Local Access to Huge Random Objects Through Partial Sampling

Amartya Shankha Biswas
CSAIL, MIT, Cambridge, MA, USA
asbiswas@mit.edu

Ronitt Rubinfeld
CSAIL, MIT, Cambridge, MA, USA
ronitt@csail.mit.edu

Anak Yodpinyanee
CSAIL, MIT, Cambridge, MA, USA
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 non-negative). 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 ordered trees, and Matching-Bracket queries on random well bracketed expressions (the Dyck language).

Finally, we introduce two features to define a new model that: (1) allows multiple independent (and even simultaneous) instantiations of the same implementation, to be consistent with each other without the need for communication, (2) allows us to generate a richer class of random objects that do not have a succinct description. Specifically, we study uniformly random *valid* q-colorings of an input graph G with maximum degree Δ. This is in contrast to prior work in the area, where the relevant random objects are defined as a distribution with $\mathcal{O}(1)$ parameters (for example, n and p in the $G(n, p)$ model). The distribution over valid colorings is instead specified via a "huge" input (the underlying graph G), that is far too large to be read by a sub-linear time algorithm. Instead, our implementation accesses G through local neighborhood probes, and is able to answer queries to the color of any given vertex in sub-linear time for $q \geq 9\Delta$, in a manner that is consistent with a specific random valid coloring of G. Furthermore, the implementation is memory-less, and can maintain consistency with non-communicating copies of itself.

2012 ACM Subject Classification Theory of computation → Generating random combinatorial structures; Theory of computation → Streaming, sublinear and near linear time algorithms

Keywords and phrases sublinear time algorithms, random generation, local computation

© Amartya Shankha Biswas, Ronitt Rubinfeld, and Anak Yodpinyanee;
licensed under Creative Commons License CC-BY
11th Innovations in Theoretical Computer Science Conference (ITCS 2020).
Editor: Thomas Vidick; Article No. 27; pp. 27:1–27:65
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Digital Object Identifier 10.4230/LIPIcs.ITCS.2020.27

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

1 Introduction

Consider an algorithm performing a computation on a *huge random object* (for example a huge 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 local query access to the object and generate it incrementally "on-the-fly" (as requested by the algorithm)? Such an *implementation* would ideally emulate the random object by answering appropriate queries, in a manner that is consistent with a specific instance of the random object, sampled from the true distribution. This paradigm is useful when we wish to simulate a sub-linear algorithm on a random object, since sampling the entire object up front would ruin its efficiency.

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 of graphs. One emerging theme that we develop further is to provide access to random objects through more complex yet natural queries. For example, consider an implementation for Erdös-Rényi $G(n, p)$ random graphs, where the simplest query $\textsc{Vertex-Pair}(u, v)$ would ask about the existence of edge (u, v), which can be answered trivially by flipping a coin with bias p (thus revealing a single entry in the adjacency matrix). However, many applications involving non-dense graphs would benefit from *adjacency list* access, which we provide through $\textsc{Next-Neighbor}$ and $\textsc{Random-Neighbor}$ queries[1]

The problem of sampling partial information about huge random objects was pioneered in [26, 24, 25], through the implementation of *indistinguishable* pseudo-random objects of exponential size. Further work in [41, 18] considers the implementation of different random graph models. [18] introduced the study of $\textsc{Next-Neighbor}$ queries which provide efficient access to the adjacency list representation. In addition to supporting $\textsc{Vertex-Pair}$ and $\textsc{Next-Neighbor}$, we also introduce and implement a new $\textsc{Random-Neighbor}$ query for undirected graphs with independent edge probabilities.

Finally, we define a new model that allows us to generate a richer class of random objects that do not have a succinct description. Specifically, we study uniformly random *valid* q-colorings of an input graph G with max degree Δ. This is in contrast to prior work in the area, where random objects are defined as a distribution with $\mathcal{O}(1)$ parameters (for example, n and p in the $G(n, p)$ model). The distribution over valid colorings is instead specified via a "huge" input (the underlying graph G), that is too large to be read by a sub-linear time algorithm. Instead, our implementation accesses G using local neighborhood probes.

This new model can be compared to *Local Computation Algorithms*, which also implement query access to a consistent valid solution, and read their input using local probes. Inspired by this connection, we extend our model to support *memory-less* implementations. This allows multiple independent (possibly simultaneous) instantiations to agree on the same

[1] $\textsc{Vertex-Pair}(u, v)$ returns whether u and v are adjacent, $\textsc{Next-Neighbor}(v)$ returns a new neighbor of v each time it is invoked (until none is left), and $\textsc{Random-Neighbor}(v)$ returns a uniform random neighbor of v (if v is not isolated).

random object, without any communication. We show how to implement local access to color of any given node in a random coloring, using sub-linear resources. Unlike LCAs which can generate an *arbitrary* valid solution, our model requires a uniformly random one.

Random Graphs (Section 3 and Section E)

Random graphs are one of the most well studied types of random object. We consider the problem of implementing local access to a number of fundamental random graph models, through natural queries, including Vertex-Pair, Next-Neighbor, and a newly introduced Random-neighbor query[1], using polylogarithmic resources per query. Our results on random graphs are summarized in Table 1.

Undirected Random Graphs with Independent Edge Probabilities. We implement the aforementioned queries 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. Under reasonable assumptions on the ability to compute certain values pertaining to consecutive edge probabilities, our implementations support Vertex-Pair, Next-Neighbor, and Random-Neighbor queries[1], using $\mathcal{O}(\text{poly}(\log n))$ time, space, and random bits. Note that in this setting, Vertex-Pair queries are trivial by themselves, since the existence of an edge depends on an independent random variable. However, maintaining all three types of queries simultaneously is much harder. As in [18] (and unlike many of the implementations presented in [24, 26]), our techniques allow unlimited queries, and the generated random objects are sampled from the true distribution (rather than just being indistinguishable). 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.

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.

Model	Vertex-Pair	Next-Neighbor	Random-Neighbor	All-Neighbors
$G(n,p)$ with $p = \frac{\log^{\mathcal{O}(1)} n}{n}$	[41]	[41]	[41]	[41]
$G(n,p)$ 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	[18]	[18]	??	✗
Small world model	This paper	This paper	This paper	This paper
Random ordered rooted trees	This paper	This paper	This paper	This paper

While Vertex-Pair and Next-Neighbor queries have been considered in prior work [18, 24, 41], we provide the first implementations of these in non-sparse random graph models. Prior results for implementing queries to $G(n,p)$ focused on the sparse case where $p = \log^{\mathcal{O}(1)} n/n$ [41]. 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 is more involved, and

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 generate a neighbor efficiently. NEXT-NEIGHBOR queries were introduced in [18] in order to access the neighborhoods of vertices in non-sparse graphs in *lexicographic order*. This query however, does not allow us to efficiently explore graphs in some natural ways, such as via random walks, since the initially returned neighbors are biased by the lexicographic ordering. We address this deficiency by introducing a new RANDOM-NEIGHBOR query, which would be useful, for instance, in sub-linear algorithms that employ random walk processes. We provide the first implementation of all three queries in *non-sparse graphs* as follows:

▶ **Theorem 1.** *Given a random graph model defined by the probability matrix $\{p_{uv}\}_{u,v\in[n]}$, and assuming that we can compute the the quantities $\prod_{u=a}^{b}(1-p_{vu})$ and $\sum_{u=a}^{b} p_{vu}$ in $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ time, there exists an implementation for this model that supports local access through* RANDOM-NEIGHBOR*,* VERTEX-PAIR*, and* NEXT-NEIGHBOR *queries using $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ running time, additional space, and random bits per query.*

We show that these assumptions can be realized in Erdös-Rényi random graphs and the Stochastic Block Model. SBM presents additional challenges for assigning community labels to vertices (Section 2.1.1).

▶ **Corollary 2.** *There exists an algorithm that implements local access to a Erdös-Rényi $G(n,p)$ random graph, through* VERTEX-PAIR*,* NEXT-NEIGHBOR*, and* RANDOM-NEIGHBOR *queries, while using $\mathcal{O}(\log^3 n)$ time, $\mathcal{O}(\log^3 n)$ random bits, and $\mathcal{O}(\log^2 n)$ additional space per query with high probability.*

▶ **Corollary 3.** *There exists an algorithm that implements local access to a random graph from the n-vertex Stochastic Block Model with r randomly-assigned communities, through* VERTEX-PAIR*,* NEXT-NEIGHBOR*, and* RANDOM-NEIGHBOR *queries, using $O(r\,\mathrm{poly}(\log n))$ time, random bits, and space per query w.h.p.*

We remark that while we are able to generate Erdös-Rényi random graphs on-the-fly supporting all three types of queries, our construction still only requires $\widetilde{\mathcal{O}}(m+n)$ time and space to generate a complete $G(n,p)$ graph, which is optimal up to logarithmic factors.

▶ **Corollary 4.** *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.*

Directed Random Graphs – The Small World Model (Section E). We consider local-access implementations for directed graphs through Kleinberg's Small World model, where the probabilities are based on distances in a 2-dimensional grid. This model was introduced in [31] to capture the geographical structure of real world networks, in addition to reproducing observed properties of short routing paths and low diameter. We implement ALL-NEIGHBORS queries for the Small World model, using $\mathcal{O}(\mathrm{poly}(\log n))$ time, space and random bits. Since such graphs are sparse, the other queries follow directly.

▶ **Theorem 5.** *There exists an algorithm that implements* ALL-NEIGHBORS *queries for a random graph from Kleinberg's Small World model, where probability of including each directed non-grid edge (u,v) in the graph is $c/(\mathrm{DIST}(u,v))^2$, where the range of allowed values of c is $\log^{\pm\mathcal{O}(1)} n$ and* DIST *denotes the Manhattan distance, using $O(\log^2 n)$ time and random words per query with high probability.*

Catalan Objects (Section 4)

Catalan objects capture a well studied combinatorial property and admit numerous interpretations that include well bracketed expressions, rooted trees and binary trees, Dyck languages etc. In this paper, we focus on the Dyck path interpretation and implement local access to a uniformly random instance of a Dyck path. Since Dyck paths have natural bijections to other Catalan objects, we can use our Dyck Path implementation to obtain implementations of these random Catalan objects.

A Dyck path is defined as a sequence of n upward $(+1)$ steps, and n downward (-1) steps, with the added constraint that the *sum of any prefix of the sequence is non-negative*. A *random* Dyck path may be viewed as a constrained one dimensional balanced random walk. A natural query on Dyck paths is $\textsc{Height}(t)$ which returns the position of the walk after t steps (equivalently, sum of the first t sequence elements). $\textsc{Height}$ queries correspond to $\textsc{Depth}$ queries on rooted trees and bracketed expressions (Section 4.1).

We also introduce and support $\textsc{First-Return}$ queries, where $\textsc{First-Return}(t)$ returns the first time when the random walk returns to the same *height* as it was at time t, as long as $\textsc{Height}(t+1) = \textsc{Height}(t) + 1$ (Section 4.1 presents the rationale for this definition). This allows us to support more involved queries that are widely used; for example, $\textsc{First-Return}$ queries are equivalent to finding the next child of a node in a random rooted tree, and also to finding a matching closing bracket in random bracketed expressions.

$\textsc{Height}$ queries for unconstrained random walks follow trivially from the implementation of interval summable functions in [26, 23]. However, the added non-negativity constraint introduces intricate non-local dependencies on the distribution of positions. We show how to overcome these challenges, and support both queries using $\mathcal{O}(\text{poly}(\log n))$ resources.

▶ **Theorem 6.** *There is an algorithm using $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ resources per query that provides access to a uniformly random Dyck path of length $2n$ by implementing the following queries:*

- $\textsc{Height}(t)$ *returns the position of the walk after t steps.*
- $\textsc{First-Return}(t)$*: If* $\textsc{Height}(t+1) > \textsc{Height}(t)$*, then* $\textsc{First-Return}(t)$ *returns the smallest index t', such that $t' > t$ and* $\textsc{Height}(t') = \textsc{Height}(t)$ *(i.e. the first time the Dyck path returns to the same height). Otherwise, if* $\textsc{Height}(t+1) < \textsc{Height}(t)$*, then* $\textsc{First-Return}(t)$ *is not defined.*

Random Coloring of Graphs – A New Model (Section 5)

So far, all the results in this area have focused on random objects with a small description size; for instance, the $G(n,p)$ model is described with only two parameters n and p. We introduce a new model for implementing random objects with *huge input description*; that is, the distribution is specified as a uniformly random solution to a huge combinatorial problem. The challenge is that now our algorithms cannot read the entire description in sub-linear time.

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 generated 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 underlying graph (in the manner of *local computation algorithms* [48, 5]). Unlike LCAs which can return an *arbitrary* 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 $q \geq 9\Delta$.

This implementation also has the feature that 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 generate exactly the same coloring, regardless of the queries asked. Since these implementations are memory-less, the resulting coloring is *oblivious* to the order of queries, and only depends on common random bits,

▶ **Theorem 7.** *Given neighborhood query access to a graph G with n nodes, maximum degree Δ, and $q = 2\alpha\Delta \geq 9\Delta$ colors, we can generate the color of any given node from a distribution of color assignments that is ϵ-close (in L_1 distance) to the uniform distribution over all valid colorings of G, in a consistent manner, using only $\mathcal{O}((n/\epsilon)^{3.06/\alpha}\Delta\log(n/\epsilon))$ time, probes, and public random bits per query.*

1.1 Related Work

The problem of computing local information of huge random objects was pioneered in [24, 26]. Further work of [41] considers the implementation of sparse $G(n,p)$ graphs from the Erdös-Rényi model [17], with $p = O(\text{poly}(\log n)/n)$, by implementing ALL-NEIGHBORS queries. 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 [18], the authors construct an oracle for the implementation of recursive trees, and BA preferential attachment graphs. Unlike prior work, their implementation allows for an arbitrary number of queries. 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 [18] provides access to the adjacency list through NEXT-NEIGHBOR queries. For additional related work, see Section G.

