Partial Sampling of Huge Random Objects

Amartya Shankha Biswas
CSAIL, MIT
asbiswas@mit.edu

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

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

Abstract

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

Our results focus on undirected graphs with independent edge probabilities, that is, each edge is chosen as an independent Bernoulli random variable. We provide a general implementation for generators in this model. Then, we use this construction to obtain the first efficient local implementations for the Erdös-Rényi $G(n, p)$ model, and the Stochastic Block model. As in previous local-access implementations for random graphs, we support VERTEX-PAIR, NEXT-NEIGHBOR queries, and ALL-NEIGHBORS queries. In addition, we introduce a new RANDOM-NEIGHBOR query. We also give the first local-access generation procedure for ALL-NEIGHBORS queries in the (sparse and directed) Kleinberg's Small-World model. Note that, in the sparse case, an ALL-NEIGHBORS query can be used to simulate the other types of queries efficiently. All of our generators require no pre-processing time, and answer each query using $\mathcal{O}(\text{poly}(\log n))$ time, random bits, and additional space.

We also show how to implement random Catalan objects, specifically focusing on Dyck paths (balanced random walks that are always positive). Here, we support HEIGHT queries to find the location of the walk and FIRST-RETURN queries to find the time when the walk returns to a specified location. As an application, we show that this can be used to implement NEXT-NEIGHBOR queries on random rooted and binary trees, and MATCHING-BRACKET queries on random well bracketed expressions (the Dyck language).

Finally, we study random q-colorings of graphs with maximum degree Δ. This is a new setting where the random object also has a "huge" description (the underlying graph) that can be accessed through adjacency list queries. This setting is similar to Local Computation Algorithms [RTVX11, ARVX12] with the added restriction that the output must follow a specific distribution in addition to being legal. We show how to sample the color of a single node in sub-linear time when $q > \alpha\Delta$ where α is a small constant.

2012 ACM Subject Classification Author: Please fill in 1 or more \ccsdesc macro

Keywords and phrases Dummy keyword

Funding *Amartya Shankha Biswas*: funding
Ronitt Rubinfeld: funding

Anak Yodpinyanee: funding

Contents

1 Introduction

The problem of computing local information of huge random objects was pioneered in [GGN03, GGN10]. Further work of [NN07] considers the generation of sparse random $G(n,p)$ graphs from the Erdös-Rényi model [ER60], with $p = O(\text{poly}(\log n)/n)$, which answers $\text{poly}(\log n)$ All-Neighbors queries, listing the neighbors of queried vertices. While these generators use polylogarithmic resources over their entire execution, they generate graphs that are only guaranteed to *appear random* to algorithms that inspect a *limited portion* of the generated graph. For example, the greedy routing algorithm on Kleinberg's small world networks [Kle00] only uses $\mathcal{O}(\log^2 n)$ probes. Using our implementation, one can execute this algorithm on a random small world instance in $\mathcal{O}(\text{poly}(\log n))$ time without incurring the $\mathcal{O}(n)$ prior-sampling overhead.

In [ELMR17], the authors construct an oracle for the generation of recursive trees, and BA preferential attachment graphs. Unlike [NN07], their implementation allows for an arbitrary number of queries. This result is particularly interesting – although the graphs in this model are generated via a sequential process, the oracle is able to locally generate arbitrary portions of it and answer queries in polylogarithmic time. Though preferential attachment graphs are sparse, they contain vertices of high degree, thus [ELMR17] provides access to the adjacency list through Next-Neighbor queries.

Segway

1.1 Undirected Graphs

In Section 4, we implement queries to both the adjacency matrix and adjacency list representation for the generic class of *undirected graphs* with *independent edge probabilities* $\{p_{uv}\}_{u,v \in V}$, where p_{uv} denotes the probability that there is an edge between u and v. Throughout, we identify our vertices via their unique IDs from 1 to n, namely $V = [n]$. We implement Vertex-Pair, Next-Neighbor, and Random-Neighbor [1] queries. Under reasonable assumptions on the ability to compute certain values pertaining to consecutive edge probabilities, our implementations support all three types of queries using $\mathcal{O}(\text{poly}(\log n))$ time, space, and random bits. In particular, our construction yields local-access generators for the Erdös-Rényi $G(n,p)$ model (for *all* values of p), and the Stochastic Block model with random community assignment. As in [ELMR17] (and unlike the generators in [GGN03, GGN10, NN07]), our techniques allow unlimited queries.

While Vertex-Pair and Next-Neighbor queries, as well as All-Neighbors queries for sparse graphs, have been considered in the prior works of [ELMR17, GGN03, GGN10, NN07], we provide the first implementation (to the best of our knowledge) of Random-Neighbor queries, which do not follow trivially from the All-Neighbor queries in *non-sparse graphs*. Such queries are useful, for instance, for sub-linear algorithms that employ random walk processes. Random-Neighbor queries present particularly interesting challenges that are outlined below.

Next-Neighbor Queries

We note that the next neighbor of a vertex can be found trivially by generating consecutive entries of the adjacency matrix, but for small edge probabilities $p_{uv} = o(1)$ this implementation can be too slow. In our algorithms, we achieve speed-up by sampling multiple neighbor values at once for a given vertex u; more specifically, we sample for the number of "non-neighbors" preceding the

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

next neighbor. To do this, we assume that we have access to an oracle which can estimate the "skip" probabilities $F(v,a,b) = \prod_{u=a}^{b}(1-p_{v,u})$, where $F(v,a,b)$ is the probability that v has no neighbors in the range $[a,b]$. We later show that it is possible to compute this quantity efficiently for the $G(n,p)$ and Stochastic block models.

A main difficulty in our setup, as compared to [ELMR17], arises from the fact that our graph is undirected, and thus we must design a data structure that "informs" all (potentially $\Theta(n)$) non-neighbors once we decide on the query vertex's next neighbor. More concretely, if u' is sampled as the next neighbor of v after its previous neighbor u, we must maintain consistency in subsequent steps by ensuring that none of the vertices in the range (u,u') return v as a neighbor. This update will become even more complicated as we later handle Random-Neighbor queries, where we may generate non-neighbors at random locations.

In Section 4.2, we present a very simple randomized generator (Algorithm 2) that supports Next-Neighbor queries efficiently, albeit the analysis of its performance is rather complicated. We remark that this approach may be extended to support Vertex-Pair queries with superior performance (given that we do not to support Random-Neighbor queries) and to provide deterministic resource usage guarantee – the full analysis can be found in Section ?? and ??, respectively.

What is deterministic here?

Random-Neighbor Queries

We provide efficient Random-Neighbor queries (Section 4.3). The ability to do so is surprising since: (1) Random-Neighbor queries affect the conditional probabilities of the remaining neighbors in a non-trivial manner [2], and (2) our implementation does not resort to explicitly sampling the degree of any vertex v in order to generate a random neighbor with the correct probability $\frac{1}{d_v}$. First, 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] – that is, from the point of view of the *agent* interacting with the generator. Second, sampling the degree of the query vertex, we suspect, is not viable for *sub-linear* generators, because this quantity alone imposes dependence on the existence of *all* of its potential incident edges. Therefore, our generator needs to return a random neighbor, with probability reciprocal to the query vertex's degree, without resorting to "knowing" its degree. The generator, however, must somehow maintain and leverage its additional *internal knowledge* of the partially-generated graph, to keep its computation tractable throughout the entire graph generation process. This requires a way of implicitly keeping track of all the resulting changes.

We formulate a *bucketing approach* (Section 4.3) which samples multiple consecutive edges at once, in such a way that the conditional probabilities of the unsampled edges remain independent and "well-behaved" during subsequent queries. For each vertex v, we divide the vertex set (potential neighbors) or v into consecutive ranges (buckets), so that each bucket contains, in expectation, roughly the same number of neighbors $\sum_{u=a}^{b} p_{v,u}$ (which we must be able to compute efficiently). The subroutine of Next-Neighbor may be applied to sample the neighbors within a bucket in expected constant time. Then, one may obtain a random neighbor of v by picking a random neighbor from a random bucket; probabilities of picking any neighbors may be normalized to the uniform distribution via rejection sampling, while stilling yielding $\mathrm{poly}(\log n)$ complexities overall. This bucketing approach also naturally leads to our data structure that requires constant space for

[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.

each bucket and for each edge, using $\Theta(n+m)$ overall memory requirement. The VERTEX-PAIR queries are implemented by sampling the relevant bucket.

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

Erdös-Rényi

In Section 5.1, we apply our construction to random $G(n,p)$ graphs for arbitrary p, and obtain VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR queries, using polylogarithmic resources (time, space and random bits) per query. We remark that, while $\Omega(n+m) = \Omega(pn^2)$ time and space is clearly necessary to generate and represent a full random graph, our implementation supports local-access via all three types of queries, and yet can generate a full graph in $\widetilde{O}(n+m)$ time and space (Corollary 17), which is tight up to polylogarithmic factors.

Stochastic Block Model

We generalize our construction to the Stochastic Block Model. In this model, the vertex set is partitioned into r *communities* $\{C_1, \ldots, C_r\}$. The probability that an edge exists depends on the communities of its endpoints: if $u \in C_i$ and $v \in C_j$, then $\{u,v\}$ exists with probability $p_{i,j}$, given in an $r \times r$ matrix $\mathbf{P}$. As communities in the observed data are generally unknown a priori, and significant research has been devoted to designing efficient algorithm for community detection and recovery, these studies generally consider the *random community assignment* condition for the purpose of designing and analyzing algorithms (see e.g., [MNS15]). Thus, in this work, we aim to construct generators for this important case, where the community assignment of vertices are independently sampled from some given distribution R.

Our approach is, as before, to sample for the next neighbor or a random neighbor directly, although our result does not simply follow closed-form formulas, as the probabilities for the potential edges now depend on the communities of endpoints. To handle this issue, we observe that it is sufficient to efficiently count the number of vertices of each community in any range of contiguous vertex indices. We then design a data structure extending a construction of [GGN10], which maintain these counts for ranges of vertices, and "sample" the partition of their counts only on an as-needed basis. This extension results in an efficient technique to sample counts from the *multivariate hypergeometric distribution* (Section 3.1). This sampling procedure may be of independent interest. For r communities, this yields an implementation with $\mathcal{O}(r \cdot \mathrm{poly}(\log n))$ overhead in required resources for each operation. This upholds all previous polylogarithmic guarantees when $r = \mathrm{poly}(\log n)$.

CDF Based Sampling

It is worth noting that our techniques for implementing local-access for the ER and SBM graphs can easily be extended to other similar models of random graphs. The only requirement is that the CDF of the probability sequences can be efficiently computed as in Section 2.3.

1.2 Directed Graphs

We then consider local-access generators for directed graphs in Kleinberg's Small World model. In this case, the probabilities are based on distances in a 2-dimensional grid. Using a modified version of our previous sampling procedure, we present such a generator supporting All-Neighbors queries in $\mathcal{O}(\text{poly}(\log n))$ time, space and random bits per query (since such graphs are sparse, the other queries follow directly).

This is duplicate

Lastly, we consider Kleinberg's Small World model ([Kle00, MN04]) in Section 6. While Small-World models are proposed to capture properties of observed data such as small shortest-path distances and large clustering coefficients [WS98], this important special case of Kleinberg's model, defined on two-dimensional grids, demonstrates underlying geographical structures of networks. The vertices are aligned on a $\sqrt{n} \times \sqrt{n}$ grid, and the edge probabilities are a function of a two-dimensional distance metric. Since the degree of each vertex in this model is $\mathcal{O}(\log n)$ with high probability, we design generators supporting All-Neighbor queries.

1.3 Catalan Objects

We also consider the problem of sampling of very long ($2n$ step) one dimensional random walks, One obvious query of interest is Height(t) which returns the position of the walk at time t. Height queries for the simple unconstrained random walk follow trivially from the implementation of interval summable functions presented in [GGN10]. Instead, we focus on an important generalization by considering balanced random walks (equal number of up and down steps) that are constrained to be always positive (commonly known as Dyck Paths). The added constraint introduces complicated non-local dependencies on the distribution of positions. However, we are able to support both queries using $\mathcal{O}(\text{poly}(\log n))$ resources.

Dyck paths are one type of Catalan object, and they have natural bijections to other Catalan objects such as bracketed expressions, random rooted trees and binary trees Thus, we can use our Dyck Path implementation to obtain useful implementations of other random Catalan objects. For instance, Height queries correspond to Depth queries on rooted trees and bracketed expressions (Section 8.1).

Why are first returns difficult?

Dependencies from past queries

However, we might want to support more interesting qaueries; for example, finding the children of a node in a random tree or finding the matching bracket in a random bracketed expression. To achieve this, we will also support First-Return queries where First-Return(t) returns the first time when the random walk returns to the same level as it was at time t. In Section 8.1, we will see that First-Return queries correspond to Next-Neighbor queries on trees and Matching-Bracket queries on bracketed expressions.

