STAT 535C Fall 2024 - Final Project - Sampling from Random Graphs to Estimate Connectivity in Unreliable Networks

Zachary Lau

December 12, 2024

Python code from this project can also be found on Github at `https://github.com/zach-lau/Graph-Connectivity`

1 Introduction

Erdos-Renyi graphs–also known as $G(n,p)$ graphs–are perhaps the most basic of the family of random graphs. A $G(n,p)$ random graph can be thought of as a graph on n nodes where each edge is independently present with probabliity p. Of particular interest is the "connectivity problem". In particular we want to answer the question: given n and p, what is the probability that a realization of a $G(n,p)$ graph is connected?

This is an interesting statistical problem for a few reasons. First of all, there are the drawbacks of existing analytic methods. Most existing analytic methods are based on graph recursions. The nature of large and dense graphs means that however that as these networks grow in size, the computational performance of these algorithms is poor. We may thus turn to Monte Carlo methods. However, Monte Carlo methods suffer from the transition behaviour of such graphs. In particular, [1] notes that $G(n,p)$ graphs have a very large probability of being conected for $p > \log n/n$ and a very small probability of being connected for $p < \log n/n$. Thus if we are interested in the probability of being connected at a particular p we are generally dealing with very low probabilites or very high probabilities. A naive Monte Carlo method would thus need an exceedingly large number of simulations to make meaningful estimates of the desired integrals.

1.1 Related work

The most relevant work to this problem occurs in the field of reliablity theory. In reliability theory the reliability polynomial $R(G,p)$ of a graph is a polynomial in p that gives the probability that graph is connected given each edge fails independently with probability p. Our problem can be seen as estimating the reliability polynomial at a single point for a complete graph. In general, [7] show that analytic derivation of the reliability polynomial is #-P complete. This class is harder than NP-complete, meaning there are no know polynomial time algorithms for its calculation. While there may be faster algorithms for specific cases such as $G(n,p)$ graphs, these suffer from numeric instability and fail to generalize to broader classes of graphs [4]. Monte Carlo methods for estimating the reliability polynomial general fall into two categories. The first class of methods such as [3] aim to estimate the coefficients of some factorized form of the reliability polynomial. The second class is those who aim to estimate this probability for a specific p such as [5]. However, both of these classes of methods generally focus on graphs with relatively large p. Moreover, they focus on estimating the reliability polynomial via edge cut sets: sets of edges whose removal makes the graph disconnected. In this work we consider alternate approaches such as adding edges or random walk methods.

2 Background

2.1 Notation and definitions

Formally we consider a graph G to be an ordered pair of a set of vertices, denoted $V(G)$ and a set of edges denoted $E(G)$. Each edge consists of an unordered pair of vertices. We denote the edge with vertices v_1 and v_2 by v_1v_2. A path is a sequence of vertices $v_1,\dots,v_n$ such that each edge v_iv_{i+1} is in $E(G)$. A graph is said to be connected if there is a path from every vertex to every other vertex. Node is a common synonym for vertex. Where not specified elsewise, we will denote the set of all connected graphs by A. We use μ to denote the counting measure on the set of graphs, and assume all densities are specified with respect to this measure. S_n denotes the group of permutations on $\{1,2,\dots\dots,n\}$. This is also referred to as "the symmetric group on n elements".

2.2 Importance Sampling

Importance sampling is a technique for reducing the variance of an estimator based on repeated sampling by changing our sampling distribution. Particularly suppose we have $X \sim F$ and we wish to calculate the expected value of some test function $\mathbb{E}_F[h(X)]$ where either F is intractable to sample from, or $h(X)$ has very high variance under F. Further assume F has density f and that there exists another tractable distribution G where we can sample $X \sim G$ with density g. We call G the proposal or majorizing distribution. Let μ denote our underlying measure. Then we can rephrase the problem as

$$\mathbb{E}_F[h(X)] = \int h(x)f(x)d\mu = \int h(x)\frac{f(x)}{g(x)}g(x)d\mu = \mathbb{E}_G[h(x)w(x)]$$

where we denote $w(x) = \frac{f(x)}{g(x)}$. The ideal proposal to reduce the variance of our estimate would have $h(x)w(x)$ be constant. This occurs exactly when $g(x) = \frac{f(x)h(x)}{\int f(x)h(x)d\mu}$. Of course, if we could calculate the denominator we would not need to perform importance sampling, thus the trick is to find a proposal that approximates this distribution, or at least shifts more weight to x values such that $f(x)h(x)$ is relatively high but where the normalization constant is tractable. Other more advanced techniques such as self-normalizing importance sampling or sequential importance sampling also exist, but for reasons of simplicity we restrict our attention to the standard importance sampler.

2.3 Metropolis Hastings and Markov Chain Monte Carlo

Markov Chain Monte Carlo methods are sampling methods that simulate from a given distribution by constructing a Markov chain on the set of desired states such that the ergodic distribution is exactly our desired distribution. One common and flexible method for doing so is Metropolis Hastings. A general algorithm for Metropolis-Hastings Monte Carlo is shown in 5. The key to making this algorithm successful is choosing a proposal which explores the sample space well, while having a high probability of being accepted.

2.4 Graph algorithms

The primary data structures we will use to keep track of graphs are adjacency lists and edge lists. The primary data structures we will use to check for connectedness are union find and DFS. More information about these data structures and algorithms is included in the appendix in section 7.2.

3 Methodology

3.1 Naive simulation

Perhaps the simplest way to estimate the probability of being connected is to simulate every edge individual. Pseudo-code for such an algorithm is presented in algorithm 1 in the appendix. Sample python code is provided in section 7.7.1. Such an algorithm uses $O(n^2p)$ space and $O(n^2)$ time in the average case. A sample implementation is provided in the appendix.

3.2 Reparametrization and analytic variance reduction

Using smarter reparametrization of the problem, we can design a simulation that will yield smaller sampling variance of our restimator and hence smaller standard error. The intuition is that start with an empty graph and just add edges until it becomes connected. Our estimate is then an importance "weight" which reflects how many edges it took. A formalization of the sampling procedure is outlined below

- Let $N = \binom{n}{2}$ be the number of edges and $[n] = \{1, \ldots, N\}$ be a labelling of the edges
- Draw Y, the ordering on edges, uniformly from S_N, the symmetric group on N elements. This represents our edge ordering.
- Draw $Z \sim \text{Binom}(N, p)$, the number of edges
- Let X be the graph consisting of the nodes $\{1, \ldots, n\}$ and the edges in $\cup_{i=1}^{Z} Y(i)$, where $Y(i)$ represents the permutation Y applied to i

We claim that this reparametrization leads to the same distribution as our traditional definition of a random $G(n, p)$ graph. A full proof is shown in the appendix in section 7.3.1. This characterization is useful because we can easily calulate $E[1_A(X)|Y]$. The key insight is that adding edges to a connected graph can never make it disconnected. Thus we only need to simulate enough edges to figure out at what threshold $Z \geq z$ the graph is connected. An algorithm to estimate $P(A)$ by drawing from the random variable $P(A|Y)$ is shown in the in algorithm 2. We claim that the average case runtime for this algorithm is $O(n \log n \alpha(n)) \approx O(n \log n)$. A rough sketch of the argument is provided in the appendix in setion 7.3.1. Finally we show that an estimator based on this simulation will have a lower variance than one based on the naive simulation. In particular we known from the law of total variance that given $h(X) = \mathbf{1}_A(x)$ we have

$$\begin{aligned} Var(h(X)) &= Var(E[h(X)|Y]) + E[Var(h(X)|Y)] \\ Var(E[h(X)|Y]) &\leq Var(h(X)) \\ Var(P(A|Y)) &\leq Var(h(X)) \end{aligned}$$

We will leave up to experimentation the question of exactly how much better this sampler is. A sample implementation is provided in section 7.7.2

3.3 MCMC

The final method we propose is an importance sampling based MCMC method. The intuition for this scheme is that if we naively simulate random $G(n, p)$ graphs, then the mass will be concentrated on small graphs which are very unlikely to be connected. We instead want to spread some of this mass out to larger graphs which are more likely to be connected. In particular, we focus on importance samplers where we can express the density as a function of the number of edges i.e. $g(x) = g(|E(x)|)$ (we overload the g label here). We do so because the number of edges is well behaved so in general we can calculate the normalizing constants required without resorting to self-normalized importance sampling, which would introduce additional variance in the normalizing constant. We suggest the following majorizing distribution which places all of its weight on graphs with enough edges to be connected but not so many that they are uninteresting. Specifically given

- n: the number of nodes
- $N = \binom{n}{2}$: the number of potential edges
- $a = n$: the lower bound
- $b = \lceil \frac{N \log n}{n} \rceil$: the upper bound
- $e = |E(x)|$: the number of edges

The probability mass function of our majorizing distribution is

$$g(x) = \frac{1}{\binom{N}{e}(b-a+1)}\mathbf{1}_{[a,b]}(e)$$

Since there are exactly $\binom{N}{e}$ graphs with e edges the marginal distribution of $E(X)$ is uniform over its support. The motivation for such a majorizing distribution is that the ideal majorizing distribution that is a function of E is given by $g(e) = P(E(X) = e)P(A|E(X) = x)$. Once we increase e beyond $N \log n/n$ the second factor is very close to one but for small p th first factor decays rapidly. Thus we want to spread the weight of our majorizing distribution somewhere between these two edge points. While our proposal does not have full support, empirically the regions it omits have very little mass. This majorizing distribution gives importance weights

$$w(x) = (b-a+1)\binom{N}{e}p^e(1-p)^{N-e}$$

We propose to sample from this distribution using MCMC methods. The motivation is that by only changing a small number of edges at time we can reduce the computational burden of regenerating the entire graph. We use as our MCMC move swapping a random number of edges. The algorithm shown in algoritihm 3 implements a single MCMC update. Combining this update step with our sampler we get algorithm 4. We expect that in general computing the importance weights could be a challenge due to the large combinatorials involved. However, for our specific choice of majorizing distribution we can express the importance weights in terms of the binomial probability mass function (PMF). Denoting by $f_{(n,p)}$ the PMF of a binomial random variable with parameters n and p, we get

$$w(x) = (b-a+1)f_{(n,p)}(e)$$

We will use the binomial PMF implementation in scipy to carry out this computation.

3.4 Experiments

3.4.1 Factors

Our experimental analysis of these algorithms will focus on two factors. Firstly is of course the algorithm of choice. The naive sampler and the variance reduction sampler do not require further implementation details. For the MCMC sampler we need to choose m the scale factor for the size of the steps. We will test $m \in \{1, \sqrt{n}, n, n \log n/4\}$ In total this gives us 6 samplers to try. The second factor is the size n of the graphs. For our purposes we run our simulations with $n \in \{10, 100, 1000\}$. These values allow us to cover a wide variety of magnitudes, while also pushing the computational limits of the algorithms. This table also provides the theoretical answers in the small cases where direct computaition is feasible.

The factors which which will not change independently include the number of samples to draw and p the probability of an edge being present. We will fix the number of samples to draw at 10^5. Taking such a large number of samples is important to get an accurate assessment of the variance of our estimators. For the naive and analytic sampler, the accuracy of the reported standard error as an estimate of the sampling standard deviation is dependent directly on the accuracy of the underlying estimate. For the MCMC samplers, we require the chain to reach stationarity for our ESS estiamtes to be reliable.

The choice of p must also be made with some thought. While we want $p < \log n/n$ so that we are in the rare event scenario, if we make p too small then the actual answer may be smaller then the machine epsilons which. A more elucidating value of p is one just below the threshold. Based on experiments with the analytic reduction sampler we will choose values such that $p \approx 0.001$. This will allow our naive sampler to at least have *some* hits but more importantly

ensure that our MCMC sampler is able to make some progress. Table 1 lists our candidate p-values for each n. The code used to generate these values by binary search is found in the appendix section 7.7.7. Future work could involve changing the value of p to see how our results change.

In addition to the timing analysis we list above, we will perform an exploratory analysis of the MCMC mixing using an instrumented version of the MCMC sampler. We will keep track of the number of edges, number of connected componets, and importance sampling weights.

3.4.2 Evaluation

We will record two primary metrics. The first metric is standard error of our estimate. Since the first two samplers are based on sample means, we apply the central limit theorem to approximate their standard error by $se = \frac{\hat{\sigma}}{\sqrt{n}}$ and $\hat{\sigma} = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x-\bar{x})^2}$. For the MCMC sample we need to apply the Markov Chain central limit theorem. In particular we will estimate the MCSE using the effective sample size, similar to what is done in Stan, a popular MCMC framework[6]. For computational reasons we will use arviz to actually calculate the ESS. It is important to note that in the case of severe underestimates of the event probability, the standard error may also be severely underestimated. This can occur for samplers which miss a few heavily weighted graphs in the sample space. The question of how to identify these underestimates in practice is an interesting one which could use further discussion. Ihe case of small n, we can calculate a theoretical answer. For large n, since we have other samplers to compare against, we will qualitatively identify underestimates based on their order of magnitude.

The second metric is computation time. All experiments will be implemented in python and timed using the in-built python timer to time user-time (as opposed to system or elapsed). Finally we will consider the compound metric given by process time per unit precision. This is ultimately what an end user will be concerned with. Lower values of this metric generally indicate a better sampler. However, as mentioned previously, we have to watch out for severe underestimates.

4 Results

4.1 Computer time

The results of the performance testing are shown in table 2 in the appendix. The order of tests was randomized to prevent warm-start effects. The order that results are listed in the table is the order the tests were run. A summarized and sorted version of the data is presented in tables 3, 4 and 5. All tests were performed on a 2021 Macbook air with M1 core. Time measurements were performed using Python's "time.process_time" function. This measures CPU time and not clock time. In particular, it does not include things like I/O and adds up compute time over all threads running the workload, even if they may be running in parallel. Since the sampling programs are not I/O bound, are we are not considering thread-level parallelism at this time, we should expect process time to be an accurate measure of performance. Furthermore, its use will decrease measurement noise caused by unrelated programs running on the system.

The time results are graphed as a function of n on a log-log scale in figure 1. The asymptotic behaviour is what we would expect. In particular the expected runtime for the naive sampler is $O(n^2)$. On the log-log plot its trend line should have a slope of 2, which is visually close to that observed. The other samplers have expected asymptotic runtime $O(n \log n)$. This would manifest on the log-log plot as a slope slightly greater than 1, reducing to 1 as $n \to \infty$. While the slope of these samplers is generally close to 1, we actually observe a slight slope upwards. This could be explained by higher startup costs such as loading the required modules or generating initial starting graphs. Vertical displacement on the log-log plot indicates constant multiplicative factors affecting the run-time. In particular, we see that the analytic sampler has a high constant multiplier which renders it slower than the other samplers unless n is large.

4.2 Accuracy

Figure 2 show the reported standard error for each n. Again, we use a log-log plot with quadratic interpolation. For the MCMC samplers, the standard error was computed as $\frac{\hat{\sigma}}{\sqrt{ESS}}$. See [6] for more information. For the MH1 sampler (MCMC with step-size 1), we did not find any connected graphs at all for $n = 1000$, and hence there is no reported standard error. As expected, the naive sampler has relatively constant standard error. Recall that we have tuned p such that the answer should be around 0.001. In this case we expect the naive standard error based on $1e5$ samples to be near $1e-4$, which we observe. Furthermore as expected, the analytic sampler has lower standard error. However, the gap narrows as n increases. The MCMC samplers actually have relatively good standard error for small n. However, this worsens as n increases.

4.3 Compound metrics

We combine the process time and standard error to produce several compound metrics. Most importantly, we have our original intended metric given by the product of CPU time and squared standard error. The values are shown in table 8 and graphed in figure 8. This holistic measure indicates the amount of compute time needed per unit precision. Based on this metric we find that the analytic sampler actually performs the best. When n is low, its lower variance improves its performance, whereas when n is high its relatively fast runtime also helps to ensure its lead. Unfortunately the MCMC samplers perform relatively poorly on this combined metric. While they are faster, the sampling variance of the weights is too high. To illustrate this point we graph the estimated sampling variance in figure 5. Assuming our ESS estimate is good, this high variance cannot be explained by poor mixing. We conjecture that the importance sampling distribution is to blame, and a better majorizing distribution could reduce the variance. Indeed, the variance is even higher than the naive sampler for both $n = 100$ and $n = 1000$. In this case we would be better off just sampling directly from the original distribution. Other compound metrics are calculated and provided in tables 6 and 7 and figures 6 and 7.