1.2 Applications

One motivation for implementing local access to huge random objects is so that we can simulate sub-linear time algorithms on them. The standard paradigm of generating the entire (input) object a priori is wasteful, because sub-linear algorithms only inspect a small fraction of the generated input. For example, the greedy routing algorithm on Kleinberg's small world networks [31] only uses $\mathcal{O}(\log^2 n)$ probes to the underlying network. 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, by generating only those parts of the graph that are accessed by the routing algorithm.

Local access implementations may also be used to design parallel generators for random objects. The model in Definition 10 is particularly suited to parallelization; different processors/machines can generate parts of the random object independently, using a *read-only* shared memory containing public random bits.

2 Model and Overview of our Techniques

We begin by formalizing our model of *local-access implementations*, inspired by [18], but we add some aspects that were not addressed in earlier models. First, we define families of random objects.

▶ **Definition 8.** *A **random object family** maps a description* Π *to a distribution* X^{Π} *over the set* $\mathbb{X}^{\Pi}$*.*

For example, the *family* of Erdös-Rényi graphs maps $\Pi = (n, p)$ to a distribution over $\mathbb{X}^{\Pi}$, which is the set of all possible n-vertex graphs, where the probability assigned to any graph containing m edges in the distribution X^{Π} is exactly $p^m \cdot (1-p)^{\binom{n}{2}-m}$.

▶ **Definition 9.** *Given a **random object family*** $\{(\mathsf{X}^{\Pi}, \mathbb{X}^{\Pi})\}$ *parameterized by* Π*, a* local access implementation *of a family of query functions* $\langle F_1, F_2, \cdots \rangle$ *where* $F_i : \mathbb{X}^{\Pi} \to \{0, 1\}$*, provides an oracle* $\mathcal{M}$ *with an internal state for storing a partially generated random object. Given a description* Π *and a query* F_i*, the oracle returns the value* $\mathcal{M}(\Pi, F_i)$*, and updates its internal state, while satisfying the following:*

- ***Consistency:*** *There must be a single* $X \in \mathbb{X}^{\Pi}$*, such that for all queries* F_i *presented to the oracle, the returned value* $\mathcal{M}(\Pi, F_i)$ *equals the true value* $F_i(X)$*.*
- ***Distribution equivalence:*** *The random object* $X \in \mathbb{X}^{\Pi}$ *consistent with the responses* $\{F_i(X)\}$ *must be sampled from some distribution* $\hat{\mathsf{X}}^{\Pi}$ *that is* ϵ*-close to the desired distribution* X^{Π} *in* L_1*-distance. In this work, we focus on supporting* $\epsilon = n^{-c}$ *for any desired constant* $c > 0$*.*
- ***Performance:*** *The computation time, and random bits required to answer a single query must be sub-linear in* $|X|$ *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.

For example, in the $G(n, p)$ family with description $\Pi = (n, p)$, we can define VERTEX-PAIR query functions $\{F_{(u,v)}\}_{u,v \in [n]}$. So, given a graph $G \in \mathbb{X}^{\Pi}$, the query $F_{(u,v)}(G) = 1$ if and only if $(u, v) \in G$.

In prior work [18, 26, 41] as well as some of our results, the input description Π is of small size (typically $\mathcal{O}(\log n)$), and the oracle $\mathcal{M}$ can read all of Π (for example, $\Pi = (n, p)$ in $G(n, p)$).

Distributions with Huge Description Size. We initiate the study of **random object families** where the description Π is too large to be read by a sub-linear time algorithm. In this setting, the oracle from Definition 9 cannot read the entire input Π, and instead accesses it through local probes. For instance, consider the **random object family** that maps a graph G to the uniform distribution over valid colorings of G. Here, the description Π includes the entire graph G, which is too large to be read by a sub-linear time algorithm. In this case, the oracle can query the underlying graph using neighborhood probes. The number of such probes used to answer a single query must be sub-linear in the input size.

Supporting Independent Query Oracles: Memory-less Implementations

The model in Definition 9 only supports sequential queries, since the response to a future query may depend on the changes in internal state caused by past queries. In some applications, we may want to have multiple independent query oracles whose responses are all consistent with each other. One way to achieve this is to restrict our attention to *memory-less* implementations; ones without any internal state. An important implication of being memory-less is that the responses to each query is oblivious to the order of queries being asked. In fact, the lack of internal state implies that independent implementations that use the same random bits and the same input description must respond to queries in the same way. Thus, instead of using the internal state to maintain consistency, memory-less implementations are given access to the same public random oracle.

For the problem of sampling a random graph coloring, we present an implementation that is memory-less and also accesses the input description through local probes, as elaborated in the following model:

▶ **Definition 10.** *Given a* ***random object family*** $\{(\mathsf{X}^\Pi, \mathbb{X}^\Pi)\}$ *parameterized by input* Π*, a* local access implementation *of a family of query functions* $\langle F_1, F_2, \cdots \rangle$*, provides an oracle* $\mathcal{M}$ *with the following properties.* $\mathcal{M}$ *has query access to the input description* Π*, and a tape of public random bits* $\mathbf{R}$*. Upon being queried with* Π *and* F_i*, the oracle uses sub-linear resources to return the value* $\mathcal{M}(\Pi, \mathbf{R}, F_i)$*, which must equal* $F_i(X)$ *for a specific* $X \in \mathbb{X}^\Pi$*, where the choice of* X *depends only on* $\mathbf{R}$*, and the distribution of* X *(over* $\mathbf{R}$*) is* $\frac{1}{n^c}$*-close to the distribution over* $\mathbb{X}^\Pi$*, for any given constant* c*. Thus, different instances of* $\mathcal{M}$ *with the same description* Π *and the same random bits* $\mathbf{R}$*, must agree on the choice of* X *that is consistent with all answered queries regardless of the order and content of queries that were actually asked.*

We can contrast Definition 10 with the one for *Local Computation Algorithms* [48, 5] which also allow query access to *some* valid solution by reading the input through local probes. The additional challenge in our setting is that we also have to make sure that we return a uniformly random solution, rather than an arbitrary one. We also note that the memory-less property may be achieved for small description size **random object families**. For instance, our implementation for the directed small world model admits such a memory-less implementation using public random bits.

2.1 Undirected Graphs

We consider 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), from which the results can be applied to Erdös-Rényi random graphs and the Stochastic Block Model. Throughout, we identify our vertices via their unique IDs from 1 to n, namely $V = [n]$. In this model, 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 [18], we provide an implementation of next-neighbor queries, which return the neighbors of any given vertex one by one in lexicographic order. Finally, we introduce a new query: Random-Neighbor that returns a uniformly random neighbor of any given vertex. This would be useful for any algorithm that performs random walks. Random-Neighbor queries in non-sparse graphs present particularly interesting challenges that are outlined below.

Next-Neighbor Queries

In Erdös-Rényi graphs, the (lexicographically) next neighbor of a vertex can be recovered by generating consecutive entries of the adjacency matrix until a neighbor is found, which takes roughly $\Omega(1/p_{uv})$ time. For small edge probabilities $p_{uv} = o(1)$, this implementation is inefficient, and we show how to improve the runtime to $\mathcal{O}(\text{poly}(\log n))$. Our main technique is to sample 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 how to compute this quantity efficiently for $G(n, p)$ and the Stochastic Block Model.

This strategy of *skip-sampling* is also used in [18]. However, in our work, the main difficulty 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. 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 randomized implementation (Algorithm 1) that supports Next-Neighbor queries efficiently, but has a complicated performance analysis. We remark that this approach may be extended to support Vertex-Pair queries (Section A) with superior performance (if we do not support Random-Neighbor queries), and also to provide deterministic resource usage guarantee (Section C).

Random-Neighbor Queries

We implement Random-Neighbor queries (Section 3.2) using poly($\log n$) resources. The ability to do so is surprising since: (1) Sampling the degree of vertex, may not be viable for *sub-linear* implementations, because this quantity alone imposes dependence on the existence of *all* of its potential incident edges and consequently on the rest of the graph (since it is undirected). Thus, our implementation needs to return a random neighbor, with probability reciprocal to the query vertex's degree, without resorting to *determining* 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 an approach which samples many consecutive edges simultaneously, 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 $\{B_v^{(i)}\}$ called blocks, so that each $B_v^{(i)}$ contains $\Theta(1)$ neighbors in expectation (i.e. $\sum_{u \in B_i} p_{vu} = \Theta(1)$). The subroutine of Next-Neighbor is applied to sample the neighbors within a block in expected $\widetilde{\mathcal{O}}(1)$ time. We can now obtain a neighbor of v by picking a random neighbor from a random block, but this introduces a bias because all blocks may not have the same number of neighbors. We remove this bias by rejecting samples from block $B_v^{(i)}$ with probability proportional to the number of neighbors in $B_v^{(i)}$.

2.1.1 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 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 4).

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

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}. A naive solution would be to simply assign communities to contiguous (by index) blocks of vertices, which would easily allow us to calculate the relevant sums/products of probabilities on continuous ranges of indices, with some additional case analysis to check when we are at a community boundary. However, this setup is unrealistic, and not particularly useful in the context of the Stochastic Block model. 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 [40]. Thus, we construct implementations where the community assignments are sampled from some given distribution R, or from a collection of specified community sizes $\langle |C_1|, \ldots, |C_r| \rangle$. The main difficulty here is to obtain a uniformly sampled assignment of vertices to communities on-the-fly.

Since the probabilities of potential edges now depend on the communities of their endpoints, we can't obtain closed form expressions for the relevant sums and products of probabilities. However, we observe that it suffices 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 [26], which maintains these counts for ranges of vertices, and determines 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 $\mathcal{O}(r \cdot \text{poly}(\log n))$ overhead in required resources for each operation.

2.2 Directed Graphs

Lastly, we consider Kleinberg's Small World model ([31, 37]) 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 [55], this important special case of Kleinberg's model, defined on two-dimensional grids, demonstrates the underlying geographical structures of networks.

In this model, each vertex is identified via its 2D coordinate $v = (v_x, v_y) \in [\sqrt{n}]^2$. Defining the Manhattan distance as $\textsc{dist}(u, v) = |u_x - v_x| + |u_y - v_y|$, 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 implement `All-Neighbor` queries, which in turn can emulate `Vertex-Pair`, `Next-Neighbor` and `Random-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 implement `All-Neighbors` queries in the small world model by listing all neighbors from closest to furthest away from the queried vertex, using $\text{poly}(\log n)$ resources per query. The main challenge is to sample for the next closest neighbor, when the probabilities are a function of the Manhattan distance on the lattice. Rather than sampling for a neighbor directly, we partition the nodes based on their distance from v (there are $\Theta(d)$ vertices at distance d). We first choose the next smallest distance partition with a neighbor using rejection sampling techniques (Lemma 11) on the appropriate distribution. In the second step, we generate all the neighbors within that partition using *skip-sampling*.

2.3 Random Catalan Objects

Many important combinatorial objects can be interpreted as Catalan objects. One such interpretation is a Dyck path; a one dimensional random walk on the line with n up and n down steps, starting from the origin, with the constraint that the height is always non-negative. We implement $\textsc{Height}(t)$, which returns the position of the walk at time t, and $\textsc{First-Return}(t)$, which returns the first time when the random walk returns to the same position as it was at time t. These queries are natural for several types of Catalan objects. As noted previously, we can use standard bijections to translate the Dyck path query implementations into natural queries for *bracketed expressions* and *ordered rooted trees*. Specifically, HEIGHT values in Dyck paths are equivalent to *depth* in bracket expressions and trees. The FIRST-RETURN queries are more involved, and are equivalent to finding the *matching bracket* in bracket expressions, and alternately to finding the *next child* of a node in an ordered rooted tree (see Section 4.1).

Over the course of the execution, our algorithm will determine the height of a random Dyck path 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 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 determined, but none of the intermediate heights are known. The important observation is that the unknown section of the path within an *interval* is entirely determined by the positions and heights of the endpoints, and in particular is completely independent of all other *intervals*. So, each interval $[x_i, x_{i+1}]$ along with corresponding heights $\{y_i, y_{i+1}\}$, represents a generalized Dyck problem with U up steps, D down steps, and a constraint that the path never dips more than y_i units below the starting height.

Height Queries

General $\textsc{Height}(x)$ queries can then be answered by recursively halving the *interval* containing x, by repeatedly sampling the height at the midpoint, until the height of position x is sampled. We start by implementing a subroutine that given an *interval* $[x_i, x_{i+1}]$ of length $2B$, containing $2U$ up and $2D$ down steps, determines the number of up steps $U' = U + d$ assigned 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 $\textsc{Height}(x_i + B)$. This is done by sampling the parameter d from a distribution $\{p_d\}$ where $p_d \equiv S_{left}(d) \cdot S_{right}(d)/S_{total}$. Here, $S_{left}(d)$ (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.

