Sub-linear Algorithms for Graph Problems

by

Anak Yodpinyanee

S.M., Massachusetts Institute of Technology (2014)
B.S., Harvey Mudd College (2012)

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2018

Signature redacted

Author .....
Department of Electrical Engineering and Computer Science
August 10, 2018

Signature redacted

Certified by...
Ronitt Rubinfeld
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Signature redacted

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Sub-linear Algorithms for Graph Problems

by

Anak Yodpinyanee

Submitted to the Department of Electrical Engineering and Computer Science
on August 10, 2018, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

In the face of massive data sets, classical algorithmic models, where the algorithm reads the entire input, performs a full computation, then reports the entire output, are rendered infeasible. To handle these data sets, alternative algorithmic models are suggested to solve problems under the restricted, namely sub-linear, resources such as time, memory or randomness. This thesis aims at addressing these limitations on graph problems and combinatorial optimization problems through a number of different models.

First, we consider the *graph spanner* problem in the *local computation algorithm* (LCA) model. A graph spanner is a subgraph of the input graph that preserves all pairwise distances up to a small multiplicative stretch. Given a query edge from the input graph, the LCA explores a sub-linear portion of the input graph, then decides whether to include this edge in its spanner or not – the answers to all edge queries constitute the output of the LCA. We provide the first LCA constructions for 3 and 5-spanners of general graphs with almost optimal trade-offs between spanner sizes and stretches, and for fixed-stretch spanners of low maximum-degree graphs.

Next, we study the *set cover* problem in the *oracle access* model. The algorithm accesses a sub-linear portion of the input set system by probing for elements in a set, and for sets containing an element, then computes an approximate minimum set cover: a collection of an approximately-minimum number of sets whose union includes all elements. We provide probe-efficient algorithms for set cover, and complement our results with almost tight lower bound constructions. We further extend our study to the *LP-relaxation* variants and to the *streaming* setting, obtaining the first streaming results for the fractional set cover problem.

Lastly, we design *local-access generators* for a collection of fundamental *random graph models*. We demonstrate how to generate graphs according to the desired probability distribution in an *on-the-fly* fashion. Our algorithms receive probes about arbitrary parts of the input graph, then construct just enough of the graph to answer these probes, using only polylogarithmic time, additional space and random bits per probe. We also provide the first implementation of random neighbor probes, which is a basic algorithmic building block with applications in various huge graph models.

Thesis Supervisor: Ronitt Rubinfeld
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I am deeply grateful to my research adviser: Ronitt Rubinfeld; my thesis committee: Piotr Indyk and Virginia Vassilevska Williams; and my academic counselor: Dina Katabi, for their invaluable guidance throughout my PhD program. I would like to thank my collaborators: Maryam Aliakbarpour, Amartya Shankha Biswas, Themistoklis Gouleakis, Piotr Indyk, Sepideh Mahabadi, Merav Parter, John Peebles, Ronitt Rubinfeld, Jonathan Ullman, and Ali Vakilian, for their helpful contributions in our research projects and publications.

This thesis is completed under the supervision of Ronitt Rubinfeld, with financial support from the NSF Grant No. CCF-1217423, CCF-1065125, CCF-1733808, CCF-1420692, CCF-1650733, IIS-1741137, as well as the Development and Promotion of Science and Technology Talented Project scholarship, Royal Thai Government.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In the face of massive data sets, classical algorithmic models, where the algorithm reads the entire input, performs a full computation, then reports the entire output, are rendered infeasible. To handle these data sets, alternative algorithmic models are suggested to solve problems under the restricted resources such as time, memory or randomness. In particular, to study sub-linear time algorithms for graph problems, it is necessary to describe how an algorithm may access its input, how it can offer its output, and how much power of computation it has, as well as how we measure the algorithm's performance. This thesis aims at addressing the massive data sets on graph problems through a number of different models.

Let us first consider sub-linear *time* algorithms, where the algorithm do not have enough time to read the entire input. To this end, many *oracle access models* have been defined: the algorithm interacts with an *oracle* that provides access to the input. For example, suppose that we wish to solve a graph problem, such as computing a minimal vertex cover or a graph spanner. Typically in this setting, the algorithm is allowed to make certain types of *probes* to the oracle and learn about the *desired* part of the graph. For instance, probes of the form "are u and v adjacent?" or "what is the i^{th} neighbor of u?" are frequently studied in sparse graphs and dense graphs, respectively. Alternative types of probes may instead offer *random* information about the input graph. For example, "sample a uniform random neighbor or u" allows us to perform random walks, or "sample a uniform random edge" aids in approximating certain graph parameters. The oracle access models can also be further extended beyond the graph setting. Consider the set cover problem: the input set system can be represented though its element-set dependency graph, which naturally suggests analogous probe types, such as "does S contain e?" or "what is the i^{th} element of S?". These allowed probes heavily depend on the nature of the problem, such as how the large input is stored in the database, or how one may actually inspect these aspects in a real-world network.

Given an oracle access, one may construct sub-linear time algorithms for computing an output, given that the output is of sufficiently small size, such as decision problems (e.g., "is the input graph bipartite?"), and parameter approximation problems (e.g., "what is the size of the minimum set cover?"). Some problems that potentially have large solutions may indeed admit sparse solutions that can be described succinctly, such as the (approximate) fractional set cover problem. However, when considering massive graphs containing millions or even billions of vertices (e.g., the underlying

graphs of WWW or Twitter), even the output can be too large for a single processor to store: the problem of computing a sparse spanning subgraph, for example, has solutions of linear size. For these problems we consider the model of *local computation algorithms* (LCAs). In addition to the oracle access, an LCA is also given a *query*[1] specifying a single location in the output. The algorithm must compute the specified part of the output, using very little resources and definitely without computing the whole output up-front. For the spanning subgraph problem, the algorithm is asked "is edge e in the subgraph?", and the yes/no answers returned by the algorithm on all edges must be consistent with a single valid spanning subgraph. Moreover, the algorithm must compute its answer independently without storing information for future invocation: the algorithm must be parallelizable and query-oblivious.

In terms of performance measurement of sub-linear time algorithms, we naturally aim to optimize the *time complexity* of our algorithms, either for the overall computation, or for answering a single query in case of LCAs. Other classical notions can also be applied, such as the *space complexity* or the required amount of *random bits*. In most study of the oracle access model, probes to the input are largely assumed to be computationally free or require very little overhead. Nonetheless, we may consider the *probe complexity* of an algorithm: how many probes does the algorithm need to make to the oracle? On the one hand, this notion considers the pure theoretical question of how much informational quantity is necessary to solve the computational problem. On the other hand, probe complexity reflects the real-world bottleneck performance when the probe access is costly: for example, the communication to the database may be overwhelmingly large that the computation time becomes negligible, or a probe may incur an actual "field work" investigation of a physical network to answer.

Real-world networks are frequently too large to be fully stored or impossible to be fully observed. One important method for studying large graphs is to propose random graph models that generate random graphs satisfying prominent behaviors observed in these real-world networks, such as the power-law degree distribution or the small-world phenomenon. Many works have been devoted to efficient generation of these random graphs; likewise, it is natural to study how to realize oracles for these random graph models. We study the *local-access generators* that provide various types of probes to random graphs, including fundamental models such as the Erdős-Rényi model, the Stochastic Block model, and the Small-World model. A local-access generator is required to provide consistent answers according to its graph, which is incrementally sampled from the distribution sufficiently close to that of the desired random graph model. As the design of sub-linear time algorithms generally considers the processing time of the oracle rather negligible, it is important for our generators to be extremely efficient in computing their answers for the given probes.

Lastly, aside from sub-linear time algorithms, we consider sub-linear *space* algorithms, namely in the *streaming model*. In the streaming model, streaming algorithms are allowed to inspect the input

[1]We clarify the difference between queries and probes for LCAs. The user gives a *query* to the LCA, specifying the desired part of the output that needs to be computed. The LCA is a sub-linear time algorithm and therefore occupies the oracle access model: it makes *probes* to the oracle to learn about the input instance in order to compute its answer to the given query. This characteristic of LCAs, computing their respective "local" parts of the output, is similar to that of Distributed Local algorithms – likewise, the LCA model is also known as the Centralized Local model.

in a sequential fashion within typically a few number of passes in order to compute their output, and may only use sub-linear computational memory overall: they can *see*, but cannot *store*, the entire input. The power of computation of streaming algorithms is very distinct from that of sub-linear time algorithms under the oracle access model: sub-linear time algorithms cannot see the entire input, but instead they have the ability to *adaptively* probe the input in an interactive fashion. Nonetheless, we instead demonstrate intimate connections between algorithm design techniques under the oracle access model and the streaming model, inventing techniques that are useful in both settings along the way in our study.

1.1 Local Computation Algorithms for Graph Spanners

In Chapter 2, we study the local computation algorithms for graph spanners. This chapter is based on joint work with Merav Parter, Ronitt Rubinfeld and Ali Vakilian. Here, we begin by giving more detailed descriptions of graph spanners and the LCA model, then formally state our contribution.

Graph spanners. A graph spanner is a fundamental graph structure, which is a sparse subset of edges of a source graph $G = (V, E)$ that faithfully preserves the pairwise distances in G up to a small multiplicative stretch. More formally, given an unweighted undirected graph $G = (V, E)$, a subgraph $H = (V, E')$ where $E' \subseteq E$ is a *k-spanner* of G if, for any pair of vertices $u, v \in V$ that are connected, $\mathsf{dist}_G(u, v) \leq \mathsf{dist}_H(u, v) \leq k \cdot \mathsf{dist}_G(u, v)$; the parameter $k \geq 1$ is called the *stretch factor*. The notion of spanners was introduced by Peleg and Schäffer [PS89] and has been used widely in different applications such as routing schemes [AP92, Pel00], synchronizers [PU89, AP90], SDD's [ST11], spectral sparsifiers [KP12].

The common objective in the computation of spanners is to achieve the best-known existential size-stretch trade-off *efficiently*. It is folklore that for every n-vertex graph G, there exists a $(2k-1)$-spanner $H \subseteq G$ with $O(n^{1+1/k})$ edges. In particular, if the *girth conjecture* of Erdős [Erd65] is true, then this size-stretch trade-off is optimal.

Local Computation Algorithms. LCAs are algorithms that compute the queried part of the output, without computing the whole output, by examining only a small (sub-linear) portion of the input; all local answers given by an LCA are consistent with a single valid output to the computational problem. Specifically, in our context, the algorithm should locally decide whether a given edge $(u, v) \in E$ belongs to the output (sparse) spanner or not: overall, the set of edges chosen by the LCA, $\{e \in E : \text{the LCA returns } \mathsf{yes} \text{ on query } e\}$ must form a valid spanner for G with the desired stretch factor and total number of edges. In a sense, such LCAs give the user the "illusion" that a *specific* sparse spanner for the graph is maintained, without ever fully computing it.

This model has been proposed in [RTVX11], aiming to capture many different types of problems, including locally list-decodable codes, local decompression and local reconstructors/filters. LCAs have been established for a large collection of problems, including Maximal Independent Set, Maximum Matching, and Vertex Cover [RTVX11, ARVX12, MV13, EMR14, MPV18, RV16, LRY17]. The study of LCAs with *sublinear* probe complexity for *nearly linear* size spanners (or *sparsifiers*) is initiated in [LRR14, LRR16] for some restricted families of graphs such as minor-closed families. Recently, Lenzen and Levi [LL18] designed the first LCA for sparsifiers in *general graphs*.

In this work, we consider the oracle access model allowing the following types of probes [Gol11, GGR98]: NEIGHBOR probe ("what is the i^{th} neighbor of u"?), DEGREE probe ("what is $\deg(u)$?") and ADJACENCY probe ("are u and v neighbors"?). In particular, the answer to an ADJACENCY probe on an ordered pair $\langle u, v\rangle$ is the index of v in $\Gamma(u)$ if the edge exists, and $\perp$ otherwise.

Our results. We provide the first sub-linear time/probe LCAs for spanners with fixed stretch values, and in particular, achieve optimal size/stretch trade-offs for 3 and 5-spanners (up to poly-logarithmic factors). Along the way, we establish a collection of fundamental sub-linear techniques for handling computational problems via access oracles, even in presence of vertices with potentially linear degrees. In more details:

- For general graphs and for parameter $r \in \{2, 3\}$, there exist an LCA for constructing a $(2r-1)$-spanner $H \subseteq G$ with $\widetilde{O}(n^{1+1/r})$ edges and probe complexity of $\widetilde{O}(n^{1-1/2r})$. These size/stretch trade-offs are best possible (up to polylogarithmic factors).

- For every $k \geq 1$ and n-vertex graph with maximum degree Δ, there exists an LCA for constructing of an $O(k^2)$-spanner $H \subseteq G$ with $\widetilde{O}(n^{1+1/k})$ edges and probe complexity of $\widetilde{O}(\Delta^4 n^{2/3})$, hence the probe complexity of the algorithm is sublinear for $\Delta = O(n^{1/12-\epsilon})$. This improves upon, and extends the recent work of [LL18], that obtained an $O(n)$ size subgraph with stretch of $\widetilde{O}(\text{poly}(\Delta, \log n))$ and a similar probe complexity.

- Despite the $n^{\Theta(1)}$ probe complexities, all of our LCAs only require $\text{poly}(\log n)$ independent random bits, via a novel analysis of graph connectivity with bounded independence.

- We complement these constructions by providing lower bound results for the probe complexity of LCAs for the simpler task of *spanning subgraphs*: Any local algorithm that with success probability of at least $2/3$ maintains a sparse spanner of the input graph with $o(m)$ edges (no matter what the stretch factor is), has probe complexity $\Omega(\min\{\sqrt{n}, n^2/m\})$, where m is the number of edges in G.

1.2 Set Cover and Fractional Set Cover

1.2.1 Set Cover in the Oracle Access Model

In Chapter 3 of this thesis, we study the classic set cover problem from the perspective of sub-linear algorithms. This chapter is based on joint work with Piotr Indyk, Sepideh Mahabadi, Ronitt Rubinfeld and Ali Vakilian. The full version of this chapter appears in [IMR+18]. We now give the formal definition of the Set Cover problem, define the oracle access model for accessing the input set cover instance (e.g., a set system), then discuss our results.

Set Cover. Set Cover is a classic combinatorial optimization problem, in which we are given a set (universe) of n elements $\mathcal{U} = \{e_1, \cdots, e_n\}$ and a collection of m sets $\mathcal{F} = \{S_1, \cdots, S_m\}$. The goal is to find a *minimum set cover* of $\mathcal{U}$, i.e., a collection of sets in $\mathcal{F}$ whose union is $\mathcal{U}$, of minimum size. Set Cover is well-studied, and finds applications in operations research [GW97, KK82, BCS09],

information retrieval and data mining [SG09], learning theory [KV94], web host analysis [CKT10], among many others.

Although the problem of finding an optimal solution is NP-complete, a natural greedy algorithm which iteratively picks the "best" remaining set (the set that covers the most number of uncovered elements) is widely used. The algorithm finds a solution of size at most $k \ln n$ where k is the optimum cover size, and can be implemented to run in time linear in the input size. The fractional variant can be approximated up to a factor arbitrarily close to 1, using an algorithm that runs in nearly-linear time as well (see e.g., [You01, KY14] and the references therein). However, the input size itself could be as large as $\Theta(mn)$, so for large data sets even reading the input might be infeasible.

This raises a natural question: is it possible to compute an approximate solution to minimum set cover in *sub-linear time*? This question was previously addressed in [NO08, YYI12], who showed that (for the integral set cover) one can design constant running-time algorithms by simulating the greedy algorithm, under the assumption that the sets are of constant size and each element occurs in a constant number of sets. However, those constant-time algorithms have a few drawbacks. In particular, they only provide a mixed multiplicative/additive guarantee (the output cover size is guaranteed to be at most $k \cdot \ln n + \epsilon n$). Furthermore, the dependence of their running times on the maximum set size is exponential, and finally, they only output the (approximate) minimum set cover size, not the cover itself.

Oracle access model for set systems. As in the prior works [NO08, YYI12] on Set Cover, our algorithms and lower bounds assume that the input can be accessed via the following two types of probes:

- EltOf: given a set S_i and an index j, the oracle returns the j-th element of S_i.
- SetOf: given an element e_i and an index j, the oracle returns the j-th set containing e_i.

Both operations are natural, providing a "two-way" connection between the sets and the elements. Furthermore, for some graph problems modeled by Set Cover (such as Dominating Set or Vertex Cover), such oracles are essentially equivalent to the NEIGHBOR probe. We also note that in another popular oracle access model employing the MEMBERSHIP probes, where we can probe whether an element e is contained in a set S (analogously to the ADJACENCY probe), it can be easily seen that even checking whether a feasible cover exists requires $\Omega(mn)$ probes.

Our results. We provide sub-linear algorithms for set cover that, unlike prior work, construct actual solutions (rather than approximating their sizes) with fully multiplicative approximate guarantee, and do not rely on any assumptions on the cardinalities of sets or occurrences of elements. Formal details are provided as follows.

- We show an adaptation of the streaming algorithm presented in [HIMV16] to the sub-linear oracle access model, that returns an α-approximate cover using $\widetilde{O}(m(n/k)^{1/(\alpha-1)} + nk)$ probes to the input, where k denotes the value of a minimum set cover.

- We complement this upper bound by proving that for lower values of k, the required number of probes is $\widetilde{\Omega}(m(n/k)^{1/(2\alpha)})$, even for estimating the optimal cover size. We prove that even checking whether a given collection of sets covers all the elements would require $\Omega(nk)$

probes. These two lower bounds provide strong evidence that the upper bound is almost tight for certain values of the parameter k.

- We show that this bound is not optimal for larger values of the parameter k, as there exists a $(1+\varepsilon)$-approximation algorithm with $\widetilde{O}(\frac{mn}{k\varepsilon^2})$ probes. We show that this bound is essentially tight for sufficiently small constant ε, by establishing a lower bound of $\widetilde{\Omega}(mn/k)$ probe complexity.

Our lower-bound results follow by carefully designing two distributions of instances that are hard to distinguish. In particular, our first lower bound involves a probabilistic construction of a certain set system with a minimum set cover of size αk, with the key property that a small number of "almost uniformly distributed" modifications can reduce the minimum set cover size down to k. Thus, these modifications are not detectable unless a large number of probes are asked. We believe that our probabilistic construction technique might find applications to lower bounds for other combinatorial optimization problems.

1.2.2 Fractional Set Cover in the Streaming Model and the Oracle Access Model

In Chapter 4, we study the Fractional Set Cover problem, both in the streaming model and the oracle access model. This chapter is based on joint work with Piotr Indyk, Sepideh Mahabadi, Ronitt Rubinfeld, Jonathan Ullman and Ali Vakilian. The full version of the streaming section of this work appears in [IMR+17]. We now define the fractional set cover problem, state our results for both models, then discuss some of the connections between algorithm design techniques in the two models that we observe in our study, in both the integral and the fractional variants of Set Cover.

Fractional Set Cover. Fractional Set Cover is the continuous relaxation of the set cover problem over a universe of n elements and a collection of m sets, where each set $S \in \mathcal{F}$ can be picked fractionally, with a value in $[0,1]$. That is, each variable x_S corresponding to the set S is assigned a value from $[0,1]$, such that for each element e its "fractional coverage" $\sum_{S:e\in S} x_S$ is at least 1, and the sum $\sum_S x_S$ is minimized.

Our results. For the streaming model, we present a randomized $(1+\varepsilon)$-approximation algorithm that makes p passes over the data, and uses $\widetilde{O}(mn^{O(1/p\varepsilon)}+n)$ memory space. The algorithm works in both the set arrival and the edge arrival models. To the best of our knowledge, this is the first streaming result for the fractional set cover problem. We design our streaming algorithms based on the multiplicative weights update (MWU) method. Using many techniques developed in the streaming model, we then extend our results to the oracle access model.

For the oracle access model, we show three sub-linear time $(1+\epsilon)$-approximate algorithms. The first two algorithms are most efficient when k is small. Specifically, one algorithm has $\widetilde{O}(\frac{mk}{\varepsilon^5}+\frac{nk}{\varepsilon^2})$ probe complexity and runtime. Our second algorithm reduces the number of probes further by a factor of (roughly) $\sqrt{k}$ while increasing the runtime by the same factor. Our third algorithm is most efficient when k is large, and uses $O(\frac{mn}{k\epsilon})$ probes and similar running time. We also show that, for constant values of k, the probe complexity of any algorithm for fractional set cover must depend linearly on both m and n: we complement these results via a lower bound of $\Omega(\frac{n+m}{k})$ for sufficiently small constant ϵ, employing similar techniques from the integral variant.

The first two algorithms for Fractional Set Cover use the MWU framework, which iteratively identifies unsatisfied constraints and re-calibrates their "importance". The first algorithm implements each constraint-checking round separately by using random sampling. Our second algorithm reduces the number of probes by re-using the random samples in multiple rounds. Since the MWU algorithm updates the constraints in an *adaptive* fashion, we employ the *adaptive data analysis* framework of Dwork et al. [DFH+15] and Bassily et al. [BNS+16], which allows us to control the probabilistic dependencies. To the best of our knowledge this is the first application of the framework to the design of sub-linear algorithms. Nonetheless, the best existing algorithm is based on a much cleaner insight from [KY14]. While our result is not the state-of-the-art algorithm, we speculate that this discovered connection may have applications in some context.

Shared techniques between the oracle access model and the streaming model. In our study of both variants of the Set Cover problem, algorithms from both models share the essential scheme of progressing in multiple rounds: in each round we detect and selectively sample the "relatively unsolved" portion of the problem, then solve it in an offline fashion, and update the maintained solution. For the Set Cover problem, we leverage existing sampling techniques from the streaming setting to construct probe-efficient algorithms in the oracle access model. These algorithms straightforwardly compute good solutions to the sampled subproblem that also turn out to significantly reduce the overall problem size in each round (e.g., by choosing sets that cover many uncovered elements).

On the other hand, for the *Fractional* Set Cover problem, we design an algorithmic framework for both models based on the multiplicative weights update (MWU) method. Unlike the integral setting, the overall problem size is not reduced due to the continuous nature of the fractional variant. As briefly discussed earlier, following the MWU method, in each MWU round, we sample subproblems (elements) with respect to their weights (importance). In our study, we invent building blocks that are useful in both settings, such as "advance sampling" for the simulation of multiple MWU rounds despite these "updated weights" – this technique aids in reducing the total number of probes/passes in our algorithms.

1.3 Local-Access Generators for Random Graphs

In Chapter 5, we study the problem of constructing local-access generators for graphs drawn from the Erdős-Rényi $G(n, p)$ model, and the Stochastic Block model, and the Small-World model. This chapter is based on joint work with Amartya Shankha Biswas and Ronitt Rubinfeld. Here, we provide a simplified definition of the local-access generator, then discuss our results.

Local-access generators for random graphs. Random graphs are used in a variety of disciplines to model communication networks, the WWW, and social networks. The naïve approach that first generates the entire random graph before executing any procedures on the generated graph becomes infeasible when the size of the random graph is too large. Moreover, in many cases it is not important to have the entire graph prepared beforehand: we may only need to probe the graph in an increasing fashion. To handle this scenario, we consider a *local-access generator* which incrementally constructs the random graph locally, at the probed 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 probe is significantly more efficient than generating the whole random graph.

The following formal definition is largely inspired by that of [ELMR17] but incorporates minor changes. A local-access generator is a data structure that provides oracle access to a random graph G drawn from a distribution $\mathcal{D}$ over a family of unweighted graphs (undirected or directed). We require the following properties from the local-access generator: (1) the responses of the local-access generator to all probes throughout the entire execution must be consistent with a single graph, (2) the random graphs provided by the generator must be sampled from some distribution $\mathcal{D}'$ that is (n^{-c})-close to the desired distribution $\mathcal{D}$, and (3) the resources, namely the computation time, additional random bits required, and additional space usage per probe, must be small, preferably polylog(n).

Our results. Our work focus on undirected graphs with independent edge probabilities; that is, each edge is chosen as an independent Bernoulli random variable (but the biases for different edges may be correlated according to a global rule). 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 works of local-access implementations for random graphs, we support VERTEX-PAIR probes ("are u and v adjacent?") and NEXT-NEIGHBOR probes ("return a neighbor of v that has not been returned before"). In addition, we introduce a new RANDOM-NEIGHBOR probe ("return a *uniform random* neighbor of v"). We also give the first local-access generation procedure for ALL-NEIGHBORS probes ("return all neighbors of v") in the (sparse and directed) Kleinberg's Small-World model. Note that, in the sparse case, an ALL-NEIGHBORS probe can be used to simulate the other types of probes efficiently. All of our generators require no pre-processing time, and answer each probe using polylog(n) time, random bits, and additional space.

Chapter 2

Graph Spanners

2.1 Introduction

One of the fundamental structural problems in graph theory is to find a *sparse* structure which preserves the pairwise distances of vertices. In many applications, it is crucial for the sparse structure to be a *subgraph* of the input graph; this problem is called the *spanner* problem. For an input graph $G = (V, E)$, a *k-spanner* $H \subseteq G$ (for $k \geq 1$) satisfies that for any $v, u \in V$, the distance from v to u in H is at most k times the distance from v to u in G, where k is referred to as the *stretch* of the spanner. Furthermore, to reduce the *cost* of the solution, it is desired to output a minimum size/weight such subgraph H. The notion of spanners was introduced by Peleg and Schäffer [PS89] and has been used widely in different applications such as routing schemes [AP92, Pel00], synchronizers [PU89, AP90], SDD's [ST11], spectral sparsifiers [KP12].

It is folklore that for every n-vertex graph G, there exists a $(2k-1)$-spanner $H \subseteq G$ with $O(n^{1+1/k})$ edges. In particular, if the *girth conjecture* of Erdős [Erd65] is true, then this size-stretch trade-off is optimal.

Spanners have been considered by now in many different models: distributed algorithms [DG08, DGP07, BS07, DGPV08, DGPV09, Pet10, EN17], dynamic streaming [AGM12, KW14], parallel computing [MPVX15] and dynamic algorithms [Elk11, BKS12, BK16]. In all these settings the objective is to compute *efficiently* (under a particular cost measure) a sparse spanner H with small stretch.

Local computation of small stretch spanners. When the graph is so large that it does not fit into the main memory, the existing algorithms are not sufficient for computing a spanner. Instead, we aim at designing an algorithm that answers *queries* of the form "is (u, v) in the spanner?" without computing the whole solution upfront. One way to get around this issue is to consider the *Local Computation Algorithms (LCA)* model (also known as the *Centralized Local model*) [RTVX11, ARVX12]. There can be many different plausible k-spanners; however, the goal of LCAs for the k-spanner problem is to design an algorithm that, given access to *primitive* probes (i.e. NEIGHBOR, DEGREE and ADJACENCY probes) on the input graph G, for each *query* on an edge $e \in E(G)$ *consistently* with respect to a *unique* k-spanner $H \subseteq G$ (picked by the LCA arbitrarily), outputs whether $e \in H$. The performance of the LCA is measured based on the *quality of solution* (i.e. number of edges in H)

and the *probe complexity* (the maximum number of probes per each query) of the algorithm.[1] An LCA gives us the "illusion" as if we have query access to a precomputed k-*spanner* of G.

LCAs have been established by now for a large collection of problems, including Maximal Independent Set, Maximum Matching, and Vertex Cover [RTVX11, ARVX12, MV13, EMR14, MPV18, RV16, LRY17]. The study of LCAs with *sublinear* probe complexity for *nearly linear* size spanners (or *sparsifiers*) is initiated in [LRR14, LRR16] for some restricted families of graphs such as minor-closed families. However, their focus is mainly on designing LCAs that *preserve* the connectivity; in the context of spanners, the stretch factor is allowed to be as large as n. Moreover, in their work, the input graph is sparse (has $O(n)$ edges), while the k-spanner problem is more challenging when the input graph is dense (has $\Omega(n)$ edges). Recently, Lenzen and Levi [LL18] designed the first LCA for sparsifiers in *general graphs*. In particular, their algorithm implies an LCA for spanners with $(1+\varepsilon)n$ edges, stretch $O(\log^2 n \cdot \text{poly}(\Delta/\varepsilon))$ and probe complexity of $O(\text{poly}(\Delta/\varepsilon) \cdot n^{2/3})$, where Δ is the maximum degree of the input graph.

In this work, we show that sublinear time LCAs for spanners are indeed possible in several cases. We give: (I) 3 and 5-spanners for general graphs with optimal trade-offs between the number of edges and the stretch parameter (up to polylogarithmic factors), and (II) general k-spanners, either in the dense regime (when the minimum degree is at least $n^{1/2-1/(2k)}$) or in the sparse regime (when the maximum degree is $n^{1/12-\varepsilon}$).

2.1.1 Our results and techniques

In this work we study the design of LCAs for graph spanners and in particular answer the following: *How can we decide quickly if a given edge e belongs to a sparse spanner (with fixed stretch) of the input graph, without preprocessing and storing any auxiliary information?* In the design of LCAs for graph problems, the set of defined *probes* to the input graph plays an important role. Here we consider the following common probes: NEIGHBOR probe ("what is the i^{th} neighbor of u"?), DEGREE probe ("what is $\deg(u)$?") and ADJACENCY probe ("are u and v neighbors"?) [Gol11, GGR98]. We emphasize that the answer to an ADJACENCY probe on an ordered pair $\langle u, v\rangle$ is the index of v in $\Gamma(u)$ if the edge exists and $\perp$ otherwise. Note that if the maximum degree in the input graph is $O(1)$, each ADJACENCY probe can be implemented by $O(1)$ number of NEIGHBOR probes.

Next, we discuss our approaches and results in more details; refer to Table 2.1 for a summary of the results.

Contribution (I): LCAs for 3 and 5-Spanners for General Graphs

Our first contribution is the local construction of 3 and 5-spanners for general graphs, while achieving the optimal trade-offs between the number of edges and the stretch factors (up to polylogarithmic factors).[2] In particular, our LCAs have $o(n)$ probe complexity even when the input graph is dense – unlike distributed local algorithms, or LCAs under the maximum degree assumption, our LCAs

[1] We may also measure the *time complexity* of an LCA. In our LCAs, the time complexities are clearly only a factor of $\text{poly}(\log n)$ higher than the corresponding probe complexities, so we focus our analysis on probe complexities.

[2] Indeed, the girth conjecture of Erdős is resolved for these stretch factors; see e.g., [Wen91].

	Reference	Graph Family	# Edges	Stretch	Probe Complexity
Prior works	[LRR14]	Bounded Degree Graphs	$(1+\varepsilon)n$	–	$\Omega(\sqrt{n})$
		Expanders	$(1+\varepsilon)n$	–	$O(\sqrt{n})$
		Subexponential growth	$(1+\varepsilon)n$	–	$O(\sqrt{n})$
	[LR15]	Minor-free	$(1+\varepsilon)n$	$\text{poly}(\Delta, 1/\varepsilon)$	$\text{poly}(\Delta, 1/\varepsilon)$
	[LRR16]	Minor-free	$(1+\varepsilon)n$	$O((\log\Delta)/\varepsilon)$	$\text{poly}(\Delta, 1/\varepsilon)$
	[LMR+17]	Expansion $(1/\log n)^{1+o(1)}$	$(1+\varepsilon)n$	super-exponential in $1/\varepsilon$	super-exponential in $1/\varepsilon$
	[LL18]	General	$(1+\varepsilon)n$	$O(\log^2 n \cdot \text{poly}(\Delta/\varepsilon))$	$O(n^{2/3} \cdot \text{poly}(\Delta/\varepsilon))$
Here	Thm. 2.1.1	General	$\widetilde{O}(n^{1+1/r})$	$2r-1$ $(r \in \{2,3\})$	$\widetilde{O}(n^{1-1/(2r)})$
	Thm. 2.2.14	Min degree $O(n^{1/2-1/(2k)})$	$\widetilde{O}(n^{1+1/k})$	5	$\widetilde{O}(n^{1-1/(2k)})$
	Thm. 2.1.2	Max degree $O(n^{1/12-\varepsilon})$	$\widetilde{O}(n^{1+1/k})$	$O(k^2)$	$\widetilde{O}(n^{1-4\epsilon})$
	Thm. 2.1.3	General	$o(m)$	any $k \leq n$	$\Omega(\min\{\sqrt{n}, n^2/m\})$

Table 2.1: Table of results on LCAs for the spanner problem. The symbol '$-$' indicates that the stretch is not analyzed. The input graph is a simple graph with n vertices, m edges, maximum degree Δ, and belongs to the indicated graph family. $\widetilde{O}$ hides a factor of $\text{poly}(\log n, k)$.

must operate on these graphs despite the inability to even learn the neighbor set of a high-degree vertex.

Most distributed spanner constructions are based on thinning the graph via clustering: construct a random set S of *centers* by adding each vertex to S independently with some fixed probability. For each vertex v sufficiently close to a center in S, include the edges of the shortest path connecting v to its closest member $s \in S$: this induces a *cluster* around each $s \in S$, where every pair of vertices in the same cluster are connected by a short path. Then, add sufficient edges connecting pairs of neighboring clusters to ensure the desired stretch factor. We design our LCAs around this standard clustering idea, but naïve attempts to construct small spanners under sub-linear probes only succeed under rather strong assumptions on the edges, such as very restricted range for their endpoints' degrees. We observe that we may partition the edge set into a constant number of classes, and separately "take care of" each class by constructing a stretch-k spanner for the subgraph formed by the edges of that respective class. The union of these constructed solutions is a k-spanner for the initial graph.

The common distributed construction of 3-spanners. The following algorithm constructs a 3-spanner $H \subseteq G$ with $\widetilde{O}(n^{3/2})$ edges. First, add to H all edges incident to vertices of degree at most $\sqrt{n}$. Second, pick a collection S of centers by sampling each vertex independently with probability $\Theta((\log n)/\sqrt{n})$. Each vertex v of degree at least $\sqrt{n}$ picks a *single* neighboring center $s \in S \cap \Gamma(v)$[3] (which exists w.h.p.) as its *center*, then adds (v, s) to H, forming a collection of $|S| = O(\sqrt{n})$ clusters (stars) around these centers. Lastly, every vertex u adds only one edge to each of its neighboring clusters. This results in a 3-spanner, because if an edge from any u to some vertex v of a neighboring cluster S is omitted, a different edge from u to w in cluster S would have been chosen, providing the path $\langle u, w, s, v\rangle$ of desired stretch 3 connecting u and v.

[3] $\Gamma(v)$ denotes the neighbor set of v, whereas $\Gamma^+(v) = \Gamma(v) \cup \{v\}$.

Overview of the LCA for 3-spanners. Here, the goal is to design an LCA that constructs a 3-spanner $H \subseteq G$ of size $\widetilde{O}(n^{3/2})$ and probe complexity of $\widetilde{O}(n^{3/4})$: the LCA is given an edge (u, v) and must answer whether $(u, v) \in E(H)$. The algorithm can straightforwardly handle the edges incident to the vertices of degree less than $\sqrt{n}$ by applying a DEGREE probe to each of its endpoint. The LCA model allows each vertex to consistently flip a coin and decide if it is a center. However, determining v's unique center, or testing its *membership* in some cluster, requires a linear scan of $\deg(v)$ probes. Lastly, as we try to connect u to the cluster of s, u must pick exactly one adjacent vertex in s's cluster; in other words, for each of $v \in \Gamma(u)$, we add (u, v) only if v belongs to a cluster not encountered by any other neighbor preceding v in u's neighbor-list. Overall, this entire process requires finding v's center s then testing, for up to $\deg(u)$ neighbors of u, its membership to the cluster of s: our total probe complexity could potentially be quadratic in the degree.

Multiple Centers.[4] To implement the cluster membership test more efficiently, we allow each vertex to join a slightly larger number of clusters. Instead of assigning each vertex v to *exactly* one of the centers in its neighbor list, assign v to the clusters of *all* centers that are among its first $\sqrt{n}$ neighbors in v's neighbor-list. W.h.p., if $\deg(v) \geq \sqrt{n}$ then it is assigned to $\Theta(\log n)$ clusters, thereby increasing the size of our spanner only by a factor of $O(\log n)$. This insight enables the algorithm to test cluster membership with a single ADJACENCY probe: the vertex v belongs to the cluster of s, if the index of s in v's neighbor-list is at most $\sqrt{n}$ (the index is returned by the ADJACENCY probe on v and s). This idea alone decreases the probe complexity of our LCA to $\widetilde{O}(\deg(u))$.

Neighbor Partitioning.[5] The multiple center technique above allows our LCA to handle edges adjacent to a vertex u of degree at most $n^{3/4}$. For $\deg(u) > n^{3/4}$, our LCA cannot even afford to look at all neighbors of u. To this end, we *partition* the neighbors of u into *blocks* of size $n^{3/4}$ each. Rather than adding one edge between u to each neighboring cluster, we now allow one edge between *each block* of u to each neighboring cluster: this increases the number of edges added from u by a factor of $\deg(u)/n^{3/4} \leq n^{1/4}$, but keeps the probe complexity down at $\widetilde{O}(n^{3/4})$ as we only need to scan the block containing v given the query (u, v) instead of u's entire neighbor-list. Now, to resolve the increase in the spanner size, we turn back to our previously-an-obstacle now-an-assumption $\deg(u) > n^{3/4}$: we only need $|S| = \Theta(n^{1/4} \log n)$ instead of $\widetilde{\Theta}(\sqrt{n})$ to ensure each such u has a neighboring center, reducing the spanner size back down by a factor of $\Theta(n^{1/4})$, as desired.

Overview of the LCA for 5-spanners. In 5-spanners, the desired bound for the LCA is a solution of size $\widetilde{O}(n^{4/3})$ and the probe complexity $\widetilde{O}(n^{5/6})$. We first observe that our 3-spanners suffice to handle of a certain subset of edges with the promised number of edges and probe complexity for LCA of 5-spanners. More specifically, we only need to design LCAs that handle edges whose both endpoints have degrees in $[n^{1/3}, n^{5/6}]$. Our previous 3-spanner construction adds a few edges between each neighboring *vertex-cluster* pair, but for 5-spanners we have two extra edges we can

[4] The construction of the spanner H_{high} that takes care of edges (u, v) where $n^{1/2} \leq \max\{\deg(u), \deg(v)\} \leq n^{3/4} = \Delta_{\text{super}}$ is illustrated in Fig. 2-2 (page 38); $\mathsf{MltCtrs}^+(v)$ denotes the set of multiple centers of v.

[5] The construction of the spanner H_{super} that takes care of edges (u, v) where $\max\{\deg(u), \deg(v)\} \geq \Delta_{\text{super}} = n^{3/4}$ is illustrated in Fig. 2-4 (page 40); each $\Gamma_{\Delta_{\text{super}},i}(u)$ denotes the i^{th} block of neighbors of size $\Theta(\Delta_{\text{super}})$ of u.

use to connect endpoints of omitted edges. Hence, we attempt to extend our 3-spanner construction in two ways: either add edges between each neighboring *cluster-cluster* pair, or stick with the vertex-cluster pairs but enlarge the clusters so they have *radius two* (no longer stars).

Cluster partitioning (bucketing).[6] The first approach adds an edge between every pair of neighboring clusters (stars). As $\deg(u), \deg(v) \geq n^{1/3}$, $|S| = \widetilde{O}(n^{2/3})$ clusters suffice to cover u and v, yielding a spanner of size $O(|S|^2) = \widetilde{O}(n^{4/3})$ as desired. Unfortunately this approach cannot be readily implemented with the desired probe complexity. To see why, consider clusters centered at s and t, containing u and v respectively. A naïve attempt spends $\deg(s) \cdot \deg(t)$ ADJACENCY probes for vertices between these clusters, as to consistently pick a unique edge between the two clusters. Instead, in lieu of the neighbor partitioning technique, we improve the probe complexity by partitioning the clusters into *buckets* of size $\Theta(n^{1/3})$, and so we may pick only one edge between any pair of buckets, yielding the desired $\Theta(n^{2/3})$ probe complexity for this step. By a careful analysis, we show that the number of created buckets is only $\widetilde{O}(n^{2/3})$, still yielding the desired spanner size $\widetilde{O}(n^{4/3})$. However, unlike partitioning a *neighbor-list*, partitioning a *cluster* requires full knowledge of the members of the cluster – they are not nicely indexed in a list any more. To this end, we only pick centers of degree $\deg(s), \deg(t) \leq n^{5/6}$ so that identifying the member of their clusters takes $\widetilde{O}(n^{5/6})$ probes (via the multiple center technique). However, these clusters are only guaranteed to cover vertices that have many neighbors with degree less than $n^{5/6}$. We handle the complimentary case in the following second approach.

Representatives.[7] We now turn to vertices of degrees $[n^{1/3}, n^{5/6}]$ that have many neighbors of degree at least $n^{5/6}$. As discussed earlier, we will increase the radius of clusters to two: these vertices will be at distance two from their centers, whereas the inner stars of these clusters are simply the previously-constructed 3-spanner. Recall that the clustering approach handle vertices of high degree (now with threshold $n^{2/3}$ in our construction): all high-degree vertices join some clusters: they are adjacent to the centers, and now forming the first-level of our radius-2 clusters. Thus, to choose which cluster to join (in the second-level), our vertex, which has many high-degree neighbors, simply chooses and connects itself to one or more high-degree neighbors, called its *representatives*. To find representatives of a vertex v, we simply pick $\Theta(\log n)$ random neighbors of v, and w.h.p. one of them will have high-degree, and hence is chosen as v's representative.

Our LCA picks a set of $|S| = \widetilde{O}(n^{1/6})$ centers for the 3-spanner. For each edge (u, v) where $\deg(u), \deg(v) \geq n^{1/3}$ and v has many high-degree neighbors (with degree at least $n^{5/6}$), v has $\log n$ representatives, and each representative has $\Theta(\log n)$ centers w.h.p., so v belongs to $O(\log^2 n)$ clusters. As per the 3-spanner case, we keep (u, v) if v is in a cluster that no previous neighbors of u belong to; this gives the spanner size of $n \cdot |S| = \widetilde{O}(n^{7/6})$. We may find the representatives of each

[6] The construction of the spanner H_{bckt} that takes care of edges (u, v) where $\deg(u), \deg(v) \in [\Delta_{\text{med}}, \Delta_{\text{super}}] = [n^{1/3}, n^{5/6}]$, such that the majority of the first $n^{1/3}$ neighbors of *both* u *and* v are of degree *at most* $n^{5/6}$, is illustrated in Fig. 2-6 (page 43); $\mathsf{Clst}(s)$ denotes the cluster of vertices that have s among their multiple centers, and $\mathsf{Bucket}(u, s)$ denotes the bucket in $\mathsf{Clst}(s)$ containing u.

[7] The construction of the spanner H_{rep} that, together with H_{super} with parameter $\Delta_{\text{super}} = n^{5/6}$, takes care of edges (u, v) where $\deg(u), \deg(v) \in [n^{1/3}, n^{5/6}]$, such that the majority of the first $n^{1/3}$ neighbors of u *or* v are of degree *at least* $n^{5/6}$, is illustrated in Fig. 2-7 (page 44); $\mathsf{Reps}(v)$ denotes the representatives of v, and $\mathsf{RepsCtrs}^+(v)$ denotes the set of v's representatives' multiple centers.

neighbor of u in $O(\log n)$ probes, and for all these $\deg(u) \cdot O(\log n) = \widetilde{O}(n^{5/6})$ representatives, check if they belong to any of v's $O(\log^2 n)$ clusters with $\widetilde{O}(n^{5/6})$ membership tests. The probe complexity is also equally contributed to by the process of finding v's representatives' centers, taking $O(n^{5/6})$ probes per representative, which is $\widetilde{O}(n^{5/6})$ probes in total.

Theorem 2.1.1 (3 and 5-spanners). *For every n-vertex graph G there exists an LCA for $(2r-1)$-spanners with $\widetilde{O}(n^{1+1/r})$ edges and probe complexity $\widetilde{O}(n^{1-1/(2r)})$ for $r \in \{2,3\}$. Moreover, the algorithm only uses $O(\log^2 n)$ random bits.*

In fact, if G has *minimum* degree $\omega(n^{1/3})$, we may apply the 5-spanner construction (with modified parameters) to obtain 5-spanners with even lower number of edges as indicated in Table 2.1 (Theorem 2.2.14): this minimum degree assumption indeed allows even sparser spanners, bypassing the girth conjecture that holds for *general* graphs. We also remark that, in the somewhat related setting of dynamic computation, spanner algorithms with worst-case *sublinear* update time are currently known only for 3 and 5-spanners as well [BK16].

Contribution (II): LCA for $O(k^2)$-Spanners for Graphs with Maximum Degree Δ

Our second contribution is the local construction of $O(k^2)$-spanners with $O(n^{1+1/k})$ edges for any $k \geq 1$, which has sub-linear probe complexity for graphs of maximum degree $\Delta = O(n^{1/12-\varepsilon})$.

Recall that there are *distributed* algorithms for constructing $(2k-1)$-spanners in k rounds. In the standard reduction from distributed algorithms to LCAs, given the query (u,v), if we visit all vertices within distance k of u and v, we will be able to simulate the distributed algorithm and answer accordingly. However, in the case $\Delta = \Omega(n^{1/k+\varepsilon})$, scanning the entire k-neighborhood may lead to the probe complexity of $\Omega(\min\{\Delta^k, \Delta n\}) = \Omega(n)$, rendering this approach prohibitively costly. Our approach instead builds on the recent work of Lenzen and Levi [LL18] – their work aims at locally constructing a spanning subgraph with only $O(n)$ edges, but the stretch parameter of their subgraph can generally be as large as $O(\text{poly}(\Delta)\log^2 n)$. Incorporating our new techniques, we design our LCA with the following guarantee:

Theorem 2.1.2 (Spanners for Graphs with Maximum Degree Δ). *For every integer $k \geq 1$ and every n-vertex graph G with maximum degree Δ, there exists an LCA for $O(k^2)$-spanner with $\widetilde{O}(n^{1+1/k})$ edges and probe complexity $\widetilde{O}(\Delta^4 n^{2/3})$. In particular, for $\Delta = O(n^{1/12-\varepsilon})$, the probe complexity is $\widetilde{O}(n^{1-4\varepsilon})$. Moreover, the algorithm only uses $O(\log^2 n)$ random bits.*

Our LCA extends the construction of [LL18] in two important aspects. Firstly, we reduce the stretch parameter of the constructed subgraph to $O(k^2)$, independent of both n and Δ, while using only $\widetilde{O}(n^{1+1/k})$ edges. In particular, the algorithm achieves a sub-linear probe complexity for $\Delta = O(n^{1/12-\varepsilon})$. Secondly, our construction can be implemented using only $\text{poly}(\log n)$ independent random bits (where [LL18] uses $\text{poly}(n)$ bits). Unlike previous works in LCAs that generally bound the required randomness trivially from the fact that their LCAs only make $\text{poly}(\log n)$ probes and hence only require $\text{poly}(\log n)$-independent random bits, our probe complexity depends polynomially on n. Thus, our LCA requires a specialized analysis that bounds the required independence based on its behavior.

We now summarize the key techniques in the design of our LCA as follows.

Sparse edges: simulating distributed algorithms. For a given stretch parameter k, we partition the edges in G into the *sparse* set E_{sparse} and the *dense* set E_{dense}. Roughly speaking, the sparse set E_{dense} only consists of edges (u, v) for which the k-neighborhood in G of either u or v contains at most $O(n^{2/3})$ vertices.[8] For this sparse region in the graph, we can simulate a standard distributed algorithm for spanners, with small probe complexity. The reason is that distributed algorithms for $(2k-1)$-spanners use k rounds of communication and hence depend only on the k^{th}-neighborhood of the vertices and the random bits. If the query edge (u, v) is in this sparse region, the LCA can probe for the entire k^{th}-neighborhood of one of its endpoints u, v by making $O(\Delta \cdot n^{2/3})$ probes (as oppose to the naïve $n^{O(k)}$ bound when the k^{th}-neighborhood is not sparse) and then simulate the distributed algorithm in this neighborhood. For this purpose, we will employ a version of Baswana-Sen that uses only a poly-logarithmic number of random bits [BS07, CPS17]. This simulation yields an LCA handling the sparse edges with $O(\Delta^2 n^{2/3})$ probe complexity. Hence, the overall probe complexity of our LCA is strictly dominated by that of the dense edges case described below.

Dense edges: partitioning of dense vertices into low-diameter Voronoi cells. To take care of the dense edges, we sample[9] a collection S of $O(n^{2/3} \log n)$ centers and partition the (dense) vertices into *Voronoi cells* around these centers.[10] This is done by connecting each dense vertex via a lexicographically-first shortest path to its closest sampled center. Our LCA can identify the Voronoi center, together with this path, of any given vertex using $O(\Delta \cdot n^{1/3})$ probes via a breadth-first search algorithm. These shortest path edges connect vertices of the same Voronoi cell, forming a tree structure of depth k (by the definition of dense vertices), ensuring a diameter of at most $2k$ (whereas in [LL18], the diameter of the Voronoi cells is $O(\Delta \log n)$).

Attempting to connect these Voronoi cells together incurs two problems. The first problem is that, we still cannot afford to add an edge between every adjacent pairs of $\widetilde{O}(n^{2/3})$ Voronoi cells. To resolve this issue, we again resort to the common distributed construction of 3-spanners (page 25): for an N-vertex graph, *mark* $\widetilde{O}(\sqrt{N})$ vertices as centers, form clusters around them, then connect them into a 3-spanner using $\widetilde{O}(N^{3/2})$ edges – further details will be discussed momentarily. Since there are $N = \widetilde{O}(n^{2/3})$ Voronoi cells, this approach only requires $\widetilde{O}(N^{3/2}) = \widetilde{O}(n)$ edges to connect our Voronoi cells. If we could actually implement such an approach, we would then obtain a 3-spanner over the Voronoi cells, or a $3 \cdot (2k) = O(k)$-spanner over the dense vertices – a result better than the $O(k^2)$ stretch that we originally claim.

Refining Voronoi cells into small clusters. We have a more demanding second problem: it is simply impossible to implement the 3-spanner construction above. Observe that a Voronoi cell may contain as many as $\Theta(n)$ vertices. In particular, given a vertex v, we cannot enumerate all

[8] An edge is *dense* if both of its endpoints are *dense vertices* that contain some *center* in their k-neighborhoods; otherwise it is *sparse*. The set $S \subset V$ of $|S| = O(n^{2/3} \log n)$ centers is chosen randomly, implying the above condition via the hitting set argument.

[9] Each vertex $v \in V$ locally decides that $v \in S$ with probability $\Theta(n^{-1/3} \log n)$, yielding a set S of size $\Theta(n^{2/3} \log n)$ in expectation.

[10] The Voronoi partitioning with random centers has also been used in other related contexts; see e.g., [MZ13].

vertices in a Voronoi cell containing v, let alone finding all of its adjacent Voronoi cells. To this end, we further subdivide the Voronoi cells into *clusters* of a more tractable size of $O(n^{1/3})$ vertices each. (In [LL18], the cluster size is linear in Δ: we introduce a different construction and analysis to remove this dependence on Δ.) We show that our LCA can identify visit its *entire* cluster using $O(\Delta^3 n^{2/3})$ probes. An essential property of this refinement is that the number of clusters (resp., clusters in marked Voronoi cells) does not significantly increase from that of Voronoi cells (resp., marked Voronoi cells), so rules (1) and (2) can still be applied, in the "cluster level" instead of the "Voronoi cell level", without significantly increasing the number of added edges. The problem, still, is that from a vertex v, we cannot find *all* Voronoi cells adjacent to v's *Voronoi cell* – but now, we can at least find *some* Voronoi cells: those adjacent to v's *cluster*. We now discuss these ideas in more details.[11] In particular, we show that the same 3-spanner connection rules, when executed on this incomplete neighborhood information, still offers the connectivity guarantee, but sacrifices the stretch factor guarantee.

Connecting Voronoi cells. We hope connect our Voronoi cells in the same fashion as the 3-spanner construction, on the *supergraph* where each Voronoi cell is contracted into a single vertex. We *mark* a random subset of $O(n^{1/3} \log n)$ Voronoi cells (among the $n^{2/3}$ Voronoi cells), then connect them according to the following rules using $\widetilde{O}(n)$ edges each. Rule (1): We connect every pair of marked Voronoi cells to their neighboring Voronoi cells. Rule (2): Let us call a Voronoi cell *bad* if it has no neighboring marked Voronoi cells – bad Voronoi cells are likely to have only $\widetilde{O}(n^{1/3})$ neighboring Voronoi cells, so we can connect them to all their neighboring Voronoi cells as well. Rule (3): We handle the remaining edges, namely the incident edges of *good* Voronoi cells b having some neighboring marked Voronoi cell c, as follows. For each pair of (not necessarily adjacent) Voronoi cell a and marked Voronoi cell c sharing common neighboring Voronoi cells $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$, we keep an edge from a to a single Voronoi cell $\mathsf{b}^* \in \Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$: for any good b where (a, b) is omitted, b will be connected to a in our spanner via $\langle \mathsf{b}, \mathsf{c}, \mathsf{b}^*, \mathsf{a} \rangle$. (where incident edges of c are added because c is marked).

Suppose that we have access to this supergraph, and let the query edge (u, v) be between Voronoi cells a and b. Locally choosing the edges for rules (1) and (2) only requires checking $\Gamma(\mathsf{a})$ and $\Gamma(\mathsf{b})$. For rule (3), we keep the edge connecting a to b if there exists some marked $\mathsf{c} \in \Gamma(\mathsf{b})$ such that the ID of b (defined as $\mathsf{ID}(v)$ of its center v) is the minimum among $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$. Unfortunately there is a major problem with this approach: we do not actually see this supergraph of Voronoi cells. For instance, from v in b, we can only see the cluster containing b, and then find out the neighboring Voronoi cells of this cluster: we do not see the entire $\Gamma(\mathsf{b})$.

Connectivity via partial neighborhood information. We first show that there are $\widetilde{O}(n^{2/3})$ clusters, $\widetilde{O}(n^{1/3})$ of which belong to marked clusters: since these quantities do not differ much from their Voronoi cell counterparts, we can apply the first two rules analogously. Next, we observe that applying rule (3) using only partial neighborhood information still yields a valid spanner. Suppose that we omit the query edge (u, v) (between Voronoi cells a and b) because we discover a marked $\mathsf{c} \in \Gamma(\mathsf{b})$ and some $\mathsf{b}' \in \Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$ with $\mathsf{ID}(\mathsf{b}') < \mathsf{ID}(\mathsf{b})$: b' has the minimum ID among the Voronoi

[11] The illustrated example on page 58, accompanied by Figure 2-12, may be helpful for understanding the upcoming description.

cells that the LCA actually sees in $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$. Observe that the path $\langle \mathsf{b}, \mathsf{c}, \mathsf{b}' \rangle$ is still in the spanner, and there is some edge (u', v') between Voronoi cells a and b', but we do not know if (u', v') will be kept by the LCA (because the LCA with query (u', v') may not see c from b'). At this point, we must instead prove connectivity between a and b' (as opposed to a and b, our original task for query (u, v)). Since $\mathsf{ID}(\mathsf{b}') < \mathsf{ID}(\mathsf{b})$, we can inductively repeat this argument with decreasing ID at each step: eventually we will reach some $\mathsf{b}^\star \in \Gamma(\mathsf{a})$ with sufficiently low ID that an edge between a and $\mathsf{b}^\star$ is kept by the LCA.

Establishing the stretch guarantee. Recall that vertices within each Voronoi cell are connected by a diameter-$2k$ tree structure, so it suffices to show that the inductive argument above terminates in $O(k)$ steps; that is, the spanner path from Voronoi cell supervertices a to b only visits $O(k)$ other Voronoi cells. To this end, we make the following modication to rule (3): instead of comparing the IDs of Voronoi cells through $\mathsf{ID}(v)$ of centers, we use an independent random *rank* assignment $r(v)$ of centers. The work of [LL18] observes that during each inductive step, conditioned on $r(\mathsf{b}') < r(\mathsf{b})$ we have $r(\mathsf{b}') \leq r(\mathsf{b})/2$ with probability $1/2$, so the argument terminates in expected $O(\log n)$ steps, yielding a path on the supergraph of length $O(\log n)$.

We enhance this idea further by modifying rule (3), thereby adding an edge from a to b if there exists a marked Voronoi cell c such that the rank $r(\mathsf{b})$ of b is among the $O(n^{1/k} \log n)$ lowest ranks in $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$ restricted to those discovered by the LCA. Via a similar argument, we have $r(\mathsf{b}') < n^{-1/k} \cdot r(\mathsf{b})$ w.h.p., and so the inductive argument only requires $O(k)$ steps, yielding the overall stretch factor of $O(k^2)$.

Implementation with bounded independence using $O(\log^2 n)$ random bits. We relax the assumption that the rank assignment for the Voronoi cells is chosen with full independence, and instead show that in order to guarantee the termination of the inductive process within $O(k)$ steps, it suffices to work with only $T = \Theta(k)$ hash functions $h_1, \cdots, h_T$ chosen uniformly at random form a family of $O(\log n)$-wise independent hash functions of the form $\{0,1\}^{\log n} \to \{0,1\}^{O((\log n)/k)}$. We define our rank function as a concatenation of h_i's on the ID of the Voronoi cell's center: for the Voronoi cell centered at v, its rank is given by $r(v) = h_1(\mathsf{ID}(v)) \circ \ldots \circ h_T(\mathsf{ID}(v))$. From the earlier example, let $\mathsf{Vor}_0 = \mathsf{b}, \mathsf{Vor}_1 = \mathsf{b}', \mathsf{Vor}_2, \ldots$ denote the sequence of Voronoi cells in $\Gamma(\mathsf{a})$ we consider in each inductive step. We then show in our analysis that as we reach the Voronoi cell Vor_i, w.h.p. the rank of its center is 0 in all its first i hash functions (i.e., $h_1 = \cdots = h_i = 0$ on Vor_i's center's ID). This in particular implies that the sequence of ranks of $\mathsf{Vor}_0, \mathsf{Vor}_1, \ldots$ may decrease at most T times before the inductive process inevitably terminates, establishing the desired $O(k^2)$ stretch bound w.h.p. even when the full independence of rank assignment is replaced with $O(\log n)$-wise independence. Hence our LCA only requires $O(\log^2 n)$ independent random bits, as desired.

Contribution (III): Lower Bounds. To establish the lower bound, we construct two distributions over undirected d-regular graph instances that contain a designated edge e. For graphs in the first family, it holds that after removing e, w.h.p., they remain connected while in the second family, removing e disconnects the endpoint of e and leave them in separate connected components. We show that for the edge e, any LCA that makes $o(\min\{\sqrt{n}, n/d\}) = o(\min\{\sqrt{n}, n^2/m\})$ probes can only distinguish whether the underlying graph is from the first family or the second family with

probability $1/2 + o(1)$.

Our approach mainly follows from the analysis of [KKR04] on the construction of [LRR14]. The lower bound of [LRR14] shows $\Omega(\sqrt{n})$ for the probe complexity of LCA for spanning graphs that only uses NEIGHBOR probes. However, in our setting, ADJACENCY probes are allowed as well. To handle the additional probe, we follow [KKR04] to show that if the number of probes performed by an algorithm is $o(m^2/nd)$, w.h.p. the answer to all ADJACENCY probes are no. Although our ADJACENCY probes return more information compared to [KKR04] if the edge exists, since the answer to all of them are no, the same argument follows.

Theorem 2.1.3 (Lower Bound). *Any local randomized LCA that computes, with success probability at least 2/3, a spanner of the simple m-edge input graph G with $o(m)$ edges, has probe complexity $\Omega(\min\{\sqrt{n}, n^2/m\})$.*

2.1.2 Additional related work: spanners in many other related settings

Local distributed algorithms. The construction of spanners in the distributed local model, where messages are unbounded, has been studied extensively in both the randomized and the deterministic settings [BS07, EN17, DG08, DGP07, DGPV08, DGPV09, Pet10]: the state of the art of both randomized and deterministic constructions is $O(k)$ rounds.

Dynamic algorithms for graph spanners. In the dynamic setting, the input graph G undergoes *updates* (e.g., edge insertions and deletions) and one wants to avoid recomputing the spanner from scratch after every update. In this model, the challenge is to dynamically maintain a spanner under the edge insertion/deletion with only a small amount of time required per update. Most of the dynamic algorithms for spanners maintain an auxiliary clustering structure that aids the modification of current spanner. Recently in [BK16], the first fully dynamic algorithms with sublinear worst-case bounds have been obtained *only* for 3-spanners and 5-spanners. We hope that the tools developed here will be useful for the dynamic setting as well.

Streaming algorithms. In the setting of dynamic streaming, the input graph is presented online as a long stream of insertions and deletions to its edges. For spanners, the goal is to maintain a sparse spanner for the graph using small space and few passes over the stream. Ahn, Guha and McGregor [AGM12] showed the first a sketch-based algorithm for spanners in this setting, yielding $(k^{\log_2 5} - 1)$-spanner with $\widetilde{O}(n^{1+1/k})$ edges and $O(\log k)$ passes. Kapralov and Woodruff [KW14] showed an alternative tradeoff yielding $O(2^k)$-spanner with $\widetilde{O}(n^{1+1/k})$ edges using only *two* passes. In dynamic streaming one can keep the entire solution and the challenge is to update the solution though the pass over the stream. In contrast, in the LCA model, one cannot afford keeping the entire solution (i.e., already the number of vertices is too large) but the input graph remains as is.

2.1.3 Model Definition and Preliminaries

Graph notation. Throughout, we consider simple unweighted undirected graphs $G = (V, E)$ on $n = |V|$ vertices and $m = |E|$ edges. Each vertex v is labeled by a unique $O(\log n)$-bit value

$\mathsf{ID}(v)$[12]. For $u \in V$, let $\Gamma(u, G) = \{v : (u, v) \in E\}$ be the neighbors of u, $\deg(u, G) = |\Gamma(u, G)|$ be its degree, and define $\Gamma^+(u, G) = \Gamma(u, G) \cup \{u\}$. Denote $V_I = \{v \in V : \deg(u, G) \in I\}$ where I is an interval. For $u, v \in V$, let $\mathrm{dist}(u, v, G)$ be the shortest-path distance between u and v in G. Let $\Gamma^k(u, G) = \{v : \mathrm{dist}(u, v, G) \leq k\}$ be the k^{th}-neighborhood of u, and denote its size $\deg_k(u, G) = |\Gamma^k(u, G)|$. A graph H is a subgraph of G, denoted $H \subseteq G$, if its edge set $E(H) \subseteq E(G)$. The union of two subgraphs H, H' of G is given by $H \cup H' = (V, E(H) \cup E(H'))$. The parameter G may, in general, be omitted for the input graph.

We assume that the input graph comes with its adjacency list representation: each neighbor set has a fixed ordering, $\Gamma(u) = \{v'_1, \ldots, v'_{\deg(u)}\}$; this ordering may be arbitrarily (e.g., not necessarily sorted by vertex IDs). Many of the algorithms in this work are based on partitioning the neighbor-list into balanced-size blocks. For $\Delta \in [n]$ and $u \in V$ such that $\deg(u) \geq \Delta$, let $\Gamma_{\Delta,1}(u), \ldots, \Gamma_{\Delta,\Theta(\deg(u)/\Delta)}(u)$ be blocks of neighbors obtained by partitioning $\Gamma(u)$ into consecutive parts. Each block is of size Δ, except possibly for the last block that is allowed to contain up to 2Δ vertices.

Local Computation Algorithms. We adopt the definition of LCAs by Rubinfeld et al. [RTVX11]. A local algorithm have access to the *adjacency list oracle* $\mathcal{O}^G$ which provides answers to the following probes (in a single step):

- **NEIGHBOR probes**: Given a vertex $v \in V$ and an index i, the i^{th} neighbor of v is returned if $i \leq \deg(v)$. Otherwise, $\perp$ is returned. The orderings of neighbor sets are fixed in advance, but can be arbitrary.
- **DEGREE probes**: Given a vertex $v \in V$, return $\deg(v)$. This probe type is defined for convenience, and can alternatively be implemented via a binary search using $O(\log n)$ NEIGHBOR probes.
- **ADJACENCY probes**: Given an ordered pair $\langle u, v \rangle$, if $v \in \Gamma(u)$ then the index i such that v is the i^{th} neighbor of u.[13] Otherwise, $\perp$ is returned.

Definition 2.1.4 (LCA for Graph Spanners)**.** *An* LCA $\mathcal{A}$ *for graph* spanners *is a (randomized) algorithm with the following properties. $\mathcal{A}$ has access to the adjacency list oracle $\mathcal{O}^G$ of the input graph G, a tape of random bits, and local read-write computation memory. When given an input (query) edge $(u, v) \in E$, $\mathcal{A}$ accesses $\mathcal{O}^G$ by making probes, then returns* yes *if (u, v) is in the spanner H, or returns* no *otherwise. This answer must only depend on the query (u, v), the graph G, and the random bits. For a fixed tape of random bits, the answers given by $\mathcal{A}$ to all possible edge queries, must be consistent with one particular sparse spanner.*

The main complexity measures of the LCA for graph spanners are the size and stretch of the output spanner, as well as the probe complexity of the LCA, defined as the maximum number of probes that the algorithm makes on $\mathcal{O}^G$ to return an answer for a single input edge. Informally speaking, imagine m instances of the same LCA, each of which is given an edge of G as a query,

[12] We do *not* require IDs to be a bijection $V \to [n]$ as in most other works on LCAs.

[13] We remark that the traditional LCA model (e.g., [RTVX11]) does not consider this type of probe, yet it is common in the oracle access model, especially for dense graphs [KKR04]. We also distinguish our ADJACENCY probes from the VERTEX-PAIR probes in Chapter 5 that only returns 1 or 0 indicating whether $(u, v) \in E(G)$ or not, without the extra index information.

while the *shared* random tape is broadcast to all. Each instance decides if its query edge is in the subgraph by making probes to $\mathcal{O}^G$ and inspecting the random tape, but may not communicate with one another by any means. The LCA succeeds for the input graph G and the random tape if the collectively-constructed subgraph is a desired spanner. All the algorithms in this chapter are randomized and, for any input graph, succeed with high probability $1 - 1/n^c$ over the random tape.

Chapter Organization. We start in Section 2.2 by describing our results for 3 and 5 spanners in general graphs. Next, in Section 2.3 by extending the recent construction of [LL18] to provide $O(k^2)$-spanners for graphs with maximum degree $n^{1/12}$. For simplicity, we first describe all our randomized algorithms as using full independence, then in Section 2.4, we explain how these algorithms can be implemented using a seed of poly-logarithmic number of random bits). Finally, in Section 2.5, we provide a lower bound result for a simpler task of computing a spanning subgraph with the specified probes.

Clarification. Throughout we use the term "spanner construction" when describing how to construct our spanners. These construction algorithms are used only to define the unique spanner, based on which the LCA makes its decisions: we never construct the full, global spanner at any point.

2.2 LCA for 3 and 5-Spanners

2.2.1 3-spanners

In this section, we give an LCA which constructs 3-spanners with $\widetilde{O}(n^{3/2})$ edges using probe complexity $\widetilde{O}(n^{3/4})$. In Section 2.2.1.1 we begin by establishing some observations that allow us to "take care" of different types of edge separately based on the degrees of their endpoints. We turn to the standard approach for spanner construction in Section 2.2.1.2, where we highlights the challenges for its LCA implementation. We then introduce our key techniques, *multiple centers* and *neighbor partitioning* respectively in Sections 2.2.1.3 and 2.2.1.4, to handle these issues, and finally establish our results in Section 2.2.1.5.

2.2.1.1 Overview and edge categorization

Taking care of edges. Consider the simple undirected unweighted input graph $G = (V, E)$. To construct our LCA for 3-spanners, we classify each edge in E into one or more categories. For each category, we propose an LCA tailored to efficiently takes care of edges of that category. More formally:

Definition 2.2.1 (Subgraphs taking care of edges). *For stretch parameter $k \geq 1$ and set of edges $E' \subseteq E$, we say that the subgraph $H' \subseteq G$* takes care *of E' if for every $(u, v) \in E'$, $\text{dist}(u, v, H') \leq k$.*

Observe that if we have a collection of subgraphs H_i's such that every edge in $(u, v) \in E$ is taken care by at least one H_i, then the union H of the H_i's constitute a k-spanner for G: if (u, v) is omitted, one can still traverse from u to v on H in k steps using edges of the H_i that takes care of (u, v). This observation suggests a method for constructing the overall LCA as follows.

Observation 2.2.2 (Spanner construction by combining subgraphs). *For a collection of subsets $E_1, \ldots, E_\ell \subseteq E$ where $\cup_{i\in[\ell]} E_i = E$, if H_i is a subgraph of G that takes care of E_i, then $H = \cup_{i\in[\ell]} H_i$ is a k-spanner of G. Further, if we have an LCA $\mathcal{A}_i$ for computing each H_i (i.e., deciding whether the query edge $(u, v) \in H_i$ and reporting* yes *or* no *accordingly), we may construct a final LCA that runs every $\mathcal{A}_i$ and answer* yes *precisely when at least one of them does so. The performance of our overall LCA (number of edges, probes, or random bits) can then be bounded by the respective sum over that of $\mathcal{A}_i$'s.*

Note that H_i may contain edges of E that are not in E_i, thus it is necessary that the overall LCA invokes every $\mathcal{A}_i$ even if $\mathcal{A}_i$ does not take care of the query edge.

Overall LCA for 3-spanners. Our LCA for 3-spanner assigns each edge of E into one or more of the subsets E_{low}, E_{high}, or E_{super} based on the degrees of its endpoints, so that $E_{\text{low}} \cup E_{\text{high}} \cup E_{\text{super}} = E$. Fix a parameter $r \geq 1$, and for ease of presentation, let $\Delta_{\text{low}} = n^{1/r}$, $\Delta_{\text{high}} = n^{1-1/r}$ and $\Delta_{\text{super}} = n^{1-1/(2r)}$. As given in the following table, we assign each $(u, v) \in E$ into some E_i, $i \in \{\text{low}, \text{high}, \text{super}\}$, based on the range of the quantity $\max\{\deg(u), \deg(v)\}$. Included also are the sizes of the created subgraphs H_i and the probe complexities of respective LCAs.

Subset	$\max\{\deg(u), \deg(v)\}$	# Edges	Probe Complexity
E_{low}	$[1, \Delta_{\text{low}}]$	$O(n \cdot \Delta_{\text{low}}) = O(n^{1+\frac{1}{r}})$	$O(1)$
E_{high}	$[\Delta_{\text{high}}, \Delta_{\text{super}}]$	$O(\frac{n^2 \log n}{\Delta_{\text{high}}}) = O(n^{1+\frac{1}{r}} \log n)$	$O(\Delta_{\text{super}} \log n) = O(n^{1-\frac{1}{2r}} \log n)$
E_{super}	$[\Delta_{\text{super}}, n)$	$O(\frac{n^3 \log n}{\Delta_{\text{super}}^2}) = O(n^{1+\frac{1}{r}} \log n)$	$O(\Delta_{\text{super}} \log n) = O(n^{1-\frac{1}{2r}} \log n)$

Table 2.2: Edge categorization for the construction of 3-spanners.

In the construction of 3-spanners for general graphs, we choose $r = 2$ so that $\Delta_{\text{low}} = \Delta_{\text{high}} = n^{1/2}$ and $\Delta_{\text{super}} = n^{3/4}$. Since $\Delta_{\text{low}} = \Delta_{\text{high}}$, we cover the entire range of the quantity $\max\{\deg(u), \deg(v)\}$, and thus have taken care of all edges. Hence, our construction leads to an LCA for 3-spanner of size $\widetilde{O}(n^{3/2})$ using $\widetilde{O}(n^{3/4})$ probes as desired. Nonetheless, observe that H_{high} and H_{super} have less edges as r becomes larger. This entails LCAs with better spanner sizes for graphs where $\max\{\deg(u), \deg(v)\} \notin (\Delta_{\text{low}}, \Delta_{\text{high}})$ holds for every edge, such as dense graphs with minimum degree Δ_{high}.

LCA for H_{low}: the trivial case. Because vertices of degree at most Δ_{low} have $O(n \cdot \Delta_{\text{low}}) = O(n^{1+1/r})$ incident edges in total, we may afford to keep all these edges, letting $H_{\text{low}} = (G, E_{\text{low}})$. Thus, an LCA simply needs to check the degrees of both endpoints (via DEGREE probes), and answer yes precisely when both (or in fact, even one) have degrees at most Δ_{low}.

2.2.1.2 The standard clustering approach and obstacles for its LCA adaptation

Our LCA is based on an application of our new technique called the *multiple-centers* method, motivated by the clustering approach used for graph thinning, explained momentarily. However we first remark that, rather than "thinning" the input graph, we construct spanners in an incremental

fashion. Beginning with an empty graph, we "add" edges from the input graph to our maintained subgraph, so that this subgraph eventually becomes a spanner. Under this notion, we may conveniently bound the size of our spanner by counting the added edges, and prove that the constructed subgraph is a valid k-spanner by showing that for every discarded edge, there exists a path formed entirely by the added edges, of length at most k that connects the two endpoints.

We are now ready to describe the global algorithm. Let H_{std} be the subgraph constructed via this "standard" approach, aiming to take care of $E(V, V_{[\Delta,n)})$ for some threshold Δ; this edge set includes E_{high} (resp., E_{super}) for $\Delta = \Delta_{\text{high}}$ (resp., Δ_{super}). We first choose a random set of vertices S by adding each vertex $v \in V$ to S independently with probability $p = \Theta((\log n)/\Delta)$; the vertices added to S are called *centers*. It follows by the Chernoff bound that w.h.p., $|S| = \Theta(n/\Delta \cdot \log n)$, and each vertex of $V_{[\Delta,n)}$ has at least one neighboring center; that is, S forms a *hitting set* for the collection $\{\Gamma(v)\}_{v \in V_{[\Delta,n)}}$.

Following the standard construction of 3-spanners (e.g., [BS07]), each vertex $v \in V_{[\Delta,n)}$ chooses a *single arbitrary* neighboring center, denoted $\mathsf{Ctr}(v)$. Each edge $(v, \mathsf{Ctr}(v))$ is added to H_{std} to form *clusters* in shape of (possibly overlapping) stars, spending up to n total edges. Let $\mathsf{Clst}(s)$ be the set of vertices in the cluster centered at s, consisting of s and every vertex that chooses s as its center. Now to connect between clusters, for each pair of vertex u and center s such that there exists an edge in E from u to some vertex in $\mathsf{Clst}(s)$, add one such arbitrary edge to H_{std}: keeping a single edge is sufficient to make u and $\mathsf{Clst}(s)$ remain connected, thinning down the number of edges from potentially n^2 to $n \cdot |S|$. We summarize the process via the following two rules for adding edges inside and between clusters. It is straightforward to verify that all added edges indeed exist in the original edge set E, so H_{std} is a subgraph of G.

Global construction of H_{std}. Each $v \in V$ is added to S with probability $p = \Theta((\log n)/\Delta)$.
(I) for each $v \in V_{[\Delta,n)}$, add $(v, \mathsf{Ctr}(v))$ to H_{std}
(B) for each $u \in V$ and $s \in S$, choose an edge from u to a vertex in $\mathsf{Clst}(s)$ (if one exists) and add it to H_{std}.

Figure 2-1: Procedure for the global construction of H_{std}.

To see why H_{std} takes care of $(u, v) \in E(V, V_{[\Delta,n)})$, suppose that (u, v) is omitted by this process. Let $s = \mathsf{Ctr}(v)$, then in step (B), some edge (u, w) where $w \in \mathsf{Clst}(s)$ must have been added to H_{std}. As edges (v, s) and (w, s) are added in step (I), u is connected to v via the path $\langle u, w, s, v\rangle$ of length 3 (unless $w = s$, where we have the path $\langle u, w = s, v\rangle$ of length 2 instead). Thus, H_{std} takes care of $(V, V_{[\Delta,n)})$.

Computing centers in the LCA model. In the LCA model, we do not generate the entire set S up front. Instead, we may verify whether $v \in S$ on-the-fly using v's ID by, e.g., applying a random map (chosen according to the given random tape) from v's ID to $\{0, 1\}$ with expectation p. In fact, this hitting set argument does not require full independence – the discussion on reducing the amount of random bits is given in Section 2.4, but for now we formalize it as the following observation.

Observation 2.2.3 (Local Computation of Centers). *Let S be a center set obtained by placing each*

vertex into S independently with probability $p = \Theta((\log n)/\Delta)$. W.h.p., S forms a hitting set for the collection of neighbor sets of all vertices of degree at least Δ. Further, under the LCA model, we may check whether $v \in S$ locally without making any probes.

2.2.1.3 LCA for E_{high}: the Multiple Centers method

Obstacles for the LCA implementation of the global algorithm. Recall that the task of our LCA is to decide whether a query edge $(u, v) \in E$ belongs to H_{std}. Let us focus on the more sophisticated step (B). To implement (B), we impose a natural priority for choosing which edge we add to our spanner. Let the neighbor-list of u be $\{v'_1, \ldots, v'_{\deg(u)}\}$ (as specified by G's adjacency list representation). For a center s, we add (u, v'_i) to H_{std} if v'_i is u's first neighbor that appears in $\mathsf{Clst}(s)$. Equivalently speaking, (u, v'_i) is added in step (B) if and only if v'_i or $\mathsf{Ctr}(v'_i)$ is a new center, not appearing in any $\{v'_j, \mathsf{Ctr}(v'_j)\}_{j<i}$. Thus, to implement this algorithm, it suffices to check whether v'_j is a center and compute $\mathsf{Ctr}(v'_j)$, for every $j \leq i$.

