Linear Time Generation and Coloring of Wireless Sensor Networks Simulated with Random Geometric Graphs

Luke Wood

April 21, 2018

1 Executive Summary

Wireless Sensor Networks (WSNs) are a group of ad hoc wireless devices that communicate amongst themselves. WSNs have tons of applications coming in the near future ranging from traffic sensing networks to weather reading sensors on the surface of Mars[3]. They are incredibly expensive to develop and test which makes them a great candidate for simulation as a preliminary way to gather data. Vlady Ravelomananana and Hichem Kenniche from the University of Paris first explored the concept of using random geometric graphs (RGGs) to attempt to model wireless sensor networks [2]. At the moment random geographic graphs are the state of the art method for simulating WSNs. This project uses RGGs as a way to gather valuable information about how wireless sensor networks will possibly function and communicate.

1.1 Introduction and Summary

This report generates and analyzes RGGs in the following ways as an attempt to gain some insight into the behavior of wireless sensor networks:

1. Generating RGGs on the geometries of a unit square, unit disc, and unit sphere
2. Color the generated graph in linear time using smallest vertex last ordering[4]
3. Find the terminal clique in the generated RGGs
4. Find a selection of bipartite subgraphs producted by an algorithm for coloring

The generation portion of this report generates graphs consisting of N vertices with an average degree of A on the geometric topologies of unit square, unit

disc, unit sphere in linear time. Some tables are included to display benchmark data for generated RGGs of various size to display the performance of the implementation of the algorithms.

One interesting thing to notice in the tables is that as the number of nodes grows the real average degree converges on the expected average degree. The lower the number of nodes the further from the expected average degree the real number is. This happens due to there not being enough nodes in the graph for the real radius to reach the expected value. The random error in the point generation is more apparent when there are fewer nodes in the graph.

The coloring algorithms also run in $O(V + E)$ which is linear time as the edges are capped by the value A which is a constant. The runtimes could also be rewritten as $O(V + A * V)$ which is $O(V)$ assuming the A is a constant value.

Reference tables 3, 3, 3 for the full benchmarks.

1.2 Programming Environment Description

The implementation of the algorithm used to gather the data supporting this report was gathered on a 15 inch Macbook pro 2017 with a 2.9 GHz Intel Core i7 processor and 16 GB of RAM. The computer is running macOS High Sierra. The python environment is used for the ability to leverage the networkx library for plotting graphs as well as matplotlib to easily create charts. The algorithms are implemented to run in linear time and despite pythons low speed still run in reasonable time spans even on large input sizes.

The generation as well as coloring algorithms were both implemented again in the Elixir programming language, however this proved problematic as all data structures are immutable. On small inputs, the Elixir implementation ran up to 5 times as fast as the python implementation, however there is no data structure with $O(1)$ insertion, access, and deletion in Elixir. The best runtime possible in pure Elixir was $O(n * log(n))$ which was worse than the python version. This made it impossible to implement the coloring algorithm in linear time without interopping with C or Erlang. Due to time constraints, it was not feasible to get the interop working with Erlang or C in time. Thus the implementation is still in python, however if I could do it again I would either follow through with the Erlang interop and continue in Elixir or use a different platform entirely.

2 Reduction to Practice

This section will describe the transition from theory to implementation. This section will also give a detailed analysis of the algorithms used in this project as well as their asymptotic runtimes.

2.1 Data Structure Design

The generation portion of this project uses several different data structures. The first one is a python object of custom type node. This serves as a tuple

of values consisting of a list of dimensions, a list of nodes, and a node number. The first two are used during graph generation and the latter during conversion to an adjacency list. This data structure could be used interchangeably with a statically indexed tuple instead of an object to avoid any overhead associated with objects in python, however the readability gained from using a custom node class heavily outweighs the marginal performance benefit gained from using a statically indexed tuple. Both of these data structures provide $O(1)$ read and write operations. The python wiki states that dictionary access time is amortized $O(1)$, however in the worst case can be $O(n)$[5]. That being said, most interpreters will convert objects with known attributes into statically indexed arrays so it is likely that there would be neglible performance difference anyways. If the goal were to generate gigantic graphs then an argument could be made to switch over the statically indexed tuples to reduce the access time for attributes by a bit as tuples use static memory address offsets.

The second data structure used in the implementation is the adjacency list. Adjacency lists are an efficient graph representation that we will use in the subsequent reports. Adjacency lists require only $O(V+E)$ storage as opposed to the $O(v^2)$ required for adjacency matrixes. Despite the huge potential savings on storage, the only sacrifice adjacency lists make is in the lookup operation to determine if there is an edge between two nodes. The sacrifice made in adjacency lists is $O(A)$ lookup time for the existence of an edge as opposed to $O(1)$ lookup time available in adjacency matrixes. This makes adjacency lists useful in situations where the expected average number of edges is significantly lower than the number of nodes in the graph.

In the coloring portion of the project, I used the same data structures above as well as a hash set. When I am keeping track of which nodes are of degree n, I frequently have to move each node's neighbors down a degree. In order to get optimal big O, I needed to use a data structure with constant time access, constant time add, and constant time deletion. Hash set seemed like a perfect match for the problem at hand as we only needed to know of each member was a set. In python, sets have amortized $O(1)$ runtime for all three of the aforementioned operations[5].

2.2 Algorithm Description

This section gives a detailed analysis of the algorithms used in the graph generation, graph coloring, and backbone selection.

Square Point Generation The algorithm to generate the points in the unit square topology is simple. The steps are as follows:

1. x=random(0, 1), y=random(0, 1)
2. add (x,y) to a list of points
3. repeat steps 1 and 2 until N points are created

That is all that is necesary to get a random point in the unit square. This algorithm is $\Theta(n)$.

Disc Point Generation Generating the points on the disc topology is slightly more challenging than generating the points for the square topology. The algorithm for generating points used in the implementation descriped in this point is as follows:

1. x=random(0, 1), y=random(0,1)
2. if the distance from (x,y) to (.5, .5) is ¡ .5, add (x,y) to a list of points
3. repeat steps 1 and 2 until N points are created

The issue with this algorithm is the nondeterminism involved in creating each point. When generating large numbers of points lots of work is wasted on generating the unused random points. In future iterations of this project a mathematical approach may be used to generated these points. Despite the non-determinism, this algorithm is still amortized $\Theta(n)$.

A possible approach to fixing this implementation would be to pick a radius between 0 and 1, and then rotate by a random number of radians between 0 and 2π. This was not implemented as I was unsure if this would produce a uniform distribution.

This is not pure $O(n)$ as due to the non-determinism it could theoretically run indefinitely. This is however of lower probability than flipping a coin repeateadly and getting heads for all eternity.

Sphere Point Generation Generating points on the surface of the unit sphere can be solved using math. The steps for the following algorithm are as follows:

1. u = (random() * 2) - 1, theta = random() * 2 * pi
2. x = sqrt(1 - u**2) * cos(theta), y = sqrt(1 - u**2) * sin(theta), z = u
3. append (x, y, z) to a list of points.
4. repeat steps 1-3 N times

In the implementation associated with this report the values x, y, and z are transformed to form a sphere around (.5, .5, .5) as opposed to (0, 0, 0) as it makes the connection algorithm simpler. This algorithm is $\Theta(n)$.

2.3 Node Connection

In order to ensure that average degree of the nodes is close to the desired average degree, a radius of a specific length can be defined surrounding each node. The nodes within that radius of each other are then connected in the graph. The

Topology	Radius Equation
Unit Square	$r = sqrt(\frac{d(G)}{N\pi})$
Unit Disc	$r = sqrt(\frac{d(G)}{N})$
Unit Sphere	$r = sqrt(\frac{2 * d(G)}{N})$

formulas to find the radius for each topology is derived from the equations found in the paper Bipartite Grid Partitioning of a Random Geometric Graph[1]. The formula used to find this radius varies for each graph tolopogy and can be found in the table 2.3.

The brute force algorithm for generating and connecting RGGs in this way is $O(n^2)$. This quickly became problematic as the algorithm took upwards of 200 seconds to run on input size of only 12,000. Figure 2.3 shows this issue. Instead, the implementation uses a bucket method to narrow down the required number of comparisons. The idea for this algorithm comes from the paprer Bipartite Grid Partitioning of a Random Geometric Graph[1]. The algorithm can be broken down into steps as follows:

1. create $(\sqrt{1/R} - 1)^2$ buckets
2. place buckets in a 2D grid arrangement
3. for each node, place the node in bucket (x, y) such that x=$floor(x * num_buckets)$, y=$floor(y * num_buckets)$
4. for each x,y such that x=0..buckets, y=0..buckets
5. check to see if each node in bucket (x,y) is within radius R to all others in bucket (x,y) as well as (x+1,y-1), (x+1,y), (x,y+1), (x+1,y+1)
6. if two nodes are under radius R from each other, place an edge between them.

This algorithm is $O(n)$ given a small enough radius and a sufficient number of buckets. This means that with sufficiently many nodes and as long as the expected average degree remains low the algorithm will be linear. The runtimes for the implementation of this algorithm can be found in figure 2.3.

Shortest Last Vertex Ordering In order to color the graph in linear time, the shortest last vertex ordering is used as a heuristic to get a reasonable coloring of the graph. To compute the ordering in $O(V+E)$ time, the following algorithm is used. Graphics are attached to visualize each step of the process.

