SUPPLEMENTARY MATERIALS: COIN-FLIPPING, BALL-DROPPING, AND GRASS-HOPPING FOR GENERATING RANDOM GRAPHS FROM MATRICES OF EDGE PROBABILITIES[*]

ARJUN S. RAMANI[†], NICOLE EIKMEIER[‡], AND DAVID F. GLEICH[§]

Problems. For the following problems, we posted solutions or solution sketches in the version of the paper on arXiv: https://arxiv.org/pdf/1709.03438v1.pdf. However, we encourage readers to solve them individually!

Problem 1 (Easy). *When we were comparing ball-dropping to grass-hopping for sampling Erdős-Rényi graphs, we derived that an approximate count for 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* $p > 0$.

Solution. We need to use L'Hôpital's rule for the first question, in which was we get $(1/(1-p))/1$ for the ratio of derivatives. This clearly tends to 1. For the second part, we take the Taylor series of $\log(1/(1-p))$, which is $p + p^2/2 + p^3/3 + \ldots$. This is always at least p when $p > 0$. (Note, that, indeed, this series divided by p gives $1 + p/2 + p^2/3 + \ldots$) ✓

Problem 2 (Easy). *Implement a grass-hopping scheme for Chung-Lu as described in Section 4.*

Solution. Note that this code uses the method grass_hop_er from listing 3.

```
"""Create a Chung-Lu graph via grass-hopping
Note this routine is only for pedagogical use as
It can be improved in many ways.
Example: chung_lu_grasshop([5,1,1,2,2,1])"""
def chung_lu_grasshop(d):
  # sort d with a permutation
  # http://stackoverflow.com/questions/7851077/how-to-return-index-of-a-sorted-list
  dperm = sorted(range(len(d)), key=lambda k: d[k])
  # build regions
  n = 0
  regions = []
  while n < len(dperm):
    first = n
    deg = d[dperm[n]]
    while n < len(dperm)-1 and d[dperm[n+1]] == deg:
      n += 1
    regions.append((first,n))
  # grasshop within regions
  vol = sum(d)
  edges = []
  for ri in xrange(len(regions)):
    for rj in xrange(len(regions)):
      rowinds = regions[ri] # start/end of row regions
```

[*]Funding: Partially supported by the NSF funded NSF CAREER award CCF-1149756, NSF Big data IIS-1546488, NSF Center for Science of Information STC, CCF-093937, DARPA SIMPLEX and the Sloan Foundation.

[†]West Lafayette Jr/Sr High School, 1105 North Grant Street, West Lafayette, IN 47906, USA (aramani@purdue.edu).

[‡]Department of Mathematics, Purdue University, eikmeier@purdue.edu

[§]Department of Computer Science, Purdue University, 305 North University Avenue, West Lafayette, IN 47907, USA (dgleich@purdue.edu).

```
    colinds = regions[rj] # start/end of col region
    p = d[dperm[rowinds[0]]]*d[dperm[colinds[0]]]/vol
    rowsize = rowinds[1] - rowinds[0] + 1
    colsize = colinds[1] - colinds[0] + 1
    er_edges = grass_hop_er(max(rowsize, colsize), p)
    edges.extend( [ (i+rowinds[0], j+colinds[0])
            for i,j in er_edges if i < rowsize and j < colsize ] )

# build inverse permutation to re-permute edge indices
iperm = [0 for _ in dperm] # inverse permutation
for i,j in enumerate(dperm):
  iperm[j] = i
# reverse permute edges
return [ (iperm[e[0]], iperm[e[1]]) for e in edges ]
```

✓

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 them was so that every single entry of the matrix was equally likely. The procedure for Chung-Lu works in a similar manner, but we won't select each cell 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

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

where we used the result of equation (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 from equation (2.6) is subtle, but meaningful. Also note that each d_i is an integer. So 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 getting 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.*