The problem of determining the number of up steps in the first half of the *interval* was solved for the unconstrained case (where the sequence is just a random permutation of up and down steps) in [26]. 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 $\mathcal{O}(\log n)$ and has an efficiently computable CDF. This allows us to sample from $\{q_d\}$ and leverage rejection sampling techniques (see Lemma 11 in Section 2.5) to obtain samples from $\{p_d\}$.

First-Return Queries

Note that $\textsc{First-Return}(x)$ is only of interest when the first step after position x is upwards (i.e. $\textsc{Height}(x+1) > \textsc{Height}(x)$), since this situation is important for complex queries to random bracketed expressions and random rooted trees. Section 4.1 details the rationale

for this definition based on bijections between these objects. FIRST-RETURN queries are challenging because we need to find the *interval* containing the first return to HEIGHT(x). Since there could be up to $\Theta(n)$ intervals, it is inefficient to iterate through all of them. To circumvent this problem, we allow each interval $[x_i, x_{i+1}]$ to sample and maintain its own boundary constraint k_i instead of using the global non-negativity constraint. This implies that the path within the interval $[x_i, x_{i+1}]$ never reaches the height $y_i - k_i$ or lower. Additionally, we maintain a crucial invariant that states that this boundary is achieved by the endpoint of lower height i.e. $\min(y_i, y_{i+1})$. If the invariant holds, we can find the interval containing FIRST-RETURN(x) by finding the smallest determined position $x_j > x$ whose sampled height $y_j \leq$ HEIGHT(x), and considering the interval $[x_{j-1}, x_j]$ preceding x_j. We use an interval tree to update and query for the known heights.

Unfortunately, every HEIGHT query creates new intervals by sub-dividing existing ones, potentially breaking the invariant. We re-establish the invariant for $[x_i, x_{i+1}]$ by generating a "mandatory boundary" h (a boundary constraint with the added restriction that some position within the interval *must* touch the boundary), and then sampling a position $x_{mid} \in [x_i, x_{i+1}]$ such that HEIGHT$(x_{mid}) = h$ (Figure 6). This creates new intervals $[x_i, x_{mid}]$ and $[x_{mid}, x_{i+1}]$, both of which have a boundary constraint at h.

The first step of sampling the *mandatory boundary* is performed by binary searching on the possible boundary locations using an appropriate CDF (Section 4.4.2). To find an intermediate position touching this boundary, we parameterize the position with d, and find the distribution $\{p_d\}$ associated with the various possibilities. Since we cannot directly sample from this complicated distribution, we define a *piecewise continuous* probability distribution $\hat{q}(\delta)$ such that $\hat{q}(\delta)$ approximates $p_{\lfloor \delta \rfloor}$ (Section 4.4.3). We then use this to define a discrete distribution $\{q_d\}$ where $q_d = \int_d^{d+1} \hat{q}(\delta)$, where we can efficiently compute the CDF of $\{q_d\}$ by integrating the piecewise continuous $\hat{q}(\delta)$. The challenge here is to construct an appropriate $\hat{q}(\delta)$ that only has $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ continuous pieces. This allows us to again use the rejection sampling technique (Lemma 11) to indirectly obtain a sample from $\{p_d\}$.

2.4 Random Coloring of a Graph

Finally, we introduce a new model (Definition 10) for implementing huge random objects, where the distribution is specified as a uniformly random solution to a huge combinatorial problem. 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. Since the implementation has to run in sub-linear time, it is not possible to read the entire input G during a single query execution.

`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 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 [22]. 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 [20] for distributed graph coloring, and mixes in $\mathcal{O}(\log n)$ epochs when $q \geq 9\Delta$. In order to implement the query COLOR(v), it suffices

to implement $\textsc{Accepted}(v, t)$ that indicates whether the proposal for v was accepted in the tth epoch. This depends on the prior colors of the potentially Δ neighbors of v. Determining the prior colors of all the neighbors w using recursive calls would result in Δ invocations of $\textsc{Accepted}(w, t-1)$ (at the preceding epoch $t-1$). Naively, this gives a bound of Δ^t on the number of invocations. We can prune the recursions by only considering neighbors who proposed the color c during *some* past epoch. This reduces the expected number of recursive calls to $t\Delta/q$, since there are $t\Delta$ potential proposals and each one is c with probability $1/q$. If q is larger than $t\Delta$, the number of recursive calls is less than 1, which gives a sub-linear bound on the total number of resulting invocations. Since the number of epochs t can be as large as $\Theta(\log n)$, this strategy will only work when $q = \Omega(\Delta \log n)$.

The improvement to $q = \Omega(\Delta)$ follows from the observation that for a neighbor w that proposed color c at epoch t', the recursive call corresponding to w can directly jump to epoch t'. If the conflicting color c was indeed accepted at that epoch, we then step *forwards* through epochs $t'+1, t'+2, \cdots$, to check whether c was overwritten by some future accepted proposal. This strategy dramatically reduces the recursive sub-problem size (given by the epoch number t'), and furthermore we show that we do not have to step through too many future epochs in order to check whether c was overwritten. This allows us to bound the runtime by $\widetilde{\mathcal{O}}\left(t\Delta(n/\epsilon)^{6.12\Delta/q}\right)$, where the overall coloring is sampled from a distribution that is ϵ-close to uniform (see Definition 10).

One requirement for our strategy is the ability to access a *valid initial coloring* (the initial state of the Markov Chain) through local probes, in addition to local probes to the underlying graph structure. This assumption can be removed by using a result of [10], that presents an LCA for $\Delta+1$ graph coloring using $\Delta^{\mathcal{O}(1)} \log n$ graph probes. Alternately, we can assign random initial colors to the vertices, which may result in an *invalid* final coloring. However, the Markov Chain will transform the initial invalid coloring into a valid one with high probability.

2.5 Basic Tools for Efficient Sampling

In this section, we describe the main techniques used to sample from a distribution $\{p_i\}_{i\in[n]}$, which differs based on the type of access to $\{p_i\}$ provided to the algorithm. If the algorithm is given cumulative distribution function (CDF) access to $\{p_i\}$, 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 $\{p_i\}$ in L_1 distance.

Sampling can be more challenging when when we can only access the probability distribution function (PDF) of $\{p_i\}$, The approach that we use in this work is to construct an auxiliary distribution $\{q_i\}$ such that: (1) $\{q_i\}$ has an efficiently computable CDF, and (2) q_i approximates p_i pointwise to within a polylogarithmic multiplicative factor for "most" of the support of $\{p_i\}$. The following Lemma inspired by [26] formalizes this concept.

▶ **Lemma 11.** *Let $\{p_i\}_{i\in[n]}$ and $\{q_i\}_{i\in[n]}$ be distributions on $[n]$ satisfying the following conditions:*

1. *There is a $\log^{\mathcal{O}(1)} n$ time algorithm to approximate p_i and q_i up to a multiplicative $\left(1 \pm \frac{1}{n^c}\right)$ factor.*
2. *We can generate an index i according to a distribution $\{\hat{q}_i\}$, where $\hat{q}_i$ is a $\left(1 \pm \frac{1}{n^c}\right)$ multiplicative approximation to q_i.*
3. *There exists a poly$(\log n)$-time recognizable set S such that*
   - $1 - \sum_{i\in S} p_i < \frac{1}{n^c}$
   - *For every $i \in S$, it holds that $p_i \le \log^{\mathcal{O}(1)} n \cdot q_i$*

Then, with high probability we can use only $\log^{\mathcal{O}(1)} n$ samples from $\{\hat{q}_i\}$ to generate an index i according to a distribution that is $\mathcal{O}\left(\frac{1}{n^c}\right)$-close to $\{p_i\}$ in L_1 distance.

Proof. We begin by setting an upper bound $U = \log^{\mathcal{O}(1)} n$ on p_i/q_i for all $i \in S$. The sampling proceeds in iterations, such that in each iteration we obtain an index i according to the distribution $\{\hat{q}_i\}$. If $i \notin S$, this index is returned with probability $\tilde{p}_i/U\tilde{q}_i$, where $\tilde{p}_i$ and $\tilde{q}_i$ are the $\left(1 \pm \frac{1}{n^c}\right)$ multiplicative approximations to p_i and q_i, Otherwise, we repeat this process until some output is returned.

The probability of returning index $i \in S$ in a particular iteration is $\hat{q}_i \cdot (\tilde{p}_i/U\tilde{q}_i)$, which is in turn a $\left(1 \pm \frac{1}{n^c}\right)$ multiplicative approximation to p_i/U. Hence, the probability of success in a single iteration is roughly $1/U$, and therefore we only need $\mathcal{O}(U \log n) = \log^{\mathcal{O}(1)} n$ iterations (and the same number of samples from $\{\hat{q}_i\}$) in order to succeed with high probability. The resulting distribution of indices approximates $\{p_i\}$ pointwise on the domain S, up to a factor of $\left(1 \pm \frac{1}{n^c}\right)$. Since the remainder of the domain contains at most $\frac{1}{n^c}$ probability mass, the output distribution is $\mathcal{O}\left(\frac{1}{n^c}\right)$ close to $\{p_i\}$ in L_1 distance. ◀

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 1.2, 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 **last** (used in the same sense as [18]), 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 **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*, 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}.

Obstacles for maintaining A explicitly. 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}: 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 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 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 12.** *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 process as an "uncovering" of the table of RVs X_{uv} (introduced in Section 3). Consider the following strategy to find

Algorithm 1 Sampling NEXT-NEIGHBOR.

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

NEXT-NEIGHBOR(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 ith coin C_{vu} corresponding to $u = \mathbf{last}[v] + i$ has bias p_{vu}, regardless of whether X_{vu} is decided or not. Then, we use the sequence $\langle C_{vu} \rangle$ to assign values to *undecided* random variables 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 be discarded. Thus, we generate coin-flips until we encounter some u satisfying both $C_{vu} = 1$ and $\mathbf{A}[v][u] = \phi$.

Let $\mathsf{F}(v, a, b)$ denote the probability distribution of the occurrence u of the first coin-flip $C_{vu} = 1$ 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. This process generates u satisfying $C_{vu} = 1$ in increasing order, until we find one that also satisfies $\mathbf{A}[u][v] = \phi$ (this outcome is captured by the condition $\mathbf{last}[u] < v$), or until the next generated u is equal to w_v. 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 approach.

3.2 Final Implementation Using Blocks

We begin this section by focusing first on RANDOM-NEIGHBOR queries, then extend the construction to the remaining queries. In order to handle RANDOM-NEIGHBOR(v), we divide the neighbors of v into *blocks* $\mathbf{B}_v = \{B_v^{(1)}, B_v^{(2)}, \ldots, B_v^{(i)}, \ldots\}$, so that each block contains, in expectation, roughly the same number of neighbors of v. We implement RANDOM-NEIGHBOR(v) by randomly selecting a block $B_v^{(i)}$, 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 block. As the block 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 block: with constant probability the block contains some neighbors, and with high probability it has at most $O(\log n)$ neighbors.

As the actual number of neighbors appearing in each block will be different, we balance out these discrepancies by performing *rejection sampling*. This equalizes the probability of choosing any neighbor implicitly, without the knowledge of $\deg(v)$. Leveraging the fact that the maximum number of neighbors in any block 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, 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 Blocks

We fix a sufficiently large constant L, and assign the vertex u to the $\lceil \sum_{i=1}^{u} p_{vi}/L \rceil^{\text{th}}$ block of v. Essentially, each block represents a contiguous range of vertices, where the expected number of neighbors of v in the block is $\approx L$ (for example, in $G(n,p)$, each block contains $\approx L/p$ vertices). We define $\Gamma^{(i)}(v) = \Gamma(v) \cap B_v^{(i)}$, the neighbors appearing in block $B_v^{(i)}$. 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 blocks except possibly the last one).

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

▶ **Lemma 13.** *With high probability, the number of neighbors in every block, $|\Gamma^{(i)}(v)|$, is at most $\mathcal{O}(\log n)$.*

▶ **Lemma 14.** *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 blocks $\{B_v^{(i)}\}_{i \in [|\mathbf{B}_v|]}$ are non-empty.*

We consider blocks to be in two possible states – filled or unfilled. Initially, all blocks are considered unfilled. In our algorithm we will maintain, for each block $B_v^{(i)}$, the set $P_v^{(i)}$ of known neighbors of u in block $B_v^{(i)}$; this is a refinement of the set P_v in Section 3.1. We define the behaviors of the procedure FILL(v,i) as follows. When invoked on an unfilled block $B_v^{(i)}$, 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 FILL to Section 3.3, instead using it as a black box.

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

Algorithm 2 Block sampling.

```
procedure RANDOM-NEIGHBOR(v)
    while True
        sample B_v^(i) ~_U B_v u.a.r.
        if B_v^(i) is not filled
            FILL(v, i)
        with probability |P_v^(i)| / M
            return u ~_U P_v^(i) u.a.r
```

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 $\Gamma(v)$.