4.4 MCMC mixing and convergence

Finally we consider mixing for our MCMC samplers. We use an instrumented version of the original MCMC sampler with code shown in section 7.7.5. The mixing is run for $100,000$ iterations at $n = 100$ with the same value of p shown in table 1. The same random seed is used as in the performance experiments. We analyze mixing of three test functions: the number of connected components, the number of edges, and the importance sampling weights. The autocorrelation function for each test function is plotted and shown in figures 9,10 and 11. As expected, we see relatively high levels of autocorrelation for the smaller scale MCMC samplers. Interestingly we do see high correlation in our importance sampling weights for the scale 1 sampler, but not for any of the larger samplers. We can also visually assess the mixing using the traceplots shown in figures 12, 13 and 14. Perhaps of most interest is the traceplot for the importance sampling weights in figure 12. We can clearly see that most of the weight is contributed by a few samples, which contributes to the higher variance. We can also see very high autocorrelation for the step-size 1 sampler. The traceplots for edges and connected components show poor mixing for smaller step sizes, and good mixing for larger step sizes, as expected.

We also examine convergence of the sample means for each test function in figures 16, 14 and 12. Of particular importance is convergence in the sample mean of the importance sampling weights in figure 15, since this is what we are using as our estimate of connectivity probability. In this figure we see a clear "saw" like pattern. This is to be expected when much of the weight comes from few values and indicates. For larger step sizes we see moderate indication of convergence, which could be quantified in the future with a R-hat calculation. For the number of edges and number of connected components, we see from figures 16 and 17 very strong indicators of convergence at higher stpe sizes. Notably the connectivity is a function of the number of connected components. However this is not enough to conclude that the mean of the importance sampled weights should also show good convergence.

5 Dicussion

5.1 Limitations

The first computational hurdle faced by this project was the computational cost of evaluating connectedness. In general for our graphs, this step had asymptotic complexity $O(n \log n)$. The cost of this step would thus dominate the cost of actually performing the update for any step size smaller than $O(n \log n)$. We can observe this asymptotic behaviour in figure 1. The slopes of the plots for each MCMC sampler are roughly similar, indicate they have the same asymptotic performance. However, if evaluating the connectivity could be made faster, then asymptotic gains could potentially be realized for smaller step sizes. Unfortunately, this is not easy. The problem of evaluating connectivity in a graph under modifications is known as the "dynamic connectivitiy" problem. Under edge addition, a union find datastructure can track connectivity in approximately constant time per insertion. However, it is much harder under edge deletion. Candidates include link-cut trees and euler-tour trees, which can do so in $O(\log n)$ per modification and query.

The second major barrier was finding an appropriate majorizing distribution to actually apply importance sampling. While we may have increased the number of non-zero weights as shown in figure 4, for $n = 100$ and $n = 1000$, the overall variance of the importacne sampling weights under our majorizing distribution is actually higher. Thus while this solution may be useful for reducing the probability of no events of interest being observed, it is not actually useful for reducing the final variance of our sampler. Better strategies might take advantage of the topological properties such as is done in [3].

The final limitation was how to evaluate the standard error of the estimators. We took a brute force approach in our case by taking a large enough number of samples such that the standard deviation of the estimator could be estimated reasonably. However, this is not always possible. Indeed, having to take such a large number of samples hindered our ability to explore the behaviour of our sampels for larger n values. Improvements in this area might include using Bayesian inspired estiamtors to find more stable confidence intervals.

5.2 Future work

We identify two major potential areas of future work. Firstly, there is still work that can be done to try to optimize the implementation of the algorithms we have already presented. For the naive sampler, a simple optimization would be to generate the number of edges from a binomial distribution first, and then generate just the required number of edges. This would improve average runtime to $O(n \log n)$, making it comparable with the analytic sampler. Furtheremore, for each sampler we could use profiling to identify the computational bottlenecks. A possible improvement for the analytic and MCMC samplers is to avoid creating a new union find object every time we want to check connectivity. A full profiling run would help to identify other possible candidates for improvement. Finally, these algorithms could be implemented in a more performant language like C++ or Julia. This would allow us to potentially explore larger n values. Another area of future work is expanding the analysis of the MCMC chains. In these experiments, we used a single MCMC chain per step size, and only visual inspection to identify convergence. A more sophisticated approach would use multiple chains in parallel. Convergence could be estimated with a quantitative measure such as $\hat{R}$. Finally we propose expanding our testing parameters. One additional types of monte carlo schemes we could test is quasi-monte carlo. It would also be useful to compare the performance of the samplers as p varies, as well as pushing n as high as possible.

6 Conclusion

While MCMC methods are a powerful tool to sample from otherwise intractable distributions, they may not always be the best choice. In particular, when dealing with graph random variables, the evaluation of the test function may be a computational constraint. In this case efficient sampling is critical. Furthermore, in graph spaces, finding appropriate majorizing distributions for importance sampling can be difficult. While importance sampling schemes may increase the

likelihood of observing an event, if poorly chosen they may nevertheless increase the sampling variance. Where possible, analytic techniques to reduce variance produce favourable outcomes compared to basic importance sampling schemes.

References

[1] Bela Bollobas. *Graph Theory An Introductory Course*. Springer, 1979.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms, fourth edition*. MIT Press, 2022.

[3] David G. Harris, Francis Sullivan, and Isabel Beichl. Fast sequential importance sampling to estimate the graph reliability polynomial. *Algorithmica*, 68(4):916–939, Apr 2014.

[4] Wakaka (https://math.stackexchange.com/users/112007/wakaka). Exact probability of random graph being connected. Mathematics Stack Exchange. URL:https://math.stackexchange.com/q/584228 (version: 2013-11-28).

[5] David R Karger. A randomized fully polynomial time approximation scheme for the all terminal network reliability problem. In *Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing*, pages 11–17, 1995.

[6] Stan Reference Manual. Posterior analysis. `https://mc-stan.org/docs/reference-manual/analysis.html#effective-sample-size.section`. [Online; accessed 2024-11-14].

[7] J. Scott Provan and Michael O. Ball. The complexity of counting cuts and of computing the probability that a graph is connected. *SIAM Journal on Computing*, 12(4):777–788, 1983.

7 Appendix

7.1 Sampling algorithms

7.1.1 Naive sampler

The naive sampling algorithm is show in algorithm 1

Algorithm 1 Naive simulation

```
procedure NAIVE(n,p)
    Set g to an empty graph
    for Each pair of vertices (v_1, v_2) in g do
        Draw U from Unif(0,1)
        if U < p then
            Add v_1 v_2 to g
        end if
    end for
    return g.connected()
end procedure
```

7.1.2 Analytic variance reduction

The sample with analytic reduction is shown in algorithm 2.

Algorithm 2 Sampling with Analytic Variance Reduction

```
procedure ANALYTICREDUCTIONSAMPLER(n,p)
    i ← 0
    u ← newUnionFind(n)
    while not u.connected do
        Pick two random vertices v_1, v_2
        if v_1 v_2 ∉ g then
            u.union(i,j)
            i ← i + 1
        end if
    end while
    return binom.survFunc(n,p,i-1)
end procedure
```

7.1.3 MCMC Importance sampling

The MCMC based importance sampling algorithm is shown in algorithm 4. The update is shown in algorithm 3.

Algorithm 3 MCMC update

procedure UPDATE(m, x)
  Draw δ from $N(0, m^2)$ and round to nearest int
  if E(x)+$\delta \geq b$ **then** Reject
  end if
  if E(x)+$\delta < a$ **then** Reject
  end if
  if $\delta > 0$ **then**
    Add δ random edges to x
  else
    Remove $-\delta$ random edges from x
  end if
end procedure

Algorithm 4 MCMC sampling from random graphs

procedure MCMCSAMPLE(n,p)
  Initialize s to empty list
  Initialize x to a valid graph
  for $i \in 1 \dots B$ **do**
    Update(x)
    s.append(w(x))
  end for
end procedure

7.2 Graph algorithms

7.2.1 Graph representations

Since graphs are complex data structures, some thought must be put into their representation in memory. We will primarily make use of two methods of representing graphs in memory: edge lists and adjacency lists. An edge list stores a graph as a list of pairs of nodes where each node is represented by a label. For example, the following python literal would represent the complete graph K_3 on the nodes $0, 1, 2$

```
my_edge_list = [(0,1),(0,2),(1,2)]
```

Similarly an adjacency list stores a list of neighbours for each vertex. The following python literal represents K_3 as an adjacency list

```
my_adjacency_list = [[1,2],[0,2],[0,1]]
```

Edges can be easily added and removed by modifying the underlying data structures. The adjacency list has the advantage has the advantage that given a node we can easily find its neighbours. This is useful for traversal algorithms such as depth first search and breadth first search. For our purposes, the edge list has the useful property that we can easily pick a random edge uniformly at random. This property will be useful for edge swap moves in our MCMC simulation.

7.2.2 Union find

A union find data structure–also known as a disjoint set data structure–is type of data structure that dynamically tracks the connected components of a graph under edge addition. It does so associating a tree with each connected component. The structure of this tree is stored by associating with each node a parent. The basic operations supported by union find data structure are

- `find(i)`: find the root of the tree containing node i
- `union(i,j)`: join the connected components containing node i and j

Basic pseudocode for each algorithm is presented in the appendix algoritihms 6 and 7. Additionally we will equip our union find datastructure with a `getComponents` operation to find the number of connected components. Notably if there is a single connected comonent then the graph is connected.

Building and using a union find datastructure has worst-case running time $O(m\alpha(n))$ where m is the number of union/find operations, n is the number of nodes and α is the inverse ackerman function. For all practical purposes $\alpha(n)$ can be seen as constant [2]. Our implementation of the union find datastructure uses $O(n)$ space, where n is the number of nodes.

7.2.3 Depth First Search (DFS) and Breadth First Search (BFS)

Depth first search and breadth first search are two algorithms for traversing a connected component of a graph. In particular, if every node is in the same connected component then we can know that a graph is connected. In the appendix algorithm 8 we present an algorithm to check if a graph on n nodes reprsented as an adjacency list is connected using a DFS.

7.3 Proofs

7.3.1 Analytic reduction sampler

First we show that the random variable described as the same distibution a $G(n,p)$ random graph. In particular, we have that the probability of any edge being present is

$$\begin{aligned} P(e \in E(X)) &= P(Y^{-1}e \leq Z) \\ &= \sum_{z=1}^{N} \frac{z}{N} P(Z = z) \\ &= \frac{1}{N}\mathbb{E}[Z] \\ &= p \end{aligned}$$

Furthermore, for any set of m edges E we can demonstrate independence by find the probability they are all included in the graph using a combinatorial approach, leveraging the fact that Y is uniformly distributed on S_N.

$$\begin{aligned}
P(E \subseteq E(G)) &= \sum_{z=1}^{N} P(Z = z) P(E \in E(G) \mid Z = z) \\
&= \sum_{z=m}^{N} \binom{N}{z} p^z (1-p)^{N-z} \frac{1}{N!} \frac{z!}{(z-m)!} (N-m)! \\
&= p^m \sum_{z-m=0}^{N-m} \binom{N-m}{z-m} p^{z-m} (1-p)^{(N-m)-(z-m)} \\
&= p^m
\end{aligned}$$

And thus the events that any set of distinct edges are included in the graph is independnet as promised.

Next we provide a rough sketch of why the runtime of the analytic reduction sampler has runtime approximately $O(n \log n)$

- We know that most graphs are connected for $p \approx \log n / n$
- Therefore most graphs with $\binom{n}{2} \times \log n / n = O(n \log n)$ edges should be connected
- We also expect that the graph will become connected before we have to worry about resampling the same edge twice
- Thus we expect our loop to run $O(n \log n)$ times
- Each loop has a a single union so by results on union-find datastructures our overall runtime should be $O(n \log n \alpha(n))$

7.3.2 Metropolis hastings sampler

A generic metropols hastings based MCMC sampler is shown in algorithm 5

Algorithm 5 Metropolis Hastings MCMC

procedure MH
  s $\leftarrow$ empty list
  for i in $1 \dots B$ **do**
    Propose $x^{(i+1)} \leftarrow q(x^{(i)}, x^{(i+1)})$
    $\rho \leftarrow \frac{p(x^{(i+1)})}{p(x^{(i)})} \frac{q(x^{(i+1)}, x^{(i)})}{q(x^{(i)}), q(x^{(i+1)})}$
    if $\rho > 1$ **then**
      $x^{(i+1)} \leftarrow x^{(i)}$
    else
      $x^{(i+1)} \leftarrow x^{(i)}$ with probability ρ
    end if
    s.append($x^{(i)}$)
  end for
end procedure

7.4 Additional algorithms

7.4.1 Union find

The find algorithm is shown in algorithm 6. It includes an optimization called "path compression" which flattens the tree to make future calls to `find` faster. The `union` algorithm is shown in algorithm 7. It includes an optimization called "union by size" which makes sure the smaller tree attachs to the larger one.

Algorithm 6 Find operation

```
procedure FIND(i)
    if parent[i] == i then return i
    else
        parent[i] ← Find(parent[i])
        return parent[i]
    end if
end procedure
```

Algorithm 7 Union operation

```
procedure UNION(i,j)
    a ← parent[i]
    b ← parent[j]
    if a == b then return
    end if
    components ← components - 1
    if size[a] > size[b] then
        parent[b] ← a
        size[a] ← size[a]+size[b]
    else
        parent[a] ← b
        size[b] ← size[a]+size[b]
    end if
end procedure
```

7.4.2 DFS

Algorithm 8 shows how a depth-first-search can be used to check if a graph is connected.

Algorithm 8 DFS for graph connectivity

```
procedure CONNECTED(g)
    visited = [False]*n
    visited[0] ← True
    s ← newStack()
    s.push(0)
    to_visit ← n − 1
    while to_visit > 0 and not s.empty() do
        n ← s.pop()
        to_visit ← to_visit - 1
        for other in n.neighbours() do
            if not visited[other] then
                visited[other] ← True
                s.push(other)
            end if
        end for
    end while
    return to_visit == 0
end procedure
```

7.5 Evaluation

Table 1 lists the values of p used for each n for testing purposes. The theoretical values were calculated using the code in section 7.7.8 using the formula from [4].

Table 1: Test p values for various n

n	p	Theoretical probability of connectivity
10	0.07544	0.001005
100	0.02732	0.001255
1000	0.005098	NA
10000	0.0007382	NA

7.6 Results

7.6.1 Performance results

All tests were run sequentially in a single process. Their appearance in table 2 reflects the order they were run. A random order was used to prevent "warm starts". The code used for testing can be found in appendix section 7.7.4.