1.4 Random Coloring of Graphs

Finally, we introduce a new model for implementating huge random objects with *huge description size*. In this model, we implement query access to random q-colorings of arbitrary graphs with maximum degree Δ. A random coloring is sampled by proposing $\mathcal{O}(n \log n)$ color updates and accepting the ones that do not create a conflict (Glauber dynamics). This is an inherently sequential process with the acceptance of a particular proposal depending on all preceding neighboring proposals. Moreover, unlike the previously considered random objects, this one has no succint representation, and we can only uncover the proper distribution by probing the graph (in the manner of *local computation algorithms* [RTVX11]). Unlike LCAs which have to return *some* valid solution, we also have to make sure that we return a solution from the correct distribution. We are able to construct an efficient implementation that returns the final color of a vertex using only a sub-linear number of probes when $q > 12\Delta$.

For additional related work, see Section **??**.

2 Preliminaries

2.1 Local-Access Generators

We begin by formalizing a model of *local-access generators* (Section 2.1), implicitly used in [ELMR17]. Our work provides local-access generators for various basic classes of graphs described in the following, with Vertex-Pair, Next-Neighbor, and Random-Neighbor queries. In all of our results, each query is processed using $\mathrm{poly}(\log n)$ time, random bits, and additional space, with *no initialization overhead*. These guarantees hold even in the case of adversarial queries. Our bounds assume constant computation time for each arithmetic operation with $O(\log n)$-bit precision. Each of our generators constructs a random graph drawn from a distribution that is $1/\mathrm{poly}(n)$-close to the desired distribution in the L_1-distance. [3]

Not good

We consider the problem of locally generating random graphs $G = (V, E)$ drawn from the desired families of simple unweighted graphs, undirected or directed. We denote the number of vertices $n = |V|$, and refer to each vertex simply via its unique ID from $[n]$. For undirected G, the set of neighbors of $v \in V$ is defined as $\Gamma(v) = \{u \in V : \{v, u\} \in E\}$; denote its degree by $\deg(v) = |\Gamma(v)|$. Inspired by the goals and results of [ELMR17], we define a model of local-access generators as follows.

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

- ***Consistency.*** *The responses of the local-access generator to all probes throughout the entire execution must be consistent with a single graph G.*
- ***Distribution equivalence.*** *The random graph G provided by the generator must be sampled from some distribution D' that is ϵ-close to the desired distribution D in the L_1-distance. In this work we focus on supporting $\epsilon = n^{-c}$ for any desired constant $c > 0$. As for* Random-Neighbor(v), *the distribution from which a neighbor is returned must be ϵ-close to the uniform distribution over neighbors of v with respect to the sampled random graph G (w.h.p $1 - n^{-c}$ for each query).*
- ***Performance.*** *The resources, consisting of (1) computation time, (2) additional random bits required, and (3) additional space required, in order to compute an answer to a single query and update the data structure, must be sub-linear, preferably* $\mathrm{poly}(\log n)$.

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

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

Supported Queries in our Model

For undirected graphs, we consider queries of the following forms. now we might want to do Next-Neighbor first for consistency.

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

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

2.2 Random Graph Models

Erdös-Rényi Model

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

Stochastic Block Model

This model is a generalization of the Erdös-Rényi Model. The vertex set V is partitioned into r communities $C_1, \ldots, C_r$. The probability that the edge $\{u, v\}$ exists is $p_{i,j}$ when $u \in C_i$ and $v \in C_j$, where the probabilities are given as an $r \times r$ symmetric matrix $\mathbf{P} = [p_{i,j}]_{i,j\in[r]}$. We assume that we are given explicitly the distribution R over the communities, and each vertex is assigned its community according to R independently at random. [4]

Small-World Model

In this model, each vertex is identified via its 2D coordinate $v = (v_x, v_y) \in [\sqrt{n}]^2$. Define the Manhattan distance as $\textsc{dist}(u, v) = |u_x - v_x| + |u_y - v_y|$, and the probability that each directed edge (u, v) exists is $c/(\textsc{dist}(u, v))^2$. Here, c is an indicator of the number of long range directed edges present at each vertex. A common choice for c is given by normalizing the distribution so that there is exactly one directed edge emerging from each vertex ($c = \Theta(1/\log n)$). We will however support a range of values of $c = \log^{\pm\Theta(1)} n$. While not explicitly specified in the original model description of [Kle00], we assume that the probability is rounded down to 1 if $c/(\textsc{dist}(u, v))^2 > 1$.

2.3 Miscellaneous

Arithmetic operations

Let N be a sufficiently large number of bits required to maintain a multiplicative error of at most a $\frac{1}{\mathrm{poly}(n)}$ factor over $\mathrm{poly}(n)$ elementary computations $(+, -, \cdot, /, \exp)$.[5] We assume that each elementary operation on words of size N bits can be performed in constant time. Likewise, a random N-bit integer can be acquired in constant time. We assume that the input is also given with N-bit precision.

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

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

Sampling via a CDF

Consider a probability distribution X over $O(n)$ consecutive integers, whose cumulative distribution function (CDF) for can be computed with at most n^{-c} additive error for constant c. Using $\mathcal{O}(\log n)$ CDF evaluations, one can sample from a distribution that is $\frac{1}{\mathrm{poly}(n)}$-close to X in L_1-distance.[6]

3 Some Basic Implementations

This section needs major overhaul

3.1 Sampling from the Multivariate Hypergeometric Distribution

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

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

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

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 $\mathbf{S}^{\mathbf{C}}_\ell$ 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 $O(r \,\mathrm{poly}(\log n))$ time with high probability. We first make use of the following procedure from [GGN10].

▶ **Lemma 2.** *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 $O(\mathrm{poly}(\log n))$ 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 3.** *Given C_1 marbles of color 1 and $C_2 = B - C_1$ marbles of color 2, there exists an algorithm that samples $\langle s_1, s_2 \rangle$, the number of marbles of each color appearing when drawing ℓ marbles from the urn without replacement, in $O(\mathrm{poly}(\log B))$ time and random words.*

Proof. For the base case where $B = 1$, we trivially have $\mathbf{S}^{\mathbf{C}}_1 = \mathbf{C}_1$ and $\mathbf{S}^{\mathbf{C}}_0 = \mathbf{C}_2$. Otherwise, for even B, we apply the following procedure.

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

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

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 $\mathbf{S}^{\mathbf{C}}_{\ell}$. □

Lastly we generalize to support larger r.

▶ **Theorem 4.** *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. 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 $\mathbf{S}^{\mathbf{D}}_{\ell} = \langle s_1, s_2 + \ldots + s_r\rangle$ via the above procedure. At this point we have obtained the first entry s_1 of the desired $\mathbf{S}^{\mathbf{C}}_{\ell}$. 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 $\mathbf{S}^{\mathbf{C}}_{\ell}$ 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 15. □

faster sampling for first k colors

▶ **Theorem 5.** *Given B marbles of r different colors in $[r]$, such that there are C_i marbles of color i and a parameter $k \leq r$, there exists an algorithm that samples $s_1 + s_2 + \cdots + s_k$, the number of marbles among the first k colors appearing when drawing ℓ marbles from the urn without replacement, in $O(\mathrm{poly}(\log B))$ time and random words.*

Proof. Since we don't have to find the individual counts, we can be more efficient by grouping half the colors together at each step. Formally, we define a pair $\mathbf{D} = \langle D_1, D_2\rangle$ where $D_1 = C_1 + C_2 + \cdots + C_{r/2}$ and $D_2 = C_{r/2+1} + \cdots + C_{r-1} + C_r$. We then generate $\langle D'_1, D'_2\rangle = \mathbf{S}^{\mathbf{D}}_{\ell}$.

- If $k < r/2$, we recursively solve the problem with the first $r/2$ colors, $B \leftarrow D'_1$, and the original value of k.
- If $k > r/2$, we recurse on the last $r/2$ colors, B set to D'_2, and k set to $k - r/2$. In this case, we add D'_1 to the returned value.
- Otherwise, $k = r/2$ and we can return D'_1.

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

3.1.1 Data structure

We now show that Theorem 4 may be used in order to create the following data structure. Recall that R denote the given distribution over integers $[r]$ (namely, the random distribution of communities for each vertex). Our data structure generates and maintains random variables $X_1, \ldots, X_n$, each of which is drawn independently at random from R: X_i denotes the community of vertex i. Then given a pair (i, j), it returns the vector $\mathbf{C}(i, j) = \langle c_1, \ldots, c_r\rangle$ where c_k counts the number of variables $X_i, \ldots, X_j$ that takes on the value k. Note that we may also find out X_i by querying for (i, i) and take the corresponding index.

We maintain a complete binary tree whose leaves corresponds to indices from $[n]$. Each node represents a range and stores the vector $\mathbf{C}$ for the corresponding range. The root represents the entire range $[n]$, which is then halved in each level. Initially the root samples $\mathbf{C}(1, n)$ from the multinomial distribution according to R (see e.g., Section 3.4.1 of [Knu97]). Then, the children are generated on-the-fly using the lemma above. Thus, each query can be processed within $O(r \operatorname{poly}(\log n))$ time, yielding Theorem 18. Then, by embedding the information stored by the data structure into the state (as in the proof of Lemma 15), we obtain the desired Corollary 19.

4 Local-Access Generators for Random Undirected Graphs

In this section, we provide an efficient implementation of local-access generators for random undirected graphs when the probabilities $p_{u,v} = \mathbb{P}[\{u, v\} \in E]$ are given. More specifically, we assume that the following quantities can be efficiently computed: (1) the probability that there is no edge between a vertex u and a range of consecutive vertices from $[a, b]$, namely $\prod_{u=a}^{b}(1 - p_{v,u})$, and (2) the sum of the edge probabilities (i.e., the expected number of edges) between u and vertices from $[a, b]$, namely $\sum_{u=a}^{b} p_{v,u}$. We will later give subroutines for computing these values for the Erdös-Rényi model and the Stochastic Block model with randomly-assigned communities in Section 5. We also begin by assuming perfect-precision arithmetic, until Section 4.5 where we show how to relax this assumption to $N = \Theta(\log n)$-bit precision.

First, we propose a simple implementation of our generator in Section 4.1 that sequentially fills out the adjacency matrix; while we do not focus on its efficiency, we establish some basic concepts for further analysis in this section. Next, we improve our subroutine for Next-Neighbor queries in Section 4.2: this algorithm samples for the next candidate of the next neighbor in a more direct manner to speed-up the process. Extending this construction, we obtain our main algorithm in Section 4.3 via the bucketing technique: partition the vertex set into contiguous ranges to normalize the expected number of neighbors in each bucket, allowing an efficient Random-Neighbor implementation by picking a random neighbor from a random bucket. The subroutine that samples for neighbors within a bucket, along with the remaining analysis of the algorithm, is given later in Section 4.4. Lastly, Section 4.5 handles the errors that may occur due to the use of finite precision.

4.1 Naïve Generator with an Explicit Adjacency Matrix

First, consider a naïve implemention that simply fills out the cells of the $n \times n$ adjacency matrix $\mathbf{A}$ of G one-by-one as required by each query. Each entry $\mathbf{A}[u][v]$ occupies exactly one of following three states: $\mathbf{A}[u][v] = 1$ or 0 if the generator has determined that $\{u, v\} \in E$ or $\{u, v\} \notin E$, respectively, and $\mathbf{A}[u][v] = \phi$ if whether $\{u, v\} \in E$ or not will be determined by future random choices. Aside from $\mathbf{A}$, our generator also maintains the vector **last**, where $\mathbf{last}[v]$ records the neighbor of v returned in the last call Next-Neighbor(v), or $\mathbf{last}[v] = 0$ if no such call has been invoked. This definition of **last** was introduced in [ELMR17]. All cells of $\mathbf{A}$ and **last** are initialized to ϕ and 0, respectively. We refer to Algorithm 1 for its straightforward implemention, but highlight some notations and useful observations here.

Figure 1 Naïve Generator

```
procedure VERTEX-PAIR($(u, v)$)
    if $\mathbf{A}[u][v] = \phi$
        draw $X_{u,v} \sim \mathsf{Bern}(p_{u,v})$
        $\mathbf{A}[v][u], \mathbf{A}[u][v] \leftarrow X_{u,v}$
    return $\mathbf{A}[u][v]$
procedure NEXT-NEIGHBOR($(v)$)
    for $u \leftarrow$ last$[v] + 1$ to $n$
        if VERTEX-PAIR$(v, u) = 1$
            last$[v] \leftarrow u$
            return $u$
    last$[v] \leftarrow n + 1$
    return $n + 1$
procedure RANDOM-NEIGHBOR($(v)$)
    $R \leftarrow V$
    while $R \neq \emptyset$
        sample $u \in R$ u.a.r.
        if VERTEX-PAIR$(v, u) = 1$
            return $u$
        else
            $R \leftarrow R \setminus \{u\}$
    return $\perp$
```

Characterizing random choices via $X_{u,v}$'s

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

Sampling from $\Gamma(v)$ uniformly without knowing $\deg(v)$

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

Capturing the state of the partially-generated graph with $\mathbf{A}$

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

Obstacles for maintaining $\mathbf{A}$

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

4.2 Improved Next-Neighbor Queries via Run-of-0's Sampling

We now speed-up our Next-Neighbor(v) procedure by attempting to sample for the first index $u > \mathbf{last}[v]$ of $X_{v,u} = 1$, from a sequence of Bernoulli RVs $\{X_{v,u}\}_{u>\mathbf{last}[v]}$, in a direct fashion. To do so, we sample a consecutive "run" of 0's with probability $\prod_{u=\mathbf{last}[v]+1}^{u'}(1-p_{v,u})$: this is the probability that there is no edge between a vertex v and any $u \in (\mathbf{last}[v], u']$, which can be computed efficiently by our assumption. The problem is that, some entries $\mathbf{A}[v][u]$'s in this run may have already been determined (to be 1 or 0) by queries Next-Neighbor(u) for $u > \mathbf{last}[v]$. To this end, we give a succinct data structure that determines the value of $\mathbf{A}[v][u]$ for $u > \mathbf{last}[v]$ and, more generally, captures the state $\mathbf{A}$, in Section 4.2.1. Using this data structure, we ensure that our sampled run does not skip over any 1. Next, for the sampled index u of the first occurrence of 1, we check against this data structure to see if $\mathbf{A}[v][u]$ is already assigned to 0, in which case we re-sample for a new candidate $u' > u$. Section 4.2.2 discusses the subtlety of this issue.

