


Miscellaneous Results

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

Our results focus on undirected graphs with independent edge probabilities, that is, each edge is chosen as an independent Bernoulli random variable. We provide a general implementation for generators in this model. Then, we use this construction to obtain the first efficient local implementations for the Erdős-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.

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

Keywords and phrases Dummy keyword

Funding *Amartya Shankha Biswas*: funding

Ronitt Rubinfeld: funding

Anak Yodpinyanee: funding

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 2 | Our Contributions and Techniques | 4 |
| 2.1 | Undirected Graphs | 4 |
| 2.2 | Directed Graphs | 6 |
| 3 | Preliminaries | 6 |
| 3.1 | Local-Access Generators | 6 |
| 3.2 | Random Graph Models | 7 |
| 3.3 | Miscellaneous | 8 |
| 4 | Local-Access Generators for Random Undirected Graphs | 8 |
| 4.1 | Naïve Generator with an Explicit Adjacency Matrix | 9 |
| 4.2 | Improved NEXT-NEIGHBOR Queries via Run-of-0's Sampling | 10 |
| 4.2.1 | Data structure | 11 |
| 4.2.2 | Queries and Updates | 12 |
| 4.3 | Final Generator via the Bucketing Approach | 13 |
| 4.3.1 | Partitioning into buckets | 13 |
| 4.3.2 | Filling a bucket | 14 |
| 4.3.3 | Putting it all together: RANDOM-NEIGHBOR queries | 15 |
| 4.4 | Implementation of FILL | 16 |
| 4.5 | Removing the Perfect-Precision Arithmetic Assumption | 18 |
| 5 | Applications to Erdős-Rényi Model and Stochastic Block Model | 19 |
| 5.1 | Erdős-Rényi Model | 20 |
| 5.2 | Stochastic Block model | 20 |
| 5.2.1 | Sampling from the Multivariate Hypergeometric Distribution | 21 |
| 5.2.2 | Data structure | 22 |
| 6 | Sampling Catalan Objects | 23 |
| 6.1 | Catalan Trapezoids and Generalized Dyck Paths | 23 |
| 6.2 | Generating Dyck Paths | 23 |
| 6.2.1 | The Simple Case | 25 |
| 6.2.2 | Path Segments Close to Zero | 26 |
| 6.3 | Supporting “First Return” Queries | 27 |
| 6.3.1 | Maintaining a Boundary Invariant | 27 |
| 6.3.2 | Sampling the Lowest Achievable Height | 28 |
| 6.3.3 | Sampling the Position of First Return | 28 |
| 6.3.4 | Estimating the CDF | 29 |
| A | Dyck Path Generator | 31 |
| A.1 | Approximating Close-to-Central Binomial Coefficients | 31 |
| A.2 | Dyck Path Boundaries and Deviations | 31 |
| A.3 | Computing Probabilities | 32 |
| A.4 | Sampling the Height | 33 |
| A.5 | First Return Sampling | 35 |

1 Introduction

The problem of computing local information of huge random objects was pioneered in [3, 4]. Further work of [9] considers the generation of sparse random $G(n, p)$ graphs from the Erdős-Rényi model [1], with $p = O(\text{poly}(\log n)/n)$, which answers $\text{poly}(\log n)$ **ALL-NEIGHBORS** queries, listing the neighbors of queried vertices. While these generators use polylogarithmic resources over their entire execution, they generate graphs that are only guaranteed to *appear random* to algorithms that inspect a *limited portion* of the generated graph.

In [2], the authors construct an oracle for the generation of recursive trees, and BA preferential attachment graphs. Unlike [9], 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 [2] provides access to the adjacency list through **NEXT-NEIGHBOR** queries.

In this work, we begin by *formalizing* a model of local-access generators implicitly used in [2]. We next construct oracles that allow queries to both the adjacency matrix and adjacency list representation of a basic class of random graph families, without generating the entire graph at the onset. Our oracles provide **VERTEX-PAIR**, **NEXT-NEIGHBOR**, and **RANDOM-NEIGHBOR** queries¹ for graphs with *independent edge probabilities*, that is, when each edge is chosen as an independent Bernoulli random variable. Using this framework, we construct the first *efficient* local-access generators for undirected graph models, supporting all three types of queries using $\mathcal{O}(\text{poly}(\log n))$ time, space, and random bits per query, under assumptions on the ability to compute certain values pertaining to consecutive edge probabilities. In particular, our construction yields local-access generators for the Erdős-Rényi $G(n, p)$ model (for *all* values of p), and the Stochastic Block model with random community assignment. As in [2] (and unlike the generators in [3, 4, 9]), 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 [2, 3, 4, 9], we provide the first implementation (to the best of our knowledge) of **RANDOM-NEIGHBOR** queries, which do not follow trivially from the **ALL-NEIGHBOR** queries in *non-sparse graphs*. Such queries are useful, for instance, for sub-linear algorithms that employ random walk processes. **RANDOM-NEIGHBOR** queries present particularly interesting challenges, since as we note in Section 2.1, (1) **RANDOM-NEIGHBOR** queries affect the conditional probabilities of the remaining neighbors in a non-trivial manner, and (2) our implementation does not resort to explicitly sampling the degree of any vertex in order to generate a random neighbor. First, sampling the degree of the query vertex, we suspect, is not viable for *sub-linear* generators, because this quantity alone imposes dependence on the existence of *all* of its potential incident edges. Therefore, our generator needs to return a random neighbor, with probability reciprocal to the query vertex’s degree, without resorting to “knowing” its degree. Second, even without committing to the degrees, answers to **RANDOM-NEIGHBOR** queries affect the conditional probabilities of the remaining adjacencies in a global and non-trivial manner – that is, from the point of view of the *agent* interacting with the generator. The generator, however, must somehow

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).

maintain and leverage its additional *internal knowledge* of the partially-generated graph, to keep its computation tractable throughout the entire graph generation process.

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

For additional related work, see Section ??.

2 Our Contributions and Techniques

We begin by formalizing a model of *local-access generators* (Section 3.1), implicitly used in [2]. Our work provides local-access generators for various basic classes of graphs described in the following, with **VERTEX-PAIR**, **NEXT-NEIGHBOR**, and **RANDOM-NEIGHBOR** queries. In all of our results, each query is processed using $\text{poly}(\log n)$ time, random bits, and additional space, with *no initialization overhead*. These guarantees hold even in the case of adversarial queries. Our bounds assume constant computation time for each arithmetic operation with $\mathcal{O}(\log n)$ -bit precision. Each of our generators constructs a random graph drawn from a distribution that is $1/\text{poly}(n)$ -close to the desired distribution in the L_1 -distance.²

2.1 Undirected Graphs

In Section 4 we construct local access generators for the generic class of undirected graphs with *independent edge probabilities* $\{p_{u,v}\}_{u,v \in V}$, where $p_{u,v}$ denote the probability that there is an edge between u and v . Throughout, we identify our vertices via their unique IDs from 1 to n , namely $V = [n]$. We assume that we can compute various values pertaining to consecutive edge probabilities for the class of graphs, as detailed below. We then show that such values can be computed for graphs generated according to the Erdős-Rényi $G(n, p)$ model and the Stochastic Block model.

Next-Neighbor Queries

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

A main difficulty in our setup, as compared to [2], 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')

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

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.

Random-Neighbor Queries

We provide efficient **RANDOM-NEIGHBOR** queries (Section 4.3). The ability to do so is surprising. First, note that after performing a **RANDOM-NEIGHBOR** query all other conditional probabilities will be affected in a non-trivial way.³ This requires a way of implicitly keeping track of all the resulting changes. Second, we can sample a **RANDOM-NEIGHBOR** with the correct probability $1/\deg(v)$, even though we do not sample or know the degree of the vertex.

We formulate a *bucketing approach* (Section 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 $\text{poly}(\log n)$ complexities overall. This bucketing approach also naturally leads to our data structure that requires constant space for each bucket and for each edge, using $\Theta(n + m)$ overall memory requirement. The **VERTEX-PAIR** queries are implemented by sampling the relevant bucket.

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

Erdős-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 $\tilde{O}(n + m)$ time and space (Corollary 5), which is tight up to polylogarithmic factors.