Table 2: Performance testing

Function	n	p	b	mu	se	hits	ess	processTime
MHRN	1,000	$5.1 \cdot 10^{-3}$	$1 \cdot 10^{5}$	$1.26 \cdot 10^{-3}$	$6.7 \cdot 10^{-3}$	4,294	358.95	97.79
MHNLN	10	$7.54 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$9.95 \cdot 10^{-4}$	$5.05 \cdot 10^{-5}$	36,096	3,334.62	1.01
Naive	10	$7.54 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.04 \cdot 10^{-3}$	$1.02 \cdot 10^{-4}$	104	$1 \cdot 10^{5}$	0.27
MHNLN	1,000	$5.1 \cdot 10^{-3}$	$1 \cdot 10^{5}$	$3.41 \cdot 10^{-3}$	$1.44 \cdot 10^{-3}$	4,573	22,819.87	73.85
MH1	100	$2.73 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.88 \cdot 10^{-4}$	$4.7 \cdot 10^{-4}$	9,261	319.53	3.88
Analytic	100	$2.73 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.23 \cdot 10^{-3}$	$5.08 \cdot 10^{-5}$	$1 \cdot 10^{5}$	$1 \cdot 10^{5}$	28.25
MH1	1,000	$5.1 \cdot 10^{-3}$	$1 \cdot 10^{5}$	0	0	0	$1 \cdot 10^{5}$	47.76
MHN	1,000	$5.1 \cdot 10^{-3}$	$1 \cdot 10^{5}$	$2.83 \cdot 10^{-3}$	$1.26 \cdot 10^{-3}$	4,725	25,341.62	97.03
MHRN	10	$7.54 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.01 \cdot 10^{-3}$	$3.92 \cdot 10^{-5}$	35,674	5,716.33	1.07
Analytic	10	$7.54 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$9.99 \cdot 10^{-4}$	$5.68 \cdot 10^{-6}$	$1 \cdot 10^{5}$	$1 \cdot 10^{5}$	4.25
MH1	10	$7.54 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.04 \cdot 10^{-3}$	$3.8 \cdot 10^{-5}$	36,005	6,333.76	1.13
Naive	100	$2.73 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.26 \cdot 10^{-3}$	$1.12 \cdot 10^{-4}$	126	$1 \cdot 10^{5}$	16.18
MHN	100	$2.73 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.06 \cdot 10^{-3}$	$2.98 \cdot 10^{-4}$	8,888	15,536.56	3.97
MHRN	100	$2.73 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.79 \cdot 10^{-3}$	$7.31 \cdot 10^{-4}$	9,446	5,418.94	5.99
MHNLN	100	$2.73 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$1.21 \cdot 10^{-3}$	$4.15 \cdot 10^{-4}$	9,073	14,048	3.63
Naive	1,000	$5.1 \cdot 10^{-3}$	$1 \cdot 10^{5}$	$2.26 \cdot 10^{-3}$	$1.5 \cdot 10^{-4}$	226	$1 \cdot 10^{5}$	1,781.48
Analytic	1,000	$5.1 \cdot 10^{-3}$	$1 \cdot 10^{5}$	$2.06 \cdot 10^{-3}$	$1.16 \cdot 10^{-4}$	$1 \cdot 10^{5}$	$1 \cdot 10^{5}$	481.51
MHN	10	$7.54 \cdot 10^{-2}$	$1 \cdot 10^{5}$	$9.18 \cdot 10^{-4}$	$6.18 \cdot 10^{-5}$	33,893	2,131.47	0.9

Table 3: Mu estimates

Function	10	100	1000
MHNLN	$9.95 \cdot 10^{-4}$	$1.21 \cdot 10^{-3}$	$3.41 \cdot 10^{-3}$
Naive	$1.04 \cdot 10^{-3}$	$1.26 \cdot 10^{-3}$	$2.26 \cdot 10^{-3}$
MHRN	$1.01 \cdot 10^{-3}$	$1.79 \cdot 10^{-3}$	$1.26 \cdot 10^{-3}$
Analytic	$9.99 \cdot 10^{-4}$	$1.23 \cdot 10^{-3}$	$2.06 \cdot 10^{-3}$
MH1	$1.04 \cdot 10^{-3}$	$1.88 \cdot 10^{-4}$	0
MHN	$9.18 \cdot 10^{-4}$	$1.06 \cdot 10^{-3}$	$2.83 \cdot 10^{-3}$

Table 4: Standard errors

Function	10	100	1000
MHNLN	$5.05 \cdot 10^{-5}$	$4.15 \cdot 10^{-4}$	$1.44 \cdot 10^{-3}$
Naive	$1.02 \cdot 10^{-4}$	$1.12 \cdot 10^{-4}$	$1.5 \cdot 10^{-4}$
MHRN	$3.92 \cdot 10^{-5}$	$7.31 \cdot 10^{-4}$	$6.7 \cdot 10^{-3}$
Analytic	$5.68 \cdot 10^{-6}$	$5.08 \cdot 10^{-5}$	$1.16 \cdot 10^{-4}$
MH1	$3.8 \cdot 10^{-5}$	$4.7 \cdot 10^{-4}$	0
MHN	$6.18 \cdot 10^{-5}$	$2.98 \cdot 10^{-4}$	$1.26 \cdot 10^{-3}$

Table 5: CPU Time (s)

Function	10	100	1000
MHNLN	1.01	3.63	73.85
Naive	0.27	16.18	1,781.48
MHRN	1.07	5.99	97.79
Analytic	4.25	28.25	481.51
MH1	1.13	3.88	47.76
MHN	0.9	3.97	97.03

Table 6: CPU time per ESS (s)

Function	10	100	1000
MHNLN	$3.01 \cdot 10^{-4}$	$2.59 \cdot 10^{-4}$	$3.24 \cdot 10^{-3}$
Naive	$2.71 \cdot 10^{-6}$	$1.62 \cdot 10^{-4}$	$1.78 \cdot 10^{-2}$
MHRN	$1.87 \cdot 10^{-4}$	$1.1 \cdot 10^{-3}$	0.27
Analytic	$4.25 \cdot 10^{-5}$	$2.82 \cdot 10^{-4}$	$4.82 \cdot 10^{-3}$
MH1	$1.78 \cdot 10^{-4}$	$1.21 \cdot 10^{-2}$	$4.78 \cdot 10^{-4}$
MHN	$4.24 \cdot 10^{-4}$	$2.55 \cdot 10^{-4}$	$3.83 \cdot 10^{-3}$

Table 7: CPU Time per non-zero weight (s)

Function	10	100	1000
MHNLN	1.01	3.63	73.85
Naive	0.27	16.18	1,781.48
MHRN	1.07	5.99	97.79
Analytic	4.25	28.25	481.51
MH1	1.13	3.88	47.76
MHN	0.9	3.97	97.03

Table 8: Product of CPU Time and squared standard error (s)

Function	10	100	1000
MHNLN	1.01	3.63	73.85
Naive	0.27	16.18	1,781.48
MHRN	1.07	5.99	97.79
Analytic	4.25	28.25	481.51
MH1	1.13	3.88	47.76
MHN	0.9	3.97	97.03

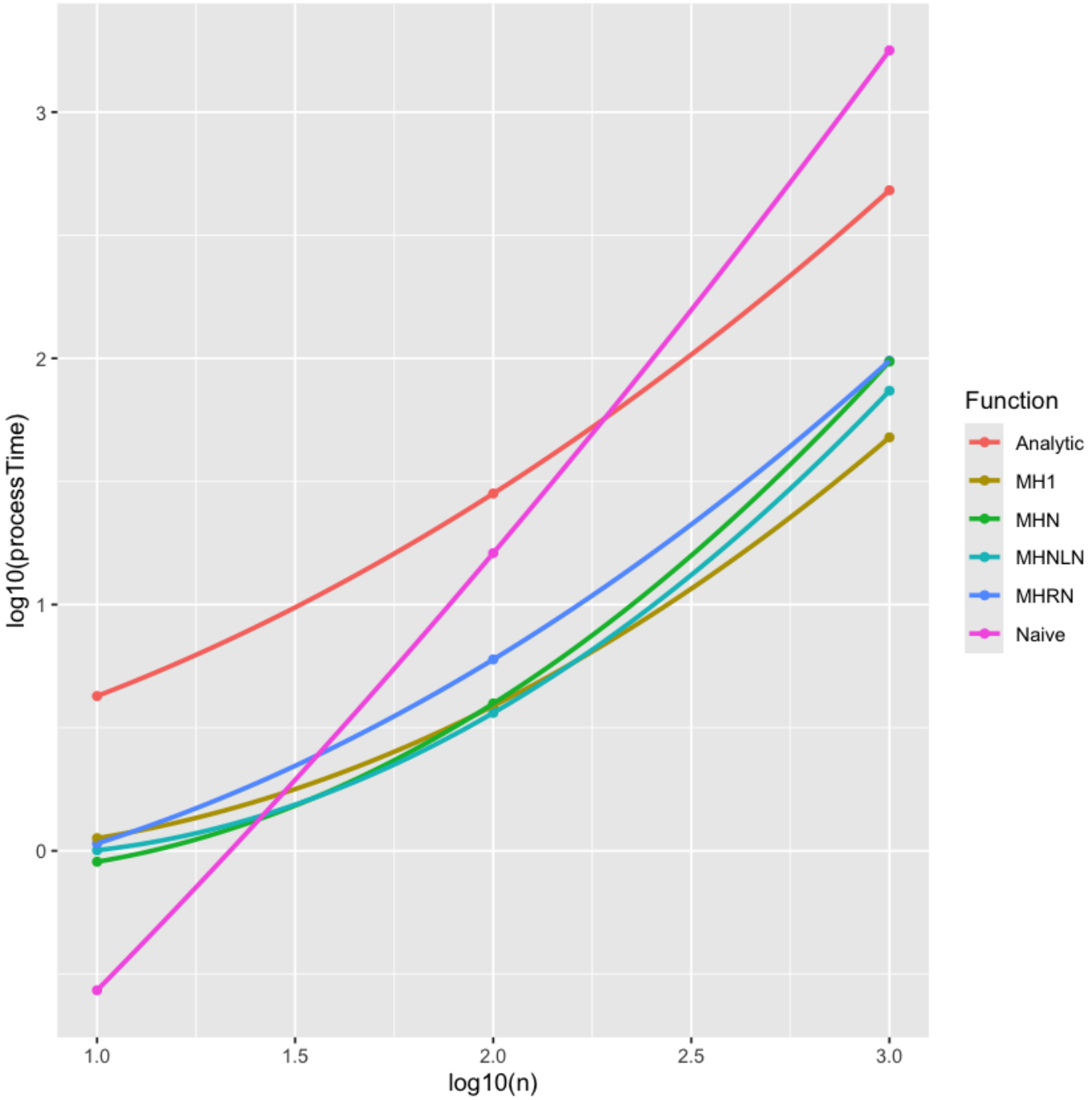

Figure 1: Process time for 100,000 samples of random graphs using different samplers. The trend-line is based on centered quadratic interpolation on the log-log scale. Process time is measured in seconds.

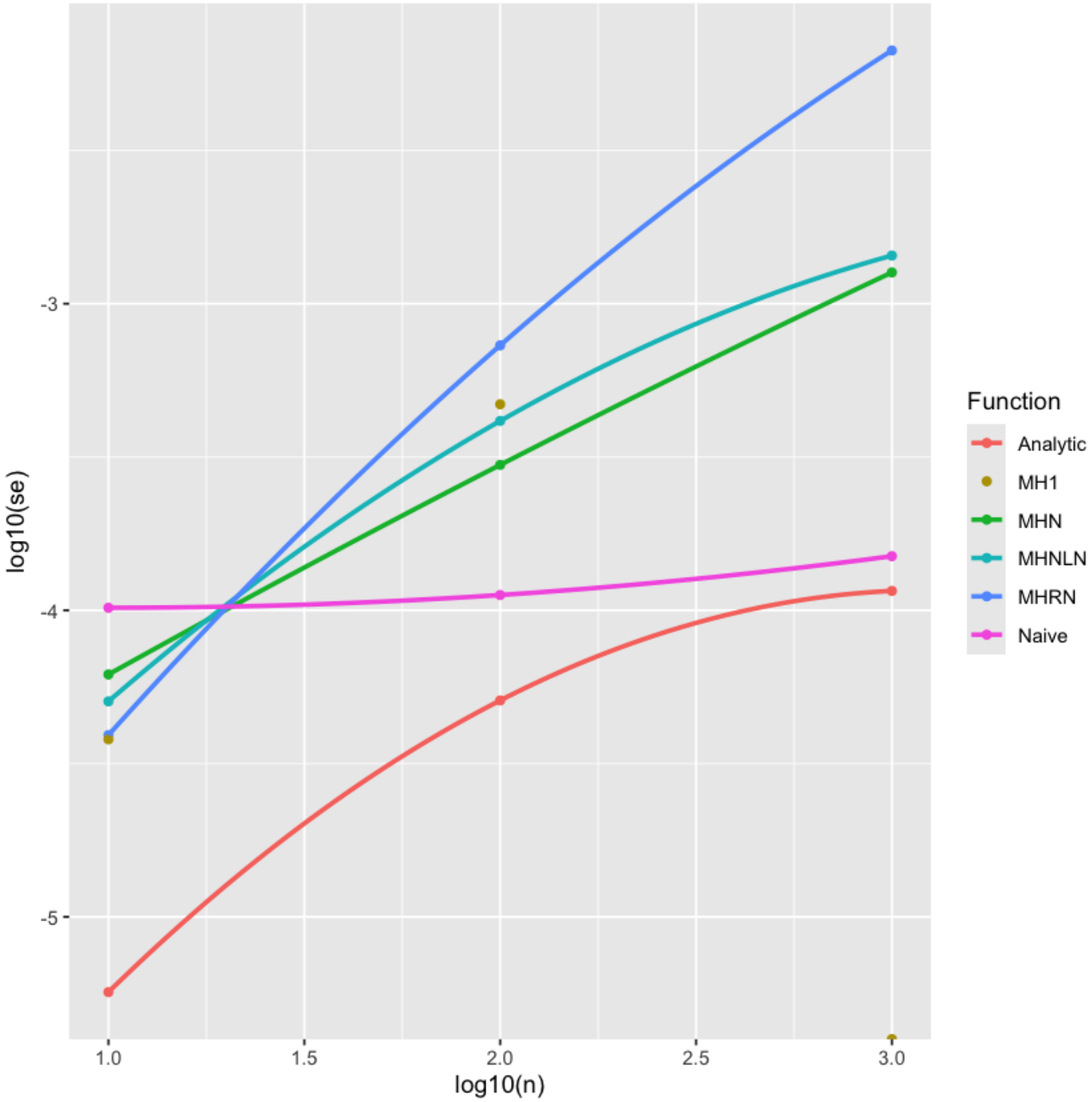

Figure 2: Standard error of estimate of probability of connectivity in $G(n, p)$ random graph

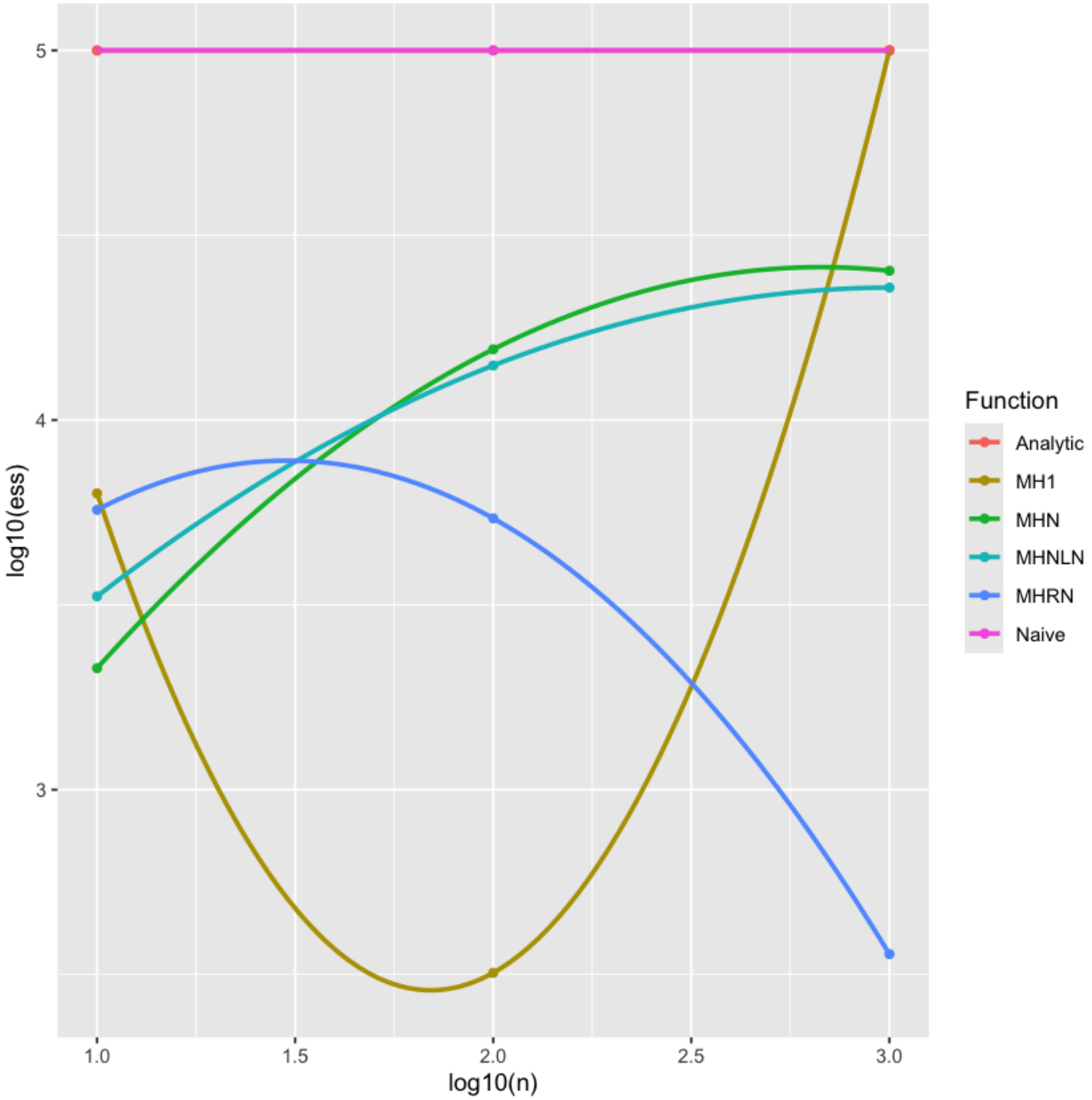

Figure 3: Effective sample size (ESS) for MCMC samplers for connectivity from graphs

