Local-Access Generators for Basic Random Graph Models

by

Amartya Shankha Biswas

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2018

Signature redacted

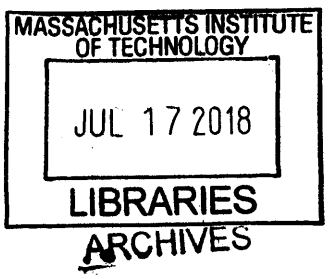
Author ....
Department of Electrical Engineering and Computer Science
February 2, 2018

Signature redacted

Certified by .. ..
Ronitt Rubinfeld
Professor
Thesis Supervisor

Signature redacted

Accepted by. .....
Christopher Terman
Chairman, Department Committee on Graduate Theses



Local-Access Generators for Basic Random Graph Models

by

Amartya Shankha Biswas

Submitted to the Department of Electrical Engineering and Computer Science
on February 2, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Consider a computation on a massive random graph: Does one need to generate the whole random graph up front, prior to performing the computation? Or, is it possible to provide an oracle to answer queries to the random graph "on-the-fly" in a much more efficient manner overall? That is, to provide a *local access generator* which incrementally constructs the random graph locally, at the queried portions, in a manner consistent with the random graph model and all previous choices. Local access generators can be useful when studying the local behavior of specific random graph models. Our goal is to design local access generators whose required resource overhead for answering each query is significantly more efficient than generating the whole random graph.

Our results focus on undirected graphs with independent edge probabilities, that is, each edge is chosen as an independent Bernoulli random variable. We provide a general implementation for generators in this model. Then, we use this construction to obtain the first efficient local implementations for the Erdös-Rényi $G(n, p)$ model, and the Stochastic Block model.

As in previous local-access implementations for random graphs, we support VERTEX-PAIR, NEXT-NEIGHBOR queries, and ALL-NEIGHBORS queries. In addition, we introduce a new RANDOM-NEIGHBOR query. We also give the first local-access generation procedure for ALL-NEIGHBORS queries in the (sparse and directed) Kleinberg's Small-World model. Note that, in the sparse case, an ALL-NEIGHBORS query can be used to simulate the other types of queries efficiently. All of our generators require no pre-processing time, and answer each query using $\mathcal{O}(\text{poly}(\log n))$ time, random bits, and additional space.

Thesis Supervisor: Ronitt Rubinfeld
Title: Professor

Acknowledgments

The results and writeup of this thesis are taken from a paper jointly written with Ronitt Rubinfeld and Anak Yodpinyanee. First and foremost, I would like to thank Ronitt and Anak for their incredible help and support. Since freshman year of my undergrad, Ronitt has been exceptionally encouraging and supportive, and has had the biggest influeuce on my research career. I am immensely grateful to my collaborator Anak for his undying perseverance and dedication. Anak's help has been invaluable in the completion of this thesis.

I would also like to thank my friends and peers for being extremely welcoming and supportive. Their participation and engagement in activities ranging from intense technical discussions to building roller coasters have made my years at MIT a wonderful experience.

Finally, I would like to thank my parents Sudeshna Sarkar, and Goutam Biswas for encouraging me to pursue research. Their unceasing commitment towards my education is the foundation of my academic career.

Contents

1 Introduction **9**

1.1 Our Contributions and Techniques 11

1.1.1 Undirected Graphs 11

1.1.2 Directed Graphs 14

2 Preliminaries **15**

2.1 Local-Access Generators 15

2.2 Random Graph Models 16

2.3 Miscellaneous 17

3 Local-Access Generators for Random Undirected Graphs **19**

3.1 Naïve Generator with an Explicit Adjacency Matrix 20

3.2 NEXT-NEIGHBOR Queries via Run-of-0s Sampling 22

3.2.1 Data structure 22

3.2.2 Queries and Updates 23

3.3 Final Generator via the Bucketing Approach 25

3.3.1 Partitioning into buckets 26

3.3.2 Filling a bucket 27

3.3.3 Putting it all together: RANDOM-NEIGHBOR queries 27

3.4 Implementation of FILL 29

3.5 Removing the Perfect-Precision Arithmetic Assumption 32

4 Applications to Erdös-Rényi Model and Stochastic Block Model **35**

4.1 Erdös-Rényi Model 36
4.2 Stochastic Block model 36
4.2.1 Sampling from the Multivariate Hypergeometric Distribution 38
4.2.2 Data structure 40

5 Local-Access Generators for Random Directed Graphs **41**
5.1 Generator for $c = 1$ 41
5.1.1 Phase 1: Sample the distance D 42
5.1.2 Phase 2: Sampling neighbors at distance D 43
5.2 Generator for $c \neq 1$ 44
5.2.1 Case $c < 1$ 44
5.2.2 Case $c > 1$ 45

A Further Analysis and Extensions of Algorithm 2 **47**
A.1 Performance Guarantee 47
A.2 Supporting VERTEX-PAIR Queries 50

B Alternative Generator with Deterministic Performance Guarantee **53**
B.1 Data structure for next-neighbor queries in the Erdös-Rényi model 53
B.2 Data structure for VERTEX-PAIR queries in the Erdös-Rényi model 56
B.3 Data structure for the Stochastic Block model 57

C Additional related work **59**

Chapter 1

Introduction

The problem of computing local information of huge random objects was pioneered in [18, 19]. Further work of [33] considers the generation of sparse random $G(n,p)$ graphs from the Erdös-Rényi model [14], with $p = O(\text{poly}(\log n)/n)$, which answers $\text{poly}(\log n)$ ALL-NEIGHBORS queries, listing the neighbors of queried vertices. While these generators use polylogarithmic resources over their entire execution, they generate graphs that are only guaranteed to *appear random* to algorithms that inspect a *limited portion* of the generated graph.

In [15], the authors construct an oracle for the generation of recursive trees, and BA preferential attachment graphs. Unlike [33], their implementation allows for an arbitrary number of queries. This result is particularly interesting – although the graphs in this model are generated via a sequential process, the oracle is able to locally generate arbitrary portions of it and answer queries in polylogarithmic time. Though preferential attachment graphs are sparse, they contain vertices of high degree, thus [15] provides access to the adjacency list through NEXT-NEIGHBOR queries.

In this work, we begin by *formalizing* a model of local-access generators implicitly used in [15]. We next construct oracles that allow queries to both the adjacency matrix and adjacency list representation of a basic class of random graph families, without generating the entire graph at the onset. Our oracles provide VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR queries[1] for graphs with *independent edge probabilities*, that is,

[1] VERTEX-PAIR(u,v) returns whether u and v are adjacent, NEXT-NEIGHBOR(v) returns a new neighbor

when each edge is chosen as an independent Bernoulli random variable. Using this framework, we construct the first *efficient* local-access generators for undirected graph models, supporting all three types of queries using $\mathcal{O}(\text{poly}(\log n))$ time, space, and random bits per query, under assumptions on the ability to compute certain values pertaining to consecutive edge probabilities. In particular, our construction yields local-access generators for the Erdös-Rényi $G(n, p)$ model (for *all* values of p), and the Stochastic Block model with random community assignment. As in [15] (and unlike the generators in [18, 19, 33]), our techniques allow unlimited queries.

While VERTEX-PAIR and NEXT-NEIGHBOR queries, as well as ALL-NEIGHBORS queries for sparse graphs, have been considered in the prior works of [15, 18, 19, 33], we provide the first implementation (to the best of our knowledge) of RANDOM-NEIGHBOR queries, which do not follow trivially from the ALL-NEIGHBOR queries in *non-sparse graphs*. Such queries are useful, for instance, for sub-linear algorithms that employ random walk processes. RANDOM-NEIGHBOR queries present particularly interesting challenges, since as we note in Section 1.1.1, (1) RANDOM-NEIGHBOR queries affect the conditional probabilities of the remaining neighbors in a non-trivial manner, and (2) our implementation does not resort to explicitly sampling the degree of any vertex in order to generate a random neighbor. First, sampling the degree of the query vertex, we suspect, is not viable for *sub-linear* generators, because this quantity alone imposes dependence on the existence of *all* of its potential incident edges. Therefore, our generator needs to return a random neighbor, with probability reciprocal to the query vertex's degree, without resorting to "knowing" its degree. Second, even without committing to the degrees, answers to RANDOM-NEIGHBOR queries affect the conditional probabilities of the remaining adjacencies in a global and non-trivial manner – that is, from the point of view of the *agent* interacting with the generator. The generator, however, must somehow maintain and leverage its additional *internal knowledge* of the partially-generated graph, to keep its computation tractable throughout the entire graph generation process.

We then consider local-access generators for directed graphs in Kleinberg's Small World

of v each time it is invoked (until none is left), and RANDOM-NEIGHBOR(v) returns a uniform random neighbor of v (if v is not isolated).

model. In this case, the probabilities are based on distances in a 2-dimensional grid. Using a modified version of our previous sampling procedure, we present such a generator supporting ALL-NEIGHBORS queries in $\mathcal{O}(\text{poly}(\log n))$ time, space and random bits per query (since such graphs are sparse, the other queries follow directly).

For additional related work, see Section C.

1.1 Our Contributions and Techniques

We begin by formalizing a model of *local-access generators* (Section 2.1), implicitly used in [15]. Our work provides local-access generators for various basic classes of graphs described in the following, with VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR queries. In all of our results, each query is processed using $\text{poly}(\log n)$ time, random bits, and additional space, with *no initialization overhead.* These guarantees hold even in the case of adversarial queries. Our bounds assume constant computation time for each arithmetic operation with $O(\log n)$-bit precision. Each of our generators constructs a random graph drawn from a distribution that is $1/\text{poly}(n)$-close to the desired distribution in the L_1-distance.[2]

1.1.1 Undirected Graphs

In Section 3 we construct local access generators for the generic class of undirected graphs with *independent edge probabilities* $\{p_{u,v}\}_{u,v\in V}$, where $p_{u,v}$ denote the probability that there is an edge between u and v. Throughout, we identify our vertices via their unique IDs from 1 to n, namely $V = [n]$. We assume that we can compute various values pertaining to consecutive edge probabilities for the class of graphs, as detailed below. We then show that such values can be computed for graphs generated according to the Erdös-Rényi $G(n,p)$ model and the Stochastic Block model.

NEXT-NEIGHBOR Queries. We note that the next neighbor of a vertex can be found trivially by generating consecutive entries of the adjacency matrix, but for small edge prob-

[2]The L_1-*distance* between two probability distributions p and q over domain D is defined as $\|\mathsf{p}-\mathsf{q}\|_1 = \sum_{x\in D}|p(x)-q(x)|$. We say that p and q are ϵ-close if $\|\mathsf{p}-\mathsf{q}\|_1 \le \epsilon$.

abilities $p_{u,v} = o(1)$ this implementation can be too slow. In our algorithms, we achieve speed-up by sampling multiple neighbor values at once for a given vertex u; more specifically, we sample for the number of "non-neighbors" preceding the next neighbor. To do this, we assume that we have access to an oracle which can estimate the "skip" probabilities $F(v, a, b) = \prod_{u=a}^{b}(1 - p_{v,u})$, where $F(v, a, b)$ is the probability that v has no neighbors in the range $[a, b]$. We later show that it is possible to compute this quantity efficiently for the $G(n, p)$ and Stochastic block models.

A main difficulty in our setup, as compared to [15], arises from the fact that our graph is undirected, and thus we must design a data structure that "informs" all (potentially $\Theta(n)$) non-neighbors once we decide on the query vertex's next neighbor. More concretely, if u' is sampled as the next neighbor of v after its previous neighbor u, we must maintain consistency in subsequent steps by ensuring that none of the vertices in the range (u, u') return v as a neighbor. This update will become even more complicated as we later handle RANDOM-NEIGHBOR queries, where we may generate non-neighbors at random locations.

In Section 3.2, we present a very simple randomized generator (Algorithm 2) that supports NEXT-NEIGHBOR queries efficiently, albeit the analysis of its performance is rather complicated. We remark that this approach may be extended to support VERTEX-PAIR queries with superior performance (given that we do not to support RANDOM-NEIGHBOR queries) and to provide deterministic resource usage guarantee – the full analysis can be found in Section A and B, respectively.

RANDOM-NEIGHBOR Queries. We provide efficient RANDOM-NEIGHBOR queries (Section 3.3). The ability to do so is surprising. First, note that after performing a RANDOM-NEIGHBOR query all other conditional probabilities will be affected in a non-trivial way. [3] This requires a way of implicitly keeping track of all the resulting changes. Second, we can sample a RANDOM-NEIGHBOR with the correct probability $1/\deg(v)$, even though we do not sample or know the degree of the vertex.

We formulate a *bucketing approach* (Section 3.3) which samples multiple consecutive

[3] Consider a $G(n, p)$ graph with small p, say $p = 1/\sqrt{n}$, such that vertices will have $\tilde{\mathcal{O}}(\sqrt{n})$ neighbors with high probability. After $\tilde{\mathcal{O}}(\sqrt{n})$ RANDOM-NEIGHBOR queries, we will have uncovered all the neighbors (w.h.p.), so that the conditional probability of the remaining $\Theta(n)$ edges should now be close to zero.

edges at once, in such a way that the conditional probabilities of the unsampled edges remain independent and "well-behaved" during subsequent queries. For each vertex v, we divide the vertex set (potential neighbors) or v into consecutive ranges (buckets), so that each bucket contains, in expectation, roughly the same number of neighbors $\sum_{u=a}^{b} p_{v,u}$ (which we must be able to compute efficiently). The subroutine of NEXT-NEIGHBOR may be applied to sample the neighbors within a bucket in expected constant time. Then, one may obtain a random neighbor of v by picking a random neighbor from a random bucket; probabilities of picking any neighbors may be normalized to the uniform distribution via rejection sampling, while stilling yielding $\mathrm{poly}(\log n)$ complexities overall. This bucketing approach also naturally leads to our data structure that requires constant space for each bucket and for each edge, using $\Theta(n+m)$ overall memory requirement. The VERTEX-PAIR queries are implemented by sampling the relevant bucket.

We now consider the application of our construction above to actual random graph models, where we must realize the assumption that $\prod_{u=a}^{b}(1-p_{v,u})$ and $\sum_{u=a}^{b} p_{v,u}$ can be computed efficiently. This holds trivially for the $G(n,p)$ model via closed-form formulas, but requires an additional back-end data structure for the Stochastic Block models.

Erdös-Rényi. In Section 4.1, we apply our construction to random $G(n,p)$ graphs for arbitrary p, and obtain VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR queries, using polylogarithmic resources (time, space and random bits) per query. We remark that, while $\Omega(n+m) = \Omega(pn^2)$ time and space is clearly necessary to generate and represent a full random graph, our implementation supports local-access via all three types of queries, and yet can generate a full graph in $\widetilde{O}(n+m)$ time and space (Corollary 3), which is tight up to polylogarithmic factors.

