Local Access to Huge Random Objects through Partial Sampling

Amartya Shankha Biswas
CSAIL, MIT
asbiswas@mit.edu

Ronitt Rubinfeld
CSAIL, MIT
ronitt@csail.mit.edu

Anak Yodpinyanee
CSAIL, MIT
anak@csail.mit.edu

Abstract

Consider an algorithm performing a computation on a *huge random object* (for example a random graph or a "long" random walk). Is it necessary to generate the entire object prior to the computation, or is it possible to provide query access to the object and sample it incrementally "on-the-fly" (as requested by the algorithm)? Such an *implementation* should emulate the random object by answering queries in a manner consistent with an instance of the random object sampled from the true distribution (or close to it). This paradigm is useful when the algorithm is sub-linear and thus, sampling the entire object up front would ruin its efficiency.

Our first set of results focus on undirected graphs with independent edge probabilities, i.e. each edge is chosen as an independent Bernoulli random variable. We provide a general implementation for this model under certain assumptions. Then, we use this to obtain the first efficient local implementations for the Erdös-Rényi $G(n, p)$ model for *all* values of p, and the Stochastic Block model. As in previous local-access implementations for random graphs, we support `Vertex-Pair` and `Next-Neighbor` queries. In addition, we introduce a new `Random-Neighbor` query. Next, we give the first local-access implementation for `All-Neighbors` queries in the (sparse and directed) Kleinberg's Small-World model. Our implementations require no pre-processing time, and answer each query using $\mathcal{O}(\text{poly}(\log n))$ time, random bits, and additional space.

Next, we show how to implement random Catalan objects, specifically focusing on Dyck paths (balanced random walks on the integer line that are always positive). Here, we support `Height` queries to find the location of the walk, and `First-Return` queries to find the time when the walk returns to a specified location. This in turn can be used to implement `Next-Neighbor` queries on random rooted and binary trees, and `Matching-Bracket` queries on random well bracketed expressions (the Dyck language).

Finally, we study random q-colorings of graphs with max degree Δ. In contrast to the prior settings, where random objects are generated according to a single distribution with $\mathcal{O}(1)$ parameters (for example, n and p in the $G(n, p)$ model), the distribution here is specified via a "huge" input (in this case, the underlying graph). When $q > \alpha\Delta$ for a small constant α, we show how to answer queries to the color of any given node in sub-linear time.

Keywords sublinear time algorithms, random generation, local computation

Funding *Amartya Shankha Biswas*: MIT Presidential Fellowship
Ronitt Rubinfeld: NSF grants CCF-1650733, CCF-1733808, IIS-1741137 and CCF-1740751
Anak Yodpinyanee: NSF grants CCF-1650733, CCF-1733808, IIS-1741137 and DPST scholarship, Royal Thai Government

Contents

1 Introduction

Consider an algorithm performing a computation on a *huge random object* (for example a random graph or a "long" random walk). Is it necessary to generate the entire object prior to the computation, or is it possible to provide query access to the object and sample it incrementally "on-the-fly" (as requested by the algorithm)? Such an *implementation* would emulate the random object by answering appropriate queries in a consistent manner. Specifically, all responses to queries must be consistent with an instance of the random object sampled from the true distribution (or close to it). This paradigm is useful when the algorithm is sub-linear and thus, sampling the entire object up front would ruin its efficiency. For example, the greedy routing algorithm on Kleinberg's small world networks [Kle00] only uses $\mathcal{O}(\log^2 n)$ probes. Using our implementation, one can execute this algorithm on a random small world instance in $\mathcal{O}(\text{poly}(\log n))$ time without incurring the $\mathcal{O}(n)$ prior-sampling overhead.

The problem of sampling partial information about huge random objects was pioneered in [GGN10, GGN03, GGM86]. Further work in [NN07, ELMR17] considers the generation of different random graph models. In this work, we focus on generating huge random objects in a number of new settings, including basic random graph models that were not previously considered, Catalan objects, and random colorings. One emerging theme that we develop further in this work is to provide common data-structure query access to huge random objects. For example, for random graph implementations, in addition to supporting adjacency matrix queries about the existence of edge (u, v), [ELMR17] introduced the study of NEXT-NEIGHBOR queries which provide efficient access to the adjacency list representation.

1.1 Random Graphs

In Section 3, we implement queries to both the adjacency matrix and adjacency list representation for the generic class of *undirected graphs* with *independent edge probabilities* $\{p_{uv}\}_{u,v\in V}$, where p_{uv} denotes the probability that there is an edge between u and v. We implement VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR [1] queries. Under reasonable assumptions on the ability to compute certain values pertaining to consecutive edge probabilities, our implementations support all three types of queries using $\mathcal{O}(\text{poly}(\log n))$ time, space, and random bits. Note that in this setting, VERTEX-PAIR queries are trivial, since the existence of an edge depends on an independent random variable. As in [ELMR17] (and unlike many of the implementations presented in [GGN03, GGN10]), our techniques allow unlimited queries. In particular, our construction yields local-access implementations for the Erdös-Rényi $G(n, p)$ model (for *all* values of p), and the Stochastic Block model with random community assignment.

While VERTEX-PAIR and NEXT-NEIGHBOR queries, as well as ALL-NEIGHBORS queries for sparse graphs, have been considered in the prior works of [ELMR17, GGN03, GGN10, NN07], we provide the first implementations of non-sparse random graph models. Prior results for implementing queries to $G(n, p)$ focused on the sparse case where $p = \log^{\mathcal{O}(1)} n/n$ [NN07]. The dense case where $p = \Theta(1)$ is also relatively simple because most of the adjacency matrix is filled, and neighbor queries can be answered by performing $\Theta(1)$ VERTEX-PAIR queries until an edge is found. The case of general p was not considered previously. For example, when $p = 1/\sqrt{n}$, each vertex has high degree $\mathcal{O}(\sqrt{n})$ but most of the adjacency matrix is empty, thus making it difficult to sample a neighbor efficiently. We also provide the first implementation (to the best of our knowledge) of RANDOM-NEIGHBOR queries in *non-sparse graphs*. Such queries are useful, for instance, in sub-linear algorithms that employ random walk processes.

[1] VERTEX-PAIR(u, v) returns whether u and v are adjacent, NEXT-NEIGHBOR(v) returns a new neighbor 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	VERTEX-PAIR	NEXT-NEIGHBOR	RANDOM-NEIGHBOR	ALL-NEIGHBORS
$G(n,p)$ with $p = \frac{\log^{\mathcal{O}(1)} n}{n}$	[NN07]	[NN07]	[NN07]	[NN07]
$G(n,p)$ for arbitrary p	This paper	This paper	This paper	✗
Stochastic Block Model with $\log^{\mathcal{O}(1)} n$ communities	This paper	This paper	This paper	✗
BA Preferential Attachment	[ELMR17]	[ELMR17]	??	✗
Small world model	This paper	This paper	This paper	This paper
Random ordered rooted trees	This paper	This paper	This paper	This paper

Table 1 (Local access implementations for random graphs): All the implementations in this table use polylogarithimc time, additional space and random bits per query. A ✗ in the ALL-NEIGHBORS column indicates that the graphs in this model may have un-bounded degree, and it is therefore impossible to sample ALL-NEIGHBORS efficiently. A ?? entry indicates a sampling problem with no known efficient solution.

Directed Random Graphs: The Small World Model

We then consider local-access implementations for directed graphs in Kleinberg's Small World model, where the probabilities are based on distances in a 2-dimensional grid. Building on our previous sampling procedure, we implement ALL-NEIGHBORS queries for the Small World model, using $\mathcal{O}(\text{poly}(\log n))$ time, space and random bits (since such graphs are sparse, the other queries follow directly).

1.2 Catalan Objects

We show how to provide query access to very long ($2n$ step) one dimensional random walks, One obvious query of interest is HEIGHT(t) which returns the position of the walk at time t. We also introduce and support FIRST-RETURN queries, where FIRST-RETURN(t) returns the first time when the random walk returns to the same level as it was at time t. HEIGHT queries for the simple unconstrained random walk follow trivially from the implementation of interval summable functions presented in [GGN10]. Here, we focus on an important generalization by considering balanced random walks (equal number of up and down steps) on the integer line, that are constrained to be always positive (commonly known as Dyck Paths). The added constraint introduces intricate non-local dependencies on the distribution of positions. However, we are able to support both queries using $\mathcal{O}(\text{poly}(\log n))$ resources.

Dyck paths are one type of Catalan object, and they have natural bijections to other Catalan objects such as bracketed expressions, random rooted trees and binary trees. Thus, we can use our Dyck Path implementation to obtain useful implementations of other random Catalan objects. For instance, HEIGHT queries correspond to DEPTH queries on rooted trees and bracketed expressions (Section 4.1). We also support more involved queries that are widely used; for example, finding the children of a node in a random tree or finding the matching bracket in a random bracketed expression. To achieve this, we note that FIRST-RETURN queries queries correspond to NEXT-NEIGHBOR queries on trees, and MATCHING-BRACKET queries on bracketed expressions (Section 4.1).

1.3 Random Coloring of Graphs

Finally, we introduce a new model for implementating huge random objects with *huge input description*; that is, the distribution is specified as a uniformly random solution to a huge combinatorial problem. In this model, we implement query access to random q-colorings of a given huge graph G with maximum degree Δ. A random coloring is sampled by proposing $\mathcal{O}(n \log n)$ color updates and accepting the ones that

do not create a conflict (Glauber dynamics). This is an inherently sequential process with the acceptance of a particular proposal depending on all preceding neighboring proposals. Moreover, unlike the previously considered random objects, this one has no succinct representation, and we can only uncover the proper distribution by probing the graph (in the manner of *local computation algorithms* [RTVX11, ARVX12]). Unlike LCAs which have to return *some* valid solution, we also have to make sure that we return a solution from the correct distribution. We are able to construct an efficient oracle that returns the final color of a vertex using only a sub-linear number of probes when $q \geq 7\Delta$.

Unlike all prior results in the area, this implementation has an additional feature that it is memoryless, and the sampled color values only depend on the input random bits. Consequently, multiple independent instances of the algorithm having access to the same random bits, will respond to queries in a manner consistent with each other; they will sample exactly the same coloring, regardless of the queries asked.

1.4 Related Work

The problem of computing local information of huge random objects was pioneered in [GGN03, GGN10]. Further work of [NN07] considers the generation of sparse random $G(n,p)$ graphs from the Erdös-Rényi model [ER60], 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 implementations 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 [ELMR17], the authors construct an oracle for the generation of recursive trees, and BA preferential attachment graphs. Unlike prior work, 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 [ELMR17] provides access to the adjacency list through `NEXT-NEIGHBOR` queries.

For additional related work, see Section G.

Computing large numbers (only poly). Maybe present a list of challenges including this.

2 Overview of our Techniques

We begin by formalizing our model of *local-access implementations*, inspired by [ELMR17].

▶ **Definition 1.** *Given a distribution X over a set of huge random objects $\mathbb{X}$, a* local access implementation *of a family of query functions $\langle F_1, F_2, \cdots \rangle$ where $F_i : \mathbb{X} \to \{0,1\}$, provides an oracle that returns the value $F_i(X)$ for $X \sim \mathbb{X}$ and a given query F_i, while satisfying the following:*

- ***Consistency:*** *All the values $F_i(X)$ returned by the local-access implementation throughout the entire execution must be consistent with a single $X \in \mathbb{X}$.*
- ***Distribution equivalence:*** *The random object $X \in \mathbb{X}$ consistent with the responses $\{F_i(X)\}$ must be sampled from some distribution X' that is ϵ-close to the desired distribution X in L_1-distance. In this work we focus on supporting $\epsilon = n^{-c}$ for any desired constant $c > 0$.*
- ***Performance:*** *The computation time, random bits, and additional space required to answer a single query must be sub-linear, and preferably $\text{poly}(\log n)$ with high probability, without any initialization overhead.*

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

2.1 Memory-less Sampling from Distributions with Huge Description Size

We also consider distributions whose description size is too large to be read by a sublinear algorithm. Specifically, we consider the uniform distribution over all valid colorings of a given input graph G. In this setting, we allow access to a uniformly random coloring of G, by implementing a query $\textsc{Color}(v)$, which returns the color of a *single vertex* v. Since $\textsc{Color}(v)$ has to run in time sublinear in the size of G, it is not even possible to read the entire input graph during its execution.

Talk about memory

Talk about LCA type model and public source of randomness.

2.2 Basic Tools for Efficient Sampling

In this section, we describe the main techniques used to sample from a distribution $\{p_d\}$, which differ based on the type of access to $\{p_d\}$ provided to the algorithm. If the algorithm is given cumulative distribution function (CDF) queries to $\{p_d\}$, then it is well known that via $\mathcal{O}(\log n)$ CDF evaluations, one can sample according to a distribution that is at most n^{-c} far from $\{p_d\}$ in L_1 distance (for constant c).

When only given access to queries to the probability dsitribution function (PDF) of $\{p_d\}$, sampling can be more challenging. The approach that we use in this work is to construct an auxiliary distribution $\{q_d\}$ with the following two properties: First, $\{q_d\}$ has an efficiently computable CDF. Second, q_d approximates p_d pointwise to within a polylogarithmic multiplicative factor for "most" of the support of $\{p_d\}$. the following Lemma from [GGN10] formalizes this concept, and shows that if we can provide such a $\{q_d\}$, we can quickly sample according to a distribution that is close to $\{p_d\}$.

▶ **Lemma 2.** *(From [GGN10]) Let $\{p_i\}$ and $\{q_i\}$ be distributions satisfying the following conditions:*

1. *There is a poly-time algorithm to approximate p_i and q_i up to $\pm n^{-2}$*
2. *Generating an index i according to q_i is closely implementable.*
3. *There exists a poly$(\log n)$-time recognizable set B such that*
   - *$1 - \sum_{i \in B} p_i$ is negligible*
   - *There exists a constant c such that for every i, it holds that $p_i \leq \log^{\mathcal{O}(1)} n \cdot q_i$*

Then, generating an index i according to the distribution $\{p_i\}$ is closely-implementable.

2.3 Undirected Graphs

In Section 3, we implement queries to both the adjacency matrix and adjacency list representation for the generic class of *undirected graphs* with *independent edge probabilities* $\{p_{uv}\}_{u,v \in V}$, where p_{uv} denotes 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 $V = [n]$. In this model, $\textsc{Vertex-Pair}$ queries by themselves can be implemented trivially, since the existence of any edge (u, v) is an independent Bernoulli random variable, but it becomes harder to maintain consistency when implementing them in conjunction with the other queries. Inspired by [ELMR17], we provide an implementation of $\textsc{next-neighbor}$ queries, which return the neighbors of any given vertex one by one in lexicographic order. Finally, we introduce a new query: $\textsc{Random-Neighbor}$ that returns a uniformly random neighbor of any given vertex. This would be useful for any algorithm that performs random walks. $\textsc{Random-Neighbor}$ queries present particularly interesting challenges that are outlined below.

`Next-Neighbor` Queries

The next neighbor of a vertex can be found trivially by generating consecutive entries of the adjacency matrix, but for small edge probabilities $p_{uv} = o(1)$ this implementation is inefficient, and uses $\Omega(1/p)$ time. As in [ELMR17], we speed this up by sampling the number of "non-neighbors" preceding the next neighbor. To do this, we assume that we can estimate the "skip" probabilities $F(v, a, b) = \prod_{u=a}^{b}(1 - p_{vu})$, 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 as compared to [ELMR17], arises from the fact that our graph is undirected, and thus we must "inform" 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 handle `Random-Neighbor` queries, where we may generate non-neighbors at random locations.

In Section 3.1, we present a very simple randomized implementation (Algorithm 1) 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 C, respectively.

`Random-Neighbor` Queries

We provide `Random-Neighbor` queries (Section 3.2) using $\text{poly}(\log n)$ resources. The ability to do so is surprising since: (1) Sampling the degree of the query vertex, we suspect, is not viable for *sub-linear* implementations, because this quantity alone imposes dependence on the existence of *all* of its potential incident edges and on the rest of the graph (Why?).

Fix This!

Therefore, our implementation needs to return a random neighbor, with probability reciprocal to the query vertex's degree, without resorting to "knowing" its degree. (2) 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. [2]

We formulate a *bucketing approach* (Section 3.2) which samples multiple consecutive 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 potential neighbors of v into consecutive ranges $\{B_v^{(i)}\}$ (buckets), so that each $B_v^{(i)}$ contains, in expectation, $\Theta(1)$ neighbors (i.e. $\sum_{u \in B_i} p_{vu} = \Theta(1)$). The subroutine of `Next-Neighbor` is applied to sample the neighbors within a bucket in expected $\widetilde{\mathcal{O}}(1)$ time. We can now obtain a neighbor of v by picking a random neighbor from a random bucket, but this introduces a bias because all buckets may not have the same number of neighbors. We remove this bias by rejecting samples from bucket $B_v^{(i)}$ with probability proportional to the number of neighbors in $B_v^{(i)}$. `Vertex-Pair` queries are implemented by sampling the relevant bucket.

[2] Consider a $G(n, p)$ graph with small p, say $p = 1/\sqrt{n}$, such that vertices will have $\widetilde{\mathcal{O}}(\sqrt{n})$ neighbors with high probability. After $\widetilde{\mathcal{O}}(\sqrt{n})$ `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.

2.3.1 Applications to Random Graph Models

We now consider the application of our construction above to actual random graph models, where we must realize the assumption that $\prod_{u=a}^{b}(1-p_{vu})$ and $\sum_{u=a}^{b} p_{vu}$ can be computed efficiently. For the Erdös-Rényi $G(n,p)$ model, these quantities have simple closed-form expressions. Thus, we obtain implementations of VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR queries, using polylogarithmic resources (time, space and random bits) per query, for *arbitrary* values of p, 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 14), which is tight up to polylogarithmic factors.

We also generalize our construction to implement 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 between $u \in C_i$ and $v \in C_j$ is p_{ij}. As communities in the observed data are generally unknown a priori, and significant research has been devoted to designing efficient algorithms 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., [MNS15]). Thus, we aim to construct implementations where the community assignment of vertices are independently sampled from some given distribution R [3]. The difficulty here is to obtain a uniformly sampled assignment of vertices to communities on-the-fly.

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 [GGN10], 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 D) which may be of independent interest. For r communities, this yields an implementation with $\mathcal{O}(r \cdot \mathrm{poly}(\log n))$ overhead in required resources for each operation.

2.4 Directed Graphs

Lastly, we consider Kleinberg's Small World model ([Kle00, MN04]) in Section E. While Small-World models are proposed to capture properties of observed data such as small shortest-path distances and large clustering coefficients [WS98], this important special case of Kleinberg's model, defined on two-dimensional grids, demonstrates underlying geographical structures of networks.

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 $c/(\textsc{dist}(u,v))^2$. A common choice for c is given by normalizing the distribution such that the expected out-degree of each vertex is 1 ($c = \Theta(1/\log n)$). We can also support a range of values of $c = \log^{\pm\Theta(1)} n$. Since the degree of each vertex in this model is $\mathcal{O}(\log n)$ with high probability, we design implementations supporting ALL-NEIGHBOR queries. In contrast to our previous cases, this model imposes an underlying two-dimensional structure of the vertex set, which governs the distance function as well as complicates the individual edge probabilities.

