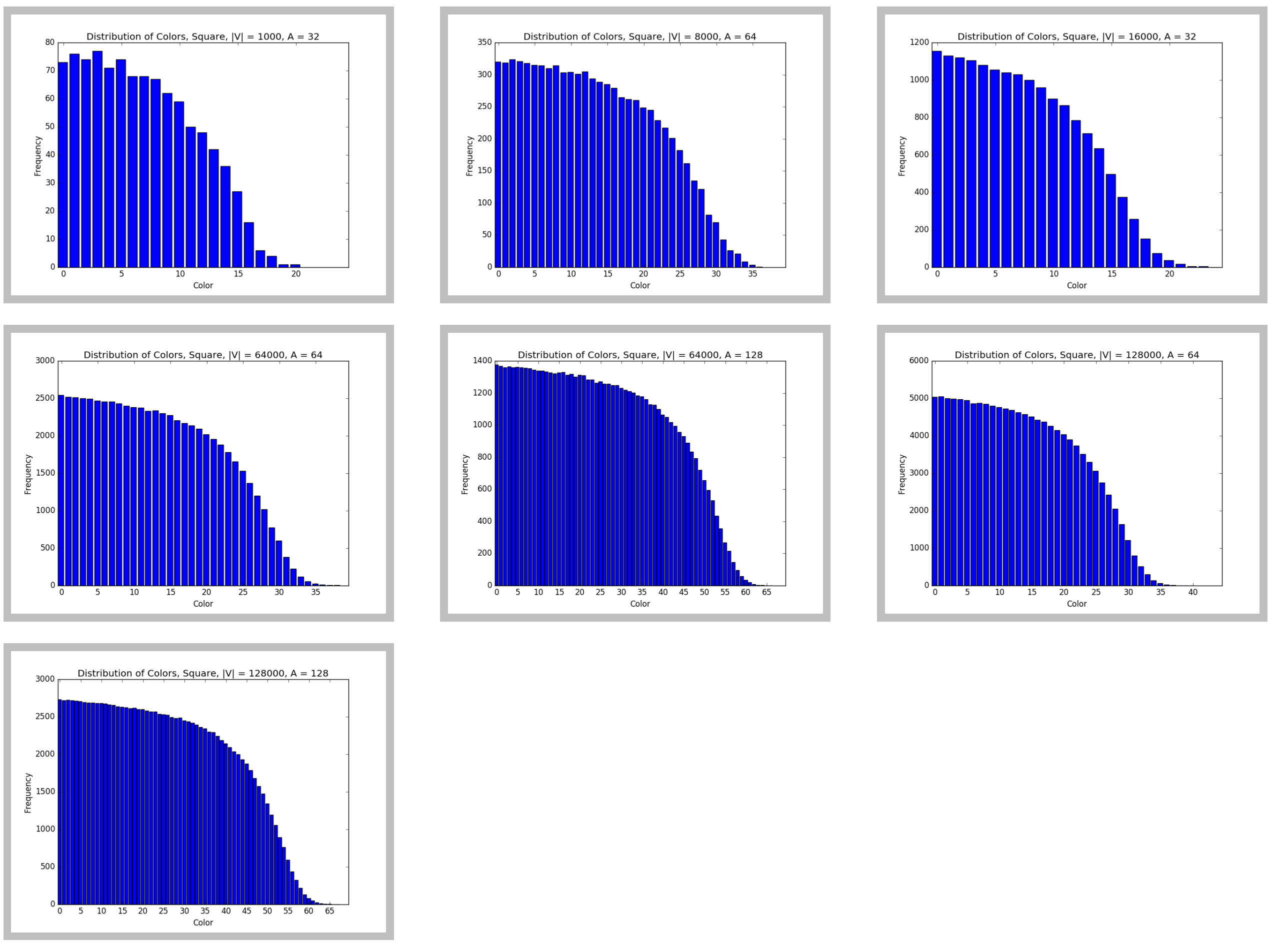
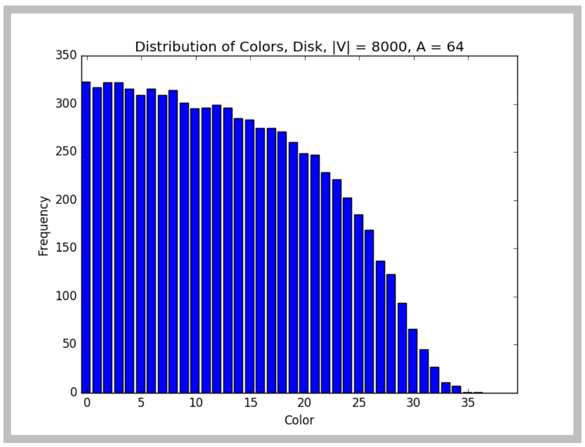
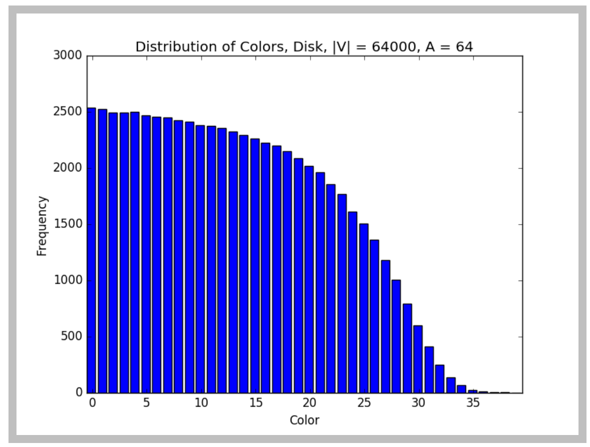