To obtain a random neighbor, we first choose a block $B_v^{(i)}$ uniformly at random, and invoke $\textsc{Fill}(v, i)$ if the block is *unfilled*. Then, we *accept* the sampled block 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 block (see Lemma 13), we accept block $B_v^{(i)}$ 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 block. If we choose to accept $B_v^{(i)}$, we return a random neighbor from $P_v^{(i)}$. Otherwise, *reject* this block 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 15.** *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 16.** *Algorithm 2 terminates in $\mathcal{O}(\log n)$ iterations in expectation, or $\mathcal{O}(\log^2 n)$ iterations w.h.p.*

Proof. Using Lemma 14, we conclude that the probability of choosing a non-empty block is at least $1/3$. Since $M = \Theta(\log n)$ by Lemma 13, 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 $\textsc{Fill}$ 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

Algorithm 3 Filling a block.

```
procedure FILL(v, i)
    (a, b) ← B_v^(i)
    while a < b
        sample u ~ F(v, a, b)
        B_u^(j) ← block containing v
        if B_u^(j) is not filled
            P_v^(i) ← P_v^(i) ∪ {u}
            P_u^(j) ← P_u^(j) ∪ {v}
        a ← u
    mark B_u^(j) as filled
```

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.

As discussed in Section 3.1, while we have recorded all occurrences of $\mathbf{A}[v][u] = 1$ in $P_v^{(i)}$, we need an 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 12. Here instead, we maintain a single bit marking whether each block is filled or unfilled: a filled block implies that $\mathbf{A}[v][u]$ for all $u \in B_v^{(i)}$ are decided. The block structure along with the mark bits, unlike **last**, is capable of handling intermittent ranges of intervals, which is sufficient for our purpose, as shown in the following lemma. This yields the implementation of Algorithm 3 for the FILL procedure fulfilling the requirement previously given in Section 3.2.1.

▶ **Lemma 17.** *The data structures $P_v^{(i)}$'s and the block 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 construction. 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 blocks 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 18.** *Each execution of Algorithm 3 (the* FILL *operation) on an* unfilled *block $B_v^{(i)}$, 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;*
- *uses an additional $\mathcal{O}(\log n)$ time, $\mathcal{O}(1)$ random $\log n$-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 16, we finally obtain polylogarithimc resource bound for our implementation of RANDOM-NEIGHBOR.

▶ **Corollary 19.** *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;*
- *uses an additional $\mathcal{O}(\log^3 n)$ time, $\mathcal{O}(\log^2 n)$ random words, and $\mathcal{O}(1)$ additional space.*

Supporting Other Query Types along with `Random-Neighbor`

- VERTEX-PAIR(u,v): We simply need to make sure that Lemma 17 holds, so we first apply FILL(u, j) on block $B_u^{(j)}$ 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 14, the probability that a particular block is empty is a small constant. Then with high probability, there exists no $\omega(\log n)$ *consecutive* empty blocks $B_v^{(i)}$'s for any vertex v, and thus NEXT-NEIGHBOR only invokes up to $\mathcal{O}(\log n)$ calls to FILL.

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

▶ **Theorem 1.** *Given a random graph model defined by the probability matrix $\{p_{uv}\}_{u,v\in[n]}$, and assuming that we can compute the the quantities $\prod_{u=a}^{b}(1-p_{vu})$ and $\sum_{u=a}^{b} p_{vu}$ in $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ time, there exists an implementation for this model that supports local access through* RANDOM-NEIGHBOR*,* VERTEX-PAIR*, and* NEXT-NEIGHBOR *queries using $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ running time, additional space, and random bits 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 1 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 1.

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 random graphs. That is, as required by Theorem 1, 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 $\text{poly}(\log n)$ 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 2.** *There exists an algorithm that implements local access to a Erdös-Rényi $G(n, p)$ random graph, through* Vertex-Pair, Next-Neighbor, *and* Random-Neighbor *queries, while using $\mathcal{O}(\log^3 n)$ time, $\mathcal{O}(\log^3 n)$ random bits, 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 ($\log n$-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 block (where there can be up to $m + n$ blocks in total). Hence, our memory usage is nearly optimal for the $G(n, p)$ model:

▶ **Corollary 4.** *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 suffices to design a data structure that is 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 $j \in [r]$), where the vertices are assigned communities according to a given distribution 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 20.** *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 bits, and additional space per query. Further, this data structure may be implemented in such a way that requires no overhead for initialization.*

▶ **Corollary 3.** *There exists an algorithm that implements local access to a random graph from the n-vertex Stochastic Block Model with r randomly-assigned communities, through* Vertex-Pair, Next-Neighbor, *and* Random-Neighbor *queries, using $O(r\,\mathrm{poly}(\log n))$ time, random bits, and space per query w.h.p.*

We provide the full details of the construction in Section D. Our construction extends a similar implementation in the work of [26] 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[3] 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 $\langle |C_1|, |C_2|, \cdots, |C_r| \rangle$ sampled according to the multinomial distribution (for n samples from the 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 21 (proven in Section D).

▶ **Theorem 21.** *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 20. 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 (a, b), it uses Theorem 20 to sample the vector $\mathbf{C}(a, b) = \langle c_1, \ldots, c_r \rangle$, where c_k counts the number of variables in $\{X_a, \ldots, X_b\}$ 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 [32]). 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 20. ◀

Then, by embedding the information stored by the data structure into the state (as in the proof of Lemma 51), we obtain the desired Corollary 3.

4 Implementing Random Catalan Objects

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.

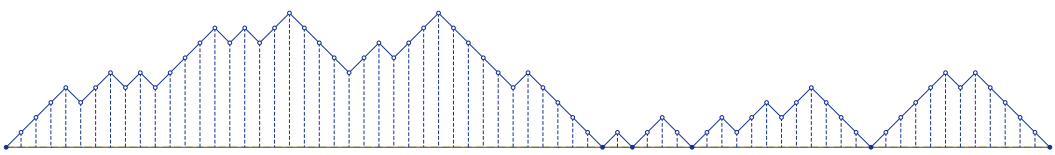

Figure 1 Simple Dyck path with $n = 35$ up and down steps.

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

One interpretation of Catalan objects is given by Dyck paths (Figure 1). A Dyck path is essentially a $2n$ step *balanced* one-dimensional walk with exactly n up and down steps. In Figure 1, each step moves one unit along the positive x-axis (time) and one unit up or down the positive y-axis (position). The prefix 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 62) is the nth 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 implement access other random Catalan objects such as rooted trees, and bracketed expressions. Specifically, we will want to answer the following queries:

- $\textsc{Direction}(t)$: Returns the value of the tth step in the Dyck path (whether the step is up or down).
- $\textsc{Height}(t)$: Returns the y-position of the path after t steps (the number of up steps minus the number of down steps among the first t steps). Since a $\textsc{Direction}(t)$ query can be simulated using the queries $\textsc{Height}(t)$ and $\textsc{Height}(t-1)$, we will not explicitly discuss the $\textsc{Direction}$ queries in what follows.
- $\textsc{First-Return}(t)$: If the $(t+1)^{th}$ step is upwards i.e. $\textsc{Height}(t+1) = \textsc{Height}(t) + 1$, it returns the smallest index $t' > t$ such that $\textsc{Height}(t') = \textsc{Height}(t)$. While it may not be clear why this query is important, it will be useful for querying bracketed expressions and random trees. (see Section 4.1).

4.1 Bijections to other Catalan objects

The $\textsc{Height}$ query is natural for Dyck paths, but the $\textsc{First-Return}$ 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 (Figure 2). The $\textsc{Height}$ query corresponds to asking for the nesting depth at a certain position in the word, and $\textsc{First-Return}(x)$ returns the position after the matched closing bracket for the step $(x \to x+1)$.

There is also a natural bijection between Dyck paths and ordered rooted trees (Figure 2), by viewing the Dyck path as a transcript of the tree's DFS traversal. 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. Thus, the $\textsc{Height}$ query returns the depth of a node. Also, since the Dyck path is a DFS transcript of the tree, a $\textsc{First-Return}$ query on the path can be used to find successive children of a tree node (each return produces the *next child*). For instance, in Figure 2, we can invoke $\textsc{First-Return}$ thrice starting at the first *yellow path position* to reveal the corresponding three children of the *yellow tree node*.

By definition, $\textsc{First-Return}(x)$ is meaningful only when the step from x to $x+1$ is upwards, i.e. when $\textsc{Height}(x+1) = \textsc{Height}(x)+1$. We can also implement a $\textsc{Reverse-First-Return}$ query, which is just a standard $\textsc{First-Return}$ query on the reversed Dyck path (consider a reversal of the green dashed arrows in Figure 2). The reversal implies that $\textsc{Reverse-First-Return}(x)$ is only meaningful when $\textsc{Height}(x-1) = \textsc{Height}(x) + 1$. In terms of bijections, $\textsc{Reverse-First-Return}$ is equivalent to finding a matching opening bracket in bracketed expressions, and a $\textsc{Previous-Child}$ query in rooted trees. We show how to implement this query in Section 4.4.7. In the case where the height at x is larger than both the heights at $x-1$ and $x+1$ (boxed nodes in Figure 2), there is no meaningful "first return" from the context of the bijections. Specifically, these nodes correspond to leaf nodes in rooted trees, or to a terminal nesting level in the bracket expression.

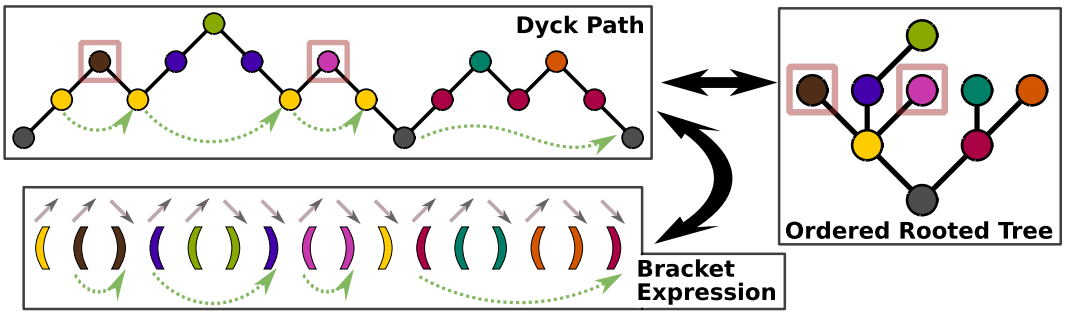

Figure 2 Bijections from Dyck paths to bracketed expressions and ordered rooted trees. Note that the color coded *positions* on the path correspond to tree nodes, and the path itself is a DFS traversal of the tree. The bijection to bracketed expressions proceeds by directly replacing up and down steps with opening and closing brackets respectively (we color the brackets with the *higher* endpoint of the corresponding step). Green dashed arrows show successive First-Return queries on the path. Using the bijection, this is equivalent to revealing three child sub-trees in order from left to right for the corresponding tree node, and also to finding matching brackets in bracketed expressions. First-Return is not defined for the red boxed nodes in either forward or reverse direction, since this position corresponds to a leaf node in the tree, or to a terminal nesting level in the bracket expression.

Moving forwards, we will focus on Dyck paths for the sake of simplicity.

4.2 Catalan Trapezoids and Generalized Dyck Paths

In order to implement local access to random Dyck paths, 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 3). Note that the case $k = 1$ corresponds to the standard description of Dyck paths, as mentioned previously (Figure 1).

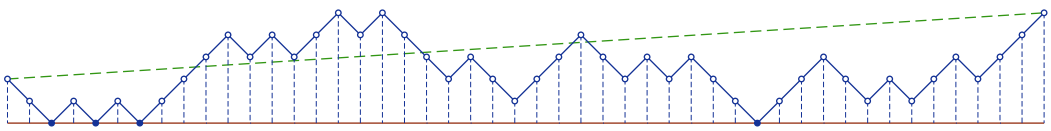

Figure 3 Generalized Dyck path with $U = 25$, $D = 22$ and $k = 3$. Note that the boundary is $k - 1 = 2$ units below the starting height.

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 [47]. We also use $\mathsf{C}_k(U, D)$ to denote the uniform distribution over $\mathbb{C}_k(n, m)$. Now, we state a result from [47] without proof:

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

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 $y_i = \text{HEIGHT}(x_i)$). We notice that these intervals can be generated 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 determine heights within such an interval, and in Section 4.4 we will move on to the more complicated FIRST-RETURN queries.

4.3 Implementing Height queries

We implement HEIGHT(t) by showing how to efficiently determine the height of the path at the midpoint of an existing interval $[x_i, x_{i+1}]$ (with corresponding endpoint heights y_i, y_{i+1}), which results in two sub-intervals that are half the size. Next, we extend this strategy to determine the heights of arbitrary positions by recursively sub-dividing the relevant interval (binary search). If the interval in question has odd length, we sample a single step from an endpoint, and proceed with a shortened even length interval. Sampling a single step is easy since there are only two outcomes (see proof of Theorem 28).

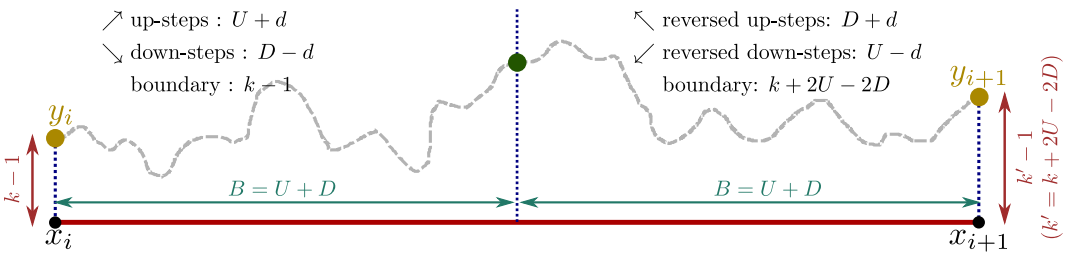

Figure 4 The $2B$-interval is split into two equal parts resulting in two separate Dyck problems. The green node (center) is the sampled height of the midpoint parameterized by the value of d. The path considered in both sub-intervals starts at a yellow node (left and right edges) and ends at the green 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.

