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

Luke Wood

March 5, 2018

1 Executive Summary

1.1 Introduction and Summary

Wireless Sensor Networks can be incredibly expensive to deploy and test which makes them an excellent candidate for simulated testing. 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]. RGGs can be used as a relatively cheap way to gather valuable information about how wireless sensor networks can function and communicate. Through a series of 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.

Nodes	Expected Avg. Deg.	Real Avg. Deg.	Max Deg.	Min Deg.	Seconds
1000	64	57.568000	82	10	0.279913
5000	64	60.658400	87	15	0.840636
25000	64	62.504080	98	11	3.828407
50000	64	62.986840	100	20	8.938070
100000	64	63.258520	102	18	14.938207

Table 1: Data on graphs generated with the square topology

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 that RGG 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 my implementation and use them elsewhere.

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

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.

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

2.2.1 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.

2.2.2 Disc Point Generation

Generating the points on the Disc topology is slightly more challenging than generating the points for the square topology. Instead of picking two numbers between 0 and 1, we apply the following formulae:

$$\theta = rand()$$
$$x = sin(\theta)$$
$$y = cos(\theta)$$

By using the same theta to determine the x and y coordinate we ensure that our points will never fall out of the unit disc. I found a formula to generate points on a unit disk using [3]

2.3 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 in Chen's Paper	Equation for Radius Derive from Chen's
Unit Square	$d(G) \approx N\pi r^2$	$r = sqrt(\frac{d(G)}{N\pi})$
Unit Disc		
Unit Sphere		

2.4 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.

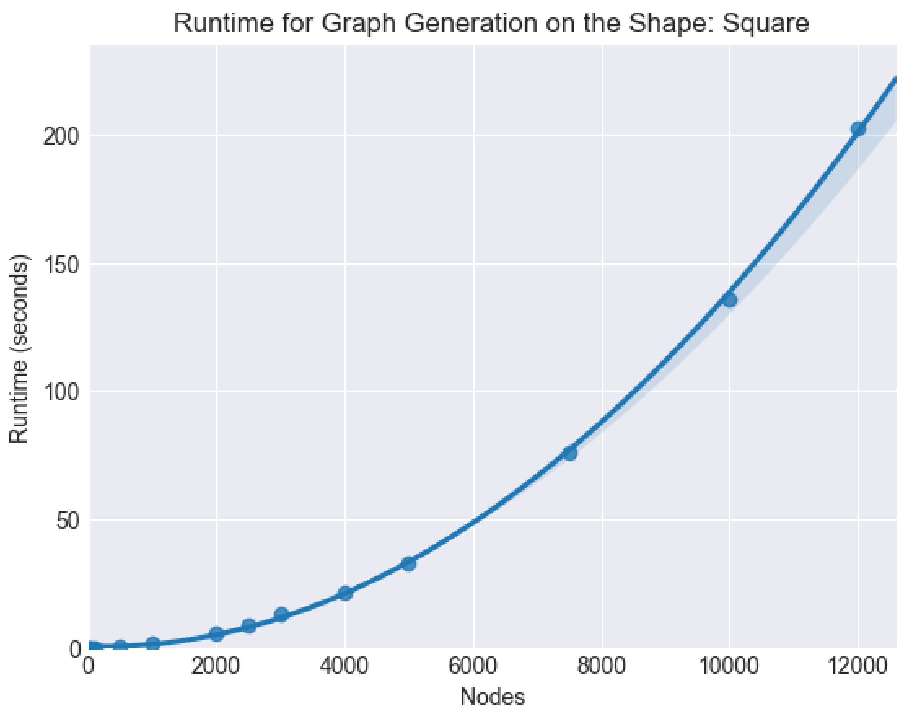

Figure 1: Runtimes of the $O(n^2)$ Algorithm

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].