However, attempting to evaluate the above condition naïvely introduces two difficulties. Firstly, if u is an arbitrary vertex, the number of candidates v'_j's we must check is up to $\deg(u)$ which can be linear, so checking all candidates is not a viable option – we will resolve this problem later when we take care of E_{super}, so that the probe complexity becomes sub-linear in the degree. Fortunately for E_{high}, the number of candidates v'_j's we must check is at most $\deg(u) \leq \Delta_{\text{super}}$, which is lower than the desired probe complexity, but is so by only a factor of $\log n$. The second difficulty arises as we must compute $\mathsf{Ctr}(v'_j)$ for each candidate v'_j, but we do not know in advance which of its neighbors are centers. As the expected size $|S \cap \Gamma(v'_j)|$ can be asymptotically much smaller than $\deg(v'_j)$ and $|S|$, we have no hope of evaluating Ctr or even discovering a neighboring center using a logarithmic number of probes, because the orderings of $\Gamma(v'_j)$'s are arbitrary. We address this second issue through the following multiple centers method.

Multiple Centers. We have established that in order to achieve a better probe complexity for computing a vertex's center, we must discard the conventional notation of Ctr. Instead, rather than assigning each vertex a *single* center, we define *multiple* centers as follows. Recall that $\Gamma_{\Delta,1}(v)$ denotes the set of the first Δ neighbors of v in its adjacency list representation.

Definition 2.2.4 (Multiple Centers)**.** *For a given integer $\Delta \in [n]$, a vertex v with degree at least Δ, and center set S, let $\mathsf{MltCtrs}(v, S, \Delta) = S \cap \Gamma_{\Delta,1}(v)$ be the (multiple) centers of v with respect to S and Δ. When Δ and S are clear from the context, we abbreviate it to $\mathsf{MltCtrs}(v)$. Define also $\mathsf{MltCtrs}^+(v) = \mathsf{MltCtrs}(v) \cup \{v\}$ if $v \in S$, and simply $\mathsf{MltCtrs}^+(v) = \mathsf{MltCtrs}(v)$ otherwise; this is the set of centers whose clusters contain v.*

The observation below shows that this new definition has two main benefits for achieving better probe complexity: (i) shows that the number of required membership tests $s \in \mathsf{MltCtrs}(v)$ in (B) will only increase by a $\log n$ factor under this new definition, and (ii) implies that each membership test for $\mathsf{MltCtrs}(v)$ (or, together with Observation 2.2.3, also for $\mathsf{MltCtrs}^+(v)$) with only constant probes.

Observation 2.2.5 (Multiple Centers Properties). *Let S be a center set obtained by placing each vertex into S independently with probability $\Theta((\log n)/\Delta)$. Then for every v with $\deg(v) \geq \Delta$:*

(i) W.h.p., $\mathsf{MltCtrs}(v, S, \Delta)$ contains $\Theta(\log n)$ centers (and in particular, $\mathsf{MltCtrs}(v, S, \Delta) \neq \emptyset$);

(ii) Given a center $s \in S$, then $s \in \mathsf{MltCtrs}(v)$ if and only if the ADJACENCY *probe with parameter $\langle v, s \rangle$ returns a value at most Δ; that is, the center s is among the first Δ neighbors of v.*

Both observations above also apply to $\mathsf{MltCtrs}^+(v)$.

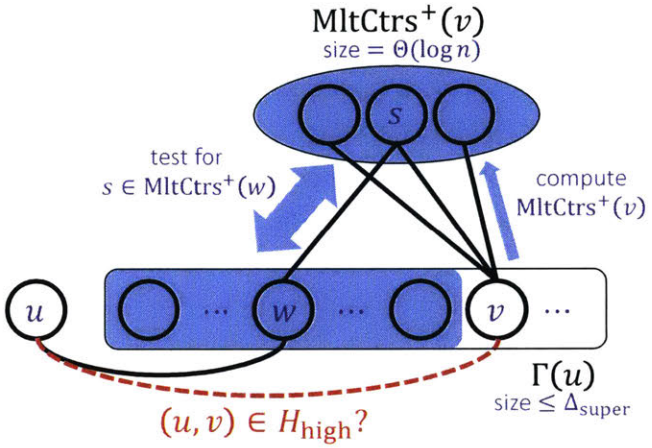


Figure 2-2: Illustration for the local construction of H_{high}.

We apply this definition for $\Delta = \Delta_{\text{high}}$, and modify the construction as follows: for (I), we must add an edge from each $v \in V_{[\Delta_{\text{high}}, \Delta_{\text{super}}]}$ to every center in $\mathsf{MltCtrs}(v)$, and for (B), we add an edge from u to a vertex in $\mathsf{Clst}(s)$, now updated to $\mathsf{Clst}(s) = \{s\} \cup \{v : s \in \mathsf{MltCtrs}(v)\}$.

2.2.1.3.1 The LCA for constructing H_{high} We apply $\mathsf{MltCtrs}$ instead of Ctr in the clustering algorithm for H_{std} with $\Delta = \Delta_{\text{high}}$. For (B), we add an edge from u to a vertex in $\mathsf{Clst}(s) = \{s\} \cup \{v : s \in \mathsf{MltCtrs}(v)\}$. This can be implemented locally by adding edge (u, v'_i) if $\mathsf{MltCtrs}^+(v'_i) \setminus \cup_{j<i} \mathsf{MltCtrs}^+(v'_j) \neq \emptyset$; that is, v'_i "introduces" some new center to the collection $\{\mathsf{MltCtrs}^+(v'_j)\}_{j<i}$. Given $(u, v) \in E$, the LCA for constructing H_{high} decides whether $(u, v) \in H_{\text{high}}$ as follows. (See Fig. 2-2).

Our LCA answers no if no rules apply. Note that for 3-spanners, we may afford to include all incident edges of centers, thereby simplifying (I) – this simplification will not extend to our 5-spanner construction. We remark that rules we give for LCAs throughout this section are inherently unsymmetrical: for example, the (I) rule below does not check for $v \in \mathsf{MltCtrs}(u)$. These unsymmetrical rules must also be reapplied with roles of u and v swapped (e.g., as (v, u)) because edges are undirected.

We now prove its correctness and performance as follows.

Lemma 2.2.6. *For $1 \leq \Delta_{\text{high}} \leq \Delta_{\text{super}} \leq n$, there exists a subgraph $H_{\text{high}} \subseteq G$ such that w.h.p.:*

(i) H_{high} has $O(n^2/\Delta_{\text{high}} \cdot \log n)$ edges,

Local construction of H_{high}. Each $v \in V$ is added to S with probability $p = \Theta((\log n)/\Delta_{\text{high}})$.
(I) If $v \in V_{[\Delta_{\text{high}},\Delta_{\text{super}}]}$ and $u \in \mathsf{MltCtrs}(v)$, answer yes (DEGREE probe and Observation 2.2.3).
(B) If $(u,v) \in E(V_{[1,\Delta_{\text{super}}]}, V_{[\Delta_{\text{high}},\Delta_{\text{super}}]})$: (DEGREE probes)

- Compute $\mathsf{MltCtrs}^+(v)$ by iterating through $\Gamma_{\Delta_{\text{high}},1}(v)$ (NEIGHBOR probes and Observation 2.2.3).
- Denote the neighbor-list of u by $\{v'_1, \ldots, v'_{\deg(u)}\}$; identify i such that $v = v'_i$ (ADJACENCY probe).
- For each $s \in \mathsf{MltCtrs}^+(v)$, iterate to check for a vertex $w \in \{v'_1, \ldots, v'_{i-1}\}$ such that $w \in V_{[\Delta_{\text{high}},\Delta_{\text{super}}]}$ and $s \in \mathsf{MltCtrs}^+(w)$ (DEGREE probe and Observation 2.2.5). Answer yes if there exists a vertex s where no such w exists.

Figure 2-3: Procedure for the local construction of H_{high}.

(ii) H_{high} *takes care of* E_{high}*; that is, for every* $(u,v) \in E_{\text{high}}$, $\text{dist}(u,v,H_{\text{high}}) \leq 3$, *and*
(iii) for a given edge $(u,v) \in E$*, one can test if* $(u,v) \in H_{\text{high}}$ *by making* $O(\Delta_{\text{super}} \log n)$ *probes.*

Proof. **(i) Size.** In step (I), by Observation 2.2.5, each vertex is connected to $O(\log n)$ centers, costing $O(n \log n)$ total edges. In step (B), we add one edge for each $v'_i \in V_{[\Delta_{\text{high}},\Delta_{\text{super}}]}$ introducing a new center. As there are at most $|S|$ centers, we add up to $|S|$ edges for each u, bounding the number of added edges by $O(n \cdot |S|)$ as well. Since $|S| = \Theta(n/\Delta_{\text{high}} \cdot \log n)$ by the Chernoff bound w.h.p., $|E(H_{\text{high}})| = O(n^2/\Delta_{\text{high}} \cdot \log n)$ as desired.

(ii) Stretch. This guarantee follows the argument of H_{std}. Suppose that $(u,v) \in E(V_{[1,\Delta_{\text{super}}]}, V_{[\Delta_{\text{high}},\Delta_{\text{super}}]})$ is omitted, fix some $s \in \mathsf{MltCtrs}^+(v)$ (which exists due to Observation 2.2.5). Let w be the first neighbor of u whose $\mathsf{MltCtrs}^+(w)$ contains s, then the edge (u,w) must have been added to H_{high} in step (B). Edges (s,v) and (s,w) are added in step (I), so u is connected to v via the path $\langle u,w,s,v\rangle$ of length 3.

(iii) Probes. Step (I) clearly takes constant probes. For (B), identifying $\mathsf{MltCtrs}^+(v)$ takes $O(\Delta_{\text{super}})$ probes. Recall by Observation 2.2.3 that $|\mathsf{MltCtrs}^+(v)| = \Theta(\log n)$. The number of candidates w we need to check is at most $\deg(u) \leq \Delta_{\text{super}}$. Checking $w \in V_{[\Delta_{\text{high}},\Delta_{\text{super}}]}$ and $s \in \mathsf{MltCtrs}^+(w)$ takes $O(1)$ probes via Observation 2.2.3 thanks to our multiple centers method. Thus the probe complexity is $\Theta(\log n) \cdot \Delta_{\text{super}} \cdot O(1) = O(\Delta_{\text{super}} \log n)$. □

2.2.1.4 LCA for E_{super}: the Neighbor Partitioning method

Neighbor Partitioning. Consider now $(u,v) \in E(V, V_{[\Delta_{\text{super}},n)}) = E_{\text{super}}$. As discussed in Section 2.2.1.2, our previous LCA can execute step (B) in sub-linear number of probes only because we previously had a sub-linear bound on $\deg(u)$, but this is not the case for E_{super}. Here, we must design an LCA whose probe complexity is sub-linear in the *degrees*: exploring the entire neighborhood of u is forbidden. To this end, we will make our rule for adding edges even "more local" and only explore a consecutive *block* of neighbors within the neighbor-list via our *neighbor partitioning* method.

For H_{super}, we choose a new set of centers S by adding each vertex $v \in V$ to S independently with probability $p = \Theta((\log n)/\Delta_{\text{super}})$, then define $\mathsf{MltCtrs}(v) = S \cap \Gamma_{\Delta_{\text{super}},1}$ accordingly. As

suggested in the preliminaries, we partition the neighbor-list $\Gamma(u)$ into $\Theta(\deg(u)/\Delta_{\text{super}})$ blocks of consecutive neighbors, $\{\Gamma_{\Delta_{\text{super}},i}(u)\}_{i\leq\Theta(\deg(u)/\Delta_{\text{super}})}$, each of size $O(\Delta_{\text{super}})$. We modify our algorithm from that of H_{high} so that we add (u,v) to H_{super} only when v introduces some new center to the collection $\{\mathsf{MltCtrs}(v')\}$ for v' preceding v *within the same block*, independent of all other blocks. More formally, suppose that v is the i^{th} neighbor in the ℓ^{th} block of $\Gamma(u)$. Let $\Gamma_{\Delta_{\text{super}},\ell}(u) = \{v'_{\ell,1},\ldots,v'_{\ell,|\Gamma_{\Delta_{\text{super}},\ell}(u)|}\}$, so $v = v'_{\ell,i}$. We add (u,v) to H_{super} if $\mathsf{MltCtrs}^+(v'_{\ell,i}) \setminus \cup_{j<i}\mathsf{MltCtrs}^+(v'_{\ell,j}) \neq \emptyset$. (See Fig. 2-4 for an illustration).

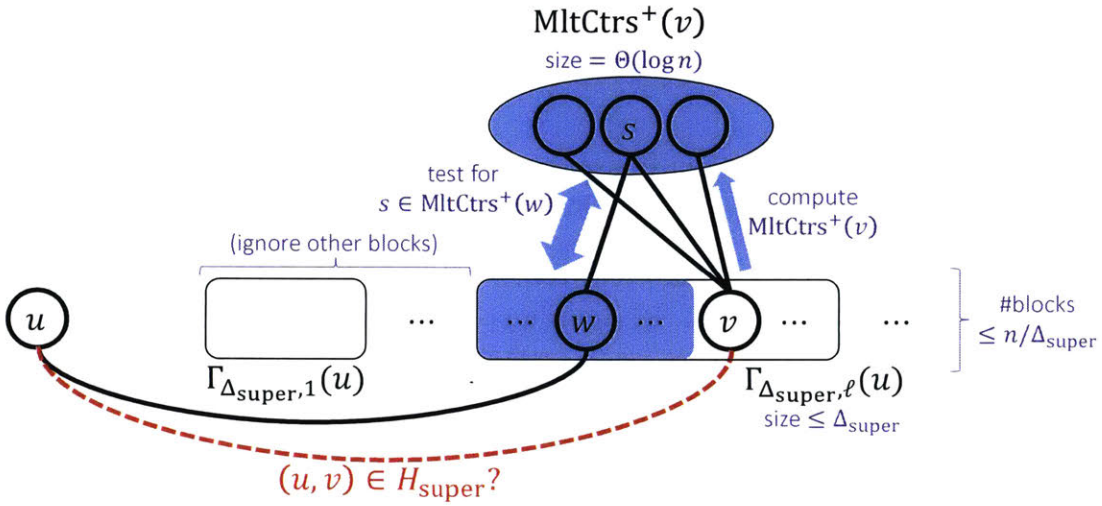


Figure 2-4: Local construction of H_{super}.

Local construction of H_{super}. Each $v \in V$ is added to S with probability $p = \Theta((\log n)/\Delta_{\text{super}})$.
(I) If $v \in V_{[\Delta_{\text{super}},n)}$ and $u \in \mathsf{MltCtrs}(v)$, answer yes.
(B) If $v \in V_{[\Delta_{\text{super}},n)}$:

- Compute $\mathsf{MltCtrs}^+(v)$ by iterating through $\Gamma_{\Delta_{\text{super}},1}(v)$.
- Denote the ℓ^{th} block of $\Gamma(u)$ by $\Gamma_{\Delta_{\text{super}},\ell}(u) = \{v'_{\ell,1},\ldots,v'_{\ell,|\Gamma_{\Delta_{\text{super}},\ell}(u)|}\}$; identify ℓ, i where $v = v'_{\ell,i}$.
- For each $s \in \mathsf{MltCtrs}^+(v)$, iterate to check for a vertex $w \in \{v'_{\ell,1},\ldots,v'_{\ell,i-1}\}$ such that $w \in V_{[\Delta_{\text{super}},n)}$ and $s \in \mathsf{MltCtrs}^+(w)$. Answer yes if there exists a vertex s where no such w exists.

Figure 2-5: Procedure for the local construction of H_{super}.

Compared to the construction of H_{high}, this process may add up to a factor of n/Δ_{super} redundant edges, but saves a factor of $\Delta_{\text{super}}/\Delta_{\text{high}}$ in the number of centers: our choices of Δ_{super} and Δ_{high} balance these two factors at $n^{1/(2r)}$, so that the total number of edges remain unchanged.

Lemma 2.2.7. *For $1 \leq \Delta_{\text{super}} \leq n$, there exists a subgraph $H_{\text{super}} \subseteq G$ such that w.h.p.:*

(i) H_{super} has $O(n^3/\Delta^2_{\text{super}} \cdot \log n)$ edges,

(ii) H_{super} takes care of E_{super}; that is, for every $(u,v) \in E_{\text{super}}$, $\text{dist}(u,v,H_{\text{super}}) \leq 3$, and

(iii) for a given edge $(u,v) \in E$, one can test if $(u,v) \in H_{\text{super}}$ by making $O(\Delta_{\text{super}} \log n)$ probes.

Proof. **(i) Size.** Here, in step (B), we add one edge for each $v'_{\ell,i} \in V_{[\Delta_{\text{super}},n)}$ introducing a new center in the ℓ^{th} block. As there are at most $|S|$ centers and $\Theta(n/\Delta_{\text{super}})$ blocks, we add up to $\Theta(n|S|/\Delta_{\text{super}})$ edges for each u, bounding the total number of added edges by $O(n^2|S|/\Delta_{\text{super}})$. Since now $|S| = \Theta(n/\Delta_{\text{super}} \cdot \log n)$ by the Chernoff bound w.h.p., $|E(H_{\text{super}})| = O(n^3/\Delta^2_{\text{super}} \cdot \log n)$ as desired.

(ii) Stretch. This guarantee follows closely from the same argument as that of H_{high}. In fact, $E(H_{\text{super}})$ includes all the edges that we would have added if we were not to apply our neighbor partitioning method.

(iii) Probes. Similarly to the analysis of H_{high}, step (I) takes constant probes, step (B) identifies $\mathsf{MltCtrs}^+(v)$ using $O(\Delta_{\text{super}})$ probes, and $|\mathsf{MltCtrs}^+(v)| = \Theta(\log n)$. Now, by our neighbor partitioning method, the number of candidates w we need to check is reduced to at most the block size, $O(\Delta_{\text{super}})$. Thus the probe complexity is $O(\Delta_{\text{super}}) \cdot \Theta(\log n) = O(\Delta_{\text{super}} \log n)$. □

2.2.1.5 Final 3-spanner results

To obtain an LCA for 3-spanners, we simply apply our LCA for constructing $H_{\text{low}}, H_{\text{high}}$ and H_{super}, then answer yes if any of them does so. Applying Lemma 2.2.6-2.2.7 with $r = 2$ (so that $\Delta_{\text{low}} = \Delta_{\text{high}} = n^{1/2}$ and $\Delta_{\text{super}} = n^{3/4}$), we obtain the following LCA result for 3-spanner in general graphs.

Theorem 2.2.8. *For every n-vertex graph $G = (V, E)$ there exists an LCA for 3-spanner with $O(n^{3/2} \log n)$ edges and probe complexity $O(n^{3/4} \log n)$.*

Moreover, by simply combining the results only for H_{high} and H_{super}, we obtain an LCA for 3-spanners of smaller sizes for graphs where every edge is incident to a vertex of degree at least $n^{1-1/r}$. This includes, in particular, graphs of minimum degree $n^{1-1/r}$.

Theorem 2.2.9. *For every $r \geq 1$ and n-vertex graph $G = (V, E)$, such that every edge is incident to a vertex of degree at least $n^{1-1/r}$, there exists an LCA for 3-spanner with $O(n^{1+1/r} \log n)$ edges and probe complexity $O(n^{1-1/(2r)} \log n)$.*

2.2.2 5-spanners

We now consider LCAs for 5-spanners, aiming for spanners of size $\widetilde{O}(n^{4/3})$ with probe complexity $\widetilde{O}(n^{5/6})$. As our construction heavily relies on the structures and tools developed for 3-spanners, we provide intuition on the modifications and extensions required to turn them into LCAs for 5-spanners, as well as introduce our new methods in Section 2.2.2.1. Detailed constructions are given in Section 2.2.2.2-2.2.2.3, leading to our results in Section 2.2.2.4.

2.2.2.1 Overview

We begin by considering cases handled by our 3-spanners: all of our constructions can still be readily applied. We may afford to set $r = 3$ to construct subgraphs of size $\widetilde{O}(n^{1+1/r}) = \widetilde{O}(n^{4/3})$ with probe complexity $\widetilde{O}(n^{1-1/(2r)}) = \widetilde{O}(n^{5/6})$, but some edges are not taken care of: (u, v) where

$\max\{\deg(u), \deg(v)\} \in (\Delta_{\text{low}}, \Delta_{\text{high}})$. Observe that the construction of H_{low} only generates a subgraph of size $O(n \cdot \Delta_{\text{low}}) = O(n^{1+1/r})$ even if one endpoint of the query edge has degree above Δ_{low}. Applying this modified construction of H_{low}, together with the construction of H_{super} (via Lemma 2.2.7), we have taken care of edges $(u, v) \in E \setminus (V_{(\Delta_{\text{low}}, \Delta_{\text{super}})} \times V_{(\Delta_{\text{low}}, \Delta_{\text{super}})})$. (Lemma 2.2.6 also applies, but is not needed in our upcoming construction.)

Let $\Delta_{\text{med}} = n^{1/2-1/(2r)}$. We will design a subgraph $H \subseteq G$ that takes care of the remaining edges $E_{\text{med}} = E(V_{[\Delta_{\text{med}}, \Delta_{\text{super}}]}, V_{[\Delta_{\text{med}}, \Delta_{\text{super}}]})$, satisfying $\text{dist}(u, v, H) \leq 5$. Since $\Delta_{\text{low}} = \Delta_{\text{med}} = n^{1/3}$ for $r = 3$, this construction, together with the previous cases above, takes care of all edges and yields a 5-spanner for any general graph G. To take care of E_{med}, we consider two different methods.

Cluster partitioning method. First, consider the clustering-based approach (local construction of H_{std} from Section 2.2.1.2), where in step (B), for each pair of vertex u and center s, we add a single edge from u to some vertex of $\mathsf{Clst}(s)$, ensuring that the stretch is 3. Now that stretch 5 is allowed, we instead consider each pair s, t of centers, then add a single edge between $\mathsf{Clst}(s)$ and $\mathsf{Clst}(t)$. Thus, if $(u, v) \in E(\mathsf{Clst}(s), \mathsf{Clst}(t))$ is omitted, then another edge $(u', v') \in E(\mathsf{Clst}(s), \mathsf{Clst}(t))$ must have been kept, providing the path $\langle u, s, u', v', t, v \rangle$ of length 5 (or less, in case $u' = s$ or $v' = t$). For $|S| = \widetilde{\Theta}(n/\Delta_{\text{med}})$ clusters, the total number of edges added in step (B) is $\widetilde{O}((n/\Delta_{\text{med}})^2) = O(n^{1+1/r} \log^2 n)$ as desired.

Unfortunately for the LCA implementation, choosing the first edge in $E(\mathsf{Clst}(s), \mathsf{Clst}(t))$, or even verifying whether one exists, requires iterating through each pair of vertices from these clusters: this process takes up to $\deg(s) \cdot \deg(t)$ ADJACENCY probes, which may not be sub-linear in n. Instead, similarly to the neighbor partitioning method, we partition each cluster into *buckets* of size (mostly) Δ_{med}, then add a single edge connecting each pair between these buckets. As the total size of all clusters is $O(n \log n)$ (since each vertex chooses $O(\log n)$ centers and hence joins $O(\log n)$ clusters), the total number of buckets is still $\widetilde{O}(n/\Delta_{\text{med}})$.

Another difficulty arises as, unlike neighbors that can be divided into blocks via indices, we must spend $\Theta(\deg(s))$ probes identifying the entire $\mathsf{Clst}(s)$ in order to break it into roughly equal-sized buckets: as illustrated in Fig. 2-6, the size of $\mathsf{Clst}(s)$, and its vertex distribution among $\Gamma^+(s)$, can be extremely arbitrary. Hence, this method only provides the desired probe complexity when we restrict that centers are of degree at most $n^{1-1/(2r)} = \Delta_{\text{super}}$. Consequently, we apply this method to take care of edges for which *both* endpoints have plenty of neighbors of degree *at most* Δ_{super} in their first block ($\Gamma_{\Delta_{\text{med}}, 1}$) of neighbors.

Representative method. Alternatively, we may connect vertices through the existing cluster structure H_{super} if they have some neighbor of degree at least Δ_{super}. We connect each vertex $v \in V_{[\Delta_{\text{med}}, \Delta_{\text{super}}]}$ to a number of *representatives*, denoted $\mathsf{Reps}(v) \subseteq \Gamma(v) \cap V_{[\Delta_{\text{super}}, n)}$; representatives that share some center are already connected via H_{super}. Thus, we may modify step (B) of the local construction of H_{high} from Section 2.2.1.3, so that we only add (u, v) to our subgraph if, *through its representatives*, v introduces a new center. Namely, if (u, v) were omitted, then for a fixed $x \in \mathsf{Reps}(v)$ and $s \in \mathsf{MltCtrs}^+(x)$, we must have added the first neighbor w in v's block of $\Gamma(u)$, such that there exists $y \in \mathsf{Reps}(w)$ with $s \in \mathsf{MltCtrs}^+(y)$, creating a path $\langle u, w, y, s, x, v \rangle$ of length 5 (or less, if $y = s$ or $x = s$). See Fig. 2-7 for illustration. The number of edges added in (B) remains the same as that of H_{high} because we still only add $|S|$ edges per each u, by the same analysis.

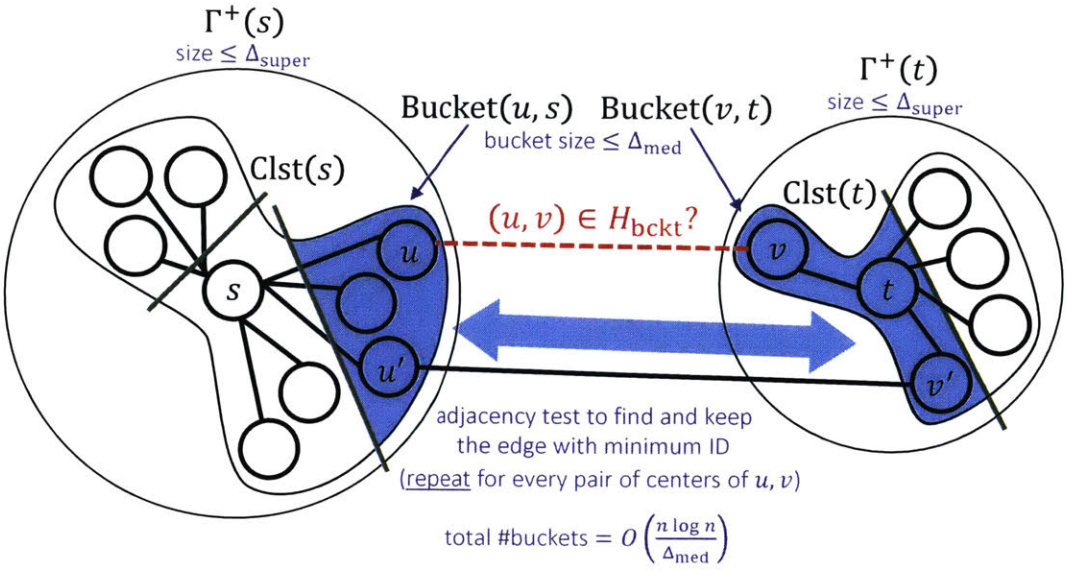


Figure 2-6: Local construction of H_{bckt}.

The only remaining issue for this approach is designing Reps. To meet the size and probe complexity requirements, we want $|\text{Reps}(v)|$ and the number of probes of each operation (computing or verifying membership of $\text{Reps}(v)$) to be at most poly-logarithmic. To this end, we inspect $O(\log n)$ random neighbors of v, chosen from $\Gamma_{\Delta_{\text{med}},1}(v)$ for consistency, hoping to find a neighbor of degree at least Δ_{super}. (These indices of neighbors are chosen as a function of the vertex's ID and the random bits, so it is consistent throughout the execution of the LCA.) So, to ensure that this approach succeeds w.h.p., we apply this method to take care of edges for which *some* endpoint have plenty of neighbors of degree *at least* Δ_{super} in their first blocks of neighbors.

Criteria for edges. We aim to take care of edges for which both endpoints are in $V_{[\Delta_{\text{med}},\Delta_{\text{super}}]}$. To categorize our edges for the purpose of constructing 5-spanners, we need the following partition of these vertices.

Definition 2.2.10 (Deserted and Crowded vertices). *A vertex $v \in V_{[\Delta_{\text{med}},\Delta_{\text{super}}]}$ is* deserted *if at least half of its neighbors in the first block $\Gamma_{\Delta_{\text{med}},1}(v)$ are of degree at most Δ_{super}; i.e., $|\Gamma_{\Delta_{\text{med}},1}(v) \cap V_{[1,\Delta_{\text{super}}]}| \geq \Delta_{\text{med}}/2$. Otherwise, the vertex is* crowded.

Let V_{dsrt} (resp., V_{crwd}) be the set of deserted (resp., crowded) vertices in $V_{[\Delta_{\text{med}},\Delta_{\text{super}}]}$. Given a vertex, we can verify whether it is in any of these sets using $O(\Delta_{\text{med}})$ probes by checking the degrees of v and each vertex in $\Gamma_{\Delta_{\text{med}},1}(v)$. We then assign each $(u,v) \in E$ into one of the four cases $\{\text{low}, \text{bckt}, \text{rep}, \text{super}\}$ as given in the table below. It is straightforward to verify that when $\Delta_{\text{low}} = \Delta_{\text{med}}$ (namely when we choose $r = 3$, which also yields the required performance), these four cases take care of all edges in E. We note that H_{rep} assumes that H_{super} is included: E_{rep} is taken care by $H_{\text{rep}} \cup H_{\text{super}}$, not by H_{rep} alone.

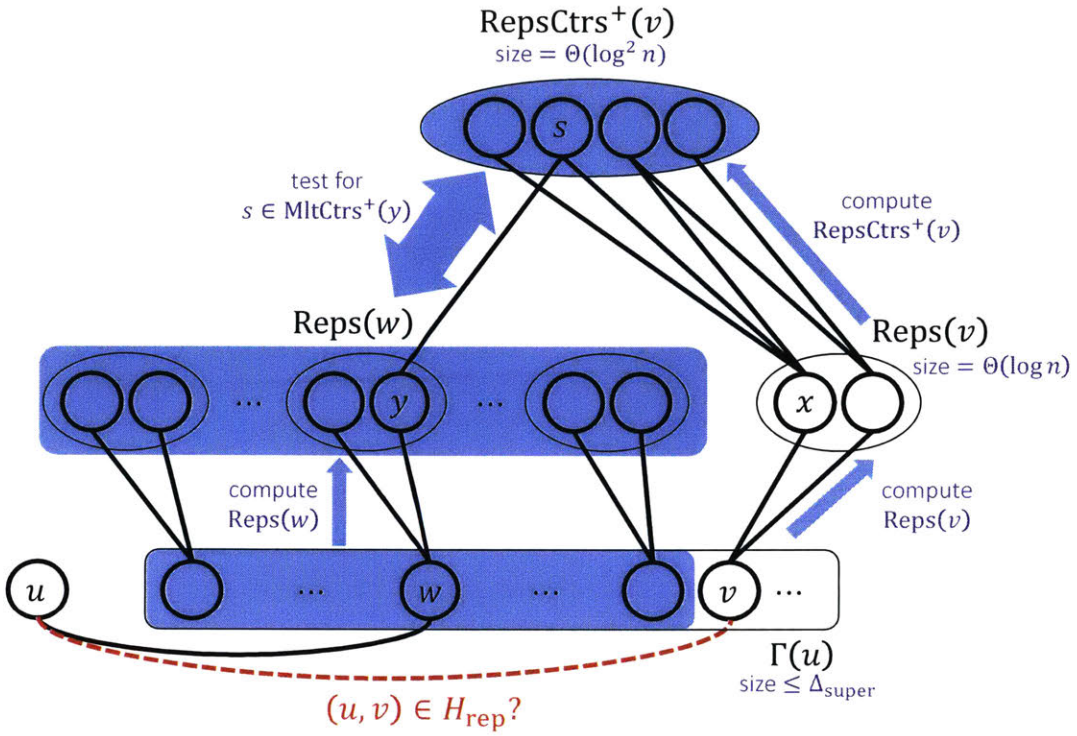


Figure 2-7: Local construction of H_{rep}.

Subset	Criteria	# Edges	Probe Complexity
E_{low}	$(u,v) \in E(V, V_{[1,\Delta_{\text{low}}]})$	$O(n \cdot \Delta_{\text{low}}) = O(n^{1+\frac{1}{r}})$	$O(1)$
E_{bckt}	$(u,v) \in E(V_{\text{dsrt}}, V_{\text{dsrt}})$	$O(\frac{n^2 \log^2 n}{\Delta_{\text{med}}^2}) = O(n^{1+\frac{1}{r}} \log^2 n)$	$O((\Delta_{\text{super}} + \Delta_{\text{med}}^2) \log^2 n) = O(n^{1-\frac{1}{2r}} \log^2 n)$
E_{rep}	$(u,v) \in E(V_{[\Delta_{\text{med}},\Delta_{\text{super}}]}, V_{\text{crwd}})$	$O(\frac{n^2}{\Delta_{\text{super}}} \cdot \log n) = O(n^{1+\frac{1}{r}} \log n)$	$O(\Delta_{\text{super}} \log^3 n) = O(n^{1-\frac{1}{2r}} \log^3 n)$
E_{super}	$(u,v) \in E(V, V_{[\Delta_{\text{super}},n)})$	$O(\frac{n^3 \log n}{\Delta_{\text{super}}^2}) = O(n^{1+\frac{1}{r}} \log n)$	$O(\Delta_{\text{super}} \log n) = O(n^{1-\frac{1}{2r}} \log n)$

Table 2.3: Edge categorization for the construction of 5-spanners.

2.2.2.2 LCA for E_{bckt}: the Cluster Partitioning method

We give the algorithm below, which requires significant clarification as follows.

- Only vertices of degree at most Δ_{super} are chosen to be in S with probability $p = \Theta((\log n)/\Delta_{\text{med}})$. Since at least half the vertices in $\Gamma_{\Delta_{\text{med}},1}(v)$ for any $v \in V_{\text{dsrt}}$ have degree smaller than Δ_{super}, Observation 2.2.5 still holds: $|\mathsf{MltCtrs}^+(v)| = \Theta(\log n)$ and condition $s \in \mathsf{MltCtrs}^+(v)$ requires constant probes to verify.
- In (B), the partitioning of clusters into buckets may be done in a consistent way (regardless of the given query edge); for instance, create a list of vertices in the cluster, sort them according to their IDs, divide the list into buckets of size Δ_{med} possibly except for the last one. Note that we partition $\mathsf{Clst}(s)$ and $\mathsf{Clst}(t)$ separately – we do not combine their elements. Similarly, once we obtain buckets containing u and v, the order in which we check the adjacency of u' and v' must be consistent. To this end, define the ID of an edge (u,v) as $(\mathsf{ID}(u),\mathsf{ID}(v))$, where the comparison between edge IDs is lexicographic. Thus, this step only adds the edge

of minimum ID between the two clusters.

- We also set the precondition $(u, v) \in E(V_{[\Delta_{\text{med}},n)}, V_{[\Delta_{\text{med}},n)})$, and consistently only allow candidate pairs $(u', v') \in E(V_{[\Delta_{\text{med}},n)}, V_{[\Delta_{\text{med}},n)})$, to ensure that the lexicographically first edge of this exact specification is added if one exists. We do not restrict to E_{bckt}, which require both endpoints to be deserted vertices, because checking whether $(u', v') \in E_{\text{bckt}}$ would take $\Theta(\Delta_{\text{med}})$ probes instead of constant probes. We restrict to edges whose endpoints have degrees at least Δ_{med} instead of considering the entire E so that $\mathsf{MltCtrs}^+$ would be well-defined.

Local construction of H_{bckt}. Each $v \in V_{[1,\Delta_{\text{super}}]}$ is added to S with probability $p = \Theta((\log n)/\Delta_{\text{med}})$.

(I) If $u \in \mathsf{MltCtrs}^+(v)$ or $v \in \mathsf{MltCtrs}^+(u)$, answer yes.

(B) If $(u, v) \in E(V_{[\Delta_{\text{med}},n)}, V_{[\Delta_{\text{med}},n)})$:

- Compute $\mathsf{MltCtrs}^+(u)$ and $\mathsf{MltCtrs}^+(v)$ by iterating through $\Gamma_{\Delta_{\text{med}},1}(u)$ and $\Gamma_{\Delta_{\text{med}},1}(v)$.
- For each pair of $s \in \mathsf{MltCtrs}^+(u)$ and $t \in \mathsf{MltCtrs}^+(v)$:
  - Partition each of $\mathsf{Clst}(s)$ and $\mathsf{Clst}(t)$ into buckets of size (mostly) Δ_{med}. Denote the buckets containing u and v by $\mathsf{Bucket}(u, s)$ and $\mathsf{Bucket}(v, t)$, respectively.
  - Iterate through each pair of $u' \in \mathsf{Bucket}(u, s)$ and $v' \in \mathsf{Bucket}(v, t)$ and check if $(u', v') \in E(V_{[\Delta_{\text{med}},n)}, V_{[\Delta_{\text{med}},n)})$. Answer yes if the edge of minimum ID found is $(u', v') = (u, v)$.

Figure 2-8: Procedure for the local construction of H_{bckt}.

Lemma 2.2.11. *For $1 \le \Delta_{\text{med}} \le \sqrt{n} \le \Delta_{\text{super}} \le n$, there exists a subgraph $H_{\text{bckt}} \subseteq G$ such that w.h.p.:*

(i) H_{bckt} has $O(\frac{n^2 \log^2 n}{\Delta_{\text{med}}^2})$ edges,

(ii) H_{bckt} takes care of E_{bckt}; that is, for every $(u, v) \in H_{\text{bckt}}$, $\text{dist}(u, v, H_{\text{bckt}}) \le 5$, and

(iii) for a given edge $(u, v) \in E$, one can test if $(u, v) \in H_{\text{bckt}}$ by making $O((\Delta_{\text{super}} + \Delta_{\text{med}}^2) \log^2 n)$ probes.

Proof. **(i) Size.** In (I) we add $|\mathsf{MltCtrs}^+(v)| = \Theta(\log n)$ edges for each $v \in V_{\text{dsrt}}$, which constitutes to $O(n \log n)$ edges in total. In (B), we add one edge between each pair of buckets. We now compute the total number of buckets. The total size of clusters $\sum_{s \in S} |\mathsf{Clst}(s)| \le |S| + \sum_{v \in V_{[\Delta_{\text{med}},n)}} |\mathsf{MltCtrs}(v)| = O(n \log n)$, so there can be up to $O((n \log n)/\Delta_{\text{med}})$ full buckets of size Δ_{med}. As buckets are formed by partitioning $|S|$ clusters, there are up to $|S| = \Theta((n \log n)/\Delta_{\text{med}})$ remainder buckets of size less than Δ_{med}. Thus, there are $\Theta((n \log n)/\Delta_{\text{med}})$ buckets, and $O(((n \log n)/\Delta_{\text{med}})^2)$ edges are added in (B).

(ii) Stretch. Suppose that (u, v) is omitted. Fix centers $s \in \mathsf{MltCtrs}^+(u)$ and $t \in \mathsf{MltCtrs}^+(v)$, then the lexicographically-first edge $(u', v') \in E(\mathsf{Bucket}(u, s), \mathsf{Bucket}(v, t))$ must have been added to H_{bckt}, forming the path $\langle u, s, u', v', t, v \rangle$ (or shorter, if there are repeated vertices), yielding $\text{dist}(u, v, H_{\text{bckt}}) \le 5$.

(iii) Probes. Computing $\mathsf{MltCtrs}^+(u)$ and $\mathsf{MltCtrs}^+(v)$ takes $O(\Delta_{\text{med}})$ probes. For each pairs of centers, we scan through the entire neighbor-lists $\Gamma(s)$ and $\Gamma(t)$ and collect all vertices in their respective clusters. This takes $O(\Delta_{\text{super}})$ probes each because we restrict to centers of degree at most

Δ_{super}. Given the clusters, we identify the buckets containing u and v each of size $O(\Delta_{\text{med}})$. We then check through candidates (u', v') between these buckets, taking $O(\Delta^2_{\text{med}})$ ADJACENCY probes. So, each pair of centers requires $O(\Delta_{\text{super}} + \Delta^2_{\text{med}})$ total probes. We repeat the process for $|\mathsf{MltCtrs}^+(u)| \cdot |\mathsf{MltCtrs}^+(v)| = O(\log^2 n)$ pairs of centers w.h.p., yielding the claimed probe complexity. □

2.2.2.3 LCA for E_{rep}: the Representative method

We first explain the computation of $\mathsf{Reps}(v)$ for $v \in V_{\text{crwd}}$. Using the random bits and the vertex ID of the parameter v, we sample a set R_v of $\Theta(\log n)$ (not necessarily distinct) values in $[\Delta_{\text{med}}]$ at random (for details, see Section 2.4). Denote the neighbor-list of v by $\{x'_1, \ldots, x'_{\deg(v)}\}$, then define $\mathsf{Reps}(v) = \{x'_i : i \in R_v \text{ and } \deg(x'_i) \geq \Delta_{\text{super}}\}$. Then since at least half of the vertices in $\Gamma_{\Delta_{\text{med}},1}(v)$ are of degree at least Δ_{super}, w.h.p. $\mathsf{Reps}(v) \neq \emptyset$. For consistency, we allow the same definition for $\mathsf{Reps}(v)$ for any $v \in V_{[\Delta_{\text{med}},n)}$ as well, even if it may result in empty sets of representatives. The selection of indices R_v can be done without any probes, so computing $\mathsf{Reps}(v)$ requires $\Theta(\log n)$ probes (by checking the degrees of these neighbors), and verifying whether $u \in \mathsf{Reps}(v)$ requires $O(1)$ probes (by finding the index of u in $\Gamma(v)$ and checking if it is in R_v).

Assume that H_{super} (with stretch 3) is present; we aim to show that $H_{\text{rep}} \cup H_{\text{super}}$ takes care of E_{rep} (with stretch 5). For convenience, define $\mathsf{RepsCtrs}^+(v) = \cup_{x' \in \mathsf{Reps}(v)} \mathsf{MltCtrs}^+(x')$, the set of (multiple) centers of any of v's reprsentatives. Observe that by adding (v, x') for every $x' \in \mathsf{Reps}(v)$, it yields that $\text{dist}(v, s, H_{\text{rep}} \cup H_{\text{super}}) \leq 2$ for any $s \in \mathsf{RepsCtrs}^+(v)$.

Consider the query (u, v), and suppose that $v = v'_i$ is the i^{th} neighbor in u's neighbor-list, $\Gamma(u) = \{v'_1, \ldots, v'_{\deg(u)}\}$. We then add (u, v) to H_{rep} if and only if v introduces a new center through some representative; that is, $\mathsf{RepsCtrs}^+(v'_i) \setminus \cup_{j<i} \mathsf{RepsCtrs}^+(v'_j) \neq \emptyset$. To verify this condition locally, we first compute $\mathsf{RepsCtrs}^+(v)$, and for each of $\{v'_j\}_{j<i}$, $\mathsf{Reps}(v'_j)$. Then, we discard (u, v) if for every center $s \in \mathsf{RepsCtrs}^+(v)$, there exists x and v'_j where $x \in \mathsf{Reps}(v'_j)$ and $s \in \mathsf{MltCtrs}^+(x)$; the last condition takes constant probes to verify due to Observation 2.2.5, which is our most economical choice here. This gives the full LCA for constructing H_{rep} below.

Local construction of H_{rep}. Each $v \in V$ is added to S with probability $p = \Theta((\log n)/\Delta_{\text{super}})$.
(I) If $v \in V_{[\Delta_{\text{med}},\Delta_{\text{super}}]}$ and $u \in \mathsf{Reps}(v)$, answer yes.
(B) If $u, v \in V_{[\Delta_{\text{med}},\Delta_{\text{super}}]}$:

- Compute $\mathsf{RepsCtrs}^+(v)$.
- Denote the neighbor-list of u by $\{v'_1, \ldots, v'_{\deg(u)}\}$; identify i such that $v = v'_i$.
- For each vertex $w \in \{v'_1, \ldots, v'_{i-1}\}$, if $w \in V_{[\Delta_{\text{med}},\Delta_{\text{super}}]}$, compute $\mathsf{Reps}(w)$.
- For each $s \in \mathsf{RepsCtrs}^+(v)$, iterate to check for a vertex y in any of the $\mathsf{Reps}(w)$'s obtained above, such that $s \in \mathsf{MltCtrs}^+(y)$. Answer yes if there exists a vertex s where no such y exists.

Figure 2-9: Procedure for the local construction of H_{rep}.

Lemma 2.2.12. *For $1 \leq \Delta_{\text{med}} \leq \Delta_{\text{super}} \leq n$, there exists a subgraph $H_{\text{rep}} \subseteq G$ such that w.h.p.:*

(i) H_{rep} has $O(n^2/\Delta_{\text{super}} \cdot \log n)$ edges,

(ii) $H_{\text{rep}} \cup H_{\text{super}}$ takes care of E_{rep}; that is, for every $(u, v) \in E_{\text{rep}}$, $\text{dist}(u, v, H_{\text{rep}} \cup H_{\text{super}}) \leq 3$, and

(iii) for a given edge $(u,v) \in E$, one can test if $(u,v) \in H_{\text{rep}}$ by making $O(\Delta_{\text{super}} \log^3 n)$ probes.

Proof. **(i) Size.** W.h.p., in (I) we add at most $\sum_{v \in V_{[\Delta_{\text{med}}, \Delta_{\text{super}}]}} |\mathsf{Reps}(v)| \leq n \cdot O(\log n) = O(n \log n)$. Similarly to the analysis of H_{high}, in (B) we add $|S| = O((n \log n)/\Delta_{\text{super}})$ edges per vertex u, so $|E(H_{\text{rep}})| = O(n^2/\Delta_{\text{super}} \cdot \log n)$.

(ii) Stretch. This claim follows from the argument given in the overview, and is similar to the analysis of H_{high}.

(iii) Probes. Computing $\mathsf{RepsCtrs}^+(v)$ takes $O(\log n) \cdot \Delta_{\text{super}} = O(\Delta_{\text{super}} \log n)$ (recall that we only check $\Gamma_{\Delta_{\text{super}},1}$ of each reprsentative). Note also that $|\mathsf{RepsCtrs}^+(v)| = O(\log^2 n)$ since v has $O(\log n)$ representative, each of which belongs to $\Theta(\log n)$ clusters. Computing Reps for each neighbor $w \in \{v'_j\}_{j<i}$ of u takes $O(\log n)$ probes each, which is $O(\Delta_{\text{super}} \log n)$ in total since $\deg(u) \leq \Delta_{\text{super}}$. This also introduces up to $\Delta_{\text{super}} \cdot O(\log n)$ representatives in total. Checking whether each of the $O(\log^2 n)$ centers in $\mathsf{RepsCtrs}^+(v)$ is a center of each of these $O(\Delta_{\text{super}} \log n)$ representative takes, in total w.h.p., $O(\Delta_{\text{super}} \log^3 n)$ probes. □

2.2.2.4 Final 5-spanner results

To obtain an LCA for 5-spanners, we again invoke all of our LCAs for the four cases. Applying Lemma 2.2.7, 2.2.11 and 2.2.12, we obtain the following LCA result for 5-spanner in general graphs.

Theorem 2.2.13. *For every n-vertex graph $G = (V, E)$ there exists an LCA for 5-spanner with $O(n^{4/3} \log^2 n)$ edges and probe complexity $O(n^{5/6} \log^3 n)$.*

Again, by combining results for larger degrees, we obtain an LCA for 5-spanners with smaller sizes on graphs with minimum degree at least $n^{1/2-1/(2r)}$.

Theorem 2.2.14. *For every $r \geq 1$ and n-vertex graph $G = (V, E)$ with minimum degree at least $n^{1/2-1/(2r)}$, there exists an LCA for 5-spanner with $O(n^{1+1/r} \log^2 n)$ edges and probe complexity $O(n^{1-1/(2r)} \log^3 n)$.*

The proofs in this section do not show that the LCA uses a polylogarithmic number of independent random bits. To obtain the main result for 3 and 5-spanner construction, we refer to Section 2.4 for missing details:

Proof of Theorem 2.1.1. The main theorem follows from Theorem 2.2.8 and Theorem 2.2.13, where the independent random bits assumption are resolved according to Section 2.4. More specifically, we show the hitting set argument under polylogarithmic number of independent random bits in Section 2.4.2. □

2.3 LCA for Graphs with Maximum Degree Δ

In this section, we show our construction of LCAs for low-stretch spanners on graphs of maximum degree $\Delta = O(n^{1/12-\epsilon})$ with sub-linear probe complexity:

Theorem 2.3.1. *For $k \geq 1$, and n-vertex graph with maximum degree Δ, there exists an LCA that computes an $O(k^2)$-spanner $H \subseteq G$ with $\widetilde{O}(n^{1+1/k})$ edges and probe complexity $\widetilde{O}(\Delta^4 n^{2/3})$. In particular, the probe complexity of the algorithm is sublinear for $\Delta = O(n^{1/12-\epsilon})$.*

Our construction is inspired by the recent work on the local construction of sparse spanning subgraphs of Lenzen and Levi [LL18]. In particular, we extend their construction in two major aspects. First, we improve upon the stretch factor of the constructed spanner from $O(\log n \cdot (\Delta + \log n))$ down to $O(k^2)$ for any $k \geq 1$, thereby removing the dependencies on Δ and n completely, at the cost of increasing the number of spanner edges from $\Theta(n)$ to $\widetilde{O}(n^{1+1/k})$. Second, we offer an improved analysis, showing that the number of required independent random bits can be reduced from linear to only polylogarithmic in n. We remark that the probe complexity in our construction is largely kept unchanged from that of [LL18].

2.3.1 Overview

We now provide some preliminaries and an outline of our $O(k^2)$-spanner construction. Throughout the main part of this section, we fix two parameters $L = \Theta(n^{1/3})$ and $p = 1/L$: these values will show up in contexts as a threshold or a probability involved in various quantities. We only need to consider $k = O(\log n)$ because, by the size-stretch tradeoff of spanners, any $k = \Omega(\log n)$ yields a spanner of (roughly) linear size, $\widetilde{O}(n^{1+1/k}) = \widetilde{O}(n)$. We note that LCAs in this section only make use of the NEIGHBOR probes. We note that the construction in Theorem 2.3.1 only use NEIGHBOR probes for finding neighbors of vertices; other probe types will only be used in the extension of our LCA to support larger maximum degree.

Sparse and dense vertices. We first sample a collection S of $O((n \log n)/L) = \widetilde{O}(n^{2/3})$ centers, which is implemented locally by having each vertex elect itself as a center with probability $p_{\text{center}} = (c_{\text{center}} \log n)/L$ for some constant $c_{\text{center}} > 0$. In particular, we remark that we never explicitly enumerate the entire set S, but only rely on the fact that we may locally determine whether a given vertex v is a center based on its ID and the randomness, without using any probes. Next, we partition our vertices into *sparse* and *dense* vertices with respect to the center set S based on their distances to the respective closest centers: a vertex v is considered sparse if it is at distance more than k away from all centers, and it is dense otherwise. By a hitting set argument, if the k^{th}-neighborhood of v is of size at least L, then it most likely contains a center, making v a dense vertex. This observation suggests that to verify that a vertex is dense, we do not necessarily need to find some center in v's potentially large k^{th}-neighborhood: it also suffices to confirm that the neighborhood itself is large.

Definition 2.3.2 (Sparse and dense)**.** *A vertex v is* sparse *in G if $\Gamma^k(v, G) \cap S = \emptyset$ and otherwise, it is* dense*. Denote the sets of sparse vertices and dense vertices by V_{sparse} and V_{dense}, respectively.*

We next partition the edge set of G into $E_{\text{sparse}} = E(V, V_{\text{sparse}})$ and $E_{\text{dense}} = E(V_{\text{dense}}, V_{\text{dense}})$, then take care[14] of them by constructing $H_{\text{sparse}} \subseteq E_{\text{sparse}}$ and $H_{\text{dense}} \subseteq E_{\text{dense}}$, so that $H =$

[14] As a reminder, to "take care" of an edge (u, v), we ensure that in the constructed spanner, there is a u-v path whose length is at most the desired stretch factor. See Definition 2.2.1 for its formal definition.

$H_{\text{sparse}} \cup H_{\text{dense}}$ gives a spanner for all edges of G. See Table 2.4 for a summary of the properties of each spanner.

Subset	Criteria	Spanner Edges	# Edges	Probe Complexity
E_{sparse}	at least one endpoint is sparse	H_{sparse}	$O(kn^{1+1/k})$	$O(\Delta^2 L^2)$
E_{dense}	both endpoints are dense	$H^{(\text{I})}_{\text{dense}}$	$O(n)$	$O(\Delta^2 L^2)$
		$H^{(\text{B})}_{\text{dense}}$	$O(n^{1+1/k}\log^4 n)$	$O(p\Delta^4 L^3 \log n)$

Table 2.4: Edge categorization for the construction of $O(k^2)$-spanners, with respective spanner sizes and probe complexities.

The outline of the construction is given as follows. For convenience, tables of various probe complexities for computing H_{sparse} and H_{dense} are provided: Table 2.5 (page 52) and Table 2.6 (page 54), respectively.

Taking care of E_{sparse}. (Section 2.3.2.) Attempting to leverage the clustering approach, we need to partition our vertices based on their distances to S. However, some vertices can be very far from all centers: connecting them to their respective closest centers would still incur a large stretch factor. We observe that every sparse vertex v has a small k^{th}-neighborhood: $\deg_k(v, G) = |\Gamma^k(v, G)| = O(L)$ (hence the name "sparse"). Thus, we may test whether some vertex v is sparse by simply examining up to $O(L)$ vertices closest to it, using $O(\Delta L)$ probes. To take care of sparse vertices' incident edges E_{sparse}, we can then afford to identify the query edge's endpoints' k^{th}-neighborhoods and simulate a k-round distributed $(2k-1)$-spanner algorithm on the subgraph $G_{\text{sparse}} = (V, E_{\text{sparse}})$. We locally obtain our spanner H_{sparse} of G_{sparse} using $O(\Delta^2 L^2)$ probes.

Partitioning of dense vertices into Voronoi cells. (Section 2.3.3.1) In the subgraph induced by dense vertices $G_{\text{dense}} = (V_{\text{dense}}, E_{\text{dense}})$, all vertices are at distance at most k from some center. We partition them into *Voronoi cells* by connecting each of them to its closest center. We show that each dense vertex can find its shortest path to its center in $O(\Delta L)$ probes. Building on this subroutine, we straightforwardly connect vertices within each Voronoi cell to their center via these shortest paths, forming a *Voronoi tree* of depth at most k, which in turn bounds the diameter of every Voronoi cell in our spanner by $2k$. In particular, our construction improves upon the construction of [LL18] that provides a diameter bound of $O(\Delta + \log n)$. We denote by $H^{(\text{I})}_{\text{dense}}$ the set of Voronoi tree edges, as each tree spans vertices inside the same Voronoi cell.

Refining Voronoi cells into small clusters. (Section 2.3.3.2) Naturally as our next step, we would like to consider our Voronoi cells as "supervertices," and connect them via an $O(k)$-spanner with respect to this "supergraph." However, determining the connectivity in this supergraph is impossible in sub-linear probes, as a Voronoi cell may contain as many as $\Theta(n)$ vertices. To handle this issue, we define a local rule based on the subtree sizes of the Voronoi tree, which *refines* our Voronoi cells. We show that this rule partitions the dense vertices into $\widetilde{O}(n/L)$ clusters of size $O(L)$ each, such that each vertex can identify its *entire* cluster using $O(\Delta^3 L^2)$ probes.

Connecting between Voronoi cells through clusters. (Section 2.3.3.3) We then formalize local criteria for connecting Voronoi cells (through clusters), forming the set of spanner edges between clusters, $H^{(\mathrm{B})}_{\mathrm{dense}}$, using $\widetilde{O}(\Delta^4 L^2)$ probes: the union $H_{\mathrm{dense}} = H^{(\mathrm{I})}_{\mathrm{dense}} \cup H^{(\mathrm{B})}_{\mathrm{dense}}$ is the desired spanner of G_{dense}. For any omitted edge between clusters, $H^{(\mathrm{B})}_{\mathrm{dense}}$ contains a path connecting the endpoints' Voronoi cells that, w.h.p., visits only $O(k)$ other Voronoi cells along the way. Since each Voronoi cell has a $2k$-diameter spanning Voronoi tree in $H^{(\mathrm{I})}_{\mathrm{dense}}$, H_{dense} achieves the desired $O(k^2)$ stretch factor. The rules for choosing $H^{(\mathrm{B})}_{\mathrm{dense}}$ are based on marking $\widetilde{O}(n^{1/3})$ random Voronoi cells along with the clusters therein, then adding at most $\widetilde{O}(n^{1/k})$ edges per each pair of cluster and marked cluster, using a total of $\widetilde{O}(n^{1+1/k})$ edges. Sections 2.3.3.4-2.3.3.5 formalize these ideas into an efficient LCA, then show the desired properties of the constructed H_{dense} and wrap up the proof, respectively.

Reducing the required amount of independent random bits. (Section 2.4) For simplicitly, our analysis in this section uses a linear number of independent random bits. This assumption for the above construction is deferred to Section 2.4.3, where we provide an implementation using only $O(\log^2 n)$ independent random bits.

Various extensions. (Section 2.3.4) We extend our construction in two different ways. Firstly, observe that our probe complexity of $\widetilde{O}(\Delta^4 n^{2/3})$ is not sub-linear for $\Delta = \Omega(n^{1/12})$, so we create a reduction that supports a larger maximum degree at the cost of using more edges. We construct a collection of random subgraphs G_i of the input graph G that altogether covers G, then efficiently simulate the behavior of the oracle for G_i. In particular, each G_i has sufficiently small maximum degree that the spanner H_i of G_i can be computed using the developed algorithm with sub-linear probes; in addition, the union of these H_i's forms the desired spanner of G. Here, we obtain an LCA for constructing $O(k^2)$-spanners with $o(n^{4/3})$ edges (e.g., not dominated by our 5-spanner results) that has sub-linear probe complexity for $\Delta = O(n^{3/8-\epsilon})$, greatly improving upon the earlier bound of $\Omega(n^{1/12-\epsilon})$. Secondly, we also consider weighted graphs and apply a similar approach, but instead G is partitioned into subgraphs where edges within the same subgraph are of similar weights.

2.3.2 LCA for computing a $(2k-1)$-spanner H_{sparse} for E_{sparse}

Checking if a vertex is sparse or dense. We first propose a variant of the breadth-first search (BFS) algorithm that, when executed starting from a vertex v, either finds v's center or verifies that v is sparse. We justify the necessity to employ a different BFS variant from that of the prior works, namely [LRR16, LL18], as follows. In these prior works, the BFS algorithm explores *all* vertices in an entire level of the BFS tree in each step until some center is encountered, and chooses the center with the lowest ID among them. This distance tie-breaking rule via ID directly ensures that the set of vertices choosing the same center induces a connected component in G.[15]

We have earlier shown that it suffices to explore L vertices closest to a dense vertex v in order to *discover* some center. However, to *choose* v's center via the above approach, we must explore the entire last level of the BFS tree in order to apply the tie-breaking rule: this last level may

[15] If v chooses s at distance d as its center, and another vertex u is at distance $d' < d$ from s, then u must also choose s because s is the center of minimum ID in $\Gamma^d(v, G) \supset \Gamma^{d'}(u, G)$, and there are no other centers in $\Gamma^{d-1}(v, G) \supset \Gamma^{d'-1}(u, G)$.

contain as many as $\Theta(\Delta L)$ vertices. Instead, we aim to further reduce a factor of Δ from the probe complexity by designing a BFS algorithm that picks the first center it discovers as v's center: this center may not be the lowest-ID center in that level. The desired connectivity guarantee does not trivially follow under this rule, and will be further discussed in Section 2.3.3.1; for now we focus on G_{sparse}.

We provide our BFS variant as follows. Note that Q denotes a first-in first-out queue, and D denotes the set of discovered vertices. We say that the BFS algorithm *discovers* a vertex w when w is added to D.

BFS variant of a search for centers starting at vertex v
  Q.enqueue(v), D.add(v)
  while Q is not empty
    $u \leftarrow Q$.dequeue
    probe for all neighbors $\Gamma(u, G)$ of u
    for each $w \in \Gamma(u, G) \setminus D$ in the increasing order of IDs
      Q.enqueue(w), D.add(w) ▷ w is *discovered*

Figure 2-10: BFS variant for finding centers.

Denote by $D_L^k(v)$ the set of the first L vertices discovered by the BFS variant, restricting to vertices at distance at most k from v. (Equivalently speaking, if we adjust the BFS algorithm above so that it also terminates as soon as we have discovered L vertices or dequeued a vertex at distance k from v, then $D_L^k(v)$ would be the set D upon termination.) Note that $D_L^k(v, G) \subseteq \Gamma^k(v, G)$, and the containment is strict when $\deg_k(v, G) > L$.

BFS probe complexity. Recall that each vertex elects itself as a center with probability $p_{\text{center}} = (c_{\text{center}} \log n)/L$. We choose a sufficiently large constant c_{center} so that, by the hitting set argument, w.h.p., $D_L^k(v, G) \cap S \neq \emptyset$ for every v with $|D_L^k(v, G)| = L$. That is, w.h.p., every vertex v with $\deg_k(v, G) \geq L$ must be dense. Equivalently:

Observation 2.3.3. *W.h.p., for every sparse vertex v, $\deg_k(v, G) < L$.*

This observation leads to a subroutine for verifying whether a vertex v is sparse or dense based on $D_L^k(v, G)$:

Claim 2.3.4. *v is sparse if and only if both of the following holds: $|D_L^k(v, G)| < L$ and $D_L^k(v, G) \cap S = \emptyset$.*

Proof. **(Sparse)** If v is sparse ($\Gamma^k(v, G) \cap S = \emptyset$), then by Observation 2.3.3, $\deg_k(v, G) < L$, so $D_L^k(v, G) = \Gamma^k(v, G)$ and both conditions follow. **(Dense)** If v is dense ($\Gamma^k(v, G) \cap S \neq \emptyset$), we assume $|D_L^k(v, G)| < L$, then $D_L^k(v, G) = \Gamma^k(v, G)$ and hence $D_L^k(v, G) \cap S = \Gamma^k(v, G) \cap S \neq \emptyset$. □

To compute $D_L^k(v, G)$ we must discover (up to) L distinct vertices. Recall that we always probe for all neighbors of a vertex at a time. Observe that for any positive integer ℓ, among the neighbor sets of $\ell - 1$ vertices in the same connected component of size at least ℓ, at least one must necessarily contain an ℓ^{th} vertex from the component. Inductively, probing for all neighbors of $\ell - 1$ vertices

subroutine	probe complexity
determine whether v is a center	none
compute $D_L^k(v, G)$, and test whether $v \in V_{\text{sparse}}$ or $v \in V_{\text{dense}}$	$O(\Delta L)$
for $(u, v) \in E_{\text{sparse}}$, compute $\Gamma^k(u, G)$ and $\Gamma^k(v, G)$	$O(\Delta^2 L)$
test $(u, v) \in E_{\text{sparse}}$ whether $(u, v) \in H_{\text{sparse}}$	$O(\Delta^2 L^2)$

Table 2.5: Probe complexities of various subroutines used for computing H_{sparse}.

during the BFS algorithm must reveal at least ℓ vertices unless the entire component containing v is exhausted. Hence, we conclude that we only need to probe for all neighbors of $L - 1$ vertices during our BFS in order to compute $D_L^k(v, G)$, requiring $O(\Delta L)$ probes in total.

Local simulation of a distributed spanner algorithm. We construct a $(2k - 1)$-spanner $H_{\text{sparse}} \subseteq E_{\text{sparse}}$ via a local simulation of a k-round distributed algorithm for constructing spanners on the subgraph G_{sparse}. Since we also want the randomized algorithm to operate on $O(\log n)$-wise independence random bits, we will use the distributed construction of Baswana-Sen [BS07] with bounded independence [CPS17]:

Theorem 2.3.5 (From [BS07, CPS17]). *There exists a randomized k-round distributed algorithm for computing a $(2k-1)$-spanner H with $O(kn^{1+1/k})$ edges for the unweighted input graph G. More specifically, for every $(u, v) \in H$, at the end of the k-round procedure, at least one of the endpoints u or v (but not necessarily both) has chosen to include (u, v) in H. Moreover, this algorithm only requires $O(\log n)$-wise independence random bits.*

For a query edge (u, v), we first verify that at least one of u or v is sparse; otherwise we handle it later during the dense case. Without loss of generality, assume that v is sparse. To simulate the distributed algorithm on G_{sparse} for vertex v, we first learn its k^{th}-neighborhood $\Gamma^k(v, G)$, and collect all the induced edges therein. We then verify every vertex in $\Gamma^k(v, G)$ whether it is dense or sparse, so that we can determine the edges that also appear in E_{sparse}, and simulate the distributed algorithm as if it is executed on G_{sparse} accordingly.

According to the description of the distributed algorithm's behavior, for a query edge (u, v), we need to simulate this algorithm on both u and v, requiring the knowledge of both $\Gamma^k(u, G)$ and $\Gamma^k(v, G)$. Since v is sparse and $\Gamma^k(u, G) \subseteq \Gamma^{k+1}(v, G)$, we have $|\Gamma^k(u, G)| \leq |\Gamma^{k+1}(v, G)| \leq \Delta \cdot |\Gamma^k(v, G)| < \Delta L$ by Observation 2.3.3. So, we need $O(\Delta^2 L)$ NEIGHBOR probes to compute the subgraph of G induced by $\Gamma^k(u, G)$ and $\Gamma^k(v, G)$. We must also test up to $O(\Delta L)$ vertices to determine whether they are sparse or not, so our simulation process requires $O(\Delta^2 L^2)$ probes in total. We conclude the analysis of our LCA for computing H_{sparse} as the following lemma; see Table 2.5 for a summary of probe complexities.

Lemma 2.3.6 (H_{sparse} properties and probe complexity). *For any stretch factor $k \geq 1$, there exists an LCA that w.h.p., given an edge $(u, v) \in E$, decides whether $(u, v) \in H_{\text{sparse}}$ using probe complexity $O(\Delta^2 L^2)$, where H_{sparse} is a k-spanner of G_{sparse} with $O(kn^{1+1/k})$ edges.*

2.3.3 LCA for computing an $O(k^2)$-spanner H_{dense} for E_{dense}

Recall that $V_{\text{dense}} = V \setminus V_{\text{sparse}}$ is the collection of dense vertices characterized as $\Gamma^k(v, G) \cap S \neq \emptyset$, and can be verified by computing $D_L^k(v, G)$ with $O(\Delta L)$ probes. We will now take care of $E_{\text{dense}} = E(V_{\text{dense}}, V_{\text{dense}})$ by constructing an $O(k^2)$-spanner $H_{\text{dense}} \subseteq E_{\text{dense}}$ so that $H = H_{\text{sparse}} \cup H_{\text{dense}}$ becomes the desired spanner of G. To do so, we follow the general approach of Lenzen and Levi [LL18] with several keys modifications along the way. Table 2.6 keeps track of the probe complexities for various useful operations for constructing H_{dense}.

In the following, we show how to partition the dense vertices into Voronoi cells, and connect vertices in each cell via a low-depth tree structure in Section 2.3.3.1. We then show how to subdivide Voronoi cells into clusters of size $O(L)$ in Section 2.3.3.2, and discuss how we connect them into the desired spanner in Sections 2.3.3.3-2.3.3.5. We denote the set of spanner edges connecting vertices **i**nside Voronoi cells by $H^{(\text{I})}_{\text{dense}}$, and edges connecting **b**etween clusters by $H^{(\text{B})}_{\text{dense}}$, so $H_{\text{dense}} = H^{(\text{I})}_{\text{dense}} \cup H^{(\text{B})}_{\text{dense}}$.

2.3.3.1 Partitioning of dense vertices into Voronoi cells

We partition the dense vertices into $|S| = O((n \log n)/L) = O(n^{2/3} \log n)$ *Voronoi cells* with respect to centers $s_i \in S$, where each dense vertex v chooses the first center s_i that it discovers when executing the proposed BFS variant. We denote by $c(v)$ the center of v, and $\mathsf{Vor}(s)$ the Voronoi cell centered at s, consisting of all vertices that choose s as its center.

Order of vertex discovery in BFS. Clearly, the vertices are discovered in increasing distance from v. We claim that the distance ties are broken according to their lexicographically-first shortest path *from* v (with respect to vertex IDs).[16] More formally, let $\pi(v, u)$ denote the lexicographically-first shortest path from v to u in G, and $|\pi(v, u)|$ denote its length (namely $\text{dist}(v, u, G)$, the number of edges in the shortest v-u path). We claim that the BFS from v discovers u before u' if either $|\pi(v, u)| < |\pi(v, u')|$, or $|\pi(v, u)| = |\pi(v, u')|$ and $\pi(v, u) \prec \pi(v, u')$: assuming the induction hypothesis that vertices at the same distance d from v are discovered (enqueued) in this lexicographical order, we dequeue them in the same order, then enqueue the neighbors of each vertex in the order of their IDs, proving the hypothesis for distance $d + 1$.

Connectedness of each Voronoi cell on G. To prove that every $\mathsf{Vor}(s_i)$ induces a connected component in G, consider a vertex v and its shortest path $\pi(v, c(v)) = \langle v_0 = v, v_1, \ldots, v_{d-1}, v_d = c(v) \rangle$: we show that *all* vertices in this path are in $\mathsf{Vor}(s_i)$. Assume the contrary: let $u = v_i$ be the first vertex on $\pi(v, c(v))$ choosing a different center $c(u)$ via $\pi(u, c(u)) = \langle v_i = u, v'_{i+1}, \ldots, v'_{d-1}, v'_d = c(u) \rangle$; note that $|\pi(u, c(u))| = d - i + 1$ because there is no center in $\Gamma^{d-1}(v, G) \supset \Gamma^{d-i}(u, G)$. Then we have that $\langle v_i, v'_{i+1}, \ldots, v'_d \rangle \prec \langle v_i, v_{i+1}, \ldots, v_d \rangle$, yielding $\langle v_0 = v, \ldots, v_i = u, v'_{i+1}, \ldots, v'_d = c(u) \rangle \prec \pi(v, c(v))$, a contradiction.

Construction of depth-k trees spanning Voronoi cells. We straightforwardly connect each v to its center $s = c(v)$ via the edges of $\pi(v, s)$. Observe that due to the lexicographic condition, the

[16] Between paths of the same length d, $\langle v_0, \ldots, v_d \rangle \prec \langle u_0, \ldots, u_d \rangle$ if, for the minimum index i such that $v_i \neq u_i$, $\mathsf{ID}(v_i) < \mathsf{ID}(u_i)$.

subroutine	probe complexity
verify that $v \in V_{\text{dense}}$, choose $c(v)$, and compute $\pi(v, c(v))$ verify if a given edge (u, v) is a Voronoi tree edge (i.e., $(u, v) \in H^{(\text{I})}_{\text{dense}}$)	$O(\Delta L)$
compute all children of v in the Voronoi tree $T(c(v))$	$O(\Delta^2 L)$
verify whether v is heavy or light, and determine $\lvert T(v)\rvert$ when v is light	$O(\Delta^2 L^2)$
compute the entire cluster containing v	$O(\Delta^3 L^2)$
given an entire cluster A, compute $c(\partial A)$ and $E(A, \mathsf{Vor}(s))$ for any $s \in c(\partial A)$	$O(\Delta^2 L^2)$
test $(u, v) \in E_{\text{dense}}$ whether $(u, v) \in H_{\text{dense}}$	$O(p\Delta^4 L^3 \log n)$

Table 2.6: Probe complexities of various subroutines used for computing H_{dense}. This table addresses $u, v \in V_{\text{dense}}$, but these probe complexities do not assume that the LCA originally knows that u and v are dense.

vertex after v on $\pi(v, s)$ must be the vertex of the minimum ID in $\Gamma(v, G) \cap \Gamma^{|\pi(v,s)|-1}(s, G)$; that is, each vertex $v \in \mathsf{Vor}(s)$ has a fixed "next vertex" to reach s. Consequently, the union of edges in $\pi(v, s)$ for every $v \in \mathsf{Vor}(s)$ forms a *tree* rooted at s, where every level d contains vertices at distance exactly d away from s.

Due to the resulting tree structure, we henceforth refer to the constructed subgraphs spanning the Voronoi cells as *Voronoi trees*. The union of these trees forms the spanner edge set $H^{(\text{I})}_{\text{dense}}$. As our BFS variant for finding a center terminates after exploring radius k, our Voronoi trees are also of depth at most k, or diameter at most $2k$, as desired. Lastly, by augmenting our proposed BFS variant to record the BFS tree edges, we can also retrieve the Voronoi tree path $\pi(v, s)$ using $O(\Delta L)$ probes. In particular, (u, v) is a Voronoi tree edge if u is on $\pi(v, c(v))$ or v is on $\pi(u, c(u))$, implying the following lemma.

Lemma 2.3.7 ($H^{(\text{I})}_{\text{dense}}$ properties and probe complexity)**.** *There exists a partition of dense vertices $v \in V_{\text{dense}}$ into $O((n \log n)/L)$ Voronoi cells $\{\mathsf{Vor}(s)\}_{s \in S}$ according to their respective first-discovered centers $c(v)$ under the provided BFS variant. The set of edges $H^{(I)}_{\text{dense}}$, defined as the collection of lexicographically-first shortest paths $\pi(v, c(v))$, forms Voronoi trees, each of which spans its corresponding Voronoi cell and has diameter at most $2k$. Further, there exists an LCA that w.h.p., given an edge $(u, v) \in E$, decides whether $(u, v) \in H^{(I)}_{\text{dense}}$ using $O(\Delta L)$ probes.*

2.3.3.2 Refinement of the Voronoi cell partition into clusters

We now further partition the Voronoi cells into *clusters*, each of size $O(L)$. Our cluster structure is based on the construction of [LL18] but has two major differences. First, whereas in [LL18] the Voronoi cells are partitioned into $\Theta(\Delta n/L)$ clusters, in our algorithm we need the number of clusters to be independent of Δ, and more specifically bounded by $O((n \log n)/L)$. Second, unlike the clusters in [LL18] that are always connected in G, each of our clusters may not necessarily induce a connected subgraph of G; they are still connected in the spanner via $H^{(\text{I})}_{\text{dense}}$, namely by the Voronoi tree of diameter at most $2k$.

Refinement of Voronoi cells into clusters. For $s \in S$, let $T(s)$ denote the Voronoi tree spanning $\mathsf{Vor}(s)$. We extend this notation for non-centers, so that $T(v) \subseteq T(s)$ denote the subtree of $T(s)$ rooted at $v \in \mathsf{Vor}(s)$. For every $v \in \mathsf{Vor}(s)$, let $p(v)$ denote the *parent* of v in $T(s)$, and $|T(v)|$ be the number of vertices in the subtree. We define heavy and light vertices as follows.

Definition 2.3.8 (Heavy and light vertices)**.** *A dense vertex v is* heavy *if $|T(v)| > L$ and otherwise, it is* light*.*

We are now ready to define the cluster of $v \in \mathsf{Vor}(s)$ using the heavy and light classification.

(a) s **is light:** That is, the Voronoi cell containing v, $\mathsf{Vor}(s)$, contains at most L vertices. Then, all vertices in $\mathsf{Vor}(s)$ form the cluster centered at s.

(b) v **is heavy:** Then the cluster of v is the singleton cluster $\{v\}$.

(c) s **is heavy and** v **is light:** Let u be the first heavy vertex on $\pi(v, s)$, and $W = \{w : p(w) = u \text{ and } w \text{ is light}\}$ be the set of u's *light* children on the Voronoi tree. Consistently ordering the vertices $W = \{w_1, \ldots, w_\ell\}$ (e.g., according to the adjacency-list order from u), we iterate through these w_i's, grouping $T(w_i)$'s into clusters of sizes between L and $2L$; the last remaining cluster is allowed to have size strictly less than L. See Figure 2-11 for an illustration of this rule.

Clearly each cluster contains at most $2L = O(L)$ vertices, and any pair of vertices in the same cluster has a path of length at most $2k$ on $T(s)$ because they belong to the same Voronoi cell. Next, we show that the number of clusters resulting from this refinement is not asymptotically larger than the number of Voronoi cells.

Claim 2.3.9. *The number of clusters is $O((n \log n)/L) = O(n^{2/3} \log n)$.*

Proof. Recall that there are $|S| = O((n \log n)/L)$ Voronoi cells: this bounds the number of clusters of type (a). Observe that in any fixed level, among *all* Voronoi trees, there can be at most n/L heavy vertices because these heavy vertices' subtrees are disjoint. Since the Voronoi tree has depth k, there are at most kn/L heavy vertices, bounding the number of clusters of type (b).

We only subdivide the subtrees of heavy vertices into clusters, and within each such subtree, all clusters, except for at most one, have size at least L. Hence, there can be up to n/L clusters of size at least L, and kn/L clusters of smaller sizes (one for each heavy parent), establishing the bound for clusters of type (c). Thus, there are in total at most $O((n \log n)/L) + (2k+1)n/L = O((n \log n)/L)$ clusters (as we only consider $k = O(\log n)$). □

Probe complexity for identifying a vertex's cluster. Recall that via our BFS variant we can find the center s and the path $\pi(v, s)$ for a dense vertex $v \in \mathsf{Vor}(s)$ using $O(\Delta L)$ probes. We begin by establishing the probe complexity for deciding whether v is light or heavy. Observe that we can find all children of v on $T(s)$ using $O(\Delta^2 L)$ probes: run the BFS on all neighbors of v, then any w with center s such that $\pi(w, s)$ passes through v is a child of v. Using this subroutine, we traverse the subtree $T(v)$ to compute $|T(v)|$ if v is light, or stop after $L+1$ and declare that v is heavy. Since $O(L)$ vertices are investigated, the probe complexity for this process is $O(L) \cdot O(\Delta^2 L) = O(\Delta^2 L^2)$.

