Proceedings of Machine Learning Research vol 201:1–16, 2023 34th International Conference on Algorithmic Learning Theory

A Query Algorithm for Learning a Spanning Forest in Weighted Undirected Graphs

Deeparnab Chakrabarty DEEPARNAB@DARTMOUTH.EDU
Department of Computer Science, Dartmouth College

Hang Liao HANG.LIAO.GR@DARTMOUTH.EDU
Department of Computer Science, Dartmouth College

Editors: Shipra Agrawal and Francesco Orabona

Abstract

We consider the problem of finding a spanning forest in an unknown *weighted* undirected graph when the access to the graph is via CUT queries, that is, one can query a subset $S \subseteq V$ of vertices and get the cut-value $\sum_{e\in\partial S} w(e)$ as the response. It is not too hard to solve this problem using $O(n \log n)$ queries mimicking a Prim-style algorithm using a binary-search style idea. In this paper we use the power of CUT queries to obtain a Monte-Carlo algorithm that makes $O(n \log\log n(\log\log\log n)^2)$ CUT queries. At the core of our contribution is a generalization of a result in [Apers et al., 2022] which studies the same problem on unweighted graphs, but to handle weights, we need to combine their ideas with ideas for *support estimation* of weighted vectors, as in [Stockmeyer, 1983], and *weighted graph reconstruction* algorithms, as in [Bshouty and Mazzawi, 2012].

Keywords: Query Algorithms, Weighted Spanning Forest, Randomized Algorithms.

1. Introduction

Imagine $G = (V, E, w)$ is an undirected weighted graph whose vertex set V is known but the edge set and their weights $w(e) \geq 0$ are unknown. The only access we have to this graph is via certain kinds of queries. What all can be learned about this graph, and what is the query complexity of doing so? Motivated by various applications in fields such as genome sequencing (cf. (Grebinski and Kucherov, 2000)), and also due to close relations to questions in sketching and streaming, such questions have been intensively studied in the past couple of decades. While many previous works studied the question of *reconstructing* the whole graph itself, more recent works have focused on the question of determining certain properties of the graph rather than wholly reconstructing it.

In this paper we consider the following basic question: given CUT query access to this unknown weighted graph, in how few queries can you decide if G is connected, or more generally, find a spanning forest in G? A CUT query takes input a subset $S \subseteq V$ of vertices and outputs the *value* $\sum_{e\in\partial S} w(e)$. Here ∂S denotes the set of edges with exactly one endpoint in S. Apart from being a natural and fundamental graph algorithms question, finding a spanning forest arises as a first step in many procedures in network analysis, clustering, etc. Therefore, a spanning forest algorithm with small query complexity can have applications to these problems. The CUT query model is equivalent (up to constant factors) to other kinds of possibly more natural queries such as additive query model where one queries a subset and obtains the total weight of edges inside that subset. Furthermore, since the cut-function is submodular, determining whether a graph is connected is a (very) special case of symmetric submodular function minimization.

It is actually not too hard to mimic a Prim/DFS/BFS-style algorithm to give a deterministic $O(n \log n)$ query algorithm to find a spanning tree in an undirected weighted graph. Fascinatingly, a better deterministic algorithm is not known and neither is it ruled out, even for unweighted graphs. Very recently, on unweighted graphs, the paper Apers et al. (2022) gave a beautiful zero-error *randomized* algorithm for this problem which makes $O(n)$ queries in expectation. Indeed, the Prim-style deterministic algorithm alluded to above also works in a much weaker query model where one only gets to know if the cut value is zero or positive, and in this weaker model the "$n \log n$" is tight for deterministic algorithms. The result in Apers et al. (2022) really used the power that the CUT queries actually give the precise value.

When weights are present, the complexity of the problem changes. To illustrate this, consider the question of figuring out just the degree of single a vertex v, that is, the *number* of edges (v, x) incident on v. When the graph is unweighted, this can be found with a single CUT query on $\{v\}$. With weights, this problem is provably difficult: it can be shown (see Appendix C) that any deterministic algorithm must make $\Omega(n)$ queries. The algorithm in Apers et al. (2022) uses the degree information of all vertices crucially. Our main result is to overcome this difficulty and solve the spanning forest problem in undirected weighted graphs in much better than $O(n \log n)$ queries, however, we fall short of getting a clean linear dependence on the number of vertices. Furthermore, our algorithm is not a zero-error algorithm.

Theorem 1 *Given* CUT *query access to an unknown weighted undirected graph $G = (V, E, w)$ with non-negative weights, there is a polynomial time Monte-Carlo algorithm that makes $O(n \log\log n (\log\log\log n)^2)$ queries and returns a spanning forest in G with probability at least* $2/3$.

We believe that the triple log factors may be removable from the above expression, although it is not completely clear how to do so. However, we don't think our current techniques alone can help remove the $O(\log\log n)$ term. We believe that understanding whether an $O(n)$-query algorithm is possible is an important question: if indeed that is the case, then it would be due to a more refined use of the fact that we get the value of the cut, and not any coarser information. However, if there is indeed a lower bound of $\omega(n)$ for this problem, then that would be (perhaps more) interesting as it would be the first $\omega(n)$ query-lower bound for symmetric submodular function minimization. In particular, it would give us a new lower bounding technique which rules out algorithms using CUT queries. We elaborate on this in the next subsection where we describe our techniques.

1.1. Technical Overview

The general strategy for finding a spanning forest is the strategy behind Borůvka's algorithm: one proceeds in rounds where in a round each connected component tries to find (at least) one edge to a different connected component. This strategy is common to many different spanning forest algorithms in many different models; in particular, the paper of Apers et al. (2022) uses this strategy. There are two main subroutines.

1. We design a subroutine ReduceConnectedComponents (described in Algorithm 2) which is a randomized algorithm that takes input t connected components, and with probability $\geq 1 - \Theta(\frac{1}{\log\log n})$, finds new edges of the graph such that after adding them, the number of connected components becomes $\leq ct$ for some constant $c < 1$. The total number of queries

made is $O(t \log\log n \cdot (\log\log\log n)^2)$. Since the "$t$" is geometrically decaying, across all rounds we take the $O(n \log\log n (\log\log\log n)^2)$ queries as promised. We keep applying this till the number of components is $\leq \frac{n}{\log n}$, at which point

2. We apply DFSSpanningForest in Lemma 9 to recover the remaining edges in the spanning forest using $O(n)$ queries.

The second part is the Prim/DFS style algorithm and is quite straightforward.