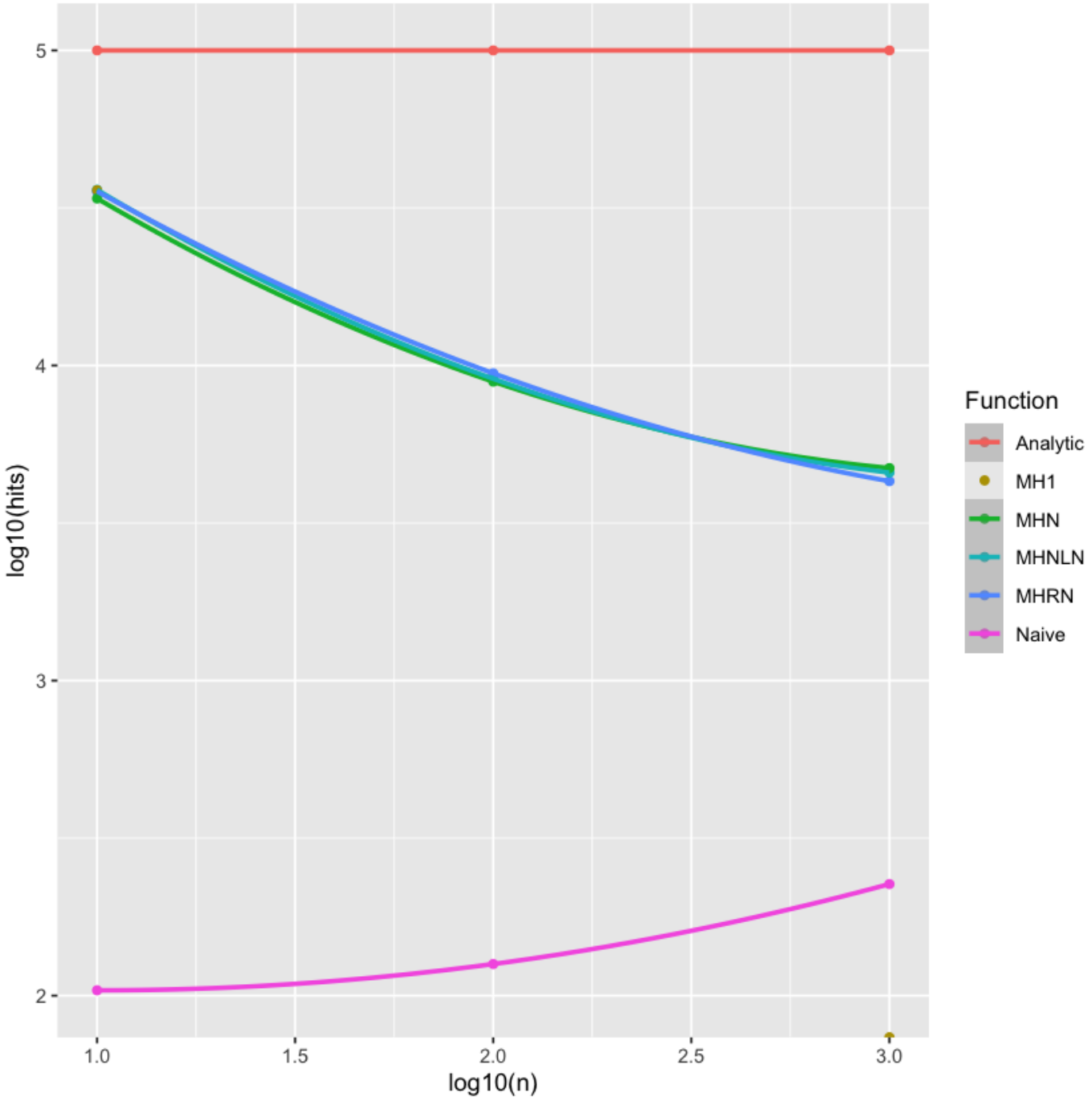

Figure 4: Number of non-zero weights out of 100,000 samples from random graphs. Non-zero weights correspond to connected graphs found.

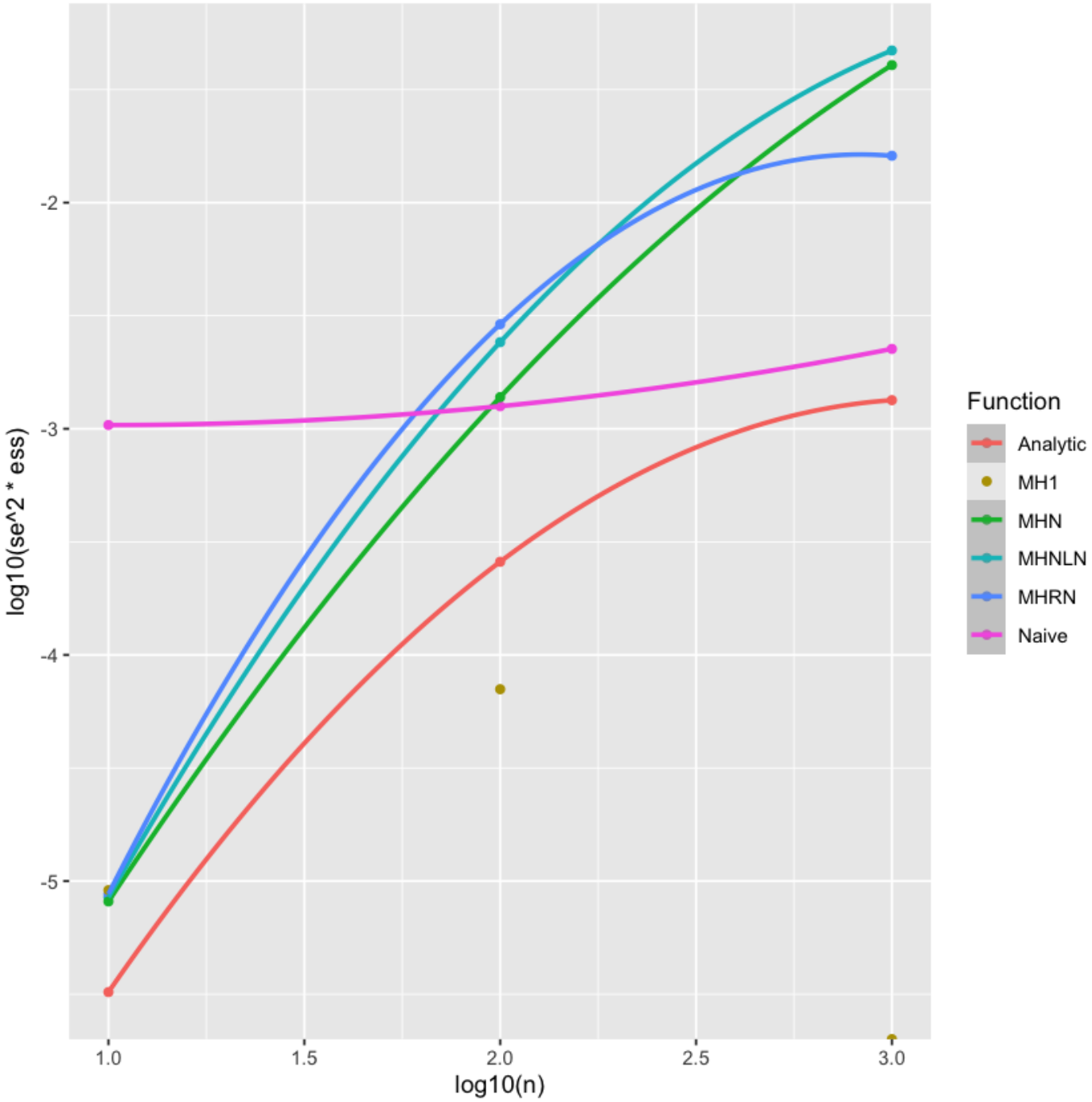

Figure 5: Estimated sampling variance of weights for samplers for connectivity in $G(n, p)$ random graphs

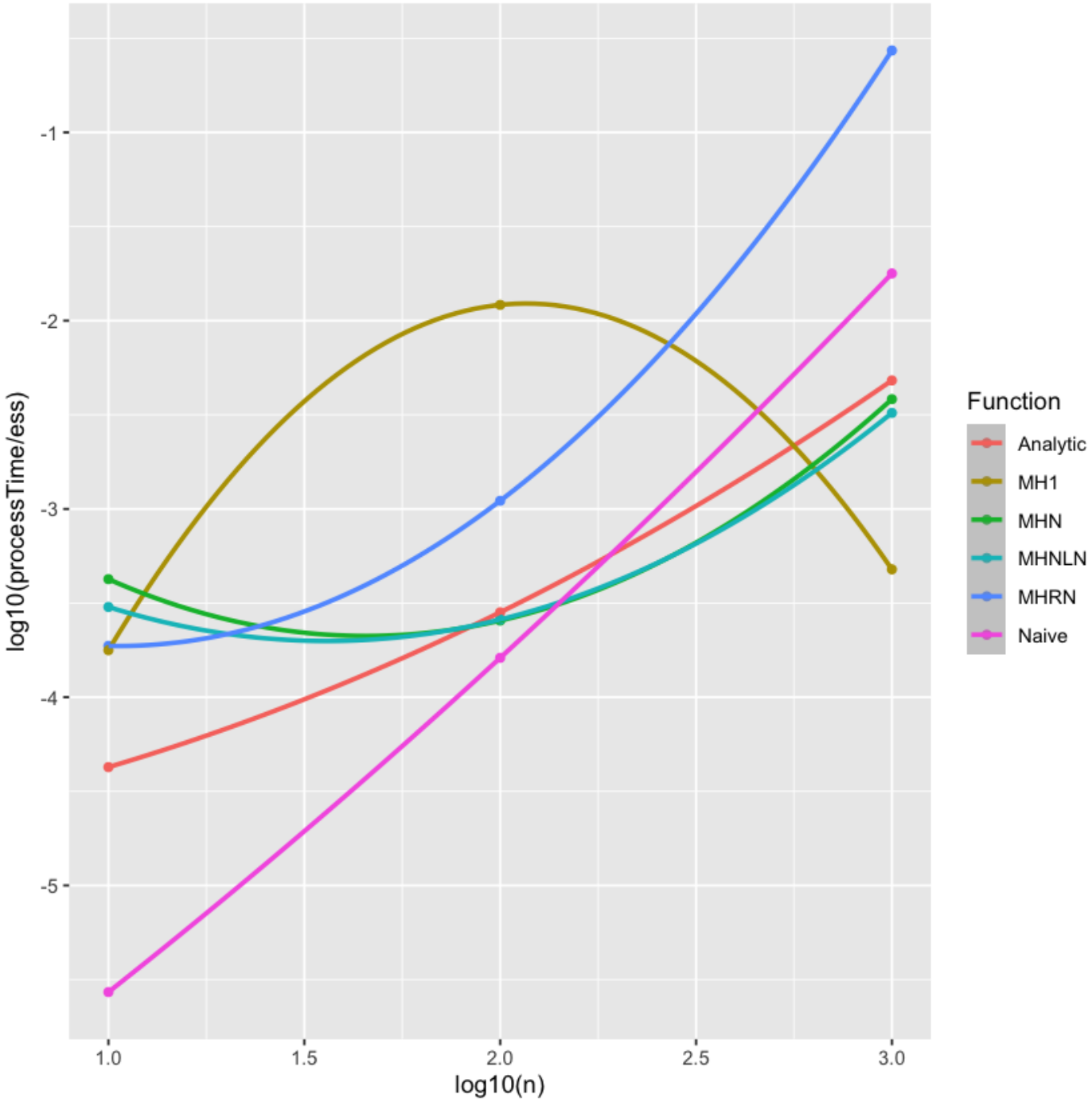

Figure 6: CPU time per effective sample size (ESS) for samplers from $G(n, p)$ random graphs

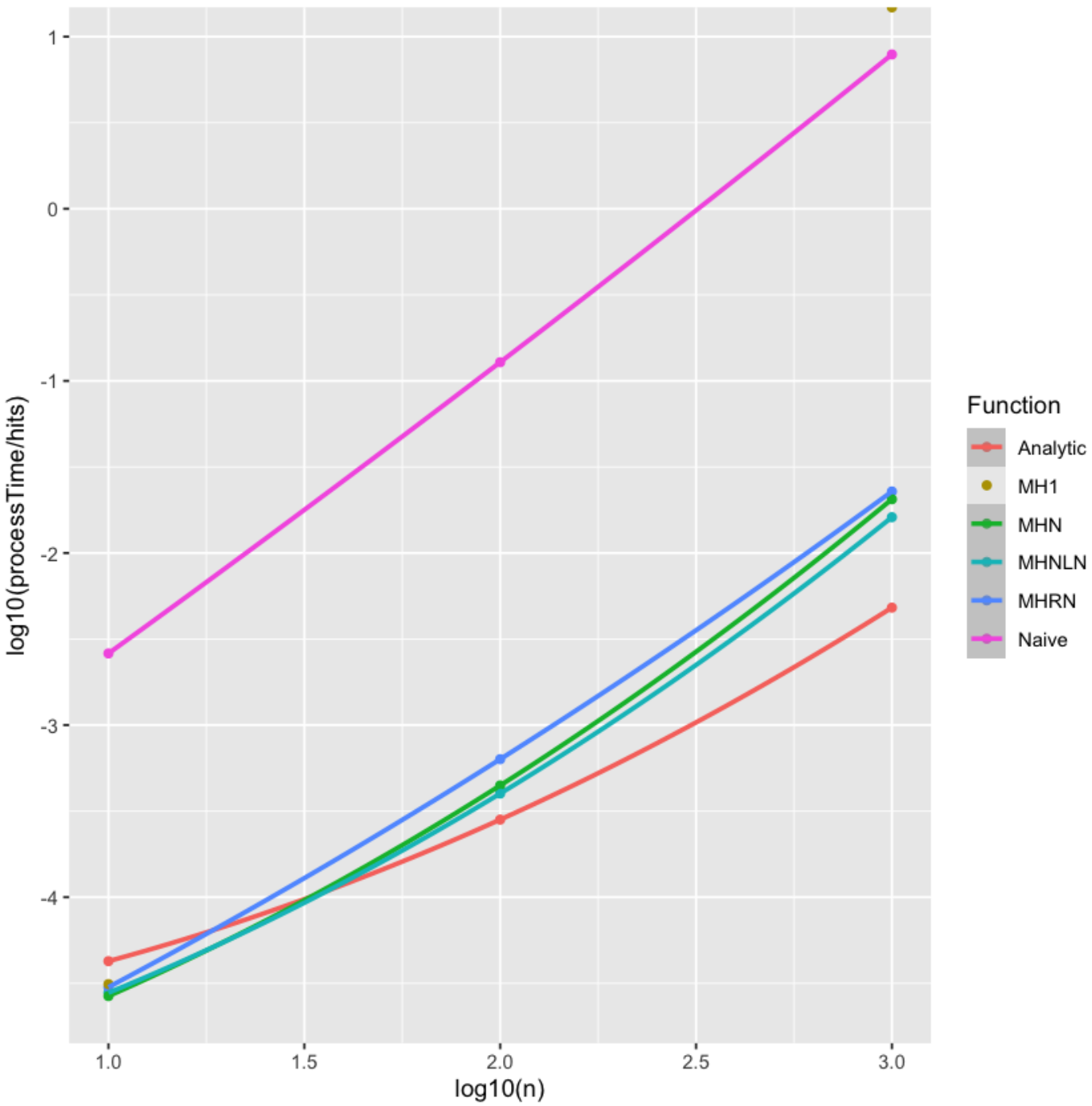

Figure 7: CPU time per non-zero weight for samples for connectivity in $G(n, p)$ random graphs

