On the Cut-Query Complexity of Approximating Max-Cut

Orestis Plevrakis
Department of Computer Science, Princeton University, NJ, USA

Seyoon Ragavan
Computer Science and Artificial Intelligence Lab,
Massachusetts Institute of Technology, Cambridge, MA, USA

S. Matthew Weinberg
Department of Computer Science, Princeton University, NJ, USA

Abstract

We consider the problem of query-efficient global max-cut on a weighted undirected graph in the value oracle model examined by [31]. Graph algorithms in this cut query model and other query models have recently been studied for various other problems such as min-cut, connectivity, bipartiteness, and triangle detection. Max-cut in the cut query model can also be viewed as a natural special case of submodular function maximization: on query $S \subseteq V$, the oracle returns the total weight of the cut between S and $V \backslash S$.

Our first main technical result is a lower bound stating that a deterministic algorithm achieving a c-approximation for any $c > 1/2$ requires $\Omega(n)$ queries. This uses an extension of the cut dimension to rule out approximation (prior work of [20] introducing the cut dimension only rules out exact solutions). Secondly, we provide a randomized algorithm with $\tilde{O}(n)$ queries that finds a c-approximation for any $c < 1$. We achieve this using a query-efficient sparsifier for undirected weighted graphs (prior work of [31] holds only for unweighted graphs).

To complement these results, for most constants $c \in (0, 1]$, we nail down the query complexity of achieving a c-approximation, for both deterministic and randomized algorithms (up to logarithmic factors). Analogously to general submodular function maximization in the same model, we observe a phase transition at $c = 1/2$: we design a deterministic algorithm for global c-approximate max-cut in $O(\log n)$ queries for any $c < 1/2$, and show that any randomized algorithm requires $\Omega(n/\log n)$ queries to find a c-approximate max-cut for any $c > 1/2$. Additionally, we show that any deterministic algorithm requires $\Omega(n^2)$ queries to find an exact max-cut (enough to learn the entire graph).

2012 ACM Subject Classification Mathematics of computing → Approximation algorithms

Keywords and phrases query complexity, maximum cut, approximation algorithms, graph sparsification

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.115

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2211.04506`

Funding *Seyoon Ragavan*: Supported by an Akamai Presidential Fellowship.
S. Matthew Weinberg: Supported by NSF CCF-1955205.

Acknowledgements SR would like to thank Georgy Noarov and Dmitry Paramonov for collaborating on a course project that led to this work. We would also like to thank Sepehr Assadi for helpful discussions and anonymous reviewers for feedback and suggested changes.

1 Introduction

For the most part, the field of graph algorithms has focused on extensive study in the "standard" model where the entire graph is given to the algorithm as input e.g. in the form of an adjacency list or an adjacency matrix. However, another growing body of work focuses on the case where the algorithm does not know the graph; rather, it has query access to an oracle that knows the graph, and aims to make as few queries as possible. Different models restrict the algorithm to make different types of queries e.g. individual edge queries, linear measurements [1, 5], matrix-vector products [32], etc.

One model that has attracted particular attention is the *cut query model* [21, 15, 31, 20, 30, 28, 4, 14]. In this model, the algorithm has query access to the cut function of a graph G with vertex set $[n]$: it provides subsets $S \subseteq [n]$ as queries and the oracle returns the total weight of all edges of G with exactly one endpoint in S. An additional reason for the wide interest in this model is that the graph's cut function is in particular a submodular function; [20] and [28] obtained new lower bounds for the query complexity of submodular function minimization by considering the special case of graph min-cut in the cut query model.

A range of graph problems has been studied in the cut query model and related models. Perhaps the most fundamental is that of learning the entire graph [21, 16, 9, 15, 10]; once the entire graph is known to the algorithm, it has all the standard graph algorithm literature at its disposal to solve the problem of interest. A more subtle question is whether the given problem can be solved with fewer queries than one would need to learn the graph. This was first answered affirmatively for the case of min-cut [31, 30, 4]: an undirected, unweighted graph requires $\Omega(n^2/\log n)$ queries to learn [21] but $O(n)$ queries suffice to find its min-cut. Since then, novel algorithms have been developed in the cut query model and related models for problems such as connectivity [1, 32, 5, 4, 14], testing bipartiteness [1], and triangle detection [32].

In this work, we initiate the study of max-cut in the cut-query model. We aim to understand how many queries are necessary and sufficient for an algorithm to find a c-approximate global max-cut in an undirected, weighted graph.

1.1 Our Results

Our two main results are stated below. Both are concerned with the setting where $c \in (1/2, 1)$ i.e. one wants to do better than a straightforward greedy algorithm or guessing a random cut, but does not want an exact max-cut. The first result is a lower bound against deterministic algorithms, and the second result is a randomized algorithm.