For the first part, we are heavily inspired by the unweighted algorithm in Apers et al. (2022) and it is worthwhile to describe their idea. First it abstracts each connected component using a "representative", a vertex with at least a neighbor in a different connected component. Then it strives to sample a matching between the representatives. For a representative, the algorithm samples a random set of neighbors with the size inversely proportional to the degree of the representative. As noted before, the degree of each node can be found in 1 query per node in the unweighted case. After sampling, every node will have exactly one neighbor with $O(1)$ probability, and this can be used to obtain a subgraph on ct vertices between which the graph is a matching, for some $c < 1$. At this point, Apers et al. (2022) uses an algorithm by Grebinski and Kucherov (2000) which gives an $O(t)$ query algorithm to recover all the edges of this matching. Thus, using $O(t)$ queries the number of connected components diminishes by a multiplicative factor, and a geometric sum gives the $O(n)$ query algorithm.

Our algorithm mimics the same skeleton, except the bottleneck is that there is no way to tell the exact degree of a vertex efficiently if the graph is weighted. In fact, it may require $\Omega(n)$ queries to learn a vertex's exact degree (see Appendix C). We observe that one doesn't quite need the exact degree, but a constant factor approximation suffices. This is where we use the DegreeEstimation algorithm in Algorithm 1 initially proposed by Stockmeyer (1983) that estimates the degree of a vertex up to a constant factor with constant probability using $O(\log\log n \log\log\log n)$ queries. This is a Monte-Carlo algorithm, and we do not know how to obtain a zero-error version of the same. The DegreeEstimation does a binary search on $\log n$ candidates that costs $\log\log n$ queries. The additional multiplicative $\log\log\log n$ is needed as we want each comparison in our binary search to be correct with failure probability $O(\frac{1}{\log\log n})$ so that with union bound one can bound the error probability of the algorithm by a constant. In the actual proof, more care is needed since we actually need to bound the expected value of the ratio of the estimation over the actual degree. In fact, this algorithm doesn't use the full power of CUT queries, and works even when one only obtains the answer whether the cut value is zero or positive. And in that regime, the $\log\log n$ factor is unavoidable.

With approximated degrees, we are no longer guaranteed to find a perfect matching, but for most of the representatives we can ensure that a constant number of neighbors outside of its own connected components are sampled. This creates a rough matching, where each representative has a constant degree. Once the hidden edges are recovered, they ensure that number of graph's connected components decreases by a constant factor. The old matching recovery algorithm can't be applied on the rough matching. So instead, we use the deterministic $O(\frac{m \log n}{\log m} + m \log\log m)$ weighted graph recovery algorithm by Bshouty and Mazzawi (2012).

With constant probability, we are in the good scenario, where the sampled edges is proportional to the number of connected components t, and the representatives being an endpoint to at least one sampled edge is at least some constant times t. If this happens, we get a graph with ct connected components with $c < 1$ using $O(t \log\log n \log\log\log n)$ queries. Finally, note that we want to use ReduceConnectedComponents to reduce the number of connected components to at most $\frac{n}{\log n}$. That means we want the above subroutine to succeed $\Theta(\log\log n)$ times. Therefore, we want to boost the constant successful probability to $1 - O(\frac{1}{\log\log n})$, which gives an additional $\log\log\log n$ factor to the query complexity of ReduceConnectedComponents.

1.2. Related Works

There is a huge literature on graph problems in the query access model depending on the kind of queries that are allowed. For brevity's sake, we focus this section only on CUT queries, and on works most relevant to our work. There is a large body of work on graph *reconstruction* using CUT queries (cf. (Grebinski and Kucherov, 2000; Alon et al., 2004; Alon and Asodi, 2005; Reyzin and Srivastava, 2007; Bshouty and Mazzawi, 2012; Choi, 2013)) and this problem is almost solved. In unweighted graphs with m edges, there exists an algorithm Choi and Kim (2010) making $O(\frac{m\log\frac{m}{n}}{\log m})$ queries, and this is information theoretically tight up to constant factors. This algorithm is *non-adaptive* but not efficient in that the set of queries isn't explicit, and given the answers, the reconstruction algorithm is not efficient. There also exists an efficient deterministic algorithm due to Mazzawi (2010), but it runs in $O(\log n)$ rounds. These were followed up for algorithms for weighted graphs with Bshouty and Mazzawi (2012) giving the best deterministic algorithm (which we use), while Choi (2013) gives a randomized $O(\frac{m\log n}{\log m})$ query algorithm which is information theoretically tight as well.

The question of understanding the edge-connectivity (ie, minimum cut) of an undirected graph using CUT queries was perhaps explicitly initiated in the paper by Rubinstein et al. (2018) who described an $\tilde{O}(n)$ algorithm to determine the edge connectivity. The paper of Apers et al. (2022) that we have mentioned multiple times above in fact gives a randomized Monte Carlo $O(n)$ query algorithm to solve this problem. The problem of determining the edge connectivity in *weighted graphs* was studied in Mukhopadhyay and Nanongkai (2020) who gave a $\tilde{O}(n)$ query algorithm for the same. The particular question of deciding whether a graph is connected or not (or more generally finding spanning forests) using CUT-queries was studied in Assadi et al. (2021). This paper was focused on deterministic algorithms and round versus query-complexity trade-off, thus ignores the poly $\log$ factors in query-complexity that this paper improves upon.

1.3. Notations and Observations

We first define the CROSS queries used throughout this paper:

CROSS queries: Given a graph $G = (V, E)$. Let $A, B \subseteq V$ be two disjoint subsets. CROSS(A, B) returns the total weights of the edges that have an endpoint in A and the other in B, $\sum_{e\in E(A,B)} w(e)$, where $E(A, B)$ denote the collection of vertex pairs $(a, b) \in A \times B$ such that there is an edge between a and b.

Proposition 2 *A* CROSS *query can be simulated by a constant number of* CUT *queries, and vice versa.*

Proof It is trivial to simulate a CUT query using a CROSS query. To simulate a CROSS query using CUT queries, observe that CROSS$(A, B) = \frac{1}{2}($CUT$(A) +$ CUT$(B) -$ CUT$(A \cup B))$, where A, B are two arbitrary disjoint subsets of V. ■

From this point on, our query model sticks to CROSS queries because it is more intuitive and versatile when used in algorithm description and analysis.

The following lemma is used in the later analysis.