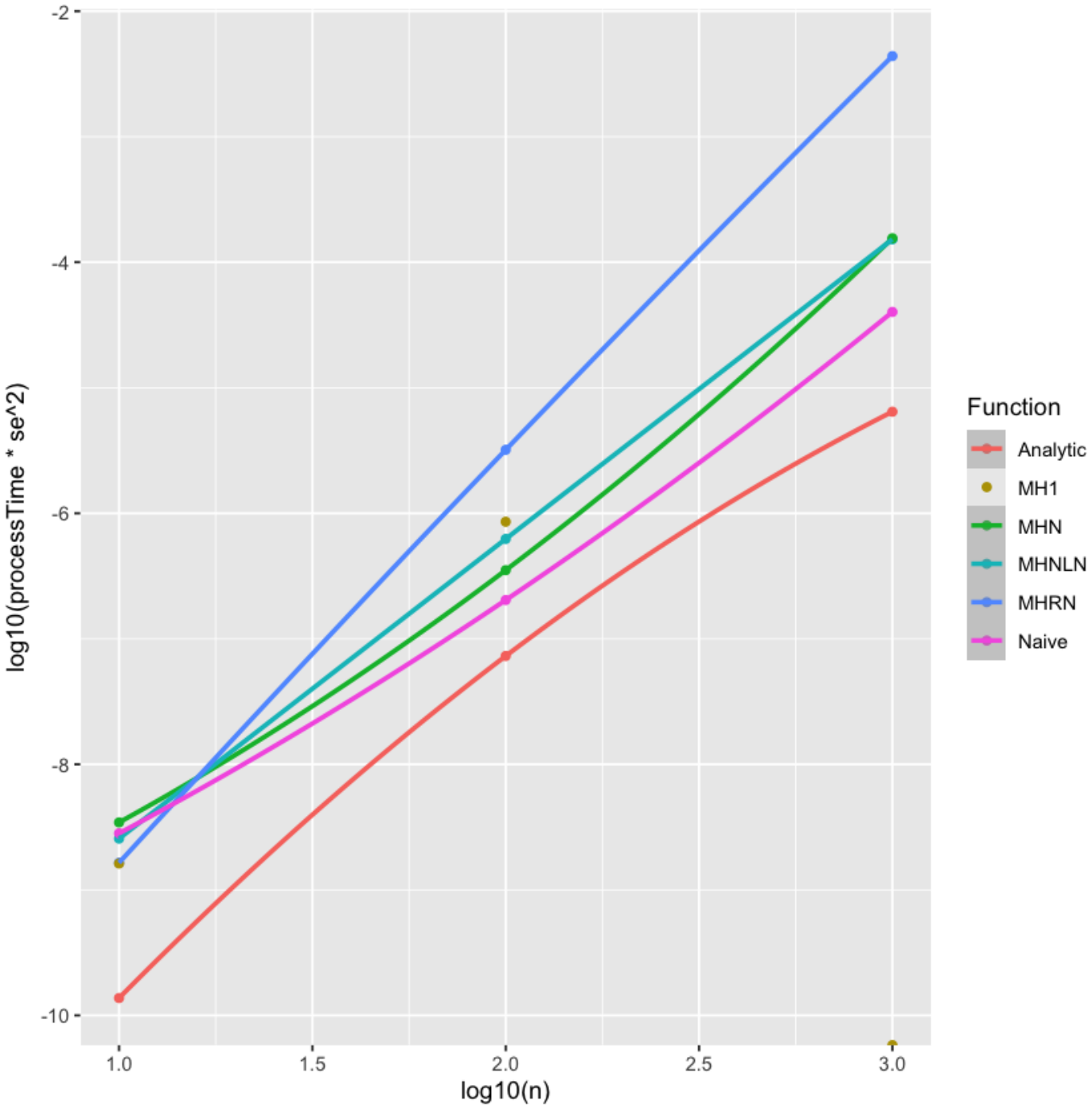

Figure 8: CPU time per unit precision for samplers for connectivity in $G(n, p)$ random graphs

7.6.2 Mixing

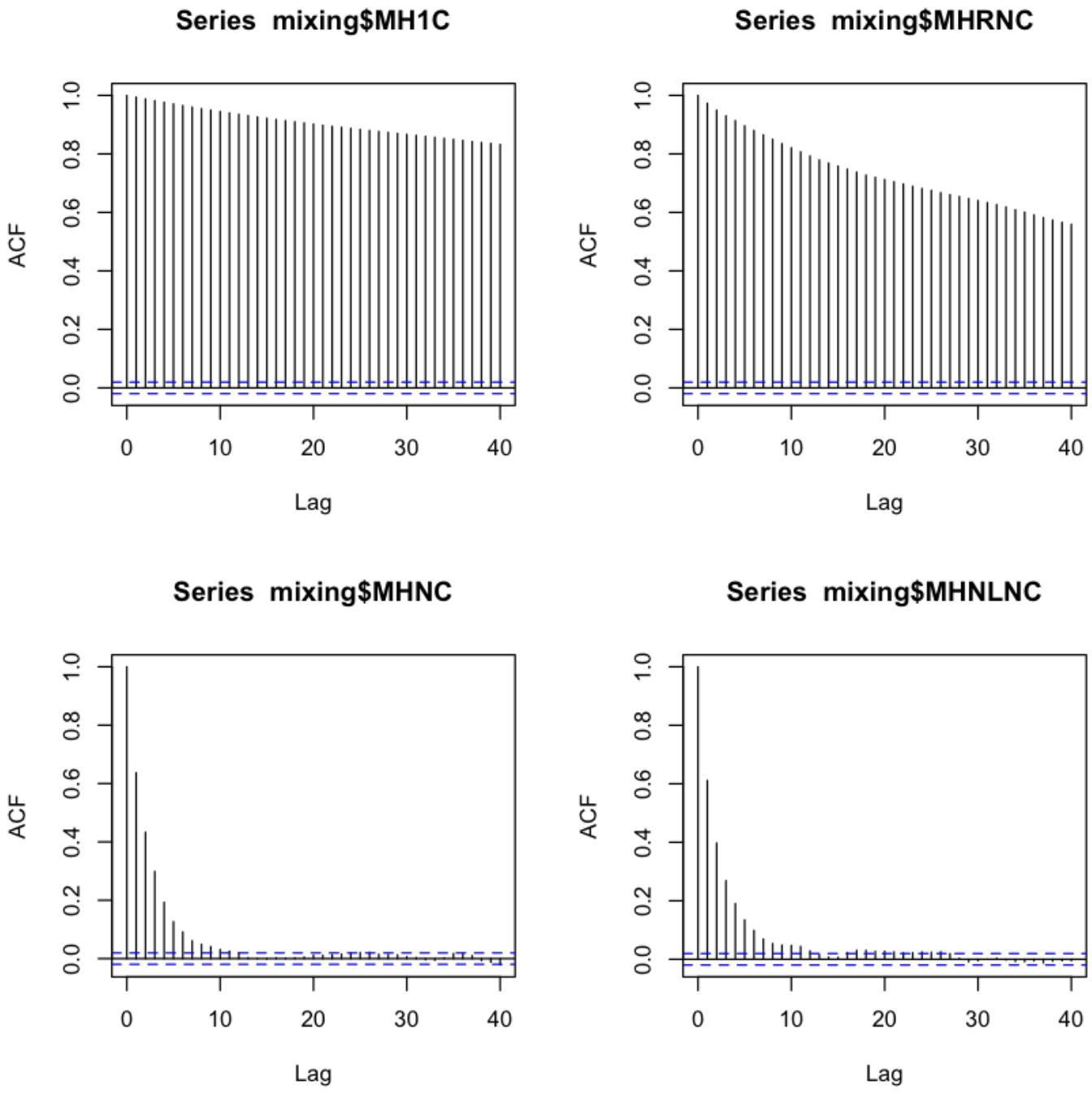

Figure 9: Autocorrelation number of connected components for MCMC samplers from random graphs