▶ **Theorem 1** (See Corollary 7 for a precise statement). *For $c > 1/2$, any deterministic algorithm achieving a c-approximation requires $\Omega(n)$ queries.*

▶ **Theorem 2** (See Corollary 29 for a precise statement). *For $c < 1$, there exists a randomized algorithm with query complexity $\tilde{O}(n)$ that achieves a c-approximation.*

These results naturally beg the question of how the query complexity behaves for other ranges of c, for both deterministic and randomized algorithms. We also answer this question in most cases. The content of all of our results is summarized in the following three theorems, and also tabulated in Figure 1. Soon after, we provide context for these results.

▶ **Theorem 3** (See Theorems A.1 and B.1 in the full version for precise statements). *For $c = 1$, the query complexity for a deterministic algorithm to achieve a c-approximation is $\Theta(n^2)$ and the query complexity for a randomized algorithm to do the same is between $\tilde{\Omega}(n)$ and $O(n^2)$.*

▶ **Theorem 4** (See Corollary 29, Corollary 7, and Theorem B.1 in the full version for precise statements). *For $c \in (1/2, 1)$, the query complexity for a deterministic algorithm to achieve a c-approximation is between $\Omega(n)$ and $O(n^2)$ and the query complexity for a randomized algorithm to do the same is $\tilde{\Theta}(n)$.*

c	Deterministic	Randomized
1	$\Theta(n^2)$	$(\tilde{\Omega}(n), O(n^2))$
$(1/2, 1)$	$(\Omega(n), O(n^2))$	$\tilde{\Theta}(n)$
$(0, 1/2)$	$\Theta(\log n)$	$\Theta(1)$

Figure 1 Summary of our results. For each range of c, we state the query complexity (up to constant and logarithmic factors) that we show for achieving a c-approximation in both the deterministic and randomized settings. (n, n^2) indicates settings where we have a lower bound of $\tilde{\Omega}(n)$ and an upper bound of $O(n^2)$.

▶ **Theorem 5** (See Corollary E.2, Theorem E.4, and Corollary C.3 in the full version for precise statements)**.** *For $c \in (0, 1/2)$, the query complexity for a deterministic algorithm to achieve a c-approximation is $\Theta(\log n)$ and the query complexity for a randomized algorithm to do the same is $\Theta(1)$.*

We now provide context for each of our results, beginning with Theorem 1. Observe that there is a straightforward non-adaptive $O(n^2)$-query deterministic algorithm to find the max-cut exactly (observed in [31], as it applies to the min-cut as well). The algorithm can query all singletons and sets of size 2 and from this learn $w_{i,j} = \frac{1}{2}(F(\{i\})+F(\{j\})-F(\{i,j\}))$ for all i, j and thus the entire graph. The algorithm can then find the max cut exactly using brute force, since no more queries need to be made. This addresses all of the $O(n^2)$ upper bounds stated in the theorems. Theorem 3 shows that this trivial algorithm is optimal among deterministic algorithms: up to constant factors, any deterministic algorithm must learn the entire graph in order to find the global max-cut with cut queries.

Next, observe that there is a phase transition at $c = 1/2$. Theorem 5 establishes that a $(1/2 - \varepsilon)$-approximation can be achieved deterministically with $O(\log n)$ queries (which happen to also be necessary). On the other hand, even a randomized algorithm needs $\tilde{\Omega}(n)$ queries to guarantee a $(1/2 + \varepsilon)$-approximation.

Finally, observe that we resolve the asymptotic query complexity for most cases. For randomized algorithms, the only unresolved cases are $c = 1/2$ and $c = 1$.[1] For deterministic algorithms, the range $[1/2, 1)$ remains unresolved.

Before continuing, we note several facts about our results. In the positive direction, we note that all of our algorithms find a set S that is a c-approximation to the global max-cut (for undirected, weighted graphs with arbitrarily large weights), rather than simply estimating the value of the max-cut within a factor of c. We refer to these two settings as the *cut finding* and *value estimation* settings respectively. On the other hand, most of our lower bounds hold even against algorithms in the value estimation setting. The only exception is our $\tilde{\Omega}(n)$ lower bound on randomized algorithms for $c > 1/2$, which we only prove in the cut finding setting. Similarly, all algorithms we discuss, except for our $\tilde{O}(n)$-query randomized algorithm for $c \in (1/2, 1)$ and the well-known $O(n)$-query deterministic greedy algorithm for $c = 1/2$, are non-adaptive. On the other hand, all of our lower bounds hold even against adaptive algorithms.