We can then compute v's cluster as follows. If v is heavy then we have $\{v\}$ as the cluster of type (b). Otherwise, we follow the path $\pi(v, s)$ up the Voronoi tree, one vertex at a time, and check

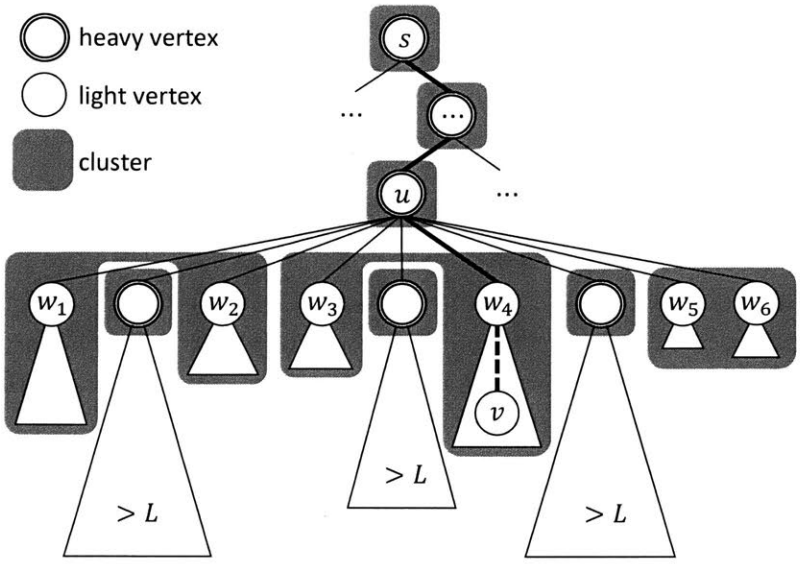


Figure 2-11: Illustration for cluster partitioning rule (c). The interesting part of the Voronoi tree $T(s)$ is shown: heavy vertices are denoted with double borderlines, and thick edges are edges of $\pi(v, s)$. Shaded areas are clusters: observe that heavy vertices form singleton clusters, while many clusters do not induce a connected subgraph of $T(s)$. In this example, u is the first heavy ancestor of v, so we compute all light children $W = \{w_1, \ldots, w_6\}$ of u, along with their subtree sizes $|T(w_i)|$'s. (For the heavy children, it suffices to only verify that they are heavy.) We group these $T(w_i)$'s into clusters of sizes in $[L, 2L]$, except possibly for the remainder cluster ($T(w_5) \cup T(w_6)$ in this case). Here, v's cluster is $T(w_3) \cup T(w_4)$.

each vertex's subtree size until we reach some heavy ancestor u of v; if there is no such u then the entire $\mathsf{Vor}(s)$ is the cluster of type (a). During this process of traversing up the Voronoi tree, we also record every computed subtree size, so that we do not need to revisit any subtree. Hence, finding the first heavy ancestor u essentially only requires visiting $O(L)$ descendants of u, which only takes $O(\Delta^2 L^2)$ probes. Once we detect u, we check each of u's children if it is light, and compute its subtree size correspondingly. Using this information, we determine all subtrees that form the cluster of type (c) containing v, as desired. This last case dominates the probe complexity: since we must check whether each of u's children is heavy or light, our algorithm require $\Delta \cdot O(\Delta^2 L^2) = O(\Delta^3 L^2)$ probes to identify v's entire cluster. The following lemma concludes the properties of the cluster partitioning of dense vertices.

Lemma 2.3.10 (Probe complexity for computing clusters)**.** *There exists a refinement of the Voronoi cell partition into $O((n \log n)/L)$ clusters of size $O(L)$ each. Further, there exists an LCA that w.h.p., given a dense vertex, compute all vertices in the cluster containing v using $O(\Delta^3 L^2)$ probes.*

2.3.3.3 Overview: connecting Voronoi cells

The supergraph intuition. To establish some intuition for connecting the Voronoi cells while maintaining a low stretch factor, let us imagine constructing an LCA for a *supergraph*, where each of the $|S| = \widetilde{O}(n/L) = \widetilde{O}(n^{2/3})$ Voronoi cells is a supervertex, and all edges between the same pair

of Voronoi cells are merged into a single superedge. Leveraging the classic clustering approach, to compute a spanner on this supergraph, we mark each supervertex independently with probability $p = n^{-1/3}$, so roughly $\widetilde{O}(pn/L) = \widetilde{O}(n^{1/3})$ supervertices are marked. These marked supervertices now act as the centers in this supergraph.

In the constructed spanner, we keep superedges between adjacent Voronoi cells according to the following three rules. Rule (1): we keep all superedges incident to a marked supervertex. There are $\widetilde{O}(n^{2/3})$ supervertices in total, and $\widetilde{O}(n^{1/3})$ supervertices are marked, contributing to $\widetilde{O}(n)$ total superedges. Rule (2): we can also keep incident superedges of supervertices without any marked neighbors: if they had more than $\widetilde{O}(n^{1/3})$ neighboring Voronoi cells, then w.h.p., one of them would have been marked. Lastly, rule (3): for each (not necessarily adjacent) pair of a supervertex a and a *marked* supervertex c, we keep a superedge from a to a *single* common neighbor $\mathsf{b}^* \in \Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$ – by consistently choosing b^* with the lowest ID, for instance. The number of added superedges is $\widetilde{O}(n)$ via the same analysis as that of rule (1).

We claim that connectivity is preserved: consider an omitted superedge (a, b). Since rule (2) does not keep (a, b), b has some marked neighbor c. By rule (3), there exists some $\mathsf{b}^* \in \Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$ with lower ID than b, such that $(\mathsf{a}, \mathsf{b}^*)$ is kept by the LCA. Recall that c is marked, so combining with rule (1), the spanner path $\langle \mathsf{a}, \mathsf{b}^*, \mathsf{c}, \mathsf{b} \rangle$ connects a and b, as desired. Thus, an LCA, given a query (a, b), keeps this superedge if there exists a supervertex $\mathsf{c} \in \Gamma(\mathsf{b})$ where b has the minimum ID among $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$, producing a 3-spanner of the supergraph with $\widetilde{O}(n)$ superedges.

However, such a supergraph-level approach cannot be implemented efficiently under the cluster refinement in the original input graph. Recall the original graph before the Voronoi cell contraction: the LCA is only given a vertex (query edge's endpoint) in the Voronoi cell, and we cannot afford to enumerate all vertices in the *entire* Voronoi cell (supervertex b) and identify *all* of its neighboring Voronoi cells (supervertex c) – finding the Voronoi cell b^* of minimum ID is outright impossible in sub-linear probes. Nonetheless, we construct an LCA based on this approach despite incomplete information of the supergraph.

Local implementation based on clusters. Employing the developed cluster refinement, as we mark a Voronoi cell, we also mark the clusters therein. We will show that the number of clusters (resp., marked clusters), do not significantly increase from the number of Voronoi cells (resp., marked Voronoi cells); hence, we may still add an edge from every cluster that is (1) marked, or (2) not adjacent to any marked clusters, to all adjacent Voronoi cells, modularly imitating the corresponding supergraph rules while still using $\widetilde{O}(n)$ edges. Nonetheless, attempting to implement rule (3) poses a problem because the LCA can only see the *clusters* containing the query edge's endpoints (while keeping the desired probe complexity). From them, we can only find out the Voronoi cells neighboring these clusters – not all Voronoi cells neighboring to the current Voronoi cell may be visible to the LCA. Due to this limitation, we cannot implement rule (3) which requires knowing *all* of b and c's neighboring Voronoi cells.

To resolve this problem, [LL18] observes that the desired connectivity is still preserved if the LCA implements a variation of rule (3) that only checks the neighboring Voronoi cells of the queried cluster in b and a canonical cluster in c. Recall that the LCA must answer "is the superedge (a, b) in the spanner?" We need to show that a and b are connected under this rule, so if the LCA keeps (a, b)

then we are done. Otherwise (a, b) is omitted, which implies that there exists a marked Voronoi cell c and a Voronoi cell $\mathsf{b}' \in \Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$ with $\mathsf{ID}(\mathsf{b}') < \mathsf{ID}(\mathsf{b})$, such that there exists a path $\langle \mathsf{b}, \mathsf{c}, \mathsf{b}' \rangle$ in the spanner thanks to rule (1). Hence, it suffices to show that a and b' are connected in the spanner. Since the supergraph contains the superedge $(\mathsf{a}, \mathsf{b}')$ (because $\mathsf{b}' \in \Gamma(\mathsf{a})$), we will inductively rely on how the LCA ensures connectivity between a and b' when it handles the query $(\mathsf{a}, \mathsf{b}')$.

So far, we have only managed to defer the original burden of proving the connectivity between a and b to the LCA's answer to the question "is the superedge $(\mathsf{a}, \mathsf{b}')$ in the spanner?" Again, even if b' indeed has the minimum ID among $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$, the LCA may not perceive this fact when it cannot see all of b''s neighboring Voronoi cells, notably c. Still, we have progress: the Voronoi cell b' in question has a *lower* ID than b. Thus we may repeat this same argument inductively on the ID of a's neighbor, which strictly decreases at each step – this argument will eventually terminate (albeit possibly in as many as $\Theta(|S|)$ steps), establishing the desired connectivity guarantee. Moreover, [LL18] enhances the LCA further by assigning random *ranks* on the Voronoi cells instead of using IDs directly, showing that a's neighbor's rank is halved at each inductive step in expectation, so the stretch of the constructed spanner (on this supergraph) is, w.h.p., $O(\log n)$.

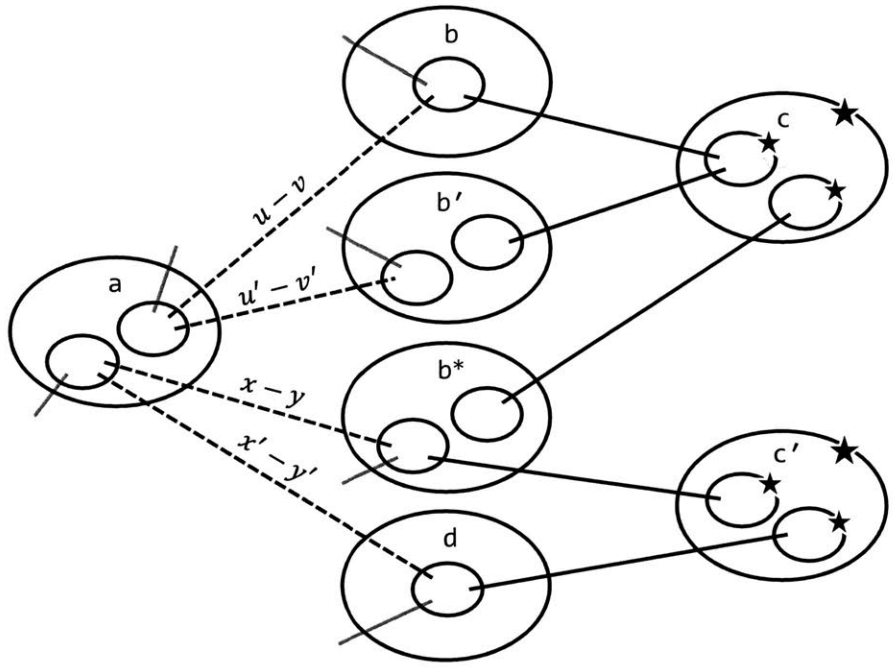


Figure 2-12: Illustration accompanying the example of clusters connection rule (3): Large and small ovals denote Voronoi cells and clusters; marked Voronoi cells and clusters therein are marked with stars. Dashed edges are query edges we consider in the example – their labels shows the names of the endpoints (vertices) inside the clusters they connect.

Illustrated example. Consider Figure 2-12. All solid edges are added by rule (1). We focus on rule (3), so to prevent an application of rule (2), we add solid grey lines to indicate that all incident clusters are adjacent to some marked Voronoi cells. Let $\mathsf{ID}(\mathsf{b}) > \mathsf{ID}(\mathsf{b}') > \mathsf{ID}(\mathsf{b}^*) > \mathsf{ID}(\mathsf{d})$. The "supergraph-level" *Voronoi cell* connection rule (3) would add (x, y) and (x', y') because b^* and d

are Voronoi cells of minimum IDs in $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$ and $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c}')$, respectively. Instead, consider now the *cluster* connection rule (3).

- Query edge (x', y'): The LCA applies the cluster connection rule (3) w.r.t. c' and keeps (x', y').
- Query edge (u, v): This edge may be omitted because the LCA finds the Voronoi cell b' also adjacent to c with lower ID than b, so rule (3) w.r.t. c does not keep this edge. The inductive argument turns to consider (u', v') (not (x, y), even if b^* actually has the lowest ID among $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$).
- Query edge (u', v'): This edge may also be omitted because the LCA cannot reach c from v' despite the fact that $\mathsf{c} \in \Gamma(\mathsf{b})$; hence it cannot apply rule (3) w.r.t. c. Note that (u', v') is engaged in another application of rule (3) w.r.t. the (undepicted) other marked endpoint of the grey edge incident to v''s cluster – (u', v') may indeed be kept by this application, and if not, the inductive argument will continue.
- Query edge (x, y): This edge is kept, but *not* because b^* is the minimum-ID Voronoi cell of $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c})$: the LCA exploring the graph from y could not have found c. Instead, it finds c', but still cannot find d. Apparently, b^* becomes the Voronoi cell of minimum ID among $\Gamma(\mathsf{a}) \cap \Gamma(\mathsf{c}')$ that it actually finds (here, the only one, in fact). Hence, the LCA applies rule (3) w.r.t. c' and keeps (x, y).

Reducing the stretch factor. Unlike the scenario of [LL18], we aim for an $O(k^2)$-spanner of size $\widetilde{O}(n^{1+1/k})$ (in the original graph); in particular, we are allowed an extra factor of $\widetilde{O}(n^{1/k})$ in the number of edges. So, between each pair of a cluster (in Voronoi cell a) and a marked cluster (in Voronoi cell c), and we add edges from the cluster in a to $\widetilde{\Theta}(n^{1+1/k})$ lowest-rank Voronoi cells b, instead of just the lowest-rank one. This adjustment reduces the ranks in the inductive argument much more rapidly: w.h.p., the argument terminates in only $O(k)$ steps, yielding an $O(k)$-spanner on this supergraph. Since each Voronoi cell has diameter at most $2k$, as we expand back our supervertices into Voronoi cells, we obtain the desired $O(k^2)$ stretch factor.

2.3.3.4 Implementation details and probe complexity analysis

Marked Voronoi cells and clusters. Recall that we randomly choose a set S of $O((n \log n)/L)$ centers, and mark each Voronoi cell *center* independently with probability $p = 1/L = n^{-1/3}$. For each marked center s_i, we also mark *all* the clusters in $\mathsf{Vor}(s_i)$. We claim that the number of marked clusters is not significantly more than the number of marked centers.

Claim 2.3.11. *The number of marked clusters is* $O((pn \log^2 n)/L) = O(n^{1/3} \log^2 n)$.

Proof. Since there are $O(\frac{n \log n}{L})$ clusters, then for any value $x > 0$, there are $O(\frac{n \log n}{xL})$ Voronoi cells with $t \in [x, 2x]$ clusters. So, we have at most $O(\frac{pn \log n}{xL})$ marked Voronoi cells with at most $2x$ clusters, yielding $O(\frac{pn \log n}{L})$ such clusters. Applying the argument for $O(\log n)$ different values of x yields $O(\frac{pn \log^2 n}{L})$ total marked clusters. □

Random ranks. We assign each center $s \in S$ an independent random *rank* $r(s) \in [0, 1)$ (e.g., a random hash function applied to their IDs): these random ranks implicitly impose a random

ordering of the centers. We sometimes refer to the rank of a Voronoi cell's center simply as the rank of that Voronoi cell. We remark that in Section 2.4.3, we will show that $\Theta(\log n)$-independence random bits suffice for our purpose of choosing centers and assigning random ranks: our algorithm can be implemented with $O(\log^2 n)$ random bits.

Adjacent clusters and Voronoi cells. The following definitions are as in [LL18]. We say that clusters A and B are *adjacent* if there exists $u \in A$ and $v \in B$ which are neighbors. In the same manner, cluster A is *adjacent* to $\mathsf{Vor}(s)$ if there exists $B \in \mathsf{Vor}(s)$ such that A and B are adjacent. For a cluster A, let $\mathsf{Vor}(A)$ denote the Voronoi cell containing A. Define the *adjacent centers* of a cluster A as $c(\partial A) = \{c(v) : \Gamma(v) \cap A \neq \emptyset\} \setminus \{c(A)\}$. Roughly speaking, this is a partial collection of neighboring Voronoi cell centers of $\mathsf{Vor}(A)$, restricted to those visible to the LCA from A.

Connecting clusters and Voronoi cells. By "connecting" two adjacent subsets of vertices A and B, we refer to the process of adding the edge of minimum ID in $E(A, B)$ to $H^{(\mathrm{B})}_{\mathrm{dense}}$, where the ID of an edge $(u, v) \in E(A, B)$ is given by $(\mathsf{ID}(u), \mathsf{ID}(v))$. The comparison is lexicographic: first compare against $\mathsf{ID}(u)$, break ties with $\mathsf{ID}(v)$.

For every marked cluster C, define the *cluster of clusters* of C, denoted by $\mathcal{C}(C)$, as the set of all clusters consisting of C and all other clusters which are adjacent to C. A cluster $B \in \mathcal{C}(C)$ is *participating* in $\mathcal{C}(C)$ if the edge of minimum ID in $E(B, \mathsf{Vor}(C))$ also belongs to $E(B, C)$. That is, if we want to connect the cluster B to a certain marked Voronoi cell by choosing the edge (u, v) of minimum ID (where $u \in B$ and v is in that Voronoi cell), then "B is participating in $\mathcal{C}(C)$" means that, C is the (unique) cluster in the Voronoi cell containing v.

Constructing $H^{(\mathrm{B})}_{\mathrm{dense}}$. Adjacent clusters are connected in H_{dense} using the following rules, where A and B denote the clusters containing the two respective endpoints of the query edges (u, v). It suffices to apply these rules when u and v belong to different Voronoi cells, $c(u) \neq c(v)$; otherwise $H^{(\mathrm{I})}_{\mathrm{dense}}$ spans them already. Note that these conditions as written are not symmetric: we must also verify them with the roles of the u and v (e.g., A and B) switched.

Global construction of $H^{(\mathrm{B})}_{\mathrm{dense}}$ for edges between clusters.

(1) Every marked cluster is connected to each of its adjacent clusters.

(2) Each cluster B that is not participating in any cluster-of-clusters (i.e., no cell adjacent to B is marked), is connected to each of its adjacent *Voronoi cells.*

(3) For each pair of cluster A and marked cluster C, consider the centers of clusters adjacent to both A and C, namely $c(\partial A) \cap c(\partial C)$. If the rank $r(s)$ of the center $s \in c(\partial A) \cap c(\partial C)$ is among the $q = \Theta(n^{1/k} \log n)$ lowest ranks of centers in $c(\partial A) \cap c(\partial C)$, then A is connected to $\mathsf{Vor}(s)$.

Figure 2-13: Procedure for the global construction of $H^{(\mathrm{B})}_{\mathrm{dense}}$.

The local algorithm and its probe complexity. We now describe the local algorithm that decides whether a query edge $(u, v) \in H^{(\mathrm{B})}_{\mathrm{dense}}$. Using the subroutines constructed so far, assume that the LCA has verified that $(u, v) \in E_{\mathrm{dense}}$, identified their centers $c(u) \neq c(v)$, and computed the entire respective clusters A and B, using $O(\Delta^3 L^2)$ probes according to Lemma 2.3.10. We then

verify the global rules of $H^{(\mathrm{B})}_{\mathsf{dense}}$ in a local fashion, answering yes indicating that $(u,v) \in H^{(\mathrm{B})}_{\mathsf{dense}}$ if any of the following condition holds.

Local construction of H_{dense} for edges between clusters.
(1) A is a marked cluster and (u,v) has the minimum edge ID amongst the edges in $E(A,B)$.
(2) B is not adjacent to any of the marked clusters, and (u,v) has the minimum edge ID among all edges in $E(B,\mathsf{Vor}(A))$.
(3) There exists a marked cluster C such that all of the following holds:
- B is participating in $\mathcal{C}(C)$,
- The rank of $c(B)$ is amongst the $q = \Theta(n^{1/k}\log n)$ lowest ranks in $c(\partial A) \cap c(\partial C)$,
- The edge (u,v) has the minimum ID among all the edges in $E(A,\mathsf{Vor}(B))$.

Figure 2-14: Procedure for the local construction of $H^{(\mathrm{B})}_{\mathsf{dense}}$.

As we have already computed the entire clusters A and B, we may verify condition (1) by checking all incident edges of A for those with the other endpoints in B. For condition (2), we compute the set $c(\partial A)$ of Voronoi cell centers $c(w)$ for neighboring vertices w of A; note that $|c(\partial A)| \leq \Delta L$. Then we check whether any of them is marked using $O(\Delta L)$ probes each. The edge of minimum ID in $E(A,\mathsf{Vor}(B))$ is among these $O(\Delta L)$ edges incident to A, allowing us to check whether $(u,v) = E(A,\mathsf{Vor}(B))$ as well. Overall condition (2) can be verified with $O(\Delta^2 L^2)$ probes.

For condition (3), we instead consider the neighboring vertices of B and compute their centers. During the process, we also keep track of the edge of minimum ID in $E(B,\mathsf{Vor}(s_i))$ of each encountered *marked* center s_i. There are up to ΔL neighboring Voronoi cells of B in total, but w.h.p., only $O(p \cdot \Delta L \cdot \log n)$ of them are marked. For each marked $\mathsf{Vor}(s_i)$, starting from the recorded endpoint in there, we compute the entire cluster C_i such that B is participating $\mathcal{C}(C_i)$ using $O(\Delta^3 L^2)$ probes. Then, we compute the centers' IDs of all neighboring vertices of C_i, namely $c(\partial C)$, spending another $O(\Delta^2 L^2)$ probes for each C_i. Combining with $c(\partial A)$ computed earlier, we can deduce if the rank of $c(B)$ is sufficiently low that $E(A,\mathsf{Vor}(B))$ must be added. In total, we require $O(p\Delta L \log n) \cdot (O(\Delta^3 L^2) + O(\Delta^2 L^2)) = O(p\Delta^4 L^3 \log n)$ probes, as desired:

Lemma 2.3.12 ($H^{(\mathrm{B})}_{\mathsf{dense}}$ probe complexity). *There exists an LCA that w.h.p., given an edge $(u,v) \in E$, decides whether $(u,v) \in H^{(B)}_{\mathsf{dense}}$ using probe complexity $O(p\Delta^4 L^3 \log n)$, where $H^{(B)}_{\mathsf{dense}}$ is as defined in Section 2.3.3.4.*

2.3.3.5 Proof of connectivity, stretch, and size analysis

Stretch and size analysis of H_{dense}. Denote by G_{Vor} the supergraph obtained from G by merging vertices within each Voronoi tree into a supervertex (e.g., by contracting $H^{(\mathrm{I})}_{\mathsf{dense}}$), and by H_{Vor} its subgraph obtained by applying the same operation in the spanner H_{dense} (e.g., the same edges as $H^{(\mathrm{B})}_{\mathsf{dense}}$ but joining corresponding supervertices instead). Since we add strictly more edges than the algorithm of [LL18] does, the connectivity follows by the exact same argument (see Lemma 4 of [LL18]); for completeness, we provide it here (with only slightly modifications). See Fig. 2-15 for an illustration.

Lemma 2.3.13 (Connectivity by $H^{(\mathrm{B})}_{\mathrm{dense}}$). *$H_{\mathsf{Vor}}$ preserves the connectivity of the Voronoi cells: if* Vor *and* Vor_0 *are connected in G_{Vor}, they remain connected in H_{Vor}.*

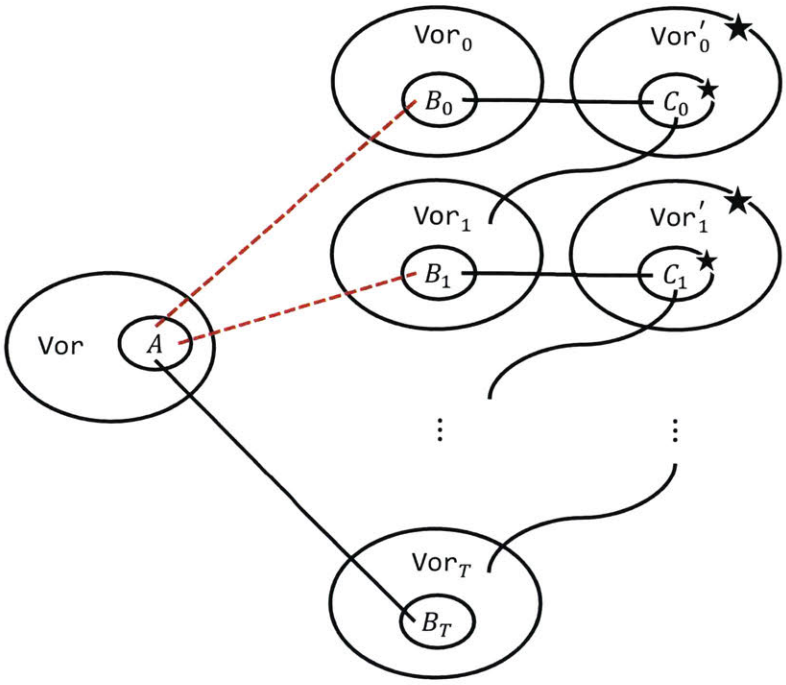


Figure 2-15: Illustration for the proof of connectivity and stretch for H_{dense}: Dashed red edges show the edges of interest at each inductive step, where the top one joining clusters A and B_0 represents the original query. Solid black edges show the path of length $2T + 1 = O(k)$ in H_{Vor} between Vor and Vor_0.

Proof. Consider clusters $A \subseteq \mathsf{Vor}$ and $B = B_0 \subseteq \mathsf{Vor}_0$ such that the edge e of minimum ID in $E(\mathsf{Vor}, \mathsf{Vor}_0)$ is in $E(A, B_0)$. If B_0 is not adjacent to any marked cell, then by condition (2) there is an edge between Vor and Vor_0 in H_{Vor}. Hence, we assume that B_0 is adjacent to a marked cell Vor'. Let $C_0 \subseteq \mathsf{Vor}'_0$ be the cluster such that B_0 is participating in $\mathcal{C}(C_0)$.

Let s_0 be the center of Vor_0. If the rank $r(s_0)$ is among the q lowest ranks of the centers $c(\partial A) \cap c(\partial C_0)$, then e is added to $H^{(\mathrm{B})}_{\mathrm{dense}}$ by condition (3). Otherwise, Vor_0 is connected to Vor'_0 in H_{Vor} as the edge of minimum ID in $E(B, C_0)$ is added to $H^{(\mathrm{B})}_{\mathrm{dense}}$ by condition (1), since $C_0 \subseteq \mathsf{Vor}'_0$ is marked. Let Vor_1 be the cell whose center has the minimum rank in $c(\partial A) \cap c(\partial C_0)$, and let $B_1 \subseteq \mathsf{Vor}_1$ be the cluster such that the edge of minimum ID in $E(A, \mathsf{Vor}_1)$ is in $E(A, B_1)$. Again by condition (1), Vor_1 is also connected to Vor'_0 in H_{Vor}.

At that point, it suffices to show that Vor is connected to Vor_1 in H_{Vor}, where the rank of Vor_1 is strictly smaller than the rank of Vor_0. We may proceed with the proof by induction, with the hypothesis that all Vor_i's are connected in H_{Vor}. Since the ranks of Vor_i's are strictly decreasing, the inductive argument halts after $T < |S|$ steps: at this point, A is connected to $B_T \subseteq \mathsf{Vor}_T$ in H_{Vor}, as desired. □

We next claim that stretch of the our spanner H_{dense} is $O(k^2)$, while [LL18] provides a stretch

factor of $O(\log n \cdot (\Delta + \log n))$. The second factor of $O(\Delta \log n)$ has been reduced down to $O(k)$ thanks to the new partitioning criteria and algorithms described so far. To remove the remaining factor of $O(\log n)$, we leverage the fact we may add a factor of $O(n^{1/k} \log n)$ more edges to the spanner H_{dense}, allowing the ranks in the inductive argument to decrease more rapidly. For simplicity, we assume now that the ranks of the centers are fully independent. In Section 2.4.3 (Theorem 2.4.5) we extend the following claim to the case where the ranks of the centers are formed by short random seed of $O(\log^2 n)$ bits.

Lemma 2.3.14 (Stretch guarantee by $H^{(\text{B})}_{\text{dense}}$). *If the ranks of the centers are assigned independently, uniformly at random from* $[0, 1)$*, then w.h.p., the stretch of* H_{Vor} *w.r.t.* G_{Vor} *is* $O(k)$.

Proof. The connectivity proof in Lemma 2.3.13 uses an inductive argument, where each step in the induction, increases the length of the path in G_{Vor} by 2. Thus it suffices to show that the induction of Lemma 2.3.13 halts, w.h.p., after $O(k)$ steps. In comparison, in Lemma 4 of [LL18], the induction uses $O(\log n)$ steps and hence the stretch in H_{Vor} is also $O(\log n)$.

Observe that while the construction of G_{Vor} heavily relies on the IDs of vertices, the rank assignment of vertices is random and independent of G_{Vor}. Again, let $A \subseteq \mathsf{Vor}, B = B_0 \subseteq \mathsf{Vor}_0$ be two adjacent clusters of interest. Folowing the argument of Lemma 2.3.13, at each step $i \geq 0$, we consider Vor_i which by the inductive hypothesis satisfies the following.

(a) A and Vor_i are adjacent.

(b) The distance between Vor_0 and Vor_i in H_{Vor} is at most $2i$.

(c) The rank of $c(\mathsf{Vor}_i)$ is the minimum rank among those of all centers in the collection $\{c(\partial A) \cap c(\partial C_j)\}_{j<i}$.

It is straightforward to verify that these conditions hold for the base case $i = 0$. For the inductive step, hypothesis (a) holds because we choose Vor_i with center $s_i \in c(\partial A) \cap c(\partial C_{i-1})$, so Vor_i is adjacent to Vor. It is also connected to the marked cluster C_{i-1}, which in turn is connected to B_{i-1} in Vor_{i-1} by rule (1), thereby proving condition (b). Lastly, condition (c) follows, because $c(\mathsf{Vor}_i)$ is the center of minimum rank in the set of centers $c(\partial A) \cap c(\partial C_{i-1})$, which contains $c(\mathsf{Vor}_{i-1})$.

It remains to show that the induction terminates after $O(k)$ steps with high probability. Let $r_i = r(c(\mathsf{Vor}_i))$. We claim that in each step, either the process terminates or, w.h.p., chooses a center of rank $r_{i+1} \leq r_i / n^{1/k}$. Suppose that the process does not terminate at step i. Observe that at this point, all ranks ever "revealed" by our algorithm so far are of the centers in condition (c): no rank lower than r_i has been encountered. Then in the beginning of step i, there are at least q cluster centers in $c(\partial A) \cap c(\partial C_i)$ whose ranks are uniformly distributed in $[0, r_i)$ (since we assume that ranks are chosen independently). For each of these $q = \Theta(n^{1/k} \log n)$ unrevealed ranks, the probability that the rank is at most $r_i / n^{1/k}$ is at least $n^{-1/k}$. By the Chernoff bound we obtain that, w.h.p., at least one of these ranks turns out to be at most $r_i / n^{1/k}$. Similarly, w.h.p., no center has rank below $\Theta(1/(n \log n))$. Thus, the algorithm terminates in $\log_{n^{1/k}}(n \log n) = \Theta(k)$ steps, as desired. □

Next, we proceed to bounding the the size of $H^{(\text{B})}_{\text{dense}}$.

Lemma 2.3.15 (Size of $H^{(\text{B})}_{\text{dense}}$). *W.h.p.,* $H^{(B)}_{\text{dense}}$ *contains* $O(\frac{pn^{2+1/k} \log^4 n}{L^2} + \frac{n \log^2 n}{pL}) = O(n^{1+1/k} \log^4 n)$ *edges.*

Proof. Recall that there are $O((n \log n)/L) = O(n^{2/3} \log n)$ clusters and $((pn \log^2 n)/L) = O(n^{1/3} \log^2 n)$ marked clusters, bounding the number of edges from condition (1) by $O((pn^2 \log^3 n)/L^2) = O(n \log^3 n)$. For condition (3), the algorithm adds $O(n^{1/k} \log n)$ edges for each such pair, which is $O((pn^{2+1/k} \log^4 n)/L^2) = O(n^{1+1/k} \log^4 n)$ edges in total. Lastly for conditon (2), every cluster that is not participating in any cluster of clusters (i.e., not adjacent to any marked Voronoi cell) w.h.p. has $O((\log n)/p) = O(n^{1/3} \log n)$ adjacent Voronoi cells, because these cells are independently marked with probability $p = n^{-1/3}$. (On the other hand, clusters are not marked independently, so in condition (2) we add one edge from A to every adjacent Voronoi cell, rather than every adjacent cluster.) Hence, the number of edges added by condition (c) is $O((n \log^2 n)/(pL)) = O(n \log^2 n)$. □

Putting everything together. Recall that our overall spanner is $H = H_{\text{sparse}} \cup H_{\text{dense}}$ where $H_{\text{dense}} = H^{(\text{I})}_{\text{dense}} \cup H^{(\text{B})}_{\text{dense}}$. Combining all results so far in this section, we achieve at our main result, Theorem 2.3.1, as follows.

Proof of Theorem 2.3.1. **(i) Size.** The size of H_{sparse}, $H^{(\text{I})}_{\text{dense}}$ and $H^{(\text{B})}_{\text{dense}}$ are $O(kn^{1+1/k})$, $O(n)$ and $O(n^{1+1/k} \log^4 n)$ due to Lemma 2.3.6, Lemma 2.3.7 (from the fact that $H^{(\text{B})}_{\text{dense}}$ is a forest), and Lemma 2.3.15, respectively. More precisely, for parameters L and p, we offer a spanner with $O(kn^{1+1/k} + \frac{pn^{2+1/k} \log^4 n}{L^2} + \frac{n \log^2 n}{pL})$ edges.

(ii) Stretch. The case of H_{sparse} taking care of E_{sparse} is immediate by Lemma 2.3.6, hence we focus on H_{dense}. The stretch argument follows by Lemma 2.3.14 for $H^{(\text{B})}_{\text{dense}}$ together with the fact that in each Voronoi cell we have a Voronoi tree of depth $O(k)$ in $H^{(\text{I})}_{\text{dense}}$ by 2.3.7. That is, between two adjacent Voronoi cells, the spanner has a path of length $O(k)$ in the Voronoi graph H_{Vor}. Within each Voronoi cell (supervertex in G_{Vor}) there exists a path of length $2k$ connecting any pair of vertices. Thus, there is a path of length $O(k^2)$ in H_{dense} between any pair of neighboring dense vertices.

(iii) Probes. The LCA can verify whether $(u, v) \in E_{\text{sparse}}$, and if so, check if $(u, v) \in H_{\text{sparse}}$ using $O(\Delta L)$ probes by Lemma 2.3.6 using $O(\Delta^2 L^2)$ total probes. Otherwise, Lemma 2.3.7 allows the LCA to verify whether u and v belongs to the same Voronoi cell, and if so, check whether $(u, v) \in H^{(\text{I})}_{\text{dense}}$ using $O(\Delta^2 L^2)$ probes. Lastly for u and v from different Voronoi cells, the LCA can check whether $(u, v) \in H^{(\text{B})}_{\text{dense}}$ using $O(p\Delta^4 L^3 \log n)$ probes via Lemma 2.3.12. Substituting $L = n^{1/3}$ and $p = 1/L$ yields the desired result. □

Theorem 2.3.1 implies that there exists an LCA with sub-linear probe complexity for any $\Delta = O(n^{1/12-\epsilon})$. In fact, we remark that by using the argument of Lemma 2.3.14, we can achieve a spanner H with $\widetilde{O}(n^{1+1/k} + nq)$ edges with stretch $O(k \log_q n) = O((k \log n)/\log q)$. As a reminder, the theorem above does not show that the LCA uses a polylogarithmic number of independent random bits. To obtain the main result for $O(k^2)$-spanner construction, we refer to Section 2.4 for missing details:

Proof of Theorem 2.1.2. The theorem follows from the proof of Theorem 2.3.1 where the independent random bits assumption are resolved according to Section 2.4. More specifically, we show the

hitting set argument under polylogarithmic number of independent random bits in Section 2.4.2, and the random ranking argument for bounding the stretch factor of H_{Vor} in Lemma 2.4.5 (Section 2.4.3). □

A summary of the differences between our algorithm and Lenzen-Levi's [LL18]. Our algorithm can be considered as an extension of [LL18] that provides a trade-off between the stretch factor and the size of the subgraph. In particular, we show that the stretch factor's dependency on Δ and n can be removed completely. We conclude by summarizing several key differences between our approaches.

- In [LL18], the distinction between dense and sparse vertices depends on a radius ℓ sampled uniformly at random from a given range that depends on Δ. In our construction, the radius is k, the stretch parameter.
- In [LL18], the sparse and dense graphs are vertex disjoint and the parameter ℓ guarantees that the number of edges between these graphs is small. In contrast, in our construction the sparse and dense graphs share vertices and in fact, these graphs are only *edge-disjoint*.
- The BFS algorithm of [LL18] for detecting a center explores an entire level of the BFS tree in each step, choosing the closest center with minimum ID. We provide a more efficient variant that explores the neighborhood of one vertex at a time, and chooses the closest center with lexicographically-first shortest path, improving the probe complexity by a factor of Δ.
- For the sparse case, [LL18] uses the distributed algorithm of Elkin and Neiman [EN17], whereas we use the algorithm of Baswana and Sen [BS07] since it has been proved to work with $O(\log n)$-wise independence [CPS17].
- For the dense case, in [LL18], the radius of the Voronoi cells is $\ell = \Theta(\Delta + \log n)$ and in our case, it is k.
- The number of clusters in [LL18] depends on ℓ and Δ. In our construction, the number of clusters is $\widetilde{O}(n^{2/3})$, each containing $O(n^{1/3})$ vertices.
- We allow $O(n^{1/k} \log n)$ edges between a cluster and neighboring clusters of a given marked clusters, whereas [LL18] only adds a single such edge.
- The algorithm of [LL18] uses random seed of size $O(\Delta \cdot n^{2/3})$. However, our algorithm only uses a poly-logarithmic number of random bits.

2.3.4 Extensions

2.3.4.1 Extension to support larger maximum degrees while maintaing sub-linear probe complexity

Our probe complexity of $\widetilde{O}(p\Delta^4 L^3) = \widetilde{O}(\Delta^4 n^{2/3})$ is only sub-linear for roughly $\Delta = O(n^{1/12-\epsilon})$, which is unfortunately quite limited. To bypass this limitation, we consider supporting an input graph G of larger maximum degree Δ, at the cost of adding more edges. Nonetheless, recall that we already have a construction for 5-spanners of general graphs with $\widetilde{O}(n^{4/3})$ edges: it is only reasonable to try to achieve a spanner with less than $\widetilde{O}(n^{4/3})$ edges for otherwise our results would be subsumed by that of 5-spanners.

A natural first attempt would be to modify the parameters L and p so that the total number of edges, $\widetilde{O}(kn^{1+1/k}+\frac{pn^{2+1/k}}{L^2}+\frac{n}{pL})$ as provided in the proof of Theorem 2.3.1, becomes $\widetilde{O}(n^{4/3})$. We unavoidably have $pL = \widetilde{\Omega}(n^{-1/3})$ due to the third term, then obtain $L = \widetilde{O}(n^{\frac{1}{9}+\frac{1}{3k}})$ by optimizing with $p = \widetilde{\Theta}(n^{-1/3}/L)$. By Lemma 2.3.12, we obtain probe complexity of $\widetilde{O}(p\Delta^4L^3) = \widetilde{O}(\Delta^4 n^{-\frac{1}{9}+\frac{2}{3k}})$, which necessarily requires $\Delta = \widetilde{O}(n^{\frac{5}{18}-\frac{1}{6k}})$ to be sub-linear. Unfortunately these graphs have $O(n^{23/18}) = o(n^{4/3})$ edges to begin with. Hence, achieving any meaningful results requires a new approach that supports at least $\Delta = \widetilde{\omega}(n^{1/3})$.

Here, we provide a different construction that bypasses the limitation above and support maximum degree Δ as large as $O(n^{3/8-\epsilon})$. In this approach, we construct a collection of t subgraphs $\{G_i\}_{i\in[t]}$ such that each edge $e \in E(G)$ appears in at least one (but not too many) G_i, so that $\cup_{i\in[t]}E(G_i) = E(G)$. The maximum degree of any G_i in our construction is w.h.p., at most some parameter $s = O(n^{1/12-\epsilon})$, so the LCA developed earlier can compute an $O(k^2)$-spanner H_i of G_i using sub-linear probes. The overall LCA can be constructed as follows: for each G_i containing the query edge e, run our LCA on that G_i; answer yes if any one of them chooses to include e in its H_i.

ALL-NEIGHBORS probes. Recall that our LCA is allowed access to the oracle that answer probes about input graph G: denote this oracle by $\mathcal{O}^G$. On the other hand, in order to simulate the developed LCA on the subgraph G_i, we must use this given oracle $\mathcal{O}^G$ to implement an oracle $\mathcal{O}^{G_i}$ for each G_i. In addition, we also need to find out, for each $e \in E(G)$, the collection G_i in which $e \in E(G_i)$.

Observe that, unlike in the constructions of 3 and 5-spanners, we have not been using important features of NEIGHBOR probes: we have the ability to specify any index i and probe for the i^{th} neighbor of a vertex v. In doing so, we would obtain not only the desired neighbor u but also the index j such that v is the j^{th} neighbor of u. Instead, in our LCA for computing $O(k^2)$-spanners, we always probe for *all* neighbors of a vertex each time, and do not make use of the returned indices. To simplify our analysis, we define the additional **ALL-NEIGHBORS probe** type that, when given a vertex $v \in V$, returns the set $\Gamma(v)$ (in an arbitrary order): this will be the only type of probe we aim to provide in $\mathcal{O}^{G_i}$. It is straightforward to verify the following observation.

Observation 2.3.16. *The LCA for computing $O(k^2)$-spanners according to Theorem 2.3.1 can be implemented with $O(p\Delta^3L^3\log n)$ ALL-NEIGHBORS probes.*

Collection of low-degree subgraphs covering G. We now provide a method for *locally* constructing a collection of low-degree subgraphs, such that the union of their edge sets is $\cup_i E(G_i) = E(G)$. Let the desired maximum degree of the subgraphs G_i be $s = O(n^{1/12-\epsilon})$, so that our $O(k^2)$-spanner construction supports these subgraphs. Let t denote the number of subgraphs G_i, and let q denote the number of NEIGHBOR probes to $\mathcal{O}^G$ needed to compute an answer to an ALL-NEIGHBORS probe of G_i (i.e., $\Gamma(v, G_i)$ for a given v). Throughout we assume that both q/s and Δ/q are $n^{\Theta(1)}$.

We define each subgraph G_i as follows (note that this construction is independent for each G_i). For each $v \in V$, we select a random subset of indices $I_v^i \subseteq [\deg(v, G)]$ of size $|I_v^i| = \min\{q, \deg(v, G)\}$. Let $R_v^i \subseteq \Gamma(v, G)$ be the set of neighbors of v whose indices in v's adjacency-list (according to $\mathcal{O}^G$) belong to I_v^i. Then, $(u, v) \in E(G_i)$ if and only if $u \in R_v^i$ and $v \in R_u^i$. That is, informally, each

vertex picks (up to) q random neighbors, and (u, v) is in G_i if u picks v and v picks u. Lastly, let $A(e) = \{i \in [t] \mid e \in E(G_i)\}$ denote the indices of G_i's where e appears.

By the following lemma, we can implement the ALL-NEIGHBORS oracles $\mathcal{O}^{G_i}$ using $q = \Theta(\sqrt{\Delta s / \log n})$ NEIGHBOR probes to $\mathcal{O}^G$, creating $t = (\Delta/s)\log^2 n$ total subgraphs.

Lemma 2.3.17. *For sufficiently large $t = \Theta((\Delta/q)^2 \log n)$, all of the following condition holds:*

(i) W.h.p., for every $e \in E(G)$, $1 \leq |A(e)| = O(\log n)$.

(ii) The answer to each ALL-NEIGHBORS *probe to G_i can be computed with q* NEIGHBOR *probes and q* ADJACENCY *probes to G.*

(iii) W.h.p., each subgraph G_i has maximum degree $s = O((q^2 \log n)/\Delta)$, assuming $q^2/\Delta = \Omega(1)$.

(iv) Each $A(e)$ can be computed using two ADJACENCY *probes to G.*

Proof. **(i) Coverage.** Since the probability that $(u, v) \in E(G_i)$ is $(q/\Delta)^2$, then $|\{i \in [t] \mid e \in E(G_i)\}| = \Theta(\log n)$ in expectation, which w.h.p., implies the desired bound.

(ii) Probe complexity. To find all neighbors of v in G_i, we probe for the j^{th} neighbor of v for every $j \in I_v^i$ with NEIGHBOR probes. Recall that an ADJACENCY probe, given $\langle u, v\rangle$, returns the index j such that v is the j^{th} neighbor of u (if $v \in \Gamma(u)$). So, for each returned neighbor u, we apply the ADJACENCY probe with parameter $\langle u, v\rangle$ to learn the index j' such that v is the j'^{th} neighbor of u. Hence, $(u, v) \in E(G_i)$ if $j' \in I_u^i$. This process requires $|I_v^i| \leq q$ NEIGHBOR probes and ADJACENCY probes.

(iii) Maximum degree. The probability that $u \in R_v^i$ and $v \in R_u^i$ are each at least q/Δ, so in expectation there are $(q/\Delta)^2 \cdot \Delta = q^2/\Delta$. The desired bound $s = O((q^2 \log n)/\Delta)$ holds w.h.p. via the Chernoff bound when $q^2/\Delta = \Omega(1)$, enforcing $s = \Omega(\log n)$.

(iv) Computing occurrences of e. We simply verify that $j \in I_u^i$ and vice versa with two probes. The LCA must to verify the conditions for all $i \in [t]$, but only needs the indices returned from these two probes. (The *running time* for this checking process is only additively $O(t)$, which is $\widetilde{O}(\Delta)$ in our application.) □

LCA results for larger maximum degree. Combining the result above with the LCA for computing $O(k^2)$-spanners, our probe complexity becomes $O(ps^3L^3 \log n) \cdot q \cdot \Theta(\log n)$: the factor of q comes from the simulation of ALL-NEIGHBORS probes, whereas the factor of $\Theta(\log n)$ reflects the fact that we may need to check whether $e \in H_i$ for as many as $\Theta(\log n)$ subgraphs G_i's. The size of the resulting spanner $H = \cup_{i \in [t]} H_i$ becomes t times that of the original size, while the stretch factor remains unchanged). Substituting the proposed values of q and t yields the following performance guarantee. Note the the probe complexity state in the lemma refers to the primitive NEIGHBOR probe type provided in our model, not the ALL-NEIGHBORS probe type defined for ease of presentation.

Lemma 2.3.18 ($O(k^2)$-spanners for larger Δ). *W.h.p., there exists an LCA for computing $O(k^2)$-spanners with $O\left(\frac{\Delta}{s}\left(kn^{1+1/k} + \frac{pn^{2+1/k}\log^4 n}{L^2} + \frac{n\log^2 n}{pL}\right)\right)$ edges, and probe complexity $O(p\Delta^{1/2}s^{7/2}L^3\log^{3/2} n)$, for any $s = \Omega(\log n)$. In particular, for the previously chosen parameters*

$L = n^{1/3}$ and $p = 1/L$, our LCA constructs $O(k^2)$-spanners with $O((\Delta/s)n^{1+1/k}\log^6 n)$ edges, and probe complexity $O(\Delta^{1/2}s^{7/2}n^{2/3}\log^{3/2} n)$.

Recall that we already have an LCA for computing 5-spanners of general graphs with $\widetilde{O}(n^{4/3})$ edges, so it is only reasonable to apply Lemma 2.3.18 while aiming for no larger number of edges. Choosing the maximum degree of the subgraphs $s = \widetilde{O}(n^{\frac{1+3\alpha-6\beta}{24}+\frac{1}{8k}})$ while keeping $L = n^{1/3}$, $p = n^{-1/3}$ yields the following result.

Corollary 2.3.19. *For positive constants $\alpha < \frac{1}{3}$, $\beta < \frac{1}{6}$ and sufficiently large k, there exists an LCA for computing $O(k^2)$-spanners for graphs of maximum degree $\Delta = O(n^{\frac{3-7\alpha-2\beta}{8}-\frac{7}{8k}})$ with $\widetilde{O}(n^{\frac{4}{3}-\alpha})$ edges using probe complexity $\widetilde{O}(n^{1-\beta})$.*

In particular, for the regime where the number of spanner edges is lower than that of 5-spanners, we can compute $O(k^2)$-spanners for graphs of maximum degree up to $\Delta = n^{3/8-\epsilon}$ given sufficiently large k while maintaining sub-linear probe complexities. This result greatly extends the restriction of the maximum degree of $n^{1/12-\epsilon}$ supported by the LCA of Theorem 2.3.1. We remark that as the original graph G contains up to $O(\Delta n)$ edges, Corollary 2.3.19 indeed provides non-trivial results when k is larger then some *constant* threshold (which may depend on constants α, β).

2.3.4.2 Extension for weighted graphs

We further remark that the "partitioning" of the edge set may be employed for weighted graphs. If the edge weights are bounded by some number W, there is a simple reduction to the unweighted setting. Namely, we apply the LCA of Theorem 2.3.1 for unweighted graphs separately for every weight scale $((1+\varepsilon)^i, (1+\varepsilon)^{i+1}]$. As a result, the stretch is increased by a factor of $(1+\varepsilon)$ and the size of the spanner by a factor of $\log_{1+\varepsilon} W$.

2.4 Bounded Independence

In this section, we show that all our LCA constructions succeed w.h.p. using $\Theta(\log n)$-*wise independent hash functions* which only require $\Theta(\log^2 n)$ random bits.

2.4.1 Preliminaries

We use the following standard notion of d-wise independent hash functions as in [Vad12]. In particular, our algorithms use the explicit construction of $\mathcal{H}$ by [Vad12], with the parameters as stated in Lemma 2.4.2.

Definition 2.4.1. *For $N, M, d \in \mathbb{N}$ such that $d \leq N$, a family of functions $\mathcal{H} = \{h : [N] \to [M]\}$ is d-wise independent if for all distinct $x_1, ..., x_d \in [N]$, the random variables $h(x_1), ..., h(x_d)$ are independent and uniformly distributed in $[M]$ when h is chosen randomly from $\mathcal{H}$.*

Lemma 2.4.2 (Corollary 3.34 in [Vad12]). *For every $\gamma, \beta, d \in \mathbb{N}$, there is a family of d-wise independent functions $\mathcal{H}_{\gamma,\beta} = \{h : \{0,1\}^\gamma \to \{0,1\}^\beta\}$ such that choosing a random function from $\mathcal{H}_{\gamma,\beta}$ takes $d \cdot \max\{\gamma, \beta\}$ random bits, and evaluating a function from $\mathcal{H}_{\gamma,\beta}$ takes time* $\mathrm{poly}(\gamma, \beta, d)$.

Then, we exploit the following result to show the concentration of d-wise independent random variables:

Fact 2.4.3 (Theorem 5(III) in [SSS95]). *If X is a sum of d-wise independent random variables, each of which is in the interval $[0,1]$ with $\mu = \mathbb{E}(X)$, then:*

- *(I) For $\delta \leq 1$ and $d \leq \lfloor \delta^2 \mu e^{-1/3} \rfloor$, it holds that $\Pr[|X - \mu| \geq \delta\mu] \leq e^{-\lfloor d/2 \rfloor}$.*
- *(II) For $\delta \geq 1$ and $d = \lceil \delta\mu \rceil$, it holds that: $\Pr[|X - \mu| \geq \delta\mu] \leq e^{-\delta\mu/3}$.*

2.4.2 Bounded independence for Section 2.2

Bounded independence for hitting set procedures. Most of our algorithms are based on the following hitting set procedure. For a given threshold $\Delta \in [1, n]$, each vertex flips a coin with probability $p = (c \log n)/\Delta$ of being head and the set of all vertices with head outcome join the set of centers S. Assuming the outcome of coin flips are fully independent, by the Chernoff bound, the followings hold w.h.p.:

(HI) There are $\Theta(pn)$ sampled vertices S.

(HII) For each vertex of degree at least Δ, the number centers among its first Δ neighbors is $\Theta(\log n)$.

Here we show that to satisfy properties (HI) and (HII), it is sufficient to assume that the outcomes of the coin flips are d-wise independent. By Lemma 2.4.2, to simulate d-wise independent coin flips for all vertices, the algorithm only requires $t = \Theta(d(\log n + \log 1/p))$ random bits: more precisely, setting $\gamma = \Theta(\log n)$ and $\beta = \log 1/p$ (for simplicity, lets assume that $\log 1/p$ is an integer), there exits a family of d-wise independent functions $\mathcal{H}$ such that a random function $h \in \mathcal{H}$[17] can be specified by a string of random bits of length t. In other words, each function $h \in \mathcal{H}$ maps the ID of each vertex to the outcome of its coin flip according to a coin with bias p. Then, from a string $\mathcal{R}$ of t random bits, the algorithm picks a function $h_{\mathcal{R}} \in \mathcal{H}$ at random to simulate the coin flips of the vertices accordingly: the outcome of the coin flip of v is head if $h_{\mathcal{R}}(\mathsf{ID}(v)) = 0$ (which happens with probability p) and the coin flips are d-wise independent. Setting $d = c \log n$ for some constant $c > 1$, we prove the following:

Claim 2.4.4. *If the coin flips are d-wise independent then properties (HI) and (HII) holds. Furthermore, the sequence of n d-wise independent coin flips can be simulated using a string of $O(\log^2 n)$ random bits.*

Proof. As described above, the sequence of n d-wise independent coin flips can be generated from a d-wise independent function $h_{\mathcal{R}} \in \mathcal{H}$ which is specified by a string of random bits $\mathcal{R}$ of size $\Theta(\log^2 n)$.

Let X_i be the event that vertex v_i is sampled into S: $v_i \in S$, if $h_{\mathcal{R}}(\mathsf{ID}(v_i)) = 0$. Since for each vertex v, $\Pr[h(\mathsf{ID}(v)) = 0] = p$, $\mathbb{E}(X) = \Theta(np) > c \log n$ where $X := \sum_{i=1}^{n} X_i$. Using Fact 2.4.3(I), w.h.p., $X = \Theta(np)$ and (HI) holds.

[17] Note that $\mathcal{H} = \{h : \{0,1\}^{\Theta(\log n)} \to \{0,1\}^{\log 1/p}\}$.

Next we show that (HII) also holds. Consider a vertex v with degree at least Δ and let $\Gamma_{\Delta,1}(v) = [u_1, \ldots, u_\Delta]$ be the first Δ neighbors of v. For each $u_j \in \Gamma_{\Delta,1}(v)$, let Y_j denote the event that $u_j \in S$ and define $Y := \sum_{j=1}^{\Delta} Y_j$. Then $\mathbb{E}(Y) = \Theta(\Delta p) = \Theta(\log n)$. Hence, by Fact 2.4.3(II), w.h.p. $Y = \Theta(\log n)$. Finally, applying the union bound over all vertices, (HII) follows. □

Construction of representatives in Section 2.2.2.3. The analysis above also extends to the process of computing Reps. Each crowded vertex chooses values $c \log n$ random indices (of its neighbor-list) in $[\Delta_{\text{med}}]$, each of which has probability $1/2$ of hitting a neighbor of degree at least Δ_{super}. Let $\{Z_i\}_{i \in [c \log n]}$ be indicators for these events and Z denote their sum, then the expected sum $\mathbb{E}(Z) \geq (c/2) \log n$. Imposing d-wise independence, Fact 2.4.3(I) implies that w.h.p., $Z > 0$, so the representative set is non-empty. We apply the union bound to show that $\mathsf{Reps}(v) \neq \emptyset$ for every $v \in V_{\text{crwd}}$, as desired.

2.4.3 Bounded independence for Section 2.3

To define the $\ell = \lceil \log n \rceil$-bit random rank $r(v)$, we will use a collection of k hash functions (where k is the stretch parameter). Letting $N = \lceil \log n / k \rceil$, each function $h_i : \{0,1\}^\ell \to \{0,1\}^N$ is an $O(\log n)$-wise independent hash function for $i \in \{1, \ldots, k\}$.

To do so, we view the rank $r(v)$ as consisting of k blocks, each with N bits. Specifically, let $r(v) = [b_1, \ldots, b_\ell] \in \{0,1\}^\ell$ and let $R_i(v) = [b_{(i-1)\cdot N}, \ldots, b_{i \cdot N - 1}]$ be the i^{th} block of N bits in $r(v)$. For every center v, define

$$R_i(v) = h_i(\mathsf{ID}(v)) \text{ and } r(v) = R_1(v) \circ R_2(v) \circ \ldots \circ R_k(v) \ .$$

The collection of these $h_1, \ldots, h_k$ functions are obtained by a uniform sampling from a family $\mathcal{H} = \{h : \{0,1\}^\ell \to \{0,1\}^N\}$ of $O(\log n)$-wise independent hash functions.

Our goal is prove Lemma 2.3.14 using these ranks instead of fully independent random ranks.

Lemma 2.4.5 (Stretch guarantee by $H^{(\text{B})}_{\text{dense}}$)**.** *If the ranks of the centers are generated according to the above construction, then w.h.p., the stretch of H_{Vor} w.r.t. G_{Vor} is $O(k)$.*

Proof. Note that G_{Vor} is independent of the rank assignment. Consider any pair of adjacent cells $\mathsf{Vor}, \mathsf{Vor}_1$ (i.e., neighbors in G_{Vor}) and let $A \subseteq \mathsf{Vor}, B \subseteq \mathsf{Vor}_1$ be two adjacent clusters of interest in these Voronoi cells.

At the beginning all vertices are unrevealed and throughout the process some of them will get revealed by exposing *one* N-size block R_j of their rank. Let $q = \lceil c \log n \cdot n^{1/k} \rceil$ for some large enough constant c, as used by our spanner construction algorithm. In each inductive step i, we either halt or we reveal the i^{th} block $R_i(v)$ in the ranks of at least q oblivious *unrevealed* centers v. At that point, we will also reveal the i^{th} block in the rank of all the centers w with $R_i(w) \neq \overline{0}$ (where $\overline{0} = [0, \ldots, 0]$).

We now describe this induction process in details. At the beginning of step $i \geq 0$, we look at $c(\mathsf{Vor}_i)$ which by induction assumption satisfies the following.

(a) A and Vor_i are adjacent.

(b) The distance between Vor_0 and Vor_i in H_{Vor} is at most $2i$.

(c) The rank of $c(\mathsf{Vor}_i)$ is the minimum rank among those of all centers in the collection $\{c(\partial A) \cap c(\partial C_j)\}_{j<i}$.

Observe that all vertices whose ranks are revealed are precisely those included in property (c). In particular, we will show property (c) as a result of two sub-properties:

(c1) The first i blocks in the rank of $c(\mathsf{Vor}_i)$ are all *zeros*.

(c2) For every center v whose rank is revealed, there is exists $j \leq i$ such that $R_j(v) \neq \overline{0}$.

For the base case, at the beginning of step i, all claims hold.

Assume that the claims hold up to the beginning of step $i \geq 1$. We will show that either we halt at that step or that all properties hold at the beginning of step $i+1$. By property (c2), each revealed center v at the beginning of step i has at least one non-zero block among the first i blocks of $r(v)$. Or, in other words, the first i blocks in the ranks of all the unrevealed vertices at the beginning of step i, are *all-zeros*.

We may assume that there is a marked cluster C_i such that B_i (the cluster in Vor_i such that the edge of minimum ID in $E(A, \mathsf{Vor}_i)$ is in $E(A, B_i)$) participates in $\mathcal{C}(C_i)$ (as otherwise, we halt). If there are less than q unrevealed centers in $c(\partial A) \cap c(\partial C_i)$, then the process terminates: by property (c), all revealed centers have a strictly larger rank than $c(\mathsf{Vor}_i)$. Otherwise, (i.e., there are at least q unrevealed centers in $c(\partial A) \cap c(\partial C_i)$), we probe the i^{th} block (using the hash function h_i) in the rank of these q unrevealed centers in $c(\partial A) \cap c(\partial C_i)$. We let Vor_{i+1} be a cell with a center $s_{i+1} = c(\mathsf{Vor}_{i+1})$ satisfying that $s_{i+1} \in c(\partial A) \cap c(\partial C_i)$ and $R_i(s_{i+1}) = \overline{0}$. If there are several such centers that satisfy these two conditions, we pick one arbitrarily. We now claim:

Claim 2.4.6. *W.h.p., there exists at least one $s_{i+1} \in c(\partial A) \cap c(\partial C_i)$ such that $R_i(s_{i+1}) = \overline{0}$.*

Proof. Let S' a subset of q unrevealed centers in $c(\partial A) \cap c(\partial C_i)$. For every $s_j \in S'$, let $X_j \in \{0, 1\}$ be the event that $R_i(s_j) = \overline{0}$. Since $R_i(s_j) = h_i(\mathsf{ID}(s_j))$, we have that $\mathbb{E}(X_j) = 1/2^N$ and $\mathbb{E}(X) = q/2^N = \Theta(\log n)$ where $X = \sum_{j=1}^{q} X_j$. Since the X_j variables are $O(\log n)$-independent, using the Chernoff bound from Fact 2.4.3(I), we obtain that w.h.p. $X \geq 1$ and hence there exists $s_j \in S'$ that satisfies the above. The claim follows. □

The proofs of the first two properties remain unchanged. Property (3a) holds by induction and by the selection of Vor_{i+1}. In particular, by induction, all the first i blocks of the rank $r(s_{i+1})$ are all zeros (as s_{i+1} is unrevealed at the beginning of step i) and we select s_{i+1} since $R_i(s_{i+1}) = \overline{0}$. Property (c2) holds by induction and by the fact that the i^{th}-block in the ranks of all those centers that got revealed in step i is *nonzero*. By combining (c1) and (c2), property (c) holds as well since s_{i+1} has the minimum rank among all those that got revealed so far.

Finally, we claim that w.h.p., the process terminates after $O(k)$ induction steps. We will show that by claiming that in every step i, at least a $(1 - c' \cdot n^{-1/k})$ fraction of the remaining unrevealed centers are revealed for some constant $c' > 0$. Let U_i be the number of unrevealed centers at the beginning of step i. Hence, $U_1 = n$. If we did not halt at step i, it means that $U_i \geq q = \Omega(\log n \cdot n^{1/k})$. We now bound the number UZ_i of unrevealed centers at the beginning of step i whose i^{th} block is all-zero. The probability of having an all-zero block for a single center is $1/2^N$ and hence in expectation there are $U_i/2^N$ such centers. Since $U_i \geq q$, and since the ranks are $O(\log n)$-wise independent, using Chernoff bound of Fact 2.4.3(I), with get that w.h.p. $UZ_i \in [c_1 \cdot U_i/2^N, c_2 \cdot U_i/2^N]$ for some

constants $0 < c_1 < c_2$. Hence, w.h.p., $U_{i+1} = U_i - UZ_i \geq (1 - c'/2^N)U_i$. Overall, after $O(k)$ induction steps, there are at most q unrevealed vertices and at that point we halt. The lemma follows. □

2.5 Lower Bounds

In this section, we establish lower bounds for the problem of locally constructing a spanner consisting of an asymptotically sub-linear number of edges from the input graph. Our results largely follows from the analysis of [KKR04] on the lower bound construction of [LRR14]; a compact version of this proof is given here for completion.

For simplicity, we assume that each vertex occupies a unique ID from $\{1, \ldots, n\}$; this assumption may only strengthen our lower bound. We define an *instance* of a d-regular graph on n vertices as a perfect matching between cells of a table of size $n \times d$: a matching between the cells (u, i) and (v, j) indicates that v is the i^{th} neighbor of u and u is the j^{th} neighbor of v. An edge can be then expressed as a quadruple (u, i, v, j); note that the endpoints are always interchangeable. For consistency with this notation, we let the NEIGHBOR probe with parameter $\langle u, i \rangle$ for $i \leq \deg(u)$ return (v, j) (instead of only v) – this change can only provide more information to the algorithm. We say that an instance G and the edge (u, i, v, j) are *compatible* if G contains (u, i, v, j). Our lower bounds are established for sufficiently large $n \equiv 2 \bmod 4$ and odd integer d.

The overall argument. First, we construct two distributions $\mathcal{D}^+_{(x,a,y,b)}$ and $\mathcal{D}^-_{(x,a,y,b)}$ over undirected d-regular graph instances for $x, y \in V$ and $a, b \in [d]$. Any graph instance G^+ in the support of $\mathcal{D}^+_{(x,a,y,b)}$ contains the edge (x, a, y, b) such that with high probability, removing this edge does not disconnect x and y. In particular, $\mathcal{D}^+_{(x,a,y,b)}$ is the uniform distribution over all instances compatible with (x, a, y, b). On the other hand, any graph instance G^- in the support of $\mathcal{D}^-_{(x,a,y,b)}$ contains the edge (x, a, y, b) such that removing this edge disconnects x and y (leaving them in separate connected components).

We show that when given the query (x, a, y, b), any *deterministic* LCA ALG that only makes $o(\min\{\sqrt{n}, \frac{n}{d}\})$ probes can only distinguish whether the underlying graph is a graph randomly drawn from $\mathcal{D}^+_{(x,a,y,b)}$ or $\mathcal{D}^-_{(x,a,y,b)}$ with probability $o(1)$. We prove this claim by defining two processes $\mathcal{P}^+_{(x,a,y,b)}$ and $\mathcal{P}^-_{(x,a,y,b)}$ which interact with ALG and generate a random subgraph from $\mathcal{D}^+_{(x,a,y,b)}$ and $\mathcal{D}^-_{(x,a,y,b)}$ respectively. We then argue that for each probe the answers that these two processes return are nearly identically distributed, and so are their *probe-answer histories.*

Aiming for an overall success probability of 2/3, ALG must keep the edge (x, a, y, b) in its spanner with probability $\frac{2}{3}(1 - o(1)) > 1/2$. Since an instance in $\mathcal{D}^+_{(x,a,y,b)}$ is chosen uniformly at random, then for more than half of the instances in the support of $\mathcal{D}^+_{(x,a,y,b)}$, which are exactly the instances compatible with (x, a, y, b), ALG returns yes on query (x, a, y, b). Applying this argument for all possible edges (quadruples (x, a, y, b)), we obtain that ALG returns yes on at least half of all compatible instance-query pairs. Consequently, over the uniform distribution over all instances, in expectation any deterministic algorithm ALG must return yes on more than $m/2$ edges. Employing Yao's principle, we conclude that any (randomized) LCA cannot compute a spanning subgraph with $o(m)$ edges using $o(\min\{\sqrt{n}, n/d\})$ probes.

2.5.1 Analysis of the probe-answer histories

Similarly to the work of [LRR14]. we construct our distributions as follow.

- **Distribution $\mathcal{D}^+_{(x,a,y,b)}$.** $\mathcal{D}^+_{(x,a,y,b)}$ is a uniform distribution over all d-regular graph instances, conditioned that (x, a, y, b) is in the instance. More precisely, the edges of G in the family is determined by the following process. Consider a two-dimensional table of size $n \times d$ which is called *matching table* and is denoted by M. Any perfect matching between cells in this table corresponds to a graph in $\mathcal{D}^+_{(x,a,y,b)}$. Note that the generated graphs are not necessarily simple.

- **Distribution $\mathcal{D}^-_{(x,a,y,b)}$.** Let $V = V_0 \uplus V_1$ be a *random* partition of the vertex set into two equal sets such that $x \in S$ and $y \in T$. Now consider two matching tables of each of size $n/2 \times d$ denoted by M_1 and M_2. For a graph G in this family, besides the edge (x, a, y, b), the rest of edges are determined by choosing a random perfect matching *within* each of M_1 and M_2 (over the remaining cells). Thus, (x, a, y, b) is the only edge connecting between M_1 and M_2.

For brevity we drop the subscript (x, a, y, b) for now as it is clear from the context. For sufficiently large values of $d = \Omega(1)$, w.h.p, each instance G from $\mathcal{D}^+$ is connected even when (x, y) is removed (see e.g., [Bol01]). On the other hand, removing (x, y) from any $G^- \in \mathcal{D}^-$ clearly disconnects x and y. Thus, unless a *deterministic* algorithm ALG can determine whether it is given (x, a, y, b) of an instance from $\mathcal{D}^+$ or $\mathcal{D}^-$, it must return yes on (x, a, y, b) for a (2/3)-fraction of these instances. For simplicity we assume that ALG has a knowledge of the construction (including the degree d), and never makes a probe that does not reveal any new information.

Let $\mathcal{L}$ denote the number of probes made by the algorithm, and Q denote the set of probes performed by ALG. Observe that ALG is a deterministic mapping from the probe-answer histories $\langle (q_1, a_1), \cdots, (q_t, a_t) \rangle \mapsto q_{t+1}$ for $t < \mathcal{L}$ and to $\{\mathsf{yes}, \mathsf{no}\}$ for $t = \mathcal{L}$. Each probe q_i is either a NEIGHBOR probe or an ADJACENCY probe.

Next, similarly to [KKR04], we define two processes $\mathcal{P}^+$ and $\mathcal{P}^-$ which interact with an arbitrary algorithm ALG and respectively construct a random graph from $\mathcal{D}^+$ and $\mathcal{D}^-$. Defining D^+_t and D^-_t to be the distribution of the probe-answer histories of the interaction of $\mathcal{P}^+$ and $\mathcal{P}^-$ respectively with ALG after t probes, we show that if $\mathcal{L} = o(\min\{\sqrt{n}, n/d\})$, then the statistical distance of $D^+_{\mathcal{L}}$ and $D^-_{\mathcal{L}}$ is $o(1)$. We now give the formal description of $\mathcal{P}^s$ for $s \in \{+, -\}$:

- Let R^s be the set of all graphs in the support of $\mathcal{D}^s$. Let $R^s_{(u,v)}$ and $R^s_{(u,i,v,j)}$ be the set of all graphs in the support of $\mathcal{D}^s$ that are compatible (u, v) and (u, i, v, j) respectively. In the former case, we require at least one matching (u, i', v, j') for some $i', j' \in [d]$ between cells in the rows of u and v in the matching table; however, in the latter case, we only allow a fixed matching (u, i, v, j). We also write $R^s_{\overline{(u,v)}}$ to denote the set of all graphs in the support of $\mathcal{D}^s$ that are not compatible with u, v.
- Starting from $R^s_0 = R^s_{(x,a,y,b)}$, for any $t > 0$, R^s_t denotes the set of all graphs in the support of $\mathcal{D}^s$ that are compatible with the first t probes and answers.
  - If q_t is an ADJACENCY probe of the form $\langle u_t, v_t \rangle$: We choose whether to add an edge between u and v with probability $|R^s_{(u_t,v_t)} \cap R^s_{t-1}|/|R^s_{t-1}|$. If so, we match a

pair of cells between the rows of u and v: sample $(i_t, i_t) = (i, j)$ with probability $|R^s_{(u_t,i_t,v_t,j_t)} \cap R^s_{t-1}|/|R^s_{t-1}|$ set $R^s_t = R^s_{(u_t,i_t,v_t,j_t)} \cap R^s_{t-1}$, and answer $a_t = i_t$. Otherwise, we simply set $R^s_t = R^s_{\overline{(u_t,v_t)}} \cap R^s_{t-1}$ and answer $a_t = \perp$.

- If q_t is a NEIGHBOR probe of the form $\langle u_t, i_t \rangle$: For each $v \in V$ and $j_t \in \{1, \ldots, d\}$, we choose a cell to match with (u_t, i_t): sample the answer $a_t = (v_t, j_t)$ with probability $|R^s_{(u_t,i_t,v_t,j_t)} \cap R^s_{t-1}|/|R^s_{t-1}|$ and set $R^s_t = R^s_{(u_t,i_t,v_t,j_t)} \cap R^s_{t-1}$.

- After $\mathcal{L}$ probes, return a random graph uniformly sampled from $R^s_{\mathcal{L}}$.

Lemma 2.5.1 (Lemma 10 in [KKR04]). *For any deterministic algorithm* ALG, *the process* $\mathcal{P}^s$ $(s \in \{+, -\})$ *when interacting with* ALG, *uniformly generates a graph from the support of* $\mathcal{D}^s_{(x,a,y,b)}$.

Next, we show that the probability that ALG can detect an edge with ADJACENCY probe (asking probe $q = (u, v)$ for which the answer is positive; an edge exists between u and v) after performing only $o(n/d)$ is small. We can define R^s_t, the set of all graphs in $\mathcal{D}^s$ as $R^s_{B,\overline{D}}$ where B is the set of edges that the graphs in R^s_t must contain (namely, all pairs of cells (u, i, v, j) created in some previous probes) and D is the set of edges that the graphs in R^s_t must not contain (namely, all pairs (u, v) disallowed by ADJACENCY probes with negative answer).

Assuming that the algorithm makes $\mathcal{L} = o(n/d)$ probes, we establish the following lemmas that will be useful in bounding the difference between the distributions of probe-answer histories generated by the two processes. In particular, assume the number of conditions $|B|, |D| = o(n/d)$, and the initial conditions $(x, a, y, b) \in B$ and $(x, y) \notin D$, in the following three lemmas.

Lemma 2.5.2. *For every* $(u, i, v, j) \neq (x, a, y, b)$, $\frac{|R^s_{(u,i,v,j)} \cap R^s_{B,\overline{D}}|}{|R^s_{B,\overline{D}}|} = O(\frac{1}{nd})$.

Proof. For process $\mathcal{P}^+$, the proof is the same as the proof of the similar statement in Lemma 11 of [KKR04]. Here, we show that the argument holds for $\mathcal{P}^-$.

$$\frac{|R^-_{(u,i,v,j)} \cap R^-_{B,\overline{D}}|}{|R^-_{B,\overline{D}}|} = \frac{|R^-_{(u,i,v,j)} \cap R^-_B|}{|R^-_B|} \cdot \frac{|R^-_{(u,i,v,j)} \cap R^-_{B,\overline{D}}|}{|R^-_{(u,i,v,j)} \cap R^-_B|} \cdot \frac{|R^-_B|}{|R^-_{B,\overline{D}}|}$$
$$\leq \frac{1}{\Omega(nd)} \cdot 1 \cdot O(1) = O\left(\frac{1}{nd}\right),$$

where the bounds on $\frac{|R^-_{(u,i,v,j)} \cap R^-_B|}{|R^-_B|}$ and $\frac{|R^-_B|}{|R^-_{B,\overline{D}}|}$ are shown in Claim 2.5.3 and 2.5.4. □

Claim 2.5.3. *For every* $(u, i, v, j) \neq (x, a, y, b)$, $\frac{|R^-_{(u,i,v,j)} \cap R^-_B|}{|R^-_B|} \leq \frac{2}{nd}$.

Proof. If u and v belong to different partitions or at least one of them is already matched in B, then $R^-_{(u,i,v,j)} = \emptyset$. Otherwise, let w denote the lower bound on the number of unmatched cells in the matching table containing rows of u and v in any instance of R^-_B. Recall that the number of matched cells is bounded by $o(nd)$, so $w \geq nd - 2|B| - 1 \geq (1 - o(1)) \cdot nd$. The probability that cells (u, i) is matched to (v, j) is given by

$$\frac{|R^-_{(u,i,v,j)} \cap R^-_B|}{|R^-_B|} = \frac{1}{w - 1} \leq \frac{2}{nd}$$

for sufficiently large n and d. □

Claim 2.5.4. $\frac{|R_B^-|}{|R_{B,\overline{D}}^-|} = O(1)$.

Proof. As we consider $(u,v) \neq (x,y)$, we have

$$\frac{|R_{(u,v)}^- \cap R_B^-|}{|R_B^-|} = \frac{|(\bigcup_{i,j\in[d]} R_{(u,i,v,j)}^-) \cap R_B^-|}{|R_B^-|} \leq \frac{\sum_{i,j\in[d]} |R_{(u,i,v,j)}^- \cap R_B^-|}{|R_B^-|} \leq d^2 \cdot \frac{2}{nd} = O\left(\frac{d}{n}\right).$$

Then by the union bound,

$$\frac{|R_{B,\overline{D}}^-|}{|R_B^-|} = \frac{|R_B^- \cap (\bigcap_{r\leq|D|} R_{\overline{e_r}}^-)|}{|R_{B,\overline{D}}^-|} = \frac{|R_B^- \setminus (\bigcup_{r\leq|D|} R_{e_r}^-)|}{|R_{B,\overline{D}}^-|} \geq 1 - \sum_{r\in[|D|]} \frac{|R_{e_r}^- \cap R_B^-|}{|R_B^-|}$$
$$= 1 - o\left(\frac{n}{d}\right) \cdot O\left(\frac{d}{n}\right) = 1 - o(1).$$

Hence, $\frac{|R_B^-|}{|R_{B,\overline{D}}^-|} = O(1)$. □

Recall again the assumption that ALG does not make probes that do not reveal any new information about the instance. Now, we are ready to formally prove the following claim on the ADJACENCY probes, that with $\mathcal{L} = o(n/d)$ probes, ALG is unlikely to obtain any positive answer.

Lemma 2.5.5. *Let* ALG *be an arbitrary deterministic algorithm interacting with process* $\mathcal{P}^s$ ($s \in \{+,-\}$) *and that has probed* $o(n/d)$ *times. The probability that* ALG *detects an edge with an* ADJACENCY *probe* of the form $\langle u_t, v_t\rangle$ *during the interaction is* $o(1)$.

