SIAM Review
Vol. 61, No. 3, pp. 549–595





# Coin-Flipping, Ball-Dropping, and Grass-Hopping for Generating Random Graphs from Matrices of Edge Probabilities*

Arjun S. Ramani†
Nicole Eikmeier‡
David F. Gleich§

**Abstract.** Common models for random graphs, such as Erdős–Rényi and Kronecker graphs, correspond to generating random adjacency matrices where each entry is nonzero based on a large matrix of probabilities. Generating an instance of a random graph based on these models is easy, although inefficient, by flipping biased coins (i.e., sampling binomial random variables) for each possible edge. This process is inefficient because most large graph models correspond to sparse graphs where the vast majority of coin flips will result in no edges. We describe some not entirely well-known, but not entirely unknown, techniques that will enable us to sample a graph by finding only the coin flips that will produce edges. Our analogies for these procedures are ball-dropping, which is easier to implement, but may need extra work due to duplicate edges, and grass-hopping, which results in no duplicated work or extra edges. Grass-hopping does this using geometric random variables. In order to use this idea for complex probability matrices such as those in Kronecker graphs, we decompose the problem into three steps, each of which is an independently useful computational primitive: (i) enumerating nondecreasing sequences; (ii) unranking multiset permutations; and (iii) decoding and encoding z-curve and Morton codes and permutations. The third step is the result of a new connection between repeated Kronecker product operations and Morton codes. Throughout, we draw connections to ideas underlying applied math and computer science, including coupon collector problems.

**Audience.** This paper is designed, primarily, for undergraduates who have some experience with mathematics classes at the level of introduction to probability, discrete math, calculus, and programming, as well as for educators teaching these classes. We try to provide pointers to additional background material where relevant—and we also provide links to various courses throughout applied math and computer science to facilitate using this material in multiple places. Our program codes are available through the github repository https://github.com/dgleich/grass-hopping-graphs.

**Key words.** graph sampling, Erdős–Rényi, Kronecker graphs, Morton codes, unranking multisets, coupon collector

**AMS subject classifications.** 05C80, 15A24, 05A15

**DOI.** 10.1137/17M1127132

*Received by the editors April 25, 2017; accepted for publication (in revised form) September 5, 2018; published electronically August 7, 2019.
https://doi.org/10.1137/17M1127132
**Funding:** This work was supported in part by DOE award DE-SC0014543, NSF CAREER CCF-1149756, IIS-1422918, IIS-1546488, and the NSF Center for the Science of Information STC CCF-0939370. The work of the third author was also supported by the DARPA SIMPLEX program and the Sloan Foundation.

†West Lafayette Jr/Sr High School, West Lafayette, IN 47906 (aramani@purdue.edu).
‡Department of Mathematics, Purdue University, West Lafayette, IN 47907 (eikmeier@purdue.edu).
§Department of Computer Science, Purdue University, West Lafayette, IN 47907 (dgleich@purdue.edu).

**Contents**



**1. Introduction and Motivation for Fast Random Graph Generation.** The utility of a random graph is akin to the utility of a random number. Random numbers, random variables, and the framework of statistical hypothesis testing provide a con-

| | ⚀ | ⚁ | ⚂ | ⚃ | ⚄ | ⚅ | |
|---|---|---|---|---|---|---|---|
| Measured | 30 | 32 | 33 | 31 | 29 | 25 | |
| Expected | 30 | 30 | 30 | 30 | 30 | 30 | |
| Statistic | 0 | 2/15 | 3/10 | 1/30 | 1/30 | 5/6 | 4/3 |

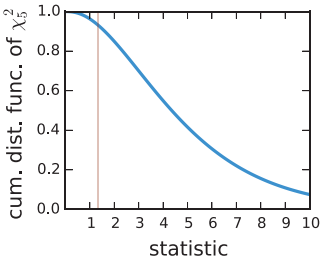



**Fig. 1** *An example of statistical hypothesis testing in the classic scenario with a closed form solution. Here, the closed form $\chi^2$ test works to determine if a dice is "fair". In this case, we conclude there is a 93% chance the dice is fair. For random graph models, we often have no analytic way to determine the quantities akin to the* expected *number or the values from the cumulative distribution function. Thus, we need to* quickly *generate a large number of random graphs to generate simulation based estimates instead.*

venient way to study and assess whether an apparent signal or pattern extracted from noisy data is likely to be real or the result of chance. In this setting, a random variable models the null-hypothesis where the effect is due to chance alone. By way of example, testing whether a standard six-sided dice is "fair" involves a random variable with a $\chi^2$ distribution. The way this test works is illustrated in Figure 1. We compare the number of times the dice rolls each number with the expected number of times under the *fair* hypothesis. For each outcome, such as ⚂, we compute $(\text{Measured} - \text{Expected})^2/\text{Expected}$. Then we sum these over all outcomes. The resulting number is a *statistic* called Pearson's cumulative test statistic. (In the figure, this is 4/3.) The test statistic is also a random variable because if we rolled the dice another set of times, we could get a different value. A random variable with a $\chi^2$ distribution is the behavior of this test statistic *when the dice is exactly fair*.

The famous $p$-value is the probability that a fair die would give rise to a test statistic value *greater* than the *computed* test statistics (which is 4/3). In other words, we look at the probability that a value from a $\chi^2$ distribution is bigger than 4/3. The resulting probability ($p = 0.931$) provides a convenient estimate of how likely the dice is to be fair. In this case, we computed the probability using an analytic description of the $\chi^2$ distribution (strictly speaking, this test involves the $\chi^2$ distribution with five degrees of freedom). The probability should be large if the dice is fair and small if it is unlikely to be fair. Often, the probability $p = 0.05$ is chosen arbitrarily as a cut-point to distinguish the two cases ($p \geq 0.05$ means fair, and $p < 0.05$ means maybe not fair), but this value should be used judiciously.

ASIDE 1. *This type of analysis and methodology is called statistical hypothesis testing and can be extended to more complex scenarios. Estimating these distributions from test statistics results in studies which are fascinating in their own right.*

Statistical hypothesis testing on graphs is largely the same (Moreno and Neville, 2013; Koyutürk, Szpankowski, and Grama, 2007; Koyutürk, Grama, and Szpankowski, 2006; Milo et al., 2002). We use a random graph model as the null hypothesis, which is akin to the *fair* hypothesis used in the example above. Then we measure something on a real-world graph and compare what we would have expected to get on a set of random graphs. Consider, for instance, the study by Milo et al. (2002) on the presence of *motifs* in networks. This study defines network motifs as patterns of connectivity that are found more frequently in a network than might be expected in a random network. In

social networks such as Facebook, a common motif is a triangle. If you have two friends, then it is much more likely that your friends are also friends, which forms a triangle structure in the graph. This property does not exist in many random models of networks. One critical difference from traditional statistics, however, is that we often lack closed-form representations for the probability of many events in random graph models. There was *no known expression* for the number of motifs that Milo should expect, unlike the case of the dice and the $\chi^2$ distribution. In fact, in the studies above, only (Koyutürk, Grama, and Szpankowski, 2006; Koyutürk, Szpankowski, and Grama, 2007) had closed-form solutions.

Aside 2. *In fact, creating random graph models with the property that they have a nontrivial number of triangles is an active area of study! Recent work in this vein includes Kolda et al.* (2014)*; Newman* (2009).

Thus, we turn to empirical simulations. One of the easiest and most common methods to do these studies (such as Milo et al. (2002)) leverages the computer to generate hundreds or hundreds of thousands of instances of a random graph and compute their properties. In comparison with the dice example, if we had access to a dice that was guaranteed to be fair, we could have generated an empirical distribution for Pearson's cumulative test statistic by tossing the fair dice a few thousand times. Then, an estimate of the probability that the statistic is larger could be computed just by checking the number of instances where the sample was larger. To do this, we would need a *fast* way of tossing dice! For random graphs, it almost goes without saying that we also need this random graph generation, which is also called random graph sampling, to be as fast and efficient as possible.

Aside 3. *Fisher* (1936) *mentions this "trivial" toss-dice-and-check idea as the basis of all rigorous work in the area of hypothesis testing. More recently, as the power of computation has grown, others including Ernst* (2004) *have suggested renewed focus on these empirical and computational tests rather than analytic results.*

Statistical hypothesis testing is not the only use for random graphs. There are a variety of other scenarios that also require fast sampling such as benchmarking high performance computers (Murphy et al., 2010) and synthetic data to evaluate algorithms (Kloumann, Ugander, and Kleinberg, 2016). As we shall see, there are a variety of efficient methods that result in slightly biased or approximate samples from these graph models. These could be appropriate in a variety of scenarios including those above, but it is important to understand the nature of the bias or approximation to avoid drawing the wrong conclusion from the computational experiments. This can often be challenging as there are subtle biases that can easily arise; we'll see an example in section 5.1. In this paper, we focus on *exactly sampling the true random graph model* as a simple alternative.

In this paper, we will explain and illustrate how to generate random graphs efficiently when the random graph model is described by a matrix of probabilities for each edge. This is admittedly a special case, but it handles some of the most widely used random graph distributions:

- *the Erdős–Rényi model* (section 2.3),
- *the Kronecker model* (section 2.6),
- *the Chung–Lu model* (section 2.7), and
- *the stochastic block model* (section 2.8).

We will pay special attention to some of the structures present in the matrices of probabilities of stochastic Kronecker graphs and how to take advantage of them to make the process go faster. Along the way, we'll see examples of a number of classic discrete structures and topics:

- *binomial and geometric random variables*,
- *the coupon collector problem*,
- *enumerating nondecreasing sequences*,
- *unranking multiset permutations*, and
- *Morton and Z-curves*.

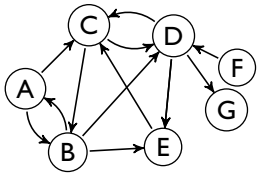
Throughout, we are going to explain these concepts and ideas both mathematically and programmatically. We include runnable Python code in the paper to make it easy to follow along; see https://github.com/dgleich/grass-hopping-graphs/ for our code.

**2. Random Graph Models and Matrices of Probabilities.** A graph consists of a set of vertices $V$ and a set of edges $E$, where each edge $e \in E$ encodes relationships between two vertices, which is often written $e = (u, v)$ for $u, v \in V$. We consider the more general setting of directed graphs in this paper, although we describe how all of the ideas specialize to undirected graphs in section 2.9. A random graph consists of a set of random edges between a fixed number of vertices. How should these random edges be chosen? That turns out to depend on the particular *random graph model*.

The models we consider generate a *random adjacency matrix* where each entry $A_{ij}$ in the adjacency matrix is 0 or 1 with probability $P_{ij}$ for some given matrix of probabilities $P$. Let's dive into these details to understand exactly what this means.

ASIDE 4. *There are more general definitions of random graphs that do not impose a fixed number of vertices, such as preferential attachment (Barabási and Albert,* 1999*), but this simple setting will serve our purposes.*

**2.1. The Adjacency Matrix.** The adjacency matrix encodes the information of the nodes and edges into a matrix. For instance,

if the graph is 

, the adjacency matrix is

$$\begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \\ G \end{array} \begin{array}{c} \begin{array}{ccccccc} A & B & C & D & E & F & G \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}.$$

Formally, the adjacency matrix is created by assigning each vertex $v \in V$ a unique number between 1 and $|V|$ often called an index. Then each edge $e = (u, v) \in E$ produces an entry of 1 in the coordinate of the matrix that results in mapping $u$ and $v$ to their indices. That is, if $i$ and $j$ are the indices of $u$ and $v$, then entry $i, j$ of the matrix has value 1. All other entries of the matrix are 0. In the case above, we mapped $A$ to index 1, $B$ to index 2, and so on.

**2.2. A Random Adjacency Matrix as a Random Graph.** Note that this process can go the other way as well. If we have any $n$-by-$n$ matrix where each entry is 0 or 1, then we can interpret that matrix as the adjacency matrix of a graph! What we are going to do is (i) generate a random matrix with each entry being 0 or 1 and (ii) interpret that matrix as a set of random edges to give us the random graph.

At this point, we need to mention a distinction between *directed* and *undirected graphs*. By convention, an undirected graph has a symmetric adjacency matrix where $A_{ij} = 1$ and $A_{ji} = 1$ for each undirected edge. Our focus will be on generating random *nonsymmetric* matrices $\boldsymbol{A}$. These techniques will still apply to generating symmetric graphs, however. For instance, we can interpret a nonsymmetric $\boldsymbol{A}$ as an undirected graph by only considering entries $A_{ij}$ where $i < j$ (or, equivalently, $A_{ij}$ where $i > j$) and then symmetrizing the matrix given one of these triangular regions. (In MATLAB, this would be: `T = triu(A,1); G = T+T'`; in Python, it would be `T=np.triu(A,1); G=T+T.T`.) We return to this point in section 2.9.

Aside 5. *In fact, there are many aspects of sparse matrix computations that are most easily understood by viewing a sparse matrix as a graph and computing a graph algorithm such as breadth-first search (Davis,* 2006*).*

**2.3. The Erdős–Rényi Model.** Perhaps the first idea that comes to mind at this point is tossing a coin to determine whether an entry in the adjacency matrix should be 0 or 1. While a simple 50%-50% coin toss is suitable for randomly choosing between the two, there is no reason we need 1's (the edges) to occur with the same probability as 0's (the nonedges). Let $p$ be the probability that we generate a 1 (an edge) and let

$$A_{ij} = \begin{cases} 1 & \text{with probability } p \\ 0 & \text{with probability } 1-p \end{cases} \quad \text{for each } i,j \text{ in an } n\text{-by-}n \text{ matrix.} \tag{2.1}$$

This model is called the Erdős–Rényi model. It is one of the simplest types of random graphs and has $n$ nodes, where any 2 nodes have probability $p$ of being connected with a directed edge. Erdős–Rényi graphs are traditionally constructed by considering each edge separately and using the method of coin-flipping, which will be discussed in section 2.5.

Aside 6. *The Erdős–Rényi graph as we consider it was actually proposed by Edgar Gilbert (Gilbert,* 1959*) at the same time at Erdős and Rényi's famous paper (Erdős and Rényi,* 1959*). Erdős and Rényi proposed a slight variation that fixed the number of edges and placed them uniformly at random.*

**2.4. Random Graph Models as Matrices of Probabilities.** Now, you might be wondering why we use only a *single* probability $p$ for each edge in the Erdős–Rényi model. The random graph models we study here have an entry $P_{ij}$ for each $i, j$, which is a more general setting than the Erdős–Rényi construction:

$$A_{ij} = \begin{cases} 1 & \text{with probability } P_{ij} \\ 0 & \text{with probability } 1-P_{ij} \end{cases} \quad \text{for each } i,j \text{ in an } n\text{-by-}n \text{ matrix.} \tag{2.2}$$

Thus, if we set $P_{ij} = p$, then this more general model corresponds to the Erdős–Rényi model. Now, of course, this begs two questions: How do we choose $P_{ij}$? How do we generate a random matrix $\boldsymbol{A}$? We will describe three common random graph models that consist of a choice of $P_{ij}$: the Kronecker model, the Chung–Lu model, and the stochastic block models. We will also describe the coin-flipping method of sampling a random graph at this point. However, in subsequent sections we will show how to sample each of these models more efficiently than using the coin-flipping method.

**2.5. The Coin-Flipping Method for Sampling a Random Graph.** Given the matrix of probabilities $\boldsymbol{P}$, the easiest way to generate a random adjacency matrix where $A_{ij}$ is 1 with probability $P_{ij}$ is to explicitly simulate biased coin flips (which

**Listing 1** A simple code to generate a random graph by coin flipping

```
import random       # after this, random.random() gives a uniform [0,1] value
import numpy as np  # numpy is the Python matrix package
"""
Generate a random graph by coin flipping. The input is a square matrix P with entries
between 0 and 1. The result is a 0 or 1 adjacency matrix for the random graph.

Example:
  coinflip(0.25*np.ones((8,8))) # generate a random Erdos-Renyi graph
"""
def coinflip(P):
  n = P.shape[0]
  assert(n == P.shape[1]) # make sure we have a square input
  A = np.zeros_like(P)    # create an empty adjacency matrix
  for j in range(n):
    for i in range(n):    # fill in each entry as 1 with prob P[i,j]
      A[i,j] = random.random() <= P[i,j]
  return A
```

are also Bernoulli random variables). A pragmatic way to do this is to use a random number generator that produces a uniform distribution of values between 0 and 1. This is pragmatic because most computer languages include such a routine and make sure it is efficient. Given such a random value $\rho$, we set $A_{ij}$ to 1 if $\rho \leq P_{ij}$ (which happens with probability $P_{ij}$). An example of this is shown in Listing 1. Note that the indices in Python range from 0 to $n-1$ rather than from 1 to $n$ as is common in mathematical descriptions of matrices.

ASIDE 7. *Computers typically use pseudorandom number generators, which have their own fascinating field (Gentle,* 2003*; Knuth,* 1997*).*

**2.6. The Kronecker Model.** The Kronecker random graph model results in a nonuniform but highly structured probability matrix $\boldsymbol{P}$. It begins with a small initiator matrix $\boldsymbol{K}$ with $n$ nodes which is then enlarged into a bigger matrix of probabilities $\boldsymbol{P}$ by taking successive Kronecker products (Chakrabarti, Zhan, and Faloutsos, 2004; Leskovec et al., 2005, 2010). For an example, suppose $\boldsymbol{K}$ is a 2-by-2 initiator matrix

$$\boldsymbol{K} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \tag{2.3}$$

where each $a, b, c, d$ is a probability. The Kronecker product of $\boldsymbol{K}$ with itself is given by

$$\boldsymbol{K} \otimes \boldsymbol{K} = \begin{bmatrix} a \cdot \boldsymbol{K} & b \cdot \boldsymbol{K} \\ c \cdot \boldsymbol{K} & d \cdot \boldsymbol{K} \end{bmatrix} = \begin{bmatrix} aa & ab & ba & bb \\ ac & ad & bc & bd \\ ca & cb & da & db \\ cc & cd & dc & dd \end{bmatrix}.$$

Finally, the $k$th Kronecker product of $\boldsymbol{K}$ is just

$$\underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \cdots \otimes \boldsymbol{K}}_{k \text{ times}}, \tag{2.4}$$

which is a $2^k$-by-$2^k$ matrix. As a concrete example, let $\boldsymbol{K} = \left[\begin{smallmatrix} 0.99 & 0.5 \\ 0.5 & 0.2 \end{smallmatrix}\right]$; then

(2.5)
$$\boldsymbol{K}\otimes\boldsymbol{K} = \begin{bmatrix} 0.9801 & 0.495 & 0.495 & 0.25 \\ 0.495 & 0.198 & 0.25 & 0.1 \\ 0.495 & 0.25 & 0.198 & 0.1 \\ 0.25 & 0.1 & 0.1 & 0.04 \end{bmatrix}, \quad \boldsymbol{K}\otimes\boldsymbol{K}\otimes\boldsymbol{K} = \begin{bmatrix} 0.97 & 0.49 & 0.49 & 0.25 & 0.49 & 0.25 & 0.25 & 0.13 \\ 0.49 & 0.20 & 0.25 & 0.10 & 0.25 & 0.10 & 0.13 & 0.05 \\ 0.49 & 0.25 & 0.20 & 0.10 & 0.25 & 0.13 & 0.10 & 0.05 \\ 0.25 & 0.10 & 0.10 & 0.04 & 0.13 & 0.05 & 0.05 & 0.02 \\ 0.49 & 0.25 & 0.25 & 0.13 & 0.20 & 0.10 & 0.10 & 0.05 \\ 0.25 & 0.10 & 0.13 & 0.05 & 0.10 & 0.04 & 0.05 & 0.02 \\ 0.25 & 0.13 & 0.10 & 0.05 & 0.10 & 0.05 & 0.04 & 0.02 \\ 0.13 & 0.05 & 0.05 & 0.02 & 0.05 & 0.02 & 0.02 & 0.01 \end{bmatrix}.$$

We use these $k$th Kronecker products as the matrix of probabilities for the random graph. This gives us (in general) an $n^k$-by-$n^k$ matrix of probabilities $\boldsymbol{P}$ for an $n^k$-node random graph. It quickly becomes tedious to write out these matrices by hand. As might be guessed from the small set of parameters underlying it, there *is structure* inside of this matrix of repeated Kronecker products, and we will return to studying and exploiting its patterns in section 5.

There are a variety of motivations for repeated Kronecker products as a graph model. On the modeling side, the idea underlying Kronecker graphs is that a graph that is twice as big should look like a perturbed version of the current graph. This type of self-similarity is a commonly assumed feature of real-world networks (Ravasz and Barabási, 2003; Dill et al., 2002; Leskovec et al., 2010). On the practical side, Kronecker graphs are extremely parsimonious and require only the entries of a small $n$-by-$n$ matrix, where $n$ is between 2 and 5. Second, they can generate a variety of graphs of different sizes by adjusting $k$. Third, the graphs they produce have a number of highly skewed properties (Seshadhri, Pinar, and Kolda, 2013). These reasons make the Kronecker model a useful synthetic network model for various real-world performance studies (Murphy et al., 2010). On the statistical side, Kronecker models have some of the same properties as real-world networks (Leskovec et al., 2010). As such, they provide nontrivial null models.

**2.7. The Chung–Lu model.** Recall that the degree, $d_u$, of a vertex $u$ is just the number of edges leaving $u$. For example, in the graph from section 2.1, the degrees of the nodes in alphabetical order are $(2, 3, 2, 3, 1, 1, 0)$. Many important and interesting features of a graph are fundamentally connected to the degrees of the vertices (Adamic et al., 2001; Litvak, Scheinhardt, and Volkovich, 2006), and this fact is the motivation for the Chung–Lu random graph model. We wish to have a random graph with vertices of roughly the same degree as the network we are studying to understand whether the properties we observe (in the real network) are due to the degrees or to the network structure. This is almost exactly what Milo et al. (2002) did when they wanted to understand if a motif pattern was significant.