Figure 11: Square benchmarks distribution of colors graphs





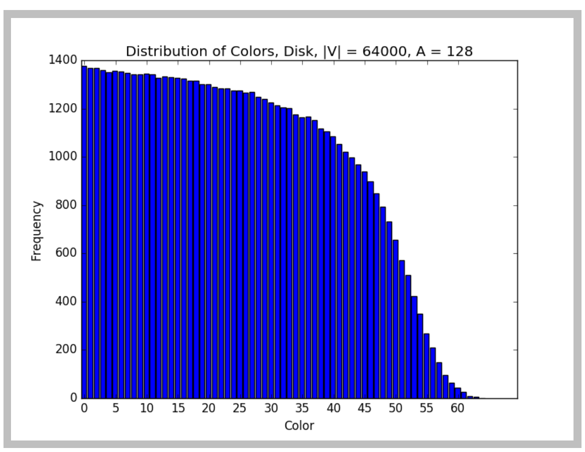


Figure 12: Disk benchmarks distribution of colors graphs

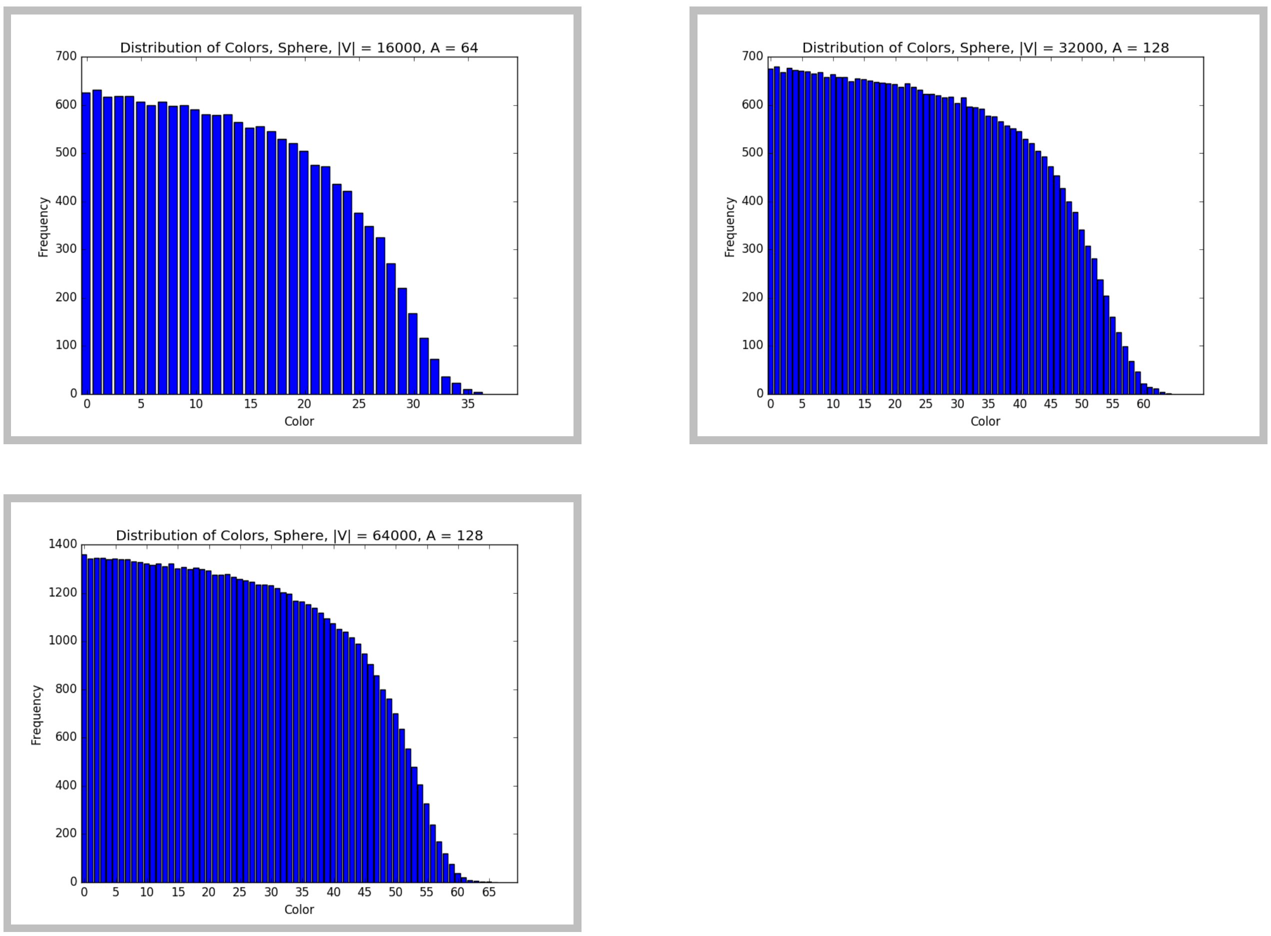


Figure 13: Sphere benchmarks distribution of colors graphs

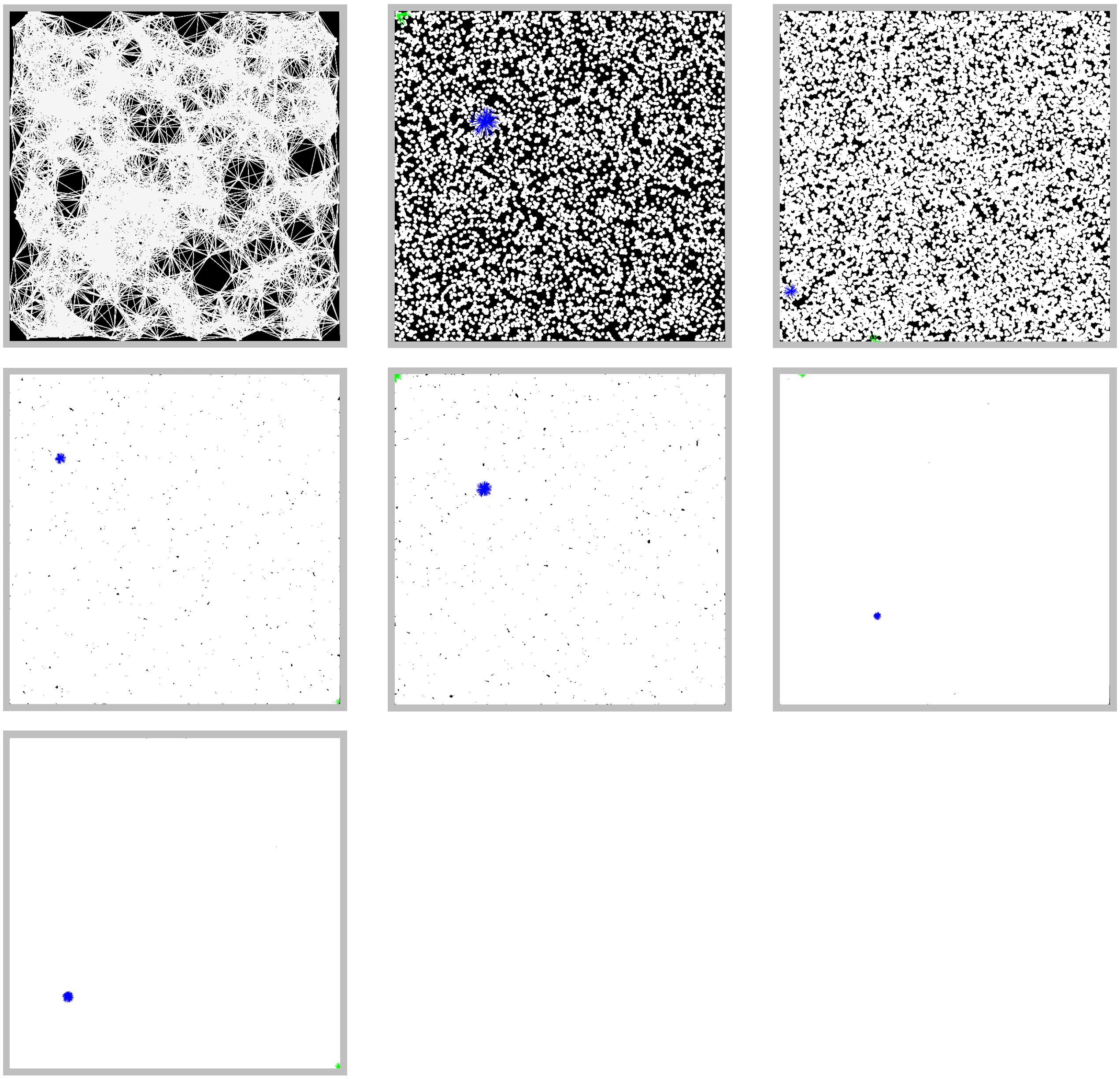


Figure 14: Square benchmark graphs

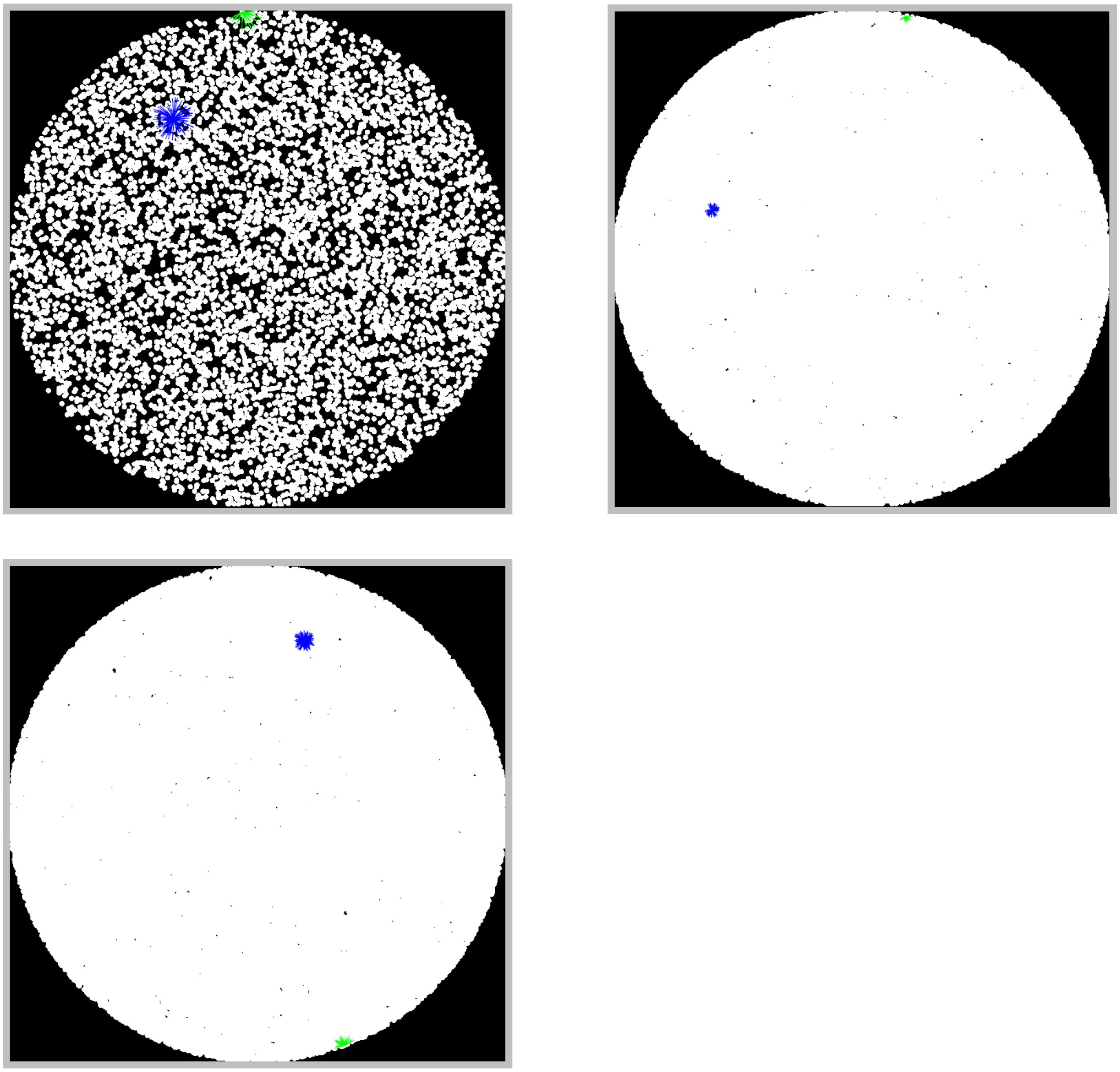


Figure 15: Disk benchmark graphs

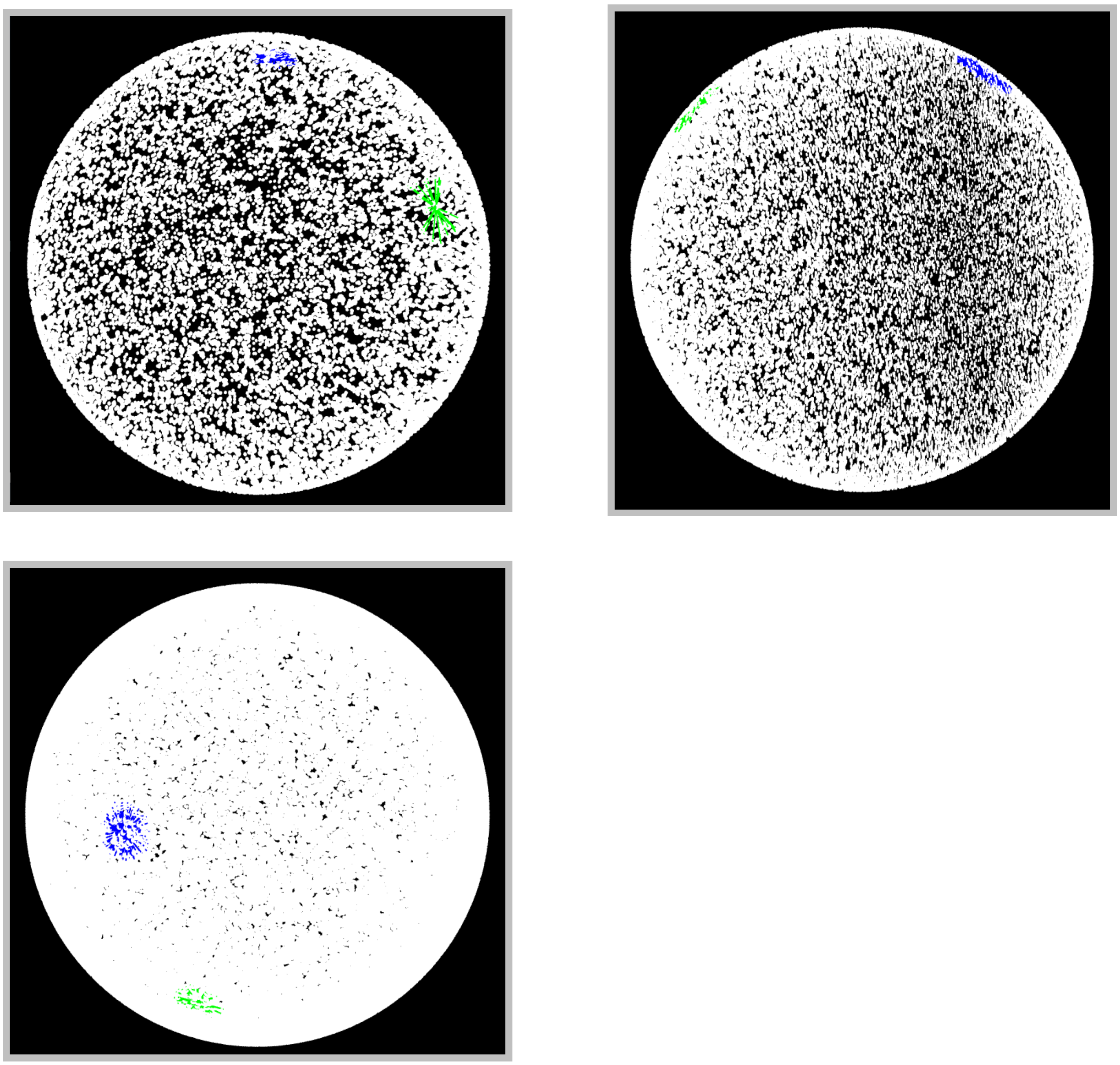


Figure 16: Sphere benchmark graphs

4 Appendix B - Code Listings

Listing 1: Processing driver

```
import random
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