³ Consider a $G(n, p)$ graph with small p , say $p = 1/\sqrt{n}$, such that vertices will have $\tilde{O}(\sqrt{n})$ neighbors with high probability. After $\tilde{O}(\sqrt{n})$ **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.

Stochastic Block Model

We generalize our construction to the Stochastic Block Model. In this model, the vertex set is partitioned into r communities $\{C_1, \dots, C_r\}$. 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., [8]). 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 \mathbf{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 [4], 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 5.2.1). This sampling procedure may be of independent interest. For r communities, this yields an implementation with $\mathcal{O}(r \cdot \text{poly}(\log n))$ overhead in required resources for each operation. This upholds all previous polylogarithmic guarantees when $r = \text{poly}(\log n)$.

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 3.3.

2.2 Directed Graphs

Lastly, we consider Kleinberg’s Small World model ([5, 7]) in Section ???. While Small-World models are proposed to capture properties of observed data such as small shortest-path distances and large clustering coefficients [11], 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.

3 Preliminaries

3.1 Local-Access Generators

We consider the problem of locally generating random graphs $G = (V, E)$ drawn from the desired families of simple unweighted graphs, undirected or directed. We denote the number of vertices $n = |V|$, and refer to each vertex simply via its unique ID from $[n]$. For undirected G , the set of neighbors of $v \in V$ is defined as $\Gamma(v) = \{u \in V : \{v, u\} \in E\}$; denote its degree by $\deg(v) = |\Gamma(v)|$. Inspired by the goals and results of [2], 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 \mathcal{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 \mathcal{D}' that is ϵ -close to the desired distribution \mathcal{D} in the L_1 -distance. In this work we focus on supporting $\epsilon = n^{-c}$ for any desired constant $c > 0$. As for $\text{RANDOM-NEIGHBOR}(v)$, 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 $\text{poly}(\log n)$.

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 $\text{NEXT-NEIGHBOR}(v)$ or $\text{RANDOM-NEIGHBOR}(v)$ 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.

- $\text{NEXT-NEIGHBOR}(v)$: 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 $n + 1$.
- $\text{RANDOM-NEIGHBOR}(v)$: The generator returns a neighbor of v uniformly at random (with probability $1/\deg(v)$ each). If v is isolated, \perp is returned.
- $\text{VERTEX-PAIR}(u, v)$: The generator returns either 1 or 0, indicating whether $\{u, v\} \in E$ or not.
- $\text{ALL-NEIGHBORS}(v)$: 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.

3.2 Random Graph Models

Erdős-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 Erdős-Rényi Model. The vertex set V is partitioned into r communities C_1, \dots, 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 \mathbf{R} over the communities, and each

vertex is assigned its community according to \mathbf{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 $\text{DIST}(u, v) = |u_x - v_x| + |u_y - v_y|$, and the probability that each directed edge (u, v) exists is $c/(\text{DIST}(u, v))^2$. 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 [5], we assume that the probability is rounded down to 1 if $c/(\text{DIST}(u, v))^2 > 1$.

3.3 Miscellaneous

Arithmetic operations

Let N be a sufficiently large number of bits required to maintain a multiplicative error of at most a $\frac{1}{\text{poly}(n)}$ factor over $\text{poly}(n)$ elementary computations $(+, -, \cdot, /, \exp)$.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.

Sampling via a CDF

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

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 $\prod_{v=a}^b (1 - p_{v,u})$, and (2) the sum of the edge probabilities (i.e., the expected number of edges) between u and vertices from $[a, b]$, namely $\sum_{v=a}^b p_{v,u}$. We will later give subroutines for computing these values for the Erdős-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.

⁴ Our algorithm also supports the alternative specification where the community sizes $\langle |C_1|, \dots, |C_r| \rangle$ 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} .

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

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

Algorithm 1 Naïve Generator

```

procedure VERTEX-PAIR( $u, v$ )
  if  $\mathbf{A}[u][v] = \phi$  then
    draw  $X_{u,v} \sim \text{Bern}(p_{u,v})$ 
     $\mathbf{A}[v][u], \mathbf{A}[u][v] \leftarrow X_{u,v}$ 
  return  $\mathbf{A}[u][v]$ 

procedure NEXT-NEIGHBOR( $v$ )
  for  $u \leftarrow \text{last}[v] + 1$  to  $n$  do
    if VERTEX-PAIR( $v, u$ ) = 1 then
       $\text{last}[v] \leftarrow u$ 
      return  $u$ 
   $\text{last}[v] \leftarrow n + 1$ 
  return  $n + 1$ 

procedure RANDOM-NEIGHBOR( $v$ )
   $R \leftarrow V$ 
  repeat
    sample  $u \in R$  u.a.r.
    if VERTEX-PAIR( $v, u$ ) = 1 then
      return  $u$ 
    else
       $R \leftarrow R \setminus \{u\}$ 
  until  $R = \emptyset$ 
  return  $\perp$ 

```

First, consider a naïve implementation 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 $\text{last}[v]$ records the neighbor of v returned in the last call **NEXT-NEIGHBOR**(v), or $\text{last}[v] = 0$ if no such call has been invoked. This definition of **last** was introduced in [2]. All cells of \mathbf{A} and **last** are initialized to ϕ and 0 , respectively. We refer to Algorithm 5 for its straightforward implementation, but highlight some notations and useful observations here.

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

Algorithm 5 updates the cell $\mathbf{A}[u][v] = \phi$ to the value of the Bernoulli random variable (RV) $X_{u,v} \sim \text{Bern}(p_{u,v})$ (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 5, 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 $u > \text{last}[v]$ of $X_{v,u} = 1$, from a sequence of Bernoulli RVs $\{X_{v,u}\}_{u > \text{last}[v]}$, in a direct fashion. To do so, we sample a consecutive “run” of 0's with probability $\prod_{u=\text{last}[v]+1}^{u'} (1-p_{v,u})$: this is the probability that there is no edge between a vertex v and any $u \in (\text{last}[v], u']$, 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 $u > \text{last}[v]$. To this end, we give a succinct data structure that determines the value of $\mathbf{A}[v][u]$ for $u > \text{last}[v]$ 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 \mathbf{last} and P_v 's provide a succinct representation of \mathbf{A} via the following observation. For simplicity, we say that $X_{u,v}$ is *decided* if $\mathbf{A}[u][v] \neq \phi$, and call it *undecided* otherwise.

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

Proof. The condition for $\mathbf{A}[v][u] = 1$ clearly holds by construction. 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. \blacktriangleleft