Stochastic Block Model. We generalize our construction to the Stochastic Block Model. In this model, the vertex set is partitioned into r *communities* $\{C_1, \ldots, C_r\}$. The probability that an edge exists depends on the communities of its endpoints: if $u \in C_i$ and $v \in C_j$, then $\{u,v\}$ exists with probability $p_{i,j}$, given in an $r \times r$ matrix $\mathbf{P}$. As communities in the observed data are generally unknown a priori, and significant research has been devoted to designing efficient algorithm for community detection and recovery, these studies generally

consider the *random community assignment* condition for the purpose of designing and analyzing algorithms (see e.g., [32]). Thus, in this work, we aim to construct generators for this important case, where the community assignment of vertices are independently sampled from some given distribution R.

Our approach is, as before, to sample for the next neighbor or a random neighbor directly, although our result does not simply follow closed-form formulas, as the probabilities for the potential edges now depend on the communities of endpoints. To handle this issue, we observe that it is sufficient to efficiently count the number of vertices of each community in any range of contiguous vertex indices. We then design a data structure extending a construction of [19], which maintain these counts for ranges of vertices, and "sample" the partition of their counts only on an as-needed basis. This extension results in an efficient technique to sample counts from the *multivariate hypergeometric distribution* (Section 4.2.1). This sampling procedure may be of independent interest. For r communities, this yields an implementation with $\mathcal{O}(r \cdot \text{poly}(\log n))$ overhead in required resources for each operation. This upholds all previous polylogarithmic guarantees when $r = \text{poly}(\log n)$.

1.1.2 Directed Graphs

Lastly, we consider Kleinberg's Small World model ([24, 29]) in Section 5. While Small-World models are proposed to capture properties of observed data such as small shortest-path distances and large clustering coefficients [41], this important special case of Kleinberg's model, defined on two-dimensional grids, demonstrates underlying geographical structures of networks. The vertices are aligned on a $\sqrt{n} \times \sqrt{n}$ grid, and the edge probabilities are a function of a two-dimensional distance metric. Since the degree of each vertex in this model is $\mathcal{O}(\log n)$ with high probability, we design generators supporting ALL-NEIGHBOR queries.

Chapter 2

Preliminaries

2.1 Local-Access Generators

We consider the problem of locally generating random graphs $G = (V, E)$ drawn from the desired families of simple unweighted graphs, undirected or directed. We denote the number of vertices $n = |V|$, and refer to each vertex simply via its unique ID from $[n]$. For undirected G, the set of neighbors of $v \in V$ is defined as $\Gamma(v) = \{u \in V : \{v, u\} \in E\}$; denote its degree by $\deg(v) = |\Gamma(v)|$. Inspired by the goals and results of [15], we define a model of local-access generators as follows.

Definition 1. *A* local-access generator *of a random graph G sampled from a distribution* D*, is a data structure that provides access to G by answering various types of* supported queries*, while satisfying the following:*

- ***Consistency.*** *The responses of the local-access generator to all probes throughout the entire execution must be consistent with a single graph G.*
- ***Distribution equivalence.*** *The random graph G provided by the generator must be sampled from some distribution D' that is ϵ-close to the desired distribution D in the L_1-distance. In this work we focus on supporting $\epsilon = n^{-c}$ for any desired constant $c > 0$. As for* RANDOM-NEIGHBOR(v)*, the distribution from which a neighbor is returned must be ϵ-close to the uniform distribution over neighbors of v with respect to the sampled random graph G (w.h.p $1 - n^{-c}$ for each query).*

- ***Performance.*** *The resources, consisting of (1) computation time, (2) additional random bits required, and (3) additional space required, in order to compute an answer to a single query and update the data structure, must be sub-linear, preferably* $\mathrm{poly}(\log n)$.

In particular, we allow queries to be made adversarially and non-deterministically. The adversary has full knowledge of the generator's behavior and its past random bits.

For ease of presentation, we allow generators to create graphs with self-loops. When self-loops are not desired, it is sufficient to add a wrapper function that simply re-invokes NEXT-NEIGHBOR(v) or RANDOM-NEIGHBOR(v) when the generator returns v.

Supported Queries in our Model. For undirected graphs, we consider queries of the following forms. now we might want to do NEXT-NEIGHBOR first for consistency.

- NEXT-NEIGHBOR(v): The generator returns the neighbor of v with the lowest ID that has not been returned during the execution of the generator so far. If all neighbors of u have already been returned, the generator returns $n + 1$.
- RANDOM-NEIGHBOR(v): The generator returns a neighbor of v uniformly at random (with probability $1/\deg(v)$ each). If v is isolated, $\perp$ is returned.
- VERTEX-PAIR(u, v): The generator returns either 1 or 0, indicating whether $\{u, v\} \in E$ or not.
- ALL-NEIGHBORS(v): The generator returns the entire list of out-neighbors of v. We may use this query for relatively sparse graphs, specifically in the Small-World model.

2.2 Random Graph Models

Erdös-Rényi Model. We consider the $G(n, p)$ model: each edge $\{u, v\}$ exists independently with probability $p \in [0, 1]$. Note that p is not assumed to be constant, but may be a function of n.

Stochastic Block Model. This model is a generalization of the Erdös-Rényi Model. The vertex set V is partitioned into r communities $C_1, \ldots, C_r$. The probability that the edge

$\{u, v\}$ exists is $p_{i,j}$ when $u \in C_i$ and $v \in C_j$, where the probabilities are given as an $r \times r$ symmetric matrix $\mathbf{P} = [p_{i,j}]_{i,j \in [r]}$. We assume that we are given explicitly the distribution R over the communities, and each vertex is assigned its community according to R independently at random.[1]

Small-World Model. In this model, each vertex is identified via its 2D coordinate $v = (v_x, v_y) \in [\sqrt{n}]^2$. Define the Manhattan distance as $\textsc{dist}(u, v) = |u_x - v_x| + |u_y - v_y|$, and the probability that each directed edge (u, v) exists is $c/(\textsc{dist}(u, v))^2$. Here, c is an indicator of the number of long range directed edges present at each vertex. A common choice for c is given by normalizing the distribution so that there is exactly one directed edge emerging from each vertex ($c = \Theta(1/\log n)$). We will however support a range of values of $c = \log^{\pm\Theta(1)} n$. While not explicitly specified in the original model description of [24], we assume that the probability is rounded down to 1 if $c/(\textsc{dist}(u, v))^2 > 1$.

2.3 Miscellaneous

Arithmetic operations. Let N be a sufficiently large number of bits required to maintain a multiplicative error of at most a $\frac{1}{\text{poly}(n)}$ factor over $\text{poly}(n)$ elementary computations $(+, -, \cdot, /, \exp)$.[2] We assume that each elementary operation on words of size N bits can be performed in constant time. Likewise, a random N-bit integer can be acquired in constant time. We assume that the input is also given with N-bit precision.

Sampling via a CDF. Consider a probability distribution X over $O(n)$ consecutive integers, whose cumulative distribution function (CDF) for can be computed with at most n^{-c} additive error for constant c. Using $\mathcal{O}(\log n)$ CDF evaluations, one can sample from a distribution that is $\frac{1}{\text{poly}(n)}$-close to X in L_1-distance.[3]

[1] Our algorithm also supports the alternative specification where the community sizes $\langle |C_1|, \ldots, |C_r| \rangle$ are given instead, where the assignment of vertices V into these communities is chosen uniformly at random.

[2] In our application of exp, we only compute a^b for $b \in \mathbb{Z}^+$ and $0 < a \leq 1 + \Theta(\frac{1}{b})$, where $a^b = \mathcal{O}(1)$. For this, $N = \mathcal{O}(\log n)$ bits are sufficient to achieve the desired accuracy, namely an additive error of n^{-c}.

[3] Generate a random N-bit number r, and binary-search for the smallest domain element x where $\mathbb{P}[X \leq x] \geq r$.

Chapter 3

Local-Access Generators for Random Undirected Graphs

In this section, we provide an efficient implementation of local-access generators for random undirected graphs when the probabilities $p_{u,v} = \mathbb{P}[\{u,v\} \in E]$ are given. More specifically, we assume that the following quantities can be efficiently computed: (1) the probability that there is no edge between a vertex u and a range of consecutive vertices from $[a,b]$, namely $\prod_{u=a}^{b}(1-p_{v,u})$, and (2) the sum of the edge probabilities (i.e., the expected number of edges) between u and vertices from $[a,b]$, namely $\sum_{u=a}^{b} p_{v,u}$. We will later give subroutines for computing these values for the Erdös-Rényi model and the Stochastic Block model with randomly-assigned communities in Section 4. We also begin by assuming perfect-precision arithmetic, until Section 3.5 where we show how to relax this assumption to $N = \Theta(\log n)$-bit precision.

First, we propose a simple implementation of our generator in Section 3.1 that sequentially fills out the adjacency matrix; while we do not focus on its efficiency, we establish some basic concepts for further analysis in this section. Next, we improve our subroutine for NEXT-NEIGHBOR queries in Section 3.2: this algorithm samples for the next candidate of the next neighbor in a more direct manner to speed-up the process. Extending this construction, we obtain our main algorithm in Section 3.3 via the bucketing technique: partition the vertex set into contiguous ranges to normalize the expected number of neighbors in each bucket, allowing an efficient RANDOM-NEIGHBOR implementation by picking a

random neighbor from a random bucket. The subroutine that samples for neighbors within a bucket, along with the remaining analysis of the algorithm, is given later in Section 3.4. Lastly, Section 3.5 handles the errors that may occur due to the use of finite precision.

3.1 Naïve Generator with an Explicit Adjacency Matrix

First, consider a naïve implemention that simply fills out the cells of the $n \times n$ adjacency matrix $\mathbf{A}$ of G one-by-one as required by each query. Each entry $\mathbf{A}[u][v]$ occupies exactly one of following three states: $\mathbf{A}[u][v] = 1$ or 0 if the generator has determined that $\{u, v\} \in E$ or $\{u, v\} \notin E$, respectively, and $\mathbf{A}[u][v] = \phi$ if whether $\{u, v\} \in E$ or not will be determined by future random choices. Aside from $\mathbf{A}$, our generator also maintains the vector $\mathbf{last}$, where $\mathbf{last}[v]$ records the neighbor of v returned in the last call NEXT-NEIGHBOR(v), or $\mathbf{last}[v] = 0$ if no such call has been invoked. This definition of $\mathbf{last}$ was introduced in [15]. All cells of $\mathbf{A}$ and $\mathbf{last}$ are initialized to ϕ and 0, respectively. We refer to Algorithm 1 for its straightforward implemention, but highlight some notations and useful observations here.

Algorithm 1 Naïve Generator

```
procedure VERTEX-PAIR(u, v)
    if A[u][v] = φ then
        draw X_{u,v} ~ Bern(p_{u,v})
        A[v][u], A[u][v] ← X_{u,v}
    return A[u][v]
procedure NEXT-NEIGHBOR(v)
    for u ← last[v] + 1 to n do
        if VERTEX-PAIR(v, u) = 1 then
            last[v] ← u
            return u
    last[v] ← n + 1
    return n + 1
procedure RANDOM-NEIGHBOR(v)
    R ← V
    repeat
        sample u ∈ R u.a.r.
        if VERTEX-PAIR(v, u) = 1 then
            return u
        else
            R ← R \ {u}
    until R = ∅
    return ⊥
```

Characterizing random choices via $X_{u,v}$'s. Algorithm 1 updates the cell $\mathbf{A}[u][v] = \phi$ to the value of the Bernoulli random variable (RV) $X_{u,v} \sim \mathsf{Bern}(p_{u,v})$ (i.e., flip a coin with bias $p_{u,v}$) only when it needs to decide whether $\{u, v\} \in E$. For the sake of analysis, we will frequently consider the *entire* table of RVs $X_{u,v}$ being sampled *up-front* (i.e., flip all coins), and the algorithm simply "uncovers" these variables instead of making coin-flips. Thus, every cell $\mathbf{A}[u][v]$ is originally ϕ, but will eventually take the value $X_{u,v}$ once the

graph generation is complete. An example application of this view of $X_{u,v}$ is the following analysis.

Sampling from $\Gamma(v)$ uniformly without knowing $\deg(v)$. Consider a RANDOM-NEIGHBOR(v) query. We create a *pool* R of vertices, draw from this pool one-by-one, until we find a neighbor of u. Then, for any fixed table $X_{u,v}$, the probability that a vertex $u \in \Gamma(v)$ is returned is simply the probability that, in the sequence of vertices drawn from the pool R, u appears first among all neighbors in $\Gamma(v)$. Hence, we sample each $u \in \Gamma(v)$ with probability $1/\deg(v)$, even without *knowing* the specific value of $\deg(v)$.

Capturing the state of the partially-generated graph with A. Under the presence of RANDOM-NEIGHBOR queries, the probability distribution of the random graphs conditioned on the past queries and answers can be very complex: for instance, the number of repeated returned neighbors of v reveals information about $\deg(v) = \sum_{u \in V} X_{u,v}$, which imposes dependencies on as many as $\Theta(n)$ variables. Our generator, on the other hand, records the neighbors and also *non-neighbors* not revealed by its answers, yet surprisingly this internal information fully captures the state of the partially-generated graph. This suggests that we should design generators that maintain $\mathbf{A}$ as done in Algorithm 1, but in a more implicit and efficient fashion in order to achieve the desired complexities. Another benefit of this approach is that any analysis can be performed on the simple representation $\mathbf{A}$ rather than any complicated data structure we may employ.

Obstacles for maintaining A. There are two problems in the current approach. Firstly, the algorithm only finds a neighbor, for a RANDOM-NEIGHBOR or NEXT-NEIGHBOR query, with probability $p_{u,v}$, which requires too many iterations: for $G(n,p)$ this requires $1/p$ iterations, which is already infeasible for $p = o(1/\mathrm{poly}(\log n))$. Secondly, the algorithm may generate a large number of non-neighbors in the process, possibly in random or arbitrary locations.

3.2 NEXT-NEIGHBOR Queries via Run-of-0s Sampling

We now speed-up our NEXT-NEIGHBOR(v) procedure by attempting to sample for the first index $u > \mathbf{last}[v]$ of $X_{v,u} = 1$, from a sequence of Bernoulli RVs $\{X_{v,u}\}_{u>\mathbf{last}[v]}$, in a direct fashion. To do so, we sample a consecutive "run" of 0's with probability $\prod_{u=\mathbf{last}[v]+1}^{u'}(1 - p_{v,u})$: this is the probability that there is no edge between a vertex v and any $u \in (\mathbf{last}[v], u']$, which can be computed efficiently by our assumption. The problem is that, some entries $\mathbf{A}[v][u]$'s in this run may have already been determined (to be 1 or 0) by queries NEXT-NEIGHBOR(u) for $u > \mathbf{last}[v]$. To this end, we give a succinct data structure that determines the value of $\mathbf{A}[v][u]$ for $u > \mathbf{last}[v]$ and, more generally, captures the state $\mathbf{A}$, in Section 3.2.1. Using this data structure, we ensure that our sampled run does not skip over any 1. Next, for the sampled index u of the first occurrence of 1, we check against this data structure to see if $\mathbf{A}[v][u]$ is already assigned to 0, in which case we re-sample for a new candidate $u' > u$. Section 3.2.2 discusses the subtlety of this issue.