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

4.2.1 Data structure

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

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

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

Lemma 6. *The data structures* **last** *and* P_v*'s together provide a succinct representation of* $\mathbf{A}$ *when only* NEXT-NEIGHBOR *queries are allowed. In particular,* $\mathbf{A}[v][u] = 1$ *if and only if* $u \in P_v$*. Otherwise,* $\mathbf{A}[v][u] = 0$ *when* $u < \mathbf{last}[v]$ *or* $v < \mathbf{last}[u]$*. In all remaining cases,* $\mathbf{A}[v][u] = \phi$*.*

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

4.2.2 Queries and Updates

Algorithm 2 Sampling NEXT-NEIGHBOR

```
procedure NEXT-NEIGHBOR($v$)
    $u \leftarrow \mathbf{last}[v]$
    $w_v \leftarrow \min\{(P_v \cap (u, n]) \cup \{n+1\}\}$
    while $u = w_v$ or $\mathbf{last}[u] < v$
        sample $F \sim \mathsf{F}(v, u, w_v)$
        $u \leftarrow F$
    if $u \neq w_v$
        $P_v \leftarrow P_v \cap \{u\}$
        $P_u \leftarrow P_u \cap \{v\}$
    $\mathbf{last}[v] \leftarrow u$
    return $u$
```

We now provide our generator (Algorithm 2), and discuss the correctness of its sampling process. The argument here is rather subtle and relies on viewing the random process as an "uncovering" process on the table of RVs $X_{u,v}$'s as previously introduced in Section 4.1. Algorithm 2, considers the following experiment for sampling the next neighbor of v in the range $(\mathbf{last}[v], w_v)$. Suppose that we generate a sequence of $w_v - \mathbf{last}[v] - 1$ independent coin-tosses, where the i^{th} coin $C_{v,u}$ corresponding to $u = \mathbf{last}[v] + i$ has bias $p_{v,u}$, regardless of whether $X_{v,u}$'s are decided or not. Then, we use the sequence $\langle C_{v,u} \rangle$ to assign values to *undecided* random variable $X_{v,u}$. The crucial observation here is that, the *decided* random variables $X_{v,u} = 0$ do not need coin-flips, and the corresponding coin result $C_{v,u}$ can simply be discarded. Thus, we need to generate coin-flips up until we encounter some u satisfying both (i) $C_{v,u} = 1$, and (ii) $\mathbf{A}[v][u] = \phi$.

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

Obstacles for extending beyond Next-Neighbor queries

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

Secondly, in developing Algorithm 2, we observe that $\mathbf{last}[v]$ and P_v together provide a succinct representation of $\mathbf{A}[v][u] = \mathtt{0}$ only for contiguous cells $\mathbf{A}[v][u]$ where $u \leq \mathbf{last}[v]$ or $v \leq \mathbf{last}[u]$: they cannot handle $\mathtt{0}$ anywhere else. Unfortunately, in order to extend our construction to support Random-Neighbor queries using the idea suggested in Algorithm 1, we must unavoidably assign $\mathbf{A}[v][u]$ to $\mathtt{0}$ in random locations beyond $\mathbf{last}[v]$ or $\mathbf{last}[u]$, which cannot be captured by the current data structure. Furthermore, unlike $\mathtt{1}$'s, we cannot record $\mathtt{0}$'s using a data structure similarly to that of P_v. More specifically, to speed-up the sampling process for small $p_{v,u}$'s, we must generate many random non-neighbors at once as suggested in Algorithm 2, but we cannot afford to spend time linear in the number of created $\mathtt{0}$'s to update our data structure. We remedy these issues via the following bucketing approach.

4.3 Final Generator via the Bucketing Approach

We now resolve both of the above issues via the bucketing approach, allowing our generator to support all remaining types of queries. We begin this section by focusing first on Random-Neighbor queries, then extend the construction to the remaining ones. In order to handle Random-Neighbor(v), we divide the neighbors of v into *buckets* $B_v = \{B_v^{(1)}, B_v^{(2)}, \ldots\}$, so that each bucket contains, in expectation, roughly the same number of neighbors of v. We may then implement Random-Neighbor(v) by randomly selecting a bucket $B_v^{(i)}$, fill in entries $\mathbf{A}[v][u]$ for $u \in B_v^{(i)}$ with $\mathtt{1}$'s and $\mathtt{0}$'s, then report a random neighbor from this bucket. As the bucket size may be too large when the probabilities are small, instead of using a linear scan, our Fill subroutine will be implemented with the Next-Neighbor subroutine in Algorithm 2 previously developed in Section 4.2. Since the number of iterations required by this subroutine is roughly proportional to the number of neighbors, we choose to allocate a constant number of neighbors in expectation to each bucket: with constant probability the bucket contains some neighbors, and with high probability it has at most $O(\log n)$ neighbors.

Nonetheless, as the actual number of neighbors appearing in each bucket may be different, we balance out these discrepancies by performing *rejection sampling*, equalizing the probability of choosing any neighbor implicitly, again without the knowledge of $\deg(v)$ as previously done in Section 4.1. Leveraging the fact that the maximum number of neighbors in any bucket is $\mathcal{O}(\log n)$, we show not only that the probabability of success in the rejection sampling process is at least $1/\mathrm{poly}(\log n)$, but the number of iterations required by Next-Neighbor is also bounded by $\mathrm{poly}(\log n)$, achieving the overall $\mathrm{poly}(\log n)$ complexities. Here in this section, we will extensively rely on the assumption that the expected number of neighbors for consecutive vertices, $\sum_{u=a}^{b} p_{v,u}$, can be computed efficiently.

4.3.1 Partitioning into buckets

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

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

▶ **Lemma 7.** *With high probability, the number of neighbors in every bucket, $|\Gamma^{(i)}(v)|$, is at most $\mathcal{O}(\log n)$.*

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

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

for any constant $c > 0$. □

▶ **Lemma 8.** *With high probability, for every v such that $|B_v| = \Omega(\log n)$ (i.e., $\mathbb{E} = \Omega(\log n)$), at least a 1/3-fraction of the buckets $\{B_v^{(i)}\}_{i \in [|B_v|]}$ are non-empty.*

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

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

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

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

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

4.3.2 Filling a bucket

We consider buckets to be in two possible states – filled or unfilled. Initially, all buckets are considered unfilled. In our algorithm we will maintain, for each bucket $B_v^{(i)}$, the set $P_v^{(i)}$ of known neighbors of u in bucket $B_v^{(i)}$; this is a refinement of the set P_v in Section 4.2. We define the behaviors of the procedure $\texttt{Fill}(v, i)$ as follows. When invoked on an unfilled bucket $B_v^{(i)}$, $\texttt{Fill}(v, i)$ performs the following tasks:

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

For the sake of presentation, we postpone our description of the implementation of Fill to Section 4.4. For now, let us use Fill as a black-box operation.

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

Figure 3 Bucketing Generator

```
procedure Random-Neighbor($(v)$)
    $R \leftarrow [|B_v|]$
    while $R = \emptyset$
        sample $i \in R$ u.a.r.
        if $B_v^{(i)}$ is not filled
            Fill $(v, i)$
        if $|P_v^{(i)}| > 0$
            with probability $\frac{|P_v^{(i)}|}{M}$
                sample $u \in P_v^{(i)}$ u.a.r
                return $u$
        else
            $R \leftarrow R \setminus \{i\}$
    return $\perp$
```

Consider Algorithm 3 for generating a random neighbor via rejection sampling, in a rather similar overall framework as the simple implementation in Section 4.1. For simplicity, throughout the analysis, we assume $|B_v| = \Omega(\log n)$; otherwise, invoke Fill(v, i) for all $i \in [|B_v|]$ to obtain the entire neighbor list $\Gamma(v)$. This does not affect the analysis because we will soon bound the number of calls that Algorithm 3 makes to Fill by $O(\log n)$ (in expectation) for $|B_v| = \Omega(\log n)$.

To obtain a random neighbor, we first choose a bucket $B_v^{(i)}$ uniformly at random. If the bucket is not yet filled, we invoke Fill(v, i) and fill this bucket. Then, we *accept* the sampled bucket for generating our random neighbor with probability proportional to $|P_v^{(i)}|$. More specifically, let $M = \Theta(\log n)$ be the upper bound on the maximum number of neighbors in any bucket, as derived in Lemma 7; we accept this bucket with probability $|P_v^{(i)}|/M$, which is well-defined (i.e., does not exceed 1) with high probability. (Note that if $P_v^{(i)} = \emptyset$, we remove i from the pool, then repeat as usual.) If we choose to accept this bucket, we return a random neighbor from $P_v^{(i)}$. Otherwise, *reject* this bucket and repeat the process again.

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

▶ **Lemma 9.** *Algorithm 3 returns a uniformly random neighbor of vertex v.*

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

▶ **Lemma 10.** *Algorithm 3 terminates in $\mathcal{O}(\log n)$ iterations in expectation, or $\mathcal{O}(\log^2 n)$ iterations with high probability.*

Proof. Following the analysis above, the probability that some vertex from $P_v^{(i)}$ is accepted in an iteration is at least $1/(|R| \cdot M)$. From Lemma 8, a (1/3)-fraction of the buckets are non-empty (with high probability), so the probability of choosing a non-empty bucket is at least 1/3. Further, $M = \Theta(\log n)$ by Lemma 7. Hence, the success probability of each iteration is at least $1/(3M) = \Omega(1/\log n)$. Thus, with high probability, the number of iterations required is $O(\log^2 n)$ with high probability. ◻

4.4 Implementation of Fill

Figure 4 Sampling in a Bucket

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

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

As discussed in Section 4.2, while we have recorded all occurrences of $\mathbf{A}[v][u] = 1$ in $P_v^{(i)}$, we need and efficient way of checking whether $\mathbf{A}[v][u] = 0$ or ϕ. In Algorithm 2, **last** serves this purpose by showing that $\mathbf{A}[v][u]$ for all $u \leq \mathbf{last}[v]$ are decided as shown in Lemma 6. Here instead, with our bucket structure, we maintain a single bit marking whether each bucket is filled or unfilled: a filled bucket implies that $\mathbf{A}[v][u]$ for all $u \in B_v^{(i)}$ are decided. The bucket structure along with mark bits, unlike **last**, are capable of handling intermittent ranges of intervals, namely buckets, which is sufficient for our purpose, as shown in the following lemma. This yields the implementation Algorithm 4 for the Fill procedure fulfilling the requirement previously given in Section 4.3.2.

▶ **Lemma 11.** *The data structures $P_v^{(i)}$'s and the bucket marking bits together provide a succinct representation of $\mathbf{A}$ as long as modifications to $\mathbf{A}$ are performed solely by the* Fill *operation in Algorithm 4. In particular, let $u \in B_v^{(i)}$ and $v \in B_u^{(j)}$. Then, $\mathbf{A}[v][u] = 1$ if and only if $u \in P_v^{(i)}$. Otherwise, $\mathbf{A}[v][u] = 0$ when at least one of $B_v^{(i)}$ or $B_u^{(j)}$ is marked as* filled. *In all remaining cases, $\mathbf{A}[v][u] = \phi$.*

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

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

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

▶ **Lemma 12.** *Each execution of Algorithm 4 (the* Fill *operation) on an* unfilled *bucket $B_v^{(i)}$, in expectation:*

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

Observe that the number of iterations required by Algorithm 4 only depends on its random coin-flips and independent of the state of the algorithm. Combining with Lemma 10, we finally obtain polylogarithimc resource bound for our implementation of Random-Neighbor.

▶ **Corollary 13.** *Each execution of Algorithm 3 (the* Random-Neighbor *query), with high probability,*

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

Extension to other query types

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

- Vertex-Pair(u,v): We simply need to make sure that Lemma 11 holds, so we first apply Fill(u, j) on bucket $B_u^{(j)}$ containing v (if needed), then answer accordingly.
- Next-Neighbor(v): We maintain **last**, and keep invoking Fill until we find a neighbor. Recall that by Lemma 8, the probability that a particular bucket is empty is a small constant. Then with high probability, there exists no $\omega(\log n)$ *consecutive* empty buckets $B_v^{(i)}$'s for any vertex v, and thus Next-Neighbor only invokes up to $\mathcal{O}(\log n)$ calls to Fill.