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 $\mathsf{C}_k(2U, 2D)$. In order to make the analysis simpler, we have assumed that the number of up and down steps are both even. The case of sampling according to $\mathsf{C}_k(2U + 1, 2D + 1)$ works similarly with slightly different formulae. Without loss of generality, we assume that $D \leq U$; if this were not the case, we could simply flip the interval, swap the up and down steps, and modify the prefix constraint to $k' = k + 2U - 2D$ (Figure 4). This ensures that the overall path in the interval is non-decreasing in height, which will simpify our analysis.

We determine the height of the path $B = U + D$ steps into the interval at the midpoint (see Figure 4). This is equivalent to finding 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 (with the remaining $U - d$ up steps and $D + d$ down steps being assigned to the second half).

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

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 4). 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 cannot directly sample from this complicated distribution $\{p_d\}$. Instead, we use the rejection sampling strategy from Lemma 11. An important point to note is that in order to apply this lemma, we must be able to approximate the p_d values. However, we cannot naively use the formula from Equation 2, since the values of $\{S_{left}, S_{right}, S_{total}\}$ are too large to compute explicitly. Lemma 67 in Section F.3 shows how to indirectly compute the probabilty approximations. We also use the following lemma to bound the deviation of the path with high probability. A proof is presented in Section F.2.

▶ **Lemma 22.** *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 direct implication 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 $\widetilde{\mathcal{O}}(\sqrt{B})$ time algorithm for determining 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 with high probability, we do not hit it even if we choose a random *unconstrained* path.

▶ **Lemma 23.** *Given a Dyck path sampling problem of length B with U up, D down steps, and 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 $\mathsf{C}_\infty(U, D)$ is $\mathcal{O}(1/n^2)$-close in L_1 distance to the distribution of Dyck paths $\mathsf{C}_k(U + D)$.*

By Lemma 23, 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 [26] (see Lemma 54 in Section D). We also used this result earlier in the paper in order to find the community assignments in the Stochastic Block Model (Section 3.4.2).

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 23 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 22). We can now use Equation 2 to evaluate the probabilties $p_d = S_{left} \cdot S_{right}/S_{total}$. Recall that S_{left} and S_{right} are the number of possible paths in the left and right half of the interval, when exactly $U + d$ up steps are assigned to the first half, and S_{total} is the total number of possible paths in the interval.

We will invoke the rejection sampling technique (Lemma 11), 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 22). In order to perform rejection sampling, we also need good approximations of p_d, which is acheved by Lemma 67 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 $S_{right} < \binom{B}{U-d}$. These are true 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.4 to bound the value of S_{total}.

▶ **Lemma 24.** *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 to bound S_{left} and S_{right} (proofs in Section F.4).

▶ **Lemma 25.** $S_{left} \leq c_1 \frac{k \cdot \sqrt{\log n}}{\sqrt{B}} \cdot \binom{B}{D-d}$ *for some constant c_1.*

▶ **Lemma 26.** $S_{right} \leq c_2 \frac{k' \cdot \sqrt{logn}}{\sqrt{B}} \cdot \binom{B}{U-d}$ *for some constant c_2.*

Finally, we obtain a diferent bound on S_{total} for the *near boundary* case.

▶ **Lemma 27.** *When $kk' \leq 2U + 1$, $S_{total} \geq c_3 \frac{k \cdot k'}{B} \cdot \binom{2B}{2D}$ for some constant c_3.*

Multiplying these inequalities together allows us to bound $p_d = S_{left} \cdot S_{right}/S_{total} \leq \Theta(q_d \log n)$, implying $p_d/q_d \leq \Theta(\log n)$. Consequently, we can leverage Lemma 11 to obtain a sample from $\{p_d\}$ using $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ samples from $\{q_d\}$, thus determining the height of the Dyck path at the midpoint of the interval.

▶ **Theorem 28.** *Given an interval $[x_i, x_{i+1}]$ (and the associated endpoint heights y_i, y_{i+1}) in a Dyck path of length $2n$, with the guarantee that no position between x_i and x_{i+1} has been sampled yet, there is an algorithm that returns the height of the path at the midpoint of x_i and x_{i+1} (or next to the midpoint if $x_{i+1} - x_i$ is odd). Moreover, this algorithm only uses $\mathcal{O}(poly(\log n))$ resources.*

Proof. If $x_{i+1} - x_i$ is even, we can set $B = (x_{i+1} - x_i)/2$. Otherwise, we first sample a single step from x_i to $x_i + 1$, and then set $B = (x_{i+1} - x_i - 1)/2$. Since there are only two possibilities for a single step, we can explicitly approximate the corresponding probabilities using the strategy outlined in the proof of Lemma 67 (see Section F.3), and then sample accordingly. This allows us to apply the rejection sampling from Lemma 11 using $\{q_d\}$ to obtain samples from $\{p_d\}$ as defined above. ◀

▶ **Theorem 29.** *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 determined. Each newly determined 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 28, 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 determine the height of position x. ◀

4.4 Supporting "First Return" Queries

In this section, we discuss more complex queries to a Dyck path. Specifically, we implement FIRST-RETURN(x) to allow the user to query the next time the path returns to HEIGHT(x). The utility of this kind of query is illustrated through bijections to other random Catalan objects in Section 4.1. An important detail here is that if the first step from x to $x + 1$ is a down-step, there is no well defined FIRST-RETURN.

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 adversarial 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 determined, 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. HEIGHT$(x_i) = y_i$.

▶ **Invariant 30.** *For any interval $[x_i, x_{i+1}]$ where the heights of the endpoints have been determined to be y_i and y_{i+1}, and every other height in the interval has yet to determined, the section of the Dyck path between positions x_i and x_{i+1} is constrained to lie above $min(y_i, y_{i+1})$.*

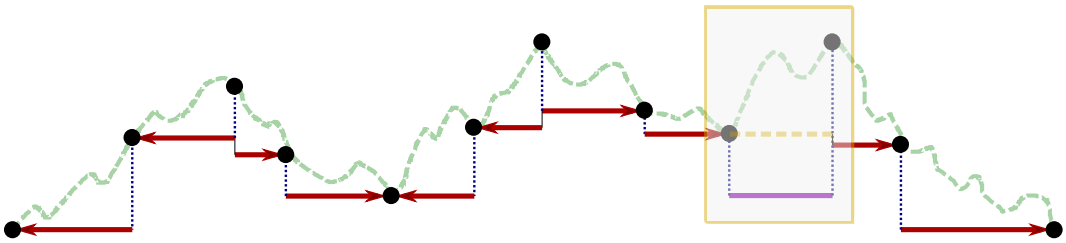

Figure 5 The set of intervals formed by a set of height samples. Each interval also has its own boundary constraint (red). Invariant 30 implies that each boundary must coincide with one of the interval endpoints. Note that the only interval which violates the invariant is the third last one (shown in yellow box).

How can one maintain such an invariant? After determining 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 determining the height of an additional point on either side (see Section 4.4.6 for details). Specifically, we use the following *two step strategy* to find the additional point for some violating interval $[x_i, x_{i+1}]$ (example violation in Figure 5):

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. Find an intermediate position x such that $x_i < x < x_{i+1}$ and $\textsc{Height}(x) = h$ (see Section 4.4.3).

The newly determined position x produces 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 6). Lemma 37 in Section 4.4.5 shows how this invariant can help us efficiently search for the interval containing the first return.

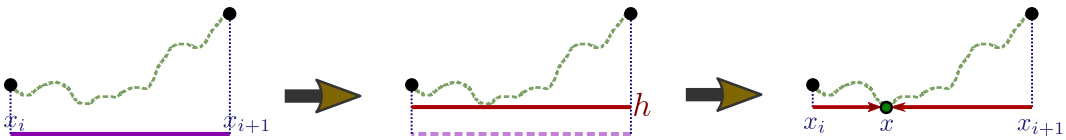

Figure 6 An interval $[x_i, x_{i+1}]$ that violates the invariant is "fixed" by first sampling the lowest height h achieved within the interval, and then generating a position $x \in [x_i, x_{i+1}]$ such that $Height(x) = h$.

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, D down steps, and a boundary that is $k - 1$ units below y_i.

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 follows (see Figure 7):

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

We define the current lower and upper boundaries as $k_{low} = k, k_{up} = 0$, and set $k_{mid} =$

Algorithm 4 Finding the Mandatory boundary.

```
function MANDATORY-BOUNDARY(U, D, k)
    k_low ← k
    k_up ← 0
    while k_low < k_up − 1
        k_mid ← ⌊(k_low + k_up)/2⌋
        P_total ← C_{k_low}(U, D) − C_{k_up}(U, D)
        P_{k_low}^{k_mid} ← C_{k_low}(U, D) − C_{k_mid}(U, D)
        with probability P_{k_low}^{k_mid} / P_total
            k_up ← k_mid
        else
            k_low ← k_mid
    return k_low
```

$(k_{low} + k_{up})/2$. 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.

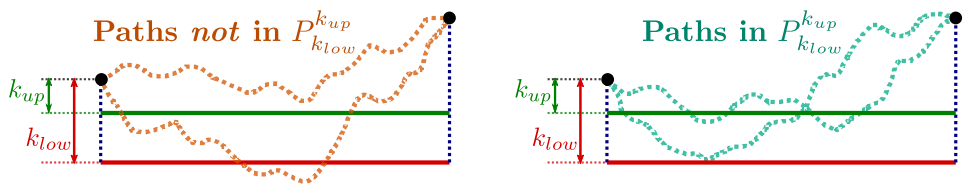

Figure 7 $P_{k_{low}}^{k_{up}}$ counts all the paths that dip at least k_{up} units below the starting point, but **do not** dip k_{low} units below the starting point. k_{mid} would lie halfway between these boundaries.

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

Now that we have a "*mandatory boundary*" k, we just need to generate a position $x \in [x_i, x_{i+1}]$ with $\textsc{Height}(x) = y_i - k + 1$, according to the uniform distribution over all paths that touch, but do not violate the boundary at k. In fact, we will do something stronger by determining 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.

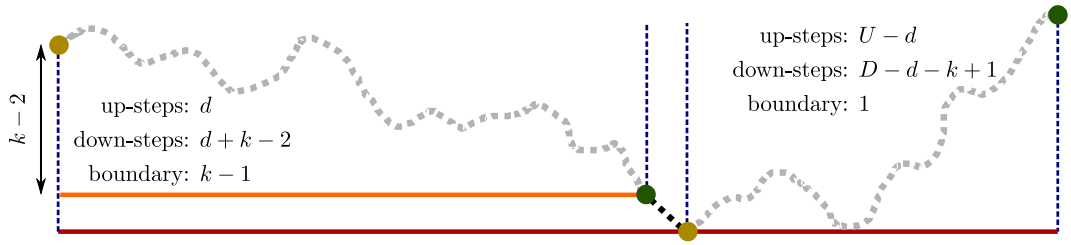

Figure 8 Zooming into the error in Figure 5. We generate a position x (yellow) on the boundary (red), such that the section of the path to the left of x never approaches the red boundary (it respects the orange boundary).

We will parameterize the position x the number of up-steps d between x_i and x (See Figure 8). 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 8), 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):

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

Our goal is to generate d from the distribution $\{p_d\}$ where $p_d = S_{left} \cdot S_{right}/S_{total}$. The application of Lemma 11 requires us to approximate $\{p_d\}$ with a "well behaved" $\{q_d\}$ (one whose CDF can be efficiently estimated). Since q_d only needs to approximate p_d up to $\text{poly}(\log n)$ factors, we focus on obtaining $\text{poly}(\log n)$ approximations to the path counts $\{S_{left}, S_{right}, S_{total}\}$. Using Equation 1, we obtain approximations for S_{left} and S_{right} (Lemma 31 and Lemma 32 with proofs in Section F.5).

▶ **Lemma 31.** *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 32.** *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, $r_{right}(d) = \mathcal{O}(\log^2 n)$.*

Our general strategy will be to integrate continuous versions of these approximations in order to obtain a CDF of some approximating distribution. Unfortunately, the 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 exponential term may be further simplified by taking a piecewise constant 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 start by considering 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 8), we get $U + D - 2d - k + 1 \geq 0$. Thus, we define the *"problematic"* set:

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

Clearly, we can bound the size of this set as $|\mathcal{R}| = \mathcal{O}(\log^4 n)$. An immediate consequence of Lemma 31 and Lemma 32 is the following.

▶ **Corollary 33.** *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 11). 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 33). 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 33), and $|\mathcal{R}| = \mathcal{O}(\log^4 n)$, we obtain the following lemma:

▶ **Lemma 34.** *Given the piecewise continuous function*

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

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 33, 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 34), each of which has a closed form integral (since $f(d)$ defined in Equation 5 has an integral).

▶ **Lemma 35.** *Given the function $\hat{q}_d$ defined in Lemma 34, 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* $\mathrm{poly}(\log n)$ *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 $d \in \mathcal{R}$ can be explicitly computed as $\int_d^{d+1} \hat{q}(\delta) = p_d$ (recall that we can compute p_d using Lemma 67 from Section F.3).