    if curr_vis == 0:
        topology.drawGraph(MAX_NODES_TO_DRAW_EDGES)
    elif curr_vis == 1:
        topology.drawSlvo()
        # toggleLooping()
    elif curr_vis == 2:
        topology.drawColoring()
        # toggleLooping()

def keyPressed():
    if key == ' ':
        toggleLooping()
    elif key == 'l':
        incrementVis()
        topology.mightResetCurrNode()
    elif key == 'h':
        decrementVis()
        topology.mightResetCurrNode()
    elif key == 'k':
        topology.incrementCurrNode()
    elif key == 'j':
        topology.decrementCurrNode()
    elif key == 'y':
        global curr_vis
        saveFrame("../report/images/{}-{}.png".format("slvo" if curr_vis == 1 else "color", topology.curr_node))

def toggleLooping():
    global is_looping
    if is_looping:
        noLoop()
        is_looping = False
    else:
        loop()
        is_looping = True

def incrementVis():
    global curr_vis
    if curr_vis < 2:
        curr_vis += 1
    background(0)
```

```python

def decrementVis():
    global curr_vis
    if curr_vis > 0:
        curr_vis -= 1
    background(0)

def main():
    global is_looping
    global curr_vis
    is_looping = True
    curr_vis = 0

    global topology
    topology = Square()