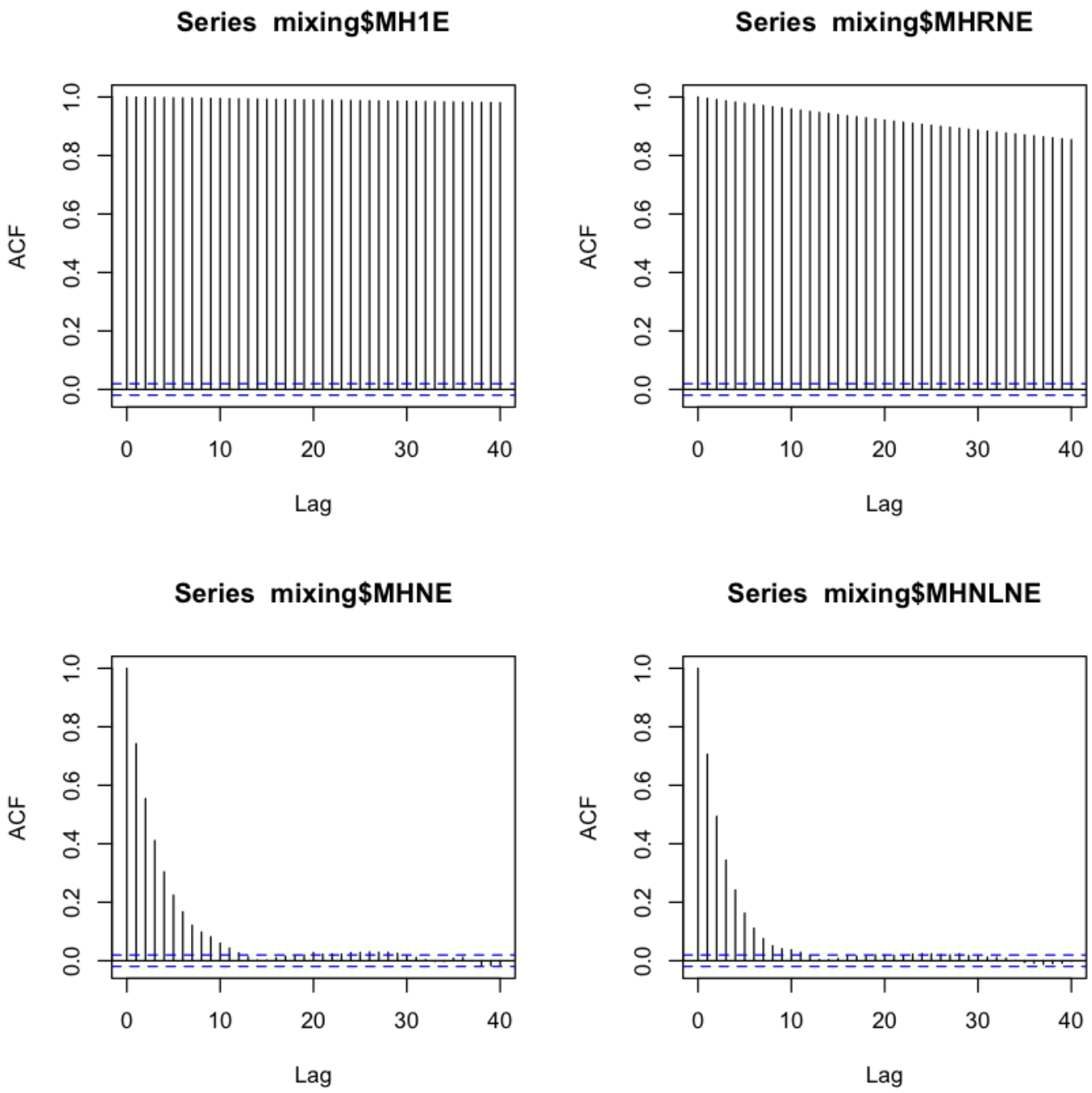

Figure 10: Autocorrelatoin of edges for MCMC samplers from random graphs

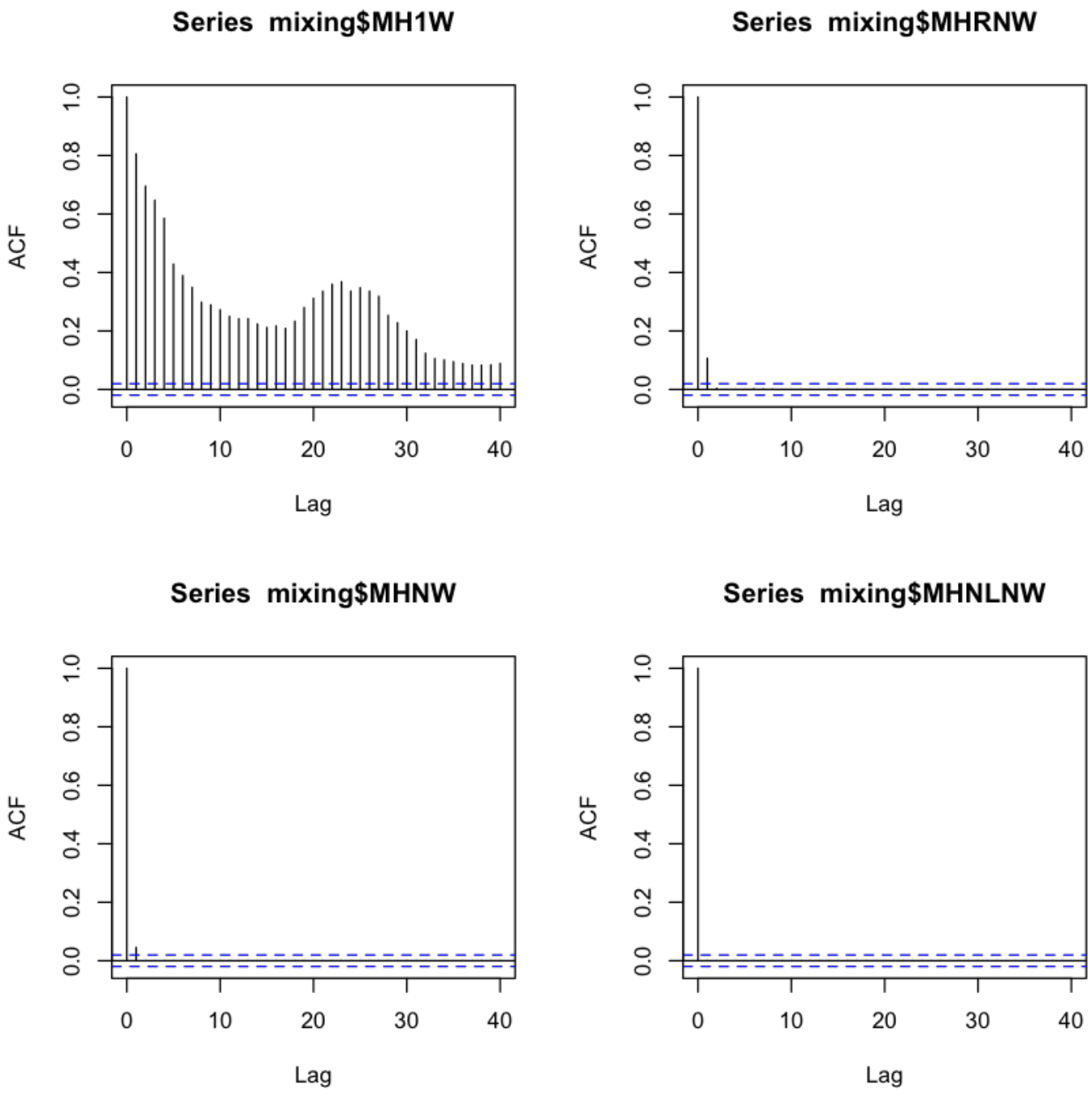

Figure 11: Autocorrelation of weight values for MCMC samplers from random graphs

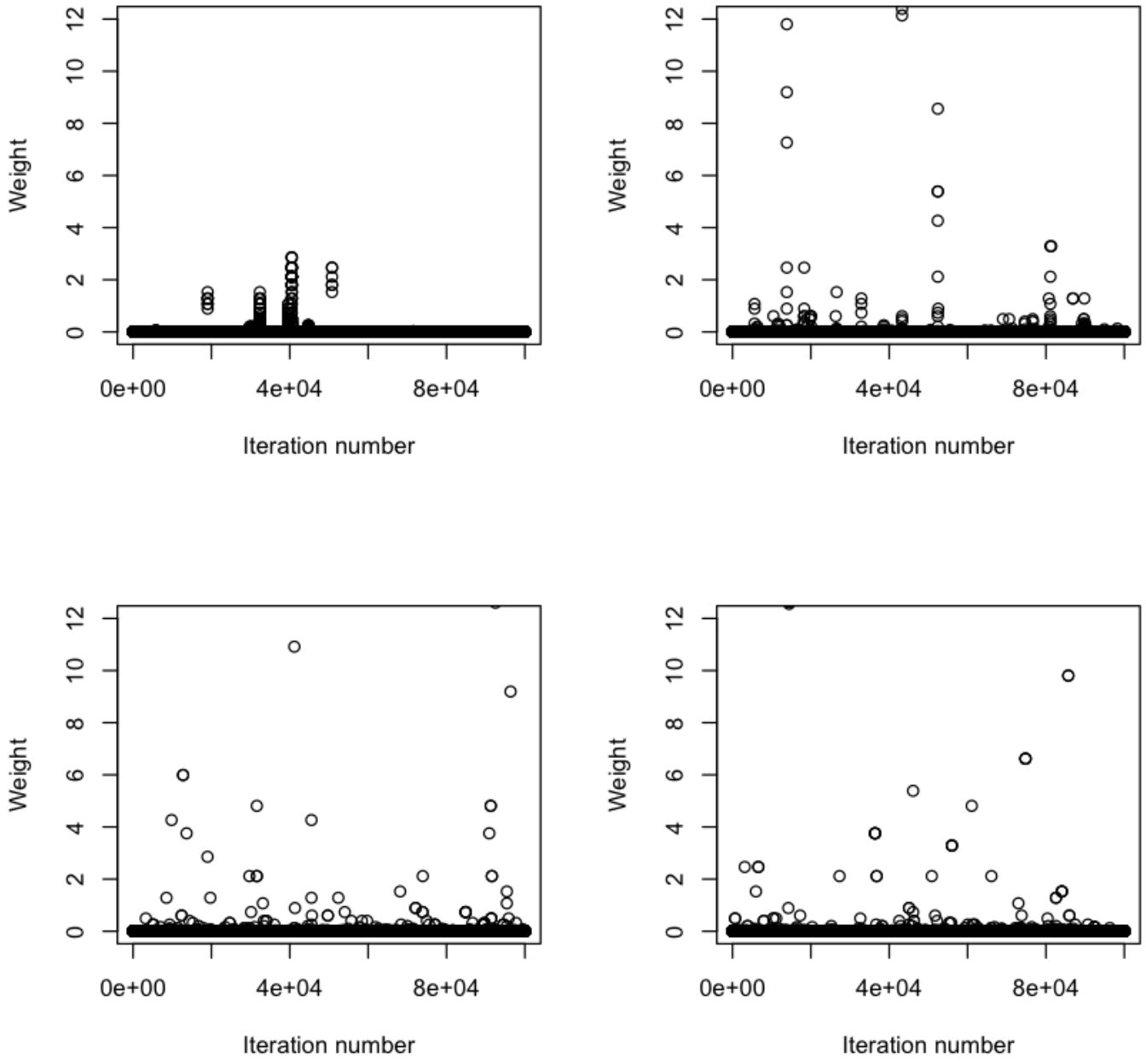

Figure 12: Weight values of samples for MCMC samplers from random graphs

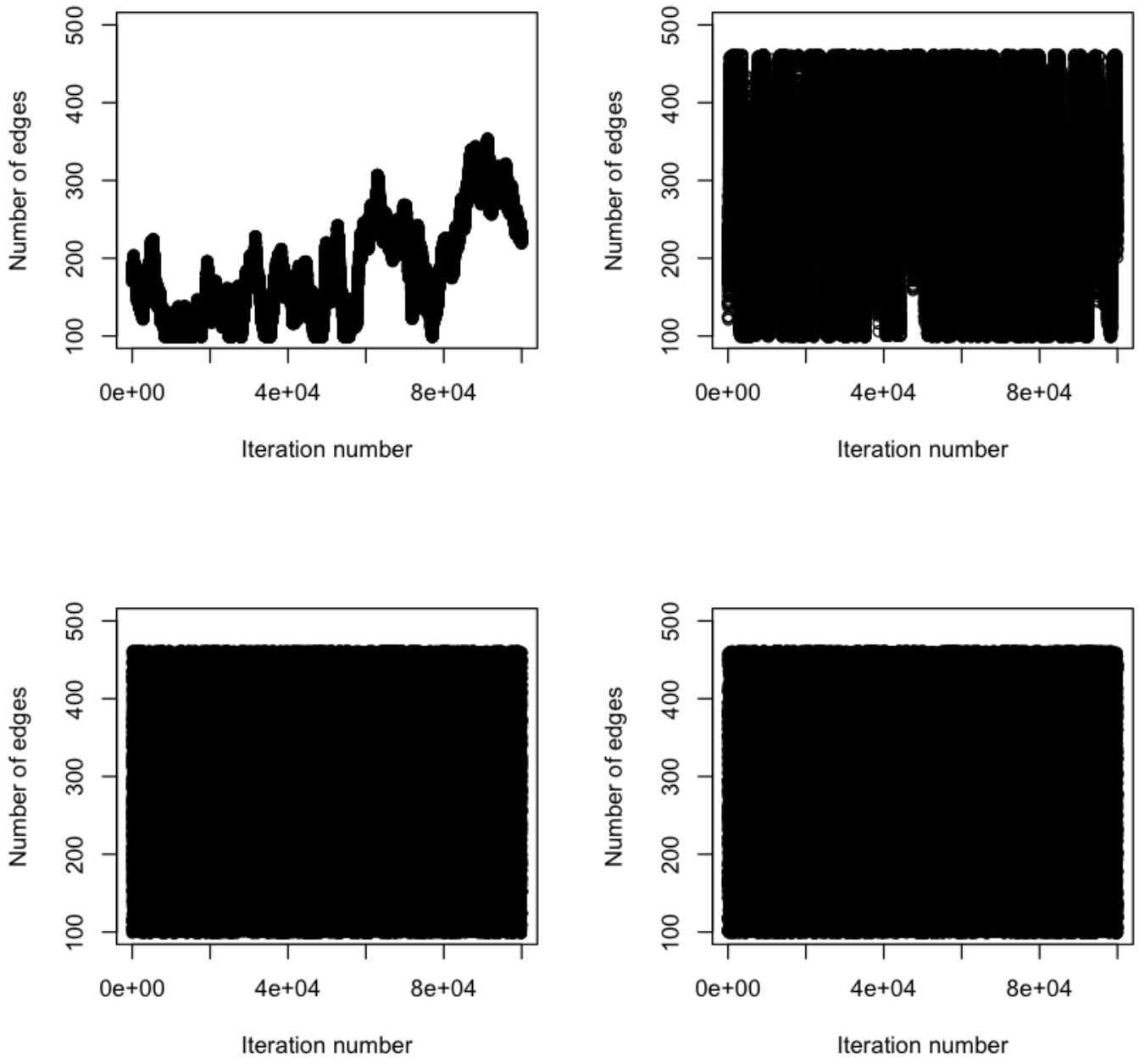

Figure 13: Number of edges in samples for MCMC samplers from random graphs

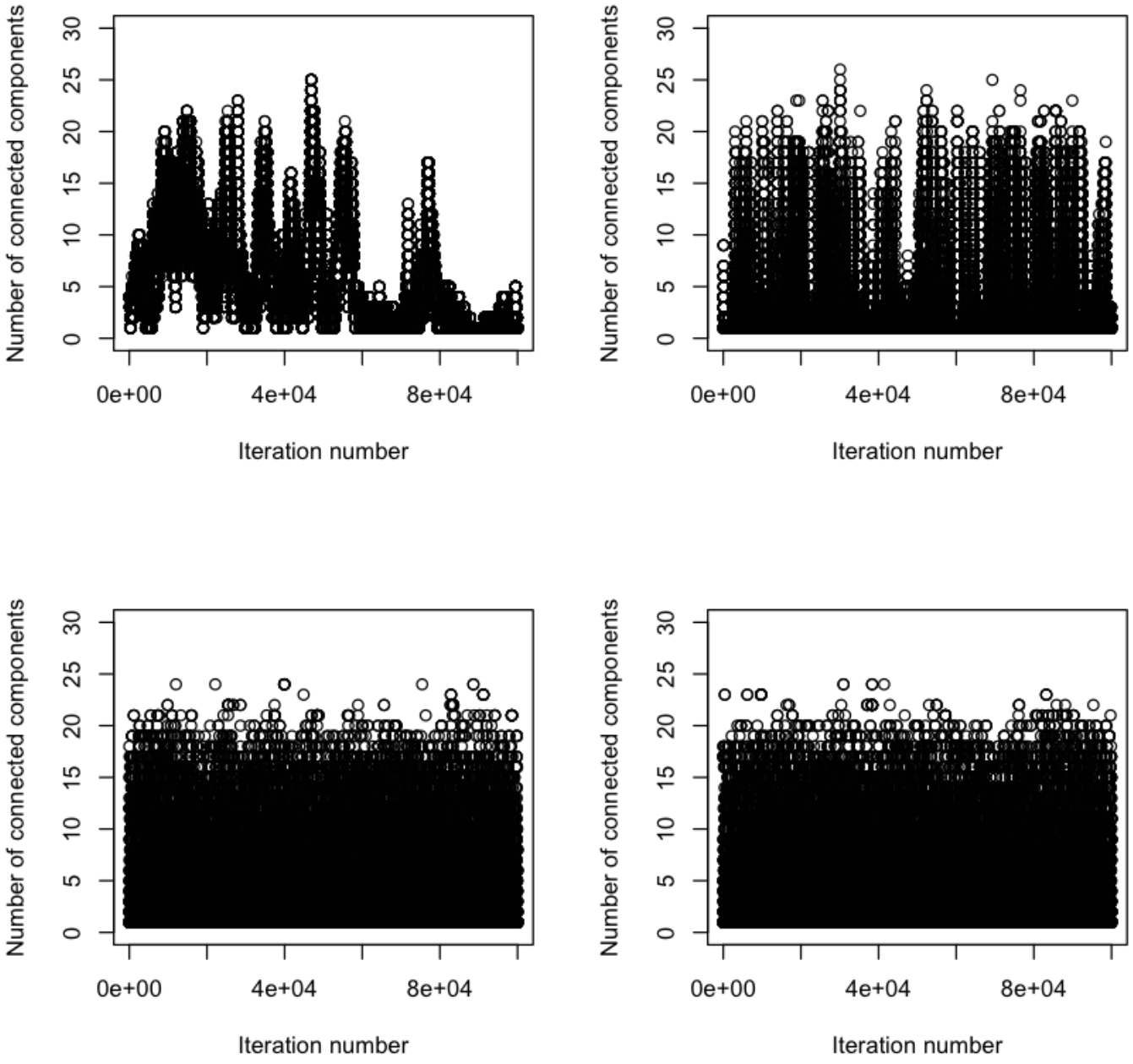

Figure 14: Number of connected components in samples for MCMC samplers from random graphs