It may appear at first glance that Theorem 1 is subsumed by the randomized lower bound in Theorem 4. Both results are lower bounds for $c > 1/2$: the former states that a deterministic algorithm in this setting requires $\Omega(n)$ queries, while the latter says that

[1] For the $c = 1/2$ case, simply outputting a random cut will achieve a $(1/2)$-approximation in expectation in $O(1)$ queries. But we are concerned with the algorithm's ability to achieve a c-approximation with some probability $p > 0$. In this setting, there exists a deterministic $O(n)$-query algorithm (see Section E.3 of the full version) but we do not have a matching lower bound.

a randomized algorithm requires $\tilde{\Omega}(n)$ queries. However, the deterministic lower bound is interesting on its own for two reasons. The first reason is that the randomized lower bound is only $\Omega(n/\log n)$, which is weaker than the $\Omega(n)$ we are able to prove in the deterministic case. Secondly, and more critically, the deterministic lower bound holds even for the value estimation setting while the randomized lower bound is only for the cut finding setting. Our argument for the randomized lower bound does not appear adaptable to the value estimation setting; indeed, the hard distribution used in our proof is actually very easy in the value estimation setting.[2]

In the negative direction, we note here that most of our algorithms have exponential time complexity even though they are query-efficient. For example, the $O(n^2)$-query algorithm we just described learns the graph in polynomial time, but then uses exponential brute force search to actually find its max-cut. This illustrates that, by focusing on query complexity, we are primarily concerned with the information theoretic question of how much the algorithm must learn about the graph to be able to estimate its max-cut, even with unlimited computation. Note that the most query-efficient algorithms for submodular function minimization are exponential time as well [24], so this is not uncommon when prioritizing query complexity.

1.2 Technical Highlights

Our results follow from a wide array of techniques. Some results (e.g. our $\tilde{\Omega}(n)$ lower bound on randomized algorithms for c-approximate max-cut and $c > 1/2$) follow from direct constructions of hard distributions – we defer further discussion of these constructions to the corresponding technical sections and appendices. Some of our results follow from novel techniques that are likely of independent interest for subsequent work – we quickly highlight these below.[3]

The key ingredient in our $\Omega(n^2)$-lower bound for exact max-cut is the *cut dimension* introduced by [20] for min-cuts (which has also been studied in follow-up work [28]). The lower bound follows immediately after establishing that the complete graph on n vertices has max-cut dimension $\Omega(n^2)$. More interestingly, we extend the concept of the cut-dimension to what is roughly a notion of "c-approximate cut-dimension" using strong LP duality. We show that this technique provides a lower bound on the number of queries needed by a deterministic algorithm to find a c-approximate max-cut for $c > 1/2$, and that for the complete graph on n vertices this gives a bound of $\Omega(n)$. This approach should also be of independent interest, given recent interest in the exact cut dimension. Completing this analysis also requires a technical lemma (Lemma 13) stating a simple geometric property of the Boolean hypercube, which may additionally be of independent interest.

Our $\tilde{O}(n)$-query randomized c-approximation for $c \in (1/2, 1)$ follows from a query-efficient sparsifier for global cuts in undirected, weighted graphs (once we have learned a sparsifier for the graph, we can just exhaust to find its max-cut, which is a $(1-\varepsilon)$-approximation). A query-efficient sparsifier for unweighted graphs appears in [31], based on ideas by [7]. We first give a natural extension of their sparsifier that accommodates graphs with weights in $[1, \text{poly}(n)]$. We then go a step further and provide a novel query-efficient sparsifier for *all*

[2] In more detail, the hard distribution is a random complete bipartite graph with edge weights 1. For this distribution in the value estimation setting, an algorithm could just immediately output $n^2/4$ without making any queries at all.

[3] Additionally, some of our results are reasonably straightforward (e.g. our $O(\log n)$ deterministic algorithm for c-approximate max-cut when $c < 1/2$) – we include discussion of these results in Appendices B, C, and E of the full version for completeness.

weighted graphs, using other ideas from [7] relating edge strengths to maximum spanning trees. For completeness, we show in Appendix D of the full version that in fact a slight modification of the [31] sparsifier also suffices for approximate max-cut for all weighted graphs (however, this modification may not produce a sparsifier). Our stronger sparsifier is hence not "necessary" to resolve approximate max-cut, but is of independent interest as not all weighted graph problems can be reduced to one with weights in $[1, \text{poly}(n)]$ (e.g. exact min-cut). In fact, our stronger sparsifier is already used in recent work [30].[4]

1.3 Related Work

Learning graphs in the cut-query model. The first natural question when an algorithm has restricted query access to a graph is how many queries it needs to learn the entire graph. For the case of cut queries, this was first studied by [21], who show that the query complexity of deterministically learning an unweighted graph is $\Theta(n^2/\log n)$, and that this can be attained non-adaptively. A later line of work [16, 9, 15, 10] answered this question for weighted graphs while also being sensitive to the number m of edges in the graph with nonzero weight, finding that the query complexity in this case is $\Theta(m \log n/\log m)$.