    # topology = Disk()
    # topology = Sphere()

    topology.num_nodes = NUM_NODES
    topology.avg_deg = AVG_DEG
    topology.canvas_height = CANVAS_HEIGHT
    topology.canvas_width = CANVAS_WIDTH

    if RUN_BENCHMARK:
        n_benchmark = 0
        topology.prepBenchmark(n_benchmark)

    run_time = time.clock()

    topology.generateNodes()
    topology.findEdges(method="cell")
    topology.colorGraph()

    print "Average degree: {}".format(topology.findAvgDegree())
    print "Min degree: {}".format(topology.getMinDegree())
    print "Max degree: {}".format(topology.getMaxDegree())
    print "Num edges: {}".format(topology.findNumEdges())
    print "Node r: {0:.3f}".format(topology.node_r)
    print "Terminal clique size: {}".format(topology.term_clique_size)
    print "Number of colors: {}".format(len(set(topology.node_colors)))
    print "Max degree when deleted: {}".format(max(topology.deg_when_del.values()))
    color_cnt = Counter(topology.node_colors)
    print "Max color set size: {}  color: {}".format(color_cnt.most_common(1)[0][1],
                                                    color_cnt.most_common(1)[0][0])

    run_time = time.clock() - run_time
    print "Run time: {0:.3f} s".format(run_time)

main()
```