We design generators for the aforementioned case of the Small-World model, supporting each ALL-NEIGHBORS query, listing all neighbors from closest to furthest away from the queried vertex, using $\mathrm{poly}(\log n)$ resources per query. Providing local access for directed graphs is simpler because the out-neighbors of vertices may be chosen independently at each vertex. So, the main challenge is to sample for the next (closest)

[3] 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.

neighbor, when the probabilities are a function of the Manhattan distance on the lattice. Rather than sampling for a neighbor directly, we sample the next smallest distance with a neighbor, employing the rejection sampling technique that allows efficient sampling through an approximate distribution that have closed-form description, then as a second step, sample for all neighbors for each chosen distance.

2.5 Random Catalan Objects

Consider a one dimensional random walk on the line with n up and n down steps, starting from the origin, with a constraint that the height is always non-negative. We implement two queries, HEIGHT and FIRST-RETURN, where HEIGHT(t) which returns the position of the walk at time t, and FIRST-RETURN(t) returns the first time when the random walk returns to the same position as it was at time t.

Over the course of the execution, our algorithm will sample the height of the walk at many different positions $\{x_1, x_2, \cdots, x_m\}$ (with $x_i < x_{i+1}$), both directly as a result of user given HEIGHT queries, and indirectly through recursive calls to HEIGHT. These sampled positions divide the sequence into contiguous *intervals* $[x_i, x_{i+1}]$, where the height of the endpoints y_i, y_{i+1} have been sampled, but none of the intermediate heights are known. The important observation is that, since the beginning and ending heights are known, the section of the path within an *interval* is completely independent of all other *intervals*. So, each interval $[x_i, x_{i+1}]$ along with the corresponding heights y_i, y_{i+1}, represents a generalized Dyck problem with U up steps, D down steps, and the constraint that the path never goes below 0.

`Height` Queries

We start by implementing a subroutine that given an *interval* $[x_i, x_{i+1}]$ of length of length $2B$ with $2U$ up and $2D$ down steps, samples the number of up steps $U' = U + d$ to the first half of the *interval* (we parameterize U' with d in order to make the analysis cleaner). Note that this is equivalent to answering the query HEIGHT$(x_i + B)$. This is done by sampling the parameter d from a distribution $\{p_d\}$ with $p_d = S_{left}(d) \cdot S_{right}(d)/S_{total}$, where $S_{left}(d)$ (respectively $S_{right}(d)$) is the number of possible paths in the left (resp. right) half of the *interval* when $U + d$ up steps and $D - d$ down steps are assigned to the first half, and S_{total} is the number of possible paths in the original $2B$-interval. General HEIGHT(x) queries can then be answered by recursively halving the *interval*, and sampling the height of the midpoint, until the height of x is sampled.

The problem of sampling the number of up steps in the first half of the *interval* was solved for the case where the sequence is fully random in [GGN10]. Adding the non-negativity constraint introduces further difficulties as the distribution over d has a CDF that is difficult to compute. We construct a different distribution $\{q_d\}$ that approximates $\{p_d\}$ pointwise to a factor of $\log n$ and has an efficiently computable CDF. This allows us to sample from $\{q_d\}$ and leverage the rejection sampling lemma (Lemma 2) to obtain samples from $\{p_d\}$.

`First-Return` Queries

FIRST-RETURN queries present an additional challenge because we don't know which *interval* contains the first return. Since there could be up to $\Theta(n)$ intervals, is it inefficient to iterate through all of them. To circumvent this problem, we allow each interval to sample its own boundary constraint $k > 0$ instead of using the global non-negativity constraint. A boundary constraint of k implies that the path within the interval $[x_i, x_{i+1}]$ never reaches the height $y_i - k$ or lower. Additionally, we maintain an invariant that states that this boundary $x_i - k$ coincides with $\min(x_i, x_{i+1})$. If this constraint is satisfied, we can find the interval containing FIRST-RETURN(x) by finding the smallest sampled position $x_i > x$ whose sampled height $y_i \leq$ HEIGHT(x), and considering the interval $[x_{i-1}, x_i]$ preceding x_i.

Every time the HEIGHT algorithm creates new intervals by sub-dividing an existing one, this invariant is potentially broken. We re-establish it by sampling a "mandatory boundary" (a boundary constraint

with the additional restriction that some position within the interval $[x_i, x_{i+1}]$ *must* touch the boundary), and then sampling a position x such that $x_i < x < x_{i+1}$ and $\textsc{Height}(x) = y$. The first step of sampling the mandatory boundary is performed by binary searching on the possible boundary locations. To find a position that touches this boundary, we parameterize the position with d and find the distribution $\{p_d\}$ associated with these events. We then define a piecewise continuous PDF $\hat{q}(\delta)$ such that $\hat{q}(\delta)$ approximates $p_{\lfloor \delta \rfloor}$. We then use this to construct $q_d = \int_d^{d+1} \hat{q}(\delta)$, where the CDF of q_d is efficently computable using integration, and use rejection sampling (Lemma 2) again to sample indirectly from $\{p_d\}$.

2.6 Random Coloring of a Graph

This should probably be reduced

Finally, we introduce a new model for implementating huge random objects where the distribution is specified as a uniformly random solution to a huge combinatorial problem. In all the problems we have considered so far as well as the ones studied in prior work [GGN10, NN07, ELMR17], the description size of the random object is small (typically $\mathcal{O}(\log n)$ to represent the size of the instance and a constant number of parameters). In this new setting, we will implement local query access to random q-colorings of a given huge graph G of size n with maximum degree Δ. The distribution in this case is defined by the graph structure which has size $\mathcal{O}(n\Delta)$. We present the following definition for local access implementations in this setting.

▶ **Definition 3.** *Given a combinatorial problem on graphs, a* local access implementation *of a family of query functions $\langle F_1, F_2, \cdots \rangle$, provides an oracle $\mathcal{A}$ with the following properties. $\mathcal{A}$ has query access to a graph G, and a tape of public random bits $\mathbf{R}$. Assuming that the solution set of the combinatorial problem on G is $\mathbb{X}$, $\mathcal{A}$ upon being queried with F_i, returns the value $F_i(X)$ for a specific solution $X \in \mathbb{X}$ where the choice of X depends only on $\mathbf{R}$, and the distribution of X (over $\mathbf{R}$) is ϵ-close to the uniform distribution over $\mathbb{X}$. Two different instances of $\mathcal{A}$ with the same graph oracle and the same random bits, must agree on the choice of X that is consistent with all answered queries regardless of what queries were actually asked.*

We can contrast this definition with the one for *Local Computation Algorithms* [RTVX11, ARVX12] which also allow query access to *some* valid solution and can read the input through local probes. An additional difficulty in our setting is that we also have to make sure that we return a solution from the correct distribution. Similarly to LCAs, we can have multiple independent instances of our algorithm answering different queries, but remaining consistent with one another.

`Color` queries

Given a graph G with maximum degree Δ, and the number of colors $q \geq 9\Delta$, we are able to construct an efficient implementation for $\textsc{Color}(v)$ that returns the final color of v in a uniformly random q-coloring of G using only a sub-linear number of probes. Random colorings of a graph are sampled using $\mathcal{O}(n \log n)$ iterations of a Markov chain [FV07]. Each step of the chain proposes a random color update for a random vertex, and accepts the update if it does not create a conflict. This is an inherently sequential process, with the acceptance of a particular proposal depending on all preceding neighboring proposals.

To make the runtime analysis simpler, we define a modified version of Glauber Dynamics that proceeds in $\mathcal{O}(\log n)$ epochs. In each epoch, all of the n vertices propose a random color and update themselves if their proposals do not conflict with any of their neighbors. This Markov chain is a special case of the one presented in [FG18] for distributed graph coloring, and mixes in $\mathcal{O}(\log n)$ epochs when $q \geq 9\Delta$. In order to implement the query $\textsc{Color}(v)$, it suffices to implement a query $\textsc{Accepted}(v, t)$ that indicates whether the proposal for v was accepted in the t^{th} epoch. The answer to this question depends on the prior colors of the potentially Δ neighbors of v. Naively sampling the colors of all these neighbors would result in Δ recursive invocations on the previous epoch $(t-1)$, and stepping *backwards* through the epochs to find the

last accepted proposal. Naively, this leads to a bound of Δ^t on the number of recursive invocations.

We can improve this somewhat by only considering neighbors w of v who had any proposal for the same color c. In this case the expected number of recursive calls is bounded by $t\Delta/q$ ($t\Delta$ proposals to consider and each one is c with probability $1/q$). So, if $q > t\Delta = \mathcal{O}(\Delta \log n)$, this allows us to bound the total number of resulting invocations. The improvement to $q \geq 9\Delta$ comes from the observation that for $w \in \Gamma(v)$ such that w proposed color c at epoch t', the recursive call for w can jump to epoch t' and then step *forwards* through the epochs to find the first accepted proposal. We show that this dramatically reduces the sub-problem size (given by t') in each recursion, thus allowing us to bound the runtime by $\mathcal{O}\left(t\Delta n^{6.12\Delta/q}\right)$ which is sub-linear for $q \geq 9\Delta$.

3 Local-Access Implementations for Random Undirected Graphs

In this section, we provide an efficient local access implementations for random undirected graphs when the probabilities $p_{uv} = \mathbb{P}[(u, v) \in E]$ are given, and we can efficiently approximate the following quantities: (1) the probability that there is no edge between a vertex u and a range of consecutive vertices $[a, b]$, namely $\prod_{u=a}^{b}(1 - p_{vu})$, and (2) the sum of the edge probabilities (i.e., the expected number of edges) between u and vertices from $[a, b]$, namely $\sum_{u=a}^{b} p_{vu}$. In Section 3.4, we provide subroutines for computing these values for the Erdös-Rényi model and the Stochastic Block model. We also begin by assuming perfect-precision arithmetic, which we relax in Section B.1.

First, consider the adjacency matrix $\mathbf{A}$ of G where each entry $\mathbf{A}[u][v]$ can exist in three possible states: $\mathbf{A}[u][v] = 1$ or 0 if the algorithm 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 (in fact, the marginal probability of $\mathbb{P}[(u, v) \in E]$ conditioned on all prior samples is still p_{uv}). Our implementation also maintains the vector $\mathbf{last}$ (used in the same sense as [ELMR17]), where $\mathbf{last}[v]$ records the neighbor of v returned in the last call $\textsc{Next-Neighbor}(v)$, or $\mathbf{last}[v] = 0$ if no such call has been invoked. All cells of $\mathbf{A}$ and $\mathbf{last}$ are initialized to ϕ and 0, respectively.

We use the Bernoulli random variable $X_{uv} \sim \mathsf{Bern}(p_{uv})$ when sampling the value of $\mathbf{A}[u][v] = \phi$. For the sake of analysis, we will frequently view our random process as if the *entire* table of random variables X_{uv} has been sampled *up-front* (i.e. all coins have already been flipped), 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_{uv} once the graph generation is complete.

Obstacles for maintaining A explicitly

First, consider a naive implemention that fills out the cells of $\mathbf{A}$ one-by-one as required by each query; equivalently, we perform $\textsc{Vertex-Pair}$ queries on successive vertices until a neighbor is found. There are two problems with this approach. Firstly, the algorithm only finds a neighbor, for a $\textsc{Random-Neighbor}$ or $\textsc{Next-Neighbor}$ query, with probability p_{uv}, which requires too many iterations: for $G(n, p)$ this requires $1/p$ iterations, which is already infeasible for $p = o(1/\text{poly}(\log n))$. Secondly, the algorithm may generate a large number of non-neighbors in the process, possibly in random or arbitrary locations.

3.1 `Next-Neighbor` Queries via Run-of-0's Sampling

We implement $\textsc{Next-Neighbor}(v)$ by sampling for the first index $u > \mathbf{last}[v]$ such that $X_{vu} = 1$, from a sequence of Bernoulli RVs $\{X_{v,u}\}_{u>\mathbf{last}[v]}$. To do so, we sample a consecutive "run" of 0's with probability $\prod_{u=\mathbf{last}[v]+1}^{u'}(1 - p_{vu})$: 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 $\textsc{Next-Neighbor}(u)$ for $u > \mathbf{last}[v]$. To mitigate this issue, 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 3.1.1. 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 3.1.2 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.

3.1.1 Data structure

From the definition of X_{uv}, NEXT-NEIGHBOR(v) is given by $\min\{u > \mathbf{last}[v] : X_{vu} = 1\}$. Let $P_v = \{u : \mathbf{A}[v][u] = 1\}$ be the set of known neighbors of v, and $w_v = \min\{(P_v \cap (\mathbf{last}[v], n])\}$ 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 $\mathbf{last}[v]$. If there is no known neighbor of v after $\mathbf{last}[v]$, we set $w_v = n + 1$. 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 $X_{vu} = 1$ in this range, or w_v if no such index exists.

We keep track of $\mathbf{last}[v]$ in a dictionary, to avoid any initialization overhead. Each P_v is maintained as an ordered set, which is also instantiated when it becomes non-empty. When NEXT-NEIGHBOR(v) returns u, we add v to P_u and u to P_v. We do not attempt to 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.

▶ **Lemma 4.** *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* (i.e. its value is changed from ϕ to 0) during the first call to NEXT-NEIGHBOR(v) that returns a value $u' > u$ thereby setting $\mathbf{last}[v] = u' \implies u < \mathbf{last}[v]$, or vice versa. □

3.1.2 Queries and Updates

We now present Algorithm 1, 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_{uv} (introduced in Section 3). Consider 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_{vu} corresponding to $u = \mathbf{last}[v] + i$ has bias p_{vu}, regardless of whether X_{vu}'s are decided or not. Then, we use the sequence $\langle C_{vu} \rangle$ to assign values to *undecided* random variable X_{vu}. The main observation here is that, the *decided* random variables $X_{vu} = 0$ do not need coin-flips, and the corresponding coin result C_{vu} can simply be discarded. Thus, we generate coin-flips until we encounter some u satisfying both $C_{vu} = 1$ and $\mathbf{A}[v][u] = \phi$.

Algorithm 1 Sampling NEXT-NEIGHBOR

```
1:  procedure NEXT-NEIGHBOR(v)
2:      u ← last[v]
3:      w_v ← min{(P_v ∩ (u, n]) ∪ {n+1}}
4:      while u = w_v or last[u] < v
5:          sample F ~ F(v, u, w_v)
6:          u ← F
7:      if u ≠ w_v
8:          P_v ← P_v ∩ {u}
9:          P_u ← P_u ∩ {v}
10:     last[v] ← u
11:     return u
```

Let $\mathsf{F}(v, a, b)$ denote the probability distribution of the occurrence u of the first coin-flip $C_{vu} = 1$ 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 $\mathbb{P}[F = f] = \prod_{u=a+1}^{f-1}(1 - p_{vu}) \cdot p_{vf}$. For convenience, let $F = b$ denote the event where all $C_{vu} = 0$. 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 1. As this process generates u satisfying $C_{vu} = 1$ in the increasing order, we repeat until we find one that also satisfies $\mathbf{A}[u][v] = \phi$. Note that once the process terminates at some u, we make no implications on the results of any uninspected coin-flips after C_{vu}.

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 starting 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_{vu}'s, and is generally not uniform. Secondly, in Section 3.1.1, 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 support Random-Neighbor queries, we will likely need to 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. Specifically, to speed-up the sampling process for small p_{vu}'s, we must generate many random non-neighbors at once, 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.

3.2 Final Implementation via the Bucketing Approach

We now resolve both of the above issues, and present an implementation that supports all 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* $\mathbf{B}_v = \{B_v^{(i)}, B_v^{(i)}, \ldots\}$, so that each bucket contains, in expectation, roughly the same number of neighbors of v. We implement Random-Neighbor(v) by randomly selecting a bucket $B_v^{(i)}$, filling in entries $\mathbf{A}[v][u]$ for $u \in B_v^{(i)}$ with 1's and 0's, and then reporting a random neighbor from this bucket. As the bucket size may be large when the probabilities are small, instead of using a linear scan, our Fill subroutine will be implemented using the "run-of-0s" sampling from Algorithm 1 (see Section 3.1). 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 $\deg(v)$. Leveraging the fact that the maximum number of neighbors in any bucket is $\mathcal{O}(\log n)$, we show not only that the probability 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 $\mathrm{poly}(\log n)$, achieving the overall $\mathrm{poly}(\log n)$ complexities. Here in this section, we will extensively rely on the assumption that the expected number of neighbors for consecutive vertices, $\sum_{u=a}^{b} p_{vu}$, can be approximated efficiently.

3.2.1 Partitioning and Filling the Buckets

We fix some sufficiently large constant L, and assign the vertex u to the $\lceil \sum_{i=1}^{u} p_{vi}/L \rceil^{\text{th}}$ bucket of v. Essentially, each bucket represents a contiguous range of vertices, where the expected number of neighbors of v in the bucket is in $[L-1, L+1]$ (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 $L - 1 < \mathbb{E}\left[|\Gamma^{(i)}(v)|\right] < L + 1$ for every $i < |\mathbf{B}_v|$ (i.e., the condition holds for all buckets except 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 (proven in Section B).

▶ **Lemma 5.** *With high probability, the number of neighbors in every bucket, $|\Gamma^{(i)}(v)|$, is at most $\mathcal{O}(\log n)$.*

▶ **Lemma 6.** *With high probability, for every v such that $|\mathbf{B}_v| = \Omega(\log n)$ (i.e., $\mathbb{E} = \Omega(\log n)$), at least a 1/3-fraction of the buckets $\{B_v^{(i)}\}_{i \in [|\mathbf{B}_v|]}$ are non-empty.*

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 3.1. We define the behaviors of the procedure $\textsc{Fill}(v, i)$ as follows. When invoked on an unfilled bucket $B_v^{(i)}$, $\textsc{Fill}(v, i)$ decides 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_{vu} is already decided; in other words, update $P_v^{(i)}$ to $\Gamma^{(i)}(v)$. Then $B_v^{(i)}$ is marked as filled. We postpone the description of our implementation of $\textsc{Fill}$ to Section 3.3, instead using it as a black box.

3.2.2 Putting it all together: `Random-Neighbor` queries

Consider Algorithm 2 for sampling a random neighbor via rejection sampling. For simplicity, throughout the analysis, we assume $|\mathbf{B}_v| = \Omega(\log n)$; otherwise, invoke $\textsc{Fill}(v, i)$ for all $i \in [|\mathbf{B}_v|]$ to obtain the entire neighbor list $\Gamma(v)$.

Algorithm 2 Bucket sampling.

procedure RANDOM-NEIGHBOR(v)
  while True
    sample $B_v^{(i)} \sim_{\mathcal{U}} \mathbf{B}_v$ u.a.r.
    if $B_v^{(i)}$ is not *filled*
      FILL(v, i)
    with probability $\frac{|P_v^{(i)}|}{M}$
      return $u \sim_{\mathcal{U}} P_v^{(i)}$ u.a.r

To obtain a random neighbor, we first choose a bucket $B_v^{(i)}$ uniformly at random, and invoke $\textsc{Fill}(v, i)$ if the bucket is *unfilled*. Then, we *accept* the sampled bucket for generating our random neighbor with probability proportional to $|P_v^{(i)}|$. More specifically, if $M = \Theta(\log n)$ is an upper bound on the maximum number of neighbors in any bucket (see Lemma 5), we accept bucket $B_v^{(i)}$ 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 sample another bucket. If we choose to accept $B_v^{(i)}$, we return a random neighbor from $P_v^{(i)}$. Otherwise, *reject* this bucket and repeat the process again.

Since the returned vertex is always a member of $P_v^{(i)}$, a valid neighbor is always returned. We now show that the algorithm correctly samples a uniformly random neighbor and bound the number of iterations required for the rejection sampling process.