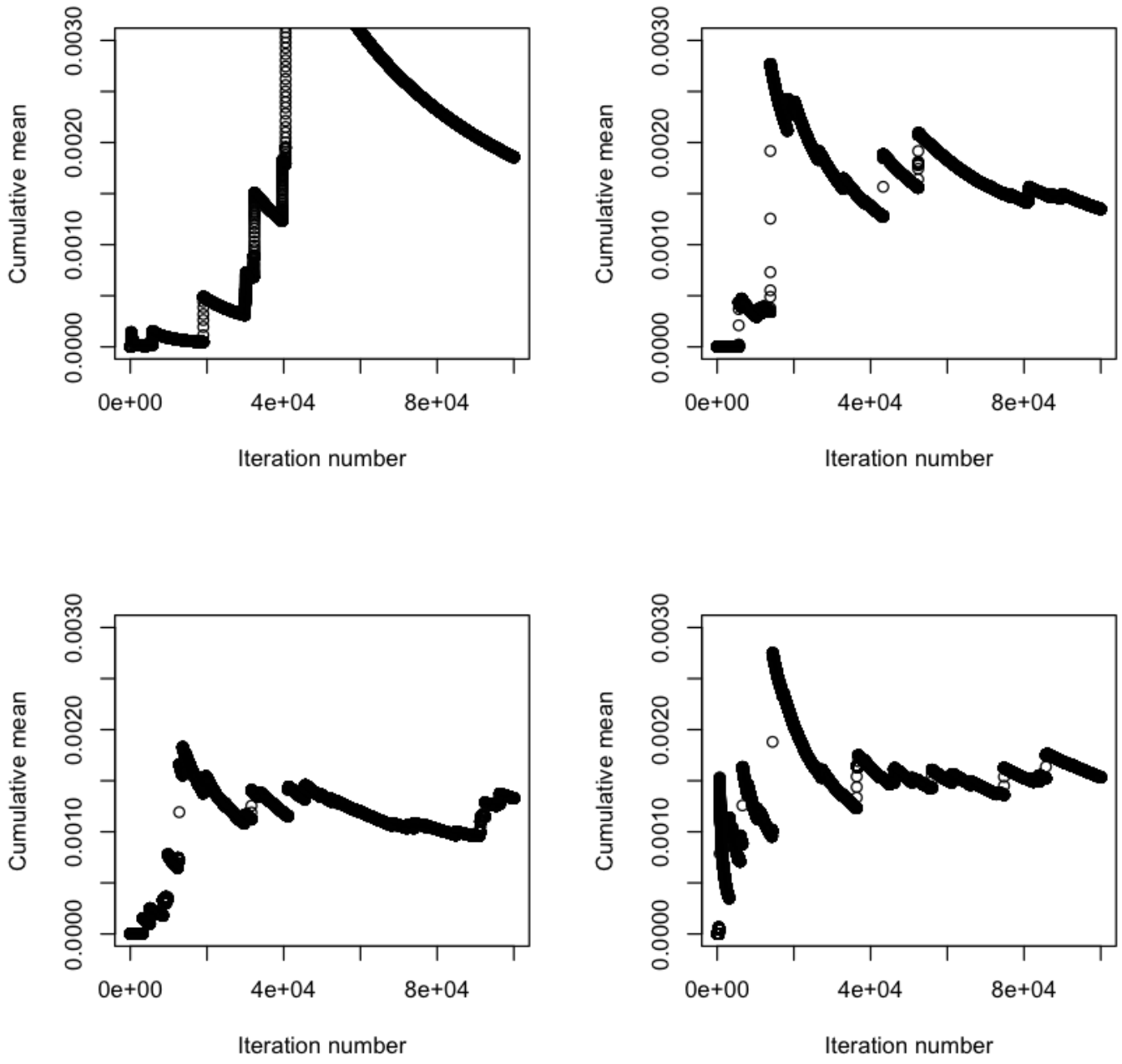

Figure 15: Cumulative mean of weight values for MCMC samplers from random graphs

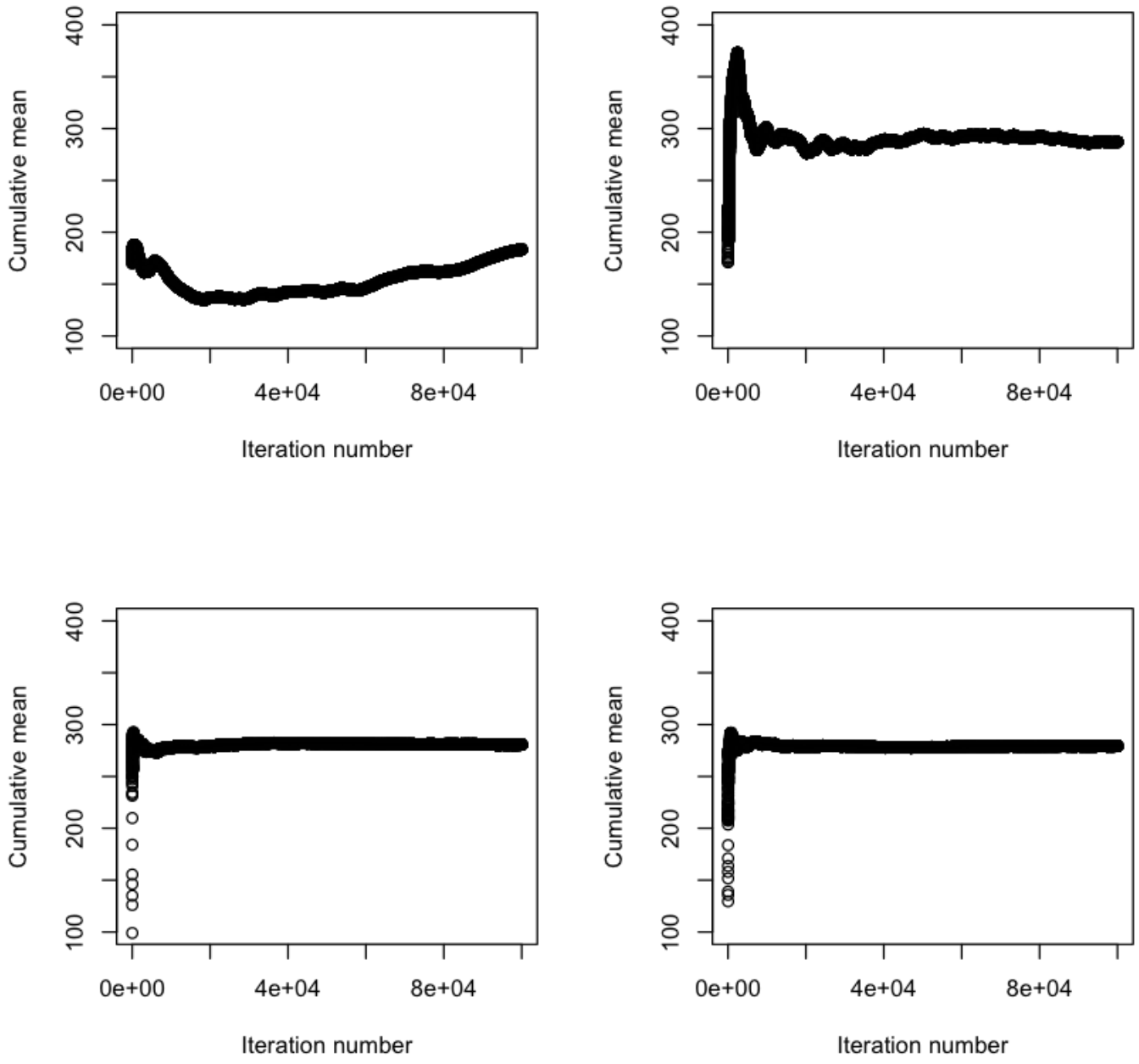

Figure 16: Cumulative mean of number of edges for MCMC samplers from random graphs

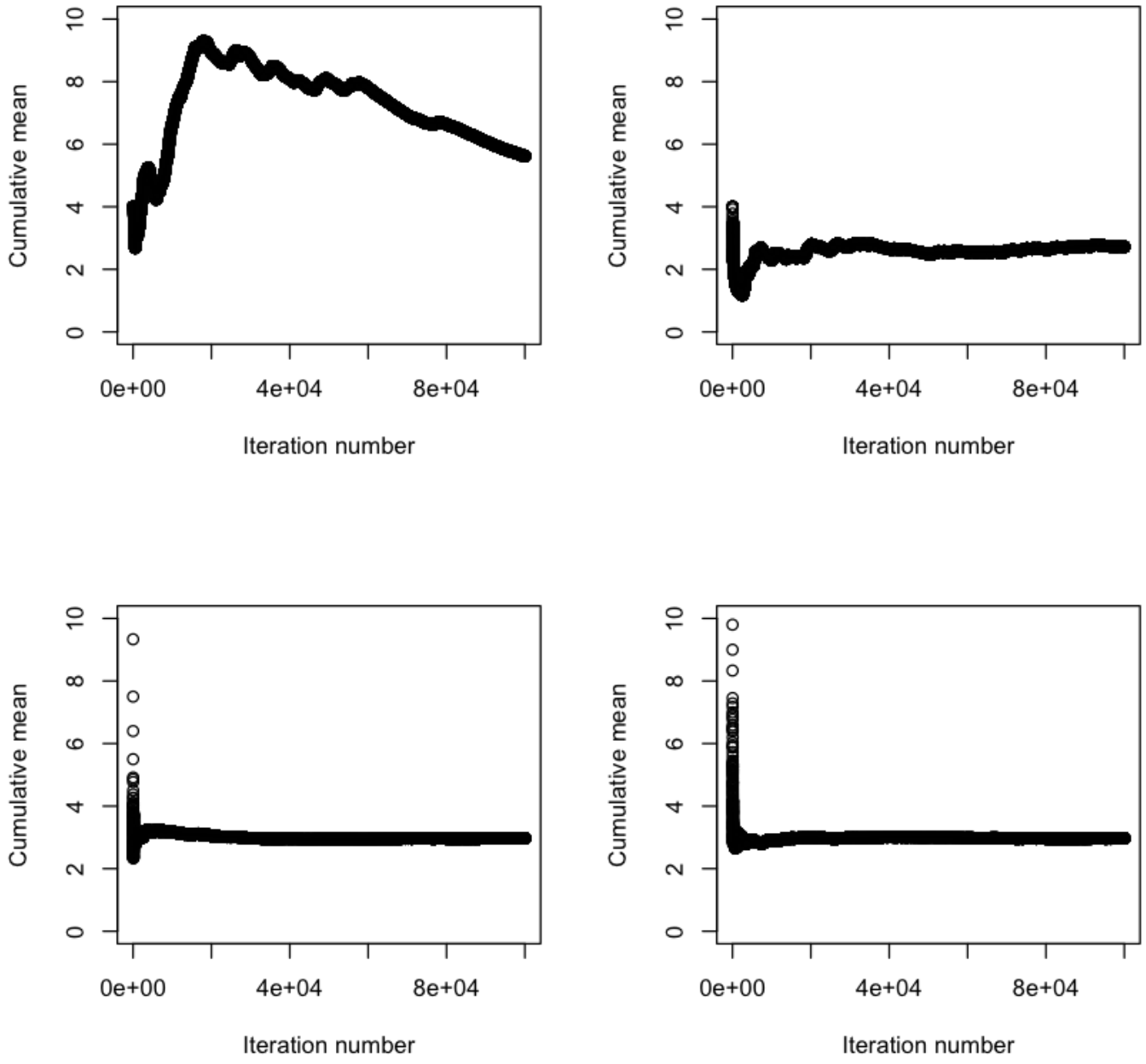

Figure 17: Cumulative mean of number of connected components for MCMC samplers from random graphs

7.7 Sample code

7.7.1 Naive sampling algorithm

```
# Naive sampler for random graphs
import random
import numpy as np

def connected(g):
    """
    Check if a graph is connected using DFS. Expects g as an adjacency list.
    Returns a boolean
    """
    # Now we check if the graph is complete using DFS
    n = len(g)
    visited = [False]*n
    s = [0] # start our stack with just 0
    n_visited = 0
    while len(s) > 0:
        next_node = s.pop()
        if (visited[next_node]):
            continue # nothing to do
        visited[next_node] = True
        n_visited += 1
        s.extend(g[next_node])
    return (n_visited == n)

def gen_graph(n,p):
    """
    Generate an n,p random graph
    """
    # First we construct our graph as an adjacency list
    # Nodes are labelled 0 through n-1
    neighbours = [[] for _ in range(n)]
    for i in range(n):
        for j in range(i+1,n):
            if (random.random() < p):
                neighbours[i].append(j)
                neighbours[j].append(i)
    return neighbours

def one_sample(n,p):
    """
    Generate a single random graph and test for connectivity

    Return true if connected, false otherwise
    """
```

```
    g = gen_graph(n,p)
    return connected(g)

def sample(n,p,b):
    """
    Take b samples from an n,p random graph. Returns a tuple containing
    the estimate and the estimated standard error
    """
    return [int(one_sample(n,p)) for _ in range(b)]

def evaluate(vals):
    """
    Evluate values from the naive sampler"""
    ss = len(vals) # sample size
    p = np.mean(vals) # estimate of p
    se = np.std(vals)/np.sqrt(ss)
    hits = sum(vals)
    return (p,se,hits,ss)

def test():
    random.seed(535)
    print(gen_graph(10, 0.1)) # [[], [], [4], [6], [2, 9], [], [3], [], [], [4]]
    print(connected([[1],[0,2],[1]])) # TRUE
    print(connected([[1],[0,2],[1],[]])) # FALSE
    print(evaluate(sample(10,0.3,10)))

if __name__ == "__main__":
    test()
```

7.7.2 Analytic variance reduction

```
import random
import scipy
import scipy.stats
import numpy as np

class UnionFind():
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1]*n
        self.components = n
    def find(self, i):
        """
        Find the root of node i
        """
```

```
        if (self.parent[i] == i):
            return i
        else:
            k = self.find(self.parent[i])
            self.parent[i] = k # Path compression
            return k
    def union(self, i, j):
        ri = self.find(i)
        rj = self.find(j)
        if ri == rj:
            return # already in the same group
        self.components -= 1
        if (self.rank[ri] > self.rank[rj]): # Union by rank
            self.parent[rj] = ri
        elif (self.rank[rj] > self.rank[ri]):
            self.parent[ri] = rj
        else:
            self.parent[ri] = rj
            self.rank[ri] += 1
    def connected(self):
        return (self.components == 1)

class Graph():
    """
    Graph on n vertices
    """
    def __init__(self, n):
        self.uf = UnionFind(n)
        self.edges = set() # edge set
    def add_edge(self, i, j):
        if i == j: # no self edges
            return
        self.edges.add((i,j))
        self.edges.add((j,i))
        self.uf.union(i,j)
    def num_edges(self):
        return len(self.edges)//2 # we want an undirected graph
    def is_connected(self):
        return self.uf.connected()

def sample_once(n,p):
    """
    Get one sample from the reduced variance sampler
    """
    g = Graph(n)
    while not g.is_connected():
```

```
            # Add random edge
            g.add_edge(random.randint(0,n-1), random.randint(0,n-1))
        return scipy.stats.binom.sf(g.num_edges()-1, n*(n-1)/2, p)

def sample(n,p,b):
    """
    Take b samples from an n,p random graph. Returns a tuple containing
    the estimate and the estimated standard error
    """
    return [sample_once(n,p) for _ in range(b)]

def evaluate(vals):
    """
    Evaluate values from the naive sampler"""
    ss = len(vals) # sample size
    p = np.mean(vals) # estimate of p
    se = np.std(vals)/np.sqrt(ss)
    hits = ss # every sample is a hit
    return (p,se,hits,ss)

def test():
    # Union find tests
    uf = UnionFind(3)
    uf.union(0,1)
    print(uf.connected()) # False
    uf.union(1,2)
    print(uf.connected()) # True
    # Graph tests
    g = Graph(4)
    g.add_edge(0,1)
    g.add_edge(1,2)
    print(g.num_edges()) # 2
    print(g.is_connected()) # False
    g.add_edge(0,3)
    print(g.is_connected()) # True
    print(evaluate(sample(100, 0.02732, 1000)))

if __name__ == "__main__":
    test()
```

7.7.3 MCMC Sampler

```
"""
Metropolis hastings for sampling from random graphs for connectedness
"""
```

```
import arviz as az
import numpy as np
import random
import scipy

class UnionFind():
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1]*n
        self.components = n
    def find(self, i):
        """
        Find the root of node i
        """
        if (self.parent[i] == i):
            return i
        else:
            k = self.find(self.parent[i])
            self.parent[i] = k # Path compression
            return k
    def union(self, i, j):
        ri = self.find(i)
        rj = self.find(j)
        if ri == rj:
            return # already in the same group
        self.components -= 1
        if (self.rank[ri] > self.rank[rj]): # Union by rank
            self.parent[rj] = ri
        elif (self.rank[rj] > self.rank[ri]):
            self.parent[ri] = rj
        else:
            self.parent[ri] = rj
            self.rank[ri] += 1
    def num_components(self):
        return self.components

class Graph():
    """
    Graph with additional functionality for adding and removing random edges
    """
    def __init__(self, n):
        self.n = n
        self.edge_set = set() # for easy lookup when adding
        self.edge_list = [] # for random removal
        self.edge_count = 0
    def add_m_edges(self, m):
```

```
        start_count = self.edge_count
        while self.edge_count < start_count + m:
            # Generate random edges
            i, j = random.randint(0, self.n-1), random.randint(0, self.n-1)
            if (i == j):
                continue
            if (i,j) in self.edge_set:
                continue
            self.edge_set.add((i,j))
            self.edge_set.add((j,i))
            self.edge_list.append((i,j))
            self.edge_count += 1
    def remove_m_edges(self, m):
        if len(self.edge_set) < m:
            raise ValueError(f"Cannot remove {m} edges from a graph with\
            {len(self.edges)} edges!")
        for _ in range(m):
            remove_idx = random.randint(0, self.edge_count-1)
            to_remove = self.edge_list[remove_idx]
            self.edge_list[remove_idx] = self.edge_list[-1]
            self.edge_list.pop() # Remove from the end
            self.edge_set.remove(to_remove)
            self.edge_set.remove(to_remove[::-1])
            self.edge_count -= 1
    def num_components(self):
        # Just add all our edges to a union find data structure
        uf = UnionFind(self.n)
        for (i,j) in self.edge_list:
            uf.union(i,j)
        return uf.num_components()

def sample(n,p,b,scale):
    g = Graph(n)
    lower = n-1
    upper = round(n*np.log(n)/2)
    start = random.randint(lower, upper) # inclusive
    g.add_m_edges(start) # Initialize our graph
    vals = np.ndarray(b, dtype=np.float64)
    comps = np.ndarray(b, dtype=np.float64)
    current_edges = start
    current_comp = g.num_components()
    for i in range(b):
        next_edges = round(np.random.normal(current_edges, scale))
        if (next_edges >= lower and next_edges <= upper and next_edges != current_edges):
            # Update
            if (next_edges > current_edges):
```

```
                g.add_m_edges(next_edges-current_edges)
            else:
                g.remove_m_edges(current_edges-next_edges)
            current_edges = next_edges
            current_comp = g.num_components()
        current_connected = (current_comp == 1)
        if current_connected:
            w = (upper+1-lower)*scipy.stats.binom.pmf(current_edges, n*(n-1)/2, p)
        else:
            w = 0
        vals[i] = w
        comps[i] = current_comp
    return vals, comps

def evaluate(val_pair):
    vals, comps = val_pair
    ans = np.mean(vals)
    ess = az.ess(vals)
    se = np.std(vals)/np.sqrt(ess)
    hits = sum([int(x > 0) for x in vals])
    return (ans, se, hits, ess)

def test():
    # sample_mh(10,0.3,10)
    # g = Graph(5)
    # g.add_m_edges(3)
    # print(g.edge_list) # 3 edges
    # g.remove_m_edges(2)
    # print(g.edge_list) # 1 edge
    # print(g.edge_set) # double for both directions
    # real mh
    # print(evaluate(sample(10, 0.07544, 10000,10)))
    # print(evaluate(sample(100, 0.02732, 100000, 25)))
    # print(evaluate(sample(1000, 0.005098, 10000, 10)))
    print(evaluate(sample(10000, 0.0007382, 100000, 100)))
    # print(np.mean(sample_mh(100, 0.02732, 100000, 10)))

if __name__ == "__main__":
    test()
```