Min-cut in the cut-query model. A very active line of work is that of understanding what properties an algorithm can learn about a graph that it only has query access to, without using up enough queries to learn the entire graph. The particular case of most relevance to this work is that of min-cut in the cut-query model. The first progress on this problem was the algorithm by [31] for *unweighted* graphs using $\tilde{O}(n)$ queries. This was later improved to $O(n)$ queries by [4]. For *weighted* graphs, an algorithm using $\tilde{O}(n)$ cut queries was presented by [30] (as previously noted, their result leverages a query-efficient sparsifier for weighted graphs, which our paper is the first to provide).

On the lower bound side, [20] show using a linear algebraic argument that $\Omega(n)$ queries are necessary for weighted graphs (the constant-factors have since been improved by [28]), thus resolving the query complexity of exact min-cut in weighted graphs up to logarithmic factors. As mentioned earlier, our deterministic lower bounds arise from adapting and extending the concept of the cut dimension to approximation.

The cut dimension was introduced by [20], and further studied in [28]. [28] further nail down that undirected graphs have a min-cut dimension of no more than $2n-3$ (and there exist graphs realizing this bound). They also introduce the ℓ_1-approximate cut dimension. The ℓ_1-approximate cut dimension is motivated by a similar thought experiment as our "c-approximate cut dimension" technique – both consider an adversary changing the original graph in a way that respects all queries made, and both consider the magnitude of the changes made to the max/min-cut (rather than just whether or not the cut is changed). However, their ℓ_1-approximate cut dimension does not imply lower bounds on the query complexity of finding approximately-optimal cuts. Finally, they further consider "the dimension of approximate min-cuts", and prove a linear upper bound on this. This concept does not have any relation to our approximate cut dimension technique (and in particular, only our technique implies lower bounds on the deterministic query complexity). In summary, the cut dimension is an active concept of study in related work, but our work is the first to use this concept to lower bound the query complexity of approximately-optimal cuts.

[4] [30, Theorem 5.2] seems to credit [31] for handling weighted graphs (whereas their query-efficient sparsifier only accommodates unweighted graphs). They use this result as part of their $\tilde{O}(n)$ query algorithm for *weighted* min-cut.

Other graph algorithms in other query models. Many works have considered other graph problems in other query models as well. These query models include cut queries [14], linear measurements [1, 5], matrix-vector products [32], OR queries [5, 8], XOR queries [8], and AND queries [8]. Some of these models e.g. linear measurements and matrix-vector products generalize the cut query model, while the others do not appear directly related. These works also consider a variety of graph problems, including connectivity [1, 32, 5, 4, 14], bipartiteness [1], triangle detection [32], minimum spanning tree [1], and maximum matching [1, 8].

Graph sparsification. [31] also provide an algorithm for graph sparsification in $\tilde{O}(n)$ queries for unweighted graphs, based on the idea of *edge strength-based sampling* used by [7] to construct sparsifiers in the classical computational model. There are also other lines of work that adapt the ideas from [7] to construct sparsifiers in other limited-access computational models, particularly (semi-)streaming [1, 2, 3, 25, 32]. Some of these algorithms are adaptable to the cut-query model. However, these algorithms either only support unweighted graphs or incur a performance cost for weighted graphs. For example, the natural extension of the algorithm in [31] to weighted graphs requires $\tilde{O}(n \log W)$ queries, where W is the ratio between the largest and smallest nonzero edge weight in the graph. Our novel sparsifier avoids this pitfall by using results from [7] connecting edge strengths to maximum spanning trees to remove the $\log W$ factor.

While these results and ours all use randomization to construct a sparsifier, there are also classical algorithms for constructing sparsifiers deterministically [6, 17]. Implementing such an algorithm directly would require being able to learn the spectrum of the graph's Laplacian. It may indeed be possible to do this (exactly or approximately) in a query-efficient way, but doing so would require significantly new ideas. A deterministic query-efficient implementation of these algorithms would also yield a deterministic query-efficient algorithm for a $(1-\epsilon)$-approximation to max cut.