In the Chung–Lu model, we need, as input, the desired degree of each vertex in the resulting random graph. For example, say we want to generate an $n$-by-$n$ graph where vertex 1 has degree $d_1$, vertex 2 has degree $d_2$, and so on. Specifically, suppose we want an eight vertex network with one node of degree 4, one node of degree 3, three nodes of degree 2, and three nodes of degree 1. The corresponding matrix is

$$\boldsymbol{P} = \begin{bmatrix} 1.00 & 0.75 & 0.50 & 0.50 & 0.50 & 0.25 & 0.25 & 0.25 \\ 0.75 & 0.56 & 0.38 & 0.38 & 0.38 & 0.19 & 0.19 & 0.19 \\ 0.50 & 0.38 & 0.25 & 0.25 & 0.25 & 0.13 & 0.13 & 0.13 \\ 0.50 & 0.38 & 0.25 & 0.25 & 0.25 & 0.13 & 0.13 & 0.13 \\ 0.50 & 0.38 & 0.25 & 0.25 & 0.25 & 0.13 & 0.13 & 0.13 \\ 0.25 & 0.19 & 0.13 & 0.13 & 0.13 & 0.06 & 0.06 & 0.06 \\ 0.25 & 0.19 & 0.13 & 0.13 & 0.13 & 0.06 & 0.06 & 0.06 \\ 0.25 & 0.19 & 0.13 & 0.13 & 0.13 & 0.06 & 0.06 & 0.06 \end{bmatrix}.$$

This matrix results from setting

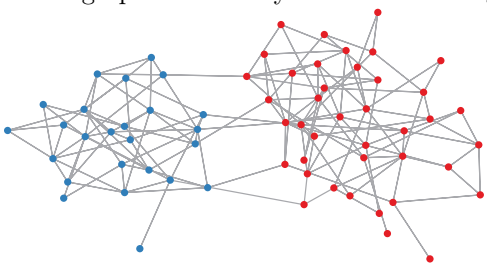
(2.6)
$$P_{ij} = \frac{d_i d_j}{\sum_k d_k}.$$

To understand why this is a good choice, let's briefly consider the *expected degree* of vertex $i$. In the adjacency matrix, we can compute the degree by taking the sum of all entries in a row. Here we have, in expectation,

$$\mathrm{E}\left[\sum_j A_{ij}\right] = \sum_j \mathrm{E}[A_{ij}] = \sum_j P_{ij} = \sum_j \frac{d_i d_j}{\sum_k d_k} = d_i. \tag{2.7}$$

This analysis shows that, in expectation, this choice of probabilities results in a random graph with the correct degree distribution.

**2.8. The Stochastic Block Model.** Another feature of real-world networks is that they have *communities* (Flake, Lawrence, and Giles, 2000; Newman and Girvan, 2004). A community is a group of vertices that are more tightly interconnected than they are connected to the rest of the graph. Pictorially this results in a graph such as



where there are two communities: blue and red. The stochastic block model is designed to mirror this structure in a random graph. Suppose we want a random graph with two communities with $n_1$ and $n_2$ nodes, respectively. We want there to be a high probability, $p$, of edges within a community, and a low probability of edges between the communities, $q < p$. Formally, this corresponds to a probability matrix

$$P_{ij} = \begin{cases} p & \text{if } i,j \text{ are in the same community,} \\ q & \text{if } i,j \text{ are in different communities,} \end{cases} \quad \text{where } p > q. \tag{2.8}$$

The reason this is called the stochastic block model is that it can be written with a set of *block* matrices. Consider $n_1 = 3$, $n_2 = 5$, $p = 0.7$, and $q = 0.1$:

$$\boldsymbol{P} = \left[\begin{array}{c|c} p & q \\ \hline q & p \end{array}\right] = \left[\begin{array}{ccc|ccccc} 0.7 & 0.7 & 0.7 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.7 & 0.7 & 0.7 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ 0.7 & 0.7 & 0.7 & 0.1 & 0.1 & 0.1 & 0.1 & 0.1 \\ \hline 0.1 & 0.1 & 0.1 & 0.7 & 0.7 & 0.7 & 0.7 & 0.7 \\ 0.1 & 0.1 & 0.1 & 0.7 & 0.7 & 0.7 & 0.7 & 0.7 \\ 0.1 & 0.1 & 0.1 & 0.7 & 0.7 & 0.7 & 0.7 & 0.7 \\ 0.1 & 0.1 & 0.1 & 0.7 & 0.7 & 0.7 & 0.7 & 0.7 \\ 0.1 & 0.1 & 0.1 & 0.7 & 0.7 & 0.7 & 0.7 & 0.7 \end{array}\right]. \tag{2.9}$$

Notice that each of the blocks is just an Erdős–Rényi matrix. The model can be extended to an arbitrary number of subsets and the values of $p$ and $q$ can be varied between different subsets. In the more general case, let $Q_{rs}$ denote the probability of an edge between a node in the $r$th block and one in the $s$th block. Then the block

adjacency matrix will look like

$$\begin{array}{c} \\ n_1 \\ n_2 \\ \\ n_k \end{array} \begin{array}{c} \begin{array}{cccc} n_1 & n_2 & & n_k \end{array} \\ \left[\begin{array}{c|c|c|c} Q_{11} & Q_{12} & \cdots & Q_{1k} \\ \hline Q_{21} & Q_{22} & & \\ \hline \vdots & \vdots & \ddots & \\ \hline Q_{k1} & Q_{k2} & & Q_{kk} \end{array}\right] \end{array}.$$

**2.9. Undirected Graphs.** The focus of our paper is on generating random graphs. To do this, we generate a random binary and nonsymmetric adjacency matrix $\boldsymbol{A}$. However, many studies on real-world data start with undirected graphs with symmetric matrices of probabilities $\boldsymbol{P}$. Thus, we would like the ability to generate undirected graphs! We have three ways to get an undirected graph (and its symmetric adjacency matrix) from these nonsymmetric adjacency matrices.

The first method has already been mentioned: take the upper or lower triangular piece of the nonsymmetric matrix $\boldsymbol{A}$ and explicitly symmetrize it (see section 2.4).

The second method builds upon the first. Recall that in many scenarios discussed in the introduction, we are interested in generating a large number of random graphs from the same distribution. In this case, we note that a nonsymmetric adjacency matrix $\boldsymbol{A}$ that results from a symmetric matrix $\boldsymbol{P}$ will give us *two* samples of a random graph! One is from the lower-triangular entries, and one from the upper-triangular entries. *This is the easiest and most pragmatic way to proceed if you are generating multiple samples, as all of the code from this paper directly applies.*

The third method proceeds by only generating edges in the upper-triangular region itself. For instance, in Listing 1, we could easily restrict the `for` loops to $i < j$, and set $A_{ji} = 1$ whenever we set $A_{ij} = 1$. This same strategy, however, will become much more complicated when we look at some of the accelerated generation schemes in subsequent sections. This is possible, and it is an exercise worth exploring for those interested, but we deemed the added layer of complexity to be too high for this particular paper. Besides, the second method above is likely to be more useful in scenarios where multiple samples are needed and it is also very easy to implement.

**3. Efficiently Generating Edges for Erdős–Rényi Graphs: Ball-Dropping and Grass-Hopping.** Recall that in an Erdős–Rényi graph, all nodes have probability $p$ of being connected. As previously explained, the simplest way to generate these edges involves flipping $n^2$ weighted coins, or "coin-flipping." The problem with coin-flipping is that it is extremely inefficient when generating an Erdős–Rényi model of a real-world network. A common characteristic of real-world networks is that most pairs of edges do not exist. Just consider how many friends you have on Facebook or followers on Twitter: It is likely to be an extremely small fraction of the billions of people present on these social networks. This makes most of the resulting adjacency matrices entries equal to zero, which results in something called a *sparse matrix*. Coin-flipping is an expensive procedure

ASIDE 8. *We say a matrix is* sparse *if the number of zero entries is so large that it makes sense just to store information on the locations of the entries that are nonzero.*

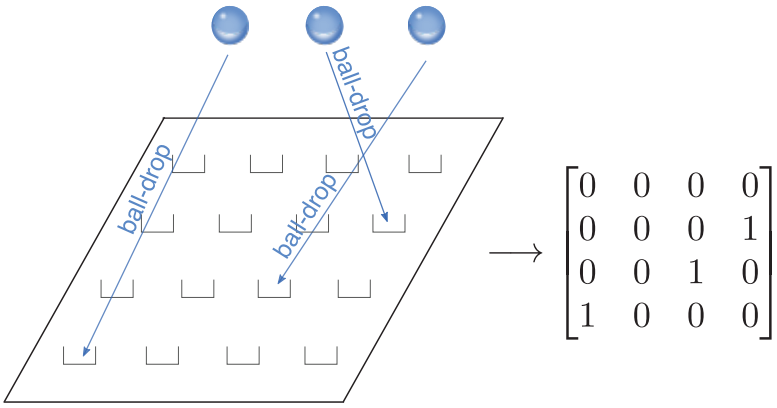



**Fig. 2** *A simple illustration of the ball-dropping procedure to generate an Erdős–Rényi graph. "Balls" are dropped with equal probability, and wherever balls are dropped, we generate an edge in the adjacency matrix.*

for this scenario because it requires a coin flip for every conceivable pair of nodes given by each entry of the matrix. Our goal is a method that does work proportional to the number of *edges* of the resulting network.

**3.1. Ball-Dropping.** The first accelerated procedure we'll consider is what we call ball-dropping. The inspiration behind this method is that we can easily "drop a ball" into a uniform random entry in a large matrix, which results in a random edge; see Figure 2. Careful and repeated use of this procedure will allow us to generate a random adjacency matrix where nonedges are never considered! This solves the efficiency problem that motivates our accelerated sampling procedures.

The ball-dropping process itself is quite simple:

1. generate a uniform random integer between 1 and $n$ for $i$;
2. generate a uniform random integer between 1 and $n$ for $j$.

The probability that we generate any entry $i, j$ is then $1/n \cdot 1/n = 1/n^2$, which is a uniform distribution over entries of the matrix. Two questions arise: (i) how many balls (or edges) do we drop, and (ii) what do we do about duplicates?

The solution to question (i) can be resolved by *binomial random variables*. The number of balls that we want to drop is given by the number of edges we will generate in an Erdős–Rényi graph. Recall that each edge or entry in $A_{ij}$ is the result of a random, independent coin flip or Bernoulli trial. The total number of edges is the number of successes in these coin flips, which is exactly what a binomial random variable describes. Thus, the number of edges in an Erdős–Rényi graph is a binomial random variable with $n^2$ trials and probability $p$ of success. Many standard programming libraries include routines for sampling binomial random variables, which makes finding the number of balls to drop an easy calculation.

ASIDE 9. *The approximate distribution that results from ignoring duplicate ball drops may still be useful. This choice induces a slightly different model. If these differences are unlikely to be relevant to the desired use, such as testing system performance, then these approximate models are a great choice. Making this judgment requires care in understanding the impact.*

That leaves question (ii): what do we do about duplicate ball drops? Suppose the sample from the binomial distribution specifies $m$ edges. Each ball drop is *exactly* the

**Listing 2** A simple code to generate an Erdős–Rényi random graph by ball dropping

```
import random      # random.randint(a,b) gives a uniform int from a to b
import numpy as np # numpy is the Python matrix package
"""
Generate a random Erdos-Renyi graph by ball-dropping. The input is:
   n: the number of nodes
   p: the probability of an edge
The result is a list of directed edges.

Example:
  ball_drop_er(8,0.25) # 8 node Erdos-Renyi with probability 0.25
"""
def ball_drop_er(n,p):
    m = int(np.random.binomial(n*n,p))      # the number of edges
    edges = set()                           # store the set of edges
    while len(edges) < m:
        # the entire ball drop procedure is one line, we use python indices in 0,n-1 here
        e = (random.randint(0,n-1),random.randint(0,n-1))
        if e not in edges:                  # check for duplicates
            edges.add(e)                    # add it to the list
    return list(edges)                      # convert the set into a list
```

procedure described above and we make no provisions to avoid duplicate entries. One strategy would be to ignore duplicate ball drops. While this is expedient, it samples from a different distribution over graphs than Erdős–Rényi does, since less than $m$ unique edges will likely be generated. The alternative is to discard duplicate ball drops and continue dropping balls until we have exactly $m$ distinct entries. This method gives the correct Erdős–Rényi distribution, and Listing 2 implements this procedure. This procedure returns the edges of a random graph instead of the adjacency matrix to support the *sparse* use case that motivates our accelerated study.

*Proof that this procedure exactly matches the Erdős–Rényi description.* Above we asserted that this procedure actually generates an Erdős–Rényi graph where each edge $i, j$ occurs with probability $p$. This is easy to prove and is well known, although proofs can be difficult to find. We follow the well-written example by Moreno et al. (2014). In order for an edge $i, j$ to occur, it must do so after we have picked $m$, the number of edges from the binomial. Thus,

$$\text{Prob}[A_{ij} = 1] = \sum_{m=0}^{n^2} \text{Prob}[A_{ij} = 1 \mid \boldsymbol{A} \text{ has } m \text{ edges}] \cdot \text{Prob}[\boldsymbol{A} \text{ has } m \text{ edges}]. \tag{3.1}$$

The first term, $\text{Prob}[A_{ij} = 1 \mid \boldsymbol{A}$ has $m$ edges$]$, is equal to $m/n^2$ because we are sampling without replacement. The second term $\text{Prob}[\boldsymbol{A}$ has $m$ edges$]$ is exactly a binomial distribution. Hence,

$$\begin{aligned} \text{Prob}[A_{ij} = 1] &= \sum_{m=0}^{n^2} (m/n^2) \cdot \binom{n^2}{m} p^m (1-p)^{n^2-m} \\ &= (1/n^2) \underbrace{\sum_{m=0}^{n^2} m \binom{n^2}{m} p^m (1-p)^{n^2-m}}_{= \text{ expectation of binomial}} = (1/n^2) \cdot n^2 p = p. \end{aligned} \tag{3.2}$$

∎

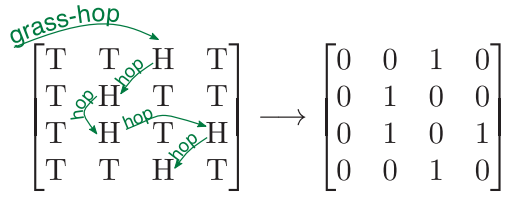



**Fig. 3** *An illustration of grass-hopping. The idea with grass-hopping is that we can sample geometric random variables to move between the coin flips that produce edges (the heads) directly and skip over all the tails.*

The downside with this method is that we do not have a precise runtime because the algorithm continues to generate edges until the number of unique edges is exactly $m$. We analyze this case further in section 3.3.

**3.2. Grass-Hopping.** The second accelerated method we present is what we have chosen to call *grass-hopping*. Note that in the real-world case, there will be many coin flips that come up as "no edge" repeatedly. The essence of the idea is that we wish to "grass hop" from edge to edge as we conceptually (but not actually) flip all $n^2$ coins in the coin-flipping method. Ideally, we'd like to "hop" over all of these "no edge" flips as illustrated in Figure 3. Is such a task possible?

Indeed it is. For a Bernoulli random variable with probability $p$, the number of consecutively failed trials can be derived from a geometric random variable (see more on these below). More specifically, a geometric random variable models the number of trials needed to obtain the next success in a series of coin flips with fixed probability. Thus, by sampling a geometric random variable, we can "grass hop" from success to success and skip over all of the failed trials. Put more algorithmically, we first generate a geometric variable to "hop" to and determine the index of the first edge, then generate a geometric variable for the next gap until a subsequent edge, and repeat this process until we have "hopped" past all the pairs of nodes. A code implementing this idea is given in Listing 3. Note that because the number of trials is limited to $n^2$, we stop sampling once the sum of our geometric random variables is greater than $n^2$.

This idea and method are also known, but not commonly described in the context of graph generation (and we still encounter many individuals unaware of it!). We discuss some of the history in section 3.4, where the core idea is traced back to Pascal and Fermat, and to the 1960s in terms of modern terminology. We first learned of it while studying the source code for the Boost Graph Library implementation of Erdős–Rényi graph generation (Siek, Lee, and Lumsdaine, 2001), and the first reference in terms of graphs dates to the 1980s (Devroye, 1986).

**Some Useful Properties of Geometric Random Variables.** A geometric random variable is a discrete random variable parameterized by the value $p$, which is the success probability of the Bernoulli trial. Let $X$ be a geometric random variable with probability $p$, the probability distribution function $\text{Prob}[X = k] = (1-p)^{k-1}p$. Note that this is just the probability that a coin comes up tails $k-1$ times in a row before coming up heads in the last trial. The expected value of $X$ is $1/p$. A straightforward calculation shows the variance to be $\frac{1-p}{p^2}$. Geometric random variables are a very useful tool and more discussion can be found in a probability textbook such as (Grinstead and Snell, 2012).

**Listing 3** A simple code to generate an Erdős–Rényi random graph by grass-hopping

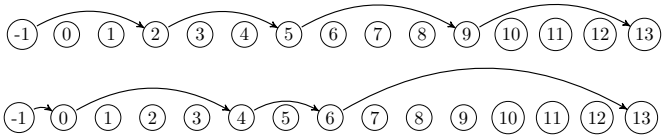
```
import numpy as np # numpy is the Python matrix package and np.random.geometric
                   # is a geometric random variate generator
"""
Generate a random Erdos-Renyi graph by grass-hopping. The input is:
   n: the number of nodes
   p: the probability of an edge
The result is a list of directed edges.

Example:
  grass_hop_er(8,0.25) # 8 node Erdos-Renyi with probability 0.25
"""
def grass_hop_er(n,p):
    edgeindex = -1                    # we label edges from 0 to n^2-1
    gap = np.random.geometric(p)      # first distance to edge
    edges = []
    while edgeindex+gap < n*n:        # check to make sure we have a valid index
        edgeindex += gap              # increment the index
        src = edgeindex // n          # use integer division, gives src in [0, n-1]
        dst = edgeindex - n*src       # identify the column
        edges.append((src,dst))
        gap = np.random.geometric(p) # generate the next gap
    return edges
```

*Proof that grass-hopping is correct.* The proof that grass-hopping is correct is essentially just the result that the geometric random variable models the gaps between successes. However, in the interest of exposition, we present a proof that $\text{Prob}[A_{ij} = 1] = p$ explicitly. In the adjacency matrix of the graph, the probability that the $i, j$ cell will have an edge can be calculated by adding the probabilities of all sequences of hops that land on that cell. Let $\ell$ be the index of the $i, j$ cell in the linear order from 0 to $n^2 - 1$ used in the code. The probability that $A_{ij} = 1$ is then the probability that the sequence of hops lands on $\ell$ or, equivalently, is simply a series of geometric random variables whose sum is equal to $\ell + 1$. (Here, we have an *off-by-one* change because the indices range from 0 to $n^2 - 1$, but the sums of gaps range from 1 to $n^2$.)

From the total probability theorem, the probability that an edge will be generated is the sum of the probabilities of all length-$k$ hop paths where $k$ goes from 1 to $\ell$+1:

$$\begin{aligned}\text{Prob}[A_\ell = 1] = \text{Prob}[A_\ell = 1 \text{ in 1 hop}] + \text{Prob}[A_\ell = 1 \text{ in 2 hops}] + \cdots \\ + \text{Prob}[A_\ell = 1 \text{ in } \ell + 1 \text{ hops}].\end{aligned} \tag{3.3}$$

There are a number of ways to land on index $l$ by $k$ hops. For example, here are two different length $k = 4$ hops landing on index $l = 13$:




Let $q = 1 - p$. Notice that the probability of the first path is

$$\underbrace{(q^2p)}_{\text{hop 1}}\underbrace{(q^2p)}_{\text{hop 2}}\underbrace{(q^3p)}_{\text{hop 3}}\underbrace{(q^3p)}_{\text{hop 4}} = q^{10}p^4.$$

The probability of the second path is $(p)(q^3p)(qp)(q^6p) = q^{10}p^4$. The fact that these two are equal is not a coincidence. In fact, the probability of a hop path with $k$ hops

landing on $(\ell+1)$ is the same for any such path! There are $\binom{\ell}{k-1}$ such paths, and so the overall probability of a hop path with $k$ hops landing on $(\ell+1)$ is $\binom{\ell}{k-1}q^{\ell-k+1}p^{k-1}p$, where again $q = 1-p$. We see this formally by observing that any hop path is a series of geometric random variables, each with distribution $\text{Prob}[X = x] = q^{x-1}p$. Furthermore, all hop paths with the same number of hops have the same probability of occurring because any sequence of $k$ geometric random variables will be an arrangement of $(\ell-k+1)$ $q$'s and $k$ $p$'s, where we will assert that the last letter is a $p$. Thus, the number of length-$k$ hop paths can be generated using binomial coefficients for the number of ways to arrange $(\ell-k+1)$ $q$'s and $k-1$ $p$'s.

On substituting these binomial coefficient probability expressions into (3.3),

$$\text{Prob}[A_\ell = 1] = \underbrace{\sum_{k=1}^{\ell+1}\binom{\ell}{k-1}q^{\ell-k+1}p^{k-1}}_{\text{binomial expansion of } (p+q)^\ell} \cdot p = p. \tag{3.4}$$

Notice we are left with the binomial expansion of $(p+q)^\ell = 1$ multiplied by $p$. ∎

The runtime of this procedure is exactly $O(E)$, where $E$ is the number of edges output, because for each instance of the loop, we generate one distinct edge. (This assumes a constant-time routine to generate a geometric random variable and an appropriate data structure to capture the results.)

**3.3. Comparing Ball-Dropping to Grass-Hopping with Coupon Collectors.** Both ball-dropping and grass-hopping produce the correct distribution, but our next question concerns their efficiency. For grass-hopping, this is easy. For ball-dropping, note that we need to drop at least $m$ balls to get $m$ edges. However, we haven't yet discussed how many duplicate entries we expect. Intuitively, if $n$ is very large and $p$ is small, then we would expect few duplicates. Our goal is a sharper analysis. Specifically, how many times do we have to drop balls in order get exactly $m$ distinct edges?