We note that we do not yet try to handle other types of queries here yet. We also do not formally bound the number of re-sampling iterations of this approach here, because the argument is not needed by our final algorithm. Yet, we remark that $O(\log n)$ iterations suffice with high probability, even if the queries are adversarial. This method can be extended to support VERTEX-PAIR queries (but unfortunately not RANDOM-NEIGHBOR queries). See Section A for full details.

3.2.1 Data structure

From the definition of $X_{u,v}$, NEXT-NEIGHBOR(v) is given by $\min\{u > \mathbf{last}[v] : X_{v,u} = 1\}$ (or $n + 1$ if no satisfying u exists). Let $P_v = \{u : \mathbf{A}[v][u] = 1\}$ be the set of known neighbors of v, and $w_v = \min\{(P_v \cap (\mathbf{last}[v], n]) \cup \{n + 1\}\}$ be its first known neighbor not yet reported by a NEXT-NEIGHBOR(v) query, or equivalently, the next occurrence of 1 in v's row on $\mathbf{A}$ after $\mathbf{last}[v]$. Note that $w_v = n + 1$ denotes that there is no known neighbor of v after $\mathbf{last}[v]$. Consequently, $\mathbf{A}[v][u] \in \{\phi, 0\}$ for all $u \in (\mathbf{last}[v], w_v)$, so NEXT-NEIGHBOR($v$) is either the index u of the first occurrence of $X_{v,u} = 1$ in this range, or w_v if no such index exists.

We keep track of $\mathbf{last}[v]$ in a dictionary, where the key-value pair $(v, \mathbf{last}[v])$ is stored only when $\mathbf{last}[v] \neq 0$: this removes any initialization overhead. Each P_v is maintained as an ordered set, which is also only instantiated when it becomes non-empty. We maintain P_v simply by adding u to v if a call NEXT-NEIGHBOR(v) returns u, and vice versa. Clearly, $\mathbf{A}[v][u] = 1$ if and only if $u \in P_v$ by construction.

As discussed in the previous section, we cannot maintain $\mathbf{A}$ explicitly, as updating it requires replacing up to $\Theta(n)$ ϕ's to 0's for a single NEXT-NEIGHBOR query in the worst case. Instead, we argue that $\mathbf{last}$ and P_v's provide a succinct representation of $\mathbf{A}$ via the following observation. For simplicity, we say that $X_{u,v}$ is *decided* if $\mathbf{A}[u][v] \neq \phi$, and call it *undecided* otherwise.

Lemma 1. *The data structures* $\mathbf{last}$ *and* P_v*'s together provide a succinct representation of* $\mathbf{A}$ *when only* NEXT-NEIGHBOR *queries are allowed. In particular,* $\mathbf{A}[v][u] = 1$ *if and only if* $u \in P_v$*. Otherwise,* $\mathbf{A}[v][u] = 0$ *when* $u < \mathbf{last}[v]$ *or* $v < \mathbf{last}[u]$*. In all remaining cases,* $\mathbf{A}[v][u] = \phi$*.*

Proof. The condition for $\mathbf{A}[v][u] = 1$ clearly holds by constuction. Otherwise, observe that $\mathbf{A}[v][u]$ becomes decided (that is, its value is changed from ϕ to 0) precisely during the first call of NEXT-NEIGHBOR(v) that returns a value $u' > u$ which thereby sets $\mathbf{last}[v]$ to u' yielding $u < \mathbf{last}[v]$, or vice versa. □

3.2.2 Queries and Updates

We now provide our generator (Algorithm 2), and discuss the correctness of its sampling process. The argument here is rather subtle and relies on viewing the random process as an "uncovering" process on the table of RVs $X_{u,v}$'s as previously introduced in Section 3.1. Algorithm 2, considers the following experiment for sampling the next neighbor of v in the range $(\mathbf{last}[v], w_v)$. Suppose that we generate a sequence of $w_v - \mathbf{last}[v] - 1$ independent coin-tosses, where the i^{th} coin $C_{v,u}$ corresponding to $u = \mathbf{last}[v]+i$ has bias $p_{v,u}$, regardless of whether $X_{v,u}$'s are decided or not. Then, we use the sequence $\langle C_{v,u} \rangle$ to assign values to *undecided* random variable $X_{v,u}$. The crucial observation here is that, the *decided* random variables $X_{v,u} = 0$ do not need coin-flips, and the corresponding coin result $C_{v,u}$ can simply

be discarded. Thus, we need to generate coin-flips up until we encounter some u satisfying both (i) $C_{v,u} = 1$, and (ii) $\mathbf{A}[v][u] = \phi$.

Let $\mathsf{F}(v, a, b)$ denote the probability distribution of the occurrence u of the first coin-flip $C_{v,u} = 1$ among the neighbors in (a, b). More specifically, $F \sim \mathsf{F}(v, a, b)$ represents the event that $C_{v,a+1} = \cdots = C_{v,F-1} = 0$ and $C_{v,F} = 1$, which happens with probability $\mathbb{P}[F = f] = \prod_{u=a+1}^{f-1}(1 - p_{v,u}) \cdot p_{v,f}$. For convenience, let $F = b$ denote the event where all $C_{v,u} = 0$. Our algorithm samples $F_1 \sim \mathsf{F}(v, \mathbf{last}[v], w_v)$ to find the first occurrence of $C_{v,F_1} = 1$, then samples $F_2 \sim \mathsf{F}(v, F_1, w_v)$ to find the second occurrence $C_{v,F_2} = 1$, and so on. These values $\{F_i\}$ are iterated as u in Algorithm 2. As this process generates u satisfying (i) in the increasing order, we repeat until we find one that also satisfies (ii). Note that once the process terminates at some u, we make no implications on the results of any uninspected coin-flips after $C_{v,u}$.

Algorithm 2 Sampling NEXT-NEIGHBOR

```
procedure NEXT-NEIGHBOR(v)
    u ← last[v]
    w_v ← min{(P_v ∩ (u, n]) ∪ {n+1}}
    repeat
        sample F ~ F(v, u, w_v)
        u ← F
    until u = w_v or last[u] < v
    if u ≠ w_v then
        P_v ← P_v ∩ {u}
        P_u ← P_u ∩ {v}
    last[v] ← u
    return u
```

Obstacles for extending beyond NEXT-NEIGHBOR queries. There are two main issues that prevent this method from supporting RANDOM-NEIGHBOR queries. Firstly, while one might consider applying NEXT-NEIGHBOR from some random location u to find the minimum $u' \geq u$ where $\mathbf{A}[v][u'] = 1$, the probability of choosing u' will depend on the probabilities $p_{v,u}$'s, and is generally not uniform. While a rejection sampling method may be applied to balance out the probabilities of choosing neighbors, these arbitrary $p_{v,u}$'s may distribute the neighbors rather unevenly: some small contiguous locations may contain so many neighbors that the rejection sampling approach requires too many iterations to obtain a single uniform neighbor.

Secondly, in developing Algorithm 2, we observe that $\mathbf{last}[v]$ and P_v together provide a succinct representation of $\mathbf{A}[v][u] = 0$ only for contiguous cells $\mathbf{A}[v][u]$ where $u \leq \mathbf{last}[v]$ or $v \leq \mathbf{last}[u]$: they cannot handle 0 anywhere else. Unfortunately, in order to extend our construction to support RANDOM-NEIGHBOR queries using the idea suggested in Al-

gorithm 1, we must unavoidably assign $\mathbf{A}[v][u]$ to 0 in random locations beyond $\mathbf{last}[v]$ or $\mathbf{last}[u]$, which cannot be captured by the current data structure. Furthermore, unlike 1's, we cannot record 0's using a data structure similarly to that of P_v. More specifically, to speed-up the sampling process for small $p_{v,u}$'s, we must generate many random non-neighbors at once as suggested in Algorithm 2, but we cannot afford to spend time linear in the number of created 0's to update our data structure. We remedy these issues via the following bucketing approach.

3.3 Final Generator via the Bucketing Approach

We now resolve both of the above issues via the bucketing approach, allowing our generator to support all remaining types of queries. We begin this section by focusing first on RANDOM-NEIGHBOR queries, then extend the construction to the remaining ones. In order to handle RANDOM-NEIGHBOR(v), we divide the neighbors of v into *buckets* $B_v = \{B_v^{(1)}, B_v^{(2)}, \ldots\}$, so that each bucket contains, in expectation, roughly the same number of neighbors of v. We may then implement RANDOM-NEIGHBOR(v) by randomly selecting a bucket $B_v^{(i)}$, fill in entries $\mathbf{A}[v][u]$ for $u \in B_v^{(i)}$ with 1's and 0's, then report a random neighbor from this bucket. As the bucket size may be too large when the probabilities are small, instead of using a linear scan, our FILL subroutine will be implemented with the NEXT-NEIGHBOR subroutine in Algorithm 2 previously developed in Section 3.2. Since the number of iterations required by this subroutine is roughly proportional to the number of neighbors, we choose to allocate a constant number of neighbors in expectation to each bucket: with constant probability the bucket contains some neighbors, and with high probability it has at most $O(\log n)$ neighbors.

Nonetheless, as the actual number of neighbors appearing in each bucket may be different, we balance out these discrepancies by performing *rejection sampling*, equalizing the probability of choosing any neighbor implicitly, again without the knowledge of $\deg(v)$ as previously done in Section 3.1. Leveraging the fact that the maximum number of neighbors in any bucket is $\mathcal{O}(\log n)$, we show not only that the probabability of success in the rejection sampling process is at least $1/\mathrm{poly}(\log n)$, but the number of iterations required by

NEXT-NEIGHBOR is also bounded by $\text{poly}(\log n)$, achieving the overall $\text{poly}(\log n)$ complexities. Here in this section, we will extensively rely on the assumption that the expected number of neighbors for consecutive vertices, $\sum_{u=a}^{b} p_{v,u}$, can be computed efficiently.

3.3.1 Partitioning into buckets

More formally, we fix some sufficiently large constant L, and assign the vertex u to the $\lceil \sum_{i=1}^{u} p_{v,i}/L \rceil^{\text{th}}$ bucket of v. Essentially, each bucket represents a contiguous range of vertices, where the expected number of neighbors of v in the bucket is (mostly) in the range $[L-1, L+1]$ (for example, for $G(n,p)$, each bucket contains roughly L/p vertices). Let us define $\Gamma^{(i)}(v) = \Gamma(v) \cap B_v^{(i)}$, the actual neighbors appearing in bucket $B_v^{(i)}$. Our construction ensures that $\mathbb{E}\left[|\Gamma^{(i)}(v)|\right] < L+1$ for every bucket, and $\mathbb{E}\left[|\Gamma^{(i)}(v)|\right] > L-1$ for every $i < |B_v|$ (i.e., the condition holds for all buckets but possibly the last one).

Now, we show that with high probability, all the bucket sizes $|\Gamma^{(i)}(v)| = \mathcal{O}(\log n)$, and at least a $1/3$-fraction of the buckets are non-empty (i.e., $|\Gamma^{(i)}(v)| > 0$), via the following lemmas.

Lemma 2. *With high probability, the number of neighbors in every bucket, $|\Gamma^{(i)}(v)|$, is at most $\mathcal{O}(\log n)$.*

Proof. Fix a bucket $B_v^{(i)}$, and consider the Bernoulli RVs $\{X_{v,u}\}_{u \in B_v^{(i)}}$. The expected number of neighbors in this bucket is $\mathbb{E}\left[|\Gamma^{(i)}(v)|\right] = \mathbb{E}\left[\sum_{u \in B_v^{(i)}} X_{v,u}\right] < L+1$. Via the Chernoff bound,

$$\mathbb{P}\left[|\Gamma^{(i)}(v)| > (1 + 3c \log n) \cdot L\right] \leq e^{-\frac{3c \log n \cdot L}{3}} = n^{-\Theta(c)}$$

for any constant $c > 0$. □

Lemma 3. *With high probability, for every v such that $|B_v| = \Omega(\log n)$ (i.e., $\mathbb{E}\left[|\Gamma(v)|\right] = \Omega(\log n)$), at least a $1/3$-fraction of the buckets $\{B_v^{(i)}\}_{i \in [|B_v|]}$ are non-empty.*

Proof. For $i < |B_v|$, since $\mathbb{E}\left[|\Gamma^{(i)}(v)|\right] = \mathbb{E}\left[\sum_{u \in B_v^{(i)}} X_{v,u}\right] > L-1$, we bound the

probability that $B_v^{(i)}$ is empty:

$$\mathbb{P}[B_v^{(i)} \text{ is empty}] = \prod_{u \in B_v^{(i)}} (1 - p_{v,u}) \leq e^{-\sum_{u \in B_v^{(i)}} p_{v,u}} \leq e^{1-L} = c$$

for any arbitrary small constant c given sufficienty large constant L. Let T_i be the indicator for the event that $B_v^{(i)}$ is *not* empty, so $\mathbb{E}[T_i] \geq 1 - c$. By the Chernoff bound, the probability that less than $|B_v|/3$ buckets are non-empty is

$$\mathbb{P}\left[\sum_{i \in [|B_v|]} T_i < \frac{|B_v|}{3}\right] < \mathbb{P}\left[\sum_{i \in [|B_v|-1]} T_i < \frac{|B_v|-1|}{2}\right] \leq e^{-\Theta(|B_v|-1)} = n^{-\Omega(1)}$$

as $|B_v| = \Omega(\log n)$ by assumption. □

3.3.2 Filling a bucket

We consider buckets to be in two possible states – filled or unfilled. Initially, all buckets are considered unfilled. In our algorithm we will maintain, for each bucket $B_v^{(i)}$, the set $P_v^{(i)}$ of known neighbors of u in bucket $B_v^{(i)}$; this is a refinement of the set P_v in Section 3.2. We define the behaviors of the procedure FILL(v, i) as follows. When invoked on an unfilled bucket $B_v^{(i)}$, FILL(v, i) performs the following tasks:

- decide whether each vertex $u \in B_v^{(i)}$ is a neighbor of v (implicitly setting $\mathbf{A}[v][u]$ to 1 or 0) unless $X_{v,u}$ is already decided; in other words, update $P_v^{(i)}$ to $\Gamma^{(i)}(v)$
- mark $B_v^{(i)}$ as filled.

For the sake of presentation, we postpone our description of the implementation of FILL to Section 3.4. For now, let us use FILL as a black-box operation.

3.3.3 Putting it all together: RANDOM-NEIGHBOR queries

Consider Algorithm 3 for generating a random neighbor via rejection sampling, in a rather similar overall framework as the simple implementation in Section 3.1. For simplicity, throughout the analysis, we assume $|B_v| = \Omega(\log n)$; otherwise, invoke FILL(v, i) for all $i \in [|B_v|]$ to obtain the entire neighbor list $\Gamma(v)$. This does not affect the analysis because

we will soon bound the number of calls that Algorithm 3 makes to FILL by $O(\log n)$ (in expectation) for $|B_v| = \Omega(\log n)$.