Proof. Consider an arbitrary step t in the interaction of ALG and $\mathcal{P}^s$ in which the algorithm performs an ADJACENCY probe. Since, $t = o(n/d)$, by the description of $\mathcal{P}^s$ and applying Lemma 2.5.2, the probability that the answer to q_t is not $\perp$ is bounded by:

$$\frac{|R_{(u_t,v_t)}^- \cap R_{t-1}^-|}{|R_{t-1}^-|} \leq \frac{\sum_{i_t,j_t\in[d]} |R_{(u_t,i_t,v_t,j_t)}^- \cap R_{t-1}^-|}{|R_{t-1}^-|} \leq d^2 \cdot O\left(\frac{1}{nd}\right) = O\left(\frac{d}{n}\right).$$

Since the total number of probes is $o(d/n)$, by the union bound, the probability that ALG detects an edge with an ADJACENCY probe during its interaction with $\mathcal{P}^s$ is $o(1)$. □

Next, we similarly show that if $\mathcal{L} = o(\sqrt{n})$, ALG is likely to obtain a new vertex from every NEIGHBOR probe it performs.

Lemma 2.5.6. *Let* ALG *be an arbitrary deterministic algorithm interacting with process* $\mathcal{P}^s$ ($s \in \{+,-\}$) *and that has probed* $o(\sqrt{n})$ *times. With probability* $1 - o(1)$, *all* NEIGHBOR *probes of* ALG *receive distinct vertices in their answers.*

Proof. Consider step t in the interaction of ALG and $\mathcal{P}^s$ and let V_{t-1} denote the set of vertices seen by ALG so far (i.e., participate in some $q_{t'}$ or $a_{t'}$ where $t' \leq t-1$); thus $|V_{t-1}| \leq 2t$. In what follows

we bound the probability p_t that a_t (the answer to of the form (u_t, i_t)) corresponds to a vertex v which belong to V_{t-1}.

$$p_t = \frac{|\bigcup_{v \in V_{t-1}, j \in [d]} (R^s_{(u_t,i_t,v,j)} \cap R^s_{t-1})|}{|R^s_{t-1}|}$$
$$\leq \sum_{v \in V_{t-1}, j \in [d]} \frac{|(R^s_{(u_t,i_t,v,j)} \cap R^s_{t-1})|}{|R^s_{t-1}|} \leq 2t \cdot d \cdot O\left(\frac{1}{nd}\right) = O\left(\frac{1}{\sqrt{n}}\right).$$

where the last inequality is implied by Lemma 2.5.2. Hence, if the total number of probes is $o(\sqrt{n})$, with probability $1 - o(1)$ the answer to every ADJACENCY probe introduces a new vertex □

Next, we prove the main result of this section. Lets D^s_t denotes the distribution over the probe-answer histories of t rounds of the interaction of ALG and $\mathcal{P}^s$.

Lemma 2.5.7. *For any arbitrary deterministic* ALG *and* $\mathcal{L} = o(\min\{\sqrt{n}, n/d\})$, *the statistical distance between* $D^+_{\mathcal{L}}$ *and* $D^-_{\mathcal{L}}$ *is* $o(1)$.

Proof. Let Π be the set of all valid probe-answer histories of length $\mathcal{L}$ and let $\Pi' \subset \Pi$ denote the set of all histories in which every ADJACENCY probe returns $\perp$ and no NEIGHBOR probe returns an already-discovered vertex.

Observe that conditioned on $\pi \in \Pi'$, the answers to all ADJACENCY probes by both $\mathcal{P}^-$ and $\mathcal{P}^+$ are $\perp$. Moreover, the answers to each NEIGHBOR probe by both processes are chosen uniformly at random among all cells from the rows corresponding to the set of all vertices not visited so far, which is the same for the both processes. That is, $D^+_{\mathcal{L}}(\pi)$ and $D^-_{\mathcal{L}}(\pi)$ are proportional to each other for every probe-answer history $\pi \in \Pi$. Hence the difference between the probe-answer histories for $\pi \in \Pi'$ in both processes are bounded simply by the difference in their total probabilities:

$$\sum_{\pi \in \Pi'} |D^+_{\mathcal{L}}(\pi) - D^-_{\mathcal{L}}(\pi)| = \left| \sum_{\pi \in \Pi'} D^+_{\mathcal{L}}(\pi) - \sum_{\pi \in \Pi'} D^-_{\mathcal{L}}(\pi) \right| = \left| \sum_{\pi \in \Pi \setminus \Pi'} D^+_{\mathcal{L}}(\pi) - \sum_{\pi \in \Pi \setminus \Pi'} D^-_{\mathcal{L}}(\pi) \right|.$$

Putting everything together, we bound the difference between the distributions of probe-answer histories when $\mathcal{L} = \min\{\sqrt{n}, n/d\}$:

$$\sum_{\pi \in \Pi} |D^+_{\mathcal{L}}(\pi) - D^-_{\mathcal{L}}(\pi)| = \sum_{\pi \in \Pi'} |D^+_{\mathcal{L}}(\pi) - D^-_{\mathcal{L}}(\pi)| + \sum_{\pi \in \Pi \setminus \Pi'} |D^+_{\mathcal{L}}(\pi) - D^-_{\mathcal{L}}(\pi)|$$
$$\leq 2 \left| \sum_{\pi \in \Pi \setminus \Pi'} D^+_{\mathcal{L}}(\pi) - \sum_{\pi \in \Pi \setminus \Pi'} D^-_{\mathcal{L}}(\pi) \right|$$
$$\leq 2 \left| \sum_{\pi \in \Pi \setminus \Pi'} D^+_{\mathcal{L}}(\pi) \right| + 2 \left| \sum_{\pi \in \Pi \setminus \Pi'} D^-_{\mathcal{L}}(\pi) \right| = o(1).$$

where the last equation follows as a result of Lemma 2.5.5 and Lemma 2.5.6. □

2.5.2 Establishing the lower bound

Here we show the main lower bound result using the outlined argument. Moreover, our construction so far allows parallel edges and self-loops which are not handled by our upper bounds; we handle this issue and establish the lower bound even for simple graphs, restated below.

Theorem 2.1.3 (Lower Bound). *Any local randomized LCA that computes, with success probability at least 2/3, a spanner of the simple m-edge input graph G with $o(m)$ edges, has probe complexity $\Omega(\min\{\sqrt{n}, n^2/m\})$.*

Proof. As outlined earlier, for the $1 - o(1)$ fraction of the instances in $\mathcal{D}^+_{(x,a,y,b)}$, a deterministic ALG must keep the edge (x, a, y, b) in its spanner with probability $\frac{2}{3}(1 - o(1)) > 1/2$ because, due to Lemma 2.5.7, with probability $1 - o(1)$ it cannot distinguish whether the given instance is from $\mathcal{D}^+_{(x,a,y,b)}$ or $\mathcal{D}^-_{(x,a,y,b)}$. Since $\mathcal{D}^+_{(x,a,y,b)}$ is the uniform distribution over instances compatible with (x, a, y, b), then for more than half of these instances, ALG returns yes on query (x, a, y, b). Applying this argument for all (x, a, y, b), we obtain that ALG returns yes on at least half of all compatible instance-query pairs. Nonetheless, our generated instances in $\mathcal{D}^+_{(x,a,y,b)}$, $\mathcal{D}^-_{(x,a,y,b)}$ often contains parallel edges and self-loops.

In order to remove these non-simple graphs, as similarly noted in [KKR04], we observe that our constructed graphs only have $O(d^2)$ parallel edges and $O(d)$ self-loops in expectation. Thus, we may simply fix each instance by modifying $O(d^2)$ matchings so that all instances become simple (assuming sufficiently large n and d). Observe that by doing so, the connectivity of the graph strictly increases: the required condition of $\mathcal{D}^+_{(x,a,y,b)}$ that x and y must be connected even when (x, a, y, b) is absent is still upheld. Similarly, for $\mathcal{D}^-_{(x,a,y,b)}$ the modifications must still respect the restriction that no edge other than (x, a, y, b) has endpoints on different tables, so that removing (x, a, y, b) disconnects them.

Due to similarly arguments as Lemma 2.5.5 and Lemma 2.5.6, the probability that ALG detects these modifications are $o(1)$, and therefore Lemma 2.5.7 still holds under these changes, as long as the query to ALG itself is not one of the modified edges. On the other hand, if a modified edge is given as a query to ALG, then we do not assume anything about the algorithm's answer for this edge. As the modified edges constitute a fraction of up to $O(d^2)/nd = O(d/n)$ of the total number of edges on each instance on average, the fraction of instance-query pairs where ALG answers yes can be potentially reduced by at most a fraction of $O(d/n)$: this still leaves a fraction of $\frac{2}{3}(1 - o(1)) - O(\frac{d}{n}) > 1/2$ for sufficiently small $d = O(n)$. That is, even when restricted to simple graphs, we still obtain that ALG returns yes on at least half of all compatible instance-query pairs

Over the uniform distribution over all instances, in expectation any deterministic algorithm ALG must return yes on more than $m/2$ edges. Employing Yao's principle, we conclude that any (randomized) LCA cannot compute a spanning subgraph with $o(m)$ edges with success probability 2/3 using $o(\min\{\sqrt{n}, n/d\})$ probes. Substituting $d = 2m/n$ yields the desired bound. □

Chapter 3

Set Cover

3.1 Overview of Set Cover in the Oracle Access Model

Set Cover is a classic combinatorial optimization problem, in which we are given a set (universe) of n elements $\mathcal{U} = \{e_1, \cdots, e_n\}$ and a collection of m sets $\mathcal{F} = \{S_1, \cdots, S_m\}$. The goal is to find a *set cover* of $\mathcal{U}$, i.e., a collection of sets in $\mathcal{F}$ whose union is $\mathcal{U}$, of minimum size. Set Cover is a well-studied problem with applications in operations research [GW97], information retrieval and data mining [SG09], learning theory [KV94], web host analysis [CKT10], and many others. Recently, this problem and other related coverage problems have gained a lot of attention in the context of massive data sets, e.g., streaming model [SG09, ER16, DIMV14, HIMV16, CW16, AKL16, MV17, Ass17, BEM17, IMR+17] or map reduce model [KMVV15, MZ15, BEM16].

Although the problem of finding an optimal solution is **NP**-complete, a natural greedy algorithm which iteratively picks the "best" remaining set (the set that covers the most number of uncovered elements) is widely used. The algorithm finds a solution of size at most $k \ln n$ where k is the optimum cover size, and can be implemented to run in time linear in the input size. However, the input size itself could be as large as $\Theta(mn)$, so for large data sets even reading the input might be infeasible.

This raises a natural question: *is it possible to solve minimum set cover in sub-linear time?* This question was previously addressed in [NO08, YYI12], who showed that one can design constant running-time algorithms by simulating the greedy algorithm, under the assumption that the sets are of constant size and each element occurs in a constant number of sets. However, those constant-time algorithms have a few drawbacks: they only provide a mixed multiplicative/additive guarantee (the output cover size is guaranteed to be at most $k \cdot \ln n + \epsilon n$), the dependence of their running times on the maximum set size is exponential, and they only output the (approximate) minimum set cover size, not the cover itself. From a different perspective, [KY14] (building on [GK95]) showed that an $O(1)$-approximate solution to the *fractional* version of the problem can be found in $\widetilde{O}(mk^2 + nk^2)$ time.[1] Combining this algorithm with the randomized rounding technique yields an $O(\log n)$-approximate solution to Set Cover with the same complexity.

In this work we initiate a systematic study of the complexity of sub-linear time algorithms for set cover with multiplicative approximation guarantees. Our upper bounds complement the aforemen-

[1] The method can be further improved to $\widetilde{O}(m + nk)$ (N. Young, personal communication).

tioned result of [KY14] by presenting algorithms which are fast when k is *large*, as well as algorithms that provide more accurate solutions (even with a constant-factor approximation guarantee) that use a sub-linear number of *probes*[2]. Equally importantly, we establish nearly matching lower bounds, some of which even hold for estimating the optimal cover size. Our algorithmic results and lower bounds are presented in Table 3.1.

Data access model. As in the prior work [NO08, YYI12] on Set Cover, our algorithms and lower bounds assume that the input can be accessed via the adjacency-list oracle[3]. More precisely, the algorithm has access to the following two oracles:

1. **EltOf:** Given a set S_i and an index j, the oracle returns the j^{th} element of S_i. If $j > |S_i|$, $\perp$ is returned.
2. **SetOf:** Given an element e_i and an index j, the oracle returns the j^{th} set containing e_i. If e_i appears in less than j sets, $\perp$ is returned.

This is a natural model, providing a "two-way" connection between the sets and the elements. Furthermore, for some graph problems modeled by Set Cover (such as Dominating Set or Vertex Cover), such oracles are essentially equivalent to the aforementioned incident-list model studied in sub-linear graph algorithms. We also note that the other popular access model employing the *membership oracle*, where we can probe whether an element e is contained in a set S, is not suitable for Set Cover, as it can be easily seen that even checking whether a feasible cover exists requires $\Omega(mn)$ time.

3.1.1 Our results

In this work we present algorithms and lower bounds for the Set Cover problem. The results are summarized in Table 3.1. The **NP**-hardness of this problem (or even its $o(\log n)$-approximate version [Fei98, RS97, AMS06, Mos15, DS14]) precludes the existence of highly accurate algorithms with fast running times, while (as we show) it is still possible to design algorithms with sub-linear probe complexities and low approximation factors. The lower bound proofs hold for the running time of any algorithm approximation set cover assuming the defined data access model.

We present two algorithms with sub-linear number of probes. First, we show that the streaming algorithm presented in [HIMV16] can be adapted so that it returns an $O(\alpha)$-approximate cover using $\widetilde{O}(m(n/k)^{1/(\alpha-1)} + nk)$ probes, which could be *quadratically* smaller than mn. Second, we present a simple algorithm which is tailored to the case when the value of k is large. This algorithm computes an $O(\log n)$-approximate cover with $\widetilde{O}(mn/k)$ *time* complexity (not just probe complexity). Hence, by combining it with the algorithm of [KY14], we get an $O(\log n)$-approximation algorithm that runs in time $\widetilde{O}(m + n\sqrt{m})$.

We complement the first result by proving that for low values of k, the required number of probes is $\widetilde{\Omega}(m(n/k)^{1/(2\alpha)})$ even for estimating the size of the optimal cover. This shows that the first algorithm is essentially optimal for the values of k where the first term in the runtime bound

[2]Note that polynomial *time* algorithm with sub-logarithmic approximation algorithms are unlikely to exist.

[3]In the context of graph problems, this model is also known as the *incidence-list model*, and has been studied extensively, see e.g., [CRT05, GKK13, BHNT15].

Problem	Approximation	Constraints	Probe Complexity	Section
Set Cover	$\alpha\rho+\varepsilon$	$\alpha \geq 2$	$\widetilde{O}\left(\frac{1}{\varepsilon}(m(\frac{n}{k})^{\frac{1}{\alpha-1}}+nk)\right)$	3.2.2
	$\rho+\varepsilon$	-	$\widetilde{O}\left(\frac{mn}{k\varepsilon^2}\right)$	3.2.3
	α	$k<\left(\frac{n}{\log m}\right)^{\frac{1}{4\alpha+1}}$	$\widetilde{\Omega}\left(m\left(\frac{n}{k}\right)^{\frac{1}{2\alpha}}\right)$	3.6
	α	$\alpha \leq 1.01$ $k=O\left(\frac{n}{\log m}\right)$	$\widetilde{\Omega}\left(\frac{mn}{k}\right)$	3.5.2
Cover Verification	-	$k \leq n/2$	$\Omega(nk)$	3.3

Table 3.1: A summary of our algorithms and lower bounds. We use the following notation: $k \geq 1$ denotes the size of the optimum cover; $\alpha \geq 1$ denotes a parameter that determines the trade-off between the approximation quality and probe/time complexities; $\rho \geq 1$ denotes the approximation factor of a "black box" algorithm for set cover used as a subroutine; We assume that $\alpha \leq \log n$ and $m \geq n$.

dominates. Moreover, we prove that even the Cover Verification problem, which is checking whether a given collection of k sets covers all the elements, would require $\Omega(nk)$ probes. This provides strong evidence that the term nk in the first algorithm is unavoidable. Lastly, we complement the second algorithm, by showing a lower bound of $\widetilde{\Omega}(mn/k)$ if the approximation ratio is a small constant.

3.1.2 Related work

Sublinear algorithms for Set Cover under the oracle model have been previously studied as an estimation problem; the goal is only to approximate the size of the minimum set cover rather than constructing one. Nguyen and Onak [NO08] consider Set Cover under the oracle model we employ in this work, in a specific setting where both the maximum cardinality of sets in $\mathcal{F}$, and the maximum number of occurrences of an element over all sets, are bounded by some constants s and t; this allows algorithms whose time and probe complexities are constant, $(2^{(st)^4}/\varepsilon)^{O(2^s)}$, containing no dependency on n or m. They provide an algorithm for estimating the size of the minimum set cover when, unlike our work, allowing both $\ln s$ multiplicative and εn additive errors. Their result has been subsequently improved to $(st)^{O(s)}/\varepsilon^2$ by Yoshida et al. [YYI12]. Additionally, the results of Kuhn et al. [KMW06] on general packing/covering LPs in the distributed $\mathcal{LOCAL}$ model, together with the reduction method of Parnas and Ron [PR07], implies that estimating the optimum set cover size to within a $O(\ln s)$-multiplicative factor (with εn additive error) can be performed in $(st)^{O(\log s \log t)}/\varepsilon^4$ time/probe complexities.

Set Cover can also be considered as a generalization of the Vertex Cover problem. The estimation variant of Vertex Cover under the adjacency-list oracle model has been studied in [PR07, MR06, ORRR12, YYI12]. Set Cover has been also studied in the sublinear *space* context, most no-

tably for the streaming model of computation [SG09, ER16, CW16, AKL16, Ass17, BEM17, IMR+17, DIMV14, HIMV16]. In this model, there are algorithms that compute approximate set covers with only multiplicative errors. Our algorithms use some of the ideas introduced in the last two papers [DIMV14, HIMV16].

3.1.3 Overview of the algorithms

The algorithmic results presented in Section 3.2, use the techniques introduced for the streaming Set Cover problem by [DIMV14, HIMV16] to get new results in the context of sub-linear time algorithms for this problem. Two components previously used for the set cover problem in the context of streaming are Set Sampling and Element Sampling. Assuming the size of the minimum set cover is k, Set Sampling randomly samples $\widetilde{O}(k)$ sets and adds them to the maintained solution. This ensures that all the elements that are well represented in the input (i.e., appearing in at least m/k sets) are covered by the sampled sets. On the other hand, the Element Sampling technique samples roughly $\widetilde{O}(k/\delta)$ elements, and finds a set cover for the sampled elements. It can be shown that the cover for the sampled elements covers a $(1-\delta)$ fraction of the original elements.

Specifically, the first algorithm performs a constant number of iterations. Each iteration uses element sampling to compute a "partial" cover, removes the elements covered by the sets selected so far and recurses on the remaining elements. However, making this process work in sub-linear time (as opposed to sub-linear space) requires new technical development. For example, the algorithm of [HIMV16] relies on the ability to test membership for a set-element pair, which generally cannot be efficiently performed in our model.

The second algorithm performs only one round of set sampling, and then identifies the elements that are not covered by the sampled sets, *without* performing a full scan of those sets. This is possible because with high probability only those elements that belong to few input sets are not covered by the sample sets. Therefore, we can efficiently enumerate all pairs (e_i, S_j), $e_i \in S_j$, for those elements e_i that were not covered by the sampled sets. We then run a black box algorithm only on the set system induced by those pairs. This approach lets us avoid the nk term present in the probe and runtime bounds for the first algorithm, which makes the second algorithm highly efficient for large values of k.

3.1.4 Overview of the lower bounds

The Set Cover lower bound for smaller optimal value k. We establish our lower bound for the problem of *estimating* the size of the minimum set cover, by constructing two distributions of set systems. All systems in the same distribution share the same optimal set cover size, but these sizes differ by a factor α between the two distributions; thus, the algorithm is required to determine from which distribution its input set system is drawn, in order to correctly estimate the optimal cover size. Our distributions are constructed by a novel use of the probabilistic method. Specifically, we first probabilistically construct a set system called *median instance* (see Lemma 3.5.6): this set system has the property that (a) its minimum set cover size is αk and (b) a small number of changes to the instance reduces the minimum set cover size to k. We set the first distribution to

be always this median instance. Then, we construct the second distribution by a random process that performs the changes (depicted in Figure 3-8) resulting in a *modified instance*. This process distributes the changes almost uniformly throughout the instance, which implies that the changes are unlikely to be detected unless the algorithm performs a large number of probes. We believe that this construction might find applications to lower bounds for other combinatorial optimization problems.

The SetCover lower bound for larger optimal value k. Our lower bound for the problem of *computing* an approximate set cover leverages the construction above. We create a combined set system consisting of multiple modified instances all chosen independently at random, allowing instances with much larger k. By the properties of the random process generating modified instances, we observe that most of these modified instances have different optimal set cover solution, and that distinguishing these instances from one another requires many probes. Thus, it is unlikely for the algorithm to be able to compute an optimal solution to a large fraction of these modified instances, and therefore it fails to achieve the desired approximation factor for the overall combined instance.

The Cover Verification lower bound for a cover of size k. For Cover Verification, however, we instead give an explicit construction of the distributions. We first create an underlying set structure such that initially, the candidate sets contain all but k elements. Then we may swap in each uncovered element from a non-candidate set. Our set structure is systematically designed so that each swap only modifies a small fraction of the answers from all possible probes; hence, each swap is hard to detect without $\Omega(n)$ probes. The distribution of valid set covers is composed of instances obtained by swapping in every uncovered element, and that of non-covers is similarly obtained but leaving one element uncovered.

3.2 Sub-Linear Algorithms for the Set Cover Problem

In this work, we present two different approximation algorithms for Set Cover with sub-linear probe in the oracle model: **smallSetCover** and **largeSetCover**. Both of our algorithms rely on the techniques from the recent developments on Set Cover in the streaming model. However, adopting those techniques in the oracle model requires novel insights and technical development.

Throughout the description of our algorithms, we assume that we have access to a black box subroutine that given the full Set Cover instance (where all members of all sets are revealed), returns a ρ-approximate solution[4].

The first algorithm (**smallSetCover**) returns a $(\alpha\rho + \varepsilon)$ approximate solution of the Set Cover instance using $\widetilde{O}(\frac{1}{\varepsilon}(m(\frac{n}{k})^{\frac{1}{\alpha-1}} + nk))$ probes, while the second algorithm (**largeSetCover**) achieves an approximation factor of $(\rho + \varepsilon)$ using $\widetilde{O}(\frac{mn}{k\varepsilon^2})$ probes, where k is the size of the minimum set cover. These algorithms can be combined so that the number of probes of the algorithm becomes asymptotically the minimum of the two:

Theorem 3.2.1. *There exists a randomized algorithm for* Set Cover *in the oracle model that*

[4]The approximation factor ρ may take on any value between 1 and $\Theta(\log n)$ depending on the computational model one assumes.

w.h.p. computes an $O(\rho \log n)$-approximate solution and uses $\widetilde{O}(\min\{m\left(\frac{n}{k}\right)^{\frac{1}{\log n}} + nk\ ,\ \frac{mn}{k}\}) = \widetilde{O}(m + n\sqrt{m})$ number of probes.

3.2.1 Preliminaries

Our algorithms use the following two sampling techniques developed for Set Cover in the streaming model [DIMV14]: Element Sampling and Set Sampling. The first technique, Element Sampling, states that in order to find a $(1-\delta)$-cover of $\mathcal{U}$ w.h.p., it suffices to solve Set Cover on a subset of elements of size $\widetilde{O}(\frac{\rho k \log m}{\delta})$ picked uniformly at random. It shows that we may restrict our attention to a subproblem with a much smaller number of elements, and our solution to the reduced instance will still cover a good fraction of the elements in the original instance. The next technique, Set Sampling, shows that if we pick ℓ sets uniformly at random from $\mathcal{F}$ in the solution, then each element that is not covered by any of picked sets w.h.p. only occurs in $\widetilde{O}(\frac{m}{\ell})$ sets in $\mathcal{F}$; that is, we are left with a much sparser subproblem to solve. The formal statements of these sampling techniques are as follows. See [DIMV14] for the proofs.

Lemma 3.2.2 (Element Sampling). *Consider an instance of* Set Cover *on $(\mathcal{U}, \mathcal{F})$ whose optimal cover has size at most k. Let $\mathcal{U}_{\text{smp}}$ be a subset of $\mathcal{U}$ of size $\Theta\left(\frac{\rho k \log m}{\delta}\right)$ chosen uniformly at random, and let $\mathcal{C}_{\text{smp}} \subseteq \mathcal{F}$ be a ρ-approximate cover for $\mathcal{U}_{\text{smp}}$. Then, w.h.p. $\mathcal{C}_{\text{smp}}$ covers at least $(1-\delta)|\mathcal{U}|$ elements.*

Lemma 3.2.3 (Set Sampling). *Consider an instance $(\mathcal{U}, \mathcal{F})$ of* Set Cover. *Let $\mathcal{F}_{\mathsf{rnd}}$ be a collection of ℓ sets picked uniformly at random. Then, w.h.p. $\mathcal{F}_{\mathsf{rnd}}$ covers all elements that appear in $\Omega(\frac{m \log n}{\ell})$ sets of $\mathcal{F}$.*

3.2.2 Efficient algorithm for instances with small optimal value

The algorithm of this section is a modified variant of the streaming algorithm of Set Cover in [HIMV16] that works in the sublinear probe model. Similarly to the algorithm of [HIMV16], our algorithm **smallSetCover** considers different guesses of the value of an optimal solution ($\varepsilon^{-1} \log n$ guesses) and performs the core *iterative* algorithm **iterSetCover** for all of them in parallel. For each guess ℓ of the size of an optimal solution, the **iterSetCover** goes through $1/\alpha$ iterations and by applying Element Sampling, guarantees that w.h.p. at the end of each iteration, the number of uncovered elements reduces by a factor of $n^{-1/\alpha}$. Hence, after $1/\alpha$ iterations all elements will be covered. Furthermore, since the number of sets picked in each iteration is at most ℓ, the final solution has at most $\rho\ell$ sets where ρ is the performance of the *offline* block **algOfflineSC** that **iterSetCover** uses to solve the reduced instances constructed by Element Sampling.

Although our general approach in **iterSetCover** is similar to the iterative core of the streaming algorithm of Set Cover, there are challenges that we need to overcome so that it works *efficiently* in the probe model. Firstly, the approach of [HIMV16] relies on the ability to test membership for a set-element pair when executing its *set filtering* subroutine: given a subset S, the algorithm of [HIMV16] requires to compute $|S \cap \mathsf{S}|$ which cannot be implemented efficiently in the probe model (in the worst case, requires $m|\mathsf{S}|$ probes). Instead, here we employ the *set sampling* which w.h.p. guarantees that the number of sets that contain an (yet uncovered) element is small.

Next challenge is achieving $m(n/k)^{1/(\alpha-1)} + nk$ probe bound for computing an α-approximate solution. As mentioned earlier, both our approach and the algorithm of [HIMV16] need to run the algorithm in parallel for different guesses ℓ of the size of an optimal solution. However, since **iterSetCover** performs $m(n/\ell)^{1/(\alpha-1)} + n\ell$ probes, if **smallSetCover** invokes **iterSetCover** with guesses in an increasing order then the probe complexity becomes $mn^{1/(\alpha-1)}+nk$; on the other hand, if it invokes **iterSetCover** with guesses in a decreasing order then the probe complexity becomes $m(n/k)^{1/(\alpha-1)} + mn$! To solve this issue, **smallSetCover** performs in two stages: in the first stage, it finds a $(\log n)$-estimate of k by invoking **iterSetCover** using $m+nk$ probes (assuming guesses are evaluated in an increasing order) and then in the second rounds it only invokes **iterSetCover** with approximation factor α in the smaller $O(\log n)$-approximate region around the $(\log n)$-estimate of k computed in the first stage. Thus, in our implementation, besides the desired approximation factor, **iterSetCover** receives an upper bound and a lower bound on the size of an optimal solution.

Now, we provide a detailed description of **iterSetCover**. It receives α, ϵ, l and u as its arguments, and it is guaranteed that the size of an optimal cover of the input instance, k, is in $[l, u]$. Note that the algorithm does not know the value of k and the sampling techniques described in Section 3.2.1 rely on k. Therefore, the algorithm needs to find a $(1+\varepsilon)$ estimate[5] of k denoted as ℓ. This can be done by trying all powers of $(1+\varepsilon)$ in $[l, u]$. The parameter α denotes the trade-off between the probe complexity and the approximation guarantee that the algorithm achieves. Moreover, we assume that the algorithm has access to a ρ-approximate black box solver of Set Cover.

iterSetCover first performs Set Sampling to cover all elements that occur in $\widetilde{\Omega}(m/\ell)$ sets. Then it goes through $\alpha-2$ iterations and in each iteration, it performs Element Sampling with parameter $\delta = \widetilde{O}((\ell/n)^{1/(\alpha-1)})$. By Lemma 3.2.2, after $(\alpha-2)$ iterations, w.h.p. only $\ell\left(\frac{n}{\ell}\right)^{1/(\alpha-1)}$ elements remain uncovered, for which the algorithm finds a cover by invoking the *offline* set cover solver. The parameters are set so that all $(\alpha-1)$ instances that are required to be solved by the offline set cover solver (the $(\alpha-2)$ instances costructed by Element Sampling and the final instance) are of size $\widetilde{O}(m\left(\frac{n}{\ell}\right)^{1/(\alpha-1)})$.

In the rest of this section, we show that **smallSetCover** w.h.p. returns an almost $(\rho\alpha)$-approximate solution of Set Cover$(\mathcal{U}, \mathcal{F})$ with probe complexity $\widetilde{O}(\frac{1}{\varepsilon}(m(n/k)^{\frac{1}{\alpha-1}} + nk))$ where k is the size of a minimum set cover.

Theorem 3.2.4. *The* **smallSetCover** *algorithm outputs a $(\alpha\rho+\varepsilon)$-approximate solution of* Set Cover$(\mathcal{U}, \mathcal{F})$ *using $\widetilde{O}(\frac{1}{\varepsilon}(m(\frac{n}{k})^{\frac{1}{\alpha-1}} + nk))$ number of probes w.h.p., where k is the size of an optimal solution of $(\mathcal{U}, \mathcal{F})$.*

To analyze the performance of **smallSetCover**, first we need to analyze the procedures invoked by **smallSetCover**: **iterSetCover** and **algOfflineSC**. The procedure **algOfflineSC**(S, ℓ) receives as an input a subset of elements S and an estimate on the size of an optimal cover of S using sets in $\mathcal{F}$. The **algOfflineSC** algorithm first determines all occurrences of S in $\mathcal{F}$. Then it invokes a black box subroutine that returns a cover of size at most $\rho\ell$ (if there exists a cover of size ℓ for S) for the reduced Set Cover instance over S.

Moreover, we assume that all subroutines have access to the EltOf and SetOf oracles, $|\mathcal{U}|$ and $|\mathcal{F}|$.

[5] The exact estimate that the algorithm works with is a $(1+\frac{\varepsilon}{2\rho\alpha})$ estimate.

iterSetCover$(\alpha, \varepsilon, l, u)$:

▷ Try all $(1 + \frac{\varepsilon}{2\alpha\rho})$-approximate guesses of k

for $\ell \in \{(1 + \frac{\varepsilon}{2\alpha\rho})^i \mid \log_{1+\frac{\varepsilon}{2\alpha\rho}} l \leq i \leq \log_{1+\frac{\varepsilon}{2\alpha\rho}} u\}$

do in order:

$\mathtt{sol}_\ell \leftarrow$ collection of ℓ sets picked uniformly at random ▷ Set Sampling

$\mathcal{U}_{\text{rem}} \leftarrow \mathcal{U} \setminus \bigcup_{\mathsf{r} \in \mathtt{sol}_\ell} \mathsf{r}$ ▷ $n\ell$ EltOf

repeat $(\alpha - 2)$ times

$\mathsf{S} \leftarrow$ sample of $\mathcal{U}_{\text{rem}}$ of size $\widetilde{O}(\rho\ell \left(\frac{n}{\ell}\right)^{\frac{1}{\alpha-1}})$

$\mathcal{D} \leftarrow$ **algOfflineSC**(S, ℓ)

if $\mathcal{D} = \mathsf{null}$ **then**

break ▷ Try the next value of ℓ

$\mathtt{sol}_\ell \leftarrow \mathtt{sol}_\ell \bigcup \mathcal{D}$

$\mathcal{U}_{\text{rem}} \leftarrow \mathcal{U}_{\text{rem}} \setminus \bigcup_{\mathsf{r} \in \mathcal{D}} \mathsf{r}$ ▷ $\rho n\ell$ EltOf

if $|\mathcal{U}_{\text{rem}}| \leq \ell \left(\frac{n}{\ell}\right)^{1/(\alpha-1)}$ ▷ Feasibility Test

$\mathcal{D} \leftarrow$ **algOfflineSC**$(\mathcal{U}_{\text{rem}}, \ell)$

if $\mathcal{D} \neq \mathsf{null}$ **then**

$\mathtt{sol}_\ell \leftarrow \mathtt{sol}_\ell \bigcup \mathcal{D}$

return $\mathtt{sol}_\ell$

Figure 3-1: **iterSetCover** is the main procedure of the **smallSetCover** algorithm for the Set Cover problem.

Lemma 3.2.5. *Suppose that each $e \in \mathsf{S}$ appears in $\widetilde{O}(\frac{m}{\ell})$ sets of $\mathcal{F}$ and lets assume that there exists a set of ℓ sets in $\mathcal{F}$ that covers S. Then* **algOfflineSC**(S, ℓ) *returns a cover of size at most $\rho\ell$ of S using $\widetilde{O}(\frac{m|\mathsf{S}|}{\ell})$ probes.*

Proof. Since each element of S is contained by $\widetilde{O}(\frac{m}{\ell})$ sets in $\mathcal{F}$, the information required to solve the reduced instance on S can be obtained by $\widetilde{O}(\frac{m|\mathsf{S}|}{\ell})$ probes (i.e. $\widetilde{O}(\frac{m}{\ell})$ SetOf probe per element in S). □

Lemma 3.2.6. *The cover constructed by the outer loop of* **iterSetCover**$(\alpha, \varepsilon, l, u)$ *with the parameter $\ell > k$, $\mathtt{sol}_\ell$, w.h.p. covers $\mathcal{U}$.*

Proof. After picking ℓ sets uniformly at random, by Set Sampling (Lemma 3.2.3), w.h.p. each element that is not covered by the sampled sets appears in $\widetilde{O}(\frac{m}{\ell})$ sets of $\mathcal{F}$. Next, by Element Sampling (Lemma 3.2.2 with $\delta = \left(\frac{\ell}{n}\right)^{1/(\alpha-1)}$), at the end of each *inner* iteration, w.h.p. the number of uncovered elements decreases by a factor of $\left(\frac{\ell}{n}\right)^{1/(\alpha-1)}$. Thus after at most $(\alpha - 2)$ iterations, w.h.p. less than $\ell \left(\frac{n}{\ell}\right)^{1/(\alpha-1)}$ elements remain uncovered. Finally, **algOfflineSC** is invoked on the remaining elements; hence, $\mathtt{sol}_\ell$ w.h.p. covers $\mathcal{U}$. □

Next we analyze the probe complexity and the approximation guarantee of **iterSetCover**. As we only apply Element Sampling and Set Sampling polynomially many times, all invocations of the corresponding lemmas during an execution of the algorithm must succeed w.h.p., so we assume their *high probability* guarantees for the proofs in rest of this section.

algOfflineSC(S, ℓ):

$\mathcal{F}_{\mathsf{S}} \leftarrow \emptyset$

for each element $e \in \mathsf{S}$ **do**

$\mathcal{F}_e \leftarrow$ the collection of sets containing e

$\mathcal{F}_{\mathsf{S}} \leftarrow \mathcal{F}_{\mathsf{S}} \cup \mathcal{F}_e$

$\mathcal{D} \leftarrow$ solution of size at most $\rho\ell$ for Set Cover on $(\mathsf{S}, \mathcal{F}_{\mathsf{S}})$ constructed by the black box solver

▷ If there exists no such cover, then $\mathcal{D} =$ null

return $\mathcal{D}$

Figure 3-2: **algOfflineSC**(S, ℓ) invokes a black box that returns a cover of size at most $\rho\ell$ (if there exists a cover of size ℓ for S) for the Set Cover instance that is the projection of $\mathcal{F}$ over S.

Lemma 3.2.7. *Given that* $l \leq k \leq \frac{u}{1+\varepsilon/(2\alpha\rho)}$, *w.h.p.* **iterSetCover**$(\alpha, \varepsilon, l, u)$ *finds a* $(\rho\alpha + \varepsilon)$*-approximate solution of the input instance using* $\widetilde{O}\left(\frac{1}{\varepsilon}(m(\frac{n}{l})^{\frac{1}{\alpha-1}} + nk)\right)$ *probes.*

Proof. Let $\ell_k = (1 + \frac{\varepsilon}{2\alpha\rho})^{\lceil \log_{1+\frac{\varepsilon}{2\alpha\rho}} k \rceil}$ be the smallest power of $1 + \frac{\varepsilon}{2\alpha\rho}$ greater than or equal to k. Note that it is guaranteed that $\ell_k \in [l, u]$. By Lemma 3.2.6, **iterSetCover** terminates with a guess value $\ell \leq \ell_k$. In the following we compute the probe complexity of the run of **iterSetCover** with a parameter $\ell \leq \ell_k$.

Set Sampling component picks ℓ sets and then update the set of elements that are not covered by those sets, $\mathcal{U}_{\text{rem}}$, using $O(n\ell)$ EltOf probes. Next, in each iteration of the inner loop, the algorithm samples a subset S of size $\widetilde{O}\left(\ell(n/\ell)^{1/(\alpha-1)}\right)$ from $\mathcal{U}_{\text{rem}}$. Recall that, by Set Sampling (Lemma 3.2.3), each $e \in \mathsf{S} \subset \mathcal{U}_{\text{rem}}$ appears in at most $\widetilde{O}(m/\ell)$ sets. Since each element in $\mathcal{U}_{\text{rem}}$ appears in $\widetilde{O}(m/\ell)$, **algOfflineSC** returns a cover $\mathcal{D}$ of size at most $\rho\ell$ using $\widetilde{O}\left(m\,(n/\ell)^{1/(\alpha-1)}\right)$ SetOf probes (Lemma 3.2.5). By the guarantee of Element Sampling (Lemma 3.2.2), the number of elements in $\mathcal{U}_{\text{rem}}$ that are not covered by $\mathcal{D}$ is at most $(\ell/n)^{1/(\alpha-1)}|\mathcal{U}_{\text{rem}}|$. Finally, at the end of each inner loop, the algorithm updates the set of uncovered elements $\mathcal{U}_{\text{rem}}$ by using $\widetilde{O}(n\ell)$ EltOf probes. The Feasibility Test which is passed w.h.p. for $\ell \leq \ell_k$ ensures that the final run of **algOfflineSC** performs $\widetilde{O}(m(n/\ell)^{1/(\alpha-1)})$ SetOf probes. Hence, the total number of probes performed in each iteration of the outer loop of **iterSetCover** with parameter $\ell \leq \ell_k$ is $\widetilde{O}\left(m\,(n/\ell)^{1/(\alpha-1)} + n\ell\right)$.

By Lemma 3.2.6, if $\ell_k \leq u$, then the outer loop of **iterSetCover** is executed for $l \leq \ell \leq \ell_k$ before it terminates. Thus, the total number of probes made by **iterSetCover** is:

$$\begin{aligned}
&\sum_{i=\lceil \log_{1+\frac{\varepsilon}{2\alpha\rho}} l \rceil}^{\log_{1+\frac{\varepsilon}{2\alpha\rho}} \ell_k} \widetilde{O}\left(m\left(\frac{n}{(1+\frac{\varepsilon}{2\alpha\rho})^i}\right)^{\frac{1}{\alpha-1}} + n(1+\frac{\varepsilon}{2\alpha\rho})^i\right) \\
&= \widetilde{O}\left(m\left(\frac{n}{l}\right)^{\frac{1}{\alpha-1}}\left(\log_{1+\frac{\varepsilon}{2\alpha\rho}}\frac{\ell_k}{l}\right) + \frac{n\ell_k}{\varepsilon/(\rho\alpha)}\right) \\
&= \widetilde{O}\left(\frac{1}{\varepsilon}\left(m\left(\frac{n}{l}\right)^{\frac{1}{\alpha-1}} + nk\right)\right).
\end{aligned}$$

Now, we show that the number of sets returned by **iterSetCover** is not more than $(\alpha\rho + \varepsilon)\ell_k$.

Set Sampling picks ℓ sets and each run of **algOfflineSC** returns at most $\rho\ell$ sets. Thus the size of the solution returned by **iterSetCover** is at most $(1+(\alpha-1)\rho)\ell_k < (\alpha\rho+\varepsilon)k$. □

Next, we prove the main theorem of the section.

smallSetCover(α, ε):
  $\texttt{sol} \leftarrow$ **iterSetCover**$(\log n, 1, 1, n)$
  $k' \leftarrow |\texttt{sol}|$ ▷ Find a $\rho \log n$ estimate of k.
  return iterSetCover$(\alpha, \epsilon, \lfloor \frac{k'}{\rho \log n} \rfloor, \lceil k'(1+\frac{\varepsilon}{2\alpha\rho}) \rceil)$

Figure 3-3: The description of the **smallSetCover** algorithm.

Proof of Theorem 3.2.4. The algorithm **smallSetCover** first finds a $(\rho \log n)$-approximate solution of Set Cover$(\mathcal{U}, \mathcal{F})$, sol, with $\widetilde{O}(m+nk)$ probes by calling **iterSetCover**$(\log n, 1, 1, n)$. Having that $k \le k' = |\texttt{sol}| \le (\rho \log n)k$, the algorithm calls **iterSetCover** with α as the approximation factor and $[\lfloor k'/(\rho \log n) \rfloor, \lceil k'(1+\frac{\varepsilon}{2\alpha\rho}) \rceil]$ as the range containing k. By Lemma 3.2.7, the second call to **iterSetCover** in **smallSetCover** returns a $(\alpha\rho+\varepsilon)$-approximate solution of Set Cover$(\mathcal{U}, \mathcal{F})$ using the following number of probes:

$$\widetilde{O}\left(\frac{1}{\varepsilon}\left(m\left(\frac{n}{k/(\rho \log n)}\right)^{\frac{1}{\alpha-1}} + nk\right)\right) = \widetilde{O}\left(\frac{1}{\varepsilon}\left(m\left(\frac{n}{k}\right)^{\frac{1}{\alpha-1}} + nk\right)\right).$$

□

3.2.3 Efficient algorithm for instances with large optimal value

The second algorithm, **largeSetCover**, works strictly better than **smallSetCover** for large values of k ($k \ge \sqrt{m}$). The advantage of **largeSetCover** is that it does not need to update the set of uncovered elements at any point and simply avoids the additive nk term in the probe complexity bound; the result of Section 3.3 suggests that the nk term may be unavoidable if one wishes to maintain the uncovered elements. Note that the guarantees of **largeSetCover** is that at the end of the algorithm, w.h.p. the ground set $\mathcal{U}$ is covered.

The algorithm **largeSetCover**, given in Figure 3-4, first randomly picks $\varepsilon\ell/3$ sets. By Set Sampling (Lemma 3.2.3), w.h.p. every element that occurs in $\widetilde{\Omega}(m/(\varepsilon\ell))$ sets of $\mathcal{F}$ will be covered by the picked sets. It then solves the Set Cover instance over the elements that occur in $\widetilde{O}(m/(\varepsilon\ell))$ sets of $\mathcal{F}$ by an offline solver of Set Cover using $\widetilde{O}(m/(\varepsilon\ell))$ probes; note that this set of elements may include some already covered elements. In order to get the promised probe complexity, **largeSetCover** enumerates the guesses ℓ of the size of an optimal set cover in the decreasing order. The algorithm returns feasible solutions for $\ell \ge k$ and once it cannot find a feasible solution for ℓ, it returns the solution constructed for the previous guess of k, i.e., $\ell(1+\varepsilon/(3\rho))$. Since **largeSetCover** performs Set Sampling for $\widetilde{O}(\varepsilon^{-1})$ iterations, w.h.p. the total probe complexity of **largeSetCover** is $\widetilde{O}(mn/(k\varepsilon^2))$. Note that testing whether the number of occurrences of an element is $\widetilde{O}(m/(\varepsilon\ell))$ only requires a single probe, namely SetOf$(e, \frac{cm \log n}{\varepsilon\ell})$.

We now prove the desired performance of **largeSetCover**.

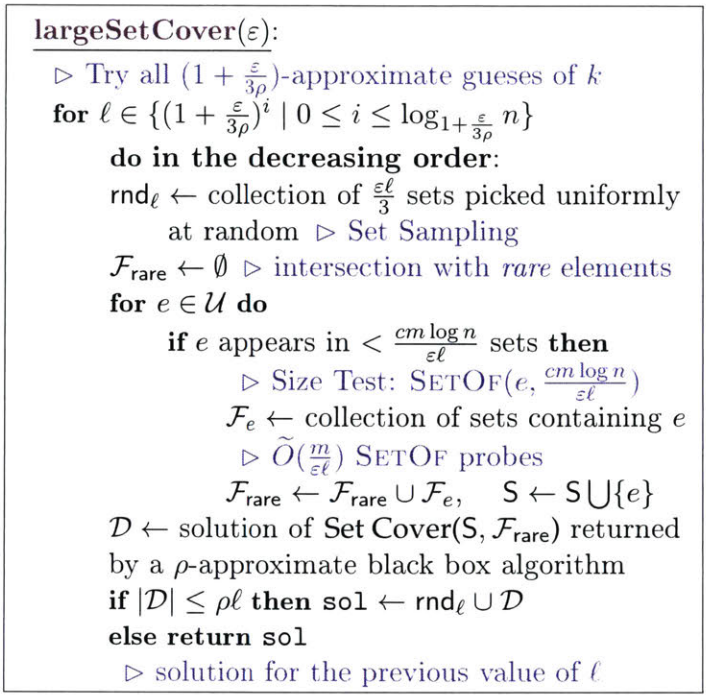


Figure 3-4: A $(\rho+\varepsilon)$-approximation algorithm for the Set Cover problem. We assume that the algorithm has access to EltOf and SetOf oracles for Set Cover$(\mathcal{U},\mathcal{F})$, as well as $|\mathcal{U}|$ and $|\mathcal{F}|$.

Lemma 3.2.8. **largeSetCover** *returns a* $(\rho+\varepsilon)$*-approximate solution of* Set Cover$(\mathcal{U},\mathcal{F})$ *w.h.p.*

Proof. The algorithm **largeSetCover** tries to construct set covers of decreasing sizes until it fails. Clearly, if $k \le \ell$ then the black box algorithm finds a cover of size at most $\rho\ell$ for any subset of $\mathcal{U}$, because k sets are sufficient to cover $\mathcal{U}$. In other words, the algorithm does not terminate with $\ell \ge k$. Moreover, since the algorithm terminates when ℓ is smaller than k, the size of the set cover found by **largeSetCover** is at most $(\frac{\varepsilon}{3}+\rho)(1+\frac{\varepsilon}{3\rho})\ell < (\frac{\varepsilon}{3}+\rho)(1+\frac{\varepsilon}{3\rho})k < (\rho+\varepsilon)k$. □

Lemma 3.2.9. *The number of probes made by* **largeSetCover** *is* $\widetilde{O}(\frac{mn}{k\varepsilon^2})$.

Proof. The value of ℓ in any *successful* iteration of the algorithm is greater than $k/(\rho+\varepsilon)$; otherwise, the size of the solution constructed by the algorithm is at most $(\rho+\varepsilon)\ell < k$ which is a contradiction.

Set Sampling guarantees that w.h.p. each uncovered element appears in $\widetilde{\Theta}(m/\varepsilon\ell)$ sets and thus the algorithm needs to perform $\widetilde{O}(\frac{mn}{\varepsilon\ell})$ SetOf probes to construct $\mathcal{F}_{\mathsf{rare}}$. Moreover, the number of required probes in the *size test* step is $O(n)$ because we only need one SetOf probe per each element in $\mathcal{U}$. Thus, the probe complexity of **largeSetCover**(ε) is bounded by

$$\sum_{i=\log_{1+\frac{\varepsilon}{3\rho}}\frac{k}{\rho+\varepsilon}}^{\log_{1+\frac{\varepsilon}{3\rho}} n} \widetilde{O}\left(n+\frac{mn}{\varepsilon(1+\frac{\varepsilon}{3\rho})^i}\right) = \widetilde{O}\left(\left(n+\frac{mn}{\varepsilon k}\right)\log_{1+\frac{\varepsilon}{3\rho}}\frac{n}{k}\right) = \widetilde{O}\left(\frac{mn}{k\varepsilon^2}\right).$$

□

3.3 Lower Bound for the Cover Verification Problem

In this section, we give a tight lower bound on a feasibility variant of the Set Cover problem which we refer to as Cover Verification. In Cover Verification$(\mathcal{U}, \mathcal{F}, \mathcal{F}_k)$, besides a collection of m sets $\mathcal{F}$ and n elements $\mathcal{U}$, we are given indices of k sets $\mathcal{F}_k \subseteq \mathcal{F}$, and the goal is to determine whether they are covering the whole universe $\mathcal{U}$ or not. We note that, throughout this section, the parameter k is a candidate for, but not necessarily the value of, the size of the minimum set cover.

A naive approach for this decision problem is to probe all elements in the given k sets and then check whether they cover $\mathcal{U}$ or not; this approach requires $O(nk)$ probes. However, in what follows we show that this approach is tight and no *randomized* protocol can decide whether the given k sets cover the whole universe with probability of success at least 0.9 using $o(nk)$ probes.

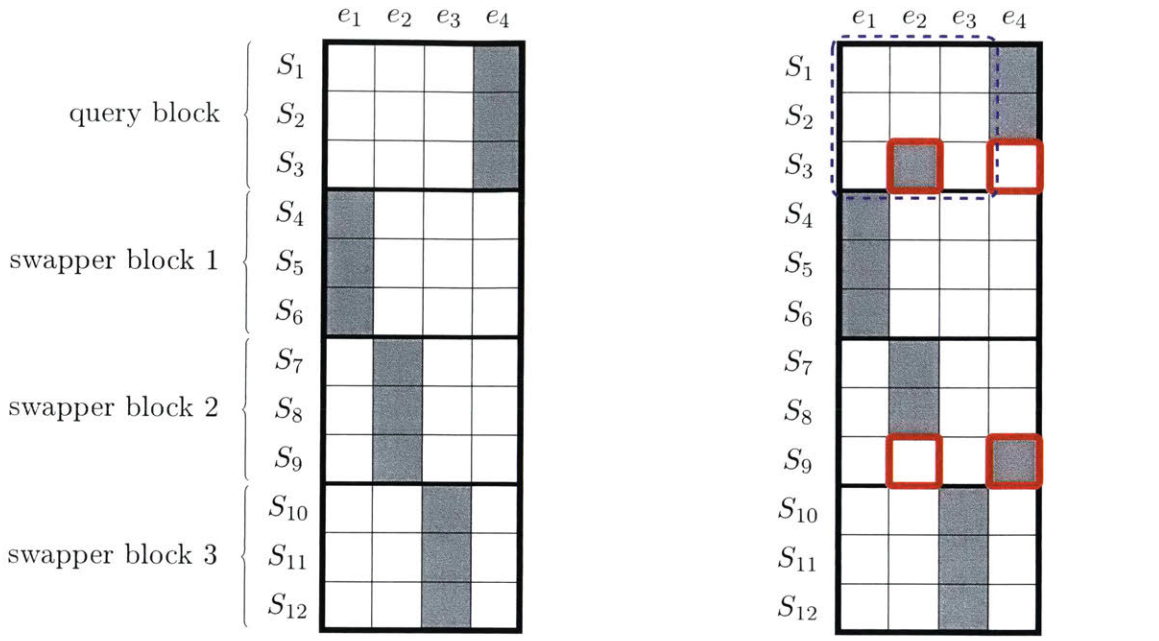
Theorem 3.3.1. *Any (randomized) algorithm for deciding whether a given $k = \Omega(\log n)$ sets covers all elements with probability of success at least 0.9, requires $\Omega(nk)$ probes.*

While this lower bound does not directly lead to a lower bound on Set Cover, it suggests that verifying the feasibility of a solution may even be more costly than finding the approximate solution itself; any algorithm bypassing this $\Omega(nk)$ lower bound may not solve Cover Verification as a subroutine.

We prove our lower bound by designing the Yes and No instances that are hard to distinguish, such that for a Yes instance, the union of the given k sets is $\mathcal{U}$, while for a No instance, their union only covers $n - 1$ elements. Each Yes instance is indistinguishable from a good fraction of No instances. Thus any algorithm must unavoidably answer incorrectly on half of these fractions, and fail to reach the desired probability of success.

3.3.1 Underlying set structure

Our instance contains n sets and n elements (so $m = n$), where the first k sets forms $\mathcal{F}_k$, the candidate for the set cover we wish to verify. We first consider the incidence matrix representation, such that the rows represent the sets and the columns represent the elements. We focus on the first n/k elements, and consider a *slab*, composing of n/k columns of the incidence matrix. We define a *basic slab* as the structure illustrated in Figure 3-5 (for $n = 12$ and $k = 3$), where the cell (i, j) is *white* if $e_j \in S_i$, and is *gray* otherwise. The rows are divided into blocks of size k, where first block, the *probe block*, contains the rows whose sets we wish to check for coverage; notice that only the last element is not covered. More specifically, in a basic slab, the probe block contains sets $S_1, \ldots, S_{n/k}$, each of which is equal to $\{e_1, \ldots, e_{n/k-1}\}$. The subsequent rows form the *swapper blocks* each consisting of n/k sets. The r^{th} swapper block consists of sets $S_{(r+1)n/k+1}, \ldots, S_{(r+2)n/k}$, each of which is equal to $\{e_1, \ldots, e_{n/k}\} \setminus \{e_r\}$. We perform one swap in this slab. Consider a parameter (x, y) representing the index of a white cell within the probe block. We exchange the color of this white cell with the gray cell on the same row, and similarly exchange the same pair of cells on swapper block y. An example is given in Figure 3-5; the dashed blue rectangle corresponds to the indices parameterizing possible swaps, and the red squares mark the modified cells. This modification corresponds to a single swap operation; in this example, choosing the index $(3, 2)$ swaps



(a) a basic slab

(b) the slab after performing a $(3, 2)$-swap

Figure 3-5: A basic slab and an example of a swapping operation.

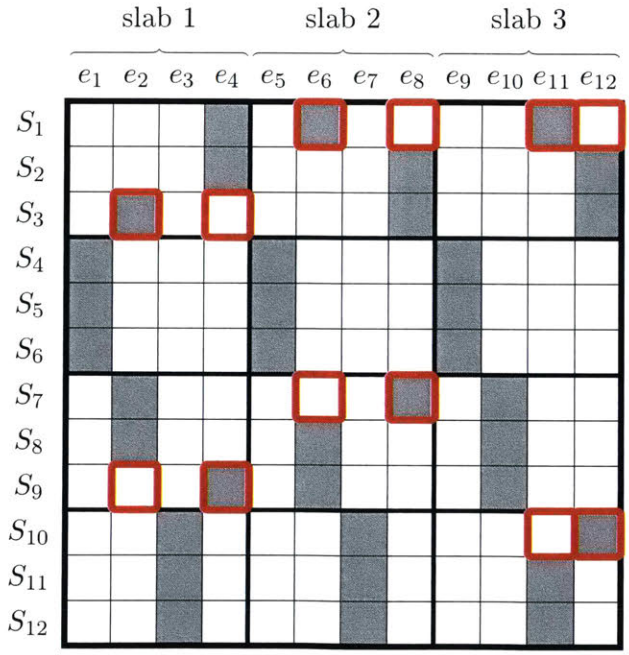


Figure 3-6: A example structure of a Yes instance; all elements are covered by the first 3 sets.

(e_2, e_4) between S_3 and S_9. Observe that there are $k \times (n/k - 1) = n - k$ possible swaps on a single slab, and any single swap allows the probe sets to cover all n/k elements.

Lastly, we may create the full instance by placing all k slabs together, as shown in Figure 3-6, shifting the elements' indices as necessary. The structure of our sets may be specified solely by the swaps made on these slabs. We define the structure of our instances as follows.

- For a Yes instance, we make one random swap on each slab. This allows the first k sets to cover all elements.
- For a No instance, we make one random swap on each slab except for exactly one of them. In that slab, the last element is not covered by any of the first k sets.

Now, to properly define an instance, we must describe our structure via EltOf and SetOf. We first create a temporary instance consisting of k basic slabs, where none of the cells are swapped. Create EltOf and SetOf lists by sorting each list in an increasing order of indices. Each instance from the above construction can then be obtained by applying up to k swaps on this temporary instance. Figure 3-7 provides a sample realization of a basic slab with EltOf and SetOf, as well as a sample result of applying a swap on this basic slab; these correspond to the incidence matrices in Figure 3-5a and Figure 3-5b, respectively. Such a construction can be extended to include all k slabs. Observe here that no two distinct swaps modify the same entry; that is, the swaps do not interfere with one another on these two functions. We also note that many entries do not participate in any swap.

3.3.2 Proof of Theorem 3.3.1

Observe that according to our instance construction, the algorithm may verify, with a single probe, whether a certain swap occurs in a certain slab. Namely, it is sufficient to probe an entry of EltOf or SetOf that would have been modified by that swap, and check whether it is actually modified or not. For simplicity, we assume that the algorithm has the knowledge of our construction. Further, without loss of generality, the algorithm does not make multiple probes about the same swap, or make a probe that is not corresponding to any swap.

We employ Yao's principle as follows: to prove a lower bound for randomized algorithms, we show a lower bound for any deterministic algorithm on a fixed distribution of input instances. Let $s = n - k$ be the number of possible swaps in each slab; assume $s = \Theta(n)$. We define our distribution of instances as follows: each of the s^k possible Yes instances occurs with probability $1/(2s^k)$, and each of the ks^{k-1} possible No instances occurs with probability $1/(2ks^{k-1})$. Equivalently speaking, we create a random Yes instance by making one swap on each basic slab. Then we make a coin flip: with probability $1/2$ we pick a random slab and undo the swap on that slab to obtain a No instance; otherwise we leave it as a Yes instance. To prove by contradiction, assume there exists a deterministic algorithm that solves the Cover Verification problem over this distribution of instances with $r = o(sk)$ probes.

Consider the Yes instances portion of the distribution, and observe that we may alternatively interpret the random process generating them as as follows. For each slab, one of its s possible swaps is chosen uniformly at random. This condition again follows the scenario considered in Section 3.5.2: we are given k urns (slabs) of each consisting of s marbles (possible swap locations), and aim to

Before: EltOf table for a basic slab

EltOf	1	2	3
S_1	e_1	e_2	e_3
S_2	e_1	e_2	e_3
S_3	e_1	e_2	e_3
S_4	e_2	e_3	e_4
S_5	e_2	e_3	e_4
S_6	e_2	e_3	e_4
S_7	e_1	e_3	e_4
S_8	e_1	e_3	e_4
S_9	e_1	e_3	e_4
S_{10}	e_1	e_2	e_4
S_{11}	e_1	e_2	e_4
S_{12}	e_1	e_2	e_4

After: EltOf table after applying a swap

EltOf	1	2	3
S_1	e_1	e_2	e_3
S_2	e_1	e_2	e_3
S_3	e_1	e_4	e_3
S_4	e_2	e_3	e_4
S_5	e_2	e_3	e_4
S_6	e_2	e_3	e_4
S_7	e_1	e_3	e_4
S_8	e_1	e_3	e_4
S_9	e_1	e_3	e_2
S_{10}	e_1	e_2	e_4
S_{11}	e_1	e_2	e_4
S_{12}	e_1	e_2	e_4

Before: SetOf table for a basic slab

SetOf	1	2	3	4	5	6	7	8	9
e_1	S_1	S_2	S_3	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}
e_2	S_1	S_2	S_3	S_4	S_5	S_6	S_{10}	S_{11}	S_{12}
e_3	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
e_4	S_4	S_5	S_6	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}

After: SetOf table after applying a swap

SetOf	1	2	3	4	5	6	7	8	9
e_1	S_1	S_2	S_3	S_7	S_8	S_9	S_{10}	S_{11}	S_{12}
e_2	S_1	S_2	S_9	S_4	S_5	S_6	S_{10}	S_{11}	S_{12}
e_3	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8	S_9
e_4	S_4	S_5	S_6	S_7	S_8	S_3	S_{10}	S_{11}	S_{12}

Figure 3-7: Tables illustrating the representation of a slab under EltOf and SetOf before and after a swap; cells modified by $\mathsf{swap}(e_2, e_4)$ between S_3 and S_9 are highlighted in red.

draw the red marble (swapped entry) from a large fraction of these urns. Following the proof of Lemmas 3.5.19-3.5.20, we obtain that if the total number of probes made by the algorithm is less than $(1-\frac{3}{b})\frac{sk}{b}$, then with probability at least 0.99, the algorithm will not see any swaps from at least $\frac{k}{b}$ slabs.

Then, consider the corresponding No instances obtained by undoing the swap in one of the slabs of the Yes instance. Suppose that the deterministic algorithm makes less than $(1-\frac{3}{b})\frac{sk}{b}$ probes, then for a fraction of 0.99 of all possible tuples $\mathcal{T}$, the output of the Yes instance is the same as the output of $\frac{1}{b}$ fraction of No instances, namely when the slab containing no swap is one of the $\frac{k}{b}$ slabs that the algorithm has not detected a swap in the corresponding Yes instance; the algorithm must answer incorrectly on half of the corresponding weight in our distribution of input instances. Thus the probability of success for any algorithm with less than $(1-\frac{3}{b})\frac{sk}{b}$ probes is at most

$$1-\mathbf{Pr}\left[|\mathcal{T}_{\text{high}}| \geq \left(1-\frac{2}{b}\right)k\right]\left(\frac{1}{b}\right)\left(\frac{1}{2}\right) \leq 1-\frac{0.495}{b} < 0.9,$$

for a sufficiently small constant $b > 3$ (e.g. $b = 4$). As $s = \Theta(n)$ and by Yao's principle, this implies the lower bound of $\Omega(nk)$ for the Cover Verification problem.

3.4 Preliminaries for the Lower Bounds

First, we formally specify the representation of the set structures of input instances, which applies to both Set Cover and Cover Verification.

Our lower bound proofs rely mainly on the construction of instances that are hard to distinguish by the algorithm. To this end, we define the swap operation that exchanges a pair of elements between two sets, and how this is implemented in the actual representation.

Definition 3.4.1 (swap operation). *Consider two sets S and S'. A swap on S and S' is defined over two elements e, e' such that $e \in S \setminus S'$ and $e' \in S' \setminus S$, where S and S' exchange e and e'. Formally, after performing* swap(e, e'), $S = (S \cup \{e'\}) \setminus \{e\}$ *and* $S' = (S' \cup \{e\}) \setminus \{e'\}$. *As for the representation via* EltOf *and* SetOf, *each application of* swap *only modifies 2 entries for each oracle. That is, if previously* $e = \textsc{EltOf}(S, i)$, $S = \textsc{SetOf}(e, j)$, $e' = \textsc{EltOf}(S', i')$, *and* $S' = \textsc{SetOf}(e', j')$, *then their new values change as follows:* $e' = \textsc{EltOf}(S, i)$, $S' = \textsc{SetOf}(e, j)$, $e = \textsc{EltOf}(S', i')$, *and* $S = \textsc{SetOf}(e', j')$.

In particular, we extensively use the property that the amount of changes to the oracle's answers incurred by each swap is minimal. We remark that when we perform multiple swaps on multiple *disjoint* set-element pairs, every swap modifies distinct entries and do not interfere with one another.

Lastly, we define the notion of probe-answer history, which is a common tool for establishing lower bounds for sub-linear algorithms under probe models.

Definition 3.4.2. *By* probe-answer history, *we denote the sequence of probe-answer pairs* $\langle (q_1, a_1), (q_2, a_2), \ldots, (q_r, a_r)\rangle$ *recording the communication between the algorithm and the oracles, where each new probe* q_{i+1} *may only depend on the probe-answer pairs* $(q_1, a_1), \ldots, (q_i, a_i)$. *In our case, each*

q_i represents either a SETOF *probe or an* ELTOF *probe made by the algorithm, and each a_i is the oracle's answer to that respective probe according to the set structure instance.*

3.5 Lower Bounds for the Set Cover Problem

In this section, we present lower bounds for Set Cover both for small values of the optimal cover size k (in Section 3.5.1), and for large values of k (in Section 3.5.2). For low values of k, we prove the following theorem whose proof is postponed to Section 3.6.

Theorem 3.5.1. *For $2 \leq k \leq (\frac{n}{16\alpha \log m})^{\frac{1}{4\alpha+1}}$ and $1 < \alpha \leq \log n$, any randomized algorithm that solves the* Set Cover *problem with approximation factor α and success probability at least $2/3$ requires $\widetilde{\Omega}(m(n/k)^{\frac{1}{2\alpha}})$ probes.*

Instead, in Section 3.5.1 we focus on the simple setting of this theorem which applies to approximation protocols for distinguishing between instances with minimum set cover sizes 2 and 3, and show a lower bound of $\widetilde{\Omega}(mn)$ (which is tight up to a polylogarithmic factor) for approximation factor $3/2$. This simplification is for the purpose of both clarity and also for the fact that the result for this case is used in Section 3.5.2 to establish our lower bound for large values of k.

High level idea. Our approach for establishing the lower bound is as follows. First, we construct a *median instance* I^* for Set Cover, whose minimum set cover size is 3. We then apply a randomized procedure **genModifiedInst**, which slightly modifies the median instance into a new instance containing a set cover of size 2. Applying Yao's principle, the distribution of the input to the deterministic algorithm is either I^* with probability $1/2$, or a modified instance generated thru **genModifiedInst**(I^*), which is denoted by $\mathcal{D}(I^*)$, again with probability $1/2$. Next, we consider the execution of the deterministic algorithm. We show that unless the algorithm asks at least $\widetilde{\Omega}(mn)$ probes, the resulting probe-answer history generated over I^* would be the same as those generated over instances constituting a constant fraction of $\mathcal{D}(I^*)$, reducing the algorithm's success probability to below $2/3$. More specifically, we will establish the following theorem.

Theorem 3.5.2. *Any algorithm that can distinguish whether the input instance is I^* or belongs to $\mathcal{D}(I^*)$ with probability of success greater than $2/3$, requires $\Omega(mn/\log m)$ probes.*

Corollary 3.5.3. *For $1 < \alpha < 3/2$, and $k \leq 3$, any randomized algorithm that approximates by a factor of α, the size of the optimal cover for the* Set Cover *problem with success probability at least $2/3$ requires $\widetilde{\Omega}(mn)$ probes.*

For simplicity, we assume that the algorithm has the knowledge of our construction (which may only strengthens our lower bounds); this includes I^* and $\mathcal{D}(I^*)$, along with their representation via ELTOF and SETOF. The objective of the algorithm is simply to distinguish them. Since we are distinguishing a distribution of instances $\mathcal{D}(I^*)$ against a single instance I^*, we may individually upper bound the probability that each probe-answer pair reveals the modified part of the instance, then apply the union bound directly. However, establishing such a bound requires a certain set of properties that we obtain through a careful design of I^* and **genModifiedInst**. We remark that our approach shows the hardness of *distinguishing* instances with with different cover sizes. That

is, our lower bound on the probe complexity also holds for the problem of approximating the size of the minimum set cover (without explicitly finding one).

Lastly, in Section 3.5.2 we provide a construction utilizing Theorem 3.5.2 to extend Corollary 3.5.3, establish the following theorem on lower bounds for larger minimum set cover sizes.

Theorem 3.5.4. *For any sufficiently small approximation factor $\alpha \leq 1.01$ and $k = O(m/\log n)$, any randomized algorithm that computes an α-approximation to the* Set Cover *problem with success probability at least 0.99 requires $\widetilde{\Omega}(mn/k)$ probes.*

3.5.1 Lower bound for small optimal value

3.5.1.1 Construction of the Median Instance I^*.

Let $\mathcal{F}$ be a collection of m sets such that (independently for each set-element pair (S, e)) S contains e with probability $1 - p_0$, where $p_0 = \sqrt{\frac{9 \log m}{n}}$ (note that since we assume $\log m \leq n/c$ for large enough c, we can assume that $p_0 \leq 1/2$). Equivalently, we may consider the incidence matrix of this instance: each entry is either 0 (indicating $e \notin S$) with probability p_0, or 1 (indicating $e \in S$) otherwise. We write $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$ denoting the collection of sets obtained from this construction.

Definition 3.5.5 (Median instance)**.** *An instance of* Set Cover*, I, is a* median instance *if it satisfies all the following properties.*

(a) No two sets cover all the elements. (The size of its minimum set cover is at least 3.)

(b) For any two sets the number of elements not covered by the union of these sets is at most $18 \log m$.

(c) The intersection of any two sets has size at least $n/8$.

(d) For any pair of elements e, e', the number of sets S s.t. $e \in S$ but $e' \notin S$ is at least $\frac{m\sqrt{9 \log m}}{4\sqrt{n}}$.

(e) For any triple of sets S, S_1 and S_2, $|(S_1 \cap S_2) \setminus S| \leq 6\sqrt{n \log m}$.

(f) For each element, the number of sets that do not contain that element is at most $6m\sqrt{\frac{\log m}{n}}$.

Lemma 3.5.6. *There exists a median instance I^* satisfying all properties from Definition 3.5.5. In fact, with high probability, an instance drawn from the distribution in which $\mathbf{Pr}[e \in S] = 1 - p_0$ independently at random, satisfies the median properties.*

The proof of the lemma follows from standard applications of concentration bounds. Specifically, it follows from the union bound and Lemmas 3.5.7–3.5.12 below.

Lemma 3.5.7. *With probability at least $1 - m^{-1}$ over $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$, the size of the minimum set cover of the instance $(\mathcal{F}, \mathcal{U})$ is greater than 2.*

Proof. The probability that an element $e \in \mathcal{U}$ is covered by two sets selected from $\mathcal{F}$ is at most:

$$\mathbf{Pr}[e \in S_1 \cup S_2] = 1 - p_0^2 = 1 - \frac{9 \log m}{n}.$$

Thus, the probability that $S_1 \cup S_2$ covers all elements in $\mathcal{U}$ is at most $(1 - \frac{9 \log m}{n})^n < m^{-9}$. Applying the union bound, with probability at least $1 - m^{-1}$ the size of optimal set cover is greater than 2. □

Lemma 3.5.8. *Let S_1 and S_2 be two sets in $\mathcal{F}$ where $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$. Then with probability at least $1 - m^{-1}$, $|\mathcal{U} \setminus (S_1 \cup S_2)| \leq 18 \log m$.*

Proof. For an element e, $\mathbf{Pr}[e \notin S_1 \cup S_2] = p_0^2 = \frac{9 \log m}{n}$. So, $\mathbb{E}[|\mathcal{U} \setminus (S_1 \cup S_2)|] = 9 \log m$. By Chernoff bound, $\mathbf{Pr}[|\mathcal{U} \setminus (S_1 \cup S_2)| \geq 18 \log m]$ is at most $e^{-9 \log m/3} \leq m^{-3}$. Thus with probability at least $1 - m^{-1}$, for any pair of sets in $\mathcal{F}$, the number of element not covered by their union is at most $18 \log m$. □

Lemma 3.5.9. *Let S_1 and S_2 be two sets in $\mathcal{F}$ where $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$. Then $|S_1 \cap S_2| \geq n/8$ with probability at least $1 - m^{-1}$.*

Proof. For each element e, it is either covered by both S_1, S_2, one of S_1, S_2 or none of them. Since $p_0 \leq 1/2$, the probability that an element is covered by both sets is greater than other cases, i.e., $\mathbf{Pr}[e \in S_1 \cap S_2] > 1/4$. Thus, $\mathbb{E}[|\mathcal{U} \setminus (S_1 \cap S_2)|] > n/4$. By Chernoff bound, $\mathbf{Pr}[|\mathcal{U} \setminus (S_1 \cap S_2)| \leq n/8]$ is exponentially small. Thus with probability at least $1 - m^{-1}$, the intersection of any pairs of sets in $\mathcal{F}$ is greater than $n/8$. □

Lemma 3.5.10. *Suppose that $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$ and let e, e' be two elements in $\mathcal{U}$. With probability at least $1 - m^{-1}$, the number of sets $S \in \mathcal{F}$ such that $e \in S$ but $e' \notin S$ is at least $\frac{m\sqrt{9 \log m}}{4\sqrt{n}}$.*

Proof. For each set S, $\mathbf{Pr}[e \in S \text{ and } e' \notin S] = (1 - p_0)p_0 \geq p_0/2$. This implies that the expected number of S satisfying the condition for e and e' is at least $\frac{m}{2} \cdot \sqrt{\frac{9 \log m}{n}}$ and by Chernoff bound, the probability that the number of sets containing e but not e' is less than $\frac{m\sqrt{9 \log m}}{4\sqrt{n}}$ is exponentially small. Thus with probability at least $1 - m^{-1}$ property (d) holds for any pair of elements in $\mathcal{U}$. □

Lemma 3.5.11. *Suppose that $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$ and let S_1, S_2 and S be sets in $\mathcal{F}$. With probability at least $1 - n^{-1}$, $|(S_1 \cap S_2) \setminus S| \leq 6\sqrt{n \log m}$.*

Proof. For each element e, $\mathbf{Pr}[e \in (S_1 \cap S_2) \setminus S] = (1 - p_0)^2 p_0 \leq p_0$. This implies that the expected size of $(S_1 \cap S_2) \setminus S$ is less than $\sqrt{9n \log m}$ and by Chernoff bound, the probability that $|(S_1 \cap S_2) \setminus S| \geq 6\sqrt{n \log m}$ is exponentially small. Thus with probability at least $1 - m^{-1}$ property (e) holds for any sets S_1, S_2 and S in $\mathcal{F}$. □

Lemma 3.5.12. *For each element, the number of sets that do not contain the element is at most $6m\sqrt{\frac{\log m}{n}}$.*

Proof. For each element e, $\mathbf{Pr}_S[e \notin S] = p_0$. This implies that $\mathbb{E}_S(|\{S \mid e \notin S\}|)$ is less than $m\sqrt{\frac{9 \log m}{n}}$ and by Chernoff bound, the probability that $|\{S \mid e \notin S\}| \geq 2m\sqrt{\frac{9 \log m}{n}}$ is exponentially small. Thus with probability at least $1 - m^{-1}$ property (f) holds for any element $e \in \mathcal{U}$. □

3.5.1.2 Distribution $\mathcal{D}(I^*)$ of Modified Instances I' Derived from I^*.

Fix a median instance I^*. We now show that we may perform $O(\log m)$ swap operations on I^* so that the size of the minimum set cover in the modified instance becomes 2. Moreover, its incidence matrix differs from that of I^* in $O(\log m)$ entries. Consequently, the number of probes to EltOf and SetOf that induce different answers from those of I^* is also at most $O(\log m)$.

We define $\mathcal{D}(I^*)$ as the distribution of instances I' generated from a median instance I^* by **genModifiedInst**(I^*) given below in Figure 3-8 as follows. Assume that $I^* = (\mathcal{U}, \mathcal{F})$. We select two different sets S_1, S_2 from $\mathcal{F}$ uniformly at random; we aim to turn these two sets into a set cover. To do so, we swap out some of the elements in S_2 and bring in the uncovered elements. For each uncovered element e, we pick an element $e' \in S_2$ that is also covered by S_1. Next, consider the candidate set that we may exchange its e with $e' \in S_2$:

Definition 3.5.13 (Candidate set)**.** *For any pair of elements e, e', the* candidate set *of (e, e') are all sets that contain e but not e'. The collection of candidate sets of (e, e') is denoted by* $\mathsf{Candidate}(e, e')$*. Note that* $\mathsf{Candidate}(e, e') \neq \mathsf{Candidate}(e', e)$ *(in fact, these two collections are disjoint).*

genModifiedInst$(I^* = (\mathcal{U}, \mathcal{F}))$:
$\mathcal{M} \leftarrow \emptyset$
pick two *different* sets S_1, S_2 from $\mathcal{F}$
    uniformly at random
for each $e \in \mathcal{U} \setminus (S_1 \cup S_2)$ **do**
    pick $e' \in (S_1 \cap S_2) \setminus \mathcal{M}$ uniformly at random
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{e'\}$
    Pick a random set S in $\mathsf{Candidate}(e, e')$
    $\mathsf{swap}(e, e')$ between S, S_2

Figure 3-8: The procedure of constructing a modified instance of I^*.

We choose a random set S from $\mathsf{Candidate}(e, e')$, and swap $e \in S$ with $e' \in S_2$ so that S_2 now contains e. We repeatedly apply this process for all initially uncovered e so that eventually S_1 and S_2 form a set cover. We show that the proposed algorithm, **genModifiedInst**, can indeed be executed without getting stuck.