This is a variation on a problem called the coupon collector problem; see (Aigner et al., 2010; Diaconis and Holmes, 2002; Dawkins, 1991) for additional discussion. The classic coupon collector problem is that there are $m$ coupons that an individual needs to collect in order to win a contest. Each round, the individual receives a random coupon with equal probability. How many rounds do we expect before the individual wins? A closely related analysis to what we present shows that we expect the game to run for $m \log m$ rounds.

In our variation of the coupon collector problem, we must find the expected number of draws needed to obtain $m$ unique objects (these are edges) out of a set of $n^2$ objects (these are entries of the matrix), where we draw objects with replacement. This can be analyzed as follows based on a post from math.stackexchange.com (Yu, 2012), or following simple extensions in the reference texts above.

Let $X$ be the random variable representing the number of draws needed to obtain the $m^{th}$ unique object. Let $X_i$ be a random variable representing the number of trials needed to obtain the $i^{th}$ unique object given that $i-1$ unique objects have already been drawn. Then, $E[X] = E[X_1]+E[X_2]+\cdots+E[X_m]$. Each random variable $X_i$ is geometrically distributed with $p = 1-\frac{i-1}{n^2}$ because the probability of drawing a new object is simply the complement of the chance of drawing an object that has already been drawn. Using the expected value of a geometric random variable, we obtain $E[X_i] = \frac{n^2}{n^2-i+1}$. Thus, $E[X] = \sum_{i=1}^{m}\frac{n^2}{n^2-i+1}$. This can be rewritten in terms of harmonic sums. The harmonic sum, $H_n$, is defined as the sum of the reciprocals of the first n integers: $H_n = \sum_{i=1}^{n}\frac{1}{n}$. Consequently, $E[X] = n^2(H_{n^2} - H_{n^2-m})$. The harmonic partial sums can be approximated as follows: $H_n = \sum_{i=1}^{n}\frac{1}{i} \approx \log(n+1)+\gamma$ (Pollack, 2015).

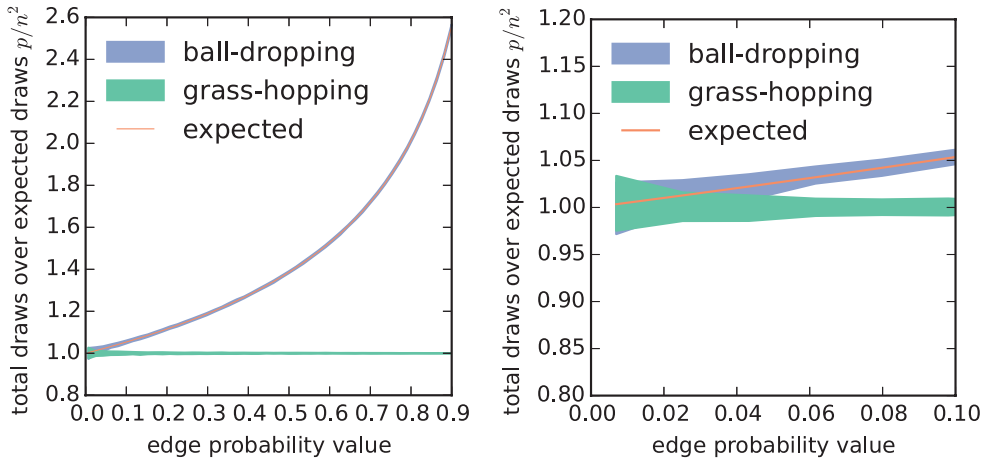



**Fig. 4** *Left: the number of random draws that the ball-dropping and grass-hopping procedures must make as a function of p, normalized by the expected number $pn^2$. Right: the same data, but zoomed into the region where $p < 0.1$. For this figure, $n = 1000$, and we show the 1% and 99% percentiles of the data in the region. The orange line plots the expected ratio $\frac{1}{p}\log(\frac{1}{1-p})$.*

This approximation is derived from the left-hand Riemann sum approximation for the integral of $\int_1^n \frac{1}{x}\,dx$. The difference between the Riemann sum, which represents the harmonic sum, and the integral, is the Euler–Mascheroni constant, $\gamma$, $\lim_{n\to\infty} \sum_{k=1}^{n} \frac{1}{k} - \int_1^n 1/x = \gamma$. After substituting in this approximation, the expected number of draws, $E[X]$, simplifies to $n^2[\log(n^2) - \log(n^2 - m)]$. To find a better expression, we need a value of $m$. Recall that $m$ is sampled from a binomial whose expectation is $n^2p$. Using $m = n^2p$ simplifies the expression to $E[X] = n^2 \log \frac{1}{1-p} = mp^{-1} \log \frac{1}{1-p}$. Thus, the extra work involved is $p^{-1} \log \frac{1}{1-p}$. In Problem 1, we show that $p^{-1} \log \frac{1}{1-p} \geq 1$ with a removable singularity at $p = 0$.

ASIDE 10. *The Euler–Mascheroni constant, $\gamma$, is approximately* 0.57721.

Consequently, ball-dropping always takes more random samples than grass-hopping. The difference between these two becomes considerable as $p$ grows larger, as shown in Figure 4.

**Large Values of *p*.** At this point, we should make the following observation. There is little reason to run ball-dropping once $p$ gets large. This is for two reasons. First, if $p$ is any value like 0.1, then we would expect $0.1n^2$ "edges," in which case we might as well do coin-flipping because it takes almost the same amount of time, so we generally expect $p$ to scale like $O(1/n)$ or $1/n^\gamma$ for $\gamma < 1$, such as $\gamma = 1/2$. In the models we will consider, however, there are often cases where $p$ does get large in small regions. Once $p > 0.5$, then we actually expect *most* edges to be present in the graph, in which case it's more computationally efficient to use ball-dropping to determine which edges are *missing* instead of which edges are present.

**3.4. A Historical Perspective on Leap-Frogging, Waiting Times, and the Geometric Method.** The core mathematical idea underlying what we call "grass-hopping" is a relationship between the geometric distribution and repeated Bernoulli trials.

The essence of these ideas is classical and applications of the idea were used by Fermat and Pascal in some of their early correspondence about probability. See, for instance, the discussions presented on the problem of points (Grinstead and Snell, 2012, p. 30); an English translation of their correspondence is available from the website https://www.york.ac.uk/depts/maths/histstat/pascal.pdf, whose translations are from (Smith, 2012; David, 1962); and a direct explanation of the hidden geometric distribution in their arguments is given in (Hald, 2003, pp. 55–56).

While the fundamental mathematics may be classical, the idea of using a geometric distribution to hop through the probabilities has been known since the 1960s (Fan, Muller, and Rezucha, 1962). That particular article called it a "leap frog" procedure. Subsequent books on sampling complex probability distributions also discussed this idea (Devroye, 1986). More generally, this style of analysis is called *waiting time analysis* in the probability literature. This methodology seeks to understand the distributions between events; the discussion in (Dawkins, 1991) draws a number of further connections between the topics we have presented in this section, including waiting time viewpoints on the coupon collector problem in which the geometric distribution also arises.

Aside 11. *The idea underlying what we call "grass-hopping" has various names in the literature including "leap frog," "waiting times," or "the geometric method." We ask that our term "grass-hopping" be used when these ideas are used in the context of generating graphs from matrices of probabilities.*

In the context of random graph generation, it is our experience that these insights are still not widely known nor always widely implemented—despite being mentioned over 30 years ago by Devroye (1986) and then again over a decade ago by Batagelj and Brandes (2005), where it was called a geometric method. More recently, Hagberg and Lemons (2015) discussed the idea and simply describe it as sampling from a waiting time distribution.

**4. Chung–Lu and Stochastic Block Models: Unions of Erdős–Rényi Graphs.** Thus far, we have only studied fast methods for Erdős–Rényi graphs. The same techniques, however, apply to Chung–Lu and stochastic block model graphs as well, because these probability matrices are *unions of Erdős–Rényi blocks.* This is immediate for the stochastic block model because that model defines a set of small regions. For instance, it is easy to use our Erdős–Rényi subroutines to create a two block stochastic block model as the following code illustrates:

```
""" Generate edges for a stochastic block model with two blocks.
n1 and n2 are the sizes of the two blocks and p is the within group
probability and q is the between group probability.
Example: sbm(20,15,0.5,0.1) # 20 nodes in group 1, 15 nodes in group 2, ...
"""
def sbm2(n1,n2,p,q):
  edges = grass_hop_er(n1,p)                              # generate the n1-by-n1 block
  edges.extend( [ (i+n1,j+n1) for i,j in grass_hop_er(n2,p) ])        # n2-by-n2 block
  edges.extend( [ (i,j+n1) for i,j in grass_hop_er(max(n1,n2),q) if i < n1 and j < n2])
  edges.extend( [ (i+n1,j) for i,j in grass_hop_er(max(n1,n2),q) if i < n2 and j < n1])
  return edges
```

This program has a runtime that scales with the number of edges in the graph.

For Chung–Lu graphs, the situation is slightly more intricate. The example Chung–Lu probability matrix $\boldsymbol{P}$ from section 2.7 has 8 nodes, 16 different probabilities, and 64 total entries in the probability matrix. It turns out that the number of distinct probabilities is given by the number of distinct degrees. That example had four distinct

degrees: 4, 3, 2, 1. Since the probability of each matrix entry is $P_{ij} = d_i d_j/\text{total-degree}$ (where total-degree $= \sum_i d_i$ is just a constant), then if there are $t$ distinct degrees, we will have $t^2$ distinct probabilities in $\boldsymbol{P}$. This corresponds with $t^2$ Erdős–Rényi blocks. This can be justified most easily if the vector of degrees $\mathbf{d}$ is sorted. Let $d_{[1]}$ be the first degree in sorted order and $d_{[2]}$ be the next, and so on. Then the entire vector is

$$\mathbf{d} = [\underbrace{d_{[1]}\ d_{[1]}\ \cdots\ d_{[1]}}_{n_1 \text{ times}}\ \underbrace{d_{[2]}\ d_{[2]}\ \cdots\ d_{[2]}}_{n_2 \text{ times}}\ \cdots\ \underbrace{d_{[t]}\ d_{[t]}\ \cdots\ d_{[t]}}_{n_t \text{ times}}]. \tag{4.1}$$

It does not matter if the vector is sorted in increasing or decreasing order, and many of the values of $n_i$ may be 1. Let $\rho =$ total-degree. The resulting matrix $\boldsymbol{P}$ now has a clear block Erdős–Rényi structure:

$$\boldsymbol{P} = \begin{array}{c} \\ n_1 \\ n_2 \\ \\ n_t \end{array} \begin{array}{c} \begin{array}{cccc} n_1 & n_2 & & n_t \end{array} \\ \left[\begin{array}{c|c|c|c} \frac{d_{[1]}d_{[1]}}{\rho} & \frac{d_{[1]}d_{[2]}}{\rho} & \cdots & \frac{d_{[1]}d_{[t]}}{\rho} \\ \hline \frac{d_{[2]}d_{[1]}}{\rho} & \frac{d_{[2]}d_{[2]}}{\rho} & & \\ \hline \vdots & \vdots & \ddots & \\ \hline \frac{d_{[t]}d_{[1]}}{\rho} & \frac{d_{[t]}d_{[2]}}{\rho} & & \frac{d_{[t]}d_{[t]}}{\rho} \end{array}\right] \end{array}. \tag{4.2}$$

We leave an implementation of this as a *solved* exercise at the conclusion of this paper. (Note that a very careful implementation can use $\binom{t}{2}$ Erdős–Rényi blocks.)

There is another grass-hop-like algorithm for sampling Chung–Lu graphs described in Miller and Hagberg (2011), which avoids the block Erdős–Rényi structure using a rejection sampling based technique. There is an elegant analysis that shows this algorithm is efficient for sparse Chung–Lu graphs that we encourage interested readers to review.

**Direct Ball-Dropping for Chung–Lu and Stochastic Block Models.** It turns out that there are equivalent procedures to ball-dropping for both Chung–Lu and and stochastic block models. We've deferred these to the problem section (Problems 3 and 6) where we explain some of the intuition behind the differences and some of the advantages and disadvantages in the procedures. Both methods need to deal with the same type of duplicate entries that arise in ball-dropping for simple Erdős–Rényi models; however, this setting is far more complicated as the different probability values can result in regions where there *will* be many duplicates. These can be discarded to easily generate a slightly biased graph, which is what is done in the Graph500 benchmark, for example (Murphy et al., 2010). On the other hand, the number of Erdős–Rényi blocks grows quadratically for both models: $\binom{t}{2}$ for Chung–Lu and $k^2$ for the block model. Thus, in the case where there is a large number of blocks, there are some nontrivial considerations as to when grass-hopping is preferable to ball-dropping. As an example, Hagberg and Lemons (2015) shows a technique that generalizes Chung–Lu and enables efficient sampling in this scenario.

**5. Fast Sampling of Kronecker Graphs via Grass-Hopping.** As far as we are aware, a ball-dropping procedure or a coin-flipping procedure was the standard method to sample a large Kronecker graph until very recently. Ball-dropping methods often generate many duplicate edges in hard to control ways (Groër, Sullivan, and Poole,

2011), as we illustrate in section 5.1. In practice, these duplicates were often ignored to generate approximate distributions that were usable in many instances (Murphy et al., 2010). As previously mentioned, coin-flipping could never scale to large graphs. This situation changed when Moreno et al. (2014) showed that a "small" number of Erdős–Rényi blocks were hiding inside the structure of the Kronecker matrix.

We will walk through a new presentation of these results that makes a number of connections to various subproblems throughout discrete mathematics. The key challenge is identifying the *entries of the matrix* where the Erdős–Rényi blocks occur. This was simple for both Chung–Lu and the stochastic block model, but it is the key challenge here.

**5.1. The Problem with Ball-Dropping.** The reason that ball-dropping was the standard way to generate a Kronecker graph for so long is that it is an extremely easy implementation because of the recursive Kronecker structure. In ball-dropping, we sample a single entry proportional to the entries of $\boldsymbol{K} \otimes \boldsymbol{K} \otimes \cdots \otimes \boldsymbol{K}$. Consider the example where $\boldsymbol{K} = \left[\begin{smallmatrix} 0.8 & 0.6 \\ 0.4 & 0.2 \end{smallmatrix}\right]$,

$$\boldsymbol{K} \otimes \boldsymbol{K} = \begin{bmatrix} aa & ab & ba & bb \\ ac & ad & bc & bd \\ ca & cb & da & db \\ cc & cd & dc & dd \end{bmatrix} = \begin{bmatrix} 0.64 & 0.48 & 0.48 & 0.36 \\ 0.32 & 0.16 & 0.24 & 0.12 \\ 0.32 & 0.24 & 0.16 & 0.12 \\ 0.16 & 0.08 & 0.08 & 0.04 \end{bmatrix}. \tag{5.1}$$

The probability that we pick any edge is then

$$\frac{1}{(a+b+c+d)^2}\boldsymbol{K} \otimes \boldsymbol{K} = \begin{bmatrix} 0.16 & 0.12 & 0.12 & 0.09 \\ 0.08 & 0.04 & 0.06 & 0.03 \\ 0.08 & 0.06 & 0.04 & 0.03 \\ 0.04 & 0.02 & 0.02 & 0.01 \end{bmatrix}$$

$$\text{or more generally} \quad \frac{\boldsymbol{K}}{\sum_{ij} K_{ij}} \otimes \frac{\boldsymbol{K}}{\sum_{ij} K_{ij}} \otimes \cdots \otimes \frac{\boldsymbol{K}}{\sum_{ij} K_{ij}}.$$

Although it seems like a complicated process to sample an edge with this probability, the repeated structure of the Kronecker products makes it easy. Recall for this example that

$$\boldsymbol{K} \otimes \boldsymbol{K} = \begin{bmatrix} a \cdot \boldsymbol{K} & b \cdot \boldsymbol{K} \\ c \cdot \boldsymbol{K} & d \cdot \boldsymbol{K} \end{bmatrix} = \begin{bmatrix} 0.8 \cdot \boldsymbol{K} & 0.6 \cdot \boldsymbol{K} \\ 0.4 \cdot \boldsymbol{K} & 0.2 \cdot \boldsymbol{K} \end{bmatrix}. \tag{5.2}$$

The idea that makes ball-dropping easy is that we can sample from this matrix by drawing twice from the entries of $\boldsymbol{K}$. The first draw gets us one of the four regions $0.8\boldsymbol{K}, 0.6\boldsymbol{K}, 0.4\boldsymbol{K}, 0.2\boldsymbol{K}$, and the second draw picks the entry *inside* this region. To describe this formally, it helps to look at the matrix $\boldsymbol{K}/\sum_{ij} K_{ij} = \left[\begin{smallmatrix} 0.4 & 0.3 \\ 0.2 & 0.1 \end{smallmatrix}\right]$ which is the normalized version of $\boldsymbol{K}$ such that the sum of the entries is 1 and each entry represents the probability of picking that entry. Drawing an entry means picking $(1,1)$ with probability 0.4, picking $(1,2)$ with probability 0.3, and so on. Suppose the two draws are $(1,2)$ and $(1,1)$. The edge that this gives us is $(3,1)$, which has probability 0.12. The probability that we drew $(1,2)$ was 0.3, and then the probability we drew $(1,1)$ was 0.4. Their product is exactly 0.12. Listing 4 provides an implementation of the standard ball-dropping procedure for sampling a Kronecker graph where we drop $(\sum_{ij} K_{ij})^k$ edges. This value represents the expected number of edges in the overall graph. If this result is unexpected, see the solution of Problem 13 for more information.

**Listing 4** A standard, but wrong, ball-dropping method for sampling a Kronecker graph

```
import numpy as np
"""Generate a single edge sampled from the
Kronecker matrix distribution where
* p = vec(K)/sum(K) is a flattened 1d array
for the normalized initiator matrix with
probability values that sum to 1,
* n is the matrix-dimension of p, and
* k is the kronecker power.
Example:
ball_drop_kron_edge([0.4,0.2,0.3,0.1],2,3)
"""
def ball_drop_kron_edge(p, n, k):
  R=0; C=0;
  offset = 1; n2 = n*n
  for i in range(k):
    # sample from K* as 1 to n^2
    ind = np.random.choice(n2, 1, p=p)[0]
    # convert to row, col index
    row = ind % n; col = ind // n
    R += row * offset
    C += col * offset
    offset *= n
  return (R,C)
```

```
import math
"""Generate a ball-dropped sample of a
Kronecker graph. The input K is the
Kronecker initiator matrix and k
is the number of levels.
Example:
K = [[0.8,0.6],[0.4,0.2]]
ball_drop_kron_wrong(K, 3)
"""
def ball_drop_kron_wrong(K, k, nedges=-1):
  if nedges < 0:
    # use the expected number of edges in K
    nedges = int(math.pow(np.sum(K), k))
  edges = set()
    # normalize to probability distribution
  n = len(K)         # and vectorize by cols
  Ksum = sum(v for ki in K for v in ki)
  p = [K[i][j]/Ksum for j in xrange(n)
    for i in xrange(n)]
  # keep dropping while we need more edges
  while len(edges) < nedges:
    edges.add(ball_drop_kron_edge(p,n,k))
  return edges
```

However, the implementation in Listing 4 is unfortunately wrong—not because of a bug in the code, but because of a subtle logic error. For the problem $\boldsymbol{K} = \left[\begin{smallmatrix} 0.99 & 0.6 \\ 0.4 & 0.2 \end{smallmatrix}\right]$, $k = 3$ we generated 10,000,000 samples of the graph and computed the empirical probabilities of finding each edge. The raw numbers are in the 0.09 to 0.001 range, and so we scale the probabilities by 10:

(5.3)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| .92 | .56 | .56 | .34 | .56 | .34 | .34 | .21 |
| .37 | .19 | .23 | .11 | .23 | .11 | .14 | .07 |
| .37 | .23 | .19 | .11 | .23 | .14 | .11 | .07 |
| .15 | .08 | .08 | .04 | .09 | .05 | .05 | .02 |
| .37 | .23 | .23 | .14 | .19 | .11 | .11 | .07 |
| .15 | .08 | .09 | .05 | .08 | .04 | .05 | .02 |
| .15 | .09 | .08 | .05 | .08 | .05 | .04 | .02 |
| .06 | .03 | .03 | .02 | .03 | .02 | .02 | .01 |

scaled true probabilities

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| .67 | .49 | .49 | .34 | .49 | .34 | .33 | .22 |
| .36 | .20 | .24 | .13 | .24 | .13 | .15 | .08 |
| .36 | .24 | .20 | .13 | .24 | .15 | .13 | .08 |
| .16 | .09 | .09 | .04 | .10 | .05 | .05 | .03 |
| .36 | .24 | .24 | .15 | .20 | .13 | .13 | .08 |
| .16 | .09 | .10 | .05 | .09 | .04 | .05 | .03 |
| .16 | .10 | .09 | .05 | .09 | .05 | .04 | .03 |
| .07 | .04 | .04 | .02 | .04 | .02 | .02 | .01 |

scaled empirical probabilities.

These two arrays should be equal. What this means is that a common and straightforward implementation of the ball-dropping procedure samples from a different distribution over edges. (Note that this is not an artifact of our choice of the total number of edges as a constant compared with a true sample of the correct number of edges; the result is essentially unchanged even if we sample from the true distribution for the number of edges.) While it is easy to illustrate evidence of the problem, articulating the reason for the problem is another matter. We explain the subtle origin of this problem in more detail and propose a corrected ball-dropping implementation, in section 5.9, after we show how to use grass-hopping to sample from the correct distribution.

