Exact and Efficient Generation of Geometric Random Variates and Random Graphs

Karl Bringmann[1] and Tobias Friedrich[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[2] Friedrich-Schiller-Universität Jena, Germany

Abstract. The standard algorithm for fast generation of Erdős-Rényi random graphs only works in the Real RAM model. The critical point is the generation of geometric random variates $\mathrm{Geo}(p)$, for which there is no algorithm that is both exact and efficient in any bounded precision machine model. For a RAM model with word size $w = \Omega(\log\log(1/p))$, we show that this is possible and present an exact algorithm for sampling $\mathrm{Geo}(p)$ in optimal expected time $\mathcal{O}(1 + \log(1/p)/w)$. We also give an exact algorithm for sampling $\min\{n, \mathrm{Geo}(p)\}$ in optimal expected time $\mathcal{O}(1+\log(\min\{1/p, n\})/w)$. This yields a new exact algorithm for sampling Erdős-Rényi and Chung-Lu random graphs of n vertices and m (expected) edges in optimal expected runtime $\mathcal{O}(n + m)$ on a RAM with word size $w = \Theta(\log n)$.

1 Introduction

Random graph generation. A large fraction of empirical research on graph algorithms is performed on random graphs. Random graph generation is also commonly used for simulating networking protocols on the Internet topology and the spread of epidemics (or rumors) on social networks (e.g. [16]) It is also an important tool in real world applications such as detecting motifs in biological networks (e.g. [21]).

We focus on homogenous and inhomogenous random graphs and consider Erdős-Rényi [9] and Chung-Lu graphs [5]. The key ingredient for generating such graphs with n vertices faster than the obvious $\Theta(n^2)$ algorithm is an efficient algorithm for exact sampling of geometric random variates.

Efficient random variate generation. Non-uniform random variates are typically generated from random variates that are uniformly distributed on $[0, 1]$. With the introduction of Intel's Ivy Bridge microarchitecture with built-in hardware digital random number generation, we can assume that we have fast access to a stream of high quality random bits. However, most non-uniform random variate generation algorithms [8, 26] assume a Real RAM, which can manipulate real numbers. This assumption is highly problematic as real numbers are infinite objects and all physical computers can only handle finite portions of these objects. Typical implementations, with e.g. double floating point precision, are efficient, but *not exact* (e.g. in C++11); that is, some outcomes might not be reachable and others might become more likely than they should.

Exact random variate generation. Knuth and Yao [18] initiated the study of exact nonuniform random number generation. They discuss the power of various restricted machine models. Several authors [10, 11, 18] provided additional results along these lines, but only presented efficient algorithms for single distributions. There also exist implementations of exact and efficient random number generators for exponential and normal distributions [15]. For parameterized families of distributions such as the geometric distribution, one can study the expected asymptotic runtime in the parameter. However, in this regard there are no exact and efficient algorithms known on any bounded precision machine model. For sampling $\mathrm{Geo}(p)$, the trivial algorithm of repeatedly sampling a coin with Bernoulli distribution $\mathrm{Ber}(p)$ until it falls heads up has expected runtime $\mathcal{O}(1/p)$, which is *not efficient* for p close to 0. (For an efficient algorithm for sampling $\mathrm{Ber}(p)$ see Appendix B.)

Exact and efficient random variate generation. Our aim is the design of exact and efficient algorithms for random variate generation. We show that this is possible in many cases and give a particularly fast algorithm for geometric random variates. This allows exact and efficient generation of Erdős-Rényi and Chung-Lu random graphs. It also allows exact and efficient generation of very large non-uniform random variates (e.g. for cryptographic applications [13]), which has been open so far.

Related work on random graph generation. There is a large body of work on generating random regular graphs (e.g. [17]), graphs with a prescribed degree distribution (e.g. [3]), and graphs with a prescribed joint degree distribution (e.g. [23]). All these algorithms converge to the desired distribution for $n \to \infty$. Note that this typically implies for finite n that only an approximation of the true distribution is reached.

The most studied random graph model is certainly the Erdős-Rényi [9] random graph $\mathcal{G}(n,p)$, where each edge of a graph of n vertices is present independently and uniformly with probability $p \in [0,1]$. Many experimental papers use algorithms with runtime $\Theta(n^2)$ to draw from $\mathcal{G}(n,p)$. The reason for this is probably that most graph algorithm software libraries such as JUNG, LEDA, BGL, and JDSL also do not contain efficient random graph generators. However, there are several algorithms which can sample from $\mathcal{G}(n,p)$ in expected time $\mathcal{O}(m+n)$ on a Real RAM, where $m = \Theta(pn^2)$ is the expected number of edges [1, 20]. This is done by using the fact that in an ordered list of all $\Theta(n^2)$ pairs of vertices the distance between two consecutive edges is geometrically distributed. The resulting distribution is *not exact* if the algorithm is run on a physical computer, which can only handle bounded precision, as it ignores the bias introduced by fixed-length number representations. The available implementation in the library NetworkX [14] therefore also does not return the desired distribution exactly. It is not obvious how to get an exact implementation even by using algebraic real numbers [19] and/or some high accuracy floating-point representation. The problem of sampling $\mathcal{G}(n,p)$ on a bounded precision model has been studied by Blanca and Mihail [2]. They showed how to achieve an approximation of the desired distribution efficiently. Our aim is an *exact and efficient* generation on a bounded precision model instead.

Generating random geometric distributions. As discussed above, all previous algorithms to sample a geometric random variate which are efficient on a Real RAM become inexact when implemented on a bounded precision machine. We assume the more realistic model of a Word RAM with word size w that can sample a random word in constant time; for a definition of the machine models see Section 2. We first observe the following lower bound for any exact sampling algorithm.

Theorem 1. *On a RAM with word size w, any algorithm sampling a geometric random variate* $\mathrm{Geo}(p)$ *with parameter $p \in (0,1)$ needs at least expected runtime $\Omega(1 + \log(1/p)/w)$.*

Theorem 1 follows from Lemma 1, which shows that the expected output size is $\Omega(\log(1/p))$ bits. Translating the well-known inversion method (typically used on a Real RAM [8]) to our bounded precision model gives the following result.

Theorem 2. *On a RAM with word size $w = \Omega(\log\log(1/p))$, a geometric random variate* $\mathrm{Geo}(p)$ *with parameter $p \in (0,1)$ can be sampled in expected runtime $\mathcal{O}(1 + \log(1/p)\operatorname{poly}\log\log(1/p)/w)$.*

To the best of our knowledge, this observation is not discussed in the literature so far. However, as the following Theorem 3 is strictly stronger, we defer the presentation of the inversion method and Theorem 2 to Appendix A. It not only applies to geometric distributions, but to all distributions where the inverse of the cumulative distribution is efficiently computable on a Word RAM. The assumption on w is needed to handle pointers to an array as large as the expected output size in constant time. This result is independent of the rest of the paper and demonstrates that the classical inversion method *does not give an optimal runtime* matching Theorem 1, since this algorithm, as well as many other approaches, *does not avoid taking logarithms*. Note that it is a long-standing open problem in analytic number theory and computational complexity whether the logarithm can be computed in linear time.

Our aim is a Word RAM algorithm which returns the exact geometric distribution in optimal runtime. In Section 3 we give a simple algorithm for this and prove the following theorem. Note that our algorithm also works for *bitstreams* p, see Section 2.

Theorem 3. *On a RAM with word size $w = \Omega(\log\log(1/p))$, a geometric random variate* $\mathrm{Geo}(p)$ *with parameter $p \in (0,1)$ can be sampled in expected runtime $\mathcal{O}(1 + \log(1/p)/w)$, which is optimal.*

Observe that, as a sample of a geometric random variate can be arbitrarily large, the aforementioned sampling algorithm cannot work in bounded worst-case time or space. Also note that on a parallel machine with P Word RAM processors the runtime decreases to $\mathcal{O}(1 + \log(1/p)/(wP))$.

Generating bounded random geometric distributions. In Appendix D we extend this to sampling bounded geometric random distributions $\mathrm{Geo}(p, n)$ (for a definition of the considered probability distributions see Section 2) and observe the following lower bound (as a corollary of Lemma 8).

Theorem 4. *On a RAM with word size w, any algorithm sampling a bounded geometric random variate $\mathrm{Geo}(p,n) = \min\{n, \mathrm{Geo}(p)\}$ with parameters $n \in \mathbb{N}$ and $p \in (0,1)$ needs at least expected runtime $\Omega(1 + \log(\min\{1/p, n\})/w)$.*

We present an algorithm which achieves this optimal runtime bound and prove the following theorem.