To begin the graph is uncolored, this can be seen in figure 2.3. The next step is to initialize the ordering of the nodes. To do so, the following data structures are created:

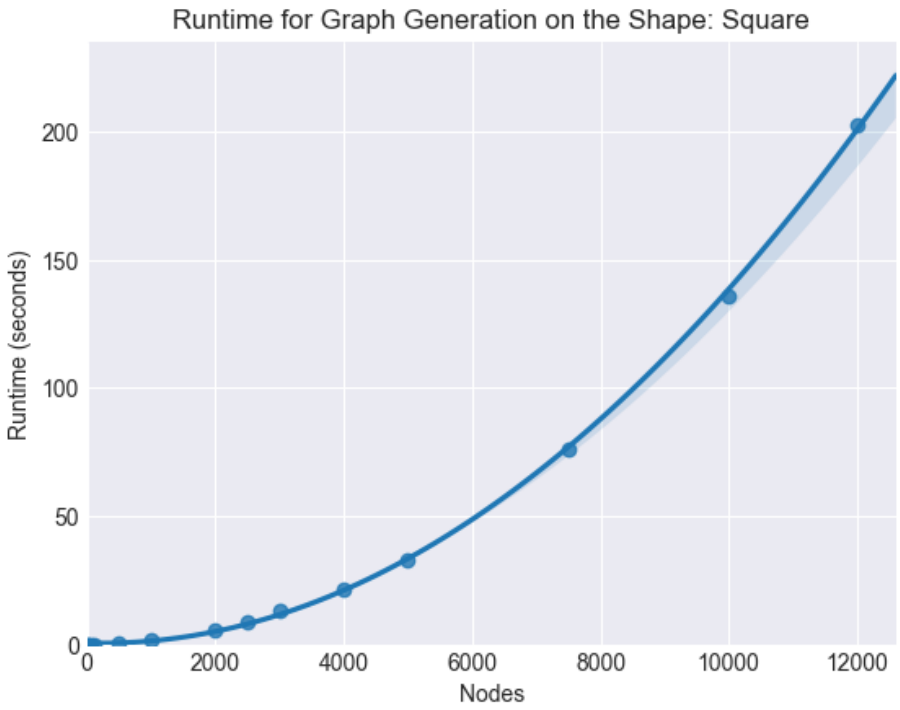

Figure 1: Runtimes of the $O(n^2)$ Algorithm

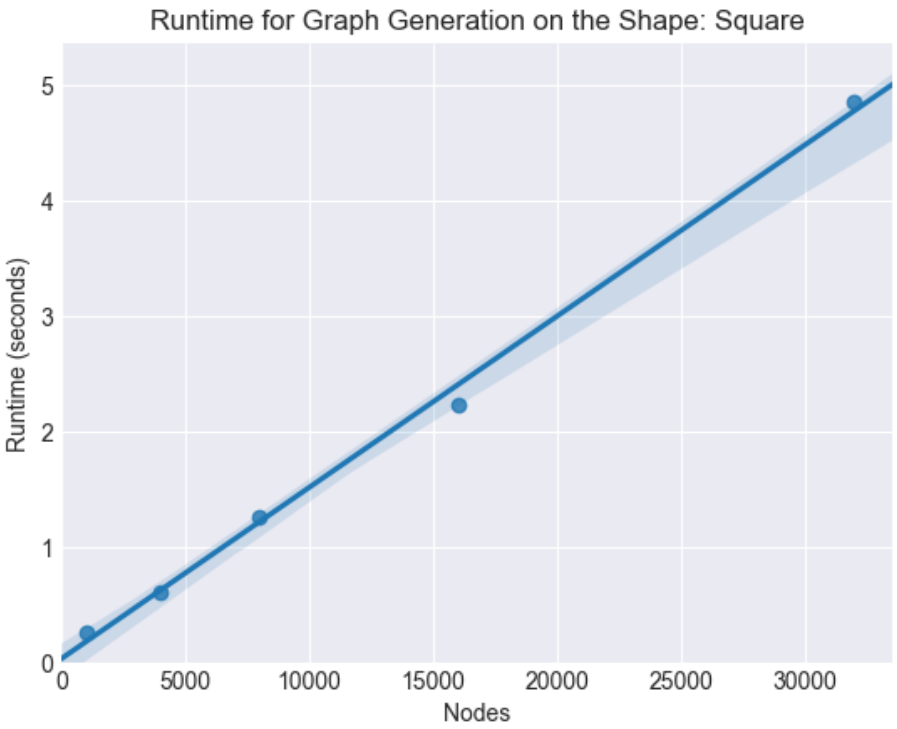

Figure 2: Runtimes of the $O(n)$ Algorithm

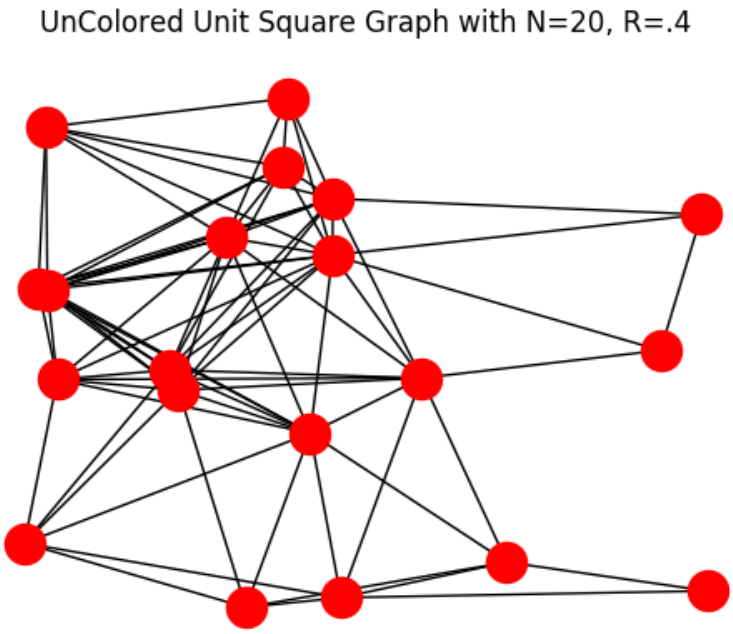

Figure 3: An Uncolored Graph on the square topology of N=20, and R=.4

1. A map mapping from degrees to nodes which are of that degree.
2. Each item mapped to in the previous map should be a set.
3. A map mapping from node numbers to the degree of that node.

Then, the following steps are executed for N iterations:

1. Select a node from a non-empty set mapped to by the smallest possible degree
2. Add this node to the ordering
3. Remove the node from the graph, this entails decrementing the degree of all neighbors and removing the node from the sets above.
4. Move all of the neighboring nodes one hash set lower in the map mapping from degree to nodes

Figure 2.3 shows this process for a graph with 20 nodes. After the ordering is complete, the ordering is then used to color the graph.

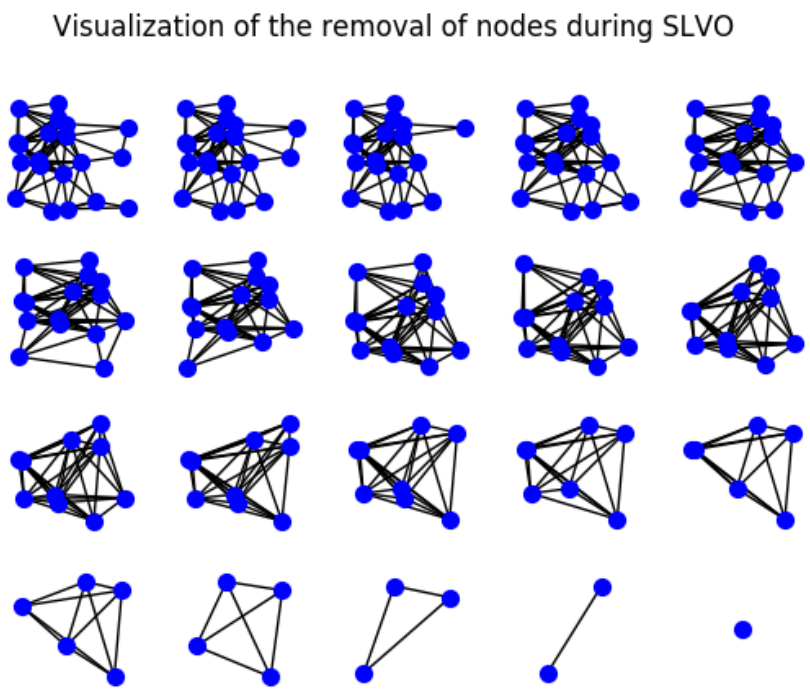

Figure 4: An Example of Shortest Vertex Last Ordering

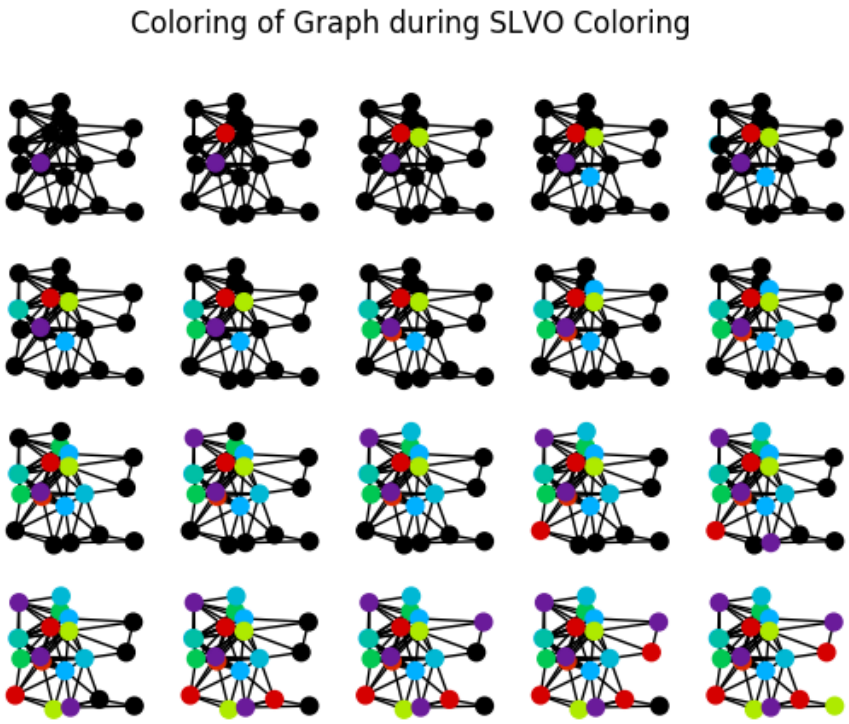

Figure 5: The Process of Coloring the Graph using SVLO

Graph Coloring The coloring process is rather simple. Each node is taken from the ordering and is given the smallest color not yet assigned to any of it's neighbors. This is repeated until all nodes are colored. Figure 2.3 shows this process for the same example above. After this is complete, the graph will be fully colored. Figure 2.3 shows a fully graphed graph.

Backbone Selection After the graph coloring is completed, the next thing done is selecting the communication backbone. This is done by first selecting the four largest color sets produced from the coloring above. Then, the backbone is created for each of the 6 combinations of the four colors. Each backbone consists of each node in the two selected colors. Six backbones from the coloring example above are included in figure 2.3.

In order to compute the domination of the backbones, an empty set is initialized. Then, for each edge in the backbone both nodes are added to the set. After all the edges are traversed the size of the set divided by N is the domination of the backbone.

Figure 2.3 shows the nodes dominated by each of these backbones..