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

▶ **Theorem 14.** *Under the assumption of*

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

there exists a local-access generator for the random graph family that supports Random-Neighbor, Vertex-Pair *and* Next-Neighbor *queries that uses polylogarithmic running time, additional space, and random words per query.*

Between these two assumptions, we first remove the assumption of perfect-precision arithmetic in the upcoming Section 4.5. Later in Section 5, we show applications of our generator to the $G(n, p)$ model, and the Stochastic Block model under random community assignment, by providing formulas and by constructing data structures for computing the quantities specified in the second assumption, respectively.

4.5 Removing the Perfect-Precision Arithmetic Assumption

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

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

▶ **Lemma 15.** *If Algorithm 4 (the* Fill *operation) is repeatedly invoked to construct a graph G by drawing the value u for at most S times in total, each of which comes from some distribution $\mathsf{F}'(v,a,b)$ that is ϵ-close in L_1-distance to the correct distribution $\mathsf{F}(v,a,b)$ that perfectly generates the desired distribution G over all graphs, then the distribution G' of the generated graph G is (ϵS)-close to G in the L_1-distance.*

Proof. For simplicity, assume that the algorithm generates the graph to completion according to a sequence of up to n^2 distinct buckets $\mathcal{B} = \langle B_{v_1}^{(u_1)}, B_{v_2}^{(u_2)}, \ldots\rangle$, where each $B_{v_i}^{(u_i)}$ specifies the unfilled bucket in which any query instigates a Fill function call. Define an *internal state* of our generator as the triplet $s = (k, u, \mathbf{A})$, representing that the algorithm is currently processing the k^{th} Fill, in the iteration (the **repeat** loop of Algorithm 4) with value u, and have generated $\mathbf{A}$ so far. Let $t_{\mathbf{A}}$ denote the *terminal state* after processing all queries and having generated the graph $G_{\mathbf{A}}$ represented by $\mathbf{A}$. We note that $\mathbf{A}$ is used here in the analysis but not explicitly maintained; further, it reflects the changes in every iteration: as u is updated during each iteration of Fill, the cells $\mathbf{A}[v][u'] = \phi$ for $u' < u$ (within that bucket) that has been skipped are also updated to 0.

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

The generator begins with the initial (internal) state $s_0 = (1, 0, \mathbf{A}_\phi)$ where all cells of $\mathbf{A}_\phi$ are ϕ's, goes through at most $S = O(n^3)$ other states (as there are up to n^2 values of k and $O(n)$ values of u), and reach some terminal state $t_{\mathbf{A}}$, generating the entire graph in the process. Let $\pi = \langle s_0^\pi = s_0, s_1^\pi, \ldots, s_{\ell(\pi)}^\pi = t_{\mathbf{A}}\rangle$ for some $\mathbf{A}$ denote a sequence ("path") of up to $S+1$ states the algorithm proceeds through, where $\ell(\pi)$ denote the number of transitions it undergoes. For simplicity, let $T_{t_{\mathbf{A}}}(t_{\mathbf{A}}) = 1$, and $T_{t_{\mathbf{A}}}(s) = 0$ for all state $s \ne t_{\mathbf{A}}$, 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 $\pi_i = \langle s_0^\pi, \dots, s_i^\pi \rangle$ of π. Observe that for $i \geq 1$,

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

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

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

5 Applications to Erdös-Rényi Model and Stochastic Block Model

In this section we demonstrate the application of our techniques to two well known, and widely studied models of randon graphs. That is, as required by Theorem 14, we must provide a method for computing the quantities $\prod_{u=a}^{b}(1-p_{v,u})$ and $\sum_{u=a}^{b} p_{v,u}$ of the desired random graph families in logarithmic time, space and random bits. Our first implementation focuses on the well known Erdös-Rényi model – $G(n,p)$: in this case, $p_{v,u} = p$ is uniform and our quantities admit closed-form formulas.

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

5.1 Erdös-Rényi Model

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

▶ **Corollary 16.** *The final algorithm in Section 4 locally generates a random graph from the Erdös-Rényi $G(n,p)$ model using $\mathcal{O}(\log^3 n)$ time, $\mathcal{O}(\log^2 n)$ random N-bit words, and $\mathcal{O}(\log^2 n)$ additional space per query with high probability.*

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

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

▶ **Corollary 17.** *The final algorithm in Section 4 can generate a complete random graph from the Erdös-Rényi $G(n, p)$ model using overall $\tilde{\mathcal{O}}(n + m)$ time, random bits and space, which is $\tilde{\mathcal{O}}(pn^2)$ in expectation. This is optimal up to $\mathcal{O}(\operatorname{poly}(\log n))$ factors.*

5.2 Stochastic Block model

For the Stochastic Block model, each vertex is assigned to some community C_i, $i \in [r]$. By partitioning the product by communities, we may rewrite the desired formulas, for $v \in C_i$, as $\prod_{u=a}^{b}(1 - p_{v,u}) = \prod_{j=1}^{r}(1 - p_{i,j})^{|[a,b] \cap C_j|}$ and $\sum_{u=a}^{b} p_{v,u} = \sum_{j=1}^{r} |[a, b] \cap C_j| \cdot p_{i,j}$. Thus, it is sufficient to design a data structure, or a *generator*, that draws a community assignment for the vertex set according to the given distribution R. This data structure should be able to efficiently count the number of occurrences of vertices of each community in any contiguous range, namely the value $|[a, b] \cap C_j|$ for each $j \in [r]$. To this end, we use the following lemma, yielding the generator for the Stochastic Block model that uses $O(r \operatorname{poly}(\log n))$ resources per query.

▶ **Theorem 18.** *There exists a data structure (generator) that samples a community for each vertex independently at random from R with $\frac{1}{\operatorname{poly}(n)}$ error in the L_1-distance, and supports queries that ask for the number of occurrences of vertices of each community in any contiguous range, using $O(r \operatorname{poly}(\log n))$ time, random N-bit words and additional space per query. Further, this data structure may be implemented in such a way that requires no overhead for initialization.*

▶ **Corollary 19.** *The final algorithm in Section 4 generates a random graph from the Stochastic Block model with randomly-assigned communities using $O(r \operatorname{poly}(\log n))$ time, random N-bit words, and additional space per query with high probability.*

Update

We provide the full details of the construction in the following Section 3.1. Our construction extends upon a similar generator in the work of [GGN10] which only supports $r = 2$. Our overall data structure is a balanced binary tree, where the root corresponds to the entire range of indices $\{1, \ldots, n\}$, and the children of each vertex corresponds to each half of the parent's range. Each node[7] holds the number of vertices of each community in its range. The tree initially contains only the root, with the number of vertices of each community sampled according to the multinomial distribution[8] (for n samples (vertices) from the probability distribution R). The children are only generated top-down on an as-needed basis according to the given queries. The technical difficulties arise when generating the children, where one needs to sample "half" of the counts of the parent from the correct marginal distribution. To this end, we show how to sample such a

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

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

count as described in the statement below. Namely, we provide an algorithm for sampling from the *multivariate hypergeometric distribution.*

Link back to previous section

6 Local-Access Generators for Random Directed Graphs

In this section, we consider Kleinberg's Small-World model [Kle00, MN04] where the probability that a *directed* edge (u, v) exists is $\min\{c/(\textsc{dist}(u, v))^2, 1\}$. Here, $\textsc{dist}(u, v)$ is the Manhattan distance between u and v on a $\sqrt{n} \times \sqrt{n}$ grid. We begin with the case where $c = 1$, then generalize to different values of $c = \log^{\pm\Theta(1)}(n)$. We aim to support All-Neighbors queries using $\mathrm{poly}(\log n)$ resources. This returns the entire list of out-neighbors of v.

6.1 Generator for $c = 1$

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

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

6.1.1 Phase 1: Sample the distance D

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

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

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

where the product enjoys telescoping as the denominator $(i^2)^{4i}$ cancels with $(i^2)^{4(i-1)}$ and $(i^2)^{4(i+1)}$ in the numerators of the previous and the next term, respectively. This gives us a closed form for the CDF, which we can compute with 2^{-N} additive error in constant time (by our computation model assumption). Thus, we may sample for the distance $D \sim \mathsf{D}(a)$ with $O(\log n)$ time and one random N-bit word.

6.1.2 Phase 2: Sampling neighbors at distance D

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

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

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

Thus at this point, we have sampled all neighbors in Γ^v_D. 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 20.** *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 N-bit words per* $\textsc{All-Neighbors}$ *query with high probability.*

6.2 Generator 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 $\frac{1}{\text{poly}(n)}$ in the L_1-distance.

6.2.1 Case $c < 1$

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

▶ **Lemma 21.** *Given a process $\mathcal{P}$ outputting the indices of successful Bernoulli trials with bias $\langle p_i \rangle$, there exists a process $\mathcal{P}^c$ outputting the indices of successful Bernoulli trials with bias $\langle c \cdot p_i \rangle$ where $c < 1$, using one additional N-bit word overhead for each answer of $\mathcal{P}$.*

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

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

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

▶ **Corollary 22.** *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities $c/(\texttt{DIST}(u,v))^2$ where $c < 1$, using $O(\log^2 n)$ time and random N-bit words per `ALL-NEIGHBORS` query with high probability.*

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

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

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

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

In Phase 1, we may apply this reduction only when the condition $c \cdot p_i \leq 1 - \frac{1}{k}$ is satisfied. For lower value of $p_i = 1/D^2$, namely for distance $D < \sqrt{c/(1-1/k)} = O(\sqrt{c})$, we may afford to sample the Bernoulli trials one-by-one as c is 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 24.** *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities $c/(\texttt{DIST}(u,v))^2$ where $c = \text{poly}(\log n)$, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *query with high probability.*

Queries

7 Permutations

In addition to a table (dictionary) containing all assigned values $\pi(i)$, we maintain the following binary trees, whose nodes are generated on-the-fly in response to queries.

T_1: Each node of T_1 corresponds to a specific range of indices, where the root represents the entire range $\{1, \dots, N\}$, and its two children represents each (approximately) half of the parent's range. Each node counts the indices i in the range, such that $\pi(i) = \perp$. Initially T_1 only contains the root node, and the number of unassigned indices are N. The children are only generated when we need to traverse down from the root; as these nodes are generated, all of the indices in their ranges are unassigned. Once an index becomes assigned, we simply update the information along the path in $\mathcal{O}(\log n)$ time.

T_2: T_2 is similar to T_1 but instead of maintaining the number of indices in the range that are still unassigned, it maintains the number of values in the range that are still unused (have not been assigned to an index). Similarly, we may sample an unused value or mark it as used within $\mathcal{O}(\log n)$ time.

To compute $\pi(i)$, first we check the table for $\pi(i)$ and return its value if $\pi(i) \neq \perp$. Otherwise, sample an unused value j from T_2 and mark that value as used. Add $\pi(i) = j$ to the table, and mark index i as used on T_1.

If we wish to support $\pi^{-1}(i)$, then also store the table of $\pi^{-1}(i)$. To assign $\pi^{-1}(i)$, sample an unassigned index j from T_1 then mark it as assigned, add $\pi(j) = i$ and $\pi^{-1}(i) = j$ to the table, and mark the value i as used on T_2.

7.1 Generating Permutations with given Cyclic Structure

We will use the technique from [NR02] to locally generate a random permutation with a given cyclic structure. This algorithm uses two permutations π and σ, where π is a uniformly random permutation and σ is a fixed permutation with the given structure. The resulting random permutation is formed by the composition $\pi^{-1}(\sigma(\pi(\cdot)))$. We will generate the permutation π as described in the previous section.;

We receive as input a list of cycle sizes $\{c_1, c_2, \cdots, c_k\}$ with the restriction that $\sum c_i = n$. Now we need to locally generate the permutation σ with the prescribed structure. Define the indices $C_j = \sum_{i=1}^{j} c_i$ with $C_0 = 0$. We will construct the cycle corresponding to c_i as all the elements in the interval $\{c_{i-1}+1, c_{i-1}+2, \cdots, c_i\}$.

Of course we will not be computing σ explicitly. Instead, we will pre-compute the C_j indices, and when given a query $\sigma(x)$, we binary search amongst C_j to find the cycle that x belongs to. Then we can report the value of $\sigma(x)$ accordingly.

So, we can now compose the generator oracles for π, σ, and π^{-1} to get the full generator.

8 Sampling Catalan Objects

Earlier, we were interested in querying the following random object. In a random permutation of n white marbles and n black marbles, how many white marbles are present in the first k positions. As we have seen before, [GGN10] gives us a method of sampling from this (hypergeometric) distribution. In constructing a generator for the Stochastic Block model, we generalized this by adding more colors (multivariate hypergeometric distribution). We also took this to the extreme where all marbles are distinguishable (i.e. a random permutation), and saw that this could also be implemented efficiently. Now we focus on a more challenging variant of this question with more complicated conditional dependences among the placement of the marbles.

Thresholding.

Mention that this is imperfect sampling (close impl.)

we have?

Connect with previous part on SBM

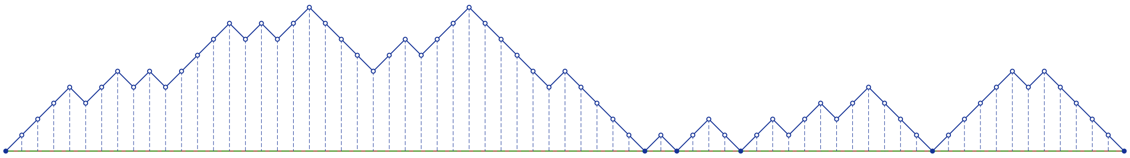

Figure 5 Simple Dyck path with $n = 35$.