Theorem 5. *On a RAM with word size $w = \Omega(\log\log(1/p))$, a bounded geometric random variate $\mathrm{Geo}(p,n) = \min\{n, \mathrm{Geo}(p)\}$ with parameters $n \in \mathbb{N}$ and $p \in (0,1)$ can be sampled in expected runtime $\mathcal{O}(1 + \log(\min\{1/p, n\})/w)$, which is optimal.*

If p is a rational number with numerator and denominator fitting in $\mathcal{O}(1)$ words, then this algorithm needs $\mathcal{O}(n)$ space in the worst case.

If p is a bitstream, we cannot bound the worst-case space usage of a sampling algorithm for $\mathrm{Geo}(n,p)$ in general. However, if p is a rational with numerator and denominator fitting in a constant number of words of the Word RAM, Theorem 5 shows that this is indeed possible.

Random graph generation. We believe our new exact and efficient sampling algorithms for bounded and unbounded geometric distributions are of independent interest, but also present in Section 4 one particular application, which is the generation of random graphs. For generating graphs with n vertices it is natural to assume $w = \Omega(\log n)$.

Theorem 6. *On a RAM with word size $w = \Omega(\log n)$, the random graph $\mathcal{G}(n,p)$ can be sampled in expected time $\Theta(n+m)$, where $m = \Theta(pn^2)$ is the expected number of edges. This is optimal if $w = \mathcal{O}(\log n)$. If p is a rational number with numerator and denominator fitting in $\mathcal{O}(1)$ words, then the worst-case space complexity of the algorithm is asymptotically equivalent to the size of the output graph, which is optimal.*

A similar algorithm achieves optimal runtime for the more general Chung-Lu random graphs $\mathcal{G}(n,W)$ [5], generating a random graph with a given sequence of expected degrees.

Theorem 7. *Let $W = (W_1, \ldots, W_n)$ be rationals with common denominator, where each numerator and the common denominator fit in $\mathcal{O}(1)$ words. Then on a RAM with word size $w = \Omega(\log n)$, the random graph $\mathcal{G}(n,W)$ can be sampled in expected time $\Theta(n+m)$, where $m = \Theta(\sum_{i=1}^{n} W_i)$ is the expected number of edges. This is optimal if $w = \mathcal{O}(\log n)$. The worst-case space complexity of the algorithm is asymptotically equivalent to the size of the output graph, which is optimal.*

Both theorems follow from plugging our algorithm for sampling geometric random variables into the best algorithm known for sampling the respective graph class.

2 Preliminaries

Machine models. We discuss two variants of random access machines (RAMs). Both are abstract computational machine models with an arbitrary number of

registers that can be indirectly addressed. In the classic RAM, each register can contain an arbitrarily large natural number $\mathbb{N}_0$ and all basic mathematical functions can be performed in constant time.

The two models relevant for this paper are the Real RAM, which allows computation even with real numbers, and the Word RAM, which only allows computation with bounded precision. In addition to the standard definitions, we assume that a uniform random number can be sampled in constant time. As we are dealing with randomized algorithms and a sample of a geometric random variate can be arbitrarily large, we also allow potentially unbounded space usage[1].

The *Real RAM* is the main model of computability in computational geometry and is also used in numerical analysis. Here, each register can contain a real number in the mathematical sense. All basic mathematical functions including the logarithm of a real number can be computed in constant time. The disadvantage of the model is that real numbers are infinite objects and all physical computers can only handle finite portions of these objects.

The *Word RAM* is a more realistic model of computation. It is parameterized by a parameter w which determines the word length. The registers are called words and contain integers in the range $\{0, \dots, 2^w - 1\}$. The execution of basic arithmetic instructions on words takes constant time; our algorithms only need constant time addition, subtraction and comparison, as well as constant time generation of random words. Long integers are represented by a string of words. Floating point numbers are represented by an exponent (a string of words of some length k) and a mantissa (a string of words of some length ℓ). Addition and multiplication can then be done in time $\mathcal{O}(\text{poly}(\ell, k))$ and with error $2^{-w\ell}$. Note that the Word RAM offers an intrinsic parallelism where, in constant time, an operation on w bits can be performed in parallel.

Random graph models. In the *Erdős-Rényi* [9] random graph model $G(n, p)$, each edge of an n vertex graph is independently present with probability p. This yields a binomial degree distribution and approaches a Poisson distribution in the limit. As many real-world networks have power-law degree distributions, we also study inhomogenous random graphs. We consider *Chung-Lu* [5] graphs $G(n, W)$ with n vertices and weights $W = (W_1, W_2, \dots, W_n) \in \mathbb{R}^n_{\geqslant 0}$. In this model, an edge between two vertices i and j is independently present with probability $p_{i,j} := \min\{W_i W_j / \sum_k W_k, 1\}$. For sufficiently large graphs, the expected degree of i converges to W_i. The related definitions of generalized random graph [25] with $p_{ij} = W_i W_j / (\sum_k W_k + W_i W_j)$ and Norros-Reittu random graphs [22] with $p_{i,j} = 1 - \exp(-W_i W_j / \sum_k W_k)$ can be handled in a similar way. However, we will focus on Chung-Lu random graphs.

Probability distributions. Let $p \in (0, 1)$. The *Bernoulli* distribution $\text{Ber}(p)$ takes values in $\{0, 1\}$ such that $\Pr[\text{Ber}(p) = 1] = 1 - \Pr[\text{Ber}(p) = 0] = p$. The *geometric* distribution $\text{Geo}(p)$ takes values in $\mathbb{N}_0$ such that for any $i \in \mathbb{N}_0$, we have $\Pr[\text{Geo}(p) = i] = p(1-p)^i$. For $n \in \mathbb{N}_0$, we define the *bounded geometric*

[1] We assume that accessing the i-th memory cell costs $\mathcal{O}(1)$ although it might make more sense to assume cost proportional to the length of a pointer to i (which is $\Theta(1 + \log(i)/w)$) or larger. However, our results remain valid as long as this cost is $\mathcal{O}((2-\varepsilon)^{i/2^w})$ for some $\varepsilon > 0$.

distribution $\mathrm{Geo}(p,n)$ to be $\min\{n, \mathrm{Geo}(p)\}$. This means that $\mathrm{Geo}(p,n)$ takes values in $\{0,\dots,n\}$ such that for any $i \in \mathbb{N}_0$, $i < n$, we have $\Pr[\mathrm{Geo}(p,n) = i] = p(1-p)^i$, and $\Pr[\mathrm{Geo}(p,n) = n] = (1-p)^n$. The *uniform* distribution $\mathrm{Uni}[0,1]$ takes values in $[0,1]$ with uniform probability. For $n \in \mathbb{N}$, we define the uniform distribution $\mathrm{Uni}(n)$ to be the uniform distribution over $\{0,\dots,n-1\}$.

Input model. We assume the input p to be given in the following form: We are given a number $k \in \mathbb{N}_0$ such that $2^{-k} \geqslant p > c2^{-k}$ for some fixed constant[2] $c > 0$. Moreover, for any $i \in \mathbb{N}$ we are able to compute a number $p_i \leqslant 1$ such that $|p_i - 2^k p| \leqslant 2^{-i}$. We can assume that p_i has at most $i+1$ bits (otherwise take the first $i+1$ bits of p_{i+1}, which are a 2^{-i}-approximation of $2^k p$). Since we assumed $w = \Omega(\log\log(1/p))$, k fits into $\mathcal{O}(1)$ words; this resembles the usual assumption that we can compute with numbers as large as the input/output size in constant time. Furthermore, we want to assume that p_i can be computed in time $\mathrm{poly}(i)$. This means that p can be approximated efficiently. However, it is sufficient even if the runtime is $\mathcal{O}((2-\varepsilon)^i)$ for some constant $\varepsilon > 0$. All numbers other than the input parameter p will be encoded as simple strings of words or floating point numbers, as discussed in the paragraph "machine models".

Notations. The base of all logarithms is 2. For integer division we use $a \operatorname{div} b := \lfloor a/b \rfloor$ for $a, b \in \mathbb{Z}$. We typically use x_i to denote the i-th bit (approximation) of x. We denote the set $\{1,\dots,n\}$ by $[n]$.

3 Sampling Geometric Random Variates

In this section we show a Word RAM algorithm for generating a geometric random variate $\mathrm{Geo}(p)$ in optimal expected runtime. We assume that the parameter p is given by a bitstream, i.e., we are given $k \in \mathbb{N}_0$ and can approximate p_* such that $p = 2^{-k} p_*$ and $c < p_* \leqslant 1$ for some $c > 0$. Approximating p_* with precision i needs time $\mathcal{O}((2-\varepsilon)^i)$ for some $\varepsilon > 0$. We first prove that the expected output size is $\Theta(\log(1/p))$, which gives a lower bound of $\Omega(1 + \log(1/p)/w)$ for the expected runtime of any algorithm sampling $\mathrm{Geo}(p)$ on the Word RAM as (at most) w bits can be processed in parallel.