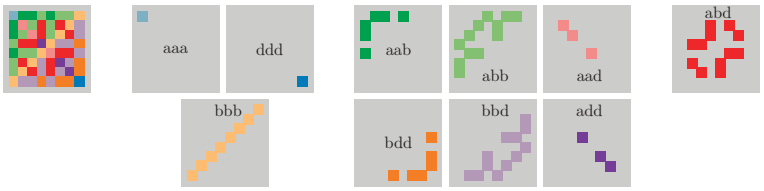



**Fig. 5** *For a Kronecker graph with a* $2 \times 2$ *initiator matrix* $\boldsymbol{K} = \left[\begin{smallmatrix} 0.99 & 0.5 \\ 0.5 & 0.2 \end{smallmatrix}\right] = \left[\begin{smallmatrix} a & b \\ b & d \end{smallmatrix}\right]$ *that has been "⊗-powered" three times to an* $8 \times 8$ *probability matrix (see* (2.5)*) the marked cells illustrate the regions of identical probability that constitute an Erdős–Rényi piece. Here, we have used* symmetry in $\boldsymbol{K}$ *to reduce the number of regions.*

**5.2. Kronecker Graphs as Unions of Erdős–Rényi.** *And why grass-hopping is hard!* You might be wondering what makes this problem hard once we demonstrate that Kronecker graphs are unions of Erdős–Rényi regions. Looking at the matrix $\boldsymbol{P}$ from section 2.6 shows that there are some repeated probabilities. (Figure 5 shows a guide to where the probabilities are the same.) However, it seems as if they are scattered and not at all square like they were in the Chung–Lu and stochastic block model cases. A larger example in Figure 6 shows what a *single* region looks like for a 64-by-64 matrix. This seems to imply that it is extremely difficult to take advantage of any Erdős–Rényi subregions. Moreover, it is unclear how many such regions there are. If there are many such regions that are all small, then it will not help us to sample each region quickly (even if we could!).

We are going to address both of these questions: (i) How many Erdős–Rényi regions are there? and (ii) How can each region be identified? More specifically, let the initiator matrix $\boldsymbol{K}$ be $n$-by-$n$, and let $k$ be the number of Kronecker terms, so there are $n^k$ nodes in the graph. In the remainder of section 5, we provide the following in answer to our questions:

- *Question* (i). There are $\binom{k+n^2-1}{k}$ regions, which is asymptotically $O(k^{n^2-1})$. This is important because it means that the number of regions grows more slowly than the number of *nodes* $n^k$, enabling us to directly enumerate the set of regions and keep the procedure efficient (section 5.3).
- *Question* (ii). The regions are easy to identify in a multiplication table view of the problem and the regions have a 1-1 correspondence with length-$k$ nondecreasing sequences of elements up to $n^2$ (section 5.5).
- *Question* (ii). We can randomly sample in the multiplication table view by unranking multiset permutations (section 5.6).
- *Question* (ii). We can map between the multiplication table view and the repeated Kronecker products via Morton codes (section 5.7).

**5.3. The Number of Erdős–Rényi Regions in a Kronecker Matrix.** Consider the result of taking a Kronecker product of a matrix $\boldsymbol{K}$ with itself. In the 2-by-2 case with $\boldsymbol{K} = \left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$,

$$\boldsymbol{K} \otimes \boldsymbol{K} = \begin{bmatrix} aa & ab & ba & bb \\ ac & ad & bc & bd \\ ca & cb & da & db \\ cc & cd & dc & dd \end{bmatrix}. \tag{5.4}$$

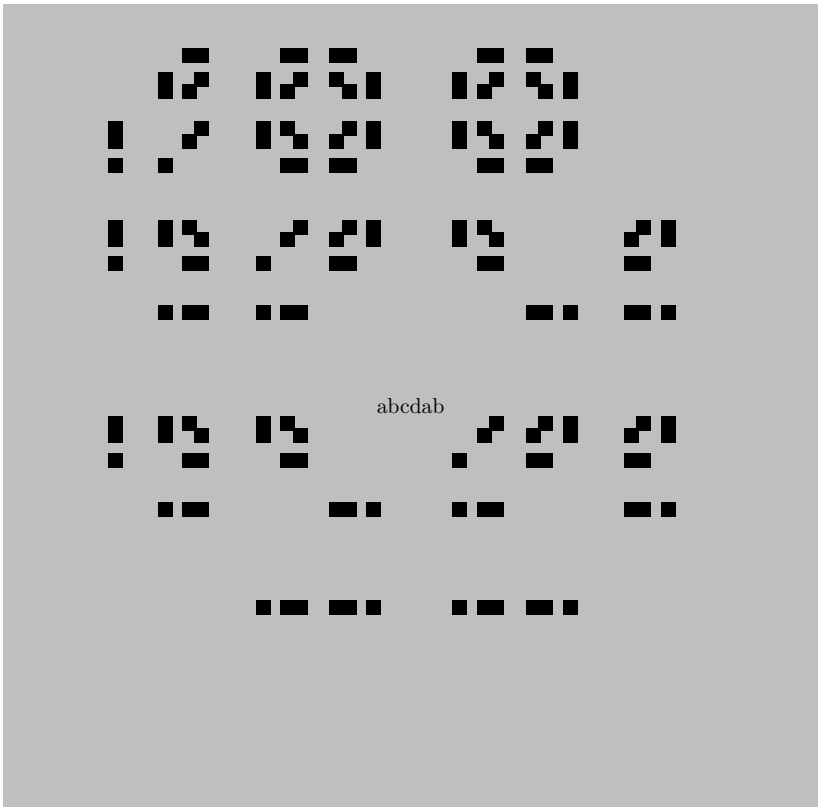



**Fig. 6** *For a Kronecker graph with a* 2*-by-*2 *initiator matrix* $\boldsymbol{K} = \left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]$ *that has been "*$\otimes$*-powered" six times (*$k = 6$*) to an* 64*-by-*64 *probability matrix, the marked cells illustrate one Erdős–Rényi piece that corresponds to the probability abcdab.*

Note that all combinations of symbols $a, b, c, d$ occur in the Kronecker product. This result holds generally. Consequently, we can identify a length-$k$ string of the symbols $a, b, c, d$ with each entry of $\boldsymbol{P} = \boldsymbol{K} \otimes^{k \text{ terms}} \cdots \otimes \boldsymbol{K}$. If $\boldsymbol{K}$ is $n$-by-$n$, then there are $(n^2)^k$ such strings. The feature we are trying to exploit here is that the string $aba$ results in the same probability as $aab$ and $baa$, and hence we want to count the number of these distinct probabilities.

An appropriate mathematical object here is the multiset. In a multiset, we have a set of items along with a multiplicity, where the order of items is irrelevant. So the strings $aab$ and $aba$ would both correspond to the multiset where $a$ occurs twice and $b$ occurs once. The cardinality of this multiset is 3. Consequently, the number of unique probabilities in $\boldsymbol{K} \otimes^{k \text{ terms}} \cdots \otimes \boldsymbol{K}$ is at most the number of distinct multisets we can derive from the $n^2$ symbols in $\boldsymbol{K}$, where each multiset has cardinality $k$. The phrase *at most* is important here. It is possible for two distinct multisets to have the same probability. For instance, if $a = 0.2, b = 0.3, c = 0.1, d = 0.6$, then $ab = cd$ and there would be fewer regions with distinct probability. In this section, we seek to upper-bound the number of these regions in the *worst case* and so we focus our attention on scenarios where these additional repeat structures do not occur.

A famous and well-known result in counting follows. Recall that $\binom{n}{k}$ is the number of distinct sets of cardinality $k$ that can be drawn from a set of $n$ items. The generalization to multisets is called "multichoose" and

$$\begin{aligned}\left(\!\!\binom{n}{k}\!\!\right) &= \text{the number of multisets of cardinality } k \text{ with } n \text{ items} \\ &= \binom{n+k-1}{k}.\end{aligned} \tag{5.5}$$

The right-hand side of this can be derived from a stars and bars argument; see, for instance, Wikipedia or a textbook in discrete math such as (Stanley, 1986).

Consequently, there are at most $\left(\!\!\binom{n^2}{k}\!\!\right) = \binom{k+n^2-1}{k}$ Erdős–Rényi regions in the probability matrix for a Kronecker graph.

Recall that for large networks, we are expecting the network to be sparse, and so we expect the number of edges generated to be about the same as the number of nodes. This expression is worrisome because there are $n^k$ nodes in a Kronecker graph and so we'd like there to be *fewer* Erdős–Rényi regions than there are nodes. If this isn't the case, then just looking at all regions would result in more work than we would do in generating edges within each region.

We now show three results that will guarantee that there are fewer Erdős–Rényi regions than nodes of the graph for sufficiently large values of $k$. The first result is that there are at most $(e+1)^k$ regions for a large enough $k$, which gives our result when $e+1 \leq n$. The second is that if $n=2$ or $n=3$ there are at most $2^k$ or $3^k$ regions when $k \geq 10$. The third is that $\binom{n^2+k-1}{k}$ is $O(k^{n^2-1})$ asymptotically, which means that the number of regions grows asymptotically much more slowly than the number of nodes.

ASIDE 12. *We use big-O notation to understand what happens for large values of k when n is considered a fixed constant.*

*Result* 1. From the Taylor expansion of $e^x$ with $x$ positive, we have that $e^x > \frac{x^k}{k!}$ for any $k$. Letting $x=k$ and rearranging our equation results in $k! > (\frac{k}{e})^k$. Furthermore, we know $\binom{n}{k} = \prod_{i=0}^{k-1} n - i/k! \leq \frac{n^k}{k!}$. Combining the lower bound on $k!$ into this equation yields $\binom{n}{k} \leq (\frac{en}{k})^k$. Substituting the expression for the number of regions yields $\binom{k+n^2-1}{k} \leq (\frac{e(k+n^2-1)}{k})^k$. Once $k \geq e(n^2-1)$, we have at most $(e+1)^k$ regions.

*Result* 2. For $n=2$, we will show by induction that $\binom{k+n^2-1}{k} \leq 2^k$ when $k$ becomes large. Note $\binom{k+n^2-1}{k} = \binom{k+n^2-1}{n^2-1}$, so when $k=7$, $\binom{k+3}{3} = 120$, $2^k = 128$, and $120 < 128$. For our inductive step we must show that our inequality holds true for $k+1$, assuming it holds true for $k$. Substituting $k+1$, we need to show that $\binom{k+4}{3} \leq 2^{k+1}$. Expanding the left-hand side,

$$\begin{aligned}\binom{k+4}{3} &= \frac{(k+4)(k+3)(k+2)}{3!} \\ &= \frac{k+1}{k+1} \cdot \frac{(k+4)(k+3)(k+2)}{3!} \\ &= \frac{k+4}{k+1}\binom{k+1}{3} \\ &\leq 2 \cdot 2^k = 2^{k+1}.\end{aligned}$$

The last statement assumes $k \geq 2$, so $\frac{k+4}{k+1} \leq 2$.

The same strategy works for $n = 3$ to show that $\binom{k+n^2-1}{k} \leq 3^k$ for $n = 3$ when $k$ becomes large. When $k = 10$, $\binom{k+8}{k} = 43758$, $3^k = 59049$, and $43758 < 59349$. For our inductive step we must show that our inequality holds true for $k+1$, assuming it holds true for $k$. Substituting $k+1$ into our inequality, we need to show $\binom{k+9}{8} \leq 3^{k+1}$, or equivalently $\frac{\prod_{i=2}^{9} k+i}{8!} \leq 3^k \cdot 3$. Multiplying the left-hand side by $\frac{k+1}{k+1}$ allows us to use our inductive assumption that $\binom{k+8}{8} \leq 3^k$. Thus, we obtain $\frac{k+9}{k+1} \leq 3$ which is true when $k \geq 3$.

*Result* 3. Finally, we note that $\binom{k+n^2-1}{k} = \frac{(k+n^2-1)!}{k!(n^2-1)!} = O(\frac{(k+n^2-1)!}{k!})$ because $n$ is a constant. If we use the simple upper bound $\frac{(k+n^2-1)!}{k!} \leq (k+n^2-1)^{n^2-1}$ and then take logs, we have $(n^2-1)\log(k+n^2-1)$. The concavity of log gives the subadditive property $\log(a+b) \leq \log(a) + \log(b)$ when $a, b \geq 2$. So $(n^2-1)\log(k+n^2-1) \leq (n^2-1)\log(k) + (n^2-1)\log(n^2-1)$. Exponentiating now yields $O(k^{n^2-1})$.

We now have the results that show there are sufficiently few Erdős–Rényi regions in a Kronecker graph, and so our grass-hopping procedure could successfully be applied to each region. This resolves question (i). We now turn to question (ii).

**5.4. The Strategy for Grass-Hopping on Kronecker Graphs: Multiplication Tables and Kronecker Products.** In order to find the Erdős–Rényi regions in a Kronecker graph easily, we need to introduce another structure: the multiplication table. Let $\mathbf{v}$ be the vector $\begin{bmatrix} a & b & c & d \end{bmatrix}^T$, then

$$\mathbf{v} \otimes \mathbf{v} = \begin{bmatrix} a\mathbf{v} \\ b\mathbf{v} \\ c\mathbf{v} \\ d\mathbf{v} \end{bmatrix}. \tag{5.6}$$

This object can be *reshaped* into a matrix called a multiplication table,

$$\text{reshape}(\mathbf{v} \otimes \mathbf{v}) = \begin{bmatrix} aa & ab & ac & ad \\ ba & bb & bc & bd \\ ca & cb & cc & cd \\ da & db & dc & dd \end{bmatrix}. \tag{5.7}$$

Note that what we call a multiplication table here is just a rank-1 matrix, $\mathbf{v}\mathbf{v}^T$. The reason this is often called a multiplication table is that it is exactly the multiplication table you likely learned in elementary school when $a = 1, b = 2, c = 3, d = 4, \ldots$, where the result has all pairs of products between the elements. The *reshape* function in this case takes a length-$n^2$ vector and assembles it into a matrix by column. For those accustomed to the mixed-product property of Kronecker products, this result is just $\text{vec}(\mathbf{v}\mathbf{v}^T) = \mathbf{v} \otimes \mathbf{v}$.

The *vec* operator is another useful tool that converts a matrix into a vector by appending the column vectors of a matrix together:

$$\text{vec}\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = \begin{bmatrix} a \\ c \\ b \\ d \end{bmatrix}. \tag{5.8}$$

In fact, the *vec* and *reshape* functions here are inverses: *vec* takes a matrix and makes it into a vector by appending columns, and *reshape* takes a vector and makes it into a matrix by dividing it into columns.

To get some sense of where this is going, compare (5.7) to (5.4) and notice that all the same entries occur, but they are just reorganized.

The idea with a multiplication table is that we can immediately generalize to using more than just the product of two numbers:

$$\text{reshape}(\mathbf{v}\otimes\mathbf{v}\otimes\mathbf{v}) = \begin{array}{|cccc|c|c|c|}\hline aaa & aab & aac & aad & bbd & ccd & ddd\\ aba & abb & abc & abd & bcd & cdd & \\ aca & acb & acc & acd & bdd & & \\ ada & adb & adc & add & & & \\ \hline\end{array}\quad \begin{array}{|cccc|c|c|}\hline baa & bab & bac & bad & cbd & dcd\\ \hline\end{array}\quad \begin{array}{|cccc|c|}\hline caa & cab & cac & cad & dbd\\ \hline\end{array}\quad \begin{array}{|cccc|}\hline daa & dab & dac & dad\\ \hline\end{array}. \tag{5.9}$$

More generally, a $k$-dimensional multiplication table arises from

$$\text{reshape}(\underbrace{\mathbf{v}\otimes\mathbf{v}\otimes\cdots\otimes\mathbf{v}}_{k \text{ times}}), \tag{5.10}$$

where now reshape takes a length-$n^k$ vector and produces an $\underbrace{n\times n\times\cdots\times n}_{k \text{ times}}$ table.

Here's how the multiplication table view helps us: *it is easy to find the Erdős–Rényi regions in a multiplication table!* Consider the entry $aab$. This occurs in cells $(1,1,2)$, $(1,2,1)$, and $(2,1,1)$. These are exactly the three permutations of the multiset with two 1's and one 2.

However, that result does not do us any good without the following fact: *the entries of the $k$-dimensional multiplication table with $\mathbf{v} = vec(\boldsymbol{K})$ can be mapped* 1–1 *to entries of the repeated Kronecker product matrix* $\underbrace{\boldsymbol{K}\otimes\boldsymbol{K}\otimes\cdots\otimes\boldsymbol{K}}_{k \text{ times}}$. More specifically, we show that

$$\text{vec}(\underbrace{\boldsymbol{K}\otimes\boldsymbol{K}\otimes\cdots\otimes\boldsymbol{K}}_{k \text{ times}}) = \boldsymbol{M}_{n,k}\,\underbrace{\text{vec}(\boldsymbol{K})\otimes\text{vec}(\boldsymbol{K})\otimes\cdots\otimes\text{vec}(\boldsymbol{K})}_{k \text{ times}}, \tag{5.11}$$

where $\boldsymbol{M}_{n,k}$ is a permutation matrix based on a Morton code. This mapping is illustrated in Figures 7 and 8. We will explain exactly what is in those figures in subsequent sections, but we hope they help provide a visual reference for the idea of mapping between the Kronecker information and the multiplication table. Note that a quick analysis of (5.7) and (5.4) and various generalizations show the existence of a permutation matrix between these two. Working out a few examples like this by hand was exactly how we got started on the theory. The key insight of the proof is the characterization of this permutation through Morton codes.

These two results, together with the small number of regions result from section 5.3, enable the following strategy to grass hop on an Kronecker graph efficiently: Grass hop independently in each region of the the multiplication table and then map the multiplication table entries back to the Kronecker matrix. More programmatically,

```
""" Generate a Kronecker graph via grass-hopping. The input K is the
Kronecker initiator matrix and the the value k is the number of levels.
Example: grass_hop_kron([[0.99,0.5],[0.5,0.2]], 3) """
def grass_hop_kron(K,k):
  n = len(K) # get the number of rows
  v = [K[i][j] for j in xrange(n) for i in xrange(n)] # vectorize by cols
  edges_mult = []
  for r in regions(v,k):            # for each region of the mult. table
    edges_mult.extend(grass_hop_region(r, v)) # get edges in mult. table
  edges_kron = []
  for e in edges_mult:              # map edges from mult. table to kron
    edges_kron.append(map_mult_to_kron(e, n))
  return edges_kron
```

The goal of the next three subsections is to write down each of the functions `regions` (section 5.5), `grass_hop_region` (section 5.6), and `map_mult_to_kron` (section 5.7). Once we have these pieces, we will be able to efficiently grass hop through a Kronecker graph!

**5.5. Enumerating All Erdős–Rényi Regions.** Recall that, in the worst case, each of the distinct probabilities that occur in $\boldsymbol{P} = \boldsymbol{K} \otimes \overset{k \text{ terms}}{\cdots} \otimes \boldsymbol{K}$ can be uniquely identified with a multiset of size $k$ from a collection of $n^2$ objects (section 5.3). Our task is to enumerate all of the regions defined by these multisets. Here, we note that each multiset can be identified with a nondecreasing sequence of length $k$ where the symbols are $0, 1, \ldots, n^2 - 1$. For instance, when $n = 2, k = 3$, these sequences are

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| [0,0,0], | [0,0,2], | [0,1,1], | [0,1,3], | [0,2,3], | [1,1,1], | [1,1,3], | [1,2,3], | [2,2,2], | [2,3,3], |
| [0,0,1], | [0,0,3], | [0,1,2], | [0,2,2], | [0,3,3], | [1,1,2], | [1,2,2], | [1,3,3], | [2,2,3], | [3,3,3]. |

Let $\mathbf{v} = \text{vec}(\left[\begin{smallmatrix} a & b \\ c & d \end{smallmatrix}\right]) = \left[\begin{smallmatrix} a & c & b & d \end{smallmatrix}\right]^T$. The sequence $[0, 2, 2]$ then refers to the probability $v(1)v(3)v(3) = abb$.

At this point, we are starting to mix programmatic indexes $(0, \ldots, n^2 - 1)$ with the mathematical indices $(1, \ldots, n^2)$. Our goal is not to be confusing, but rather to ensure that the discussion in the text matches the programs more precisely from this point forward.

As justification, we hope it is easy to see that each nondecreasing sequence of length $k$ corresponds to a multiset of cardinality $k$. (For instance, $[0, 2, 2, 3]$ corresponds to one 0, two 2's, and one 3.) The other direction, that each multiset can be represented as a nondecreasing sequence, is also straightforward. Given a multiset of elements from 0 to $n^2 - 1$ with $k$ total elements, place them into a sequence with repetitions in sorted order. Thus, if we had two 5's, three 1's, and one 2, we would obtain the sequence $[1, 1, 1, 2, 5, 5, 5]$. This sequence is nondecreasing and so we can do this for any multiset.

*Thus, we have justified that the length-k nondecreasing sequences are in* 1–1 *correspondence with the Erdős–Rényi regions.* We now continue with the problem of computationally enumerating nondecreasing sequences.

ASIDE 13. *There could be fewer Erdős–Rényi regions due to additional structure in the specific choice of probabilities. From now on, we assume the worst case scenario that each multiset corresponds to a distinct probability.*

For remainder of this section, $k$ will represent the length of the sequence and $m$ will represent the largest entry $+ 1$ that is valid in the sequence in order to match the code precisely. In the previous example, $k = 4$ and $m = 4$. Our code for this task is implemented in the subroutine

**Listing 5** Code to update a nondecreasing sequence representing a subregion to the next nondecreasing sequence

```
def next_region(cur, m):
  k = len(cur)
  cur[k-1] += 1      # increment the last element
  if cur[k-1] == m:  # if there is spill
    if len(cur) > 1: # there is an array
      cur[0:k-1] = next_region(cur[0:-1],m) # recur on prefix
      cur[k-1] = cur[k-2]                   # update suffix
    else:
      cur[k-1] = -1   # singleton, no room left!
  return cur

""" Generate the set of Erdos-Renyi regions in a Kronecker graph
where v = vec(K), and k = number of levels. Each region gives indices
into v that produce a distinct Erdos-Renyi probability.
Example: regions([0.99,0.5,0.5,0.2],3) """
def regions(v,k):
  m = len(v)
  rval = []
  cur = [ 0 for _ in xrange(k) ] # initialize the regions to zero
  while cur[0] != -1:
    rval.append(list(cur)) # make a copy
    next_region(cur, m)
  return rval
```

`regions` in Listing 5. It begins with the first sequence $[0,0,0,0]$ and iteratively steps through all of them via the function `next_region`.