Proposition 3 *Let $t_1, t_2, \cdots, t_{\lceil \log n \rceil}$ be integers at least 2. If $\sum_{i=1}^{\lceil \log n \rceil} t_i = t$ where $\frac{n}{\log n} \le t \le n$, then $\sum_{i=1}^{\lceil \log n \rceil} \frac{t_i}{\log t_i} = O(\frac{t}{\log t})$.*

Proof See Appendix A. ■

Support of a vector v, or $\mathsf{supp}(v)$ in notation, is the number of non-zero entries of v.

All the logs in the paper are $\log_2$ by default.

2. Subroutines

In this section, we provide a few subroutines that we need for our final algorithm. These are mostly minor modifications of algorithms that have appeared in previous works. We also state and prove some extra properties that we need for our purpose.

The first subroutine is a degree estimation algorithm which, for any given vertex, returns an estimate of its degree. This is equivalent to finding the support of a non-negative vector, and this problem was studied in Stockmeyer (1983). We provide a version of the algorithm below both for completeness, and also to prove an extra property that we need. We note that the algorithm actually works with a much weaker query access than what we have: for any subset S, we only need to know if its weight $w(S) > 0$ or not, that is, if S intersects the support of the vector or not. This query is called an OR query in the literature.

Given a parameter i, an (i,t)-test samples t independent random subsets $S_1, \ldots, S_t$ where in each set every element is present independently with probability $1/2^i$. We query $w(S_j)$ and use f_i to denote the fraction of samples with $w(S_j) > 0$. Below, C is a large constant which can be thought of as 10^4, but we don't optimize this.

Lemma 4 *(Paraphrasing and modifying Stockmeyer, 1983, Theorem 2.1). Given a vector $v \in (\mathbb{R}_0^+)^n$, the algorithm* $\mathsf{DegreeEstimation}$ *makes $O(\log\log n \log\log\log n)$ queries and returns an estimate $\mathsf{est}(v)$ such that (i) $\frac{\mathsf{supp}(v)}{2} \le \mathsf{est}(v) \le 2\mathsf{supp}(v)$ holds with probability 0.99, and (ii) $\frac{2}{5\mathsf{supp}(v)} < \mathbf{E}[\frac{1}{\mathsf{est}(v)} \mid \text{not FAIL}] < \frac{3}{\mathsf{supp}(v)}$.*

Proof The idea of the algorithm is similar to the algorithms for probabilistic counting. It tries to find the *scale* i such that $\mathsf{supp}(v) \in [2^i, 2^{i+1}]$. Given such a guess i, it samples a set where every element is picked with probability $1/2^i$. If the guess of i is accurate, then there is a $\Theta(1)$ probability of obtaining a non-empty subset. If the guess is too high, then the chances of getting an empty set is too low; otherwise it's too high. We can thus perform binary search on the scale. Since the number of scales is $\log n$, the number of rounds is $O(\log\log n)$. To union bound over all possible such rounds, in each round we make $O(\log\log\log n)$ queries, thus giving the total number.

The proof of (i) is already present in Stockmeyer (1983) (also see Ron and Tsur (2016)). We need the statement (ii), but the proof is not much more involved. We give the details below.

We begin by proving (i) which also implies the probability of returning FAIL is ≤ 0.01. Fix an iteration i. We say that this iteration is *good* if the following is true depending on $\mathsf{supp}(v)$.

Algorithm 1 DegreeEstimation

```
Input: OR query access to $v \in (\mathbb{R}_0^+)^n$
Output: Returns $\mathsf{est}(v)$, an estimate of the support of $v$.
$i \leftarrow 1, \mathsf{lo} \leftarrow 1, \mathsf{hi} \leftarrow \lceil \log n \rceil$
  while $\mathsf{hi} > \mathsf{lo}$ do
    Perform $(i,t)$ test with $t = C \log\log\log n$, and obtain $f_i$.
    if $f_i \leq 0.6$ then
      // conclude that $\mathsf{supp}(v) < 2^{i-1}$
      $\mathsf{hi} \leftarrow i-1; i \leftarrow \lceil (\mathsf{lo}+i)/2 \rceil$
    else if $f_i \geq 0.7$ then
      // conclude that $\mathsf{supp}(v) > 2^{i+1}$
      $\mathsf{lo} \leftarrow i+1; i \leftarrow \lfloor (i+\mathsf{hi})/2 \rfloor$
    else
      break while loop.
    end
end
// All conclusions correct implies $2^{i-1} \leq \mathsf{supp}(v) \leq 2^{i+1}$.
Perform $(i,C)$-test and obtain $f_i$.
if $f \in [0.35, 0.95]$ then
  return $\mathsf{est}(v) \leftarrow 2^i$.
return FAIL.
```

- $\mathsf{supp}(v) < 2^{i-1}$ and the algorithm concludes $\mathsf{supp}(v) < 2^i$, or
- $\mathsf{supp}(v) > 2^{i+1}$ and the algorithm concludes $\mathsf{supp}(v) > 2^i$,

That is, when support is much smaller or much larger than the scale, then the binary search moves in the correct direction.

Proposition 5 *Fix an iteration i. The probability that i is not good is $\leq \frac{1}{200 \log\log n}$.*

Proof When $\mathsf{supp}(v) < 2^{i-1}$, we expect each set S_j sampled in the ith iteration to have $\Pr[w(S_j) = 0] = (1 - \frac{1}{2^i})^{\mathsf{supp}(v)} > (1 - \frac{1}{2^i})^{2^{i-1}} \geq 0.5$. Thus, in this case we get that the expected fraction of sets with $w(S_j) > 0$ is $\mathbf{E}[f_i] \leq 0.5$. Therefore, in this case, using Chernoff bounds we get that $\mathbf{Pr}[f_i > 0.55] \leq 2^{-ct}$ for some constant c. When $t = C \log\log\log n$, for large enough constant C, this number is $\leq \frac{0.0001}{\log\log n}$.

Similarly, when $\mathsf{supp}(v) > 2^{i+1}$, we expect each set S_j sampled in the ith iteration to have $\Pr[w(S_j) = 0] = (1 - \frac{1}{2^i})^{\mathsf{supp}(v)} \leq (1 - \frac{1}{2^i})^{2^{i+1}} \leq 0.14$. Thus, in this case we get $\mathbf{E}[f_i] \geq 0.86$, and so $\mathbf{Pr}[f_i < 0.7] \leq \frac{0.0001}{\log\log n}$. ■

Now consider the i's encountered by the binary search algorithm. There are at most $2 \log\log n$ such iterations, and thus by union bound, with probability ≥ 0.999 all of them are good. This implies that when the while loop breaks, we are guaranteed $2^{i-1} \leq \mathsf{supp}(v) \leq 2^{i+1}$. The probability that the final test fails is ≤ 0.0001. To see this, when $2^{i-1} \leq \mathsf{supp}(v) \leq 2^{i+1}$, the probability a single S_j is empty is $\mathbf{Pr}[w(S_j) = 1] = 1 - \left(1 - \frac{1}{2^i}\right)^{\mathsf{supp}(v)} \in [1 - \frac{1}{e^{1/2}}, \frac{15}{16}] \subseteq [0.35, 0.95]$. And therefore, the probability the final test aborts, again by Chernoff bounds, is ≤ 0.0001. This proves (i).