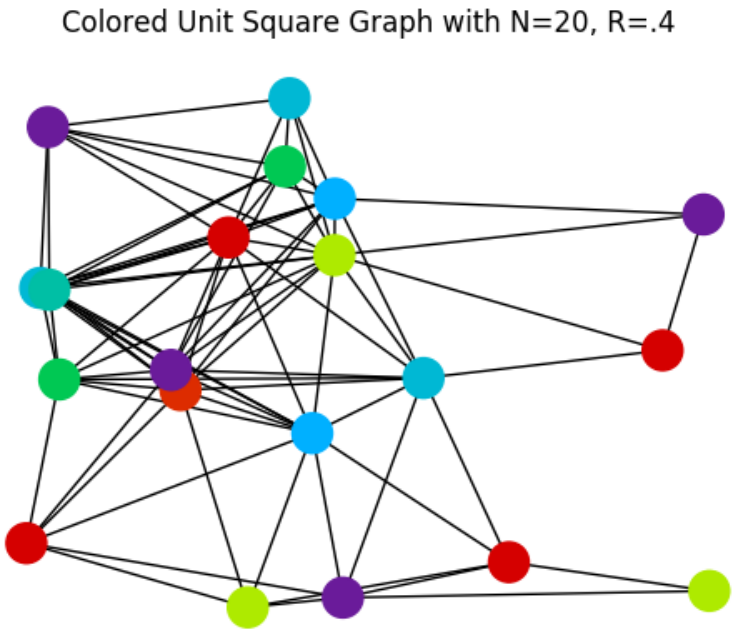

Figure 6: A Fully Colored Graph

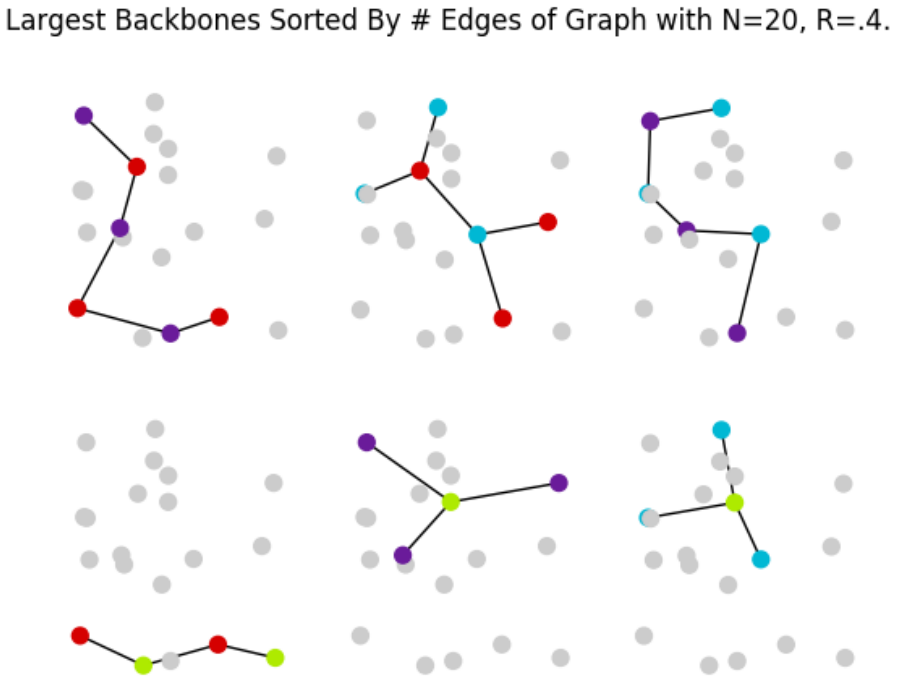

Figure 7: The Six Largest Backbones for a graph with N=20, R=.4

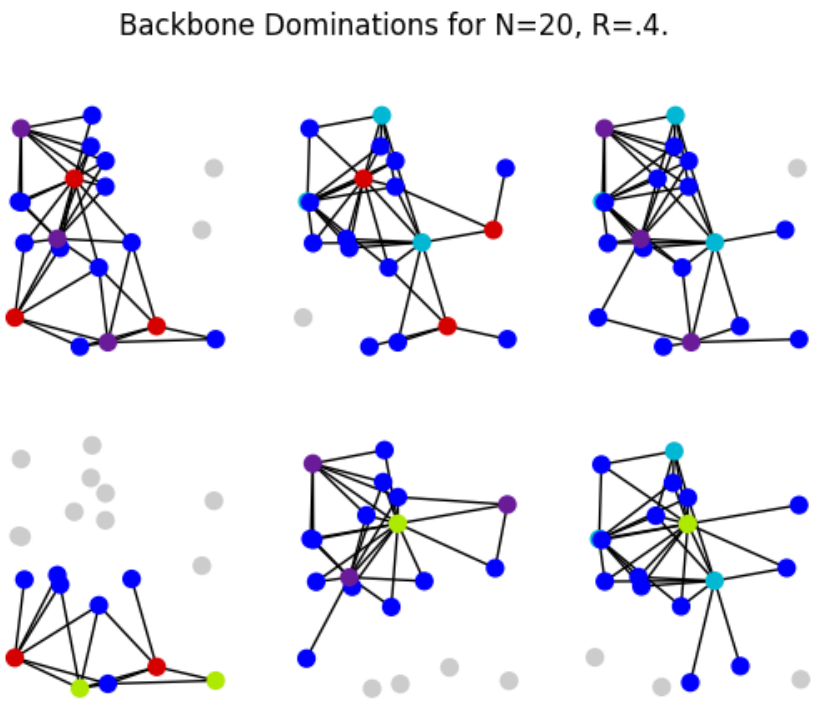

Figure 8: Nodes Dominated by the Six Largest Backbones for a Graph of N=20, R=.4

Nodes	$O(n^2)$	$O(n)$
1000	0.258400	0.657174
2000	0.382116	2.630040
3000	0.473916	5.874501
5000	0.831753	16.809004
10000	1.560216	65.398991

Table 1: Runtimes of the $O(n)$ and $O(n^2)$ algorithms for generating graphs

Requirement	Data Structure	Access Time	Insertion Time	Deletion Time
Degree to set of Nodes	Dictionary	$O(1)$	$O(1)$	$O(1)$
Set of Nodes	Set	$O(1)$	$O(1)$	$O(1)$
Node to Degree	Dictionary	$O(1)$	$O(1)$	$O(1)$
Ordering	Python List(Array)	$O(1)$	$O(1)$	$O(N)$

Table 2: Data Structures Used in the Initialization for SLVO

All of these algorithms can be implemented in linear time trivially given the proper data structure selections.

2.4 Algorithm Engineering

Graph Generation As mentioned above, the brute force algorithm to find adjacent nodes becomes slow at a fast rate. To fix this, the implementation presented uses linear time algorithms for all of it's operations. Table 2.4 compares the runtimes of the linear implementation to that of the quadratic implementation.

The $O(n)$ algorithm is far superior even on small input sizes such as 1000. The final implementation uses $O(n)$ time as well as $O(n * A)$ space where A=aveage edge density. This is accomplished by ensuring that each node is stored in a representation that only references adjacent edges. The implementation also only compares nodes to nodes in adjacent buckets to determine if an edge should be placed between them. Each node only has at most $5nr^2$ comparisons[1]. With reasonably spare graphs this maintains a runtime of $O(n)$ however with large enough radii this can begin to behave like $O(n^2)$. This yields an $O(n)$ time and $O(n)$ space on the type of inputs this implementation targets.

Shortest Last Vertex Ordering The SLVO algorithm used in this paper is an $O(V + E)$ algorithm. This section describes the implementation of the shortest last vertex ordering algorithm described in the previous section. The first step is the initialization. Table 2.4 shows the data structures and access times for the data structures used. All of these runtimes are confirmed by the official python documentation[5]. Please note some of these runtimes are amortized, in the worst case many of these operations are $O(n)$.

After the initialization step, we move into the meat of the algorithm. The

Select a node from a non-empty set mapped to by the smallest possible degree	$O(1)$
Add this node to the ordering	$O(1)$
Remove the node from the graph	$O(1)$
Decrement all neighboring nodes one degree	$O(max(E))$

Table 3: Big O of Each Step

steps are run the same was as described in the algorithm Table 2.4 describes the big O of each step when using the data structures used above.

The steps in the table run V times. While it may look like this is $O(V * E)$ due to the last step being $O(E)$, this is not the case as we only traverse each edge once and then remove the nodes.

Graph Coloring The graph coloring algorithm is also $O(V + E)$. To implement the algorithm, I used the following. First, a map is initialized from nodes to colors. I used a list of size N to get constant time access and update as well as to avoid the overhead required to use a hash table. All of the values in the coloring have an initial color of -1. For each node in the ordering, I select all of the neighbors from the adjacency list. I then select the colors at the index of each neighbor and place them into a hash set. I then try each color from 0 until there is a color not in the hash set and assign that color to the node. Due to the use of the hash set, each color check is $O(1)$.

Overall, this yields $O(V + E)$ runtime.

Backbone Selection The only meaningful data structure choice in the backbone selection is the data structure used to represent the set during domination calculation. A hash set is a great choice for this. The built in set data structure in python has $O(1)$ insertion which is all that is needed for this algorithm. By doing this, all of the algorithms required for backbone selection can run in $O(V + E)$.

2.5 Verification

One way that we verified our results was checking the distribution of edge densities in our graph. We expect to see a gaussian distribution in the edge densities with the center being around our calculated radius. We can also verify the runtime of our algorithms by plotting the input size on the x axis and the runtime on the y axis. If we have a linear algorithm we should be able to fit the distribution of points to a linear equation with minimal error. Both of these verification methods were successful and can be seen in the results section of this report.

Visualizing the Points One way to validate that the points are distributing correctly is by plotting out the points in a scatter plot. Early on the implementation had a bug where the points distributed around the radius of the unit disc

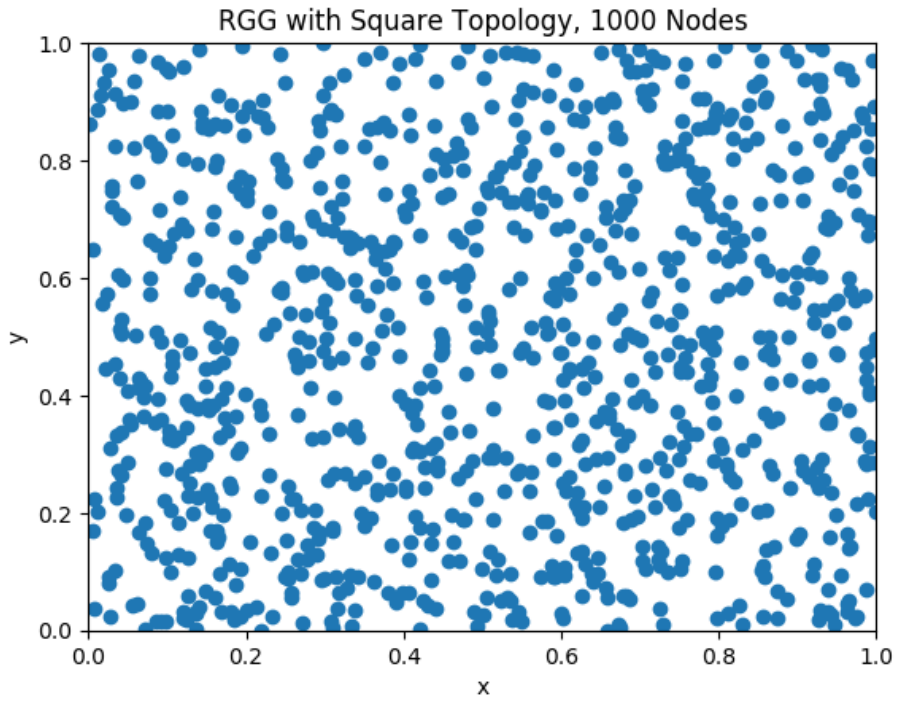

Figure 9: 1000 Points on the Square Topology

instead of evenly inside the unit disc. Using a scatter plot made this trivial to spot.