Lemma 1. *For any $p \in (0,1)$, we have $\mathbb{E}[\log(1+\mathrm{Geo}(p))] = \Theta(\log(1/p))$, where the lower bound holds for $1/p$ large enough.*

The proof of Lemma 1 can be found in Appendix C. We now present an algorithm achieving this optimal expected runtime. The main trick is that we split up $\mathrm{Geo}(p)$ into $\mathrm{Geo}(p) \operatorname{div} 2^k$ and $\mathrm{Geo}(p) \bmod 2^k$. It is easy to see that both parts are independent random variables. Now, $\mathrm{Geo}(p) \operatorname{div} 2^k$ has constant expected value, so we can iteratively check whether it equals $0, 1, 2, \dots$. On the other hand, $\mathrm{Geo}(p) \bmod 2^k$ is sufficiently well approximated by the uniform distribution over $\{0,\dots,2^k-1\}$; the rejection method suffices for fast sampling. These ideas are brought together in Algorithm 1.

[2] One could set $c = 1/2$, but our results hold more generally, in the case where we cannot compute such a good approximation of p.

Algorithm 1 GenGeo(p) samples Geo(p) given a bitstream $p = 2^{-k}p_*$.

```
D ← 0
while Ber((1 − p)^{2^k}) do
    D ← D + 1
repeat
    M ←(lazy)— Uni(2^k)
until Ber((1 − p)^M)
fill up M with random bits
return 2^k D + M
```

Here, D represents $\mathrm{Geo}(p) \operatorname{div} 2^k$, initialized to 0. It is increased by 1 as long as a Bernoulli random variate $\mathrm{Ber}((1-p)^{2^k})$ turns out to be 1. Then M, corresponding to $\mathrm{Geo}(p) \bmod 2^k$, is chosen uniformly from the interval $\{0, \ldots, 2^k - 1\}$, but rejected with probability $(1-p)^M$. We sample M lazily, i.e., a bit of M is sampled only if needed by the test $\mathrm{Ber}((1-p)^M)$. After we leave the loop, M is filled up with random bits, so that we return the same value as if we had sampled M completely inside of the second loop. The result is, naturally, $2^k D + M$.

We will next discuss correctness of this algorithm, describe the details of how to implement it efficiently, and analyze its runtime. We postpone the issue of how to sample $\mathrm{Ber}((1-p)^n)$ to the end of this section. For the moment we will just assume that this can be done in expected constant time, looking at the first expected constant many bits of p and n.

Correctness. Let $n \geqslant 0$. The probability of outputting $n = 2^k D + M$ should be $p(1-p)^n$, i.e., it should be proportional to $(1-p)^n$. Following the algorithm step by step we see that the probability is

$$\underbrace{\left((1-p)^{2^k}\right)^D \cdot \left(1 - (1-p)^{2^k}\right)}_{\text{first loop}} \cdot \underbrace{\sum_{t \geqslant 0} \left(1 - \sum_{i=0}^{2^k-1} 2^{-k}(1-p)^i\right)^t 2^{-k}(1-p)^M}_{\text{second loop}},$$

where t is the number of iterations of the second loop; note that $2^{-k}(1-p)^i$ is the probability of outputting i in the first iteration of the second loop, so that $\sum_{i=0}^{2^k-1} 2^{-k}(1-p)^i$ is the probability of leaving the second loop after the first iteration. Collecting the factors dependent on D and M we see that this probability is proportional to $(1-p)^{2^k D + M} = (1-p)^n$, showing correctness of the algorithm.

Implementation Details. We encode D and M as strings of words, the easiest way of representing large integers. Then incrementing the counter D can be done in amortized constant time. Also, computing $2^k D + M$ can be done by shifting D by k and overwriting the last k bits of $2^k D$ (which are all 0) with M (which is smaller than 2^k), which has the cost of reading the output once. Note that these operations can even be avoided, if we store D and M at the right positions in a string of words right away.

Moreover, we want to sample M uniformly in the interval $\{0, \dots, 2^k - 1\}$. This can be done by generating $\lceil k/w \rceil$ uniformly random words and truncating the first one to $k \bmod w$ bits, in case w does not divide k. Thus, this can be done in time $\mathcal{O}(1 + \log(1/p)/w)$. However, for our purposes it actually makes more sense to sample bits of M on demand: Sampling $\mathrm{Ber}((1-p)^M)$ requires an expected number of $\mathcal{O}(1)$ bits of M, so we can sample these bits on demand, storing already sampled bits, and filling up the rest of the bits of M after we leave the second loop. This has the advantage that we sample a large number of random bits exactly once, not an expected number of $\mathcal{O}(1)$ times, so that the runtime of the algorithm is more concentrated.

Runtime. We show that the expected runtime of Algorithm 1 is $\mathcal{O}(1 + \log(1/p)/w)$. Again, assume that we can sample $\mathrm{Ber}((1-p)^n)$ in expected constant time. By the last section, incrementing the counter D can be done in amortized constant time, and we only need an expected constant number of bits of M during the second loop, after which we fill up M with random bits in time $\mathcal{O}(1+\log(1/p)/w)$. Hence, if we show that the two loops run in expected constant time, then Algorithm 1 runs in expected time $\mathcal{O}(1 + \log(1/p)/w)$.

We consider the probabilities of dropping out of the two loops. Since $2^{-k} \geqslant p > c2^{-k}$, for the first loop this is

$$1 - (1-p)^{2^k} \geqslant 1 - (1-p)^{c/p} \geqslant 1 - e^{-c}, \tag{1}$$

so we have constant probability to drop out of this loop in every iteration. Moreover, the second loop terminates immediately if $k = 0$; otherwise we have

$$(1-p)^M \geqslant (1-p)^{2^k} \geqslant (1-2^{-k})^{2^k} \geqslant (1-1/2)^2 = 1/4, \tag{2}$$

so for the second loop we also have constant probability of dropping out.

To show that each loop runs in expected constant time, let T be a random variable denoting the number of iterations of the loop; note that $\mathbb{E}[T] = \mathcal{O}(1)$, since the probability of dropping out of each loop is $\Omega(1)$. Furthermore, let X_i be the runtime of the i-th iteration of the loop; note that by assumption we can sample $\mathrm{Ber}((1-p)^n)$ in expected constant time, so that $\mathbb{E}[X_i \mid T \geqslant i] = \mathcal{O}(1)$. The total runtime of the loop is $X_1 + \ldots + X_T$. Thus, the following lemma shows that the expected runtime of the loop is $\mathcal{O}(1)$. This finishes the proof of Theorem 3, aside from sampling $\mathrm{Ber}((1-p)^n)$.

Lemma 2. *Let T be a random variable with values in $\mathbb{N}_0$ and X_i, $i \in \mathbb{N}$, be random variables with values in $\mathbb{R}$; we assume no independence. Let $\alpha \in \mathbb{R}$ with $\mathbb{E}[X_i \mid T \geqslant i] \leqslant \alpha$ for all $i \in \mathbb{N}$. Then we have $\mathbb{E}[X_1 + \ldots + X_T] \leqslant \alpha \cdot \mathbb{E}[T]$.*

We remark that the above lemma is an easy special case of Wald's equation.

Note that the only points where this algorithm is using the Word RAM parallelism are when we fill up M and when we compute with exponents. The generation of $\mathrm{Ber}((1-p)^n)$, discussed in the remainder of this section, will use Word RAM parallelism only for working with exponents. The filling of M can be done in time $\mathcal{O}(1 + \log(1/p)/w)$ as we assumed that we can generate random words

in unit time. Also note that given P processors, each one capable of performing Word RAM operations, we can trivially further parallelize this algorithm to run in expected time $\mathcal{O}\big(1 + \frac{\log(1/p)}{wP}\big)$.

Sampling $\mathbf{Ber((1-p)^n)}$. It is left to show how to sample a Bernoulli random variable with parameter $(1-p)^n$. We can use the fact that we know k with $2^{-k} \geqslant p > c2^{-k}$ and can approximate $2^k p$ by p_i, and that $n \in \mathbb{N}$, $n \leqslant 2^k$. Note that we can easily get an approximation n_i of n of the form $|2^{-k}n - n_i| \leqslant 2^{-i}$ in the situation of Algorithm 1: In the first loop we have $n = 2^k$, then simply pick $n_i = 1$; in the second loop $n = M$ is uniform in $\{0, \dots, 2^k - 1\}$, so that we get n_i by determining (i.e. flipping) the highest i bits of n. In this situation we can show the following lemma.