4.2.2 Queries and Updates

Algorithm 2 Sampling NEXT-NEIGHBOR

```

procedure NEXT-NEIGHBOR( $v$ )
   $u \leftarrow \text{last}[v]$ 
   $w_v \leftarrow \min\{(P_v \cap (u, n]) \cup \{n + 1\}\}$ 
  repeat
    sample  $F \sim F(v, u, w_v)$ 
     $u \leftarrow F$ 
  until  $u = w_v$  or  $\text{last}[u] < v$ 
  if  $u \neq w_v$  then
     $P_v \leftarrow P_v \cap \{u\}$ 
     $P_u \leftarrow P_u \cap \{v\}$ 
   $\text{last}[v] \leftarrow u$ 
  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 $(\text{last}[v], w_v)$. Suppose that we generate a sequence of $w_v - \text{last}[v] - 1$ independent coin-tosses, where the i^{th} coin $C_{v,u}$ corresponding to $u = \text{last}[v] + i$ 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 $F(v, a, b)$ 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, $F \sim F(v, a, b)$ represents the event that $C_{v,a+1} = \dots = C_{v,F-1} = 0$ 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 $F_1 \sim F(v, \text{last}[v], w_v)$ to find the first occurrence of $C_{v,F_1} = 1$, then samples $F_2 \sim F(v, F_1, w_v)$ 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 $\mathbf{A}[v][u] = 0$ only for contiguous cells $\mathbf{A}[v][u]$ where $u \leq \mathbf{last}[v]$ or $v \leq \mathbf{last}[u]$: they cannot handle 0 anywhere else. Unfortunately, in order to extend our construction to support **RANDOM-NEIGHBOR** queries using the idea suggested in Algorithm 5, we must unavoidably assign $\mathbf{A}[v][u]$ to 0 in random locations beyond $\mathbf{last}[v]$ or $\mathbf{last}[u]$, which cannot be captured by the current data structure. Furthermore, unlike 1's, we cannot record 0's using a data structure similarly to that of P_v . More specifically, to speed-up the sampling process for small $p_{v,u}$'s, we must generate many random non-neighbors at once as suggested in Algorithm 2, but we cannot afford to spend time linear in the number of created 0's to update our data structure. We remedy these issues via the following bucketing approach.

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* $B_v = \{B_v^{(1)}, B_v^{(2)}, \dots\}$, so that each bucket contains, in expectation, roughly the same number of neighbors of v . We may then implement **RANDOM-NEIGHBOR**(v) by randomly selecting a bucket $B_v^{(i)}$, fill in entries $\mathbf{A}[v][u]$ for $u \in B_v^{(i)}$ with 1's and 0's, then report a random neighbor from this bucket. As the bucket size may be too large when the probabilities are small, instead of using a linear scan, our **FILL** subroutine will be implemented with the **NEXT-NEIGHBOR** subroutine in Algorithm 2 previously developed in Section 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 probability of success in the rejection sampling process is at least $1/\text{poly}(\log n)$, but the number of iterations required by **NEXT-NEIGHBOR** is also bounded by $\text{poly}(\log n)$, achieving the overall $\text{poly}(\log n)$ complexities. Here in this section, we will extensively rely on the assumption that the expected number of neighbors for consecutive vertices, $\sum_{u=a}^b p_{v,u}$, can be computed efficiently.

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 $\mathbb{E}[|\Gamma^{(i)}(v)|] < L+1$ for every bucket, and $\mathbb{E}[|\Gamma^{(i)}(v)|] > L-1$ 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.

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 $\mathbb{E}[|\Gamma^{(i)}(v)|] = \mathbb{E}\left[\sum_{u \in B_v^{(i)}} X_{v,u}\right] < L + 1$. 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 $c > 0$. ◀

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 $\mathbb{E}[|\Gamma^{(i)}(v)|] = \mathbb{E}\left[\sum_{u \in B_v^{(i)}} X_{v,u}\right] > L - 1$, 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 sufficiently large constant L . Let T_i be the indicator for the event that $B_v^{(i)}$ is *not* empty, so $\mathbb{E}T_i = 1 - c$. 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 **FILL**(v, i) as follows. When invoked on an unfilled bucket $B_v^{(i)}$, **FILL**(v, i) performs the following tasks:

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

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

4.3.3 Putting it all together: Random-Neighbor queries

Algorithm 3 Bucketing Generator

```

procedure RANDOM-NEIGHBOR( $v$ )
   $R \leftarrow [|B_v|]$ 
  repeat
    sample  $i \in R$  u.a.r.
    if  $B_v^{(i)}$  is not filled then
       $\text{FILL}(v, i)$ 
    if  $|P_v^{(i)}| > 0$  then
      with probability  $\frac{|P_v^{(i)}|}{M}$ 
        sample  $u \in P_v^{(i)}$  u.a.r.
        return  $u$ 
    else
       $R \leftarrow R \setminus \{i\}$ 
  until  $R = \emptyset$ 
  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 $\text{FILL}(v, i)$ 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 $\text{FILL}(v, i)$ 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 4.3.1; 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. 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. \blacktriangleleft

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 4.3.1, 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 4.3.1. 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 $\mathcal{O}(\log^2 n)$ with high probability. \blacktriangleleft

4.4 Implementation of Fill

Algorithm 4 Sampling in a Bucket

```

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

```

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 an 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 4.2.1. Here instead, with our bucket structure, we maintain a single bit marking whether each bucket is filled or unfilled: a filled bucket implies that $\mathbf{A}[v][u]$ for all $u \in B_v^{(i)}$ are decided. The bucket structure along with mark bits, unlike **last**, are capable of handling intermittent ranges of intervals, namely buckets, which is sufficient for our purpose, as shown in the following lemma. This yields the implementation Algorithm 4 for the **FILL** procedure fulfilling the requirement previously given in Section 4.3.2.

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 construction. Otherwise, observe that $\mathbf{A}[v][u]$ becomes decided precisely during a **FILL**(v, i) or a **FILL**(u, j) operation, which thereby

marks one of the corresponding buckets as filled. \blacktriangleleft

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 $F(u, a, b)$, and to maintain the ordered sets $P_v^{(i)}$ and $P_u^{(j)}$.

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 4.3.3, we finally obtain polylogarithmic resource bound for our implementation of **RANDOM-NEIGHBOR**.

► **Corollary 2.** *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 4.4 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 4.3.1, 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 3.** *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 perfect-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 $F(v, a, b)$ in Algorithm 4. By our assumption, the CDF of any $F(v, a, b)$ can be efficiently computed from $\prod_{u=a}^{u'} (1 - p_{v,u})$, and thus sampling with $\frac{1}{\text{poly}(n)}$ 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.

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 $F'(v, a, b)$ that is ϵ -close in L_1 -distance to the correct distribution $F(v, a, b)$ that perfectly generates the desired distribution \mathbf{G} over all graphs, then the distribution \mathbf{G}' of the generated graph G is (ϵS) -close to \mathbf{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 $\mathcal{B} = \langle B_{v_1}^{(u_1)}, B_{v_2}^{(u_2)}, \dots \rangle$, 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 $F'(v, a, b)$ where $\|F(v, a, b) - F'(v, a, b)\|_1 \leq \epsilon = \frac{1}{\text{poly}(n)}$, then moves to a new state according to u . In other words, there is an induced pair of collection of distributions over the states: $(\mathcal{T}, \mathcal{T}')$ where $\mathcal{T} = \{\mathbf{T}_s\}_{s \in \mathcal{S}}$, $\mathcal{T}' = \{\mathbf{T}'_s\}_{s \in \mathcal{S}}$, such that $\mathbf{T}_s(s')$ and $\mathbf{T}'_s(s')$ denote the probability that the algorithm advances from s to s' by using a sample from the correct $F(v, a, b)$ and from the approximated $F'(v, a, b)$, respectively. Consequently, $\|\mathbf{T}_s - \mathbf{T}'_s\|_1 \leq \epsilon$ for every $s \in \mathcal{S}$.

The generator begins with the initial (internal) state $s_0 = (1, 0, \mathbf{A}_\phi)$ where all cells of \mathbf{A}_ϕ 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 $\pi = \langle s_0^\pi = s_0, s_1^\pi, \dots, s_{\ell(\pi)}^\pi = t_{\mathbf{A}} \rangle$ 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 $T_{t_A}(t_A) = 1$, and $T_{t_A}(s) = 0$ for all state $s \neq t_A$, 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 $q(\pi) = \prod_{i=1}^S T_{s_{i-1}}(s_i)$, and thus $G(G_A) = \sum_{\pi: s_S^\pi = t_A} q(\pi)$.

Let $\mathcal{T}^{\min} = \{\mathbf{T}_s^{\min}\}_{s \in \mathcal{S}}$ where $\mathbf{T}_s^{\min}(s') = \min\{\mathbf{T}_s(s'), \mathbf{T}'_s(s')\}$, and note that each \mathbf{T}_s^{\min} is not necessarily a probability distribution. Then, $\sum_{s'} \mathbf{T}_s^{\min}(s') = 1 - \|\mathbf{T}_s - \mathbf{T}'_s\|_1 \geq 1 - \epsilon$. Define $q', q^{\min}, G'(G_A), G^{\min}(G_A)$ analogously, and observe that $q^{\min}(\pi) \leq \min\{q(\pi), q'(\pi)\}$ for every π , so $G^{\min}(G_A) \leq \min\{G(G_A), G'(G_A)\}$ for every G_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, $G^{\min}(G_A)$ lower bounds the probability that the algorithm generates the graph G_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 $i \geq 1$,

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

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, $\sum_{G_A} \min\{G(G_A), G'(G_A)\} \geq \sum_{G_A} G^{\min}(G_A) \geq 1 - \epsilon S$, implying that $\|G - G'\|_1 \leq \epsilon S$, as desired. In particular, by substituting $\epsilon = \frac{1}{\text{poly}(n)}$ and $S = O(n^3)$, we have shown that Algorithm 4 only creates a $\frac{1}{\text{poly}(n)}$ error in the L_1 -distance. \blacktriangleleft

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 than 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 Erdős-Rényi Model and Stochastic Block Model

In this section we demonstrate the application of our techniques to two well known, and widely studied models of random graphs. That is, as required by Theorem 3, 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 Erdős-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 \mathbf{R} over the communities. We formulate a method of sampling community assignments locally. This essentially allows us to sample from the *multivariate hypergeometric distribution*, using $\text{poly}(\log n)$ random bits, which may be of independent interest. We remark that, as our first step, we sample for the number of vertices of each community. That is, our construction can alternatively support

the community assignment where the number of vertices of each community is given, under the assumption that the *partition* of the vertex set into communities is chosen uniformly at random.