Figure 2.5 shows the distribution on the square topology, Figure 2.5 shows the distribution on the disc topology, and Figure 2.5 shows the distribution on the sphere topology.

Edge Density Checking the distribution of the edge densities is one way to verify that the edges are connecting as expected The edge distributions should follow a gaussian distribution. Figures 2.5, 2.5, and 2.5 all show gaussian distrubutions for their respective topologies. All of the edge density distributions follow a gaussian distribution which is what is expected meaning that the edges likely connected the nodes in the expected way.

Ordering A plot displaying the degrees of the graph against the degrees of nodes when they are removed from the graph during shortest last vertex ordering can be used to make sure the ordering process is behaving as expected. Figure 2.5 shows the expected behavior. Both of the plots follow a normal distribution. The degrees when removed having a much lower standard deviation along with a lower average. The degrees when the graph is generated have a higher standard

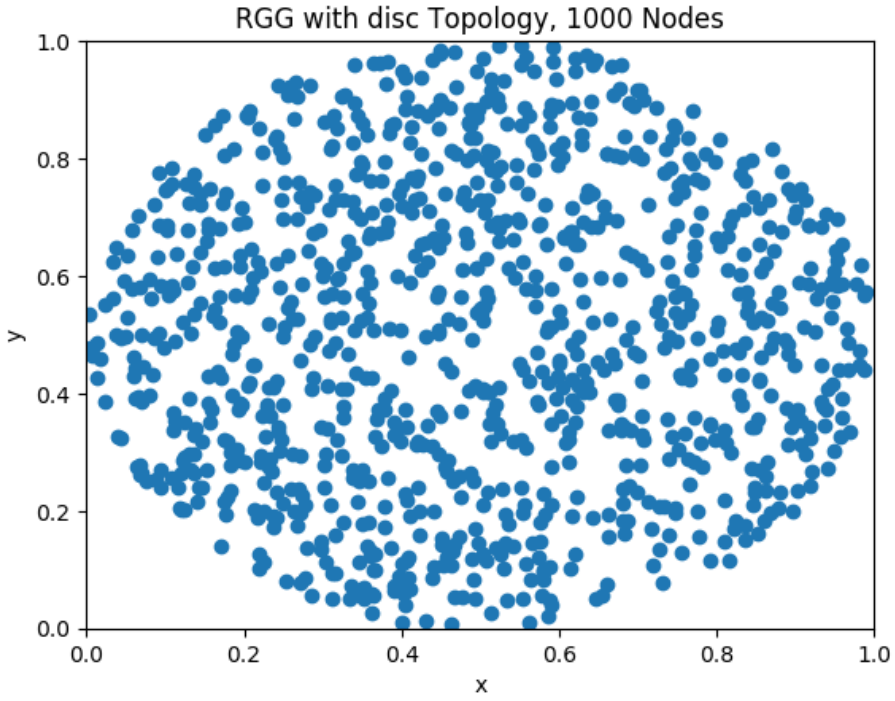

Figure 10: 1000 Points on the Disc Topology

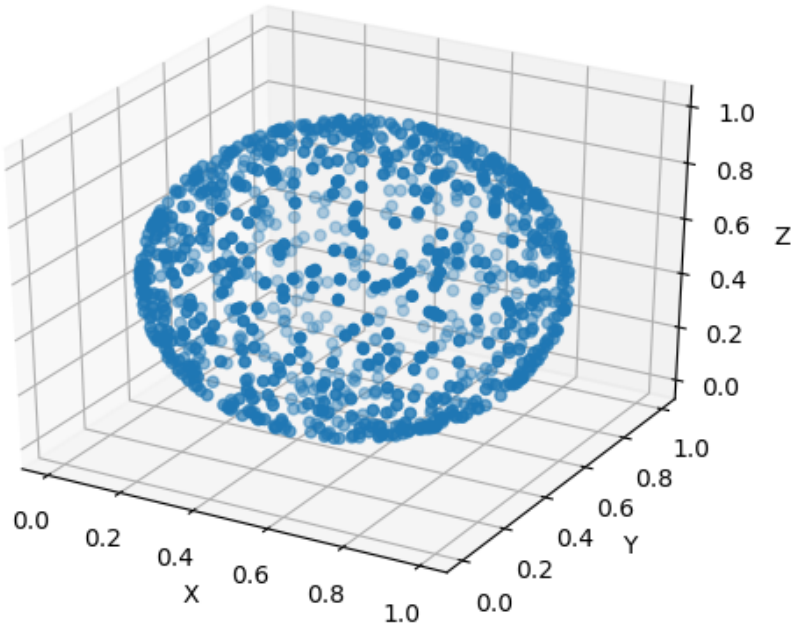

Figure 11: 1000 Points on the Sphere Topology

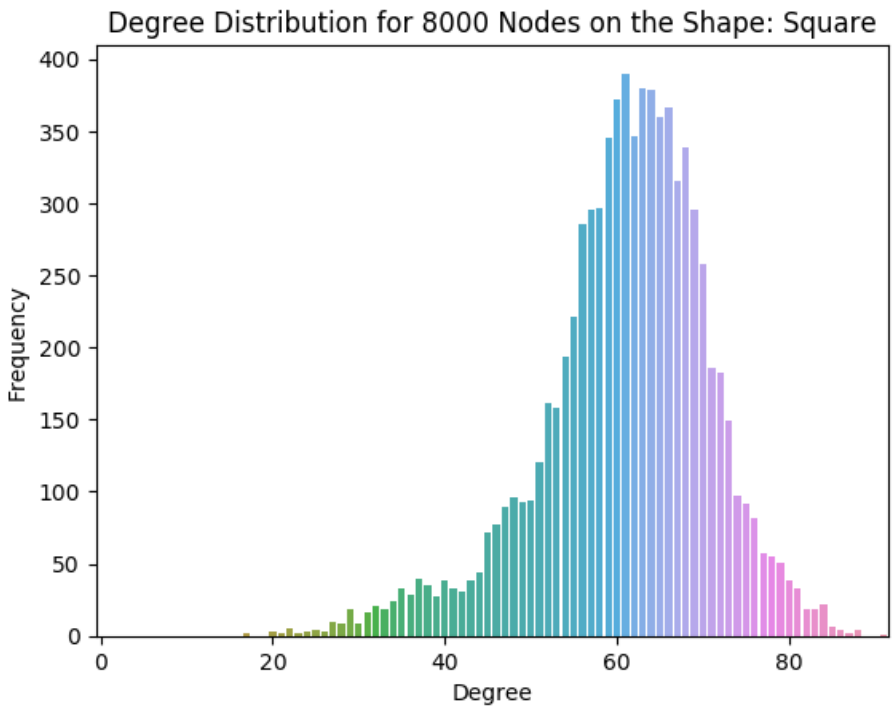

Figure 12: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Square

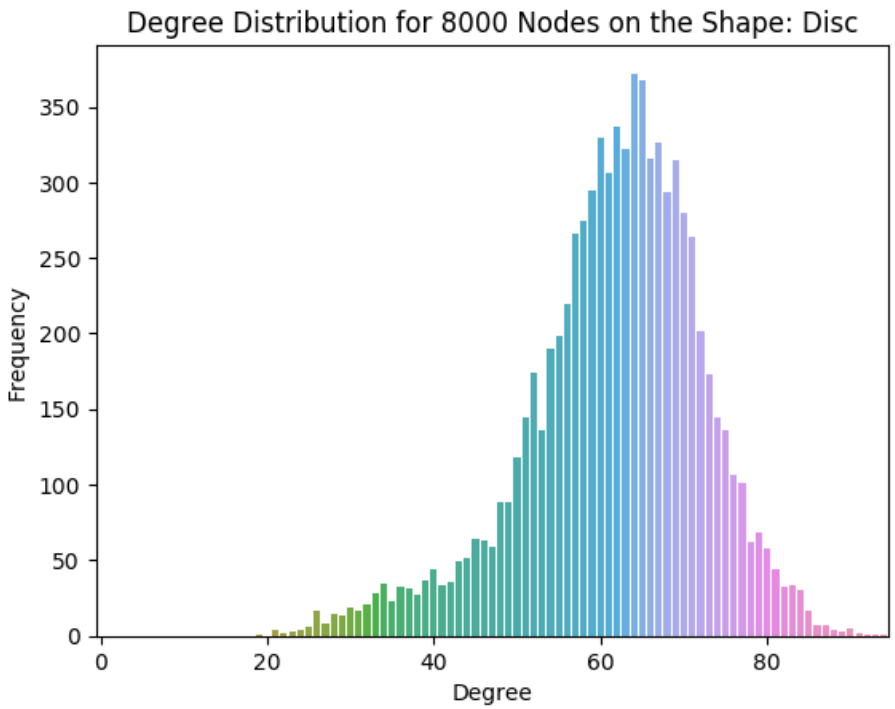

Figure 13: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Disc

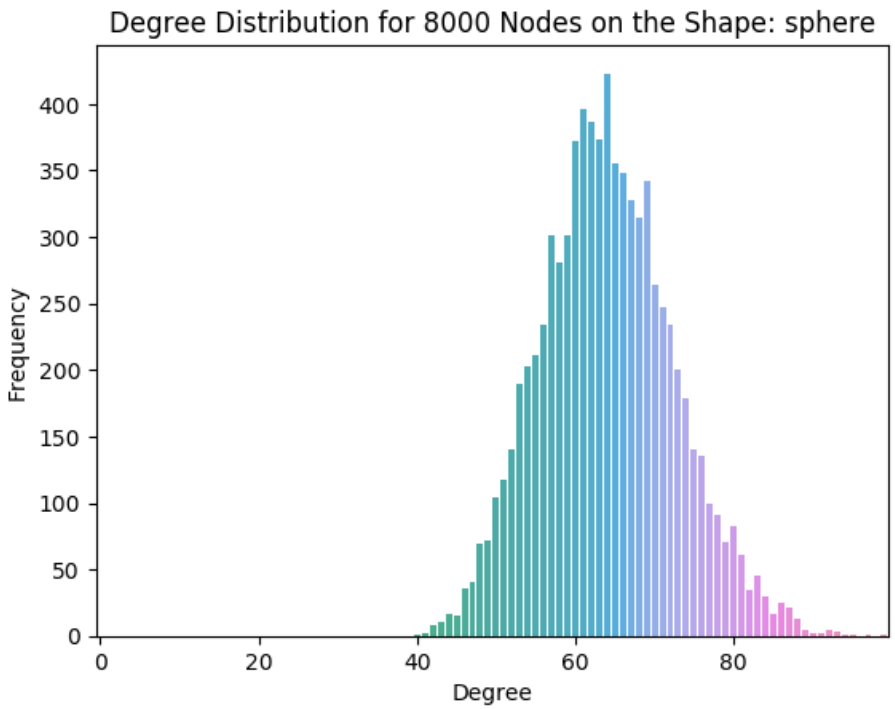

Figure 14: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Sphere

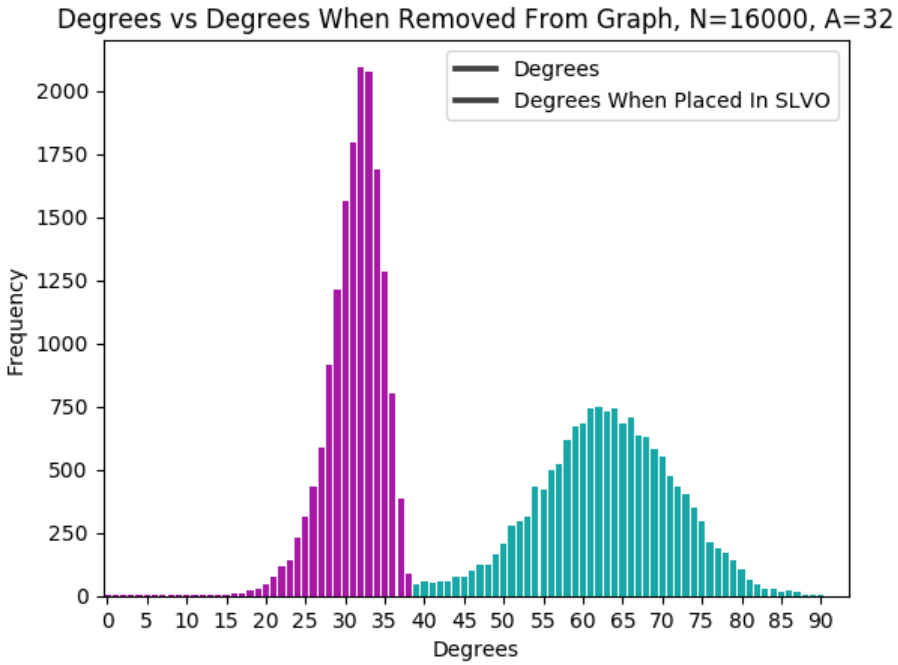

Figure 15: Degree Densities when Nodes are Removed During Shortest Vertex Last Ordering

deviation and a higher average. This is the expected result.

Coloring To verify that the coloring was working properly we can plot the size of the colors in the graph. Figure 2.5 shows the distribution of the size of each color. It makes sense that the earlier colors are used more given that the algorithm assigns the lowest possible color to each node.

Some benchmark data related to the graph coloring is included in table 3.

Shortest Vertex Last Ordering vs Random Ordering In order to further sell the implementation, I have plotted the number of colors used against various values of N for both shortest last vertex ordering and random ordering. Both of these algorithms run in linear time. Figure 2.5 shows this comparison. It is clear that slvo is superior in terms of colors used.

Backbone In order to verify that the backbones were being selected correctly, the top two backbones from each benchmark have been included in this report.

Additionally, some backbone specific benchmark data can be found in table 3.