To establish (ii), first note that the lower bound $\mathbf{E}[\frac{1}{\mathsf{est}(v)}] \geq \frac{2}{5\mathsf{supp}(v)}$ follows from (i): the LHS is $\geq 0.99 \cdot \frac{1}{2\mathsf{supp}(v)}$. To establish the upper bound on the LHS, we need to worry that there may be a "largish" 0.0001 chance that $\mathsf{est}(v) \leq \mathsf{supp}(v)/10^{12}$. This is not ruled out by the statement (i), but indeed, it's easy to see this is not possible.

Proposition 6 *For any $L \geq 40$, $\mathbf{Pr}[\mathsf{est}(v) \leq \mathsf{supp}(v)/L] \leq e^{-L}$.*

Proof Suppose the algorithm returns $\mathsf{est}(v) = 2^i$ but $\mathsf{supp}(v) \geq L \cdot 2^i$ for some $L \geq 40$. In the final (i, C) test, the probability that a sample S_j has $w(S_j) = 0$ is $= \left(1 - \frac{1}{2^i}\right)^{\mathsf{supp}(v)} \leq e^{-L}$. Therefore, the probability $f \leq 0.95$, that is, $0.05C$ of these sets are indeed empty, is at most $2^C \cdot e^{-0.05CL} \leq e^{-0.0025CL}$ for large enough $L \geq 40$. When C is large enough, this is $\leq e^{-L}$. ■

This is enough to upper bound $\mathbf{E}[\frac{1}{\mathsf{est}(v)}]$.

$$\mathbf{E}[\frac{1}{\mathsf{est}(v)}] \leq 0.99 \cdot \frac{2}{\mathsf{supp}(v)} + 0.01 \cdot \frac{40}{\mathsf{supp}(v)} + \sum_{L \geq 40} \frac{Le^{-L}}{\mathsf{supp}(v)} \leq \frac{3}{\mathsf{supp}(v)}$$

where the first term corresponds to when $\mathsf{est}(v) \geq \mathsf{supp}(v)/2$ and the latter two sums is an (overestimation) using the previous proposition. ■

The next tool we need is an algorithm which *reconstructs* weighted graphs using CROSS queries. In particular, we use an algorithm by Bshouty and Mazzawi (2012) which reconstructs graphs on n vertices and m edges. We state this as a lemma.

Lemma 7 *(Paraphrasing Bshouty and Mazzawi, 2012, Theorem 3). There is a deterministic polynomial time algorithm* BMGraphReconstruction *for reconstructing a hidden matrix $A \in (\mathbb{R}_0^+)^{n \times n}$ with at most m non-zero entries that uses*

$$O\left(\frac{m \log n}{\log m} + m \log \log m\right)$$

queries of the form $Q(w, q) = w^\intercal A q$, where $w, q \in \{0, 1\}^n$.

For our purposes, the number of edges would be unknown (in fact, random variables), and so we use a simple modification to get an efficient algorithm that recovers the hidden edges of a graph using CROSS queries that doesn't require to know the number of the hidden edges to be known.

Lemma 8 *(Follows readily from Bshouty and Mazzawi, 2012, Theorem 3). There is an algorithm* GraphReconstruction *that can recover a bipartite graph (V_1, V_2, E) with non-negative weights $|V_1| \leq n, |V_2| \leq n$ and $|E| = m$ using*

$$O\left(\frac{m \log n}{\log m} + m \log \log m\right)$$

CROSS *queries even if m is unknown to the algorithm.*

Proof An observation about Lemma 7 is that if A had more than m non-zero entries, then the deterministic algorithm detects this and aborts. This is a standard idea: we run BMGraphReconstruction with number of edge guesses growing in powers of 2 and accept the first time that algorithm doesn't abort. We provide some details in Appendix B for completeness. ■

The final tool we need is the simple DFS style algorithm to find a spanning forest. In particular, if the number of connected components is q, then the remaining edges can be found in $O(q \log n)$ many queries. This was also used by Apers et al. (2022) for unweighted graphs but an inspection of their proof shows that it readily works with weights as well. The idea is that even in a weighted non-negative vector, a single element in the support can be found using binary search, and every connected component can find an edge coming out of it (if they exist) in $O(\log n)$ queries.

Lemma 9 *(Paraphrasing Apers et al., 2022, Lemma 5.1) Let $G = (V, E)$ be an n-vertex weighted graph with non-negative weights. Let G' be a contraction of G with q many supervertices, which are given explicitly as the partition $P = A_1, \cdots, A_q$ of V. There is a deterministic algorithm* DFSSpanningForest *that takes in G, G' and outputs a set of edges $F \subseteq E$ that form a spanning forest of G' and makes $O(q \log n)$* CUT *queries to G.*

3. Main Algorithm

The algorithm proceeds in phases. At the beginning of each phase, the algorithm has a collection of t connected components with $t = n$ in the first phase and the connected components are all singletons. The main workhorse of each phase is Algorithm 2 which is a Monte Carlo algorithm that makes $O(t \log \log n \log \log \log n)$ queries and returns a collection of connected components which is a *constant* factor smaller, with probability $\geq 1 - \frac{1}{100 \log \log n}$. We first describe this, and using this algorithm one can get the final spanning forest algorithm in a straightforward fashion as in Apers et al. (2022).

We further assume for now that each component C_i has a *representative* vertex $c_i \in C_i$ with the guarantee that c_i has an edge to $V \setminus C_i$. It is not too hard to maintain this with an $O(n)$ overhead on the queries, and we describe this when we describe the final algorithm. For now, let us go with this assumption.

Let $C_1, \cdots, C_t$ be the connected components of G_1. Algorithm 2 begins by coloring the components to red or blue uniformly at random. Let B be the set of vertices colored blue. Let $b : V \mapsto [n]$ maps a vertex to the number of its blue neighbors. Call a representative r "good" if it is colored red and $b(r) \geq 1$. Note that a representative is good with probability $\geq 1/4$; its component has to be colored red and one of its neighbor's component has to be colored blue. Since it has at least one neighbor outside its component, the probability follows. Call a component good if its representative is good; the expected number of good components is $\geq t/4$.

