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

Luke Wood

March 6, 2018

1 Executive Summary

Wireless Sensor Networks are a huge area in modern networks research. 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]. In the aforementioned paper it is proven that randomly generated points is the best way to model the incoming wireless sensor networks. 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
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 topology generation algorithm runs in approximately 15 seconds for graph size 100000 which is reasonable given the large input size.

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

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. This is a possible optimization point for future iterations. Despite the algorithm being slower than the square version, it still terminated in 48 seconds for 100000 nodes. This is an acceptable runtime.

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

One interesting thing to notice in the above 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.

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 programmign language to utilize the high level of parralelism that comes from immutable data.

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. Converting RGGs to an Adjacency List
4. Serializing Adjacency Lists to files

Being able to serialize the Adjacency Lists to files is a noteworthy feature as if there are algorithms that require faster runtimes than a dynamically typed language can support (such as python) we can still generate the RGGs from the generation implementation and use them in other computation ecosystems.

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. 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 pythons dynamic typing).

2 Reduction to Practice

In this section I will discuss how the transition from theory to implementation went. I will discuss optimizations I made as well as optimizations that I was aware that I could make but decided not to for reasons.

2.1 Data Structure Design

In the generation portion of this project I use several different data structures. The first one is a python object of custom type node. This basically serves as a tuple of values consisting of an X location, a Y location, a list of nodes, and a node number. All of these are used during the connecting of the nodes in graph generation excluding node number. Node number is assigned during object construction time and is only used when converting the list of nodes into 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 I believe that 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 we 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.

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

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

nodes. This takes $O(e)$ in an adjacency list as opposed to $O(1)$ in an adjacency matrix.

2.2 Algorithm Description

In the following sections I will give a detailed analysis of the algorithms used in this project. I will discuss the runtimes of each algorithm in terms of Ω, O, and Θ when relevant.

2.3 Square Point Generation

The algorithm to generate the points in the unit square topology is incredibly simple. Simply pick two random numbers between 0 and 1 for all nodes.

2.4 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 in the unit disc is as follows: Pick a random x and y between 0 and 1. Calculate the distance between that point and the center of the unit circle (0.5, 0.5). If the distance is less than or equal to .5 we take this point, otherwise we generate another point.

2.5 Node Connection

In order to ensure that average degree of the nodes is close to the desired average degree we define a radius surrounding each node. 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 displayed below:

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	$r = sqrt(\frac{d(G)}{N\pi})$	$r = sqrt(\frac{d(G)}{N})$
Unit Sphere		

To derive the disc radius formula, I simply multiplied the unit square formula by pi because that is the amount of extra area that the unit disc has over the unit square

2.6 Conversion From Node List to Adjacency Matrix

The algorithm to convert from a list of the node class I defined to an adjacency list is extremely straight forward. The algorithm consists of a pair of nested map operations. The first operation maps each node to it's edges list. The second operation maps each item of the edge lists to it's respective node number. 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.

2.7 Algorithm Engineering

Originally I had a brute force algorithm that ran in $O(n^2)$ time. This quickly became problematic as the algorithm took upwards of 200 seconds to run on input size of only 12,000.