5.1 Erdős-Rényi Model

As $p_{v,u} = p$ for all edges $\{u, v\}$ in the Erdős-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 4.** *The final algorithm in Section 4 locally generates a random graph from the Erdős-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 $F \sim F(v, a, b)$ 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 $\text{poly}(\log n)$ factors in the resources, given the perfect-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 5.** *The final algorithm in Section 4 can generate a complete random graph from the Erdős-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 $\mathcal{O}(\text{poly}(\log n))$ 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 \mathbf{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 $\mathcal{O}(r \text{poly}(\log n))$ resources per query.

► **Theorem 6.** *There exists a data structure (generator) that samples a community for each vertex independently at random from \mathbf{R} with $\frac{1}{\text{poly}(n)}$ 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 $\mathcal{O}(r \text{poly}(\log n))$ 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 7.** *The final algorithm in Section 4 generates a random graph from the Stochastic Block model with randomly-assigned communities using $\mathcal{O}(r \text{poly}(\log n))$ time, random N -bit words, and additional space per query with high probability.*

We provide the full details of the construction in the following Section 5.2.1. Our construction extends upon a similar generator in the work of [4] which only supports $r = 2$. Our overall data structure is a balanced binary tree, where the root corresponds to the entire range of indices $\{1, \dots, n\}$, and the children of each vertex corresponds to each half of the parent's range. Each node⁷ 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 distribution⁸ (for n samples (vertices) from the probability distribution \mathbf{R}). The children are only generated top-down on an as-needed basis according to the given queries. The technical difficulties arise when generating the children, where one needs to sample "half" of the counts of the parent from the correct marginal distribution. To this end, we show how to sample such a count as described in the statement below. Namely, we provide an algorithm for sampling from the *multivariate hypergeometric distribution*.

5.2.1 Sampling from the Multivariate Hypergeometric Distribution

Consider the following random experiment. Suppose that we have an urn containing $B \leq n$ marbles (representing vertices), each occupies one of the r possible colors (representing communities) represented by an integer from $[r]$. The number of marbles of each color in the urn is known: there are C_k indistinguishable marbles of color $k \in [r]$, where $C_1 + \dots + C_r = B$. 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 $\mathbf{C} = \langle c_1, \dots, c_r \rangle$, then we would like to (approximately) sample a vector $\mathbf{S}_\ell^{\mathbf{C}}$ of r non-negative integers such that

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

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

The sample $\mathbf{S}_\ell^{\mathbf{C}}$ above may be generated easily by simulating the drawing process, but this may take $\Omega(\ell)$ iterations, which have linear dependency in n in the worst case: $\ell = \Theta(B) = \Theta(n)$. Instead, we aim to generate such a sample in $O(r \text{ poly}(\log n))$ time with high probability. We first make use of the following procedure from [4].

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 $O(\text{poly}(\log n))$ 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 $r = 2$).

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 [6]

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 $O(\text{poly}(\log n))$ time and random words.

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

- If $\ell \leq B/2$, generate $\mathbf{C}' = \mathbf{S}_{B/2}^{\mathbf{C}}$ using Claim 5.2.1.
 - If $\ell = B/2$ then we are done.
 - Else, for $\ell < B/2$ we recursively generate $\mathbf{S}_{\ell}^{\mathbf{C}'}$.
- Else, for $\ell > B/2$, we generate $\mathbf{S}_{B-\ell}^{\mathbf{C}'}$ as above, then output $\mathbf{C} - \mathbf{S}_{B-\ell}^{\mathbf{C}'}$.

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

Lastly we generalize to support larger r .

► **Theorem 8.** *Given B marbles of r different colors, such that there are C_i marbles of color i , there exists an algorithm that samples $\langle s_1, s_2, \dots, s_r \rangle$, the number of marbles of each color appearing when drawing l marbles from the urn without replacement, in $O(r \cdot \text{poly}(\log n))$ time and random words.*

Proof. Observe that we may reduce $r > 2$ to the two-color case by sampling the number of marbles of the first color, collapsing the rest of the colors together. Namely, define a pair $\hat{\mathbf{C}} = \langle C_1, C_2 + \dots + C_r \rangle$, then generate $\mathbf{S}_{\ell}^{\hat{\mathbf{C}}} = \langle s_1, s_2 + \dots + s_r \rangle$ via the above procedure. At this point we have obtained the first entry s_1 of the desired $\mathbf{S}_{\ell}^{\mathbf{C}}$. So it remains to generate the number of marbles of each color from the remaining $r - 1$ colors in $\ell - s_1$ remaining draws. In total, we may generate $\mathbf{S}_{\ell}^{\mathbf{C}}$ by performing r iterations of the two-colored case. The error in the L_1 -distance may be established similarly to the proof of Lemma 4.5. ◀

5.2.2 Data structure

We now show that Theorem 8 may be used in order to create the following data structure. Recall that \mathbf{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, \dots, X_n , each of which is drawn independently at random from \mathbf{R} : X_i denotes the community of vertex i . Then given a pair (i, j) , it returns the vector $\mathbf{C}(i, j) = \langle c_1, \dots, c_r \rangle$ where c_k counts the number of variables X_i, \dots, 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 \mathbf{R} (see e.g., Section 3.4.1 of [6]). Then, the children are generated on-the-fly using the lemma above. Thus, each query can be processed within $O(r \text{poly}(\log n))$ time, yielding Theorem 6. Then, by embedding the information stored by the data structure into the state (as in the proof of Lemma 4.5), we obtain the desired Corollary 7.

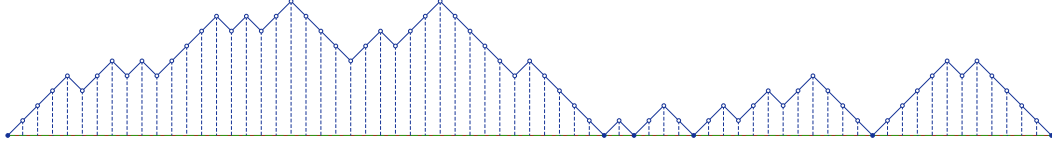
6 Sampling Catalan Objects

we have?

As we have seen before, [4] gives us a method of sampling Binomial objects. In constructing a generator for the Stochastic Block model, we generalized this to help us sample from the multivariate hypergeometric distribution. Now we focus on a more interesting variant of the question.

Dyck paths are one interpretation of the Catalan numbers.

Connect with
previous part
on SBM



■ **Figure 1** Simple Dyck path with $n = 35$.

A Dyck path can be constructed as a $2n$ step one-dimensional random walk (Figure 1). Each step in the walk moves one unit along the positive x -axis and one unit up or down the positive y -axis. Given these restrictions, we would obtain a 1D random walk pinned to zero on both sides. A Dyck path also has the additional restriction that the y -coordinate of any point on the random walk is ≥ 0 . i.e. the walk is always north of the origin. The number of possible Dyck paths (see Theorem 24) is the n^{th} Catalan number $C_n = \frac{1}{n+1} \cdot \binom{2n}{n}$.

We will attempt to support queries to a uniformly random instance of a Dyck path. 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).

The **HEIGHT** query seems natural, but it isn't obvious why we care about the **FIRST-RETURN** query.