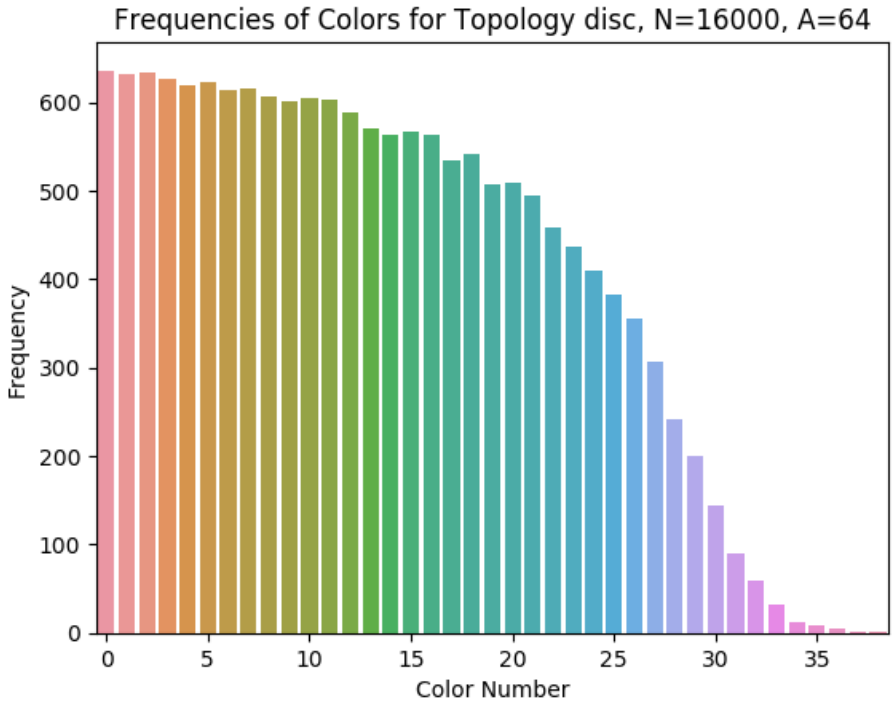

Figure 16: Distribution of the Size of Colors Used

Figure 17: Random Order Coloring vs Shortest Last Vertex ordering

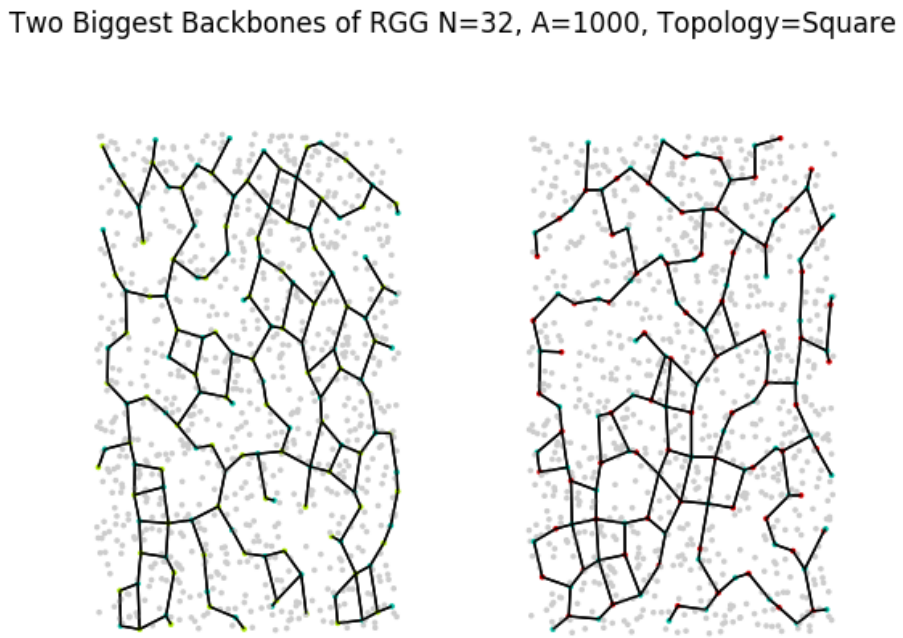

Figure 18: Top Two Backbones, N=1000, A=32, Topology Square

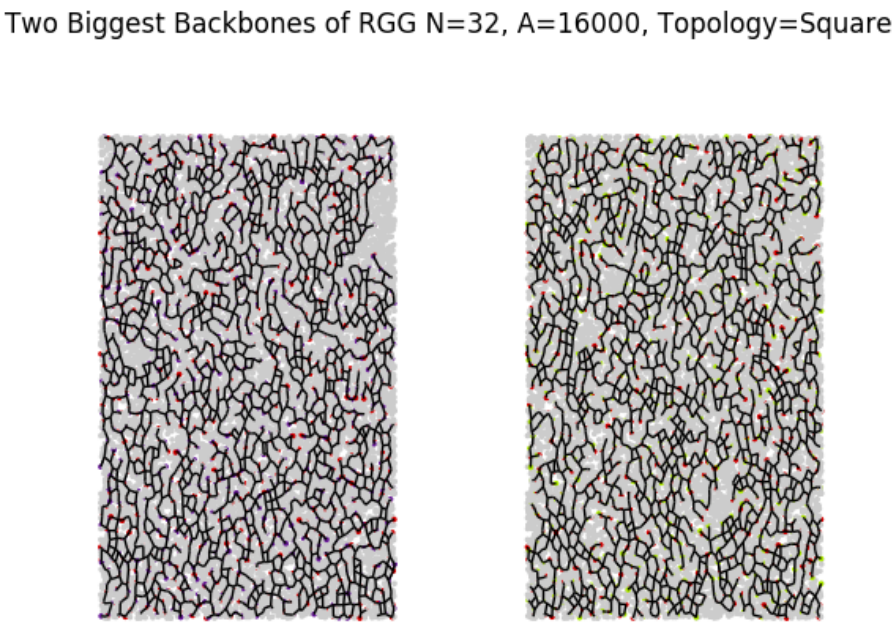

Figure 19: Top Two Backbones, N=16000, A=32, Topology Square

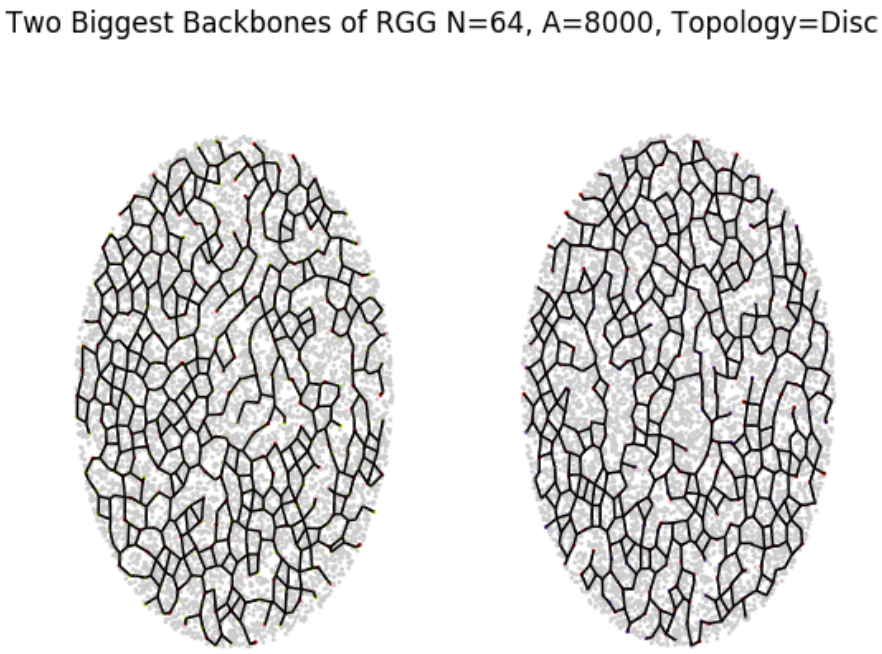

Figure 20: Top Two Backbones, N=8000, A=64, Topology Disc

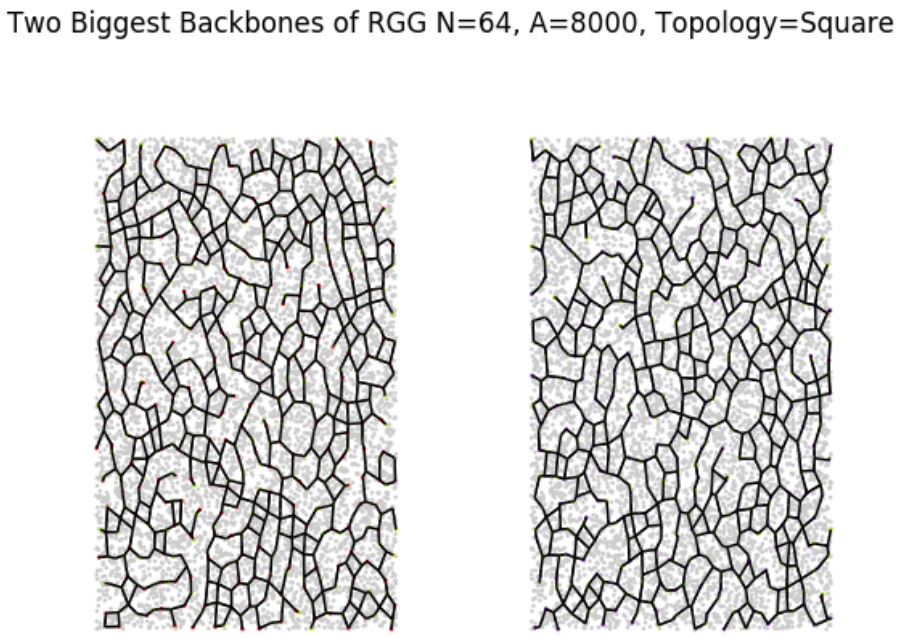

Figure 21: Top Two Backbones, N=8000, A=64, Topology Square

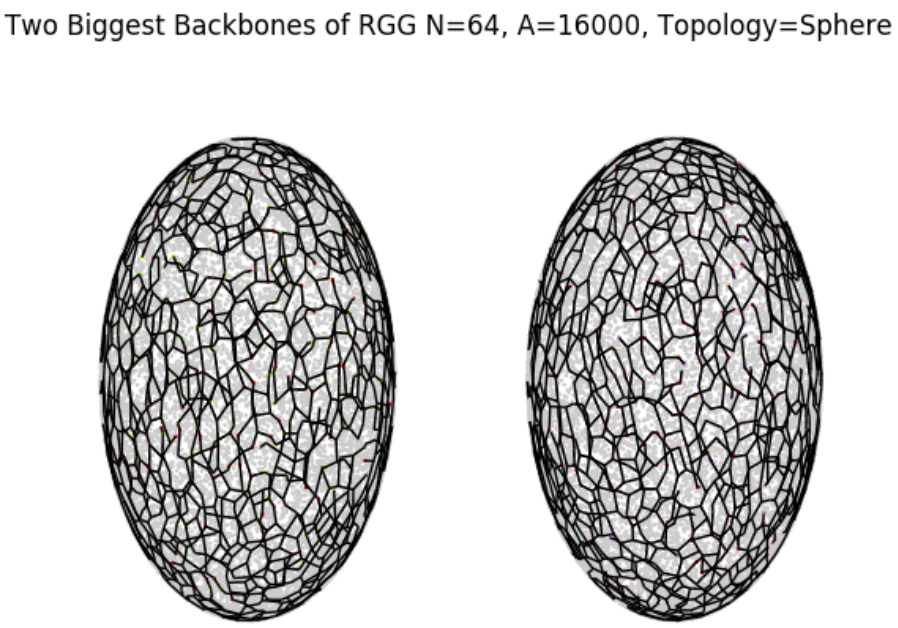

Figure 22: Top Two Backbones, N=16000, A=64, Topology Sphere

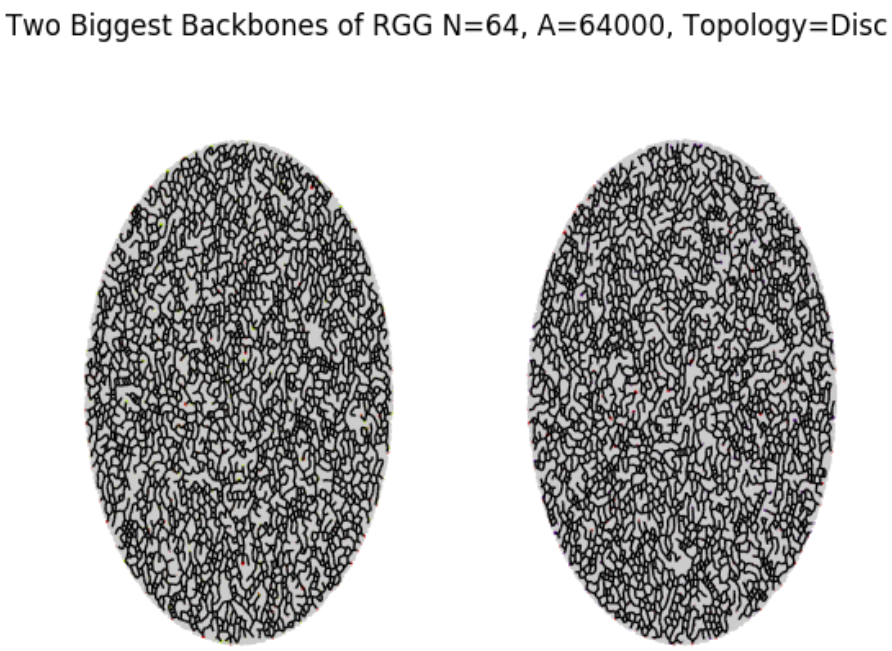

Figure 23: Top Two Backbones, N=64000, A=64, Topology Disc

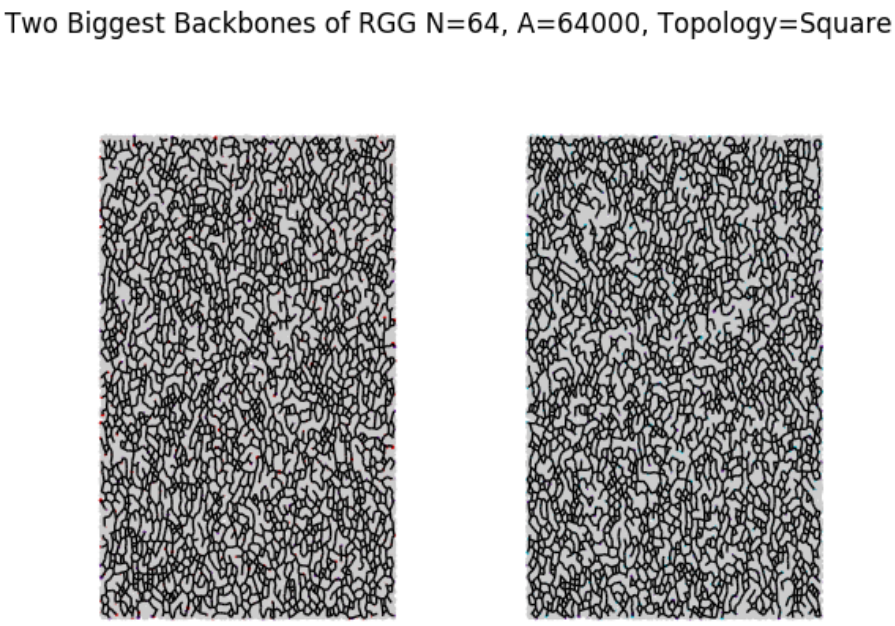

Figure 24: Top Two Backbones, N=64000, A=64, Topology Square

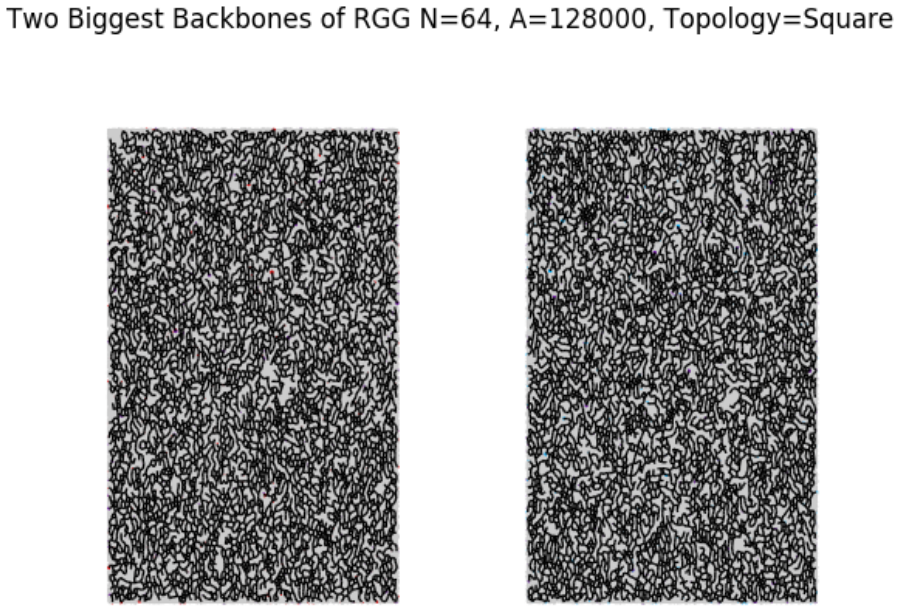