Lemma 3.5.14. *The procedure* **genModifiedInst** *is well-defined under the precondition that the input instance I^* is a median instance.*

Proof. To carry out the algorithm, we must ensure that the number of the initially uncovered elements is at most that of the elements covered by both S_1 and S_2. This follows from the properties of median instances (Definition 3.5.5): $|\mathcal{U} \setminus (S_1 \cup S_2)| \leq 18 \log m$ by property (b), and that the size of the intersection of S_1 and S_2 is greater than $n/8$ by property (c). That is, in our construction there are sufficiently many possible choices for e' to be matched and swapped with each uncovered element e. Moreover, by property (d) there are plenty of candidate sets S for performing $\mathsf{swap}(e, e')$ with S_2. □

3.5.1.3 Bounding the Probability of Modification.

Let $\mathcal{D}(I^*)$ denote the distribution of instances generated by **genModifiedInst**(I^*). If an algorithm were to distinguish between I^* or $I' \sim \mathcal{D}(I^*)$, it must find some cell in the EltOf or SetOf tables that would have been modified by **genModifiedInst**, to confirm that **genModifiedInst** is indeed executed; otherwise it would make wrong decisions half of the time. We will show an additional

property of this distribution: none of the entries of EltOf and SetOf are significantly more likely to be modified during the execution of genModifiedInst. Consequently, no algorithm may strategically detect the difference between I^* or I' with the desired probability, unless the number of probes is asymptotically the reciprocal of the maximum probability of modification among any cells.

Define $P_{\mathsf{Elt-Set}} : \mathcal{U} \times \mathcal{F} \to [0,1]$ as the probability that an element is swapped by a set. More precisely, for an element $e \in \mathcal{U}$ and a set $S \in \mathcal{F}$, if $e \notin S$ in the median instance I^*, then $P_{\mathsf{Elt-Set}}(e,S) = 0$; otherwise, it is equal to the probability that S swaps e. We note that these probabilities are taken over $I' \sim \mathcal{D}(I^*)$ where I^* is a fixed median instance. That is, as per Figure 3-8, they correspond to the random choices of S_1, S_2, the random matching $\mathcal{M}$ between $\mathcal{U} \setminus (S_1 \cup S_2)$ and $S_1 \cap S_2$, and their random choices of choosing each candidate set S. We bound the values of $P_{\mathsf{Elt-Set}}$ via the following lemma.

Lemma 3.5.15. *For any $e \in \mathcal{U}$ and $S \in \mathcal{F}$, $P_{\mathsf{Elt-Set}}(e,S) \le \frac{4800 \log m}{mn}$ where the probability is taken over $I' \sim \mathcal{D}(I^*)$.*

Proof. Let S_1, S_2 denote the first two sets picked (uniformly at random) from $\mathcal{F}$ to construct a modified instance of I^*. For each element e and a set S such that $e \in S$ in the basic instance I^*,

$$\begin{aligned} P_{\mathsf{Elt-Set}}(e,S) = \mathbf{Pr}[S = S_2] \cdot \mathbf{Pr}[e \in S_1 \cap S_2] \\ \cdot \mathbf{Pr}[e \text{ matches to } \mathcal{U} \setminus (S_1 \cup S_2) \mid e \in S_1 \cap S_2] \\ + \mathbf{Pr}[S \notin \{S_1, S_2\}] \cdot \mathbf{Pr}[e \in S \setminus (S_1 \cup S_2) \mid e \in S] \\ \cdot \mathbf{Pr}[S \text{ swaps } e \text{ with } S_2 \mid e \in S \setminus (S_1 \cup S_2)]\,. \end{aligned}$$

where all probabilities are taken over $I' \sim \mathcal{D}(I^*)$. Next we bound each of the above six terms. Since we choose the sets S_1, S_2 randomly, $\mathbf{Pr}[S = S_2] = 1/m$. We bound the second term by 1. For the third term, since we pick a matching uniformly at random among all possible (maximum) matchings between $\mathcal{U} \setminus (S_1 \cup S_2)$ and $S_1 \cap S_2$, by symmetry, the probability that a certain element $e \in S_1 \cap S_2$ is in the matching is (by properties (b) and (c) of median instances),

$$\frac{|\mathcal{U} \setminus (S_1 \cup S_2)|}{|S_1 \cap S_2|} \le \frac{18 \log m}{n/8} = \frac{144 \log m}{n}.$$

We bound the fourth term by 1. To compute the fifth term, let d_e denote the number of sets in $\mathcal{F}$ that do not contain e. By property (f) of median instances, the probability that $e \in S$ is in $S \setminus (S_1 \cup S_2)$ given that $S \notin \{S_1, S_2\}$ is at most,

$$\frac{d_e(d_e - 1)}{(m-1)(m-2)} \le \frac{36m^2 \cdot \frac{\log m}{n}}{m^2/2} = \frac{72 \log m}{n}.$$

Finally for the last term, note that by symmetry, each pair of matched elements ee' is picked by genModifiedInst equiprobably. Thus, for any $e \in S \setminus (S_1 \cup S_2)$, the probability that each element $e' \in S_1 \cap S_2$ is matched to e is $\frac{1}{|S_1 \cap S_2|}$. By properties (c)–(e) of median instances, the last term is

at most

$$\sum_{e' \in (S_1 \cap S_2) \setminus S} \mathbf{Pr}[ee' \in \mathcal{M}] \cdot \frac{1}{|\mathsf{Candidate}(e,e')|} = |(S_1 \cap S_2) \setminus S| \cdot \frac{1}{|S_1 \cap S_2|} \cdot \frac{1}{\mathsf{Candidate}(e,e')}$$
$$\leq 6\sqrt{n \log m} \cdot \frac{1}{n/8} \cdot \frac{1}{\frac{m\sqrt{9 \log m}}{4\sqrt{n}}}$$
$$= \frac{64}{m}.$$

Therefore,

$$P_{\mathsf{Elt-Set}}(e,S) \leq \frac{1}{m} \cdot 1 \cdot \frac{144 \log m}{n} + 1 \cdot \frac{72 \log m}{n} \cdot \frac{64}{m} \leq \frac{4800 \log m}{mn}.$$

□

3.5.1.4 Proof of Theorem 3.5.2.

Now we consider a median instance I^*, and its corresponding family of modified sets $\mathcal{D}(I^*)$. To prove the promised lower bound for randomized protocols distinguishing I^* and $I' \sim \mathcal{D}(I^*)$, we apply Yao's principle and instead show that no deterministic algorithm $\mathcal{A}$ may determine whether the input is I^* or $I' \sim \mathcal{D}(I^*)$ with success probability at least 2/3 using $r = o(\frac{mn}{\log m})$ probes. Recall that if $\mathcal{A}$'s probe-answer history $\langle (q_1, a_1), \ldots, (q_r, a_r) \rangle$ when executed on I' is the same as that of I^*, then $\mathcal{A}$ must unavoidably return a wrong decision for the probability mass corresponding to I'. We bound the probability of this event as follows.

Lemma 3.5.16. *Let Q be the set of probes made by $\mathcal{A}$ on I^*. Let $I' \sim \mathcal{D}(I^*)$ where I^* is a given median instance. Then the probability that $\mathcal{A}$ returns different outputs on I^* and I' is at most $\frac{4800 \log m}{mn}|Q|$.*

Proof of Theorem 3.5.2. If $\mathcal{A}$ does not output correctly on I^*, the probability of success of $\mathcal{A}$ is less than 1/2; thus, we can assume that $\mathcal{A}$ returns the correct answer on I^*. This implies that $\mathcal{A}$ returns an incorrect solution on the fraction of $I' \sim I'(I^*)$ for which $\mathcal{A}(I^*) = \mathcal{A}(I')$. Now recall that the distribution in which we apply Yao's principle consists of I^* with probability 1/2, and drawn uniformly at random from $\mathcal{D}(I^*)$ also with probability 1/2. Then over this distribution, by Lemma 3.5.16,

$$\mathbf{Pr}[\mathcal{A} \text{ suceeds}] \leq 1 - \frac{1}{2}\mathbf{Pr}_{I' \sim \mathcal{D}(I^*)}[\mathcal{A}(I^*) = \mathcal{A}(I')]$$
$$\leq 1 - \frac{1}{2}\left(1 - \frac{4800 \log m}{mn}|Q|\right)$$
$$= \frac{1}{2} + \frac{2400 \log m}{mn}|Q|.$$

Thus, if the number of probes made by $\mathcal{A}$ is less than $\frac{mn}{14400 \log m}$, then the probability that $\mathcal{A}$ returns the correct answer over the input distribution is less than 2/3 and the proof is complete. □

3.5.2 Lower bound for large optimal value

Our construction of the median instance I^* and its associated distribution $\mathcal{D}(I^*)$ of modified instances also leads to the lower bound of $\widetilde{\Omega}(\frac{mn}{k})$ for the problem of computing an approximate solution to Set Cover. This lower bound matches the performance of our algorithm for large optimal value k and shows that it is tight for some range of value k, albeit it only applies to sufficiently small approximation factor $\alpha \leq 1.01$.

Proof overview. We construct a distribution over *compounds*: a compound is a Set Cover instance that consists of $t = \Theta(k)$ smaller instances $I_1, \ldots, I_t$, where each of these t instances is either the median instance I^* or a random modified instance drawn from $\mathcal{D}(I^*)$. By our construction, a large majority of our distribution is composed of compounds that contains at least $0.2t$ modified instances I_i such that, any deterministic algorithm $\mathcal{A}$ must fail to distinguish I_i from I^* when it is only allowed to make a small number of probes. A deterministic $\mathcal{A}$ can safely cover these modified instances with three sets, incurring a cost (sub-optimality) of $0.2t$. Still, $\mathcal{A}$ may choose to cover such an I_i with two sets to reduce its cost, but it then must err on a different compound where I_i is replaced with I^*. We track down the trade-off between the amount of cost that $\mathcal{A}$ saves on these compounds by covering these I_i's with two sets, and the amount of error on other compounds its scheme incurs. $\mathcal{A}$ is allowed a small probability δ to make errors, which we then use to upper-bound the expected cost that $\mathcal{A}$ may save, and conclude that $\mathcal{A}$ still incurs an expected cost of $0.1t$ overall. We apply Yao's principle (for algorithms with errors) to obtain that randomized algorithms also incur an expected cost of $0.05t$, on compounds with optimal solution size $k \in [2t, 3t]$, yielding the impossibility result for computing solutions with approximation factor $\alpha = \frac{k+0.1t}{k} > 1.01$ when given insufficient probes.

3.5.2.1 Overall Lower Bound Argument

Compounds. Consider the median instance I^* and its associated distribution $\mathcal{D}(I^*)$ of modified instances for Set Cover with n elements and m sets, and let $t = \Theta(k)$ be a positive integer parameter. We define a *compound* $\mathfrak{I} = \mathfrak{I}(I_1, I_2, \ldots, I_t)$ as a set structure instance consisting of t median or modified instances $I_1, I_2, \ldots, I_t$, forming a set structure $(\mathcal{U}^t, \mathcal{F}^t)$ of $n' \triangleq nt$ elements and $m' \triangleq mt$ sets, in such a way that each instance I_i occupies separate elements and sets. Since the optimal solution to each instance I_i is 3 if $I_i = I^*$, and 2 if I_i is any modified instance, the optimal solution for the compound is $2t$ plus the number of occurrences of the median instance; this optimal objective value is always $\Theta(k)$.

Random distribution over compounds. Employing Yao's principle, we construct a distribution $\mathfrak{D}$ of compounds $\mathfrak{I}(I_1, I_2, \ldots, I_t)$: it will be applied against any deterministic algorithm $\mathcal{A}$ for computing an approximate minimum set cover, which is allowed to err on at most a δ-fraction of the compounds from the distribution (for some small constant $\delta > 0$). For each $i \in [t]$, we pick $I_i = I^*$ with probability $c/\binom{m}{2}$ where $c > 2$ is a sufficiently large constant. Otherwise, simply draw a random modified instance $I_i \sim \mathcal{D}(I^*)$. We aim to show that, in expectation over $\mathfrak{D}$, $\mathcal{A}$ must output a solution that of size $\Theta(t)$ more than the optimal set cover size of the given instance $\mathfrak{I} \sim \mathfrak{D}$.

$\mathcal{A}$ frequently leaves many modified instances undetected. Consider an instance $\mathfrak{I}$ containing

at least $0.95t$ modified instances. These instances constitute at least a 0.99-fraction of $\mathfrak{D}$: the expected number of occurrences of the median instance in each compound is only $c/\binom{m}{2} \cdot t = O(t/m^2)$, so by Markov's inequality, the probablity that there are more than $0.05t$ median instances is at most $O(1/m^2) < 0.01$ for large m. We make use of the following useful lemma, whose proof is deferred to Section 3.5.2.2. In what follow, we say that the algorithm "distinguishes" or "detects the difference" between I_i and I^* if it makes a probe that induces different answers, and thus may deduce that one of I_i or I^* cannot be the input instance. In particular, if $I_i = I^*$ then detecting the difference between them would be impossible.

Lemma 3.5.17. *Fix $M \subseteq [t]$ and consider the distribution over compounds $\mathfrak{I}(I_1, \ldots, I_t)$ with $I_i \sim \mathcal{D}(I^*)$ for $i \in M$ and $I_i = I^*$ for $i \notin M$. If $\mathcal{A}$ makes at most $o(\frac{mnt}{\log m})$ probes to $\mathfrak{I}$, then it may detect the differences between I^* and at least $0.75t$ of the modified instances $\{I_i\}_{i \in M}$, with probability at most 0.01.*

We apply this lemma for any $|M| \geq 0.95t$ (although the statement holds for any M, even vacuously for $|M| < 0.75t$). Thus, for $0.99 \cdot 0.99 > 0.98$-fraction of $\mathfrak{D}$, $\mathcal{A}$ fails to identify, for at least $0.95t - 0.75t = 0.2t$ modified instances I_i in $\mathfrak{I}$, whether it is a median instance or a modified instance. Observe that the probe-answer history of $\mathcal{A}$ on such $\mathfrak{I}$ would not change if we were to replace any combination of these $0.2t$ modified instances by copies of I^*. Consequently, if the algorithm were to correctly cover $\mathfrak{I}$ by using two sets for some of these I_i, it must unavoidably err (return a non-cover) on the compound where these I_i's are replaced by copies of the median instance.

Charging argument. We call a compound $\mathfrak{I}$ *tough* if $\mathcal{A}$ does not err on $\mathfrak{I}$, and $\mathcal{A}$ fails to detect at least $0.2t$ modified instances; denote by $\mathfrak{D}^{\mathsf{tough}}$ the conditional distribution of $\mathfrak{D}$ restricted to tough instances. For tough $\mathfrak{I}$, let $\mathsf{cost}(\mathfrak{I})$ denote the number of modified instances I_i that the algorithm decides to cover with three sets. That is, for each tough compound $\mathfrak{I}$, $\mathsf{cost}(\mathfrak{I})$ measures how far the solution returned by $\mathcal{A}$ is, from the optimal set cover size. Then, there are at least $0.2t - \mathsf{cost}(\mathfrak{I})$ modified instances I_i that $\mathcal{A}$ chooses to cover with only two sets despite not being able to verify whether $I_i = I^*$ or not. Let $R_{\mathfrak{I}}$ denote the set of the indices of these modified instances, so $|R_{\mathfrak{I}}| = 0.2t - \mathsf{cost}(\mathfrak{I})$. By doing so, $\mathcal{A}$ then errs on the *replaced compound* $r(\mathfrak{I}, R_{\mathfrak{I}})$, denoting the compound similar to $\mathfrak{I}$, except that each modified instance I_i for $i \in R_{\mathfrak{I}}$ is replaced by I^*. In this event, we say that the tough compound $\mathfrak{I}$ *charges* the replaced compound $r(\mathfrak{I}, R_{\mathfrak{I}})$ via $R_{\mathfrak{I}}$. Recall that the total error of $\mathcal{A}$ is δ: this quantity upper-bounds the total probability masses of charged instances, which we will then manipulate to obtain a lower bound on $\mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}}[\mathsf{cost}(\mathfrak{I})]$.

Instances must share optimal solutions for R to charge the same replaced instance. Observe that many tough instances may charge to the same replaced instance: we must handle these duplicities. First, consider two tough instances $\mathfrak{I}^1 \neq \mathfrak{I}^2$ charing the same $\mathfrak{I}_{\mathsf{r}} = r(\mathfrak{I}^1, R) = r(\mathfrak{I}^2, R)$ via the same $R = R_{\mathfrak{I}^1} = R_{\mathfrak{I}^2}$. As $\mathfrak{I}^1 \neq \mathfrak{I}^2$ but $r(\mathfrak{I}^1, R) = r(\mathfrak{I}^2, R)$, these tough instances differ on some modified instances with indices in R. Nonetheless, the probe-answer histories of $\mathcal{A}$ operating on $\mathfrak{I}^1$ and $\mathfrak{I}^2$ must be the same as their instances in R are both indistinguishable from I^* by the deterministic $\mathcal{A}$. Since $\mathcal{A}$ does not err on tough instances (by definition), both tough $\mathfrak{I}^1$ and $\mathfrak{I}^2$ must share the same optimal set cover on every instance in R. Consequently, for each fixed R, only tough instances that have the same optimal solution for modified instances in R may charge the

same replaced instance via R.

Charged instance is much heavier than charging instances combined. By our construction of $\mathfrak{I}(I_1, \ldots, I_t)$ drawn from $\mathfrak{D}$, $\Pr[I_i = I^*] = c/\binom{m}{2}$ for the median instance. On the other hand, $\sum_{j=1}^{\ell} \Pr[I_i = I^j] \le (1 - c/\binom{m}{2}) \cdot (1/\binom{m}{2}) < 1/\binom{m}{2}$ for modified instances $I^1, \ldots, I^\ell$ sharing the same optimal set cover, because they are all modified instances constructed to have the two sets chosen by **genModifiedInst** as their optimal set cover: each pair of sets is chosen uniformly with probability $1/\binom{m}{2}$. Thus, the probability that I^* is chosen is more than c times the total probability that any I^j is chosen. Generalizing this observation, we consider tough instances $\mathfrak{I}^1, \mathfrak{I}^2, \ldots, \mathfrak{I}^\ell$ charging the same $\mathfrak{I}_{\mathsf{r}}$ via R, and bound the difference in probabilities that $\mathfrak{I}_{\mathsf{r}}$ and any $\mathfrak{I}^j$ are drawn. For each index in R, it is more than c times more likely for $\mathfrak{D}$ to draw the median instance, rather than any modified instances of a fixed optimal solution. Then, for the replaced compound $\mathfrak{I}_{\mathsf{r}}$ that $\mathcal{A}$ errs, $p(\mathfrak{I}_{\mathsf{r}}) \ge c^{|R|} \cdot \sum_{j=1}^{\ell} p(\mathfrak{I}^j)$ (where p denotes the probability mass in $\mathfrak{D}$, not in $\mathfrak{D}^{\mathsf{tough}}$). In other words, the probability mass of the replaced instance charged via R is always at least $c^{|R|}$ times the total probability mass of the charging tough instances.

Bounding the expected cost using δ. In our charging argument by tough instances above, we only bound the amount of charges on the replaced instances via a fixed R. As there are up to 2^t choices for R, we scale down the total amount charged to a replaced instance by a factor of 2^t, so that $\sum_{\mathsf{tough}\ \mathfrak{I}} c^{|R_{\mathfrak{I}}|} p(\mathfrak{I})/2^t$ lower bounds the total probability mass of the replaced instances that $\mathcal{A}$ errs.

Let us first focus on the conditional distribution $\mathfrak{D}^{\mathsf{tough}}$ restricted to tough instances. Recall that at least a $(0.98 - \delta)$-fraction of the compounds in $\mathfrak{D}$ are tough: $\mathcal{A}$ fails to detect differences between $0.2t$ modified instances from the median instance with probability 0.98, and among these compounds, $\mathcal{A}$ may err on at most a δ-fraction. So in the conditional distribution $\mathfrak{D}^{\mathsf{tough}}$ over tough instances, the individual probability mass is scaled-up to $p^{\mathsf{tough}}(\mathfrak{I}) \le \frac{p(\mathfrak{I})}{0.98-\delta}$. Thus,

$$\frac{\sum_{\mathsf{tough}\ \mathfrak{I}} c^{|R_{\mathfrak{I}}|} p(\mathfrak{I})}{2^t} \ge \frac{\sum_{\mathsf{tough}\ \mathfrak{I}} c^{|R_{\mathfrak{I}}|} (0.98 - \delta) p^{\mathsf{tough}}(\mathfrak{I})}{2^t} = \frac{(0.98 - \delta) \mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}^{\mathsf{tough}}} \left[c^{|R_{\mathfrak{I}}|} \right]}{2^t}.$$

As the probability mass above cannot exceed the total allowed error δ, we have

$$\frac{\delta}{0.98 - \delta} \cdot 2^t \ge \mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}^{\mathsf{tough}}} \left[c^{|R_{\mathfrak{I}}|} \right] \ge \mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}^{\mathsf{tough}}} \left[c^{0.2t - \mathsf{cost}(\mathfrak{I})} \right] \ge c^{0.2t - \mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}^{\mathsf{tough}}} [\mathsf{cost}(\mathfrak{I})]},$$

where Jensen's inequality is applied in the last step above. So,

$$\mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}^{\mathsf{tough}}} [\mathsf{cost}(\mathfrak{I})] \ge 0.2t - \frac{t + \log \frac{\delta}{0.98-\delta}}{\log c} = \left(0.2 - \frac{1}{\log c} \right) t - \frac{\log \frac{\delta}{0.98-\delta}}{\log c} \ge 0.11t,$$

for sufficiently large c (and m) when choosing $\delta = 0.02$.

We now return to the expected cost over the entire distribution $\mathfrak{I}$. For simplicity, define $\mathsf{cost}(\mathfrak{I}) = 0$ for any non-tough $\mathfrak{I}$. This yields $\mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}} [\mathsf{cost}(\mathfrak{I})] \ge (0.98 - \delta) \mathbf{E}_{\mathfrak{I} \sim \mathfrak{D}^{\mathsf{tough}}} [\mathsf{cost}(\mathfrak{I})] \ge (0.98 - \delta) \cdot 0.11t \ge 0.1t$, establishing the expected cost of any deterministic $\mathcal{A}$ with probability of error at most 0.02 over $\mathfrak{D}$.

Establishing the lower bound for randomized algorithms. Lastly, we apply Yao's principle[6] to obtain that, for any randomized algorithm with error probability $\delta/2 = 0.01$, its expected cost under the worst input is at least $\frac{1}{2} \cdot 0.1t = 0.05t$. Recall now that our cost here lower-bounds the sub-optimality of the computed set cover (that is, the algorithm uses at least cost more sets to cover the elements than the optimal solution does). Since our input instances have optimal solution $k \in [2t, 3t]$ and the randomized algorithm returns a solution with cost at least $0.05t$ in expectation, it achieves an approximation factor of no better than $\alpha = \frac{k+0.05t}{k} > 1.01$ with $o(\frac{mnt}{\log m})$ probes. Theorem 3.5.4 then follows, noting the substitution of our problem size: $\frac{mnt}{\log m} = \frac{(m'/t)(n'/t)t}{\log(m'/t)} = \Theta(\frac{m'n'}{k' \log m'})$.

3.5.2.2 Proof of Lemma 3.5.17

First, we recall the following result from Lemma 3.5.16 for distinguishing between I^* and a random $I' \sim \mathcal{D}(I^*)$.

Corollary 3.5.18. *Let q be the number of probes made by $\mathcal{A}$ on $I_i \sim \mathcal{D}(I^*)$ over n elements and m sets, where I^* is a median instance. Then the probability that $\mathcal{A}$ detects a difference between I_i and I^* in one of its probes is at most $\frac{4800q \log m}{mn}$.*

Marbles and urns. Fix a compound $\mathfrak{I}(I_1, \ldots, I_t)$. Let $s \triangleq \frac{mn}{4800 \log m}$, and then consider the following, entirely different, scenario. Suppose that we have t urns, where each urn contains s marbles. In the i^{th} urn, in case I_i is a modified instance, we put in this urn one red marble and $s-1$ white marbles; otherwise if $I_i = I^*$, we put in s white marbles. Observe that the probability of obtaining a red marble by drawing q marbles from a single urn *without replacement* is exactly q/s (for $q \leq s$). Now, we will relate the probability of drawing red marbles to the probability of successfully distinguishing instances. We emphasize that we are only comparing the probabilities of events for the sake of analysis, and we do not imply or suggest any direct analogy between the events themselves.

Corollary 3.5.18 above bounds the probability that the algorithm successfully distinguishes a modified instance I_i from I^* with $\frac{4800q \log m}{mn} = q/s$. Then, the probability of distinguishing between I_i and I^* using q probes, is bounded from above by the probability of obtaining a red marble after drawing q marbles from an urn. Consequently, the probability that the algorithm distinguishes $3t/4$ instances is bounded from above by the probability of drawing the red marbles from at least $3t/4$ urns. Hence, to prove that the event of Lemma 3.5.17 occurs with probability at most 0.01, it is sufficient to upper-bound the probability that an algorithm obtains $3t/4$ red marbles by 0.01.

Consider an instance of t urns; for each urn $i \in [t]$ corresponding to a modified instance I_i, exactly one of its s marbles is red. An algorithm may draw marbles from each urn, one by one without replacement, for potentially up to s times. By the principle of deferred decisions, the red marble is equally likely to appear in any of these s draws, independent of the events for other urns. Thus, we can create a tuple of t random variables $\mathcal{T} = (T_1, \ldots, T_t)$ such that for each $i \in [t]$, T_i is chosen uniformly at random from $\{1, \ldots, s\}$. The variable T_i represents the number of draws required to obtain the red marble in the i^{th} urn; that is, only the T_i^{th} draw from the i^{th} urn finds

[6]Here we use the Monte Carlo version where the algorithm may err, and use cost instead of the time complexity as our measure of performance. See, e.g., Proposition 2.6 in [MR95] and the description therein.

the red marble from that urn. In case I_i is a median instance, we simply set $T_i = s + 1$ indicating that the algorithm never detects any difference as I_i and I^* are the same instance.

We now show the following two lemmas in order to bound the number of red marbles the algorithm may encounter throughout its execution.

Lemma 3.5.19. *Let $b > 3$ be a fixed constant and define $\mathcal{T}_{\text{high}} = \{i \mid T_i \geq \frac{s}{b}\}$. If $t \geq 14b$, then $|\mathcal{T}_{\text{high}}| \geq (1 - \frac{2}{b})t$ with probability at least 0.99.*

Proof. Let $\mathcal{T}_{\text{low}} = \{1, \ldots, t\} \setminus \mathcal{T}_{\text{high}}$. Notice that for the i^{th} urn, $\mathbf{Pr}[i \in \mathcal{T}_{\text{low}}] < \frac{1}{b}$ independently of other urns, and thus $|\mathcal{T}_{\text{low}}|$ is stochastically dominated by $\mathrm{B}(t, \frac{1}{b})$, the binomial distribution with t trials and success probability $\frac{1}{b}$. Applying Chernoff bound, we obtain

$$\mathbf{Pr}\left[|\mathcal{T}_{\text{low}}| \geq \frac{2t}{b}\right] \leq e^{-\frac{t}{3b}} < 0.01.$$

Hence, $|\mathcal{T}_{\text{high}}| \geq t - \frac{2t}{b} = (1 - \frac{2}{b})t$ with probability at least 0.99, as desired. □

Lemma 3.5.20. *If the total number of draws made by the algorithm is less than $(1 - \frac{3}{b})\frac{st}{b}$, then with probability at least 0.99, the algorithm will not obtain* red *marbles from at least $\frac{t}{b}$ urns.*

Proof. If the total number of such draws is less than $(1 - \frac{3}{b})\frac{st}{b}$, then the number of draws from at least $\frac{3t}{b}$ urns is less than $\frac{s}{b}$ each. Assume the condition of Lemma 3.5.19: for at least $(1 - \frac{2}{b})t$ urns, $T_i \geq \frac{s}{b}$. That is, the algorithm will not encounter a red marble if it makes less than $\frac{s}{b}$ draws from such an urn. Then, there are at least $\frac{t}{b}$ urns with $T_i \geq \frac{s}{b}$ from which the algorithm makes less than $\frac{s}{b}$ draws, and thus does not obtain a red marble. Overall this event holds with probability at least 0.99 due to Lemma 3.5.19. □

We substitute $b = 4$ and assume sufficiently large t. Suppose that the deterministic algorithm makes less than $(1 - \frac{3}{4})\frac{st}{4} = \frac{st}{16}$ probes, then for a fraction of 0.99 of all possible tuples $\mathcal{T}$, there are $t/4$ instances I_i that the algorithm fails to detect their differences from I^*: the probability of this event is lower-bounded by that of the event where the red marbles from those corresponding urns i are not drawn. Therefore, the probability that the algorithm makes probes that detect differences between I^* and more than $3t/4$ instances I_i's is bounded by 0.01, concluding our proof of Lemma 3.5.17.

3.6 Generalized Lower Bounds for the Set Cover Problem

In this section we generalize the approach of Section 3.5 and prove our main lower bound result (Theorem 3.5.1) for the number of probes required for approximating with factor α the size of an optimal solution to the SetCover problem, where the input instance contains m sets, n elements, and a minimum set cover of size k. The structure of our proof is largely the same as the simplified case, but the definitions and the details of our analysis will be more complicated. The size of the minimum set cover of the median instance will instead be at least $\alpha k + 1$, and **genModifiedInst** reduces this down to k. We now aim to prove the following statement which implies the lower bound in Theorem 3.5.1.

Theorem 3.6.1. *Let k be the size of an optimal solution of I^* such that $1 < \alpha \leq \log n$ and $2 \leq k \leq \left(\frac{n}{16\alpha \log m}\right)^{\frac{1}{4\alpha+1}}$. Any algorithm that distinguishes whether the input instance is I^* or belongs to $\mathcal{D}(I^*)$ with probability of success at least $2/3$ requires $\widetilde{\Omega}(m(\frac{n}{k})^{1/(2\alpha)})$ probes.*

3.6.1 Construction of the median instance I^*

Let $\mathcal{F}$ be a collection of m sets such that independently for each set-element pair (S, e), S contains e with probability $1 - p_0$, where we modify the probability to $p_0 = \left(\frac{8(\alpha k+2)\log m}{n}\right)^{1/(\alpha k)}$. We start by proving some inequalities involving p_0 that will be useful later on, which hold for any k in the assumed range.

Lemma 3.6.2. *For $2 \leq k \leq \left(\frac{n}{16\alpha \log m}\right)^{\frac{1}{4\alpha+1}}$, we have that*

(a) $1 - p_0 \geq p_0^{k/4}$,

(b) $p_0^{k/4} \leq 1/2$,

(c) $\frac{p_0^k}{(1-p_0)^2} \leq \left(\frac{8(\alpha k+2)\log m}{n}\right)^{\frac{1}{2\alpha}}$.

Proof. Recall as well that $\alpha > 1$. In the given range of k, we have $k^{4\alpha} \leq \frac{n}{16\alpha k \log m} \leq \frac{n}{8(\alpha k+2)\log m}$ because $k\alpha \geq 2$. Thus

$$p_0 = \left(\frac{8(\alpha k+2)\log m}{n}\right)^{\frac{1}{\alpha k}} \leq \left(\frac{1}{k^{4\alpha}}\right)^{\frac{1}{\alpha k}} = k^{-4/k}.$$

Next, rewrite $k^{-4/k} = e^{-\frac{4\ln k}{k}}$ and observe that $\frac{4\ln k}{k} \leq \frac{4}{e} < 1.5$. Since $e^{-x} \leq 1 - \frac{x}{2}$ for any $x < 1.5$, we have $p_0 \leq e^{-\frac{4\ln k}{k}} < 1 - \frac{2\ln k}{k}$. Further, $p_0^{k/4} \leq e^{-\ln k} = 1/k$. Hence $p_0 + p_0^{k/4} \leq 1 - \frac{2\ln k}{k} + \frac{1}{k} \leq 1$, implying the first statement.

The second statement easily follows as $p_0^{k/4} \leq 1/k \leq 1/2$ since $k \geq 2$. For the last statement, we make use of the first statement:

$$\frac{p_0^k}{(1-p_0)^2} \leq \frac{p_0^k}{(p_0^{k/4})^2} = p_0^{k/2} = \left(\frac{8(\alpha k+2)\log m}{n}\right)^{\frac{1}{2\alpha}}$$

which completes the proof of the lemma. □

Next, we give the new, generalized definition of median instances.

Definition 3.6.3 (Median instance). *An instance of* Set Cover, *$I = (\mathcal{U}, \mathcal{F})$, is a* median instance *if it satisfies all the following properties.*

(a) No αk sets cover all the elements. (The size of its minimum set cover is greater than αk.)

(b) The number of uncovered elements of the union of any k sets is at most $2np_0^k$.

(c) For any pair of elements e, e', the number of sets $S \in \mathcal{F}$ s.t. $e \in S$ but $e' \notin S$ is at least $(1 - p_0)p_0 m/2$.

(d) For any collection of k sets $S_1, \cdots, S_k$, $|S_k \cap (S_1 \cup \cdots \cup S_{k-1})| \geq (1 - p_0)(1 - p_0^{k-1})n/2$.

(e) For any collection of $k+1$ sets $S, S_1, \cdots, S_k$, $|(S_k \cap (S_1 \cup \cdots \cup S_{k-1})) \setminus S| \leq 2p_0(1-p_0)(1-p_0^{k-1})n$.

(f) For each element, the number of sets that do not contain the element is at most $(1 + \frac{1}{k})p_0 m$.

Lemma 3.6.4. *For* $k \leq \min\{\sqrt{\frac{m}{27 \ln m}}, (\frac{n}{16\alpha \log m})^{\frac{1}{4\alpha+1}}\}$, *there exists a median instance* I^* *satisfying all the median properties from Definition 3.6.3. In fact, most of the instances constructed by the described randomized procedure satisfy the median properties.*

Proof. The lemma follows from applying the union bound on the results of Lemmas 3.6.5–3.6.10. □

The proofs of the Lemmas 3.6.5–3.6.10 follow from standard applications of concentration bounds. We include them here for the sake of completeness.

Lemma 3.6.5. *With probability at least* $1 - m^{-2}$ *over* $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$, *the size of the minimum set cover of the instance* $(\mathcal{F},\mathcal{U})$ *is at least* $\alpha k + 1$.

Proof. The probability that an element $e \in \mathcal{U}$ is covered by a specific collection of αk sets in $\mathcal{F}$ is at most $1 - p_0^{\alpha k} = 1 - \frac{8(\alpha k+2)\log m}{n}$. Thus, the probability that the union of the αk sets covers all elements in $\mathcal{U}$ is at most $(1 - \frac{8(\alpha k+2)\log m}{n})^n < m^{-8(\alpha k+2)}$. Applying the union bound, with probability at least $1 - m^{-2}$ the size of an optimal set cover is at least $\alpha k + 1$. □

Lemma 3.6.6. *With probability at least* $1 - m^{-2}$ *over* $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$, *any collection of* k *sets has at most* $2np_0^k$ *uncovered elements.*

Proof. Let $S_1, \cdots, S_k$ be a collection of k sets from $\mathcal{F}$. For each element $e \in \mathcal{U}$, the probability that e is not covered by the union of the k sets is p_0^k. Thus,

$$\mathbb{E}[|\mathcal{U} \setminus (S_1 \cup \cdots \cup S_k)|] = p_0^k n \geq p_0^{\alpha k} n = 8(\alpha k + 2)\log m.$$

By Chernoff bound,

$$\mathbf{Pr}\Big[|\mathcal{U} \setminus (S_1 \cup \cdots \cup S_k)| \geq 2p_0^k n\Big] \leq e^{-\frac{p_0^k n}{3}} \leq e^{-(\alpha k+2)\log m} \leq m^{-k-2}.$$

Thus with probability at least $1 - m^{-2}$, for any collection of k sets in $\mathcal{F}$, the nnumber of uncovered elements by the union of the sets is at most $2p_0^k n$. □

Lemma 3.6.7. *Suppose that* $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$ *and let* e, e' *be two elements in* $\mathcal{U}$. *Given* $k \leq \left(\frac{n}{16\alpha \log m}\right)^{\frac{1}{4\alpha+1}}$, *with probability at least* $1 - m^{-2}$, *the number of sets* $S \in \mathcal{F}$ *such that* $e \in S$ *but* $e' \notin S$ *is at least* $mp_0(1 - p_0)/2$.

Proof. For each set S, $\mathbf{Pr}[e \in S \text{ and } e' \notin S] = (1 - p_0)p_0$. This implies that the expected number of such sets S satisfying the condition for e and e' is

$$p_0(1 - p_0)m \geq p_0 \cdot p_0^{k/4} \cdot m \geq p_0^{\alpha k} n = 8(\alpha k + 2)\log m$$

by Lemma 3.6.2 and $m \geq n$. By Chernoff bound, the probability that the number of sets containing e but not e' is less than $mp_0(1 - p_0)/2$ is at most

$$e^{-\frac{p_0(1-p_0)m}{8}} \leq e^{-(\alpha k+2)\log m} \leq m^{-\alpha k-2}.$$

Thus with probability at least $1 - m^{-2}$ property (c) holds for any pair of elements in $\mathcal{U}$. □

Lemma 3.6.8. *Suppose that $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$ and let $S_1, \cdots, S_k$ be k different sets in $\mathcal{F}$. Given $k \leq \left(\frac{n}{16\alpha \log m}\right)^{\frac{1}{4\alpha+1}}$, with probability at least $1-m^{-2}$, $|S_k \cap (S_1 \cup \cdots \cup S_{k-1})| \geq (1-p_0)(1-p_0^{k-1})n/2$.*

Proof. For each element e, $\mathbf{Pr}[e \in S_k \cap (S_1 \cup \cdots \cup S_{k-1})] = (1 - p_0)(1 - p_0^{k-1})$. This implies that the expected size of $S_k \cap (S_1 \cup \cdots \cup S_{k-1})$ is

$$(1 - p_0)(1 - p_0^{k-1})n \geq p_0^{k/4} \cdot p_0^{k/4} \cdot n \geq p_0^{\alpha k} n = 8(\alpha k + 2) \log m.$$

by Lemma 3.6.2. By Chernoff bound, the probability that $|S_k \cap (S_1 \cup \cdots \cup S_{k-1})| \leq (1 - p_0)(1 - p_0^{k-1})n/2$ is at most

$$e^{-\frac{(1-p_0)(1-p_0^{k-1})n}{8}} \leq e^{-(\alpha k+2)\log m} \leq m^{-\alpha k-2}.$$

Thus with probability at least $1 - m^{-2}$ property (d) holds for any sets $S_1, \cdots, S_k$ in $\mathcal{F}$. □

Lemma 3.6.9. *Suppose that $\mathcal{F} \sim \mathcal{I}(\mathcal{U}, p_0)$ and let $S_1, \cdots, S_k$ and S be $k + 1$ different sets in $\mathcal{F}$. Given $k \leq \left(\frac{n}{16\alpha \log m}\right)^{\frac{1}{4\alpha+1}}$, with probability at least $1 - m^{-2}$, $|(S_k \cap (S_1 \cup \cdots \cup S_{k-1})) \setminus S| \leq 2p_0(1 - p_0)(1 - p_0^{k-1})n$.*

Proof. For each element e, $\mathbf{Pr}[e \in (S_k \cap (S_1 \cup \cdots \cup S_{k-1})) \setminus S] = p_0(1 - p_0)(1 - p_0^{k-1})$. Then,

$$\begin{aligned}
\mathbb{E}(|(S_k \cap (S_1 \cup \cdots \cup S_{k-1})) \setminus S|) &= p_0(1 - p_0)(1 - p_0^{k-1})n \\
&\geq p_0 \cdot p_0^{k/4} \cdot p_0^{k/4} \\
&\geq p_0^{\alpha k} n \\
&= 8(\alpha k + 2) \log m
\end{aligned}$$

by Lemma 3.6.2. By Chernoff bound, the probability that $|(S_k \cap (S_1 \cup \cdots \cup S_{k-1})) \setminus S| \geq 2p_0(1 - p_0)(1 - p_0^{k-1})n$ is

$$e^{-\frac{p_0(1-p_0)(1-p_0^{k-1})n}{3}} \leq e^{-2(\alpha k+2)\log m} \leq m^{-2\alpha k-4}.$$

Thus with probability at least $1 - m^{-2}$ property (e) holds for any sets $S_1, \cdots, S_k$ and S in $\mathcal{F}$. □

Lemma 3.6.10. *Given that $k \leq \left(\frac{n}{16\alpha \log m}\right)^{\frac{1}{4\alpha+1}}$, for each element, the number of sets that do not contain the element is at most $(1 + \frac{1}{k})p_0 m$.*

Proof. First, note that $k \leq \left(\frac{n}{16\alpha \log m}\right)^{\frac{1}{4\alpha+1}} \leq \sqrt{\frac{m}{27 \ln m}}$ as $m \geq n$ and $\alpha \geq 1$.

Next, for each element e, $\mathbf{Pr}_{S \sim \mathcal{F}}[e \notin S] = p_0$. This implies that $\mathbb{E}_S(|\{S \mid e \notin S\}|) = p_0 m$. By Chernoff bound, the probability that $|\{S \mid e \notin S\}| \geq (1 + \frac{1}{k})p_0 m$ is at most $e^{\frac{-mp_0}{3k^2}}$. Now if $k \geq \log n$, then $p_0 \geq 1/e$ and thus this probability would be at most $\exp(\frac{-m}{3ek^2}) \leq m^{-3}$ for any $k \leq \sqrt{\frac{m}{27 \ln m}}$.

Otherwise, we have that the above probability is at most $\exp(\frac{-mn^{-1/\alpha k}}{3\log^2 n}) \leq \exp(\frac{-m^{1-1/\alpha k}}{3\log^2 m}) \leq m^{-3}$ given $m \geq n$ and sufficiently large n. Thus with probability at least $1 - m^{-2}$ property (f) holds for any element $e \in \mathcal{U}$. □

3.6.2 Distribution $\mathcal{D}(I^*)$ of the modified instances derived from I^*

Fix a median instance I^*. We now show that we may perform $\tilde{O}(n^{1-1/\alpha}k^{1/\alpha})$ swap operations on I^* so that the size of the minimum set cover in the modified instance becomes k. So, the number of probes to EltOf and SetOf that induce different answers from those of I^* is at most $\widetilde{O}(n^{1-1/\alpha}k^{1/\alpha})$. We define $\mathcal{D}(I^*)$ as the distribution of instances I' that is generated from a median instance I^* by **genModifiedInst**(I^*) given below in Figure 3-9. The main difference from the simplified version are that we now select k different sets to turn them into a set cover, and the swaps may only occur between S_k and the candidates.

genModifiedInst$(I^* = (\mathcal{U}, \mathcal{F}))$:
  $\mathcal{M} \leftarrow \emptyset$
  pick k *different* sets $S_1, \cdots S_k$ from $\mathcal{F}$
    uniformly at random
  for each $e \in \mathcal{U} \setminus (S_1 \cup \cdots \cup S_k)$ **do**
    pick $e' \in (S_k \cap (S_1 \cup \cdots \cup S_{k-1})) \setminus \mathcal{M}$
      uniformly at random
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{ee'\}$
    pick a random set S in Candidate(e, e')
    swap(e, e') between S, S_k

Figure 3-9: The procedure of constructing a modified instance of I^*.

Lemma 3.6.11. *The procedure* **genModifiedInst** *is well-defined under the precondition that the input instance I^* is a median instance.*

Proof. To carry out the algorithm, we must ensure that the number of the initially uncovered elements is at most that of the elements covered by both S_k and some other set from $S_1, \ldots, S_{k-1}$. Since I^* is a median instance, by properties (b) and (d) from Definition 3.6.3, these values satisfy $|\mathcal{U} \setminus (S_1 \cup \cdots \cup S_k)| \leq 2p_0^k n$ and $|S_k \cap (S_1 \cup \cdots \cup S_{k-1})| \geq (1 - p_0)(1 - p_0^{k-1})n/2$, respectively. By Lemma 3.6.2, $p_0^{k/4} \leq 1/2$. Using this and Lemma 3.6.2 again,

$$(1 - p_0)(1 - p_0^{k-1})n/2 \geq p_0^{k/4} \cdot p_0^{k/4} \cdot n/2 \geq p_0^{k/2} n/2 \geq 2p_0^k n.$$

That is, in our construction there are sufficiently many possible choices for e' to be matched and swapped with each uncovered element e. Moreover, since I^* is a median instance, $|\text{Candidate}(e, e')| \geq (1 - p_0)p_0 m/2$ (by property (c)), and there are plenty of candidates for each swap. □

3.6.2.1 Bounding the Probability of Modification

Similarly to the simplified case, define $P_{\text{Elt}-\text{Set}} : \mathcal{U} \times \mathcal{F} \to [0, 1]$ as the probability that an element is swapped by a set, and upper bound it via the following lemma.

Lemma 3.6.12. *For any $e \in \mathcal{U}$ and $S \in \mathcal{F}$, $P_{\mathsf{Elt-Set}}(e, S) \leq \frac{64p_0^k}{(1-p_0)^2 m}$ where the probability is taken over the random choices of $I' \sim \mathcal{D}(I^*)$.*

Proof. Let $S_1, \ldots, S_k$ denote the first k sets picked (uniformly at random) from $\mathcal{F}$ to construct a modified instance of I^*. For each element e and a set S such that $e \in S$ in the basic instance I^*,

$$\begin{aligned} P_{\mathsf{Elt-Set}}(e, S) = \mathbf{Pr}[S = S_k] \cdot \mathbf{Pr}\big[e \in \cup_{i\in[k-1]} S_i \mid e \in S_k\big] \\ \cdot \mathbf{Pr}\big[e \text{ matches to } \mathcal{U} \setminus (\cup_{i\in[k]} S_i) \mid e \in S_k \cap (\cup_{i\in[k-1]} S_i)\big] \\ + \mathbf{Pr}[S \notin \{S_1, \ldots, S_k\}] \cdot \mathbf{Pr}\big[e \in S \setminus (\cup_{i\in[k]} S_i) \mid e \in S\big] \\ \cdot \mathbf{Pr}[S \text{ swaps } e \text{ with } S_k \mid e \in S \setminus (S_1 \cup \cdots \cup S_k)], \end{aligned}$$

where all probabilities are taken over $I' \sim \mathcal{D}(I^*)$. Next we bound each of the above six terms. Clearly, since we choose the sets $S_1, \cdots, S_k$ randomly, $\mathbf{Pr}[S = S_k] = 1/m$. We bound the second term by 1. Next, by properties (b) and (d) of median instances, the third term is at most

$$\frac{|\mathcal{U} \setminus (\cup_{i\in[k]} S_i)|}{|S_k \cap (\cup_{i\in[k-1]} S_i)|} \leq \frac{2p_0^k n}{(1-p_0)(1-p_0^{k-1})\frac{n}{2}} \leq \frac{4p_0^k}{(1-p_0)^2}.$$

We bound the fourth term by 1. Let d_e denote the number of sets in $\mathcal{F}$ that do not contain e. Using property (f) of median instances, the fifth term is at most

$$\frac{d_e(d_e-1)\cdots(d_e-k+1)}{(m-1)(m-2)\cdots(m-k)} \leq \left(\frac{d_e}{m-1}\right)^k \leq \left(\frac{(1+1/k)p_0 m}{m(1-\frac{1}{k+1})}\right)^k \leq e^2 p_0^k,$$

Finally for the last term, note that by symmetry, each pair of matched elements ee' is picked by **genModifiedInst** equiprobably. Thus, for any $e \in S \setminus (S_1 \cup \cdots \cup S_k)$, the probability that each element $e' \in S_k \cap (S_1 \cup \cdots \cup S_{k-1})$ is matched to e is $\frac{1}{|S_k \cap (S_1 \cup \cdots \cup S_{k-1})|}$. By properties (c)-(e) of median instances, the last term is at most

$$\begin{aligned} &\sum_{e' \in (S_k \cap (\cup_{i\in[k-1]} S_i)) \setminus S} \mathbf{Pr}\big[ee' \in \mathcal{M}\big] \cdot \mathbf{Pr}\big[(S, S_k) \text{ swaps } (e, e')\big] \\ &\leq |(S_k \cap (\cup_{i\in[k-1]} S_i)) \setminus S| \cdot \frac{1}{|S_k \cap (\cup_{i\in[k-1]} S_i)|} \cdot \frac{1}{|\mathsf{Candidate}(e, e')|} \\ &\leq 2p_0(1-p_0)(1-p_0^{k-1})n \cdot \frac{1}{(1-p_0)(1-p_0^{k-1})n/2} \cdot \frac{1}{p_0(1-p_0)m/2} \\ &\leq \frac{8}{(1-p_0)m}. \end{aligned}$$

Therefore,

$$\begin{aligned} P_{\mathsf{Elt-Set}}(e,S) &\leq \frac{1}{m}\cdot 1\cdot\frac{4p_0^k}{(1-p_0)^2} + 1\cdot e^2 p_0^k\cdot\frac{8}{(1-p_0)m} \\ &\leq \frac{4p_0^k}{(1-p_0)^2} + \frac{60p_0^k}{(1-p_0)m} \\ &\leq \frac{64p_0^k}{(1-p_0)^2 m}. \end{aligned}$$

□

3.6.3 Proof of Theorem 3.6.1

The remaining part of our proof follows that of the simplified version almost exactly.

Proof of Theorem 3.6.1. Applying the same argument as that of Lemma 3.5.16, we derive that the probability that $\mathcal{A}$ returns different outputs on I^* and I' is at most

$$\begin{aligned} \mathbf{Pr}\big[\mathcal{A}(I^*) \neq \mathcal{A}(I')\big] &\leq \sum_{t=1}^{|Q|} \mathbf{Pr}[\mathsf{ans}_{I^*}(q_t) \neq \mathsf{ans}_{I'}(q_t)] \\ &\leq \sum_{t=1}^{|Q|} P_{\mathsf{Elt-Set}}(e(q_t), S(q_t)) \\ &\leq \frac{64p_0^k}{m(1-p_0)^2}|Q|, \end{aligned}$$

via the result of Lemma 3.6.12. Then, over the distribution in which we applied Yao's lemma, we have

$$\begin{aligned} \mathbf{Pr}[\mathcal{A}\text{ succeeds}] &\leq 1 - \frac{1}{2}\mathbf{Pr}_{I'\sim\mathcal{D}(I^*)}[\mathcal{A}(I^*) = \mathcal{A}(I')] \\ &\leq 1 - \frac{1}{2}\left(1 - \frac{64p_0^k}{m(1-p_0)^2}|Q|\right) \\ &= \frac{1}{2} + \frac{32p_0^k}{m(1-p_0)^2}|Q| \\ &\leq \frac{1}{2} + \frac{32}{m}\left(\frac{8(k\alpha+2)\log m}{n}\right)^{\frac{1}{2\alpha}}|Q| \end{aligned}$$

where the last inequality follows from Lemma 3.6.2. Thus, if the number of probes made by $\mathcal{A}$ is less than $\frac{m}{192}(\frac{n}{8(k\alpha+2)\log m})^{1/(2\alpha)}$, then the probability that $\mathcal{A}$ returns the correct answer over the input distribution is less than 2/3 and the proof is complete. □

Chapter 4

Fractional Set Cover

4.1 Overview of Fractional Set Cover in the Streaming Model

Recall that in the Set Cover problem, the goal is to find the minimum size *set cover* of $\mathcal{U}$, i.e., a collection of sets in $\mathcal{F}$ whose union is $\mathcal{U}$: the input consists of a set (universe) of n elements $\mathcal{U} = \{e_1, \cdots, e_n\}$ and a collection of m sets $\mathcal{F} = \{S_1, \cdots, S_m\}$. The LP relaxation of Set Cover (called SetCover-LP) is also well-studied. It is a continuous relaxation of the problem where each set $S \in \mathcal{F}$ can be selected "fractionally", i.e., assigned a number x_S from $[0, 1]$, such that for each element e its "fractional coverage" $\sum_{S:e \in S} x_S$ is at least 1, and the sum $\sum_S x_S$ is minimized.

A natural $\ln n$-approximation greedy algorithm of Set Cover, which in each iteration picks the *best* remaining set, is widely used and known to be the best possible under $\mathbf{P} \neq \mathbf{NP}$ [LY94, Fei98, RS97, AMS06, Mos15, DS14]. However, the greedy algorithm is sequential in nature and does not perform efficiently in the standard models developed for *massive data analysis*; in particular, in the *streaming* model. In streaming Set Cover [SG09], the ground set $\mathcal{U}$ is stored in the memory, the sets $S_1, \cdots, S_m$ are stored consecutively in a read-only repository and the algorithm can only access the sets by performing sequential scans (or passes) over the repository. Moreover, the amount of (read-write) memory available to the algorithm is much smaller than the input size (which can be as large as mn). The objective is to design a *space-efficient* algorithm that returns a (nearly)-optimal feasible cover of $\mathcal{U}$ after performing only a few passes over the data. Streaming Set Cover has witnessed a lot of developments in recent years, and tight upper and lower bounds are known, in both *low space* [ER16, CW16] and *low approximation* [DIMV14, HIMV16, AKL16, BEM17, Ass17] regimes.

Despite the above developments, the results for the *fractional* variant of the problem are still unsatisfactory. To the best of our knowledge, it is not known whether there exists an efficient and accurate algorithm for this problem that uses only a logarithmic (or even a *poly logarithmic*) number of passes. This state of affairs is perhaps surprising, given the many recent developments on fast LP solvers [KY14, You14, LS14, AO15b, AO15a, WRM16]. To the best of our knowledge, the only prior results on streaming Packing/Covering LPs were presented in paper [AG13], which studied the LP relaxation of Maximum Matching.

4.1.1 Our results

In this work, we present the first $(1+\varepsilon)$-approximation algorithm for the fractional Set Cover in the streaming model with constant number of passes. Our algorithm performs p passes over the data stream and uses $\widetilde{O}(mn^{O(\frac{1}{p\varepsilon})}+n)$ memory space to return a $(1+\varepsilon)$ approximate solution of the LP relaxation of Set Cover for positive parameter $\varepsilon \leq 1/2$.

We emphasize that similarly to the previous work on variants of Set Cover in streaming setting, our result also holds for the *edge arrival* stream in which the pair of (S_i, e_j) (edges) are stored in the read-only repository and all elements of a set are not necessarily stored consecutively.

4.1.2 Related work

Set Cover Problem. The Set Cover problem was first studied in the streaming model in [SG09], which presented an $O(\log n)$-approximation algorithm in $O(\log n)$ passes and using $\widetilde{O}(n)$ space. This approximation factor and the number of passes can be improved to $O(\log n)$ by adapting the greedy algorithm *thresholding* idea presented in [CKW10] . In the low space regime ($\widetilde{O}(n)$ space), Emek and Rosen [ER16] designed a *deterministic* single pass algorithm that achieves an $O(\sqrt{n})$-approximation. This is provably the best guarantee that one can hope for in a single pass even considering randomized algorithms. Later Chakrabarti and Wirth [CW16] generalized this result and provided a *tight* trade-off bounds for Set Cover in multiple passes. More precisely, they gave an $O(pn^{1/(p+1)})$-approximate algorithm in p-passes using $\widetilde{O}(n)$ space and proved that this is the best possible approximation ratio up to a factor of poly(p) in p passes and $\widetilde{O}(n)$ space.

A different line of work started by Demaine et al. [DIMV14] focused on designing a "low" approximation algorithm (between $\Theta(1)$ and $\Theta(\log n)$) in the smallest possible amount of space. In contrast to the results in the $\widetilde{O}(n)$ space regime, [DIMV14] showed that randomness is necessary: any constant pass deterministic algorithm requires $\Omega(mn)$ space to achieve constant approximation guarantee. Further, they provided a $O(4^p \log n)$-approximation algorithm that makes $O(4^p)$ passes and uses $\widetilde{O}(mn^{1/p}+n)$. Later Har-Peled et al. [HIMV16] improved the algorithm to a $2p$-pass $O(p\log n)$-approximation with memory space $\widetilde{O}(mn^{1/p}+n)$[1]. The result was further improved by Bateni et al. where they designed a p-pass algorithm that returns a $(1+\varepsilon)\log n$-approximate solution using $mn^{\Theta(1/p)}$ memory [BEM17].

As for the lower bounds, Assadi et al. [AKL16] presented a lower bound of $\Omega(mn/\alpha)$ memory for any single pass streaming algorithm that computes a α-approxime solution. For the problem of estimating the size of an optimal solution they prove $\Omega(mn/\alpha^2)$ memory lower bound. For both settings, they complement the results with matching tight upper bounds. Very recently, Assadi [Ass17] proved a lower bound for streaming algorithms with multiple passes which is tight up to polylog factors: any α-approximation algorithm for Set Cover requires $\Omega(mn^{1/\alpha})$ space, even if it is allowed polylog(n) passes over the stream, and even if the sets are arriving in a random order in the stream. Further, [Ass17] provided the matching upper bound: a $(2\alpha+1)$-pass algorithm that computes a $(\alpha+\varepsilon)$-approximate solution in $\widetilde{O}(\frac{mn^{1/\alpha}}{\varepsilon^2}+\frac{n}{\varepsilon})$ memory (assuming exponential

[1] In streaming model, space complexity is of interest and one can assume exponentital computation power. In this case the algorithms of [DIMV14, HIMV16] save a factor of $\log n$ in the approximation ratio.

computational resource).

Max Cover Problem. The first result on streaming Max k-Cover showed how to compute a $(1/4)$-approximate solution in one pass using $\widetilde{O}(kn)$ space [SG09]. It was improved by Badanidiyuru et al. [BMKK14] to a $(1/2-\varepsilon)$-approximation algorithm that requires $\widetilde{O}(n/\varepsilon)$ space. Moreover, their algorithm works for a more general problem of Submodular Maximization with cardinality constraints. This result was later generalized for the problem of non-monotone submodular maximization under constraints beyond cardinality [CGQ15]. Recently, McGregor and Vu [MV17] and Bateni et al. [BEM17] independently obtained single pass $(1-1/e-\varepsilon)$-approximation with $\widetilde{O}(m/\varepsilon^2)$ space. On the lower bound side, [MV17] showed a lower bound of $\widetilde{\Omega}(m)$ for constant pass algorithm whose approximation is better than $(1-1/e)$. Moreover, [Ass17] proved that any streaming $(1-\varepsilon)$-approximation algorithm of Max k-Cover in polylog(n) passes requires $\widetilde{\Omega}(m/\varepsilon^2)$ space even on random order streams and the case $k=O(1)$. This bound is also complemented by the $\widetilde{O}(mk/\varepsilon^2)$ and $\widetilde{O}(m/\varepsilon^3)$ algorithms of [BEM17, MV17]. For more detailed survey of the results on streaming Max k-Cover refer to [BEM17, MV17, Ass17].

Covering/Packing LPs. The study of LPs in streaming model was first discussed in the work of Ahn and Guha [AG13] where they used *multiplicative weights update* (MWU) based techniques to solve the LP relaxation of Maximum (Weighted) Matching problem. They used the fact that MWU returns a near optimal fractional solution with small size support: first they solve the fractional matching problem, then solve the actual matching only considering the edges in the support of the returned fractional solution.

Our algorithm is also based on the MWU method, which is one of the main key techniques in designing fast approximation algorithms for Covering and Packing LPs [PST95, You95, GK07, AHK12]. We note that the MWU method has been previously studied in the context of *streaming* and *distributed* algorithms, leading to efficient algorithms for a wide range of graph optimization problems [AG13, BGM14, AG15].

For a related problem, *covering integer LP* (covering ILP), Assadi et al. [AKL16] designed a one-pass streaming algorithm that estimates the optimal solution of $\{\min \mathbf{c}^\top \mathbf{x} \mid \mathbf{A}^\top \mathbf{x} \geq \mathbf{b}, \mathbf{x} \in \{0,1\}^n\}$ within a factor of α using $\widetilde{O}(\frac{mn}{\alpha^2} \cdot b_{\max} + m + n \cdot b_{\max})$ where $b_{\max}$ denotes the largest entry of $\mathbf{b}$. In this problem, they assume that columns of $\mathbf{A}$, constriants, are given one by one in the stream.

In a different regime, [DKM05] studied approximating the feasibility LP in streaming model with additive approximation. Their algorithm performs two passes and is most efficient when the input is dense.

4.1.3 Our techniques

Preprocessing. Let k denote the value of the optimal solution. The algorithm starts by picking a uniform *fractional* vector (each entry of value $O(\frac{k}{m})$) which covers all frequently occurring elements (those appearing in $\Omega(\frac{m}{k})$ sets), and updates the uncovered elements in one pass. This step considerably reduces the memory usage as the uncovered elements have now lower occurrence (roughly $\frac{m}{k}$). Note that we do not need to assume the knowledge of the correct value k: in parallel we try all powers of $(1+\varepsilon)$, denoting our guess by ℓ.

Multiplicative Weight Update. To cover the remaining elements, we employ the MWU framework and show how to implement it in the streaming setting. In each iteration of MWU, we have a probability distribution $\mathbf{p}$ corresponding to the constraints (elements) and we need to satisfy the *average* covering constraint. More precisely, we need an *oracle* that assigns values to x_S for each set S so that $\sum_S p_S x_S \geq 1$ subject to $\|\mathbf{x}\|_1 \leq \ell$, where p_S is the sum of probabilities of the elements in the set S. Then, the algorithm needs to update $\mathbf{p}$ according to the amount each element has been covered by the oracle's solution. The simple greedy realization of the oracle can be implemented in the streaming setting efficiently by computing all p_S while reading the stream in one pass, then choosing the heaviest set (i.e., the set with largest p_S) and setting its x_S to ℓ. This approach works, except that the number of rounds T required by the MWU framework is large. In fact, $T = \Omega(\frac{\phi \log n}{\varepsilon^2})$, where ϕ is the width parameter (the maximum amount an oracle solution may over-cover an element), which is $\Theta(\ell)$ in this naïve realization. Next, we show how to decrease T in two steps.

Step 1. A first hope would be that there is a more efficient implementation of the oracle which gives a better width parameter. Nonetheless, no matter how the oracle is implemented, if all sets in $\mathcal{F}$ contain a fixed element e, then the width is inevitably $\Omega(\ell)$. This observation implies that we need to work with a different set system that has small width, but at the same time, it has the same objective value as of the optimal solution. Consequently, we consider the *extended set system* where we replace $\mathcal{F}$ with all subsets of the sets in $\mathcal{F}$. This extended system preserves the optimality, and under this system we may avoid over-covering elements and obtain $T = O(\log n)$ (for constant ε).

In order to turn a solution in our set system into a solution in the extended set system with small width, we need to remove the repeated elements from the sets in the solution so that every covered element appears exactly once, and thereby getting constant width. However, as a side effect, this reduces the total weight of the solution ($\sum_{S \in \mathtt{sol}} p_S x_S$), and thus the average covering constraint might not be satisfied anymore. In fact, we need to come up with a guarantee that, on one hand, is preserved under the pruning step, and on the other hand, implies that the solution has large enough total weight

Therefore, to fulfill the average constraint under the pruning step, the oracle must instead solve the *maximum coverage* problem: given a budget, choose sets to cover the largest (fractional) amount of elements. We first show that this problem can be solved approximately via the MWU framework using the simple oracle that picks the heaviest set, but this MWU algorithm still requires T passes over the data. To improve the number of passes, we perform *element sampling* and apply the MWU algorithm to find an approximate maximum coverage of a small number of sampled elements, whose subproblem can be stored in memory. Fortunately, while the number of fractional solutions to maximum coverage is unbounded, by exploiting the structure of the solutions returned by the MWU method, we can limit the number of plausible solutions of this oracle and approximately solve the average constraint, thereby reducing the space usage to $\widetilde{O}(m)$ for a $O(\frac{\log n}{\varepsilon^2})$-pass algorithm.

Step 2. To further reduce the number of required passes, we observe that the weights of the constraints change slowly. Thus, in a single pass, we can sample the elements for multiple rounds in advance, and then perform rejection (sub-)sampling to obtain an unbiased set of samples for each subsequent round. This will lead to a streaming algorithm with p passes and $mn^{O(1/p)}$ space.

Extension. We also extend our result to handle general covering LPs. More specifically, in the LP relaxation of Set Cover, maximize $\mathbf{c}^\top \mathbf{x}$ subject to $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$, $\mathbf{A}$ has entries from $\{0, 1\}$ whereas entries of $\mathbf{b}$ and $\mathbf{c}$ are all ones. If the non-zero entries instead belong to a range $[1, M]$, we increase the number of sampled elements by poly(M) to handle discrepancies between coefficients, leading to a poly(M)-multiplicative overhead in the space usage.

4.2 MWU Framework for Fractional Set Cover Streaming Algorithm

In this section, we present a basic streaming algorithm that computes a $(1+\varepsilon)$-approximate solution of the LP-relaxation of Set Cover for any $\varepsilon > 0$ via the MWU framework. We will, in the next section, improve it into an efficient algorithm that achieves the claimed $O(p)$ passes and $\widetilde{O}(mn^{1/p})$ space complexity.

SetCover-LP $\langle\langle$*Input:* $\mathcal{U}, \mathcal{F}\rangle\rangle$

$$\begin{aligned} \text{minimize} \quad & \sum_{S \in \mathcal{F}} x_S & \\ \text{subject to} \quad & \sum_{S: e \in S} x_S \geq 1 & \forall e \in \mathcal{U} \\ & x_S \geq 0 & \forall S \in \mathcal{F} \end{aligned}$$

Figure 4-1: LP relaxation of Set Cover.

Let $\mathcal{U}$ and $\mathcal{F}$ be the ground set of elements and the collection of sets, respectively, and recall that $|\mathcal{U}| = n$ and $|\mathcal{F}| = m$. Let $\mathbf{x} \in \mathbb{R}^m$ be a vector indexed by the sets in $\mathcal{F}$, where x_S denotes the value assigned to the set S. Our goal is to compute an approximate solution to the LP in Figure 4-1. Throughout the analysis we assume $\varepsilon \leq 1/2$, and ignore the case where some element never appears in any set, as it is easy to detect in a single pass that no cover is valid. For ease of reading, we write $\widetilde{O}$ and $\widetilde{\Theta}$ to hide polylog$(m, n, \frac{1}{\varepsilon})$ factors.

Outline of the algorithm. Let k denote the optimal objective value, and $0 < \varepsilon \leq 1/2$ be a parameter. The outline of the algorithm is shown in **fracSetCover** (Figure 4-2). This algorithm makes calls to the subroutine **feasibilityTest**, that given a parameter ℓ, with high probability, either returns a solution of objective value at most $(1+\varepsilon/3)\ell$, or detects that the optimal objective value exceeds ℓ. Consequently, we may search for the right value of ℓ by considering all values in $\{(1+\varepsilon/3)^i \mid 0 \leq i \leq \log_{1+\varepsilon/3} n\}$. As for some value of ℓ it holds that $k \leq \ell \leq k(1+\varepsilon/3)$, we obtain a solution of size $(1+\varepsilon/3)\ell \leq (1+\varepsilon/3)(1+\varepsilon/3)k \leq (1+\varepsilon)k$ which gives an approximation factor $(1+\varepsilon)$. This whole process of searching for k increases the space complexity of the algorithm by at most a multiplicative factor of $\log_{1+\varepsilon/3} n \approx \frac{3 \log n}{\varepsilon}$.

The **feasibilityTest** subroutine employs the multiplicative weights update method (MWU) which is described next.

fracSetCover(ε):

▷ Finds a feasible $(1+\varepsilon)$-approximate solution in $O(\frac{\log n}{\varepsilon})$ iterations

for $\ell \in \{(1+\varepsilon/3)^i \mid 0 \le i \le \log_{1+\varepsilon/3} n\}$ **do in parallel**: $\mathbf{x}_\ell \leftarrow$ **feasibilityTest**$(\ell, \varepsilon/3)$

return x_{ℓ^*} where $\ell^* \leftarrow \min\{\ell : x_\ell \text{ is not INFEASIBLE}\}$

Figure 4-2: **fracSetCover** returns a $(1+\varepsilon)$-approximate solution of SetCover-LP, where **feasibilityTest** is an algorithm that returns a solution of objective value at most $(1+\varepsilon/3)\ell$ when $\ell \ge k$.

4.2.1 Preliminaries of the MWU method for solving covering LPs

In the following, we describe the MWU framework. The claims presented here are standard results of the MWU method. For more details, see e.g. Section 3 of [AHK12]. Note that we introduce the general LP notation as it simplifies the presentation later on.

Let $\mathbf{Ax} \ge \mathbf{b}$ be a set of linear constraints, and let $\mathcal{P} \triangleq \{\mathbf{x} \in \mathbb{R}^m : \mathbf{x} \ge \mathbf{0}\}$ be the polytope of the non-negative orthant. For a given error parameter $0 < \beta < 1$, we would like to solve an approximate version of the feasibility problem by doing one of the following:

- Compute $\hat{\mathbf{x}} \in \mathcal{P}$ such that $\mathbf{A}_i\hat{\mathbf{x}} - b_i \ge -\beta$ for *every* constraint i.
- Correctly report that the system $\mathbf{Ax} \ge \mathbf{b}$ has no solution in $\mathcal{P}$.

The MWU method solves this problem assuming the existence of the following oracle that takes a distribution $\mathbf{p}$ over the constraints and finds a solution $\hat{\mathbf{x}}$ that satisfies the constraints on average over $\mathbf{p}$.

Definition 4.2.1. *Let $\phi \ge 1$ be a width parameter and $0 < \beta < 1$ be an error parameter. A* $(1, \phi)$-bounded $(\beta/3)$-approximate oracle *is an algorithm that takes as input a distribution $\mathbf{p}$ and does one of the following:*

- *Returns a solution $\hat{\mathbf{x}} \in \mathcal{P}$ satisfying*
  - $\mathbf{p}^\top\mathbf{A}\hat{\mathbf{x}} \ge \mathbf{p}^\top\mathbf{b} - \beta/3$, *and*
  - $\mathbf{A}_i\hat{\mathbf{x}} - b_i \in [-1, \phi]$ *for* every *constraint i.*
- *Correctly reports that the inequality $\mathbf{p}^\top\mathbf{Ax} \ge \mathbf{p}^\top\mathbf{b}$ has no solution in $\mathcal{P}$.*

The MWU algorithm for solving covering LPs involves T rounds. It maintains the (non-negative) weight of each constraint in $\mathbf{Ax} \ge \mathbf{b}$, which measures how much it has been satisfied by the solutions chosen so far. Let $\mathbf{w}^t$ denote the weight vector at the beginning of round t, and initialize the weights to $\mathbf{w}^1 \triangleq \mathbf{1}$. Then, for rounds $t = 1, \ldots, T$, define the probability vector $\mathbf{p}^t$ proportional to those weights $\mathbf{w}^t$, and use the oracle above to find a solution $\mathbf{x}^t$. If the oracle reports that the system $\mathbf{p}^\top\mathbf{Ax} \ge \mathbf{p}^\top\mathbf{b}$ is infeasible, the MWU algorithm also reports that the original system $\mathbf{Ax} \ge \mathbf{b}$ is infeasible, and terminates. Otherwise, define the cost vector incurred by $\mathbf{x}^t$ as $\mathbf{m}^t \triangleq \frac{1}{\phi}(\mathbf{Ax} - \mathbf{b})$, then update the weights so that $w_i^{t+1} \triangleq w_i^t(1 - \beta m_i^t/6)$ and proceed to the next round. Finally, the algorithm returns the average solution $\overline{\mathbf{x}} = \frac{1}{T}\sum_{t=1}^{T}\mathbf{x}^t$.

The MWU theorem (e.g., Theorem 3.5 of [AHK12]) shows that $T = O(\frac{\phi \log n}{\beta^2})$ is sufficient to correctly solve the problem, yielding $\mathbf{A}_i\hat{\mathbf{x}} - b_i \geq -\beta$ for every constraint, where n is the number of constraints. In particular, the algorithm requires T calls to the oracle.

Theorem 4.2.2 (MWU Theorem [AHK12]). *For every $0 < \beta < 1, \phi \geq 1$ the* MWU *algorithm either solves the* Feasibility-Covering-LP *problem up to an additive error of β (i.e., solves $\mathbf{A}_i\mathbf{x} - b_i \geq -\beta$ for every i) or correctly reports that the LP is infeasible, making only $O(\frac{\phi \log n}{\beta^2})$ calls to a $(1, \phi)$-bounded $\beta/3$-approximate oracle of the LP.*

4.2.2 Streaming MWU-based algorithm for Fractional Set Cover

Setting up our MWU algorithm. As described in the overview, we wish to solve, as a subroutine, the decision variant of SetCover-LP known as Feasibility-SC-LP given in Figure 4-3a, where the parameter ℓ serves as the guess for the optimal objective value.

Feasibility-SC-LP $\langle\langle$*Input:* $\mathcal{U}, \mathcal{F}, \ell\rangle\rangle$

$$\sum_{S \in \mathcal{F}} x_S \leq \ell$$
$$\sum_{S: e \in S} x_S \geq 1 \qquad \forall e \in \mathcal{U}$$
$$x_S \geq 0 \qquad \forall S \in \mathcal{F}$$

(a) LP relaxation of Feasibility Set Cover.

Feasibility-Covering-LP $\langle\langle$*Input:* $\mathbf{A}, \mathbf{b}, \mathbf{c}, \ell\rangle\rangle$

$\mathbf{c}^\top \mathbf{x} \leq \ell$	(objective value)
$\mathbf{A}\mathbf{x} \geq \mathbf{b}$	(covering)
$\mathbf{x} \geq \mathbf{0}$	(non-negativity)

(b) LP relaxation of the Feasibility Covering problem.

Figure 4-3: LP relaxations of the feasibility variant of set cover and general covering problems.

To follow the conventional notation for solving LPs in the MWU framework, consider the more standard form of covering LPs denoted as Feasibility-Covering-LP given in Figure 4-3b. For our purpose, $\mathbf{A}_{n \times m}$ is the element-set incidence matrix indexed by $\mathcal{U} \times \mathcal{F}$; that is, $A_{e,S} = 1$ if $e \in S$, and $A_{e,S} = 0$ otherwise. The vectors $\mathbf{b}$ and $\mathbf{c}$ are both all-ones vectors indexed by $\mathcal{U}$ and $\mathcal{F}$, respectively. We emphasize that, unconventionally for our system $\mathbf{A}\mathbf{x} \geq \mathbf{b}$, there are n constraints (i.e. elements) and m variables (i.e. sets).

Employing the MWU approach for solving covering LPs, we define the polytope

$$\mathcal{P}_\ell \triangleq \{\mathbf{x} \in \mathbb{R}^m : \mathbf{c}^\top \mathbf{x} \leq \ell \text{ and } \mathbf{x} \geq \mathbf{0}\}.$$

Observe that by applying the MWU algorithm to this polytope $\mathcal{P}$ and constraints $\mathbf{A}\mathbf{x} \geq \mathbf{b}$, we obtain a solution $\overline{\mathbf{x}} \in \mathcal{P}_\ell$ such that $\mathbf{A}_e\left(\frac{\overline{\mathbf{x}}}{1-\beta}\right) \geq \frac{b_e - \beta}{1-\beta} = 1 = b_e$, where $\mathbf{A}_e$ denotes the row of $\mathbf{A}$ corresponding to e. This yields a $(1 + O(\varepsilon))$-approximate solution for $\beta = O(\varepsilon)$.

Unfortunately, we cannot implement the MWU algorithm on the full input under our streaming context. Therefore, the main challenge is to implement the following two subtasks of the MWU algorithm in the streaming settings. First, we need to design an oracle that solves the average constraint in the streaming setting. Moreover, we need to be able to efficiently update the weights for the subsequent rounds.

Covering the common elements. Before we proceed to applying the MWU framework, we add a simple first step to our implementation of **feasibilityTest** (Figure 4-4) that will greatly reduce the amount of sapce required in implementing the MWU algorithm. This can be interpreted as the fractional version of Set Sampling described in [DIMV14]. In our subroutine, we partition the elements into the common elements that occur more frequently, which will be covered if we simply choose a uniform vector solution, and the rare elements that occur less frequently, for which we perform the MWU algorithm to compute a good solution. In one pass we can find all frequently occurring elements by counting the number of sets containing each element. The amount of required space to perform this task is $O(n \log m)$.

we call an element that appears in at least $\frac{m}{\alpha\ell}$ sets *common*, and we call it *rare* otherwise, where $\alpha = \Theta(\varepsilon)$. Since we are aiming for a $(1+\varepsilon)$-approximation, we can define $\mathbf{x}^{\mathsf{cmn}}$ as a vector whose all entries are $\frac{\alpha\ell}{m}$. The total cost of $\mathbf{x}^{\mathsf{cmn}}$ is $\alpha\ell$ and all common elements are covered by $\mathbf{x}^{\mathsf{cmn}}$. Thus, throughout the algorithm we may restrict our attention to the rare elements.

Our goal now is to construct an efficient MWU-based algorithm, which finds a solution $\mathbf{x}^{\mathsf{rare}}$ covering the rare elements, with objective value at most $\frac{\ell}{1-\beta} \leq (1+\varepsilon-\alpha)\ell$. We note that our implementation does not explicitly maintain the weight vector $\mathbf{w}^t$ described in Section 4.2.1, but instead updates (and normalizes) its probability vector $\mathbf{p}^t$ in every round.

4.2.3 First attempt: simple oracle and large width

A greedy solution for the oracle. We implement the oracle for MWU algorithm such that $\phi = \ell$, and thus requiring $\Theta(\ell \log n/\beta^2)$ iterations (Theorem 4.2.2). In each iteration, we need an oracle that finds some solution $\mathbf{x} \in \mathcal{P}_\ell$ satisfying $\mathbf{p}^\top \mathbf{A}\mathbf{x} \geq \mathbf{p}^\top \mathbf{b} - \beta/3$, or decides that no solution in $\mathcal{P}_\ell$ satisfies $\mathbf{p}^\top \mathbf{A}\mathbf{x} \geq \mathbf{p}^\top \mathbf{b}$.