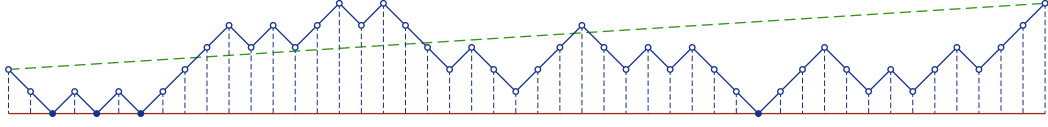
6.1 Catalan Trapezoids and Generalized Dyck Paths

In order to generate these objects locally, we will need to analyze more general Catalan objects which correspond to numbers in the *Catalan Trapezoid*. First, we define Catalan trapezoids as presented in [10]. Let $C_k(n, m)$ be the $(n, m)^{\text{th}}$ entry of the Catalan trapezoid of order k , where $C_1(n, m)$ corresponds to the Catalan triangle. We can interpret $C_k(n, m)$ as the number of *generalized* Dyck paths. Specifically, we consider a sequence of n up-steps and m 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 2). The total number of such paths is exactly $C_k(n, m)$. For $k = 1$, we obtain the definition of the simple Dyck path (Figure 1). Now, we state a result from [10] without proof

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

6.2 Generating Dyck Paths

Our general recursive step is as follows. We consider a sequence of length $2S$ comprising of $2U$ up moves (+1) and $2D$ down moves (−1). Additionally, the sum of any initial sequence



■ **Figure 2** Complex Dyck path with $n = 25$, $m = 22$ and $k = 3$. Notice that the boundary is shifted.

prefix? can be less than $k - 1$. Without loss of generality, let's assume that $2D \leq S$. If this were not the case, we could simply flip the sequence and negate the elements. This essentially means that the overall Dyck path is non-decreasing.

► **Lemma 9.** $S - 2D = \mathcal{O}(\log n \sqrt{S}) \implies U - D = \mathcal{O}(\log n \sqrt{S})$

We want to sample the height of this path after S steps. This is the same as sampling the number of $(+1)$ s that get assigned to the first half of the elements in the sequence. We define p_d as the probability that exactly $D - d$ (-1) s get assigned to the first half. This means that exactly $U + d$ $(+1)$ s get assigned to the first half. Consequently, the second half will contain exactly $D + d$ (-1) s and $U - d$ $(+1)$ s.

What is d is negative?

Let us first compute this probability.

$$p_d = \frac{D_{left} \cdot D_{right}}{D_{tot}}$$

Here, D_{left} denotes the number of valid starting sequences (first half) and D_{right} denotes the number of valid ending sequences. Here, *valid* means that each half sequence gets the appropriate number of ups and downs and the initial sums never drop below $1 - k$. For, D_{right} , we will start the Dyck path from the end of the $2S$ sequence. In this case the invalidation threshold will be a different k' . This k' is the final height of the $2S$ sequence. So, $k' = k + 2U - 2D = k + 4S - 2D$. We will use this fact extensively moving forward.

Also, D_{tot} is the total number of possible sequences of length $2S$, given the initial conditions. Note that in this case the threshold remains at k .

Frequently?

We will use the following rejection sampling lemma from [4].

► **Lemma 10.** Let $\{p_i\}$ and $\{q_i\}$ be distributions satisfying the following conditions

1. There is a poly-time algorithm to approximate p_i and q_i up to $\pm n^{-2}$
2. Generating an index i according to q_i is closely implementable.
3. There exists a poly($\log n$)-time recognizable set S such that

- $1 - \sum_{i \in S} p_i$ 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.

Algorithm 5 Naïve Generator

```

procedure SPLIT( $U, D, k$ )
   $S \leftarrow U + D$ 

   $d \sim \left\{ \frac{\binom{S-d}{D+d} \binom{S}{2D}}{\binom{2S}{2D}} \right\}_d$ 

   $k' \leftarrow k + U - D$ 

   $p_d \leftarrow \frac{\binom{S-d}{D-d} - \binom{S-d-k}{D-d-k} \binom{S}{2D} - \binom{S}{2D-k}}{\binom{2S}{2D} - \binom{2S}{2D-k}}$ 
   $q_d \leftarrow \frac{\binom{S-d}{D+d} \binom{S}{2D}}{\binom{2S}{2D}}$ 

  if  $p_d < q_d$  then
    return  $d$ 
  draw  $X \sim \text{Bern}(p_d/q_d)$ 
  if  $X = 0$  then
    return  $d$ 
  return SPLIT( $U, D, k, k'$ )

procedure HEIGHT( $x$ )
  if  $x \in \text{heights}$  then
    return  $\text{heights}[x]$ 

   $l \leftarrow \text{LOWER-BOUND}(x)$ 
   $r \leftarrow \text{UPPER-BOUND}(x)$ 
   $h_l \leftarrow \text{HEIGHT}(l)$ 
   $h_r \leftarrow \text{HEIGHT}(r)$ 
   $\text{extra} \leftarrow (r - l) - (h_r - h_l)$ 
   $U \leftarrow (h_r - h_l) + \text{extra}/2$ 
   $D \leftarrow \text{extra}/2$ 
   $k \leftarrow 1 + h_l$ 
   $d \leftarrow \text{SPLIT}(U, D, k)$ 
   $\text{heights}[(r+l)/2] \leftarrow h_l + U + d$ 
  return HEIGHT( $x$ )

```

6.2.1 The Simple Case

The problem of sampling reduces to the binomial sampling case when $k > \mathcal{O}(\log n)\sqrt{S}$ for some constant c . This is because with high probability, will never dip below the threshold. In this case, the we can simply approximate the probability as

$$\frac{\binom{S}{D-d} \cdot \binom{S}{D+d}}{\binom{2S}{2D}}$$

This is because unconstrained random walks will not dip below the $1 - k$ threshold with high probability. This problem was solved in [4] using $\mathcal{O}(\text{poly}(\log n))$ resources.

6.2.2 Path Segments Close to Zero

The problem arises when we $k < \mathcal{O}(\log n)\sqrt{S}$. In this case we need to compute the actual probability, Using the formula from [10], we find that.

$$D_{left} = \binom{S}{D-d} - \binom{S}{D-d-k} \quad D_{right} = \binom{S}{U-d} - \binom{S}{U-d-k'} \quad D_{tot} = \binom{2S}{2D} - \binom{2S}{2D-k} \quad (1)$$

Here, $k' = k + 2U - 2D$, and so $k' = \mathcal{O}(\log n)\sqrt{S}$ (using Lemma 9).

The final distribution we wish to sample from is given by $\{p_d\}_d$ where $p_d = \frac{D_{left} \cdot D_{right}}{D_{tot}}$. To achieve this, we will use Lemma 10 from [4]. An important point to note is that in order to apply this lemma, we must be able to compute the p_d values. For now, we will assume that we have access to an oracle that will compute the value for us. Later, in Section , we will see how to construct such an oracle.

Fix reference

bound variance of path

In this process, we will first disregard all values of d where $|d| > \Theta(\log n\sqrt{S})$. The probability mass associated with these values can be shown to be negligible .

Next, we will construct an appropriate $\{q_i\}$ and show that $p_d < \log^{\mathcal{O}(1)} n \cdot q_d$ for all $|d| < \Theta(\sqrt{S})$ and some constant c . We will use the following distribution

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

It is shown in [4] that this distribution is closely implementable.

First, we consider the case where $k \cdot k' \leq 2U + 1$. In this case, we use loose bounds for $D_{left} < \binom{S}{D-d}$ and $D_{right} < \binom{S}{U-d}$. We also use the following lemma (proven in Section A).

► **Lemma 11.** *When $kk' > 2U + 1$, $D_{tot} > \frac{1}{2} \cdot \binom{2S}{2D}$.*

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 (no boundary).

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 12.** *$D_{left} \leq c_1 \frac{k \cdot \log n}{\sqrt{S}} \cdot \binom{S}{D-d}$ for some constant c_1 .*

► **Lemma 13.** *$D_{right} < c_2 \frac{k' \cdot \log n}{\sqrt{S}} \cdot \binom{S}{U-d}$ for some constant c_2 .*

► **Lemma 14.** *When $kk' \leq 2U + 1$, $D_{tot} < c_3 \frac{k \cdot k'}{S} \cdot \binom{2S}{2D}$ for some constant c_3 .*

We can now put these lemmas together to show that $p_d/q_d \leq \Theta(\log^2 n)$. Now, we can apply Lemma 10 to sample the value of d , which gives us the height of the Dyck path at the midpoint of the two given points.