Important extension of interval summable queries.

We 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. This can be interpreted as a Dyck path; a $2n$ step *balanced* one-dimensional walk with exactly n up and down steps. In Figure 5, 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 50) is the n^{th} Catalan number $C_n = \frac{1}{n+1} \cdot \binom{2n}{n}$. Many important combinatorial objects occur in Catalan families of which these are an example.

Our goal will be to support queries to a uniformly random instance of a Dyck path, which will in turn allow us to sample other random Catalan objects such as rooted trees, and bracketed expressions. Specifically, we will want to answer the following queries:

- Height(i): Returns the position of the path after i steps
- First-Return(i): Returns an index $j > i$ such that Height(j) $=$ Height(i) and for any k between i and j, Height(k) is strictly greater than Height(i).

8.1 Bijections to other Catalan objects

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

There is also a natural bijection between Dyck paths and rooted ordered trees by letting the path be a transcript of the DFS traversal of a tree. Starting with the root, for each "up-step" we move to a new child of the current node, and for each "down-step", we backtrack towards the root. Here, the HEIGHT query returns the depth of a node and the FIRST-RETURN query can be used to find the *next child* of a node.

Figure? Degree queries by repeated application.

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

8.2 Catalan Trapezoids and Generalized Dyck Paths

In order to sample Dyck paths locally, we will need to analyze more general Catalan objects. Specifically, we consider a sequence of U up-steps and D down-steps, such that the sum of any initial sub-string is not less than $1 - k$. This means that we start our Dyck path at a height of $k - 1$, and we are never allowed to cross below zero (Figure 6).

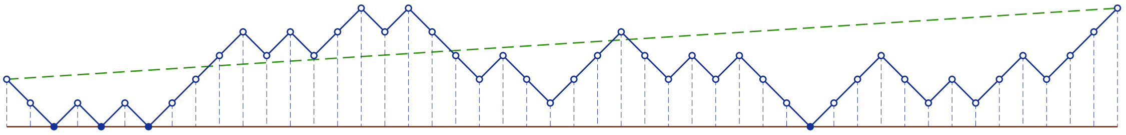

Figure 6 Complex Dyck path with $U = 25$, $D = 22$ and $k = 3$. Notice that the boundary is shifted.

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 (presented in [Reu14]). 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 [Reu14] without proof

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

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

Consider the situation after sampling the height of the Dyck path at various locations $\langle x_1, x_2, \cdots, x_m \rangle$. The revealed locations partition the path into disjoint *intervals* $[x_i, x_{i+1}]$ where the heights of the endpoints have been sampled. We define $y_i = \textsc{Height}(x_i)$ and notice that these intervals are independent of each other. Specifically, the path in 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 $U - D = y_{i+1} - y_i$, and this happens independent of any samples outside the interval. Next, we will show how one can sample heights within such an interval, and in Section 8.4 we will move on to the more complicated FIRST-RETURN queries.

Is this necessary?

We also maintain a threshold $\mathcal{T} = \Theta(\log^4 n)$. If a query lands in an interval that has length less than $\mathcal{T}$, then we brute force sample the entire interval one step at a time. Assuming that the probabilities of these events can be approximated efficiently (Lemma 55, this take poly$(\log n)$ time.

8.3 Sampling the Height

We will implement HEIGHT(t) by showing how to (efficiently) sample the position of the path in the midpoint of an existing interval. We can then extend this to arbitrary positions by running a binary search on the appropriate interval using the midpoint samples. If the interval in question has odd length, we sample one step on the boundary and proceed with a shortened interval.

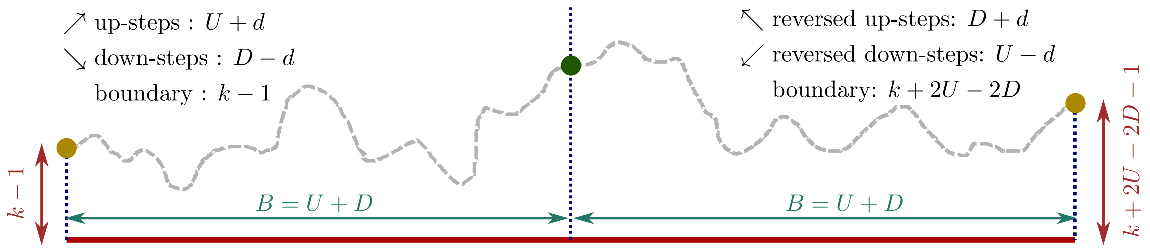

Figure 7 The $2B$-interval is split into two equal parts resulting in two separate Dyck problems. The green node (center) is the sampled height of the midpoint corresponding to a specific value of d. The path considered in both sub-intervals starts at a yellow node (left and right edges) and ends at 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. this interval should be sampled from $\mathsf{C}_k(2U, 2D)$ (the factor of two makes the analysis cleaner). Without loss of generality, we assume that $2D \leq B$; if this were not the case, we could simply flip the sequence and negate the elements. This essentially means that the overall path in the interval is non-decreasing in height.

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

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

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). Moving forwards, we will drop the d when referring to the path counts. We (conceptually) flip the second half of the interval, such that the corresponding path begins from the end of the $2B$-interval and terminates at the midpoint (Figure 7). This results in a different starting point, and the 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 interval (of length $2B$).

We will use the following rejection sampling lemma from [GGN10].

Frequently?

▶ **Lemma 25.** *Let $\{p_i\}$ and $\{q_i\}$ be distributions satisfying the following conditions*

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

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

An important point to note is that in order to apply this lemma, we must be able to compute the p_d values at least approximately. For now, we will assume that we have access to an oracle that will compute the value for us. Lemma 55 shows how to construct such an oracle. We also use the following lemmas to bound the deviation of the path.

▶ **Lemma 26.** *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.*

unconstrained random walks will not dip below $1 - k$ threshold whp

8.3.1 The Simple Case: Far Boundary

▶ **Lemma 27.** *Given a Dyck path sampling problem of length B with U up and D down steps with a bounary at k, there exists a constant c such that if $k > c\sqrt{B \log n}$, then the distribution of paths sampled without a boundary $\mathsf{C}_\infty(U, D)$ (hypergeometric sampling) is statistically $\mathcal{O}(1/n^2)$-close in L_1 distance to the distribution of Dyck paths $\mathsf{C}_k(U + D)$.*

By Lemma 27, 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 equivalent to the interval summable functions implemented in [GGN10] (using $\mathcal{O}(poly(\log n))$ resources).

8.3.2 Path Segments Close to Zero

A difficult case is when $k = \mathcal{O}(\sqrt{B \log n})$ and we need to compute the actual probability p_d, For the definitions in Section A.4, we see that:

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

Here, $k' = k + 2U - 2D$, and so $k' = \mathcal{O}(\sqrt{B \log n})$ (using Lemma 26). The distribution $\mathsf{C}_k(2U, 2D)$ we wish to sample from has probabilties $p_d = S_{left} \cdot S_{right}/S_{total}$. We will use rejection sampling (Lemma 25) 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 26). To invoke the rejection sampling lemma, we present a method to approximate the probabilities p_d in Lemma 55. The only thing left to do is to find an appropriate q_d that also has an *efficiently computable CDF*. Surprisingly, as in Section 8.3.1, we will be able to use the hypergeometric distribution for q_d,

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

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 $S_{right} < \binom{B}{U-d}$ along with the following lemma (proven in Section A).

▶ **Lemma 28.** *When $kk' > 2U+1$, $S_{total} > \frac{1}{2} \cdot \binom{2B}{2D}$.*

Combining the three bounds we obtain $p_d < \frac{1}{2}q_d$. 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 8.3.1). The case where the boundaries are closer (i.e. $k \cdot k' \leq 2U+1$) is trickier, since the individual counts need not be close to the corresponding binomial counts. However, in this case we can still ensure that the sampling probability is within poly-logarithmic factors of the binomial sampling probability. We use the following lemmas (proven in Section A).

▶ **Lemma 29.** *$S_{left} \leq c_1 \frac{k \cdot \sqrt{\log n}}{\sqrt{B}} \cdot \binom{B}{D-d}$ for some constant c_1.*

▶ **Lemma 30.** *$S_{right} < c_2 \frac{k' \cdot \sqrt{logn}}{\sqrt{B}} \cdot \binom{B}{U-d}$ for some constant c_2.*

▶ **Lemma 31.** *When $kk' \leq 2U+1$, $S_{total} < c_3 \frac{k \cdot k'}{B} \cdot \binom{2B}{2D}$ for some constant c_3.*

We can now put these lemmas together to show that $p_d/q_d \leq \Theta(\log n)$ and invoke Lemma 25 to sample the value of d. This gives us the height of the Dyck path at the midpoint of the two given points.

▶ **Theorem 32.** *Given two positions a and b (and the associated heights) in a Dyck path of length $2n$ with the guarantee that no position between a and b has been sampled yet, there is an algorithm that returns the height of the path halfway between a and b. Moreover, this algorithm only uses $\mathcal{O}(poly(\log n))$ resources.*

Proof. If $b-a$ is even, we can set $B = (b-a)/2$. Otherwise, we first sample a single step from a to $a+1$, and then set $B = (b-a-1)/2$. Since there are only two possibilities for a single step, we can explicitly approximate the probabilities, and then sample accordingly. This allows us to apply the rejection sampling from Lemma 25 using $\{q_d\}$ to obtain samples from $\{p_d\}$ as defined above. □

▶ **Theorem 33.** *There is an algorithm that provides sample access to a Dyck path of length $2n$, by answering queries of the form* Height(x) *with the correctly sampled height of the Dyck path at position x using only $\mathcal{O}(poly(\log n))$ resources per query.*

Proof. The algorithm maintains a successor-predecessor data structure (e.g. Van Emde Boas tree) to store all positions x that have already been sampled. Each newly sampled position is added to this structure. Given a query Height(x), the algorithm first finds the successor and predecessor (say a and b) of x among the alredy queried positions. This provides us the quarantee required to apply Theorem 32, which allows us to query the height at the midpoint of a and b. We then binary search by updating either the successor or predecessor of x and repeat until we sample the height of position x. □

8.4 Supporting "First Return" Queries

We also support more complex queries to a Dyck path. Specifically, in addition to querying the height after t steps, we introduce a First-Return query that allows the user to query the next time the path returns to that height (if at all i.e. only if the step from x to $x+1$ is an up-step).

The utility of this kind of query can be seen in other interpretations of Catalan objects. For instance, if we interpret it as a well bracketed expression, First-Return(x) returns the position of the bracket matching the one started at x. If we consider a uniformly random rooted tree, the function effectively returns the next child of a vertex (see Section 8.1).

8.4.1 Maintaining a Boundary Invariant

Notice that over a sequence of HEIGHT queries $\langle x_1, x_2, \cdots, x_m \rangle$ to the Dyck path, many different positions are revealed (possibly in adverserial locations). This partitions the path into at most $m+1$ disjoint and independent *intervals* with set boundary conditions. The first step towards finding the FIRST-RETURN from position t would be to locate the *interval* where the 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 inefficiency, the fact that a specific interval *does not* contain the first return causes dependencies for all subsequent samples.

We resolve this issue by maintaining the following invariant. 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 $\langle y_1, y_2, \cdots, y_m \rangle$.

▶ **Invariant 34.** *For any interval $[x_i, x_{i+1}]$ where the heights of the endpoints have been sampled to be y_i and y_{i+1}, and no other position in the interval has yet been sampled, the section of the Dyck path between positions x_i and x_{i+1} is constrained to lie above $min(y_i, y_{i+1})$.*

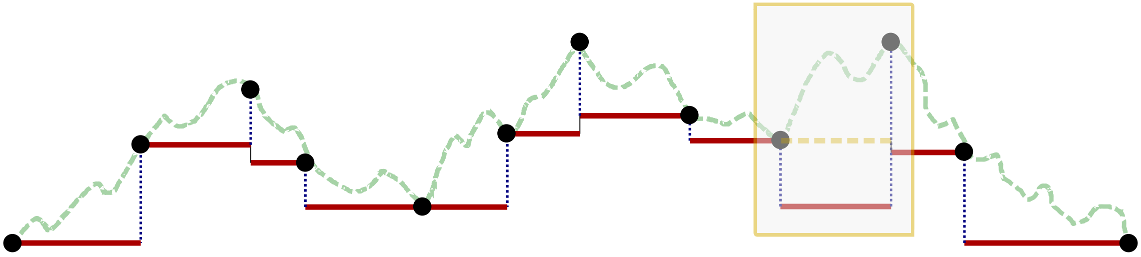

Figure 8 Error in third segment.

It's not even clear that it is always possible to maintain such an invariant. After sampling the height of a particular position x_i as y_i (with $x_{i-1} < x_i < x_{i+1}$), the invariant is potentially broken on either side of x_i. We will re-establish the invariant by sampling an additional point on either side. This proceeds as follows for the interval between x_i and x_{i+1} (see error in Figure 8):

1. Sample the lowest height h achieved by the walk between x_i and x_{i+1}.
2. Sample a position x such that $x_i < x < x_{i+1}$ and HEIGHT$(x) = h$.