The function `next_region` handles scenarios exemplified by the following three cases:

$$\begin{aligned} &[0,1,1,2] \to [0,1,1,3] && \text{easy,} \\ &[1,3,3,3] \to [2,2,2,2] && \text{spill,} \\ &[3,3,3,3] \to [-1,-1,-1,-1] && \text{spill and done.} \end{aligned} \tag{5.12}$$

Recall that Python indexes from 0. This `next_region` update works by incrementing the last, or $(k$–$1)$th, entry of the sequence to the next value. In the easy case, this results in $[0,1,1,2] \to [0,1,1,3]$. In the spill case, we get $[1,3,3,3] \to [1,3,3,4]$. We check for these cases by examining the value of this last element again. If it is any value $< m$, then we have the easy case and there is nothing left to do. However, if the last value is equal to $m$, then we need to handle the spill. In the spill scenario, we recursively ask for the next nondecreasing sequence for all but the last element. In our example, this call produces `next_update([1, 3, 3], 4)`, which yields $[1,3,3] \to [2,2,2]$. At this point, `cur = [2, 2, 2, 4]`. Because the sequence must be nondecreasing, we set the last element (currently at 4) to be the first value possible. This is given by the second-to-last element (2). And so, this update produces $[1,3,3,3] \to [2,2,2,2]$. The final case is when the array has length 1, and so there is no prefix to update. In this case, we simply flag this scenario by introducing a $-1$ sentinel value which propagates through the array.

*Proof that regions and update work.* The following is a sketch of a proof that `next_region` gives the next region in lexicographic order. First, for arrays of length 1, this is true because at each step we increment the element up until we generate the

$-1$ termination symbol. We now inductively assume it is true for arrays of length $< k$. When `next_region` runs on an array of length $k$, then either we increment the last element, in which case we have proven the result, or the last element is already $n-1$. If the last element is already $n-1$, then we generate the next element in lexicographic order in the prefix array, which is of length $k-1$, and this occurs via our induction hypothesis. Finally, note that the last element of the updated prefix has the same value as our suffix.

**5.6. Grass-Hopping within an Erdős–Rényi Region in the Multiplication Table: Unranking Multiset Permutations.** Given an Erdős–Rényi subregion, we now turn to the problem of how to grass hop within that region in the multiplication table. Let $\mathbf{v}$ be a length-$m$ vector, which is $\text{vec}(\boldsymbol{K})$, and consider the $k$-dimensional multiplication table $\underline{\boldsymbol{M}}$. The entry $(i, j, \ldots, l)$ in the multiplication table is simply equal to the product $v(i)v(j)\cdots v(l)$. In other words,

$$\underline{\boldsymbol{M}}(i, j, \ldots, \ell) = \underbrace{v(i)v(j)\cdots v(\ell)}_{k \text{ total terms}}, \tag{5.13}$$

where $v(i)$ is the $i$th entry in the vector $\mathbf{v}$. Recall from section 5.5 that each Erdős–Rényi subregion exactly corresponds with a length-$k$ nondecreasing sequence. Let $r$ be the length-$k$ nondecreasing sequence that labels the current region. In the multiplication table, this region corresponds to the element

$$\underline{\boldsymbol{M}}(r_1 + 1, r_2 + 1, \ldots, r_k + 1) = v(r_1 + 1)v(r_2 + 1)\cdots v(r_k + 1). \tag{5.14}$$

(Note that we index the region programmatically, but the vector entries mathematically, hence the addition of 1.) The locations in the multiplication table in which this Erdős–Rényi subregion occurs are the distinct permutations of the sequence $r$. For example,

$$\begin{aligned} r = [0,1,1,1] \rightarrow \quad & [0,1,1,1], [1,0,1,1], [1,1,0,1], [1,1,1,0], \\ r = [0,1,2,2] \rightarrow \quad & [0,1,2,2], [0,2,1,2], [0,2,2,1], [1,0,2,2], \\ & [1,2,0,2], [1,2,2,0], [2,0,1,2], [2,0,2,1], \\ & [2,1,0,2], [2,1,2,0], [2,2,0,1], [2,2,1,0]. \end{aligned} \tag{5.15}$$

All of these entries have exactly the same probability because the multiplication operation in (5.14) is associative.

Consequently, it is easy to find the regions of the multiplication table that share the same probability. What is not entirely clear at this point is how we use this observation with a grass-hopping procedure. In grass-hopping, we move around indices of the Erdős–Rényi region in hops. How can we *hop* from element $[0, 2, 1, 2]$ to element $[2, 0, 1, 2]$?

The answer arises from a *ranking* and *unranking* perspective on combinatorial enumeration (Bonet, 2008). For a complex combinatorial structure, such as multiset permutations, we can assign each permutation a rank from 0 to the total number of objects minus one. Thus, in the second example in (5.15) we have

$$\begin{aligned} [0,1,2,2] &\rightarrow 0, \\ [0,2,1,2] &\rightarrow 1, \\ &\vdots \\ [2,2,1,0] &\rightarrow 11. \end{aligned} \tag{5.16}$$

This rank is based on the lexicographic order of the objects. Lexicographic order is exactly the order you'd expect things to be in based on your experience with traditional sequences of numbers. The permutation $[2, 1, 0, 2]$ occurs before $[2, 2, 0, 1]$ for the same reason we say 2102 is before 2201 in order. *Unranking* is the opposite process. Given an integer between 0 and the total number of objects minus one, the *unranking* problem is to turn that integer into the combinatorial object. Hence,

$$4 \overset{\text{unrank}}{\to} [1, 2, 0, 2]. \tag{5.17}$$

More generally, given the initial sequence $r = [0, 1, 2, 2]$ and any integer between 0 and 11, the unranking problem is to turn that integer into the sequence with that rank in lexicographic order.

This is exactly what we need in order to do grass-hopping on multiset permutations. We grass hop on the integers between 0 and the total number of multiset permutations $-1$. The total number of multiset permutations is given by the the formula

$$\frac{m!}{a_1!a_2!\dots a_k!}, \tag{5.18}$$

where $a_i$ is the number of times the $i$th element appears and $m$ is the cardinality of the multiset. (See the permutation page on Wikipedia for more details.) At a high level, the unrank algorithm (Listing 6) takes as input a multiset permutation in nondecreasing form represented as an array, as well as an index or "rank" of the desired permutation. The algorithm is recursively defined and simply returns the original array if the input rank is 0 as the base case. In the other case, the algorithm searches for the first element of the multiset permutation. (The details of this search are below.) Once it finds the first element, it repeats the search on the remainder of the sequence after removing that element.

To make this code somewhat efficient, we maintain a counter representation of the multiset. This counts the number of times each element occurs in the multiset and keeps a sorted set of elements. For the multiset $[0, 1, 2, 2]$, the counter representation is

$$\underbrace{\text{keys} = [0, 1, 2]}_{\text{sorted elements}} \text{ and } mset = \begin{cases} 0 & \to 1, \\ 1 & \to 1, \\ 2 & \to 2. \end{cases} \tag{5.19}$$

The functions `ndseq_to_counter` and `counter_to_ndseq` convert between the counter and sequence representations. The function `num_multiset_permutations` returns the number of permutations of a given multiset via (5.18). For our example, $[0, 1, 2, 2]$, equation (5.18) is equal to 12, which matches our direct enumeration in (5.16). The main algorithm is `unrank_mset_counter`, which takes as input the counter representation of the multiset as well as the "rank" index. The objective of `unrank_mset_counter` is to find the first element of the permutation in lexicographic order. We examine the sorted elements in the `keys` array in order. For each element, we tentatively remove it by decreasing its count in `mset` and then we count the number of multiset permutations with the remaining elements. For our running example, this yields

$$[0, 1, 2, 2] \to \begin{array}{l} \text{starts with 0 and ends with a permutation of } [1, 2, 2] = 3, \\ \text{starts with 1 and ends with a permutation of } [0, 2, 2] = 3, \\ \text{starts with 2 and ends with a permutation of } [0, 1, 2] = 6. \end{array} \tag{5.20}$$

**Listing 6** Unranking a multiset permutation

```
# unrank takes as input:
# - C: a multiset represented by a list
# - n: the lexicographic rank to find
# and returns the nth permutation of C in
# lexicographic order.
#
# Examples:
# unrank([0,1,1,3], 0) returns [0,1,1,3]
# unrank([0,1,1,3], 1) returns [0,1,3,1]
# unrank([0,1,1,3], 2) returns [0,3,1,1]

from math import factorial

def ndseq_to_counter(seq):
  mset = {}
  for c in seq:
    # get a value with a default
    # of zero if it isn't there
    mset[c] = mset.get(c,0)+1
  return mset, sorted(mset.keys())

def counter_to_ndseq(mset,keys):
  seq = []
  for k in keys: # keys in sorted order
    # append k mset[k] times
    for v in xrange(mset[k]):
      seq.append(k)
  return seq

def num_multiset_permutations(mset):
  count = factorial(sum(mset.values()))
  for k in mset.keys():
    count = count//factorial(mset[k])
  return count

def unrank_mset_counter(mset,keys,n):
  if n==0:       # easy if rank == 0
    return counter_to_ndseq(mset,keys)
  for s in keys: # else find prefix key
    mset[s] -= 1 # decrease count of s
    # determine number of prefixes with s
    place = num_multiset_permutations(mset)
    if place > n:
      # then the first element is s
      if mset[s] == 0: # remove the key
        keys.remove(s) # if the count is 0
      suffix = unrank_mset_counter(
                mset, keys, n) # recurse!
      suffix.insert(0, s)      # append s
      return suffix
    else:  # here it does not start with s
      mset[s] += 1 # restore the count
      n -= place   # update search offset
  raise(ValueError("rank too large"))

def unrank(seq,n):
  mset,keys = ndseq_to_counter(seq)
  return unrank_mset_counter(mset,keys,n)
```

We interpret this output as follows. Let the desired rank be $R$. There are three permutations which start with a zero, and they correspond to the first three ranks 0, 1, or 2. So if $R < 3$, in that case we can recurse and unrank the permutation with sequence $[1, 2, 2]$ and exactly the same rank $R$. If the rank is 3, 4, or 5, then the permutation starts with 1 and we can recurse and unrank the permutation of $[0, 2, 2]$ with rank $R - 3$. Finally, if the rank is $6, \ldots, 11$, then the permutation starts with 2 and we can recurse and unrank the permutation of $[0, 1, 2]$ with rank $R - 6$. These recursions yield the suffix that we return.

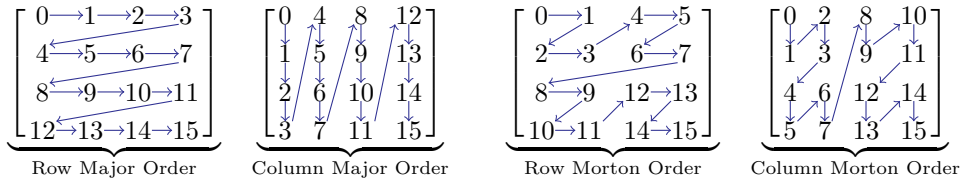
**5.7. Mapping from the Multiplication Table to the Kronecker Graph: Morton Codes.** In this section we prove a novel connection between repeated Kronecker product graph matrices and Morton codes that was mentioned in (5.11). But first, some background on Morton codes!

Morton codes or Z-order codes are a neat primitive that arises in a surprising diversity of applications (Morton, 1966). Relevant details of these applications are beyond the scope of our tutorial, but they include high-performance computing (Buluç et al., 2009), database search (Orenstein and Merrett, 1984), and computer graphics (Vinkler, Bittner, and Havran, 2017) (see Wikipedia for additional references). When Morton codes and Z-order are used in the context of matrices, the terms refer to a specific way to order the matrix elements when they are represented as a data array with a single index. Common strategies to accomplish this include *row* and *column* major orders, which order elements by rows or columns, respectively. Morton codes adopt a recursive, hierarchical pattern of indices. Here are four different ways to

**Listing 7** Grass-hopping in an Erdős–Rényi region of a Kronecker product matrix

```
import numpy as np # use np.random.geometric for the geometric random variables
# grass_hop_region takes as input
# - r: the region to be sampled represented by a non-decreasing array
# - v: the initiator matrix represented as a n^2-by-1 column vector
# and returns a list of edges represented by indexes in mult. table
# Example: v = [0.99,0.5,0.5,0.2]; grass_hop_region(regions(v,3)[2], v)
def grass_hop_region(r,v):
    p = multtable(r,v)               # p is the common prob value of the region
    n = num_multiset_permutations(ndseq_to_counter(r)[0]) # total size of region
    edges_mult = []                  # the initially empty list of edges
    i = -1                           # starting index of the grass-hopping
    gap = np.random.geometric(p)     # the first hop
    while i+gap < n:                 # check to make sure we haven't hopped out
        i += gap                     # increment the current index
        edges_mult.append(unrank(r,i))  # add the
        gap = np.random.geometric(p) # generate the next gap
    return edges_mult

# multtable takes as input:
# - r: an array with k elements specifying a cell in the multiplication table
# - v: the initiator matrix represented as a n^2-by-1 column vector
# and returns the value at the specified location in the multiplication table
def multtable(r,v):
    final = 1.0
    for val in r:
        final *= v[val]
    return final
```

organize the 16 values in a 4-by-4 matrix:



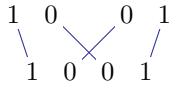

The name "Z-curve" comes from the description of the row Morton order, which enumerates the first 2-by-2 block in a "Z" shape and then recursively repeats it in growing 2-by-2 blocks. For the general Morton code, notice the recursive pattern as well. We are going to use *column* Morton orders in this paper.

More formally, a Morton code refers to a specific method, which we now describe, that is used to generate the linear index from the row and column. It can also refer to the inverse map, which takes a linear Morton index and produces the row and column in the matrix. We refer to these as follows:

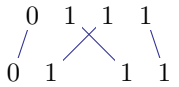
$$\begin{aligned} &\text{MortonEncode}(i,j) \to I \in [0, n^2-1], \\ &\text{MortonDecode}(I) \to (i,j) \in [0, n-1]^2. \end{aligned} \tag{5.21}$$

At this point, we've fully switched to programmatic indices since otherwise the confusion between the codes and the text would become extreme. Matrices and vectors are now indexed from 0 as well. In the previous 4-by-4 example of the orders, $\text{MortonEncode}(1,2) = 9$ and $\text{MortonDecode}(7) = (3,1)$. (Our row and column indices start at zero if these numbers seem off by one!) The encoding and decoding procedures

work on bit-strings for the respective numbers. In fact, the procedure is surprisingly elegant: take the bit-strings for the row and column indices and interleave the digits. For example, let's say we want to encode the numbers 1 and 2 into a single value. First, we convert both numbers into base 2: $1 \to 01_2$ and $2 \to 10_2$. Next, we interleave the digits starting from the column:




The resulting code is $1001_2 = 9$: the desired Morton index. To decode a Morton index into its row and column numbers, we simply reverse the process. We consider 7 with it's base-2 representation $0111_2$ and divide the digits into two groups, alternating every binary digit:




This gives a column index of 1 and a row index of 3.

Now, suppose we consider the Morton code as a map between two linear indices: the column Morton order and the column major order. This is easy to build if we are given a MortonDecode function because we can quickly move from the row and column values to the linear index in column major order. (In case this isn't clear, given the row $r$ and column $c$ indices, the column major index is $r + cn$, where $n$ is the number of rows of the matrix.) The result is a permutation $[0, n^2 - 1] \to [0, n^2 - 1]$:

$$\begin{array}{llll} 0 \to 0 & 4 \to 2 & 8 \to 8 & 12 \to 10 \\ 1 \to 1 & 5 \to 3 & 9 \to 9 & 13 \to 11 \\ 2 \to 4 & 6 \to 6 & 10 \to 12 & 14 \to 14 \\ 3 \to 5 & 7 \to 7 & 11 \to 13 & 15 \to 15 \end{array} . \tag{5.22}$$

This permutation induces a permutation matrix $\boldsymbol{M}$:

$$\left[\begin{array}{cccccccccccccccc} 1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&1 \end{array}\right] . \tag{5.23}$$

The essence of our final primitive is that this Morton map translates between the multiplication table and the Kronecker product, as illustrated in Figure 7. In this case, we have validated the case of (5.11):

$$\underbrace{\text{vec}(\boldsymbol{K} \otimes \boldsymbol{K})}_{\substack{\text{Kronecker matrix} \\ \text{in column major}}} = \underbrace{\boldsymbol{M}}_{\substack{\text{MortonDecode} \\ \text{to column major}}} \underbrace{[\text{vec}(\boldsymbol{K}) \otimes \text{vec}(\boldsymbol{K})]}_{\substack{\text{linear index} \\ \text{from multiplication table}}} , \tag{5.24}$$

where $\boldsymbol{K}$ is two-by-two. We prove this statement in full generality shortly.

More generally, Morton codes can be defined with respect to any base. We illustrate a case of (5.11) in Figure 8, where $\boldsymbol{K}$ is 3-by-3 and $k = 3$. Suppose, in the multiplication table, we generate an edge at index 477. The value 477 in base 3 is

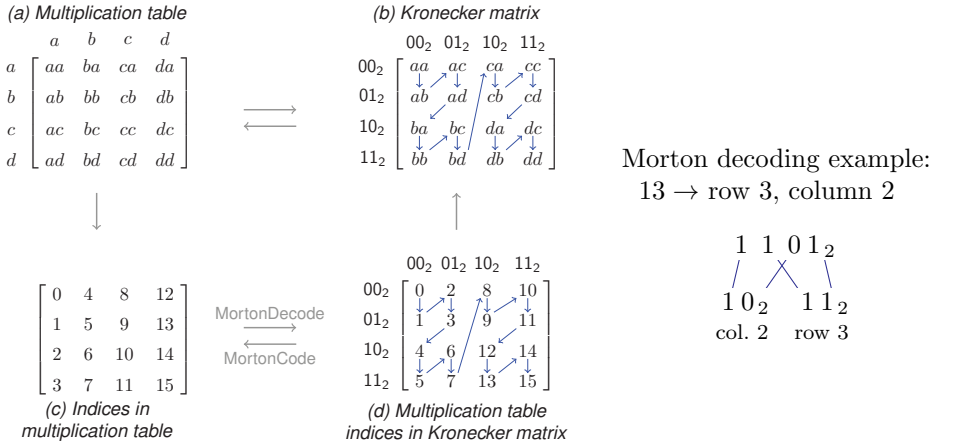
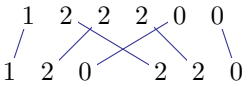



**Fig. 7** *The Morton code relationship between the multiplication table* (a) *and the Kronecker matrix* (b) *for* $k = 2$ *in the Kronecker product of* $\boldsymbol{K} = \left[\begin{smallmatrix} a & c \\ b & d \end{smallmatrix}\right]$. *Our relationship builds on the Morton decoding transformation from indices in the multiplication table* (c) *to the reordered entries that arise in the Kronecker matrix* (d). *The reordering is the result of a Morton decoding process where we take an index from the multiplication table, write it in base* 2, *and then separate alternating digits into two different indices, one for the row and one for the column. (Note that these are all* 0 *indexed.) A larger example is given in Figure* 8.

$122200_3$. The resulting MortonDecode is




which gives a column index of $1 \cdot 3^2 + 2 \cdot 3 + 0 = 15$ and a row index of $2 \cdot 3^2 + 2 \cdot 3 + 0 = 24$. Looking up this entry in the Kronecker matrix (Figure 8, upper right) yields a value of $fia$. Checking back in the multiplication table (Figure 8, upper left) shows that entry 477 corresponds to entry $fai$ (a permutation)! (Note that given the base 10 value 477 we can determine its value in the multiplication table by doing a base 9 decode. Indeed, $477_{10} = 580_9$, which gives a multiset index of $[3, 8, 4]$ corresponding to the values $v(6) = f, v(9) = i, v(1) = a$.)

Returning to the problem of generating a random Kronecker graph, recall that (5.11) gave us a way of mapping from elements of the multiplication table to elements of the Kronecker matrix. This relationship, then, provides a computational tool to determine where an arbitrary element of the multiplication table generated by `grass_hop_region` occurs in the Kronecker matrix itself, providing the final piece of the fast Kronecker sampling methodology.

The code that maps indices of the multiplication table through Morton codes to the Kronecker indices is given in Listing 8. This code assumes that (5.11) is correct, which we will prove in the next section. In the code, the index in the multiplication table is converted into a single linear index. This conversion is just a base $n^2$ to base 10 conversion. Next, the linear index is MortonDecoded in base $n$. The decoding proceeds by assigning the least significant digit to the row index, then the next digit to the column, the next digit to the row, and so on. The most-significant digit then goes to the column value.

What remains is the proof that (5.11) is correct, which we tackle in the next section.

