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

Luke Wood

March 27, 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[**mahjoub2010employing**]. 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. In this project I will use RGGs as a way to gather valuable information about how wireless sensor networks will possibly function and communicate.

1.1 Introduction and Summary

Through a series of three reports I will be analyzing 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[**matula1983smallest**]
3. Find the terminal clique in the generated RGGs
4. Find a selection of bipartite subgraphs producted by an algorithm for coloring

This report is the first of the series and will describe an implementation for a linear time algorithm to generate graphs consisting of N vertices with an average degree of A on the geometric topologies of: unit square, unit disc, unit sphere. Some tables are included to display some metrics for generated RGGs of various size to display the performance of the generation algorithm. The square

Nodes	E(Avg. Deg)	Avg. Deg	Max Deg.	Min Deg.	Seconds
1000	64	57.426000	86	15	0.247774
5000	64	60.332400	90	13	0.833398
25000	64	62.563600	95	19	3.624590
50000	64	62.852480	104	15	7.565354
100000	64	63.317680	101	16	15.464250

Table 1: Data on Graphs Generated with the Square Topology

Nodes	E(Avg. Deg)	Avg. Deg	Max Deg.	Min Deg.	Seconds
1000	64	57.142000	85	19	0.477228
5000	64	60.569600	86	22	2.561189
25000	64	62.607760	94	20	12.667523
50000	64	63.050840	96	18	22.993427
100000	64	63.241720	97	20	46.860248

Table 2: Data on Graphs Generated with the Disc Topology

topology generation algorithm runs in approximately 15 seconds for graph size 100000 which is fast given the large input size. Reference table **??** for in depth information on this algorithm.

The disc topology also runs in linear time but the constant factor is much larger than that of the square topology. This is due to some overhead that I incurred when generating the points themselves. Despite the algorithm being slower than the square version, it still terminated in 48 seconds for 100000 nodes. At the moment this is the worst generation algorithm and could be improved by using mathematical formulas instead of repeated guessing to generate points. Reference table **??** for in depth information on this algorithm.

Sphere generation was also a linear time algorithm. Unlike the disc topology I generated the points using a mathematical formula instead of brute force finding points. Thie yielded a runtime of under 35 seconds for generating 32000 points. Reference table **??** for in depth information on this algorithm.

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

Nodes	E(Avg. Deg)	Avg. Deg	Max Deg.	Min Deg.	Seconds
1000	64	77.984000	138	41	2.806046
4000	64	63.985000	92	37	5.296109
8000	64	63.966000	95	37	8.456132
16000	64	63.948000	94	35	18.688274
32000	64	64.007937	95	33	34.921683

Table 3: Data on Graphs Generated with the Sphere Topology

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.

Going forwards I will continue to implement linear time algorithms to solve the problems at hand. I will possible move from python to the Elixir programming environment to utilize the high level of parralelism and the huge speed gain from type inference.

Currently, my implementation has support for:

1. Generating a RGG with a unit square topology
2. Generating a RGG with a unit disc topology
3. Generating a RGG with a unit sphere topology
4. Converting RGGs to an Adjacency List
5. Serializing Adjacency Lists to files

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 graph generation is written in python 3 as generating and connection a graph is not super computationally expensive with even decently large inputs such as 100000 (assuming $O(n)algorithms$). The later algorithms may be implemented in a different language such as Elixir to get high levels of concurrency and higher efficiency due to type inference (as opposed to python's dynamic typing).

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. If the goal were to generate gigantic graphs then an argument

Algorithm	Ω	O
Square Point Generation	$\Omega(n)$	$O(n)$
Disc Point Generation	$\Omega(n)$	$O(n)$
Sphere Point Generation	$\Omega(n)$	$O(n)$
Node Connection	$\Omega(n)$	$O(n)$

Table 4: big O and big Ω of all of the algorithms described

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 mentioned data structure 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. This is handy in situations where the expected average number of edges is significantly lower than the number of nodes in the graph. 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. This takes $O(e)$ in an adjacency list as opposed to $O(1)$ in an adjacency matrix.