Figure 25: Top Two Backbones, N=128000, A=64, Topology Square

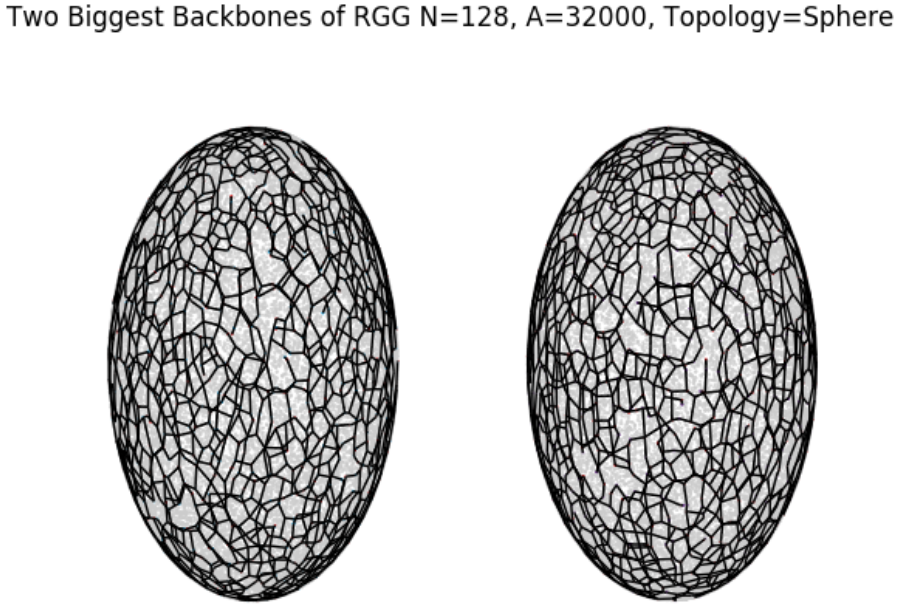

Figure 26: Top Two Backbones, N=32000, A=128, Topology Sphere

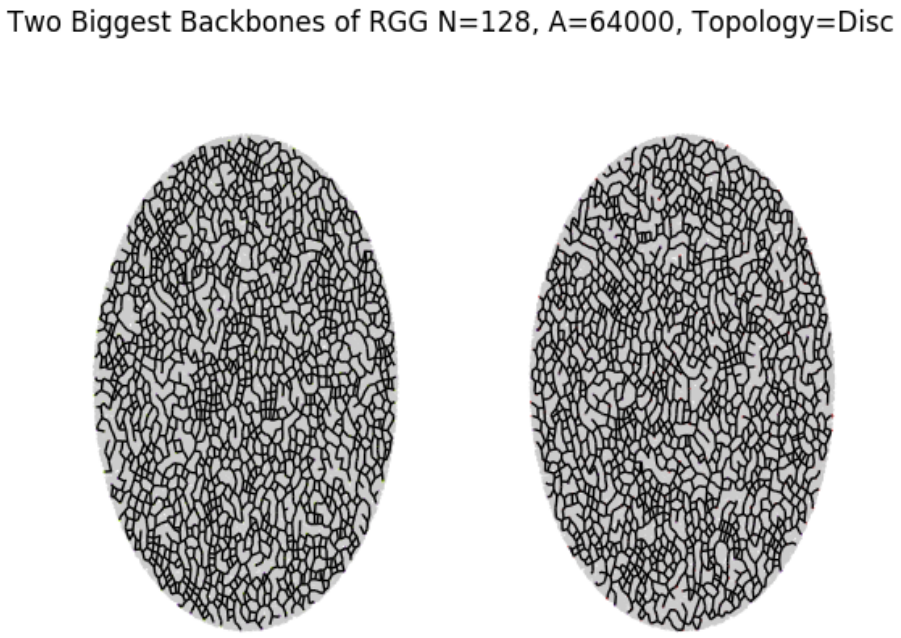

Figure 27: Top Two Backbones, N=64000, A=128, Topology Disc

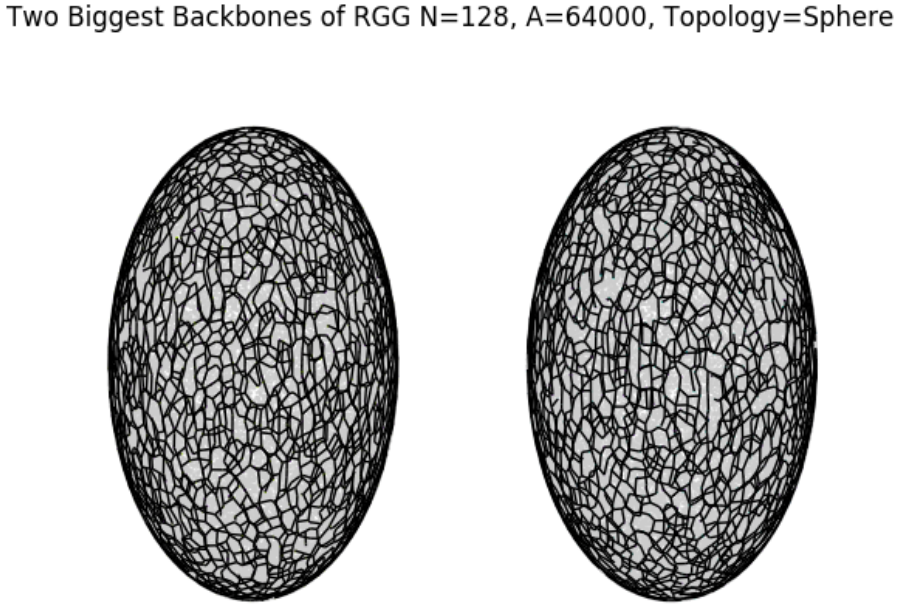

Figure 28: Top Two Backbones, N=64000, A=128, Topology Sphere

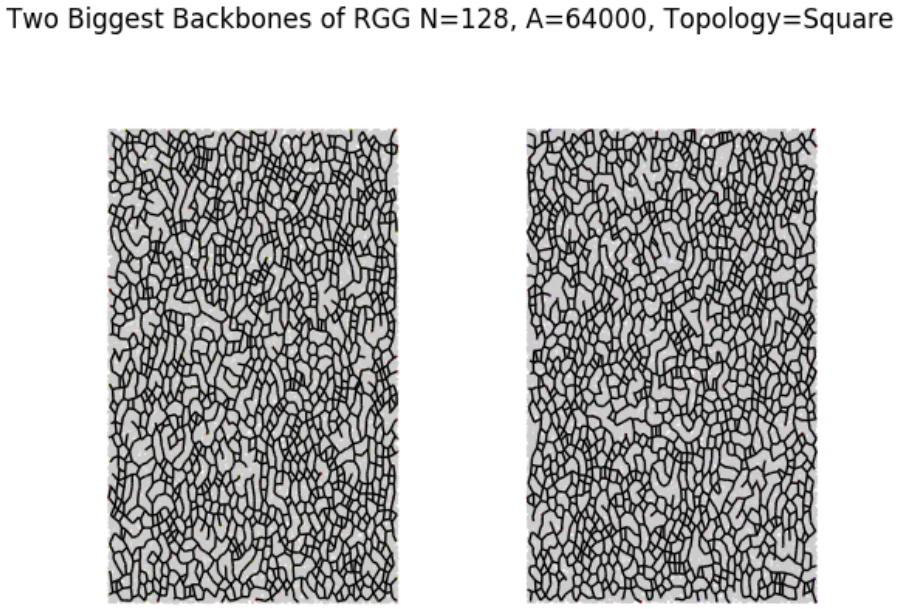

Figure 29: Top Two Backbones, N=64000, A=128, Topology Square

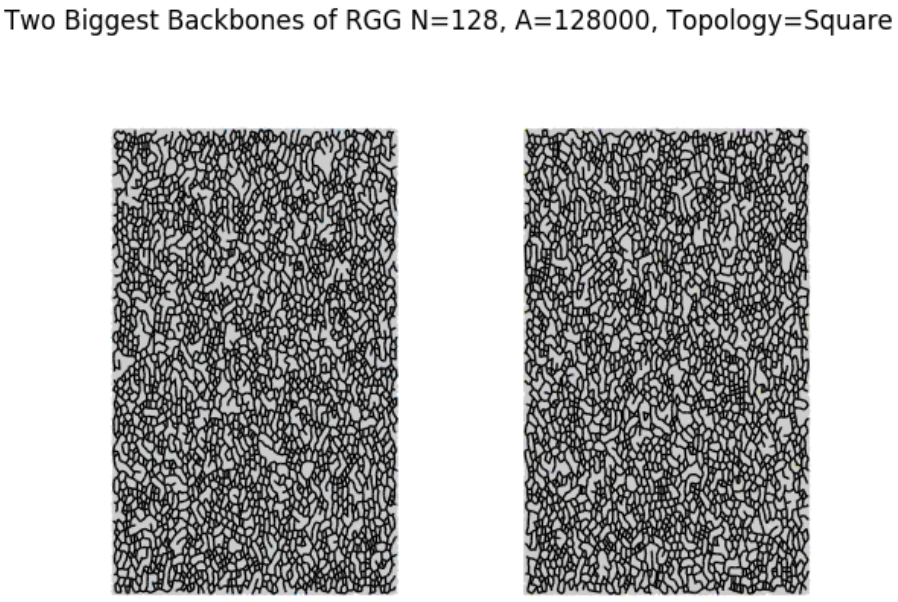

Figure 30: Top Two Backbones, N=128000, A=128, Topology Square

N	A	Square Runtime	Disc Runtime	Sphere Runtime
1000	64	0.444113	0.872433	2.983946
5000	64	1.811515	4.316646	6.846584
10000	64	2.931157	7.800921	10.893038
25000	64	7.639537	22.379140	30.186049
50000	64	16.041881	46.400750	63.725632
100000	64	30.260184	89.923685	125.474381

Table 4: Comparison of Runtimes of Generating the Different Topologies

Benchmark	N	A	Topology	Avg. Degree	Min Degree	Max Degree	Generation Runtime
1	1000	32	Square	28	6	48	0.768379
2	8000	64	Square	61	14	101	10.670343
3	16000	32	Square	31	7	53	10.027430
4	64000	64	Square	62	19	97	77.688297
5	64000	128	Square	125	38	173	154.284222

Table 5: RGG Generation Specific Benchmark Data

3 Result Summary

All of the implementations of algorithms used in this paper run in $O(n)$ time. There was variance between the runtime of the different topologies due to the way the points were generated combined with the calculation required to compute 2d distance vs 3d distance. Table 3 shows the runtime of all of the topologies in a table. Figures 3, 3, and 3 show the individual runtime charts. Figure 3 shows all of the runtimes of the different RGG generation algorithms overlaying each other. As discussed in the verification section, multiple plots have been used to verify the accuracy of the algorithms.