Solution. If we draw two samples from the list. The draws are independent, so the probability of the event is the product of each draw's probability. Each occurs with probability $d_i / \sum_k d_k$. So if we saw i and then j, this even occurs with probability

$$\frac{d_i}{\sum_k d_k} \cdot \frac{d_j}{\sum_k d_k} = \frac{d_i d_j}{\sum_{k,\ell} d_k d_l}.$$

```
""" Build a list where entry i occurs d[i] times. """
def build_list(d):
  L = []
  for i in xrange(len(d)):
     L.extend( [i for _ in xrange(d[i])] )
  return L

"""Create a Chung-Lu graph via ball-dropping.
Note this routine is only for pedagogical use as
It can be improved in many ways.
Example: chung_lu_ball_drop([5,1,1,2,2,1])"""
def chung_lu_ball_drop(d):
  M = sum(d)
  L = build_list(d)
  edges = set()
  dups = 0
  while len(edges) < M:
    src = L[random.randint(0,len(L)-1)]
    dst = L[random.randint(0,len(L)-1)]
    if (src,dst) not in edges:
      edges.add((src,dst))
    else:
      dups += 1
  return (list(edges), dups)
```

✓

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*

Solution.

```
def grass_hop_er(m,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 < m*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
def sbm2(n1,n2,p,q):
  edges = grass_hop_er(n1,n1,p)                          # generate the n1-by-n1 block
  edges.extend( [ (i+n1,j+n1) for i,j in grass_hop_er(n2,n2,p) ])  # n2-by-n2 block
  edges.extend( [ (i,j+n1) for i,j in grass_hop_er(n1,n2,q) ])     # n1-by-n2 block
  edges.extend( [ (i+n1,j) for i,j in grass_hop_er(n2,n1,q) ])     # n2-by-n1 block
  return edges
```

✓

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.

Solution. Recall from section 4 that we can arrange the nodes of our graph in increasing degree order, which forms a block structure. Sort the nodes in this way, and set $ns = [n_1, \ldots, n_k]$, where n_i is the number of degrees with the *ith* largest unique degree. Store the values $\frac{d_{[i]}d_{[j]}}{\rho}$ in Q_{ij}. Then run the sbm procedure on this output. ✓

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 within that Erdős-Rényi block.*

Solution. We are going to sketch a solution and outline some interesting directions to take as far as extending. The solution involves sampling an entry at random based on the probabilities in $\boldsymbol{Q}$. This can be done with a routine such as those described here: http://stackoverflow.com/questions/4437250/choose-list-variable-given-probability-of-each-variable. Then, once you have the region of $\boldsymbol{Q}$, you ball-drop like in Erdős-Rényi. This assumes there is no structure in $\boldsymbol{Q}$. A more interesting case is where there is structure in $\boldsymbol{Q}$. For instance, where there is a single within group probability p and a single between group probability q such that $\boldsymbol{Q} = (p-q)\boldsymbol{I} + q\mathbf{e}\mathbf{e}^T$ or $Q_{ij} = p$ if $i = j$ and $Q_{ij} = q$ if $i \neq j$, and where all the sizes of each group are equal. In this case, you can first choose if the ball will go into the diagonal or non-diagonal region. Then conditional on that choice, pick an element of $\boldsymbol{Q}$. Then do an Erdős-Rényi ball-drop within the region of $\boldsymbol{P}$ that corresponds to an element of $\boldsymbol{Q}$. ✓

Problem 7 (Medium). *Write an implementation to generate a stochastic block model graph using* grass_hop_er *as a sub-routine.*

Solution.

```
#Generate a general stochastic block model using grass-hopping 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_er(ns, Q):
    edges = [];
    for i in range(len(ns)):
        for j in range(len(ns)): #loop through each block
            m=ns[i]; n=ns[j]; #find m x n size of block
            row_loc=0;
            for k in range(i):
                row_loc+=ns[k]; #locate row of top left corner of block
            col_loc=0;
            for l in range(j):
                col_loc+=ns[l]; #locate column of top left corner of block
            for e in grass_hop_er(m,n,Q[i][j]): #find edges
                edges.append([e[0]+row_loc,e[1]+col_loc]); #add edges updated with top left corner
    return edges;
```

✓

Problem 8 (Easy). *Extend all the programs in this tutorial to have 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 the paper, we worked out the number of Erdős-Rényi regions for a Kronecker graph assuming that the probabilities in $\boldsymbol{K}$ were non-symmetric and there is no other structure in the coefficients that reduces the number of Erdős-Rényi regions. But in Figure 5, we showed the Erdős-Rényi regions where the matrix $\boldsymbol{K}$ was 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.*