Observe that $\mathbf{p}^\top \mathbf{A}\mathbf{x}$ is maximized when we place value ℓ on x_{S^*} where S^* achieves the maximum value $p_S \triangleq \sum_{e \in S} p_e$. Further, for our application, $\mathbf{b} = \mathbf{1}$ so $\mathbf{p}^\top \mathbf{b} = 1$. Our implementation **heavySetOracle** of **oracle** given in Figure 4-5 below is a deterministic greedy algorithm that finds a solution based on this observation. As $\mathbf{A}_e \mathbf{x} \leq \|\mathbf{x}\|_1 \leq \ell$, **heavySetOracle** implements a $(1,\ell)$-bounded $(\beta/3)$-approximate oracle. Therefore, the implementation of **feasibilityTest** with **heavySetOracle** computes a solution of objective value at most $(\alpha + \frac{1}{1-\beta})\ell < (1+\frac{\varepsilon}{3})\ell$ when $\ell \geq k$ as promised.

Finally, we track the space usage which concludes the complexities of the current version of our algorithm: it only stores vectors of length m or n, whose entries each requires a logarithmic number of bits, yielding the following theorem.

Theorem 4.2.3. *There exists a streaming algorithm that w.h.p. returns a $(1+\varepsilon)$-approximate fractional solution of* SetCover-LP$(\mathcal{U}, \mathcal{F})$ *in $O(\frac{k \log n}{\varepsilon^2})$ passes and using $\widetilde{O}(m+n)$ memory for any positive $\varepsilon \leq 1/2$. The algorithm works in both set arrival and edge arrival streams.*

The presented algorithm suffers from large number of passes over the input. In particular, we are interested in solving the fractional Set Cover in constant number of passes using sublinear space. To this end, we first reduce the required number of rounds in MWU by a more complicated implementation of **oracle**.

feasibilityTest(ℓ, ε):

$\alpha, \beta \leftarrow \frac{\varepsilon}{3}$, $\mathbf{p}^{\mathsf{curr}} \leftarrow \mathbf{1}_{m\times 1}$ ▷ The initial prob. vector for the MWU algorithm on $\mathcal{U}$

▷ Compute a cover of *common* elements in **one** pass
$\mathbf{x}^{\mathsf{cmn}} \leftarrow \frac{\alpha\ell}{m} \cdot \mathbf{1}_{m\times 1}$, $\mathbf{freq} \leftarrow \mathbf{0}_{n\times 1}$
for each set S in the stream **do**
  for each element $e \in S$ **do**
    $\mathsf{freq}_e \leftarrow \mathsf{freq}_e + 1$
    if e appears in more than $\frac{m}{\alpha\ell}$ sets (i.e. $\mathsf{freq}_e > \frac{m}{\alpha\ell}$) **then** ▷ Common element
      $p_e^{\mathsf{curr}} \leftarrow 0$
$\mathbf{p}^{\mathsf{curr}} \leftarrow \frac{\mathbf{p}^{\mathsf{curr}}}{\|\mathbf{p}^{\mathsf{curr}}\|}$ ▷ $\mathbf{p}^{\mathsf{curr}}$ represents the current prob. vector

$\mathbf{x}^{\mathsf{total}} \leftarrow \mathbf{0}_{m\times 1}$

▷ MWU algorithm for covering *rare* elements
repeat T times
  ▷ Solve the corresp. oracle of MWU and decide if the solution is feasible
  try $\mathbf{x} \leftarrow$ **oracle**$(\mathbf{p}^{\mathsf{curr}}, \ell, \mathcal{F})$
  $\mathbf{x}^{\mathsf{total}} \leftarrow \mathbf{x}^{\mathsf{total}} + \mathbf{x}$
  ▷ In **one** pass, update $\mathbf{p}$ according to $\mathbf{x}$
  $\mathbf{z} \leftarrow \mathbf{0}_{n\times 1}$
  for each set S in the stream **do**
    for each element $e \in S$ **do**
      $z_e \leftarrow z_e + x_S$
  if $(\mathbf{p}^{\mathsf{curr}})^\top \mathbf{z} < 1 - \beta/3$ **then** ▷ Detect infeasible solutions returned by **oracle**
    report INFEASIBLE
  $\mathbf{p}^{\mathsf{curr}} \leftarrow$ **updateProb**$(\mathbf{p}^{\mathsf{curr}}, \mathbf{z})$
$\mathbf{x}^{\mathsf{rare}} \leftarrow \frac{\mathbf{x}^{\mathsf{total}}}{(1-\beta)T}$ ▷ Scaled up the solution to cover *rare* elements

return $\mathbf{x}^{\mathsf{cmn}} + \mathbf{x}^{\mathsf{rare}}$

Figure 4-4: A generic implementation of **feasibilityTest**. Its performance depend on the implementations of **oracle**, **updateProb**. We will investigate different implementations of **oracle** in the gray box.

4.3 Max Cover Problem and its Application to Width Reduction

In this section, we improve the described algorithm in the previous section and prove the following result.

Theorem 4.3.1. *There exists a streaming algorithm that w.h.p. returns a $(1+\varepsilon)$-approximate fractional solution of* SetCover-LP$(\mathcal{U}, \mathcal{F})$ *in p passes and uses $\widetilde{O}(mn^{O(1/p\varepsilon)} + n)$ memory for any $2 \leq p \leq \text{polylog}(n)$ and $0 < \varepsilon \leq 1/2$. The algorithm works in both set arrival and edge arrival streams.*

Recall that in implementing **oracle**, we must find a solution $\mathbf{x}$ of total size $\|\mathbf{x}\|_1 \leq \ell$ with a

heavySetOracle$(\mathbf{p}, \ell, \mathcal{F})$:

Compute p_S for every $S \in \mathcal{F}$ while reading the set system ▷ either from stream or memory
$S^* \leftarrow \mathbf{argmax}_{S \in \mathcal{F}} p_S$
if $p_S < (1 - \beta/3)/\ell$ **then report** INFEASIBLE
$\mathbf{x} \leftarrow \mathbf{0}_{n \times 1}, x_S \leftarrow \ell$
return $\mathbf{x}$

Figure 4-5: **heavySetOracle** computes p_S of every set given the set system in a stream or stored memory, then returns the solution $\mathbf{x}$ that optimally places value ℓ on the corresponding entry. It reports INFEASIBLE if there is no sufficiently good solution, concluding that the set system is infeasible.

sufficiently large weight $\mathbf{p}^\top \mathbf{A}\mathbf{x}$. Our previous implementation chooses only one good entry x_S and places its entire *budget* ℓ on this entry. As the width of the solution is roughly the maximum amount an element is over-covered by $\mathbf{x}$, this implementation induces a width of ℓ. In this section, we design an oracle that returns a solution in which the budget is distributed more evenly among the entries of $\mathbf{x}$ to reduce the width. To this end, we design an implementation of **oracle** of the MWU approach based on the Max ℓ-Cover problem (whose precise definition will be given shortly). The solution to our Max ℓ-Cover aids in reducing the width of our **oracle** solution to a constant, so the required number of rounds of the MWU algorithm decreases to $O(\frac{\log n}{\varepsilon^2})$, independent of ℓ. Note that, if the objective value of an optimal solution of Set Cover$(\mathcal{U}, \mathcal{F})$ is ℓ, then a solution of width $o(\ell)$ may not exist, as shown in Lemma 4.3.2. This observation implies that we need to work with a different set system. Besides having small width, an optimal solution of the Set Cover instance on the new set system should have the same objective value of the optimal solution of Set Cover$(\mathcal{U}, \mathcal{F})$.

Lemma 4.3.2. *There exists a set system in which, under the direct application of the MWU framework in computing a $(1+\varepsilon)$-approximate solution, induces width $\phi = \Omega(k)$, where k is the optimal objective value. Moreover, the exists a set system in which the approach from the previous section (which handles the frequent and rare elements differently) has width $\phi = \Theta(n) = \Theta(\sqrt{m/\varepsilon})$.*

Proof. For the first claim, we consider an arbitrary set system, then modify it by adding a common element e to all sets. Recall that the MWU framework returns an average of the solutions from all rounds. Thus there must exist a round where the oracle returns a solution $\mathbf{x}$ of size $\|\mathbf{x}\|_1 = \Theta(k)$. For the added element e, this solution has $\sum_{S: e \in S} x_S = \sum_{S \in \mathcal{F}} x_S = \Theta(k)$, inducing width $\phi = \Omega(k)$.

For the second claim, consider the following set system with $k = \sqrt{m/\varepsilon}$ and $n = 2k + 1$. For $i = 1, \ldots, k$, let $S_i = \{e_i, e_{k+i}, e_{2k+1}\}$, whereas the remaining $m - k$ sets are arbitrary subsets of $\{e_1, \ldots, e_k\}$. Observe that e_{k+i} is contained only in S_i, so $x_{S_i} = 1$ in any valid set cover. Consequently the solution $\mathbf{x}$ where $x_{S_1} = \cdots = x_{S_k} = 1$ and $x_{S_{k+1}} = \cdots = x_{S_{2k+1}} = 0$ forms the unique (fractional) minimum set cover of size $k = \sqrt{m/\varepsilon}$. Next, recall that an element is considered rarely occurring if it appears in at most $\frac{m}{\alpha\ell} > \frac{m}{\varepsilon k}$ sets. As $e_{k+1}, \ldots, e_{2k}$ each only occurs once, and e_{2k+1} only appears in $k = \sqrt{m/\varepsilon} = \frac{m}{\varepsilon k}$ sets, these $k+1$ elements are deemed rare and thus handled by the MWU framework.

The solution computed by the MWU framework satisfies $\sum_{S: e \in S} x_S \geq 1 - \beta$ for every e, and in particular, for each $e \in \{e_{k+1}, \ldots, e_{2k}\}$. Therefore, the average solution places a total weight

MaxCover-LP $\langle\langle$*Input:* $\mathcal{U}, \mathcal{F}, \ell, \mathbf{p}\rangle\rangle$

$$\begin{aligned}
\text{maximize} \quad & \sum_{e\in\mathcal{U}} p_e z_e & \\
\text{subject to} \quad & \sum_{S:e\in S} x_S \geq z_e & \forall e \in \mathcal{U} \\
& \sum_{S\in\mathcal{F}} x_S = \ell & \\
& 0 \leq z_e \leq 1 & \forall e \in \mathcal{U} \\
& x_S \geq 0 & \forall S \in \mathcal{F}
\end{aligned}$$

Figure 4-6: LP relaxation of weighted Max k-Cover.

of at least $(1-\beta)\cdot\Theta(k)$ on $x_{S_1},\ldots,x_{S_k}$, so there must exist a round that places at least the same total weight on these sets. However, these k sets all contain e_{2k+1}, yielding $\sum_{S:e_{2k+1}\in S} x_S \geq (1-\beta)\cdot\Theta(k) = \Omega(k)$, implying a width of $\Omega(k) = \Omega(\sqrt{m/\varepsilon})$. □

Extended Set System. First, we consider the *extended set system* $(\mathcal{U}, \breve{\mathcal{F}})$, where $\breve{\mathcal{F}}$ is the collection containing all subsets of sets in $\mathcal{F}$; that is,

$$\breve{\mathcal{F}} \triangleq \{R : R \subseteq S \text{ for some } S \in \mathcal{F}\}.$$

It is straightforward to see that the optimal objective value of Set Cover over $(\mathcal{U}, \breve{\mathcal{F}})$ is equal to that of $(\mathcal{U}, \mathcal{F})$: we only add subsets of the original sets to create $\breve{\mathcal{F}}$, and we may replace any subset from $\breve{\mathcal{F}}$ in our solution with its original set in $\mathcal{F}$. Moreover, we may *prune* any collection of sets from $\mathcal{F}$ into a collection from $\breve{\mathcal{F}}$ of the same cardinality so that, this pruned collection not only covers the same elements, but also each of these elements is covered exactly once. This extended set system is defined for the sake of analysis only: we will never explicitly handle an exponential number of sets throughout our algorithm.

We define *ℓ-cover* as a collection of sets of total weight ℓ. Although the pruning of an ℓ-cover reduces the width, the total weight $\mathbf{p}^\top\mathbf{A}\mathbf{x}$ of the solution will decrease. Thus, we consider the weighted constraint of the form

$$\sum_{e\in\mathcal{U}} \left(p_e \cdot \min\{1, \sum_{S:e\in S} x_S\} \right) \geq 1;$$

that is, we can only gain the value p_e without any multiplicity larger than 1. The problem of maximizing the left hand side is known as the *weighted max coverage* problem: for a parameter ℓ, find an ℓ-cover such that the total value p_e's of the covered elements is maximized.

4.3.1 The Maximum Coverage problem

In the design of our algorithm, we consider the *weighted* Max k-Cover problem, which is closely related to Set Cover. Extending upon the brief description given earlier, we fully specify the LP relaxation of this problem. In the weighted Max k-Cover$(\mathcal{U}, \mathcal{F}, \ell, \mathbf{p})$, given a ground set of elements $\mathcal{U}$, a collection of sets $\mathcal{F}$ over the ground set, a budget parameter ℓ, and a weight vector $\mathbf{p}$, the goal is to return ℓ sets in $\mathcal{F}$ whose weighted *coverage*, the total weight of all covered elements, is maximized. Moreover, since we are aiming for a fractional solution of Set Cover, we consider the LP relaxation of weighted Max k-Cover, MaxCover-LP (see Figure 4-6); in this LP relaxation, z_e denotes the fractional amount that an element is covered, and hence is capped at 1.

As an intermediate goal, we aim to compute an approximate solution of MaxCover-LP, given that the optimal solution covers all elements in the ground set, or to correctly detect that no solution has weighted coverage of more than $(1-\varepsilon)$. In our application, the vector $\mathbf{p}$ is always a probability vector: $\mathbf{p} \geq \mathbf{0}$ and $\sum_{e \in \mathcal{U}} p_e = 1$. We make the following useful observation.

Observation 4.3.3. *Let k be the value of an optimal solution of* SetCover-LP$(\mathcal{U}, \mathcal{F})$ *and let $\mathbf{p}$ be an arbitrary probability vector over the ground set. Then there exists a fractional solution of* MaxCover-LP$(\mathcal{U}, \mathcal{F}, \ell, \mathbf{p})$ *whose weighted coverage is one if $\ell \geq k$.*

δ-integral near optimal solution of MaxCover-LP. Our plan is to solve MaxCover-LP over a randomly projected set system, and argue that with high probability this will result in a valid **oracle**. Such an argument requires an application of the union bound over the set of solutions, which is generally of unbounded size. To this end, we consider a more restrictive domain of δ-*integral* solutions: this domain has bounded size, but is still guaranteed to contain a sufficiently good solution.

Definition 4.3.4 (δ-integral solution)**.** *A fractional solution $\mathbf{x}_{n\times 1}$ of an LP is δ-integral if $\frac{1}{\delta} \cdot \mathbf{x}$ is an integral vector. That is, for each $i \in [n]$, $x_i = v_i\delta$ where each v_i is an integer.*

Next we claim that **maxCoverOracle** given in Figure 4-7 below, which is the MWU algorithm with **heavySetOracle** for solving MaxCover-LP, results in a δ-integral solution.

Lemma 4.3.5. *Consider a* MaxCover-LP *with the optimal objective value* OPT *(where the weights of elements form a probability vector). There exists a $\Theta(\frac{\varepsilon_{\text{MC}}^2}{\log n})$-integral solution of* MaxCover-LP *whose objective value is at least $(1-\varepsilon_{\text{MC}})$OPT. In particular, if an optimal solution covers all elements $\mathcal{U}$ ($\ell \geq k$),* **maxCoverOracle** *returns a solution whose weighted coverage is at least $1-\varepsilon_{\text{MC}}$ in polynomial time.*

Proof. Let $(\mathbf{x}^*, \mathbf{z}^*)$ denote the optimal solution of value OPT to MaxCover-LP, which implies that $\|\mathbf{x}^*\|_1 \leq \ell$ and $\mathbf{A}\mathbf{x}^* \geq \mathbf{z}^*$. Consider the following covering LP: minimize $\|\mathbf{x}\|_1$ subject to $\mathbf{A}\mathbf{x} \geq \mathbf{z}^*$ and $\mathbf{x} \geq \mathbf{0}$. Clearly there exists an optimal solution of objective value ℓ, namely $\mathbf{x}^*$. This covering LP may be solved via the MWU framework. In particular, we may use the oracle that picks one set S with maximum weight (as maintained in the MWU framework) and places its entire budget on x_S. For an accurate guess $\ell' = \Theta(\ell)$ of the optimal value, this algorithm returns an average of $T = \Theta(\frac{\ell' \log n}{\varepsilon_{\text{MC}}^2}) = \Theta(\frac{\ell \log n}{\varepsilon_{\text{MC}}^2})$ oracle solutions. Observe that the outputted solution $\mathbf{x}$ is of the

form $x_S = \frac{v_S \ell'}{T} = v_S \delta$ where v_S is the number of rounds in which S is chosen by the oracle, and $\delta = \frac{\ell'}{T} = \frac{\ell' \varepsilon_{\mathrm{MC}}^2}{\ell \log n} = \Theta(\frac{\varepsilon_{\mathrm{MC}}^2}{\log n})$. In other words, $\mathbf{x}$ is $(\frac{\varepsilon_{\mathrm{MC}}^2}{\log n})$-integral. By Theorem 4.2.2, $\mathbf{x}$ satisfies $\mathbf{A}\mathbf{x} \geq (1 - \varepsilon_{\mathrm{MC}})\mathbf{z}^*$. Then in MaxCover-LP, the solution $(\mathbf{x}, (1 - \varepsilon_{\mathrm{MC}})\mathbf{z}^*)$ yields coverage at least $\mathbf{p}^\top((1 - \varepsilon_{\mathrm{MC}})\mathbf{z}^*) = (1 - \varepsilon_{\mathrm{MC}})\mathbf{p}^\top \mathbf{z}^* = (1 - \varepsilon_{\mathrm{MC}})\mathrm{OPT}$. □

maxCoverOracle$(\mathcal{U}, \mathcal{F}, \ell)$:
  $\mathbf{x} \leftarrow$ MWU solution of SetCover LP relaxation implemented with **heavySetOracle**
  return $\mathbf{x}$

Figure 4-7: **maxCoverOracle** returns a fractional ℓ-cover with weighted coverage at least $1 - \beta/3$ w.h.p. if $\ell \geq k$. It provides no guarantee on its behavior if $\ell < k$.

Pruning a fractional ℓ-cover. In our analysis, we aim to solve the SetCover problem under the extended set system. We claim that any solution $\mathbf{x}$ with coverage $\mathbf{z}$ in the actual set system may be turned into a pruned solution $\breve{\mathbf{x}}$ in the extended set system that provides the same coverage $\mathbf{z}$, but satisfies the strict equality $\sum_{\breve{S} \in \breve{\mathcal{F}}: e \in \breve{S}} \breve{x}_{\breve{S}} = z_e$. Since $z_e \leq 1$, the pruned solution satisfies the condition for an oracle with width *one*. We give an algorithm **prune** for pruning $\mathbf{x}$ into $\breve{\mathbf{x}}$ below (Figure 4-8) and show the desired property in Lemma 4.3.6.

prune$(\mathbf{x})$:
  $\breve{\mathbf{x}} \leftarrow \mathbf{0}_{|\breve{\mathcal{F}}| \times 1}$, $\mathbf{z} \leftarrow \mathbf{0}_{n \times 1}$ ▷ Maintain the pruned solution and its coverage amount
  for each $S \in \mathcal{F}$ **do**
    $\breve{S} \leftarrow S$
    while $x_S > 0$ **do**
      $r \leftarrow \min(x_S, \min_{e \in \breve{S}}(1 - z_e))$ ▷ Weight to be moved from x_S to $\breve{x}_{\breve{S}}$
      $x_S \leftarrow x_S - r$, $x_{\breve{S}} \leftarrow x_{\breve{S}} + r$ ▷ Move weight to the pruned solution
      for each $e \in \breve{S}$ **do** $z_e \leftarrow z_e + r$ ▷ Update coverage accordingly
      $\breve{S} \leftarrow \breve{S} \setminus \{e \in \breve{S} : z_e = 1\}$ ▷ Remove e with $z_e = 1$ from $\breve{S}$
  return $\mathbf{z}$

Figure 4-8: The **prune** subroutine lifts a solution in $\mathcal{F}$ to a solution in $\breve{\mathcal{F}}$ with the same MaxCover-LP objective value and width 1. The subroutine returns $\mathbf{z}$, the amount by which members of $\breve{\mathcal{F}}$ cover each element. The actual pruned solution $\breve{\mathbf{x}}$ may be computed but has no further use in our algorithm and thus not returned.

Lemma 4.3.6. *A fractional ℓ-cover $\mathbf{x}$ of $(\mathcal{U}, \mathcal{F})$ can be converted, in polynomial time, to a fractional ℓ-cover $\breve{\mathbf{x}}$ of $(\mathcal{U}, \breve{\mathcal{F}})$ such that for each element e, its coverage $z_e = \sum_{\breve{S} \in \breve{\mathcal{F}}: e \in \breve{S}} \breve{x}_{\breve{S}} = \min(\sum_{S: e \in S} x_S, 1)$.*

Proof. Consider the algorithm **prune** in Figure 4-8. As we pick a valid amount $r \leq x_S$ to move from x_S to $\breve{x}_{\breve{S}}$ at each step, $\breve{\mathbf{x}}$ must be an ℓ-cover (in the extended set system) when **prune** finishes. Observe that if $\sum_{S: e \in S} x_S < 1$ then e will never be removed from any $\breve{S}$, so z_e is increased by x_S for every S, and thus $z_e = \sum_{S: e \in S} x_S$. Otherwise, the condition $r \leq 1 - z_e$ ensures that z_e stops increasing precisely when it reaches 1. Each S takes up to $n + 1$ rounds in the while loop as one

element $e \in S$ is removed at the end of each round. There are at most m sets, so the algorithm must terminate (in polynomial time).

We note that in Section 4.3.5, we need to adjust **prune** to instead achieves the condition $z_e = \min(\mathbf{A}_e\mathbf{x}, 1)$ where entries of $\mathbf{A}$ are arbitrary non-negative values. We simply make the following modifications: choose $r \leftarrow \min(x_S, \min_{e \in \breve{S}} \frac{1-z_e}{A_{e,S}})$ and update $z_e \leftarrow z_e + r \cdot A_{e,S}$, and the same proof follows. □

Remark that to update the weights in the MWU framework, it is sufficient to have the coverage $\sum_{\breve{S} \in \breve{\mathcal{F}}: e \in \breve{S}} \breve{x}_{\breve{S}}$, which are the z_e's returned by **prune**; the actual solution $\breve{\mathbf{x}}$ is not necessary. Observe further that our MWU algorithm can still use $\mathbf{x}$ instead of $\breve{\mathbf{x}}$ as its solution because $\mathbf{x}$ has no worse coverage than $\breve{\mathbf{x}}$ in every iteration, and so does the final, average solution. Lastly, notice that the coverage $\mathbf{z}$ returned by **prune** has the simple formula $z_e = \min(\sum_{S: e \in S} x_S, 1)$. That is, we introduce **prune** to show an existence of $\breve{\mathbf{x}}$, but will never run **prune** in our algorithm. Note also that, in order to update the weights in the MWU framework, it is sufficient to know the vector $\mathbf{z}$, which has a simple formula given in the lemma above. The actual solution $\breve{\mathbf{x}}$ is not necessary.

4.3.2 Sampling-based oracle for Fractional Max Coverage

In the previous section, we simply needed to compute the values p_S's in order to construct a solution for the **oracle**. Here as we aim to bound the width of **oracle**, our new task is to find a fractional ℓ-cover $\mathbf{x}$ whose weighted coverage is at least $1 - \beta/3$. The *element sampling* technique, which is also known from prior work in streaming Set Cover and Max k-Cover, is to sample a few elements and solve the problem over the sampled elements only. Then, by applying the union bound over all possible candidate solutions, it is shown that w.h.p. a nearly optimal cover of the sampled elements also covers a large fraction of the whole ground set. This argument applies to the aforementioned problems precisely because there are standard ways of bounding the number of all integral candidate solutions (e.g. ℓ-covers).

However, in the fractional setting, there are infinitely many solutions. Consequently, we employ the notion of δ-integral solutions where the number of such solutions is bounded. In Lemma 4.3.6, we showed that there always exists a δ-integral solution to MaxCover-LP whose coverage is at least a $(1 - \varepsilon_{\text{MC}})$-fraction of an optimal solution. Moreover, the number of all possible solutions is bounded by the number of ways to divide the budget ℓ into ℓ/δ equal parts of value δ and distribute them (possibly with repetition) among m entries:

Observation 4.3.7. *The number of feasible δ-integral solutions to* MaxCover-LP$(\mathcal{U}, \mathcal{F}, \ell, \mathbf{p})$ *is* $O(m^{\ell/\delta})$ *for any multiple ℓ of δ.*

Next, we design our algorithm using the element sampling technique: we show that a $(1-\beta/3)$-approximate solution of MaxCover-LP can be computed using the projection of all sets in $\mathcal{F}$ over a set of elements of size $\Theta(\frac{\ell \log n \log mn}{\beta^4})$ picked according to $\mathbf{p}$. For every fractional solution $(\mathbf{x}, \mathbf{z})$ and subset of elements $\mathcal{V} \subseteq \mathcal{U}$, let $\mathcal{C}_{\mathcal{V}}(\mathbf{x}) \triangleq \sum_{e \in \mathcal{V}} p_e z_e$ denote the coverage of elements in $\mathcal{V}$ where $z_e = \min(1, \sum_{S: e \in S} x_S)$. We may omit the subscript $\mathcal{V}$ in $\mathcal{C}_{\mathcal{V}}$ if $\mathcal{V} = \mathcal{U}$.

The following lemma, which is essentially an extension of the Element Sampling lemma of [DIMV14] for our application, MaxCover-LP, shows that a $(1 - \varepsilon_{\text{MC}})$-approximate ℓ-cover

over a set of sampled elements of size $\Theta(\ell \log n \log mn/\gamma^4)$ w.h.p. has a weighted coverage of at least $(1-2\gamma)(1-\varepsilon_{\text{MC}})$ if there exists a fractional ℓ-cover whose coverage is 1. Thus, choosing $\varepsilon_{\text{MC}} = \gamma = \beta/9$ yields the desired guarantee for **maxCoverOracle**, leading to the performance given in Theorem 4.3.9.

Lemma 4.3.8. *Let ε_{MC} and γ be parameters. Consider the* MaxCover-LP$(\mathcal{U}, \mathcal{F}, \ell, \mathbf{p})$ *with optimal solution of value* OPT*, and let $\mathcal{L}$ be a multi-set of $s = \Theta(\ell \log n \log(mn)/\gamma^4)$ elements sampled independently at random according to the probability vector $\mathbf{p}$. Let $\mathbf{x}^{\text{sol}}$ be a $(1-\varepsilon_{\text{MC}})$-approximate $\Theta(\frac{\gamma^2}{\log n})$-integral ℓ-cover over the sampled elements. Then with high probability, $\mathcal{C}(\mathbf{x}^{\text{sol}}) \geq (1-2\gamma)(1-\varepsilon_{\text{MC}})\text{OPT}$.*

Proof. Consider the MaxCover-LP$(\mathcal{U}, \mathcal{F}, \ell, \mathbf{p})$ with optimal solution $(\mathbf{x}^{\text{OPT}}, \mathbf{z}^{\text{OPT}})$ of value OPT, and let $\mathbf{x}^{\text{sol}}$ be a $(1-\varepsilon_{\text{MC}})$-approximate $\Theta(\frac{\gamma^2}{\log n})$-integral ℓ-cover over the sampled elements and $\mathbf{z}^{\text{sol}}$ be its corresponding coverage vector. Denote the sampled elements with $\mathcal{L} = \{\hat{e}_1, \cdots, \hat{e}_s\}$. Observe that by defining each X_i as a random variable that takes the value $z^{\text{OPT}}_{\hat{e}_i}$ with probability $p_{\hat{e}_i}$ and 0 otherwise, the expected value of $\mathsf{X} = \sum_{i=1}^{s} \mathsf{X}_i$ is

$$\mathbf{E}[\mathsf{X}] = \sum_{i=1}^{s} \mathbf{E}[\mathsf{X}_i] = s \sum_{e \in \mathcal{U}} p_e \cdot z_e^{\text{OPT}} = s \cdot \mathcal{C}(\mathbf{x}^{\text{OPT}}) = s \cdot \text{OPT}.$$

Let $\tau = s(1-\gamma)\text{OPT}$. Since $\mathsf{X}_i \in [0,1]$, by applying Chernoff bound on X, we obtain

$$\begin{aligned}\mathbf{Pr}\big[\mathcal{C}_{\mathcal{L}}(\mathbf{x}^{\text{OPT}}) \leq \tau\big] &= \mathbf{Pr}[\mathsf{X} \leq (1-\gamma)\mathbf{E}[\mathsf{X}]] \\ &\leq e^{-\frac{\gamma^2 \mathbf{E}[\mathsf{X}]}{3}} \leq e^{-\frac{\Omega(\ell \log(mn) \log n/\gamma^2)}{3}} = (mn)^{-\Omega(\ell \log n/\gamma^2)}.\end{aligned}$$

Therefore, since $\mathbf{x}^{\text{sol}}$ is a $(1-\varepsilon_{\text{MC}})$-approximate solution of MaxCover-LP$(\mathcal{L}, \mathcal{F}, \ell, \mathbf{p})$, with probability $1-(mn)^{-\Omega(\ell \log n/\gamma^2)}$, we have $\mathcal{C}_{\mathcal{L}}(\mathbf{x}^{\text{sol}}) \geq (1-\varepsilon_{\text{MC}})\tau$.

Next, by a similar approach, we show that for any fractional solution $\mathbf{x}$, if $\mathcal{C}_{\mathcal{L}}(\mathbf{x}) \geq \mathcal{C}_{\mathcal{L}}(\mathbf{x}^{\text{OPT}})$, then with probability $1-(mn)^{-\Omega(\ell \log n/\gamma^2)}$, $\mathcal{C}(\mathbf{x}) \geq (\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\text{MC}})\text{OPT}$. Consider a fractional ℓ-cover $(\mathbf{x}, \mathbf{z})$ whose coverage is less than $(\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\text{MC}})\text{OPT}$. Let Y_i denote a random variable that takes value $z_{\hat{e}_i}$ with probability $p_{\hat{e}_i}$, and define $\mathsf{Y} = \sum_{i=1}^{s} \mathsf{Y}_i$. Then, $\mathbf{E}[\mathsf{Y}_i] = \mathcal{C}(\mathbf{x}) < (\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\text{MC}})\text{OPT}$. For ease of analysis, let each $\overline{\mathsf{Y}}_i \in [0,1]$ be an auxiliary random variable that stochastically dominates Y_i with expectation $\mathbf{E}[\overline{\mathsf{Y}}_i] = (\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\text{MC}})\text{OPT}$, and $\overline{\mathsf{Y}} = \sum_{i=1}^{s} \overline{\mathsf{Y}}_i$ which stochastically dominates $\overline{\mathsf{Y}}$ with expectation $\mathbf{E}[\overline{\mathsf{Y}}] = s \cdot (\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\text{MC}})\text{OPT} = \frac{(1-\varepsilon_{\text{MC}})\tau}{1+\gamma}$. We then have

$$\begin{aligned}\mathbf{Pr}[\mathcal{C}_{\mathcal{L}}(\mathbf{x}) > (1-\varepsilon_{\text{MC}})\tau] = \mathbf{Pr}[\mathsf{Y} > (1-\varepsilon_{\text{MC}})\tau] &= \mathbf{Pr}\big[\mathsf{Y} > (1+\gamma)\mathbf{E}[\overline{\mathsf{Y}}]\big] \\ &\leq \mathbf{Pr}\big[\overline{\mathsf{Y}} > (1+\gamma)\mathbf{E}[\overline{\mathsf{Y}}]\big] \leq e^{-\frac{\gamma^2 \mathbf{E}[\overline{\mathsf{Y}}]}{3}} \leq (mn)^{-\Omega(\ell \log n/\gamma^2)},\end{aligned}$$

using the fact that $(\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\text{MC}}) = \Theta(1)$ for our interested range of parameters. Thus,

$$\mathbf{Pr}\left[\mathcal{C}(\mathbf{x}) \leq (\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\text{MC}})\text{OPT} \text{ and } \mathcal{C}_{\mathcal{L}}(\mathbf{x}) > (1-\varepsilon_{\text{MC}})\tau\right] \leq (mn)^{-\Omega(\ell \log n/\gamma^2)}.$$

In other words, except with probability $(mn)^{-\Omega(\ell \log n/\gamma^2)}$, a chosen solution $\mathbf{x}$ that offers at least as good empirical coverage over $\mathcal{L}$ as $\mathbf{x}^{\mathrm{OPT}}$ (namely $\mathbf{x}^{\mathtt{sol}}$) does have actual coverage of at least $(\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\mathrm{MC}})\mathrm{OPT}$.

Since the total number of $\Theta(\frac{\gamma^2}{\log n})$-integral ℓ-covers is $O(m^{\ell \log n/\gamma^2})$ (Observation 4.3.7), applying union bound, with probability at least

$$1 - O(m^{\ell \log n/\gamma^2}) \cdot (mn)^{-\Omega(\ell \log n/\gamma^2)} = 1 - \frac{1}{\mathrm{poly}(mn)} ,$$

a $(1-\varepsilon_{\mathrm{MC}})$-approximate $\Theta(\frac{\gamma^2}{\log n})$-integral solution of Max k-Cover$(\mathcal{L}, \mathcal{F}, \ell, \mathbf{p})$ has weighted coverage of at least

$$(\frac{1-\gamma}{1+\gamma})(1-\varepsilon_{\mathrm{MC}})\mathrm{OPT} > (1-2\gamma)(1-\varepsilon_{\mathrm{MC}})\mathrm{OPT}$$

over $\mathcal{U}$. □

Theorem 4.3.9. *There exists a streaming algorithm that w.h.p. returns a $(1+\varepsilon)$-approximate fractional solution of* SetCover-LP$(\mathcal{U}, \mathcal{F})$ *in $O(\log n/\varepsilon^2)$ passes and uses $\widetilde{O}(m/\varepsilon^6 + n)$ memory for any positive $\varepsilon \leq 1/2$. The algorithm works in both set arrival and edge arrival streams.*

Proof. The algorithm clearly requires $\Theta(T)$ passes to simulate the MWU algorithm. The required amount of memory, besides $\widetilde{O}(n)$ for counting elements, is dominated by the projected set system. In each pass over the stream, we sample $\Theta(\ell \log mn \log n/\varepsilon^4)$ elements, and since they are rarely occurring, each is contained in at most $\Theta(\frac{m}{\varepsilon\ell})$ sets. Finally, we run $\log_{1+\Theta(\varepsilon)} n = O(\log n/\varepsilon)$ instances of the MWU algorithm in parallel to compute a $(1+\varepsilon)$-approximate solution. In total, our space complexity is $\Theta(\ell \log mn \log n/\varepsilon^4) \cdot \Theta(\frac{m}{\varepsilon\ell}) \cdot O(\log n/\varepsilon) = \widetilde{O}(m/\varepsilon^6)$. □

4.3.3 Final step: running several MWU rounds together

We complete our result by further reducing the number of passes at the expense of increasing the required amount of memory, yielding our full algorithm **fastFeasibilityTest** in Figure 4-9. More precisely, aiming for a p-pass algorithm, we show how to execute $R \triangleq \frac{T}{\Theta(p)} = \Theta(\frac{\log n}{p\beta^2})$ rounds of the MWU algorithm in a single pass. We show that this task may be accomplished with a multiplicative factor of $f \cdot \Theta(\log mn)$ increase in memory usage, where $f \triangleq n^{\Theta(1/(p\beta))}$.

Advance sampling. Consider a sequence of R consecutive rounds $i = 1, \ldots, R$. In order to implement the MWU algorithm for these rounds, we need (multi-)sets of sampled elements $\mathcal{L}_1, \ldots, \mathcal{L}_R$ according to probabilities $\mathbf{p}^1, \ldots, \mathbf{p}^R$, respectively (where $\mathbf{p}^i$ is the probability corresponding to round i). Since the probabilities of subsequent rounds are not known in advance, we circumvent this problem by choosing these sets $\mathcal{L}_i$'s with probabilities according to $\mathbf{p}^1$, but the number of samples in each set will be $|\mathcal{L}_i| = s \cdot f \cdot \Theta(\log mn)$ instead of s. Then, once $\mathbf{p}^i$ is revealed, we sub-sample the elements from $\mathcal{L}_i$ to obtain $\mathcal{L}'_i$ as follow: for a (copy of) sampled element $\hat{e} \in \mathcal{L}_i$, add $\hat{e}$ to $\mathcal{L}'_i$ with probability $\frac{p^i_{\hat{e}}}{p^1_{\hat{e}} f}$; otherwise, simply discard it. Note that it is still left to be shown that the probability above is indeed at most 1.

Since each e was originally sampled with probability p^1_e, then in $\mathcal{L}'_i$, the probability that a sampled element $\hat{e} = e$ is exactly p^i_e/f. By having $f \cdot \Theta(\log mn)$ times the originally required

number of samples s in the first place, in expectation we still have $\mathbf{E}[|\mathcal{L}'_i|] = |\mathcal{L}_i| \sum_{e \in \mathcal{U}} \frac{p^i_e}{f} = (s \cdot f \cdot \Theta(\log mn)) \frac{1}{f} = s \cdot \Theta(\log mn)$. Due to the $\Theta(\log mn)$ factor, by the Chernoff bound, we conclude that with w.h.p. $|\mathcal{L}'_i| \geq s$. Thus, we have a sufficient number of elements sampled with probability according to $\mathbf{p}^i$ to apply Lemma 4.3.8, as needed.

Change in probabilities. As noted above, we must show that the probability that we sub-sample each element is at most 1; that is, $p^i_e/p^1_e \leq f = n^{\Theta(1/(p\beta))}$ for every element e and every round $i = 1, \ldots, R$. We bound the multiplicative difference between the probabilities of two consecutive rounds as follows.

Lemma 4.3.10. *Let* $\mathbf{p}$ *and* $\mathbf{p}'$ *be the probability of elements before and after an update. Then for every element* e, $p'_e \leq (1 + O(\beta))p_e$.

4.3.4 Proof of Lemma 4.3.10

Proof. Recall the weight update formula $w^{t+1}_e = w^t_e(1 - \frac{\beta(\breve{\mathbf{A}}_e \breve{\mathbf{x}} - b_e)}{6\phi})$ for the MWU framework, where $\breve{\mathbf{A}}_{n \times |\breve{\mathcal{F}}|}$ represents the membership matrix corresponding to the extended set system $(\mathcal{U}, \breve{\mathcal{F}})$. In our case, the desired coverage amount is $b_e = 1$. By construction, we have $\breve{\mathbf{A}}_e \breve{\mathbf{x}} = z_e \leq 1$; therefore, our width is $\phi = 1$, and $-1 \leq \breve{\mathbf{A}}_e \breve{\mathbf{x}} - b_e \leq 0$. That is, the weight of each element cannot decrease, but may increase by at most a multiplicative factor of $1 + \beta/6$, before normalization. Thus even after normalization no weight may increase by more than a factor of $1 + \beta/6 = 1 + O(\beta)$. □

Therefore, after $R = \Theta(\frac{\log n}{p\beta^2})$ rounds, the probability of any element may increase by at most a factor of $(1 + O(\beta))^{\Theta(\frac{\log n}{p\beta^2})} \leq e^{\Theta(\frac{\log n}{p\beta})} = n^{\Theta(1/(p\beta))} = f$, as desired. This concludes the proof of Theorem 4.3.1.

Implementation details. We make a few remarks about the implementation given in Figure 4-9. First, even though we perform all sampling in advance, the decisions of **maxCoverOracle** do not depend on any $\mathcal{L}_i$ of later rounds, and **updateProb** is entirely deterministic: there is no dependency issue between rounds. Next, we only need to perform **updateProb** on the sampled elements $\mathcal{L} = \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_R$ during the current R rounds. We therefore denote the probabilities with a different vector $\mathbf{q}^i$ over the sampled elements $\mathcal{L}$ only. Probabilities of elements outside $\mathcal{L}$ are not required by **maxCoverOracle** during these rounds, but we simply need to spend one more pass after executing R rounds of MWU to aggregate the new probability vector $\mathbf{p}$ over all (rare) elements. Similarly, since **maxCoverOracle** does not have the ability to verify, during the MWU algorithm, that each solution $\mathbf{x}^i$ returned by the oracle indeed provides a sufficient coverage, we check all of them during this additional pass. Lastly, we again remark that this algorithm operates on the extended set system: the solution $\mathbf{x}$ returned by **maxCoverOracle** has at least the same coverage as $\breve{\mathbf{x}}$. While $\breve{\mathbf{x}}$ is not explicitly computed, its coverage vector $\mathbf{z}$ can be computed exactly.

4.3.5 Extension to general covering LPs

We remark that our MWU-based algorithm can be extended to solve a more general class of covering LPs. Consider the problem of finding a vector $\mathbf{x}$ that minimizes $\mathbf{c}^\top \mathbf{x}$ subject to constraints $\mathbf{A}\mathbf{x} \geq \mathbf{b}$

fastFeasibilityTest(ℓ, ε):

$\alpha, \beta \leftarrow \frac{\varepsilon}{3}$, $\mathbf{p}^{\mathsf{curr}} \leftarrow \mathbf{1}_{m\times 1}$ ▷ The initial prob. vector for the MWU algorithm on $\mathcal{U}$

Compute a cover of *common* elements in **one** pass ▷ See Fig. 4-4's **feasibilityTest** block

$\mathbf{x}^{\mathsf{total}} \leftarrow \mathbf{0}_{m\times 1}$

▷ MWU algorithm for covering *rare* elements

repeat p times

$R \leftarrow \Theta(\frac{\log n}{p\beta^2})$ ▷ Number of MWU iterations performed together

▷ In **one** pass, projects all sets in $\mathcal{F}$ over the collections of samples $\mathcal{L}_1, \cdots \mathcal{L}_R$

sample $\mathcal{L}_1, \ldots, \mathcal{L}_R$ according to $\mathbf{p}^{\mathsf{curr}}$ each of size $\ell n^{\Theta(1/(p\beta))}\,\mathrm{poly}(\log mn)$

$\mathcal{L} \leftarrow \mathcal{L}_1 \cup \cdots \cup \mathcal{L}_R$, $\mathcal{F}_{\mathcal{L}} \leftarrow \emptyset$ ▷ $\mathcal{L}$ is a set whereas $\mathcal{L}_1, \ldots, \mathcal{L}_R$ are multi-sets

for each set S in the stream **do** $\mathcal{F}_{\mathcal{L}} \leftarrow \mathcal{F}_{\mathcal{L}} \cup \{S \cap \mathcal{L}\}$

▷ Each pass simulates R rounds of MWU

for each $e \in \mathcal{L}$ **do** $q_e^1 \leftarrow p_e^{\mathsf{curr}}$ ▷ Project $\mathbf{p}^{\mathsf{curr}}_{n\times 1}$ to $\mathbf{q}^1_{|\mathcal{L}|\times 1}$ over sampled elements

$\mathbf{q}^1 \leftarrow \frac{\mathbf{q}^1}{\|\mathbf{q}^1\|}$

for each round $i = 1, \ldots, R$ **do**

$\mathcal{L}'_i \leftarrow$ **sample** each elt $e \in \mathcal{L}_i$ with probab. $\frac{q_e^i}{q_e^1 n^{\Theta(1/(p\beta))}}$ ▷ Rejection Sampling

$\mathbf{x}^i \leftarrow$ **maxCoverOracle**$(\mathcal{L}'_i, \mathcal{F}_{\mathcal{L}}, \ell)$ ▷ w.h.p. $\mathcal{C}(\mathbf{x}^i) \geq 1 - \beta/3$ when $\ell \geq k$

▷ In **no additional** pass, updates probab. $\mathbf{q}$ over sampled elts according to $\mathbf{x}^i$

$\mathbf{z} \leftarrow \mathbf{0}_{|\mathcal{L}|\times 1}$ ▷ Compute coverage over *sampled* elements

for each element-set pair $e \in S$ where $S \in \mathcal{F}_{\mathcal{L}}$ **do** $z_e \leftarrow \min(z_e + x_S^i, 1)$

$\mathbf{q}^{i+1} \leftarrow$ **updateProb**$(\mathbf{q}^i, \mathbf{z})$ ▷ Only update weights of elements in $\mathcal{L}$

▷ In **one** pass, updates probab. $\mathbf{p}^{\mathsf{curr}}$ over *all* (rare) elts according to $\mathbf{x}^1, \ldots, \mathbf{x}^R$

$\mathbf{z}^1, \ldots, \mathbf{z}^R \leftarrow \mathbf{0}_{n\times 1}$ ▷ Compute coverage over *all* (rare) elements

for each element-set pair $e \in S$ in the stream **do**

for each round $i = 1, \ldots, R$ **do** $z_e^i \leftarrow \min(z_e^i + x_S^i, 1)$

for each round $i = 1, \ldots, R$ **do**

if $(\mathbf{p}^{\mathsf{curr}})^\top \mathbf{z}^i < 1 - \beta/3$ **then** ▷ Detect infeasible solutions

report INFEASIBLE

$\mathbf{x}^{\mathsf{total}} \leftarrow \mathbf{x}^{\mathsf{total}} + \mathbf{x}^i$, $\mathbf{p}^{\mathsf{curr}} \leftarrow$ **updateProb**$(\mathbf{p}^{\mathsf{curr}}, \mathbf{z}^i)$ ▷ Perform actual updates

$\mathbf{x}^{\mathsf{rare}} \leftarrow \frac{\mathbf{x}^{\mathsf{total}}}{(1-\beta)T}$ ▷ Scaled up the solution to cover *rare* elements

return $\mathbf{x}^{\mathsf{cmn}} + \mathbf{x}^{\mathsf{rare}}$

Figure 4-9: An efficient implementation of **feasibilityTest** which performs in p passes and consumes $\widetilde{O}(mn^{O(\frac{1}{p\varepsilon})} + n)$ space.

and $\mathbf{x} \geq \mathbf{0}$. In terms of the Set Cover problem, $A_{e,S} \geq 0$ indicates the multiplicity of an element e in the set S, $b_e > 0$ denotes the number of times we wish e to be covered, and $c_S > 0$ denotes the cost per unit for the set S. Now define

$$L \triangleq \min_{(e,S):A_{e,S}\neq 0} \frac{A_{e,S}}{b_e c_S} \quad \text{and} \quad U \triangleq \max_{(e,S)} \frac{A_{e,S}}{b_e c_S}.$$

Then, we may modify our algorithm to obtain the following result.

Theorem 4.3.11. *There exists a streaming algorithm that w.h.p. returns a $(1+\varepsilon)$-approximate fractional solution to general covering LPs in p passes and using $\widetilde{O}(\frac{mU}{\varepsilon^6 L} \cdot n^{O(\frac{1}{p\varepsilon})} + n)$ memory for any $3 \leq p \leq \text{polylog}(n)$, where parameters L and U are defined above. The algorithm works in both set arrival and edge arrival streams.*

Proof. We modify our algorithm and provide an argument of its correctness as follows. First, observe that we can convert the input LP into an equivalent LP with all entries $b_e = c_S = 1$ by simply replacing each $A_{e,S}$ with $\frac{A_{e,S}}{b_e c_S}$. Namely, let the new parameters be $\mathbf{A}', \mathbf{b}'$ and $\mathbf{c}'$, and we consider the variable $\mathbf{x}'$ where $x'_S = c_S x_S$. It is straightforward to verify that $\mathbf{c}'^\top \mathbf{x}' = \mathbf{c}^\top \mathbf{x}$ and $\mathbf{A}'_e \mathbf{x}' = \frac{\mathbf{A}_e \mathbf{x}}{b_e}$, reducing the LP into the desired case. Thus, we may afford to record $\mathbf{b}$ and $\mathbf{c}$, so that each value $\frac{A_{e,S}}{b_e c_S}$ may be computed on-the-fly. Henceforth we assume that all entries $b_e = c_S = 1$ and $A_{e,S} \in \{0\} \cup [L, U]$. Observe as well that the optimal objective value k may be in the expanded range $[1/U, n/L]$, so the number of guesses must be increased from $\frac{\log n}{\varepsilon}$ to $\frac{\log(nU/L)}{\varepsilon}$.

Next consider the process for covering the rare elements. We instead use a uniform solution $\mathbf{x}^{\mathsf{cmn}} = \frac{\alpha \ell L}{m} \cdot \mathbf{1}$. Observe that if an element occurs in at least $\frac{m}{\alpha \ell L}$ sets, then $\mathbf{A}_e \mathbf{x}^{\mathsf{cmn}} = \sum_{S: e \in S} A_{e,S} \cdot \frac{\alpha \ell}{m} \geq \frac{m}{\alpha \ell L} \cdot L \cdot \frac{\alpha \ell}{m} = 1$. That is, we must adjust our definition so that an element is considered common if it appears in at least $\frac{m}{\alpha \ell L}$ sets. Consequently, whenever we perform element sampling, the required amount of memory to store information of each element increases by a factor of $1/L$.

Next consider Lemma 4.3.5, where we show an existence of integral solutions via the MWU algorithm with a greedy oracle. As the greedy implementation chooses a set S and places the entire budget ℓ on x_S, the amount of coverage $A_{e,S} x_S$ may be as large as ℓU as $A_{e,S}$ is no longer bounded by 1. Thus this application of the MWU algorithm has width $\phi = \Theta(\ell U)$ and requires $T = \Theta(\frac{\ell U \log n}{\varepsilon_{\text{MC}}^2})$ rounds. Consequently, its solution becomes $\Theta(\frac{\ell}{T}) = \Theta(\frac{\varepsilon_{\text{MC}}^2}{U \log n})$-integral. As noted in Observation 4.3.7, the number of potential solutions from the greedy oracle increases by a power of U. Then, in Lemma 4.3.8, we must reduce the error probability of each solution by the same power. We increase the number of samples s by a factor of U to account for this change, increasing the required amount of memory by the same factor.

As in the previous case, any solution $\mathbf{x}$ may always be pruned so that the width is reduced to 1: our algorithm **prune** still works as long as the entries of $\mathbf{A}$ are non-negative (Lemma 4.3.6). Therefore, the fact that entries of $\mathbf{A}$ may take on values other than 0 or 1 does not affect the number of rounds (or passes) of our overall application of the MWU framework. Thus, we may handle general covering LPs using a factor of $\widetilde{O}(U/L)$ larger memory within the same number of passes. In particular, if the non-zero entries of the input are bounded in the range $[1, M]$, this introduces a factor of $\widetilde{O}(U/L) \leq \widetilde{O}(M^3)$ overhead in memory usage. □

4.4 Overview of Fractional Set Cover in the Oracle Access Model

We now revisit the oracle access model and consider the Fractional Set Cover problem. As previously mentioned, the fractional variant can be approximated up to a factor arbitrarily close to 1, using an algorithm that runs in nearly-linear time. In particular, [KY14], via a reduction to an algorithm

from [GK95], showed that an $O(1)$-approximate solution to the fractional version of the problem can be found in $\widetilde{O}(mk^2 + nk^2)$ probe complexity. The latter paper employs the "randomized fictitious play" technique for approximately solving a two-player zero-sum game, that could be viewed as a randomized variant of the MWU method. We re-emphasize that the algorithm of [KY14] can be further improved to $\widetilde{O}(m + nk)^2$, which is currently the best known algorithm for this problem.

4.4.1 Our results and techniques

In this work, we show two algorithms which offer tradeoffs between the probe complexity and the running time. Both algorithms employ MWU framework [AHK12], which iteratively identifies unsatisfied constraints and re-calibrates their "importance". The algorithms differ in the way the constraint checking procedure is implemented. The first algorithm implements each constraint checking round separately by using random sampling. This bounds the running time (and the probe complexity) by the product of the number of rounds and the number of samples taken in each round, which is $\widetilde{O}(\frac{mk}{\varepsilon^5} + \frac{nk}{\varepsilon^2})$.

To reduce the probe complexity, the second algorithm relies on the key observation that the distribution from which the random samples are taken changes very little from round to round. This makes it possible to (re-)use random samples in several rounds. However, the updates to the distribution can be *adaptive*, i.e., they depend on the outcome of the random sampling process in the previous rounds. This makes it impossible to guarantee the correctness of the algorithm by employing a simple union bound. To overcome this challenge, we cast our algorithm in the *adaptive data analysis* framework of [DFH+15, BNS+16], which handles the dependencies by utilizing only a limited amount of information about the samples in each round. To the best of our knowledge this is the first application of the framework to the design of sub-linear algorithms. We show that this approach significantly reduces the probe complexity (by roughly a factor of $\sqrt{k}$), at the price of increasing the running time by (roughly) the same factor. Note that the running times of both algorithms are sub-linear in the input size for low values of k.

We partially complement the aforementioned upper bounds by showing that for small values of k, the probe complexity of any algorithm for fractional set cover must depend linearly on both m and n. Unlike the integral case where the lower bound construction employs the probabilistic method, the lower bound construction of the fractional case is more explicit. A new key idea of this lower bound construction is that, we leverage the LP duality to prove that the constructed instances have large minimum fractional set cover, by providing large solutions to the dual packing LPs.

Finally, we also show a variant of the algorithm which is most efficient when k is large. Our algorithmic results and lower bounds are presented in Table 4.1. While we may not accomplish the state-of-the-art algorithm in this work, we demonstrate an interesting framework for designing and improving algorithms based on the MWU method that makes a meaningful connection to the field of adaptive data analysis. We expect that this technique may find its applications in other contexts, especially in improving the complexities of MWU-based algorithms.

[2] N. Young, personal communication

Probe Complexity	Extra Running Time	Section
$\widetilde{O}\left(\frac{m\sqrt{k}}{\varepsilon^{9/2}}+\frac{nk}{\varepsilon^2}\right)$	$\widetilde{O}\left(\frac{mk^{3/2}}{\varepsilon^{11/2}}\right)$	4.5.1.1
$\widetilde{O}\left(\frac{mk}{\varepsilon^5}+\frac{nk}{\varepsilon^2}\right)$	-	4.5.1.3
$\widetilde{O}\left(\frac{mn}{k\varepsilon}\right)$	$\widetilde{O}\left(\frac{m+n}{\varepsilon^2}\right)$	4.5.2
$\Omega\left(\frac{n+m}{k}\right)$		4.6

Table 4.1: A summary of our algorithms and lower bounds for computing an $(1+\epsilon)$-approximate solution to Fractional Set Cover, where k denote the optimal value. For the lower bound, ϵ is assumed to be a sufficiently small constant.

4.5 Sub-Linear Algorithms for the Fractional Set Cover Problem

In this section, we present a sublinear-time algorithm that computes a $(1+\varepsilon)$-approximate solution of the LP-relaxation of Set Cover for any $\varepsilon > 0$. We retain our notation from the streaming setting.

Overview of the algorithms. Let k denote the optimal objective value, and $\varepsilon > 0$ be a parameter. Our approach combines two algorithms that are efficient for different values of k. Both algorithms first pick a *fractional* vector that covers *frequently occurring* (which will be defined precisely shortly) elements; one can interpret this step as the fractional version of Set Sampling described in more details in Section 3.2.1. The two algorithms handle the remaining elements differently.

For the first algorithm, **smallFracCover**, we create a subroutine **feasibilityTest-Small** based on the multiplicative weights update method (MWU) such that, given a parameter ℓ, with high probability, either returns a solution of objective value at most $(1+\varepsilon/3)\ell$, or detects that the optimal objective exceeds ℓ. Consequently, we may search for the right value ℓ in two steps. First we consider values ℓ in an increasing power of 2 until we find a value ℓ yielding a valid solution. Then we perform a binary search between $\ell/2$ and ℓ for $O(\log 1/\varepsilon)$ iterations to obtain a $(1+\varepsilon/3)$-approximation to the value of k, which results in a $(1+\varepsilon)$-approximate solution overall. This overall algorithm is described as **smallFracCover** in Figure 4-10.

Assuming that the complexities of **feasibilityTest-Small** are upper bounded by those of the execution with the largest value of ℓ (which is $\Theta(k)$), the whole process of searching k increases our asymptotic complexities by at most a multiplicative factor of $\log k + \log 1/\varepsilon$. The details of **feasibilityTest-Small** is given in Section 4.5.1. The main challenge of this approach lies in designing the oracle for the MWU framework and updating weights using only a sub-linear number of probes. To this end, we employ recent results in adaptive data analysis from [BNS+16].

To handle the case where ℓ is large, we propose the simpler second algorithm, **largeFracCover**, using a subroutine **feasibilityTest-Large**, which is very much the same as the algorithm described in Section 3.2.3, offering a similar guarantee to the MWU based approach, but requires $\widetilde{O}(mn/\ell)$ probes. Now the complexity is inversely proportional to ℓ. Thus in the first step we consider ℓ

smallFracCover(ε):

▷ Find a feasible 2-approximate solution in $O(\log k)$ iterations

for $\ell \in \{2^i \mid 0 \le i \le \log n\}$ **do in the increasing order**:

  try $\mathbf{x} \leftarrow$ **feasibilityTest-Small**$(\ell, \varepsilon/3)$

  if feasibilityTest-Small$(\ell, \varepsilon/3)$ reports INFEASIBLE **then continue**

  else $\mathbf{x}^{\mathsf{best}} \leftarrow \mathbf{x}$ and **break**

$L \leftarrow \ell/2, H \leftarrow \ell$

▷ Binary search for a $(1+\varepsilon/3)$-approximation using $\log 1/\varepsilon$ more iterations

repeat $\Theta(\log 1/\varepsilon)$ iterations

  try $\mathbf{x} \leftarrow$ **feasibilityTest-Small**$(\frac{L+H}{2}, \varepsilon/3)$

  if feasibilityTest-Small$(\frac{L+H}{2}, \varepsilon/3)$ reports INFEASIBLE **then** $L \leftarrow \frac{L+H}{2}$

  else $\mathbf{x}^{\mathsf{best}} \leftarrow \mathbf{x}$, $H \leftarrow \frac{L+H}{2}$

return $\mathbf{x}^{\mathsf{best}}$

Figure 4-10: **smallFracCover** returns a $(1+\varepsilon)$-approximate solution of SetCover-LP, where **feasibilityTest-Small** is an algorithm that returns a solution of objective value at most $(1+\varepsilon/3)\ell$ when $\ell \ge k$.

in the decreasing order, before applying binary search to compute a $(1+\varepsilon)$-approximate solution, as described in Section 4.5.2. We may combine the two algorithms by running them in parallel, answering the probes for the algorithms alternatively, until one algorithm returns an answer. The result is an algorithm with probe complexity $\widetilde{O}(\min\{m\sqrt{k} + nk, mn/k\}) = \widetilde{O}(n\sqrt{m} + m\sqrt[3]{n})$.

4.5.1 Efficient algorithm for instances with small optimal value

Our algorithm follows the same general framework of the streaming setting. First, we begin by creating $\mathbf{x}^{\mathsf{cmn}}$ that covers the common elements. Next, we must construct an efficient MWU-based algorithm, which finds a solution $\mathbf{x}^{\mathsf{rare}}$ covering the rare elements, with objective value at most $\frac{\ell}{1-\beta} \le (1+\varepsilon-\alpha)\ell$. To do so, consider the MWU algorithm applied with $\phi = \ell$, which requires $\Theta(\ell \log n/\beta^2)$ iterations (Theorem 4.2.2). In each iteration, we need an oracle that finds some solution $\mathbf{x} \in \mathcal{P}_\ell$ satisfying $\mathbf{p}^\top \mathbf{A}\mathbf{x} \ge \mathbf{p}^\top \mathbf{b} - \beta/3$, or decides that no solution in $\mathcal{P}_\ell$ satisfies $\mathbf{p}^\top \mathbf{A}\mathbf{x} \ge \mathbf{p}^\top \mathbf{b}$. We make the following useful observation.

Lemma 4.5.1. *Define $p_S \triangleq \sum_{e \in S} p_e$, the total probability of elements in S. Assume $\mathbf{b} = \mathbf{1}$. The constraint $\mathbf{p}^\top \mathbf{A}\mathbf{x} \ge \mathbf{p}^\top \mathbf{b} - \beta/3$ has a solution in $\mathcal{P}_\ell$ if $p_S \ge (1-\beta/3)/\ell$ for some $S \in \mathcal{F}$. Conversely, if $p_S < 1/\ell$ for all $S \in \mathcal{F}$ then the constraint $\mathbf{p}^\top \mathbf{A}\mathbf{x} \ge \mathbf{p}^\top \mathbf{b}$ has no solution in $\mathcal{P}_\ell$.*

Proof. Observe that $p_S = (\mathbf{p}^\top \mathbf{A})_S$ and $\mathbf{p}^\top \mathbf{b} = 1$. Clearly, this constraint $\mathbf{p}^\top \mathbf{A}\mathbf{x} \ge \mathbf{p}^\top \mathbf{b} - \beta/3 = 1 - \beta/3$ can be solved greedily, by simply choosing a set S with $p_S \ge \frac{1-\beta/3}{\ell}$ and assign $x_S = \frac{1-\beta/3}{p_S}$ (and $x_{S'} = 0$ for all other sets S'). On the other hand, if $p_S < 1/\ell$ for all sets, then increasing any entry of $\mathbf{x}$ by some positive amount may increase the value of $\mathbf{p}^\top \mathbf{A}\mathbf{x}$ by strictly less than $1/\ell$ per unit amount. Thus, for any $\mathbf{x} \in \mathcal{P}_\ell$, $\mathbf{p}^\top \mathbf{A}\mathbf{x} < 1 = \mathbf{p}^\top \mathbf{b}$. □

Our implementation of **feasibilityTest-Small** will make use of the solution suggested in this lemma. We note that our implementation does not explicitly maintain the weight vector $\mathbf{w}^t$ de-

scribed in Section 4.5.1, but instead updates (and normalizes) its probability vector $\mathbf{p}^t$ in every round.

4.5.1.1 First attempt: Independent sampling approach

We would like to find a set with large p_S without using a large number of probes. Observe that if we sample an element according to the distribution $\mathbf{p}$, then the probability that we obtain an element $e \in S$ is exactly $\sum_{e \in S} p_e = p_S$. Thus, by sampling $\widetilde{\Theta}(\ell/\beta^2)$ elements, via the Chernoff bound, we will be able to detect a set with sufficiently large p_S or decide that none exists. For $T = \widetilde{\Theta}(\ell/\beta^2)$ rounds of the MWU framework, this approach would require $\widetilde{\Theta}(\ell^2/\beta^4)$ samples.

In Figure 4-11, **feasibilityTest-IS**, we provide the implementation of **feasibilityTest-Small** with a fresh set of samples for each round of MWU. This sampling-based oracle is implemented as **heavySet** in Figure 4-12. Observe that this implementation is considerably more complicated than the **heavySetOracle** counterpart since we do not have access to the all the values p_S. By an application of the Chernoff bound as outlined above, the following lemma may be proved for **heavySet**:

Lemma 4.5.2. *Consider the execution of* **heavySet** *with probability distribution* $\mathbf{p}$*, and let* S *be the set returned by the subroutine. If* $p_S < (1 - \beta/3)/\ell$*, then w.h.p.* $\max_{S^* \in \mathcal{F}} p_{S^*} < 1/\ell$.

We omit the proof as it is essentially a simpler version of the proof of Lemma 4.5.7. In other words, w.h.p., upon verifying p_S, **heavySet** implements a $(1, \ell)$-bounded $\beta/3$-approximate oracle for the polytope $\mathcal{P}_\ell$, the linear system $\mathbf{Ax} \geq \mathbf{b}$ and probability distribution $\mathbf{p}$. Therefore, **feasibilityTest-IS** computes a solution of objective value at most $(\alpha + \frac{1}{1-\beta})\ell < (1 + \frac{\varepsilon}{3})\ell$ when $\ell \geq k$ (recall that **heavySet** is invoked by **feasibilityTest-IS** with parameter $\varepsilon/3$). Our process **smallFracCover** eventually computes a value ℓ satisfying $k \leq \ell \leq (1 + \frac{\varepsilon}{3})k$; hence, we obtain a $(1 + \frac{\varepsilon}{3})(1 + \frac{\varepsilon}{3}) < (1 + \varepsilon)$-approximate solution to SetCover-LP.

Our subroutine **heavySet** performs $\Theta(\frac{\ell \log mn}{\beta^2})$ rounds of sampling, each of which takes $O(\frac{m}{\alpha\ell})$ SETOF probes by the guarantee of $\mathbf{x}^{\mathsf{cmn}}$. As **heavySet** returns exactly one set, **feasibilityTest-IS** requires n ELTOF probes to compute VAL and update the probability vector $\mathbf{p}^{\mathsf{curr}}$. Thus, the probe complexity of each MWU round is $O(\frac{m \log mn}{\alpha\beta^2} + n)$. The subroutine **feasibilityTest-IS** makes n SETOF probes to detect common elements, and makes $\Theta(\frac{\ell \log n}{\beta^2})$ calls to **heavySet**, giving a total probe complexity of $n + O(\frac{m \log mn}{\alpha\beta^2} + n) \cdot \Theta(\frac{\ell \log n}{\beta^2}) = \widetilde{O}(\frac{m\ell}{\alpha\beta^4} + \frac{n\ell}{\beta^2})$. We only execute **feasibilityTest-IS** with $\ell = O(k)$, so the probe complexity of **smallSetCover** for invoking each **feasibilityTest-IS** is at most $\widetilde{O}(\frac{mk}{\varepsilon^5} + \frac{nk}{\varepsilon^2})$ per iteration. Since **smallSetCover** increases the overall complexities by only a factor of $\log k + \log 1/\varepsilon$, our overall probe complexity is still asymptotically $(\log k + \log \frac{1}{\varepsilon}) \cdot \widetilde{O}(\frac{mk}{\varepsilon^5} + \frac{nk}{\varepsilon^2}) = \widetilde{O}(\frac{mk}{\varepsilon^5} + \frac{nk}{\varepsilon^2})$.

Assuming that each probe takes constant time, the processing time of the algorithm is dominated by its probe complexity as well, constituting the following claim.

Theorem 4.5.3. *For any constant* $\varepsilon > 0$, **smallSetCover** *computes a* $(1+\varepsilon)$*-approximate solution to* SetCover-LP *with* $\widetilde{O}(\frac{mk}{\varepsilon^5} + \frac{nk}{\varepsilon^2})$ *probes and processing time, with high probability.*

Now, we move on to our second implementation of **feasibilityTest-Small** that achieves more efficient probe complexity, by making use of the recent developments in adaptive data analysis.

feasibilityTest-IS(ℓ, ε):

$\alpha, \beta \leftarrow \frac{\varepsilon}{3}$ ▷ Approximation parameters for covering common and rare elements
$\mathbf{x}^{\mathsf{cmn}} \leftarrow \frac{\alpha\ell}{m} \cdot \mathbf{1}_{m\times 1}$ ▷ $\mathbf{x}^{\mathsf{cmn}}$ is the solution covering all common elements
$\mathbf{p}^{\mathsf{curr}} \leftarrow \mathbf{0}_{m\times 1}$ ▷ Compute the initial probability vector for the MWU algorithm on rare elements
for each $e \in \mathcal{U}$ **do**
    if e appears in less than $\frac{m}{\alpha\ell}$ sets **then** ▷ Single SETOF probe
        $p_e^{\mathsf{curr}} \leftarrow 1$
$\mathbf{p}^{\mathsf{curr}} \leftarrow \frac{\mathbf{p}^{\mathsf{curr}}}{\|\mathbf{p}^{\mathsf{curr}}\|}$ ▷ $\mathbf{p}^{\mathsf{curr}}$ will henceforth represent the current probability vector
$\mathbf{x}^{\mathsf{total}} \leftarrow \mathbf{0}_{m\times 1}$ ▷ Maintain the cumulative solution from the MWU algorithm
repeat $T = \Theta(\frac{\ell \log n}{\beta^2})$ times
    try $S \leftarrow$ **heavySet**$(\mathbf{p}^{\mathsf{curr}}, \ell, \beta/3)$ ▷ Ask the oracle to find a set S with $p_S^{\mathsf{curr}} \geq \frac{1-\beta/3}{\ell}$
    VAL $\leftarrow \sum_{e\in S} p_e^{\mathsf{curr}}$ ▷ Compute VAL $= p_S^{\mathsf{curr}}$ exactly, requiring up to n ELTOF probes
    if VAL $< \frac{1-\beta/3}{\ell}$ **then report** INFEASIBLE ▷ Update the maintained cumulative solution
    $x_S^{\mathsf{total}} \leftarrow x_S^{\mathsf{total}} + \ell$
    for each $e \in S$ **do** ▷ Update the current probability vector
        $p_e^{\mathsf{curr}} \leftarrow (1 - \frac{\beta(\ell-1)}{6\ell})/(1 + \frac{\beta}{6\ell}) \cdot p_e^{\mathsf{curr}}$
    $\mathbf{p}^{\mathsf{curr}} \leftarrow \frac{\mathbf{p}^{\mathsf{curr}}}{\|\mathbf{p}^{\mathsf{curr}}\|}$
$\mathbf{x}^{\mathsf{rare}} \leftarrow \frac{\mathbf{x}^{\mathsf{total}}}{(1-\beta)T}$ ▷ Solution from the MWU method, scaled up to cover rare elements
return $\mathbf{x}^{\mathsf{cmn}} + \mathbf{x}^{\mathsf{rare}}$

Figure 4-11: **feasibilityTest-IS** is an implementation of **feasibilityTest-Small** that uses a new set of independent samples in each round of MWU.

heavySet$(\mathbf{p}, \ell, \beta)$:

▷ Approximate the total probability of elements in each set, p_S
$\mathbf{c} \leftarrow \mathbf{0}_{m\times 1}$ ▷ c_S counts the number of samples that are also in S
repeat $r = \Theta(\frac{\ell}{\beta^2} \log mn)$ times
    sample an element e according to the probability distribution $\mathbf{p}$
    for each $S \ni e$ ▷ $O(\frac{m}{\alpha\ell})$ SETOF probes
        $c_S \leftarrow c_S + 1$
return $\operatorname{argmax}_{S\in\mathcal{F}} c_S$ ▷ $\mathbf{E}[c_S] = r \cdot p_S$

Figure 4-12: **heavySet** returns a set maximizing c_S, its approximation for $r \cdot p_S$. With high probability, it correctly finds a set S satisfying $p_S \geq \frac{1-\beta/3}{\ell}$ when there exists some set S^* satisfying $p_{S^*} \geq \frac{1}{\ell}$. The number of samples required is $r = \widetilde{\Theta}(\frac{\ell}{\beta^2})$.

4.5.1.2 Preliminaries of Adaptive Data Analysis

At a high level, we will implement the MWU algorithm in a probe-efficient way by taking *samples* from the probability distribution $\mathbf{p}$. To make the algorithm as probe-efficient as possible, we want to actually *reuse* the samples across multiple rounds of the MWU algorithm, but doing so introduces dependencies between the samples and $\mathbf{p}$ that make standard probabilistic analysis impossible. To address this, we will use results from *adaptive data analysis*, specifically the recent work of [BNS+16].

Consider a *data set* $\mathbf{D}$, which is a sequence of *data points* $\langle \mathbf{d}_1, \ldots, \mathbf{d}_N \rangle$, where each $\mathbf{d}_i$ is sampled independently from a distribution $\mathcal{D}$ representing the *population*, i.e. $\mathbf{D} \sim \mathcal{D}^N$. There is also an

analyst $\mathcal{A}$ who does not know $\mathbf{D}$. This analyst asks a sequence of K probes $q_1, \ldots, q_K$ about $\mathcal{D}$. Most importantly, $\mathcal{A}$ may choose his probes adaptively based on the previous probe-answer pairs. The goal is to create a *mechanism* $\mathcal{M}$ that, given $\mathbf{D}$ as an input, approximately answers the analysts probes, while minimizing the number of samples N. The analyst may be completely adversarial, and may be designed to learn about $\mathbf{D}$, and adversarially force $\mathcal{M}$ to eventually return an inaccurate answer by asking specifically for some aspect of $\mathcal{D}$ that is not well-represented by $\mathbf{D}$. We remark that in our setting, the analyst also knows $\mathcal{D}$.

For our application, we consider *optimization probes*. Each probe q is specified by a function $f(\mathbf{d}, \psi)$, and asks for a parameter ψ from a set of parameters Ψ that maximizes the expected value of $f(\cdot, \psi)$ evaluated on a random $\mathbf{d} \sim \mathcal{D}$. We denote this objective by $\overline{f}(\mathcal{D}; \psi) \triangleq \mathbf{E}_{\mathbf{d} \sim \mathcal{D}}[f(\mathbf{d}; \psi)]$. We focus on the case where $f(\cdot, \psi)$ is a predicate function taking values from $\{0, 1\}$ for every $\psi \in \Psi$. That is, we wish to find a parameter $\psi \in \Psi$ such that the predicate $f(\mathbf{d}; \psi)$ holds for the largest fraction of data points $\mathbf{d}$ from the population $\mathcal{D}$. We also define $\tilde{f}(\mathbf{D}; \psi) \triangleq \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{d}_i; \psi)$, the empirical fraction of data points from $\mathbf{D}$ for which the predicate $f(\mathbf{d}_i; \psi)$ holds. We expect $\tilde{f}(\mathbf{D}; \psi)$ to be an accurate estimator for $\overline{f}(\mathcal{D}; \psi)$ when N is sufficiently large.

We require that $\mathcal{M}$ returns a parameter ψ with certain accuracy compared to the optimal parameter ψ^*. We define the error with respect to the population $\mathcal{D}$ as

$$\mathsf{err}^{\mathcal{D}}(\overline{f}, \psi) \triangleq \mathbf{E}_{\mathbf{D} \sim \mathcal{D}^N} \left[\max_{\psi^* \in \Psi} \tilde{f}(\mathbf{D}; \psi^*) \right] - \mathbf{E}_{\mathbf{D} \sim \mathcal{D}^N} \left[\tilde{f}(\mathbf{D}; \psi) \right].$$

Since $\mathbf{E}_{\mathbf{D} \sim \mathcal{D}^N} \left[\max_{\psi^* \in \Psi} \tilde{f}(\mathbf{D}; \psi^*) \right] \geq \max_{\psi^* \in \Psi} \mathbf{E}_{\mathbf{D} \sim \mathcal{D}^N} \left[\tilde{f}(\mathbf{D}; \psi^*) \right]$, we also have that

$$\mathsf{err}^{\mathcal{D}}(\overline{f}, \psi) \geq \max_{\psi^* \in \Psi} \mathbf{E}_{\mathbf{D} \sim \mathcal{D}^N} \left[\tilde{f}(\mathbf{D}; \psi^*) \right] - \mathbf{E}_{\mathbf{D} \sim \mathcal{D}^N} \left[\tilde{f}(\mathbf{D}; \psi) \right].$$

Moreover, recall that for the family of probes we consider, $\mathbf{E}_{\mathbf{D} \sim \mathcal{D}^N} \left[\tilde{f}(\mathbf{D}; \psi) \right] = \mathbf{E}_{\mathbf{d} \sim \mathcal{D}} \left[f(\mathbf{d}; \psi) \right]$. Hence,

$$\mathsf{err}^{\mathcal{D}}(\overline{f}, \psi) \geq \max_{\psi^* \in \Psi} \mathbf{E}_{\mathbf{d} \sim \mathcal{D}} \left[f(\mathbf{d}; \psi^*) \right] - \mathbf{E}_{\mathbf{d} \sim \mathcal{D}} \left[f(\mathbf{d}; \psi) \right].$$

We now state the theorem we will use in our construction of the approximate oracle. Let the sequence of probes and answers between the analyst $\mathcal{A}$ and the mechanism $\mathcal{M}$ be denoted by $\overline{f}^1, \ldots, \overline{f}^K$ and $\psi^1, \ldots, \psi^K$, respectively. Without loss of generality, $\mathcal{A}$ is deterministic.

Theorem 4.5.4 (Corollary 6.4 of [BNS+16]). *Let Ψ be a finite set of parameters and F be the set of functions of the form $\tilde{f}(\mathbf{D}; \psi) = \frac{1}{N} \sum_{i=1}^{N} f(\mathbf{d}_i; \psi)$ taking values from $[0, 1]$. Then there exists a mechanism $\mathcal{M}$ such that*

$$\mathbf{Pr} \left[\max_{i=1}^{K} \mathsf{err}^{\mathcal{D}}(\overline{f}^i, \psi^i) \leq 0.1 \right] \geq 1 - \delta$$

where the probability is taken over random data points $\mathbf{D} \sim \mathcal{D}^N$ given to $\mathcal{M}$ as well as $\mathcal{M}$'s randomness, for K adaptively chosen probes from F, where

$$N = O\left(\sqrt{K} \cdot \log |\Psi| \cdot \log^{3/2}(\tfrac{1}{\delta}) \right).$$

Moreover, the running time of $\mathcal{M}$ is dominated by $O((K+\log(\frac{1}{\delta}))\cdot|\Psi|)$ total evaluations of function f^i corresponding to the probes $\overline{f}^i$.

4.5.1.3 Second Algorithm: reusing the samples

One idea for reducing the number of required samples is to reuse our samples throughout many rounds. Because the algorithm chooses its solution and updates the probabilities according to these random samples, our next *probe* which asks for a set with large p_S with respect to the new probabilities is *adaptive*. This dependency influences the accuracy of the answer to this next probe, and thus we cannot directly reuse our samples this way.

To circumvent this issue, we make use of the adaptive data analysis. We simulate our oracle via an interaction between an analyst $\mathcal{A}$ (our algorithm) and a mechanism $\mathcal{M}$. Namely, $\mathcal{M}$ takes samples which consist of elements drawn from $\mathbf{p}$, and $\mathcal{A}$ asks $\mathcal{M}$ to *suggest* a set with large p_S. The mechanism $\mathcal{M}$ strategically obfuscates its suggested set S, but as long as it is accurate enough, we can verify S and use it to construct a solution for the approximate oracle. Our algorithm forwards the data set (samples) to the mechanism without ever inspecting it, and the obfuscation made by $\mathcal{M}$ prevents the algorithm from learning the data set. Consequently, $\mathcal{M}$ can continue to provide accurate suggestions throughout a number of rounds, even in the presence of adaptive probes. For the mechanism we are using (in Theorem 4.5.4), $\mathcal{M}$ takes roughly $\sqrt{K}$ rounds' worth of data points, but can reuse the samples and give accurate answers for up to K rounds.

