Fast Generation of Spatially Embedded Random Networks

Matthew Roughan, *Senior Member IEEE* and Eric Parsonage

Abstract—Spatially Embedded Random Networks such as the Waxman random graph have been used in many settings for synthesizing networks. Prior to our work, there existed no software for generating these efficiently. Existing techniques are $O(n^2)$ where n is the number of nodes in the network; in this paper we present an $O(n+e)$ algorithm, where e is the number of edges.

Index Terms—Random graph model, Waxman graph, random plane networks, random geometric graphs, spatial networks, range-dependent random graphs, random connection models, random distance graphs, partially structured random graphs.

1 Introduction

Random graphs are frequently used as the underlying model in fields such as computer networking, sociology, biology and physics. Interesting questions arise regarding asymptotic behavior of large graphs, and many new datasets involve large numbers of elements. However, despite the emphasis on large graphs, there is relatively little work on their *efficient* generation.

Spatially Embedded Random Networks (SERN) [1, 2] (also known as random geometric graphs with general connection functions, and soft random geometric graphs [3]) are a large class of random graphs. They generalize many simpler models: *e.g.,* the Gilbert-Erdős-Rényi (GER) random graph [4, 5], the random plane network [6] and the Waxman random graph [7]. This paper is concerned with efficient generation of SERNs.

The GER random graph links every pair of vertices independently with a fixed probability, whereas a SERN allows that in real-world networks longer links are often more costly or difficult to construct, and their existence is therefore less likely. In a SERN, nodes are embedded in a metric space, and the probability they are connected is given by a function of the distance $d_{i,j}$ between them.

There are many examples of SERNs, including random plane networks, (soft) random geometric graphs, spatial networks, range-dependent random graphs, random connection models, random distance graphs, and partially structured random graphs. However, little effort has gone in their synthesis.

Generation of synthetic random graphs is one of the basic requirements for modeling, particularly when the model overlaying the graph is quite complex, for instance information flows in a social network [8], or routing protocols in computer networking [7]. Thus analysis of the graph may not provide the insights of simulation. The ability to generate an ensemble of graphs that match some characteristics of real-world graphs allows us to test predictions, as well as their sensitivity to the underlying assumptions. The facility to synthesize graphs is also needed in estimation procedures such as Approximate Bayesian Computation (ABC) [9].

The only existing software available for generating SERNs uses the approach of generating the node locations, calculating all distances, and then generating edges using a series of Bernoulli trials. This naïve algorithm has computational complexity $O(n^2)$ in the number of nodes n.

However, many real-world graphs are very sparse in the sense that the number of edges e is much smaller than the number of possible edges [10]. Sparsity is often modeled as the number of edges growing as $O(n)$. Thus for many real-world examples an $O(n^2)$ generation algorithm is highly inefficient.

Here we develop a fast, efficient method for creating large sparse SERNs. Our method takes $O(n+e)$ computation and memory, which is the best possible for an exact method. We have used it to generate graphs with more than four billion nodes in approximately 20 minutes (on a single core of an Intel i7, 6900K, running at 3.9 GHz). We demonstrate in detail in the paper the $O(n+e)$ algorithm for generating Waxman graphs, the best previous algorithm being $O(n^2)$.

We also demonstrate a multi-threaded implementation that shows that the method parallelizes.

Further details are provided in [11] and the software described here is publicly available at git@github.com: lamestllama/conSERN.git.

2 Background

A graph (or network) consists of a set of n nodes or vertices which, without loss of generality, we label $\mathcal{V} = \{1, 2, \ldots, n\}$, and edges or links $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$. We are primarily concerned here with undirected graphs (though our work is easy to generalize to directed graphs). We say that two nodes i and j are *adjacent* or *neighbors* if $(i, j) \in \mathcal{E}$.

The classic example of a *random graph* is the GER random graph [4, 5], $G_{n,p}$ of n vertices, which is constructed by assigning each edge (i, j) to be in $\mathcal{E}$ independently, with fixed probability p. A SERN generalizes this by making

• *M. Roughan and E. Parsonage are with the ARC Centre of Excellence for Mathematical & Statistical Frontiers (ACEMS), in the School of Mathematical Sciences at the University of Adelaide.*

the probability of each edge dependent on the geometric distance between the two nodes.

SERNs [1–3] constitute a large class of useful random-graph models including GER random graphs, Gilbert's random plane network [6] (also known by other names such as the *random geometric graph* [3, 12]) and the Waxman random graph [7].

Formally, we create a SERN by placing n nodes randomly within some defined region R of a metric space Ω with distance metric $d(x, y)$. Each pair of nodes is linked, *i.e.,* made adjacent in the graph, independently, with link probability given by a function of distance $d_{i,j} = d(x_i, x_j)$ between nodes i and j. For instance we could define a space with one of the standard distance metrics: *e.g.,* the Euclidean distance metric, or any of the standard ℓ_p distances.

There are many such models in the literature, using different probability functions [3–6, 13–18]. All of the examples of which we are aware have non-increasing link-probability functions (beyond some point) so we refer to these as *distance deterrence functions*, and exploit this property in our algorithm. In our work, we choose to represent distance functions in general in the form $q\, f_\theta(s\, d_{i,j})$, where parameters $s \in [0, \infty)$, and $q \in (0, 1]$. The existing functions fit this formulation, at least approximately, though their parameterizations differ. The advantage of this common formulation is that in this representation the parameters have a consistent meaning: q is a *thinning* parameter, s a *scale* parameter, and θ represents a possible set of *shape* parameters.

For example, the distance deterrence function chosen by Waxman was the negative exponential

$$p(d_{i,j}) = q\, e^{-s d_{i,j}}. \tag{1}$$

The Waxman graph[1] has been used in many settings from computer networks [7] to biological cell networks [21], typically to synthesize random networks. However, our algorithms apply to the wider class of SERNs, for instance covering models such as

- Threshold: $p_{i,j} = q\, H(1 - s d_{i,j})$, where $H(\cdot)$ is the Heavyside step function, motivated by the *random plane network* [6];
- Cauchy: $p_{i,j} = q\, (1 + (s d_{i,j})^2)^{-1}$ [17].