Max-cut in the classical computational model. Max-cut is extremely well-studied in the computational model, but algorithms in the cut-query model differ significantly. For example, [19] present a randomized polynomial-time algorithm using semidefinite programming that achieves a ≈ 0.878 approximation in expectation. However, this algorithm does not imply anything in our model; even formulating the necessary SDP requires access to the entire graph.

It has since been shown by [23] that it is NP-hard to beat a $16/17 \approx 0.941$ approximation. Moreover, [27] show that it is NP-hard to beat the constant achieved by Goemans and Williamson assuming the unique games conjecture [26]. These lower bounds do not imply anything in our model either. In fact, many of our algorithms (and those in prior work) involve exponential computation even if they use an efficient number of queries. Once we have either learned the entire graph ($c = 1$) or constructed a sparsifier of the graph ($c \in (1/2, 1)$), we use brute force search to find the exact max cut on the graph we have learned.

Submodular function maximization. Our problem can be viewed as a special case of symmetric submodular function maximization, by taking $f(\cdot)$ to be the cut function of a graph. However, the additional structure imposed by the cut function of a graph opens up possibilities for new algorithms and requires new lower bound arguments. For example, when $c > 1/2$, [18] show that achieving a c-approximation to symmetric submodular function maximization requires $\exp(\Omega(n))$ queries to the oracle. For global max-cut, the situation is significantly different: $O(n^2)$ queries suffice to trivially learn the entire function (and therefore, there seems to be little hope of embedding hard SFM instances as graphs). Still, we show that this trivial algorithm is optimal among deterministic algorithms when $c = 1$, and show that $\tilde{\Omega}(n)$ queries are necessary for even a randomized algorithm to achieve a c-approximation for $c > 1/2$.

For $c \leq 1/2$, randomized [12] and deterministic [11] algorithms are known that achieve a $1/2$-approximation with $O(n)$ and $O(n^2)$ queries respectively, even when the submodular function is not symmetric. [18] show in the symmetric case that picking a random set ($O(1)$ queries) achieves a $1/2$-approximation in expectation. They also provide a deterministic algorithm based on local search that achieves a c-approximation for any $c < 1/2$ in $\tilde{O}(n^3)$ queries. When restricting to max-cut, [18] immediately implies a randomized algorithm for all $c \leq 1/2$ with $O(1)$ queries (which is optimal). For deterministic algorithms, however, our bound of $\Theta(\log n)$ is again an exponential improvement over the general case of submodular function maximization.

In summary, global max-cut demonstrates a similar phase transition at $c = 1/2$ as general submodular function maximization: the query complexity for $c < 1/2$ is exponentially smaller than the query complexity for $c > 1/2$. However, the distinction for max-cut is between logarithmic and polynomial queries, whereas the distinction for general submodular functions is between polynomial and exponential.

Submodular function minimization. Just as submodular function maximization generalizes max-cut in the cut query model, submodular function minimization generalizes min-cut. Unlike the maximization case which requires exponentially many queries, general submodular function minimization can be solved exactly with polynomially many queries; state-of-the-art algorithms use $\tilde{O}(n^2)$ queries [24, 29].

However, until relatively recently, most query lower bounds for submodular function minimization were only $\Omega(n)$ [22, 20, 28]. The last two results proceed using the aforementioned cut dimension technique. This was recently improved to $\Omega(n \log n)$ by [13], using different ideas unrelated to graph cuts.

2 Preliminaries

G is an undirected, weighted graph with weights $w_{i,j}$ on the edge (i, j). G induces a cut function F. We denote the cut function by $F(\cdot)$, so $F(S) := \sum_{i \in S, j \notin S} w_{i,j}$. Additionally, for any cut $S \subseteq [n]$, we define the indicator vector $v_S \in \mathbb{R}^{\binom{n}{2}}$ by $(v_S)_{i,j} = 1$ if (i, j) crosses the cut defined by S and 0 otherwise.

We consider algorithms that have black-box access to $F(\cdot)$, and aim to find $\arg\max_{S \subseteq [n]}\{F(S)\}$ (exact maximization) or a set T such that $F(T) \geq c \cdot \max_{S \subseteq [n]}\{F(S)\}$ (c-approximation). We refer to the query complexity as the number of queries to $F(\cdot)$ that the algorithm makes (the algorithm has no other access to G or F, and may perform unlimited computation).