Listing 2: Topology class and subclasses

```python
import random
import math
import time

# benchmarks (num_nodes, avg_deg)
SQUARE_BENCHMARKS = [(1000,32), (8000,64), (16000,32), (64000,64), (64000,128),
                     (128000,64), (128000, 128)]
DISK_BENCHMARKS = [(8000,64), (64000,64), (64000,128)]
SPHERE_BENCHMARKS = [(16000,64), (32000,128), (64000,128)]

"""
Topology - super class for the shape of the random geometric graph
"""
```

```python
class Topology(object):

    num_nodes = 100
    avg_deg = 0
    canvas_height = 720
    canvas_width = 720

    def __init__(self):
        self.nodes = []
        self.edges = {}
        self.node_r = 0.0
        self.minDeg = ()
        self.maxDeg = ()
        self.slvo = []
        self.deg_when_del = {}
        self.node_colors = []
        self.curr_node = 0

    # public funciton for generating nodes of the graph, must be subclassed
    def generateNodes(self):
        print "Method for generating nodes not subclassed"

    # public function for finding edges
    def findEdges(self, method="brute"):
        self._getRadiusForAverageDegree()
        self._addNodesAsEdgeKeys()

        if method == "brute":
            self._bruteForceFindEdges()
        elif method == "sweep":
            self._sweepFindEdges()
        elif method == "cell":
            self._cellFindEdges()
        else:
            print "Find edges method not defined: {}".format(method)

        self._findMinAndMaxDegree()

    # brute force edge detection
    def _bruteForceFindEdges(self):
        for i, n in enumerate(self.nodes):
            for j, m in enumerate(self.nodes):
                if i != j and self._distance(n, m) <= self.node_r:
                    self.edges[n].append(j)

    # sweep edge detection
    def _sweepFindEdges(self):
        self.nodes.sort(key=lambda x: x[0])

        for i, n in enumerate(self.nodes):
            search_space = []
            for j in range(1,self.num_nodes-i):
                if abs(n[0] - self.nodes[i+j][0]) <= self.node_r:
                    search_space.append(i+j)
                else:
                    break
            for j in search_space:
                if self._distance(n, self.nodes[j]) <= self.node_r:
                    self.edges[n].append(j)
                    self.edges[self.nodes[j]].append(i)

    # cell edge detection
    def _cellFindEdges(self):
        num_cells = int(1/self.node_r) + 1
        cells = []
        for i in range(num_cells):
            cells.append([[] for j in range(num_cells)])