2.2 Algorithm Description

This section gives a detailed analysis of the algorithms used in the graph generation implementation. Table **??** contains information on the asymptotic runtimes of the algorithms in terms of O and Ω.

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

Topology	Equation Used to Derive Radius	Radius Equation
Unit Square	$d(G) \approx N\pi r^2$	$r = sqrt(\frac{d(G)}{N\pi})$
Unit Disc	$r = sqrt(\frac{d(G)}{N\pi})$	$r = sqrt(\frac{d(G)}{N})/2$
Unit Sphere	$r = sqrt(\frac{d(G)}{N})/2$	$r = sqrt(\frac{d(G)}{N}) * 2/3$

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.

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 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 **??**.

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 **??** shows this issue. Instead, the implementation uses a bucket method to narrow down the required

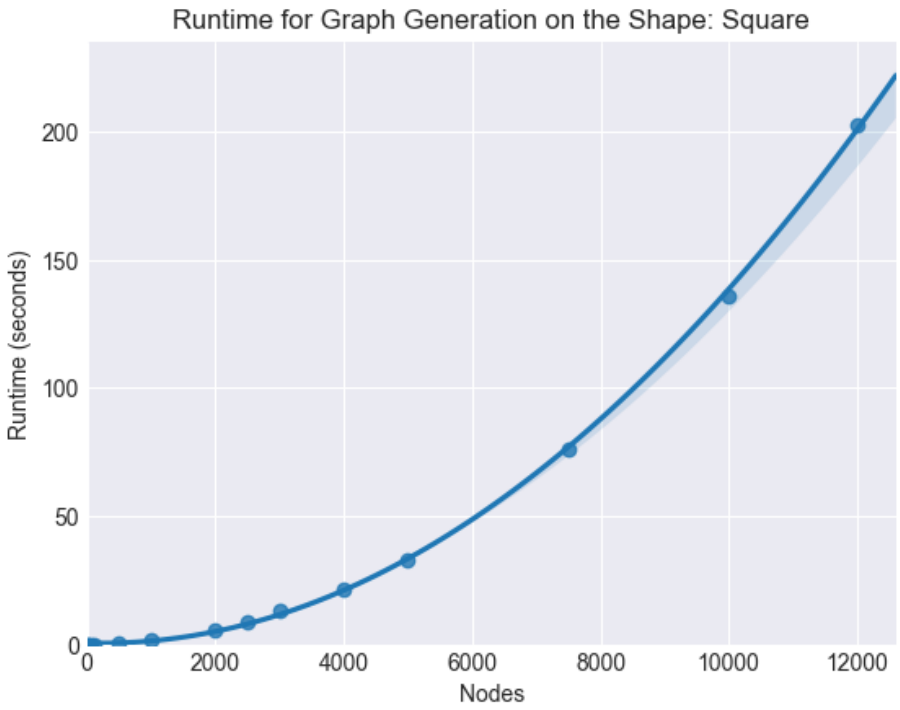

Figure 1: Runtimes of the $O(n^2)$ Algorithm

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 **??**.

Conversion From Node List to Adjacency Matrix The algorithm to convert from a node list to an adjacency matrix is as follows:

1. map each node to it's edge list attribute
2. map each edge list to the contained nodes' node numbers

This quickly yields an $O(v*e)$ algorithm to change my node list to an adjacenecy matrix. If the performance of $O(v * e)$ is deemed unacceptable in the future, then we can simply append the node number to the list of edges as opposed to a pointer to the node object deeming the second map operation unneccesary yieling an $O(v)$ algorithm.