▶ **Lemma 7.** *Algorithm 2 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. Fixing a single iteration and consider a vertex $u \in P_v^{(i)}$, we compute the probability that u is accepted. The probability that $B_v^{(i)} i$ is chosen is $1/|\mathbf{B}_v|$, 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 $1/(|\mathbf{B}_v| \cdot M)$, which is positive and independent of u. Therefore, each vertex is returned with the same probability. ◻

▶ **Lemma 8.** *Algorithm 2 terminates in $\mathcal{O}(\log n)$ iterations in expectation, or $\mathcal{O}(\log^2 n)$ iterations with high probability.*

Proof. From Lemma 6, a (1/3)-fraction of the buckets are non-empty with high probability, and hence the probability of choosing a non-empty bucket is at least 1/3. Since $M = \Theta(\log n)$ by Lemma 5, the success probability of each iteration is at least $1/(3M) = \Omega(1/\log n)$, Thus, the number of iterations required is $O(\log^2 n)$ with high probability. ◻

3.3 Implementation of `Fill`

Lastly, we describe the implementation of the FILL procedure, employing the approach of skipping non-neighbors, as developed for Algorithm 1. We aim to simulate the following process: perform coin-tosses C_{vu} with probability p_{vu} 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 $C_{vu} = 1$, then add u to P_v and vice versa if X_{vu} has not previously been decided. Thus, once $B_v^{(i)}$ is filled, we will obtain $P_v^{(i)} = \Gamma^{(i)}(v)$ as desired.

Algorithm 3 Filling a bucket

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

As discussed in Section 3.1, 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 1, **last** serves this purpose by showing that $\mathbf{A}[v][u]$ for all $u \leq \mathbf{last}[v]$ are decided as shown in Lemma 4. 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 3 for the FILL procedure fulfilling the requirement previously given in Section 3.2.1.

▶ **Lemma 9.** *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 3. 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 implementation, 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 3 is the same as the number of coins $C_{vu} = 1$ which is, in expectation, a constant (namely $\sum_{u \in B_v^{(i)}} \mathbb{P}[C_{vu} = 1] = \sum_{u \in B_v^{(i)}} p_{vu} \leq L + 1$).

By tracking the resource required by Algorithm 3 we obtain the following lemma; note that "additional space" refers to the enduring memory that the implementation 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 10.** *Each execution of Algorithm 3 (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 $\prod_{u \in [a,b]}(1 - p_{vu})$ and $\sum_{u \in [a,b]} p_{vu}$ 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 3 only depends on its random coin-flips and independent of the state of the algorithm. Combining with Lemma 8, we finally obtain polylogarithimc resource bound for our implementation of RANDOM-NEIGHBOR.

▶ **Corollary 11.** *Each execution of Algorithm 2 (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* $\prod_{u\in[a,b]}(1-p_{vu})$ *and* $\sum_{u\in[a,b]} p_{vu}$ *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 9 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 6, 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 12.** *Given a random graph model defined by the probabilty matrix* $\{p_{uv}\}$ *and assuming that we can compute the the quantities* $\prod_{u=a}^{b}(1-p_{vu})$ *and* $\sum_{u=a}^{b} p_{vu}$ *in polylogarithmic time, there exists a local-access implementation for this random graph model that supports* RANDOM-NEIGHBOR, VERTEX-PAIR *and* NEXT-NEIGHBOR *queries using polylogarithmic running time, additional space, and random words per query.*

We have also been implicitly assuming perfect-precision arithmetic and we relax this assumption in Section B.1. In the following Section 3.4, we show applications of Theorem 12 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 Theorem 12.

3.4 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 12, we must provide a method for computing the quantities $\prod_{u=a}^{b}(1-p_{vu})$ and $\sum_{u=a}^{b} p_{vu}$ 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, $p_{vu} = p$ 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.

3.4.1 Erdös-Rényi Model

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

▶ **Corollary 13.** *The final algorithm in Section 3 locally implements 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 14.** *The final algorithm in Section 3 can generate a complete random graph from the Erdös-Rényi $G(n, p)$ model using overall $\widetilde{\mathcal{O}}(n + m)$ time, random bits and space, which is $\widetilde{\mathcal{O}}(pn^2)$ in expectation. This is optimal up to $\mathcal{O}(\mathrm{poly}(\log n))$ factors.*

3.4.2 Stochastic Block model

In 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 $\prod_{u=a}^{b}(1 - p_{vu}) = \prod_{j=1}^{r}(1 - p_{ij})^{|[a,b] \cap C_j|}$ and $\sum_{u=a}^{b} p_{vu} = \sum_{j=1}^{r} |[a, b] \cap C_j| \cdot p_{ij}$. Thus, it is sufficient to design a data structure, 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 an implementation for the Stochastic Block model using $O(r\,\mathrm{poly}(\log n))$ resources per query.

▶ **Theorem 15.** *There exists a data structure that samples a community for each vertex independently at random from R with $\frac{1}{\mathrm{poly}(n)}$ 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 $O(r\,\mathrm{poly}(\log n))$ 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 16.** *The final algorithm in Section 3 implements a random graph from the Stochastic Block model with randomly-assigned communities using $O(r\,\mathrm{poly}(\log n))$ time, random N-bit words, and additional space per query with high probability.*

We provide the full details of the construction in Section D. Our construction extends a similar implementation in the work of [GGN10] which only supports $r = 2$. The overall data structure is a balanced binary tree, where the root corresponds to the entire range of indices $[n]$, and the children of each vertex

correspond to the first and second half of the parent's range. Each node[4] 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 [5] (for n samples (vertices) from the probability distribution R). The children are 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 the counts assigned to either child from the correct marginal distribution. We show how to sample such a count from the *multivariate hypergeometric distribution*, below in Theorem 17 (proven in Section D).

▶ **Theorem 17.** *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 $O(r \cdot \mathrm{poly}(\log B))$ time and random words.*

Proof of Theorem 15. Recall that R denotes the given distribution over integers $[r]$ (namely, the random distribution of communities for each vertex). Our algorithm generates and maintains random variables $X_1, \ldots, X_n$ (denoting the community assignment), each of which is drawn independently from R. Given a pair (i, j), it uses Theorem 15 to sample the vector $\mathbf{C}(i, j) = \langle c_1, \ldots, c_r \rangle$, where c_k counts the number of variables in $\{X_i, \ldots, X_j\}$ that take on the value k.

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 [Knu97]). Then, the children are generated on-the-fly as described above. Thus, each query can be processed within $O(r \,\mathrm{poly}(\log n))$ time, yielding Theorem 15. □

Then, by embedding the information stored by the data structure into the state (as in the proof of Lemma 47), we obtain the desired Corollary 16.

4 Sampling Random Catalan Objects

S_{left} etc are too large to compute, but we can approximate teh ratios.

In the previous Section 3.4.2 on the Stochastic Block Model, we considered random sequences of colored marbles. Next, we focus on an important variant of these sequences as Catalan objects, which impose a global constraint on the types of allowable sequences. Specifically, consider a sequence of n white and n black marbles, such that every *prefix* of the sequence has at least as many white marbles as black ones. Our goal will be to support queries to a uniformly random instance of such an object.

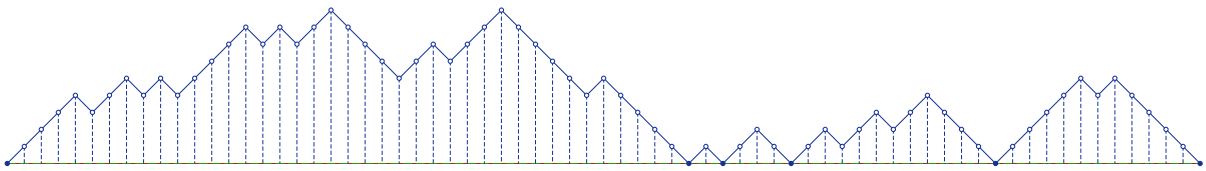

Figure 4 Simple Dyck path with $n = 35$ up and down steps.

One interpretation of Catalan objects is given by Dyck paths (Figure 4). A Dyck path is essentially a $2n$ step *balanced* one-dimensional walk with exactly n up and down steps. In Figure 4, each step moves one unit along the positive x-axis (time) and one unit up or down the positive y-axis (position). The prefix

[4] For clarity, "vertex" is only used in the sampled graph, and "node" is only used in the internal data structures of the algorithm.

[5] See e.g., section 3.4.1 of [Knu97]

constraint implies that the y-coordinate of any point on the walk is ≥ 0 i.e. the walk never crosses the x-axis. The number of possible Dyck paths (see Theorem 58) is the n^{th} Catalan number $C_n = \frac{1}{n+1} \cdot \binom{2n}{n}$. Many important combinatorial objects occur in Catalan families of which these are an example.

We will approach the problem of partially sampling Catalan objects through Dyck paths. This, in turn, will allow us to sample other random Catalan objects such as rooted trees, and bracketed expressions. Specifically, we will want to answer the following queries:

- DIRECTION(i): Returns the value of the i^{th} step in the Dyck path (whether the step is up or down).
- HEIGHT(i): Returns the position of the path after i steps.
- FIRST-RETURN(i): Returns an index $j > i$ such that HEIGHT(j) = HEIGHT(i), and for any other k between i and j, HEIGHT(k) is strictly greater than HEIGHT(i). While it may not be clear why this kind of query is important, it will be useful for querying bracketed expressions and random trees. We defer this discussion to Section 4.1.

Since a DIRECTION(i) query can be simulated using the queries HEIGHT(i) and HEIGHT($i-1$), we will not explicitly discuss the DIRECTION queries in what follows.

4.1 Bijections to other Catalan objects

The HEIGHT query is natural for Dyck paths, but the FIRST-RETURN query is important in exploring other Catalan objects. For instance, consider a random well bracketed expression; equivalently an uniform distribution over the Dyck language. One can construct a trivial bijection between Dyck paths and words in this language by replacing up and down steps with opening and closing brackets respectively. The HEIGHT query corresponds to asking for the nesting depth at a certain position in the word, and FIRST-RETURN(i) returns the position of the matching bracket for position i.

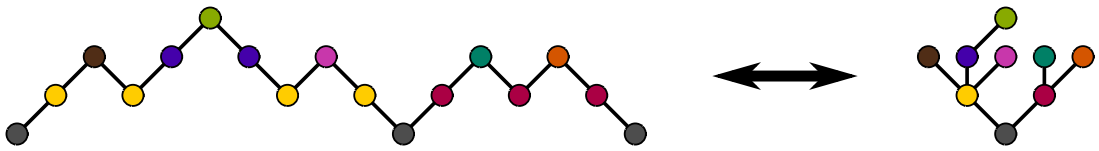

Figure 5 Bijection between Dyck paths and ordered rooted trees. For example, successive FIRST-RETURN queries on the yellow node would reveal its three children in order from left to right.

There is also a natural bijection between Dyck paths and ordered rooted trees (Figure 5), by considering the Dyck path to be a transcript of a DFS traversal on the tree. Starting with the root, for each "up-step" we move to a new child of the current node, and for each "down-step", we backtrack towards the root. Here, the HEIGHT query returns the depth of a node and the FIRST-RETURN query can be used to find the *next child* of a node. A similar argument shows a bijection to the set of all ordered binary trees. Moving forwards, we will focus on Dyck paths for the sake of simplicity.

4.2 Catalan Trapezoids and Generalized Dyck Paths

In order to sample Dyck paths locally, we will need to analyze more general Catalan objects. Specifically, we consider a sequence of U up-steps and D down-steps, such that any prefix of the sequence containing U' up and D' down steps satisfies $U' - D' \geq 1 - k$. This means that we start our Dyck path at a height of $k-1$, and we are never allowed to cross below zero (Figure 6). Note that the case $k = 1$ corresponds to the standard description of Dyck paths, as mentioned previously (Figure 4).

We will denote the set of such *generalized Dyck paths* as $\mathbb{C}_k(U, D)$ and the number of paths as $C_k(U, D) = |\mathbb{C}_k(U, D)|$, which is an entry in the *Catalan Trapezoid* of order k [Reu14]. We also use $\mathsf{C}_k(U, D)$ to denote

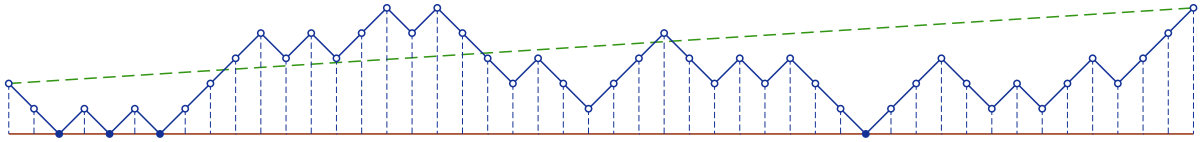

Figure 6 Generalized Dyck path with $U = 25$, $D = 22$ and $k = 3$. Note that the boundary is $k - 1 = 2$ units below the starting height.

the uniform distribution over $\mathbb{C}_k(n, m)$. Now, we state a result from [Reu14] without proof:

$$C_k(U, D) = \begin{cases} \binom{U+D}{D} & 0 \le D < k \\ \binom{U+D}{D} - \binom{U+D}{D-k} & k \le D \le U + k - 1 \\ 0 & D > U + k - 1 \end{cases} \tag{1}$$

For $k = 1$ and $n = m$, these represent the vanilla Catalan numbers i.e. $C_n = C_1(n, n)$ (number of simple Dyck paths). Our goal is to sample from the distribution $\mathsf{C}_1(n, n)$.

Consider the situation after a sequence of HEIGHT queries to the Dyck path at various locations $\langle x_1, x_2, \cdots, x_m \rangle$, such that the corresponding heights were sampled to be $\langle y_1, y_2, \cdots, y_m \rangle$. These revealed locations partition the path into disjoint *intervals* $[x_i, x_{i+1}]$, where the heights of the endpoints of each interval have been determined (as $y_i = \textsc{Height}(x_i)$). We notice that these intervals can be sampled independently of each other. Specifically, the path within the interval $[x_i, x_{i+1}]$ will be sampled from $\mathsf{C}_k(U, D)$, where $k - 1 = y_i$, $U + D = x_{i+1} - x_i$, and $U - D = y_{i+1} - y_i$. Moreover, since the heights of the endpoints y_i and y_{i+1} are known, this choice is independent of any samples outside the interval. Next, in Section 4.3, we will show how one can sample heights within such an interval, and in Section 4.4 we will move on to the more complicated FIRST-RETURN queries.

4.3 Sampling the Height

We implement $\textsc{Height}(t)$ by showing how to (efficiently) sample the height of the path in the midpoint of an existing interval $[x_i, x_{i+1}]$ (where the heights of the endpoitns are y_i and y_{i+1}). We can then extend this to arbitrary positions by running a binary search on the appropriate interval using the midpoint samples. If the interval in question has odd length, we sample one step on the boundary, and proceed with a shortened even length interval.

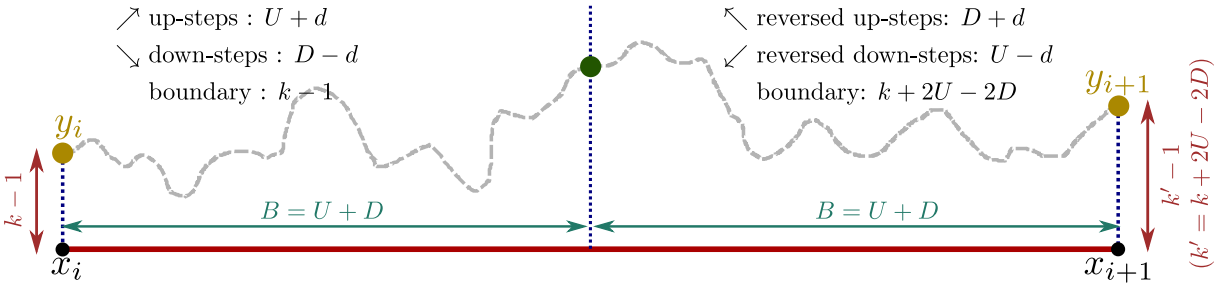

Figure 7 The $2B$-interval is split into two equal parts resulting in two separate Dyck problems. The green node (center) is the sampled height of the midpoint parameterized by the value of d. The path considered in both sub-intervals starts at a yellow node (left and right edges) and ends at the green node. From this perspective, the path on the right is reversed with up and down steps being swapped. A possible path is shown in gray.

What if U and D are not even?

Our general recursive step is as follows. We consider an interval of length $2B$ comprising of $2U$ up-steps and $2D$ down-steps where the sum of any prefix cannot be less than $k-1$ i.e. the path within this interval should be sampled from $\mathsf{C}_k(2U, 2D)$ (the factors of two makes the analysis cleaner). Without loss of generality, we assume that $D \leq U$; if this were not the case, we could simply flip the interval, and swap the up and down steps. This ensures that the overall path in the interval is non-decreasing in height.

We sample the height of the path $B = U + D$ steps into the interval at the midpoint (see Figure 7). This is equivalent to sampling the number of up or down steps that get assigned to the first half of the interval. We parameterize the possibilities by d and define p_d to be the probability that exactly $U + d$ up-steps and $D - d$ down steps get assigned to the first half, and therefore the second half gets exactly $U - d$ up steps and $D + d$ down steps.

$$p_d = \frac{S_{left}(d) \cdot S_{right}(d)}{S_{total}(d)} \tag{2}$$

Here, $S_{left}(d)$ denotes the number of possible paths in the first half (using $U + d$ up steps) and $S_{right}(d)$ denotes the number of possible paths in the second half (using $U - d$ up steps). Note that all of these paths have to respect the k-boundary constraint (cannot dip more than $k-1$ units below the starting height), where $k = y_i + 1$. Moving forwards, we will drop the d when referring to the path counts. We (conceptually) flip the second half of the interval, such that the corresponding path begins from the end of the $2B$-interval and terminates at the midpoint (Figure 7). This results in a different starting point, and the prefix/boundary constraint will also be different. Hence, we define $k' = k + 2U - 2D$ to represent the new boundary constraint (since the final height of the $2B$-interval is $k' - 1$). Finally, S_{total} is the total number of possible paths in the $2B$ interval.

We will now use the rejection sampling lemma (Lemma 2). An important point to note is that in order to apply this lemma, we must be able to compute the p_d values at least approximately. For now, we assume that we have access to an oracle that will compute the value for us. Lemma 63 in Section F shows how to construct such an oracle. We also use the following lemma to bound the deviation of the path with high probability.

▶ **Lemma 18.** *Consider a contiguous* sub-path *of a simple Dyck path of length* $2n$ *where the sub-path is of length* $2B$ *comprising of* U *up-steps and* D *down-steps (with* $U + D = 2B$*). Then there exists a constant* c *such that the quantities* $|B - U|$, $|B - D|$, *and* $|U - D|$ *are all* $< c\sqrt{B \log n}$ *with probability at least* $1 - 1/n^2$ *for every possible sub-path.*

This lemma allows us to ignore potential midpoint heights that cause a deviation greater than $c\sqrt{B \log n}$. A proof is presented in Section F.2. One implication of Lemma 18 is that with high probability, the correctly sampled value for d will be $\mathcal{O}(\sqrt{B \log n})$. In other words, The height of the midpoint takes on one of only $\mathcal{O}(\sqrt{B \log n})$ distinct values with high probability. This immediately suggests a $\tilde{\mathcal{O}}(\sqrt{B})$ time algorithm for sampling the midpoint height, by explicitly computing the probabilities of each of these potential heights, and directly sampling from the resulting distribution. However, we can go further and obtain a $\mathcal{O}(poly(\log n))$ time algorithm.