To obtain a random neighbor, we first choose a bucket $B_v^{(i)}$ uniformly at random. If the bucket is not yet filled, we invoke FILL(v, i) and fill this bucket. Then, we *accept* the sampled bucket for generating our random neighbor with probability proportional to $|P_v^{(i)}|$. More specifically, let $M = \Theta(\log n)$ be the upper bound on the maximum number of neighbors in any bucket, as derived in Lemma 2; we accept this bucket with probability $|P_v^{(i)}|/M$, which is well-defined (i.e., does not exceed 1) with high probability. (Note that if $P_v^{(i)} = \emptyset$, we remove i from the pool, then repeat as usual.) If we choose to accept this bucket, we return a random neighbor from $P_v^{(i)}$. Otherwise, *reject* this bucket and repeat the process again.

Algorithm 3 Bucketing Generator

```
procedure RANDOM-NEIGHBOR(v)
    R ← [|B_v|]
    repeat
        sample i ∈ R u.a.r.
        if B_v^(i) is not filled then
            FILL (v, i)
        if |P_v^(i)| > 0 then
            with probability |P_v^(i)| / M
                sample u ∈ P_v^(i) u.a.r
                return u
        else
            R ← R \ {i}
    until R = ∅
    return ⊥
```

Since the returned value is always a member of $P_v^{(i)}$, a valid neighbor is always returned. Further, i is removed from R only if $B_v^{(i)}$ does not contain any neighbors. So, if v has any neighbor, RANDOM-NEIGHBOR does not return $\perp$. We now proceed to showing the correctness of the algorithm and bound the number of iterations required for the rejection sampling process.

Lemma 4. *Algorithm 3 returns a uniformly random neighbor of vertex v.*

Proof. It suffices to show that the probability that any neighbor in $\Gamma(v)$ is return with uniform positive probability, within the same iteration. Fix a single iteration and consider a vertex $u \in P_v^{(i)}$: we compute the probability that u is accepted. The probability that i is picked is $1/|R|$, the probability that $B_v^{(i)}$ is accepted is $|P_v^{(i)}|/M$, and the probability that u is chosen among $P_v^{(i)}$ is $1/|P_v^{(i)}|$. Hence, the overall probability of returning u in a single iteration of the loop is $1/(|R| \cdot M)$, which is positive and independent of u. Therefore, each vertex is returned with the same probability. □

Lemma 5. *Algorithm 3 terminates in $\mathcal{O}(\log n)$ iterations in expectation, or $\mathcal{O}(\log^2 n)$ iterations with high probability.*

Proof. Following the analysis above, the probability that some vertex from $P_v^{(i)}$ is accepted in an iteration is at least $1/(|R| \cdot M)$. From Lemma 3, a $(1/3)$-fraction of the buckets are non-empty (with high probability), so the probability of choosing a non-empty bucket is at least $1/3$. Further, $M = \Theta(\log n)$ by Lemma 2. Hence, the success probability of each iteration is at least $1/(3M) = \Omega(1/\log n)$. Thus, with high probability, the number of iterations required is $O(\log^2 n)$ with high probability. □

3.4 Implementation of FILL

Lastly, we describe the implementation of the FILL procedure, employing the approach of skipping non-neighbors, as developed for Algorithm 2. We aim to simulate the following process: perform coin-tosses $C_{v,u}$ with probability $p_{v,u}$ for every $u \in B_v^{(i)}$ and update $\mathbf{A}[v][u]$'s according to these coin-flips unless they are decided (i.e., $\mathbf{A}[v][u] \neq \phi$). We directly generate a sequence of u's where the coins $C_{v,u} = 1$, then add u to P_v and vice versa if $X_{v,u}$ has not previously been decided. Thus, once $B_v^{(i)}$ is filled, we will obtain $P_v^{(i)} = \Gamma^{(i)}(v)$ as desired.

Algorithm 4 Sampling in a Bucket

procedure FILL(v, i)
  $(a, b) \leftarrow B_j^{(i)}$
  repeat
    sample $u \sim \mathsf{F}(v, a, b)$
    $B_u^{(j)} \leftarrow u$'s bucket containing v
    if $B_u^{(j)}$ is not filled **then**
      $P_v^{(i)} \leftarrow P_v^{(i)} \cup \{u\}$
      $P_u^{(j)} \leftarrow P_u^{(j)} \cup \{v\}$
    $a \leftarrow u$
  until $a \geq b$
  mark $B_u^{(j)}$ as filled

As discussed in Section 3.2, while we have recorded all occurrences of $\mathbf{A}[v][u] = 1$ in $P_v^{(i)}$, we need and efficient way of checking whether $\mathbf{A}[v][u] = 0$ or ϕ. In Algorithm 2, **last** serves this purpose by showing that $\mathbf{A}[v][u]$ for all $u \leq \mathbf{last}[v]$ are decided as shown in Lemma 1. Here instead, with our bucket structure, we maintain a single bit marking whether each bucket is filled or unfilled: a filled bucket implies that $\mathbf{A}[v][u]$ for all $u \in B_v^{(i)}$ are decided. The bucket structure along with mark bits, unlike **last**, are capable of handling intermittent ranges of intervals, namely buckets, which is sufficient for our purpose, as shown in the following lemma. This yields the implementation Algorithm 4 for the FILL

procedure fulfilling the requirement previously given in Section 3.3.2.

Lemma 6. *The data structures $P_v^{(i)}$'s and the bucket marking bits together provide a succinct representation of* $\mathbf{A}$ *as long as modifications to* $\mathbf{A}$ *are performed solely by the* FILL *operation in Algorithm 4. In particular, let $u \in B_v^{(i)}$ and $v \in B_u^{(j)}$. Then, $\mathbf{A}[v][u] = 1$ if and only if $u \in P_v^{(i)}$. Otherwise, $\mathbf{A}[v][u] = 0$ when at least one of $B_v^{(i)}$ or $B_u^{(j)}$ is marked as* filled. *In all remaining cases, $\mathbf{A}[v][u] = \phi$.*

Proof. The condition for $\mathbf{A}[v][u] = 1$ still holds by constuction. Otherwise, observe that $\mathbf{A}[v][u]$ becomes decided precisely during a FILL(v, i) or a FILL(u, j) operation, which thereby marks one of the corresponding buckets as filled. □

Note that $P_v^{(i)}$'s, maintained by our generator, are initially empty but may not still be empty at the beginning of the FILL function call. These $P_v^{(i)}$'s are again instantiated and stored in a dictionary once they become non-empty. Further, observe that the coin-flips are simulated independently of the state of $P_v^{(i)}$, so the number of iterations of Algorithm 4 is the same as the number of coins $C_{v,u} = 1$ which is, in expectation, a constant (namely $\sum_{u \in B_v^{(i)}} \mathbb{P}[C_{v,u} = 1] = \sum_{u \in B_v^{(i)}} p_{v,u} \leq L + 1$).

By tracking the resource required by Algorithm 4 we obtain the following lemma; note that "additional space" refers to the enduring memory that the generator must allocate and keep even after the execution, not its computation memory. The $\log n$ factors in our complexities are required to perform binary-search for the range of $B_v^{(i)}$, or for the value u from the CDF of $\mathsf{F}(u, a, b)$, and to maintain the ordered sets $P_v^{(i)}$ and $P_u^{(j)}$.

Lemma 7. *Each execution of Algorithm 4 (the* FILL *operation) on an* unfilled *bucket $B_v^{(i)}$, in expectation:*

- *terminates within $\mathcal{O}(1)$ iterations (of its* **repeat** *loop);*
- *computes $\mathcal{O}(\log n)$ quantities of $\prod_{u \in [a,b]}(1 - p_{v,u})$ and $\sum_{u \in [a,b]} p_{v,u}$ each;*
- *aside from the above computations, uses $\mathcal{O}(\log n)$ time, $\mathcal{O}(1)$ random N-bit words, and $\mathcal{O}(1)$ additional space.*

Observe that the number of iterations required by Algorithm 4 only depends on its random coin-flips and independent of the state of the algorithm. Combining with Lemma 5, we finally obtain polylogarithimc resource bound for our implementation of RANDOM-NEIGHBOR.

Corollary 1. *Each execution of Algorithm 3 (the* RANDOM-NEIGHBOR *query), with high probability,*

- *terminates within* $\mathcal{O}(\log^2 n)$ *iterations (of its* **repeat** *loop);*
- *computes* $\mathcal{O}(\log^3 n)$ *quantities of* $\prod_{u\in[a,b]}(1-p_{v,u})$ *and* $\sum_{u\in[a,b]} p_{v,u}$ *each;*
- *aside from the above computations, uses* $\mathcal{O}(\log^3 n)$ *time,* $\mathcal{O}(\log^2 n)$ *random* N*-bit words, and* $\mathcal{O}(\log^2 n)$ *additional space.*

Extension to other query types. We finally extend our algorithm to support other query types as follows.

- VERTEX-PAIR(u,v): We simply need to make sure that Lemma 6 holds, so we first apply FILL(u, j) on bucket $B_u^{(j)}$ containing v (if needed), then answer accordingly.
- NEXT-NEIGHBOR(v): We maintain **last**, and keep invoking FILL until we find a neighbor. Recall that by Lemma 3, the probability that a particular bucket is empty is a small constant. Then with high probability, there exists no $\omega(\log n)$ *consecutive* empty buckets $B_v^{(i)}$'s for any vertex v, and thus NEXT-NEIGHBOR only invokes up to $\mathcal{O}(\log n)$ calls to FILL.

We summarize the results so far with through the following theorem.

Theorem 1. *Under the assumption of*

1. *perfect-precision arithmetic, including the generation of random real numbers in* $[0, 1)$*, and*
2. *the quantities* $\prod_{u=a}^{b}(1-p_{v,u})$ *and* $\sum_{u=a}^{b} p_{v,u}$ *of the random graph family can be computed with perfect precision in logarithmic time, space and random bits,*

there exists a local-access generator for the random graph family that supports RANDOM-NEIGHBOR, VERTEX-PAIR *and* NEXT-NEIGHBOR *queries that uses polylogarithmic running time, additional space, and random words per query.*

Between these two assumptions, we first remove the assumption of perfect-precision arithmetic in the upcoming Section 3.5. Later in Section 4, we show applications of our generator to the $G(n, p)$ model, and the Stochastic Block model under random community

assignment, by providing formulas and by constructing data structures for computing the quantities specified in the second assumption, respectively.

3.5 Removing the Perfect-Precision Arithmetic Assumption

In this section we remove the prefect-precision arithmetic assumption. Instead, we only assume that it is possible to compute $\prod_{u=a}^{b}(1-p_{v,u})$ and $\sum_{u=a}^{b} p_{v,u}$ to N-bit precision, as well as drawing a random N-bit word, using polylogarithmic resources. Here we will focus on proving that the family of the random graph we generate via our procedures is statistically close to that of the desired distribution. The main technicality of this lemma arises from the fact that, not only the generator is randomized, but the agent interacting with the generator may choose his queries arbitrarily (or adversarially): our proof must handle any sequence of random choices the generator makes, and any sequence of queries the agent may make.

Observe that the distribution of the graphs constructed by our generator is governed entirely by the samples u drawn from $\mathsf{F}(v,a,b)$ in Algorithm 4. By our assumption, the CDF of any $\mathsf{F}(v,a,b)$ can be efficiently computed from $\prod_{u=a}^{u'}(1-p_{v,u})$, and thus sampling with $\frac{1}{\mathrm{poly}(n)}$ error in the L_1-distance requires a random N-bit word and a binary-search in $\mathcal{O}(\log(b-a+1)) = \mathcal{O}(\log n)$ iterations. Using this crucial fact, we prove our lemma that removes the perfect-precision arithmetic assumption.

Lemma 8. *If Algorithm 4 (the* FILL *operation) is repeatedly invoked to construct a graph G by drawing the value u for at most S times in total, each of which comes from some distribution $\mathsf{F}'(v,a,b)$ that is ϵ-close in L_1-distance to the correct distribution $\mathsf{F}(v,a,b)$ that perfectly generates the desired distribution G over all graphs, then the distribution G' of the generated graph G is (ϵS)-close to G in the L_1-distance.*

Proof. For simplicity, assume that the algorithm generates the graph to completion according to a sequence of up to n^2 distinct buckets $\mathcal{B} = \langle B_{v_1}^{(u_1)}, B_{v_2}^{(u_2)}, \ldots \rangle$, where each $B_{v_i}^{(u_i)}$ specifies the unfilled bucket in which any query instigates a FILL function call. Define an

internal state of our generator as the triplet $s = (k, u, \mathbf{A})$, representing that the algorithm is currently processing the k^{th} FILL, in the iteration (the **repeat** loop of Algorithm 4) with value u, and have generated $\mathbf{A}$ so far. Let $t_{\mathbf{A}}$ denote the *terminal state* after processing all queries and having generated the graph $G_{\mathbf{A}}$ represented by $\mathbf{A}$. We note that $\mathbf{A}$ is used here in the analysis but not explicitly maintained; further, it reflects the changes in every iteration: as u is updated during each iteration of FILL, the cells $\mathbf{A}[v][u'] = \phi$ for $u' < u$ (within that bucket) that has been skipped are also updated to 0.

Let $\mathcal{S}$ denote the set of all (internal and terminal) states. For each state s, the generator samples u from the corresponding $\mathsf{F}'(v, a, b)$ where $\|\mathsf{F}(v, a, b) - \mathsf{F}'(v, a, b)\|_1 \leq \epsilon = \frac{1}{\text{poly}(n)}$, then moves to a new state according to u. In other words, there is an induced pair of collection of distributions over the states: $(\mathcal{T}, \mathcal{T}')$ where $\mathcal{T} = \{\mathsf{T}_s\}_{s \in \mathcal{S}}, \mathcal{T}' = \{\mathsf{T}'_s\}_{s \in \mathcal{S}}$, such that $\mathsf{T}_s(s')$ and $\mathsf{T}'_s(s')$ denote the probability that the algorithm advances from s to s' by using a sample from the correct $\mathsf{F}(v, a, b)$ and from the approximated $\mathsf{F}'(v, a, b)$, respectively. Consequently, $\|\mathsf{T}_s - \mathsf{T}'_s\|_1 \leq \epsilon$ for every $s \in \mathcal{S}$.

The generator begins with the initial (internal) state $s_0 = (1, 0, \mathbf{A}_\phi)$ where all cells of $\mathbf{A}_\phi$ are ϕ's, goes through at most $S = O(n^3)$ other states (as there are up to n^2 values of k and $O(n)$ values of u), and reach some terminal state $t_{\mathbf{A}}$, generating the entire graph in the process. Let $\pi = \langle s_0^\pi = s_0, s_1^\pi, \ldots, s_{\ell(\pi)}^\pi = t_{\mathbf{A}} \rangle$ for some $\mathbf{A}$ denote a sequence ("path") of up to $S + 1$ states the algorithm proceeds through, where $\ell(\pi)$ denote the number of transitions it undergoes. For simplicity, let $T_{t_{\mathbf{A}}}(t_{\mathbf{A}}) = 1$, and $T_{t_{\mathbf{A}}}(s) = 0$ for all state $s \neq t_{\mathbf{A}}$, so that the terminal state can be repeated and we may assume $\ell(\pi) = S$ for every π. Then, for the correct transition probabilities described as $\mathcal{T}$, each π occurs with probability $q(\pi) = \prod_{i=1}^{S} \mathsf{T}_{s_{i-1}}(s_i)$, and thus $\mathsf{G}(G_{\mathbf{A}}) = \sum_{\pi : s_S^\pi = t_{\mathbf{A}}} q(\pi)$.