2.4 Algorithm Engineering

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 **??** 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 $5nr^2$

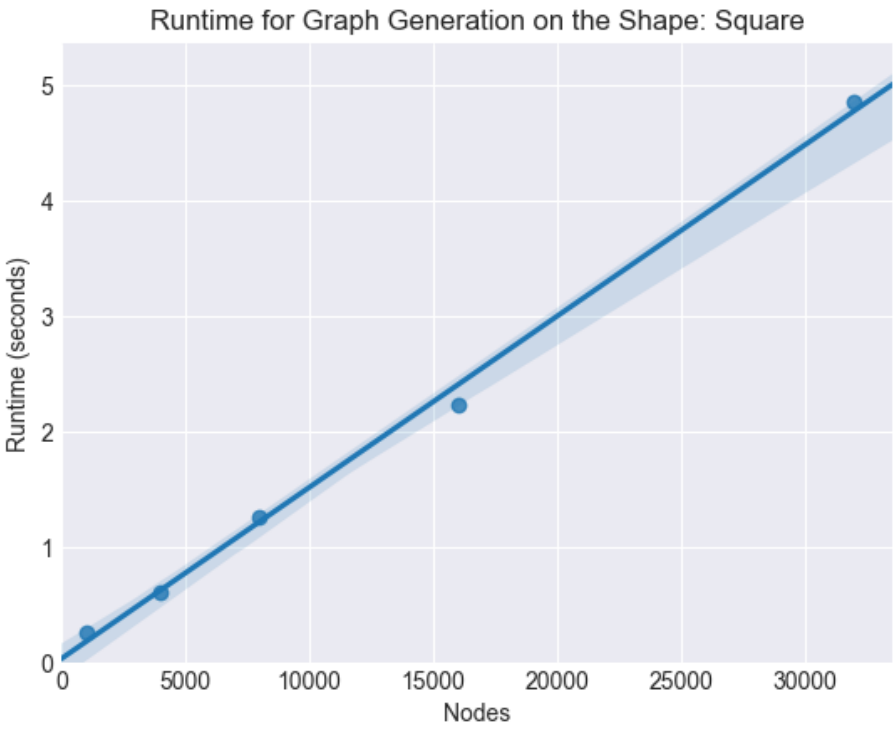

Figure 2: Runtimes of the $O(n)$ Algorithm

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 5: Runtimes of the $O(n)$ and $O(n^2)$ algorithms in seconds

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.

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

Figure **??** shows the distribution on the square topology, Figure **??** shows the distribution on the disc topology, and Figure **??** 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 **??**, **??**, and **??** 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.

3 Result Summary

All of my algorithms ran 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 **??** shows the runtime of all of the topologies in a table. Figures **??**, **??**, and **??** show the individual runtime charts. Figure **??** 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.

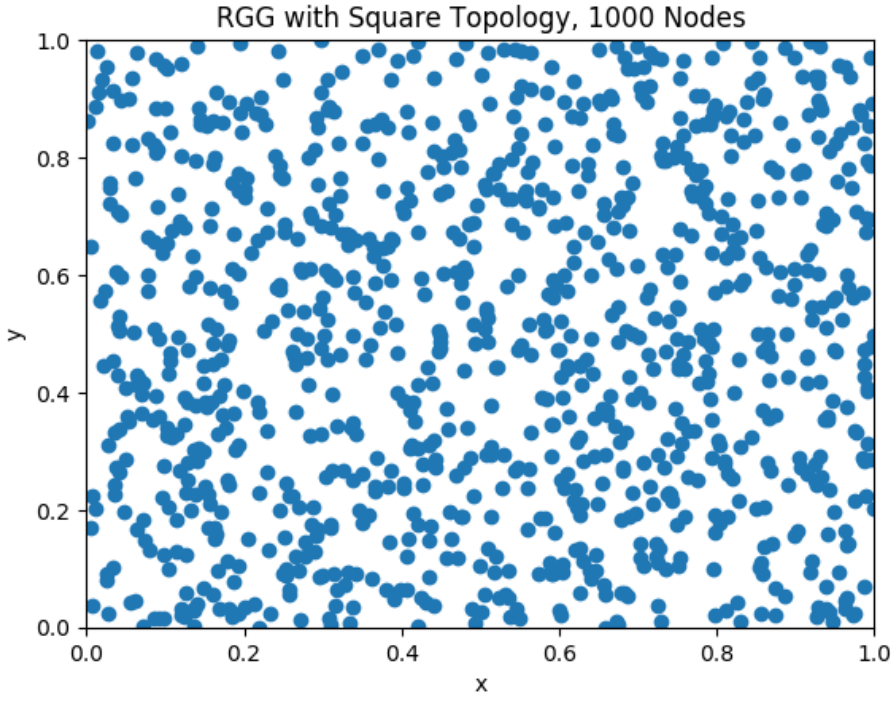

Figure 3: 1000 Points on the Square Topology

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 6: Comparison of Runtimes of Generating the Different Topologies