4.3.1 The Simple Case: Far Boundary

We first consider the case when the boundary constraint is far away from the starting point, i.e. k is large. The following lemma (proof in Section F.2) shows that in this case, we can safely *ignore* the constraint. Intuitively, this is because the boundary is so far away, that we do not hit it with high probability even if we choose a random *unconstrained* path.

▶ **Lemma 19.** *Given a Dyck path sampling problem of length* B *with* U *up and* D *down steps with a boundary at* k*, there exists a constant* c *such that if* $k > c\sqrt{B \log n}$*, then the distribution of paths sampled without a boundary* $\mathsf{C}_\infty(U, D)$ *(hypergeometric sampling) is statistically* $\mathcal{O}(1/n^2)$*-close in* L_1 *distance to the distribution of Dyck paths* $\mathsf{C}_k(U + D)$.

By Lemma 19, the problem of sampling from $C_k(2U, 2D)$ reduces to sampling from the hypergeometric distribution $C_\infty(2U, 2D)$ when $k > \mathcal{O}(\sqrt{B \log n})$ i.e. the probabilities p_d can be approximated by:

$$q_d = \frac{\binom{B}{D-d} \cdot \binom{B}{D+d}}{\binom{2B}{2D}}$$

This problem of sampling from the hypergeometric distribution is implemented using $\mathcal{O}(poly(\log n))$ resources in [GGN10] (see Lemma 50 in Section D). We also used this result earlier in the paper in order to find the community assignments in the Stochastic Block Model.

4.3.2 The Difficult Case: Intervals Close to Zero

The difficult case is when $k = \mathcal{O}(\sqrt{B \log n})$, and the previous approximation due to Lemma 19 no longer works. In this case, we cannot just ignore the boundary constraint, and instead we have to analyze the true probability distribution given by p_d. We obtain an expression for p_d by substituting the formula for generalized Catalan numbers as follows: (Equation 1) into Equation 2.

$$S_{left} = C_k(U + d, D - d) \qquad S_{right} = C_{k'}(U - d, D + d) \qquad S_{total} = C_k(2U, 2D) \tag{3}$$

Since the right interval is flipped in our analysis, this changes the prefix/boundary constraint, and hence, the expression for S_{right} uses $k' = k + 2U - 2D$. This also implies that $k' = \mathcal{O}(\sqrt{B \log n})$ (using Lemma 18). We can now use Equation 2 to evaluate the probabilties $p_d = S_{left} \cdot S_{right}/S_{total}$. Recall that $p_d = S_{left}(d) \cdot S_{right}(d)/S_{total}(d)$, where S_{left} and S_{right} are the number of possible paths in the left and right half of the interval respectively, when exactly $U + d$ up steps are assigned to the first half. S_{total} is the total number of possible paths in the interval.

We will invoke the rejection sampling technique (Lemma 2), by constructing a different distribution q_d that approximates p_d up to logarithmic factors over the vast majority of its support (we ignore all $|d| > \Theta(\sqrt{B \log n})$ since the associated probability mass is negligible by Lemma 18). In order to perform rejection sampling, we also need good approximations of p_d, which is acheved by Lemma 63 in Section F.3. Next, we define an appropriate q_d that approximates p_d and also has an *efficiently computable CDF*. Surprisingly, as in Section 4.3.1, we will be able to use the hypergeometric distribution for q_d,

$$q_d \equiv \frac{\binom{B}{D-d} \cdot \binom{B}{D+d}}{\binom{2B}{2D}} = \frac{\binom{B}{D-d} \cdot \binom{B}{U-d}}{\binom{2B}{2D}}$$

However, the argument for why this q_d is a good approximation to p_d is far less straightforward.

First, we consider the case where $k \cdot k' \leq 2U + 1$. In this case, we use loose bounds for $S_{left} < \binom{B}{D-d}$ and $S_{right} < \binom{B}{U-d}$. The preceding upper bounds hold because $\binom{B}{D-d}$ and $\binom{B}{U-d}$ are the total number of *unconstrained* paths in the left and right half respectively, and adding the boundary constraint can only reduce the number of paths. We also prove the following lemma in Section F to bound the value of S_{right}.

▶ **Lemma 20.** *When $kk' > 2U + 1$, $S_{total} > \frac{1}{2} \cdot \binom{2B}{2D}$.*

Combining the three bounds, we conclude that $p_d < \frac{1}{2} q_d$. Intuitively, in this case the Dyck boundary is far away, and therefore the number of possible paths is only a constant factor away from the number of unconstrained paths (see Section 4.3.1). The case where the boundaries are closer (i.e. $k \cdot k' \leq 2U + 1$) is trickier, since the individual counts need not be close to the corresponding binomial counts. However, in this case we can still ensure that the sampling probability is within poly-logarithmic factors of the binomial sampling probability. We use the following lemmas (proven in Section F).

▶ **Lemma 21.** *$S_{left} \leq c_1 \frac{k \cdot \sqrt{\log n}}{\sqrt{B}} \cdot \binom{B}{D-d}$ for some constant c_1.*

▶ **Lemma 22.** *$S_{right} \leq c_2 \frac{k' \cdot \sqrt{\log n}}{\sqrt{B}} \cdot \binom{B}{U-d}$ for some constant c_2.*

▶ **Lemma 23.** *When $kk' \leq 2U+1$, $S_{total} \geq c_3 \frac{k \cdot k'}{B} \cdot \binom{2B}{2D}$ for some constant c_3.*

We can now put these lemmas together to show that $p_d/q_d \leq \Theta(\log n)$ and invoke Lemma 2 to sample the value of d. This gives us the height of the Dyck path at the midpoint of the two given points.

▶ **Theorem 24.** *Given two positions a and b (and the associated heights) in a Dyck path of length $2n$, with the guarantee that no position between a and b has been sampled yet, there is an algorithm that returns the height of the path at the midpoint of a and b (or next to the midpoint if $b-a$ is odd). Moreover, this algorithm only uses $\mathcal{O}(poly(\log n))$ resources.*

Proof. If $b-a$ is even, we can set $B = (b-a)/2$. Otherwise, we first sample a single step from a to $a+1$, and then set $B = (b-a-1)/2$. Since there are only two possibilities for a single step, we can explicitly approximate the probabilities, and then sample accordingly. This allows us to apply the rejection sampling from Lemma 2 using $\{q_d\}$ to obtain samples from $\{p_d\}$ as defined above. □

▶ **Theorem 25.** *There is an algorithm that provides sample access to a Dyck path of length $2n$, by answering queries of the form* HEIGHT(x) *with the correctly sampled height of the Dyck path at position x using only $\mathcal{O}(poly(\log n))$ resources per query.*

Proof. The algorithm maintains a successor-predecessor data structure (e.g. Van Emde Boas tree) to store all positions x that have already been sampled. Each newly sampled position is added to this structure. Given a query HEIGHT(x), the algorithm first finds the successor and predecessor (say a and b) of x among the already queried positions. This provides us the guarantee required to apply Theorem 24, which allows us to query the height at the midpoint of a and b. We then binary search by updating either the successor or predecessor of x and repeat until we sample the height of position x. □

4.4 Supporting "First Return" Queries

In this section, we show how to implement more complex queries to a Dyck path. Specifically, we introduce FIRST-RETURN(x) to allow the user to query the next time the path returns to HEIGHT(x) (if at all). The utility of this kind of query can be seen through other random Catalan objects. For instance, if we consider a random well bracketed expression, FIRST-RETURN(x) is the position of the bracket matching the x^{th} one. If we consider a random rooted tree, FIRST-RETURN corresponds to the next child of a vertex (see Section 4.1). An important detail here is that if the first step from x to $x+1$ is an down-step, there is no well defined "FIRST-RETURN". In case of trees, this would correspond to a leaf which has no children.

4.4.1 Maintaining a Boundary Invariant

Notice that after performing a set of HEIGHT queries $\langle x_1, x_2, \cdots, x_m \rangle$ to the Dyck path, many different positions are revealed (possibly in adverserial locations). This partitions the path into at most $m+1$ disjoint and independent *intervals* with known boundary conditions. The first step towards finding the FIRST-RETURN from position t would be to locate the *interval* where the first return occurs. Even if we had an efficient technique to filter intervals, we would want to avoid considering all $\Theta(m)$ intervals to find the correct one. In addition, the fact that a specific interval *does not* contain the first return implies dependencies for all subsequent samples.

We resolve these difficulties by maintaining a new invariant. Consider all positions whose heights have have already been sampled, either directly as a result of user given HEIGHT queries, or indirectly due to

recursive HEIGHT calls; $\langle x_1, x_2, \cdots, x_m \rangle$ in increasing order i.e. $x_i < x_{i+1}$. Let the corresponding heights be $\langle y_1, y_2, \cdots, y_m \rangle$ i.e. $\textsc{Height}(x_i) = y_i$.

▶ **Invariant 26.** *For any interval $[x_i, x_{i+1}]$ where the heights of the endpoints have been sampled to be y_i and y_{i+1}, and no other position in the interval has yet been sampled, the section of the Dyck path between positions x_i and x_{i+1} is constrained to lie above $min(y_i, y_{i+1})$.*

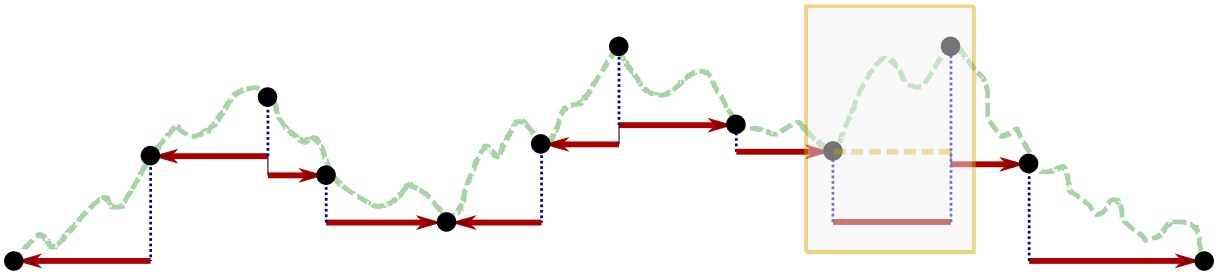

Figure 8 The set of intervals formed by a set of height samples. Each interval also has its own boundary constraint (red). Invariant 26 implies that each boundary must coincide with one of the interval endpoints. Note that the only interval which violates the invariant is the third last one (shown in yellow box).

How can one maintain such an invariant? After sampling the height of a particular position x_i as y_i (with $x_{i-1} < x_i < x_{i+1}$), the invariant is potentially broken on either side of x_i. We re-establish the invariant by sampling an additional point on either side. This proceeds as follows for some violating interval $[x_i, x_{i+1}]$ (example violation in Figure 8):

1. Sample the lowest height h achieved by the walk between x_i and x_{i+1} according to the uniform distribution over all possible paths that respect the current boundary constraint (see Section 4.4.2).
2. Sample an intermediate position x such that $x_i < x < x_{i+1}$ and $\textsc{Height}(x) = h$ (see Section 4.4.3).

The sample $\textsc{Height}(x)$ results in two sub-intervals $[x_i, x]$ and $[x, x_{i+1}]$. Since $h = \textsc{Height}(x)$ has been determined to be the minimum height in the overall range $[x_i, x_{i+1}]$, the invariant is preserved in the new intervals (see Figure 9). Lemma 33 in Section 4.4.5 shows how this invariant can help us efficiently search for the interval containing the first return.

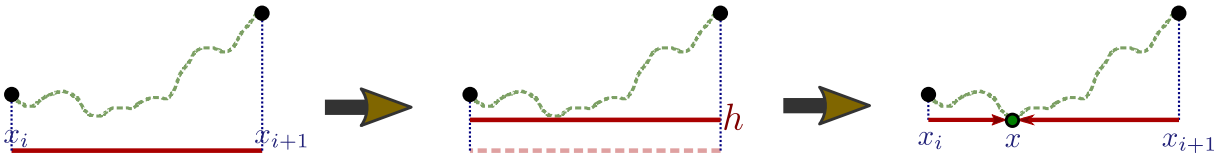

Figure 9 An interval $[x_i, x_{i+1}]$ that violates the invariant is "fixed" by first sampling the lowest height h achieved within the interval, and then sampling a position $x \in [x_i, x_{i+1}]$ such that $Height(x) = h$.

4.4.2 Sampling the Lowest Achievable Height: Mandatory Boundary

First, we need to sample the lowest height h of the walk between x_i and x_{i+1} (with corresponding heights y_i and y_{i+1}). We will refer to h as the *"mandatory boundary"* in this interval; i.e. no height in the interval may be lower than the boundary, but at least one point *must* touch the boundary (have height h). We assume that $y_i \leq y_{i+1}$ without loss of generality; if this is not the case, swap x_i and x_{i+1} and consider the reversed path. Say this interval defines a generalized Dyck problem with U up steps and D down steps and a boundary that is $k - 1$ units below y_i.

Picture?

Algorithm 10 Finding the Mandatory boundary

```
1: function MANDATORY-BOUNDARY(U, D, k)
2:     k_low ← k
3:     k_up ← 0
4:     while k_low < k_up − 1
5:         k_mid ← ⌊(k_low + k_up)/2⌋
6:         P_total ← C_{k_low}(U, D) − C_{k_up}(U, D)
7:         P^{k_mid}_{k_low} ← C_{k_low}(U, D) − C_{k_mid}(U, D)
8:         with probability P^{k_mid}_{k_low} / P_total
9:             k_up ← k_mid
10:        else
11:            k_low ← k_mid
12:    return k_low
```

Given any two boundaries k_{lower} and k_{upper} on this interval (with $k_{lower} < k_{upper} \leq y_i$), we can count the number of possible generalized Dyck paths that violate the k_{upper} boundary but *not* the k_{lower} boundary as:

$$P_{k_{lower}}^{k_{upper}} = C_{k_{lower}}(U, D) - C_{k_{upper}}(U, D)$$

We define the current lower and upper boundaries as $k_{low} = k, k_{up} = 0$, and set $k_{mid} = (k_{low} + k_{up})/2$. Since we can compute the quantities $P_{k_{mid}}^{k_{up}}$, $P_{k_{low}}^{k_{mid}}$, and $P_{total} = P_{k_{low}}^{k_{up}}$, we can sample a single bit to decide if the *"lower boundary"* should move up or if the *"upper boundary"* should move down. We then repeat this binary search until we find $k' = k_{low} = k_{up} - 1$ and k' becomes the *"mandatory boundary"* (i.e. the walk reaches the height exactly $k' - 1$ units below y_i but no lower.

4.4.3 Sampling First Position that Touches the *"Mandatory Boundary"*

Now that we have a "*mandatory boundary*" k, we just need to sample a position x with height $h = x_i - k + 1$. In fact, we will do something stronger by sampling the *first* time the walk touches the boundary after x_i. As before, we assume that this interval contains U up steps and D down steps.

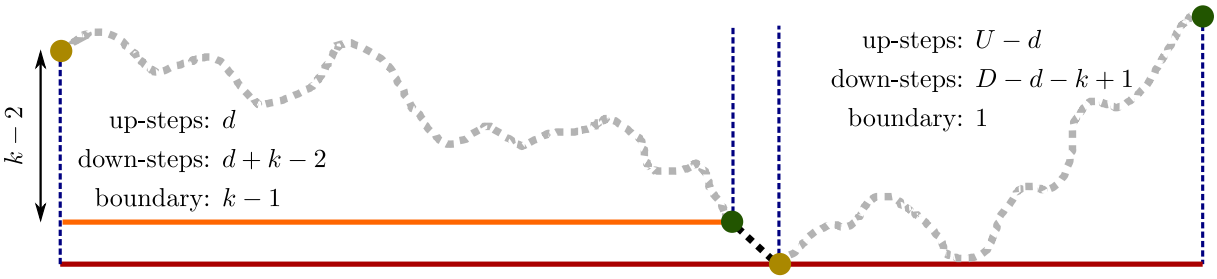

Figure 11 Zooming into the error in Figure 8. We sample a position x (yellow) on the boundary (red), such that the section of the path to the left of x never approaches the red boundary (it respects the orange boundary).

We will parameterize the position x the number of up-steps d between x_i and x (See Figure 11). implying that $x = x_i + 2d + k - 1$. Given a specific d, we want to compute the number of valid paths that result in d up-steps before the first approach to the boundary. Note that unlike Section 4.3, d is used here to parameterize the (horizontal) x-position of the desired point. We will calculate the probability p_d associated with a particular position by counting the total number of paths to the left and right of the first approach and multiplying them together.

As in Section 4.3.2, we define S_{left} to be the number of paths in the sub-interval before the first approach (left side of Figure 11), S_{right} to be the number of paths following the first approach, and S_{total} to be the total number of paths that touch the "mandatory boundary" at k (note that these quantities are functions of U, D, k and d, but we drop the parameters for the sake of clarity):

$$S_{left} = C_k(d, d + k - 2) \qquad S_{right} = C_1(U - d, D - d - k + 1) \qquad S_{total} = C_k(U, D) - C_{k-1}(U, D)$$

Our goal is to sample d from the distribution $\{p_d\}$ where $p_d = S_{left} \cdot S_{right}/S_{total}$. The application of Lemma 2 requires us to approximate $\{p_d\}$ with a "well behaved" $\{q_d\}$ (one whose CDF can be efficiently estimated). Since we only require a asymptotic (up to poly$(\log n)$ factors) approximation to $\{p_d\}$, it suffices

to estimate the number of paths asymptotically as well. Using Equation 1, we obtain approximations for S_{left} and S_{right} (Lemma 27 and Lemma 28 with proofs in Section F.5). Our general strategy will be to integrate continuous versions of these approximations in order to obtain a CDF of some approximating distribution.

▶ **Lemma 27.** *If $d > \log^4 n$, then $S_{left}(d) = \Theta\left(\frac{2^{2d+k}}{\sqrt{d}} e^{-r_{left}(d)} \cdot \frac{k-1}{d+k-1}\right)$ where $r_{left}(d) = \frac{(k-2)^2}{2(2d+k-2)}$. Furthermore, $r_{left}(d) = \mathcal{O}(\log^2 n)$.*

▶ **Lemma 28.** *If $U + D - 2d - k > \log^4 n$, then $S_{right}(d) = \Theta\left(\frac{2^{U+D-2d-k}}{\sqrt{U+d-2d-k}} e^{-r_{right}(d)} \cdot \frac{U-D+k}{U-d+1}\right)$ where $r_{right}(d) = \frac{(U-D-k-1)^2}{4(U+D-2d-k+1)}$. Furthermore, $r_{right}(d) = \mathcal{O}(\log^2 n)$.*

Unfortunately, these continuous approximation functions obtained do not admit closed form integrals. The main culprit is the exponential term in both expressions. We tackle this issue by noticing that the values of the exponents are bounded by $\mathcal{O}(\log^2 n)$ over the majority of the range of d. Within this range of d values, the approximating functions may be further simplified by taking a piecewise linear approximation, where each of the pieces corresponds to a fixed value of the *floor of the corresponding exponent*. This technique is elaborated in Section 4.4.4.

We now consider the *"problematic"* values of d that are outside the range of the two preceding lemmas. These values are the ones where $d < \log^4 n$ or $2d > U + D - k - \log^4 n$. Since d is the number of up steps in the left sub-interval, $d \geq 0$. Further, since the length of the right sub-interval nust be non-negative (see Figure 11), we get $U + D - 2d - k + 1 \geq 0$. Thus, we define the *"problematic"* set:

$$\mathcal{R} = \left\{ d \;\middle|\; 0 \leq d < \log^4 n \text{ or } -1 < 2d - U - D + k < \log^4 n \right\} \tag{4}$$

Clearly, we can bound the size of this set as $|\mathcal{R}| = \mathcal{O}(\log^4 n)$. An immediate consequence of Lemma 27 and Lemma 28 is the following.