► **Theorem 15.** *There is an algorithm that given two points at distance a and b (with $a < b$) along a Dyck path of length $2n$, with the guarantee that no position between a and b has been sampled yet, returns the height of the path halfway between a and b . Moreover, this algorithm only uses $\mathcal{O}(\text{poly}(\log n))$ resources.*

Proof. If $b - a$ is even, we can set $S = (b - a)/2$. Otherwise, we first sample a single step from a to $a + 1$, and then set $S = (b - a - 1)/2$. Since there are only two possibilities for a single step, we can explicitly compute an approximation of the probabilities, and then sample accordingly. Now, if $S > \Theta(\log^2 n)$ we can simply use the rejection sampling procedure described above to obtain a $\mathcal{O}(\text{poly}(\log n))$ algorithm. Otherwise, we sample each step individually. Since there are only $2S = \Theta(\log^2 n)$ steps, the sampling is still efficient. ◀

► **Theorem 16.** *There is an algorithm that provides sample access to a Dyck path of length $2n$, by answering queries of the form $\text{HEIGHT}(x)$ with the correctly sampled height of the Dyck path at position x using only $\mathcal{O}(\text{poly}(\log n))$ 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 queried. Each newly queried position is added to this structure. Given a query $\text{HEIGHT}(x)$, the algorithm first finds the successor and predecessor (say a and b) of x among the already queried positions. This provides us the guarantee required to apply Theorem 15, 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 . Once the interval length becomes less than $\Theta(\log^2 n)$, we perform the full sampling (as in Theorem 15) which provides us the height at position x . ◀

6.3 Supporting “First Return” Queries

We might want to support more complex queries to a Dyck path. Specifically, in addition to querying the height of a position, we might want to know the next time the path return to that height (if at all). We introduce a new query $\text{FIRST-RETURN}(x)$ which returns the first time the walk returns to $\text{HEIGHT}(x)$ 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, $\text{FIRST-RETURN}(x)$ 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.

Explain why

We will use the following asymptotic formula for *close-to-central* binomial coefficients.

► **Lemma 17.** *If $k = \frac{n \pm c\sqrt{n}}{2}$ where $c = o(n^{1/6})$, we can approximate $\binom{n}{k}$ up to constant factors by the expression:*

$$\frac{2^n}{\sqrt{n}} \cdot e^{-c^2/2}$$

We maintain a threshold $\mathcal{T} = \Theta(\log^7 n)$. If an un-sampled interval in the Dyck path has length less than \mathcal{T} , then we sample the entire interval. So, for intervals with length $S > \mathcal{T}$, the maximum deviations are bounded by $\mathcal{O}(\log n \sqrt{S}) = \mathcal{O}(\log^{4.5} n)$ with high probability. Specifically, this means that if we write the deviation as $c\sqrt{n}$, we see that $c = \log n$ which is $o(S^{1/6})$.

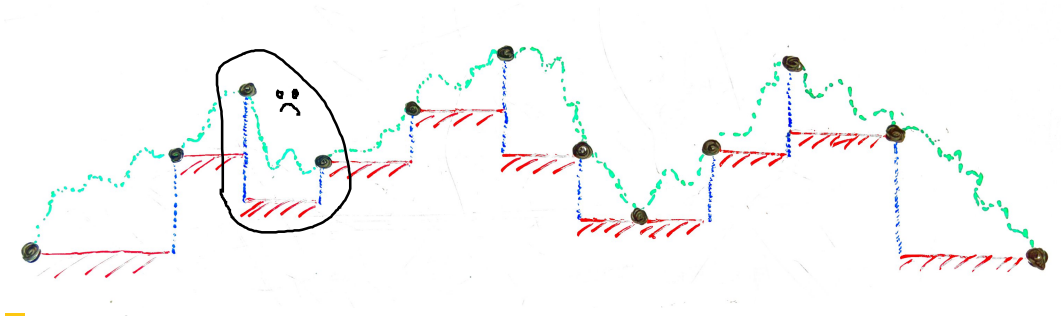
Formalize this notion of deviations

6.3.1 Maintaining a Boundary Invariant

Consider all positions that have been queried already $\langle x_1, x_2, \dots, x_m \rangle$ (in increasing order) along with their corresponding heights $\langle h_1, h_2, \dots, h_m \rangle$. We maintain an invariant that for each $i < m$, the Dyck path between positions x_i and x_{i+1} is constrained to lie above $\min(h_i, h_{i+1})$.

Why?

It is not even clear that this is always possible. After sampling the height of a particular position x_i as h_i (with $x_{i-1} < x_i < x_{i+1}$), the invariant is potentially broken on either side



■ **Figure 3** Error in third segment.

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 3):

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 $\text{HEIGHT}(x) = h$.

Since h is the minimum height along this interval, sampling the point x suffices to preserve the invariant.

6.3.2 Sampling the Lowest Achievable Height

For the first step, we need to sample the lowest height of the walk between x_i and x_{i+1} . Notice that we can assume $x_i < x_{i+1}$ without loss of generality (if $x_i > x_{i+1}$, swap them and proceed). Let's say that the boundary is currently $k' - 1$ units below h_i .

We know how to count the number of possible Dyck paths for any given boundary. Dividing by the total number of possible paths gives us precisely the CDF we need. This allows us to binary search to find the boundary.

We will use D_k to denote the number of paths that respect a boundary which is $k - 1$ units below h_i . So, in the first step, we compute $p = D_{k'/2}/D_{k'}$. This means that with probability p , the path never reaches height $h_i - k'/2$. Otherwise, the path must reach $h_i - k'/2$ but not $h_i - k'$. Note that we can also calculate the total number of such paths as $D_{k'} - D_{k'/2}$. We repeat this procedure, essentially performing binary search, until we find a k such that the path reaches height $h_i - k + 1$ (potentially multiple times), but never goes below it.

6.3.3 Sampling the Position of First Return

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 .

We will parameterize the position x the number of up-steps between x_i and x (See Figure 4). This quantity will be referred to as d such that $x - x_{i+1} = 2d + k - 1$. 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 $\text{poly}(\log n)$ factors) estimate of the probabilities, it suffices to estimate the number of paths asymptotically as well.

► **Lemma 18.** *If $d > \log^7 n$, then $D_{\text{left}}(d) = \Theta\left(\frac{2^{2d+k}}{\sqrt{d}} e^{-r_{\text{left}}(d)} \cdot \frac{k-1}{d+k-1}\right)$ where $r_{\text{left}}(d) = \frac{(k-2)^2}{2(2d+k-2)}$.*

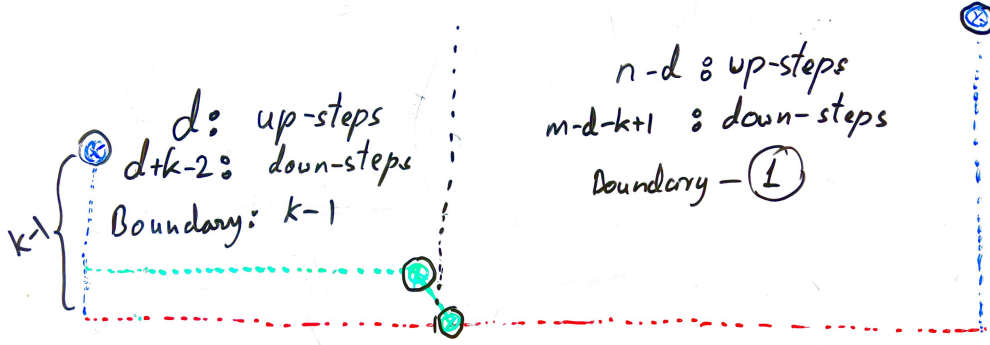


Figure 4 Zooming into (and flipping) the error in Figure 4

► **Lemma 19.** If $U+D-2d-k > \log^7 n$, then $D_{right}(d) = \Theta \left(\frac{2^{U+D-2d-k}}{\sqrt{U+D-2d-k}} e^{-r_{right}(d)} \cdot \frac{U-D+k}{U-d+1} \right)$ where $r_{right}(d) = \frac{(U-D-k-1)^2}{4(U+D-2d-k+1)}$.