There are many others: *e.g.,* the "Exponential" [18] and "Power law" [13–16], as well as hybrid functions. The fact that in many real-world networks longer links have a higher cost and thus exist with lower probability is the intuition behind many SERN models. The distance deterrence functions above reflect this by being monotone non-increasing, and reducing in the limit to 0.

Given the variety of deterrence functions, the types of metric spaces on which the model can be applied (our code supports Euclidean, Manhattan, Max, and Discrete distance metrics), and the arbitrary convex region R on which they can be applied, there are many possible cases encompassed by the idea of a SERN.

1. Note that despite the presence of the exponential function, the Waxman graph is not in the class of Exponential Random Graphs (ERG) [19, 20], where the exponential applies to the probability of a particular graph (not a particular link), as a function of the overall graph properties.

In modern problems, networks of millions of nodes are common, and billion node networks exist. For instance, Facebook claims (as of July 2015) over a billion active users, who form part of a large graph. As network modeling moves towards encompassing such graphs, the need to synthesize very large graphs increases.

We are not aware of any general tools to generate wide classes of SERNs. There are a number that have been designed for generating Waxman random graphs [18, 22–25], but none seriously consider how to generate these graphs quickly. All of the software that generates true Waxman graphs are $O(n^2)$ in computation time [26], and the vectorized Matlab algorithm is $O(n^2)$ in memory as well. We demonstrate our approach on Waxman graphs for comparison to the literature, but the reader should keep in mind that our implementation caters for SERNs in general.

Many of the properties of SERNs are known [1–3], but results for the Waxman graph are particularly clear, *e.g.,* see [27], and thus illustrative of the more general case. This is another motivation for using these graphs as our main example.

Our work is helped by the fact that many, if not most, large real-world graphs are sparse, *i.e.,* the number of edges is $O(n)$. Since the deterrence functions don't take n as a parameter it is necessary as we increase n to scale the parameters of the deterrence function in order to maintain $e = O(n)$. Examination of the deterrence functions above shows that the parameter q acts as a filter on the overall number of edges, but has no effect on the distribution of the lengths of the edges in a graph. Thus, it is natural to scale these graphs taking $q \propto 1/n$. The result of this scaling is just a reduction in edge density (per node pair), leaving the edge-length distribution untouched.

The other parameters do however have an effect on the distribution of the edge lengths thus we call them scale or shape parameters. An alternative scaling, for instance taking s as a function of n, leads to a distortion of the link distance distribution, dependent on the particular deterrence function, and so is less appealing. For instance, in many models this distance distribution arises from a physical cost. The implication of scaling with s would be that the costs change non-linearly as a function of node density. We discuss this possibility further below, but note that because of its simplicity and transparency, q is the natural choice for scaling as n increases, keeping the other parameters constant.

2.1 Fast generation of GER graphs

The common method for generation of the GER random graphs $G_{n,p}$ is simply to perform $O(n^2)$ Bernoulli trials, one for each possible edge. Batagelj and Brandes [28] noted that this algorithm is naïve, and proposed an $O(n+e)$ algorithm. We use this algorithm as a component of our own, and also use the underlying idea of their algorithm to develop a new approach.

The naïve algorithm considers each potential edge in turn, and flips a biased coin – *i.e.,* generates a Bernoulli trial – to see if the edge exists. Batagelj and Brandes list all the potential edges in order, and generate jumps across all of the "non-edges". Given a series of Bernoulli trials the jumps between these will take the geometric distribution.

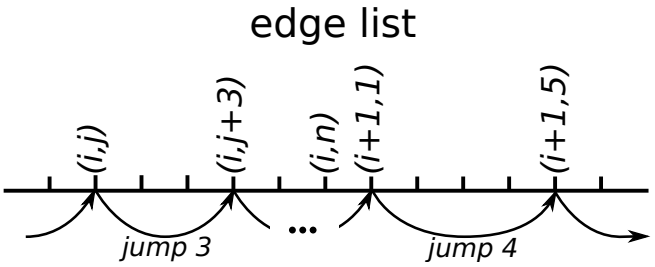

Fig. 1. An illustration of Batagelj and Brandes' algorithm: the edges are listed in lexicogaphic order, and instead of testing each edge by an independent Bernoulli trial, we generate a series of geometric jumps, resulting in this instance, in the selected edges being (i, j), $(i, j + 3)$, $(i + 1, 1)$ and $(i + 1, 5)$. In practice, we don't need to actually list the edges, but only provide a means for indexing into the putative list.

So one can generate the same stochastic process with these jumps by listing the edges in some order, and then jumping between edges as shown in Figure 1.

We jump from edge to edge, so we need only generate a jump for each edge (plus one last jump), *i.e.*, the algorithm needs $O(e)$ jumps. If the graph is sparse, this is much faster than the naïve approach. In practice, we don't need to actually list the edges, but only provide a means for indexing into the putative list, and as long at this index step is $O(1)$, the resulting algorithm is $O(n + e)$. The $O(n)$ term comes from the initial construction of the nodes, presuming fixed sized integers are used to represent the node labels, and the $O(e)$ from the construction of the edges.

We cannot simply reuse this idea to generate SERNs, because in a SERN all links do not have equal probabilities. The appropriate size jumps would not therefore follow a simple distribution, and if they could be calculated at all, it would require as much work as a naïve approach.

However, we can use Batagelj and Brandes' algorithm to create a set of initial edges. Then, we use our novel filtering technique to obtain a subset of these edges that then form the desired graph. We then further improve this using a divide-and-conquer approach in our second algorithm.

It is worth noting also that Batagelj and Brandes' algorithm underlies another idea [29] for generation of a different class of random graphs, highlighting its versatility.