Let R be the set of good red representatives. We first estimate the number of blue neighbors, $\mathsf{est}(r_i)$ for each $r_i \in R$ using DegreeEstimation in Algorithm 1. More specifically, we treat the vertex pairs with an endpoint r_i and the other colored blue as a hidden non-negative vector of dimension at most n, to which we apply DegreeEstimation. If the algo doesn't FAIL, DegreeEstimation ensures $\frac{1}{3b(r_i)} < \mathbf{E}[\frac{1}{\mathsf{est}(r_i)}] < \frac{3}{b(r_i)}$. We apply DegreeEstimation on every r_i, which takes $O(t \log \log n \log \log \log n)$ queries in total. Note in Lemma 4 we showed that the probability of returning FAIL is bounded by 0.01. Let R' be the set of good red representatives that gets a valid estimate est. We have $\mathbf{E}[|R'|] \geq 0.99\mathbf{E}[|R|] \geq 0.99t/4$.

```
Algorithm 2 ReduceConnectedComponents
Input: $G_1$ with $t$ connected components $C_1, \cdots, C_t$, each with $\geq 1$ representative
Output: A graph with $\leq \frac{47}{48}t$ connected components w.p. $\geq 1 - \frac{1}{100 \log \log n}$
count ← 0.
while $count < 35 \log \log \log n$ do
    Color all vertices in $C_i$ to red or blue uniformly at random.
    $B \leftarrow$ vertices colored blue.
    $R \leftarrow$ red representatives that have at least one blue neighbor.
    Estimate the number of blue neighbors of each $r \in R$ with DegreeEstimation in Algorithm 1
    to get $\mathsf{est}(r)$ or FAIL.
    $R_i \leftarrow \{r \in R | 2^i \leq \mathsf{est}(r) < 2^{i+1}\}$.
    For each integer $0 \leq i \leq \lceil \log n \rceil$, sample a set $B_i$ with every $b \in B$ sampled w.p. $\frac{1}{2^i}$ with
    replacement.
    try recover edges $E_1 := \cup_{i \in \lceil \log n \rceil} E(R_i, B_i)$ with $\lceil \log n \rceil$ calls of GraphReconstruction de-
    scribed in Lemma 8 using a total of $t \log \log n \log \log \log n$ queries, and see if number of
    connected components is $< 47t/48$:
        return $G_1 \cup E_1$.
    catch:
        $count \leftarrow count + 1$.
    end
end
```

Break $[1, n-1]$ into $O(\log n)$ disjoint intervals $I_0, I_1, \cdots, I_{\lceil \log n \rceil - 1}$ where $I_i = [2^i, 2^{i+1})$. For each i, Algorithm 2 samples **a single** set B_i from B with replacement at rate $\frac{1}{2^i}$. Define

$$R_i := \{r \in R' | 2^i \leq \mathsf{est}(r) < 2^{i+1}\}$$

for $0 \leq i < \lceil \log n \rceil$. The algorithm then tries to reconstruct the $O(\log n)$ different graphs $E(R_i, B_i)$ using the GraphReconstruction described in Lemma 8, with a *total* budget of $t \log \log n \log \log \log n$. If budget runs out we abort. If all graphs are reconstructed, then we add all these edges to G_1. If the number of connected components drops significantly, we return. Otherwise, we abort. Once aborted, we repeat *everything* all over again. The total number of trials is $C \log \log \log n$, and if all these tries fail, we abort our algorithm. We now analyze the algo, and in particular prove the following.

Lemma 10 *Let $G = (V, E)$ be a weighted graph with non-negative edge weights. Given a subgraph G_1 of G with t connected components for some $t \geq \frac{n}{\log n}$. Algorithm 2 returns a graph G_2 with at most $\frac{47}{48}t$ connected components with probability $1 - \frac{1}{100 \log \log n}$ using $O(t \log \log n (\log \log \log n)^2)$ CROSS queries.*

Proof

Let us begin by understanding the number of edges in $E(R_i, B_i)$'s. Fix an i. Define Y_{r_i} to be the random variable representing the number of edges with an endpoint $r_i \in R_i$ and the other endpoint in B_i. Note that

$$\mathbf{E}[Y_{r_i}] = \mathbf{E}[\frac{b(r_i)}{2^i}]$$

By construction, $2^i \leq \mathsf{est}(r_i) < 2^{i+1}$. Thus $\frac{1}{\mathsf{est}(r_i)} \leq \frac{1}{2^i} < \frac{1}{\mathsf{est}(r_i)/2}$. It follows that

$$\mathbf{E}[\frac{b(r_i)}{\mathsf{est}(r_i)}] \leq \mathbf{E}[\frac{b(r_i)}{2^i}] < \mathbf{E}[\frac{2b(r_i)}{\mathsf{est}(r_i)}]$$

By Lemma 4, $\frac{2}{5b(r_i)} < \mathbf{E}[\frac{1}{\mathsf{est}(r_i)}] < \frac{3}{b(r_i)}$. From it we have $\mathbf{E}[\frac{b(r_i)}{\mathsf{est}(r_i)}] > \frac{2}{5}$ and $\mathbf{E}[\frac{2b(r_i)}{\mathsf{est}(r_i)}] < 6$. So

$$\frac{2}{5} < \mathbf{E}[Y_{r_i}] < 6.$$

That is, each r_i has $\Theta(1)$ edges, in expectation, to vertices in B_i.

Let $Y = \sum_{r_i \in R_i, 0 \leq i < \lceil \log n \rceil} Y_{r_i}$. Y counts the total number of edges in these $O(\log n)$ subgraphs. Also note that the Y_{r_i}s are not necessarily independent. Since $\frac{t}{4} \leq \mathbf{E}[|R|] \leq t$, and in the DegreeEstimation each vertex fails with probability 0.01, we know that $\frac{t}{4} \cdot \frac{99}{100} \leq \frac{9}{40}t \leq \mathbf{E}[|R'|] \leq t$. By linearity of expectation,

$$\frac{t}{12} \leq \mathbf{E}[\sum_{r_i \in R'} Y_{r_i}] \leq 6t.$$

That is, the expected number of edges in the union of $E(R_i, B_i)$'s is $\Theta(t)$. Let $\mathcal{E}_1$ be the event that $Y \geq 300t$. By Markov's inequality,

$$\mathbf{Pr}[\mathcal{E}_1] \leq \frac{1}{50},$$

i.e., the number of sampled edges is greater than $300t$ with probability at most $\frac{1}{50}$. Thus, we have established the following proposition.