Lemma 3. *Given bitstream p with $2^{-k} \geqslant p = \Omega(2^{-k})$, for $n = 2^k$ or for uniformly random n in $\{0, \dots, 2^k - 1\}$ we can sample $\mathrm{Ber}((1-p)^n)$ in expected constant time.*

We discuss in Appendix B that the only thing we need to efficiently sample $\mathrm{Ber}(q)$ is to be able to compute an approximation q_i of q with $|q - q_i| \leqslant 2^{-i}$ in time $\mathcal{O}((2-\varepsilon)^i)$. To get such an approximation for $(1-p)^n$, we make use of the binomial theorem $(1-p)^n = \sum_{j=0}^{n} \binom{n}{j}(-p)^j$. Noting that $\binom{n}{j} \leqslant \frac{n^j}{j!}$ and $n \leqslant 1/p$, we see that the j-th summand is absolutely bounded by $1/j!$. Moreover, the absolute value of the summands is monotonically decreasing in j, and their sign is $(-1)^j$, implying

$$\left| \sum_{j=i+2}^{n} \binom{n}{j}(-p)^j \right| \leqslant 1/(i+2)! \leqslant 2^{-i-1}. \tag{3}$$

Thus, by summing up only the first $i+2$ summands we get a good approximation of $(1-p)^n$.

Moreover, we have

$$\sum_{j=0}^{i+1} \binom{n}{j}(-p)^j = \frac{1}{(i+1)!} \sum_{j=0}^{i+1} (-p)^j \Bigg(\prod_{h=j+1}^{i+1} h \Bigg) \prod_{h=0}^{j-1} (n-h). \tag{4}$$

We will compute the right-hand side of this with working precision r. This means that we work with floating point numbers, with an exact exponent encoded by a string of words, and a mantissa which is a string of $\lceil r/w \rceil$ words. We get p and n up to working precision r by plugging in $2^{-k}p_r$ and $2^k n_r$. Then we calculate the numerator and denominator of the right-hand side independently with working precision r. Note that adding or multiplying the floating point numbers takes time $\mathcal{O}(\mathrm{poly}(r))$ for adding/multiplying the mantissas (even using the school method for multiplication is fine for this), and $\mathcal{O}(1 + \log(i))$ for subtracting/adding the exponents, as all exponents in equation (4) are absolutely bounded by $\mathcal{O}(\mathrm{poly}(i) \cdot k)$ and k fits in $\mathcal{O}(1)$ words.

Regarding runtime, noting that there are $\mathcal{O}(\mathrm{poly}(i))$ operations to carry out in computing the right-hand side of equation (4), we see that we can compute the

latter with working precision r in time $\mathcal{O}(\text{poly}(r, i))$. If we choose r large enough so that this yields an approximation of equation (4) with absolute error at most 2^{-i-1}, then combined with the error analysis from using only the first $i+2$ terms (equation (3)), we get a runtime of $\mathcal{O}(\text{poly}(r, i))$ to compute an approximation of $(1-p)^n$ with absolute error 2^{-i}. Now, as long as we can choose $r = \text{poly}(i)$, this runtime is small enough to use the result from Appendix B, since we only needed an approximation of $(1-p)^n$ with absolute error 2^{-i} in time $\mathcal{O}((2-\varepsilon)^i)$ for some $\varepsilon > 0$. Under this assumption on r, we are done proving Lemma 3. The following lemma shows that $r = \text{poly}(i)$ is indeed sufficient. Its proof can be found in the Appendix C.

Lemma 4. *The absolute error of computing equation* (4) *with working precision* $r = i + \alpha(1 + \log(i))$ *is at most* 2^{-i-1}*, for a large enough constant* α*.*

4 Generating Random Graphs

In this section we show that Erdős-Rényi and Chung-Lu random graphs can be efficiently generated. For this we simply take the efficient generation on Real RAMs from [1, 20] and replace the generation of bounded geometric variables by our algorithm from the last section. In the following we discuss why this is sufficient and leads to the runtimes claimed in Theorems 6 and 7.

Consider the original efficient generation algorithm of Erdős-Rényi random graphs described in [1], which is essentially the following. For each vertex $u \in [n]$ we want to sample its neighbors $v \in [u-1]$ in decreasing order. Defining $v_0 := u$, the first neighbor v_1 of u is distributed as $v_1 \sim v_0 - 1 - \text{Geo}(p, v_0 - 1)$, where the event $v_1 = 0$ represents that u has no neighbor. Then the next neighbor is distributed as $v_2 \sim v_1 - 1 - \text{Geo}(p, v_1 - 1)$ and so on. Sampling the graph in this way, we use $m + n$ bounded geometric variables, where m is the number of edges in the final graph (which is a random variable).

In this algorithm we have to cope with indices of vertices, thus, it is natural to assume $w = \Omega(\log n)$. Under this assumption, all single operations of the original algorithm can be performed in worst-case constant time on a Word RAM, except for the generation of bounded geometric variables $\text{Geo}(p, k)$, with $k \leqslant n$. The latter, however, can be done in expected time $\mathcal{O}(1 + \log(\min\{n, 1/p\})/w) = \mathcal{O}(1)$ using our algorithm from Theorem 5. Hence, the expected runtime of the modified algorithm (with replaced sampling of bounded geometric variables) should be the same as that of the original algorithm. To prove this, consider the runtime the modified algorithm spends on sampling bounded geometric variables. This random variable can be written as $X_1 + \ldots + X_T$, where T is a random variable denoting the number of bounded geometric variables sampled by the algorithm, and X_i is the time spent on sampling the i-th such variable. Note that $\mathbb{E}[X_i \mid T \geqslant i] = \mathcal{O}(1)$. Thus, by Lemma 2 we can bound $\mathbb{E}[X_1 + \ldots + X_T]$ by $\mathcal{O}(\mathbb{E}[T])$. Since the original algorithm spends time $\Omega(T)$, the total expected runtime of the modified algorithm is asymptotically the same as the expected runtime of the original algorithm, namely $\mathcal{O}(n + pn^2)$. This runtime is optimal, as writing down a graph takes time $\Omega(n + m)$ (each index needs $\Theta(1)$ words; this depends, however, on the

representation of the graph). Noting that the space requirements of the algorithm are met by Theorem 5, this proves Theorem 6.

A similar result applies to the more general Chung-Lu random graphs $\mathcal{G}(n, W)$. Again we assume $w = \Omega(\log n)$. Let us further assume, for simplicity, that all given weights W_u, $u \in V$ are rational numbers with the same denominator, with each numerator and the common denominator fitting in $\mathcal{O}(1)$ words. In this case, the sum $S = \sum_{u \in V} W_u$ has the same denominator as all W_u and numerator bounded by n times the numerator of the largest W_u. Since $w = \Omega(\log n)$, the numerator of S fits in $\mathcal{O}(1)$ more words than used for the largest W_u. Hence, numerator and denominator of S fit in $\mathcal{O}(1)$ words and can be computed in $\mathcal{O}(n)$ time. Moreover, the edge probabilities $p_{u,v} = \min\{W_u W_v / S, 1\}$ are also rationals with numerator and denominator fitting in $\mathcal{O}(1)$ words that can be computed in constant time if S is available.

Carefully examining the efficient sampling algorithm for Chung-Lu random graphs, Algorithm 2 of Miller and Hagberg [20], we see that now every step can be performed in the same deterministic time bound as on a Real RAM, except for the generation of bounded geometric variables and Bernoulli variables. Note that for any $p \in (0, 1)$ we have $\mathrm{Ber}(p) \sim \mathrm{Geo}(1 - p, 1)$, so Theorem 5 shows that the bounded geometric as well as the Bernoulli random variables can be sampled in expected constant time and bounded space (for $w = \Omega(\log n)$). Thus, we can bound the expected runtime of the modified generation for Chung-Lu graphs analogously to the Erdős-Rényi case, proving Theorem 7.

5 Conclusions and Future Work

We have presented new exact algorithms which can sample $\mathrm{Geo}(p)$ and $\min\{n, \mathrm{Geo}(p)\}$ in optimal time and space on a Word RAM. It remains open to find similar algorithms for other non-uniform random variates besides exponential and normal distributions. Moreover, it would be interesting to see whether our theoretically optimal algorithms are also practical, e.g., for generating very large geometric random variates for cryptographic applications [13].