3 Fast Waxman Generation

3.1 The Waxman SERN

In what follows we shall use a number of results regarding the Waxman SERN, where the Euclidean distance metric is used, and the distance deterrence function is (1).

The probability that an arbitrary link exists (prior to knowing the distances) is

$$\mathbb{P}\{(i,j) \in \mathcal{E} \mid q, s\} = q \int_0^\infty \exp(-st) g(t)\, dt = q\tilde{G}(s), \quad (2)$$

for any $i \neq j$, where $\tilde{G}(s)$ is the Laplace transform of $g(t)$, which is the Probability Density Function (PDF) of the distance between an arbitrary pair of random points in the space in question. The calculation of this density is commonly described as the *Line-Picking-Problem* (the length distribution of random lines in a region), and analytic expressions exist for many cases. Most notably, Waxman SERNs have been typically generated on the unit square, for which an analytic expression is known [30]:

$$g(t) = \begin{cases} 2t(t^2 - 4t + \pi) & \text{for } 0 \leq t \leq 1, \\ 2t\left[4\sqrt{t^2-1} - (t^2 + 2 - \pi) - 4\tan^{-1}\left(\sqrt{t^2-1}\right)\right] & \\ \qquad \text{for } 1 \leq t \leq \sqrt{2}. & \end{cases} \quad (3)$$

The solutions to the line-picking problem for other region shapes and metrics are also known. We consider the common case here, but other cases are shown in Figure 2 for comparison, the most salient feature being that symmetric 2D regions (the square and disk for example) have very similar densities, with some difference in the mode if we consider 3D regions, and a larger difference in the tail if we consider a long-thin rectangular region.

The corresponding functions $\tilde{G}(s)$ are shown in Figure 3.

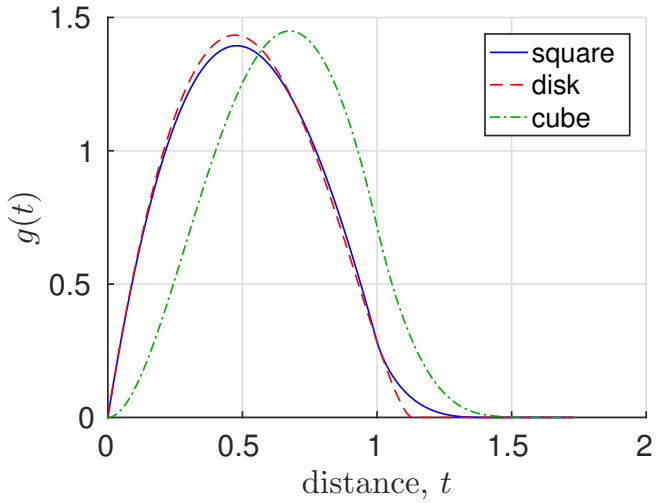

Fig. 2. The probability density function $g(t)$ for the Line-Picking-Problem on various regions (with area or volume 1), given a Euclidean metric.

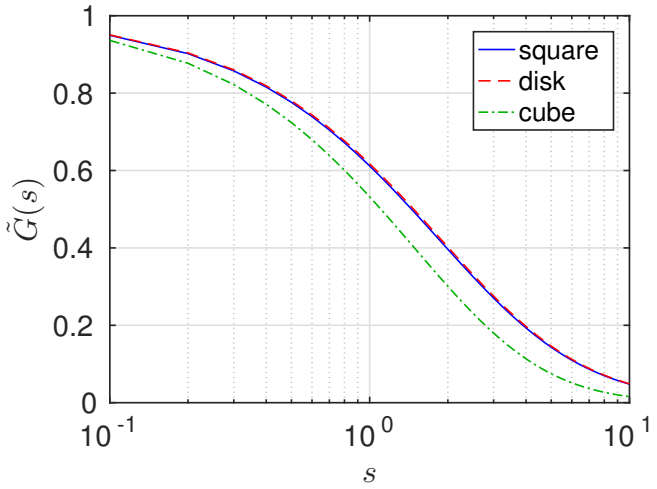

Fig. 3. The Laplace transforms $\tilde{G}(s)$ for different regions.

We could equivalently note that $\tilde{G}(s)$ is the moment generating function (w.r.t. to $-s$) of $g(t)$. We also know that the Laplace transform at $s = 0$ of a probability density is the normalization constraint, so $\tilde{G}(0) = 1$. Hence when $s = 0$

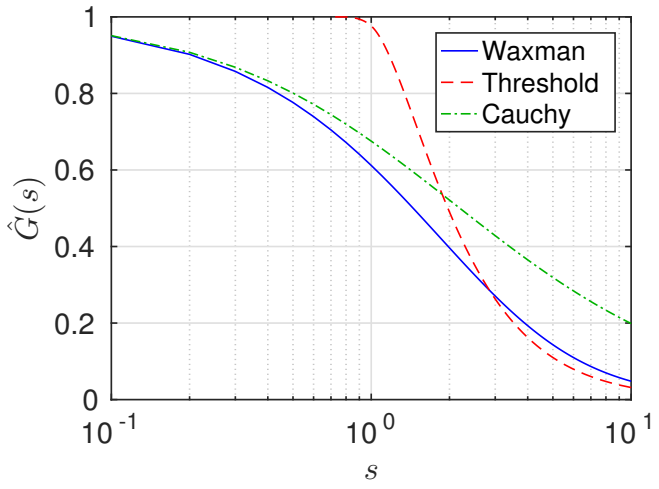

Fig. 4. The graph metric $\hat{G}(s)$ for three SERNs.

there is no distance dependence and the Waxman graph is equivalent to the GER random graph $G_{n,q}$.

From this probability we can also compute features of the graph such as the average node degree

$$\bar{k} = (n-1)q\tilde{G}(s), \tag{4}$$

from which we can derive values of q that produce given average degree for a given network size and s. From the Handshake Theorem we can derive the average number of edges to be

$$\bar{e} = n(n-1)q\tilde{G}(s)/2. \tag{5}$$

From this we can see that if $q \propto 1/n$, and s is held constant, then $\bar{e} = O(n)$.