Since h is the minimum height along this interval, sampling the point x suffices to preserve the invariant. Lemma 41 shows how this invariant can be used to efficiently find the interval containing the first return.

8.4.2 Sampling the Lowest Achievable Height in an Interval

For the first step, we need to sample the lowest height h of the walk between x_i and x_{i+1} (with $x_i < x_{i+1}$). Say that this interval defines a generalized Dyck problem with U up steps and D down steps with a boundary that is $k-1$ units below y_i.

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

k_{lower} boundary as $P(k_{lower}, k_{upper}) = C_{k_{lower}}(U,D) - C_{k_{upper}}(U,D)$. We set $k_{low} = k, k_{up} = 0$, and $k_{mid} = (k_{low} + k_{up})/2$. Since we can compute $P(k_{low}, k_{up})$, $P(k_{low}, k_{mid})$, and $P(k_{mid}, k_{up})$, 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.

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

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

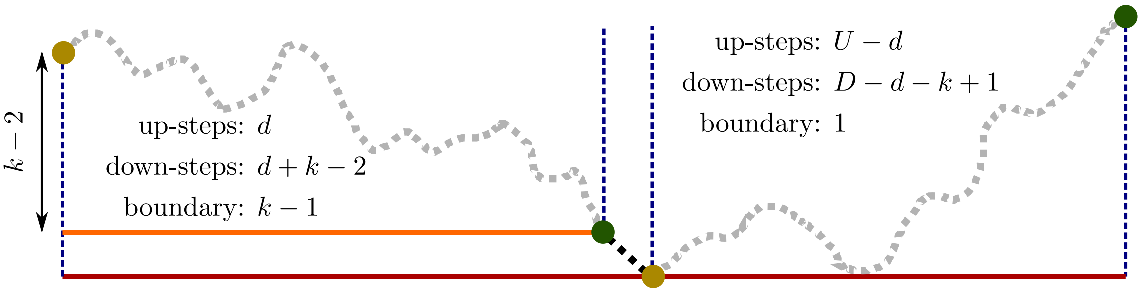

Figure 9 Zooming into the error in Figure 8

We will parameterize the position x the number of up-steps d between x_i and x (See Figure 9). 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. We will calculate this quantity by counting the total number of paths to the left and right of the first approach and multiplying them together.

Since we only care about getting a asymptotic (up to poly$(\log n)$ factors) estimate of the probabilities, it suffices to estimate the number of paths asymptotically as well. As in Section 8.3.2, we define S_{left} to be the number of paths in the sub-interval before the first approach (left side of Figure 9), S_{right} to be the number of paths following the first approach, and S_{total} to be the toal number of paths that touch the "mandatory boundary" at k:

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

Our goal is to sample d from the distribution $\{p_d\}$ where $p_d = S_{left} \cdot S_{right}/S_{total}$. Using Equation 1, we obtain the following approximations for S_{left} and S_{right}.

▶ **Lemma 35.** *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 36.** *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, $r_{right}(d) = \mathcal{O}(\log^2 n)$.*

Deal with other values of d

We now consider the 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), $d \geq 0$ and since the length of the right sub-interval nust be non-negative, we get $U + D - 2d - k + 1 \geq 0$. Thus, we define the set

$$\mathcal{R} = \left\{ d \;\middle|\; 0 \leq d < \log^4 n \textbf{ or } -1 < 2d - U - D + k < \log^4 n \right\} \tag{3}$$

Clearly, we can bound the size of this set as $|\mathcal{R}| = \mathcal{O}(\log^4 n)$. An immediate consequence of Lemma 35 and Lemma 36 is the following.

▶ **Corollary 37.** *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 $r(d) = \mathcal{O}(\log^2 n)$.*

8.4.4 Estimating the CDF

We will now use these observations to construct a suitable $\{q_d\}$ that can be used to invoke the rejection sampling lemma. In addition to beign a good approximation to $\{p_d\}$, $\{q_d\}$ must be of a form that allows us to compute its CDF efficiently. First, we rewrite the approximate probability as $p_d = \Theta\left(\mathcal{K} \cdot f(d) \cdot e^{-r(d)}\right)$ where:

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

Notice that $\mathcal{K}$ is a constant and $f(d)$ is a function whose integral has a closed form. Using the fact that $r(d) = \mathcal{O}(\log^2 n)$ (from Corollary 37), we obtain the following lemma:

▶ **Lemma 38.** *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 evaluateds 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 in $\hat{q}(\delta)$ is within a factor of e of the original $e^{-r(d)}$ term.

For all $\mathcal{O}(\log^4 n)$ values $d \in \mathcal{R}$, $\hat{q}(\delta)$ is constant on the interval $[d, d+1]$. □

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:

$$q_d = \frac{\int_d^{d+1} \hat{q}(\delta)}{\mathcal{N}} \qquad\qquad Q_d = \frac{\int_0^{d+1} \hat{q}(\delta)}{\mathcal{N}} \tag{4}$$

Here the normalizing factor $\mathcal{N}$ is $\int_0^{d_{max}+1} \hat{q}(\delta)$. To show that these can be computed efficiently, it suffices to show that any integral of $\hat{q}_d$ can be efficiently evaluated.

▶ **Lemma 39.** *Given the function $\hat{q}_d$ defined in Lemma 38, it is possible to compute the integral $\int\limits_{d_1}^{d_2+1} \hat{q}(\delta)$ in $\mathrm{poly}(\log n)$ time for any valid d_1, d_2 (i.e. the bounds must be such that $d_i \geq 0$ and $U + D - 2d - k + 1 \geq 0$).*

Proof. As noted in Lemma 38, $\hat{q}(\delta)$ is piecewise continuous function with $\mathcal{O}(\log^4 n)$ pieces. For all $\mathcal{O}(\log^4 n)$ values $d \in \mathcal{R}$, $\hat{q}(\delta)$ is constant on the interval $[d, d+1]$. Now consider values We claim that the number of continuous segments in the *valid* range is $\mathcal{O}(\log^4 n)$. □

▶ **Theorem 40.** *Kinesthetics*

Needs a theorem

8.4.5 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 $\langle y_1, y_2, \cdots, y_m \rangle$.

▶ **Lemma 41.** *For any position x_i, assuming that Invariant 34 holds, we can find the interval $(x_{k-1}, x_k]$ that contains* First-Return(x_i) *We do this by by setting k to be either the smallest index $f > i$ such that $y_f \leq y_i$ or setting $k = i + 1$.*

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 34 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. Now the challenge is to find the smallest index $f > i$ such that $y_f \leq y_i$. One solution is to maintain an interval tree over the range $[2n]$ storing the position of the boundary. Specifically, we have a balanced binary tree with $2n$ leaves with the i^{th} leaf storing the boundary at position i. Each internal node stores the minimum value amongst all the leaves in its sub-tree. In this setting, we can binary search for f by guessing a bound f' and performing a *range minimum query* over the interval $(x_i, x_{f'}]$. Overall, this requires $\mathcal{O}(\log n)$ range queries each of which makes $\mathcal{O}(\log n)$ probes to the binary tree.

cite

However, we cannot explicitly maintain or even construct this tree, and updates can be as expensive as $\Theta(n)$. To mitigate this, we start with just a root node (indicating that the initial boundary is 1 everywhere) and build the tree dynamically as needed. We perform updates using *lazy propagation* by only propagating updates down to the children (creating children if necessary) when needed. So, at any given time, some nodes in the tree may not hold the correct value, but the correct value must be present on the path to the root.

cite

This probably needs more explanation

▶ **Theorem 42.** *There is an algorithm using $\mathcal{O}(poly(\log n))$ resources per query that provides sample access to a Dyck path of length $2n$ by answering queries of the form* First-Return(x_i) *with the correctly sampled position y; where $y > x_i$ is the position where the Dyck path first returns to* Height(x_i) *after position x_i.*

Proof. We first query the interval $(x_i, x_{i+1}]$ to find a first return using Theorem 40. If a return is not found, we calculate f using . Since $x_{f-1} < x_i \leq x_f$ by definition, the interval $(x_{f-1}, x_f]$ must contain a position at height y_i. We sample a point in the middle of this interval and fix the boundary invariant by sampling another point, essentially breaking it up into $\mathcal{O}(1)$ sub-intervals

what?

each at most half the size of the original. Based on the new samples, we find the sub-interval containing the first return in $\mathcal{O}(1)$ time. We repeat up to $\mathcal{O}(\log n)$ times until the current interval size drops below the threshold $\mathcal{T}$. Then we spend $\tilde{\mathcal{O}}(\mathcal{T})$ time to brute force sample this interval and find the first return position (if it wasn't revealed in previous steps). □

9 Domino Tiling the $2 \times n$ grid

We will consider the problem of tiling a square grid with dominos. This problem has a long history and various importtant applications in statistical physics. Specifically, we will focus on the local generation of domino tilings of a $2 \times n$ grid from the uniform distribution. The queries will be as an index, and the generator should report the orientation of the domino at the i^{th} position in the grid.

cite

It is a well known result that the number of tilings of a $2 \times n$ grid is exactly F_n. To aid with generalization, we will instead allow the generator to respond with the splitting boundary of the current tiling instead. For example, in Figure ??, the boundary is a vertical line at the specified position. In Figure ??, the boundary is horizontal, indicating that there are two horizontal dominos at that location. Note that Figure ?? is impossible for a $2 \times n$ grid. It should be clear that this query model is equivalent.

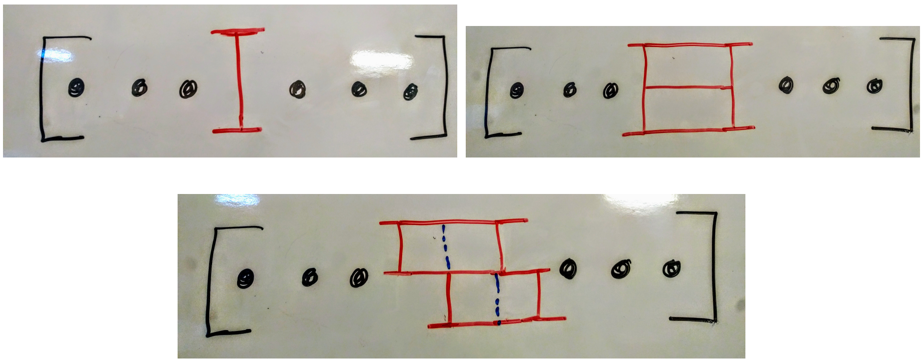

Figure 10 Caption for this figure with two images

Now, consider a query to the location i, such that all positions between $i - a$ and $i + b$ have not been queriesd so far. So, there is a blank $2 \times (a + b)$ size sub-grid that we have to sample from. Let us consider the number of possible tilings resulting from each possible splitting boundary.

1. Vertical Boundary – This indicates that we divide the region into two sub-grids with sizes $2 \times a$ and $2 \times b$. So, the total number of possible tilings is exactly $F_a \cdot F_b$.
2. Horizontal Boundary – This indicates that we divide the region into two sub-grids with sizes $2 \times (a - 1)$ and $2 \times (b - 1)$. So, the total number of possible tilings is exactly $F_{a-1} \cdot F_{b-1}$.

So the probabilities are computed as $\frac{F_a \cdot F_b}{F_a \cdot F_b + F_{a-1} \cdot F_{b-1}}$ and $\frac{F_{a-1} \cdot F_{b-1}}{F_a \cdot F_b + F_{a-1} \cdot F_{b-1}}$. Now, we face the issue of approximating these fractions. If either of the values a or b are less than $\Theta(\sqrt{n})$, then we can compute the exact value of the corresponding F_a or F_b. Otherwise, we use Lemma **??** to approximate $F_a = \phi \cdot F_{a-1}$ and $F_b = \phi \cdot F_{b-1}$. So, the probability of the vertical boundary becomes

$$\frac{F_a \cdot F_b}{F_a \cdot F_b + F_{a-1} \cdot F_{b-1}} = \frac{\phi^2}{\phi^2 + 1}$$

Similarly, the probability of a horizontal split with a top and bottom domino becomes $1/(\phi^2+1)$. Note that this also determines the two adjacent boundaries.

The only information we needed to make this query was the extent of the un-queried interval $[i-a, i+b]$. We can use any standard data-structure that allows insertion in positions $\{1, 2 \cdots, n+1\}$, and provides successor and predecessor queries.

Here's we can be fancy and use Van-Emde-Boas trees to get a $\mathcal{O}(\log \log n)$ query time. However, in some cases, the exact value of a Fibonacci number still needs to be computed, and this takes $\mathcal{O}(\log n)$ time. The faster queries only work when the new query is "far enough" ($\mathcal{O}(\log n)$ distance) away from all previous queries.

10 Random Coloring of a Graph

Query access

We wish to locally sample an uniformly random coloring of a graph. A q-coloring of a graph $G = (V, E)$ is a function $\sigma : V \to [q]$, such that for all $(u, v) \in E$, $\sigma_u \neq \sigma_v$. We will consider only bounded degree graphs, i.e. graphs with max degree $\leq \Delta$. Otherwise, the coloring problem becomes NP-hard.

cite

Using the technique of path-coupling, Vigoda showed that for $q > 2\Delta$, one can sample an uniformly random coloring by using a MCMC algorithm.

cite

The Markov Chain proceeds in T steps. The state of the chain at time t is given by $\mathbf{X}^t \in [q]^{|V|}$. Specifically, the color of vertex v at step t is $\mathbf{X}_v^t$.