Solution. Note that the kth Kronecker power of $\boldsymbol{K}$ contains all length-k permutations of the probabilities contained in $\boldsymbol{K}$. Thus, it suffices to count the number of unique probabilities in $\boldsymbol{K}$ and apply the same stars-and-bars technique shown in 5.3. The number of unique values in an n x n symmetric matrix is equivalent to the number of values in the upper-triangular matrix: $\binom{n+1}{2}$. In order to count the number of length-k sequences of these values, we apply the stars-and-bars formula and attain $\binom{\binom{n+1}{2}+k-1}{k}$ ✓

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 $\binom{n}{2}$.*

Solution.

```python
import random
from scipy.special import comb

# unrank takes as input
#   I: the rank of the graph
#   max_val: the number of potential edges
#   length: the number of edges
# and returns the Ith non-decreasing sequence of edges
# of size 'length' and maximum edge of value (max_val-1)
def unrank(I, max_val, length):
  array = [];
  # finds the next edge value
  for i in range(max_val):
    if length == 0:
      break;
    if length == 1:
      array.append(int(i + I));
      length=length-1;
    # checks if next edge value is i
    elif(comb(max_val-i-1, length-1) > I):
      array.append(int(i));
      length = length - 1;
    else:
      I = I - comb(max_val-i-1, length-1);
  return array;

# fixed_er takes as input:
#   n: the number of nodes
#   m: the number of edges
# And returns a list of edges for a graph with n nodes and m edges

def fixed_er(n,m):
  graph = [];
  #computes the number of possible edges
  possible_edges = int(n*(n-1)/2);
  #computes the number of possible graphs
  possible_graphs = int(comb(possible_edges,m));

  #randomly generates a "rank" or number of the graph
  I = int(random.random()*possible_graphs)

  #finds the m edges that make up the graph
  edges = unrank(I, possible_edges, m);

  #finds the nodes of edges: a set of edges.
  for edge in edges:
    nodes = unrank(edge, n, 2);
    graph.append([nodes[0], nodes[1]]);
  return graph;
```

✓

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

Solution.

```
def ball_drop_er(n,p):
  if p > 0.5:
    nonedges = ball_drop_er(n,1-p)
    alledges = set((i,j) for i in xrange(n) for j in xrange(n))
    return list(alledges.difference(nonedges))
  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)
```

✓

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:* $\log n! = n \log n - n + \frac{1}{2}\log n + \frac{1}{2}\log(2\pi) + \epsilon_n$

Solution. The number of subregions is given by $\binom{n^2+k-1}{k}$, which expands to $\left(\frac{(n^2+k-1)!}{(n^2-1)!k!}\right)$. Taking the log of our expression allows us separate out the components of the combination and arrive at: $\log(n^2+k-1)! - \log(n^2-1)! - \log(k!)$. Now we can substitute in Stirling's approximation for each $\log k!$ term and simplify arriving at: $k \log \frac{n^2+k-1}{k} + (n^2-1)(\log(n^2+k-1)-1) + \frac{1}{2}\log \frac{n^2+k-1}{k}$. Note that our simplification process is made easier by eliminating all constant terms (those with no k). Using L'Hopital's Rule, it can be found that $\lim_{k\to\infty} k \log \frac{n^2+k-1}{k} = n^2 - 1$. Furthermore, at the limit, $\frac{1}{2}\log \frac{n^2+k-1}{k}$ goes to 0. Thus, we can simplify our expression further to $(n^2-1)\log n^2 + k - 1$. Exponentiating this expression yields $(n^2+k-1)^{n^2-1}$, which is $O(k(n^2-1)$ as desired. ✓

Problem 13 (Easy). *In Section 5.1, we claimed 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.*