Let $\mathcal{T}^{\min} = \{\mathsf{T}_s^{\min}\}_{s \in \mathcal{S}}$ where $\mathsf{T}_s^{\min}(s') = \min\{\mathsf{T}_s(s'), \mathsf{T}'_s(s')\}$, and note that each $\mathsf{T}_s^{\min}$ is not necessarily a probability distribution. Then, $\sum_{s'} \mathsf{T}_s^{\min}(s') = 1 - \|\mathsf{T}_s - \mathsf{T}'_s\|_1 \geq 1 - \epsilon$. Define $q', q^{\min}, \mathsf{G}'(G_{\mathbf{A}}), \mathsf{G}^{\min}(G_{\mathbf{A}})$ analogously, and observe that $q^{\min}(\pi) \leq \min\{q(\pi), q'(\pi)\}$ for every π, so $\mathsf{G}^{\min}(G_{\mathbf{A}}) \leq \min\{\mathsf{G}(G_{\mathbf{A}}), \mathsf{G}'(G_{\mathbf{A}})\}$ for every $G_{\mathbf{A}}$ as well. In other words, $q^{\min}(\pi)$ lower bounds the probability that the algorithm, drawing samples from the correct distributions or the approximated distributions, proceeds through states of π; consequently, $\mathsf{G}^{\min}(G_{\mathbf{A}})$ lower bounds the probability that the algorithm generates the graph $G_{\mathbf{A}}$.

Next, consider the probability that the algorithm proceeds through the prefix $\pi_i = \langle s_0^\pi, \ldots, s_i^\pi \rangle$ of π. Observe that for $i \geq 1$,

$$\sum_\pi q^{\min}(\pi_i) = \sum_\pi q^{\min}(\pi_{i-1}) \cdot \mathsf{T}^{\min}_{s^\pi_{i-1}}(s^\pi_i) = \sum_{s,s'} \sum_{\pi : s^\pi_{i-1}=s, s^\pi_i = s'} q^{\min}(\pi_{i-1}) \cdot \mathsf{T}^{\min}_s(s')$$

$$= \sum_{s'} \mathsf{T}^{\min}_s(s') \cdot \sum_s \sum_{\pi : s^\pi_{i-1}=s} q^{\min}(\pi_{i-1}) \geq (1-\epsilon) \sum_\pi q^{\min}(\pi_{i-1}).$$

Roughly speaking, at least a factor of $1 - \epsilon$ of the "agreement" between the distributions over states according to $\mathcal{T}$ and $\mathcal{T}'$ is necessarily conserved after a single sampling process. As $\sum_\pi q^{\min}(\pi_0) = 1$ because the algorithm begins with $s_0 = (1, 0, \mathbf{A}_\phi)$, by an inductive argument we have $\sum_\pi q^{\min}(\pi) = \sum_\pi q^{\min}(\pi_S) \geq (1-\epsilon)^S \geq 1 - \epsilon S$. Hence, $\sum_{G_\mathbf{A}} \min\{\mathsf{G}(G_\mathbf{A}), \mathsf{G}'(G_\mathbf{A})\} \geq \sum_{G_\mathbf{A}} \mathsf{G}^{\min}(G_\mathbf{A}) \geq 1 - \epsilon S$, implying that $\|\mathsf{G} - \mathsf{G}'\|_1 \leq \epsilon S$, as desired. In particular, by substituting $\epsilon = \frac{1}{\mathrm{poly}(n)}$ and $S = O(n^3)$, we have shown that Algorithm 4 only creates a $\frac{1}{\mathrm{poly}(n)}$ error in the L_1-distance. □

We remark that RANDOM-NEIGHBOR queries also require that the returned edge is drawn from a distribution that is close to a uniform one, but this requirement applies only *per query* rather then over the entire execution of the generator. Hence, the error due to the selection of a random neighbor may be handled separately from the error for generating the random graph; its guarantee follows straightforwardly from a similar analysis.

Chapter 4

Applications to Erdös-Rényi Model and Stochastic Block Model

In this section we demonstrate the application of our techniques to two well known, and widely studied models of randon graphs. That is, as required by Theorem 1, we must provide a method for computing the quantities $\prod_{u=a}^{b}(1-p_{v,u})$ and $\sum_{u=a}^{b} p_{v,u}$ of the desired random graph families in logarithmic time, space and random bits. Our first implementation focuses on the well known Erdös-Rényi model – $G(n,p)$: in this case, $p_{v,u} = p$ is uniform and our quantities admit closed-form formulas.

Next, we focus on the Stochastic Block model with randomly-assigned communities. Our implementation assigns each vertex to a community in $\{C_1, \ldots, C_r\}$ identically and independently at random, according to some given distribution R over the communities. We formulate a method of sampling community assignments locally. This essentially allows us to sample from the *multivariate hypergeometric distribution*, using $\mathrm{poly}(\log n)$ random bits, which may be of independent interest. We remark that, as our first step, we sample for the number of vertices of each community. That is, our construction can alternatively support the community assignment where the number of vertices of each community is given, under the assumption that the *partition* of the vertex set into communities is chosen uniformly at random.

4.1 Erdös-Rényi Model

As $p_{v,u} = p$ for all edges $\{u, v\}$ in the Erdös-Rényi $G(n, p)$ model, we have the closed-form formulas $\prod_{u=a}^{b}(1-p_{v,u}) = (1-p)^{b-a+1}$ and $\sum_{u=a}^{b} p_{v,u} = (b-a+1)p$, which can be computed in constant time according to our assumption, yielding the following corollary.

Corollary 2. *The final algorithm in Section 3 locally generates a random graph from the Erdös-Rényi $G(n,p)$ model using $\mathcal{O}(\log^3 n)$ time, $\mathcal{O}(\log^2 n)$ random N-bit words, and $\mathcal{O}(\log^2 n)$ additional space per query with high probability.*

We remark that there exists an alternative approach that picks $F \sim \mathsf{F}(v, a, b)$ directly via a closed-form formula $a + \lceil \frac{\log U}{\log(1-p)} \rceil$ where U is drawn uniformly from $[0, 1)$, rather than binary-searching for U in its CDF. Such an approach may save some $\mathrm{poly}(\log n)$ factors in the resources, given the prefect-precision arithmetic assumption. This usage of the log function requires $\Omega(n)$-bit precision, which is not applicable to our computation model.

While we are able to generate our random graph on-the-fly supporting all three types of queries, our construction still only requires $\mathcal{O}(m + n)$ space (N-bit words) in total at any state; that is, we keep $\mathcal{O}(n)$ words for **last**, $\mathcal{O}(1)$ words per neighbor in P_v's, and one marking bit for each bucket (where there can be up to $m + n$ buckets in total). Hence, our memory usage is nearly optimal for the $G(n, p)$ model:

Corollary 3. *The final algorithm in Section 3 can generate a complete random graph from the Erdös-Rényi $G(n,p)$ model using overall $\tilde{\mathcal{O}}(n+m)$ time, random bits and space, which is $\tilde{\mathcal{O}}(pn^2)$ in expectation. This is optimal up to $\mathcal{O}(\mathrm{poly}(\log n))$ factors.*

4.2 Stochastic Block model

For the Stochastic Block model, each vertex is assigned to some community C_i, $i \in [r]$. By partitioning the product by communities, we may rewrite the desired formulas, for $v \in C_i$, as $\prod_{u=a}^{b}(1 - p_{v,u}) = \prod_{j=1}^{r}(1 - p_{i,j})^{|[a,b] \cap C_j|}$ and $\sum_{u=a}^{b} p_{v,u} = \sum_{j=1}^{r} |[a, b] \cap C_j| \cdot p_{i,j}$. Thus, it is sufficient to design a data structure, or a *generator*, that draws a community assignment for the vertex set according to the given distribution R. This data structure

should be able to efficiently count the number of occurrences of vertices of each community in any contiguous range, namely the value $|[a, b] \cap C_j|$ for each $j \in [r]$. To this end, we use the following lemma, yielding the generator for the Stochastic Block model that uses $O(r \operatorname{poly}(\log n))$ resources per query.

Theorem 2. *There exists a data structure (generator) that samples a community for each vertex independently at random from* R *with* $\frac{1}{\operatorname{poly}(n)}$ *error in the* L_1*-distance, and supports queries that ask for the number of occurrences of vertices of each community in any contiguous range, using* $O(r \operatorname{poly}(\log n))$ *time, random* N*-bit words and additional space per query. Further, this data structure may be implemented in such a way that requires no overhead for initialization.*

Corollary 4. *The final algorithm in Section 3 generates a random graph from the Stochastic Block model with randomly-assigned communities using* $O(r \operatorname{poly}(\log n))$ *time, random* N*-bit words, and additional space per query with high probability.*

We provide the full details of the construction in the following Section 4.2.1. Our construction extends upon a similar generator in the work of [19] which only supports $r = 2$. Our overall data structure is a balanced binary tree, where the root corresponds to the entire range of indices $\{1, \ldots, n\}$, and the children of each vertex corresponds to each half of the parent's range. Each node[1] holds the number of vertices of each community in its range. The tree initially contains only the root, with the number of vertices of each community sampled according to the multinomial distribution[2] (for n samples (vertices) from the probability distribution R). The children are only generated top-down on an as-needed basis according to the given queries. The technical difficulties arise when generating the children, where one needs to sample "half" of the counts of the parent from the correct marginal distribution. To this end, we show how to sample such a count as described in the statement below. Namely, we provide an algorithm for sampling from the *multivariate hypergeometric distribution.*

[1] For clarity, "vertex" is only used in the generated graph, and "node" is only used in the internal data structures of the generator.

[2] See e.g., section 3.4.1 of [25]

4.2.1 Sampling from the Multivariate Hypergeometric Distribution

Consider the following random experiment. Suppose that we have an urn containing $B \leq n$ marbles (representing vertices), each occupies one of the r possible colors (representing communities) represented by an integer from $[r]$. The number of marbles of each color in the urn is known: there are C_k indistinguishable marbles of color $k \in [r]$, where $C_1 + \cdots + C_r = B$. Consider the process of drawing $\ell \leq B$ marbles from this urn *without replacement*. We would like to sample how many marbles of each color we draw.

More formally, let $\mathbf{C} = \langle c_1, \ldots, c_r \rangle$, then we would like to (approximately) sample a vector $\mathbf{S}_\ell^{\mathbf{C}}$ of r non-negative integers such that

$$\Pr[\mathbf{S}_\ell^{\mathbf{C}} = \langle s_1, \ldots, s_r \rangle] = \frac{\binom{C_1}{s_1} \cdot \binom{C_2}{s_2} \cdots \binom{C_r}{s_r}}{\binom{B}{C_1+C_2+\cdots+C_r}}$$

where the distribution is supported by all vectors satisfying $s_k \in \{0, \ldots, C_k\}$ for all $k \in [r]$ and $\sum_{k=1}^{r} s_k = \ell$. This distribution is referred to as the *multivariate hypergeometric distribution.*

The sample $\mathbf{S}_\ell^{\mathbf{C}}$ above may be generated easily by simulating the drawing process, but this may take $\Omega(\ell)$ iterations, which have linear dependency in n in the worst case: $\ell = \Theta(B) = \Theta(n)$. Instead, we aim to generate such a sample in $O(r \operatorname{poly}(\log n))$ time with high probability. We first make use of the following procedure from [19].

Lemma 9. *Suppose that there are T marbles of color 1 and $B - T$ marbles of color 2 in an urn, where $B \leq n$ is even. There exists an algorithm that samples $\langle s_1, s_2 \rangle$, the number of marbles of each color appearing when drawing $B/2$ marbles from the urn without replacement, in $O(\operatorname{poly}(\log n))$ time and random words. Specifically, the probability of sampling a specific pair $\langle s_1, s_2 \rangle$ where $s_1 + s_2 = T$ is approximately $\binom{B/2}{s_1}\binom{B/2}{T-s_1}/\binom{B}{T}$ with error of at most n^{-c} for any constant $c > 0$.*

In other words, the claim here only applies to the two-color case, where we sample the number of marbles when drawing exactly half of the marbles from the entire urn ($r = 2$ and $\ell = B/2$). First we generalize this claim to handle any desired number of drawn marbles ℓ (while keeping $r = 2$).

Lemma 10. *Given C_1 marbles of color 1 and $C_2 = B - C_1$ marbles of color 2, there exists an algorithm that samples $\langle s_1, s_2 \rangle$, the number of marbles of each color appearing when drawing l marbles from the urn without replacement, in $O(\mathrm{poly}(\log n))$ time and random words.*

Proof. For the base case where $B = 1$, we trivially have $\mathbf{S}_1^{\mathbf{C}} = \mathbf{C}$ and $\mathbf{S}_0^{\mathbf{C}} = \vec{\mathbf{0}}$. Otherwise, for even B, we apply the following procedure.

- If $\ell \leq B/2$, generate $\mathbf{C}' = \mathbf{S}_{B/2}^{\mathbf{C}}$ using Claim 9.
  - If $\ell = B/2$ then we are done.
  - Else, for $\ell < B/2$ we recursively generate $\mathbf{S}_{\ell}^{\mathbf{C}'}$.
- Else, for $\ell > B/2$, we generate $\mathbf{S}_{B-\ell}^{\mathbf{C}'}$ as above, then output $\mathbf{C} - \mathbf{S}_{B-\ell}^{\mathbf{C}'}$.

On the other hand, for odd B, we simply simulate drawing a single random marble from the urn before applying the above procedure on the remaining $B - 1$ marbles in the urn. That is, this process halves the domain size B in each step, requiring $\log B$ iterations to sample $\mathbf{S}_{\ell}^{\mathbf{C}}$. □

Lastly we generalize to support larger r.

Theorem 3. *Given B marbles of r different colors, such that there are C_i marbles of color i, there exists an algorithm that samples $\langle s_1, s_2, \cdots, s_r \rangle$, the number of marbles of each color appearing when drawing l marbles from the urn without replacement, in $O(r \cdot \mathrm{poly}(\log n))$ time and random words.*

Proof. Observe that we may reduce $r > 2$ to the two-color case by sampling the number of marbles of the first color, collapsing the rest of the colors together. Namely, define a pair $\hat{\mathbf{C}} = \langle C_1, C_2 + \cdots + C_r \rangle$, then generate $\mathbf{S}_{\ell}^{\hat{\mathbf{C}}} = \langle s_1, s_2 + \ldots + s_r \rangle$ via the above procedure. At this point we have obtained the first entry s_1 of the desired $\mathbf{S}_{\ell}^{\mathbf{C}}$. So it remains to generate the number of marbles of each color from the remaining $r - 1$ colors in $\ell - s_1$ remaining draws. In total, we may generate $\mathbf{S}_{\ell}^{\mathbf{C}}$ by performing r iterations of the two-colored case. The error in the L_1-distance may be established similarly to the proof of Lemma 8. □

4.2.2 Data structure

We now show that Theorem 3 may be used in order to create the following data structure. Recall that R denote the given distribution over integers $[r]$ (namely, the random distribution of communities for each vertex). Our data structure generates and maintains random variables $X_1, \ldots, X_n$, each of which is drawn independently at random from R: X_i denotes the community of vertex i. Then given a pair (i, j), it returns the vector $\mathbf{C}(i, j) = \langle c_1, \ldots, c_r \rangle$ where c_k counts the number of variables $X_i, \ldots, X_j$ that takes on the value k. Note that we may also find out X_i by querying for (i, i) and take the corresponding index.