The results above generalize to other SERNs, where instead of the Laplace transform, we calculate the convolution of $g(t)$ with the distance deterrence function, denoting the resulting metric $\hat{G}(s;\theta)$, which we show in Figure 4 for the cases described earlier. We can see that each has a different form, but experiences an approximately similar range of variation.

3.2 Algorithms

All SERN generators must start by generating a set of n nodes, which takes $O(n)$ operations. We discuss methods for doing so efficiently in § 5, once we have described the additional requirements imposed on this process by our algorithm. Here we concentrate on the main performance bottleneck: generating the edges. For simplicity, we describe our edge generation algorithms for the Waxman SERN, though our code works for a much larger class of SERNs.

Our first algorithm – q-jumping – uses the observation that distances $d_{i,j} \geq 0$ and so

$$p(d_{i,j}) = q\, e^{-sd_{i,j}} \leq q. \tag{6}$$

That is, we could imagine generating the graph in two steps. In the first we generate a GER $G_{n,q}$ random graph with probability q for each edge. We can generate the GER graph quickly using Batagelj and Brandes' algorithm. In the second step, we note that the Waxman probabilities for each edge are smaller than those for the GER graph, and so we can randomly filter the GER graph to obtain a Waxman graph. We do so in two ways: a simple iterative filter (see Algorithm 1), which we then further improve using a divide-and-conquer approach in our second algorithm (bucket-jumping).

1: Input: n, q, s
2: $\mathcal{E} \leftarrow \phi$
3: $\mathcal{E}_1 \leftarrow G_{n,q}$
4: **for** $(i,j) \in \mathcal{E}_1$ **do**
5: calculate $d_{i,j}$
6: calculate $p'_{i,j} \leftarrow \exp(-sd_{i,j})$
7: generate $r \sim U[0,1]$
8: **if** $r \leq p'_{i,j}$ **then**
9: $\mathcal{E} \leftarrow \mathcal{E} \cup (i,j)$
10: **end if**
11: **end for**

ALGORITHM 1. The q-jumping algorithm for generating the edges of an undirected Waxman graph. Note that there is no q in Step 6 because this factor has already been incorporated in the formation of $\mathcal{E}_1 \leftarrow G_{n,q}$. The algorithm can be generalized to SERNs simply by replacing line 6 with $p'_{i,j} \leftarrow f_\theta(sd_{ij})$, the general distance deterrence function.

Theorem 1. *Algorithm 1 (q-jumping) generates a Waxman SERN.*

Proof. Step 3, by definition, generates a GER $G_{n,q}$, *i.e.*, a graph where each link is chosen independently with probability q.

Each link from $\mathcal{E}_1$ is added independently to $\mathcal{E}$ with probability $\exp(-sd_{i,j})$, conditional on d_{ij}, so the overall probability of each link is $q\exp(-sd_{i,j})$, and the existence of each link is independent conditional on the distances. That is, we have generated a Waxman SERN. □

Theorem 2. *Algorithm 1 is $O(n+e)$. Moreover, the efficiency of the algorithm, as measured by the ratio of time to generate the edges of a GER graph to those of the Waxman graph with the same average node degree is the Laplace transform $\tilde{G}(s)$, of (3).*

Proof. The initial generation of the nodes is $O(n)$ (we generate a list of n node locations).

Step 3 (generation of $\mathcal{E}_1 \leftarrow G_{n,q}$) is $O(e_1)$ (see [28]), where e_1 is the number of edges generated in $\mathcal{E}_1$ (noting that the edge generation requires 1 jump per generated edge plus one additional jump at the end).

The second component of the algorithm iterates through each edge of $\mathcal{E}_1$, adding it to $\mathcal{E}$ if it passes the random criteria whose calculation is $O(1)$, hence the overall algorithm is $O(n+e_1)$, where $e_1 = |\mathcal{E}_1|$.

The efficiency takes into account that we generate the GER graph first, and iterate through each edge. Each edge from $\mathcal{E}_1$ is included with probability $\exp(-sd_{i,j})$, *i.e.*, the edge probability of a Waxman graph with $q = 1$. Hence from (2), the expected number of edges in the final graph is $e_1\tilde{G}(s)$.

As e is related to e_1 by a fixed ratio, the algorithm is $O(n+e)$, and the overall efficiency is $\tilde{G}(s)$, where we measure this as a ratio of the number of edges in the GER compared to the number in the final graph. Equivalently, we can consider this as a ratio of the times to generate GER and Waxman graphs of equivalent average node degree. □

The extensions of the theorems to SERNs in general should be obvious.

The theorem above shows that the algorithm's computational complexity is $O(n+e)$, but also that the efficiency depends on $\tilde{G}(s)$. A Laplace transform of a PDF obeys certain properties: $\tilde{G}(0) = 1$, and $\tilde{G}(s) \to 0$ for large s, so the q-jumping algorithm will be quite fast for small s, but less so as s grows. We can see the size of this effect if we examine Figure 3.

On the other hand, the main property of SERNs is that longer links are less likely, and that for larger s this effect is increased. Thus the very nature of these graphs creates geometric structure that we can exploit in their generation.

We do so by breaking the region into M^2 "buckets" in a regular, rectangular grid (see Figure 5). Given nodes i and j in buckets I and J, respectively, we can put a lower bound $D_{I,J} \leq d_{i,j}$ on the distance between the nodes, and thus an upper bound on the probability of a link.

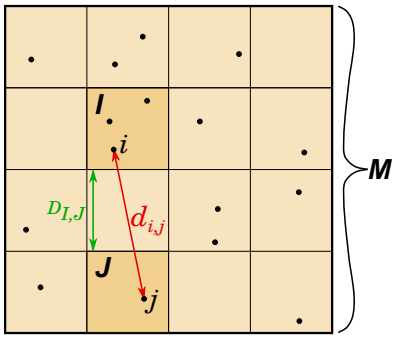

Fig. 5. Region broken into buckets. We refer to these as buckets rather than the more obvious grid, or other terms, because in general they might not form a regular grid.