The more interesting pieces are over the range of δ where $\lfloor \delta \rfloor \notin \mathcal{R}$. Such a piece is given by a continuous range of values $[\delta_{min}, \delta_{max}] \subseteq [d_1, d_2+1]$, such that for any $\delta \in [\delta_{min}, \delta_{max}]$, the value of $\lfloor r(\lfloor \delta \rfloor) \rfloor$ is a constant E. We begin the calculation of $\hat{q}(\delta) = \mathcal{K} \cdot f(\delta) \cdot exp(-E)$, by first computing a $\left(1 \pm 1/n^3\right)$ multiplicative approximation to $\ln \mathcal{K} = (U+D)\ln 2 - \ln S_{total}$, using the strategy in Lemma 67 (Section F.3). Since $f(\delta)$ has a closed form integral (Equation 5), we can also compute $F = \int_{\delta_{min}}^{\delta_{max}} f(\delta)$. Using the fact that the exponent is constant over the range $[\delta_{min}, \delta_{max}]$, we can write the logarithm of the integral as:

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

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 $\mathcal{O}(1)$. This upper bound is a result of the fact that $\int \hat{q}(\delta) = \mathcal{O}(\sum p_d) = \mathcal{O}(1)$, 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 the integrals of all the $\mathcal{O}(\log^4 n)$ pieces together produces the final desired value of the integral. ◀

Now, we are finally ready to use Lemma 11 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 67.

▶ **Theorem 36.** *Given a sub-interval $[x_i, x_{i+1}]$ of a random Dyck path of length $2n$, such that the only determined 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 generates 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 30 to find the interval $\mathcal{I} = [x_k, x_{k+1}]$ containing $\textsc{First-Return}(x_i)$.

▶ **Lemma 37.** *Assuming that Invariant 30 holds, the interval $(x_{j-1}, x_j]$ containing $\textsc{First-Return}(x_i)$ is obtained by setting j to be either the smallest index $f > i$ such that $y_f \leq y_i$ or setting $j - 1 = i$.*

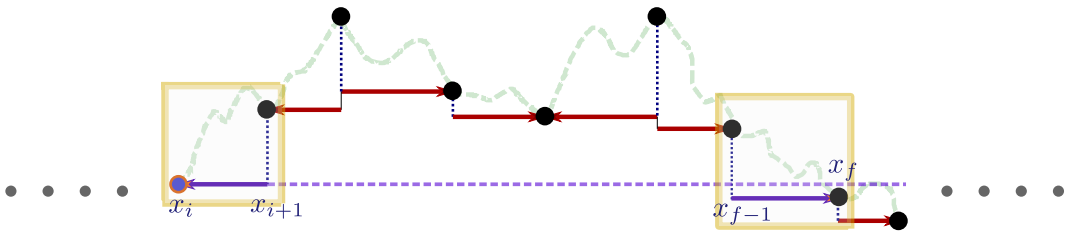

Figure 9 There are only two possible intervals (yellow boxes) that could contain $\textsc{First-Return}(x_i)$; either the interval adjacent to x_i, or the interval $[x_{f-1}, x_f]$, where f is the smallest index such that $y_f < x_i$.

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 $i+1 < k \leq f-1$. By the definition of y_f, we know that both y_k and y_{k-1} are strictly larger than y_i. Invariant 30 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 $y_f \leq y_i$. To this end, we define $M_{[a,b]}$ as the minimum determined height in the range of positions $[a, b]$.

One solution is to maintain a range tree $\mathbf{R}$ [14] 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 determined 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 $M_{[a_i,b_i]} = \min\left(M_{[a_i,b_i]}, y\right)$.
- $\textsc{Query}(a, b)$: 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 $\textsc{Query}(x_i, x_{f'}) \leq y_i$. Overall, this requires $\mathcal{O}(\log n)$ calls to $\textsc{Query}$, 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 $\textsc{Update}$ affecting a position $x \in [a, b]$. Since a call to $\textsc{Update}$ can create at most $\mathcal{O}(\log n)$ new nodes in $\mathbf{R}$, the additional space required for each $\textsc{Height}$ or $\textsc{First-Return}$ query is still bounded.

▶ **Theorem 38.** *There is an algorithm using $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ resources per query that provides access to a random Dyck path of length $2n$, by answering queries of the form $\textsc{First-Return}(x_i)$ with the correctly sampled smallest position $x' > x_i$, where the Dyck path first returns to $\textsc{Height}(x_i)$.*

Proof. In order to make the presentation simpler, we ensure that the next determined position after x_i is x_i+1 (in other words $x_{i+1} = x_i+1$). This can be done by invoking $\textsc{Height}(x_i+1)$, if needed. If $\textsc{Height}(x_i+1) = y_{i+1} < y_i$, we can terminate because $\textsc{First-Return}(x_i)$ 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 $y_f \leq y_i$, by using the range tree data structure described above. Since $\textsc{Height}(x_{f-1}) > y_i \leq \textsc{Height}(x_f)$ by definition, the interval $(x_{f-1}, x_f]$ must contain at least one position at height y_i. We determine the height at the midpoint of this interval, and then fix the potential violation of Invariant 30 by finding another point along with its height. This essentially breaks up the interval $[x_{f-1}, x_f]$ into $\mathcal{O}(1)$ sub-intervals, each at most half the size of the original. Based on the newly revealed heights, 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, reducing the size of the intervals in consideration, until the position of the first return is revealed. ◀

4.4.6 Maintaining `Height` Queries under Invariant 30

Finally, we show that the boundary constraints introduced in order to maintain Invariant 30 do not interfere with the implementation of $\textsc{Height}$ 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 query $\textsc{Height}(x)$, 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 30, 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 algorithm for determining the height at the midpoint x_{mid} of $[x_i, x_{i+1}]$ 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. The second step is to re-establish Invariant 30 in the newly created sub-interval $[x_{mid}, x_{i+1}]$[4], using the two step strategy from Section 4.4.1. This involves determining the height of one additional point $x' \in [x_{mid}, x_{i+1}]$. Finally, we can continue with the height sampling algorithm from Theorem 29, by recursively halving one of the newly created intervals (the one containing x). Figure 10 illustrates the aforementioned three steps.

Hence, we can combine results from Theorem 29 and Theorem 38 to obtain the following:

▶ **Theorem 6.** *There is an algorithm using $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ resources per query that provides access to a uniformly random Dyck path of length $2n$ by implementing the following queries:*

- $\textsc{Height}(t)$ *returns the position of the walk after t steps.*
- $\textsc{First-Return}(t)$*: If* $\textsc{Height}(t+1) > \textsc{Height}(t)$*, then* $\textsc{First-Return}(t)$ *returns the smallest index t', such that $t' > t$ and* $\textsc{Height}(t') = \textsc{Height}(t)$ *(i.e. the first time the Dyck path returns to the same height). Otherwise, if* $\textsc{Height}(t+1) < \textsc{Height}(t)$*, then* $\textsc{First-Return}(t)$ *is not defined.*

[4]Note that the invariant is not broken for the $[x_i, x_{mid}]$ sub-interval since $y_i + 1$ was the original boundary constraint. In general, the invariant will be automatically preserved for one of the sub-intervals; the side corresponding to $\min(y_i, y_{i+1})$.

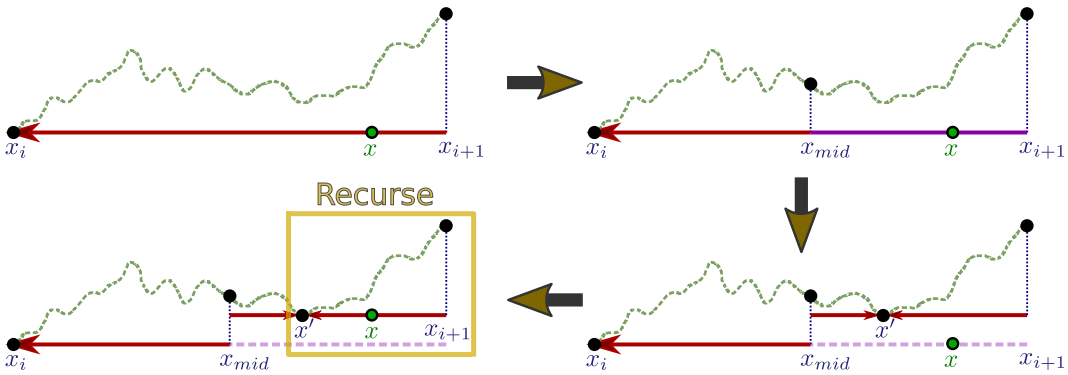

Figure 10 Performing HEIGHT queries while maintaining Invariant 30. The first step is to determine the height at the midpoint x_{mid} of an existing interval $[x_i, x_{i+1}]$ that contains x. Next, we re-establish Invariant 30 by sampling an additional point x' within the violating interval Finally, we recurse on the appropriate sub-interval (yellow box).

4.4.7 Reverse-First-Return Queries

For the sake of completeness, we also sketch how to implement a REVERSE-FIRST-RETURN query, which is just a standard FIRST-RETURN query on the reversed Dyck path. This type of query was motivated in Section 4.1, and can be applied to positions x where $\textsc{Height}(x-1) = \textsc{Height}(x) + 1$.

Note that Invariant 30 remains unchanged as it is symmetric to reversal of the Dyck path. Specifically, the invariant on a particular interval does not change if we swap the endpoints of the interval. First, we can find the correct interval containing $\textsc{Reverse-First-Return}(x)$, by using an analog of Lemma 37.

▶ **Lemma 39.** *Assuming Invariant 30, the interval $[x_j, x_{j+1})$ containing* REVERSE-FIRST-RETURN(x_i) *is obtained by setting j to be either the largest index $f < i$ such that $y_f \leq y_i$ or setting $j + 1 = i$.*

Since the range tree defined in Section 4.4.5 allows us to query the minimum in arbitrary contiguous ranges, we can also use it to find the appropriate f for Lemma 39 through binary search. Once we have found an interval that is known to contain the REVERSE-FIRST-RETURN, we can follow the strategy from the proof of Theorem 38; recursively sub-dividing the relevant interval and finding the newly created sub-interval that contains the return, until we converge on the correct return position.

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 sub-linear time algorithm to provide local access to a uniformly random valid q-coloring of an input graph. Specifically, we implement $\textsc{Color}(v)$, 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 $\textsc{All-Neighbors}(v)$ 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 10. This model is essentially a generalization of *Local Computation Algorithms* [48]. Given a computation problem with input x and a set of valid outputs $F(x)$, LCAs provide query access to some $y \in F(x)$ using only a sub-linear number of probes to x. This makes it feasible to consider sub-linear algorithms for problems where the output size is super-linear. Our model has the additional restriction that the output must be drawn uniformly at random from $F(x)$, rather than just being an arbitrary member. Since the number of possible outputs ($|F(x)|$) may be exponential, it is not possible to encode all the requisite random bits in sub-linear space. Therefore, we use a source of public randomness $\mathbf{R}$.

Sequential Algorithm for Random Coloring

We consider graphs with max degree Δ, and $q = \Theta(\Delta)$, since this is the regime where this problem is feasible [22]. In the sequential setting, [22] 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 $\mathbf{X}^t_v$. 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. $c \notin \{\mathbf{X}^t_u : u \in \Gamma(v)\}$, then the vertex is recolored i.e. $\mathbf{X}^{t+1}_v \leftarrow c$. 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.

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 [20]. 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 kth epoch by $\mathbf{X}^k$. In the kth 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 $\mathbf{X}^k_v$ to c_v, if and only if for all neighbors w of v, $\mathbf{X}^{k-1}_w \neq c_v$ 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 [20] as a distributed algorithm for sampling a random coloring.[5] 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, [20] uses the standard technique of *path coupling*, introduced in [8]. 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

[5] Note that [20] 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 $\gamma = 1$.

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 [20].

▶ **Lemma 40.** *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 41.** *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 [8] uses a coupling on adjacent states to bound the mixing time.

▶ **Lemma 42** (Simplified Path Coupling from [8])**.** *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, the mixing time $\tau_{mix}(\epsilon) = \mathcal{O}(\ln(n\epsilon^{-1}) / \ln \beta^{-1})$.*

▶ **Corollary 43.** *If $q \geq 9\Delta$, then the chain is mixed after $\tau_{mix}(\epsilon) = 3\left(\ln n + \ln(\frac{1}{\epsilon})\right)$ epochs.*

5.1.1 Naive Reduction from Distributed Algorithms

Using the technique of Parnas and Ron [46], one can modify a distributed algorithm for a graph problem to construct a Local Computation Algorithm for the same problem. Specifically, given a k-round distributed algorithm on a network of max degree Δ, we can simulate the behaviour and outcome of a single node v by simulating the full algorithm on the k-neighborhood of v. The simulation only requires us to probe this k-neighborhood, which contains $\mathcal{O}(\Delta^k)$ nodes. However, the aforementioned distributed algorithm for Glauber Dynamics used $\mathcal{O}(\log n)$ rounds, implying a probe complexity of $\Delta^{\mathcal{O}(\log n)}$, which is super-linear. We show how to reduce the probe complexity by appropriately pruning the k-neighborhood.

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 sub-linear 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 tth epoch. Since each of these $\mathbf{C}^t_v$ values are independent uniform samples from $[q]$, instances of our algorithm will be able to access them in a consistent manner using the public random bits $\mathbf{R}$.