Regarding our new exact algorithm for sampling Erdős-Rényi and Chung-Lu random graphs in optimal time and space on a Word RAM, we believe that similar results can be proven for the more general case where the weights W_u are given as bitstreams.

Bibliography

[1] V. Batagelj and U. Brandes. Efficient generation of large random networks. *Phys. Rev. E*, 71:036113, 2005.

[2] A. Blanca and M. Mihail. Efficient generation ε-close to $G(n, p)$ and generalizations. `arxiv.org/abs/1204.5834`, 2012.

[3] J. K. Blitzstein and P. Diaconis. A sequential importance sampling algorithm for generating random graphs with prescribed degrees. *Internet Mathematics*, 6:489–522, 2011.

[4] R. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2010.

[5] F. Chung and L. Lu. Connected components in random graphs with given expected degree sequences. *Annals of Combinatorics*, 6:125–145, 2002.

[6] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *J. Comput. Syst. Sci.*, 7:354–375, 1973.

[7] A. De, P. P. Kurur, C. Saha, and R. Saptharishi. Fast integer multiplication using modular arithmetic. In *40th Symp. Theory of Computing (STOC)*, pp. 499–506, 2008.

[8] L. Devroye. *Non-uniform random variate generation*. Springer-Verlag, 1986.

[9] P. Erdős and A. Rényi. On random graphs. *Publ Math Debrecen*, 6:290–297, 1959.

[10] P. Flajolet and N. Saheb. The complexity of generating an exponentially distributed variate. *J. Algorithms*, 7:463–488, 1986.

[11] P. Flajolet, M. Pelletier, and M. Soria. On Buffon machines and numbers. In *22nd Symp. Discrete Algorithms (SODA)*, pp. 172–183, 2011.

[12] M. Fürer. Faster integer multiplication. *SIAM J. Comput.*, 39:979–1005, 2009.

[13] A. Ghosh, T. Roughgarden, and M. Sundararajan. Universally utility-maximizing privacy mechanisms. In *41st Symp. Theory of Computing (STOC)*, pp. 351–360, 2009.

[14] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *7th Conf. Python in Science*, pp. 11–15, 2008.

[15] C. Karney. Random number library, version 1.5. `sourceforge.net/projects/randomlib`, 2012.

[16] M. J. Keeling and K. T. Eames. Networks and epidemic models. *Journal of The Royal Society Interface*, 2:295–307, 2005.

[17] J. H. Kim and V. H. Vu. Generating random regular graphs. In *35th Symp. Theory of Computing (STOC)*, pp. 213–222. ACM, 2003.

[18] D. E. Knuth and A. C.-C. Yao. The complexity of nonuniform random number generation. In J. F. Traub, editor, *Symposium on Algorithms and Complexity: New Directions and Recent Results*, pp. 357–428, 1976.

[19] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[20] J. C. Miller and A. A. Hagberg. Efficient generation of networks with given expected degrees. In *8th Intl. Workshop Algorithms & Models for the Web Graph (WAW)*, pp. 115–126, 2011.

[21] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298:824–827, 2002.

[22] I. Norros and H. Reittu. On a conditionally Poissonian graph process. *Advances in Applied Probability*, 38:59–75, 2006.

[23] J. Ray, A. Pinar, and C. Seshadhri. Are we there yet? When to stop a Markov chain while generating random graphs. In *9th Intl. Workshop Algorithms & Models for the Web Graph (WAW)*, pp. 153–164, 2012.

[24] R. Solovay and V. Strassen. A fast Monte-Carlo test for primality. *SIAM J. Comput.*, 6:84–85, 1977.

[25] R. van der Hofstad. Random graphs and complex networks. Available at `www.win.tue.nl/~rhofstad/NotesRGCN.pdf`, 2009.

[26] J. von Neumann. Various techniques used in connection with random digits. In A. S. Householder et al., editors, *The Monte Carlo Method*, Vol. 12, pp. 36–38. National Bureau of Standards, Applied Mathematics Series, 1951.

A Multi-precision Approach

In this appendix we consider the question of why exact and efficient sampling of non-uniform random variates on bounded precision machines should be possible at all.

To illustrate this, let us first consider the following situation on a Real RAM: Let $X = X_p$ be a random variable with some parameter p, that we omit in the following discussion as a subscript. Let X have cumulative distribution function $F(x) = \Pr[X \leqslant x]$ and let U be a uniform real in $[0, 1]$. Then X can be sampled as $X \sim F^{-1}(U)$; this is called the inversion method (see, e.g., Devroye [8]). On a Real RAM this method is applicable as long as F^{-1} is efficiently computable; the uniform variable U can be sampled in unit time by assumption. It is easy to see that for a geometric random variable $X \sim \mathrm{Geo}(p)$ we have $F^{-1}(x) = \lfloor \log_{1-p}(1-x) \rfloor$. Thus, since a logarithm of a real number can be computed in unit time by assumption, $\mathrm{Geo}(p)$ can be sampled in unit time on a Real RAM, i.e., in time that is independent of the parameter p. Note that typical implementations for sampling geometric random variables use the above algorithm (e.g. in C++11), but with the usual floating point precision, although the algorithm is only exact if used with infinite precision.

The situation changes when we consider a bounded precision machine like the Word RAM instead. We focus on the case where $F^{-1}(x)$ can be written as $\lfloor g(x) \rfloor$, with g being a smooth and monotonically increasing function; note that this is the case for geometric random variables. Suppose that we sample only an n-bit approximation $\tilde{U}$ of U, i.e., we sample the first n bits of U, so that $\tilde{U} \leqslant U < \tilde{U} + 2^{-n}$. Suppose further that we compute an n-bit approximation $\tilde{G}$ of $g(\tilde{U})$, such that $\tilde{G} \leqslant g(\tilde{U}) \leqslant (1+2^{-n})\tilde{G}$. $\lfloor \tilde{G} \rfloor$ is not exactly distributed as X is; we make a certain amount of error. So let us also compute an n-bit approximation $\tilde{G}'$ of $g(\tilde{U} + 2^{-n})$, such that $\tilde{G}' \geqslant g(\tilde{U} + 2^{-n}) \geqslant (1 - 2^{-n})g(\tilde{U} + 2^{-n})$. Then $\tilde{G} \leqslant g(U) \leqslant \tilde{G}'$. Thus, if $\tilde{G}$ and $\tilde{G}'$ lie in the same interval $[k, k+1)$, for some $k \in \mathbb{N}$, we have correctly identified $F^{-1}(U) = \lfloor g(U) \rfloor$ as the number k. Of course, it may be that both values do not lie in the same interval. In this case we can increase n (e.g., double it) and repeat the process, until at some point we have approximated $g(U)$ well enough to identify its integral part.

This method terminates with probability 1: Since $g(U) \notin \mathbb{N}$ with probability 1, an approximation of $g(U)$ with (strictly positive) additive error $\min_{m \in \mathbb{N}} |g(U) - m|$ suffices. Since $\tilde{G}, \tilde{G}' \to g(U)$ for $n \to \infty$ (as g is smooth), we reach the necessary precision after a finite number of incrementations of n. Thus, we have described an exact algorithm for sampling X.

Let us now turn to the question of whether the above method is also efficient. More precisely, let us bound its expected runtime (asymptotically in terms of p). We will first bound the final n in terms of $g(U)$, $g'(U)$, and $\delta := \min_{m \in \mathbb{N}} |g(U) - m|$.

Lemma 5. *Let $U \in [0, 1]$ with $g(U) \notin \mathbb{N}$ be fixed and consider the final n of the sampling algorithm. Then $n = \mathcal{O}(1 + \log(g(U) + g'(U)) + \log(1/\delta))$.*

Proof. Note that the smoothness of g implies $g(\tilde{U} + 2^{-n}) - g(\tilde{U}) = \mathcal{O}(2^{-n} g'(U))$. This yields $g(\tilde{U}) = g(U) - \mathcal{O}(2^{-n} g'(U))$, and since $(1+2^{-n})\tilde{G} \geqslant g(\tilde{U})$, we have $(1+$

$2^{-n})\tilde{G}+\mathcal{O}(2^{-n}g'(U)) \geqslant g(U)$. Using $\tilde{G} \leqslant g(U)$, we get $\tilde{G} \geqslant g(U)-\mathcal{O}(2^{-n}(g(U)+g'(U)))$. Analogously, we have $\tilde{G}' \leqslant g(U)+\mathcal{O}(2^{-n}(g(U)+g'(U)))$. Hence, it suffices to have $n = c+\log(g(U)+g'(U))+\log(1/\delta)$, for c large enough, to have $\tilde{G} \geqslant g(U)-\delta$ and $\tilde{G}' < g(U)+\delta$, i.e., the method terminates for some $n = \mathcal{O}(1+\log(g(U)+g'(U))+\log(1/\delta))$. □

Now that we have bounded n, let the runtime of one iteration be $f(n) = f_w(n)$. We have $f(n) = \Omega(n)$, since for sampling the first n bits of U we need runtime $\Theta(n/w)$. Thus, doubling n after each iteration, the runtime of the algorithm is bounded by the runtime of the last iteration. Moreover, assume that f is at most polynomial in n, i.e., $f(n) \leqslant \mathcal{O}(n^c)$ for some $c > 0$. Then plugging in the above bound on n, we get a runtime of at most

$$\mathcal{O}(1+f(\log(g(U)+g'(U)))+f(\log(1/\delta))),$$

since for polynomial $q(x)$ we have $q(x+y) = \mathcal{O}(q(x)+q(y))$. Now, if the expected value of this term is small, then the above method is efficient.