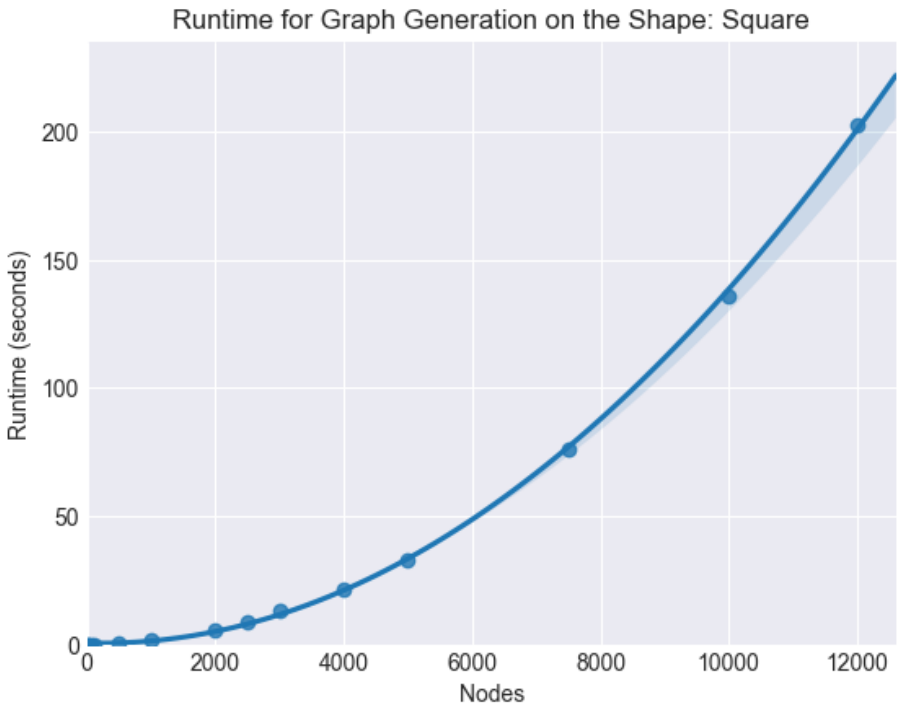

Figure 1: Runtimes of the $O(n^2)$ Algorithm

To fix this, I overhauled the algorithm to be $O(n)$. I did this by implementing the cell method described above in the Algorithm Description section as well as

in Chen's paper Bipartite Grid Partitioning of a Random Geometric Graph[1].

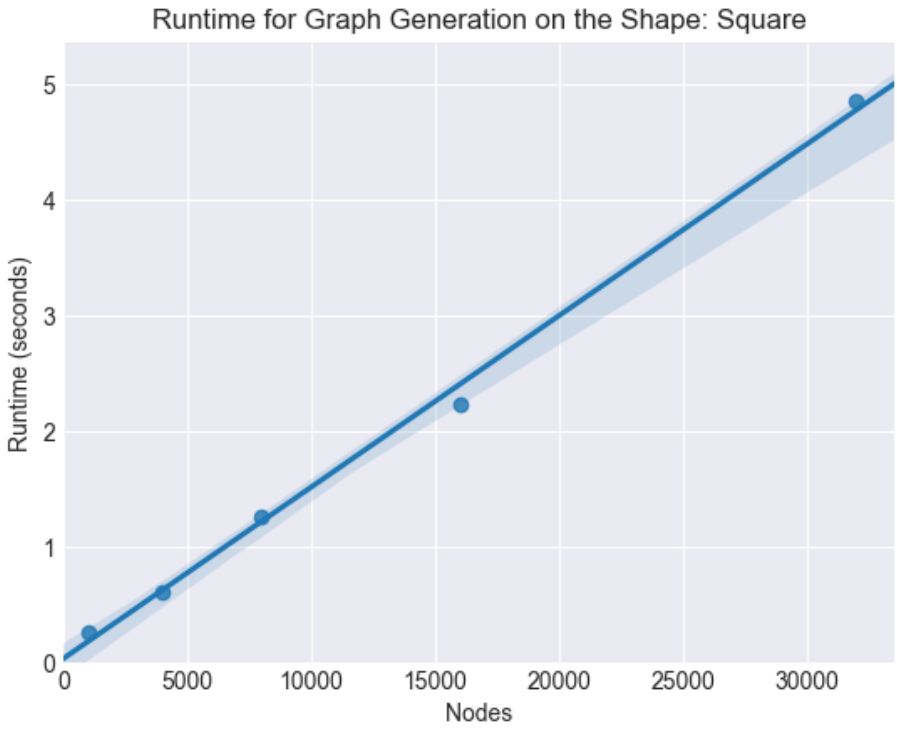

Figure 2: Runtimes of the $O(n)$ Algorithm

I have also included a table to compare some of the runtimes for the same size inputs below.

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

The $O(n)$ algorithm is far superior even on small input sizes such as 1000.

I originally was using a python object to store the data of each node but instead switched over to a statically indexed tuple.

2.8 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.