The results for the rgg generation can be found in table 3

The results for the rgg generation can be found in table 3

The results for the rgg generation can be found in table 3

Benchmark	N	A	Topology	Average Degree When Removed	Colors	Largest Color	Coloring Runtime
1	1000	32	Square	19	19	77	0.010196
2	8000	64	Square	41	38	325	0.190919
3	16000	32	Square	24	23	1140	0.198154
4	64000	64	Square	41	39	2539	1.556953
5	64000	128	Square	72	66	1370	2.983963

Table 6: Coloring Specific Benchmark Data

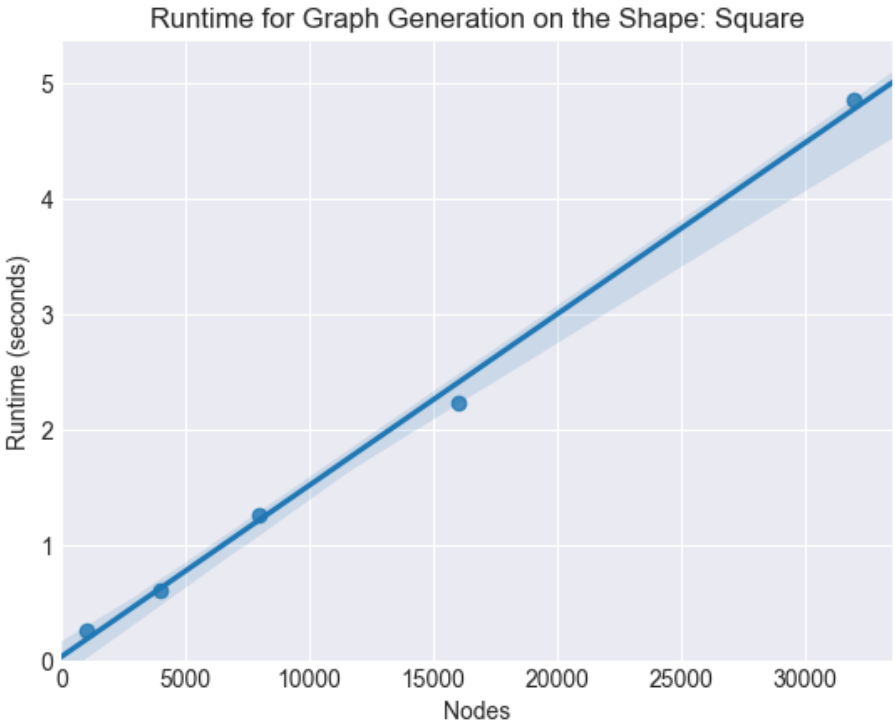

Figure 31: Runtimes of the $O(n)$ Square RGG Generation Algorithm

Benchmark	N	A	Topology	Backbone Order	Backbone Size	Backbone Domination	Number of Backbone Faces	Backbone Runtime
1	1000	32	Square	356	149	0.992000	N/A	0.002872
2	8000	64	Square	1670	639	0.997125	N/A	0.031183
3	16000	32	Square	5370	2216	0.986938	N/A	0.056506
4	64000	64	Square	13046	5027	0.998875	N/A	0.245150
5	64000	128	Square	7524	2725	0.999609	N/A	0.395333

Table 7: Backbone Specific Benchmark Data

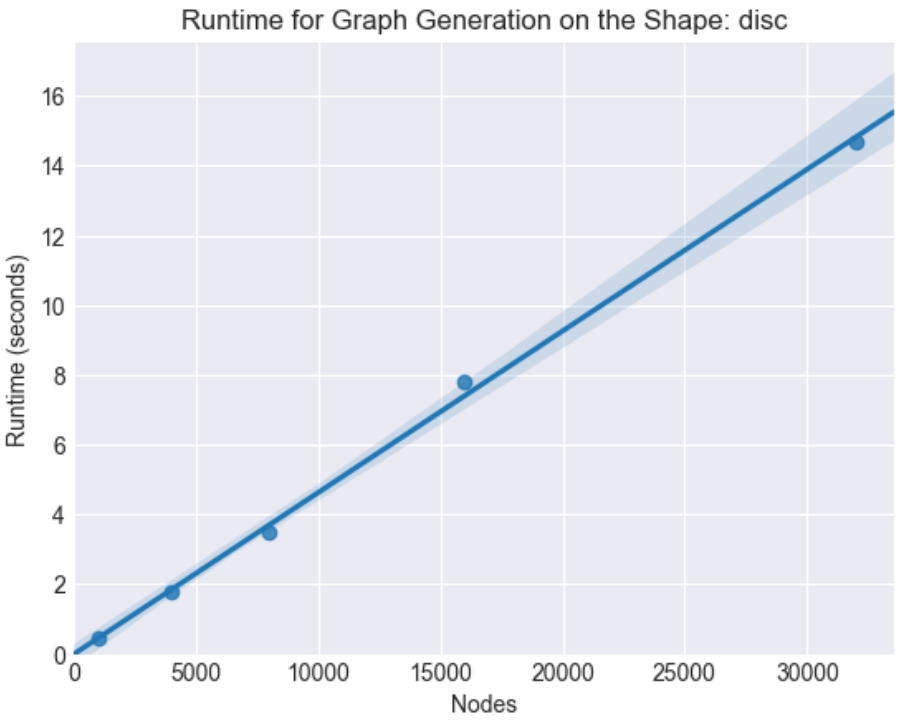

Figure 32: Runtimes of the $O(n)$ Disc RGG Generation Algorithm

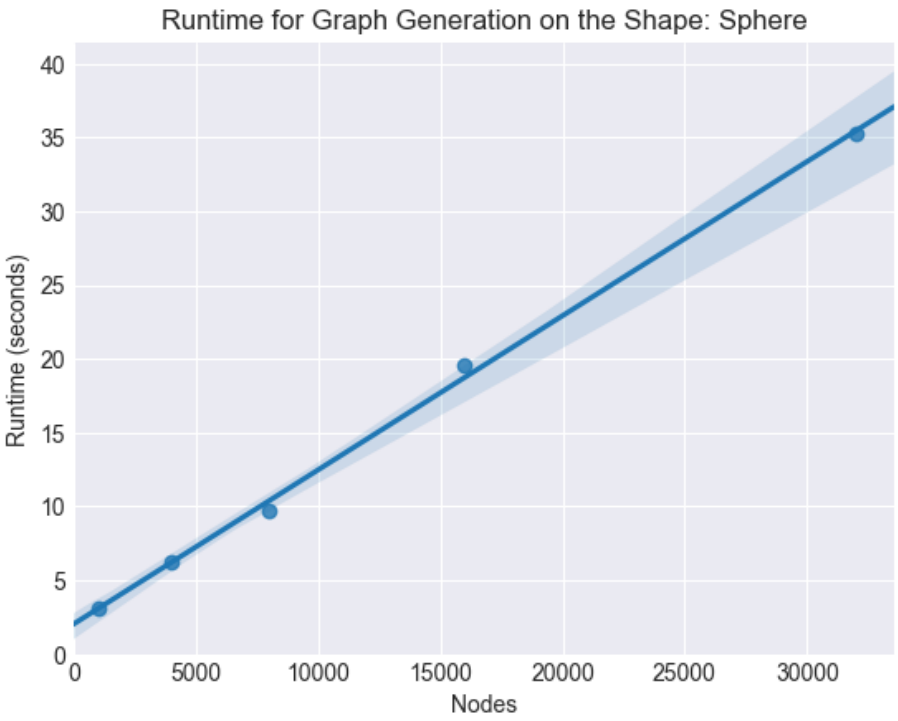

Figure 33: Runtimes of the $O(n)$ Sphere RGG Generation Algorithm

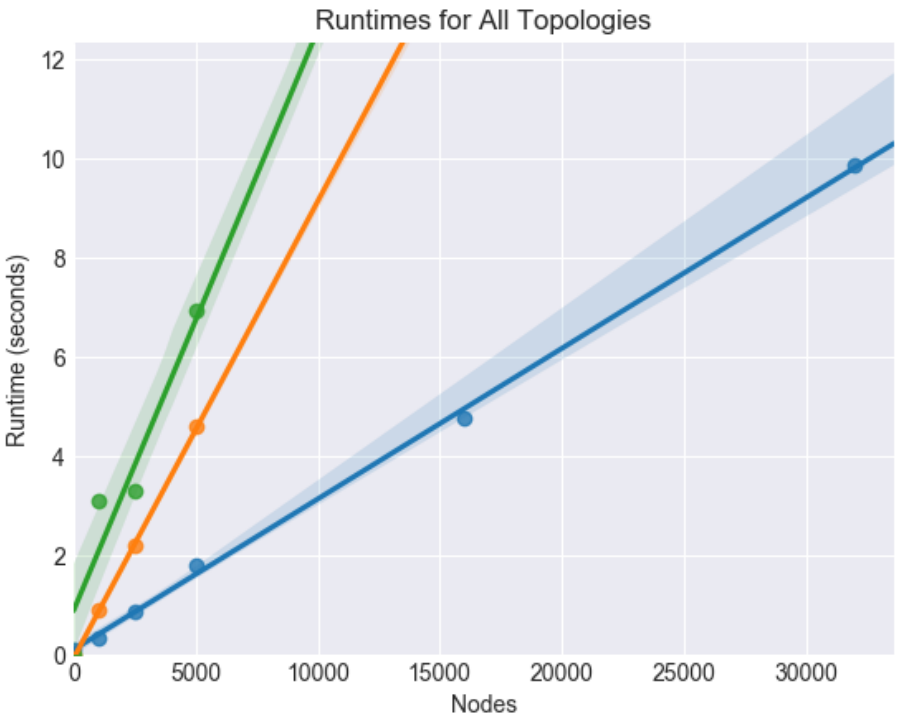

Figure 34: Runtimes of the $O(n)$ RGG Generation Algorithms overlayed

References

[1] Zizhen Chen and David W Matula. "Bipartite Grid Partitioning of a Random Geometric Graph". In: *2017 13th International Conference on Distributed Computing in Sensor Systems (DCOSS)*. IEEE. 2017, pp. 163–169.

[2] Hichem Kenniche and Vlady Ravelomananana. "Random geometric graphs as model of wireless sensor networks". In: *Computer and Automation Engineering (ICCAE), 2010 The 2nd International Conference on*. Vol. 4. IEEE. 2010, pp. 103–107.

[3] Dhia Mahjoub and David W Matula. "Employing (1- ε) dominating set partitions as backbones in wireless sensor networks". In: *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Society for Industrial and Applied Mathematics. 2010, pp. 98–112.

[4] David W Matula and Leland L Beck. "Smallest-last ordering and clustering and graph coloring algorithms". In: *Journal of the ACM (JACM)* 30.3 (1983), pp. 417–427.

[5] PythonWiki. *TimeComplexity - Python Wiki*. 2017. URL: `https://wiki.python.org/moin/TimeComplexity` (visited on 06/05/2017).