We maintain a complete binary tree whose leaves corresponds to indices from $[n]$. Each node represents a range and stores the vector $\mathbf{C}$ for the corresponding range. The root represents the entire range $[n]$, which is then halved in each level. Initially the root samples $\mathbf{C}(1, n)$ from the multinomial distribution according to R (see e.g., Section 3.4.1 of [25]). Then, the children are generated on-the-fly using the lemma above. Thus, each query can be processed within $O(r \operatorname{poly}(\log n))$ time, yielding Theorem 2. Then, by embedding the information stored by the data structure into the state (as in the proof of Lemma 8), we obtain the desired Corollary 4.

Chapter 5

Local-Access Generators for Random Directed Graphs

In this section, we consider Kleinberg's Small-World model [24, 29] where the probability that a *directed* edge (u, v) exists is $\min\{c/(\text{DIST}(u, v))^2, 1\}$. Here, $\text{DIST}(u, v)$ is the Manhattan distance between u and v on a $\sqrt{n} \times \sqrt{n}$ grid. We begin with the case where $c = 1$, then generalize to different values of $c = \log^{\pm\Theta(1)}(n)$. We aim to support ALL-NEIGHBORS queries using $\text{poly}(\log n)$ resources. This returns the entire list of out-neighbors of v.

5.1 Generator for $c = 1$

Observe that since the graphs we consider here are directed, the answers to the ALL-NEIGHBOR queries are all independent: each vertex may determine its out-neighbors independently. Given a vertex v, we consider a partition of all the other vertices of the graph into sets $\{\Gamma_1^v, \Gamma_2^v, \ldots\}$ by distance: $\Gamma_k^v = \{u : \text{DIST}(v, u) = k\}$ contains all vertices at a distance k from vertex v. Observe that $|\Gamma_k^v| \leq 4k = O(k)$. Then, the expected number of edges from v to vertices in Γ_k^v is therefore $|\Gamma_k^v| \cdot 1/k^2 = O(1/k)$. Hence, the expected degree of v is at most $\sum_{k=1}^{2(\sqrt{n}-1)} O(1/k) = O(\log n)$. It is straightforward to verify that this bound holds with high probability (use Hoeffding's inequality). Since the degree of v is small, in this model we can afford to perform ALL-NEIGHBORS queries instead of NEXT-NEIGHBOR queries using an additional $\text{poly}(\log n)$ resources.

Nonetheless, internally in our generator, we sample for our neighbors one-by-one similarly to how we process NEXT-NEIGHBOR queries. We perform our sampling in two phases. In the first phase, we sample a distance d, such that the next neighbor closest to v is at distance d. We maintain $\mathbf{last}[v]$ to be the last sampled distance. In the second phase, we sample all neighbors of v at distance d, under the assumption that there must be at least one such neighbor. For simplicity, we sample these neighbors as if there are *full* $4d$ vertices at distance d from v: some sampled neighbors may lie outside our $\sqrt{n} \times \sqrt{n}$ grid, which are simply discarded. As the running time of our generator is proportional to the number of generated neighbors, then by the bound on the number of neighbors, this assumption does not asymptotically worsen the performance of the generator.

5.1.1 Phase 1: Sample the distance D

Let $a = \mathbf{last}[v] + 1$, and let $\mathsf{D}(a)$ to denote the probability distribution of the distance where the next closest neighbor of v is located, or $\perp$ if there is no neighbor at distance at most $2(\sqrt{n} - 1)$. That is, if $D \sim \mathsf{D}(a)$ is drawn, then we proceed to Phase 2 to sample all neighbors at distance D. We repeat the process by sampling the next distance from $\mathsf{D}(a+D)$ and so on until we obtain $\perp$, at which point we return our answers and terminate.

To sample the next distance, we perform a binary search: we must evaluate the CDF of $\mathsf{D}(a)$. The CDF is given by $\mathbb{P}[D \leq d]$ where $D \sim \mathsf{D}(a)$, the probability that there is *some* neighbor at distance at most d. As usual, we compute the probability of the negation: there is *no* neighbor at distance at most d. Recall that each distance i has exactly $|\Gamma_i^v| = 4i$ vertices, and the probability of a vertex $u \in \Gamma_i^v$ is not a neighbor is exactly $1 - 1/i^2$. So, the probability that there is no neighbor at distance i is $(1 - 1/i^2)^{4i}$. Thus, for $D \sim \mathsf{D}(a)$ and $d \leq 2(\sqrt{n} - 1)$,

$$\mathbb{P}[D \leq d] = 1 - \prod_{i=a}^{d} \left(1 - \frac{1}{i^2}\right) = 1 - \prod_{i=a}^{d} \left(\frac{(i-1)(i+1)}{i^2}\right)^{4i} = 1 - \left(\frac{(a-1)^a}{a^{a-1}} \cdot \frac{(d+1)^d}{d^{d+1}}\right)^4$$

where the product enjoys telescoping as the denominator $(i^2)^{4i}$ cancels with $(i^2)^{4(i-1)}$ and $(i^2)^{4(i+1)}$ in the numerators of the previous and the next term, respectively. This gives us a

closed form for the CDF, which we can compute with 2^{-N} additive error in constant time (by our computation model assumption). Thus, we may sample for the distance $D \sim \mathsf{D}(a)$ with $O(\log n)$ time and one random N-bit word.

5.1.2 Phase 2: Sampling neighbors at distance D

After sampling a distance D, we now have to sample all the neighbors at distance D. We label the vertices in Γ_D^v with unique indices in $\{1, \ldots, 4D\}$. Note that now each of the $4D$ vertices in Γ_D^v is a neighbor with probability $1/D^2$. However, by Phase 1, this is conditioned on the fact that there is at least one neighbor among the vertices in Γ_D^v, which may be difficult to sample when $1/D^2$ is very small. We can emulate this naïvely by repeatedly sampling a "block", composing of the $4D$ vertices in Γ_D^v, by deciding whether each vertex is a neighbor of v with uniform probability $1/D^2$ (i.e., $4D$ identical independent Bernoulli trials), and then discarding the entire block if it contains no neighbor. We repeat this process until we finally sample one block that contains at least one neighbor, and use this block as our output.

For the purpose of making the sampling process more efficient, we view this process differently. Let us imagine that we are given an infinite sequence of independent Bernoulli variables, each with bias $1/D^2$. We then divide the sequence into contiguous blocks of length $4D$ each. Our task is to find the *first* occurrence of success (a neighbor), then report the whole block hosting this variable.

This first occurrence of a successful Bernoulli trial is given by sampling from the geometric distribution, $X \sim \mathsf{Geo}(1/D^2)$. Since the vertices in each block are labeled by $1, \ldots, 4D$, then this first occurrence has label $X' = X \bmod 4D$. By sampling $X \sim \mathsf{Geo}(1/D^2)$, the first X' Bernoulli variables of this block is also implicitly determined. Namely, the vertices of labels $1, \ldots, X' - 1$ are non-neighbors, and that of label X' is a neighbor. The sampling for the remaining $4D - X'$ vertices can then be performed in the same fashion we sample for next neighbors in the $G(n, p)$ case: repeatedly find the next neighbor by sampling from $\mathsf{Geo}(1/D^2)$, until the index of the next neighbor falls beyond this block.

Thus at this point, we have sampled all neighbors in Γ_D^v. We can then update $\mathbf{last}[v] \leftarrow D$ and continue the process of larger distances. Sampling each neighbor takes $O(\log n)$ time and one random N-bit word; the resources spent sampling the distances is also bounded by that of the neighbors. As there are $O(\log n)$ neighbors with high probability, we obtain the following theorem.

Theorem 4. *There exists an algorithm that generates a random graph from Kleinberg's Small World model, where probability of including each directed edge (u, v) in the graph is $1/(\text{DIST}(u, v))^2$ where* DIST *denote the Manhattan distance, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *query with high probability.*

5.2 Generator for $c \neq 1$

Observe that to support different values of c in the probability function $c/(\text{DIST}(u, v))^2$, we do not have a closed-form formula for computing the CDF for Phase 1, whereas the process for Phase 2 remains unchanged. To handle the change in the probability distribution Phase 1, we consider the following, more general problem. Suppose that we have a process P that, one-by-one, provide occurrences of successes from the sequence of independent Bernoulli trials with success probabilities $\langle p_1, p_2, \ldots \rangle$. We show how to construct a process $\mathcal{P}^c$ that provide occurrences of successes from Bernoulli trials with success probabilities $\langle c \cdot p_1, c \cdot p_2, \ldots \rangle$ (truncated down to 1 as needed). For our application, we assume that c is given in N-bit precision, there are $O(n)$ Bernoulli trials, and we aim for an error of $\frac{1}{\text{poly}(n)}$ in the L_1-distance.

5.2.1 Case $c < 1$

We use rejection sampling in order to construct a new Bernoulli process.

Lemma 11. *Given a process $\mathcal{P}$ outputting the indices of successful Bernoulli trials with bias $\langle p_i \rangle$, there exists a process $\mathcal{P}^c$ outputting the indices of successful Bernoulli trials with bias $\langle c \cdot p_i \rangle$ where $c < 1$, using one additional N-bit word overhead for each answer of $\mathcal{P}$.*

Proof. Consider the following rejection sampling process to generating the Bernoulli trials. In addition to each Bernoulli variable X_i with bias p_i, we sample another coin-flip C_i with bias c. Set $Y_i = X_i \cdot C_i$, then $\mathbb{P}[Y_i = 1] = \mathbb{P}[X_i = 1] \cdot \mathbb{P}[C_i] = c \cdot p_i$, as desired. That is, we keep a success of a Bernoulli trial with probability c, or reject it with probability $1 - c$.

Now, we are already given the process $\mathcal{P}$ that "handles" X_i's, generating a sequence of indices i with $X_i = 1$. The new process $\mathcal{P}^c$ then only needs to handle the C_i's. Namely, for each i reported as success by $\mathcal{P}$, $\mathcal{P}^c$ flips a coin C_i to see if it should also report i, or discard it. As a result, $\mathcal{P}^c$ can generate the indices of successful Bernoulli trials using only one random N-bit word overhead for each answer from $\mathcal{P}$. □

Applying this reduction to the distance sampling in Phase 1, we obtain the following corollary.

Corollary 5. *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities $c/(\text{DIST}(u,v))^2$ where $c < 1$, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *query with high probability.*

5.2.2 Case $c > 1$

Since we aim to sample with larger probabilities, we instead consider making $k \cdot c$ independent copies of each process $\mathcal{P}$, where $k > 1$ is a positive integer. Intuitively, we hope that the probability that one of these process returns an index i will be at least $c \cdot p_i$, so that we may perform rejection sampling to decide whether to keep i or not. Unfortunately such a process cannot handle the case where $c \cdot p_i$ is large, notably when $c \cdot p_i > 1$ is truncated down to 1, while there is always a possibility that none of the processes return i.

Lemma 12. *Let $k > 1$ be a constant integer. Given a process $\mathcal{P}$ outputting the indices of successful Bernoulli trials with bias $\langle p_i \rangle$, there exists a process $\mathcal{P}^c$ outputting the indices of successful Bernoulli trials with bias $\langle \min\{c \cdot p_i, 1\} \rangle$ where $c > 1$ and $c \cdot p_i \leq 1 - \frac{1}{k}$ for every i, using one additional N-bit word overhead for each answer of $k \cdot c$ independent copies of $\mathcal{P}$.*

Proof. By applying the following form of Bernoulli's inequality, we have

$$(1-p_i)^{k\cdot c} \leq 1 - \frac{k\cdot c\cdot p_i}{1+(k\cdot c-1)\cdot p_i} = 1 - \frac{k\cdot c\cdot p_i}{1+k\cdot c\cdot p_i - p_i} \leq 1 - \frac{k\cdot c\cdot p_i}{1+(k-1)} = 1 - c\cdot p_i$$

That is, the probability that at least one of the generators report an index i is $1-(1-p_i)^{k\cdot c} \geq c\cdot p_i$, as required. Then, the process $\mathcal{P}^c$ simply reports i with probability $(c\cdot p_i)/(1-(1-p_i)^{k\cdot c})$ or discard i otherwise. Again, we only require N-bit of precision for each computation, and thus one random N-bit word suffices. □

In Phase 1, we may apply this reduction only when the condition $c\cdot p_i \leq 1-\frac{1}{k}$ is satisfied. For lower value of $p_i = 1/D^2$, namely for distance $D < \sqrt{c/(1-1/k)} = O(\sqrt{c})$, we may afford to sample the Bernoulli trials one-by-one as c is $\mathrm{poly}(\log n)$. We also note that the degree of each vertex is clearly bounded by $O(\log n)$ with high probability, as its expectation is scaled up by at most a factor of c. Thus, we obtain the following corollary.

Corollary 6. *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities* $c/(\mathrm{DIST}(u,v))^2$ *where* $c = \mathrm{poly}(\log n)$, *using* $O(\log^2 n)$ *time and random* N*-bit words per* ALL-NEIGHBORS *query with high probability.*

Appendix A

Further Analysis and Extensions of Algorithm 2

A.1 Performance Guarantee

This section is devoted to showing the following lemma that bounds the required resources per query of Algorithm 2. We note that we only require efficient computation of $\prod_{u\in[a,b]}(1-p_{v,u})$ (and not $\sum_{u\in[a,b]} p_{v,u}$), and that for the $G(n,p)$ model, the resources required for such computation is asymptotically negligible.

Theorem 5. *Each execution of Algorithm 2 (the* NEXT-NEIGHBOR *query), with high probability,*

- *terminates within $\mathcal{O}(\log n)$ iterations (of its* **repeat** *loop);*
- *computes $\mathcal{O}(\log^2 n)$ quantities of $\prod_{u\in[a,b]}(1-p_{v,u})$;*
- *aside from the above computations, uses $\mathcal{O}(\log^2 n)$ time, $\mathcal{O}(\log n)$ random N-bit words, and $\mathcal{O}(\log n)$ additional space.*

Proof. We focus on the number of iterations as the remaining results follow trivially. This proof is rather involved and thus is divided into several steps.

Specifying random choices. The performance of the algorithm depends on not only the random variables $X_{v,u}$'s, but also the unused coins $C_{v,u}$'s. We characterize the two col-

lections of Bernoulli variables $\{X_{v,u}\}$ and $\{Y_{v,u}\}$ that cover all random choices made by Algorithm 2 as follows.