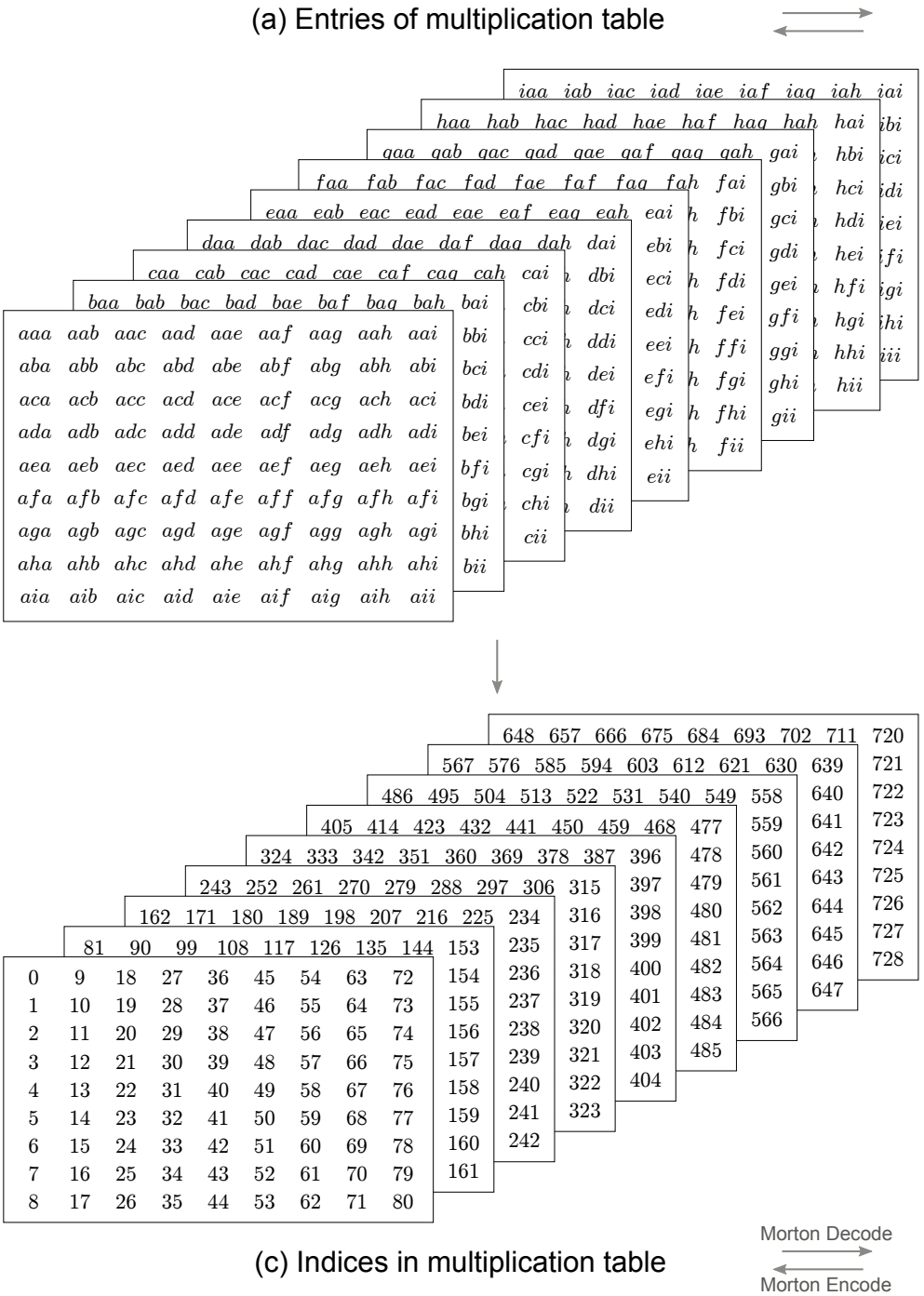



**Fig. 8** *The connection between the multiplication table and a repeated Kronecker matrix with a* 3*-by-*3 *matrix* $\boldsymbol{K}$ *with vector form* $\mathbf{v} = \begin{bmatrix} a & b & c & d & e & f & g & h & i \end{bmatrix}^T$. *This yields a* $9 \times 9 \times 9$ *multiplication table and a* 27*-by-*27 *matrix of probabilities.*

(b) Kronecker matrix

| | $000_3$ | $001_3$ | $002_3$ | $010_3$ | $011_3$ | $012_3$ | $020_3$ | $021_3$ | $022_3$ | $100_3$ | $101_3$ | $102_3$ | $110_3$ | $111_3$ | $112_3$ | $120_3$ | $121_3$ | $122_3$ | $200_3$ | $201_3$ | $202_3$ | $210_3$ | $211_3$ | $212_3$ | $220_3$ | $221_3$ | $222_3$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $000_3$ | *aaa* | *aad* | *aag* | *ada* | *add* | *adg* | *aga* | *agd* | *agg* | *daa* | *dad* | *dag* | *dda* | *ddd* | *ddg* | *dga* | *dgd* | *dgg* | *gaa* | *gad* | *gag* | *gda* | *gdd* | *gdg* | *gga* | *ggd* | *ggg* |
| $001_3$ | *aab* | *aae* | *aah* | *adb* | *ade* | *adh* | *agb* | *age* | *agh* | *dab* | *dae* | *dah* | *ddb* | *dde* | *ddh* | *dgb* | *dge* | *dgh* | *gab* | *gae* | *gah* | *gdb* | *gde* | *gdh* | *ggb* | *gge* | *ggh* |
| $002_3$ | *aac* | *aaf* | *aai* | *adc* | *adf* | *adi* | *agc* | *agf* | *agi* | *dac* | *daf* | *dai* | *ddc* | *ddf* | *ddi* | *dgc* | *dgf* | *dgi* | *gac* | *gaf* | *gai* | *gdc* | *gdf* | *gdi* | *ggc* | *ggf* | *ggi* |
| $010_3$ | *aba* | *abd* | *abg* | *aea* | *aed* | *aeg* | *aha* | *ahd* | *ahg* | *dba* | *dbd* | *dbg* | *dea* | *ded* | *deg* | *dha* | *dhd* | *dhg* | *gba* | *gbd* | *gbg* | *gea* | *ged* | *geg* | *gha* | *ghd* | *ghg* |
| $011_3$ | *abb* | *abe* | *abh* | *aeb* | *aee* | *aeh* | *ahb* | *ahe* | *ahh* | *dbb* | *dbe* | *dbh* | *deb* | *dee* | *deh* | *dhb* | *dhe* | *dhh* | *gbb* | *gbe* | *gbh* | *geb* | *gee* | *geh* | *ghb* | *ghe* | *ghh* |
| $012_3$ | *abc* | *abf* | *abi* | *aec* | *aef* | *aei* | *ahc* | *ahf* | *ahi* | *dbc* | *dbf* | *dbi* | *dec* | *def* | *dei* | *dhc* | *dhf* | *dhi* | *gbc* | *gbf* | *gbi* | *gec* | *gef* | *gei* | *ghc* | *ghf* | *ghi* |
| $020_3$ | *aca* | *acd* | *acg* | *afa* | *afd* | *afg* | *aia* | *aid* | *aig* | *dca* | *dcd* | *dcg* | *dfa* | *dfd* | *dfg* | *dia* | *did* | *dig* | *gca* | *gcd* | *gcg* | *gfa* | *gfd* | *gfg* | *gia* | *gid* | *gig* |
| $021_3$ | *acb* | *ace* | *ach* | *afb* | *afe* | *afh* | *aib* | *aie* | *aih* | *dcb* | *dce* | *dch* | *dfb* | *dfe* | *dfh* | *dib* | *die* | *dih* | *gcb* | *gce* | *gch* | *gfb* | *gfe* | *gfh* | *gib* | *gie* | *gih* |
| $022_3$ | *acc* | *acf* | *aci* | *afc* | *aff* | *afi* | *aic* | *aif* | *aii* | *dcc* | *dcf* | *dci* | *dfc* | *dff* | *dfi* | *dic* | *dif* | *dii* | *gcc* | *gcf* | *gci* | *gfc* | *gff* | *gfi* | *gic* | *gif* | *gii* |
| $100_3$ | *baa* | *bad* | *bag* | *bda* | *bdd* | *bdg* | *bga* | *bgd* | *bgg* | *eaa* | *ead* | *eag* | *eda* | *edd* | *edg* | *ega* | *egd* | *egg* | *haa* | *had* | *hag* | *hda* | *hdd* | *hdg* | *hga* | *hgd* | *hgg* |
| $101_3$ | *bab* | *bae* | *bah* | *bdb* | *bde* | *bdh* | *bgb* | *bge* | *bgh* | *eab* | *eae* | *eah* | *edb* | *ede* | *edh* | *egb* | *ege* | *egh* | *hab* | *hae* | *hah* | *hdb* | *hde* | *hdh* | *hgb* | *hge* | *hgh* |
| $102_3$ | *bac* | *baf* | *bai* | *bdc* | *bdf* | *bdi* | *bgc* | *bgf* | *bgi* | *eac* | *eaf* | *eai* | *edc* | *edf* | *edi* | *egc* | *egf* | *egi* | *hac* | *haf* | *hai* | *hdc* | *hdf* | *hdi* | *hgc* | *hgf* | *hgi* |
| $110_3$ | *bba* | *bbd* | *bbg* | *bea* | *bed* | *beg* | *bha* | *bhd* | *bhg* | *eba* | *ebd* | *ebg* | *eea* | *eed* | *eeg* | *eha* | *ehd* | *ehg* | *hba* | *hbd* | *hbg* | *hea* | *hed* | *heg* | *hha* | *hhd* | *hhg* |
| $111_3$ | *bbb* | *bbe* | *bbh* | *beb* | *bee* | *beh* | *bhb* | *bhe* | *bhh* | *ebb* | *ebe* | *ebh* | *eeb* | *eee* | *eeh* | *ehb* | *ehe* | *ehh* | *hbb* | *hbe* | *hbh* | *heb* | *hee* | *heh* | *hhb* | *hhe* | *hhh* |
| $112_3$ | *bbc* | *bbf* | *bbi* | *bec* | *bef* | *bei* | *bhc* | *bhf* | *bhi* | *ebc* | *ebf* | *ebi* | *eec* | *eef* | *eei* | *ehc* | *ehf* | *ehi* | *hbc* | *hbf* | *hbi* | *hec* | *hef* | *hei* | *hhc* | *hhf* | *hhi* |
| $120_3$ | *bca* | *bcd* | *bcg* | *bfa* | *bfd* | *bfg* | *bia* | *bid* | *big* | *eca* | *ecd* | *ecg* | *efa* | *efd* | *efg* | *eia* | *eid* | *eig* | *hca* | *hcd* | *hcg* | *hfa* | *hfd* | *hfg* | *hia* | *hid* | *hig* |
| $121_3$ | *bcb* | *bce* | *bch* | *bfb* | *bfe* | *bfh* | *bib* | *bie* | *bih* | *ecb* | *ece* | *ech* | *efb* | *efe* | *efh* | *eib* | *eie* | *eih* | *hcb* | *hce* | *hch* | *hfb* | *hfe* | *hfh* | *hib* | *hie* | *hih* |
| $122_3$ | *bcc* | *bcf* | *bci* | *bfc* | *bff* | *bfi* | *bic* | *bif* | *bii* | *ecc* | *ecf* | *eci* | *efc* | *eff* | *efi* | *eic* | *eif* | *eii* | *hcc* | *hcf* | *hci* | *hfc* | *hff* | *hfi* | *hic* | *hif* | *hii* |
| $200_3$ | *caa* | *cad* | *cag* | *cda* | *cdd* | *cdg* | *cga* | *cgd* | *cgg* | *faa* | *fad* | *fag* | *fda* | *fdd* | *fdg* | *fga* | *fgd* | *fgg* | *iaa* | *iad* | *iag* | *ida* | *idd* | *idg* | *iga* | *igd* | *igg* |
| $201_3$ | *cab* | *cae* | *cah* | *cdb* | *cde* | *cdh* | *cgb* | *cge* | *cgh* | *fab* | *fae* | *fah* | *fdb* | *fde* | *fdh* | *fgb* | *fge* | *fgh* | *iab* | *iae* | *iah* | *idb* | *ide* | *idh* | *igb* | *ige* | *igh* |
| $202_3$ | *cac* | *caf* | *cai* | *cdc* | *cdf* | *cdi* | *cgc* | *cgf* | *cgi* | *fac* | *faf* | *fai* | *fdc* | *fdf* | *fdi* | *fgc* | *fgf* | *fgi* | *iac* | *iaf* | *iai* | *idc* | *idf* | *idi* | *igc* | *igf* | *igi* |
| $210_3$ | *cba* | *cbd* | *cbg* | *cea* | *ced* | *ceg* | *cha* | *chd* | *chg* | *fba* | *fbd* | *fbg* | *fea* | *fed* | *feg* | *fha* | *fhd* | *fhg* | *iba* | *ibd* | *ibg* | *iea* | *ied* | *ieg* | *iha* | *ihd* | *ihg* |
| $211_3$ | *cbb* | *cbe* | *cbh* | *ceb* | *cee* | *ceh* | *chb* | *che* | *chh* | *fbb* | *fbe* | *fbh* | *feb* | *fee* | *feh* | *fhb* | *fhe* | *fhh* | *ibb* | *ibe* | *ibh* | *ieb* | *iee* | *ieh* | *ihb* | *ihe* | *ihh* |
| $212_3$ | *cbc* | *cbf* | *cbi* | *cec* | *cef* | *cei* | *chc* | *chf* | *chi* | *fbc* | *fbf* | *fbi* | *fec* | *fef* | *fei* | *fhc* | *fhf* | *fhi* | *ibc* | *ibf* | *ibi* | *iec* | *ief* | *iei* | *ihc* | *ihf* | *ihi* |
| $220_3$ | *cca* | *ccd* | *ccg* | *cfa* | *cfd* | *cfg* | *cia* | *cid* | *cig* | *fca* | *fcd* | *fcg* | *ffa* | *ffd* | *ffg* | *fia* | *fid* | *fig* | *ica* | *icd* | *icg* | *ifa* | *ifd* | *ifg* | *iia* | *iid* | *iig* |
| $221_3$ | *ccb* | *cce* | *cch* | *cfb* | *cfe* | *cfh* | *cib* | *cie* | *cih* | *fcb* | *fce* | *fch* | *ffb* | *ffe* | *ffh* | *fib* | *fie* | *fih* | *icb* | *ice* | *ich* | *ifb* | *ife* | *ifh* | *iib* | *iie* | *iih* |
| $222_3$ | *ccc* | *ccf* | *cci* | *cfc* | *cff* | *cfi* | *cic* | *cif* | *cii* | *fcc* | *fcf* | *fci* | *ffc* | *fff* | *ffi* | *fic* | *fif* | *fii* | *icc* | *icf* | *ici* | *ifc* | *iff* | *ifi* | *iic* | *iif* | *iii* |

| | $000_3$ | $001_3$ | $002_3$ | $010_3$ | $011_3$ | $012_3$ | $020_3$ | $021_3$ | $022_3$ | $100_3$ | $101_3$ | $102_3$ | $110_3$ | $111_3$ | $112_3$ | $120_3$ | $121_3$ | $122_3$ | $200_3$ | $201_3$ | $202_3$ | $210_3$ | $211_3$ | $212_3$ | $220_3$ | $221_3$ | $222_3$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $000_3$ | 0 | 3 | 6 | 27 | 30 | 33 | 54 | 57 | 60 | 243 | 246 | 249 | 270 | 273 | 276 | 297 | 300 | 303 | 486 | 489 | 492 | 513 | 516 | 519 | 540 | 543 | 546 |
| $001_3$ | 1 | 4 | 7 | 28 | 31 | 34 | 55 | 58 | 61 | 244 | 247 | 250 | 271 | 274 | 277 | 298 | 301 | 304 | 487 | 490 | 493 | 514 | 517 | 520 | 541 | 544 | 547 |
| $002_3$ | 2 | 5 | 8 | 29 | 32 | 35 | 56 | 59 | 62 | 245 | 248 | 251 | 272 | 275 | 278 | 299 | 302 | 305 | 488 | 491 | 494 | 515 | 518 | 521 | 542 | 545 | 548 |
| $010_3$ | 9 | 12 | 15 | 36 | 39 | 42 | 63 | 66 | 69 | 252 | 255 | 258 | 279 | 282 | 285 | 306 | 309 | 312 | 495 | 498 | 501 | 522 | 525 | 528 | 549 | 552 | 555 |
| $011_3$ | 10 | 13 | 16 | 37 | 40 | 43 | 64 | 67 | 70 | 253 | 256 | 259 | 280 | 283 | 286 | 307 | 310 | 313 | 496 | 499 | 502 | 523 | 526 | 529 | 550 | 553 | 556 |
| $012_3$ | 11 | 14 | 17 | 38 | 41 | 44 | 65 | 68 | 71 | 254 | 257 | 260 | 281 | 284 | 287 | 308 | 311 | 314 | 497 | 500 | 503 | 524 | 527 | 530 | 551 | 554 | 557 |
| $020_3$ | 18 | 21 | 24 | 45 | 48 | 51 | 72 | 75 | 78 | 261 | 264 | 267 | 288 | 291 | 294 | 315 | 318 | 321 | 504 | 507 | 510 | 531 | 534 | 537 | 558 | 561 | 564 |
| $021_3$ | 19 | 22 | 25 | 46 | 49 | 52 | 73 | 76 | 79 | 262 | 265 | 268 | 289 | 292 | 295 | 316 | 319 | 322 | 505 | 508 | 511 | 532 | 535 | 538 | 559 | 562 | 565 |
| $022_3$ | 20 | 23 | 26 | 47 | 50 | 53 | 74 | 77 | 80 | 263 | 266 | 269 | 290 | 293 | 296 | 317 | 320 | 323 | 506 | 509 | 512 | 533 | 536 | 539 | 560 | 563 | 566 |
| $100_3$ | 81 | 84 | 87 | 108 | 111 | 114 | 135 | 138 | 141 | 324 | 327 | 330 | 351 | 354 | 357 | 378 | 381 | 384 | 567 | 570 | 573 | 594 | 597 | 600 | 621 | 624 | 627 |
| $101_3$ | 82 | 85 | 88 | 109 | 112 | 115 | 136 | 139 | 142 | 325 | 328 | 331 | 352 | 355 | 358 | 379 | 382 | 385 | 568 | 571 | 574 | 595 | 598 | 601 | 622 | 625 | 628 |
| $102_3$ | 83 | 86 | 89 | 110 | 113 | 116 | 137 | 140 | 143 | 326 | 329 | 332 | 353 | 356 | 359 | 380 | 383 | 386 | 569 | 572 | 575 | 596 | 599 | 602 | 623 | 626 | 629 |
| $110_3$ | 90 | 93 | 96 | 117 | 120 | 123 | 144 | 147 | 150 | 333 | 336 | 339 | 360 | 363 | 366 | 387 | 390 | 393 | 576 | 579 | 582 | 603 | 606 | 609 | 630 | 633 | 636 |
| $111_3$ | 91 | 94 | 97 | 118 | 121 | 124 | 145 | 148 | 151 | 334 | 337 | 340 | 361 | 364 | 367 | 388 | 391 | 394 | 577 | 580 | 583 | 604 | 607 | 610 | 631 | 634 | 637 |
| $112_3$ | 92 | 95 | 98 | 119 | 122 | 125 | 146 | 149 | 152 | 335 | 338 | 341 | 362 | 365 | 368 | 389 | 392 | 395 | 578 | 581 | 584 | 605 | 608 | 611 | 632 | 635 | 638 |
| $120_3$ | 99 | 102 | 105 | 126 | 129 | 132 | 153 | 156 | 159 | 342 | 345 | 348 | 369 | 372 | 375 | 396 | 399 | 402 | 585 | 588 | 591 | 612 | 615 | 618 | 639 | 642 | 645 |
| $121_3$ | 100 | 103 | 106 | 127 | 130 | 133 | 154 | 157 | 160 | 343 | 346 | 349 | 370 | 373 | 376 | 397 | 400 | 403 | 586 | 589 | 592 | 613 | 616 | 619 | 640 | 643 | 646 |
| $122_3$ | 101 | 104 | 107 | 128 | 131 | 134 | 155 | 158 | 161 | 344 | 347 | 350 | 371 | 374 | 377 | 398 | 401 | 404 | 587 | 590 | 593 | 614 | 617 | 620 | 641 | 644 | 647 |
| $200_3$ | 162 | 165 | 168 | 189 | 192 | 195 | 216 | 219 | 222 | 405 | 408 | 411 | 432 | 435 | 438 | 459 | 462 | 465 | 648 | 651 | 654 | 675 | 678 | 681 | 702 | 705 | 708 |
| $201_3$ | 163 | 166 | 169 | 190 | 193 | 196 | 217 | 220 | 223 | 406 | 409 | 412 | 433 | 436 | 439 | 460 | 463 | 466 | 649 | 652 | 655 | 676 | 679 | 682 | 703 | 706 | 709 |
| $202_3$ | 164 | 167 | 170 | 191 | 194 | 197 | 218 | 221 | 224 | 407 | 410 | 413 | 434 | 437 | 440 | 461 | 464 | 467 | 650 | 653 | 656 | 677 | 680 | 683 | 704 | 707 | 710 |
| $210_3$ | 171 | 174 | 177 | 198 | 201 | 204 | 225 | 228 | 231 | 414 | 417 | 420 | 441 | 444 | 447 | 468 | 471 | 474 | 657 | 660 | 663 | 684 | 687 | 690 | 711 | 714 | 717 |
| $211_3$ | 172 | 175 | 178 | 199 | 202 | 205 | 226 | 229 | 232 | 415 | 418 | 421 | 442 | 445 | 448 | 469 | 472 | 475 | 658 | 661 | 664 | 685 | 688 | 691 | 712 | 715 | 718 |
| $212_3$ | 173 | 176 | 179 | 200 | 203 | 206 | 227 | 230 | 233 | 416 | 419 | 422 | 443 | 446 | 449 | 470 | 473 | 476 | 659 | 662 | 665 | 686 | 689 | 692 | 713 | 716 | 719 |
| $220_3$ | 180 | 183 | 186 | 207 | 210 | 213 | 234 | 237 | 240 | 423 | 426 | 429 | 450 | 453 | 456 | 477 | 480 | 483 | 666 | 669 | 672 | 693 | 696 | 699 | 720 | 723 | 726 |
| $221_3$ | 181 | 184 | 187 | 208 | 211 | 214 | 235 | 238 | 241 | 424 | 427 | 430 | 451 | 454 | 457 | 478 | 481 | 484 | 667 | 670 | 673 | 694 | 697 | 700 | 721 | 724 | 727 |
| $222_3$ | 182 | 185 | 188 | 209 | 212 | 215 | 236 | 239 | 242 | 425 | 428 | 431 | 452 | 455 | 458 | 479 | 482 | 485 | 668 | 671 | 674 | 695 | 698 | 701 | 722 | 725 | 728 |

Morton Decode

Morton Encode

(d) Multiplication table indices in Kronecker matrix

**Fig. 8** *(continued)*

**Listing 8** Mapping multiplication table indices to row and column indices of the Kronecker matrix

```
""" Map a multi-index from the mult. table
table to a row and column in the Kronecker
matrix. The input is:
  mind: the multi-index for the mult table
  n: the size of the initiator matrix K
Example:
  map_mult_to_kron([1,3],2)   # = (3,1)
  map_mult_to_kron([4,0,7],3) # = (10,11)
"""
def map_mult_to_kron(mind,n):
  I = multiindex_to_linear(mind,n*n)
  return morton_decode(I,n)

def multiindex_to_linear(mind,n2):
  I = 0
  base = 1
  for i in xrange(len(mind)-1,-1,-1):
    I += mind[i]*base
    base *= n2
  return I
```

```
def morton_decode(I,n):
  row = 0
  rowbase = 1
  col = 0
  colbase = 1
  i = 0
  while I > 0:
    digit = I%n
    I = I // n
    if i%2 == 0:
      row += rowbase*digit
      rowbase *= n
    else:
      col += colbase*digit
      colbase *= n
    i += 1
  return (row,col)
```

### 5.8. The Proof of the Multiplication Table to Kronecker Matrix Permutation.

In this section, we will prove (5.11) via the following theorem.