Proposition 11 $\mathbf{Pr}[\mathcal{E}_1] = \mathbf{Pr}[\sum_i |E(R_i, B_i)| \geq 300t] \leq \frac{1}{50}$.

Next, let us argue that with decent probability a significant number of the r_i's do sample at least one edge. This will lead to the decrease in the number of connected components. To this end, define

$$Z_{r_i} = \begin{cases} 1, & \text{if } Y_{r_i} \geq 1 \\ 0, & \text{otherwise.} \end{cases}$$

That is, Z_{r_i} is the indicator if r_i has an edge to B_i.

Let $\mathcal{G}$ denote the event that our estimate for $b(r_i)$ was good. That is, $\frac{\mathsf{est}(r_i)}{2} \leq b(r_i) \leq 2\mathsf{est}(r_i)$. By Lemma 4,

$$\mathbf{Pr}[\mathcal{G}] \geq 0.99.$$

Now note that

$$\mathbf{Pr}[Z_{r_i} = 0|\mathcal{G}] = (1 - \frac{1}{2^i})^{b(r_i)} \leq (1 - \frac{1}{2^i})^{\mathsf{est}(r_i)/2} \leq (1 - \frac{1}{2^i})^{2^{i-1}} \leq 0.61$$

Z_{r_i}s are Bernoulli random variables, so $\mathbf{Pr}[Z_{r_i} = 1|\mathcal{G}] \geq 0.39$. Therefore,

$$\begin{aligned} \mathbf{Pr}[Z_{r_i} = 1] &= \mathbf{Pr}[Z_{r_i} = 1|\mathcal{G}]\mathbf{Pr}[\mathcal{G}] + \mathbf{Pr}[Z_{r_i} = 1|\mathcal{G}^c]\mathbf{Pr}[\mathcal{G}^c] \\ &\geq 0.39 \cdot 0.99 \\ &> 0.38 \end{aligned}$$

and because $\frac{9}{40}t \le \mathbf{E}[|R'|] \le t$, by linearity of expectation,

$$\frac{t}{12} \le \mathbf{E}[\sum_{r_i \in R'} Z_{r_i}] \le t.$$

Let $\mathcal{E}_2$ represent the event $\sum_{r \in R} Z_r \le \frac{t}{24}$. Apply reverse Markov's inequality on $\sum_{r \in R} Z_r$,

$$\mathbf{Pr}[\mathcal{E}_2] \le \frac{t - \mathbf{E}[\sum_{r_i \in R'} Z_{r_i}]}{t - \frac{t}{24}} \le \frac{22}{23}.$$

i.e., the number of representatives being an endpoint of at least one sampled edges is at most $\frac{t}{24}$ with probability at most $\frac{22}{23}$. Thus, we have established the following proposition.

Proposition 12 $\mathbf{Pr}[\mathcal{E}_2] = \mathbf{Pr}[\text{fewer than } t/24 \text{ of } r_i\text{'s have an edge to } B_i] \le \frac{22}{23}$.

Let S_i be the number of edges in the induced subgraph (R_i, B_i). The subgraph (R_i, B_i) has at most n vertices. By Lemma 8, it takes

$$O\left(\frac{S_i \log n}{\log S_i} + S_i \log\log S_i\right)$$

queries to recover the edges.

If $\mathcal{E}_1$ doesn't occur, we have $Y < 300t$. Since $\sum_{0 \le i \le \lceil \log n \rceil} S_i = Y$, summing S_i up for all groups,

$$\begin{aligned}
\sum_{i=0}^{\lceil \log n \rceil} O(\frac{S_i \log n}{\log S_i} + S_i \log\log S_i) &\le \log n \sum_{i=0}^{\lceil \log n \rceil} O(\frac{S_i}{\log S_i}) + \sum_{i=0}^{\lceil \log n \rceil} O(S_i \log\log n) \\
&= \log n \cdot O(\frac{Y}{\log Y}) + O(Y \log\log n) \quad \text{(by Proposition 3)} \\
&= O(\frac{t \log n}{\log t}) + O(t \log\log n) \\
&= O(t \log\log n)
\end{aligned}$$

That is, if $\mathcal{E}_1$ doesn't occur in some while loop, then in that loop the "try" would work.

Furthermore, if $\mathcal{E}_2$ doesn't occur, it guarantees that the number of representatives being an endpoint of at least one sampled edges is at least $\frac{t}{24}$. That is, after the edges are recovered, these $\frac{t}{24}$ connected components now connect to other connected components. The number of connected components after this round is at most $(1 - \frac{1}{48})t$ as the $\frac{t}{24}$ connected components might connect to each other.

By union bound, the probability that neither of $\mathcal{E}_1, \mathcal{E}_2$ occurs is

$$1 - \mathbf{Pr}[\mathcal{E}_1 \cup \mathcal{E}_2] \ge 1 - \mathbf{Pr}[\mathcal{E}_1] - \mathbf{Pr}[\mathcal{E}_2] \ge \frac{1}{50}.$$

And therefore, we get the following.

Proposition 13 *The probability that a while loop succeeds in returning a E_1 is $\ge 1/50$.*

Therefore, if we run for $35 \log\log\log n$ trials, one of them succeeds with probability $\geq 1 - (\frac{49}{50})^{35 \log\log\log n} \geq 1 - \frac{1}{100 \log\log n}$. ■

We introduce the definition of “active” and “inactive” vertices to help maintain representatives.

Definition 14 *Let C be the set of vertices of a connected component in G. Define the “active” vertices of C to be the vertices $v \in C$ such that* $\mathsf{CROSS}(\{v\}, V\backslash C) > 0$. *Define the “inactive” vertices of C to be the vertices $v \in C$ such that* $\mathsf{CROSS}(\{v\}, V\backslash C) = 0$.

At any point we maintain a set of connected components. A vertex in a connected component is inactive if it doesn't have any edges to vertices outside the component. This can occur even when G is connected. The reason we need to track the connected components is that we don't want to involve any inactive vertices in queries (otherwise we would pay $\Omega(n)$ in each iteration which is too expensive). Given t connected components, we want to make $O(t)$ queries for maintaining representatives even when $t \ll n$.

Now we are ready to state the final algorithm.