3 Lower Bound for Deterministic $(1/2 + \epsilon)$-approximation

In this section, we extend the cut dimension technique by [20] using linear programming and the strong duality theorem to show the deterministic hardness part of Theorem 4. (We refer the reader to Appendix A of the full version for a discussion of this technique and its direct application to show the deterministic lower bound in Theorem 3.) Our exact result is as follows:

▶ **Theorem 6.** *Suppose we have $c \in (1/2, 1), \epsilon \in (0,1), \alpha > 0$ such that $c > \frac{1}{1+\epsilon^2}$ and $\alpha < \frac{(1-\epsilon)^3}{108(1+\epsilon)}$. Then for n sufficiently large, αn queries do not suffice for a deterministic algorithm to estimate the max cut value within a factor of c.*

Before proving this theorem, we state a cleaner lower bound as a corollary:

▶ **Corollary 7.** *For $c \in (1/2, 1)$, a deterministic algorithm that estimates the max cut value within a factor of c requires at least $n(\frac{(\sqrt{c}-\sqrt{1-c})^4}{108c(2c-1)} - o(1))$ queries.*

This implies that the query complexity for a deterministic algorithm to achieve a c-approximation for global max-cut on a weighted undirected graph in the value estimation setting is $\Omega(n)$.

Proof. See Appendix F.1 of the full version. ◀

The first step is conceptually similar to the cut dimension argument from Theorem A.1 in the full version. We consider an adversary that answers all queries as if the graph were K_n (there is an edge of weight 1 between all pairs of vertices). Then we would like to find a perturbation z to the weight vector of K_n such that $w, w+z$ agree on all queries but have differing max cut values. The difference here is that we would like to show that the algorithm cannot even achieve a c-approximation, so we require the perturbation to be so large that the max cut value of $w+z$ is a multiplicative factor greater than the max cut of w. To do this, we will have to go beyond linear algebraic tools and consider linear programming instead. Our argument comprises the following steps:

1. Write the conditions we require of the perturbation z as linear constraints and thus formulate a linear program LP1, which we would like to show has high value.
2. LP1 works with vectors in $\mathbb{R}^{\binom{n}{2}}$ that represent cuts, which are unwieldy and unnatural. Rewrite this in terms of indicator vectors in $\mathbb{R}^n$.
3. Define another linear program LP2 and show that a high value for LP2 implies that LP1 must also have high value.
4. Show that LP2 has high value by taking its dual, and showing that the dual has high value. This comes down to showing a key technical lemma, which essentially states that the n-dimensional hypercube cannot be covered by an ℓ_1 neighborhood of an αn-dimensional subspace of $\mathbb{R}^n$. We show this using an ℓ_1 ϵ-net argument.

We now work through each step in detail. We retain all notation used in Appendix A of the full version for the cut dimension argument, and introduce additional notation as necessary.

3.1 Step 1: Formulating LP1

Throughout these proofs, we use $\mathbf{1}$ to denote a vector with all entries equal to 1 in either $\mathbb{R}^n$ or $\mathbb{R}^{\binom{n}{2}}$. It will be clear from context which of these we are referring to at any given time, but for now we are taking $\mathbf{1} \in \mathbb{R}^{\binom{n}{2}}$ to denote the weight vector of K_n. Let $q = \alpha n$ and $Q_1, Q_2, \ldots, Q_q \subseteq [n]$ be the q queried cuts. As in Appendix A of the full version, they have the corresponding 0/1 indicator vectors $v_{Q_1}, v_{Q_2}, \ldots, v_{Q_q} \in \mathbb{R}^{\binom{n}{2}}$. We are interested in finding a perturbation z such that $\mathbf{1}, \mathbf{1}+z$ are both weighted, undirected graphs (with non-negative edge weights) and agree on all queries but have max cut values differing by a multiplicative factor.

First, we require $\mathbf{1} + z$ to define a valid graph i.e. its entries should all be non-negative since these correspond to edge weights:

$$\mathbf{1} + z \geq 0 \Leftrightarrow z \geq -\mathbf{1}. \tag{1}$$

Next, we need $\mathbf{1}, \mathbf{1} + z$ to agree on all queries. This guarantees that the algorithm cannot tell the difference between $\mathbf{1}$ and $\mathbf{1} + z$ based only on the queries made so far.

$$\mathbf{1}^T v_{Q_i} = (\mathbf{1} + z)^T v_{Q_i}, \ \forall i \Leftrightarrow z^T v_{Q_i} = 0, \ \forall i. \tag{2}$$

Finally, we would like the graph corresponding to $\mathbf{1} + z$ to have a much larger max cut value than the graph corresponding to $\mathbf{1}$. We capture this in the following definition and lemma:

▶ **Definition 8.** *Define a* near-max cut *to be any cut $C \subseteq [n]$ such that $n/2 - \sqrt{n \log n} \leq |C| \leq n/2 + \sqrt{n \log n}$.*

Note that a near-max cut is nearly a max-cut in K_n. As hinted at earlier, we will show that we can find a near-max cut C and a perturbation z to the graph that will preserve the value of all queries while blowing up the value of C by a factor of c. In this case, the algorithm cannot distinguish between K_n and the perturbed graph and thus cannot achieve a c-approximation.