As the GER jumping algorithm does not depend on the order of the potential edges, or even that we generated them all at once, we can use this approach to generate the set of edges between any pair of buckets using the upper bound given above. The resulting algorithm is shown in Algorithm 2.

Theorem 3. *Algorithm 2 (bucket-jumping) generates a (exact) Waxman SERN.*

Proof. The set of nodes is partitioned into M^2 disjoint buckets that cover all of the nodes exactly once each. The algorithm considers each pair of the M^2 buckets once, and thus considers each potential edge between every pair of nodes exactly once.

If $I = J$, then we are forming a subgraph consisting of the nodes n_I that are in bucket I, and $D_{I,J} = 0$. This is just formation of a Waxman graph on a smaller region defined by the bucket. Hence the resulting subgraph is a Waxman graph on B_I.

For $I \neq J$, we adapt Batagelj and Brandes' algorithm in the following way. Their original algorithm lists the potential edges in lexicographic order, and takes geometric jumps with probability distribution

$$p(\text{jump} = k) = q(1-q)^k,$$

where q is the probability of an edge, and $k = 0, 1, \ldots$ is the size of the jump between the edges in the aforementioned list. The results will be the same as determining the set of

```
1:  Input: n, q, s, M
2:  E ← φ
3:  for I=1..M^2 do
4:    for J=I..M^2 do
5:      N_{I,J} ← number of possible node pairs
6:      Q_{I,J} ← q exp(−sD_{I,J})
7:      E_{I,J} ← G_{N_{I,J},Q_{I,J}}
8:      for (i, j) ∈ E_{I,J} do
9:        calculate d_{i,j}
10:       calculate p'_{i,j} ← exp( − s(d_{i,j} − D_{I,J}))
11:       generate r ~ U[0, 1]
12:       if r ≤ p'_{i,j} then
13:         E ← E ∪ (i, j)
14:       end if
15:     end for
16:   end for
17: end for
```

ALGORITHM 2. The bucket-jumping algorithm for generating the edges of an undirected Waxman graph. The algorithm can be generalized to other SERNs by replacing line 6 with $Q_{I,J} \leftarrow qf_\theta(sD_{I,J})$ and line 10 by $p'_{i,j} \leftarrow qf_\theta(sd_{ij})/Q_{I,J}$.

edges via a sequence of Bernoulli trials with probability q [31, p.165]. The first edge is generated with a jump from the beginning of the list, and the last occurs when a jump passes beyond the length of the list.

The important detail to realize is that there is nothing special about Batagelj and Brandes' ordering. The potential links (in their algorithm) are independent, and hence, we could have ordered them in any arbitrary fashion.

In our case, we list a set of potential links by taking one node from bucket I, and one from bucket J, also in lexicographic order. The jump process will then generate a set of equi-likely edges forming a bipartite graph $G_{N_{I,J},Q_{I,J}}$, where each edge has one end in bucket I the other in bucket J. Each edge has constant probability $Q_{I,J} = q\exp(-sD_{I,J})$.

These form a graph from which we then filter the actual edges of our Waxman graph with probability $\exp\left(-s(d_{i,j} - D_{I,J})\right)$. The two steps are independent, and so the overall probability of selection is

$$p_{i,j} = q\exp(-sD_{I,J})\exp\left(-s(d_{i,j}-D_{I,J})\right) = q\exp(-sd_{i,j}).$$

Thus, the overall selection process is a heterogeneous Bernoulli process, where the probability of selection of a given edge is given by the Waxman distance deterrence function of the distance between the two nodes. That is, we have generated an exact Waxman SERN. □

Theorem 4. *Algorithm 2 is $O(n+e)$.*

Proof. The algorithm generates n nodes, and allocates them to buckets. We discuss this step in depth below, but for instance, with regular buckets the allocation can be performed with simple arithmetic, so this stage is $O(n)$.

The algorithm considers M^2 buckets partitioning the n nodes. Each pair of such buckets is considered once. If $I = J$, then we are forming a subgraph consisting of the nodes n_I that are in bucket I. This is identical to the formation of a Waxman graph on any other region, and so the edge

generation step is $O(e_{I,I})$ where $e_{I,I}$ is the resulting set of edges in this bucket.

If $I \neq J$, then the two buckets are disjoint. Assume buckets I and J contain n_I and n_J nodes, respectively. The resulting subgraph $\mathcal{E}_{I,J}$ will be a bipartite graph, where each edge has one end in bucket I the other in bucket J. The considered node pairs take one element from each bucket, so the number of nodes pairs is $N_{I,J} = n_I n_J$, but as in the other case, generation of the edges depends on the number of edges found, not the number of potential edges, so this step is $O(e_{I,J})$.

The total computation cost is therefore

$$O\left(\sum_I e_{I,I} + \sum_{I=1}^{M^2}\sum_{J=I+1}^{M^2} e_{I,J}\right) = O(e),$$

where e is the final number of edges in the graph. □

Note that in the complexity analysis above, we assume s is a constant, and we scale $q \propto 1/n$. As discussed earlier, this is the natural scaling for these graphs.

If, however, one were to scale the graph by keeping q constant, and changing s, then we can see from Figures 3 and 4 and Equation (4) that s would change non-linearly. We can derive asymptotic expansions for $G(s)$ for large s from Tauberian theory [32, Theorem 2, pp.445-6], resulting in the approximation (for 2D regions)

$$\tilde{G}(s) \overset{s\to\infty}{\sim} \pi s^{-2}. \tag{7}$$

Given this expansion, we can see that to assure constant node degree, according to (4), we would need to have $s \sim \sqrt{n}$, and the "efficiency" of Algorithm 1 (as measured by the ratio of its computation time to that of the GER bounding graph) would be $O(1/n)$, *i.e.*, the q-jumping algorithm would be $O(n^2)$ losing the efficiency gains. However, analysis of Algorithm 2 for this case is more complicated, because if s changes, then it is natural to change the number of buckets M. We will illustrate below the choice of M is not intrinsically difficult, and that it obviates in large part the reduction in efficiency of Algorithm 1.