Algorithm 3 SpanningForestAlgorithm

```
1  Input: $G = (V, E)$ with $w(e) \geq 0$ for all $e \in E$.
2  Output: A spanning forest of $G$ w.p. $\geq \frac{2}{3}$.
3  $n \leftarrow |V|$.
4  $i \leftarrow 1$.
5  $G_i \leftarrow (V, \emptyset)$.
6  for $v \in V$ do
7      if $|E(\{v\}, V\backslash v)| > 0$ then
8          Mark $v$ as active.
9      else
10         Mark $v$ as inactive.
   end
11 while $i < 33 \log\log n$ do
12     $G_{i+1} \leftarrow$ ReduceConnectedComponents($G_i$)
13     If any of old representatives in $G_i$ is still active, mark it as the next representative in $G_{i+1}$.
       Otherwise keep marking vertices in $G_{i+1}$ inactive until one finds an active vertex.
14     $i \leftarrow i + 1$
   end
15 $\mathcal{F} \leftarrow$ Find the spanning forest of $G_i$ with DFSSpanningForest in Lemma 9.
16 return $\mathcal{F}$.
```

Theorem 15 *Let $G = (V, E, w)$ be a weighted graph in which we want to find a spanning forest where $|V| = n$. V is known in advance but edges E and edge weights $w \in (\mathbb{R}_0^+)^{|E|}$ are hidden. Algorithm 3 is an $O(n \log\log n (\log\log\log n)^2)$* CUT *query complexity algorithm that reconstructs a spanning forest of G with probability at least $\frac{2}{3}$.*

Proof The queries we make in Algorithm 3 are from 3 parts.

1. Calling Algorithm 2 in line 12;
2. Finding active vertices in line 6 - 10 and 13;
3. Finding the spanning forest with subgraph G_i in line 15.

Line 12: If all of the $33 \log \log n$ calls of Algorithm 2 succeed, by Lemma 10, the total number of queries that used in calling Algorithm 2 is at most

$$\sum_{i=0}^{k} c^i n \log \log n (\log \log \log n)^2 = O(n \log \log n (\log \log \log n)^2).$$

Line 6 - 10 and 13: To learn whether a vertex is active takes 1 query. Every time Algorithm 3 invokes Algorithm 2, in each connected component, we find one "active" vertex (if there is any) by iterating over vertices that are not "inactive" at the end. Once a vertex is "inactive", it becomes "inactive" forever. The only vertices that we may query more than once are the representatives, in which case in the ith call of Algorithm 2 there are at most $c^i n$ of them. So the total number of queries used to find representatives is at most $n + \sum_{i=0}^{k} c^i n = O(n)$.

Line 15: If all of the $33 \log \log n$ calls of Algorithm 2 succeed, the number of connected components of $G_{33 \log \log n}$ is at most $(\frac{47}{48})^{33 \log \log n} \leq \frac{1}{\log n}$. We apply Lemma 9 to find the remaining tree edges deterministically which takes $O(n)$ CROSS queries. Summing up the queries from the three parts, we conclude the total number of queries is $O(n \log \log n (\log \log \log n)^2)$.

Error probability: Algorithm 2 is called at most $33 \log \log n$ times by the analysis above and each time fails with probability at most $\frac{1}{100 \log \log n}$ by Lemma 10. The error probability of Algorithm 3 is thus $\leq \frac{1}{3}$. ■

4. Concluding Thoughts

In this paper, we showed a randomized algorithm which makes $O(n \log \log n (\log \log \log n)^2)$ cut queries and can recover a spanning forest of an undirected *weighted* graph with probability $> 2/3$. On weighted graphs, this is the first algorithm which is better than the simple $O(n \log n)$-query deterministic algorithm.

Several questions are left open. The natural one is of course: can one actually get an $O(n)$-query algorithm? At this point, even a deterministic algorithm is not ruled out. If we consider our approach, then the biggest bottleneck is the degree estimation problem, and we believe this may be a problem worth studying in its own right. More precisely, given a non-negative weighted N-dimensional vector where for any subset $S \subseteq [N]$ we can query and obtain the $\sum_{i \in S} w_i$, can we *estimate* the size of the support (F_0 norm) up to say a constant factor making only $O(1)$ queries? Or is there a dependence on N necessary?

Another big question left open is about *minimum cuts* in weighted graphs. Currently, the best known algorithm is by Mukhopadhyay and Nanongkai (2020) who give an $O(n\text{polylog}(n))$ query algorithm to solve the minimum cut problem, where the exponent in the polylog is unspecified but seems to be at least 3. Is there an $O(n)$ algorithm for this problem? The paper Apers et al. (2022) actually gives such a Monte-Carlo algorithm for unweighted graphs, but generalizing to weighted graphs would need new techniques.

Acknowledgments

This research was partly supported by NSF grant # 2041920

References

Noga Alon and Vera Asodi. Learning a hidden subgraph. *SIAM Journal on Discrete Mathematics (SIDMA)*, 18(4):697–712, 2005.

Noga Alon, Richard Beigel, Simon Kasif, Steven Rudich, and Benny Sudakov. Learning a hidden matching. *SIAM Journal on Computing (SICOMP)*, 33(2):487–501, 2004.

Simon Apers, Yuval Efron, Pawel Gawrychowski, Troy Lee, Sagnik Mukhopadhyay, and Danupon Nanongkai. Cut query algorithms with star contraction. *Proc., FOCS*, 2022.

Sepehr Assadi, Deeparnab Chakrabarty, and Sanjeev Khanna. Graph connectivity and single element recovery via linear and or queries. In *Proc., European Symposium on Algorithms*, volume 204, page 7, 2021.

Nader H. Bshouty and Hanna Mazzawi. Toward a deterministic polynomial time algorithm with optimal additive query complexity. *Theoretical Computer Science*, 417:23–35, 2012.

Sung-Soon Choi. Polynomial time optimal query algorithms for finding graphs with arbitrary real weights. In Shai Shalev-Shwartz and Ingo Steinwart, editors, *Proc., Conf. on Learning Theory*, volume 30, pages 797–818, 2013.

Sung-Soon Choi and Jeong Han Kim. Optimal query complexity bounds for finding graphs. *Artif. Intell.*, 174(9-10):551–569, 2010.

Andrei Graur, Tristan Pollner, Vidhya Ramaswamy, and S Matthew Weinberg. New query lower bounds for submodular function minimization. *Proc., Innovations in Theoretical Computer Science (ITCS)*, page 64, 2020.

Vladimir Grebinski and Gregory Kucherov. Optimal reconstruction of graphs under the additive model. *Algorithmica*, 28(1):104–124, 2000.