We also use $\mathbf{X}^t$ to denote the final vector of vertex colors at the end of the tth 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 tth 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 tth 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 $\mathbf{C}^t_v$ can simply be read off the public random tape $\mathbf{R}$, it is not clear how we can determine the $\boldsymbol{\chi}^t_v$ values efficiently. Computing $\mathbf{X}^t_v$ is quite simple 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, but we have to be able to access the initial colors of requisite vertices in sub-linear time. One option is to simply assume that we have oracle access to an arbitrary valid coloring. We can also invoke a result from [10] that provides a *Local Computation Algorithm* for $(\Delta+1)$-coloring of a graph. Specifically, they provide local access to the color of any vertex using $\mathcal{O}(\Delta^{\mathcal{O}(1)}\log n)$ queries to the underlying graph, such that the returned colors are consistent with some valid coloring. Integrating their routine into our algorithm incurs a multiplicative $\text{poly}(\Delta)$ overhead for each query.

Another option is to (uniformly) randomly intialize the colors of the vertex independently of its neighbors. Although the initial coloring may be invalid, one can show that with high probability, the final coloring after running the Markov Chain is valid. The intuitive reasoning is that each vertex attempts to recolor itself $\mathcal{O}(\log n)$ times, and each attempt suceeds with constant probabiltity at least $1-1/\alpha$ (Lemma 44). Furthermore, we can union bound to claim that *all* the vertices get re-colored at least once with high probability. After a vertex is re-colored once, it can no longer be in conflict with any of its neighbor's colors, and therefore we obtain a final coloring that is valid. For the sake of simplicity, we will assume that our algorithm can access the initial colors of any vertex v through a public $\mathbf{C}_v^0$.

5.2.2 Naive Coloring Implementations

Our general strategy to determine $\boldsymbol{\chi}_v^t$ will be to check for all neighbors w of v, whether w causes a conflict with v's proposed color in the tth 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 $\boldsymbol{\chi}_w^{t'} = 1$) is found. At this point, if $\mathbf{C}_w^{t'} = \mathbf{C}_v^t$, 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 $\mathbf{C}_v^t$ 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 sub-linear. 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 Sub-linear 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.

Algorithm 5 shows our final procedure for generating $\boldsymbol{\chi}_v^t$, where $c = \mathbf{C}_v^t$ is v's proposal at epoch t. We iterate through all neighbors w of v, checking for conflicts (Line 3). The condition $c \neq \mathbf{C}_w^t$ can be checked by reading $\mathbf{C}_w^t$ off the public random tape (Line 4). If no conflict is seen, we proceed to check whether $c \neq \mathbf{X}_w^{t-1}$.

Algorithm 5 Checking if proposal is accepted.

```
1:  procedure ACCEPTED(v, t)
2:      c ← C_v^t                          ▷ Find the current proposal using the public random bits
3:      for w ← Γ(v)                       ▷ Iterate through the neighbors, checking for conflicts
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, 0]   ▷ Check for conflict with neighbor's prior state
7:              if C_w^{t'} = c and ACCEPTED(w, t')   ▷ Potential conflict with neighbor's color
8:                  overwritten ← false    ▷ Check if color c is overwritten by a future proposal
9:                  for t̃ ← [t'+1, t'+2, t'+3, ··· , 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
```

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. 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; this implies 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 $\boldsymbol{\chi}_v^t = 1$.

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 44.** *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 $\{\mathbf{C}_w^t, \mathbf{X}_w^{t-1} | w \in \Gamma(v)\}$. Since there are $2\alpha\Delta$ colors, the rejection probability is at most $1/\alpha$. ◀

▶ **Definition 45.** *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 46.** *We define $W_{t'}^t$ 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 $W_{t'}^t$ (using $\leq$ to denote stochastic dominance).

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

The above Equation 7 conservatively assumes that the call to ACCEPTED$(v, t'+1)$ in line 10 is always invoked (resulting in $T_{t'+1}$ invocations of ACCEPTED). However, the next call to ACCEPTED$(v, t'+2)$ occurs only if the previous one was not accepted, which happens with probability $\leq 1/\alpha$ (Lemma 44). This produces the $\mathcal{B}(1/\alpha) \cdot T_{t'+2}$ term in the expression. In general, 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 $1/\alpha^{i-1}$.

Next, we prove our main lemma that bounds the number of recursive calls to ACCEPTED.

▶ **Lemma 47.** *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 $\chi_v^t =$* ACCEPTED(v, t) *is $\mathbb{E}[T_t] = \mathcal{O}\left(e^{1.02t/\alpha}\right)$.*

Proof. We start by constructing a recurrence for the expected number of 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 (which happens with probability $1/q$ independently for each trial), there is a recursive call to ACCEPTED(w, t'), which in turn results in $T_{t'}$ recursive calls to ACCEPTED. If we find that $\chi_w^{t'} = 1$ (i.e. w was colored to c at epoch t'), we will need to proceed to check whether the color was subsequently overwritten, which requires an additional $W_{t'}^t$ calls to ACCEPTED. Summing up over all neighbors and epochs, we obtain the following bound:

$$T_t \leq \sum_{w \in \Gamma(v)} \sum_{t'=1}^{t} \mathbb{P}\left[\mathbf{C}_w^{t'} = c\right] \cdot \left[T_{t'} + W_{t'}^t\right] \tag{8}$$

$$\leq \Delta \cdot \sum_{t'=1}^{t} \mathcal{B}\left(\frac{1}{q}\right) \cdot \left[T_{t'} + T_{t'+1} + \mathcal{B}\left(\frac{1}{\alpha}\right) \cdot T_{t'+2} + \mathcal{B}\left(\frac{1}{\alpha^2}\right) \cdot T_{t'+3} + \cdots \right.$$
$$\left. \cdots + \mathcal{B}\left(\frac{1}{\alpha^{t-t'-2}}\right) \cdot T_{t-1}\right] \tag{9}$$

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

In the last step, we just grouped together all the terms corresponding to the same epoch (note that we include additional terms since it's an upper bound). Using Lemma 44 and the fact that $\mathbb{P}[\mathbf{C}_w^{t'} = c]$ 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 7.** *Given neighborhood query access to a graph G with n nodes, maximum degree Δ, and $q = 2\alpha\Delta \geq 9\Delta$ colors, we can generate the color of any given node from a distribution of color assignments that is ϵ-close (in L_1 distance) to the uniform distribution over all valid colorings of G, in a consistent manner, using only $\mathcal{O}((n/\epsilon)^{3.06/\alpha}\Delta \log(n/\epsilon))$ time, probes, and public random bits per query.*

Proof. Since $q \geq 9\Delta$, we can use Corollary 43 to obtain $\tau_{mix}(\epsilon) \leq 3(\ln n + \ln 1/\epsilon)$. Also, since $\alpha > 4.5$, we can invoke Lemma 47 to conclude that the number of calls to ACCEPTED is $\mathcal{O}(n^{3.06/\alpha}\epsilon^{-3.06/\alpha})$. Finally, we note that each call to ACCEPTED(v, t) potentially reads $\mathcal{O}(t\Delta)$ color proposals from the public random tape, while iterating through all $\leq \Delta$ neighbors of v in all t epochs. Since $t \leq 3\ln(n/\epsilon)$, this implies that the algorithm uses $\mathcal{O}((n/\epsilon)^{3.06/\alpha}\Delta \log(n/\epsilon))$ time and random bits, which is sub-linear for $\alpha > 3.06$. ◀

6 Open Problems

There are many interesting directions to pursue in this area. Below, we provide a few examples of random objects that may admit local access implementations.

Small Description Size

- Provide a local access implementation of degree queries for undirected random graphs, even for $G(n, p)$. How about i^{th} neighbor queries?
- For simple models such as $G(n, p)$, provide a local access implementation of a RANDOM-TRIANGLE(v) query, that returns a uniformly random triangle containing vertex v.
- Provide a memory-less local access implementation of basic queries for undirected random graphs.
- Given an ordered graph such as a lattice, provide an implementation to locally access a random perfect matching. Interesting special cases of this problem include random domino and lozenge tilings.

Huge Description Size

- 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.
- Improve the local access implementation for sampling the color of a specified vertex in a random q-coloring of G, by supporting smaller values of q (smaller than 9Δ). 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 query access to an input 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. This may be feasible in certain restricted classes of graphs.
- Given query access to an input DNF formula, provide an implementation to access the truth value of a single variable in a uniformly random satisfying assignment.

References

1. Emmanuel Abbe. Community detection and stochastic block models: recent developments. *The Journal of Machine Learning Research*, 18(1):6446–6531, 2017.
2. Emmanuel Abbe, Afonso S Bandeira, and Georgina Hall. Exact recovery in the stochastic block model. *IEEE Transactions on Information Theory*, 62(1):471–487, 2016.
3. Emmanuel Abbe and Colin Sandon. Community detection in general stochastic block models: Fundamental limits and efficient algorithms for recovery. In *Foundations of Computer Science (FOCS), 2015 IEEE 56th Annual Symposium on*, pages 670–688. IEEE, 2015.
4. Maksudul Alam and Maleq Khan. Parallel algorithms for generating random networks with given degree sequences. *International Journal of Parallel Programming*, 45(1):109–127, 2017.
5. 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.
6. Danielle Smith Bassett and ED Bullmore. Small-world brain networks. *The neuroscientist*, 12(6):512–523, 2006.
7. Vladimir Batagelj and Ulrik Brandes. Efficient generation of large random networks. *Physical Review E*, 71(3):036113, 2005.
8. 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.
9. 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.
10. Yi-Jun Chang, Manuela Fischer, Mohsen Ghaffari, Jara Uitto, and Yufan Zheng. The Complexity of $(\Delta+1)$ 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.
11. Jingchun Chen and Bo Yuan. Detecting functional modules in the yeast protein–protein interaction network. *Bioinformatics*, 22(18):2283–2290, 2006.
12. 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.
13. 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.
14. Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Cheong Schwarzkopf. Computational geometry. In *Computational geometry*, pages 105–109. Springer, 2000.
15. Peter Sheridan Dodds, Roby Muhamad, and Duncan J Watts. An experimental study of search in global social networks. *science*, 301(5634):827–829, 2003.
16. David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.
17. Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

18 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. doi:10.4230/LIPIcs.ICALP.2017.6.

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

20 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.

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

22 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.

23 Anna C Gilbert, Sudipto Guha, Piotr Indyk, Yannis Kotidis, Sivaramakrishnan Muthukrishnan, and Martin J Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 389–398. ACM, 2002.

24 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.

25 Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986. doi:10.1145/6490.6503.

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

27 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.

28 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.

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

30 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.

31 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.

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

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

34 Michael Luby and Charles Rackoff. How to Construct Pseudorandom Permutations from Pseudorandom Functions. *SIAM J. Comput.*, 17(2):373–386, 1988. doi:10.1137/0217022.

35 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. doi:10.1007/978-3-642-31594-7_55.

36 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.

37 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.

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

39 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*, 2003. `arXiv:cond-mat/0312028`.

40 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.

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

42 Moni Naor and Omer Reingold. On the Construction of Pseudo-Random Permutations: Luby-Rackoff Revisited. *IACR Cryptology ePrint Archive*, 1996:11, 1996. URL: `http://eprint.iacr.org/1996/011`.

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

44 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.

45 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.

46 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.

47 Shlomi Reuveni. CATALAN'S TRAPEZOIDS. *Probability in the Engineering and Informational Sciences*, 28(03):353–361, 2014.

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

49 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.

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

51 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.

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

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

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

55 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 Blocks

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 48.** *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.[6]

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

Then, the upper bound on the number of failed iterations of a call NEXT-NEIGHBOR(v) is given by the maximum number of cells $Y_{v,u'} = 1$ of $u' \in (\mathbf{last}[v], u)$, over any $u \in (\mathbf{last}[v], n]$ satisfying $X_{v,\mathbf{last}[v]+1} = \cdots = X_{vu} = 0$. 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 $(v, \mathrm{last}[v])$

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 48. ◀

A.2 Supporting Vertex-Pair Queries

We extend our implementation (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 $\mathbf{last}$ and P_v's cannot record arbitrary modifications to $\mathbf{A}$.

Observe that if we were to apply the trivial implementation of VERTEX-PAIR, then by Lemma 12, **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.[7] 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 49.** *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 19).

▶ **Corollary 50.** *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 48.*

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 implementation is randomized, but the agent interacting with the implementation may choose their queries arbitrarily (or adversarially): our proof must handle any sequence of random choices the implementation makes, and any sequence of queries the agent may make.

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

Observe that the distribution of the graphs constructed by our implementation 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 51.** *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 blocks $\mathcal{B} = \langle B_{v_1}^{(u_1)}, B_{v_2}^{(u_2)}, \ldots \rangle$, where each $B_{v_i}^{(u_i)}$ specifies the unfilled block in which any query instigates a FILL function call. Define an *internal state* of our implementation as the triplet $s = (k, u, \mathbf{A})$, representing that the algorithm is currently processing the kth 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 block) 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 implementation 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 implementation 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 $s \ne t_{\mathbf{A}}$, 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 $\pi_i = \langle s_0^\pi, \ldots, s_i^\pi \rangle$ of π. Observe that for $i \geq 1$,

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

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

▶ **Lemma 13.** *With high probability, the number of neighbors in every block, $|\Gamma^{(i)}(v)|$, is at most $\mathcal{O}(\log n)$.*

Proof. Fix a block $B_v^{(i)}$, and consider the Bernoulli RVs $\{X_{vu}\}_{u \in B_v^{(i)}}$. The expected number of neighbors in this block 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,

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

for any constant $c > 0$. ◀

▶ **Lemma 14.** *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 blocks $\{B_v^{(i)}\}_{i \in [|\mathbf{B}_v|]}$ 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:

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

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

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

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

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 implementations 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

Algorithm 6 Alternate implementation.