We calculate this expected value for the geometric random variable $\mathrm{Geo}(p)$.

Lemma 6. *For geometric random variables, i.e., for $g(x) = \log_{1-p}(1-x)$, we have in the situation of this section*

$$\mathbb{E}[1+f(\log(g(U)+g'(U)))+f(\log(1/\delta))] = \mathcal{O}(1+f(\log(1/p))).$$

Proof. In this case, $g(x) = \log_{1-p}(1-x)$ and $g'(x) = -1/((1-x)\log(1-p))$. Since $1/(1-x) \geqslant -\log(1-x)$ holds for all $x \in [0,1]$, we have $g(U)+g'(U) = \Theta(g'(U))$. Because $\log(1-p)$ is asymptotically equal to $-p$, we even have $\log(g(U)+g'(U)) = \mathcal{O}(1)+\log(1/(1-U))+\log(1/p)$, so we can bound the runtime of the algorithm by $\mathcal{O}(1+f(\log(1/(1-U)))+f(\log(1/p))+f(\log(1/\delta)))$. We first analyze δ. Since $\Pr[\delta \leqslant x] = g^{-1}(x)+\sum_{k\in\mathbb{N}} g^{-1}(k+x)-g^{-1}(k-x)$, the density function of δ is $1/g'(g^{-1}(x))+\sum_{k\in\mathbb{N}} 1/g'(g^{-1}(k+x))-1/g'(g^{-1}(k-x))$. Since, furthermore, g' is monotonically increasing we have $1/g'(g^{-1}(k+x))-1/g'(g^{-1}(k-x)) \leqslant 0$, so the density is bounded by $1/g'(g^{-1}(\delta)) \leqslant 1/g'(0) = -1/\log(1-p) \leqslant 1/p$. Thus, δ is a random variable of the following form (with $q = 1/p$): Let Y be a random variable with values in $[0,1]$ and density bounded from above by $q \geqslant 1$. Then we show that $\mathbb{E}[f(\log(1/Y))] \leqslant \mathcal{O}(f(\log(q))+1)$. The expected value is maximized if Y is uniform in $[0,1/q]$, and then it is equal to $\int_0^{1/q} q\cdot f(\log(1/y))dy$. Setting $x = \frac{1}{qy}$, this is equal to $\int_1^\infty f(\log(xq))x^{-2}dx \leqslant \int_1^\infty f(\log(q))x^{-2}dx + \int_1^\infty \log^c(x)x^{-2}dx$ using our assumption on f. This yields the desired bound. Note that this bound yields not only an upper bound of $\mathcal{O}(f(\log(1/p))+1)$ for the expected value of $f(\log(1/\delta))$, but also an upper bound of $\mathcal{O}(1)$ for the expected value of $f(\log(1/(1-U)))$. □

Hence, the geometric random variable $\mathrm{Geo}(p)$ can be sampled in time $\mathcal{O}(1+f(\log(1/p)))$. Note that $f(n)$, the runtime of one iteration, is dominated by the time for approximating a logarithm of an n-bit number up to precision 2^{-n}. This can be done in time $\mathcal{O}(M(n)\log(n))$ (see, e.g., [4]), where $M(n)$ is the time to multiply two n-bit numbers. By using fast Fourier transforms, $M(n) = \mathcal{O}(\frac{n}{w}\log(\frac{n}{w})\log\log(\frac{n}{w}))$ [24]. The best known bound is $M(n) =$

$\mathcal{O}(\frac{n}{w}\log(\frac{n}{w})2^{\log^*(\frac{n}{w})})$ [7, 12]. It is conjectured that $M(n) = \Omega(\frac{n}{w}\log(\frac{n}{w}))$ [24]. This gives $f(n) = \mathcal{O}(1 + \frac{n}{w}\operatorname{poly}\log(n))$, which is efficient. A geometric random variable can thus be sampled in expected time $\mathcal{O}(1 + \frac{\log(1/p)}{w}\operatorname{poly}\log\log(1/p))$ which proves Theorem 2. Despite its simplicity, to the best of our knowledge this approach has not been formalized yet. Note that this approach does not, however, give linear runtime $\mathcal{O}(1 + \log(1/p)/w)$.

B Sampling Bernoulli Random Variates

For sampling geometric random variates as presented in Section 3, it is essential to efficiently sample Bernoulli random variates. As this cannot be found in standard text books, this appendix describes a simple method for generating Bernoulli random variates $\mathrm{Ber}(p)$ on a Word RAM which is closely related to Flajolet and Saheb [10].

We assume that, for any i, we can compute an approximation p_i of p such that $|p_i - p| \leqslant 2^{-i}$ in time $\mathcal{O}((2-\varepsilon)^i)$ for some fixed $\varepsilon > 0$. Under this assumption we show that $\mathrm{Ber}(p)$ can be sampled in expected time $\mathcal{O}(1)$, i.e., in runtime independent of p. Note that we can assume that p_i has at most $i+1$ bits; otherwise take the first $i+1$ bits b of p_{i+1}, which fulfill $|b-p| \leqslant 2^{-i}$, since $|p_{i+1}-p| \leqslant 2^{-i-1}$ and $|b - p_{i+1}| \leqslant 2^{-i-1}$.

Observe that we can generate $\mathrm{Ber}(p)$ by generating a uniformly random real $r \in [0,1]$ and returning 1 if $r \leqslant p$ and 0, otherwise. We now describe how this can be efficiently simulated without having to cope with the real number r at once.

Let $r \sim \mathrm{Uni}[0,1]$. We get an approximation r_i of r by sampling i random bits. Then $r_i \leqslant r < r_i + 2^{-i}$. Comparing r_i and p_i, it can happen that the intervals $[r_i, r_i + 2^{-i}]$ and $[p_i - 2^{-i}, p_i + 2^{-i}]$ are non-intersecting. In this case, $r_i < p_i$ implies $r < p$, and similarly $r_i > p_i$ implies $r > p$, so we are done. Otherwise, we can increase the precision i by 1 and repeat this process (remembering the former random choices we made for r). Note that the probability that these intervals are non-intersecting is at most $3 \cdot 2^{-i}$; there are at most 3 choices for r_i such that they intersect. Hence, the probability that we need precision at least i is at most $3 \cdot 2^{-i}$. Since we can compute p_i in time $\mathcal{O}((2-\varepsilon)^i)$, and can clearly sample r_i and compare both intervals in this time bound also, we get an expected time of at most

$$\sum_{i=1}^{\infty} \mathcal{O}((2-\varepsilon)^i) \cdot 3 \cdot 2^{-i} = \mathcal{O}(1).$$

Lemma 7. *A Bernoulli random variate* $\mathrm{Ber}(p)$ *with parameter* $p \in (0,1)$ *can be sampled in constant expected runtime on a Word RAM.*

Note that we did not use any parallelism provided by the Word RAM in this section. Regarding concentration, this has $\Pr[T > t] \leqslant t^{1+\varepsilon'}$ for some $\varepsilon' > 0$. If we can compute p_i in the lower time bound $\mathcal{O}(\operatorname{poly}(i))$ then we even have $\Pr[T > t] \leqslant 2^{-\Omega(t^{\varepsilon''})}$ for some $\varepsilon'' > 0$. It seems hard to achieve real exponential bounds, i.e., $2^{-\Omega(t)}$ for the parameters we encounter in this paper.

C Sampling Geometric Random Variates

This appendix presents all proofs omitted from Section 3.