▶ **Lemma 9.** *To prove Theorem 6, it suffices to show that there exists a near-max cut C such that the following linear program has value at least $\epsilon^2 n^2/4$. We call this linear program LP1.*

$$\begin{aligned} &\textit{Maximize } z^T v_C \\ &\textit{subject to } z^T v_{Q_j} = 0 \quad \textit{for all } j \in [q], \textit{ and} \\ &\qquad\qquad z \geq -\mathbf{1}. \end{aligned}$$

Proof. We have already explained how the constraints arise. To justify that the objective corresponds to a bound on the approximation ratio, see Appendix F.2 of the full version. ◀

3.2 Step 2: Rewriting LP1

As already mentioned, the vectors $v_S \in \mathbb{R}^{\binom{n}{2}}$ are unnatural and difficult to work with. Intuitively, the reason for this is that a cut only has n degrees of freedom (each vertex can be included or not included in S), but we are representing it with a vector with $\binom{n}{2}$ entries, thereby creating unwanted dependencies between entries of these vectors.

We would thus like to find a more natural parametrization of these cuts that can still be connected naturally to the vectors v_S. To construct such a parametrization, we assign to each cut S a ± 1 indicator vector $u_S \in \mathbb{R}^n$. Entries are indexed by vertices in $[n]$, and for each $i \in [n]$ we have $(u_S)_i = 1$ if $i \in S$ and -1 otherwise.

Now we connect these new indicator vectors to v_S as follows. Define the matrix $M_S \in \mathbb{R}^{n \times n}$ by $M_S = \frac{\mathbf{1}\mathbf{1}^T - u_S u_S^T}{2}$. (Note that from here onwards, $\mathbf{1}$ now refers to the vector in $\mathbb{R}^n$ with all entries equal to 1.) M_S's entries are indexed by ordered pairs of vertices. Now observe that:

$$\begin{aligned}(M_S)_{i,j} &= \frac{\mathbf{1}_i\mathbf{1}_j - (u_S)_i(u_S)_j}{2} \\ &= \frac{1 - (u_S)_i(u_S)_j}{2} \\ &= \begin{cases} 0, & i,j \in S \text{ or } i,j \notin S, \\ 1, & \text{otherwise} \end{cases} \\ &= \begin{cases} (v_S)_{i,j}, & i \neq j, \\ 0, & i = j. \end{cases}\end{aligned}$$

Thus if we flatten M_S into a vector, it consists of two copies of v_S (each unordered vertex pair (i, j) in v_S appears twice in M_S since the vertex pairs indexing M_S are ordered) and n 0's (corresponding to vertex pairs (i, i) for $i \in [n]$). This allows us to rewrite LP1 in terms of the u_S's as stated in the following lemma. For matrices A, B of the same shape, we use $\langle A, B\rangle$ to denote the matrix inner product $\operatorname{tr}(A^T B) = \sum_{i,j} A_{i,j}B_{i,j}$.

▶ **Lemma 10.** *For any C, LP1 has value $\geq \epsilon^2 n^2/4$ if and only if the following LP has value at least $\epsilon^2 n^2/2$. We call this the "matrix LP". Here, $Z \in \mathbb{R}^{n\times n}$.*

$$\begin{aligned}&\textit{Maximize } \langle Z, M_C\rangle \\ &\textit{subject to } \langle Z, M_{Q_j}\rangle = 0 \quad \textit{for all } j \in [q], \textit{ and} \\ &\qquad Z \geq -\mathbf{1}.\end{aligned}$$

Proof. See Appendix F.3 of the full version. ◀

3.3 Step 3: Defining LP2 and Connecting LP2 to LP1

For a near-max cut C, define a new linear program which we call LP2 as follows. Here $y \in \mathbb{R}^n$.

$$\begin{aligned}&\text{Maximize } y^T u_C \\ &\text{subject to } y^T u_{Q_j} = 0 \quad \text{for all } j \in [q], \\ &\qquad y^T\mathbf{1} = 0, \text{ and} \\ &\qquad -\mathbf{1} \leq y \leq \mathbf{1}.\end{aligned}$$

We claim that it suffices to show that LP2 has value at least ϵn:

▶ **Lemma 11.** *If there exists a near-max cut C such that LP2 has value at least ϵn then the matrix LP for C has value at least $\epsilon^2 n^2/2$, which would imply Theorem 6.*