Hanna Mazzawi. Optimally reconstructing weighted graphs using queries. In *Proc., SODA*, pages 608–615, 2010.

Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: sequential, cut-query, and streaming algorithms. In *Proc., STOC*, pages 496–509, 2020.

Lev Reyzin and Nikhil Srivastava. Learning and verifying graphs using queries with a focus on edge counting. In *Proc., International Conference on Algorithmic Learning Theory (ALT)*, pages 285–297. Springer, 2007.

Dana Ron and Gilad Tsur. The power of an example: Hidden set size approximation using group queries and conditional sampling. *ACM Transactions on Computation Theory (TOCT)*, 8(4): 1–19, 2016.

Aviad Rubinstein, Tselil Schramm, and S. Matthew Weinberg. Computing exact minimum cuts without knowing the graph. In *Proc., Innovations in Theoretical Computer Science (ITCS)*, pages 39:1–39:16, 2018.

Larry Stockmeyer. The complexity of approximate counting. In *Proc., STOC*, pages 118–126, 1983.

Appendix A. Proof of Proposition 3

Proposition 16 *Let $t_1, t_2, \cdots, t_{\lceil \log n \rceil}$ be integers at least 2. If $\sum_{i=1}^{\lceil \log n \rceil} t_i = t$ where $\frac{n}{\log n} \le t \le n$, then $\sum_{i=1}^{\lceil \log n \rceil} \frac{t_i}{\log t_i} = O(\frac{t}{\log t})$.*

Proof Partition $\{t_i\}$ into sets $P_1 = \{t_i | t_i \in [2, t/\log^2 t)\}, P_2 = \{t_i | t_i \in [t/\log^2 t, t]\}$. For $t_i \in P_1$, note $|P_1| \le \log n \le 2 \log t$, it follows that

$$\sum_{i \in P_1} \frac{i}{\log i} \le \sum_{i \in P_1} i \le 2 \log t \cdot \frac{t}{\log^2 t} = \frac{2t}{\log t}.$$

For $t_i \in P_2$, since $t_i > \sqrt{t}, \log t_i \ge \frac{1}{2} \log t$. So $t_i / \log t_i \le t_i / \frac{1}{2} \log t = \frac{2t_i}{\log t}$. For those t_i,

$$\sum_{i \in P_2} \frac{i}{\log i} \le \sum_{i \in P_2} \frac{2i}{\log t} \le \frac{2t}{\log t}.$$

Combine the two inequalities,

$$\sum_{i \in P} \frac{i}{\log i} = \sum_{i \in P_1} \frac{i}{\log i} + \sum_{i \in P_2} \frac{i}{\log i} \le \frac{4t}{\log t} = O(\frac{t}{\log t}).$$

■

Appendix B. Proof of Lemma 8

Lemma 17 *There is an algorithm* GraphReconstruction *that can recover a (V_1, V_2, E) non-negative weighted bipartite graph with $|V_1| \le n, |V_2| \le n$ and $|E| = m$ using*

$$O\left(\frac{m \log n}{\log m} + m \log \log m\right)$$

CROSS *queries even if m is unknown to the algorithm.*

Proof Lemma 7 uses queries of the form $Q(w, q) = w^{\intercal} A q$ where $w, q \in \{0, 1\}^n$. Let $S_w = \{i \in V_1 | w_i = 1\}, S_q = \{j \in V_2 | q_j = 1\}$. Any such $w^{\intercal} A q$ query can be converted to a CROSS query on the bipartite graph (V_1, V_2, E) because $w^{\intercal} A q = \sum_{w_i=1} \sum_{q_j=1} A_{ij} = CROSS(S_w, S_q)$.

We move on to prove the claim that the lemma holds regardless of if m is known.

We can guess the magnitude of m in growing powers of 2. Assume GraphReconstruction in Lemma 8 requires $c_1 \frac{m \log n}{\log m} + c_2 m \log \log m$ many queries for some constant c_1, c_2. We run the algorithm with parameter 2^i from $i = 0$ to $2^i \geq m$, in which case the algorithm guarantees success. Each run stops when the number of queries reaches $\frac{c_1 2^i \log n}{\log 2^i} + c_2 2^i \log\log 2^i$. Let q be the smallest integer such that $2^q \geq m$. The total number of queries it takes until $i = q$ is

$$\begin{aligned}\sum_{i=1}^{q}\left(c_1\frac{2^i \log n}{\log 2^i} + c_2 2^i \log\log 2^i\right) &= O\left(\frac{2^{q+1}\log n}{\log 2^{q+1}} + 2^{q+1}\log\log 2^{q+1}\right) \qquad \text{(see Proposition 3)}\\ &= O\left(\frac{m\log n}{\log m} + m\log\log m\right)\end{aligned}$$

■

Appendix C. A simple lower bound

We show a simple lower bound that no deterministic algorithm cannot obtain the exact degree of any vertex in $o(n)$ queries. In fact, we prove this for a stronger set of queries, namely LINEAR queries. Given a vector $v \in \mathbb{R}^N$. A LINEAR query LINEAR(w) with input $w \in \mathbb{R}^N$ returns $v \cdot w$. It is easy to see that a LINEAR query is more powerful than a CUT or CROSS query.

Theorem 18 *Let $v \in \mathbb{R}^n_{\geq 0}$ be an unknown vector. Any **deterministic** algorithm using* LINEAR *queries that returns* supp(v) *exactly has query complexity at least n.*

Proof The proof technique is similar to that in Graur et al. (2020, Claim 29). Fix any deterministic algorithm $\mathcal{A}$ that uses at most $n-1$ LINEAR queries. The coefficients of the $n-1$ LINEAR queries form an $(n-1)\times n$ matrix. Let $\boldsymbol{f}$ be a non-zero vector in its nullspace. We can find a constant c such that $c\boldsymbol{f} + \mathbf{1}$ has at least 1 zero entry with the remaining entries non-negative. $\mathcal{A}$ fails against $c\boldsymbol{f} + \mathbf{1}$ and $\mathbf{1}$ because the two sets of queries would get the exact same answers yet supp$(c\boldsymbol{f} + \mathbf{1}) < n$ and supp$(\mathbf{1}) = n$. ■

A slight modification of this proof shows that any deterministic algorithm solving the graph connectivity problem using LINEAR queries has query complexity at least n. (The graph connectivity is a decision problem that asks one to determine whether a graph is connected.) The same technique would show any $n-1$ queries algorithm can't differentiate between an n star graph and an $n-1$ star graph with a disconnected singleton vertex.