4 Results

We test the performance of the algorithms described above using a C implementation, for which we provide stand-alone code, library functions, and Matlab MEX bindings. The latter allows us to compare the computation times of various algorithms through the common mechanism of Matlab's `tic()/toc()` functions, which provide comparable wall-clock time estimates between Matlab and C implementations. We test timing by generating 100 networks and taking the shortest times for each on a Ubuntu 12.10 Linux box running on an Intel i7 X990 CPU with 6 cores running at 3.47 GHz, with Matlab (R2013a), and gcc 4.7.2. In each case we generate a network with fixed average node degree, *i.e.*, a sparse graph with $O(e) = O(n)$, achieved by scaling the parameter $q \propto 1/n$ as described above.

Figure 6 shows the results for a small s value, over a range of network sizes n, and for two bucket grid sizes $M = 1$ and 10. The dashed blue curve shows results for a vectorized Matlab implementation as a benchmark. The dashed red curve is the naïve algorithm implemented in C, which shows clear $O(n^2)$ performance, with roughly a two times speed up in comparison to the Matlab implementation. Note that Matlab has the capability to use multiple threads to speed up vectorized computations, whereas this C-version uses a single thread, hence the C-code speedup is not as great as might be expected. The Matlab implementation uses $O(n^2)$ memory, so we do not attempt very large Matlab tests. All others use $O(n+e)$ memory.

The solid curves show the bucket-based algorithm for two bucket grid sizes $M = 1$ and 10 (when $M = 1$ the bucket algorithm is equivalent to the q-jumping algorithm).

We can see for both values of M that the performance for large n is $O(e)$, and that the bucket grid size M has negligible impact for large n. For small n we can see the overhead of having more buckets.

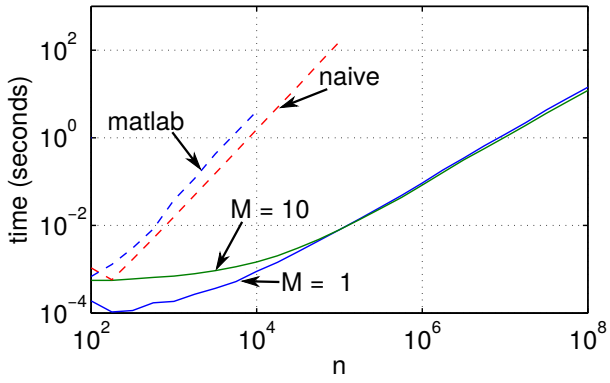

Fig. 6. Performance of the bucket algorithm compared to the naïve algorithm for $s = 0.1$ (single thread). Note that when $M = 1$ the bucket algorithm is equivalent to the q-jumping algorithm.

Figure 7 shows the performance for large s, and although we see the same broad features as in the previous figure, we now also see the benefit of the buckets. A larger number of buckets improves the algorithm for larger s, though there is a diminishing return as M increases.

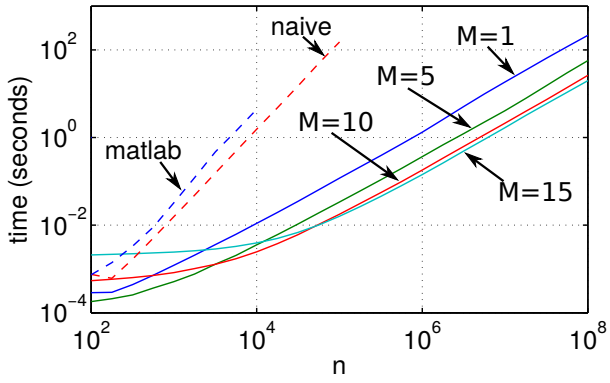

Fig. 7. Performance comparison for $s = 10$ (single thread).

We consider the effect of M more carefully in Figure 8, which shows the performance for fixed n over a range of s values. Most obviously, any fixed number of buckets has a "sweet spot" where it best balances the initial overhead of

bucket creation with the performance drop off as s increases. However, a relatively small number of buckets (around $M = 20$) provides good performance over a very wide range of parameters (note the log-log axes).

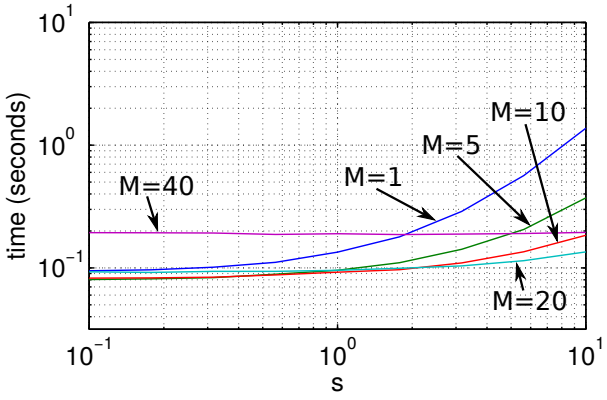

Fig. 8. Performance as a function of s ($n = 10^6$, single thread).

The asymptotic overhead of increasing M is $O(M^4)$, and so choice of M might seem to involve a difficult trade-off, but the figure shows that over two orders of magnitude of s, the choice of M is easy. Moreover, for such small M values we have not reached the asymptotic performance limits: for instance, increasing M from 1 to 10 does not involve a 10^4 increase in compute time (for small s where the overhead is most obvious). Thus choosing M for a particular application is not challenging, and could be accomplished by a simple set of pilot runs with moderate sized networks.

The final results shown in Figure 9 show the multi-threading performance compared with the ideal parallelized performance. The figure shows that the parallelization works, but the multi-thread implementation has significant overhead in bringing the edges back together. If we were aiming to calculate statistical properties of the graph that did not require it to be stored as a whole (for instance, average node degrees or link distances), then we could construct the information required, in parallel, without this overhead, and thus attain the ideal performance.