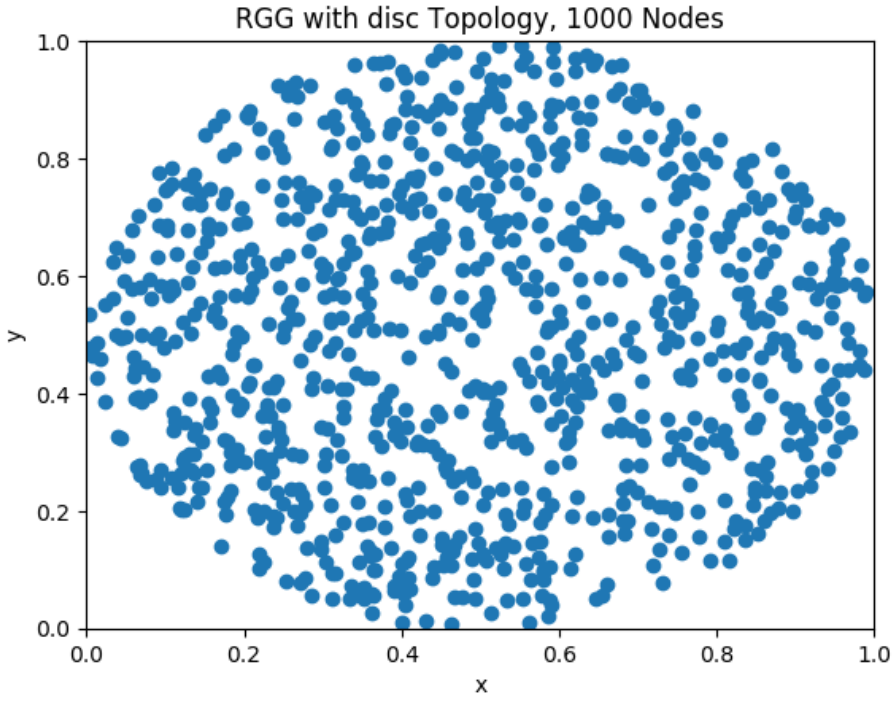

Figure 4: 1000 Points on the Disc Topology

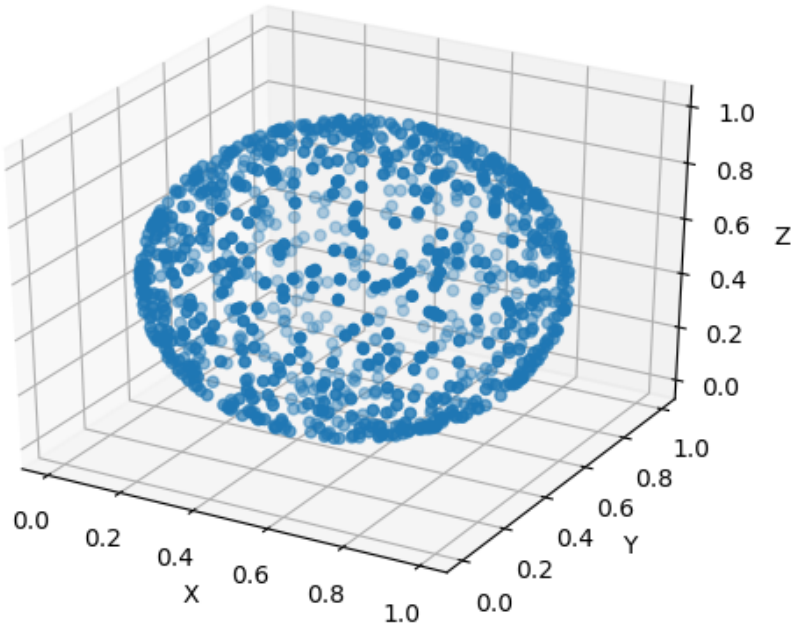

Figure 5: 1000 Points on the Sphere Topology

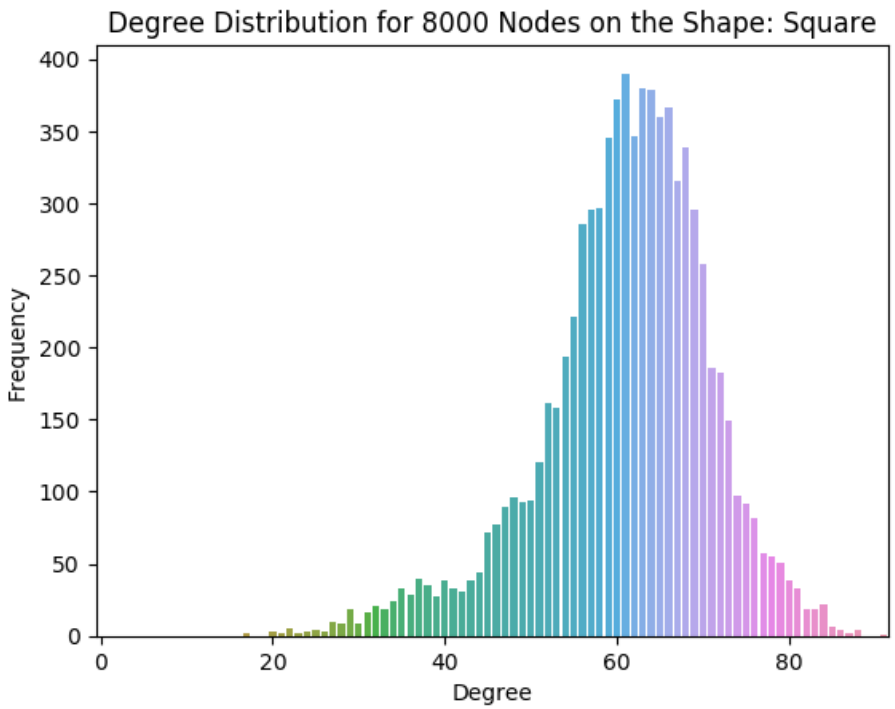

Figure 6: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Square