Simulating changing probabilities. The immediate issue that arises from this approach is that our distribution $\mathbf{p}$ for the MWU algorithm changes in every iteration. To resolve this issue, $\mathcal{A}$ must somehow tailor its probe to $\mathcal{M}$ in such a way that $\mathcal{M}$'s answer reflects the probability for the current iteration $\mathbf{p}'$, not the probability $\mathbf{p}$ used to generate the data points for $\mathcal{M}$. In particular, in the sampling and counting scheme suggested above, the contribution of each occurrence of e should not be equal, but instead scaled proportionally to p'_e/p_e. For this approach to work, p'_e/p_e must not increase by a large amount, or we would not have enough samples of these elements. Similarly to the streaming setting, we show that this is indeed the case through the following lemma.[3]

Lemma 4.5.5. *Let $\mathbf{p}$ and $\mathbf{p}'$ be the probability of elements before and after an update, such that the solution from the oracle $\mathbf{x}$ is of the form $x_S = \frac{1-\beta/3}{p_S}$ for some set S with $p_S \geq \frac{1-\beta/3}{\ell}$, and $x_{S'} = 0$ for all other sets S'. Then for every element e, $p'_e \leq (1+\frac{1}{\ell})p_e$.*

Proof. Before normalization, the probability of each $e \in S$ is multiplied by a factor of

$$1 - \frac{\beta(A_{e,S}x_S - b_e)}{6\ell} = 1 - \frac{\beta(\frac{1-\beta/3}{p_S} - 1)}{6\ell} > 1 - \frac{\beta}{6\ell p_S}$$

since $A_{e,S} = 1$, $x_S = \frac{1-\beta/3}{p_S}$, and $\beta < 1$ by assumption. On the other hand, the probability of each $e' \notin S$ is multiplied by a factor of

$$1 - \frac{\beta(A_{e',S}x_S - b_{e'})}{6\ell} = 1 + \frac{\beta}{6\ell}$$

[3]If we set $x_S = \ell$, then this lemma does not necessarily hold.

since $A_{e',S} = 0$. The original total probability of $e \in S$ is p_S and that of $e' \notin S$ is $1 - p_S$, so the total probability before normalization is at least

$$\left(1 - \frac{\beta}{6\ell p_S}\right) p_S + \left(1 + \frac{\beta}{6\ell}\right)(1 - p_S) = 1 - \frac{\beta p_S}{6\ell} \geq 1 - \frac{\beta}{6\ell}$$

since $p_S \leq 1$. Therefore, the probability of any element after normalization is increased from its original probability by at most a factor of

$$\left(1 + \frac{\beta}{6\ell}\right)\left(1 - \frac{\beta}{6\ell}\right)^{-1} < 1 + \frac{\beta}{\ell}$$

since $\beta < 1$ and $\ell \geq 1$. □

The inequality $(1 + \frac{\beta}{\ell})^{\frac{\ell}{\beta}} \leq \mathsf{e}$ (Euler's number)[4] immediately yields the following corollary.

Corollary 4.5.6. *After $\frac{\ell}{\beta}$ iterations of the* MWU *algorithm, the probability of an element may only increase by at most a constant multiplicative factor* e.

Setting up the analyst-mechanism interaction. Now we are ready to elaborate on our second implementation of the **feasibilityTest-Small**, **feasibilityTest-RS** in Figure 4-13. As suggested by Corollary 4.5.6, we divide the T rounds of the MWU algorithm into epochs of ℓ/β rounds each; within each epoch the probability of any element increases by at most a constant factor. At the beginning of each epoch, we create a mechanism $\mathcal{M}$ using the samples drawn from $\mathbf{p}^{\mathsf{mech}}$, which is the same as the probability $\mathbf{p}^{\mathsf{curr}}$ of the current iteration. We pack our samples into s-tuples, each of which is considered a data point, where $s = \Theta(\ell/\beta^2)$. That is, each data point has the format $\mathbf{d} = \langle \hat{e}_1, \ldots, \hat{e}_s \rangle$, and is drawn independently from $\mathcal{D} = (\mathbf{p}^{\mathsf{mech}})^s$. The total number of data points in our data set is $N = \Theta(\sqrt{\frac{\ell}{\beta}}\,\mathrm{polylog}(m, n, \frac{1}{\beta}))$, and thus the number of samples required is $N \cdot s = \widetilde{\Theta}(\ell^{3/2}/\beta^{5/2})$. Our goal is to find a set with large p_S, so in this case, the parameter $\psi \in \Psi$ we aim to choose is simply a set $S \in \mathcal{F}$.

We design the predicate $f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, v}$ for evaluating a data point $\mathbf{d}$ with respect to parameter S. Intuitively, this predicate represents $\mathbf{d}$'s opinion whether $\mathbf{d}$ *supports* the claim that $p_S^{\mathsf{curr}} \geq v$ or not. We first define an unbiased estimator for p_S^{curr} as follows. Let

$$q^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S) \triangleq \frac{1}{s} \sum_{i=1}^{s} \mathbf{1}_S(\hat{e}_i) \frac{p_{\hat{e}_i}^{\mathsf{curr}}}{p_{\hat{e}_i}^{\mathsf{mech}}},$$

where $\mathbf{1}_S(\hat{e}_i)$ is the indicator function for the condition $\hat{e}_i \in S$. Then, we set $f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, v}(\mathbf{d}; S) = 1$ if $q^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S) \geq v$, and set it to 0 otherwise. Note that the factor $p_{\hat{e}_i}^{\mathsf{curr}}/p_{\hat{e}_i}^{\mathsf{mech}}$ is used to rebalance the contribution of each sample, simulating the distribution $\mathbf{p}^{\mathsf{curr}}$ using the samples from $\mathbf{p}^{\mathsf{mech}}$. Subsequently, both $\tilde{f}^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, v}(\mathbf{D}; S)$ and $\overline{f}^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, v}(\mathcal{D}; S)$ are defined on the data set and the population, respectively, as given in Section 4.5.1.2. These two functions represent the fractions of data points in $\mathbf{D}$ and $\mathcal{D}$ that support the claim that $p_S^{\mathsf{curr}} \geq v$.

[4]We consistently write Euler's number e in this font so that it is distinguishable from an element e.

feasibilityTest-RS(ℓ, ε):

$\alpha, \beta \leftarrow \frac{\varepsilon}{3}$ ▷ Approximation parameters for covering common and rare elements
$\mathbf{x}^{\mathsf{cmn}} \leftarrow \frac{\alpha\ell}{m} \cdot \mathbf{1}_{m\times 1}$ ▷ $\mathbf{x}^{\mathsf{cmn}}$ is the solution covering all common elements
$\mathbf{p}^{\mathsf{curr}} \leftarrow \mathbf{0}_{m\times 1}$ ▷ Compute the initial probability vector for the MWU algorithm on rare elements
for each $e \in \mathcal{U}$ **do**
  if e appears in less than $\frac{m}{\alpha\ell}$ sets **then** ▷ Single SETOF probe
    $p_e^{\mathsf{curr}} \leftarrow 1$
$\mathbf{p}^{\mathsf{curr}} \leftarrow \frac{\mathbf{p}^{\mathsf{curr}}}{\|\mathbf{p}^{\mathsf{curr}}\|}$ ▷ $\mathbf{p}^{\mathsf{curr}}$ will henceforth represent the current probability vector
$\mathbf{x}^{\mathsf{total}} \leftarrow \mathbf{0}_{m\times 1}$ ▷ Maintain the cumulative solution from the MWU algorithm
repeat $T' = \Theta(\frac{\log n}{\beta})$ times ▷ We divide $T = \Theta(\frac{\ell \log n}{\beta^2})$ rounds into $T' = \Theta(\frac{\log n}{\beta})$ epochs
  $\mathbf{p}^{\mathsf{mech}} \leftarrow \mathbf{p}^{\mathsf{curr}}$ ▷ $\mathbf{p}^{\mathsf{mech}}$ is the probability distribution of samples given to the mechanism
  let $\mathcal{D} = (\mathbf{p}^{\mathsf{mech}})^s$ where $s = \Theta(\frac{\ell}{\beta^2})$ ▷ Distribution of s-tuples of elements from $\mathbf{p}^{\mathsf{mech}}$
  draw $\mathbf{D} \sim \mathcal{D}^N$ where $N = \Theta(\sqrt{\frac{\ell}{\beta}}\,\mathrm{polylog}(m, n, \frac{1}{\beta}))$ ▷ $\mathbf{D}$ contains $\widetilde{O}(\frac{\ell^{3/2}}{\beta^{5/2}})$ elements in total
  probe for all sets containing elements in $\mathbf{D}$ ▷ $\widetilde{O}(\frac{m\sqrt{\ell}}{\alpha\beta^{5/2}})$ SETOF probes
  create mechanism $\mathcal{M}^{\mathbf{D}}$ on the data set $\mathbf{D}$
  repeat ℓ/β times ▷ Perform each round of the MWU algorithm
    probe $\mathcal{M}^{\mathbf{D}}$ on function $\overline{f}^{\mathbf{p}^{\mathsf{mech}}, \mathbf{p}^{\mathsf{curr}}, \frac{1-\beta/6}{\ell}}$ for an answer S ▷ $\mathcal{M}^{\mathbf{D}}$ suggests a set S
    $\mathrm{VAL} \leftarrow \sum_{e\in S} p_e^{\mathsf{curr}}$ ▷ Compute $\mathrm{VAL} = p_S^{\mathsf{curr}}$ exactly, requiring up to n ELTOF probes
    if $\mathrm{VAL} < \frac{1-\beta/3}{\ell}$ **then report** INFEASIBLE
    $x_S^{\mathsf{total}} \leftarrow x_S^{\mathsf{total}} + \frac{1-\beta/3}{\mathrm{VAL}}$ ▷ Update the maintained cumulative solution
    for each $e \in S$ **do** ▷ Update the current probability vector
      $p_e^{\mathsf{curr}} \leftarrow (1 - \frac{\beta(\frac{1-\beta/3}{\mathrm{VAL}} - 1)}{6\ell})/(1 + \frac{\beta}{6\ell}) \cdot p_e^{\mathsf{curr}}$
    $\mathbf{p}^{\mathsf{curr}} \leftarrow \frac{\mathbf{p}^{\mathsf{curr}}}{\|\mathbf{p}^{\mathsf{curr}}\|}$
$\mathbf{x}^{\mathsf{rare}} \leftarrow \frac{\mathbf{x}^{\mathsf{total}}}{(1-\beta)} \cdot \frac{\beta}{T'\ell}$ ▷ Solution from the MWU method, scaled up to cover rare elements
return $\mathbf{x}^{\mathsf{cmn}} + \mathbf{x}^{\mathsf{rare}}$

Figure 4-13: **feasibilityTest-RS** is an implementation of **feasibilityTest-Small** that saves on the total number of samples by reusing them for several rounds of MWU.

A rather unanimous suggestion. Recall that each data point $\mathbf{d}$ consists of s samples from $\mathbf{p}^{\mathsf{mech}}$. Intuitively, if we assume that $\mathbf{p}^{\mathsf{curr}}$ and $\mathbf{p}^{\mathsf{mech}}$ are similar, then for a sufficiently large s, each data point should be able to provide a rather accurate estimation of p_S^{curr}. Consequently, we could expect the data points to be almost unanimous whether they support the claim that $p_S^{\mathsf{curr}} \geq v$ or not. We formalize this intuition for the case where $\mathbf{p}^{\mathsf{curr}}$ and $\mathbf{p}^{\mathsf{mech}}$ are within the same epoch, via the following lemma.

Lemma 4.5.7. *Let* $\mathbf{p}^{\mathsf{curr}}$ *and* $\mathbf{p}^{\mathsf{mech}}$ *be probabilities of elements according to our* MWU *algorithm, where* $\mathbf{p}^{\mathsf{curr}}$ *occurs at most* β/ℓ *iterations after* $\mathbf{p}^{\mathsf{mech}}$*. If each data point* $\mathbf{d}$ *consists of* s *independent samples from* $\mathbf{p}^{\mathsf{mech}}$ *where* $s \geq \frac{1352\ell}{\beta^2}$*, then:*

(a) If $p_S^{\mathsf{curr}} \geq 1/\ell$*, then* $\mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S)] \geq 0.9$.

(b) If $p_S^{\mathsf{curr}} < (1 - \beta/3)/\ell$*, then* $\mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S)] \leq 0.1$.

Proof. Consider the first statement. It is sufficient to show that with probability at least 0.9, a random data point $\mathbf{d} \sim \mathcal{D}$ satisfies $f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S) = 1$. In other words, the data point

$\mathbf{d} = \langle \hat{e}_1, \ldots, \hat{e}_s \rangle$ supports the claim that $p_S^{\mathsf{curr}} \geq \frac{1-\beta/6}{\ell}$ because its approximated value

$$q^{\mathbf{p}^{\mathsf{curr}},\mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S) = \frac{1}{s}\sum_{i=1}^{s} \mathbf{1}_S(\hat{e}_i)\frac{p_{\hat{e}_i}^{\mathsf{curr}}}{p_{\hat{e}_i}^{\mathsf{mech}}} \geq \frac{1-\beta/6}{\ell}.$$

To apply the Chernoff bound, we define a random variable $Y_i = \mathbf{1}_S(\hat{e}_i)\frac{p_{\hat{e}_i}^{\mathsf{curr}}}{\mathsf{e}\cdot p_{\hat{e}_i}^{\mathsf{mech}}}$ and $Y = \sum_{i=1}^{s} Y_i = \frac{s}{\mathsf{e}}\cdot q^{\mathbf{p}^{\mathsf{curr}},\mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S)$. Observe that $Y_i \in [0,1]$ because $p_{\hat{e}_i}^{\mathsf{curr}} \leq \mathsf{e}\cdot p_{\hat{e}_i}^{\mathsf{mech}}$ within the same epoch. Moreover, since each $\hat{e}_i$ is drawn independently from $\mathbf{p}^{\mathsf{mech}}$,

$$\mathbf{E}[Y_i] = \sum_{\hat{e}_i \in \mathcal{U}} \mathbf{1}_S(\hat{e}_i)\frac{p_{\hat{e}_i}^{\mathsf{curr}}}{\mathsf{e}\cdot p_{\hat{e}_i}^{\mathsf{mech}}} p_{\hat{e}_i}^{\mathsf{mech}} = \frac{1}{\mathsf{e}}\sum_{\hat{e}_i \in S} p_{\hat{e}_i}^{\mathsf{curr}} = \frac{p_S^{\mathsf{curr}}}{\mathsf{e}}$$

and $\mathbf{E}[Y] = \frac{s\cdot p_S^{\mathsf{curr}}}{\mathsf{e}} \geq \frac{s}{\mathsf{e}\ell}$ by assumption. Then,

$$\begin{aligned}\mathbf{Pr}\left[q^{\mathbf{p}^{\mathsf{curr}},\mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S) < \frac{1-\beta/6}{\ell}\right] &= \mathbf{Pr}\left[\frac{\mathsf{e}}{s}Y < \frac{1-\beta/6}{\ell}\right] = \mathbf{Pr}\left[Y < \left(1-\frac{\beta}{6}\right)\frac{s}{\mathsf{e}\ell}\right] \\ &\leq \mathbf{Pr}\left[Y < \left(1-\frac{\beta}{6}\right)\mathbf{E}[Y]\right] \leq e^{-\frac{1}{2}(\frac{\beta}{6})^2\mathbf{E}[Y]} \leq e^{-\frac{1}{2}(\frac{\beta}{6})^2\frac{s}{\mathsf{e}\ell}}.\end{aligned}$$

The probability above is bounded by 0.1 when $s \geq \frac{451\ell}{\beta^2}$.

Now consider the second statement; we need to show that with probability at most 0.1, a random data point $\mathbf{d} \sim \mathcal{D}$ satisfies $q^{\mathbf{p}^{\mathsf{curr}},\mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S) > \frac{1-\beta/6}{\ell}$. Define Y_i's and Y in the same fashion as the first case. Since each $Y_i \in [0,1]$ and $\mathbf{E}[Y_i] = \frac{p_S^{\mathsf{curr}}}{\mathsf{e}} < \frac{1-\beta/3}{\mathsf{e}\ell}$, we can define a different random variable Y_i' that stochastically dominates Y_i while satisfying $Y_i' \in [0,1]$ and $\frac{1-\beta/2}{\mathsf{e}\ell} \leq \mathbf{E}[Y_i'] < \frac{1-\beta/3}{\mathsf{e}\ell}$. (Namely, if $\mathbf{E}[Y_i] < \frac{1-\beta/2}{\mathsf{e}\ell}$ then move some of its probability mass in its probability density function to a higher value in $[0,1]$.) Consequently $Y' = \sum_{i=1}^{s} Y_i'$ stochastically dominates Y, and $\mathbf{E}[Y'] \geq \frac{(1-\beta/2)s}{\mathsf{e}\ell}$. Using a similar concentration analysis,

$$\begin{aligned}\mathbf{Pr}\left[q^{\mathbf{p}^{\mathsf{curr}},\mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S) > \frac{1-\beta/6}{\ell}\right] &= \mathbf{Pr}\left[\frac{\mathsf{e}}{s}Y > \frac{1-\beta/6}{\ell}\right] \leq \mathbf{Pr}\left[\frac{\mathsf{e}}{s}Y' > \frac{1-\beta/6}{\ell}\right] \\ &= \mathbf{Pr}\left[Y' > \left(1-\frac{\beta}{6}\right)\frac{s}{\mathsf{e}\ell}\right] \leq \mathbf{Pr}\left[Y' > \left(1+\frac{\beta}{6}\right)\mathbf{E}[Y']\right] \\ &\leq e^{-\frac{1}{3}(\frac{\beta}{6})^2\mathbf{E}[Y']} \leq e^{-\frac{1}{3}(\frac{\beta}{6})^2\frac{(1-\beta/2)s}{\mathsf{e}\ell}} < e^{-\frac{1}{3}(\frac{\beta}{6})^2\frac{s}{2\mathsf{e}\ell}}.\end{aligned}$$

The probability above is bounded by 0.1 when $s \geq \frac{1352\ell}{\beta^2}$. □

Putting it all together. We finally apply Theorem 4.5.4 from adaptive data analysis to conclude the correctness of our simulation of the approximate oracle.

Lemma 4.5.8. *The probe-answer interaction between the analyst $\mathcal{A}$ (our algorithm) and the mechanism $\mathcal{M}^{\mathbf{D}}$ where $\mathbf{D}$ is the data set with $N = \widetilde{\Theta}(\sqrt{\ell/\beta})$ data points of s-tuples of elements drawn from $\mathbf{p}^{\mathsf{mech}}$, simulates a $(1,\ell)$-bounded $\beta/3$-approximate oracle with respect to the probability $\mathbf{p}^{\mathsf{curr}}$ in the same epoch as $\mathbf{p}^{\mathsf{mech}}$, with high probability over the entire execution of the algorithm.*

Proof. First, it is easy to verify that if we can find a set S with $p_S^{\mathsf{curr}} \geq (1-\beta/3)/\ell$, then our solution from Lemma 4.5.1 satisfies the $(1,\ell)$-bounded and $\beta/3$-approximate conditions required by the oracle. As justified by Lemma 4.5.1, it is sufficient to show that at least one of the following cases holds: $\mathcal{M}^{\mathbf{D}}$ returns a set S satisfying $p_S^{\mathsf{curr}} \geq (1-\beta/3)/\ell$, or for every set S, $p_S^{\mathsf{curr}} < 1/\ell$.

Let us apply Theorem 4.5.4 with number of parameters $|\Psi| = m$ (because $\Psi = \mathcal{F}$ in our case), number of probes $K = \frac{\ell}{\beta}$ (number of rounds per epoch), and $\delta = 1/\mathrm{poly}(m, n, \frac{1}{\beta})$ (overall error probability). Then, for every probe-answer pair $(\overline{f}, S)$ during the entire execution of the algorithm, $\mathsf{err}^{\mathcal{D}}(\overline{f}, S) \leq 0.1$ holds with high probability. That is,

$$\max_{S^* \in \mathcal{F}} \mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f(\mathbf{d}; S^*)] - \mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f(\mathbf{d}; S)] \leq 0.1.$$

First, suppose that there exists a set S' such that $p_{S'}^{\mathsf{curr}} \geq 1/\ell$; in this case we need $\mathcal{M}^{\mathbf{D}}$ to return a set S with $p_S^{\mathsf{curr}} \geq (1-\beta/3)/\ell$. Since $p_{S'}^{\mathsf{curr}} \geq 1/\ell$, by Lemma 4.5.7a, $\mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S')] \geq 0.9$ and thus $\max_{S^* \in \mathcal{F}} \mathbf{E}_{\mathbf{d}\sim\mathcal{D}}\left[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S^*)\right] \geq 0.9$ as well. Then for the answer S returned by $\mathcal{M}^{\mathbf{D}}$,

$$\mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S)] \geq \max_{S^* \in \mathcal{F}} \mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S^*)] - 0.1 \geq 0.8.$$

Thus, by Lemma 4.5.7b, $p_S^{\mathsf{curr}} \geq (1-\beta/3)/\ell$.

Similarly, now suppose that for the answer S returned by $\mathcal{M}^{\mathbf{D}}$, $p_S^{\mathsf{curr}} < (1-\beta/3)/\ell$. In this case we must show that $p_{S'}^{\mathsf{curr}} < 1/\ell$ for every $S' \in \mathcal{F}$. By Lemma 4.5.7b, $\mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S)] \leq 0.1$. We then have

$$\max_{S^* \in \mathcal{F}} \mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S^*)] \leq \mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S)] + 0.1 \leq 0.2.$$

This implies that for every set $S' \in \mathcal{F}$, $\mathbf{E}_{\mathbf{d}\sim\mathcal{D}}[f^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}, \frac{1-\beta/6}{\ell}}(\mathbf{d}; S')] \leq 0.2$. Thus $p_{S'}^{\mathsf{curr}} < 1/\ell$ by Lemma 4.5.7a. □

By applying Theorem 4.2.2 to Lemma 4.5.8 above, we have shown that either we return the solution $\mathbf{x}^{\mathsf{rare}}$ with objective value at most $\frac{\ell}{1-\beta}$, or the constraint $\mathbf{Ax} \geq \mathbf{b}$ has no solution in $\mathcal{P}_\ell$.

Probe complexity. The probe complexity can be derived straightforwardly from the pseudocode. In each epoch of our **feasibilityTest-RS** algorithm, we draw $\widetilde{O}(\frac{\ell^{3/2}}{\beta^{5/2}})$ elements and probe for all sets containing them. Since these are rare elements, each of the sampled elements requires $O(\frac{m}{\alpha\ell})$ SETOF probes. Over $T' = \widetilde{\Theta}(\frac{1}{\beta})$ epochs, $\widetilde{O}(\frac{m\sqrt{\ell}}{\alpha\beta^{7/2}})$ SETOF probes are required.

In each round, once we choose a set S, we need to make $O(n)$ ELTOF probes to learn all of its elements in order to compute p_S as well as to update the solution and probabilities. Over $T = \widetilde{\Theta}(\frac{\ell}{\beta^2})$ rounds, $\widetilde{O}(\frac{n\ell}{\beta^2})$ ELTOF probes are required. Altogether, for a guess ℓ of the optimal objective value, the probe complexity for **feasibilityTest-RS** is $\widetilde{O}(\frac{m\sqrt{\ell}}{\varepsilon^{9/2}} + \frac{n\ell}{\varepsilon^2})$. The probe complexity of **smallFracCover** with **feasibilityTest-RS** implementation of **feasibilityTest-Small** becomes $O(\log k + \log\frac{1}{\varepsilon}) \cdot \widetilde{O}(\frac{m\sqrt{k}}{\varepsilon^{9/2}} + \frac{nk}{\varepsilon^2}) = \widetilde{O}(\frac{m\sqrt{k}}{\varepsilon^{9/2}} + \frac{nk}{\varepsilon^2})$, as concluded in the theorem below. This probe complexity is $\widetilde{O}(m\sqrt{k} + nk)$ for constant $\varepsilon > 0$.

Time complexity. The time complexity of **feasibilityTest-RS** is mostly asymptotically dominated by its probe complexity, except for the time spent simulating the mechanism. Within each epoch, the mechanism's running time is dominated by $\widetilde{O}(\frac{m\ell}{\beta})$ evaluations of the function f. Since we have full knowledge about the set structure of the sampled elements, we can create a hash table so that checking the membership (whether $e \in S$) takes constant time. Thus each evaluation of f takes $O(s) = \widetilde{O}(\frac{\ell}{\beta^2})$ time. The total running time for the mechanism over the entire execution of **feasibilityTest-RS** becomes $\widetilde{\Theta}(\frac{1}{\beta}) \cdot \widetilde{O}(\frac{m\ell}{\beta}) \cdot \widetilde{O}(\frac{\ell}{\beta^2}) = \widetilde{O}(\frac{m\ell^2}{\beta^4})$.

However, suppose that we evaluate f on a data point $\mathbf{d}$ for every set S as we probe for the sets containing the elements in $\mathbf{d}$, then we store the results. In this case, the running time is simply $\frac{m}{\alpha\ell} \cdot O(s) = \widetilde{O}(\frac{m}{\alpha\beta^2})$, which is more efficient in an amortized perspective. Thus the running time for evaluating f on all data points in each epoch takes $\widetilde{O}(\frac{m\ell^{3/2}}{\alpha\beta^{7/2}})$ time, which is $\widetilde{O}(\frac{m\ell^{3/2}}{\alpha\beta^{9/2}})$ over the entire execution.

We conclude the result of **smallFracCover** via the following theorem.

Theorem 4.5.9. *For any constant $\varepsilon > 0$,* **smallFracCover** *computes a $(1+\varepsilon)$-approximate solution to* SetCover-LP *with $\widetilde{O}(\frac{m\sqrt{k}}{\varepsilon^{9/2}} + \frac{nk}{\varepsilon^2})$ probes and $\widetilde{O}(\frac{mk^{3/2}}{\varepsilon^{11/2}} + \frac{nk}{\varepsilon^2})$ time with high probability.*

4.5.1.4 Extension to general covering LPs

Similarly to the streaming setting, we remark that our MWU-based algorithm can be extended to solve a more general class of covering LPs. Consider the problem of finding a vector $\mathbf{x}$ that minimizes $\mathbf{c}^\top \mathbf{x}$ subject to constraints $\mathbf{A}\mathbf{x} \geq \mathbf{b}$ and $\mathbf{x} \geq \mathbf{0}$. In terms of the Set Cover problem, $A_{e,S} \geq 0$ indicates the multiplicity of an element e in the set S, $b_e > 0$ denotes the number of times we wish e to be covered, and $c_S > 0$ denotes the cost per unit for the set S. We may modify our model so that the algorithm is readily given full information on vectors $\mathbf{b}$ and $\mathbf{c}$. Moreover, each SetOf and EltOf probe, in addition to returning a set or an element, also returns the multiplicity $A_{e,S} > 0$ corresponding to the relationship $e \in S$ (whereas $A_{e,S} = 0$ is interpreted as $e \notin S$ and will not be returned by the oracle).

Now define

$$L \triangleq \min_{(e,S):A_{e,S}\neq 0} \frac{A_{e,S}}{b_e c_S} \quad \text{and} \quad U \triangleq \max_{(e,S)} \frac{A_{e,S}}{b_e c_S}.$$

Then, we may modify our algorithm to obtain the following result.

Corollary 4.5.10. *For any constant $\varepsilon > 0$, there exist variations of* **smallFracCover** *that compute a $(1+\varepsilon)$-approximate solution to general covering LPs with high probability, each of which providing the following respective asymptotic guarantees:*
(a) $\widetilde{O}(\frac{mkU^2}{\varepsilon^5 L} + \frac{nkU}{\varepsilon^2})$ probes and time,
(b) $\widetilde{O}(\frac{mk^{1/2}U^{3/2}}{\varepsilon^{9/2}L} + \frac{nkU}{\varepsilon^2})$ probes and $\widetilde{O}(\frac{mk^{3/2}U^{5/2}}{\varepsilon^{11/2}L} + \frac{nkU}{\varepsilon^2})$ time,
(c) $\widetilde{O}(\frac{mk^2U^2}{\varepsilon^4} + \frac{nkU}{\varepsilon^2})$ probes and time, and
(d) $\widetilde{O}(\frac{mk^{3/2}U^{3/2}}{\varepsilon^{7/2}} + \frac{nkU}{\varepsilon^2})$ probes and $\widetilde{O}(\frac{mk^{5/2}U^{5/2}}{\varepsilon^{9/2}} + \frac{nkU}{\varepsilon^2})$ time.
Note that for conditions (c)-(d), $\widetilde{O}$ hides a factor of $\log 1/L$.

In our result, conditions (a), (b) are accomplished by modifying the method for covering the common elements, whereas conditions (c), (d) are obtained by applying the MWU framework on

all elements. We provide the latter two results for two reasons. First, they only depend logarithmically on $1/L$; this allows us to reasonably compare our results with other previous approaches such as that of [KY14] which also have no dependence on L. Second, we will soon discuss the extension to (set) packing LPs, where partitioning elements into common elements and rare elements do not reduce the complexities. Conditions (a), (c) are derived from the simpler algorithm **feasibilityTest-IS**, whereas conditions (b), (d) are derived from the adaptive data analysis approach **feasibilityTest-RS**. We now provide a description for all the required modifications to obtain these algorithms.

General formulas for complexities. First, recall that For **feasibilityTest-IS**, the probe and time complexities are both

$$\begin{aligned}&\#\text{ rounds}\\ &\times (\#\text{ elements/round} \times \#\ \textsc{SetOf}\text{ probes/element } + \#\ \textsc{EltOf}\text{ probes/round}),\end{aligned}$$

where "elements" are the elements we need to sample. For **feasibilityTest-RS**, the probe complexity is

$$\begin{aligned}\#\text{ epochs} \times (&\#\text{ points/epoch} \times \#\text{ elements/point } \times \#\ \textsc{SetOf}\text{ probes/element}\\ &+ \#\text{ rounds/epoch} \times \#\ \textsc{EltOf}\text{ probes/round}),\end{aligned}$$

whereas its time complexity is

$$\begin{aligned}\#\text{ rounds} \times (&\#\text{ points/epoch} \times \#\text{ elements/point } \times \#\ \textsc{SetOf}\text{ probes/element}\\ &+ \#\ \textsc{EltOf}\text{ probes/round}),\end{aligned}$$

where "points" are the data points for the mechanism. Note that the first term of the time complexity represents the fact that for each round the number of points on which we need to evaluate f is (# points/epoch).

Normalizing b and c. First, observe that we can convert the input LP into an equivalent LP with all entries b_e and c_S to 1 by simply replacing each $A_{e,S}$ with $\frac{A_{e,S}}{b_e c_S}$, which is in the range $\{0\} \cup [L, U]$. This operation can be done by the algorithm; namely, since it knows $\mathbf{b}$ and $\mathbf{c}$, it can replace each value $A_{e,S}$ on-the-fly when this value is returned by the SetOf or EltOf oracles.

Handling the rare elements. Next consider the process for covering the rare elements. If we do not cover the rare elements separately but instead cover them during our MWU algorithm, then for each sample, we may need to make as many as m SetOf probes for sets containing it. That is, (# SetOf probes/element) is bounded by m.

Now suppose that we instead use a uniform solution $\mathbf{x}^{\mathsf{cmn}} = \frac{\alpha\ell L}{m} \cdot \mathbf{1}$. Observe that if an element occurs in at least $\frac{m}{\alpha\ell L}$ sets, then

$$\mathbf{A}_e \mathbf{x}^{\mathsf{cmn}} = \sum_{S: e \in S} A_{e,S} \frac{\alpha\ell}{m} \geq \frac{m}{\alpha\ell L} \cdot L \cdot \frac{\alpha\ell}{m} = 1.$$

In other words, we must adjust our definition so that an element is considered rare if it appears in at least $\frac{m}{\alpha\ell L}$ sets. Consequently, we must set (# SETOF probes/element) to $\frac{m}{\alpha\ell L}$.

Modifying the number of rounds. Consider now the MWU algorithm. In each iteration, we again create an approximate oracle that picks a set S of probability $p_S \geq \frac{1-\beta/3}{\ell}$, then choose a solution $\mathbf{x}$ with $x_S = \frac{1-\beta/3}{p_S} \leq \ell$. Consequently, $\mathbf{A}_e\mathbf{x} - b_e = A_{e,S} \cdot \frac{1-\beta/3}{p_S} - 1 \in [-1, \ell U - 1]$. Thus, the width of our algorithm is now $\rho = \ell U$, and we need to increase the number of rounds by a factor of U to (# rounds) $= T = \Theta(\frac{\rho \log n}{\beta^2}) = \widetilde{\Theta}(\frac{\ell U}{\beta^2})$.

For **feasibilityTest-RS**, we note that as ρ increase, the (additive) change in probabilities between rounds also decreases by a factor of U. That is, the probability p_e of any element increases only by a factor of $1 + \frac{\beta}{\rho}$; we now have (# rounds/epoch) $= K = \frac{\rho}{\beta} = \frac{\ell U}{\beta}$ rounds, but (# epochs) $= T' = \Theta(\frac{\log n}{\beta})$ remains unchanged. Due to the increased number of rounds per epoch, we must also increase the number of data points given to the mechanism: (# points/epoch) $= N = \widetilde{\Theta}(\sqrt{K}) = \widetilde{\Theta}(\sqrt{kU/\beta})$.

Fixing the sampling process. We generalize the probability of a set to $p_S \triangleq \sum_{e \in S} p_e A_{e,S}$ so that $p_S = (\mathbf{p}^\top \mathbf{A})_S$ still holds. To obtain the desired guarantee, for each sample e, we probe for every S containing e, but only add $A_{e,S}/U$ to our unbiased estimator for p_S. To apply the Chernoff bound, we need U times the original number of samples; as a result, the number of samples in each round of **feasibilityTest-IS** must be increased to (# points/epoch) $= \widetilde{\Theta}(\frac{\ell U}{\beta^2})$.

For **feasibilityTest-RS**, this means each data point must contain (# points/epoch) $= s = \Theta(\frac{\ell U}{\beta^2})$ samples. We note that in the analysis, the unbiased estimator becomes

$$q^{\mathbf{p}^{\mathsf{curr}}, \mathbf{p}^{\mathsf{mech}}}(\mathbf{d}; S) \triangleq \frac{1}{s} \sum_{i=1}^{s} \mathbf{1}_S(\hat{e}_i) \frac{p^{\mathsf{curr}}_{\hat{e}_i}}{p^{\mathsf{mech}}_{\hat{e}_i}} A_{\hat{e}_i,S},$$

and the variable Y_i for applying the Chernoff bound must be modified to

$$Y_i = \frac{A_{\hat{e}_i,S}}{\mathsf{e}U} \mathbf{1}_S(\hat{e}_i) \frac{p^{\mathsf{curr}}_{\hat{e}_i}}{p^{\mathsf{mech}}_{\hat{e}_i}}$$

to ensure that $Y_i \in [0, 1]$.

Finally, once we plug each term into our complexity formulas for each algorithm, Corollary 4.5.10 follows from the fact that we only run our algorithm for solving the feasibility problem with guesses $\ell = O(k)$, and up to $O(\log(nU/L) + \log(1/\varepsilon))$ iterations are required in total.

4.5.1.5 Extension to set packing LPs

In this section, we consider the dual problem of Fractional Set Cover called Fractional Set Packing, and outline how we may solve this problem using a similar approach. Consider the same set system $(\mathcal{U}, \mathcal{F})$. We aim to compute a vector $\mathbf{y} \in \mathbb{R}^n$ where each value y_e is the *weight* of an element e, under the packing constraints $\sum_{e \in S} y_e \leq 1$ restricting the total weight of elements of any set to at most 1, and the non-negativity constraints $y_e \geq 0$. The goal is to maximize the total weight $\sum_{e \in \mathcal{U}} y_e$. This problem is restated as SetPacking-LP in Figure 4-14a.

We may again describe Fractional Set Packing in its matrix form: maximize $\mathbf{b}^\top \mathbf{y}$ subject to

$\mathbf{A}^\top \mathbf{y} \le \mathbf{c}$ and $\mathbf{y} \ge \mathbf{0}$. This matrix form represents a more general packing problem, but for simplicity we only restrict our attention to the case where every entry $A_{e,S} \in \{0,1\}$ and $b_e, c_S = 1$; its generalization may be performed in the same fashion as accomplished in Section 4.5.1.4. Due to the nature of the Set Packing problem, we do not have a method for dealing with common elements (or large sets) in the same fashion we did for Set Cover; hence, we must solve the whole system via the MWU framework.

SetPacking-LP $\langle\langle$*Input:* $\mathcal{U}, \mathcal{F}\rangle\rangle$

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in \mathcal{U}} y_e & \\ \text{subject to} \quad & \sum_{e \in S} y_e \le 1 & \forall S \in \mathcal{F} \\ & y_e \ge 0 & \forall e \in \mathcal{U} \end{aligned}$$

(a) LP relaxation of the Set Packing problem.

Feasibility-Packing-LP $\langle\langle$*Input:* $\mathbf{A}_{n\times m}, \mathbf{b}, \mathbf{c}, \ell\rangle\rangle$

$$\begin{aligned} \mathbf{b}^\top \mathbf{y} &\ge \ell & \text{(objective value)} \\ \mathbf{A}^\top \mathbf{y} &\le \mathbf{c} & \text{(packing)} \\ \mathbf{y} &\ge \mathbf{0} & \text{(non-negativity)} \end{aligned}$$

(b) LP relaxation of the Feasibility Packing problem.

Figure 4-14: LP relaxations of the set packing problem, and the feasibility variant for general packing problems.

Modifying feasibilityTest-IS for packing. To apply the MWU approach, we again consider its feasibility variant, Feasibility-Packing-LP in Figure 4-14b. In this case, we define our polytope $\mathcal{P}_\ell \triangleq \{\mathbf{y} \in \mathbb{R}^n : \mathbf{y} \ge \mathbf{0} \text{ and } \mathbf{b}^\top \mathbf{y} \ge \ell\}$, and aim to solve $\mathbf{q}^\top \mathbf{A}^\top \mathbf{y} \le \mathbf{q}^\top \mathbf{c} = 1$ in each oracle call, where $\mathbf{q}$ is the probability vector proportional to the constraints' weights (which now corresponds to the sets, not elements). More specifically, we must find some $\mathbf{y} \in \mathcal{P}_\ell$ satisfying $\mathbf{q}^\top \mathbf{A}^\top \mathbf{y} \le 1 + \beta$ whenever $\mathbf{q}^\top \mathbf{A}^\top \mathbf{y} \le 1$ has a solution in $\mathcal{P}_\ell$.

Then, the oracle needs to find an element e such that $q_e \triangleq \sum_{S: e \in S} q_S = (\mathbf{q}^\top \mathbf{A}^\top)_e \le \frac{1+\beta}{\ell}$ so that we may construct a solution $\mathbf{y}$ where $y_e = \ell$ and $y_{e'} = 0$ for all other elements e'. (In contrast to covering, we look for an element with low q_e for packing.) To approximate q_e, we need to sample $\widetilde{\Theta}(\ell/\beta^2)$ sets to compute q_e with the desired accuracy; each set requires $O(n)$ EltOf probes for checking its elements. Once an element is chosen, $O(m)$ SetOf probes are required to compute q_e and update the probabilities q_S for the next round. It is straightforward to verify that the width of the algorithm is still $\rho = \ell$. Then, using the same analysis, we obtain an algorithm similar to **feasibilityTest-IS** that finds an approximation to SetPacking-LP using

$$\begin{aligned} &\#\text{ rounds} \times (\#\text{ sets/round} \times \#\text{ EltOf probes/set} \; + \; \#\text{ SetOf probes/round}) \\ &= \widetilde{O}\left(\frac{\ell}{\beta^2}\right)\left(\widetilde{\Theta}\left(\frac{\ell}{\beta^2}\right) \cdot n + m\right) = \widetilde{O}\left(\frac{n\ell^2}{\beta^4} + \frac{m\ell}{\beta^2}\right) \end{aligned}$$

probes and running time per iteration of feasibility checking (corresponding to each guess ℓ), resulting in $\widetilde{O}(\frac{nk^2}{\varepsilon^4} + \frac{mk}{\varepsilon^2})$ overall complexities. Unlike **feasibilityTest-IS**, the dependencies on m and n are swapped since our linear constraints now correspond to $\mathbf{A}^\top$, and the first term is multiplied by a factor of εk due to the fact that we solve the whole system with the MWU approach.

Modifying feasibilityTest-RS for packing. For **feasibilityTest-RS**, the required modification

is very similar to the case of **feasibilityTest-IS**. It is also trivial to verify that the probability q_S does not increase by more than a factor of $(1 + \frac{\beta}{\ell})$, and therefore the number of rounds per epoch and the number of sets in a data point need not be changed. This yields an algorithm with $\widetilde{O}(\frac{nk^{3/2}}{\varepsilon^{7/2}} + \frac{mk}{\varepsilon^2})$ probes and $\widetilde{O}(\frac{nk^{5/2}}{\varepsilon^{9/2}} + \frac{mk}{\varepsilon^2})$ time.

We conclude our results for Fractional Set Packing below.

Corollary 4.5.11. *For any constant $\varepsilon > 0$, there exist variations of* **smallFracCover** *that compute a $(1+\varepsilon)$-approximate solution to set packing LPs with high probability, each of which providing the following respective asymptotic guarantees:*
(a) $\widetilde{O}(\frac{nk^2}{\varepsilon^4} + \frac{mk}{\varepsilon^2})$ probes and time, by modifying **feasibilityTest-IS**, *and*
(b) $\widetilde{O}(\frac{nk^{3/2}}{\varepsilon^{7/2}} + \frac{mk}{\varepsilon^2})$ probes and $\widetilde{O}(\frac{nk^{5/2}}{\varepsilon^{9/2}} + \frac{mk}{\varepsilon^2})$ time, by modifying **feasibilityTest-RS**.

4.5.2 Efficient algorithm for instances with large optimal value

Note that MWU-based approach performs well when k is small. In this section, we complement our result by giving a simple algorithm whose probe complexity becomes better as k increases. As described earlier we pick the vector $\mathbf{x}^{\mathsf{cmn}}$ whose all entries are $\frac{\alpha\ell}{m}$. The total cost of the vector is $\alpha\ell$ and the elements that are not covered by $\mathbf{x}^{\mathsf{cmn}}$ appear in at most $\frac{m}{\alpha\ell}$ sets. By probeing the sets containing all uncovered elements, we can fully construct a reduced instance of SetCover-LP over uncovered elements using $O(\frac{mn}{\alpha\ell})$ probes. Moreover, we can apply the following result of Koufogiannakis and Young [KY14] to upper bound the required running time for constructing a $(1+\beta)$-approximate solution with only an additive overhead of $\widetilde{O}(\frac{m+n}{\beta^2})$ to the time complexity.

Theorem 4.5.12 (Theorem 3 of [KY14])**.** *For mixed/packing covering, there is a $(1+\beta)$-approximation algorithms running in time $O(\frac{(n+m)\log \mathsf{NZ}}{\beta^2} + \mathsf{NZ})$ where NZ denotes the number of non-zero entries in the linear program.*

Since the size of the reduced SetCover-LP instance is $O(\frac{mn}{\alpha\ell})$, by Theorem 4.5.12, a $(1+\beta)$-approximate solution can be computed in time $\widetilde{O}(\frac{m+n}{\beta^2} + \frac{mn}{\alpha\ell})$. Using this result, we give our implementation **largeFracCover** in Figure 4-15 and analyze its correctness and complexities below.

Theorem 4.5.13. *For any constant $\varepsilon > 0$,* **largeFracCover** *computes a $(1+\varepsilon)$-approximate solution to* SetCover-LP *with $\widetilde{O}(\frac{mn}{\varepsilon k})$ probes and $\widetilde{O}(\frac{m+n}{\varepsilon^2} + \frac{mn}{\varepsilon k})$ time with high probability.*

Proof. To detect the rare elements, at most one SetOf probe per element is required; in total, n probes are sufficient for this task. Then **largeFracCover** probes the sets containing rare elements (at most $\frac{m}{\alpha\ell}$ SetOf probes per rare element); the total number of probes made by the algorithm is $O(\frac{mn}{\alpha\ell})$. As **feasibilityTest-Large** verifies that the value of the solution returned by the SetCover-LP solver is at most $(1+\beta)\ell$, the algorithm will always return a feasible solution with objective value at most $(\alpha+(1+\beta))\ell \leq (1+\varepsilon/3)\ell$ (recall that **feasibilityTest-Large** is invoked by **largeFracCover** with parameter $\varepsilon/3$). Via the search in **largeFracCover**, we obtain an approximation ℓ satisfying $k \leq \ell \leq (1+\varepsilon/3)k$. Since we only invoke **feasibilityTest-Large** with $\ell = \Omega(k)$, the total number of probes is bounded by $O(\log n + \log 1/\varepsilon)$ times the complexity of each iteration of **feasibilityTest-Large**, yielding the overall probe complexity of $\widetilde{O}(\frac{mn}{\varepsilon k})$.

largeFracCover(ε):

▷ Find a feasible 2-approximate solution in $O(\log n)$ iterations

for $\ell \in \{2^i \mid 0 \le i \le \log n\}$ **do in the decreasing order**:

  try $\mathbf{x} \leftarrow$ **feasibilityTest-Large**$(\ell, \varepsilon/3)$

  if feasibilityTest-Large$(\ell, \varepsilon/3)$ reports INFEASIBLE **then break**

  else $\mathbf{x}^{\mathsf{best}} \leftarrow \mathbf{x}$ and **continue**

$L \leftarrow \ell, H \leftarrow 2\ell$

▷ Binary search for a $(1+\varepsilon/3)$-approximation using $\log 1/\varepsilon$ more iterations

repeat $\Theta(\log 1/\varepsilon)$ iterations

  try $\mathbf{x} \leftarrow$ **feasibilityTest-Large**$(\frac{L+H}{2}, \varepsilon/3)$

  if feasibilityTest-Large$(\frac{L+H}{2}, \varepsilon/3)$ reports INFEASIBLE **then** $L \leftarrow \frac{L+H}{2}$

  else $\mathbf{x}^{\mathsf{best}} \leftarrow \mathbf{x}$, $H \leftarrow \frac{L+H}{2}$

return $\mathbf{x}^{\mathsf{best}}$

feasibilityTest-Large(ℓ, ε):

$\alpha, \beta \leftarrow \frac{\varepsilon}{2}$ ▷ Approximation parameters for covering common and rare elements

$\mathbf{x}^{\mathsf{cmn}} \leftarrow \frac{\alpha\ell}{m} \cdot \mathbf{1}_{m\times 1}$ ▷ $\mathbf{x}^{\mathsf{cmn}}$ is the solution covering all common elements

$\mathcal{U}^{\mathsf{rare}} \leftarrow \emptyset$ ▷ Compute the rare elements, which are not covered by $\mathbf{x}^{\mathsf{cmn}}$

for each $e \in \mathcal{U}$ **do**

  if e appears in less than $\frac{m}{\alpha\ell}$ sets **then** ▷ Single SETOF probe

    let $\mathcal{F}_e$ be the collection of sets containing e ▷ $O(\frac{m}{\alpha\ell})$ SETOF probes

    $\mathcal{U}^{\mathsf{rare}} \leftarrow \mathcal{U}^{\mathsf{rare}} \cup \{e\}, \quad \mathcal{F}^{\mathsf{rare}} \leftarrow \mathcal{F}^{\mathsf{rare}} \cup \mathcal{F}_e$

$\mathbf{x}^{\mathsf{rare}} \leftarrow (1+\beta)$-approximate solution of SetCover-LP $(\mathcal{U}^{\mathsf{rare}}, \mathcal{F}^{\mathsf{rare}})$ returned by [KY14]

if $\|\mathbf{x}^{\mathsf{rare}}\| \le (1+\beta)\ell$ **then return** $\mathbf{x}^{\mathsf{cmn}} + \mathbf{x}^{\mathsf{rare}}$

else report INFEASIBLE

Figure 4-15: **largeFracCover** returns a $(1+\varepsilon)$-approximate solution of SetCover-LP, where **feasibilityTest-Large** is an algorithm that returns a solution of objective value at most $(1+\varepsilon/3)\ell$ when $\ell \ge k$.

By Theorem 4.5.12, there exists a $(1+\beta)$-approximate SetCover-LP solver that runs in $O(\frac{(n+m)\log \mathsf{NZ}}{\beta^2} + \mathsf{NZ})$; thus, a $(1+\beta)$-approximate solution of the reduced SetCover-LP instance can be computed in $\widetilde{O}(\frac{m+n}{\beta^2} + \frac{mn}{\alpha k})$. Therefore the total runtime of **largeFracCover** is $\widetilde{O}(\frac{m+n}{\varepsilon^2} + \frac{mn}{\varepsilon k})$. □

We again remark that **largeFracCover** may handle general covering LPs by setting $\mathbf{x}^{\mathsf{cmn}} = \frac{\alpha\ell L}{m} \cdot \mathbf{1}$, which increases the number of probes per rare element to $O(\frac{m}{\alpha\ell L})$ SETOF probes. Consequently NZ may be as large as $\frac{mn}{\alpha\ell L}$. This yields the following corollary.

Corollary 4.5.14. *For any constant $\varepsilon > 0$, there exist a variation of* **largeFracCover** *that computes a $(1+\varepsilon)$-approximate solution to general packing LPs with $\widetilde{O}(\frac{mn\log U}{\alpha k L})$ probes and $\widetilde{O}(\frac{(m+n)\log U}{\varepsilon^2} + \frac{mn\log U}{\varepsilon k L})$ time with high probability.*

4.6 Lower Bounds for the Fractional Set Cover Problem

In this section, we present our lower bounds for solving Set Cover LPs when $\varepsilon \le 1$. The analysis for our lower bound proofs are sufficiently similar to that of in the (integral) Set Cover problem

in Section 3.3, so we focus on describing the constructions and bounding their optimal objective values.

4.6.1 Construction of basic blocks

Let $q \geq 3$ be a parameter. Consider a set system (U, F) of size $|\mathsf{U}| = |\mathsf{F}| = 2q$. We partition $\mathsf{U} = \{e_1, \cdots, e_{2q}\}$ into two disjoint equal-sized sets $\mathsf{U}^{\mathsf{left}} = \{e_1, \ldots, e_q\}$ and $\mathsf{U}^{\mathsf{right}} = \{e_{q+1}, \ldots, e_{2q}\}$. The collection $\mathcal{F}$ consists of $2q$ sets $S_1, \cdots, S_{2q}$ such that

$$S_r = \begin{cases} \mathsf{U} \setminus \{e_r, e_{r+q}\}, & r \leq q \\ \mathsf{U}^{\mathsf{left}} \setminus \{e_{r-q}\}, & r > q \end{cases}$$

The set system (U, F) is called a *basic block*. An example basic block with $q = 4$ is given in Figure 4-16a. It is straightforward to check that each set in $\{S_1, \cdots, S_q\}$ has size exactly $2(q-1)$, each set in $\{S_{q+1}, \cdots, S_{2q}\}$ has size $q-1$, each element $e \in \mathsf{U}^{\mathsf{left}}$ appears in exactly $2(q-1)$ sets, and each element $e \in \mathsf{U}^{\mathsf{right}}$ appears in exactly $q-1$ sets.

Next we define *swapped blocks*. Consider integers $1 \leq i, j \leq q$ such $i \neq j$. In a basic block (U, F), observe that $e_i \in S_{q+j}$ but $e_i \notin S_i$, while $e_{q+j} \in S_i$ but $e_{q+j} \notin S_{q+j}$. The swapped block $(\mathsf{U}, \mathsf{F}^{(i,j)})$ is constructed by applying the swap operation between $e_{q+j} \in S_i$ and $e_i \in S_{q+j}$ on the basic block. More formally, the set system of the swapped block $(\mathsf{U}, \mathsf{F}^{(i,j)} = \{S_1^{(i,j)}, \cdots, S_n^{(i,j)}\})$ is defined as follows:

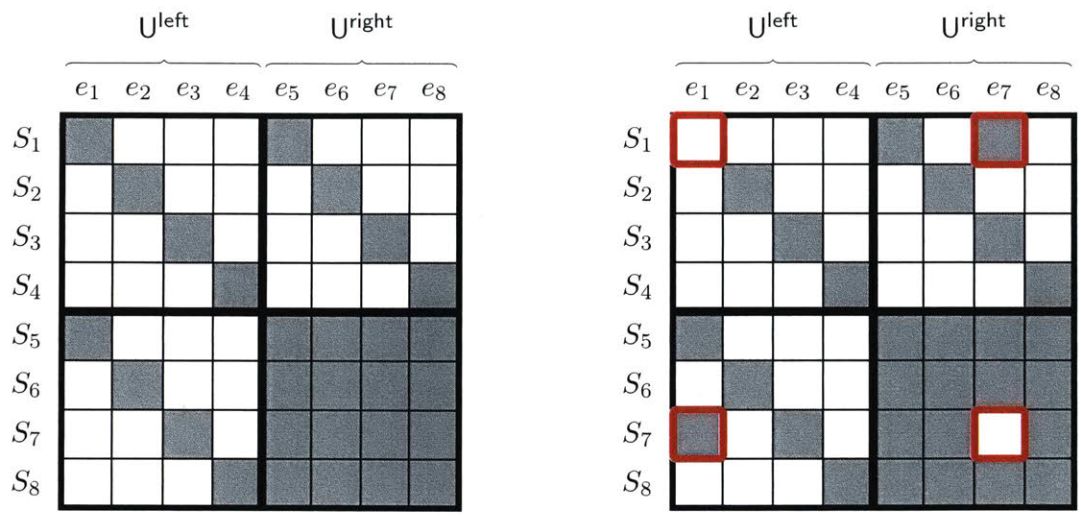
$$S_r^{(i,j)} = \begin{cases} S_r, & r \notin \{i, q+j\} \\ (S_r \cup \{e_i\}) \setminus \{e_{q+j}\}, & r = i \\ (S_r \cup \{e_{q+j}\}) \setminus \{e_i\}, & r = q+j \end{cases}.$$

An example swapped block $(\mathsf{U}, \mathsf{F}^{(1,3)})$ for $q = 4$ is given in Figure 4-16b. Observe that all sets other than S_i and S_{q+j} remain unchanged, while S_i, S_{q+j} only swap one pair of elements e_{q+j}, e_i. Moreover, for each pair (i, j) such that $i \neq j$ we can construct a distinct swapped block; this pair of parameters fully specifies all possible swapped blocks resulting from a basic block. Following the same argument from Section 3.3 for Cover Verification, each swapped block only differs in exactly two answers of each SetOf and EltOf oracles from those of its corresponding basic block, and no two distinct swapped blocks modify any same oracle answer.

4.6.2 Main construction

In this section, using the basic blocks and the swap operations defined in the previous section (see Figure 4-16), we construct two families of instances for the lower bound argument. The underlying instance consists of $\frac{m}{2q} \times \frac{n}{2q}$ basic blocks, each of size $2q \times 2q$. Then we perform a series of swap operations on some of the blocks in the structure to obtain our families of Yes instances and No instances.

First consider an $m \times n$ matrix which consists of $\frac{m}{2q} \times \frac{n}{2q}$ basic blocks. More precisely, the set



(a) A basic block (U, F) **(b)** A swapped block $(\mathsf{U}, \mathsf{F}^{(1,3)})$

Figure 4-16: A basic block (U, F) and an example of a swap operation for $q = 4$. Recall that the cell corresponding to the pair (e, S) is white if $e \in S$ and is shaded if $e \notin S$.

system $(\mathcal{U} = \cup_{c=0}^{\frac{n}{2q}-1} \mathsf{U}_c, \mathcal{F} = \cup_{r=0}^{\frac{m}{2q}-1} \mathsf{F}_r)$ is constructed so that each of $(\mathsf{U}_c = \{e_{2cq+1,\ldots,(2c+2)q}\}, \mathsf{F}_r = \{S_{2rq+1}, \ldots, S_{(2r+2)q}\})$ where $0 \leq r < \frac{m}{2q}$ and $0 \leq c < \frac{n}{2q}$ forms a basic block of size $2q \times 2q$, as shown in Figure 4-17.

Next we define the No family and the Yes family of instances with respect to this underlying structure.

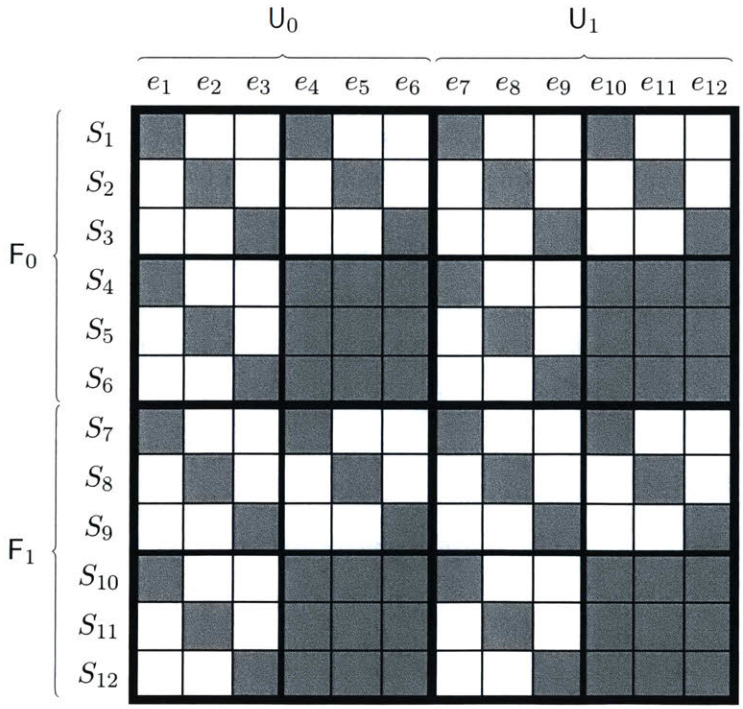


Figure 4-17: An underlying set system of basic blocks, with $q = 3$ and $m = n = 2 \cdot 2q$.

No instances. Pick a uniformly random column (of blocks) $0 \leq c < \frac{n}{2q}$ and a uniformly random pair (i, j) (such that $1 \leq i, j \leq q$ and $i \neq j$). Then, replace every basic block of column c by a swapped block with parameter (i, j). More formally, for each $0 \leq r < \frac{m}{2q}$, we replace the basic block $(\mathsf{U}_c, \mathsf{F}_r)$ with the swapped block $(\mathsf{U}_c, \mathsf{F}_r^{(i,j)})$. Note that the indices i, j are with respect to the elements and sets in the corresponding basic block. That is, the swap occurs between $e_{(2c+1)q+j} \in S_{2rq+i}$ and $e_{2cq+i} \in S_{(2r+1)q+j}$. An example can be found in Figure 4-18a.

Yes instances. For each No instance with parameter (c, i, j), create a Yes instance by undoing one of the swaps performed on a uniform random row r. In other words, we can specify each Yes instance with parameters (r, c, i, j) which indicate that all basic blocks in column c except the one in row r are replaced with the swapped block defined with parameters i and j. An example of this construction can be found in Figure 4-18b.

4.6.3 Bounding the optimal objective values

We now establish useful bounds on the optimal objective value for the instances in our constructed families. We remark that these bounds are given in terms of q; they are independent of m and n.

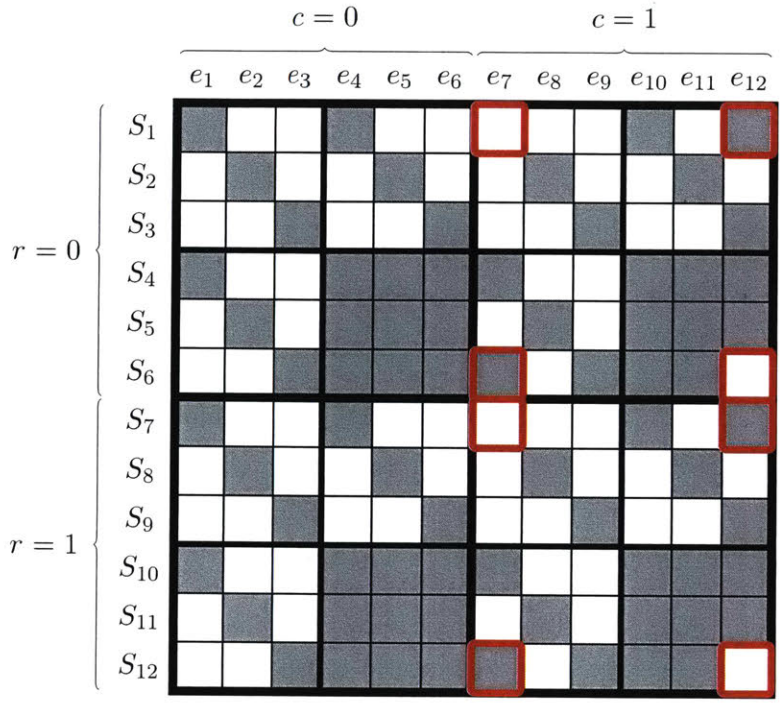
Lemma 4.6.1. *The optimal value of the* Yes *instance with block size* $2q$ *is at most* $\frac{q}{q-1}$.

Proof. For all Yes instances, there exists a row consisted entirely of basic (not swapped) blocks. Let us denote the index of this row by r. Observe that every element belongs to all but one set from the collection $\{S_{2rq+i}, \ldots, S_{(2r+1)q}\}$. Thus, the fractional set cover $\mathbf{x}$, such that $x_{S_{2rq+i}} = \frac{1}{q-1}$ for $1 \leq i \leq q$ and 0 for all remaining sets, covers this Yes instance. (As an example, consider setting $x_{S_1} = x_{S_2} = x_{S_3} = \frac{1}{2}$ in Figure 4-18b.) This solution $\mathbf{x}$ has objective value $\frac{q}{q-1}$, which upper-bounds the optimal solution. □

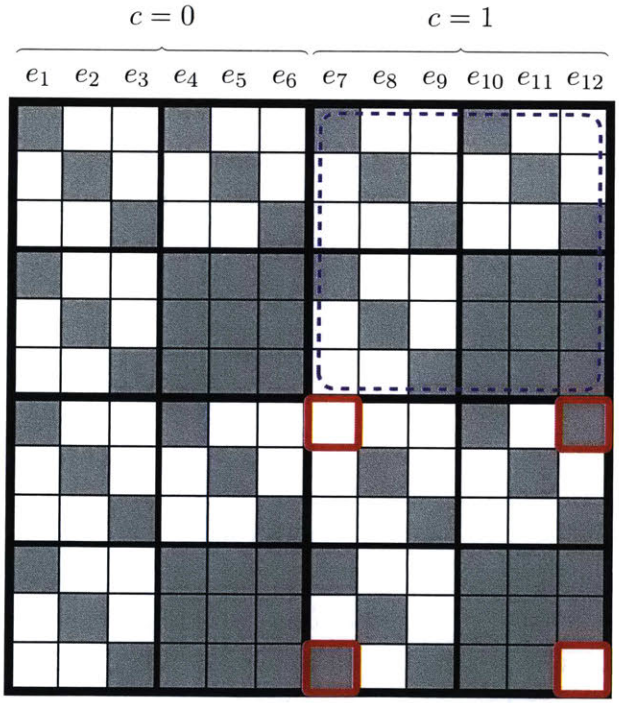
Lemma 4.6.2. *The optimal value of the* No *instance with block size* $2q$ *is at least* $\frac{q-1}{q-2}$.

Proof. To prove a lower bound on the optimal value for the fractional set cover of any No instance, we construct a feasible solution of the dual program of its SetCover-LP with value $\frac{q-1}{q-2}$. Then, by weak duality we conclude that the optimal objective value for the SetCover-LP of the No instance is at least $\frac{q-1}{q-2}$. The dual problem, given in Figure 4-19, is a packing problem: the objective is to assign a value y_e to each element e so that the total value of elements within any set is at most 1, while maximizing the total value assigned to all elements.

By the construction of No instances, there exists a column c where all blocks undergo the same swap specified by (i, j). Let $\mathsf{U}_c^{\mathsf{left}} = \{e_{2cq+1}, \cdots, e_{(2c+1)q}\}$ and $\mathsf{U}_c^{\mathsf{right}} = \{e_{(2c+1)q+1}, \cdots, e_{(2c+2)q}\}$ (analogously to the partition within a basic block). Set the value $y_e = \frac{1}{q-2}$ for each element $e \in \mathsf{U}_c^{\mathsf{right}} \setminus \{e_{(2c+1)q+j}\}$, and $y_e = 0$ for all remaining elements. (As an example, consider setting $y_{e_{11}} = y_{e_{12}} = 1$ in Figure 4-18a.) By our construction, every set contains at most $q - 2$ elements of $\mathsf{U}_c^{\mathsf{right}} \setminus \{e_{2cq+q+j}\}$; thus, all packing constraints of the dual program over the No instance are satisfied. Moreover, the total value of the constructed solution $\mathbf{y}$ is $\frac{q-1}{q-2}$. □



(a) A No instance with parameter $(1, 1, 3)$



(b) A Yes instance with parameter $(0, 1, 1, 3)$

Figure 4-18: A No instance and a Yes instance from the constructed families.

Dual of SetCover-LP $\langle\langle$*Input:* $\mathcal{U}, \mathcal{F}\rangle\rangle$

$$\begin{aligned} \text{maximize} \quad & \sum_{e \in \mathcal{U}} y_e & \\ \text{subject to} \quad & \sum_{e \in S} y_e \leq 1 & \forall S \in \mathcal{F} \\ & y_e \geq 0 & \forall e \in \mathcal{U} \end{aligned}$$

Figure 4-19: LP relaxation of the dual of Set Cover problem.

4.6.4 Establishing lower bounds

To prove the lower bound for randomized protocols, applying Yao's principle, we instead show a lower bound for any deterministic algorithm on a fixed distribution of input instances. We define the distribution $\mathcal{D}$ of instances as follows: each of $\frac{mn(q-1)}{q}$ possible Yes instances has probability $\frac{q}{2mn(q-1)}$, and each of the possible $n(q-1)$ No instances has probability $\frac{1}{2n(q-1)}$. We may also equivalently describe the process for generating a random instance as follows. First, pick a tuple (c, i, j) uniformly at random, and create a No instance according to this parameter. Then, with probability $1/2$, we pick a random row r and undo the swap performed on the block located at column c and row r to obtain a Yes instance corresponding to the parameter (r, c, i, j).

As similarly argued in Section 3.3, we may characterize each probe as a unique parameter (r, c, i, j) if it instigates different answers from the basic block $(\mathsf{U}_c, \mathsf{F}_r)$ and the swapped block $(\mathsf{U}_c, \mathsf{F}_r^{(i,j)})$. Observe that the algorithm may only detect a swap if it correctly specifies the triplet (c, i, j), and may only detect the difference between the Yes instance and the corresponding No instance if it correctly specifies the 4-tuple (r, c, i, j).

First, we show that any deterministic $(1 + O(\frac{1}{q^2}))$-approximation algorithm of SetCover-LP over this distribution requires $\Omega(nq)$ probes. Then we provide another lower bound showing that any deterministic $(1 + O(\frac{1}{q^2}))$-approximation algorithm of the problem over this same distribution also requires $\Omega(\frac{m}{q})$ probes. Because these lower bounds are established from same distribution, we achieve the combined lower bound of $\Omega(nq + \frac{m}{q})$ probes.

$\Omega(nq)$ lower bound. Here we provide the algorithm with a stronger oracle. Once the algorithm makes a probe corresponding to (c, i, j), it receives the values of all EltOf and SetOf probes corresponding to (r, c, i, j) for all values of $0 \leq r < \frac{m}{q}$.

Lemma 4.6.3. *If a deterministic algorithm makes at most $\frac{nq}{5}$ probes on the input from distribution $\mathcal{D}$, then its probability of success is less than $2/3$.*

Proof. Since the algorithm is deterministic, as long as it detects no swap, its probe-answer history must be the same no matter whether the input is from the No family or the Yes family. Thus the algorithm must return the same answer. Suppose that the algorithm makes less than $\frac{nq}{5}$ probes. Then for all instances in $\mathcal{D}$ whose swaps are specified with the $n(q-1) - \frac{nq}{5} \geq \frac{7}{10}n(q-1)$ triplets (c, i, j) that are not probed (which constitute at least $\frac{7}{10}$ fraction of $\mathcal{D}$), the algorithm must not see any swap and return the same answer. Moreover, by the described process for constructing

instances of $\mathcal{D}$, it is straightforward to see that in expectation, exactly half of these instances are Yes instances, and the remaining half are No instances. Since the algorithm returns the same answer for this $\frac{7}{10}$ fraction of $\mathcal{D}$, its answer is incorrect for $\frac{7}{20}$ fraction of $\mathcal{D}$. Thus the success probability of the algorithm is at most $1 - 7/20 < 2/3$. □

Lemma 4.6.4. *For $\varepsilon \leq 1$, any (randomized) $(1+\varepsilon)$-approximation algorithm of* SetCover-LP *with probability of success at least $2/3$ requires $\Omega(\frac{n}{\sqrt{\varepsilon}})$ probes.*

Proof. By Lemma 4.6.1, the value of an optimal solution of SetCover-LP over Yes instances is at most $\frac{q}{q-1}$, whereas by Lemma 4.6.2, the value of an optimal solution of SetCover-LP over No instances is at least $\frac{q-1}{q-2}$. Moreover, Lemma 4.6.3 shows that in order to distinguish between Yes and No instances with success probability at least $2/3$, $\Omega(nq)$ probes are required. Hence, setting $q = \frac{3}{\sqrt{\varepsilon}}$ and applying Yao's principle, completes our proof. □

$\Omega(\frac{m}{q})$ lower bound. Similarly, we provide the algorithm with a different type of stronger oracle. Once the algorithm makes a probe corresponding to r, it receives the values of all EltOf and SetOf probes corresponding to (r, c, i, j) for all values of $1 \leq i, j \leq q, i \neq j$ and $0 \leq c < \frac{n}{q}$.

Lemma 4.6.5. *If a deterministic algorithm makes less than $\frac{m}{5q}$ probes on the input from distribution $\mathcal{D}$, then its probability of success is less than $2/3$.*

Proof. Suppose that the Yes instance is specified by the parameter (r, c, i, j). If the algorithm does not make any probe corresponding to the row r, the answers it receives from the oracle will be the same: no swaps anywhere except for those corresponding to (c, i, j). Since the algorithm is deterministic, it must return the same answer. Suppose that the algorithm makes less than $\frac{m}{5q}$ probes. Then for all instances in $\mathcal{D}$ whose swaps are specified with the $\frac{m}{q} - \frac{m}{5q}$ values r that are not probed (which constitutes $\frac{4}{5}$ fraction of $\mathcal{D}$), the algorithm must return the same value no matter whether they are Yes instances or No instances. Thus, it must answer incorrectly for $\frac{2}{5}$ fraction of $\mathcal{D}$, yielding the success probability of at most $1 - \frac{2}{5} < \frac{2}{3}$. □

We omit the proof of the next lemma since it closely resembles that of Lemma 4.6.4 but instead makes use of Lemma 4.6.5.

Lemma 4.6.6. *For $\varepsilon \leq 1$, any (randomized) $(1+\varepsilon)$-approximation algorithm of* SetCover-LP *with probability of success at least $2/3$ requires $\Omega(m\sqrt{\varepsilon})$ probes.*

Combining Lemma 4.6.4 and Lemma 4.6.6, we obtain the following lower bound for probe complexity of approximating SetCover-LP. For constant ϵ and k, this result shows that our algorithm **smallSetCover** is tight (up to polylogarithmic factors).

Theorem 4.6.7. *For $\varepsilon \leq 1$, any (randomized) $(1+\varepsilon)$-approximation algorithm of* SetCover-LP *with probability of success at least $2/3$ requires $\Omega(\frac{n}{\sqrt{\varepsilon}} + m\sqrt{\varepsilon})$ probes.*

We remark that we may again apply the reduction from Section 3.5.2, by constructing compounds of independent SetCover-LP instances and thereby increasing the optimal value of the LP. This yields the following result.

Corollary 4.6.8. *For sufficiently small positive constant ε, any (randomized) $(1+\varepsilon)$-approximation algorithm of* SetCover-LP *with probability of success at least $2/3$ requires $\Omega(\frac{n+m}{k})$ probes, where k is the size of the optimal objective value of* SetCover-LP.

Chapter 5

Local-Access Generators for Random Graphs

5.1 Overview

The problem of computing local information of huge random objects was pioneered in [GGN03, GGN10]. Further work of [NN07] considers the generation of sparse random $G(n,p)$ graphs from the Erdős-Rényi model [ER60], with $p = O(\text{poly}(\log n)/n)$, which answers $\text{poly}(\log n)$ ALL-NEIGHBORS probes, listing the neighbors of probed 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 [ELMR17], the authors construct an oracle for the generation of recursive trees, and BA preferential attachment graphs. Unlike [NN07], their implementation allows for an arbitrary number of probes. 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 probes in polylogarithmic time. Though preferential attachment graphs are sparse, they contain vertices of high degree, thus [ELMR17] provides access to the adjacency list through NEXT-NEIGHBOR probes.

In this work, we construct oracles that allow probes 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 probes[1] 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 probes using $\mathcal{O}(\text{poly}(\log n))$ time, space, and random bits per probe (with high probability), 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 (while introducing linear dependencies on the

[1] VERTEX-PAIR(u,v) returns whether u and v are adjacent, NEXT-NEIGHBOR(v) returns the neighbor of v with *lowest* ID that has not been returned by any previous NEXT-NEIGHBOR(v) call yet (until none is left), and RANDOM-NEIGHBOR(v) returns a uniform random neighbor of v (if v is not isolated).

number of communities). As in [ELMR17] (and unlike the generators in [GGN03, GGN10, NN07]), our techniques allow unlimited probes.

While VERTEX-PAIR and NEXT-NEIGHBOR probes, as well as ALL-NEIGHBORS probes for sparse graphs, have been considered in the prior works of [ELMR17, GGN03, GGN10, NN07], we provide the first implementation (to the best of our knowledge) of RANDOM-NEIGHBOR probes, which do not follow trivially from the ALL-NEIGHBOR probes in *non-sparse graphs*. Such probes are useful, for instance, for sub-linear algorithms that employ random walk processes. RANDOM-NEIGHBOR probes present particularly interesting challenges, since as we note in Section 5.1.1.1, (1) our implementation does not resort to explicitly sampling the degree of any vertex in order to generate a random neighbor, and (2) RANDOM-NEIGHBOR probes affect the conditional probabilities of the remaining neighbors in a non-trivial manner. The former issue arises as sampling the degree of the probe vertex, we suspect, is not viable for *sub-linear* generators: this quantity alone imposes dependence on the existence of *all* of its potential incident edges. Hence, our generator needs to return a random neighbor with probability reciprocal to the probe vertex's degree, without the knowledge of this quantity itself. For the latter issue, even without committing to the degrees, answers to RANDOM-NEIGHBOR probes still affect the conditional probabilities of the remaining adjacencies in a global and non-trivial manner – that is, from the point of view of the *agent* interacting with the generator. The generator, however, must somehow maintain and leverage its additional *internal knowledge* of the partially-generated graph, to keep its computation tractable throughout the entire graph generation process.

We then consider local-access generators for directed graphs in Kleinberg's Small World 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 probes in $\mathcal{O}(\text{poly}(\log n))$ time, space and random bits per probe (since such graphs are sparse, the other probes follow directly).

5.1.1 Our results and techniques

We begin by stating our formalization of local-access generators (Section 5.2.1) inspired by that of [ELMR17] (see also [ELMR16] for the original definition). Our work provides local-access generators for various basic classes of graphs described in the following, with VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR probes. In all of our results, each probe 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 probes. Our bounds assume constant computation time for each arithmetic operation with $O(\log n)$-bit precision. Each of our generators constructs a random graph drawn from a distribution that is $1/\text{poly}(n)$-close to the desired distribution in the L_1-distance.[2]

5.1.1.1 Undirected Graphs

In Section 5.3 we construct local access generators for the generic class of undirected graphs with *independent edge probabilities* $\{p_{u,v}\}_{u,v\in V}$, where $p_{u,v}$ denote the probability that there is an edge

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

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 Probes. 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 when assuming access to the oracle that computes "skip" probabilities $F(v, a, b) = \prod_{u=a}^{b}(1 - p_{v,u})$, where $F(v, a, b)$ is the probability that v has no neighbors in the range $[a, b]$. We later show that it is possible to compute this quantity efficiently for the $G(n, p)$ and Stochastic block models.

A main difficulty in our setup, as compared to [ELMR17], arises from the fact that our graph is undirected, and thus we must design a data structure that "informs" all (potentially $\Theta(n)$) non-neighbors once we decide on the probe vertex's next neighbor. More concretely, if u' is sampled as the next neighbor of v after its previous neighbor u, we must maintain consistency in subsequent steps by ensuring that none of the vertices in the range (u, u') return v as a neighbor. The method given in [ELMR17] handles the preferential attachment graphs which are generated in an incremental process, where each new vertex arrives sequentially and connects itself to an existing vertex, forming a rooted tree structure. However, our update requires a more sophisticated data structure as we must later support RANDOM-NEIGHBOR probes, where we generate neighbors (and non-neighbors) at random locations.

Random-Neighbor Probes. We provide efficient RANDOM-NEIGHBOR probes. The ability to do so is surprising. First, note that after performing a RANDOM-NEIGHBOR probe, all other conditional probabilities will be affected in a non-trivial way.[3] 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 5.3.3) 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 probes. For each vertex v, we divide the vertex set into consecutive ranges (buckets), so that each bucket contains, in expectation, roughly the same number of neighbors $\sum_{u=a}^{b} p_{v,u}$. 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. 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.