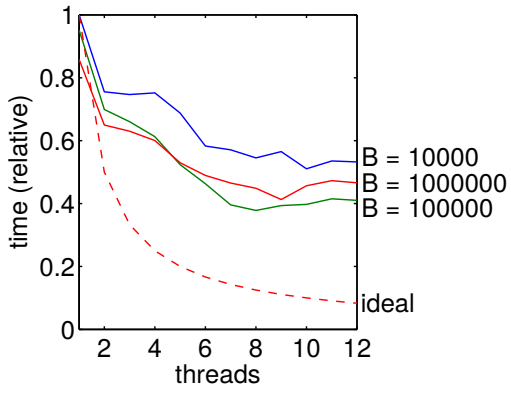

Fig. 9. Multi-thread performance ($n = 10^6$, $M = 20$) for different thread-buffer sizes B.

These results concern the Waxman case, but the key matter to note here is that they apply for any sparse SERN because either distance matters only a little, in which case straight q-jumping will work, or distance is significant, in which case buckets will reduce the computational workload.

5 Implementation Details

Our implementation is based on a shared C library using only standard POSIX packages including the `pthread` library. We also provide Matlab MEX bindings.

The aim is to create very large graphs, and so the use of memory is important. We restrict integers to 32 bits to reduce the memory footprint (though this limits the number of nodes to 2^{32}). We also avoid the use of data structures which fragment memory and thus cause the processor to try to cache memory from both ends of the available address space simultaneously. Using linear data structures maximizes the effectiveness of the cache.

The running time of our algorithm is dominated by the time spent creating links, but most of the code is devoted to setting the preconditions for the algorithm to work efficiently. We need to make discovering the correct bucket for a node an $O(1)$ operation. Naïvely, it is easy to create the buckets, but creating the memory structure above requires some care. We synthesize the number of nodes in each bucket in advance using a multinomial distribution [33] to allow memory allocation to be performed once. That makes node creation and allocation to buckets *embarrassingly parallel*. Also, all nodes can be stored in a single contiguous memory block with a separate pointer to the start of each bucket, rather than separate memory for each bucket.

The buckets are straight forward when the SERN is embedded in a square region, but our implementation allows for three types of regions in which to embed the SERN:

- A rectangular region, generalizing the square region initially investigated by Waxman [7];
- An elliptical region allowing investigation of SERNs where there are no corner effects; and
- A user defined polygon allowing real world boundary data to be used.

We also provide efficient random number generators based on [34] that are thread safe, and fast for the types of random variables needed here.

The parallel execution of link generation is more complex than that of node generation because we do not know in advance how many links will be generated and we want to maximize the size of the network that we can create in a given amount of memory. We also want to be able to return the data in a contiguous block of memory so that the MEX code used to incorporate the library into Matlab need only construct appropriate pointers in order to return the data into Matlab's standard data structures, and similarly for linking the code into other high-level tools. Schemes of node generation where each thread writes to its own memory until all links are generated are thus not possible, as they require either much more memory or a juggling act where some sets of data are shrunk and others grown until all data is in a contiguous space. The former is restrictive and the latter likely to end up with a fragmented heap and deadlock because it is not possible to transition to the next state. The implementation chosen is a compromise, with each thread generating the links for a pair of buckets at a time and writing this data into a buffer which, when full, is

written to shared memory using a common pointer to the next available free location.

The current code allows generation of SERNs on rectangles, ellipses and arbitrary polygons on $\mathbb{R}^2$. Our standard wrappers implement the four distance metrics based on the ℓ_p norm for $p = 0, 1, 2$ and ∞, and nine link probability functions. The routines for both bucket and link generation have been parameterized to accept pointers to functions that implement distance and link probabilities, thus creating new models involves only five or six lines of code.

6 Conclusion and Future Work

This paper describes an algorithm to perform fast $O(n + e)$ generation of SERNs. The results from the implementation described show that the performance is several orders of magnitude faster than competing code for large graphs.

Further details of the implementation are available at [11], and the code itself is available at git@github.com: lamestllama/conSERN.git.

Following presentation of the algorithm in [35], a similar idea was exploited to generate a different class of random graphs in [29], so this approach can clearly be extended: *e.g.*,

- Higher-dimensional spaces, and non-Euclidean manifolds.
- Better threading by predicting the amount of links that will be generated and allocating memory in advance.
- The current implementation of polygon shapes can be optimized by recursively finding the intersection of buckets with the defined region.

Acknowledgments

This work was partially supported through ARC grants CE140100049 and DP110103505.

References

[1] L. Barnett, E. Di Paolo, and S. Bullock, "Spatially embedded random networks," *Phys. Rev. E*, vol. 76, p. 056115, Nov. 2007. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.76.056115

[2] K. Kosmidis, S. Havlin, and A. Bunde, "Structural properties of spatially embedded networks," *Europhysics Letters*, vol. 82, no. 4, 2008. [Online]. Available: http://iopscience.iop.org/0295-5075/82/4/48005

[3] C. P. Dettmann and O. Georgiou, "Random geometric graphs with general connection functions," *Phys. Rev. E*, vol. 93, p. 032313, Mar 2016. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.93.032313

[4] E. Gilbert, "Random graphs," *Annals of Mathematical Statistics*, vol. 30, pp. 1441–1144, 1959.

[5] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, vol. 5, pp. 17–61, 1960.

[6] E. N. Gilbert, "Random plane networks," *Journal of the Society for Industrial and Applied Mathematics*, vol. 9, no. 4, pp. 533–543, 1961.

[7] B. Waxman, "Routing of multipoint connections," *IEEE J. Select. Areas Commun.*, vol. 6, no. 9, pp. 1617–1622, 1988.

[8] D. J. Watts, "A simple model of global cascades on random networks," *Proceedings of the National Academy of the Sciences of the United States of America*, vol. 99, no. 9, pp. 5766–5771, 2002.