Proof of Lemma 1. For the upper bound we can use

$$\mathbb{E}[\log(1 + \mathrm{Geo}(p))] \leqslant \log(1 + \mathbb{E}[\mathrm{Geo}(p)]) = \log(1/p).$$

For the lower bound we have

$$\begin{aligned}
\mathbb{E}[\log(1 + \mathrm{Geo}(p))] &= \sum_{i=1}^{\infty} \log(i) \cdot p(1-p)^{i-1} \\
&\geqslant \sum_{i=\lceil 1/p \rceil}^{\infty} \log(i) \cdot p(1-p)^{i-1} \\
&\geqslant p \cdot \log(1/p) \cdot \sum_{i=\lceil 1/p \rceil}^{\infty} (1-p)^{i-1} \\
&= p \cdot \log(1/p) \cdot (1-p)^{\lceil 1/p \rceil - 1} \cdot 1/p \\
&\geqslant \Omega(\log(1/p)),
\end{aligned}$$

since e.g. for $p \leqslant 1/2$ we have $(1-p)^{\lceil 1/p \rceil - 1} \geqslant (1-p)^{1/p} \geqslant (1-1/2)^2 = 1/4$. □

Proof of Lemma 2. The variable X_i is part of the sum if and only if $T \geqslant i$. Hence, we have

$$\mathbb{E}[X_1 + \ldots + X_T] = \sum_{i \geqslant 1} \mathbb{E}[X_i \mid T \geqslant i] \cdot \Pr[T \geqslant i].$$

Using $\mathbb{E}[X_i \mid T \geqslant i] \leqslant \alpha$ and the definition of $\mathbb{E}[T]$, this yields

$$\mathbb{E}[X_1 + \ldots + X_T] \leqslant \alpha \sum_{i \geqslant 1} \Pr[T \geqslant i] = \alpha \cdot \mathbb{E}[T].$$ □

Proof of Lemma 4. Consider computing a product $c = a \cdot b$ of floating point numbers. Note that we work with precision r, so already a and b should have some relative error $1 + \varepsilon_a$ and $1 + \varepsilon_b$ relative to their correct values. Then what is the relative error of c? We compute c to be the product of a and b rounded to the next floating point number with precision r, so c has relative error at most $(1 + \varepsilon_a)(1 + \varepsilon_b)(1 + \mathcal{O}(2^{-r}))$. As long as $\varepsilon_a, \varepsilon_b \leqslant 2^{-r/2}$ this product is less than $1 + \varepsilon_a + \varepsilon_b + \mathcal{O}(2^{-r})$. Thus, computing a product like p^j with working precision r we get a relative error of $1 + \mathcal{O}(j2^{-r})$, since we plugged in p with relative error at most $\mathcal{O}(2^{-r})$. This assumes $j \leqslant 2^{r/2}$. Similarly, we get a relative error of $\mathcal{O}(i2^{-r})$ for $(i+1)!$ and $\prod_{h=j+1}^{i+1} h$, assuming that $i + 1 < 2^r$ (so that each factor can be represented exactly).

The subtraction $n - h$ can be performed with relative error $1 + \mathcal{O}(i2^{-r})$ if $h \leqslant i + 1 < 2^r$, since in this case the floating point representation of h is actually exact, and the exponent of $n - h$ can drop from the exponent of n by at most

$\log(i)$. Thus, the product $\prod_{h=0}^{j-1}(n-h)$ can be computed with relative error at most $1+\mathcal{O}(i^2 2^{-r})$

In general, for an addition $c=a+b$ we cannot hope for a relative approximation of the resulting number c, at least not if a and b can have opposing signs. However, if e_a and e_b are the exponents of a and b, we can hope for an approximation relative to $2^{\max\{e_a,e_b\}}$. So assume that we computed a and b with relative error $1+\varepsilon_a$, $1+\varepsilon_b$ relative to some $2^{e'_a} \geqslant 2^{e_a}$, $2^{e'_b} \geqslant 2^{e_b}$. Then their sum will have relative error at most $1+\varepsilon_a+\varepsilon_b+\mathcal{O}(2^{-r}))$ relative to $2^{\max\{e'_a,e'_b\}}$. Observe that this is sufficient for the sum of equation (4): Each summand is bounded by $(i+1)!$, so we get approximations relative to $2^{e'} = \mathcal{O}((i+1)!)$. Since we start with a relative approximation error of $1+\mathcal{O}(i^2 2^{-r})$ for each summand, this adds up to an error of $1+\mathcal{O}(i^3 2^{-r})$ for the numerator, assuming $i^3 \leqslant 2^{r/2}$. Since this error is relative to $2^{e'} = \mathcal{O}((i+1)!)$, and the denominator is $(i+1)!$, we end up with an approximation of equation (4) with *absolute* error at most $\mathcal{O}(i^3 2^{-r})$. Thus, to get an absolute error of at most 2^{-i-1} it suffices to set $r = i+\mathcal{O}(\log(i))$ with a large enough hidden constant. Moreover, the above construction is explicit enough to allow for computing the involved constants. □

D Sampling Bounded Geometric Random Variates

For the generation of $\mathcal{G}(n,p)$ and Chung-Lu random graphs we do not necessarily need a geometric random variate, but rather a bounded geometric random variate. Generating $\mathrm{Geo}(p,n)$ in the obvious way, $\min\{n, \mathrm{Geo}(p)\}$, as defined in Section 2, takes expected time $\mathcal{O}(1+\log(1/p)/w)$. We show in this appendix how to reduce the expected runtime to $\mathcal{O}(1+\log(\min\{1/p,n\})/w)$.

Lemma 8. *For any $p \in (0,1)$ and $n \in \mathbb{N}_0$, we have $\mathbb{E}[\log(1+\mathrm{Geo}(p,n))] = \Theta(\log(\min\{1/p, n+1\}))$, where the lower bound holds for $1/p$ and n large enough.*

Proof. We have

$$\mathbb{E}[\log(1+\mathrm{Geo}(p,n))] = \log(n+1)\cdot(1-p)^n + \sum_{i=0}^{n-1} \log(i+1)\cdot p(1-p)^i.$$

Thus, for n large enough and $n \leqslant 1/p$, we have

$$\begin{aligned}\mathbb{E}[\log(1+\mathrm{Geo}(p,n))] &\geqslant \log(n+1)\cdot(1-p)^n \\ &\geqslant \log(n+1)\cdot(1-1/n)^n \\ &\geqslant \log(n+1)\cdot(1-1/2)^2 = \Omega(\log(n+1)).\end{aligned}$$

On the other hand, for $1/p$ large enough and $1/p < n$, we have

$$\begin{aligned}\mathbb{E}[\log(1+\mathrm{Geo}(p,n))] &\geqslant \sum_{i=\lfloor 1/(2p)\rfloor}^{\lfloor 1/p\rfloor} \log(i+1)\cdot p(1-p)^i \\ &\geqslant \log(1/2p)\, p \sum_{i=\lfloor 1/(2p)\rfloor}^{\lfloor 1/p\rfloor} (1-p)^i,\end{aligned}$$

In this interval, $(1-p)^i \geqslant (1-p)^{1/p} = \Omega(1)$, and the interval is of length $\Omega(1/p)$, yielding

$$\mathbb{E}[\log(1 + \mathrm{Geo}(p, n))] = \Omega(\log(1/p)).$$

For the upper bound we may use

$$\mathbb{E}[\log(1 + \mathrm{Geo}(p, n))] \leqslant \log(1 + \mathbb{E}[\mathrm{Geo}(p, n)]).$$

As $\mathrm{Geo}(p, n) = \min\{n, \mathrm{Geo}(p)\}$, we have $\mathbb{E}[\mathrm{Geo}(p, n)] \leqslant \min\{n, \mathbb{E}[\mathrm{Geo}(p)]\} = \min\{n, 1/p - 1\}$, which finishes the proof. □

In order to get the reduced runtime, we assume that we are given p as a bitstream and n as a string of words together with an exponent m with $2^m \geqslant n = \Omega(2^m)$.

Algorithm 2 BoundedGeo(p, n) samples $\mathrm{Geo}(p, n)$ given bitstream $p = 2^{-k}p_*$, number n, and exponent m with $2^m \geqslant n = \Omega(2^m)$.