THEOREM 5.1. *Let $\boldsymbol{K}$ be an $n$-by-$n$ matrix and $\mathbf{v} = \text{vec}(\boldsymbol{K})$ be the column major representation of $\boldsymbol{K}$. Consider an element in the multiplication table*

$$\underbrace{\mathbf{v} \otimes \mathbf{v} \otimes \cdots \otimes \mathbf{v}}_{k \text{ terms}}$$

*with index $(r_1, r_2, \ldots, r_k)$. Let $I$ be the lexicographic index of the element $(r_1, \ldots, r_k)$. Then the base $n$ Morton decoding of $I$ provides the row and column indices of an element in*

$$\underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \cdots \otimes \boldsymbol{K}}_{k \text{ terms}}$$

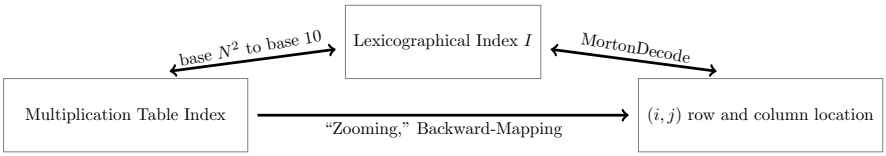
*with the same value. If we convert the row and column indices provided by Morton into a column major index, then we can build a permutation matrix $\boldsymbol{M}_{n,k}$ in order to write*

$$\text{vec}(\underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \cdots \otimes \boldsymbol{K}}_{k \text{ terms}}) = \boldsymbol{M}_{n,k}(\underbrace{\mathbf{v} \otimes \mathbf{v} \otimes \cdots \otimes \mathbf{v}}_{k \text{ terms}}).$$

*Proof.* To state the proof concisely, let $\boldsymbol{K}^{\otimes k} = \boldsymbol{K} \otimes \overset{k \text{ terms}}{\cdots} \otimes \boldsymbol{K}$ and $\mathbf{v}^{\otimes k}$ be the same. In the proof, we will just show that the Morton code for $I$ gives the correct row and column indices for $\boldsymbol{K}^{\otimes k}$. The resulting permutation matrix follows by converting those row and column indices to a column major index and building the full permutation matrix. We will prove this statement by induction on the power of the Kronecker product, $k$.

*Base case.* When $k = 1$, $\boldsymbol{K}^{\otimes k} = \boldsymbol{K}$ is simply the initiator matrix. Consequently, the theorem reads $\text{vec}(\boldsymbol{K}) = \boldsymbol{M}_{n,1}\mathbf{v}$; however, the length-1 Morton code is simply the row and column indices in column major order, and so this result follows because the map $\boldsymbol{M}_{n,1}$ is the identity.

*Inductive step overview.* We will prove our statement inductively by calculating the new row and column values in two ways: first by recursively defining the row and column values using a "zooming" argument, and second by applying the Morton decoding process on a recursively defined lexicographic index, $I_{k+1}$. The following figure gives an illustrative overview of how we will show the result in general.




*Inductive step formal.* Assume $\boldsymbol{M}_{n,k}$ correctly maps between $\mathbf{v}^{\otimes k}$ and $\boldsymbol{K}^{\otimes k}$. More specifically, this means we can find the row and column indices $(R_k, C_k)$ for a given index of the multiplication table $(r_1, r_2, \ldots, r_k)$.

Let $(r_1, r_2, \ldots, r_k, r_{k+1})$ be the multiplication table index for an entry of $\mathbf{v}^{\otimes k+1}$, then notice that $(r_1, r_2, \ldots, r_k)$ corresponds to an index in $\mathbf{v}^{\otimes k}$. By assumption, the Morton code correctly gives the correct row and column values in $\boldsymbol{K}^{\otimes k}$: $R_k$ and $C_k$. Note that the structure of the $\boldsymbol{K}^{\otimes k+1}$ is

$$\left[\begin{array}{c|c|c|c} K_{1,1}\boldsymbol{K}^{\otimes k} & K_{1,2}\boldsymbol{K}^{\otimes k} & \cdots & K_{1,n}\boldsymbol{K}^{\otimes k} \\ \hline K_{2,1}\boldsymbol{K}^{\otimes k} & K_{2,2}\boldsymbol{K}^{\otimes k} & \cdots & K_{2,n}\boldsymbol{K}^{\otimes k} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline K_{n,1}\boldsymbol{K}^{\otimes k} & K_{n,2}\boldsymbol{K}^{\otimes k} & \cdots & K_{n,n}\boldsymbol{K}^{\otimes k} \end{array}\right], \tag{5.25}$$

so the row value for $(r_1, r_2, \ldots, r_k, r_{k+1})$ is

$$R_{k+1} = R_k + (r_{k+1} \bmod n)n^k. \tag{5.26}$$

The reasoning for this is as follows. The size of the previous iteration's matrix is $n^k$. When we incorporate the new value $r_{k+1}$, we move to one of the $n^2$ areas (of dimension $n^k$ by $n^k$) of the Kronecker matrix, where $(r_1, r_2, \ldots, r_k)$ is the suffix. The specific row of the region can be found by simply taking the last multiplication table index $r_{k+1}$ mod $n$, which corresponds to looking up the row index for $r_{k+1}$ in column major order. The same reasoning can be used to find the column index, $j_{k+1}$:

$$C_{k+1} = C_k + \left\lfloor \frac{r_{k+1}}{n} \right\rfloor n^k. \tag{5.27}$$

By taking the floor of $r_{k+1}$ divided by $n$, we know the column index of the value for $r_{k+1}$. We call this procedure *backward-mapping* or *zooming* because we use the inductive step to *zoom* into the right cell.

Now we must show that the Morton decoding process arrives at the same updated row and column values. First we will calculate the lexicographic index of our length-$(k+1)$ multiplication table index. A given multiplication table index $(r_1, r_2, \ldots, r_k)$ has a lexicographic index $I_k$. When we move to the next Kronecker power, we incorporate the value $r_{k+1}$. We can determine the new lexicographic index $I_{k+1}$ using $I_k$ and $r_{k+1}$:

$$I_{k+1} = I_k + r_{k+1}(n^2)^k. \tag{5.28}$$

This relationship arises because the lexicographic index can be calculated by converting our multiplication table index from base $n^2$ to base 10. The last value $r_{k+1}$ will occupy the place value at the front of the linearized index, i.e., $(n^2)^k$.

Now we must show that Morton decoding correctly maps $I_{k+1}$ to $R_{k+1}$ and $C_{k+1}$. Using the division and remainder theorem on $r_{k+1}$ divided by $n$ results in

$$r_{k+1} = \left\lfloor \frac{r_{k+1}}{n} \right\rfloor n + (r_{k+1} \bmod n). \tag{5.29}$$

Substituting this representation into (5.28) yields

$$I_{k+1} = \left\lfloor \frac{r_{k+1}}{n} \right\rfloor n^{2k+1} + (r_{k+1} \bmod n) n^{2k} + I_k. \tag{5.30}$$

Thus, the last two digits that Morton decoding will find here are $\lfloor \frac{r_{k+1}}{n} \rfloor$ and $(r_{k+1} \bmod n)$, because these correspond to the final terms in a base $n$ representation of $I_{k+1}$. Consequently, Morton decoding of $I_{k+1}$ is equivalent to decoding $I_k$ and appending the digits $(r_{k+1} \bmod n)$ and $\lfloor \frac{r_{k+1}}{n} \rfloor$ to the beginning of the new row and column digit-strings, respectively. This completes our proof because (5.30) gives precisely the row and column values found in (5.26) and (5.27). ∎

Note that this proof actually suggests a more straightforward way to generate the row and column indices of the Kronecker matrix entries; see Problem 14 for more information.

**5.9. Fixing Ball-Dropping Using What We've Learned.** Recall that at the start of this section, we showed that a simple implementation of a ball-dropping procedure yielded the *wrong* distribution over edges for the Kronecker graph. The error in the implementation is subtle. First, we can easily check that the implementation of ball-dropping for a single edge is correct. The problem is that *after the first edge* is selected, subsequent edges are drawn from a different distribution because we ignore duplicate edges! For some quick intuition to help understand what happens, note that we are less likely to see duplicate samples on low-probability edges and more likely to see duplicates on high-probability edges. Also, in the data from (5.3) we saw the probability of low-probability edges increase and high-probability edges decrease.

Let us illustrate the precise issue in a simplified setting. Suppose we wanted to use ball-dropping to generate two distinct samples from the set $\{a, b, c, d\}$, where we have probabilities of $0.8, 0.6, 0.4, 0.2$ of sampling each one, respectively. In ball-dropping, we generate each sample with probability $0.4, 0.3, 0.2, 0.1$ (respectively). Say we sample $a$ first, with probability 0.4. Because we demand distinct samples, the probability of sampling $b$ next is $0.3/(0.3 + 0.2 + 0.1) = 0.5$. This gives a joint probability of $0.4 \cdot 0.5 = 1/5 = 3/15$. Each of these scenarios is listed in Table 1. By taking a sum over these possibilities, we end up with effective sampling probabilities $0.36, 0.30, 0.22, 0.12$, not $0.4, 0.3, 0.2, 0.1$ as desired.

We should mention that if all the probabilities are equal, as in the Erdős–Rényi case, then this problem does not occur. Working out the same type of probability table as above quickly confirms this intuition: This is why ball-dropping worked for the Erdős–Rényi graph. This insight suggests a strategy to adapt ball-dropping to correctly sample a Kronecker graph:

1. Determine the number of edges within each Erdős–Rényi region inside the Kronecker graph by sampling a binomial distribution.
2. Ball-drop edges, identify the Erdős–Rényi region they correspond to, and keep an edge *if* there are edges remaining for that region.

The only detail of this implementation we need to prescribe is how to determine the Erdős–Rényi region from the ball-dropped edges. This is unexpectedly easy, and we simply record the choices made within the Kronecker graph at each step; this

**Table 1** *A probability table describing what happens when we use ball-dropping to sample two distinct items from the four items $a, b, c, d$ with probabilities $0.8, 0.6, 0.4, 0.2$. The effective probability is the sum of all entries in the column divided by two, which gives the probability that we will see that item in the output. This table shows that even in a highly simplified setting, we would sample from a slightly different distribution using ball-dropping.*

| | | a | b | c | d |
|---|---|---|---|---|---|
| Probability of first choice | | 0.4 | 0.3 | 0.2 | 0.1 |
| Joint probability | a and then | – | 3/15 | 2/15 | 1/15 |
| | b and then | 12/70 | – | 6/70 | 3/70 |
| | c and then | 4/40 | 3/40 | – | 1/40 |
| | d and then | 4/90 | 3/90 | 2/90 | – |
| Effective probability | | 0.36 | 0.30 | 0.22 | 0.12 |

corresponds to the multiset for an Erdős–Rényi region. The overall procedure is implemented in Listing 9, which uses some of the other routines we've developed throughout this section. Note the only difference between `ball_drop_kron_mset` (Listing 9) and `ball_drop_kron_edge` (Listing 4) is that `ball_drop_kron_mset` records the information for the multiset corresponding to the edge. Because the edge corresponds with a permutation of the multiset information, we sort the multiset to uniquely identify each region. (As a technical detail, Python cannot use lists to index into a dictionary structure, and so we convert them into tuples.)

If we repeat the experiment from section 5.1 ($\boldsymbol{K} = \left[\begin{smallmatrix} 0.99 & 0.6 \\ 0.4 & 0.2 \end{smallmatrix}\right]$, $k = 3$, and 10,000,000 random trials), then we arrive at the following scaled edge probabilities arrive:

(5.31)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| .924 | .560 | .560 | .339 | .560 | .339 | .339 | .206 |
| .373 | .187 | .226 | .113 | .226 | .113 | .137 | .069 |
| .373 | .226 | .187 | .113 | .226 | .137 | .113 | .069 |
| .151 | .075 | .075 | .038 | .091 | .046 | .046 | .023 |
| .373 | .226 | .226 | .137 | .187 | .113 | .113 | .069 |
| .151 | .075 | .091 | .046 | .075 | .038 | .046 | .023 |
| .151 | .091 | .075 | .046 | .075 | .046 | .038 | .023 |
| .061 | .030 | .030 | .015 | .030 | .015 | .015 | .008 |

scaled true probabilities

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| .924 | .560 | .560 | .339 | .560 | .339 | .339 | .206 |
| .373 | .187 | .226 | .113 | .226 | .113 | .137 | .068 |
| .373 | .226 | .187 | .113 | .226 | .137 | .113 | .068 |
| .151 | .075 | .075 | .038 | .091 | .046 | .046 | .023 |
| .373 | .226 | .226 | .137 | .187 | .113 | .113 | .069 |
| .151 | .075 | .092 | .046 | .075 | .038 | .046 | .023 |
| .151 | .091 | .075 | .046 | .075 | .046 | .038 | .023 |
| .061 | .031 | .030 | .015 | .030 | .015 | .015 | .008 |

scaled empirical probabilities from Listing 9

(The elements that are different are flagged in red.) Note that we need to show results to three decimal places, which is actually the 4th decimal place because of the scaling for display, to see any differences between the two. Using additional trials show that these values converge as we would expect.

The downside to this implementation relates back to the overall issue with ball-dropping: we might have to resample for a long time until we satisfy all the requirements for all regions. This situation is worsened for sampling from Kronecker graphs because the probabilities are highly imbalanced. To measure this, we instrumented the code to record the number of ball drops we need sample a single edge for a few problems as $k$ varies. The results are given in Table 2. These data show that we often need hundreds of extra extra ball drops *per edge* and that this number exhibits very high standard deviations. Together, these results suggest that the corrected ball dropping implementation is unlikely to be efficient for sampling a Kronecker graph as $k$ grows.

**Listing 9** A correct implementation of ball-dropping to sample a Kronecker graph that first picks the number of edges in each Erdős–Rényi region and continues sampling until no region has any edges left

```
"""Correctly generate a ball-dropped
sample of a Kronecker graph.
Example: K = [[0.8,0.6],[0.4,0.2]]
ball_drop_kron_good(K, 3)
"""
def ball_drop_kron_good(K, k):
  # normalize to probability distribution
  n = len(K)  # and vectorize by columns
  Ksum = sum(v for ki in K for v in ki)
  p = [K[i][j]/Ksum for j in xrange(n)
    for i in xrange(n)]
  # for each ER region, sample a binomial
  # to get the num of edges in the region
  redges = {}
  for r in regions(p,k):
    pr = multtable(p,v) # get region prob
    nr = num_multiset_permutations(
           ndseq_to_counter(r)[0])
    redges[tuple(r)] =  # sample the binom
            int(np.random.binomial(nr,pr))
  edges = set() # the full set of edges
  num_edges = sum(redges.values())
  while len(edges) < num_edges:
    i,j,ms = ball_drop_kron_mset(p, n, k)
    ms.sort() # sort to order the multiset
    ms = tuple(ms)      # convert to a tuple
    if redges[ms] > 0:        # check if we
      if (i,j) not in edges: # need the edge
        edges.add((i,j))
        # decrease edges remaining in region
        redges[ms] -= 1
  return edges

"""Generate a single edge along with the
corresponding Erdos-Renyi region.
Example:
ball_drop_kron_mset([0.4,0.2,0.3,0.1],2,3)
"""
def ball_drop_kron_mset(p, n, k):
  R=0; C=0;
  offset = 1; n2 = n*n
  mset = [0 for _ in range(k)]
  for i in range(k):
    ind = np.random.choice(n2, 1, p=p)[0]
    mset[i] = ind
    # update row and col value
    row = ind % n; col = ind // n
    R += row * offset
    C += col * offset
    offset *= n
  return (R,C,mset)
```

**Table 2** *The mean and standard deviation, over* 100 *trials, of the number of ball drops required in Listing* 5.9 *for each edge of the final sample. As k grows, there are more drops required. These results show that the corrected ball-dropping implementation is unlikely to be efficient for sampling a Kronecker graph. Because of the high standard deviation, these results are not overly reliable.*

| Problem | Drops per edge | Size $k$ 6 | 8 | 10 | 12 | 14 | 16 |
|---|---|---|---|---|---|---|---|
| $\left[\begin{smallmatrix}0.99 & 0.5\\ 0.5 & 0.2\end{smallmatrix}\right]$ | Mean | 35 | 55 | 56 | 67 | 147 | 107 |
| | St. dev. | 96 | 188 | 87 | 79 | 496 | 172 |
| $\left[\begin{smallmatrix}0.8 & 0.6\\ 0.4 & 0.2\end{smallmatrix}\right]$ | Mean | 36 | 80 | 61 | 133 | 175 | 166 |
| | St. dev. | 90 | 356 | 132 | 710 | 488 | 458 |

The space of hybrid grass-hopping and ball-dropping solutions, however, could be fruitful.

**6. Problems and Pointers.** We have achieved our major goal, which was to establish a new link between Morton codes and repeated Kronecker products, and have described how that result is useful in efficiently generating a Kronecker graph. While we have not focused on producing the most efficient implementation possible (see Problem 14 for a nontrivial improvement), we did focus on a procedure that does only a small amount of additional work per edge and is faster than the ball-dropping procedures that have dominated the experimental landscape.

Our work builds strongly upon the work of Moreno et al. (2014) and highlights new connections to classic problems in combinatorics including ranking and unrank-

ing (Bonet, 2008). Indeed, the ranking and unranking perspective has a tremendous amount to offer for ever more complicated models of graphs. For instance, the original specification of the Erdős–Rényi graph due to Erdős and Rényi (1959) was a random choice of graphs with $n$ vertices and $m$ edges. It is straightforward to sample such an object from the ranking and unranking perspective. To each graph we associate a binary-string of length $\binom{n}{2}$, which represents the strictly upper-triangular region of an adjacency matrix. This binary-string must have exactly $m$ ones. To randomly generate such a string, first generate a random number between 0 and $\binom{\binom{n}{2}}{m} - 1$, then *unrank* that random number in lexicographic order. This procedure follows by testing whether the first number is 0 or 1. There are $\binom{\binom{n}{2}-1}{m}$ choices where the leading value is a 0 and $\binom{\binom{n}{2}-1}{m-1}$ choices where it is a 1, so we test whether our number is larger than $\binom{\binom{n}{2}-1}{m}$ and then assign the first digit. This process can be repeated to deterministically generate the string. Note that when straightforwardly implemented, as in Problem 10, this procedure is not at all efficient compared with alternatives, yet there are a tremendous number of effective sampling strategies that seek ways to optimize away inefficiencies; see, for instance, (Bonet, 2008) for some existing ideas to improve the speed of these processes. Our point is that a general view of ranking and unranking might open some analytical doors that would not have otherwise been opened.

We conclude our paper by (i) reviewing the major timelines of the literature, (ii) pointing out some strongly related work that falls outside of our specific scope, and (iii) providing some follow-up problems.

### 6.1. Recap of the Major Literature.

- 1959: Erdős–Rényi paper (Erdős and Rényi, 1959) with $m$ fixed edges
- 1959: Gilbert's model for $G(n, p)$ (Gilbert, 1959)
- 1962: Paper on grass-hopping or leap-frogging for sequential trials in statistics (Fan, Muller, and Rezucha, 1962)
- 1983: Stochastic block model proposed (Holland, Laskey, and Leinhardt, 1983)
- 1986: Grass-hopping for Erdős–Rényi graphs (Devroye, 1986)
- 2002: Chung–Lu graphs proposed (Chung and Lu, 2002)
- 2004: RMAT, Kronecker predecessor, proposed (Chakrabarti, Zhan, and Faloutsos, 2004)
- 2005: Kronecker graphs proposed (Leskovec et al., 2005)
- 2005: Grass-hopping for Erdős–Rényi graphs, again (Batagelj and Brandes, 2005)
- 2011: Grass-hopping-like procedure proposed for Chung–Lu (Miller and Hagberg, 2011)
- 2013: Observation on Erdős–Rényi blocks in Kronecker (Moreno et al., 2014)

**6.2. Pointers to More Complex Graph Models.** The models we consider here are not considered state of the art, although they are often used as synthetic examples due to their simplicity and ubiquity. In this section, we provide a few references to more intricate models that could be considered state of the art. We've attempted to limit our discussion to the most strongly related literature, since a full survey of random graph models is well beyond our scope.

**BTER: Kolda et al. (2014)** The block two-level Erdős–Rényi (BTER) graph models a collection of Erdős–Rényi subgraphs akin to the diagonal blocks of stochastic block models. However, it also includes a degree constraint that better models real-world networks.

**mKPGM: Moreno et al. (2010)** Another model of interest is called the mixed Kronecker product graph model (mKPGM), which is a generalization of the Kronecker model but allows for more variance as you might expect in real-world populations of networks. The main difference in this model compared to the standard Kronecker model that we have explored is that it is based on realizing an adjacency matrix at stages of the Kronecker product calculation. This corresponds to increasing or zeroing probabilities in the final Kronecker product, e.g., $\boldsymbol{P} = \boldsymbol{A}_\ell \otimes (\boldsymbol{K} \otimes \cdots \; k - \ell \text{ terms } \cdots \otimes \boldsymbol{K})$, where $\boldsymbol{A}_\ell$ is a realization of an $\ell$-level Kronecker graph with $\boldsymbol{K}$.