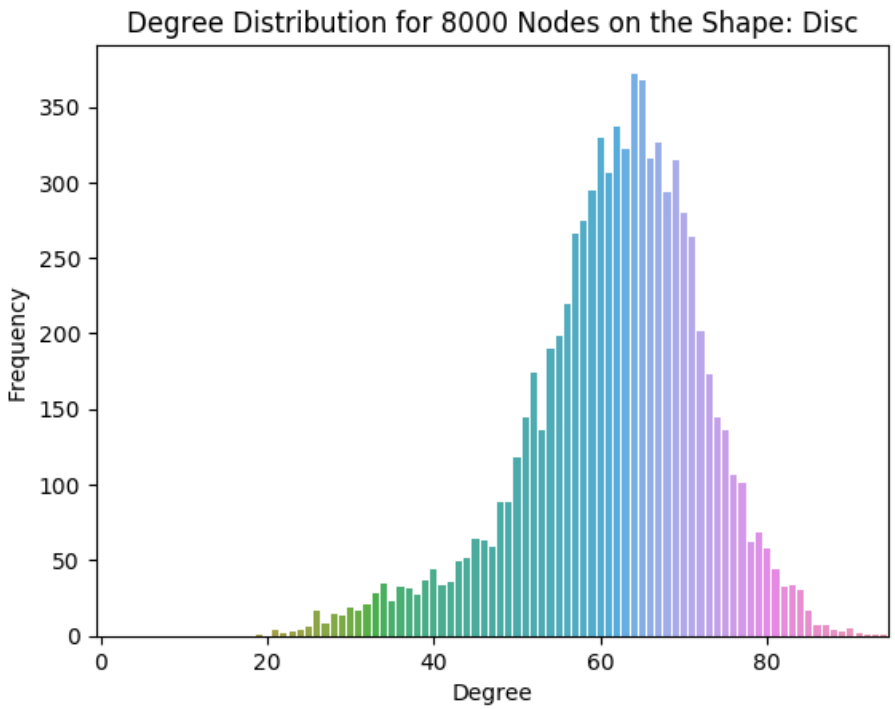

Figure 7: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Disc

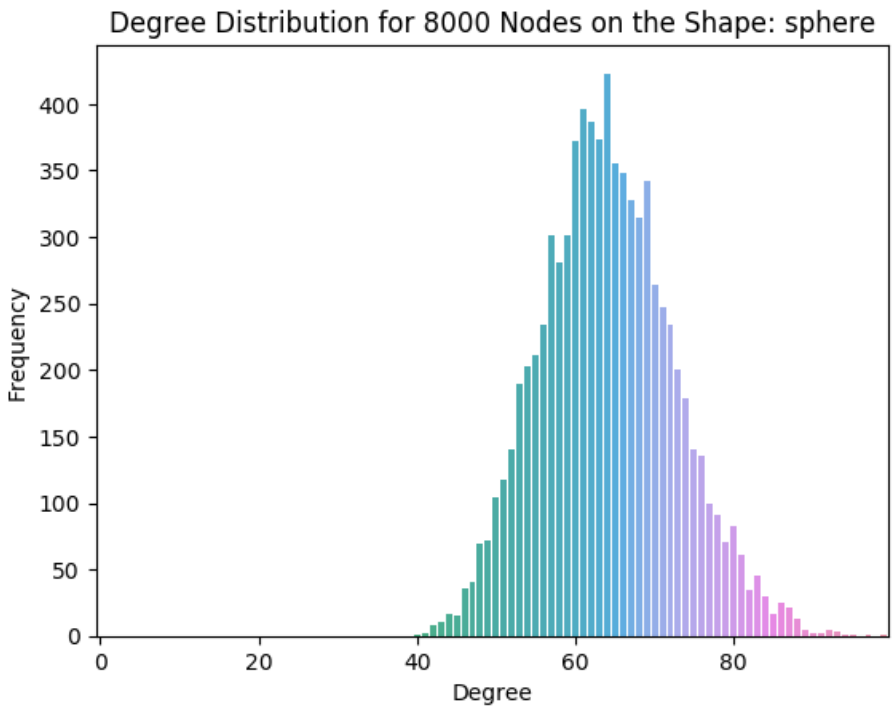

Figure 8: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Sphere