▶ **Corollary 29.** *When $d \notin \mathcal{R}$, $S_{left}(d) \cdot S_{right}(d) = \Theta\left(\frac{2^{U+D}}{\sqrt{d(U+D-2d-k)}} \cdot e^{-r(d)} \cdot \frac{k-1}{d+k-1} \cdot \frac{U-D+k}{U-d+1}\right)$ where $r(d) = \mathcal{O}(\log^2 n)$.*

4.4.4 Estimating the CDF

We use these observations to construct a suitable $\{q_d\}$ that can be used to invoke the rejection sampling lemma (Lemma 2). We will achieve this by constructing a piecewise continuous function $\hat{q}$, such that $\hat{q}(\delta)$ approximates $p_{\lfloor \delta \rfloor}$, and then use the integral of $\hat{q}$ to define the discrete distribution $\{q_d\}$. As stated in the previous section, we can leverage the fact that when $d \notin \mathcal{R}$, the floor of the exponent $\lfloor r(d) \rfloor$ only takes $\mathcal{O}(\log^2 n)$ distinct values (consequence of Corollary 29). Since the "problematic" set $\mathcal{R}$ only has $\mathcal{O}(\log^4 n)$ values, we can also deal with these remaining values by simply creating $|\mathcal{R}|$ additional continuous pieces in the function $\hat{q}$. We begin by rewriting $p_d = \Theta\left(\mathcal{K} \cdot f(d) \cdot e^{-r(d)}\right)$ where:

$$\mathcal{K} = \frac{2^{U+D}}{S_{total}} = \frac{2^{U+D}}{C_k(U,D) - C_{k-1}(U,D)} \qquad f(d) = \frac{(k-1)(U-D+k)}{\sqrt{d(U+D-2d-k)}(d+k-1)(U-d+1)} \tag{5}$$

Notice that $\mathcal{K}$ is a constant and $f(d)$ is a function whose integral has a closed form. Using the fact that $r(d) = \mathcal{O}(\log^2 n)$ (Corollary 29), and $|\mathcal{R}| = \mathcal{O}(\log^4 n)$, we obtain the following lemma:

▶ **Lemma 30.** *Given the piecewise continuous function*

$$\hat{q}(\delta) = \begin{cases} p_{\lfloor \delta \rfloor} & \text{if } \lfloor \delta \rfloor \in \mathcal{R} \\ \mathcal{K} \cdot f(\delta) \cdot exp\left(-\lfloor r(\lfloor \delta \rfloor) \rfloor\right) & \text{if } \lfloor \delta \rfloor \notin \mathcal{R} \end{cases} \implies p_d = \Theta\left(\int_d^{d+1} \hat{q}(\delta)\right)$$

Furthermore, $\hat{q}(\delta)$ has $\mathcal{O}(\log^4 n)$ continuous pieces.

Proof. For $d \in \mathcal{R}$, the integral trivially evaluates to exactly p_d. For $d \notin \mathcal{R}$, it suffices to show that $p_d = \Theta\left(\hat{q}(\delta)\right)$ for all $\delta \in [d, d+1)$. We already know that $p_d = \Theta\left(\mathcal{K} \cdot f(d) \cdot e^{-r(d)}\right)$. Moreover, for any $\delta \in [d, d+1)$, the exponential term $e^{-\lfloor r(\lfloor \delta \rfloor) \rfloor}$ in $\hat{q}(\delta)$ is within a factor of e of the original $e^{-r(\lfloor \delta \rfloor)}$ term.

For all $\mathcal{O}(\log^4 n)$ values $d \in \mathcal{R}$, $\hat{q}(\delta)$ is constant on the interval $[d, d+1]$. Since $r(d) = \mathcal{O}(\log^2 n)$ by Corollary 29, the exponential term $e^{-\lfloor r(\lfloor \delta \rfloor) \rfloor}$ in $\hat{q}(\delta)$ taken on at most $\mathcal{O}(\log^2 n)$ values. Thus, $\hat{q}$ is continuous for a range of δ correspoding to a fixed value of $\lfloor r(\lfloor \delta \rfloor) \rfloor$, and so, we conclude that $\hat{q}$ is piecewise continuous with $\mathcal{O}(\log^2 n)$ pieces. ◻

Now, we have everything in place to define the distribution $\{q_d\}$ that we will be sampling from. Specifically, we will define q_d and it's CDF Q_d as follows ($\mathcal{N}$ is the normalizing factor):

$$q_d = \left(\int_d^{d+1} \hat{q}(\delta)\right) \cdot \frac{1}{\mathcal{N}} \qquad Q_d = \left(\int_0^{d+1} \hat{q}(\delta)\right) \cdot \frac{1}{\mathcal{N}} \qquad \text{where } \mathcal{N} = \int_0^{d_{max}+1} \hat{q}(\delta) \tag{6}$$

To show that these can be computed efficiently, it suffices to show that any integral of $\hat{q}(\delta)$ can be efficiently evaluated. This follows from the fact that $\hat{q}$ is piecewise continuous with $\mathcal{O}(\log^4)$ pieces (Lemma 30), each of which has a closed form integral (since $f(d)$ defined in Equation 5 has an integral).

▶ **Lemma 31.** *Given the function $\hat{q}_d$ defined in Lemma 30, it is possible to approximate the integral $\int_{d_1}^{d_2+1} \hat{q}(\delta)$ to a multiplicative factor of $\left(1 \pm \frac{1}{n^2}\right)$, in $\operatorname{poly}(\log n)$ time for any valid $d_1, d_2 \in \mathbb{Z}$ (the bounds must be such that $d_i \geq 0$ and $U + D - 2d_i - k + 1 \geq 0$).*

Proof. We will compute the integral by splitting it up into $\mathcal{O}(\log^4 n)$ continuous pieces and then approximating the integral over each piece. The pieces corresponding to values of δ where $\lfloor \delta \rfloor \in \mathcal{R}$ are explicitly computed as $p_d = \int_d^{d+1} \hat{q}(\delta)$. We can do this using Lemma 63 from Section F.3.

Otherwise, if $\lfloor \delta \rfloor \notin \mathcal{R}$, we consider a range of values $[d_{min}, d_{max}] \subseteq [d_1, d_2 + 1]$, such that for any $d \in [d_{min}, d_{max}]$, the value of $\lfloor r(\lfloor d \rfloor) \rfloor$ is a constant E. Recall that we can also compute a $\left(1 \pm 1/n^3\right)$ multiplicative approximation to $\ln \mathcal{K} = \ln 2 \cdot (U + D) - \ln S_{total}$, by using the strategy in Lemma 63 (Section F.3). Finally, we can compute $F = \int_{d_{min}}^{d_{max}} f(\delta)$. Now, the logarithm of the integral is be written as:

$$\ln\left(\int_{d_{min}}^{d_{max}} \hat{q}(\delta)\right) = \ln\left(\mathcal{K} \cdot E \cdot F \cdot\right) = \ln 2 \cdot (U + D) - \ln S_{total} + \ln E + \ln F$$

If this value is smaller than $-3 \ln n$, we can safely ignore it since it contrbuted less than $1/n^3$ to the probability mass. On the flipside, the logarithm is guaranteed to be bounded by $\mathcal{O}(1)$. The upper bound is a result of the fact that $\int \hat{q}(\delta) = \Theta(\sum p_d) = \mathcal{O}(1)$, where both the sum and the integral are taken over the entire *valid* range of d. This means that we can exponentiate to obtain the true value of the piecewise integral up to a multiplicative approximation of $(1 \pm 1/n^3)$. Adding all the $\mathcal{O}(\log^4 n)$ pieces together produces the desired value of the integral.

Fix issues in this proof

◻

Now, we are finally ready to use Lemma 2 to sample d from the distribution $\{p_d\}$, using the efficient sampling procedure for $\{q_d\}$. The only other requirement is the ability to approximate the p_d values, which follows from Lemma 63.

▶ **Theorem 32.** *Given a sub-interval $[x_i, x_{i+1}]$ of a random Dyck path of length $2n$, such that the only sampled heights in the interval are $y_i = \textsc{Height}(x_i)$ and $y_{i+1} = \textsc{Height}(x_{i+1})$, and a mandatory boundary constraint at k, there exists an algorithm that samples a point x within the interval such that $\textsc{Height}(x) = y_i - k + 1$, using $\mathrm{poly}(\log n)$ time, random bits, and additional space.*

4.4.5 Finding the Correct Interval: `First-Return` Query

As before, consider all positions that have been queried already $\langle x_1, x_2, \cdots, x_m \rangle$ (in increasing order) along with their corresponding heights $\langle y_1, y_2, \cdots, y_m \rangle$. We wish to find the first return to height y_i after x_i (where $y_i = \textsc{Height}(x_i)$). Our strategy begins by using Invariant 26 to find the interval $\mathcal{I} = [x_k, x_{k+1}]$ containing $\textsc{First-Return}(x_i)$.

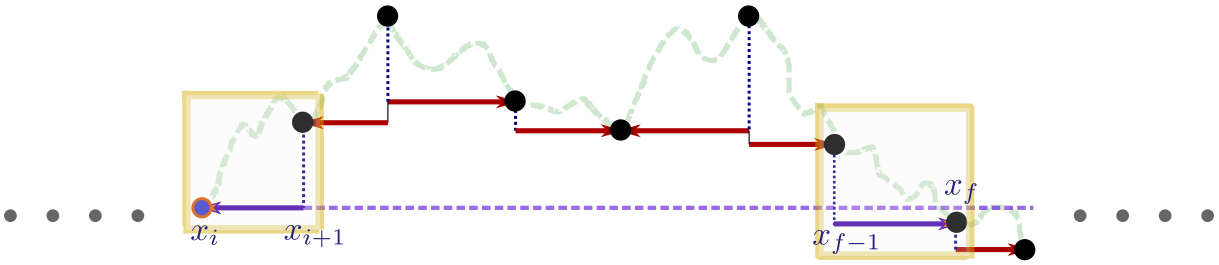

Figure 12 There are only two possible intervals (yellow boxes) that could contain $\textsc{First-Return}(x_i)$; either the interval adjacent to x_i, or the interval $[x_{f-1}, x_f]$, where f is the smallest index such that $y_f < x_i$.

▶ **Lemma 33.** *For any position x_i, assuming that Invariant 26 holds, the interval $(x_{k-1}, x_k]$ containing $\textsc{First-Return}(x_i)$ is obtained by setting k to be either the smallest index $f > i$ such that $y_f \le y_i$ or setting $k - 1 = i$.*

Proof. We assume the contrary i.e. there exists some $k \neq f$ and $k \neq i + 1$ such that the correct interval is $(x_{k-1}, x_k]$. Since $y_f < y_i$, the position of first return to y_i happens in the range $(x_i, x_f]$. So, the only possibility is $i + 1 < k \le f - 1$. By the definition of y_f, we know that both y_k and y_{k-1} are strictly larger than y_i. Invariant 26 implies that the boundary for this interval $(y_{k-1}, y_k]$ is at $\min(y_{k-1}, y_k) > y_i$. So, it is not possible for the first return to be in this interval. □

The good news is that there are only two intervals that we need to worry about, one of which is just the adjacent one $[x_i, x_{i+1}]$. The problem of finding the other interval that may contain the first return boils down to finding the smallest index $f > i$ such that $y_f \le y_i$. To this end, we define $M_{[a,b]}$ as the minimum sampled height in the range of positions $[a, b]$.

One solution is to maintain a range tree $\mathbf{R}$ [DBVKOS00] over the range $[2n]$. Assuming that $2n = 2^l$, we can view $\mathbf{R}$ as a complete binary tree with depth r. Every non-leaf node is denoted by $\mathbf{R}_{[a,b]}$, and corresponds to a range $[a, b] \subseteq [2n]$ that is the union of the ranges of its children. Each $\mathbf{R}_{[a,b]}$ stores the value $M_{[a,b]}$ which is is the minimum sampled height in the range of positions $[a, b]$, or ∞ if none of the heights have been revealed. The leaf nodes are denoted as $\{\mathbf{R}_i\}_{i \in [2n]}$, and correspond to the singleton range corresponding to position $i \in [2n]$. Note that a node at depth l' will correspond to a range of size $2^{l-l'}$, with the root being associated with the entire range $[2n]$.

We say that the range $[a, b]$ is *canonical* if it corresponds to a range of some $\mathbf{R}_{[a,b]}$ in $\mathbf{R}$. By the property of range trees, any arbitrary range can be decomposed into a disjoint union of $O(\log n)$ canonical ranges. We implement $\mathbf{R}$ to support the following operations:

- $\textsc{Update}(x, y)$: Update the height of the position x to y.
  This update affects all ranges $[a_i, b_i]$ containing x. So, for each $[a_i, b_i]$ we set $M_{[a_i,b_i]} = \min\left(M_{[a_i,b_i]}, y\right)$.

- QUERY(a, b): Return the minimum boundary height in the range $[a, b]$.
  We decompose $[a, b]$ into $\mathcal{O}(\log n)$ *canonical* ranges $\langle r_1, r_2, \cdots \rangle$, and return the minimum of all the M_{r_i} values as $M_{[a,b]}$ (since $[a, b]$ is the union of all r_i).

Now, we can binary search for f by guessing a value f' and checking if QUERY$(x_i, x_{f'}) \leq y_i$. Overall, this requires $\mathcal{O}(\log n)$ calls to QUERY, each of which makes $\mathcal{O}(\log n)$ probes to the range tree. To avoid an initialization overhead, we only create the node $\mathbf{R}_{[a,b]}$ during the first UPDATE affecting a position $x \in [a, b]$. Since a call to UPDATE can create at most $\mathcal{O}(\log n)$ new nodes in $\mathbf{R}$, the additional space required for each HEIGHT or FIRST-RETURN query is still bounded.

▶ **Theorem 34.** *There is an algorithm using $\mathcal{O}(poly(\log n))$ resources per query that provides sample access to a Dyck path of length $2n$ by answering queries of the form* FIRST-RETURN(x_i) *with the correctly sampled position y; where $y > x_i$ is the position where the Dyck path first returns to* HEIGHT(x_i) *after position x_i.*

Proof. In order to make the presentation simpler, we ensure that the next determined position after x_i is $x_i + 1$ (i.e. $x_{i+1} = x_i + 1$). This can be done by invoking HEIGHT$(x_i + 1)$, if it has not been sampled already. If HEIGHT$(x_i + 1) = y_{i+1} < y_i$, we can terminate because FIRST-RETURN(x_i) is not defined. Otherwise, we notice that in this setting, the first return cannot lie in the adjacent interval $[x_i, x_{i+1}] = [x_i, x_i + 1]$.

Hence, we proceed to finding the smallest value f such that $y_f < y_i$, by using the range tree data structure described above. Since HEIGHT$(x_{f-1}) \geq y_i >$ HEIGHT(x_f) by definition, the interval $(x_{f-1}, x_f]$ must contain a position at height y_i. We sample a point in the middle of this interval and fix the boundary invariant by sampling another point, essentially breaking it up into $\mathcal{O}(1)$ sub-intervals each at most half the size of the original. Based on the new samples, we again find the (newly created) sub-interval containing the first return in $\mathcal{O}(1)$ time. We repeat up to $\mathcal{O}(\log n)$ times, performing binary search to find the position of the first return. ◻

4.4.6 Maintaining `Height` Queries under Invariant 26

Finally, we show that the boundary constraints introduced in order to maintain Invariant 26 do not interfere with the implementation of HEIGHT queries. As before, we consider the currently revealed heights $\langle y_1, y_2, \cdots, y_m \rangle$, along with the corresponding positions $\langle x_1, x_2, \cdots, x_m \rangle$ (in increasing order). Say that we are now presented with a a query HEIGHT(x), where $x_i < x < x_{i+1}$. As in Section 4.3, we swap x_i and x_{i+1} if necessary in order to ensure that $y_i < y_{i+1}$. Due to Invariant 26, we know that the lowest achievable height in the interval $[x_i, x_{i+1}]$ is y_i, i.e. the boundary constraint for the left half becomes $k = 1$ instead of $k = y_i + 1$, since the constrained boundary is at height y_i. Similarly, the boundary constraint for the right half becomes $k' = 2U - 2D + 1$. The rest of the algorithm can proceed as described in Section 4.3. Of course, in this scenario, the boundary is never far away, and therefore we should always use the strategy in Section 4.3.2.

5 Random Coloring of a Graph

A *valid* q-coloring of a graph $G = (V, E)$ is a vector of colors $\mathbf{X} \in [q]^V$, such that for all $(u, v) \in E$, $\mathbf{X}_u \neq \mathbf{X}_v$. We present a sublinear time algorithm to provide local access to a uniformly random valid q-coloring of an input graph. Specifically, we implement COLOR(v), which returns the color $\mathbf{X}_v$ of node v, where $\mathbf{X}$ is a uniformly random valid coloring. The implementation can access the input graph G through a sub-linear number of *neighborhood queries*. A neighborhood query of the form ALL-NEIGHBORS(v) returns a list of neighbors of v. The implementation can also access a tape of public random bits $\mathbf{R}$.

Moreover, multiple independent instances of COLOR that are given access to the same public tape of random bits $\mathbf{R}$, should output color values consistent with a single $\mathbf{X}$, regardless of the order and content of the queries received. Unlike our previous results, the choice of $\mathbf{X}$ only depends on $\mathbf{R}$, and the COLOR

implementations do not need to use any additional memory to maintain consistency. For a formal description of this model, see Definition 3. We consider graphs with max degree Δ, and $q = \Theta(\Delta)$, since this is the regime where this problem is feasible [FV07].

Comparison to LCAs

In the sequential setting, [FV07] used the technique of path coupling to show that for $q > 2\Delta$, one can sample an uniformly random coloring by using a simple Markov chain. The Markov chain proceeds in T steps. The state of the chain at time t is given by $\mathbf{X}^t \in [q]^{|V|}$. Specifically, the color of vertex v at step t is $\mathbf{X}_v^t$. In each step of the Markov process, a vertex and a color are sampled uniformly at random i.e. a pair $(v, c) \sim_{\mathcal{U}} V \times [q]$. Subsequently, if the recoloring of vertex v with color c does not result in a conflict with v's neighbors, i.e. $c \notin \{\mathbf{X}_u^t : u \in \Gamma(v)\}$, then the vertex is recolored i.e. $\mathbf{X}_v^{t+1} \leftarrow c$. After running this chain for $T = \mathcal{O}(n \log(n/\epsilon))$ steps, the Markov chain is mixed, implying that the distribution of resulting colors is ϵ close to the uniform distribution in L_1 distance.

Use $q = 2\alpha\Delta$ everywhere.

5.1 Modified Glauber Dynamics based on a Distributed Algorithm

Now we define a modified Markov chain as a special case of the *Local Glauber Dynamics* presented in [FG18]. The modified Markov chain proceeds in epochs. We denote the initial coloring of the graph by the vector $\mathbf{X}^0$ and the state of the coloring after the k^{th} epoch by $\mathbf{X}^k$. In the k^{th} epoch, every node attempts to recolor itself simultaneously in a conservative manner, as described below:

- Sample $|V|$ colors $\langle c_1, c_2, \cdots, c_n \rangle$ from $[q]$, where c_v is the color proposed by vertex v.
- For each vertex v, we set $\mathbf{X}_v^k$ to c_v, if and only if for all neighbors w of v, $\mathbf{X}_w^{k-1} \neq c_v$ and $c_w \neq c_v$. Specifically, a vertex v is recolored if and only if its proposed color c_v does not conflict with any of it's neighbors current colors (at the end of the previous epoch), or their current proposals.