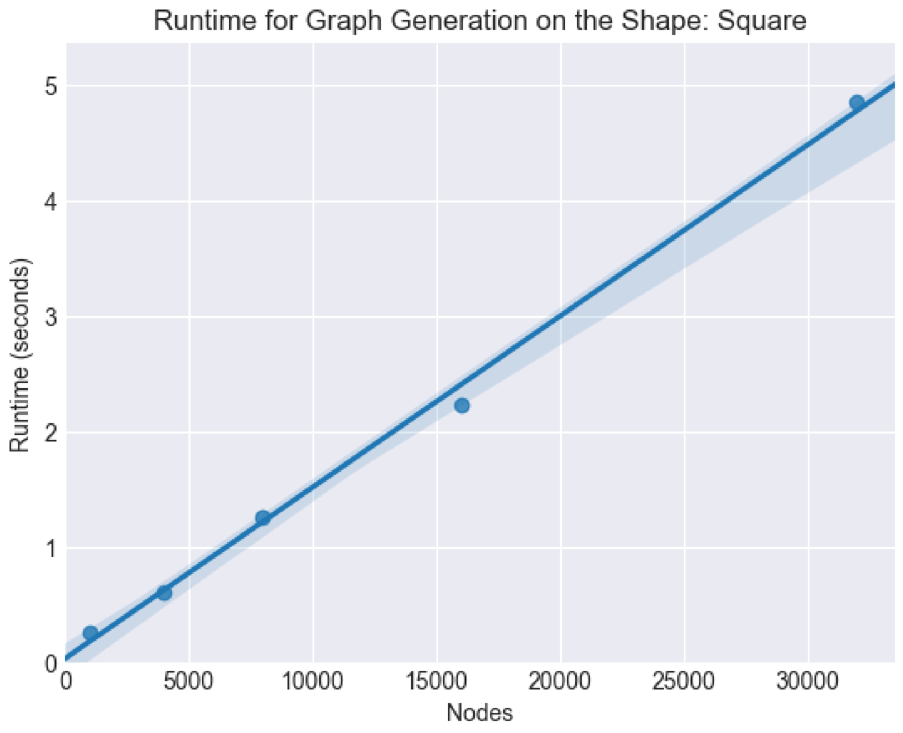

Figure 2: Runtimes of the $O(n)$ Algorithm

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 2: 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.

2.5 Verification

3 Result Summary

3.1 Edge Density

As expected I got a gaussian distribution for my edge densities.

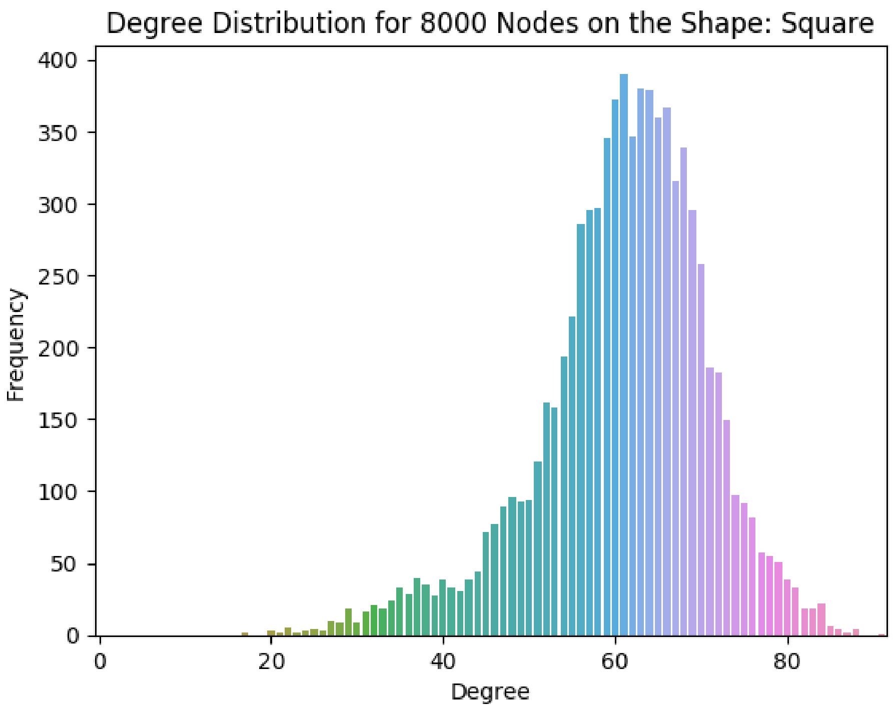

Figure 3: Edge Densities of an 8000 Node Graph with E(Degree)=64

3.2 Performance Rates

Nodes	Topology	Runtime

Table 3: Runtimes of the $O(n)$ vertex connection algorithm

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`.