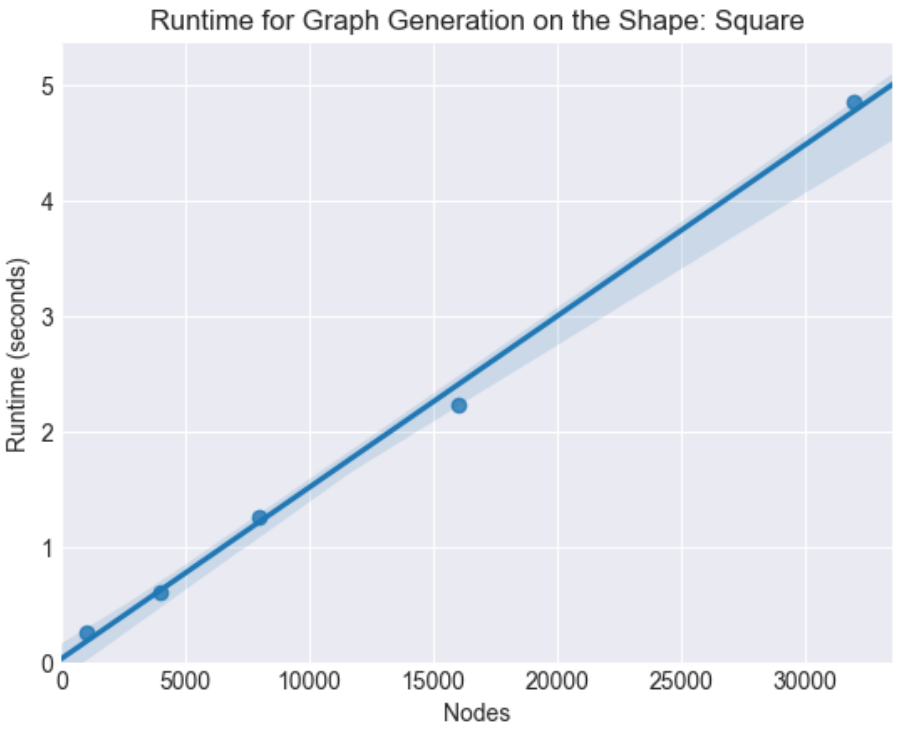

Figure 9: Runtimes of the $O(n)$ Square RGG Generation Algorithm

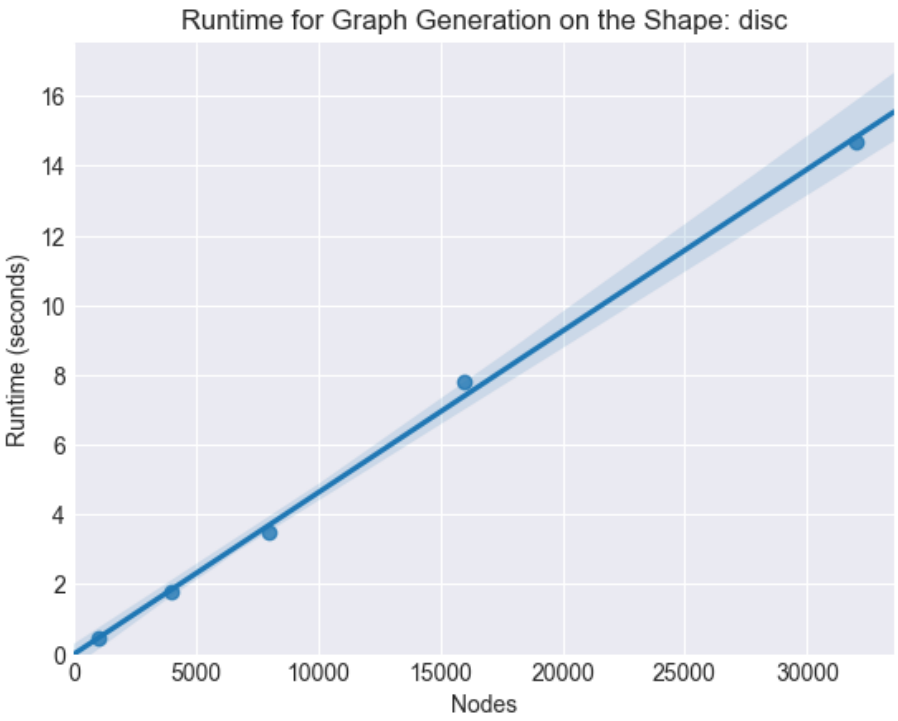

Figure 10: Runtimes of the $O(n)$ Disc RGG Generation Algorithm

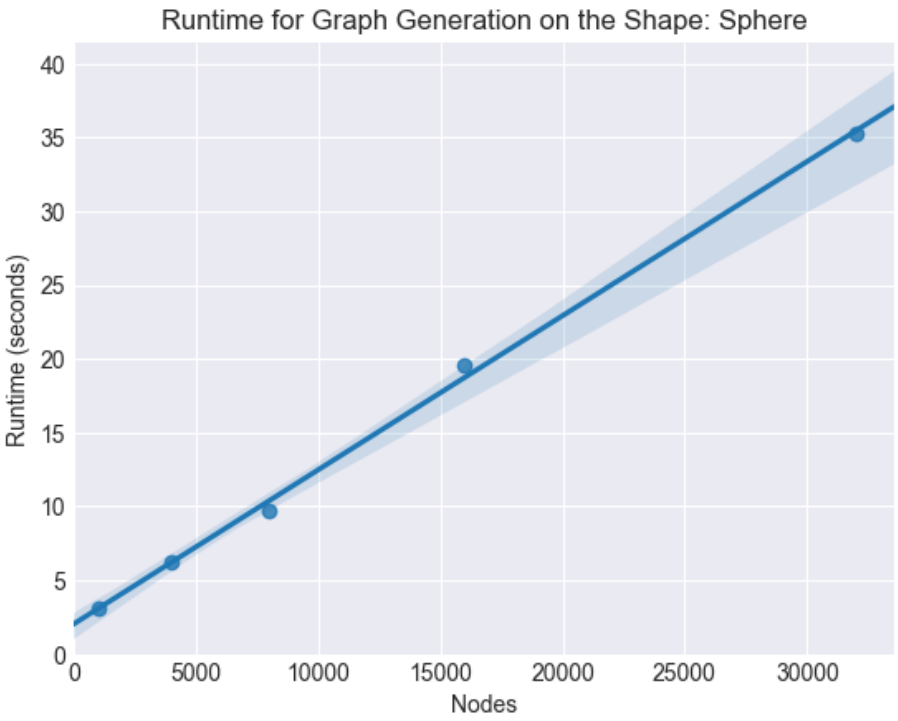