This procedure is a special case of the *Local Glauber Dynamics*, which was presented in [FG18] as a distributed algorithm for sampling a random coloring.[6] In the distributed setting, our epochs correspond to synchronous rounds, where many vertices recolor themselves simultaneously.

In order to bound the mixing time of this Markov chain, [FG18] uses the standard technique of *path coupling*, introduced in [BD97]. The argument begins by considering two initial states of the Markov Chains, say two colorings $\mathbf{X}^0$ and $\mathbf{Y}^0$, that differ at only one vertex. Formally, we can define the distance between two colorings $d(\mathbf{X}, \mathbf{Y})$ as the number of vertices v such that $\mathbf{X}_v \neq \mathbf{Y}_v$, which results in the condition $d(\mathbf{X}^0, \mathbf{Y}^0) = 1$. A *coupling* is a joint evolution rule for a pair of states $(\mathbf{X}^0, \mathbf{Y}^0) \to (\mathbf{X}^1, \mathbf{Y}^1)$, such that both of the individual evolutions $(\mathbf{X}^0 \to \mathbf{X}^1)$ and $(\mathbf{Y}^0 \to \mathbf{Y}^1)$ have the same transition probabilities as the original Markov Chain. We can directly use the result from the coupling defined in [FG18].

▶ **Lemma 35.** *If $q = 2\alpha\Delta$, then there exists a coupling $(\mathbf{X}^0, \mathbf{Y}^0) \to (\mathbf{X}^1, \mathbf{Y}^1)$, such that if $d(\mathbf{X}^0, \mathbf{Y}^0) = 1$, then $\mathbb{E}[d(\mathbf{X}^1, \mathbf{Y}^1)] \leq 1 - \left(1 - \frac{1}{2\alpha}\right) e^{-3/\alpha} + \frac{1/2\alpha}{1-1/\alpha}$*

▶ **Corollary 36.** *If $q \geq 9\Delta$ and $d(\mathbf{X}^0, \mathbf{Y}^0) = 1$, then $\mathbb{E}[d(\mathbf{X}^1, \mathbf{Y}^1)] < \frac{1}{e^{1/3}}$*

The *path coupling* lemma from [BD97] uses a coupling on adjacent states to bound the mixing time.

▶ **Lemma 37.** ***(Simplified Path Coupling from [BD97])*** *If there exists a coupling $(\mathbf{X}^0, \mathbf{Y}^0) \to (\mathbf{X}^1, \mathbf{Y}^1)$ defined for states where $d(\mathbf{X}^0, \mathbf{Y}^0) = 1$, such that $\mathbb{E}[d(\mathbf{X}^1, \mathbf{Y}^1) \mid \mathbf{X}^0, \mathbf{Y}^0] \leq \beta$ (for $\beta < 1$), then,*

[6] Note that [FG18] also uses a marking probability γ, which indicates the likelihood of any vertex participating in a given round. For our purposes, it suffices to set $\gamma = 1$.

the mixing time $\tau_{mix}(\epsilon) = \mathcal{O}(\ln(n\epsilon^{-1})/\ln\beta^{-1})$.

▶ **Corollary 38.** *If* $q \geq 9\Delta$*, then the chain is mixed after* $\tau_{mix}(\epsilon) = 3\left(\ln n + \ln(\frac{1}{\epsilon})\right)$ *epochs.*

Distributed Algo + Parnas-Ron [PR07] and how we do better.

Similar to LCA for MIS

5.2 Local Coloring Algorithm

Given query access to the adjacency matrix of a graph G with maximum degree Δ and a vertex v, the algorithm has to output the color assigned to v after running $t = \mathcal{O}(\ln n)$ epochs of *Modified Glauber Dynamics*. We want to be able to answer such queries in sublinear time, without simulating the entire Markov Chain. We will define the number of colors as $q = 2\alpha\Delta$ where $\alpha > 1$.

The proposals at each epoch are a vector of color samples $\mathbf{C}^t \sim_{\mathcal{U}} [q]^n$, where $\mathbf{C}^t_v$ is the color proposed by v in the t^{th} epoch. Note that these values are fully independent and as such any $\mathbf{C}^t_v$ can be sampled trivially. We also use $\mathbf{X}^t$ to denote the final vector of vertex colors at the end of the t^{th} epoch. Finally, we define indicator variables $\boldsymbol{\chi}^t_v$ to indicate whether the color $\mathbf{C}^t_v$ proposed by vertex v was accepted at the t^{th} epoch: $\boldsymbol{\chi}^t_v = 1$ if and only if for all neighbors $w \in \Gamma(v)$, we satisfy the condition $\mathbf{C}^t_v \neq \mathbf{X}^{t-1}_w$ and $\mathbf{C}^t_v \neq \mathbf{C}^t_w$ (i.e. the proposed color does not conflict with any neighboring proposal or any neighbor's color from the preceding epoch). So, the color of a vertex v after the t^{th} epoch $\mathbf{X}^t_v$ is set to be $\mathbf{C}^i_v$ where $i \leq t$ is the largest index such that $\boldsymbol{\chi}^i_v = 1$. While the proposals $\mathbf{C}^t_v$ are easy to sample, it is much less clear how we can determine the $\boldsymbol{\chi}^t_v$ values. Note that we can compute $\mathbf{X}^t_v$ quite easily if we know the values $\boldsymbol{\chi}^i_v$ for all $i \leq t$. So, we focus our attention on the query $\textsc{Accepted}(v, t)$ that returns $\boldsymbol{\chi}^t_v$.

5.2.1 Local Access to an Initial Valid Coloring

One caveat that we have not addressed is how we should initialize the Markov Chain. The starting state can be any valid coloring of G. One way to

How to initialize? [CFG+19]

5.2.2 Naive Coloring Implementations

Our general strategy to determine $\boldsymbol{\chi}^t_v$ will be to check for all neighbors w of v, whether w causes a conflict with v's proposed color in the t^{th} epoch. One naive way to achieve this, is to iterate backwards from epoch t, querying to find out whether w's proposal was accepted, until the most recent accepted proposal (latest epoch $t' < t$ such that $\boldsymbol{\chi}^{t'}_w = 1$) is found. At this point, if $\mathbf{C}^{t'}_w = \mathbf{C}^t_v$, then the current color of w conflicts with v's proposal. Otherwise there is no conflict, and we can proceed to the next neighbor. However, this process potentially makes Δ recursive calls to a sub-problem that is only slightly smaller i.e. $T(t) \leq \Delta \cdot T(t-1)$. This leads to a running time upper bound of Δ^t which is superlinear for the desired number of epochs $t = \Omega(\log n)$ (the mixing time).

We can prune the number of recursive calls by only processing the neighbors w which actually proposed the color $\mathbf{C}^t_v$ during *some* epoch. In this case, the expected number of neighbors that have to be probed recursively is less than $t\Delta/q$ (since the total number of neighbor proposals over t epochs is at most $t\Delta$, and there are q possible colors). So, the overall runtime is upper bounded by $(t\Delta/q)^t$. For this algorithm, if we allow $q > t\Delta = \Omega(\Delta \log n)$ colors, the runtime becomes sublinear. So, we can use this simple algorithm only when q is sufficiently large. However, we want a sub-linear time algorithm for $q = \mathcal{O}(\Delta)$.

5.2.3 A Sublinear Time Algorithm for $q = \mathcal{O}(\Delta)$

The expected number of neighbors that need to be checked recursively can always be $t\Delta/q$ in the worst case. The crucial observation is that even though these recursive calls seem unavoidable, we can aim to reduce the size of the recursive sub-problem and thus bound the number of levels of recursion. Because of the more complex structure of this epoch jumping process, the main challenge is to analyze the runtime.

High Level algorithm.

Algorithm 13 Checking if proposal is accepted

```
1:  procedure ACCEPTED(v, t)
2:      c ← C_v^t                                          ▷ Sample the color.
3:      for w ← Γ(v)
4:          if C_w^t = c                                   ▷ Check for conflict with neighbor's current proposal
5:              return 0
6:          for t' ← [t, t − 1, t − 2, ··· , 1]
7:              if C_w^{t'} = c and ACCEPTED(w, t')        ▷ Potential conflict with neighbor's color
8:                  overwritten ← false                    ▷ Check if color c was overwritten by a future proposal
9:                  for t̃ ← [t' + 1, t' + 2, ··· , t − 1]
10:                     if ACCEPTED(w, t̃)
11:                         overwritten ← true
12:                         break
13:                 if not overwritten
14:                     return 0                           ▷ Conflict! This proposal is not accepted
15:                 break
16:     return 1                                           ▷ No conflicts! This proposal is accepted
```

Algorithm 13 shows our final procedure for sampling $\boldsymbol{\chi}_v^t$ where $c = \mathbf{C}_v^t$ is the color proposed by v in epoch t. We iterate through all neighbors w of v, checking for conflicts. The condition $c \neq \mathbf{C}_w^t$ is can easily be checked by sampling $\mathbf{C}_w^t$. If no conflict is seen, the next step is to check whether $c \neq \mathbf{X}_w^{t-1}$.

To achieve this, we iterate through all the epochs in reverse order (line 6) to check whether the color c was ever proposed for vertex w. If not, we can ignore w, and otherwise let's say that the most recent proposal for c was at epoch t' i.e. $\mathbf{C}_w^{t'} = c$. Now, we directly "jump" to the $(t')^{th}$ epoch and recursively check if this proposal was accepted (line 7). If the proposal $\mathbf{C}_w^{t'}$ was not accepted, we keep iterating back in time until we find the next most recent epoch when c was proposed by w, or until we run out of epochs. When we find the most recent epoch t' in which c was accepted i.e. $\boldsymbol{\chi}_w^{t'} = 1$, we successively consider epochs $t'+1, t'+2, t'+3, \cdots, t-1$ to see whether the color c was overwritten (line 9) by an accepted proposal in a future epoch. This is done by recursively invoking $\textsc{Accepted}(w, t'+i)$ in order to compute $\boldsymbol{\chi}_w^{t'+i}$ (line 10). If at any of these subsequent iterations, we see that a different proposal was accepted (thus overwriting the color c), then neighbor w does not cause a conflict, and we can move on to the next neighbor. Otherwise, we have seen that $\boldsymbol{\chi}_w^{t'} = 1$ (color c was accepted) and every subsequent proposal until the current epoch t was rejected, implying that color c *survived* as the color of neighbor w, i.e. $\mathbf{X}_w^{t-1} = c$. This leads to a conflict with v's current proposal for color c (line 14) and hence $\boldsymbol{\chi}_v^t = 0$. If we exhaust all the neighbors and don't find any conflicts (line 16) then $\boldsymbol{\chi}_v^t = 1$.

Picture?

Now we analyze the runtime of $\textsc{Accepted}$ by constructing and solving a recurrence relation. We will use the following lemma to evaluate the expectation of products of relevant random variables.

▶ **Lemma 39.** *The probability that any given proposal is rejected* $\mathbb{P}[\boldsymbol{\chi}_v^t = 0]$ *is at most* $1/\alpha$*. Moreover, this upper bound holds even if we condition on all the values in* $\mathbf{C}$ *except* $\mathbf{C}_v^t$*.*

Proof. A rejection can occur due to a conflict with at most 2Δ possible values in $\{\mathbf{C}_w^t, X_w^{t-1} | w \in \Gamma(v)\}$. Since there are $2\alpha\Delta$ colors, the rejection probability is at most $1/\alpha$. ◻

▶ **Definition 40.** *We define* T_t *to be a random variable indicating the number of recursive calls performed during the execution of* $\textsc{Accepted}(v, t)$ *while computing a single* $\boldsymbol{\chi}_v^t$*.*

▶ **Definition 41.** *We define* $R_{t'}^t$ *to be a random variable indicating the number of calls to* `Accepted` *that are required, to check whether a color* c *assigned at epoch* t' *was overwritten at some epoch before* t*.*

Using $\mathcal{B}(p)$ to denote the Bernoulli random variable with bias p, we obtain an expression for $R_{t'}^t$.

$$R_{t'}^t = \left[T_{t'+1} + \mathcal{B}\left(\frac{1}{\alpha}\right) \cdot T_{t'+2} + \mathcal{B}\left(\frac{1}{\alpha^2}\right) \cdot T_{t'+3} + \cdots + \mathcal{B}\left(\frac{1}{\alpha^{t-t'-2}}\right) \cdot T_{t-1} \right] \tag{7}$$

The aforementioned Equation 7 indicates that the call to $\textsc{Accepted}(v, t'+1)$ (line 10) is always invoked (resulting in $T_{t'+1}$ invocations of `Accepted`). However, the next call to $\textsc{Accepted}(v, t'+2)$ is invoked only if the previous one was not accepted, which occurs with probability at most $1/\alpha$ (Lemma 39). This gives us the $\mathcal{B}(1/\alpha) \cdot T_{t'+2}$ term in the expression. In general, $\textsc{Accepted}(v, t'+i)$ is only invoked if the preceding $i-1$ calls to `Accepted` all returned 0. This event happens with probability at most $1/\alpha^{i-1}$.

▶ **Lemma 42.** *Given graph* G *and* $q = 2\alpha\Delta$ *colors, for* $\alpha > 4.5$*, the expected number of recursive calls to the procedure* `Accepted` *while computing a single* $\boldsymbol{\chi}_v^t = \textsc{Accepted}(v, t)$ *is* $\mathbb{E}[T_t] = \mathcal{O}\left(e^{1.02t/\alpha}\right)$*.*

Proof. We start with the recurrence for the expected number of probes to $\{\boldsymbol{\chi}^{t'}\}_{t' \in [t]}$ (equivalently calls to `Accepted`) used by the algorithm. When checking a single neighbor w, the algorithm iterates through all the epochs t' such that $\mathbf{C}_w^{t'} = c$ (in reality, only the last occurence matters, but we are looking for an upper bound). If such a t' is found (this happens with probability $1/q$ independently for each trial), there is one recursive call to $T_{t'}$. Regardless of what happens, let's say the algorithm queries $T_{t'+1}, T_{t'+2}, \cdots, T_{t-1}$ until an `Accepted` proposal is found. Adding an extra $T_{t'}$ term to Equation 7 and summing up over all neighbors and epochs we get the following:

$$T_t \leq \Delta \cdot \sum_{t'=1}^{t} \mathbb{P}[C_w^{t'} = c] \cdot \left[T_{t'} + T_{t'+1} + \mathcal{B}\left(\frac{1}{\alpha}\right) \cdot T_{t'+2} + \mathcal{B}\left(\frac{1}{\alpha^2}\right) \cdot T_{t'+3} + \cdots \right. \tag{8}$$

$$\left. \cdots + \mathcal{B}\left(\frac{1}{\alpha^{t-t'-2}}\right) \cdot T_{t-1} \right] \tag{9}$$

$$\leq \Delta \cdot \mathcal{B}\left(\frac{1}{q}\right) \left[\sum_{t'=1}^{t-1} T_{t'} + \sum_{t'=1}^{t-1} T_{t'} \cdot \left(1 + \mathcal{B}\left(\frac{1}{\alpha}\right) + \mathcal{B}\left(\frac{1}{\alpha^2}\right) + \cdots \right) \right] \tag{10}$$

In the second step, we just group all the terms from the same epoch together. Using Lemma 39 and the fact that $\mathbb{P}[C_w^{t'} = c]$ is independent of all other events, we can write a recurrence for the expected number of probes.

$$\mathbb{E}[T_t] \leq \Delta \cdot \frac{1}{2\alpha\Delta} \left[\sum_{t'=1}^{t-1} T_{t'} + \sum_{t'=1}^{t-1} T_{t'} \cdot \left(1 + \frac{1}{\alpha} + \frac{1}{\alpha^2} + \cdots \right) \right] \leq \frac{1}{2\alpha} \cdot \sum_{t'=1}^{t-1} T_{t'} \cdot \left[1 + \frac{\alpha}{\alpha - 1} \right] \tag{11}$$

Now, we make the assumption that $\mathbb{E}[T_{t'}] \leq e^{kt/\alpha}$, and show that this satisfies the expectation recurrence for the desired value of k. First, we sum the geometric series:

$$\sum_{t'=1}^{t-1} \mathbb{E}[T_{t'}] = \sum_{t'=1}^{t-1} e^{kt'/\alpha} < \frac{e^{kt/\alpha} - 1}{e^{k/\alpha} - 1} < \frac{e^{kt/\alpha}}{e^{k/\alpha} - 1}$$

The expectation recurrence to be satisfied then becomes:

$$\mathbb{E}[T_t] \leq \frac{1}{2\alpha} \cdot \frac{e^{kt/\alpha}}{e^{k/\alpha}-1} \cdot \left[1 + \frac{\alpha}{\alpha - 1}\right] = e^{kt/\alpha} \cdot \frac{2\alpha - 1}{2\alpha(\alpha-1)(e^{k/\alpha}-1)} = e^{kt/\alpha} \cdot f(\alpha, k)$$

We notice that for $k = 1.02$ and $\alpha > 4.5$, $f(\alpha) < 1$. This can easily be verified by checking that $f(\alpha, 1.02)$ decreases monotonically with α in the range $\alpha > 4.5$. Thus, our recurrence is satisfied for $k = 1.02$, and therefore the expected number of calls is $\mathcal{O}(e^{1.02t/\alpha})$. □

▶ **Theorem 43.** *Given adjacency list query access to a graph with n nodes, maximum degree Δ, and $q = 2\alpha\Delta \geq 9\Delta$ colors, we can sample the color of any given node from a distribution of color assignments that is $\frac{1}{n}$-close (in L_1 distance) to the uniform distribution over all colorings of the graph, in a consistent manner, using only $\mathcal{O}(n^{6.12/\alpha}\Delta \log n)$ time and random bits.*

Proof. Since $q \geq 9\Delta$, we can use Corollary 38 to obtain $\tau_{mix}(1/n) \leq 6 \ln n$. Since $\alpha > 4.5$, we can invoke Lemma 42 to conclude that the number of calls to `ACCEPTED` is $\mathcal{O}(n^{6.12/\alpha})$. Finally, we note that each call to `ACCEPTED`(v, t) potentially samples $\mathcal{O}(t\Delta)$ color proposals, while iterating through all the neighbors of v in all t epochs. Since $t \leq 6 \ln n$, this implies that the algorithm uses $\mathcal{O}(n^{6.12/\alpha}\Delta \log n)$ time and random bits, which is sublinear for $\alpha > 6.12$. □

One final observation is that the value returned (sampled value of $\boldsymbol{\chi}_v^t$) by `ACCEPTED` only depends on the sampled values of C_v^t, which in turn depends only on the random bits used by the algorithm. This implies that unlike the prior implementations, this one does not need to store any state in memory. Importantly, multiple independent instances of the algorithm that have access to the same random bits, will invariably answer queries in a manner consistent with each other i.e. they will sample exactly the same coloring.

6 Open Problems

There are many interesting directions to pursue in this area. We give a few examples.

- Provide a local access implementation of degree queries for undirected random graphs, even for $G(n, p)$.
- Provide a faster local access implementation for sampling the color of a specified vertex in a random q-coloring of a bounded degree graph G.
- Given a graph with maximum degree Δ and $q < 9\Delta$, provide a sublinear time local access implementation for sampling the color of a specified vertex in a random q-coloring of G. We remark that this problem in particular should be feasible, by simulating a faster mixing Markov chain. The important question is whether you can you get down to $q = 2\Delta$?
- Given a graph G and starting vertex v, provide a local access implementation for sampling the location of a random walk starting at v after t steps.

References

Abb16 Emmanuel Abbe. Community detection and the stochastic block model. 2016.