- Each $X_{v,u}$ (same as $X_{u,v}$) represents the result for the *first* coin-toss corresponding to cells $\mathbf{A}[v][u]$ and $\mathbf{A}[u][v]$, which is the coin-toss obtained when $X_{v,u}$ becomes decided: either $C_{v,u}$ during a NEXT-NEIGHBOR(v) call when $\mathbf{A}[v][u] = \phi$, or $C_{v,u}$ during a NEXT-NEIGHBOR(u) call when $\mathbf{A}[u][v] = \phi$, whichever occurs first. This description of $X_{v,u}$ respects our invariant that, if the generation process is executed to completion, we will have $\mathbf{A}[v][u] = X_{v,u}$ in all entries.
- Each $Y_{v,u}$ represents the result for the *second* coin-toss corresponding to cell $\mathbf{A}[v][u]$, which is the coin-toss $C_{v,u}$ obtained during a NEXT-NEIGHBOR(v) call when $X_{v,u}$ is already decided. In other words, $\{Y_{v,u}\}$'s are the coin-tosses that should have been skipped but still performed in Algorithm 2 (if they have indeed been generated). Unlike the previous case, $Y_{v,u}$ and $Y_{u,v}$ are two independent random variables: they may be generated during a NEXT-NEIGHBOR(v) call and a NEXT-NEIGHBOR(u) call, respectively.

As mentioned earlier, we allow any sequence of probabilities $p_{v,u}$ in our proof. The success probabilities of these indicators are therefore given by $\mathbb{P}[X_{v,u} = 1] = \mathbb{P}[Y_{v,u} = 1] = p_{v,u}$.

Characterizing iterations. Suppose that we compute NEXT-NEIGHBOR(v) and obtain an answer u. Then $X_{v,\mathbf{last}[v]+1} = \cdots = X_{v,u-1} = 0$ as none of $u' \in (\mathbf{last}[v], u)$ is a neighbor of v. The vertices considered in the loop of Algorithm 2 that do not result in the answer u, are $u' \in (\mathbf{last}[v], u)$ satisfying $\mathbf{A}[v][u'] = 0$ and $Y_{v,u'} = 1$; we call the iteration corresponding to such a u' a *failed iteration*. Observe that if $X_{v,u'} = 0$ but is undecided ($\mathbf{A}[v][u'] = \phi$), then the iteration is not failed, even if $Y_{v,u'} = 1$ (in which case, $X_{v,u'}$ takes the value of $C_{v,u'}$ while $Y_{v,u'}$ is never used). Thus we assume the worst-case scenario where all $X_{v,u'}$ are revealed: $\mathbf{A}[v][u'] = X_{v,u'} = 0$ for all $u' \in (\mathbf{last}[v], u)$. The number of failed iterations in this case stochastically dominates those in all other cases.[1]

[1]There exists an adversary who can enforce this worst case. Namely, an adversary that first makes NEXT-NEIGHBOR queries to learn all neighbors of every vertex except for v, thereby filling out the whole $\mathbf{A}$ in the process. The claimed worst case then occurs as this adversary now repeatedly makes NEXT-NEIGHBOR queries on v. In particular, a committee of n adversaries, each of which is tasked to perform this series of calls corresponding to each v, can always expose this worst case.

Then, the upper bound on the number of failed iterations of a call NEXT-NEIGHBOR(v) is given by the maximum number of cells $Y_{v,u'} = 1$ of $u' \in (\mathbf{last}[v], u)$, over any $u \in (\mathbf{last}[v], n]$ satisfying $X_{v,\mathbf{last}[v]+1} = \cdots = X_{v,u} = 0$. Informally, we are asking "of all consecutive cells of 0's in a single row of $\{X_{v,u}\}$-table, what is the largest number of cells of 1's in the corresponding cells of $\{Y_{v,u}\}$-table?"

Bounding the number of iterations required for a fixed pair $(v, \mathbf{last}[v])$**.** We now proceed to bounding the number of iterations required over a sampled pair of $\{X_{v,u}\}$ and $\{Y_{v,u}\}$, from any probability distribution. For simplicity we renumber our indices and drop the index $(v, \mathbf{last}[v])$ as follows. Let $p_1, \ldots, p_L \in [0, 1]$ denote the probabilities corresponding to the cells $\mathbf{A}[v][\mathbf{last}[v] + 1 \ldots n]$ (where $L = n - \mathbf{last}[v]$), then let $X_1, \ldots, X_L$ and $Y_1, \ldots, Y_L$ be the random variables corresponding to the same cells on $\mathbf{A}$.

For $i = 1, \ldots, L$, define the random variable Z_i in terms of X_i and Y_i so that

- $Z_i = 2$ if $X_i = 0$ and $Y_i = 1$, which occurs with probability $p_i(1 - p_i)$.
  This represents the event where i is not a neighbor, and the iteration fails.
- $Z_i = 1$ if $X_i = Y_i = 0$, which occurs with probability $(1 - p_i)^2$.
  This represents the event where i is not a neighbor, and the iteration does not fail.
- $Z_i = 0$ if $X_i = 1$, which occurs with probability p_i.
  This represents the event where i is a neighbor.

For $\ell \in [L]$, define the random variable $M_\ell := \prod_{i=1}^{\ell} Z_i$, and $M_0 = 1$ for convenience. If $X_i = 1$ for some $i \in [1, \ell]$, then $Z_i = 0$ and $M_\ell = 0$. Otherwise, $\log M_\ell$ counts the number of indices $i \in [\ell]$ with $Y_i = 1$, the number of failed iterations. Therefore, $\log(\max_{\ell \in \{0,\ldots,L\}} M_\ell)$ gives the number of failed iterations this NEXT-NEIGHBOR(v) call.

To bound M_ℓ, observe that for any $\ell \in [L]$, $\mathbb{E}[Z_\ell] = 2p_\ell(1-p_\ell) + (1-p_\ell)^2 = 1 - p_\ell^2 \leq 1$ regardless of the probability $p_\ell \in [0, 1]$. Then, $\mathbb{E}[M_\ell] = \mathbb{E}[\prod_{i=1}^{\ell} Z_i] = \prod_{i=1}^{\ell} \mathbb{E}[Z_i] \leq 1$ because Z_ℓ's are all independent. By Markov's inequality, for any (integer) $r \geq 0$, $\Pr[\log M_\ell > r] = \Pr[M_\ell > 2^r] < 2^{-r}$. By the union bound, the probability that more than r failed iterations are encountered is $\Pr[\log(\max_{\ell \in \{0,\ldots,L\}} M_\ell) > r] < L \cdot 2^{-r} \leq n \cdot 2^{-r}$.

Establishing the overall performance guarantee. So far we have deduced that, for each pair of a vertex v and its $\mathbf{last}[v]$, the probability that the call NEXT-NEIGHBOR(v) encoun-

ters more than r failed iterations is less that $n \cdot 2^{-r}$, which is at most n^{-c-2} for any desired constant c by choosing a sufficiently large $r = \Theta(\log n)$. As Algorithm 2 may need to support up to $\Theta(n^2)$ NEXT-NEIGHBOR calls, one corresponding to each pair $(v, \mathbf{last}[v])$, the probability that it ever encounters more than $O(\log n)$ failed iterations to answer a single NEXT-NEIGHBOR query is at most n^{-c}. That is, with high probability, $O(\log n)$ iterations are required per NEXT-NEIGHBOR call, which concludes the proof of Theorem 5. □

A.2 Supporting VERTEX-PAIR Queries

We extend our generator (Algorithm 2) to support the VERTEX-PAIR queries: given a pair of vertices (u, v), decide whether there exists an edge $\{u, v\}$ in the generated graph. To answer a VERTEX-PAIR query, we must first check whether the value $X_{u,v}$ for $\{u, v\}$ has already been assigned, in which case we answer accordingly. Otherwise, we must make a coin-flip with the corresponding bias $p_{u,v}$ to assign $X_{u,v}$, deciding whether $\{u, v\}$ exists in the generated graph. If we maintained the full $\mathbf{A}$ as done in the naïve Algorithm 1, we would have been able to simply set $\mathbf{A}[u][v]$ and $\mathbf{A}[v][u]$ to this new value. However, our more efficient Algorithm 2 that represents $\mathbf{A}$ compactly via $\mathbf{last}$ and P_v's cannot record arbitrary modifications to $\mathbf{A}$.

Observe that if we were to apply the trivial implementation of VERTEX-PAIR in Algorithm 1, then by Lemma 1, $\mathbf{last}$ and P_v's will only fail capture the state $\mathbf{A}[v][u] = 0$ when $u > \mathbf{last}[v]$ and $v > \mathbf{last}[u]$. Fortunately, unlike NEXT-NEIGHBOR queries, a VERTEX-PAIR query can only set one cell $\mathbf{A}[v][u]$ to 0 per query, and thus we may afford to store these changes explicitly.[2] To this end, we define the set $Q = \{\{u, v\} : X_{u,v}$ is assigned to 0 during a VERTEX-PAIR query$\}$, maintained as a hash table. Updating Q during VERTEX-PAIR queries is trivial: we simply add $\{u, v\}$ to Q before we finish processing the query if we set $\mathbf{A}[u][v] = 0$. Conversely, we need to add u to P_v and add v to P_u if the VERTEX-PAIR query sets $\mathbf{A}[u][v] = 1$ as usual, yielding the following observation. It is straightforward to verify that each VERTEX-PAIR query requires $O(\log n)$ time, $O(1)$

[2]The disadvantage of this approach is that the generator may allocate more than $\Theta(m)$ space over the entire graph generation process, if VERTEX-PAIR queries generate many of these 0's.

random N-bit word, and $O(1)$ additional space per query.

Lemma 13. *The data structures* last, P_v*'s and* Q *together provide a succinct representation of* $\mathbf{A}$ *when* NEXT-NEIGHBOR *queries (modified Algorithm 2) and* VERTEX-PAIR *queries (modified Algorithm 1) are allowed. In particular,* $\mathbf{A}[v][u] = 1$ *if and only if* $u \in P_v$*. Otherwise,* $\mathbf{A}[v][u] = 0$ *if* $u < \mathbf{last}[v]$*,* $v < \mathbf{last}[u]$*, or* $\{v, u\} \in Q$*. In all remaining cases,* $\mathbf{A}[v][u] = \phi$*.*

We now explain other necessary changes to Algorithm 2. In the implementation of NEXT-NEIGHBOR, an iteration is not failed when the chosen $X_{v,u}$ is still undecided: $\mathbf{A}[v][u]$ must still be ϕ. Since $X_{v,u}$ may also be assigned to 0 via a VERTEX-PAIR(v, u) query, we must also consider an iteration where $\{v, u\} \in Q$ failed. That is, we now require one additional condition $\{v, u\} \notin Q$ for termination (which only takes $O(1)$ time to verify per iteration). As for the analysis, aside from handling the fact that $X_{v,u}$ may also become decided during a VERTEX-PAIR call, and allowing the states of the algorithm to support VERTEX-PAIR queries, all of the remaining analysis for correctness and performance guarantee still holds.

Therefore, we have established that our augmentation to Algorithm 2 still maintains all of its (asymptotic) performance guarantees for NEXT-NEIGHBOR queries, and supports VERTEX-PAIR queries with complexities as specified above, concluding the following corollary. We remark that, as we do not aim to support RANDOM-NEIGHBOR queries, this simple algorithm here provides significant improvement over the performance of RANDOM-NEIGHBOR queries (given in Corollary 1).

Corollary 7. *Algorithm 2 can be modified to allow an implementation of* VERTEX-PAIR *query as explained above, such that the resource usages per query still asymptotically follow those of Theorem 5.*

Appendix B

Alternative Generator with Deterministic Performance Guarantee

In this section, we construct data structures that allow us to sample for the next neighbor directly by considering only the cells $\mathbf{A}[v][u] = \phi$ in the Erdös-Rényi model and the Stochastic Block model. This provides $\text{poly}(\log n)$ *worst-case* performance guarantee for generators supporting only the NEXT-NEIGHBOR queries. We may again extend this data structure to support VERTEX-PAIR queries, however, at the cost of providing $\text{poly}(\log n)$ *amortized* performance guarantee instead.

In what follows, we first focus on the $G(n, p)$ model, starting with NEXT-NEIGHBOR queries (Section B.1) then extend to VERTEX-PAIR queries (Section B.2. We then explain how this result may be generalized to support the Stochastic Block model with random community assignment in Section B.3.

B.1 Data structure for next-neighbor queries in the Erdös-Rényi model

Recall that NEXT-NEIGHBOR(v) is given by $\min\{u > \mathbf{last}[v] : X_{v,u} = 1\}$ (or $n+1$ if no satisfying u exists). To aid in computing this quantity, we define:

$$K_v = \{u \in (\mathbf{last}[v], n] : \mathbf{A}[v][u] = 1\},$$
$$w_v = \min K_v, \text{ or } n+1 \text{ if } K_v = \emptyset,$$
$$T_v = \{u \in (\mathbf{last}[v], w_v) : \mathbf{A}[v][u] = \phi\}.$$

The ordered set K_v is only defined for ease of presentation within this section: it is equivalent to $(\mathbf{last}[v], n] \cap P_v$, recording the known neighbors of v after $\mathbf{last}[v]$ (i.e., those that have not been returned as an answer by any NEXT-NEIGHBOR(v) query yet). The quantity w_v remains unchanged but is simply restated in terms of K_v. T_v specifies the list of candidates u for NEXT-NEIGHBOR(v) with $\mathbf{A}[v][u] = \phi$; in particular, all candidates u's, such that the corresponding RVs $X_{v,u} = 0$ are decided, are explicitly excluded from T_v.

Unlike the approach of Algorithm 2 that simulates coin-flips even for decided $X_{v,u}$'s, here we only flip undecided coins for the indices in T_v: we have $|T_v|$ Bernoulli trials to simulate. Let F be the random variable denoting the first index of a successful trial out of $|T_v|$ coin-flips, or $|T_v|+1$ if all fail; denote the distribution of F by ExactF$(p, |T|)$. The CDF of F is given by $\mathbb{P}[F = f] = 1-(1-p)^f$ for $f \leq |T_v|$ (i.e., there is some success trial in the first f trials), and $\mathbb{P}[F = |T_v|+1] = 1$. Thus, we must design a data structure that can compute w_v, compute $|T_v|$, find the F^{th} minimum value in T_v, and update $\mathbf{A}[v][u]$ for the F lowest values $u \in T_v$ accordingly.

Algorithm 5 Alternative Generator

```
procedure NEXT-NEIGHBOR(v)
    w ← min K_v, or n+1 if K_v = ∅
    t ← COUNT(v)
    sample F ~ ExactF(p, t)
    if F ≤ t then
        u ← PICK(v, F)
        K_u ← K_u ∪ {v}
    else
        u ← w
        if u ≠ n+1 then
            K_v ← K_v \ {u}
    UPDATE(v, u)
    last[v] ← u
    return u
```

Let $k = \lceil \log n \rceil$. We create a range tree, where each node itself contains a balanced binary search tree (BBST), storing **last** values of its corresponding range. Formally, for $i \in [0, n/2^j)$ and $j \in [0, k]$, the i^{th} node of the j^{th} level of the range tree, stores $\mathbf{last}[v]$ for every $v \in (i \cdot 2^{k-j}, (i+1) \cdot 2^{k-j}]$. Denote the range tree by $\mathbf{R}$, and each BBST corresponding

to the range $[a, b]$ by $\mathbf{B}_{[a,b]}$. We say that the range $[a, b]$ is *canonical* if it corresponds to a range of some $\mathbf{B}_{[a,b]}$ in $\mathbf{R}$.