7.7.4 Testing framework

```
import analytic
import mh
import naive
```

```
import random
import time
import numpy as np

import pandas

class TestFunction():
    def __init__(self, sample, evaluate, name):
        self.sample = sample
        self.evaluate = evaluate
        self.name = name

class Test():
    def __init__(self, function : TestFunction, n, p, b = 10):
        self.function = function
        self.n = n
        self.p = p
        self.b = b
    def run(self):
        start_time = time.process_time()
        # Run the test and report results
        vals = self.function.sample(self.n,self.p,self.b)
        end_time = time.process_time()
        elapsed_time = end_time - start_time
        mu, se, hits, ess = self.function.evaluate(vals)
        return (elapsed_time, mu, se, hits, ess)
    def __str__(self):
        return f"{self.function.name}: n = {self.n}, p = {self.p}, b = {self.b}"

# Global tfs
naive_tf = TestFunction(naive.sample, naive.evaluate, "Naive")
analytic_tf = TestFunction(analytic.sample, analytic.evaluate, "Analytic")
mh_1 = TestFunction(lambda n,p,b : mh.sample(n,p,b,1), mh.evaluate, "MH1")
mh_rn = TestFunction(lambda n,p,b : mh.sample(n,p,b,np.sqrt(n)), mh.evaluate, "MHRN")
mh_n = TestFunction(lambda n,p,b : mh.sample(n,p,b,n), mh.evaluate, "MHN")
mh_nln = TestFunction(lambda n,p,b : mh.sample(n,p,b,n*np.log(n)/4), mh.evaluate, "MHNLN")

def main():
    random.seed(535)
    tests = []
    # Create our list of tests to carry out
    # for 100000 use 9.5583032816648 48e-05
    times = (10, 100, 1000, 10000)# , 10000)
    probs = (0.0754, 0.02732, 0.005098)# , 0.0007382)
    test_funcs = (naive_tf, analytic_tf, mh_1, mh_rn, mh_n, mh_nln)
    for n,p in zip(times, probs):
```

```
        for tf in test_funcs:
            tests.append(Test(tf, n, p, b = 100000))
    print("The test plan is")
    print("\n".join([str(t) for t in tests]))
    print("Tests will be shuffled")
    input("Press enter to continue:")
    print("Continuing...")
    random.shuffle(tests)
    results = []
    for i,t in enumerate(tests):
        random.seed(535)
        print(f"Running test {t}. [{i+1}/{len(tests)}]")
        process_time, mu, se, hits, ess = t.run()
        results.append((t.function.name, t.n, t.p, t.b, mu, se, hits, ess, process_time))
        print("Done")
    print("Done running tests. Writing out...")
    out_df = pandas.DataFrame(results, columns=["Function", "n", "p", "b", "mu", "se", "hits",
    out_df.to_csv("results.csv", index=False)

if __name__ == "__main__":
    main()
```

7.7.5 Mixing analysis

```
"""
This is an instrumented version of mh.py which is used to measure mixingn of the
chain
"""
import numpy as np
import random
import scipy
import pandas

class UnionFind():
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [1]*n
        self.components = n
    def find(self, i):
        """
        Find the root of node i
        """
        if (self.parent[i] == i):
            return i
        else:
```

```
            k = self.find(self.parent[i])
            self.parent[i] = k # Path compression
            return k
    def union(self, i, j):
        ri = self.find(i)
        rj = self.find(j)
        if ri == rj:
            return # already in the same group
        self.components -= 1
        if (self.rank[ri] > self.rank[rj]): # Union by rank
            self.parent[rj] = ri
        elif (self.rank[rj] > self.rank[ri]):
            self.parent[ri] = rj
        else:
            self.parent[ri] = rj
            self.rank[ri] += 1
    def connected(self):
        return (self.components == 1)
    def num_components(self):
        return (self.components)

class Graph():
    """
    Graph with additional functionality for adding and removing random edges
    """
    def __init__(self, n):
        self.n = n
        self.edge_set = set() # for easy lookup when adding
        self.edge_list = [] # for random removal
        self.edge_count = 0
    def add_m_edges(self, m):
        start_count = self.edge_count
        while self.edge_count < start_count + m:
            # Generate random edges
            i, j = random.randint(0, self.n-1), random.randint(0, self.n-1)
            if (i == j):
                continue
            if (i,j) in self.edge_set:
                continue
            self.edge_set.add((i,j))
            self.edge_set.add((j,i))
            self.edge_list.append((i,j))
            self.edge_count += 1
    def remove_m_edges(self, m):
        if len(self.edge_set) < m:
            raise ValueError(f"Cannot remove {m} edges from a graph with\
```

```
            {len(self.edges)} edges!")
        for _ in range(m):
            remove_idx = random.randint(0,self.edge_count-1) # pick edge
            to_remove = self.edge_list[remove_idx] # save the edge
            self.edge_list[remove_idx] = self.edge_list[-1] # replace with end
            self.edge_list.pop() # remove end
            self.edge_set.remove(to_remove) # remove from edge set as well
            self.edge_set.remove(to_remove[::-1])
            self.edge_count -= 1
    def is_connected(self):
        # Just add all our edges to a union find data structure
        uf = UnionFind(self.n)
        for (i,j) in self.edge_list:
            uf.union(i,j)
        return uf.connected()
    def num_components(self):
        # Just add all our edges to a union find data structure
        # We have to rebuild this each time because the graph is dynamic
        uf = UnionFind(self.n)
        for (i,j) in self.edge_list:
            uf.union(i,j)
        return uf.num_components()

def sample(n,p,b,scale):
    g = Graph(n)
    lower = n-1
    upper = round(n*np.log(n))
    start = random.randint(lower, upper) # inclusive
    g.add_m_edges(start) # Initialize our graph
    comps = np.ndarray(b, dtype=np.float64)
    edges = np.ndarray(b, dtype=np.float64)
    vals = np.ndarray(b, dtype=np.float64)
    current_edges = start
    current_comp = g.num_components()
    for i in range(b):
        next_edges = round(np.random.normal(current_edges, scale))
        if (next_edges >= lower and next_edges <= upper and next_edges != current_edges):
            # Update
            if (next_edges > current_edges):
                g.add_m_edges(next_edges-current_edges)
            else:
                g.remove_m_edges(current_edges-next_edges)
            current_edges = next_edges
            current_comp = g.num_components()
        if current_comp == 1:
            w = (upper+1-lower)*scipy.stats.binom.pmf(current_edges, n*(n-1)/2, p)
```

```
            else:
                w = 0
            # Add our graph to the last n
            vals[i] = w
            comps[i] = current_comp
            edges[i] = current_edges
        return (vals, edges, comps)

# def evaluate(vals):
#     ans = np.mean(vals)
#     ess = az.ess(vals)
#     se = np.std(vals)/np.sqrt(ess)
#     hits = sum([int(x > 0) for x in vals])
#     return (np.mean(vals),se, hits)

def main():
    """
    Instrumented MCMC to evaluate mixing
    """
    n = 100
    p =  0.02732
    # p = 0.02
    # n = 10
    # p = 0.07544
    b = 100000
    agg_vals = []
    agg_edges = []
    agg_comps = []
    tests = ((1,np.sqrt(n),n,n*np.log(n)/4),("MH1","MHRN","MHN","MHNLN"))
    for step in tests[0]:
        random.seed(535)
        vals, edges, comps = sample(n,p,b,step)
        agg_vals.append(vals)
        agg_edges.append(edges)
        agg_comps.append(comps)

    df_dict = {}
    for i, name in enumerate(tests[1]):
        df_dict[name+"W"] = agg_vals[i]
        df_dict[name+"E"] = agg_edges[i]
        df_dict[name+"C"] = agg_comps[i]
    df = pandas.DataFrame(df_dict)
    df.to_csv(f"mixing_{n}_{b}.csv")


if __name__ == "__main__":
```

```
    main()
```

7.7.6 Edge distribution

```
import random
import scipy
import numpy as np
import matplotlib.pyplot as plt
import math

class UnionFind:
    def __init__(self, n):
        self.n = n
        self.parents = list(range(n))
        self.sizes = [1]*n # only need sizes of roots
        self.components = n
    def find(self, i):
        # Find the parent of i and perform path compression
        if self.parents[i] == i:
            return i
        else:
            j = self.find(self.parents[i])
            self.parents[i] = j # path compression
            return j
    def union(self, i, j):
        a = self.find(i)
        b = self.find(j)
        # now need them to point to each other
        if (a != b):
            self.components -=1
            if (self.sizes[a] > self.sizes[b]):
                self.parents[b] = a # small point to big (less children to fix)
                self.sizes[a] += self.sizes[b]
            else:
                self.parents[a] = b
                self.sizes[b] += self.sizes[a]

class SmartGraph:
    def __init__(self, n : int, p : float):
        self.n = n
        self.p = p
        self.neighbours = [[] for i in range(n)] # no neighbours
        # Basic stats
        self._edges = 0
        self._parents = range(n) # Create our union find data structure
```

```
        self._n_components = n
        self._union_find = UnionFind(n)
    def edges(self):
        return self._edges
    def add_edge(self, i, j):
        self.neighbours[i].append(j)
        self.neighbours[j].append(i)
        self._edges += 1
        self._union_find.union(i,j)
    def connected_components(self):
        return self._union_find.components
    def connected(self):
        return self.connected_components() == 1
    def to_string(self):
        ans = "Graph"
        for i, n in enumerate(self.neighbours):
            ans += f"\n{i}: "
            for v in n:
                ans += f"{v}, "
        return ans
    def add_random_edge(self):
        # just add random edges by trial and error should be fine for sparse
        # graphs
        # can we profile this? what is failing?
        success = False
        while not success:
            i = random.randint(0, self.n-1)
            j = random.randint(0, self.n-1)
            if i != j and j not in self.neighbours[i]:
                self.add_edge(i,j)
                success = True

def run_once(n,p):
    g = SmartGraph(n,p)
    # so we need to weight our samples by based on what they represent
    # so should be times (n choose e) divide by p(e)
    count = 0
    while not g.connected():
        count += 1
        g.add_random_edge()
    # use sf
    # print(count)
    # print(g.to_string())
    return count
    # could we optimize this? # numerical stability?
    # we could use a self balancing binary search tree ro avl tree?
```

```
def estimate_edge_dependence(vals):
    m = max(vals)
    sorted_vals = sorted(vals) # so we can count
    # now we just basically count the number that are less than whatever
    ans = [0]*m
    count = 0
    i = 0 # index into ans
    while i < m:
        while sorted_vals[count] <= i:
            count += 1
        ans[i] = count/len(vals)
        i = i+1
    return (list(range(0, m)),ans)

# There might bemore speed ups lets see
def run_many(n,p,b):
    vals = [run_once(n,p) for _ in range(b)]
    print(f"Mean: {sum(vals)/b}")
    print(f"Sd: {np.sqrt(np.var(vals))}")
    edge_counts = estimate_edge_dependence(vals)
    # Estimate the probability of a graph with e edges being connected
    plt.plot(edge_counts[0], edge_counts[1])
    pmf_vals = scipy.stats.binom.pmf(edge_counts[0], n*(n-1)/2, p) # pmf(k,n,p)
    # plt.plot(edge_counts[0], pmf_vals)
    joint_p = pmf_vals*edge_counts[1] # renormalize
    plt.plot(edge_counts[0], joint_p/max(joint_p))
    N = n*(n-1)/2
    plt.axvline(N*np.log(n)/n) # this is how many we expect at threshold
    plt.axvline(N*p, color="red") # this is the average number of edges
    plt.axvline(n, color="green") # minimum number of edges for connected graph
    # plt.plot(edge_counts[0], pmf_vals)
    plt.show()

if __name__ == "__main__":
    run_many(100, 0.07, 100000)
    # run_many(1000, [0.005, 0.006], 100) # we could generate the whole curve this way
# Could I do this in some library?
```

7.7.7 Generating p-values

```
import random
import scipy
import numpy as np
import matplotlib.pyplot as plt
```

```
import math

class UnionFind:
    def __init__(self, n):
        self.n = n
        self.parents = list(range(n))
        self.sizes = [1]*n # only need sizes of roots
        self.components = n
    def find(self, i):
        # Find the parent of i and perform path compression
        if self.parents[i] == i:
            return i
        else:
            j = self.find(self.parents[i])
            self.parents[i] = j # path compression
            return j
    def union(self, i, j):
        a = self.find(i)
        b = self.find(j)
        # now need them to point to each other
        if (a != b):
            self.components -=1
            if (self.sizes[a] > self.sizes[b]):
                self.parents[b] = a # small point to big (less children to fix)
                self.sizes[a] += self.sizes[b]
            else:
                self.parents[a] = b
                self.sizes[b] += self.sizes[a]

class SmartGraph:
    def __init__(self, n : int, p : float):
        self.n = n
        self.p = p
        self.neighbours = [[] for i in range(n)] # no neighbours
        # Basic stats
        self._edges = 0
        self._parents = range(n) # Create our union find data structure
        self._n_components = n
        self._union_find = UnionFind(n)
    def edges(self):
        return self._edges
    def add_edge(self, i, j):
        self.neighbours[i].append(j)
        self.neighbours[j].append(i)
        self._edges += 1
        self._union_find.union(i,j)
```

```
    def connected_components(self):
        return self._union_find.components
    def connected(self):
        return self.connected_components() == 1
    def to_string(self):
        ans = "Graph"
        for i, n in enumerate(self.neighbours):
            ans += f"\n{i}: "
            for v in n:
                ans += f"{v}, "
        return ans
    def add_random_edge(self):
        # just add random edges by trial and error should be fine for sparse
        # graphs
        # can we profile this? what is failing?
        success = False
        while not success:
            i = random.randint(0, self.n-1)
            j = random.randint(0, self.n-1)
            if i != j and j not in self.neighbours[i]:
                self.add_edge(i,j)
                success = True

def run_once(n,p):
    g = SmartGraph(n,p)
    # so we need to weight our samples by based on what they represent
    # so should be times (n choose e) divide by p(e)
    count = 0
    while not g.connected():
        count += 1
        g.add_random_edge()
    # use sf
    # print(count)
    # print(g.to_string())
    return count
    # could we optimize this? # numerical stability?
    # we could use a self balancing binary search tree ro avl tree?

def estimate_edge_dependence(vals):
    m = max(vals)
    sorted_vals = sorted(vals) # so we can count
    # now we just basically count the number that are less than whatever
    ans = [0]*m
    count = 0
    i = 0 # index into ans
    while i < m:
```

```
        while sorted_vals[count] <= i:
            count += 1
        ans[i] = count/len(vals)
        i = i+1
    return (list(range(0, m)),ans)

def find_prob(vals, n, p):
    N = n*(n-1)/2
    return sum(scipy.stats.binom.sf([v-1 for v in vals], N, p))/len(vals)

# There might bemore speed ups lets see
def run_many(n,p,b,target):
    vals = [run_once(n,p) for _ in range(b)]
    # binary earch to find the ideal value of p
    low = 0
    high = 1
    mid = (low + high)/2
    for _ in range(30): # iterations is 3x precision
        prob = find_prob(vals, n, mid)
        # print(f"At {mid} got {prob}")
        if (prob > target):
            high = mid
        else:
            low = mid
        mid = (low + high)/2
    return mid


if __name__ == "__main__":
    # p = run_many(10, 0.07, 100000, 0.001)
    # p = run_many(100, 0.07, 10000, 0.001)
    # p = run_many(1000, 0.07, 1000, 0.001)
    # p = run_many(10000, 0.07, 1000, 0.001)
    p = run_many(100000, 0.07, 1000, 0.001)
    print(p)
    p = run_many(1000000, 0.07, 1000, 0.001)
    print(p)
    # run_many(1000, [0.005, 0.006], 100) # we could generate the whole curve this way
# Could I do this in some library?
```

7.7.8 Theoretical calculation

```
import math

def f(n,p):
```

```
    if n == 1:
        return 1
    c = [0]*(n+1)
    c[1] = 1
    for k in range(2,n+1):
        c[k] = 1-sum([c[i]*math.comb(k-1,i-1)*(1-p)**(i*(k-i)) for i in range(1,k)])
    return c[-1]

if __name__ == "__main__":
    # print(f(3,0.07544)) # should be 0.016213 = p^3 +3p^2(1-p)
    print(f(10,0.07544))
    print(f(100,0.02732))
    print(f(1000,0.005098))
    # print(f(10000,0.000732))
```