Deal with smaller values of d

Deal with other values of d

6.3.4 Estimating the CDF

► **Lemma 20.** $D_{left}(d) \cdot D_{right}(d) = \Theta \left(\frac{2^{U+D}}{\sqrt{d(U+D-2d-k)}} e^{-r(d)} \cdot \frac{k-1}{d+k-1} \cdot \frac{U-D+k}{U-d+1} \right)$ where $r(d) = \mathcal{O}(\log^2 n)$.

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

► **Corollary 21.** The probability p_d of sampling d as the number of up-steps before the first approach to the boundary can be approximated as:

$$p_d = \Theta \left(\frac{2^{U+D} \cdot \frac{(k-1)(U-D+k)}{\sqrt{d(U+D-2d-k)(d+k-1)(U-d+1)}} \cdot e^{-\lfloor r(d) \rfloor}}{\binom{U+D}{U} - \binom{U+D}{D-k}} \right)$$

This is because the floor function only affects the value of the exponential by a factor of at most e .

D_{total} is not correct

► **Corollary 22.** We define a piecewise continuous function

$$\hat{q}(d) = \frac{2^{U+D} \cdot \frac{(k-1)(U-D+k)}{\sqrt{d(U+D-2d-k)(d+k-1)(U-d+1)}} \cdot e^{-\lfloor r(d) \rfloor}}{\binom{U+D}{U} - \binom{U+D}{D-k}}$$

We claim that $p_d = \Theta \left(\int_d^{d+1} \hat{q}(d) \right)$. Note that this integral has a closed form for a fixed value of $\lfloor r(d) \rfloor$.

Let the maximum value of $r(d)$ be $r_{max} = \mathcal{O}(\log^2 n)$.

► **Corollary 23.** We can compute the approximate normalized probabilities

$$q_d = \frac{\int_d^{d+1} \hat{q}(d)}{\int_1^n \hat{q}(d)}$$

such that $p_d = \Theta(q_d)$. Furthermore, we can also compute the CDF of q_d as:

$$Q_d = \frac{\int_1^{d+1} \hat{q}(d)}{\int_1^n \hat{q}(d)}$$

This allows us to sample from the distribution q_d and use Lemma 10 to indirectly sample from p_d .

References

- 1 Paul Erdos and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci.*, 5(1):17–60, 1960.
- 2 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. URL: <https://doi.org/10.4230/LIPIcs.ICALP.2017.6>, doi:10.4230/LIPIcs.ICALP.2017.6.
- 3 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.
- 4 Oded Goldreich, Shafi Goldwasser, and Asaf Nussboim. On the implementation of huge random objects. *SIAM Journal on Computing*, 39(7):2761–2822, 2010.
- 5 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.
- 6 Donald E Knuth. The art of computer programming, 3rd edn. seminumerical algorithms, vol. 2, 1997.
- 7 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.
- 8 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.
- 9 Moni Naor and Asaf Nussboim. Implementing huge sparse random graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 596–608. Springer, 2007.
- 10 Shlomi Reuveni. Catalan’s trapezoids. *Probability in the Engineering and Informational Sciences*, 28(03):353–361, 2014.
- 11 Duncan J Watts and Steven H Strogatz. Collective dynamics of ‘small-world’ networks. *Nature*, 393(6684):440–442, 1998.

A Dyck Path Generator

► **Theorem 24.** *There are $\frac{1}{n+1} \binom{2n}{n}$ Dyck paths for length $2n$.*

Proof from
catalan book

Proof. We first consider ballot sequences which are another catalan object. Formally, this is a sequence of $n+1$ +1s and n -1s such that any prefix sum of the sequence is nonnegative. Note that a ballot sequence can be transformed into a valid Dyck path by removing the last occurrence of +1. Similarly, a Dyck path can be converted to a valid ballot sequence by inserting a +1 after the last occurrence of a First we consider a *balanced* walk with n up and n down steps (without the boundary constraint). We will define a function that maps balanced random walks to Dyck paths. To do this, we represent a balanced walk with a sequence of $2n$ variables $\{X_1, X_2, \dots, X_{2n}\}$ consisting of $n+1$ and -1 values. ◀

A.1 Approximating Close-to-Central Binomial Coefficients

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

► **Lemma 25.**

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

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

Cite Asymptopia

► **Lemma 26.**

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

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

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

$$\Rightarrow \log R = \sum_{i=1}^{c\sqrt{n}/2} \log \left(\frac{n/2 - i + 1}{n/2 + i} \right) \quad (3)$$

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

$$\Rightarrow \binom{n}{k} = \binom{n}{n/2} e^{-c^2/2} \exp(\mathcal{O}(c^3/\sqrt{n})) \quad (5)$$

◀

A.2 Dyck Path Boundaries and Deviations

► **Lemma 27.** *Given a random walk of length $2n$ with exactly n up and down steps, consider a contiguous sub-path of length $2S$ that comprises of U up-steps and D down-steps i.e. $U + D = 2S$. Both $|S - U|$ and $|S - D|$ are $\mathcal{O}(\sqrt{S \log n})$ with probability at least $1 - 1/n^3$.*

Proof. We consider the random walk as a sequence of unbiased random variables $\{X_i\}_{i=1}^{2n} \in \{0, 1\}^{2n}$ with the constraint $\sum_{i=1}^{2n} X_i = n$. Here, 1 corresponds to an up-step and 0 corresponds

to a down step. Because of the constraint, X_i, X_j are negatively correlated for $i \neq j$ which allows us to apply Chernoff bounds. Now we consider a sub-path of length $2S$ and let U denote the sum of the X_i s associated with this subpath. Using Chernoff bound with $\mathbb{E}[X] = S$, we get:

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

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

► **Corollary 28.** *With high probability, every contiguous sub-path in the random walk (with U up and D down steps) satisfies the above property with high probability. Specifically, if $U + D = 2S$, then $|S - U|$ is upper bounded by $\sqrt{S \log n}$ with high probability for all contiguous sub-paths.*

Proof. We can simply apply Lemma 27 and union bound over all n^2 possible contiguous sub-paths. ◀

► **Lemma 29.** *Consider a contiguous sub-path of a simple Dyck path of length $2n$ where the sub-path is of length $2S$ comprising of U up-steps and D down-steps (with $U + D = 2S$). Both $|S - U|$ and $|S - D|$ are $\mathcal{O}(\log n \sqrt{S})$ with probability at least $1 - 1/n^3$.*

Proof. Because of Theorem 24, we can sample a Dyck path by first sampling a *balanced* random walk and then converting it to a Dyck path. In the worst case, the transformation only increases the *imbalance* in any interval by a factor of at most $\sqrt{2}$. So, we can simply use Corollary 28 to finish the proof. ◀

Finish

► **Lemma 30.** *There exists a constant c such that if $k > c \log n \sqrt{S}$, then the distribution of paths sampled without a boundary (normal random walks) is statistically $1/n^2$ -close in L_1 distance to the distribution of Dyck paths.*

Proof. ◀

Change statement. What if other end is much lower?

A.3 Computing Probabilities

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:

$$\log(m!) = m \log m - m + \frac{1}{2} \log(2\pi m) + \frac{1}{12m} - \frac{1}{360m^3} + \frac{1}{1260m^5} - \dots \quad (6)$$

Oracle for estimating probabilities:

► **Lemma 31.**

Given a probability expression of the form $p_d = \frac{D_{left} \cdot D_{right}}{D_{total}}$ and a parameter n , there exists a $\text{poly}(\log n)$ time oracle that returns a $(1 \pm 1/n^2)$ multiplicative approximation to p_d .

Proof. We first compute a $1/n^2$ additive approximation to $\log p_d$. Note that this is possible because $\log p_d$ can be written as a sum of logarithms of factorials. So, we can use the series expansion from Equation 6 up to $\mathcal{O}(\log n)$ terms.