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

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.

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 $\mathsf{ExactF}(p, |T|)$. 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 Fth 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 ith node of the jth 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 Fth 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 ith 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 ith smallest element of T. This is accomplished 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 6, 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 6, 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:

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

This definition differs from that of Q in Section A.2 in two aspects. First, we ensure that each $\mathbf{A}[v][u] = 0$ is recorded by either **last** (via Lemma 12) 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 48 and Corollary 50).

▶ **Theorem 52.** *Consider the Erdös-Rényi $G(n,p)$ model. For* NEXT-NEIGHBOR *queries only, Algorithm 6 is an implementation 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 6 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 $F \sim$ 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 53.** *Consider the Stochastic Block model with randomly-assigned communities. For* NEXT-NEIGHBOR *queries only, Algorithm 6 is an implementation that answers each query using $O(r\,\mathrm{poly}(\log n))$ time, random words, and additional space per query. For* NEXT-NEIGHBOR *and* VERTEX-PAIR *queries, Algorithm 6 answers each query using $O(r\,\mathrm{poly}(\log n))$ amortized time, $O(r\,\mathrm{poly}(\log n))$ random words, and $O(r\,\mathrm{poly}(\log n))$ additional space per query additional space, and one N-bit random word.*

D Sampling from the Multivariate Hypergeometric Distribution

Consider the following random experiment. Suppose that we have an urn containing $B \le n$ 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 [26].

▶ **Lemma 54.** *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 55.** *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 Lemma 54.
  - 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 21.** *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 51. ◀

▶ **Theorem 56.** *Given B marbles of r different colors in $[r]$, such that there are C_i marbles of color i and a parameter $k \le r$, 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 $r \le B$). So, the overall runtime is $\mathcal{O}(\mathrm{poly}(\log B))$. ◀

E Local-Access Implementations for Random Directed Graphs

In this section, we consider Kleinberg's Small-World model [31, 37] where the probability that a *directed* edge (u, v) exists is $\min\{c/(\mathtt{DIST}(u, v))^2, 1\}$. Here, $\mathtt{DIST}(u, v)$ 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 $\mathrm{poly}(\log n)$ resources. This returns the entire list of out-neighbors of v.

E.1 Implementation 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: $\Gamma_k^v = \{u : \mathtt{DIST}(v, u) = k\}$ 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 $\mathrm{poly}(\log n)$ resources.

Nonetheless, internally in our implementation, we generate 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 compute a distance d, such that the next neighbor closest to v is at distance d. We maintain $\mathbf{last}[v]$ to be the last computed distance. In the second phase, we

generate all neighbors of v at distance d, under the assumption that there must be at least one such neighbor. For simplicity, we generate these neighbors as if there are *full* $4d$ vertices at distance d from v: some generated neighbors may lie outside our $\sqrt{n} \times \sqrt{n}$ grid, which are simply discarded. As the running time of our implementation is proportional to the number of implementation neighbors, then by the bound on the number of neighbors, this assumption does not asymptotically worsen the performance of the implementation.

E.1.1 Phase 1: Generate 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 generate 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 generate the next distance, we perform a binary search: we must evaluate the CDF of $\mathsf{D}(a)$. The CDF is given by $\mathbb{P}[D \leq d]$ where $D \sim \mathsf{D}(a)$, the probability that there is *some* neighbor at distance at most d. As usual, we compute the probability of the negation: there is *no* neighbor at distance at most d. Recall that each distance i has exactly $|\Gamma_i^v| = 4i$ vertices, and the probability of a vertex $u \in \Gamma_i^v$ is not a neighbor is exactly $1 - 1/i^2$. So, the probability that there is no neighbor at distance i is $(1 - 1/i^2)^{4i}$. Thus, for $D \sim \mathsf{D}(a)$ and $d \leq 2(\sqrt{n} - 1)$,

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

where the product enjoys telescoping as the denominator $(i^2)^{4i}$ cancels with $(i^2)^{4(i-1)}$ and $(i^2)^{4(i+1)}$ in the numerators of the previous and the next term, respectively. This gives us a closed form for the CDF, which we can compute with 2^{-N} additive error in constant time (by our computation model assumption). Thus, we may generate the distance $D \sim \mathsf{D}(a)$ using $O(\log n)$ 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 generate 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 generate 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 generate 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 generate 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 generated 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 57.** *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 $1/(\textsc{dist}(u, v))^2$ where $\textsc{dist}$ denote the Manhattan distance, using $O(\log^2 n)$ time and random $\log n$-bit words per $\textsc{All-Neighbors}$ query with high probability.*

E.2 Implementation for $c \neq 1$

Observe that to support different values of c in the probability function $c/(\textsc{dist}(u, v))^2$, 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 $\frac{1}{\mathrm{poly}(n)}$ in the L_1-distance.

E.2.1 Case $c < 1$

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

▶ **Lemma 58.** *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 simulate the Bernoulli trials. In addition to each Bernoulli variable X_i with bias p_i, we generate 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 59.** *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 $\log n$-bit words per $\textsc{All-Neighbors}$ 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 60.** *Let $k > 1$ be a constant integer. Given a process $\mathcal{P}$ outputting the indices of successful Bernoulli trials with bias $\langle p_i \rangle$, there exists a process $\mathcal{P}^c$ outputting the indices of successful Bernoulli trials with bias $\langle \min\{c \cdot p_i, 1\} \rangle$ where $c > 1$ and $c \cdot p_i \leq 1 - \frac{1}{k}$ for every i, using one additional N-bit word overhead for each answer of $k \cdot c$ independent copies of $\mathcal{P}$.*

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

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

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

In Phase 1, we may apply this reduction only when the condition $c \cdot p_i \leq 1 - \frac{1}{k}$ is satisfied. For lower value of $p_i = 1/D^2$, namely for distance $D < \sqrt{c/(1 - 1/k)} = O(\sqrt{c})$, we may afford to generate 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 61.** *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 words per* ALL-NEIGHBORS *query with high probability.*

F Omitted Proofs for the Dyck Path Implementation

▶ **Theorem 62.** *There are $\frac{1}{n+1}\binom{2n}{n}$ Dyck paths for length $2n$ (construction from [53]).*

Proof from [53]. 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 63.** *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 64.**

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

Proof from [52]. We consider the ratio: $R = \binom{n}{k}/\binom{n}{n/2}$:

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

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

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

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

◀

F.2 Dyck Path Boundaries and Deviations

▶ **Lemma 65.** *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:

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

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

▶ **Corollary 66.** *With high probability, every contiguous sub-path of length $2B$ (with U up and D down steps such that $U + D = 2B$) in the random walk satisfies the property that $|B - U|$ and $|B - D|$ are upper bounded by $c\sqrt{B\log n}$ w.h.p. $1 - 1/n^2$ (for some constant c).*

Proof. We can simply apply Lemma 65 and union bound over all n^2 possible contiguous sub-paths. ◀

▶ **Lemma 22.** *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 62, 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 66 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 23.** *Given a Dyck path sampling problem of length B with U up, D down steps, and 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 $\mathsf{C}_\infty(U, D)$ is $\mathcal{O}(1/n^2)$-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, $\mathcal{D} \subseteq \mathcal{R}$. Since the random walk/sequence distribution is uniform on $\mathcal{R}$, and by Corollary 66 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 the uniform distributions on $\mathcal{D}$ and $\mathcal{R}$ is $\mathcal{O}(1/n^2)$. ◀

F.3 Estimating the Sampling Probabilities

▶ **Lemma 67.** *Given a Dyck 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 $\ln(ab) = l_a + l_b$. 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 $\widetilde{l_a} = l_a \cdot (1 \pm \mathcal{O}(1/n^c))$ and $\widetilde{l_b} = l_b \cdot (1 \pm \mathcal{O}(1/n^c))$. In case $\widetilde{l_b} - \widetilde{l_a} < -c \ln n \implies b/a < 1/n^c$, we approximate $\ln(a + b)$ by $\ln a$ since $\ln(1 + b/a) = \mathcal{O}(1/n^c)$ in this case.

Otherwise, we notice that $\max(a, b) = \mathcal{O}\left((n!)^2\right) \implies \max(l_a, l_b) = o(n^3) \implies l_a - l_b = o(n^3)$. This is true because if we write out the expression for $p_d = S_{left} \cdot S_{right}/S_{total}$ in terms of sums and products of factorials, the largest possible value will be a product of at most two terms that are present in the numerator/denominator of a binomial term (see the expression

for generalized Catalan numbers in Equation 1). Hence, the claim $\max(a,b) = \mathcal{O}\left((n!)^2\right)$ follows from the fact that the numerators/denominators of the relevant binomial coefficient in Equation 1 are at most $n!$. Using this fact, we obtain the following:

$$1+e^{\widetilde{l}_b-\widetilde{l}_a} = 1+\frac{b}{a}\cdot e^{\mathcal{O}\left(\frac{l_b-l_a}{n^c}\right)} = 1+\frac{b}{a}\cdot\left(1\pm\mathcal{O}\left(\frac{1}{n^{c-3}}\right)\right) = \left(1+\frac{b}{a}\right)\cdot\left(1\pm\mathcal{O}\left(\frac{1}{n^{c-3}}\right)\right)$$

In other words, the value of c decreases every time we have a sum 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, cewhich allows us to obtain the desired $1\pm 1/n^3$ multiplicative 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 $\widetilde{p}_d = p_d\cdot e^{-\mathcal{O}(\ln n/n^3)} = p_d\left(1\pm\mathcal{O}(1/n^2)\right)$. ◀

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

▶ **Lemma 68.** *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 25.** *$S_{left}\leq c_1\frac{k\cdot\sqrt{\log n}}{\sqrt{B}}\cdot\binom{B}{D-d}$ for some constant c_1.*

Proof. This involves some simple manipulations.

$$\begin{aligned}
S_{left} &= \binom{B}{D-d}-\binom{B}{D-d-k} && (17)\\
&= \binom{B}{D-d}\cdot\left[1-\frac{(D-d)(D-d-1)\cdots(D-d-k+1)}{(B-D-d+k)(B-D-d+k-1)\cdots(B-D-d+1)}\right] && (18)\\
&\leq \binom{B}{D-d}\cdot\left[1-\left(\frac{D-d-k+1}{B-D+d+k}\right)^k\right] && (19)\\
&\leq \binom{B}{D-d}\cdot\left[1-\left(\frac{U+d+k-(U-D+d+k-1)}{U+d+k}\right)^k\right] && (20)\\
&\leq \binom{B}{D-d}\cdot\left[1-\left(\frac{U+d+k-\mathcal{O}(\sqrt{B\log n})}{U+d+k}\right)^k\right] && (21)\\
&\leq \Theta\left(\frac{k\sqrt{\log n}}{\sqrt{B}}\right)\cdot\binom{B}{D-d} && (22)
\end{aligned}$$

◀

▶ **Lemma 26.** *$S_{right}\leq c_2\frac{k'\cdot\sqrt{logn}}{\sqrt{B}}\cdot\binom{B}{U-d}$ for some constant c_2.*

Proof.

$$\begin{aligned}
S_{right} &= \binom{B}{U-d}-\binom{B}{U-d-k'} && (23)\\
&= \binom{B}{U-d}\cdot\left[1-\frac{(U-d)(U-d-1)\cdots(U-d-k'+1)}{(B-U-d+k')(B-U-d+k'-1)\cdots(B-U-d+1)}\right] && (24)
\end{aligned}$$

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

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

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

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

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

◀

▶ **Lemma 69.** $S_{tot} \geq \binom{2B}{2D} \cdot \left[1 - \left(1 - \frac{k'}{2U+1}\right)^k\right]$.

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 24.** *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 27.** *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: `First-Return` Queries

▶ **Lemma 31.** *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 8 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 22, 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 64 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 32.** *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 22, 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 64 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 [17], 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 [29] originally for modeling social networks; nonetheless, it has proven to be an useful general statistical model in numerous fields, including recommender systems [33, 49], medicine [51], social networks [21, 44], molecular biology [11, 36], genetics [9, 30, 13], and image segmentation [50]. Canonical problems for this model are the community detection and community recovery problems: some recent works include [12, 40, 3, 2]; see e.g., [1] for survey of recent results. The study of Small-World networks is originated in [55] has frequently been observed, and proven to be important for the modeling of many real world graphs such as social networks [15, 54], brain neurons [6], 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) [31, 37]. See also e.g., [43] and Chapter 20 of [16].

Generation of random graphs

The problem of local-access implementation of random graphs has been considered in the aforementioned work [24, 41, 18], as well as in [35] 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 [39, 7, 45, 38], and the parallel computation model [4].

Query models

In the study of sub-linear time graph algorithms where reading the entire input is infeasible, it is necessary to specify how the algorithm may access the input graph, normally by defining the type of queries that the algorithm may ask about the input graph; the allowed types of queries can greatly affect the performance of the algorithms. While Next-Neighbor query is only recently considered in [18], there are other query models providing a neighbor of a vertex, such as asking for an entry in the adjacency-list representation [28], or traversing to a random neighbor. On the other hand, the Vertex-Pair query is common in the study of dense graphs as accessing the adjacency matrix representation [27]. The All-Neighbors query has recently been explicitly considered in local algorithms [19].

Other constructions of huge pseudorandom functions that are permutations or random hash functions were given in [34, 42, 35].