Again, to allow fast initialization, we make the following adjustments from the given formalization above: (1) values $\mathbf{last}[v] = 0$ are never stored in any $\mathbf{B}_{[a,b]}$, and (2) each $\mathbf{B}_{[a,b]}$ is created on-the-fly during the first occasion it becomes non-empty. Further, we augment each $\mathbf{B}_{[a,b]}$ so that each of its node maintains the size of the subtree rooted at that node: this allows us to count, in $O(\log n)$ time, the number of entries in $\mathbf{B}_{[a,b]}$ that is no smaller than a given threshold.

Observe that each v is included in exactly one $\mathbf{B}_{[a,b]}$ per level in $\mathbf{R}$, so $k+1 = O(\log n)$ copies of $\mathbf{last}[v]$ are stored throughout $\mathbf{R}$. Moreover, by the property of range trees, any interval can be decomposed into a disjoint union of $O(\log n)$ canonical ranges. From these properties we implement the data structure $\mathbf{R}$ to support the following operations. (Note that $\mathbf{R}$ is initially an empty tree, so initialization is trivial.)

- COUNT(v): compute $|T_v|$.

  We break $(\mathbf{last}[v], w_v)$ into $O(\log n)$ disjoint canonical ranges $[a_i, b_i]$'s each corresponding to some $\mathbf{B}_{[a_i,b_i]}$, then compute $t_{[a_i,b_i]} = |\{u \in [a_i, b_i] : \mathbf{last}[u] < v\}|$, and return $\sum_i t_{[a_i,b_i]}$. The value $t_{[a_i,b_i]}$ is obtained by counting the entries of $\mathbf{B}_{[a_i,b_i]}$ that is at least v, then subtract it from $b_i - a_i + 1$; we cannot count entries less than v because $\mathbf{last}[u] = 0$ are not stored.

- PICK(v, F): find the F^{th} minimum value in T_v (assuming $F \leq |T_v|$).

  We again break $(\mathbf{last}[v], w_v)$ into $O(\log n)$ canonical ranges $[a_i, b_i]$'s, compute $t_{[a_i,b_i]}$'s, and identify the canonical range $[a^*, b^*]$ containing the i^{th} smallest element (i.e., $[a_i, b_i]$ with the smallest b satisfying $\sum_{j \leq i} t_{[a_j,b_j]} \geq F$ assuming ranges are sorted). Binary-search in $[a^*, b^*]$ to find exactly the i^{th} smallest element of T. This is ccomplished by traversing $\mathbf{R}$ starting from the range $[a^*, b^*]$ down to a leaf, at each step computing the children's $T_{[a,b]}$'s and deciding which child's range contains the desired element.

- UPDATE(v, u): simulate coin-flips, assigning $X_{v,u} \leftarrow 1$, and $X_{v,u'} \leftarrow 0$ for $u' \in (\mathbf{last}[v], u) \cap T_v$.

  This is done implicitly by handling the change $\mathbf{last}[v] \leftarrow u$: for each BBST $\mathbf{B}_{[a,b]}$

where $v \in [a, b]$, remove the old value of $\mathbf{last}[v]$ and insert u instead.

It is straightforward to verify that all operations require at most $O(\log^2 n)$ time and $O(\log n)$ additional space per call. The overall implementation is given in Algorithm 5, using the same asymptotic time and additional space. Recall also that sampling $F \sim \mathsf{ExactF}(p, t)$ requires $O(\log n)$ time and one N-bit random word for the $G(n, p)$ model.

B.2 Data structure for VERTEX-PAIR queries in the Erdös-Rényi model

Recall that we define Q in Algorithm 2 as the set of pairs (u, v) where $X_{u,v}$ is assigned to 0 during a VERTEX-PAIR query, allowing us to check for modifications of $\mathbf{A}$ not captured by $\mathbf{last}[v]$ and K_v. Here in Algorithm 5, rather than checking, we need to be able to count such entries. Thus, we instead create a BBST Q'_v for each v defined as:

$$Q'_v = \{u : u > \mathbf{last}[v], v > \mathbf{last}[u], \text{ and } X_{u,v} \text{ is assigned to 0 during a VERTEX-PAIR query}\}.$$

This definition differs from that of Q in Section A.2 in two aspects. First, we ensure that each $\mathbf{A}[v][u] = 0$ is recorded by either $\mathbf{last}$ (via Lemma 1) or Q'_v (explicitly), but *not both*. In particular, if u were to stay in Q'_v when $\mathbf{last}[v]$ increases beyond u, we would have double-counted these entries 0 not only recorded by Q'_v but also implied by $\mathbf{last}[v]$ and K_v. By having a BBST for each Q'_v, we can compute the number of 0's that must be excluded from T_v, which cannot be determined via $\mathbf{last}[v]$ and K_v alone: we subtract these from any counting process done in the data structure $\mathbf{R}$.

Second, we maintain Q'_v separately for each v as an ordered set, so that we may identify non-neighbors of v within a specific range – this allows us to remove non-neighbors in specific range, ensuring that the first aspect holds. More specifically, when we increase $\mathbf{last}[v]$, we must go through the data structure Q'_v and remove all $u < \mathbf{last}[v]$, and for each such u, also remove v from Q'_u. There can be as many as linear number of such u, but the number of removals is trivially bounded by the number of insertions, yielding an amortized time performance guarantee in the following theorem. Aside from the deterministic

guarantee, unsurprisingly, the required amount of random words for this algorithm is lower than that of the algorithm from Section A (given in Theorem 5 and Corollary 7).

Theorem 6. *Consider the Erdös-Rényi $G(n,p)$ model. For* NEXT-NEIGHBOR *queries only, Algorithm 5 is a generator that answers each query using $O(\log^2 n)$ time, $O(\log n)$ additional space, and one N-bit random word. For* NEXT-NEIGHBOR *and* VERTEX PAIR *queries,an extension of Algorithm 5 answers each query using $O(\log^2 n)$ amortized time, $O(\log n)$ additional space, and one N-bit random word.*

B.3 Data structure for the Stochastic Block model

We employ the data structure for generating and counting the number of vertices of each community in a specified range from Section 4.2. We create r different copies of the data structure $\mathbf{R}$ and Q'_v, one for each community, so that we may implement the required operations separately for each color, including using the COUNT subroutine to sample $F \sim$ ExactF via the corresponding CDF, and picking the next neighbor according to F. Recall that since we do not store $\mathbf{last}[v] = 0$ in $\mathbf{R}$, and we only add an entry to K_v, P_v or Q'_v after drawing the corresponding $X_{u,v}$, the communities of the endpoints, which cover all elements stored in these data structures, must have already been determined. Thus, we obtain the following corollary for the Stochastic Block model.

Corollary 8. *Consider the Stochastic Block model with randomly-assigned communities. For* NEXT-NEIGHBOR *queries only, Algorithm 5 is a generator that answers each query using $O(r\,\mathrm{poly}(\log n))$ time, random words, and additional space per query. For* NEXT-NEIGHBOR *and* VERTEX-PAIR *queries, Algorithm 5 answers each query using $O(r\,\mathrm{poly}(\log n))$ amortized time, $O(r\,\mathrm{poly}(\log n))$ random words, and $O(r\,\mathrm{poly}(\log n))$ additional space per query additional space, and one N-bit random word.*

Appendix C

Additional related work

Random graph models. The Erdös-Rényi model, given in [14], is one of the most simple theoretical random graph model, yet more specialized models are required to capture properties of real-world data. The Stochastic Block model (or the planted partition model) was proposed in [22] originally for modeling social networks; nonetheless, it has proven to be an useful general statistical model in numerous fields, including recommender systems [26, 37], medicine [39], social networks [17, 35], molecular biology [9, 28], genetics [8, 23, 11], and image segmentation [38]. Canonical problems for this model are the community detection and community recovery problems: some recent works include [10, 32, 3, 2]; see e.g., [1] for survey of recent results. The study of Small-World networks is originated in [41] has frequently been observed, and proven to be important for the modeling of many real world graphs such as social networks [12, 40], brain neurons [5], among many others. Kleinberg's model on the simple lattice topology (as considered in this paper) imposes a geographical that allows navigations, yielding important results such as routing algorithms (decentralized search) [24, 29]. See also e.g., [34] and Chapter 20 of [13].

Generation of random graphs. The problem of local-access implementation of random graphs has been considered in the aforementioned work [18, 33, 15], as well as in [27] that locally generates out-going edges on bipartite graphs while minimizing the maximum in-degree. The problem of generating full graph instances for random graph models have been frequently considered in many models of computations, such as sequential algorithms

[31, 6, 36, 30], and the parallel computation model [4].

Query models. In the study of sub-linear time graph algorithms where reading the entire input is infeasible, it is necessary to specify how the algorithm may access the input graph, normally by defining the type of queries that the algorithm may ask about the input graph; the allowed types of queries can greatly affect the performance of the algorithms. While NEXT-NEIGHBOR query is only recently considered in [15], there are other query models providing a neighbor of a vertex, such as asking for an entry in the adjacency-list representation [21], or traversing to a random neighbor [7]. On the other hand, the VERTEX-PAIR query is common in the study of dense graphs as accessing the adjacency matrix representation [20]. The ALL-NEIGHBORS query has recently been explicitly considered in local algorithms [16].

Bibliography

[1] Emmanuel Abbe. Community detection and the stochastic block model. 2016.

[2] Emmanuel Abbe, Afonso S Bandeira, and Georgina Hall. Exact recovery in the stochastic block model. *IEEE Transactions on Information Theory*, 62(1):471–487, 2016.

[3] Emmanuel Abbe and Colin Sandon. Community detection in general stochastic block models: Fundamental limits and efficient algorithms for recovery. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 670–688. IEEE, 2015.

[4] Maksudul Alam and Maleq Khan. Parallel algorithms for generating random networks with given degree sequences. *International Journal of Parallel Programming*, 45(1):109–127, 2017.

[5] Danielle Smith Bassett and ED Bullmore. Small-world brain networks. *The neuroscientist*, 12(6):512–523, 2006.

[6] Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

[7] Michael Brautbar and Michael J Kearns. Local algorithms for finding interesting individuals in large networks. 2010.

[8] Irineo Cabreros, Emmanuel Abbe, and Aristotelis Tsirigos. Detecting community structures in hi-c genomic data. In *Information Science and Systems (CISS), 2016 Annual Conference on*, pages 584–589. IEEE, 2016.

[9] Jingchun Chen and Bo Yuan. Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.

[10] Peter Chin, Anup Rao, and Van Vu. Stochastic block model and community detection in sparse graphs: A spectral algorithm with optimal rate of recovery. In *Conference on Learning Theory*, pages 391–423, 2015.

[11] Melissa S Cline, Michael Smoot, Ethan Cerami, Allan Kuchinsky, Nerius Landys, Chris Workman, Rowan Christmas, Iliana Avila-Campilo, Michael Creech, Benjamin Gross, et al. Integration of biological networks and gene expression data using cytoscape. *Nature protocols*, 2(10):2366–2382, 2007.

[12] Peter Sheridan Dodds, Roby Muhamad, and Duncan J Watts. An experimental study of search in global social networks. *science*, 301(5634):827–829, 2003.

[13] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[14] Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[15] Guy Even, Reut Levi, Moti Medina, and Adi Rosén. Sublinear random access generators for preferential attachment graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 6:1–6:15, 2017.

[16] Uriel Feige, Boaz Patt-Shamir, and Shai Vardi. On the probe complexity of local computation algorithms. *arXiv preprint arXiv:1703.07734*, 2017.

[17] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

[18] O Goldreich, S Goldwasser, and A Nussboim. On the implementation of huge random objects. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 68–79. IEEE, 2003.

[19] Oded Goldreich, Shafi Goldwasser, and Asaf Nussboim. On the implementation of huge random objects. *SIAM Journal on Computing*, 39(7):2761–2822, 2010.

[20] Oded Goldreich, Shari Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *Journal of the ACM (JACM)*, 45(4):653–750, 1998.

[21] Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. In *Proceedings of the twenty-ninth annual ACM Symposium on Theory of Computing*, pages 406–415. ACM, 1997.

[22] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.

[23] Daxin Jiang, Chun Tang, and Aidong Zhang. Cluster analysis for gene expression data: a survey. *IEEE Transactions on knowledge and data engineering*, 16(11):1370–1386, 2004.

[24] Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the thirty-second annual ACM Symposium on Theory of Computing*, pages 163–170. ACM, 2000.

[25] Donald E Knuth. The art of computer programming, 3rd edn. seminumerical algorithms, vol. 2, 1997.

[26] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

[27] Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. Converting online algorithms to local computation algorithms. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, pages 653–664, 2012.

[28] Edward M Marcotte, Matteo Pellegrini, Ho-Leung Ng, Danny W Rice, Todd O Yeates, and David Eisenberg. Detecting protein function and protein-protein interactions from genome sequences. *Science*, 285(5428):751–753, 1999.

[29] Chip Martel and Van Nguyen. Analyzing kleinberg's (and other) small-world models. In *Proceedings of the twenty-third annual ACM Symposium on Principles of Distributed Computing*, pages 179–188. ACM, 2004.

[30] Joel Miller and Aric Hagberg. Efficient generation of networks with given expected degrees. *Algorithms and models for the web graph*, pages 115–126, 2011.

[31] Ron Milo, Nadav Kashtan, Shalev Itzkovitz, Mark EJ Newman, and Uri Alon. On the uniform generation of random graphs with prescribed degree sequences. *arXiv preprint cond-mat/0312028*, 2003.

[32] Elchanan Mossel, Joe Neeman, and Allan Sly. Reconstruction and estimation in the planted partition model. *Probability Theory and Related Fields*, 162(3-4):431–461, 2015.

[33] Moni Naor and Asaf Nussboim. Implementing huge sparse random graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 596–608. Springer, 2007.

[34] Mark EJ Newman. Models of the small world. *Journal of Statistical Physics*, 101(3):819–841, 2000.

[35] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002.

[36] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 331–342. ACM, 2011.

[37] Shaghayegh Sahebi and William W Cohen. Community-based recommendations: a solution to the cold start problem. In *Workshop on recommender systems and the social web, RSWEB*, page 60, 2011.

[38] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

[39] Therese Sørlie, Charles M Perou, Robert Tibshirani, Turid Aas, Stephanie Geisler, Hilde Johnsen, Trevor Hastie, Michael B Eisen, Matt Van De Rijn, Stefanie S Jeffrey, et al. Gene expression patterns of breast carcinomas distinguish tumor subclasses with clinical implications. *Proceedings of the National Academy of Sciences*, 98(19):10869–10874, 2001.

[40] Jeffrey Travers and Stanley Milgram. The small world problem. *Phychology Today*, 1:61–67, 1967.

[41] Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.