$D \leftarrow 0$
while $\mathrm{Ber}((1-p)^{2^k})$ **do**
  $D \leftarrow D + 1$
  if $2^k D \geqslant 2^m$ **then return** n
repeat
  $M \xleftarrow{\text{lazy}} \mathrm{Uni}(2^k)$
until $\mathrm{Ber}((1-p)^M)$
sample remaining bits of M (highest to lowest) until one of the bits is 1,
D and the current bits of M form a 2-approximation X of $2^k D + M$
if $X \geqslant 2^m$ **then return** n
fill up M with random bits
return $\min\{n, 2^k D + M\}$

We now slightly change Algorithm 1 and obtain Algorithm 2. Again, we sample $D = \mathrm{Geo}(p) \operatorname{div} 2^k$ and $M = \mathrm{Geo}(p) \bmod 2^k$, the latter lazily as in Algorithm 1. However, now we ensure that whenever we know that the output will be larger than n, we exit and return n. To this end, we check after each incrementation of D whether $2^k D \geqslant 2^m$ (note that this is a cheaper test than $2^k D \geqslant n$), returning n if true. Moreover, after having sampled D and exiting the second loop (so that we have sampled some initial bits of M already), we sample more bits of M from highest to lowest until we find a 1-bit, or until M is completely sampled. At this point, we have a 2-approximation X of $2^k D + M$, since for a number $N \in \mathbb{N}$ with highest bit j we have $2^j \leqslant N < 2^{j+1}$. Since $2^k D + M$ is at least X, we can return n if $X \geqslant 2^m$.

Correctness of this algorithm is clear, since Algorithm 1 was correct and we return n only if we are sure that the output will be at least n.

Implementation Details. It is easy to argue that the tests $2^k D \geqslant 2^m$ and $X \geqslant 2^{m+1}$ are implementable in constant time, again assuming that the exponents k and m fit in $\mathcal{O}(1)$ words: We only have to know the highest bit j of the left hand side. Then we have $2^k D \geqslant 2^m$ if and only if $2^j \geqslant 2^m$, which is trivial to test. However, the highest bit of the counter D is easy to remember, and we can remember the highest bit of X during its construction.

We can also argue that computing $\min\{n, Y\}$, with $Y = 2^k D + M$, in the last step of the algorithm works in time $\mathcal{O}(1 + \log(Y)/w)$: First, we scan Y to determine its exponent. If this is sufficiently smaller than m, we know that $Y \leqslant n$ and return Y. Otherwise, n is not much larger than Y, so we can afford to compare them and return the minimum. Observe that at this point of the algorithm Y cannot be much larger than n, otherwise we would have aborted and returned n earlier. In fact, we have $Y = \mathcal{O}(\min\{n, Y\})$. Thus, the runtime of this step is $\mathcal{O}(1 + \log(1 + Y)/w) = \mathcal{O}(1 + \log(1 + \min\{n, Y\})/w)$, where $\min\{n, Y\}$ is the output. Thus, the second to last line takes asymptotically only as much time as the return statement in the last line.

Runtime. Consider Algorithm 2 without the second to last line and without the return statements. Observe that the remaining parts have an expected runtime of $\mathcal{O}(1)$, as shown by the analysis of Algorithm 1: The tests $2^k D \geqslant 2^m$ and $X \geqslant 2^m$ are one difference to Algorithm 1, but we argued that they can be implemented in $\mathcal{O}(1)$ worst-case time. Another difference is the sampling of further bits of M until we see the first 1-bit, but this clearly takes expected time $\mathcal{O}(1)$.

We also argued in the last section that the second-to-last line and the computation of $\min\{n, 2^k D + M\}$ can be done in the same asymptotic runtime as returning the result. Thus, the lines we did not consider so far take time $\mathcal{O}(1+\log(1+Z)/w)$, where Z is the returned value in this case. The expected value of any of the return statements is thus bounded by $\mathcal{O}(1+\mathbb{E}[\log(1+\mathrm{Geo}(p, n))]/w)$, which is itself bounded by $\mathcal{O}(1 + \log(\min\{n, 1/p\})/w)$ by Lemma 8.

Space Usage. Since $\mathrm{Geo}(p, n)$ is bounded, we could hope for a sampling algorithm with bounded space usage, in contrast to $\mathrm{Geo}(p)$. In this section we show that Algorithm 2 can be adapted to need $\mathcal{O}(n)$ words in the worst case. This is, of course, possibly exponentially large in the input size $\Theta(\log(n) + \log(1/p))$, but it will help for generating $\mathcal{G}(n, p)$'s.

Again, we delay the discussion of the tests $\mathrm{Ber}((1-p)^n)$. Observe that apart from this, Algorithm 2 can be made to use $\mathcal{O}(1 + \log(n)/w)$ words in the worst case: We have $D \leqslant 2^m = \mathcal{O}(n)$, otherwise we exit. Moreover, if we throw away initial 0 bits of $2^k D + M$ in the process of computing X, we need only $\mathcal{O}(1)$ words in the worst case for storing D and the initial bits of M when we arrive at X. At this point, either we exit, or $X < 2^m$, so that we have $2^k D + M \leqslant 2X \leqslant 2^{m+2} = \mathcal{O}(n)$, implying that the remainder of the algorithm uses, again, only $\mathcal{O}(1 + \log(n)/w)$ words in the worst case. This proves that Algorithm 2 has a space usage of $\mathcal{O}(1 + \log(n)/w)$ words in the worst-case, apart from the tests $\mathrm{Ber}((1-p)^n)$.

We cannot hope for the test $\mathrm{Ber}((1-p)^n)$ to run in bounded space if p is a bitstream in general form. However, we can achieve this if p is a rational with given numerator a and denominator b fitting into a constant number of words:

Lemma 9. *Let p be a given rational number with numerator and denominator fitting in $\mathcal{O}(1)$ words, and let $2^{-k} \geqslant p = \Omega(2^{-k})$. For $n = 2^k$ or for uniformly random n in $\{0, \dots, 2^k - 1\}$, we can sample $\mathrm{Ber}((1-p)^n)$ using $\mathcal{O}(n)$ words in the worst case. The algorithm runs in expected constant time using an expected constant number of words.*

Proof. In this case we can afford to compute and store the numerator and denominator of $(1-p)^n = (b-a)^n/b^n$ explicitly in $\mathcal{O}(n)$ words (and $\mathcal{O}(\mathrm{poly}(n))$ time); call them A and B. We now show that $\mathrm{Ber}(A/B)$ can be sampled using $\mathcal{O}(n)$ space in the worst case and expected time $\mathcal{O}(\mathrm{poly}(n))$. Later we discuss how to reduce the expected time and space.

In order to sample $\mathrm{Ber}(A/B)$, we can take a random real number $r \in [0,1)$ and return 1, if $r \leqslant A/B$ and 0 otherwise, as usual. Such an r can be written as $\sum_{i \geqslant 1} b_i 2^{-i}$ with $b_i \in \{0,1\}$ uniformly at random and independent. Let $s_k := \sum_{i>k} b_i 2^{-i+k}$; s_k is again a uniform random number in $[0,1)$. Then we have $r = s_0$ and $s_k = \frac{1}{2}(b_{k+1} + s_{k+1})$. Moreover, $s_k \leqslant A'/B$, if and only if

$$s_{k+1} \leqslant \frac{2A' - b_{k+1}B}{B}.$$

Thus, we get a recursion for numerators A_k such that $r \leqslant A/B$ if and only if $s_0 \leqslant A_0/B$ if and only if $s_k \leqslant A_k/B$. The recursion is $A_0 = A$ and $A_k = 2A_{k-1} - b_k B$ for $k \geqslant 1$. Note that we are done with this test as soon as $A_k/B < 0$, since then $s_k \geqslant 0 > A_k/B$, or $A_k/B \geqslant 1$, since then $s_k \leqslant 1 \leqslant A_k/B$. Also note that before we are done, we always have $0 \leqslant A_k \leqslant B$, so we can store A_k/B using $\mathcal{O}(n)$ words. Thus, this algorithm uses $\mathcal{O}(n)$ words in the worst-case.

We show that this algorithm aborts after an expected number of $\mathcal{O}(1)$ incrementations of k. Observe that we can solve the recursion to

$$A_k/B = 2^k A/B - 2^k r_k,$$

for any $k \geqslant 0$, where $r_k = \sum_{i=1}^{k} b_i 2^{-i}$ is an approximation of r by its first k bits. Hence, we have $A_k/B \in [0,1]$ if and only if $A/B - r_k \in [0, 2^{-k}]$, which can only hold if $A/B - r \in [-2^{-k}, 2^{-k}]$. Since A/B is fixed, this happens with probability at most 2^{1-k}. Thus, the expected runtime of the algorithm is bounded by

$$\sum_{k \geqslant 1} \mathcal{O}(\mathrm{poly}(n,k)) \cdot 2^{1-k} = \mathcal{O}(\mathrm{poly}(n)).$$

Note that above algorithm can be combined with the algorithm from Lemma 3: As long as it needs $\mathcal{O}(n)$ storage we run the latter algorithm, but if we reach the point where it would need more than n words, we switch to the new algorithm. This way we have constant expected runtime and space, while using $\mathcal{O}(n)$ words in the worst case. □