In each step of the Markov process, a pair $(v, c) \in V \times [q]$ is sampled uniformly at random. Subsequently, if the recoloring of vertex v with color c does not result in a conflict with v's neighbors, i.e. $c \notin \{X_u^t : u \in \Gamma(v)\}$, then the vertex is recolored i.e. $X_v^{t+1} \leftarrow c$.

After running the MC for $T = \mathcal{O}(n \log n)$ steps we reach the stationary distribution (ϵ close), and the coloring is an uniformly random one.

Exact Bound: $t_{mix}(\epsilon) \leq \left(\frac{q-\Delta}{q-2\Delta}\right) n \left(\log n + \log(1/\epsilon)\right)$

cite book (Peres, Lyons)

10.1 Modified Glauber Dynamics

Now we define a modified Markov Chain as a special case of the *Local Glauber Dynamics* presented in [FG18]. The modified Markov chain procceds in epochs. We denote the initial coloring of the graph by $\mathbf{X}^0$ and the state of the coloring after the k^{th} epoch by $\mathbf{X}^k$. In the k^{th} epoch $\mathcal{E}_k$:

- Sample $|V|$ colors $\langle c_1, c_2, \cdots, c_n \rangle$ from $[q]$, where c_v is the proposed color for vertex v.
- For each vertex v, we set $\mathbf{X}_v^k$ to c_v if for all neighbors w of v, $\mathbf{X}_w^k \neq c_v$ and $\mathbf{X}_w^{k-1} \neq c_v$.

This procedure is a special case of the *Local Glauber Dynamics* presented in [FG18]. The goal in [FG18] is to find a simultaneous update rule that causes few conflicts among neighbors (and converges to the correct distribution). Notice that we *can* have adjacent nodes update in the same epoch. However for the sake of succinctness we use their update rule and avoid a tedious path coupling argument.

Cite Path Coupling

We can directly use the path coupling argument from [FG18] which be briefly describe below. Given two colorings $\mathbf{X}$ and $\mathbf{Y}$, we define $d(\mathbf{X}, \mathbf{Y})$ as the number of vertices v such that $\mathbf{X}_v \neq \mathbf{Y}_v$. We define the coupling $(\mathbf{X}, \mathbf{Y}) \to (\mathbf{X}', \mathbf{Y}')$ where $\mathbf{X}$ and $\mathbf{Y}$ differ only at a single vertex v such that

$\mathbf{X}_v = c_X$ and $\mathbf{Y}_v = c_Y$. Now, we pick a random permutation of the vertices along with uniformly sampled colors:

$$\langle (v_1, c_1), (v_2, c_2), \cdots, (v_n, c_n) \rangle = \langle (\pi_1, c_1), (\pi_2, c_2), \cdots, (\pi_n, c_n) \rangle$$

Now, for each (v_i, c_i) in order, we update the coloring of X and Y as follows:

- If the current color of v_i as well as c_i are both in $\{c_X, c_Y\}$, then the $\mathbf{X}$ chain picks the color c_i and the $\mathbf{Y}$ chain picks the other color.
- Otherwise, both chains pick the same color c_i for the vertex v_i.

We use the following result from [FG18] that bounds the coupled distance.

▶ **Lemma 43.** *If $q = 2\alpha\Delta$ and $d(\mathbf{X}, \mathbf{Y}) = 1$, then $\mathbb{E}[d(\mathbf{X}', \mathbf{Y}')] \le 1 - \left(1 - \frac{1}{2\alpha}\right) e^{-3/\alpha} + \frac{1/2\alpha}{1-1/\alpha}$*

▶ **Corollary 44.** *If $q \ge 9\Delta$ and $d(\mathbf{X}, \mathbf{Y}) = 1$, then $\mathbb{E}[d(\mathbf{X}', \mathbf{Y}')] < \frac{1}{e^{1/3}}$*

▶ **Theorem 45.** *If $q \ge 9\Delta$, then the chain is mixed after $\tau_{mix}(\epsilon) = 3\left(\ln n + \ln(\frac{1}{\epsilon})\right)$ epochs.*

Proof. Starting for a maximum distance of n, the distance decreases to 1 after at most $3 \ln n$ epochs, and it takes a further $3 \ln\left(\frac{1}{\epsilon}\right)$ to reduce the distance to ϵ. □

10.2 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 of v after running $t = \mathcal{O}(\ln n)$ epochs of *Modified Glauber Dynamics*. 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 $\mathbf{C}^t \sim_{\mathcal{U}} [q]^n$. Note that these values are fully independent and as such any $\mathbf{C}_v^t$ can be sampled trivially. We also use $\mathbf{X}^t$ to denote the final vector of vertex colors at the end of the t^{th} epoch. Finally, we define indicator variables $\boldsymbol{\chi}_v^t$ to denote if the color for vertex v was accepted at the t^{th} epoch; $\boldsymbol{\chi}_v^t = 1$ if and only if for all neighbors $w \in \Gamma(v)$, we satisfy the condition $\mathbf{C}_v^t \ne \mathbf{X}_w^{t-1}$ and $\mathbf{C}_v^t \ne \mathbf{C}_w^t$. So, the color of a vertex v after the t^{th} epoch $\mathbf{X}_v^t$ is set to be $\mathbf{C}_v^i$ where $i \le t$ is the largest index such that $\boldsymbol{\chi}_v^i = 1$. While the proposals $\mathbf{C}_v^t$ are easy to sample, it is much less clear how we can sample the $\boldsymbol{\chi}_v^t$ values. Note that we can compute $\mathbf{X}_v^t$ quite easily if we know the values $\boldsymbol{\chi}_v^i$ for all $i \le t$. So, we focus our attention on the query $\textsc{Accept}(v, t)$ that returns $\boldsymbol{\chi}_v^t$.

10.2.1 Naive Coloring Implementations

The general strategy to implement this is to iterate over all neighbors w of v, and for each of them check if they conflict with v's proposed color. Given a neighbor w, one naive way to do this is to iterate backwards from epoch t querying to find if w's proposal was accepted until the first accepted proposal (from the latest epoch $t' < t$) is found. At this point, if $C_w^{t'} = C_v^t$, then the current color of w conflicts with v's proposal. Otherwise there is no conflict and we can proceed to the next neighbor. THis process however makes Δ recursive calls to a sub-problem that is only slightly smaller i.e. $T(t) \le \Delta \cdot T(t-1)$. This leads to a running time upper bound of Δ^t which is superlinear for the desired $t = \Omega(\log n)$.

We can prune the number of recursive calls by only processing the neighbors w which actually proposed the color C_v^t during *some* epoch. In this case, the expected number of neighbors that

Algorithm 11 Generator

```
procedure ACCEPT($v, t$)
    $c \leftarrow C_v^t$
    for $w \leftarrow \Gamma(v)$
        if $C_w^t = c$
            return 0
        for $t' \leftarrow [t, t-1, t-2, \cdots, 1]$
            if $C_w^{t'} = c$ and ACCEPT($w, t'$)
                $flag \leftarrow 1$
                while $t' < t - 1$
                    $t' \leftarrow t' + 1$
```

have to be probed recursively is $\leq t\Delta/q$ (since the total number of neighbor proposals over t epochs is at most $t\Delta$). So, the overall runtime is upper bounded by $(t\Delta/q)^t$. For this algorithm, if we allow $q > t\Delta = \Omega(\Delta \log n)$ colors, the runtime becomes sublinear. This lower bound on q is however asymptotically worse that the sequential requirement $q > 2\Delta = \mathcal{O}(\Delta)$.

10.2.2 Jumping Back to Past Epochs

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

Algorithm 11 shows our final procedure for sampling $\boldsymbol{\chi}_v^t$ where $c = C_v^t$ is the color proposed by v in epoch t. As before, we iterate through all neighbors w of v. The condition $c \neq \mathbf{C}_w^t$ is can easily be checked by sampling $\mathbf{C}_w^t$ in the current epoch. If no conflict is seen, the next step is to check whether $c \neq \mathbf{X}_w^{t-1}$.

To achieve this, we iterate through all the epochs in reverse order (without making recursive calls) to check whether the color c was ever proposed for vertex w. If not, we can ignore w, and otherwise let's say that the last 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. If the proposal $C_w^{t'}$ was not accepted, we keep iterating back until we find another candidate proposal for color c or we run out of epochs. Otherwise if $\boldsymbol{\chi}_w^{t'} = 1$ (proposal accepted), we move to epoch $t'+1$ to see if w's color was replaced. If not, we check epoch $t'+2$, $t'+3$, and so on until we reach epoch $t-1$. At this point we have seen that $\boldsymbol{\chi}_w^{t'} = 1$ (color c was accepted) and every subsequent proposal until the current epoch was rejected i.e. $\mathbf{X}_w^{t-1} = c$ and this leads to a conflict with v's current proposal for color c and hence $\boldsymbol{\chi}_v^t = 0$. If at any of the iterations, we see that a different proposal was accepted, then w does not cause a conflict and we can move on to the next neighbor. If we exhaust all the neighbors and don't find any conflicts then $\boldsymbol{\chi}_v^t = 1$.

Now we analyze the runtime of Accept by constructing and solving a recurrence relation. We will use the following lemma to evaluate the expectation of products of relevant random variables.

▶ **Lemma 46.** *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 $\{C_w^t, X_w^{t-1} | w \in \Gamma(v)\}$. Since there are $2\alpha\Delta$ colors, the rejection probability is at most $1/\alpha$. □

▶ **Definition 47.** *We define T_t to be a random variable indicating the number of recursive calls performed during the execution of Accept(v, t) while sampling a single $\boldsymbol{\chi}_v^t$.*

What probes?

So, the number of probes required to check whether a color c (assigned at epoch t') was overwritten at some epoch before t is:

$$\left[T_{t'+1} + \mathcal{B}\left(\frac{1}{\alpha}\right) \cdot T_{t'+2} + \mathcal{B}\left(\frac{1}{\alpha^2}\right) \cdot T_{t'+3} + \cdots + \mathcal{B}\left(\frac{1}{\alpha^{t-t'-2}}\right) \cdot T_{t-1}\right] \tag{5}$$

What probes? Given graph G and q colors ...

▶ **Lemma 48.** *For $\alpha > 4.5$, the expected number of calls to the procedure* Accept *while sampling a single $\boldsymbol{\chi}_v^t$ is $\mathbb{E}[T_t] = \mathcal{O}\left(e^{1.02t/\alpha}\right)$.*

Proof. We start with the recurrence for the expected number of probes to $\{\boldsymbol{\chi}^{t'}\}_{t' \in [t]}$ (equivalently calls to Accept) used by the algorithm. We will use $\mathcal{B}(p)$ to refer to the Bernoulli random variable with bias p. When checking a single neighbor w, the algorithm iterates through all the epochs t' such that $\mathbf{C}_w^{t'} = c$ (in reality, only the last occurence matters, but we are looking for an upper bound). If such a t' is found (this happens with probability $1/q$ independently for each trial), there is one recursive call to $T_{t'}$. Regardless of what happens, let's say the algorithm queries $T_{t'+1}, T_{t'+2}, \cdots, T_{t-1}$ until an Accept proposal is found. Adding an extra $T_{t'}$ term to Equation 5 and summing up over all neighbors and epochs we get the following:

$$\begin{aligned} T_t \leq \Delta \cdot \sum_{t'=1}^{t} \mathbb{P}[C_w^{t'} = c] \cdot \Bigg[T_{t'} + T_{t'+1} + \mathcal{B}\left(\frac{1}{\alpha}\right) \cdot T_{t'+2} + \mathcal{B}\left(\frac{1}{\alpha^2}\right) \cdot T_{t'+3} + \cdots \qquad && (6) \\ \cdots + \mathcal{B}\left(\frac{1}{\alpha^{t-t'-2}}\right) \cdot T_{t-1} \Bigg] && (7) \\ \leq \Delta \cdot \mathcal{B}\left(\frac{1}{q}\right) \left[\sum_{t'=1}^{t-1} T_{t'} + \sum_{t'=1}^{t-1} T_{t'} \cdot \left(1 + \mathcal{B}\left(\frac{1}{\alpha}\right) + \mathcal{B}\left(\frac{1}{\alpha^2}\right) + \cdots \right)\right] && (8) \end{aligned}$$

In the second step, we just group all the terms from the same epoch together. Using Lemma 46 and the fact that $\mathbb{P}[C_w^{t'} = c]$ is independent of all other events, we can write a recurrence for the expected number of probes.

$$\mathbb{E}[T_t] \leq \Delta \cdot \frac{1}{2\alpha\Delta} \left[\sum_{t'=1}^{t-1} T_{t'} + \sum_{t'=1}^{t-1} T_{t'} \cdot \left(1 + \frac{1}{\alpha} + \frac{1}{\alpha^2} + \cdots \right)\right] \leq \frac{1}{2\alpha} \cdot \sum_{t'=1}^{t-1} T_{t'} \cdot \left[1 + \frac{\alpha}{\alpha - 1}\right] \tag{9}$$

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 $\mathcal{O}(e^{1.02t/\alpha})$.