Figure 11: Runtimes of the $O(n)$ Sphere RGG Generation Algorithm

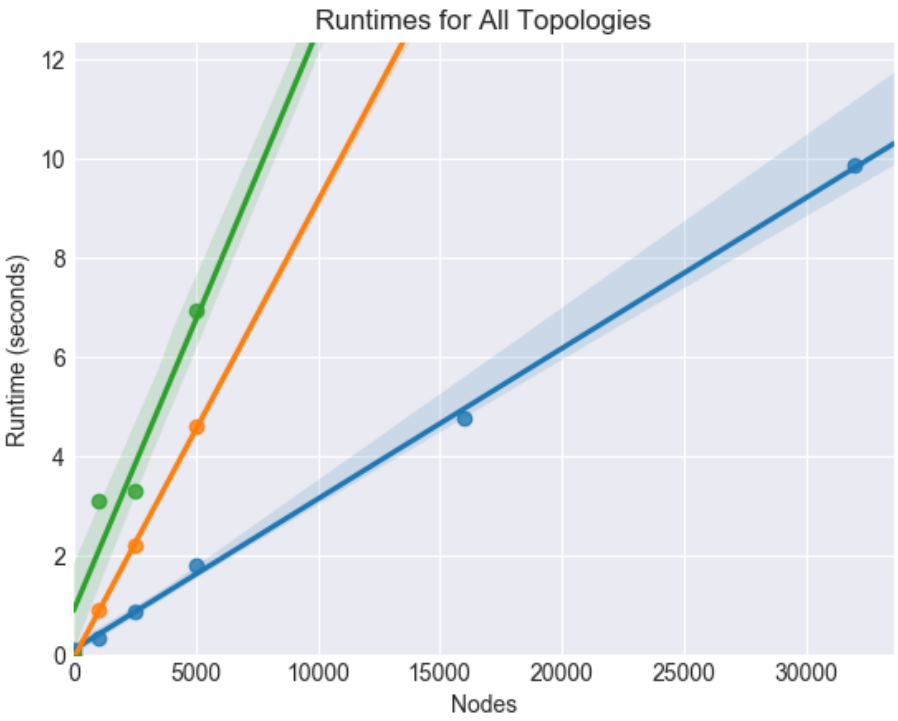

Figure 12: Runtimes of the $O(n)$ RGG Generation Algorithms overlayed

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] Siyfion (https://math.stackexchange.com/users/11104/siyfion). *Generating a random point on the unit circle*. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/4 (version: 2011-05-19). eprint: `https://math.stackexchange.com/q/40023`. URL: `https://math.stackexchange.com/q/40023`.