ABH16 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.

AK17 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.

ARVX12 Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 1132–1139. Society for Industrial and Applied Mathematics, 2012.

AS15 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.

BB05 Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.

BB06 Danielle Smith Bassett and ED Bullmore. Small-world brain networks. *The neuroscientist*, 12(6):512–523, 2006.

BD97 Russ Bubley and Martin Dyer. Path coupling: A technique for proving rapid mixing in markov chains. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*, pages 223–231. IEEE, 1997.

BK10 Michael Brautbar and Michael J Kearns. Local algorithms for finding interesting individuals in large networks. 2010.

CAT16 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.

CFG+19 Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The complexity of (δ+ 1) coloring in congested clique, massively parallel computation, and centralized local computation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, pages 471–480. ACM, 2019.

CRV15 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.

CSC+07 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.

CY06 Jingchun Chen and Bo Yuan. Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.

DBVKOS00 Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 105–109. Springer, 2000.

DMW03 Peter Sheridan Dodds, Roby Muhamad, and Duncan J Watts. An experimental study of search in global social networks. *science*, 301(5634):827–829, 2003.

EK10 David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

ELMR17 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.

ER60 Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

FG18 Manuela Fischer and Mohsen Ghaffari. A simple parallel and distributed sampling technique: Local glauber dynamics. In *32nd International Symposium on Distributed Computing (DISC 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

For10 Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3):75–174, 2010.

FPSV17 Uriel Feige, Boaz Patt-Shamir, and Shai Vardi. On the probe complexity of local computation algorithms. *arXiv preprint arXiv:1703.07734*, 2017.

FV07 Alan Frieze and Eric Vigoda. A survey on the use of markov chains to randomly sample colourings. *Oxford Lecture Series in Mathematics and its Applications*, 34:53, 2007.

GGM86 Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

GGN03 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.

GGN10 Oded Goldreich, Shafi Goldwasser, and Asaf Nussboim. On the implementation of huge random objects. *SIAM Journal on Computing*, 39(7):2761–2822, 2010.

GGR98 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.

GR97 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.

HLL83 Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.

JTZ04 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.

Kle00 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.

Knu97 Donald E Knuth. The art of computer programming, 3rd edn. seminumerical algorithms, vol. 2, 1997.

LR88 Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM J. Comput.*, 17(2):373–386, 1988.

LSY03 Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *IEEE Internet computing*, 7(1):76–80, 2003.

MH11 Joel Miller and Aric Hagberg. Efficient generation of networks with given expected degrees. *Algorithms and models for the web graph*, pages 115–126, 2011.

MKI+03 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.

MN04 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.

MNS15 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.

MPN+99 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.

MRVX12 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.

New00 Mark EJ Newman. Models of the small world. *Journal of Statistical Physics*, 101(3):819–841, 2000.

NLKB11 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.

NN07 Moni Naor and Asaf Nussboim. Implementing huge sparse random graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 596–608. Springer, 2007.

NR96 Moni Naor and Omer Reingold. On the construction of pseudo-random permutations: Luby-rackoff revisited. *IACR Cryptology ePrint Archive*, 1996:11, 1996.

NWS02 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.

PR07 Michal Parnas and Dana Ron. Approximating the minimum vertex cover in sublinear time and a connection to distributed algorithms. *Theoretical Computer Science*, 381(1-3):183–196, 2007.

Reu14 Shlomi Reuveni. Catalan's trapezoids. *Probability in the Engineering and Informational Sciences*, 28(03):353–361, 2014.

RTVX11 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. *arXiv preprint arXiv:1104.1377*, 2011.

SC11 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.

SM00 Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(8):888–905, 2000.

Spe14 Joel Spencer. *Asymptopia*, volume 71. American Mathematical Soc., 2014.

SPT+01 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.

Sta15 Richard P Stanley. *Catalan numbers*. Cambridge University Press, 2015.

TM67 Jeffrey Travers and Stanley Milgram. The small world problem. *Phychology Today*, 1:61–67, 1967.

WS98 Duncan J Watts and Steven H Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, 1998.

A Further Analysis and Extensions of Algorithm 1: Sampling `Next-Neighbor` without Buckets

A.1 Performance Guarantee

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

▶ **Theorem 44.** *Each execution of Algorithm 1 (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 $\prod_{u\in[a,b]}(1-p_{vu})$;*
- *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_{vu}'s, but also the unused coins C_{vu}'s. We characterize the two collections of Bernoulli variables $\{X_{vu}\}$ and $\{Y_{vu}\}$ that cover all random choices made by Algorithm 1 as follows.

- Each X_{vu} (same as X_{uv}) 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_{vu} becomes decided: either C_{vu} during a `Next-Neighbor`(v) call when $\mathbf{A}[v][u]=\phi$, or C_{vu} during a `Next-Neighbor`(u) call when $\mathbf{A}[u][v]=\phi$, whichever occurs first. This description of X_{vu} respects our invariant that, if the generation process is executed to completion, we will have $\mathbf{A}[v][u]=X_{vu}$ in all entries.
- Each Y_{vu} represents the result for the *second* coin-toss corresponding to cell $\mathbf{A}[v][u]$, which is the coin-toss C_{vu} obtained during a `Next-Neighbor`(v) call when X_{vu} is already decided. In other words, $\{Y_{vu}\}$'s are the coin-tosses that should have been skipped but still performed in Algorithm 1 (if they have indeed been generated). Unlike the previous case, Y_{vu} and Y_{uv} 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_{vu} in our proof. The success probabilities of these indicators are therefore given by $\mathbb{P}[X_{vu}=1]=\mathbb{P}[Y_{vu}=1]=p_{vu}$.

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 1 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.[7]

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 $X_{v,\mathbf{last}[v]+1} = \cdots = X_{vu} = 0$. Informally, we are asking "of all consecutive cells of 0's in a single row of $\{X_{vu}\}$-table, what is the largest number of cells of 1's in the corresponding cells of $\{Y_{vu}\}$-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 $\{X_{vu}\}$ and $\{Y_{vu}\}$, 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) encounters 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 1 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 44. □

[7] 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.

A.2 Supporting Vertex-Pair Queries

We extend our generator (Algorithm 1) 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_{uv} 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_{uv} to assign X_{uv}, deciding whether $\{u, v\}$ exists in the generated graph. If we maintained the full $\mathbf{A}$, 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 1 that represents $\mathbf{A}$ compactly via **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, then by Lemma 4, **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.[8] To this end, we define the set $Q = \{\{u, v\} : X_{uv}$ 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, $O(1)$ random N-bit word, and $O(1)$ additional space per query.

▶ **Lemma 45.** *The data structures* **last**, P_v*'s and* Q *together provide a succinct representation of* $\mathbf{A}$ *when* NEXT-NEIGHBOR *queries (modified Algorithm 1) and* VERTEX-PAIR *queries 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 1. In the implementation of NEXT-NEIGHBOR, an iteration is not failed when the chosen X_{vu} is still undecided: $\mathbf{A}[v][u]$ must still be ϕ. Since X_{vu} 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_{vu} 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 1 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 11).

▶ **Corollary 46.** *Algorithm 1 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 44.*

[8] 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.

B Omitted Details from Section 3: Undirected Random Graph Implementations

B.1 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 $\prod_{u=a}^{b}(1-p_{vu})$ and $\sum_{u=a}^{b} p_{vu}$ 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 3. By our assumption, the CDF of any $\mathsf{F}(v,a,b)$ can be efficiently computed from $\prod_{u=a}^{u'}(1-p_{vu})$, 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 47.** *If Algorithm 3 (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 3) 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 $\|\mathsf{F}(v,a,b) - \mathsf{F}'(v,a,b)\|_1 \le \epsilon = \frac{1}{\mathrm{poly}(n)}$, 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, $\|\mathsf{T}_s - \mathsf{T}'_s\|_1 \le \epsilon$ 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, $\sum_{s'} \mathsf{T}_s^{\min}(s') = 1 - \|\mathsf{T}_s - \mathsf{T}'_s\|_1 \ge 1 - \epsilon$. Define $q', q^{\min}, \mathsf{G}'(G_{\mathbf{A}}), \mathsf{G}^{\min}(G_{\mathbf{A}})$

analogously, and observe that $q^{\min}(\pi) \le \min\{q(\pi), q'(\pi)\}$ for every π, so $\mathsf{G}^{\min}(G_{\mathbf{A}}) \le \min\{\mathsf{G}(G_{\mathbf{A}}), \mathsf{G}'(G_{\mathbf{A}})\}$ 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 $\pi_i = \langle s_0^\pi, \dots, s_i^\pi \rangle$ of π. Observe that for $i \ge 1$,

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

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 $\sum_\pi q^{\min}(\pi) = \sum_\pi q^{\min}(\pi_S) \ge (1-\epsilon)^S \ge 1 - \epsilon S$. Hence, $\sum_{G_{\mathbf{A}}} \min\{\mathsf{G}(G_{\mathbf{A}}), \mathsf{G}'(G_{\mathbf{A}})\} \ge \sum_{G_{\mathbf{A}}} \mathsf{G}^{\min}(G_{\mathbf{A}}) \ge 1 - \epsilon S$, implying that $\|\mathsf{G} - \mathsf{G}'\|_1 \le \epsilon S$, as desired. In particular, by substituting $\epsilon = \frac{1}{\mathrm{poly}(n)}$ and $S = O(n^3)$, we have shown that Algorithm 3 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.

B.2 Bounding bucket sizes

▶ **Lemma 5.** *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 $\{X_{vu}\}_{u \in B_v^{(i)}}$. The expected number of neighbors in this bucket is $\mathbb{E}\left[|\Gamma^{(i)}(v)|\right] = \mathbb{E}\left[\sum_{u \in B_v^{(i)}} X_{vu}\right] < L + 1$. Via the Chernoff bound,

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

for any constant $c > 0$. ◻

▶ **Lemma 6.** *With high probability, for every v such that $|\mathbf{B}_v| = \Omega(\log n)$ (i.e., $\mathbb{E} = \Omega(\log n)$), at least a $1/3$-fraction of the buckets $\{B_v^{(i)}\}_{i \in [|\mathbf{B}_v|]}$ are non-empty.*

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

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

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 $\mathbb{E}1 - c$. By the Chernoff bound, the probability that less than $|B_v|/3$ buckets are non-empty is

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

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

C Next-Neighbor Implementation 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 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 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 C.1) then extend to VERTEX-PAIR queries (Section C.2. We then explain how this result may be generalized to support the Stochastic Block model with random community assignment in Section C.3.

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

Recall that NEXT-NEIGHBOR(v) is given by $\min\{u > \mathbf{last}[v] : X_{vu} = 1\}$ (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: 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 $X_{vu} = 0$ are decided, are explicitly excluded from T_v.

Algorithm 14 Alternate implementation

procedure NEXT-NEIGHBOR(v)
  $w \leftarrow \min K_v$, or $n+1$ if $K_v = \emptyset$
  $t \leftarrow$ COUNT(v)
  sample $F \sim$ ExactF(p, t)
  if $F \le t$
    $u \leftarrow$ PICK(v, F)
    $K_u \leftarrow K_u \cup \{v\}$
  else
    $u \leftarrow w$
    if $u \neq n+1$
      $K_v \leftarrow K_v \setminus \{u\}$
  UPDATE(v, u)
  $\mathbf{last}[v] \leftarrow u$
  return u

Unlike the approach of Algorithm 1 that simulates coin-flips even for decided X_{vu}'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 $f \le |T_v|$ (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.

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.)

- $\textsc{count}(v)$: 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.
- $\textsc{pick}(v, F)$: find the F^{th} minimum value in T_v (assuming $F \le |T_v|$).
  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 $\sum_{j \le i} t_{[a_j,b_j]} \ge F$ 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.
- $\textsc{update}(v, u)$: simulate coin-flips, assigning $X_{vu} \leftarrow 1$, 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 14, 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.

C.2 Data structure for `Vertex-Pair` queries in the Erdös-Rényi model

Recall that we define Q in Algorithm 1 as the set of pairs (u, v) where X_{uv} is assigned to 0 during a $\textsc{Vertex-Pair}$ query, allowing us to check for modifications of $\mathbf{A}$ not captured by $\mathbf{last}[v]$ and K_v. Here in Algorithm 14, 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:

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

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 4) 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 44 and Corollary 46).

▶ **Theorem 48.** *Consider the Erdös-Rényi $G(n, p)$ model. For $\textsc{Next-Neighbor}$ queries only, Algorithm 14 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 14 answers each query using* $O(\log^2 n)$ *amortized time,* $O(\log n)$ *additional space, and one* N*-bit random word.*

C.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 3.4.2. 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 $F \sim \mathsf{ExactF}$ 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_{uv}, 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 49.** *Consider the Stochastic Block model with randomly-assigned communities. For* NEXT-NEIGHBOR *queries only, Algorithm 14 is a generator that answers each query using* $O(r \operatorname{poly}(\log n))$ *time, random words, and additional space per query. For* NEXT-NEIGHBOR *and* VERTEX-PAIR *queries, Algorithm 14 answers each query using* $O(r \operatorname{poly}(\log n))$ *amortized time,* $O(r \operatorname{poly}(\log n))$ *random words, and* $O(r \operatorname{poly}(\log n))$ *additional space per query additional space, and one* N*-bit random word.*

D 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 [GGN10].

▶ **Lemma 50.** *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 51.** *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 $O(\operatorname{poly}(\log B))$ time and random words.*

Proof. For the base case where $B = 1$, we trivially have $\mathbf{S}_1^{\mathbf{C}} = \mathbf{C}_1$ and $\mathbf{S}_0^{\mathbf{C}} = \mathbf{C}_2$. 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 **??**.
  - 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 17.** *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 $O(r \cdot \operatorname{poly}(\log B))$ 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 $\mathbf{D} = \langle C_1, C_2 + \cdots + C_r \rangle$, then generate $\mathbf{S}_\ell^{\mathbf{D}} = \langle s_1, s_2 + \ldots + s_r \rangle$ 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 47. ◻

▶ **Theorem 52.** *Given B marbles of r different colors in $[r]$, such that there are C_i marbles of color i and a parameter $k \leq r$, there exists an algorithm that samples $s_1 + s_2 + \cdots + s_k$, the number of marbles among the first k colors appearing when drawing ℓ marbles from the urn without replacement, in $O(\mathrm{poly}(\log B))$ time and random words.*

Proof. Since we don't have to find the individual counts, we can be more efficient by grouping half the colors together at each step. Formally, we define a pair $\mathbf{D} = \langle D_1, D_2 \rangle$ where $D_1 = C_1 + C_2 + \cdots + C_{r/2}$ and $D_2 = C_{r/2+1} + \cdots + C_{r-1} + C_r$. We then generate $\langle D_1', D_2' \rangle = \mathbf{S}_\ell^{\mathbf{D}}$.

- If $k < r/2$, we recursively solve the problem with the first $r/2$ colors, $B \leftarrow D_1'$, and the original value of k.
- If $k > r/2$, we recurse on the last $r/2$ colors, B set to D_2', and k set to $k - r/2$. In this case, we add D_1' to the returned value.
- Otherwise, $k = r/2$ and we can return D_1'.

The number of recursive calls is $\mathcal{O}(\log r) = \mathcal{O}(\log B)$ (since $r \leq B$). So, the overall runtime is $\mathcal{O}(\mathrm{poly}(\log B))$. ◻

E Local-Access Generators for Random Directed Graphs

In this section, we consider Kleinberg's Small-World model [Kle00, MN04] where the probability that a *directed* edge (u, v) exists is $\min\{c/(\textsc{dist}(u, v))^2, 1\}$. Here, $\textsc{dist}(u, v)$ 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.

E.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: $\Gamma_k^v = \{u : \textsc{dist}(v, u) = k\}$ contains all vertices at a distance k from vertex v. Observe that $|\Gamma_k^v| \le 4k = O(k)$. 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.

E.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 $\mathbb{P}[D \le d]$ 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 $d \le 2(\sqrt{n} - 1)$,

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

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.

E.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 53.** *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 $1/(\texttt{DIST}(u, v))^2$ where* DIST *denote the Manhattan distance, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *query with high probability.*

E.2 Generator for $c \neq 1$

Observe that to support different values of c in the probability function $c/(\texttt{DIST}(u, v))^2$, 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.

E.2.1 Case $c < 1$

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

▶ **Lemma 54.** *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 55.** *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities $c/(\textsc{dist}(u,v))^2$ where $c < 1$, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *query with high probability.*

E.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 56.** *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 $c \cdot p_i \le 1 - \frac{1}{k}$ 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

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

That is, the probability that at least one of the generators report an index i is $1 - (1 - p_i)^{k \cdot c} \ge c \cdot p_i$, 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 $c \cdot p_i \le 1 - \frac{1}{k}$ 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 $\text{poly}(\log n)$. 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 57.** *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities $c/(\textsc{dist}(u,v))^2$ where $c = \text{poly}(\log n)$, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *query with high probability.*

F Omitted Proofs for the Dyck Path Implementation

▶ **Theorem 58.** *There are* $\frac{1}{n+1}\binom{2n}{n}$ *Dyck paths for length* $2n$ *(construction from [Sta15]).*

Proof from [Sta15]. Consider all possible sequences containing $n+1$ up-steps and n down-steps with the restriction that the first step is an up-step. We say that two sequences belong to the same *class* if they are cyclic shifts of each other. Because of the restriction, the total number of sequences is $\binom{2n}{n}$ and each class is of size $n+1$. Now, within each class, exactly one of the sequences is such that the prefix sums are *strictly greater* than zero. From such a sequence, we can obtain a Dyck sequence by deleting the first up-step. Similarly, we can start with a Dyck sequence, add an initial up-step and consider all $n+1$ cyclic shifts to obtain a *class*. This bijection shows that the number of Dyck paths is $\frac{1}{n+1}\binom{2n}{n}$. ◻

F.1 Approximating Close-to-Central Binomial Coefficients

We start with Stirling's approximation which states that

$$m! = \sqrt{2\pi m}\left(\frac{m}{e}\right)^m\left(1+\mathcal{O}\left(\frac{1}{m}\right)\right)$$

We will also use the logarithm approximation when a better approximation is required:

$$\log(m!) = m\log m - m + \frac{1}{2}\log(2\pi m) + \frac{1}{12m} - \frac{1}{360m^3} + \frac{1}{1260m^5} - \cdots \tag{12}$$

This immediately gives us an asymptotic formula for the central binomial coefficient as:

▶ **Lemma 59.** *The central binomial coefficient can be approximated as:*

$$\binom{n}{n/2} = \sqrt{\frac{2}{\pi n}}2^n\left(1+\mathcal{O}\left(\frac{1}{n}\right)\right)$$

Now, we consider a "off-center" Binomial coefficient $\binom{n}{k}$ where $k = \frac{n+c\sqrt{n}}{2}$.