Solution. The expected number of edges in the Kronecker graph is just the expected number of times the Bernoulli random variables come up heads. So for

$$\boldsymbol{P} = \underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \ldots \otimes \boldsymbol{K}}_{k\text{ times}},$$

the expected number of edges is $\sum_{ij} P_{ij}$. Hence we really need to compute $\sum_{ij}(\boldsymbol{K} \otimes \boldsymbol{K} \otimes \ldots \otimes \boldsymbol{K})$. We can do this inductively on k. Note that the result is simple for $k = 1$. Recall the recursive nature of $\boldsymbol{P}$:

$$\boldsymbol{P} = \begin{bmatrix} K_{11}\underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \ldots \otimes \boldsymbol{K}}_{k-1\text{ times}} & K_{1,2}\underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \ldots \otimes \boldsymbol{K}}_{k-1\text{ times}} & \ldots \\ K_{2,1}\underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \ldots \otimes \boldsymbol{K}}_{k-1\text{ times}} & \ddots & \\ \vdots & & \end{bmatrix}$$

Let σ be the sum of entries in $\underbrace{\boldsymbol{K} \otimes \boldsymbol{K} \otimes \ldots \otimes \boldsymbol{K}}_{k-1\text{ times}}$. So the sum of entries of $\boldsymbol{P}$ is just $\sum_{ij} K_{ij}\sigma = (\sum_{ij} K_{ij})\sigma$ where $\sigma = (\sum_{ij} K_{ij})^{k-1}$ by induction. ✓

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 index 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`.

Solution.

```
# backward_map takes as input
#    s: an array (edge probability sequence) consisting of a
#       set letters represented as numbers e.g. [0,1,3,1]
#    n: the size of the n x n initiator matrix
# and returns the corresponding row or column index of the
# probability sequence in the kronecker product graph matrix
def backward_map(s,n):
    row=0
    col=0
    # we want to "zoom" further into the matrix for each
    # value of our cell
    for i, r in enumerate(s):
        length = len(s)-i;
        row += (r mod n) * math.pow(n,length-1);
        col += int(r/n) * math.pow(n,length-1);
    return [row,col];
```

✓

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*

Solution. The number of multiset permutations is given by

$$\frac{k!}{\prod_{i=0}^{n^2} r_i!} \tag{0.1}$$

Removing an instance of the first letter results in $\frac{(k-1)!r_0}{\prod_{i=0}^{n^2} r_i!}$. In general, we can compute the sum of all permutations with one digit removed as follows:

$$\sum_{j=1}^{n^2} \frac{(k-1)!r_j}{\prod_{i=0}^{n^2} r_i!} \tag{0.2}$$

It suffices to show that

$$\frac{k!}{\prod_{i=0}^{n^2} r_i!} = \sum_{j=1}^{n^2} \frac{(k-1)!r_j}{\prod_{i=0}^{n^2} r_i!} \tag{0.3}$$

Canceling common terms yields

$$k = \sum_{j=1}^{n^2} r_j \tag{0.4}$$

This is true because the sum of all r_j, the count of each letter, is just the length of the multiset. ✓

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

Solution. regions: In listing 5, every region is visited and the $update_region$ method is performed. $Update_region$ requires k calculations in the worst case because we must spill over to every index in the region. Thus, our complexity is $O(number_of_regions \cdot k)$.

grass_hop_region: From viewing 7, the number of operations is on the order of the number of edges, $|E|$ times the number of operations in the unrank algorithm. Unranking requires k^3 calculations bringing our total count to $O(|E|k^3)$. (To place one digit, we might have to look at k entries and do $O(k)$ work for each entry to compute the number of multisets; this can be improved to $O(k^2)$ time by using an updating formula that avoids full recomputation of the number of multiset permutations. See our Julia codes for this technique.)

mult_to_kron: determining the linear index takes $2k$ calculations because we simply have to loop through all the elements in our multiset index and make 2 calculations in the loop. Morton Decoding takes $10k$ calculations because we must loop through 2k digits and do 5 calculations in each iteration. $12k$ calculations is $O(k)$. ✓

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 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: 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 order. The research problem is to generalize these results. Possible directions include: 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 if these digit interchanging permutations are generators of the symmetric group.*

REFERENCES

M. M. Mansour. *Pruned bit-reversal permutations: Mathematical characterization, fast algorithms and architectures*. IEEE Transactions on Signal Processing, 61 (12), pp. 3081–3099, 2013. `doi:10.1109/TSP.2013.2245656`. Cited on page 9.