Now, we can exponentiate the approximation to obtain $p_d \cdot e^{\mathcal{O}(1/n^2)} \approx p_d (1 + \mathcal{O}(1/n^2))$ ◀

Point to section referencing the left right/ total.

Not obvious how

A.4 Sampling the Height

fix

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

► **Lemma 32.** For $x < 1$ and $k \geq 1$,

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

► **Lemma 12.** $D_{left} \leq c_1 \frac{k \cdot \log n}{\sqrt{S}} \cdot \binom{S}{D-d}$ for some constant c_1 .

Proof. This involves some simple manipulations.

$$D_{left} = \binom{S}{D-d} - \binom{S}{D-d-k} \tag{7}$$

$$= \binom{S}{D-d} \cdot \left[1 - \frac{(D-d)(D-d-1) \cdots (D-d-k+1)}{(S-D-d+k)(S-D-d+k-1) \cdots (S-D-d+1)} \right] \tag{8}$$

$$\leq \binom{S}{D-d} \cdot \left[1 - \left(\frac{D-d-k+1}{S-D-d+k} \right)^k \right] \tag{9}$$

$$\leq \binom{S}{D-d} \cdot \left[1 - \left(\frac{U+d+k-(U-D+d+k-1)}{U+d+k} \right)^k \right] \tag{10}$$

$$\leq \binom{S}{D-d} \cdot \left[1 - \left(\frac{U+d+k - \mathcal{O}(\log n \sqrt{S})}{U+d+k} \right)^k \right] \tag{11}$$

$$\leq \Theta \left(\frac{k \log n}{\sqrt{S}} \right) \cdot \binom{S}{D-d} \tag{12}$$

◀

► **Lemma 13.** $D_{right} < c_2 \frac{k' \cdot \log n}{\sqrt{S}} \cdot \binom{S}{U-d}$ for some constant c_2 .

Proof.

$$D_{right} = \binom{S}{U-d} - \binom{S}{U-d-k'} \quad (13)$$

$$= \binom{S}{U-d} \cdot \left[1 - \frac{(U-d)(U-d-1) \cdots (U-d-k'+1)}{(S-U-d+k')(S-U-d+k'-1) \cdots (S-U-d+1)} \right] \quad (14)$$

$$\leq \binom{S}{U-d} \cdot \left[1 - \left(\frac{U-d-k'+1}{S-U+d+k'} \right)^{k'} \right] \quad (15)$$

$$\leq \binom{S}{U-d} \cdot \left[1 - \left(\frac{2D-U-d-k+1}{2U-D+k+d} \right)^{k'} \right] \quad (16)$$

$$\leq \binom{S}{U-d} \cdot \left[1 - \left(\frac{U+k+d-(2U-2D+2d+2k-1)}{U+k+d} \right)^{k'} \right] \quad (17)$$

$$\leq \binom{S}{U-d} \cdot \left[1 - \left(\frac{U+k+d-\mathcal{O}(\log n \sqrt{S})}{U+k+d} \right)^{k'} \right] \quad (18)$$

$$\leq \Theta \left(\frac{k' \log n}{\sqrt{S}} \right) \cdot \binom{S}{U-d} \quad (19)$$

◀

change state-
ment

► **Lemma 33.** $D_{tot} \geq \binom{2S}{2D} \cdot \left[1 - \left(1 - \frac{k'}{2U+1} \right)^k \right]$.

Proof.

$$D_{tot} = \binom{2S}{2D} - \binom{2S}{2D-k} \quad (20)$$

$$= \binom{2S}{2D} \cdot \left[1 - \frac{(2D)(2D-1) \cdots (2D-k+1)}{(2S-2D+k)(2S-2D+k-1) \cdots (2S-2D+1)} \right] \quad (21)$$

$$\geq \binom{2S}{2D} \cdot \left[1 - \left(\frac{2D-k+1}{2S-2D+1} \right)^k \right] \quad (22)$$

$$\geq \binom{2S}{2D} \cdot \left[1 - \left(\frac{2U-(2U-2D+k-1)}{2U+1} \right)^k \right] \quad (23)$$

$$\geq \binom{2S}{2D} \cdot \left[1 - \left(\frac{(2U+1)-k'}{2U+1} \right)^k \right] \quad (24)$$

$$\geq \binom{2S}{2D} \cdot \left[1 - \left(1 - \frac{k'}{2U+1} \right)^k \right] \quad (25)$$

$$(26)$$

◀

Reference pre-
vious lemma

► **Lemma 11.** When $kk' > 2U+1$, $D_{tot} > \frac{1}{2} \cdot \binom{2S}{2D}$.

Proof. When $kk' > 2U+1 \implies k > \frac{2U+1}{k'}$, we will show that the above expression is greater than $\frac{1}{2} \binom{2S}{2D}$. 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 14.** When $kk' \leq 2U + 1$, $D_{tot} < c_3 \frac{k \cdot k'}{S} \cdot \binom{2S}{2D}$ 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

$$1 - \left(1 - \frac{k'}{2U+1}\right)^k \tag{27}$$

$$\leq \frac{kk'}{2U+1} - \frac{k(k-1)}{2} \cdot \frac{k'^2}{(2U+1)^2} \tag{28}$$

$$\leq \frac{kk'}{2U+1} - \frac{k^2 k'^2}{2(2U+1)^2} \tag{29}$$

$$\leq \frac{kk'}{2U+1} \left(1 - \frac{kk'}{2(2U+1)}\right) \tag{30}$$

$$\leq \frac{kk'}{2(2U+1)} \leq \frac{kk'}{\Theta(S)} \tag{31}$$

$$\tag{32}$$

◀

A.5 First Return Sampling

► **Lemma 18.** If $d > \log^7 n$, then $D_{left}(d) = \Theta\left(\frac{2^{2d+k}}{\sqrt{d}} e^{-r_{left}(d)} \cdot \frac{k-1}{d+k-1}\right)$ where $r_{left}(d) = \frac{(k-2)^2}{2(2d+k-2)}$.

Proof. In what follows, we will drop constant factors: Refer to Figure 4 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 includes 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:

$$D_{left}(d) = \binom{2d+k-2}{d} - \binom{2d+k-2}{d-1} \tag{33}$$

$$= \binom{2d+k-2}{d} \left[1 - \frac{d}{d+k-1}\right] = \binom{2d+k-2}{d} \frac{k-1}{d+k-1} \tag{34}$$

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 29, we see that $\frac{k-2}{\sqrt{z}}$ should be $\mathcal{O}(\log n)$. If this is not the case, we can simply return 0 because the probability associated with this value of d is negligible. Since $z > \log^7 n$, we can apply Lemma 26 to get:

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

◀

► **Lemma 19.** *If $U+D-2d-k > \log^7 n$, then $D_{right}(d) = \Theta\left(\frac{2^{U+D-2d-k}}{\sqrt{U+d-2d-k}} e^{-r_{right}(d)} \cdot \frac{U-D+k}{U-d+1}\right)$ where $r_{right}(d) = \frac{(U-D-k-1)^2}{4(U+D-2d-k+1)}$.*

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:

$$D_{right}(d) = \binom{U+D-2d-k+1}{U-d} - \binom{U+D-2d-k+1}{U-d+1} \quad (35)$$

$$= \binom{U+D-2d-k+1}{U-d} \left[1 - \frac{D-d-k-1}{U-d+1}\right] \quad (36)$$

$$= \binom{U+D-2d-k+1}{U-d} \frac{U-D+k}{U-d+1} \quad (37)$$

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 29, we see that $\frac{k-2}{\sqrt{z}}$ should be $\mathcal{O}(\log n)$. If this is not the case, we can simply return 0 because the probability associated with this value of d is negligible. Since $z > \log^7 n$, we can apply Lemma 26 to get:

$$D_{right}(d) = \Theta\left(\left(\frac{z}{z/2}\right) e^{\frac{(U-D+k-1)^2}{2z}} \frac{U-D+k}{U-d+1}\right) \quad (38)$$

$$= \Theta\left(\frac{2^{U+D-2d-k}}{\sqrt{U+D-2d-k}} e^{\frac{(U-D+k-1)^2}{2(U+D-2d-k+1)}} \frac{U-D+k}{U-d+1}\right) \quad (39)$$

◀