▶ **Lemma 60.**

$$\binom{n}{k} = \binom{n}{n/2}e^{-c^2/2}exp\left(\mathcal{O}(c^3/\sqrt{n})\right)$$

Proof from [Spe14]. We consider the ratio: $R = \binom{n}{k}/\binom{n}{n/2}$:

$$R = \frac{\binom{n}{k}}{\binom{n}{n/2}} = \frac{(n/2)!(n/2)!}{k!(n-k)!} = \prod_{i=1}^{c\sqrt{n}/2}\frac{n/2-i+1}{n/2+i} \tag{13}$$

$$\implies \log R = \sum_{i=1}^{c\sqrt{n}/2}\log\left(\frac{n/2-i+1}{n/2+i}\right) \tag{14}$$

$$= \sum_{i=1}^{c\sqrt{n}/2} -\frac{4i}{n} + \mathcal{O}\left(\frac{i^2}{n^2}\right) = -\frac{c^2n}{2n} + \mathcal{O}\left(\frac{(c\sqrt{n})^3}{n^2}\right) = -\frac{c^2}{2} + \mathcal{O}\left(\frac{c^3}{\sqrt{n}}\right) \tag{15}$$

$$\implies \binom{n}{k} = \binom{n}{n/2}e^{-c^2/2}exp\left(\mathcal{O}(c^3/\sqrt{n})\right) \tag{16}$$

◻

F.2 Dyck Path Boundaries and Deviations

▶ **Lemma 61.** *Given a random walk of length $2n$ with exactly n up and down steps, consider a contiguous* sub-path *of length $2B$ that comprises of U up-steps and D down-steps i.e. $U + D = 2B$. Both $|B - U|$ and $|B - D|$ are $\mathcal{O}(\sqrt{B \log n})$ with probability at least $1 - 1/n^4$.*

Proof. We consider the random walk as a sequence of unbiased random variables $\{X_i\}_{i=1}^{2n} \in \{0,1\}^{2n}$ with the constraint $\sum_{i=1}^{2n} X_i = n$. Here, 1 corresponds to an up-step and 0 corresponds to a down step. Because of the constraint, X_i, X_j are negatively correlated for $i \neq j$ which allows us to apply Chernoff bounds. Now we consider a sub-path of length $2B$ and let U denote the sum of the X_is associated with this subpath. Using Chernoff bound with $\mathbb{E}[X] = B$, we get:

$$\mathbb{P}\left[|U - B| < 3\sqrt{B \log n}\right] = \mathbb{P}\left[|U - B| < 3\frac{\sqrt{\log n}}{\sqrt{B}}B\right] < e^{\frac{9 \log n}{3}} \approx \frac{1}{n^3}$$

Since U and D are symmetric, the same argument applies. ◻

▶ **Corollary 62.** *With high probability, every contiguous sub-path in the random walk (with U up and D down steps) satisfies the property with high probability. Specifically, if $U + D = 2B$, then $|B - U|$ and $|B - D|$ are upper bounded by $c\sqrt{B \log n}$ w.h.p. $1 - 1/n^2$ for all contiguous sub-paths (for some constant c).*

Proof. We can simply apply Lemma 61 and union bound over all n^2 possible contiguous sub-paths. ◻

▶ **Lemma 18.** *Consider a contiguous* sub-path *of a simple Dyck path of length $2n$ where the sub-path is of length $2B$ comprising of U up-steps and D down-steps (with $U + D = 2B$). Then there exists a constant c such that the quantities $|B - U|$, $|B - D|$, and $|U - D|$ are all $< c\sqrt{B \log n}$ with probability at least $1 - 1/n^2$ for every possible sub-path.*

Proof. As a consequence of Theorem 58, we can sample a Dyck path by first sampling a *balanced* random walk with n up steps and n down steps and adding an initial up step. We can then find the corresponding Dyck path by taking the unique cyclic shift that satisfies the Dyck constraint (after removing the initial up-step). Any interval in a cyclic shift is the union of at most two intervals in the original sequence. This affects the bound only by a constant factor. So, we can simply use Corollary 62 to finish the proof. Notice that since $|U - D| \leq |B - U| + |B - D|$, $|U - D| = \mathcal{O}(\sqrt{B \log n})$ comes for free. ◻

▶ **Lemma 19.** *Given a Dyck path sampling problem of length B with U up and D down steps with a boundary at k, there exists a constant c such that if $k > c\sqrt{B \log n}$, then the distribution of paths sampled without a boundary $\mathsf{C}_\infty(U, D)$ (hypergeometric sampling) is statistically $\mathcal{O}(1/n^2)$-close in L_1 distance to the distribution of Dyck paths $\mathsf{C}_k(U + D)$.*

Proof. We use $\mathcal{D}$ and $\mathcal{R}$ to denote the set of all valid Dyck paths and all random sequences respectively. Clearly, $\mathcal{D} \subseteq \mathcal{R}$. Let c be a constant satisfying Corollary 62. Since the random walk/sequence distribution is uniform on $\mathcal{R}$, and by Corollary 62 we see that at least $1 - 1/n^2$ fraction of the elements of $\mathcal{R}$ do not violate the boundary constraint. Therefore, $|\mathcal{D}| \geq (1 - 1/n^2)|\mathcal{R}|$ and so the L_1 distance between $\mathcal{U}_\mathcal{D}$ and $\mathcal{U}_\mathcal{R}$ is $\mathcal{O}(1/n^2)$. ◻

F.3 Estimating the Sampling Probabilities

▶ **Lemma 63.** *Given a Dych sub-path problem within a global Dyck path of size $2n$ and a probability expression of the form $p_d = \frac{S_{left} \cdot S_{right}}{S_{total}}$, there exists a $\mathrm{poly}(\log n)$ time oracle that returns a $\left(1 \pm 1/n^2\right)$ multiplicative approximation to p_d if $p_d = \Omega(1/n^2)$ and returns 0 otherwise.*

Proof. We first compute a $1 + 1/n^3$ multiplicative approximation to $\ln p_d$. Using $\mathcal{O}(\log n)$ terms of the series in Equation 12, it is possible to estimate the logarithm of a factorial up to $1/n^c$ additive error. So, we can use the series expansion from Equation 12 up to $\mathcal{O}(\log n)$ terms. The additive error can also be cast as multiplicative since factorials are large positive integers.

The probability p_d can be written as an arithmetic expression involving sums and products of a constant number of factorial terms. Given a $1 \pm 1/n^c$ multiplicative approximation to $l_a = \ln a$ and $l_a = \ln b$, we wish to approximate $\ln(ab)$ and $\ln(a+b)$. The former is trivial since $\ln(ab) = \ln a + \ln b$. For the latter, we assume $a > b$ and use the identity $\ln(a+b) = \ln a + \ln(1 + b/a)$ to note that it suffices to approximate $\ln(1 + b/a)$. We define $\hat{l}_a = l_a \cdot (1 \pm \mathcal{O}(1/n^c))$ and $\hat{l}_b = l_b \cdot (1 \pm \mathcal{O}(1/n^c))$. In case $\hat{l}_b - \hat{l}_a < c \ln n \implies b/a < 1/n^c$, we approximate $\ln(a+b)$ by $\ln a$ since $\ln(1 + b/a) = \mathcal{O}(1/n^c)$ in this case. Otherwise, using the fact that $l_a - l_b = o(n^2)$, we compute:

$$1 + e^{\hat{l}_b - \hat{l}_a} = 1 + \frac{b}{a} \cdot e^{\mathcal{O}\left(\frac{l_b - l_a}{n^c}\right)} = 1 + \frac{b}{a} \cdot \left(1 \pm \mathcal{O}\left(\frac{1}{n^{c-2}}\right)\right) = \left(1 + \frac{b}{a}\right) \cdot \left(1 \pm \mathcal{O}\left(\frac{1}{n^{c-2}}\right)\right)$$

In other words, the value of c decreases every time we have a sunm operation. Since there are only a constant number of such arithmetic operations in the expression for p_d, we can set c to be a high enough constant (when approximating the factorials) and obtain the desired $1 \pm 1/n^3$ approximation to $\ln p_d$. If $\ln p_d < -3 \ln n$, we approximate $p_d = 0$. Otherwise, we can exponentiate the approximation to obtain $p_d \cdot e^{-\mathcal{O}(\ln n / n^3)} = p_d \left(1 \pm \mathcal{O}(1/n^2)\right)$. ◻

F.4 Omitted Proofs from Section 4.3: Sampling the Height

▶ **Lemma 64.** *For $x < 1$ and $k \geq 1$, we claim that $1 - kx < (1-x)^k < 1 - kx + \frac{k(k-1)}{2} x^2$*

▶ **Lemma 21.** *$S_{left} \leq c_1 \frac{k \cdot \sqrt{\log n}}{\sqrt{B}} \cdot \binom{B}{D-d}$ for some constant c_1.*

Proof. This involves some simple manipulations.

$$S_{left} = \binom{B}{D-d} - \binom{B}{D-d-k} \tag{17}$$

$$= \binom{B}{D-d} \cdot \left[1 - \frac{(D-d)(D-d-1)\cdots(D-d-k+1)}{(B-D-d+k)(B-D-d+k-1)\cdots(B-D-d+1)}\right] \tag{18}$$

$$\leq \binom{B}{D-d} \cdot \left[1 - \left(\frac{D-d-k+1}{B-D+d+k}\right)^k\right] \tag{19}$$

$$\leq \binom{B}{D-d} \cdot \left[1 - \left(\frac{U+d+k-(U-D+d+k-1)}{U+d+k}\right)^k\right] \tag{20}$$

$$\leq \binom{B}{D-d} \cdot \left[1 - \left(\frac{U+d+k-\mathcal{O}(\sqrt{B \log n})}{U+d+k}\right)^k\right] \tag{21}$$

$$\leq \Theta\left(\frac{k\sqrt{\log n}}{\sqrt{B}}\right) \cdot \binom{B}{D-d} \tag{22}$$

◻

▶ **Lemma 22.** *$S_{right} \leq c_2 \frac{k' \cdot \sqrt{logn}}{\sqrt{B}} \cdot \binom{B}{U-d}$ for some constant c_2.*

Proof.

$$S_{right} = \binom{B}{U-d} - \binom{B}{U-d-k'} \tag{23}$$

$$= \binom{B}{U-d} \cdot \left[1 - \frac{(U-d)(U-d-1)\cdots(U-d-k'+1)}{(B-U-d+k')(B-U-d+k'-1)\cdots(B-U-d+1)}\right] \tag{24}$$

$$\leq \binom{B}{U-d} \cdot \left[1 - \left(\frac{U-d-k'+1}{B-U+d+k'}\right)^{k'}\right] \tag{25}$$

$$\leq \binom{B}{U-d} \cdot \left[1 - \left(\frac{2D-U-d-k+1}{2U-D+k+d}\right)^{k'}\right] \tag{26}$$

$$\leq \binom{B}{U-d} \cdot \left[1 - \left(\frac{U+k+d-(2U-2D+2d+2k-1)}{U+k+d}\right)^{k'}\right] \tag{27}$$

$$\leq \binom{B}{U-d} \cdot \left[1 - \left(\frac{U+k+d-\mathcal{O}(\sqrt{B\log n})}{U+k+d}\right)^{k'}\right] \tag{28}$$

$$\leq \Theta\left(\frac{k'\sqrt{\log n}}{\sqrt{B}}\right) \cdot \binom{B}{U-d} \tag{29}$$

□

▶ **Lemma 65.** $S_{tot} \geq \binom{2B}{2D} \cdot \left[1 - \left(1 - \frac{k'}{2U+1}\right)^{k}\right]$.

Proof.

$$S_{tot} = \binom{2B}{2D} - \binom{2B}{2D-k} \tag{30}$$

$$= \binom{2B}{2D} \cdot \left[1 - \frac{(2D)(2D-1)\cdots(2D-k+1)}{(2B-2D+k)(2B-2D+k-1)\cdots(2B-2D+1)}\right] \tag{31}$$

$$\geq \binom{2B}{2D} \cdot \left[1 - \left(\frac{2D-k+1}{2B-2D+1}\right)^{k}\right] \tag{32}$$

$$\geq \binom{2B}{2D} \cdot \left[1 - \left(\frac{2U-(2U-2D+k-1)}{2U+1}\right)^{k}\right] \tag{33}$$

$$\geq \binom{2B}{2D} \cdot \left[1 - \left(\frac{(2U+1)-k'}{2U+1}\right)^{k}\right] \tag{34}$$

$$\geq \binom{2B}{2D} \cdot \left[1 - \left(1 - \frac{k'}{2U+1}\right)^{k}\right] \tag{35}$$

$$\tag{36}$$

□

▶ **Lemma 20.** *When $kk' > 2U+1$, $S_{total} > \frac{1}{2} \cdot \binom{2B}{2D}$.*

Proof. When $kk' > 2U+1 \implies k > \frac{2U+1}{k'}$, we will show that the above expression is greater than $\frac{1}{2}\binom{2B}{2D}$. Defining $\nu = \frac{2U+1}{k'} > 1$, we see that $(1-\frac{1}{\nu})^k \leq (1-\frac{1}{\nu})^{\nu}$. Since this is an increasing function of ν and since the limit of this function is $\frac{1}{e}$, we conclude that

$$1 - \left(1 - \frac{k'}{2U+1}\right)^{k} > \frac{1}{2}$$

□

▶ **Lemma 23.** *When $kk' \leq 2U+1$, $S_{total} \geq c_3 \frac{k \cdot k'}{B} \cdot \binom{2B}{2D}$ for some constant c_3.*

Proof. Now we bound the term $1 - \left(1 - \frac{k'}{2U+1}\right)^k$, given that $kk' \leq 2U+1 \implies \frac{kk'}{2U+1} \leq 1$. Using Bernoulli's inequality, we know that $(1-x)^n \leq 1/(1+nx)$ for $x \in [0,1]$ and $n \in \mathbb{N}$. Since the term $k'/(2U+1)$ is positive and ≤ 1 (since $kk' \leq 2U+1$), we can apply this as follows:

$$1 - \left(1 - \frac{k'}{2U+1}\right)^k \tag{37}$$

$$\geq 1 - \frac{1}{1 + \frac{kk'}{2U+1}} = \frac{\frac{kk'}{2U+1}}{1 + \frac{kk'}{2U+1}} = \frac{kk'}{2U+1} \cdot \frac{1}{1 + \frac{kk'}{2U+1}} \tag{38}$$

$$\geq \frac{kk'}{2U+1} \cdot \frac{1}{2} \qquad \left(\text{Since } \frac{kk'}{2U+1} \leq 1\right) \tag{39}$$

$$\geq \frac{kk'}{\Theta(B)} \tag{40}$$

□

F.5 Omitted Proofs from Section 4.4: Supporting `First-Return` Queries

▶ **Lemma 27.** *If $d > \log^4 n$, then $S_{left}(d) = \Theta\left(\frac{2^{2d+k}}{\sqrt{d}} e^{-r_{left}(d)} \cdot \frac{k-1}{d+k-1}\right)$ where $r_{left}(d) = \frac{(k-2)^2}{2(2d+k-2)}$. Furthermore, $r_{left}(d) = \mathcal{O}(\log^2 n)$.*

Proof. In what follows, we will drop constant factors: Refer to Figure 11 for the setup. The left section of the path reaches one unit above the boundary (the next step would make it touch the boundary). The number of up-steps on the left side is d and therefore the number of down steps must be $d+k-2$. This inclues d down steps to cancel out the upwards movement, and $k-2$ more to get to one unit above the boundary. The boundary for this section is $k' = k-1$. This gives us:

$$S_{left}(d) = \binom{2d+k-2}{d} - \binom{2d+k-2}{d-1} \tag{41}$$

$$= \binom{2d+k-2}{d}\left[1 - \frac{d}{d+k-1}\right] = \binom{2d+k-2}{d}\frac{k-1}{d+k-1} \tag{42}$$

Now, letting $z = 2d+k-2$, we can write $d = \frac{z-(k-2)}{2} = \frac{z - \frac{k-2}{\sqrt{z}}\sqrt{z}}{2}$. Using Lemma 18, we see that $\frac{k-2}{\sqrt{z}}$ should be $\mathcal{O}(\sqrt{\log n})$. If this is not the case, we can simply return 0 because the probability associated with this value of d is negligible. Since $z > \log^4 n$, we can apply Lemma 60 to get:

$$S_{left}(d) = \Theta\left(\binom{z}{z/2} e^{\frac{(k-2)^2}{2z}} \frac{k-1}{d+k-1}\right) = \Theta\left(\frac{2^{2d+k}}{\sqrt{d}} e^{\frac{(k-2)^2}{2(2d+k-2)}} \frac{k-1}{d+k-1}\right)$$

□

▶ **Lemma 28.** *If $U+D-2d-k > \log^4 n$, then $S_{right}(d) = \Theta\left(\frac{2^{U+D-2d-k}}{\sqrt{U+d-2d-k}} e^{-r_{right}(d)} \cdot \frac{U-D+k}{U-d+1}\right)$ where $r_{right}(d) = \frac{(U-D-k-1)^2}{4(U+D-2d-k+1)}$. Furthermore, $r_{right}(d) = \mathcal{O}(\log^2 n)$.*

Proof. The right section of the path starts from the original boundary. Consequently, the boundary for this section is at $k' = 1$. The number of up-steps on the right side is $U - d$ and the number of down steps is $D - d - k + 1$. This gives us:

$$S_{right}(d) = \binom{U + D - 2d - k + 1}{U - d} - \binom{U + D - 2d - k + 1}{U - d + 1} \tag{43}$$

$$= \binom{U + D - 2d - k + 1}{U - d} \left[1 - \frac{D - d - k - 1}{U - d + 1}\right] \tag{44}$$

$$= \binom{U + D - 2d - k + 1}{U - d} \frac{U - D + k}{U - d + 1} \tag{45}$$

Now, letting $z = U + D - 2d - k + 1$, we can write $U - d = \frac{z + (U - D + k - 1)}{2} = \frac{z + \frac{U - D + k - 1}{\sqrt{z}}\sqrt{z}}{2}$. Using Lemma 18, we see that $\frac{k-2}{\sqrt{z}}$ should be $\mathcal{O}(\sqrt{\log n})$. If this is not the case, we can simply return 0 because the probability associated with this value of d is negligible. Since $z > \log^4 n$, we can apply Lemma 60 to get:

$$S_{right}(d) = \Theta\left(\binom{z}{z/2} e^{\frac{(U - D + k - 1)^2}{2z}} \frac{U - D + k}{U - d + 1}\right) \tag{46}$$

$$= \Theta\left(\frac{2^{U + D - 2d - k}}{\sqrt{U + D - 2d - k}} e^{\frac{(U - D + k - 1)^2}{2(U + D - 2d - k + 1)}} \frac{U - D + k}{U - d + 1}\right) \tag{47}$$

□

G Additional related work

Random graph models

The Erdös-Rényi model, given in [ER60], 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 [HLL83] originally for modeling social networks; nonetheless, it has proven to be an useful general statistical model in numerous fields, including recommender systems [LSY03, SC11], medicine [SPT+01], social networks [For10, NWS02], molecular biology [CY06, MPN+99], genetics [CAT16, JTZ04, CSC+07], and image segmentation [SM00]. Canonical problems for this model are the community detection and community recovery problems: some recent works include [CRV15, MNS15, AS15, ABH16]; see e.g., [Abb16] for survey of recent results. The study of Small-World networks is originated in [WS98] has frequently been observed, and proven to be important for the modeling of many real world graphs such as social networks [DMW03, TM67], brain neurons [BB06], 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) [Kle00, MN04]. See also e.g., [New00] and Chapter 20 of [EK10].

Generation of random graphs

The problem of local-access implementation of random graphs has been considered in the aforementioned work [GGN03, NN07, ELMR17], as well as in [MRVX12] 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 [MKI+03, BB05, NLKB11, MH11], and the parallel computation model [AK17].

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 [ELMR17], there are other query models providing a neighbor of a vertex, such as asking for an entry in the adjacency-list representation [GR97], or traversing to a random neighbor [BK10]. On the other hand, the `Vertex-Pair` query is common in the study of dense graphs as accessing the adjacency matrix representation [GGR98]. The `All-Neighbors` query has recently been explicitly considered in local algorithms [FPSV17].

Other constructions of huge pseudorandom functions that are permutations or random hash functions were given in [LR88, NR96, MRVX12].