[9] K. Csiléry, M. G. B. Blum, O. E. Gaggiotti, and O. Francois, "Approximate bayesian computation (ABC) in practice," *Trends in Ecology & Evolution*, vol. 25, pp. 410–418, 2010. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0169534710000662

[10] P. E. Black, "Sparse graph," in Dictionary of Algorithms and Data Structures, National Institute of Standards and Technology (NIST), 2008, https://xlinux.nist.gov/dads//HTML/sparsegraph.html, accessed Feb, 2016.

[11] E. Parsonage and M. Roughan, "Fast generation of spatially embedded random networks," *ArXiv e-prints, 1512.03532*, December 2015.

[12] M. Penrose, *Random Geometric Graphs*. Oxford Studies in Probability, 2003.

[13] P. Grindrod, "Range-dependent random graphs and their application to modeling large small-world proteome datasets," *Phys. Rev. E*, vol. 66, no. 066702, 2002.

[14] A. Farago, "Scalable analysis and design of ad hoc networks via random graph theory," in *Dial-M'02*, 2002.

[15] C. Gunduz-Demir, "Mathematical modeling of the malignancy of cancer using graph evolution," *Mathematical Biosciences*, vol. 209, no. 2, pp. 514–527, 2007.

[16] R. Hekmat and P. V. Mieghem, "Degree distribution and hopcount in wireless ad-hoc networks," in *Networks, 2003. ICON2003*, 2003, pp. 603–609.

[17] C. Avin, "Distance graphs: from random geometric graphs to Bernoulli graphs and between," in *Fifth International Workshop on Foundations of Mobile Computing*, ser. DIALM-POMC '08. New York, NY, USA: ACM, 2008, pp. 71–78. [Online]. Available: http://doi.acm.org/10.1145/1400863.1400878

[18] E. W. Zegura, K. L. Calvert, and M. J. Donahoo, "A quantitative comparison of graph-based models for Internet topology," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 770–783, 1997.

[19] O. Frank and D. Strauss, "Markov graphs," *Journal of the American Statistical Association*, vol. 81, no. 395, pp. 832–842, 1986.

[20] G. Robins, P. Pattison, Y. Kalish, and D. Lusher, "An introduction to exponential random graph (p*) models for social networks," *Social Networks*, vol. 29, no. 2, pp. 173–191, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0378873306000372

[21] C. Gunduz, B. Yener, and S. H. Gultekin, "The cell graphs of cancer," *Bioinformatics*, vol. 20, no. 1, pp. 145–151, 2004.

[22] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: an approach to universal topology generation," in *Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Washington, DC, USA, 2001. [Online]. Available: http://dl.acm.org/citation.cfm?id=882459.882563

[23] D. Magoni, "Nem: A software for network topology analysis and modeling," in *10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, ser. MASCOTS '02. Washington, DC, USA: IEEE Computer Society, 2002. [Online]. Available: http://dl.acm.org/citation.cfm?id=882460.882599

[24] D. Magoni and J.-J. Pansiot, "Influence of network topology on protocol simulation," in *Networking ICN 2001*, ser. Lecture Notes in Computer Science, P. Lorenz, Ed. Springer Berlin / Heidelberg, 2001, vol. 2093, pp. 762–770. [Online]. Available: http://dx.doi.org/10.1007/3-540-47728-4_75

[25] M.-A. Weisser and J. Tomasik, "aSHIIP: autonomous generator of random Internet-like topologies with inter-domain hierarchy," in *18th IEEE Symposium on Modeling Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'10)*, 2010. [Online]. Available: http://wwwdi.supelec.fr/software/ashiip/

[26] J. Lothian, S. Powers, B. D. Sullivan, M. Baker, J. Schrock, and S. W. Poole, "Synthetic graph generation for data-intensive HPC benchmarking: Background and framework," Oak Ridge National Laboratory, Tech. Rep. ORNL/TM-2013/339, October 2013.

[27] M. Roughan, J. Tuke, and E. Parsonage, "Estimating the parameters of the Waxman random graph," 2015, arXiv:1506.07974.

[28] V. Batagelj and U. Brandes, "Efficient generation of large random networks," *Phys. Rev. E*, vol. 71, p. 036113, Mar. 2005. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevE.71.036113

[29] K. Bringmann, R. Keusch, and J. Lengler, "Geometric inhomogeneous random graphs," *ArXiv e-prints, 1511.00576*, November 2015.

[30] B. Ghosh, "Random distance within a rectangle and between two rectangles," *Bulletin of the Calcutta Mathematical Society*, vol. 43, no. 1, pp. 17–24, 1951.

[31] W. Feller, *An Introduction to Probability Theory and its Applications*, 2nd ed. John Wiley and Sons, New York, 1971, vol. I.

[32] ——, *An Introduction to Probability Theory and its Applications*, 2nd ed. John Wiley and Sons, New York, 1971, vol. II.
[33] D. Knuth, "The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching," 1968.
[34] G. Marsaglia, "Random number generation," in *Encyclopedia of Computer Science*. John Wiley and Sons, pp. 1499–1503. [Online]. Available: http://dl.acm.org/citation.cfm?id=1074100.1074752
[35] E. Parsonage and M. Roughan, "Fast generation of spatially embedded random networks," in *International Workshop on Monte Carlo Methods for Spatial Stochastic Systems*, July 2015, acems.org.au/acems_wp/wp-content/uploads/2015/09/Matthew-Roughan.pdf.

Matthew Roughan (M97-SM09) received the Ph.D. degree in applied probability from the University of Adelaide, Adelaide, Australia, in 1994. He joined the School of Mathematical Sciences, University of Adelaide, in 2004. Prior to that, he was with AT&T Labs - Research, Florham Park, NJ. His research interests lie in measurement and modeling of the Internet, and his background is in stochastic modeling.

Eric Parsonage is a M.Phil student in the School of Mathematical Sciences at the University of Adelaide. His interests include structural modelling of computer networks, and routing algebras.