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

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

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 5.4.1, we apply our construction to random $G(n,p)$ graphs for arbitrary p, and obtain VERTEX-PAIR, NEXT-NEIGHBOR, and RANDOM-NEIGHBOR probes, using polylogarithmic resources (time, space and random bits) per probe. 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 probes, and yet can generate a full graph in $\widetilde{O}(n+m)$ time and space (Corollary 5.4.2), which is tight up to polylogarithmic factors.

Stochastic Block Model. We generalize our construction to the Stochastic Block Model. In this model, the vertex set is partitioned into r *communities* $\{C_1, \ldots, C_r\}$. The probability that an edge exists between $u \in C_i$ and $v \in C_j$, is $p_{i,j}$, given 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, we aim to construct generators where the community assignment of vertices are independently sampled from some given distribution R.

Our approach is, as before, to sample for the next neighbor or a random neighbor directly, although our result does not simply follow closed-form formulas, as the probabilities for the potential edges now depend on the communities of endpoints. To handle this issue, we observe that it is sufficient to efficiently count the number of vertices of each community in any range of contiguous vertex indices. We then design a data structure extending a construction of [GGN10], which maintain these counts for ranges of vertices, and "sample" the partition of their counts only on an as-needed basis. This extension results in an efficient technique to sample counts from the *multivariate hypergeometric distribution* (Section 5.4.2.1): this sampling procedure may be of independent interest. For r communities, this yields an implementation with $\mathcal{O}(r \cdot \text{poly}(\log n))$ overhead in required resources for each operation.

5.1.1.2 Directed Graphs

Lastly, we consider Kleinberg's Small World model [Kle00,MN04] in Section 5.5. While Small-World models are proposed to capture properties of observed data such as small shortest-path distances and large clustering coefficients [WS98], this important special case of Kleinberg's model, defined on two-dimensional grids, demonstrates underlying geographical structures of networks. The vertices are aligned on a $\sqrt{n} \times \sqrt{n}$ grid, and the edge probabilities are a function of a two-dimensional distance metric. Since the degree of each vertex in this model is $O(\log n)$ with high probability, we design generators supporting ALL-NEIGHBOR probes.

5.1.2 Additional related work

Random graph models. The Erdős-Rényi model, given in [ER60], is one of the most simple theoretical random graph model, yet more specialized models are required to capture properties of real-world data. The Stochastic Block model (or the planted partition model) was proposed in [HLL83] originally for modeling social networks; nonetheless, it has proven to be an useful general statistical model in numerous fields, including recommender systems [LSY03,SC11], medicine [SPT+01], social

networks [For10, NWS02], molecular biology [CY06, MPN+99], genetics [CAT16, JTZ04, CSC+07], and image segmentation [SM00]. Canonical problems for this model are the community detection and community recovery problems: some recent works include [CRV15, MNS15, AS15, ABH16]; see e.g., [Abb18] for survey of recent results. The study of Small-World networks is originated in [WS98] has frequently been observed, and proven to be important for the modeling of many real world graphs such as social networks [DMW03, TM67], brain neurons [BB06], among many others. Kleinberg's model on the simple lattice topology (as considered in this chapter) imposes a geographical that allows navigations, yielding important results such as routing algorithms (decentralized search) [Kle00, MN04]. See also e.g., [New00] and Chapter 20 of [EK10].

Generation of random graphs. The problem of local-access implementation of random graphs has been considered in the aforementioned work [GGN03, NN07, ELMR17], as well as in [MRVX12] that locally generates out-going edges on bipartite graphs while minimizing the maximum in-degree. The problem of generating full graph instances for random graph models have been frequently considered in many models of computations, such as sequential algorithms [MKI+03, BB05, NLKB11, MH11], and the parallel computation model [AK17].

Probe models. In the study of sub-linear time graph algorithms where reading the entire input is infeasible, it is necessary to specify how the algorithm may access the input graph, normally by defining the type of probes that the algorithm may ask about the input graph; the allowed types of probes can greatly affect the performance of the algorithms. While NEXT-NEIGHBOR probe is only recently considered in [ELMR17], there are other probe models providing a neighbor of a vertex, such as asking for an entry in the adjacency-list representation [GR02], or traversing to a random neighbor [BK10]. On the other hand, the VERTEX-PAIR probe is common in the study of dense graphs as accessing the adjacency matrix representation [GGR98]. The ALL-NEIGHBORS probe has recently been explicitly considered in local algorithms [FPV18].

5.2 Preliminaries

For clarity, we first describe the model of local-access generators, with focus on the undirected case; we explain the directed case within their own section.

5.2.1 Local-access generators

We consider the problem of locally generating random graphs $G = (V, E)$ drawn from the desired families of simple unweighted graphs, undirected or directed. We denote the number of vertices $n = |V|$, and refer to each vertex simply via its unique ID from $[n]$. For undirected G, the set of neighbors of $v \in V$ is defined as $\Gamma(v)$ and its degree is defined as $\deg(v) = |\Gamma(v)|$. Inspired by the model of [ELMR17] (formally described in [ELMR16]), we propose our adaptation of local-access generators as follows.

Definition 5.2.1. *A* local-access generator *of a random graph G sampled from a distribution* D*, is a data structure that provides access to G by answering various types of* supported probes*, 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 generated by the generator must be sampled from some distribution D' that is ϵ-close to the desired distribution D in the L_1-distance.*[4] *In this work we focus on supporting $\epsilon = n^{-c}$ for any desired constant $c > 0$. As for* RANDOM-NEIGHBOR(v)*, the distribution from which a neighbor is returned must be ϵ-close to the uniform distribution over neighbors of v with respect to the sampled random graph G (with probability $1 - n^{-c}$ for each probe).*

- ***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 probe and update the data structure, must be sub-linear, preferably* $\mathrm{poly}(\log n)$*. In the majority of this work, we aim to satisfy these conditions with high probability (for each probe), but we also further consider different trade-offs for deterministic and amortized guarantees in Section 5.6 and Section 5.7.*

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

Supported Probes. For undirected graphs, we consider probes of the following forms.

- NEXT-NEIGHBOR(v): The generator returns the neighbor of v with the lowest ID that has not been returned during the execution of the generator so far. If all neighbors of u have already been returned, the generator returns $n + 1$.

- RANDOM-NEIGHBOR(v): The generator returns a neighbor of v uniformly at random (with probability $1/\deg(v)$ each). If v is isolated, $\perp$ is returned.

- VERTEX-PAIR(u, v): The generator returns 1 or 0, indicating whether $\{u, v\} \in E$ or not.

Differences from the oracle access model in Chapter 2. We remark the following differences compared to the oracle access model employed by the LCAs in Chapter 2. First, we represent vertices via their unique IDs in $[n]$, whereas LCAs in Chaper 2 do not rely on this assumption. Second, we explicitly exclude the DEGREE probes due to the discussed difficulties. Third, because we do not have an explicit adjacency-list representation of the generated graphs, the VERTEX-PAIR(u, v) probe, unlike the ADJACENCY probe with parameter $\langle u, v\rangle$, does not return the index i indicating that v is the i^{th} neighbor of u.

5.2.2 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.

[4]Previous works on graph generators such as [BB05, ELMR17] do not consider errors (or additional resource requirements) resulting from the use of finite-precision arithmetic, and hence provide identical equivalence (where $\epsilon = 0$). Our definition, however, handles these issues explicitly, under reasonable assumptions on the computation model.

Stochastic Block Model. This model is a generalization of the Erdős-Rényi Model. The vertex set V is partitioned into r communities $C_1, \ldots, C_r$. The probability that the edge $\{u, v\}$ exists is $p_{i,j}$ when $u \in C_i$ and $v \in C_j$, where the probabilities are given as an $r \times r$ symmetric matrix $\mathbf{P} = [p_{i,j}]_{i,j \in [r]}$. We assume that we are given explicitly the distribution R over the communities, and each vertex is assigned its community according to R independently at random. We remark that our algorithm also supports the alternative specification where the community sizes $\langle |C_1|, \ldots, |C_r| \rangle$ are given instead, where the assignment of vertices V into these communities is chosen uniformly at random.

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 [Kle00], we assume that the probability is rounded down to 1 if $c/(\text{dist}(u, v))^2 > 1$.

Since the Small-World model generates relatively sparse graphs, we support an ALL-NEIGHBORS(v) probe which returns the entire list of out-neighbors of v.

5.2.3 Miscellaneous

Arithmetic operations. Let N be a sufficiently large number of bits required to maintain a multiplicative error of at most a $\frac{1}{\text{poly}(n)}$ factor over $\text{poly}(n)$ elementary computations $(+, -, \cdot, /, \exp)$.[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 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 $O(\log n)$ CDF evaluations, one can sample from a distribution that is $\frac{1}{\text{poly}(n)}$-close to X in L_1-distance.[6]

5.3 Local-Access Generators for Random Undirected Graphs

In this section, we provide an efficient implementation of local-access generators for random undirected graphs when the probabilities $p_{u,v} = \mathbb{P}[\{u, v\} \in E]$ are given. More specifically, we assume that we can efficiently compute: (1) the probability that there is no edge between a vertex u and a

[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 $a^b = O(1)$. For this, $N = O(\log n)$ 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 $\mathbb{P}[X \leq x] \geq r$.

range of consecutive vertices from $[a, b]$, namely $\prod_{u=a}^{b}(1 - p_{v,u})$, and (2) the sum of the edge probabilities (i.e., the expected number of edges) between u and vertices from $[a, b]$, namely $\sum_{u=a}^{b} p_{v,u}$. In Section 5.4, we provide subroutines for computing these for the Erdős-Rényi model and the Stochastic Block model with randomly-assigned communities. We also begin by assuming perfect-precision arithmetic, until Section 5.3.5 where we to relax this assumption to $N = \Theta(\log n)$-bit precision.

First, we propose a simple implementation of our generator in Section 5.3.1 that sequentially fills out the adjacency matrix; while we do not focus on its efficiency, we establish some basic concepts for further analysis in this section. Next, we improve our subroutine for NEXT-NEIGHBOR probes in Section 5.3.2; this algorithm samples for the next candidate of the next neighbor in a more direct manner to speed-up the process. Extending this construction to support RANDOM-NEIGHBOR probes, we obtain our main algorithm in Section 5.3.3 via the bucketing technique; partition the vertex set into contiguous ranges to normalize the expected number of neighbors in each bucket. The subroutine that samples for neighbors within a bucket, along with the remaining analysis of the algorithm, is given later in Section 5.3.4.

We remark that throughout this chapter, for convenient, we generally index our algorithms via the corresponding figure numbers. For example, Algorithm 5-1 refers to the algorithm with pseudocode provided in Figure 5-1.

5.3.1 Naïve Generator with an explicit adjacency matrix

Each entry $\mathbf{A}[u][v]$ occupies exactly one of following three states: $\mathbf{A}[u][v] = 1$ or 0 if the generator has determined that $\{u, v\} \in E$ or $\{u, v\} \notin E$, respectively, and $\mathbf{A}[u][v] = \phi$ if whether $\{u, v\} \in E$ or not will be determined by future random choices. Aside from $\mathbf{A}$, our generator also maintains the vector **last**, where $\mathbf{last}[v]$ records the neighbor of v returned in the last call NEXT-NEIGHBOR(v), or 0 if no such call has been invoked. This definition of **last** was introduced in [ELMR17]. All cells of $\mathbf{A}$ and **last** are initialized to ϕ and 0, respectively. We refer to Algorithm 5-1 for its straightforward implemention, but highlight some notations and useful observations here.

Characterizing random choices via $X_{u,v}$'s. Algorithm 5-1 updates the cell $\mathbf{A}[u][v] = \phi$ to the value of the Bernoulli random variable (RV) $X_{u,v} \sim \mathsf{Bern}(p_{u,v})$ (i.e., flip a coin with bias $p_{u,v}$) only when it needs to decide whether $\{u, v\} \in E$. For the sake of analysis, we will frequently consider the *entire* table of RVs $X_{u,v}$ being sampled *up-front* (i.e., flip all coins), and the algorithm simply "uncovers" these variables instead of making coin-flips. Thus, every cell $\mathbf{A}[u][v]$ is originally ϕ, but will eventually take the value $X_{u,v}$ once the graph generation is complete.

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

Capturing the state of the partially-generated graph with A. Under the presence of RANDOM-NEIGHBOR probes, the probability distribution of the random graphs conditioned on the

```
VERTEX-PAIR(u, v)
  if A[u][v] = φ then
      draw X_{u,v} ~ Bern(p_{u,v})
      A[v][u], A[u][v] ← X_{u,v}
  return A[u][v]

NEXT-NEIGHBOR(v)
  for u ← last[v] + 1 to n do
      if VERTEX-PAIR(v, u) = 1 then
          last[v] ← u
          return u
  last[v] ← n + 1
  return n + 1

RANDOM-NEIGHBOR(u, v)
  R ← V
  repeat
      sample u ∈ R u.a.r.
      if VERTEX-PAIR(v, u) = 1 then
          return u
      else
          R ← R \ {u}
  until R = ∅
  return ⊥
```

Figure 5-1: Naïve generator

past probes 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 5-1, but in a more implicit and efficient fashion in order to achieve the desired complexities. Another benefit of this approach is that any analysis can be performed on the simple representation $\mathbf{A}$ rather than any complicated data structure we may employ.

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

5.3.2 Improved NEXT-NEIGHBOR probes via run-of-0's sampling

We now speed-up our NEXT-NEIGHBOR(v) procedure by attempting to sample for the first index $u > \mathbf{last}[v]$ of $X_{v,u} = 1$, from a sequence of Bernoulli RVs $\{X_{v,u}\}_{u > \mathbf{last}[v]}$. To do so, we sample a

consecutive "run" of 0's with probability $\prod_{u=\mathbf{last}[v]+1}^{u'}(1-p_{v,u})$ (probability that $(\mathbf{last}[v], u']$ contains no neighbors of v), 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 probes NEXT-NEIGHBOR(u) for $u > \mathbf{last}[v]$. To this end, we give a succinct data structure that determines the value of $\mathbf{A}[v][u]$, and more generally, captures the state of $\mathbf{A}$, in Section 5.3.2.1. Using this data structure, we ensure that our sampled run does not skip over any 1. Next, for the sampled index u of the first occurrence of 1, we check against this data structure to see if $\mathbf{A}[v][u]$ is already assigned to 0, in which case we re-sample for a new candidate $u' > u$. Section 5.3.2.2 discusses the subtlety of this issue.

While this sampling process has been used in [ELMR17], we include the full description here, as it will become a building block for further sections, as well as clarifying some implementation differences, particularly in our succinct data structure for representing $\mathbf{A}$. Since we aim to support arbitrary edge probabilities (resulting in either dense or sparse graphs), this warrants a new method for bounding the number of re-sampling iterations of this process. In particular, we show (in Section 5.6) that $\mathcal{O}(\log n)$ iterations suffice with high probability even if the probes are adversarial, and that this method may be extended to support VERTEX-PAIR probes. Lastly, we use this section to highlight its limitation: why this approach alone cannot handle RANDOM-NEIGHBOR probes.

5.3.2.1 Data structure

From the definition of $X_{u,v}$, NEXT-NEIGHBOR(v) is given by $\min\{u > \mathbf{last}[v] : X_{v,u} = 1\}$ (or $n+1$ if no satisfying u exists). Let $P_v = \{u : \mathbf{A}[v][u] = 1\}$ be the set of known neighbors of v, and $w_v = \min\{(P_v \cap (\mathbf{last}[v], n]) \cup \{n+1\}\}$ be its first known neighbor not yet reported by a NEXT-NEIGHBOR(v) probe, 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.

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 probe 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. We say that $X_{u,v}$ is *decided* if $\mathbf{A}[u][v] \neq \phi$, and call it *undecided* otherwise.

Lemma 5.3.1. *The data structures* $\mathbf{last}$ *and* P_v*'s together provide a succinct representation of* $\mathbf{A}$ *when only* NEXT-NEIGHBOR *probes are allowed. That is,* $\mathbf{A}[v][u] = 1$ *precisely when* $u \in P_v$*. Otherwise,* $\mathbf{A}[v][u] = 0$ *when* $u < \mathbf{last}[v]$ *or* $v < \mathbf{last}[u]$*. In all other cases,* $\mathbf{A}[v][u] = \phi$*.*

Proof. The condition for $\mathbf{A}[v][u] = 1$ clearly holds by constuction. Otherwise, observe that $\mathbf{A}[v][u]$ becomes decided (its value changes 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. □

5.3.2.2 Probes and Updates

We now provide our generator (Algorithm 5-2), 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. Algorithm 5-2 considers the following experiment for sampling the next neighbor of v in the range $(\mathbf{last}[v], w_v)$. Suppose that we generate a sequence of $w_v - \mathbf{last}[v] - 1$ independent coin-tosses, where the i^{th} coin $C_{v,u}$ corresponding to $u = \mathbf{last}[v] + i$ has bias $p_{v,u}$, regardless of whether $X_{v,u}$'s are decided or not. Then, we use the sequence $\langle C_{v,u} \rangle$ to assign values to *undecided* random variable $X_{v,u}$. The crucial observation here is that the *decided* random variables $X_{v,u} = 0$ do not need coin-flips, and the corresponding coin result $C_{v,u}$ can simply be discarded. Thus, we need to generate coin-flips up until we encounter some u satisfying both $C_{v,u} = 1$ and $\mathbf{A}[v][u] = \phi$.

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

Figure 5-2: NEXT-NEIGHBOR procedure

Let $\mathsf{F}(v, a, b)$ denote the probability distribution of the occurrence u of the first coin-flip $C_{v,u} = 1$ among the neighbors in (a, b). More specifically, $F \sim \mathsf{F}(v, a, b)$ represents the event that $C_{v,a+1} = \cdots = C_{v,F-1} = 0$ and $C_{v,F} = 1$, which happens with probability $\mathbb{P}[F = f] = \prod_{u=a+1}^{f-1}(1 - p_{v,u}) \cdot p_{v,f}$. For convenience, let $F = b$ denote the event where all $C_{v,u} = 0$. Our algorithm samples $F_1 \sim \mathsf{F}(v, \mathbf{last}[v], w_v)$ to find the first occurrence of $C_{v,F_1} = 1$, then samples $F_2 \sim \mathsf{F}(v, F_1, w_v)$ to find the second occurrence $C_{v,F_2} = 1$, and so on. These values $\{F_i\}$ are iterated as u in Algorithm 5-2. We repeat until we find u such that $\mathbf{A}[v][u] = \phi$. 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 probes. There are two main issues that prevent this method from supporting RANDOM-NEIGHBOR probes. 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 5-2, 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 $v \leq \mathbf{last}[u]$. Unfortunately, in order to extend our construction to support RANDOM-NEIGHBOR probes, 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. More specifically, to speed-up the sampling process for small $p_{v,u}$'s, we must generate many random non-neighbors at once, but we cannot afford to spend time linear in the number of created 0's to update our data structure. We remedy these issues via the following bucketing approach.

5.3.3 Final generator via the bucketing approach

We now resolve both of the above issues via the bucketing approach, allowing our generator to support all remaining types of probes. We begin this section by focusing first on RANDOM-NEIGHBOR probes. we divide the neighbors of v into *buckets* $B_v = \{B_v^{(1)}, B_v^{(2)}, \ldots\}$, so that each bucket contains, in expectation, roughly the same number of neighbors of v. We may then implement RANDOM-NEIGHBOR(v) by randomly selecting a bucket $B_v^{(i)}$, fill in entries $\mathbf{A}[v][u]$ for $u \in B_v^{(i)}$ with 1's and 0's, then report a random neighbor from this bucket. As the bucket size may be too large, instead of using a linear scan, we will use the NEXT-NEIGHBOR subroutine from Algorithm 5-2. 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.

Nonetheless, as the actual number of neighbors appearing in each bucket may be different, we balance out these discrepancies by performing *rejection sampling*, again without the knowledge of $\deg(v)$ (Section 5.3.1). Leveraging the fact that the maximum number of neighbors in any bucket is $O(\log n)$, we show not only that the probability of success in the rejection sampling process is at least $1/\operatorname{poly}(\log n)$, but the number of iterations required by NEXT-NEIGHBOR is also bounded by $\operatorname{poly}(\log n)$, achieving the overall desired 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.

5.3.3.1 Partitioning into buckets

We pick 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 is a contiguous range of vertices, where the total probability of vertices being neighbors of v is mostly between $L-1$ and $L+1$ (for example, for $G(n,p)$ each bucket contains approximately L/p neighbors). Define $\Gamma^{(i)}(v) = \Gamma(v) \cap B_v^{(i)}$, the actual neighbors appearing in bucket $B_v^{(i)}$. By construction, $L - 1 < \mathbb{E}[|\Gamma^{(i)}(v)|] < L + 1$ for every bucket (except for $i = |B_v|$ where the lower bound may fail).

Via this property, we show that with high probability, all the bucket sizes $|\Gamma^{(i)}(v)|$ are at most $\mathcal{O}(L \log n)$ and at least 2/3 fraction of the buckets are non-empty.

Lemma 5.3.2. *With high probability, the number of neighbors in every bucket, $|\Gamma^{(i)}(v)|$, is at most $\mathcal{O}(L \log n)$.*

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

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

for any constant $c > 0$. □

Lemma 5.3.3. *With high probability, for every v such that $|B_v| = \Omega(\log n)$, at least a fraction of $2/3$ of the buckets $\{B_v^{(i)}\}_{i \in [|B_v|]}$ are non-empty.*

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

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

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

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

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

5.3.3.2 Filling a bucket

We consider buckets to be in two possible states – filled or unfilled. Initially, all buckets are considered unfilled. In our algorithm we will maintain, for each bucket $B_v^{(i)}$, the set $P_v^{(i)}$ of known neighbors of u in bucket $B_v^{(i)}$; this is a refinement of the set P_v in Section 5.3.2. We define the behaviors of the procedure FILL(v, i) as follows. When invoked on an unfilled bucket $B_v^{(i)}$, FILL(v, i) performs the following tasks:

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

For the sake of presentation, we postpone our description of the implementation of FILL to Section 5.3.4. For now, let us use FILL as a black-box operation.

5.3.3.3 Putting it all together: Random-Neighbor probes

Consider Algorithm 5-3 for generating a random neighbor via rejection sampling, in a rather similar overall framework as the simple implementation in Section 5.3.1. For simplicity, throughout the analysis, we assume $|B_v| = \Omega(\log n)$; otherwise, invoke FILL(v, i) for all $i \in [|B_v|]$ to obtain the

entire neighbor list $\Gamma(v)$. This does not affect the analysis because we will bound the (expected) number of calls that Algorithm 5-3 makes to FILL by $O(\log n)$.

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

Figure 5-3: Bucketing generator

To obtain a random neighbor, we first choose a bucket $B_v^{(i)}$ uniformly at random. If the bucket is not filled, we invoke FILL(v, i) and fill it. 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 5.3.2; we accept this bucket with probability $|P_v^{(i)}|/M$, which does not exceed 1 with high probability. (We also remove i from the pool precisely when $P_v^{(i)} = \emptyset$, before proceeding to the next iteration – the **else** statement is coupled with the **if** directly above it.) If we choose to accept this bucket, we return a random neighbor from $P_v^{(i)}$. Otherwise, *reject* 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.

Lemma 5.3.4. *Algorithm 5-3 returns a uniformly random neighbor of vertex v.*

Proof. It suffices to find the probability that a neighbor $u \in \Gamma(v)$ is returned, in a single iteration. During an iteration, 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 is $1/|P_v^{(i)}|$. Hence, the overall probability of returning u is $1/(|R|M)$, which is independent of u. So, each vertex is returned with the same probability. □

Lemma 5.3.5. *Algorithm 5-3 terminates in $\mathcal{O}(\log n)$ iterations in expectation, or $\mathcal{O}(\log^2 n)$ iterations with high probability.*

Proof. The probability that some vertex from $P_v^{(i)}$ is accepted in an iteration is at least $1/(|R|M)$. From Lemma 5.3.3, (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 5.3.2. Hence, the success probability of each iteration is at least $1/(3M) = \Omega(1/\log n)$. Thus, the number of iterations required is $O(\log^2 n)$ with high probability. □

5.3.4 Implementation of the FILL function

```
FILL(v, i)
(a, b) ← B_j^(i)
repeat
    sample u ~ F(v, a, b)
    B_u^(j) ← bucket containing v
    if B_u^(j) is not filled then
        P_v^(i) ← P_v^(i) ∪ {u}
        P_u^(j) ← P_u^(j) ∪ {v}
    a ← u
until a ≥ b
mark B_u^(j) as filled
```

Figure 5-4: FILL procedure

Lastly, we describe the implementation of the FILL procedure, employing the approach of skipping non-neighbors. 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 5.3.2, while we have recorded all occurrences of $\mathbf{A}[v][u] = 1$ in $P_v^{(i)}$, we need and efficient way of checking whether $\mathbf{A}[v][u] = 0$ or ϕ. In Algorithm 5-2, **last** serves this purpose by showing that $\mathbf{A}[v][u]$ for all $u \leq \mathbf{last}[v]$ are decided as shown in Lemma 5.3.1. 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 5-4 for the FILL procedure fulfilling the requirement previously given in Section 5.3.3.2.

Lemma 5.3.6. *The data structures $P_v^{(i)}$'s and the bucket marking bits together provide a succinct representation of $\mathbf{A}$ as long as modifications to $\mathbf{A}$ are performed solely by the* FILL *operation in Algorithm 5-4. In particular, let $u \in B_v^{(i)}$ and $v \in B_u^{(j)}$. Then, $\mathbf{A}[v][u] = 1$ if and only if $u \in P_v^{(i)}$. Otherwise, $\mathbf{A}[v][u] = 0$ when at least one of $B_v^{(i)}$ or $B_u^{(j)}$ is marked as* filled. *In all remaining cases, $\mathbf{A}[v][u] = \phi$.*

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

Note that $P_v^{(i)}$'s, maintained by our generator, are initially empty but may not still be empty at the beginning of the FILL function call. These $P_v^{(i)}$'s are again instantiated and stored in a dictionary once they become non-empty. Further, observe that the coin-flips are simulated independently of the state of $P_v^{(i)}$, so the number of iterations of Algorithm 5-4 is the same as the number of coins $C_{v,u} = 1$ which is, in expectation, a constant (namely $\sum_{u \in B_v^{(i)}} \mathbb{P}[C_{v,u} = 1] = \sum_{u \in B_v^{(i)}} p_{v,u} \leq L + 1$).

We have completed the description of our implementation of RANDOM-NEIGHBOR. As Algorithm 5-4 requires poly$(\log n)$ resources per call with high probability, combining with Lemma 5.3.5, we obtain the desired polylogarithmic resource bound for RANDOM-NEIGHBOR. More formally, by tracking the resource required by Algorithm 5-4 we obtain the following lemma; note that "additional space" refers to the enduring memory that the generator must allocate and keep even after the execution, not its computation memory. The $\log n$ factors in our complexities are required to perform binary-search for the range of $B_v^{(i)}$, or for the value u from the CDF of $\mathsf{F}(u, a, b)$, and to maintain the ordered sets $P_v^{(i)}$ and $P_u^{(j)}$.

Lemma 5.3.7. *Each execution of Algorithm 5-4 (the* FILL *operation) on an* unfilled *bucket $B_v^{(i)}$, in expectation:*

- *terminates within $O(1)$ iterations (of its* **repeat** *loop);*
- *computes $O(\log n)$ 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 $O(\log n)$ time, $O(1)$ random N-bit words, and $O(1)$ additional space.*

Observe that the number of iterations required by Algorithm 5-4 only depends on its random coin-flips and independent of the state of the algorithm. Combining with Lemma 5.3.5, we finally obtain polylogarithimc resource bound for our implementation of RANDOM-NEIGHBOR.

Corollary 5.3.8. *Each execution of Algorithm 5-3 (the* RANDOM-NEIGHBOR *probe), with high probability,*

- *terminates within $O(\log^2 n)$ iterations (of its* **repeat** *loop);*
- *computes $O(\log^3 n)$ 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 $O(\log^3 n)$ time, $O(\log^2 n)$ random N-bit words, and $O(\log^2 n)$ additional space.*

Extension to other probe types. We now extend our algorithm to support remaining probes.

- VERTEX-PAIR(u, v): We simply need to make sure that Lemma 5.3.6 holds, so we first apply FILL(u, j) on bucket $B_u^{(j)}$ containing v (if needed), then answer accordingly.

- NEXT-NEIGHBOR(v): We maintain **last**, and keep invoking FILL until we find a neighbor. Recall that the probability that a particular bucket is empty is a constant (Lemma 5.3.3). 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 $O(\log n)$ calls to FILL.

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

Theorem 5.3.9. *Under the assumption of*

1. *perfect-precision arithmetic, including the generation of random real numbers in* $[0, 1)$
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 *probes that uses polylogarithmic running time, additional space, and random words per probe with high probability.*

5.3.5 Removing the perfect-precision arithmetic assumption

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

Observe that the distribution of the graphs constructed by our generator is governed entirely by the samples u drawn from $\mathsf{F}(v, a, b)$ in Algorithm 5-4. By our assumption, the CDF of any $\mathsf{F}(v, a, b)$ can be efficiently computed from $\prod_{u=a}^{u'}(1-p_{v,u})$, and thus sampling with $\frac{1}{\mathrm{poly}(n)}$ error in the L_1-distance requires a random N-bit word and a binary-search in $O(\log(b-a+1)) = O(\log n)$ iterations. Using this crucial fact, we prove our lemma that removes the perfect-precision arithmetic assumption.

Lemma 5.3.10. *If Algorithm 5-4 (the* FILL *operation) is repeatedly invoked to construct a graph* G *by drawing the value* u *for at most* S *times in total, each of which comes from some distribution* $\mathsf{F}'(v, a, b)$ *that is* ϵ*-close in* L_1*-distance to the correct distribution* $\mathsf{F}(v, a, b)$ *that perfectly generates the desired distribution* G *over all graphs, then the distribution* G' *of the generated graph* G *is* (ϵS)*-close to* G *in the* L_1*-distance.*

Proof. For simplicity, assume that the algorithm generates the graph to completion according to a sequence of up to n^2 distinct buckets $\mathcal{B} = \langle B_{v_1}^{(u_1)}, B_{v_2}^{(u_2)}, \ldots \rangle$, where each $B_{v_i}^{(u_i)}$ specifies the unfilled bucket in which any probe 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 5-4) with value u, and have generated $\mathbf{A}$ so far. Let $t_{\mathbf{A}}$ denote the *terminal state* after processing all probes and having generated the graph $G_{\mathbf{A}}$ represented by $\mathbf{A}$. We note that $\mathbf{A}$ is used here in the analysis but not explicitly maintained; further, it reflects the changes in every iteration: as u is updated during each iteration of FILL, the cells $\mathbf{A}[v][u'] = \phi$ for $u' < u$ (within that bucket) that has been skipped are also updated to 0.

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

probability that the algorithm advances from s to s' by using a sample from the correct $\mathsf{F}(v,a,b)$ and from the approximated $\mathsf{F}'(v,a,b)$, respectively. Consequently, $\|\mathsf{T}_s - \mathsf{T}'_s\|_1 \le \epsilon$ for every $s \in \mathcal{S}$.

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

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

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

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

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 $\sum_\pi q^{\min}(\pi) = \sum_\pi q^{\min}(\pi_S) \ge (1-\epsilon)^S \ge 1 - \epsilon S$. Hence, $\sum_{G_{\mathbf{A}}} \min\{\mathsf{G}(G_{\mathbf{A}}), \mathsf{G}'(G_{\mathbf{A}})\} \ge \sum_{G_{\mathbf{A}}} \mathsf{G}^{\min}(G_{\mathbf{A}}) \ge 1 - \epsilon S$, implying that $\|\mathsf{G} - \mathsf{G}'\|_1 \le \epsilon S$, as desired. In particular, by substituting $\epsilon = \frac{1}{\mathrm{poly}(n)}$ and $S = O(n^3)$, we have shown that Algorithm 5-4 only creates a $\frac{1}{\mathrm{poly}(n)}$ error in the L_1-distance. □

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

5.4 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 randon graphs. That is, as required by Theorem 5.3.9, 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 $\{C_1,\ldots,C_r\}$ identically and independently at random, according to some given distribution R over the communities. We formulate a method of sampling community assignments locally. This essentially allows us to sample from the *multivariate hypergeometric distribution*, using $\mathrm{poly}(\log n)$ random bits, which may be of independent interest. We remark that, as our first step, we sample for the number of vertices of each community. That is, our construction can alternatively support the community assignment where the number of vertices of each community is given, under the assumption that the *partition* of the vertex set into communities is chosen uniformly at random.

Refer to Section 5.2.2 for specifications of our random graph models. We also construct local-access generators for these models with deterministic guarantees in Section 5.7.

5.4.1 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 5.4.1. *The final algorithm in Section 5.3 locally generates a random graph from the Erdős-Rényi $G(n,p)$ model using $O(\log^3 n)$ time, $O(\log^2 n)$ random N-bit words, and $O(\log^2 n)$ additional space per probe with high probability.*

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

While we are able to generate our random graph on-the-fly supporting all three types of probes, our construction still only requires $O(m+n)$ space (N-bit words) in total at any state; that is, we keep $O(n)$ words for **last**, $O(1)$ 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 5.4.2. *The final algorithm in Section 5.3 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 $\tilde{O}(pn^2)$ in expectation. This is optimal up to $\mathcal{O}(\mathrm{poly}(\log n))$ factors.*

5.4.2 Stochastic Bock Model

For the Stochastic Block model, each vertex is assigned to some community C_i, $i\in[r]$. By partitioning the product by communities, we may rewrite the desired formulas, for $v\in C_i$, as

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

Theorem 5.4.3. *There exists a data structure (generator) that samples a community for each vertex independently at random from* R *with* $\frac{1}{\text{poly}(n)}$ *error in the* L_1*-distance, and supports probes that ask for the number of occurrences of vertices of each community in any contiguous range, using* $O(r \text{ poly}(\log n))$ *time, random* N*-bit words and additional space per probe (in the worst case). Further, this data structure may be implemented in such a way that requires no overhead for initialization.*

Corollary 5.4.4. *The final algorithm in Section 5.3 generates a random graph from the Stochastic Block model with randomly-assigned communities using* $O(r \text{ poly}(\log n))$ *time, random* N*-bit words, and additional space per probe with high probability.*

We provide the full details of the construction in the following Section 5.4.2.1. Our construction extends upon a similar generator in the work of [GGN10] which only supports $r = 2$. Our overall data structure is a balanced binary tree, where the root corresponds to the entire range of indices $\{1,\ldots,n\}$, and the children of each vertex corresponds to each half of the parent's range. Each node[7] holds the number of vertices of each community in its range. The tree initially contains only the root, with the number of vertices of each community sampled according to the multinomial distribution[8] (for n samples (vertices) from the probability distribution R). The children are only generated top-down on an as-needed basis according to the given probes. 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.*

5.4.2.1 Sampling from the Multivariate Hypergeometric Distribution

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

[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 [Knu98]

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

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

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

The sample $\mathbf{S}_\ell^{\mathbf{C}}$ above may be generated easily by simulating the drawing process, but this may take $\Omega(\ell)$ iterations, which have linear dependency in n in the worst case: $\ell = \Theta(B) = \Theta(n)$. Instead, we aim to generate such a sample in $O(r \operatorname{poly}(\log n))$ time with high probability. We first make use of the following procedure from [GGN10].

Lemma 5.4.5. *Suppose that there are T marbles of color 1 and $B-T$ marbles of color 2 in an urn, where $B \leq n$ is even. There exists an algorithm that samples $\langle s_1, s_2 \rangle$, the number of marbles of each color appearing when drawing $B/2$ marbles from the urn without replacement, in $O(\operatorname{poly}(\log n))$ time and random words. Specifically, the probability of sampling a specific pair $\langle s_1, s_2 \rangle$ where $s_1 + s_2 = T$ is approximately $\binom{B/2}{s_1}\binom{B/2}{T-s_1}/\binom{B}{T}$ with error of at most n^{-c} for any constant $c > 0$.*

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

Lemma 5.4.6. *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 $O(\operatorname{poly}(\log n))$ time and random words.*

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

- If $\ell \leq B/2$, generate $\mathbf{C}' = \mathbf{S}_{B/2}^{\mathbf{C}}$ using Lemma 5.4.5.
  - 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 5.4.7. *Given B marbles of r different colors, such that there are C_i marbles of color i, there exists an algorithm that samples $\langle s_1, s_2, \cdots, s_r \rangle$, the number of marbles of each color appearing when drawing l marbles from the urn without replacement, in $O(r \cdot \operatorname{poly}(\log n))$ time and random words.*

Proof. Observe that we may reduce $r > 2$ to the two-color case by sampling the number of marbles of the first color, collapsing the rest of the colors together. Namely, define a pair $\hat{\mathbf{C}} = \langle C_1, C_2 + \cdots + C_r \rangle$, then generate $\mathbf{S}_\ell^{\hat{\mathbf{C}}} = \langle s_1, s_2 + \ldots + s_r \rangle$ via the above procedure. At this point we have obtained the first entry s_1 of the desired $\mathbf{S}_\ell^{\mathbf{C}}$. So it remains to generate the number of marbles of each color from the remaining $r - 1$ colors in $\ell - s_1$ remaining draws. In total, we may generate $\mathbf{S}_\ell^{\mathbf{C}}$ by performing r iterations of the two-colored case. The error in the L_1-distance may be established similarly to the proof of Lemma 5.3.10. □

5.4.2.2 Data structure

We now show that Theorem 5.4.7 may be used in order to create the following data structure. Recall that R denote the given distribution over integers $[r]$ (namely, the random distribution of communities for each vertex). Our data structure generates and maintains random variables $X_1, \ldots, X_n$, each of which is drawn independently at random from R: X_i denotes the community of vertex i. Then given a pair (i, j), it returns the vector $\mathbf{C}(i, j) = \langle c_1, \ldots, c_r \rangle$ where c_k counts the number of variables $X_i, \ldots, X_j$ that takes on the value k. Note that we may also find out X_i by probeing for (i, i) and take the corresponding index.

We maintain a complete binary tree whose leaves corresponds to indices from $[n]$. Each node represents a range and stores the vector $\mathbf{C}$ for the corresponding range. The root represents the entire range $[n]$, which is then halved in each level. Initially the root samples $\mathbf{C}(1, n)$ from the multinomial distribution according to R (see e.g., Section 3.4.1 of [Knu98]). Then, the children are generated on-the-fly using the lemma above. Thus, each probe can be processed within $O(r \operatorname{poly}(\log n))$ time, yielding Theorem 5.4.3. Then, by embedding the information stored by the data structure into the state (as in the proof of Lemma 5.3.10), we obtain the desired Corollary 5.4.4.

5.5 Local-Access Generators for Random Directed Graphs

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

Refer to Section 5.2.2 for specifications of this random graph model.

5.5.1 Generator for $c = 1$

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

Since the degree of v is small, in this model we can afford to perform ALL-NEIGHBORS probes instead of NEXT-NEIGHBOR probes using an additional $\text{poly}(\log n)$ resources.

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

5.5.1.1 Phase 1: Sample the distance D

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

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

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

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

5.5.1.2 Phase 2: Sampling neighbors at distance D

After sampling a distance D, we now have to sample all the neighbors at distance D. We label the vertices in Γ_D^v with unique indices in $\{1, \ldots, 4D\}$. Note that now each of the $4D$ vertices in Γ_D^v is a neighbor with probability $1/D^2$. However, by Phase 1, this is conditioned on the fact that there is at least one neighbor among the vertices in Γ_D^v, which may be difficult to sample when $1/D^2$ is very small. We can emulate this naïvely by repeatedly sampling a "block", composing of the $4D$ vertices in Γ_D^v, by deciding whether each vertex is a neighbor of v with uniform probability $1/D^2$

(i.e., $4D$ identical independent Bernoulli trials), and then discarding the entire block if it contains no neighbor. We repeat this process until we finally sample one block that contains at least one neighbor, and use this block as our output.

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

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

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

Theorem 5.5.1. *There exists an algorithm that generates a random graph from Kleinberg's Small World model, where probability of including each directed edge (u, v) in the graph is $1/(\mathrm{dist}(u, v))^2$ where* dist *denote the Manhattan distance, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *probe with high probability.*

5.5.2 Generator for $c \neq 1$

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

5.5.2.1 Case $c < 1$

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

Lemma 5.5.2. *Given a process $\mathcal{P}$ outputting the indices of successful Bernoulli trials with bias $\langle p_i \rangle$, there exists a process $\mathcal{P}^c$ outputting the indices of successful Bernoulli trials with bias $\langle c \cdot p_i \rangle$ where $c < 1$, using one additional N-bit word overhead for each answer of $\mathcal{P}$.*

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

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

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

Corollary 5.5.3. *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities $c/(\mathrm{dist}(u,v))^2$ where $c < 1$, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *probe with high probability.*

5.5.2.2 Case $c > 1$

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

Lemma 5.5.4. *Let $k > 1$ be a constant integer. Given a process $\mathcal{P}$ outputting the indices of successful Bernoulli trials with bias $\langle p_i \rangle$, there exists a process $\mathcal{P}^c$ outputting the indices of successful Bernoulli trials with bias $\langle \min\{c \cdot p_i, 1\} \rangle$ where $c > 1$ and $c \cdot p_i \leq 1 - \frac{1}{k}$ for every i, using one additional N-bit word overhead for each answer of $k \cdot c$ independent copies of $\mathcal{P}$.*

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

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

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

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

Corollary 5.5.5. *There exists an algorithm that generates a random graph from Kleinberg's Small World model with edge probabilities $c/(\mathrm{dist}(u,v))^2$ where $c = \mathrm{poly}(\log n)$, using $O(\log^2 n)$ time and random N-bit words per* ALL-NEIGHBORS *probe with high probability.*

5.6 Further Analysis and Extensions of Algorithm 5-2

5.6.1 Performance guarantee

This section is devoted to showing the following lemma that bounds the required resources per probe of Algorithm 5-2. We note that we only require efficient computation of $\prod_{u\in[a,b]}(1-p_{v,u})$ (and not $\sum_{u\in[a,b]} p_{v,u}$), and that for the $G(n,p)$ model, the resources required for such computation is asymptotically negligible.

Theorem 5.6.1. *Each execution of Algorithm 5-2 (the* NEXT-NEIGHBOR *probe), with high probability,*

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

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

Specifying random choices. The performance of the algorithm depends on not only the random variables $X_{v,u}$'s, but also the unused coins $C_{v,u}$'s. We characterize the two collections of Bernoulli variables $\{X_{v,u}\}$ and $\{Y_{v,u}\}$ that cover all random choices made by Algorithm 5-2 as follows.

- Each $X_{v,u}$ (same as $X_{u,v}$) represents the result for the *first* coin-toss corresponding to cells $\mathbf{A}[v][u]$ and $\mathbf{A}[u][v]$, which is the coin-toss obtained when $X_{v,u}$ becomes decided: either $C_{v,u}$ during a NEXT-NEIGHBOR(v) call when $\mathbf{A}[v][u] = \phi$, or $C_{v,u}$ during a NEXT-NEIGHBOR(u) call when $\mathbf{A}[u][v] = \phi$, whichever occurs first. This description of $X_{v,u}$ respects our invariant that, if the generation process is executed to completion, we will have $\mathbf{A}[v][u] = X_{v,u}$ in all entries.

- Each $Y_{v,u}$ represents the result for the *second* coin-toss corresponding to cell $\mathbf{A}[v][u]$, which is the coin-toss $C_{v,u}$ obtained during a NEXT-NEIGHBOR(v) call when $X_{v,u}$ is already decided. In other words, $\{Y_{v,u}\}$'s are the coin-tosses that should have been skipped but still performed in Algorithm 5-2 (if they have indeed been generated). Unlike the previous case, $Y_{v,u}$ and $Y_{u,v}$ are two independent random variables: they may be generated during a NEXT-NEIGHBOR(v) call and a NEXT-NEIGHBOR(u) call, respectively.

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

Characterizing iterations. Suppose that we compute NEXT-NEIGHBOR(v) and obtain an answer u. Then $X_{v,\mathbf{last}[v]+1} = \cdots = X_{v,u-1} = 0$ as none of $u' \in (\mathbf{last}[v], u)$ is a neighbor of v. The vertices

considered in the loop of Algorithm 5-2 that do not result in the answer u, are $u' \in (\mathbf{last}[v], u)$ satisfying $\mathbf{A}[v][u'] = 0$ and $Y_{v,u'} = 1$; we call the iteration corresponding to such a u' a *failed iteration.* Observe that if $X_{v,u'} = 0$ but is undecided ($\mathbf{A}[v][u'] = \phi$), then the iteration is not failed, even if $Y_{v,u'} = 1$ (in which case, $X_{v,u'}$ takes the value of $C_{v,u'}$ while $Y_{v,u'}$ is never used). Thus we assume the worst-case scenario where all $X_{v,u'}$ are revealed: $\mathbf{A}[v][u'] = X_{v,u'} = 0$ for all $u' \in (\mathbf{last}[v], u)$. The number of failed iterations in this case stochastically dominates those in all other cases.[9]

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

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

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

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

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

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

Establishing the overall performance guarantee. So far we have deduced that, for each pair of a vertex v and its $\mathbf{last}[v]$, the probability that the call NEXT-NEIGHBOR(v) encounters more than r failed iterations is less that $n \cdot 2^{-r}$, which is at most n^{-c-2} for any desired constant c by choosing a

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

sufficiently large $r = \Theta(\log n)$. As Algorithm 5-2 may need to support up to $\Theta(n^2)$ NEXT-NEIGHBOR calls, one corresponding to each pair $(v, \mathbf{last}[v])$, the probability that it ever encounters more than $O(\log n)$ failed iterations to answer a single NEXT-NEIGHBOR probe is at most n^{-c}. That is, with high probability, $O(\log n)$ iterations are required per NEXT-NEIGHBOR call, which concludes the proof of Theorem 5.6.1. □

5.6.2 Supporting VERTEX-PAIR probes

We extend our generator (Algorithm 5-2) to support the VERTEX-PAIR probes: given a pair of vertices (u, v), decide whether there exists an edge $\{u, v\}$ in the generated graph. To answer a VERTEX-PAIR probe, we must first check whether the value $X_{u,v}$ for $\{u, v\}$ has already been assigned, in which case we answer accordingly. Otherwise, we must make a coin-flip with the corresponding bias $p_{u,v}$ to assign $X_{u,v}$, deciding whether $\{u, v\}$ exists in the generated graph. If we maintained the full $\mathbf{A}$ as done in the naïve Algorithm 5-1, we would have been able to simply set $\mathbf{A}[u][v]$ and $\mathbf{A}[v][u]$ to this new value. However, our more efficient Algorithm 5-2 that represents $\mathbf{A}$ compactly via $\mathbf{last}$ and P_v's cannot record arbitrary modifications to $\mathbf{A}$.

Observe that if we were to apply the trivial implementation of VERTEX-PAIR in Algorithm 5-1, then by Lemma 5.3.1, $\mathbf{last}$ and P_v's will only fail capture the state $\mathbf{A}[v][u] = 0$ when $u > \mathbf{last}[v]$ and $v > \mathbf{last}[u]$. Fortunately, unlike NEXT-NEIGHBOR probes, a VERTEX-PAIR probe can only set one cell $\mathbf{A}[v][u]$ to 0 per probe, and thus we may afford to store these changes explicitly.[10] To this end, we define the set $Q = \{\{u, v\} : X_{u,v} \text{ is assigned to } 0 \text{ during a VERTEX-PAIR probe}\}$, maintained as a hash table. Updating Q during VERTEX-PAIR probes is trivial: we simply add $\{u, v\}$ to Q before we finish processing the probe if we set $\mathbf{A}[u][v] = 0$. Conversely, we need to add u to P_v and add v to P_u if the VERTEX-PAIR probe sets $\mathbf{A}[u][v] = 1$ as usual, yielding the following observation. It is straightforward to verify that each VERTEX-PAIR probe requires $O(\log n)$ time, $O(1)$ random N-bit word, and $O(1)$ additional space per probe.

Lemma 5.6.2. *The data structures* $\mathbf{last}$, P_v*'s and* Q *together provide a succinct representation of* $\mathbf{A}$ *when* NEXT-NEIGHBOR *probes (modified Algorithm 5-2) and* VERTEX-PAIR *probes (modified Algorithm 5-1) are allowed. In particular,* $\mathbf{A}[v][u] = 1$ *if and only if* $u \in P_v$*. Otherwise,* $\mathbf{A}[v][u] = 0$ *if* $u < \mathbf{last}[v]$*,* $v < \mathbf{last}[u]$*, or* $\{v, u\} \in Q$*. In all remaining cases,* $\mathbf{A}[v][u] = \phi$*.*

We now explain other necessary changes to Algorithm 5-2. In the implementation of NEXT-NEIGHBOR, an iteration is not failed when the chosen $X_{v,u}$ is still undecided: $\mathbf{A}[v][u]$ must still be ϕ. Since $X_{v,u}$ may also be assigned to 0 via a VERTEX-PAIR(v, u) probe, we must also consider an iteration where $\{v, u\} \in Q$ failed. That is, we now require one additional condition $\{v, u\} \notin Q$ for termination (which only takes $O(1)$ time to verify per iteration). As for the analysis, aside from handling the fact that $X_{v,u}$ may also become decided during a VERTEX-PAIR call, and allowing the states of the algorithm to support VERTEX-PAIR probes, all of the remaining analysis for correctness and performance guarantee still holds.

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

Therefore, we have established that our augmentation to Algorithm 5-2 still maintains all of its (asymptotic) performance guarantees for NEXT-NEIGHBOR probes, and supports VERTEX-PAIR probes with complexities as specified above, concluding the following corollary. We remark that, as we do not aim to support RANDOM-NEIGHBOR probes, this simple algorithm here provides significant improvement over the performance of RANDOM-NEIGHBOR probes (given in Corollary 5.3.8).

Corollary 5.6.3. *Algorithm 5-2 can be modified to allow an implementation of* VERTEX-PAIR *probe as explained above, such that the resource usages per probe still asymptotically follow those of Theorem 5.6.1.*

5.7 Alternative Generator with Deterministic Performance Guarantee

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

In what follows, we first focus on the $G(n,p)$ model, starting with NEXT-NEIGHBOR probes (Section 5.7.1) then extend to VERTEX-PAIR probes (Section 5.7.2. We then explain how this result may be generalized to support the Stochastic Block model with random community assignment in Section 5.7.3.

5.7.1 Data structure for NEXT-NEIGHBOR probes for $G(n,p)$

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

Figure 5-5: Alternative generator

Recall that NEXT-NEIGHBOR(v) is given by $\min\{u > \mathbf{last}[v] : X_{v,u} = 1\}$ (or $n+1$ if no satisfying

u exists). To aid in computing this quantity, we define:

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

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

Unlike the approach of Algorithm 5-2 that simulates coin-flips even for decided $X_{v,u}$'s, here we only flip undecided coins for the indices in T_v: we have $|T_v|$ Bernoulli trials to simulate. Let F be the random variable denoting the first index of a successful trial out of $|T_v|$ coin-flips, or $|T_v|+1$ if all fail; denote the distribution of F by $\mathsf{ExactF}(p, |T|)$. The CDF of F is given by $\mathbb{P}[F = f] = 1 - (1-p)^f$ for $f \leq |T_v|$ (i.e., there is some success trial in the first f trials), and $\mathbb{P}[F = |T_v| + 1] = 1$. Thus, we must design a data structure that can compute w_v, compute $|T_v|$, find the F^{th} minimum value in T_v, and update $\mathbf{A}[v][u]$ for the F lowest values $u \in T_v$ accordingly.

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

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

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

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

- PICK(v, F): find the F^{th} minimum value in T_v (assuming $F \leq |T_v|$).
  We again break $(\mathbf{last}[v], w_v)$ into $O(\log n)$ canonical ranges $[a_i, b_i]$'s, compute $t_{[a_i,b_i]}$'s, and

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

- UPDATE(v, u): simulate coin-flips, assigning $X_{v,u} \leftarrow 1$, and $X_{v,u'} \leftarrow 0$ for $u' \in (\mathbf{last}[v], u) \cap T_v$. This is done implicitly by handling the change $\mathbf{last}[v] \leftarrow u$: for each BBST $\mathbf{B}_{[a,b]}$ where $v \in [a, b]$, remove the old value of $\mathbf{last}[v]$ and insert u instead.

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

5.7.2 Data structure for VERTEX-PAIR probes for $G(n, p)$

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

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

This definition differs from that of Q in Section 5.6.2 in two aspects. First, we ensure that each $\mathbf{A}[v][u] = 0$ is recorded by either **last** (via Lemma 5.3.1) or Q'_v (explicitly), but *not both*. In particular, if u were to stay in Q'_v when $\mathbf{last}[v]$ increases beyond u, we would have double-counted these entries 0 not only recorded by Q'_v but also implied by $\mathbf{last}[v]$ and K_v. By having a BBST for each Q'_v, we can compute the number of 0's that must be excluded from T_v, which cannot be determined via $\mathbf{last}[v]$ and K_v alone: we subtract these from any counting process done in the data structure $\mathbf{R}$.

Second, we maintain Q'_v separately for each v as an ordered set, so that we may identify non-neighbors of v within a specific range – this allows us to remove non-neighbors in specific range, ensuring that the first aspect holds. More specifically, when we increase $\mathbf{last}[v]$, we must go through the data structure Q'_v and remove all $u < \mathbf{last}[v]$, and for each such u, also remove v from Q'_u. There can be as many as linear number of such u, but the number of removals is trivially bounded by the number of insertions, yielding an amortized time performance guarantee in the following theorem. Aside from the deterministic guarantee, unsurprisingly, the required amount of random words for this algorithm is lower than that of the algorithm from Section 5.6 (given in Theorem 5.6.1 and Corollary 5.6.3).

Theorem 5.7.1. *Consider the Erdős-Rényi $G(n, p)$ model. For* NEXT-NEIGHBOR *probes only, Algorithm 5-5 is a generator that answers each probe using $O(\log^2 n)$ time, $O(\log n)$ additional space,*

and one N-bit random word. For NEXT-NEIGHBOR *and* VERTEX PAIR *probes, an extension of Algorithm 5-5 answers each probe using $O(\log^2 n)$ amortized time, $O(\log n)$ additional space, and one N-bit random word.*

5.7.3 Data structure for the stochastic block model

We employ the data structure for generating and counting the number of vertices of each community in a specified range from Section 5.4.2. We create r different copies of the data structure $\mathbf{R}$ and Q'_v, one for each community, so that we may implement the required operations separately for each color, including using the COUNT subroutine to sample $F \sim \mathsf{ExactF}$ via the corresponding CDF, and picking the next neighbor according to F. Recall that since we do not store $\mathbf{last}[v] = 0$ in $\mathbf{R}$, and we only add an entry to K_v, P_v or Q'_v after drawing the corresponding $X_{u,v}$, the communities of the endpoints, which cover all elements stored in these data structures, must have already been determined. Thus, we obtain the following corollary for the Stochastic Block model.

Corollary 5.7.2. *Consider the Stochastic Block model with randomly-assigned communities. For* NEXT-NEIGHBOR *probes only, Algorithm 5-5 is a generator that answers each probe using $O(r \operatorname{poly}(\log n))$ time, random words, and additional space per probe (in the worst case). For* NEXT-NEIGHBOR *and* VERTEX-PAIR *probes, Algorithm 5-5 answers each probe using $O(r \operatorname{poly}(\log n))$ amortized time, $O(r \operatorname{poly}(\log n))$ random words, and $O(r \operatorname{poly}(\log n))$ additional space per probe additional space, and one N-bit random word.*

Bibliography

[Abb18] Emmanuel Abbe. Community detection and stochastic block models. *Foundations and Trends in Communications and Information Theory*, 14(1-2):1–162, 2018.

[ABH16] Emmanuel Abbe, Afonso S. Bandeira, and Georgina Hall. Exact recovery in the stochastic block model. *IEEE Trans. Information Theory*, 62(1):471–487, 2016.

[AG13] Kook Jin Ahn and Sudipto Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. *Inf. Comput.*, 222:59–79, 2013.

[AG15] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 202–211, 2015.

[AGM12] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 5–14, 2012.

[AHK12] Sanjeev Arora, Elad Hazan, and Satyen Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[AK17] Md. Maksudul Alam and Maleq Khan. Parallel algorithms for generating random networks with given degree sequences. *International Journal of Parallel Programming*, 45(1):109–127, 2017.

[AKL16] Sepehr Assadi, Sanjeev Khanna, and Yang Li. Tight bounds for single-pass streaming complexity of the set cover problem. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 698–711, 2016.

[AMS06] Noga Alon, Dana Moshkovitz, and Shmuel Safra. Algorithmic construction of sets for k-restrictions. *ACM Trans. Algorithms*, 2(2):153–177, 2006.

[AO15a] Zeyuan Allen Zhu and Lorenzo Orecchia. Nearly-linear time positive LP solver with faster convergence rate. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 229–236, 2015.

[AO15b] Zeyuan Allen Zhu and Lorenzo Orecchia. Using optimization to break the epsilon barrier: A faster and simpler width-independent algorithm for solving positive linear

programs in parallel. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1439–1456, 2015.

[AP90] Baruch Awerbuch and David Peleg. Network synchronization with polylogarithmic overhead. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume II*, pages 514–522, 1990.

[AP92] Baruch Awerbuch and David Peleg. Routing with polynomial communication-space trade-off. *SIAM J. Discrete Math.*, 5(2):151–162, 1992.

[ARVX12] Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1132–1139, 2012.

[AS15] Emmanuel Abbe and Colin Sandon. Community detection in general stochastic block models: Fundamental limits and efficient algorithms for recovery. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 670–688, 2015.

[Ass17] Sepehr Assadi. Tight space-approximation tradeoff for the multi-pass streaming set cover problem. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 321–335, 2017.

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

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

[BCS09] Nikhil Bansal, Alberto Caprara, and Maxim Sviridenko. A new approximation method for set covering problems, with applications to multidimensional bin packing. *SIAM J. Comput.*, 39(4):1256–1278, 2009.

[BEM16] MohammadHossein Bateni, Hossein Esfandiari, and Vahab S. Mirrokni. Distributed coverage maximization via sketching. *CoRR*, abs/1612.02327, 2016.

[BEM17] MohammadHossein Bateni, Hossein Esfandiari, and Vahab S. Mirrokni. Almost optimal streaming algorithms for coverage problems. In *Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2017, Washington DC, USA, July 24-26, 2017*, pages 13–23, 2017.

[BGM14] Bahman Bahmani, Ashish Goel, and Kamesh Munagala. Efficient primal-dual graph algorithms for mapreduce. In *Algorithms and Models for the Web Graph - 11th International Workshop, WAW 2014, Beijing, China, December 17-18, 2014, Proceedings*, pages 59–78, 2014.

[BHNT15] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on

one-pass dynamic streams. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 173–182, 2015.

[BK10] Mickey Brautbar and Michael J. Kearns. Local algorithms for finding interesting individuals in large networks. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 5-7, 2010. Proceedings*, pages 188–199, 2010.

[BK16] Greg Bodwin and Sebastian Krinninger. Fully dynamic spanners with worst-case update time. In *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, pages 17:1–17:18, 2016.

[BKS12] Surender Baswana, Sumeet Khurana, and Soumojit Sarkar. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Algorithms*, 8(4):35:1–35:51, 2012.

[BMKK14] Ashwinkumar Badanidiyuru, Baharan Mirzasoleiman, Amin Karbasi, and Andreas Krause. Streaming submodular maximization: massive data summarization on the fly. In *The 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, New York, NY, USA - August 24 - 27, 2014*, pages 671–680, 2014.

[BNS+16] Raef Bassily, Kobbi Nissim, Adam D. Smith, Thomas Steinke, Uri Stemmer, and Jonathan Ullman. Algorithmic stability for adaptive data analysis. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 1046–1059, 2016.

[Bol01] Béla Bollobás. *Random Graphs*. Number 73. Cambridge University Press, 2001.

[BS07] Surender Baswana and Sandeep Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Struct. Algorithms*, 30(4):532–563, 2007.

[CAT16] Irineo Cabreros, Emmanuel Abbe, and Aristotelis Tsirigos. Detecting community structures in hi-c genomic data. In *2016 Annual Conference on Information Science and Systems, CISS 2016, Princeton, NJ, USA, March 16-18, 2016*, pages 584–589, 2016.

[CGQ15] Chandra Chekuri, Shalmoli Gupta, and Kent Quanrud. Streaming algorithms for submodular function maximization. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, pages 318–330, 2015.

[CKT10] Flavio Chierichetti, Ravi Kumar, and Andrew Tomkins. Max-cover in map-reduce. In *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 231–240, 2010.

[CKW10] Graham Cormode, Howard J. Karloff, and Anthony Wirth. Set cover algorithms for very large datasets. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, pages 479–488, 2010.

[CPS17] Keren Censor-Hillel, Merav Parter, and Gregory Schwartzman. Derandomizing local distributed algorithms under bandwidth restrictions. In *31st International Symposium*

on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria, pages 11:1–11:16, 2017.

[CRT05] Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM J. Comput.*, 34(6):1370–1379, 2005.

[CRV15] Peter Chin, Anup Rao, and Van Vu. Stochastic block model and community detection in sparse graphs: A spectral algorithm with optimal rate of recovery. In *Proceedings of The 28th Conference on Learning Theory, COLT 2015, Paris, France, July 3-6, 2015*, pages 391–423, 2015.

[CSC+07] Melissa S Cline, Michael Smoot, Ethan Cerami, Allan Kuchinsky, Nerius Landys, Chris Workman, Rowan Christmas, Iliana Avila-Campilo, Michael Creech, Benjamin Gross, et al. Integration of biological networks and gene expression data using cytoscape. *Nature protocols*, 2(10):2366, 2007.

[CW16] Amit Chakrabarti and Anthony Wirth. Incidence geometries and the pass complexity of semi-streaming set cover. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1365–1373, 2016.

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

[DFH+15] Cynthia Dwork, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Leon Roth. Preserving statistical validity in adaptive data analysis. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 117–126, 2015.

[DG08] Bilel Derbel and Cyril Gavoille. Fast deterministic distributed algorithms for sparse spanners. *Theor. Comput. Sci.*, 399(1-2):83–100, 2008.

[DGP07] Bilel Derbel, Cyril Gavoille, and David Peleg. Deterministic distributed construction of linear stretch spanners in polylogarithmic time. In *Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings*, pages 179–192, 2007.

[DGPV08] Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. On the locality of distributed sparse spanner construction. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 2008, Toronto, Canada, August 18-21, 2008*, pages 273–282, 2008.

[DGPV09] Bilel Derbel, Cyril Gavoille, David Peleg, and Laurent Viennot. Local computation of nearly additive spanners. In *Distributed Computing, 23rd International Symposium, DISC 2009, Elche, Spain, September 23-25, 2009. Proceedings*, pages 176–190, 2009.

[DIMV14] Erik D. Demaine, Piotr Indyk, Sepideh Mahabadi, and Ali Vakilian. On streaming and communication complexity of the set cover problem. In *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, pages 484–498, 2014.

[DKM05] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. Sampling sub-problems of heterogeneous max-cut problems and approximation algorithms. In *STACS 2005, 22nd Annual Symposium on Theoretical Aspects of Computer Science, Stuttgart, Germany, February 24-26, 2005, Proceedings*, pages 57–68, 2005.

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

[DS14] Irit Dinur and David Steurer. Analytical approach to parallel repetition. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 624–633, 2014.

[EK10] David A. Easley and Jon M. Kleinberg. *Networks, Crowds, and Markets - Reasoning About a Highly Connected World.* Cambridge University Press, 2010.

[Elk11] Michael Elkin. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms*, 7(2):20:1–20:17, 2011.

[ELMR16] Guy Even, Reut Levi, Moti Medina, and Adi Rosén. Sublinear random access generators for preferential attachment graphs. *CoRR*, abs/1602.06159, 2016.

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

[EMR14] Guy Even, Moti Medina, and Dana Ron. Deterministic stateless centralized local algorithms for bounded degree graphs. In *Algorithms - ESA 2014 - 22th Annual European Symposium, Wroclaw, Poland, September 8-10, 2014. Proceedings*, pages 394–405, 2014.

[EN17] Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 652–669, 2017.

[ER60] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. In *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, volume 5, pages 17–61, 1960.

[ER16] Yuval Emek and Adi Rosén. Semi-streaming set cover. *ACM Trans. Algorithms*, 13(1):6:1–6:22, 2016.

[Erd65] Paul Erdős. On some extremal problems in graph theory. *Israel Journal of Mathematics*, 3(2):113–116, 1965.

[Fei98] Uriel Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[For10] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[FPV18] Uriel Feige, Boaz Patt-Shamir, and Shai Vardi. On the probe complexity of local computation algorithms. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 50:1–50:14, 2018.

[GGN03] Oded Goldreich, Shafi Goldwasser, and Asaf Nussboim. On the implementation of huge random objects. In *44th Symposium on Foundations of Computer Science (FOCS 2003), 11-14 October 2003, Cambridge, MA, USA, Proceedings*, pages 68–79, 2003.

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

[GGR98] Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, 1998.

[GK95] Michael D. Grigoriadis and Leonid G. Khachiyan. A sublinear-time randomized approximation algorithm for matrix games. *Oper. Res. Lett.*, 18(2):53–58, 1995.

[GK07] Naveen Garg and Jochen Könemann. Faster and simpler algorithms for multicommodity flow and other fractional packing problems. *SIAM J. Comput.*, 37(2):630–652, 2007.

[GKK13] Ashish Goel, Michael Kapralov, and Sanjeev Khanna. Perfect matchings in o(nlog n) time in regular bipartite graphs. *SIAM J. Comput.*, 42(3):1392–1404, 2013.

[Gol11] Oded Goldreich. A brief introduction to property testing. In *Studies in Complexity and Cryptography. Miscellanea on the Interplay between Randomness and Computation - In Collaboration with Lidor Avigad, Mihir Bellare, Zvika Brakerski, Shafi Goldwasser, Shai Halevi, Tali Kaufman, Leonid Levin, Noam Nisan, Dana Ron, Madhu Sudan, Luca Trevisan, Salil Vadhan, Avi Wigderson, David Zuckerman*, pages 465–469. 2011.

[GR02] Oded Goldreich and Dana Ron. Property testing in bounded degree graphs. *Algorithmica*, 32(2):302–343, 2002.

[GW97] Tal Grossman and Avishai Wool. Computational experience with approximation algorithms for the set covering problem. *European Journal of Operational Research*, 101(1):81 – 92, 1997.

[HIMV16] Sariel Har-Peled, Piotr Indyk, Sepideh Mahabadi, and Ali Vakilian. Towards tight bounds for the streaming set cover problem. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 371–383, 2016.

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

[IMR+17] Piotr Indyk, Sepideh Mahabadi, Ronitt Rubinfeld, Jonathan Ullman, Ali Vakilian, and Anak Yodpinyanee. Fractional set cover in the streaming model. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2017, August 16-18, 2017, Berkeley, CA, USA*, pages 12:1–12:20, 2017.

[IMR+18] Piotr Indyk, Sepideh Mahabadi, Ronitt Rubinfeld, Ali Vakilian, and Anak Yodpinyanee. Set cover in sub-linear time. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2467–2486, 2018.

[JTZ04] Daxin Jiang, Chun Tang, and Aidong Zhang. Cluster analysis for gene expression data: A survey. *IEEE Trans. Knowl. Data Eng.*, 16(11):1370–1386, 2004.

[KK82] Narendra Karmarkar and Richard M. Karp. An efficient approximation scheme for the one-dimensional bin-packing problem. In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 312–320, 1982.

[KKR04] Tali Kaufman, Michael Krivelevich, and Dana Ron. Tight bounds for testing bipartiteness in general graphs. *SIAM J. Comput.*, 33(6):1441–1483, 2004.

[Kle00] Jon M. Kleinberg. The small-world phenomenon: an algorithmic perspective. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, May 21-23, 2000, Portland, OR, USA*, pages 163–170, 2000.

[KMVV15] Ravi Kumar, Benjamin Moseley, Sergei Vassilvitskii, and Andrea Vattani. Fast greedy algorithms in mapreduce and streaming. *TOPC*, 2(3):14:1–14:22, 2015.

[KMW06] Fabian Kuhn, Thomas Moscibroda, and Roger Wattenhofer. The price of being near-sighted. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 980–989, 2006.

[Knu98] Donald Ervin Knuth. *The art of computer programming, Volume II: Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[KP12] Michael Kapralov and Rina Panigrahy. Spectral sparsification via random spanners. In *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 393–398, 2012.

[KV94] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory.* MIT Press, 1994.

[KW14] Michael Kapralov and David P. Woodruff. Spanners and sparsifiers in dynamic streams. In *ACM Symposium on Principles of Distributed Computing, PODC '14, Paris, France, July 15-18, 2014*, pages 272–281, 2014.

[KY14] Christos Koufogiannakis and Neal E. Young. A nearly linear-time PTAS for explicit fractional packing and covering linear programs. *Algorithmica*, 70(4):648–674, 2014.

[LL18] Christoph Lenzen and Reut Levi. A centralized local algorithm for the sparse spanning graph problem. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, pages 87:1–87:14, 2018.

[LMR+17] Reut Levi, Guy Moshkovitz, Dana Ron, Ronitt Rubinfeld, and Asaf Shapira. Constructing near spanning trees with few local inspections. *Random Struct. Algorithms*, 50(2):183–200, 2017.

[LR15] Reut Levi and Dana Ron. A quasi-polynomial time partition oracle for graphs with an excluded minor. *ACM Trans. Algorithms*, 11(3):24:1–24:13, 2015.

[LRR14] Reut Levi, Dana Ron, and Ronitt Rubinfeld. Local algorithms for sparse spanning graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2014, September 4-6, 2014, Barcelona, Spain*, pages 826–842, 2014.

[LRR16] Reut Levi, Dana Ron, and Ronitt Rubinfeld. A local algorithm for constructing spanners in minor-free graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2016, September 7-9, 2016, Paris, France*, pages 38:1–38:15, 2016.

[LRY17] Reut Levi, Ronitt Rubinfeld, and Anak Yodpinyanee. Local computation algorithms for graphs of non-constant degrees. *Algorithmica*, 77(4):971–994, 2017.

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in õ(sqrt(rank)) iterations and faster algorithms for maximum flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 424–433, 2014.

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

[LY94] Carsten Lund and Mihalis Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.

[MH11] Joel C. Miller and Aric A. Hagberg. Efficient generation of networks with given expected degrees. In *Algorithms and Models for the Web Graph - 8th International Workshop, WAW 2011, Atlanta, GA, USA, May 27-29, 2011. Proceedings*, pages 115–126, 2011.

[MKI+03] Ron Milo, Nadav Kashtan, Shalev Itzkovitz, Mark EJ Newman, and Uri Alon. On the uniform generation of random graphs with prescribed degree sequences. *arXiv preprint cond-mat/0312028*, 2003.

[MN04] Charles U. 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, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, pages 179–188, 2004.

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

[Mos15] Dana Moshkovitz. The projection games conjecture and the np-hardness of ln n-approximating set-cover. *Theory of Computing*, 11:221–235, 2015.

[MPN+99] Edward M Marcotte, Matteo Pellegrini, Ho-Leung Ng, Danny W Rice, Todd O Yeates, and David Eisenberg. Detecting protein function and protein-protein interactions from genome sequences. *Science*, 285(5428):751–753, 1999.

[MPV18] Yishay Mansour, Boaz Patt-Shamir, and Shai Vardi. Constant-time local computation algorithms. *Theory Comput. Syst.*, 62(2):249–267, 2018.

[MPVX15] Gary L. Miller, Richard Peng, Adrian Vladu, and Shen Chen Xu. Improved parallel algorithms for spanners and hopsets. In *Proceedings of the 27th ACM on Symposium on Parallelism in Algorithms and Architectures, SPAA 2015, Portland, OR, USA, June 13-15, 2015*, pages 192–201, 2015.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms.* Cambridge University Press, 1995.

[MR06] Sharon Marko and Dana Ron. Distance approximation in bounded-degree and general sparse graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 9th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems, APPROX 2006 and 10th International Workshop on Randomization and Computation, RANDOM 2006, Barcelona, Spain, August 28-30 2006, Proceedings*, pages 475–486. 2006.

[MRVX12] Yishay Mansour, Aviad Rubinstein, Shai Vardi, and Ning Xie. Converting online algorithms to local computation algorithms. In *Automata, Languages, and Programming - 39th International Colloquium, ICALP 2012, Warwick, UK, July 9-13, 2012, Proceedings, Part I*, pages 653–664, 2012.

[MV13] Yishay Mansour and Shai Vardi. A local computation approximation scheme to maximum matching. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques - 16th International Workshop, APPROX 2013, and 17th International Workshop, RANDOM 2013, Berkeley, CA, USA, August 21-23, 2013. Proceedings*, pages 260–273. 2013.

[MV17] Andrew McGregor and Hoa T. Vu. Better streaming algorithms for the maximum coverage problem. In *20th International Conference on Database Theory, ICDT 2017, March 21-24, 2017, Venice, Italy*, pages 22:1–22:18, 2017.

[MZ13] Claire Mathieu and Hang Zhou. Graph reconstruction via distance oracles. In *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part I*, pages 733–744, 2013.

[MZ15] Vahab S. Mirrokni and Morteza Zadimoghaddam. Randomized composable core-sets for distributed submodular maximization. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing, STOC 2015, Portland, OR, USA, June 14-17, 2015*, pages 153–162, 2015.

[New00] Mark EJ Newman. Models of the small world. *Journal of Statistical Physics*, 101(3-4):819–841, 2000.

[NLKB11] Sadegh Nobari, Xuesong Lu, Panagiotis Karras, and Stéphane Bressan. Fast random graph generation. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 331–342, 2011.

[NN07] Moni Naor and Asaf Nussboim. Implementing huge sparse random graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, 10th International Workshop, APPROX 2007, and 11th International Workshop, RANDOM 2007, Princeton, NJ, USA, August 20-22, 2007, Proceedings*, pages 596–608. 2007.

[NO08] Huy N. Nguyen and Krzysztof Onak. Constant-time approximation algorithms via local improvements. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 327–336, 2008.

[NWS02] Mark EJ Newman, Duncan J Watts, and Steven H Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Sciences*, 99(suppl 1):2566–2572, 2002.

[ORRR12] Krzysztof Onak, Dana Ron, Michal Rosen, and Ronitt Rubinfeld. A near-optimal sublinear-time algorithm for approximating the minimum vertex cover size. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1123–1131, 2012.

[Pel00] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* Society for Industrial and Applied Mathematics, 2000.

[Pet10] Seth Pettie. Distributed algorithms for ultrasparse spanners and linear size skeletons. *Distributed Computing*, 22(3):147–166, 2010.

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

[PS89] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[PST95] Serge A. Plotkin, David B. Shmoys, and Éva Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20(2):257–301, 1995.

[PU89] David Peleg and Jeffrey D. Ullman. An optimal synchronizer for the hypercube. *SIAM J. Comput.*, 18(4):740–747, 1989.

[RS97] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997*, pages 475–484, 1997.

[RTVX11] Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. In *Innovations in Computer Science - ICS 2010, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 223–238, 2011.

[RV16] Omer Reingold and Shai Vardi. New techniques and tighter bounds for local computation algorithms. *J. Comput. Syst. Sci.*, 82(7):1180–1200, 2016.

[SC11] Shaghayegh Sahebi and William Cohen. Community-based recommendations: a solution to the cold start problem. In *Workshop on Recommender Systems and the Social Web (RSWEB)*, page 60, 2011.

[SG09] Barna Saha and Lise Getoor. On maximum coverage in the streaming model & application to multi-topic blog-watch. In *Proceedings of the SIAM International Conference on Data Mining, SDM 2009, April 30 - May 2, 2009, Sparks, Nevada, USA*, pages 697–708, 2009.

[SM00] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 22(8):888–905, 2000.

[SPT+01] Therese Sørlie, Charles M. Perou, Robert Tibshirani, Turid Aas, Stephanie Geisler, Hilde Johnsen, Trevor Hastie, Michael B. Eisen, Matt van de Rijn, Stefanie S. Jeffrey, Thor Thorsen, Hanne Quist, John C. Matese, Patrick O. Brown, David Botstein, Per Eystein Lønning, and Anne-Lise Børresen-Dale. Gene expression patterns of breast carcinomas distinguish tumor subclasses with clinical implications. *Proceedings of the National Academy of Sciences*, 98(19):10869–10874, 2001.

[SSS95] Jeanette P. Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995.

[ST11] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM J. Comput.*, 40(4):981–1025, 2011.

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

[Vad12] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends in Theoretical Computer Science*, 7(1-3):1–336, 2012.

[Wen91] Rephael Wenger. Extremal graphs with no C^4's, C^6's, or C^{10}'s. *J. Comb. Theory, Ser. B*, 52(1):113–116, 1991.

[WRM16] Di Wang, Satish Rao, and Michael W. Mahoney. Unified acceleration method for packing and covering problems via diameter reduction. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 50:1–50:13, 2016.

[WS98] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393:440, Jun 1998.

[You95] Neal E. Young. Randomized rounding without solving the linear program. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1995. San Francisco, California, USA.*, pages 170–178, 1995.

[You01] Neal E. Young. Sequential and parallel algorithms for mixed packing and covering. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 538–546, 2001.

[You14] Neal E. Young. Nearly linear-time approximation schemes for mixed packing/covering and facility-location linear programs. *CoRR*, abs/1407.3015, 2014.

[YYI12] Yuichi Yoshida, Masaki Yamamoto, and Hiro Ito. Improved constant-time approximation algorithms for maximum matchings and other optimization problems. *SIAM J. Comput.*, 41(4):1074–1093, 2012.