Finally, we note that each probe potentially takes time $\mathcal{O}(t\Delta)$ to iterate through all the neighbors in all epochs resulting in a total runtime of $\mathcal{O}(t\Delta e^{kt/\alpha})$. □

▶ **Theorem 49.** *Given adjacency list query access to a graph with n nodes, maximum degree Δ, and $q = 2\alpha\Delta \geq 9\Delta$ colors, we can sample the color of any given node in an ($1/n$-approximate) uniformly random coloring of the graph in a consistent manner using only $\mathcal{O}(n^{6.12/\alpha}\Delta \log n)$ time space and random bits. This is sublinear for $\alpha > 6.12$ and the sampled coloring is $1/n$-close to the uniform distribution in L_1 distance.*

Proof. We compute the mixing time from Theorem 45 to obtain $\tau_{mix}(1/n) = 6 \ln n$ (this is valid since $q > 9\Delta$). Since $\alpha > 4.5$, we can invoke Lemma 48 to conclude that the number of calls to Accept is $\mathcal{O}(n^{6.12/\alpha}\Delta \log n)$ which is sublinear for $\alpha > 6.12$. Each call to Accept(v, t) potentially spends $t\Delta$ time looking for neighbors in each epoch before t. Since $t \leq 6 \ln n$, the overall runtime becomes $\mathcal{O}(n^{6.12/\alpha}\Delta \log n)$. □

11 Open Problems

- Degree queries for undirected random graphs?
- Faster implementation of coloring $\mathcal{O}(\text{poly}(\log n))$?
- Reduce the required value of α?
- Random walks on other networks? Ideally, any network.

References

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

ELMR17 Guy Even, Reut Levi, Moti Medina, and Adi Rosén. Sublinear random access generators for preferential attachment graphs. In *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, pages 6:1–6:15, 2017.

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

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

GGN03 O Goldreich, S Goldwasser, and A Nussboim. On the implementation of huge random objects. In *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*, pages 68–79. IEEE, 2003.

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

Kle00 Jon Kleinberg. The small-world phenomenon: An algorithmic perspective. In *Proceedings of the thirty-second annual ACM Symposium on Theory of Computing*, pages 163–170. ACM, 2000.

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

MN04 Chip Martel and Van Nguyen. Analyzing kleinberg's (and other) small-world models. In *Proceedings of the twenty-third annual ACM Symposium on Principles of Distributed Computing*, pages 179–188. ACM, 2004.

MNS15 Elchanan Mossel, Joe Neeman, and Allan Sly. Reconstruction and estimation in the planted partition model. *Probability Theory and Related Fields*, 162(3-4):431–461, 2015.

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

NR02 Moni Naor and Omer Reingold. Constructing pseudo-random permutations with a prescribed structure. *Journal of Cryptology*, 15(2):97–102, 2002.

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

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

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

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

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

A Dyck Path Generator

▶ **Theorem 50.** *There are $\frac{1}{n+1}\binom{2n}{n}$ Dyck paths for length $2n$ (construction from [Sta15]).*

Proof. 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 $\frac{1}{n+1}\binom{2n}{n}$. □

A.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:

$$\log(m!) = m\log m - m + \frac{1}{2}\log(2\pi m) + \frac{1}{12m} - \frac{1}{360m^3} + \frac{1}{1260m^5} - \cdots \tag{10}$$

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

▶ **Lemma 51.** *The central binomial coefficient can be approximated as:*

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

Cite Asymptopia

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

▶ **Lemma 52.** *Proof from [Spe14]*

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

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

$$R = \frac{\binom{n}{k}}{\binom{n}{n/2}} = \frac{(n/2)!(n/2)!}{k!(n-k)!} = \prod_{i=1}^{c\sqrt{n}/2}\frac{n/2-i+1}{n/2+i} \tag{11}$$

$$\implies \log R = \sum_{i=1}^{c\sqrt{n}/2}\log\left(\frac{n/2-i+1}{n/2+i}\right) \tag{12}$$

$$= \sum_{i=1}^{c\sqrt{n}/2} -\frac{4i}{n} + \mathcal{O}\left(\frac{i^2}{n^2}\right) = -\frac{c^2 n}{2n} + \mathcal{O}\left(\frac{(c\sqrt{n})^3}{n^2}\right) = -\frac{c^2}{2} + \mathcal{O}\left(\frac{c^3}{\sqrt{n}}\right) \tag{13}$$

$$\implies \binom{n}{k} = \binom{n}{n/2}e^{-c^2/2}exp\left(\mathcal{O}(c^3/\sqrt{n})\right) \tag{14}$$

□

A.2 Dyck Path Boundaries and Deviations

▶ **Lemma 53.** *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 $\sum\limits_{i=1}^{2n} X_i = n$. Here, 1 corresponds to an up-step and 0 corresponds to a down step. Because of the constraint, X_i, X_j are negatively correlated for $i \neq j$ which allows us to apply Chernoff bounds. Now we consider a sub-path of length $2B$ and let U denote the sum of the X_is associated with this subpath. Using Chernoff bound with $\mathbb{E}[X] = B$, we get:

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

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

▶ **Corollary 54.** *With high probability, every contiguous sub-path in the random walk (with U up and D down steps) satisfies the property with high probability. Specifically, if $U + D = 2B$, then $|B - U|$ and $|B - D|$ are upper bounded by $c\sqrt{B \log n}$ w.h.p. $1 - 1/n^2$ for all contiguous sub-paths (for some constant c).*

Proof. We can simply apply Lemma 53 and union bound over all n^2 possible contiguous sub-paths. □

▶ **Lemma 26.** *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 50, 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 54 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 27.** *Given a Dyck path sampling problem of length B with U up and D down steps with a bounary at k, there exists a constant c such that if $k > c\sqrt{B \log n}$, then the distribution of paths sampled without a boundary $\mathsf{C}_\infty(U, D)$ (hypergeometric sampling) is statistically $\mathcal{O}(1/n^2)$-close in L_1 distance to the distribution of Dyck paths $\mathsf{C}_k(U + D)$.*

Proof. We use $\mathcal{D}$ and $\mathcal{R}$ to denote the set of all valid Dyck paths and all random sequences respectively. Clearly, $\mathcal{D} \subseteq \mathcal{R}$. Let c be a constant satisfying Corollary 54. Since the random walk/sequence distribution is uniform on $\mathcal{R}$, and by Corollary 54 we see that at least $1 - 1/n^2$ fraction of the elements of $\mathcal{R}$ do not violate the boundary constraint. Therefore, $|\mathcal{D}| \geq (1 - 1/n^2)|\mathcal{R}|$ and so the L_1 distance between $\mathcal{U}_\mathcal{D}$ and $\mathcal{U}_\mathcal{R}$ is $\mathcal{O}(1/n^2)$. □

A.3 Computing Probabilities

Oracle for estimating probabilities:

▶ **Lemma 55.** *Given a Dych sub-path problem within a global Dyck path of size $2n$ and a probability expression of the form $p_d = \frac{S_{left} \cdot S_{right}}{S_{total}}$, there exists a* $\mathrm{poly}(\log n)$ *time oracle that returns a $\left(1 \pm 1/n^2\right)$ multiplicative approximation to p_d if $p_d = \Omega(1/n^2)$ and returns 0 otherwise.*

Point to section referencing the left right/ total.

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 10, 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 10 up to $\mathcal{O}(\log n)$ terms. The additive error can also be cast as multiplicative since factorials are large positive integers.

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

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

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

A.4 Sampling the Height

fix

- $d < c \cdot \sqrt{B} \log n$
- $k < c \cdot \sqrt{B} \log n \implies U - D < c \cdot \sqrt{B} \log n$
- $k' < c \cdot \sqrt{B} \log n$
- $B > \log^2 n \implies \sqrt{B} \log n < B$

▶ **Lemma 56.** *For $x < 1$ and $k \geq 1$,*

$$1 - kx < (1 - x)^k < 1 - kx + \frac{k(k-1)}{2} x^2.$$

▶ **Lemma 29.** *$S_{left} \leq c_1 \frac{k \cdot \sqrt{\log n}}{\sqrt{B}} \cdot \binom{B}{D-d}$ for some constant c_1.*

Proof. This involves some simple manipulations.

$$
\begin{aligned}
S_{left} &= \binom{B}{D-d} - \binom{B}{D-d-k} \qquad (15)\\
&= \binom{B}{D-d} \cdot \left[1 - \frac{(D-d)(D-d-1)\cdots(D-d-k+1)}{(B-D-d+k)(B-D-d+k-1)\cdots(B-D-d+1)}\right] \qquad (16)\\
&\le \binom{B}{D-d} \cdot \left[1 - \left(\frac{D-d-k+1}{B-D+d+k}\right)^k\right] \qquad (17)\\
&\le \binom{B}{D-d} \cdot \left[1 - \left(\frac{U+d+k-(U-D+d+k-1)}{U+d+k}\right)^k\right] \qquad (18)\\
&\le \binom{B}{D-d} \cdot \left[1 - \left(\frac{U+d+k-\mathcal{O}(\sqrt{B\log n})}{U+d+k}\right)^k\right] \qquad (19)\\
&\le \Theta\left(\frac{k\sqrt{\log n}}{\sqrt{B}}\right) \cdot \binom{B}{D-d} \qquad (20)
\end{aligned}
$$

□

▶ **Lemma 30.** *$S_{right} < c_2 \frac{k' \cdot \sqrt{logn}}{\sqrt{B}} \cdot \binom{B}{U-d}$ for some constant c_2.*

Proof.

$$
\begin{aligned}
S_{right} &= \binom{B}{U-d} - \binom{B}{U-d-k'} \qquad (21)\\
&= \binom{B}{U-d} \cdot \left[1 - \frac{(U-d)(U-d-1)\cdots(U-d-k'+1)}{(B-U-d+k')(B-U-d+k'-1)\cdots(B-U-d+1)}\right] \qquad (22)\\
&\le \binom{B}{U-d} \cdot \left[1 - \left(\frac{U-d-k'+1}{B-U+d+k'}\right)^{k'}\right] \qquad (23)\\
&\le \binom{B}{U-d} \cdot \left[1 - \left(\frac{2D-U-d-k+1}{2U-D+k+d}\right)^{k'}\right] \qquad (24)\\
&\le \binom{B}{U-d} \cdot \left[1 - \left(\frac{U+k+d-(2U-2D+2d+2k-1)}{U+k+d}\right)^{k'}\right] \qquad (25)\\
&\le \binom{B}{U-d} \cdot \left[1 - \left(\frac{U+k+d-\mathcal{O}(\sqrt{B\log n})}{U+k+d}\right)^{k'}\right] \qquad (26)\\
&\le \Theta\left(\frac{k'\sqrt{\log n}}{\sqrt{B}}\right) \cdot \binom{B}{U-d} \qquad (27)
\end{aligned}
$$

□

change statement

▶ **Lemma 57.** $S_{tot} \ge \binom{2B}{2D} \cdot \left[1 - \left(1 - \frac{k'}{2U+1}\right)^k\right]$.

Proof.

$$\begin{aligned}
S_{tot} &= \binom{2B}{2D} - \binom{2B}{2D-k} && (28)\\
&= \binom{2B}{2D} \cdot \left[1 - \frac{(2D)(2D-1)\cdots(2D-k+1)}{(2B-2D+k)(2B-2D+k-1)\cdots(2B-2D+1)}\right] && (29)\\
&\geq \binom{2B}{2D} \cdot \left[1 - \left(\frac{2D-k+1}{2B-2D+1}\right)^k\right] && (30)\\
&\geq \binom{2B}{2D} \cdot \left[1 - \left(\frac{2U-(2U-2D+k-1)}{2U+1}\right)^k\right] && (31)\\
&\geq \binom{2B}{2D} \cdot \left[1 - \left(\frac{(2U+1)-k'}{2U+1}\right)^k\right] && (32)\\
&\geq \binom{2B}{2D} \cdot \left[1 - \left(1 - \frac{k'}{2U+1}\right)^k\right] && (33)\\
& && (34)
\end{aligned}$$

□

Reference previous lemma

▶ **Lemma 28.** *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 31.** *When $kk' \leq 2U+1$, $S_{total} < c_3 \frac{k \cdot k'}{B} \cdot \binom{2B}{2D}$ 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 Taylor expansion, we see that

$$\begin{aligned}
&1 - \left(1 - \frac{k'}{2U+1}\right)^k && (35)\\
&\leq \frac{kk'}{2U+1} - \frac{k(k-1)}{2} \cdot \frac{k'^2}{(2U+1)^2} && (36)\\
&\leq \frac{kk'}{2U+1} - \frac{k^2k'^2}{2(2U+1)^2} && (37)\\
&\leq \frac{kk'}{2U+1}\left(1 - \frac{kk'}{2(2U+1)}\right) && (38)\\
&\leq \frac{kk'}{2(2U+1)} \leq \frac{kk'}{\Theta(B)} && (39)\\
& && (40)
\end{aligned}$$

□

A.5 First Return Sampling

▶ **Corollary 37.** *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 $r(d) = \mathcal{O}(\log^2 n)$.*

Proof. This follows from the fact that both $r_{left}(d)$ and $r_{right}(d)$ are $\mathcal{O}(\log^2 n)$. □

▶ **Lemma 35.** *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 9 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 26, 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 52 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 36.** *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 26, 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 52 to get:

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

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

□