**Random Kernel Graph: Bollobás, Janson, and Riordan (2007)** This models a general, but specific, structure for $\boldsymbol{P}$, where $P_{i,j} = \min(f(v_i, v_j)/n, 1)$ and $f$ is a kernel function with a few special properties, and $v_i = i/n$ for each vertex. The Chung–Lu model can be expressed in this form and there is an efficient sampling algorithm due to Hagberg and Lemons (2025).

**Multifractal Networks: Palla, Lovász, and Vicsek (2010)** These are a continuous generalization of the Kronecker matrix models we consider in this paper and they generate a network based on a function $P(x, y)$ where $x$ and $y$ are in $[0, 1]$, but also where $P(x, y)$ is the result of repeated convolution of a generator (akin to repeated Kronecker products; see the referenced paper for details). With the resulting measure $P(x, y)$, we draw random coordinates $v_i$ for each node in $[0, 1]$ and generate a graph where $P_{ij} = P(v_i, v_j)$.

There are strong relationships among these classes of models. We conjecture that this idea of *grass-hopping* could be extended to *all* of these random graph models and, more generally, to all random graph generation mechanisms that have a small number of parameters and computationally efficient strategies to compute the probability of a given edge. (For an $n^k$ vertex Kronecker graph, there are $n^2$ parameters and each probability can be computed in $k$ steps.)

**6.3. Problems.** For the following problems, we have posted solutions or solution sketches in the supplementary material to this SIAM paper as well as to the version of the paper on arXiv: https://arxiv.org/pdf/1709.03438v1.pdf. However, we encourage readers to solve them independently!

PROBLEM 1 (easy). *When we were comparing ball-dropping to grass-hopping for sampling Erdős–Rényi graphs, we derived that an approximate count of the number of edges used in ball-dropping is* $m \log(1/(1-p))/p$. *First show that the term* $\log(1/(1-p))/p$ *has a removable singularity at* $p = 0$ *by showing that* $\lim_{p\to 0} \log(1/(1-p))/p = 1$. *Second, show that* $\log(1/(1-p))/p > 1$ *if* $p > 0$.

PROBLEM 2 (easy). *Implement a grass-hopping scheme for Chung–Lu as described in section* 4.

PROBLEM 3 (easy). *While we considered coin-flipping, ball-dropping, and grass-hopping for Erdős–Rényi, we did not do so for some of the other models. Of these, the Chung–Lu ball-dropping procedure is perhaps the most interesting. Recall how ball-dropping works for Erdős–Rényi. We determine how many edges we expect the graph to have. Then we drop balls into random cells until we obtain the desired number of edges. The way we dropped the balls was such that every single entry of the matrix was equally likely. The procedure for Chung–Lu works in a similar manner, but the selection of each cell is not equally likely. For simplicity, we describe it using the exact expected number of edges.*

*Note that the expected number of edges in a Chung–Lu graph is just*

$$E\left[\sum_i \sum_j A_{ij}\right] = \sum_i \sum_j E[A_{ij}] = \sum_i d_i,$$

*where we have used the result of* (2.7) *in order to introduce* $d_i$.

*Let* $m = \sum_i d_i$. *We will ball drop until we have* $m$ *edges.*

*The goal of each ball drop is to pick cell* $(i, j)$ *with probability* $d_i d_j / \sum_{k,l} d_k d_l$. *This forms a valid probability distribution over cells. The difference to* (2.6) *is subtle, but meaningful. Also note that each* $d_i$ *is an integer. Thus, to sample from this distribution, what we do is build a list where entry* $i$ *occurs exactly* $d_i$ *times. If we draw an entry out of this list, then the probability of drawing* $i$ *is exactly* $d_i / \sum_k d_k$. *Your first task is to show that if we draw two entries out of this list, sampling with replacement, the probability of the two drawn entries being* $i$ *and* $j$ *is*

$$d_i d_j / \sum_{k,l} d_k d_l.$$

*This sampling procedure is easy to implement: Just build the list and randomly pick an entry!*

*Your problem is then*

- *show that the probability we computed above is correct;*
- *implement this procedure on the computer akin to the Python codes we've seen;*
- *prepare a plot akin to Figure* 2 *for the Chung–Lu graphs as you vary the properties of the degree distribution.*

PROBLEM 4 (easy). *In section* 4, *we wrote a short code using* `grass_hop_er` *to generate a two-block stochastic block model (2SBM). In that short code, we generated larger Erdős–Rényi blocks for the* off-diagonal blocks *then necessary and only included edges if they fell in a smaller region. This inefficiency, which is severe if* $n_1 \gg n_2$, *can be addressed for a more general* `grass_hop_er` *function.*

*Implement* `grass_hop_er` *for an* $m \times n$ *region and rewrite the short two-block stochastic block model to use this more general routine to avoid the inefficiency.*

PROBLEM 5 (easy). *Suppose that you have the following function implemented:*

```
""" Generate a general stochastic block model where the input is:
ns: a list of integers for the size of each block (length k).
 Q: a k-by-k matrix of probabilities for the within block sizes. """
def sbm(ns, Q)
```

*Describe how you could implement a program to generate a Chung–Lu graph that makes a single call to* `sbm`.

PROBLEM 6 (medium, partially solved). *Implement and evaluate a ball-dropping procedure for the stochastic block model. To determine the number of edges, use the result about the number of edges in an Erdős–Rényi block and apply it to each block. To run ball-dropping, first pick a region based on the probability of* $\boldsymbol{Q}$ *(hint: you can do this with binary search and a uniform random variable), then pick from within that Erdős–Rényi block.*

PROBLEM 7 (medium). *Write an implementation to generate a stochastic block model graph using* `grass_hop_er` *as a subroutine.*

PROBLEM 8 (easy). *Extend all the programs in this paper and give them proper input validation so that they throw easy-to-understand errors if the input is invalid. (For instance, Erdős–Rényi graphs need the* `p` *input to be a probability.)*

PROBLEM 9 (medium). *In this paper, we calculated the number of Erdős–Rényi regions for a Kronecker graph assuming that the probabilities in* $\boldsymbol{K}$ *were nonsymmetric and there is no other structure in the coefficients that reduces the number of Erdős–Rényi regions. However, in Figure* 5*, we showed the Erdős–Rényi regions where the matrix* $\boldsymbol{K}$ *is symmetric. Determine a tight closed-form expression for the number of Erdős–Rényi regions when* $\boldsymbol{K}$ *is symmetric, again assuming no additional structure in the coefficients that would reduce the number of Erdős–Rényi regions.*

PROBLEM 10 (hard). *Implement the algorithm described in section* 6 *to generate an Erdős–Rényi graph with exactly* $m$ *entries by unranking binary-strings of length* $\binom{n}{2}$.

PROBLEM 11 (easy). *Implement a simple change to* `ball_drop_er` *that will generate nonedges when* $p > 0.5$ *and filter the list of all edges. Our own solution to this involves adding four lines (we used a few pieces of python syntactic sugar, but the point is that it isn't a complicated idea). Compare the runtimes of this approach with the standard ball-dropping approach.*

PROBLEM 12 (medium). *In section* 5.3*, we showed that the number of subregions is on the order of* $O(k^{n^2-1})$*. There are many ways to arrive at this result. Using Stirling's approximation for combinations, find a different way to count the number of subregions. Stirling's approximation is* $\log n! = n \log n - n + \frac{1}{2} \log n + \frac{1}{2} \log(2\pi) + \epsilon_n$.

PROBLEM 13 (easy). *In section* 5.1*, we claimed that the number of expected edges in the Kronecker graph with matrix* $\boldsymbol{K}$ *and* $k$ *levels was* $(\sum_{ij} K_{ij})^k$*. Prove this result.*

PROBLEM 14 (medium). *Note that in the proof of Theorem* 5.1*, we began by showing that the multiset region could be easily decoded into the row and column indices of the entry in the Kronecker matrix. Adapt our programs to use this insight to avoid computing the linear multiindex index returned by* `multiindex_to_linear`.

PROBLEM 15 (easy). *When we were unranking multisets, we would compute the number of multisets that started with a digit as the prefix and then enumerate the number of such sets. We used this to identify the first digit of the unranked set. Double check that we didn't miss any possibilities by summing.*

PROBLEM 16 (easy). *Determine the number of operations involved in each of the components of the procedure to generate a Kronecker graph.*

PROBLEM 17 (research level with appropriate generality). *Another view of Theorem* 5.1 *is that we have a permutation induced by two different representations of the same number. More specifically, the multi-index representation is in base* $n^2$*, which we convert to base* $n$*, and then we interleave digits to build a row and column representation. Hence, there are two equivalent representations of the number in base* 10*. This hints at a far more general set of permutations based on this type of digit interchange and base conversion. As an example, the* stride *permutation can be expressed as follows: represent a number from* $[0, n^2 - 1]$ *in base* $n$*. This gives a two-digit representation. Swap the digits and convert back to* $[0, n^2 - 1]$*. This gives a permutation matrix that exactly corresponds with what is called a stride permutation, which is closely related to the matrix transpose and the difference between row and column major orders. The*

*research problem is to generalize these results. Possible directions include the following questions: can any permutation matrix be expressed in this fashion? If not, can you exhibit one and characterize the class of permutations possible? A solution may have applications in terms of implementing circuits to compute permutations that arise in many common signal processing applications including cell phone signals (Mansour,* 2013*). One strategy to employ is to look at this problem from the perspective of group theory and study whether these digit interchanging permutations are generators of the symmetric group.*

**Acknowledgments.** We thank Sebastian Moreno for mentioning the original problem during his thesis defense and providing an excellent starting point, as well as Jennifer Neville. Along the way, we received helpful feedback from Eric Chi and C. Seshadhri. Finally, we wish to thank all of our research team and especially Caitlin Kennedy for a careful reading.

## REFERENCES

L. A. Adamic, R. M. Lukose, A. R. Puniyani, and B. A. Huberman, *Search in power-law networks*, Phys. Rev. E, 64 (2001), https://doi.org/10.1103/physreve.64.046135. (Cited on p. 556)

M. Aigner, G. M. Ziegler, K. H. Hofmann, and P. Erdos, *Proofs from the Book*, Springer, 2010. (Cited on p. 563)

A.-L. Barabási and R. Albert, *Emergence of scaling in random networks*, Science, 286 (1999), pp. 509–512, https://doi.org/10.1126/science.286.5439.509. (Cited on p. 553)

V. Batagelj and U. Brandes, *Efficient generation of large random networks*, Phys. Rev. E, 71 (2005), art. 036113, https://journals.aps.org/pre/abstract/10.1103/PhysRevE.71.036113. (Cited on pp. 565, 589)

B. Bollobás, S. Janson, and O. Riordan, *The phase transition in inhomogeneous random graphs*, Random Structures Algorithms, 31 (2007), pp. 3–122, https://doi.org/10.1002/rsa.20168. (Cited on p. 590)

B. Bonet, *Efficient algorithms to rank and unrank permutations in lexicographic order*, in AAAI Workshop on Search in AI and Robotics, 2008, pp. 142–151. (Cited on pp. 576, 589)

A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, *Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks*, in Proceedings of the Twenty-First Annual Symposium on Parallelism in Algorithms and Architectures, SPAA '09, ACM, New York, 2009, pp. 233–244, https://doi.org/10.1145/1583991.1584053. (Cited on p. 578)

D. Chakrabarti, Y. Zhan, and C. Faloutsos, *R-MAT: A recursive model for graph mining*, in Proceedings of the 2004 SIAM International Conference on Data Mining, 2004, pp. 442–446, https://doi.org/10.1137/1.9781611972740.43. (Cited on pp. 555, 589)

F. Chung and L. Lu, *Connected components in random graphs with given expected degree sequences*, Ann. Comb., 6 (2002), pp. 125–145, https://doi.org/10.1007/PL00012580. (Cited on p. 589)

F. N. David, *Games, Gods and Gambling: The Origins and History of Probability and Statistical Ideas from the Earliest Times to the Newtonian Era*, Hafner Publishing Company, 1962. (Cited on p. 565)

T. A. Davis, *Direct Methods for Sparse Linear Systems*, SIAM, Philadelphia, 2006, https://doi.org/10.1137/1.9780898718881. (Cited on p. 554)

B. Dawkins, *Siobhan's problem: The coupon collector revisited*, Amer. Statist., 45 (1991), pp. 76–82, http://www.jstor.org/stable/2685247. (Cited on pp. 563, 565)

L. Devroye, *Nonuniform Random Variate Generation*, Springer-Verlag, 1986, http://luc.devroye.org/rnbookindex.html. (Cited on pp. 561, 565, 589)

P. Diaconis and S. Holmes, *A Bayesian peek into Feller volume* I, Sankhyā Ser. A, 64 (2002), pp. 820–841. (Cited on p. 563)

S. Dill, R. Kumar, K. S. Mccurley, S. Rajagopalan, D. Sivakumar, and A. Tomkins, *Self-similarity in the web*, ACM Trans. Internet Tech., 2 (2002), pp. 205–223. (Cited on p. 556)

P. Erdős and A. Rényi, *On random graphs* I, Publ. Math., 6 (1959), pp. 290–297. (Cited on pp. 554, 589)

M. D. Ernst, *Permutation methods: A basis for exact inference*, Statist. Sci., 19 (2004), pp. 676–685, https://doi.org/10.1214/088342304000000396. (Cited on p. 552)

C. T. Fan, M. E. Muller, and I. Rezucha, *Development of sampling plans by using sequential (item by item) selection techniques and digital computers*, J. Amer. Statist. Assoc., 57 (1962), pp. 387–402. (Cited on pp. 565, 589)

R. A. Fisher, *"The coefficient of racial likeness" and the future of craniometry*, J. Royal Anthropol. Inst. Great Britain and Ireland, 66 (1936), pp. 57–63, http://www.jstor.org/stable/2844116. (Cited on p. 552)

G. W. Flake, S. Lawrence, and C. L. Giles, *Efficient identification of web communities*, in Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '00, ACM, New York, 2000, pp. 150–160, https://doi.org/10.1145/347090.347121. (Cited on p. 557)

J. E. Gentle, *Random Number Generation and Monte Carlo Methods*, Springer, New York, 2003, https://doi.org/10.1007/b97336. (Cited on p. 555)

E. N. Gilbert, *Random graphs*, Ann. Math. Statist., 30 (1959), pp. 1141–1144, https://doi.org/10.1214/aoms/1177706098. (Cited on pp. 554, 589)

C. M. Grinstead and J. L. Snell, *Introduction to Probability*, American Mathematical Society, 2012. (Cited on pp. 561, 565)

C. Groër, B. D. Sullivan, and S. Poole, *A mathematical analysis of the R-MAT random graph generator*, Networks, 58 (2011), pp. 159–170, https://doi.org/10.1002/net.20417. (Cited on p. 566)

A. Hagberg and N. Lemons, *Fast generation of sparse random kernel graphs*, PLoS One, (2015), http://journals.plos.org/plosone/article/metrics?id=10.1371/journal.pone.0135177. (Cited on pp. 565, 566, 590)

A. Hald, *A History of Probability and Statistics and Their Applications before* 1750, Wiley Ser. Probab. Math. Statist. 501, John Wiley & Sons, 2003. (Cited on p. 565)

P. W. Holland, K. B. Laskey, and S. Leinhardt, *Stochastic blockmodels: First steps*, Social Networks, 5 (1983), pp. 109–137, https://doi.org/10.1016/0378-8733(83)90021-7. (Cited on p. 589)

I. M. Kloumann, J. Ugander, and J. Kleinberg, *Block models and personalized PageRank*, Proc. Natl. Acad. Sci. USA, 114 (2016), pp. 33–38, https://doi.org/10.1073/pnas.1611275114. (Cited on p. 552)

D. E. Knuth, *The Art of Computer Programming, Volume* 2*: Seminumerical Algorithms*, 3rd ed., Addison-Wesley Longman, Boston, MA, 1997. (Cited on p. 555)

T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, *A scalable generative graph model with community structure*, SIAM J. Sci. Comput., 36 (2014), pp. C424–C452, https://doi.org/10.1137/130914218. (Cited on pp. 552, 589)

M. Koyutürk, A. Grama, and W. Szpankowski, *Assessing significance of connectivity and conservation in protein interaction networks*, in Research in Computational Molecular Biology, A. Apostolico, C. Guerra, S. Istrail, P. A. Pevzner, and M. Waterman, eds., Lecture Notes in Comput. Sci. 3909, Springer, Berlin, 2006, pp. 45–59, https://doi.org/10.1007/11732990_4. (Cited on pp. 551, 552)

M. Koyutürk, W. Szpankowski, and A. Grama, *Assessing significance of connectivity and conservation in protein interaction networks*, J. Comput. Biol., 14 (2007), pp. 747–764, https://doi.org/10.1089/cmb.2007.R014. (Cited on pp. 551, 552)

J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos, *Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication*, in Knowledge Discovery in Databases: PKDD 2005, Lecture Notes in Comput. Sci. 3721, Springer, Berlin, 2005, pp. 133–145, https://doi.org/10.1007/11564126_17. (Cited on pp. 555, 589)

J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani, *Kronecker graphs: An approach to modeling networks*, J. Mach. Learn. Res., 11 (2010), pp. 985–1042, http://www.jmlr.org/papers/volume11/leskovec10a/leskovec10a.pdf. (Cited on pp. 555, 556)

N. Litvak, W. Scheinhardt, and Y. Volkovich, *In-degree and PageRank of Web pages: Why do they follow similar power laws?*, Internet Math., 7 (2007), pp. 175–198. (Cited on p. 556)

M. M. Mansour, *Pruned bit-reversal permutations: Mathematical characterization, fast algorithms and architectures*, IEEE Trans. Signal Process., 61 (2013), pp. 3081–3099, https://doi.org/10.1109/TSP.2013.2245656. (Cited on p. 593)

J. C. Miller and A. Hagberg, *Efficient generation of networks with given expected degrees*, in Algorithms and Models for the Web Graph, Lecture Notes in Comput. Sci. 6732, Springer, Berlin, 2011, pp. 115–126. (Cited on pp. 566, 589)

R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, *Network motifs: Simple building blocks of complex networks*, Science, 298 (2002), pp. 824–827, https://doi.org/10.1126/science.298.5594.824. (Cited on pp. 551, 552, 556)

S. Moreno, S. Kirshner, J. Neville, and S. V. N. Vishwanathan, *Tied Kronecker product graph models to capture variance in network populations*, in 2010 48th Annual Allerton Conference on

Communication, Control, and Computing, Allerton, 2010, pp. 1137–1144, https://doi.org/10.1109/ALLERTON.2010.5707038. (Cited on p. 590)

S. Moreno and J. Neville, *Network hypothesis testing using mixed Kronecker product graph models*, in 2013 IEEE 13th International Conference on Data Mining, 2013, pp. 1163–1168, https://doi.org/10.1109/ICDM.2013.165. (Cited on p. 551)

S. Moreno, J. J. Pfieffer, J. Neville, and S. Kirshner, *A scalable method for exact sampling from Kronecker family models*, IEEE International Conference on Data Mining (ICDM), 2014, pp. 440–449. (Cited on pp. 560, 567, 588, 589)

G. M. Morton, *A Computer Oriented Geodetic Data Base; and a New Technique in File Sequencing*, Tech. report, IBM, 1966, https://domino.research.ibm.com/library/cyberdig.nsf/0/0dabf9473b9c86d48525779800566a39. (Cited on p. 578)

R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, *Introducing the Graph* 500, Cray User's Group, 2010. (Cited on pp. 552, 556, 566, 567)

M. E. J. Newman, *Random graphs with clustering*, Phys. Rev. Lett., 103 (2009), art. 058701, https://doi.org/10.1103/PhysRevLett.103.058701. (Cited on p. 552)

M. E. J. Newman and M. Girvan, *Finding and evaluating community structure in networks*, Phys. Rev. E, 69 (2004), art. 026113, https://doi.org/10.1103/PhysRevE.69.026113. (Cited on p. 557)

J. A. Orenstein and T. H. Merrett, *A class of data structures for associative searching*, in Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS '84, ACM, New York, 1984, pp. 181–190, https://doi.org/10.1145/588011.588037. (Cited on p. 578)

G. Palla, L. Lovász, and T. Vicsek, *Multifractal network generator*, Proc. Natl. Acad. Sci. USA, 107 (2010), pp. 7640–7645, https://doi.org/10.1073/pnas.0912983107. (Cited on p. 590)

P. Pollack, *Euler and the partial sums of the prime harmonic series*, Elem. Math, 70 (2015), pp. 13–20. (Cited on p. 563)

E. Ravasz and A.-L. Barabási, *Hierarchical organization in complex networks*, Phys. Rev. E, 67 (2003), art. 026112. (Cited on p. 556)

C. Seshadhri, A. Pinar, and T. G. Kolda, *An in-depth analysis of stochastic Kronecker graphs*, J. ACM, 60 (2013), pp. 13:1–13:32, https://doi.org/10.1145/2450142.2450149. (Cited on p. 556)

J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *The Boost Graph Library: User Guide and Reference Manual*, Addison-Wesley Professional, 2001. (Cited on p. 561)

D. E. Smith, *A Source Book in Mathematics*, Courier Corporation, 2012. (Cited on p. 565)

R. P. Stanley, *What is enumerative combinatorics?*, in Enumerative Combinatorics, Springer, 1986, pp. 1–63. (Cited on p. 571)

M. Vinkler, J. Bittner, and V. Havran, *Extended Morton codes for high performance bounding volume hierarchy construction*, in Proceedings of High Performance Graphics, HPG '17, ACM, New York, 2017, pp. 9:1–9:8, https://doi.org/10.1145/3105762.3105782. (Cited on p. 578)

E. Yu, http://math.stackexchange.com/questions/247569, 2012. Accessed January 4, 2017. (Cited on p. 563)