3 Result Summary

3.1 Edge Density

As expected I got a gaussian distribution for my edge densities. This is apparent in the attached edge distribution charts for each topology

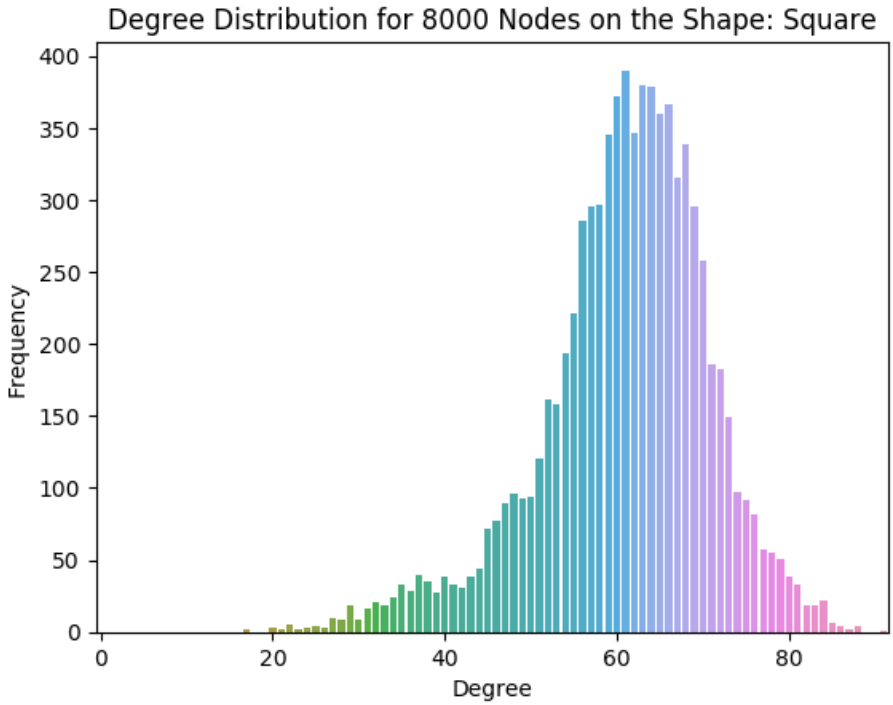

Figure 3: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Square

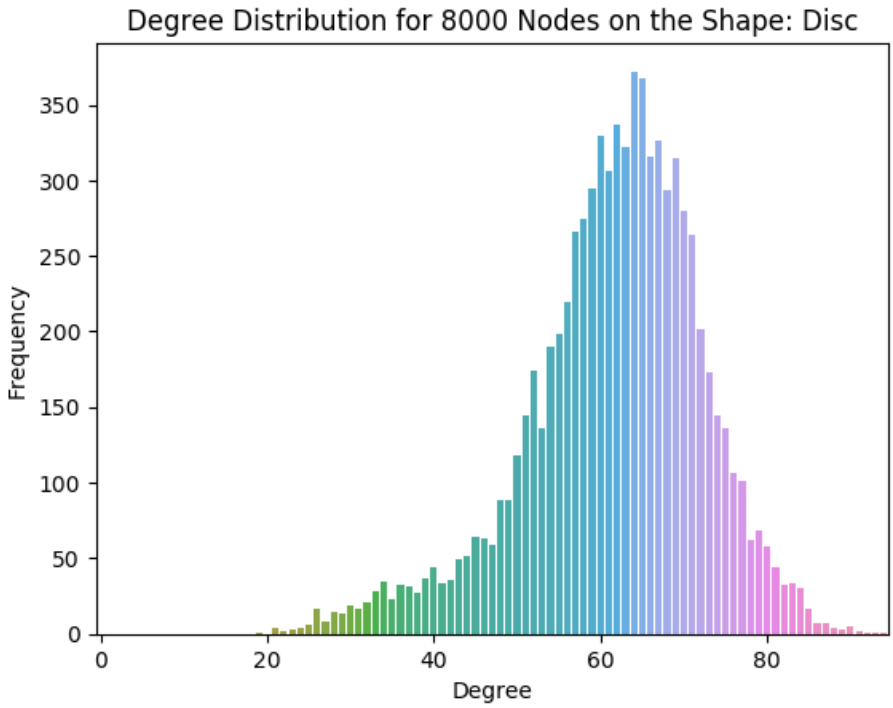

Figure 4: Edge Densities of an 8000 Node Graph with E(Degree)=64 on Topology Disc

3.2 Performance Rates

N	A	Square Time	Disc Time
1000	64	0.260793	0.488157
5000	64	0.834028	2.720183
10000	64	1.651358	4.596511
25000	64	4.034038	12.849979
50000	64	7.192651	24.288077
100000	64	13.939330	43.767211

Table 4: Runtimes of the $O(n)$ Graph Generations in Seconds

References

[1] Zizhen Chen and David W Matula. "Bipartite Grid Partitioning of a Random Geometric Graph". In: *2017 13th International Conference on Dis-*

tributed 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`.