```

```
        for i, n in enumerate(self.nodes):
            cells[int(n[0]/self.node_r)][int(n[1]/self.node_r)].append(i)

        for i in range(num_cells):
            for j in range(num_cells):
                for n_i in cells[i][j]:
                    for c in self._findAdjCells(i, j, num_cells):
                        for m_i in cells[c[0]][c[1]]:
                            if self._distance(self.nodes[n_i], self.nodes[m_i]) <= self.node_r:
                                self.edges[self.nodes[n_i]].append(m_i)
                                self.edges[self.nodes[m_i]].append(n_i)
                    for m_i in cells[i][j]:
                        if self._distance(self.nodes[n_i], self.nodes[m_i]) <= self.node_r and n_i != m_i:
                            self.edges[self.nodes[n_i]].append(m_i)

    # cell edge detection helper function
    def _findAdjCells(self, i, j, n):
        adj_cells = [(1,-1), (0,1), (1,1), (1,0)]
        return (((i+x[0])%n,(j+x[1])%n) for x in adj_cells)

    # function for finding the radius needed for the desired average degree
    # must be subclassed
    def _getRadiusForAverageDegree(self):
        print "Method for finding necessary radius for average degree not subclassed"

    # helper function for findEdges, initializes edges dict
    def _addNodesAsEdgeKeys(self):
        self.edges = {n:[] for n in self.nodes}

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

    # public function for finding the number of edges
    def findNumEdges(self):
        sigma_edges = 0
        for k in self.edges.keys():
            sigma_edges += len(self.edges[k])

        return sigma_edges/2

    # public function for finding the average degree of nodes
    def findAvgDegree(self):
        return 2*self.findNumEdges()/self.num_nodes

    # 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
```

```python
    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(255)
        fill(255)

        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(245)
        fill(255)

        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(0,255,0)
        fill(255)
        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(0,0,255)
        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.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
```

```python
        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

    def _mapColorsToRGB(self, color_list):
        s = set(color_list)
        color_map = {}
        while len(s) > 0:
            c = s.pop()
            color_map[c] = (random.randint(0,255), random.randint(0,255), random.randint(0,255))

        return color_map

    # 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._drawNodes(l)
        self._drawEdges(l)

    # increments curr_node, used to limit the number of nodes drawn
    def incrementCurrNode(self):
        if self.curr_node < self.num_nodes:
            self.curr_node += 1
            background(0)

    # decrements curr_node, used to limit the number of nodes drawn
    def decrementCurrNode(self):
        if self.curr_node > 0:
            self.curr_node -= 1
            background(0)
```

```python

    # used to reset curr node if all nodes have been drawn and the method changes
    def mightResetCurrNode(self):
        if self.curr_node == self.num_nodes:
            curr_node = 0
            background(0)

    def drawColoring(self):
        l = [self.nodes[i] for i in self.slvo[0:self.curr_node]]
        self._drawNodes(l)
        self._applyColors(self.slvo[0:self.curr_node])
        self._drawEdges(l)

    def _applyColors(self, node_i_list):
        strokeWeight(5)

        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])
            ellipse(self.nodes[n_i][0]*self.canvas_width, self.nodes[n_i][1]*self.canvas_height, 5, 5)

"""
Square - inherits from Topology, overloads generateNodes and _getRadiusForAverageDegree
for a unit square topology
"""
class Square(Topology):

    def __init__(self):
        super(Square, self).__init__()

    # places nodes uniformly in a unit square
    def generateNodes(self):
        for i in range(self.num_nodes):
            self.nodes.append((random.uniform(0,1), random.uniform(0,1)))

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

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

"""
Disk - inherits from Topology, overloads generateNodes and _getRadiusForAverageDegree
for a unit circle topology
"""
class Disk(Topology):

    def __init__(self):
        super(Disk, self).__init__()

    # places nodes uniformly in a unit square and regenerates the node if it falls
    # outside of the circle
    def generateNodes(self):
        for i in range(self.num_nodes):
            p = (random.uniform(0,1), random.uniform(0,1))
            while self._distance(p, (0.5,0.5)) > 0.5:
                p = (random.uniform(0,1), random.uniform(0,1))
            self.nodes.append(p)
```

```python

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

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

"""
Sphere - inherits from Topology, overloads generateNodes, _getRadiusForAverageDegree,
and _distance for a unit sphere topology. Also updates the drawGraph function for
a 3D canvas
"""
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

    # 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(0)
        strokeWeight(2)
        stroke(255)
        fill(255)

        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
            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 self.num_nodes <= 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(0,255,0)
                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(255)
            # draw edges for max degree node
            elif n == self.maxDeg:
                stroke(0,0,255)
                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(255)

            popMatrix()
```