Proof Sketch. Take such a near-max cut C and $y \in \mathbb{R}^n$ such that y is in the feasible region of LP2 and $y^T u_C \geq \epsilon n$. Then we claim that $Z = -yy^T$ is feasible for the matrix LP and attains a value $\geq \epsilon^2 n^2/2$. Intuitively, y can be thought of as a vector representing a "pseudo-cut" in the same way that u_S represents S. LP2 having high value means that y aligns non-trivially with C, and what we are claiming is that perturbing towards the "pseudo-cut" corresponding to y will align the graph's weights with the cut corresponding to C. We provide details in Appendix F.4 of the full version. ◀

3.4 Step 4: Showing that LP2 has High Value

Finally, we show that we can find a near-max cut C such that LP2 has value at least ϵn, which by Lemma 11 will complete the proof of Theorem 6. We do this by taking the dual of LP2, which has a simple characterization captured by the following lemma:

▶ **Lemma 12.** *Consider vectors $w, w_1, w_2, \dots, w_k \in \mathbb{R}^d$, and the following LP:*

$$\begin{aligned}
&\textit{Maximize } z^T w\\
&\textit{subject to } z^T w_i = 0 \quad \textit{for all } i \in [k], \textit{ and}\\
&\qquad\qquad -\mathbf{1} \le z \le \mathbf{1}.
\end{aligned}$$

Let $W = \operatorname{span}(w_1, w_2, \dots, w_k)$. Then the value of this LP is $\min_{v \in W} ||v - w||_1$. (If there is no such v then by this minimum we mean ∞.)

Proof. See Appendix F.5 of the full version. ◀

To use this lemma, let $V = \operatorname{span}(u_{Q_1}, \dots, u_{Q_q}, \mathbf{1})$. Then Lemma 12 tells us that LP2 has value equal to $\min_{u \in V} ||u - u_C||_1$.

Now note that V depends only on the set of queries and not at all on C. Thus we would like to show that there exists a near-max cut C such that $\min_{u \in V} ||u - u_C||_1 \ge \epsilon n$. We will show the strict version of this inequality i.e. that $\min_{u \in V} ||u - u_C||_1 > \epsilon n$. Denote by B_r the ℓ_1 ball of radius r in $\mathbb{R}^n$. Then what we want to show is that there exists a near-max cut C such that $u_C \notin V + B_{\epsilon n}$. This brings us to our key technical lemma, which has little to do with graphs and may be of independent interest:

▶ **Lemma 13.** *Let $\epsilon \in (0, 1)$ and $d \le \alpha' n$ for $\alpha' < \frac{(1-\epsilon)^3}{108(1+\epsilon)}$. Suppose D is a d-dimensional subspace of $\mathbb{R}^n$. Denote by B_r the ℓ_1 ball of radius r in $\mathbb{R}^n$. Then there exists $p \in \{-1, 1\}^n$ such that $p \notin D + B_{\epsilon n}$ and $|\mathbf{1}^T p| \le 2\sqrt{n \log n}$.*

Proof. We show this by a volume argument. Specifically, we estimate the size of $(D + B_{\epsilon n}) \cap \{-1, 1\}^n$ using an ℓ_1 ϵ-net argument and show that this must be much less than 2^n. We provide details in Appendix F.6 of the full version. ◀

With this lemma, we can complete this step and thus the proof of Theorem 6:

▶ **Corollary 14.** *For n sufficiently large, there exists a near-max cut C such that $u_C \notin V + B_{\epsilon n}$.*

Proof. See Appendix F.7 of the full version. ◀

4 Sparsifier-based Randomized Algorithms for $(1 - \epsilon)$-approximation

Here we address the randomized upper bound part of Theorem 4, namely that a $(1 - \epsilon)$-approximation can be achieved in the cut finding setting with $\tilde{O}(n)$ queries. Our algorithms are adaptive. The key notion is that of a *sparsifier*:

▶ **Definition 15.** *Given weighted graphs G, H on the same set of n vertices with non-negative weights, we say H is an ϵ-sparsifier of G if all of the following conditions hold:*

1. *H has $\tilde{O}_\epsilon(n)$ edges with nonzero weight.*
2. *For any cut $S \subseteq [n]$, we have $(1 - \epsilon)F(S; G) \le F(S; H) \le (1 + \epsilon)F(S; G)$.*

Here $F(S; G)$ denotes the value of the cut defined by S on the graph G, and similarly for $F(S; H)$.

▶ **Lemma 16.** *If an algorithm can compute an ϵ-sparsifier of G in $\tilde{O}(n/\epsilon^2)$ queries with high probability, then it can also find a $(1-2\epsilon)$-approximate max cut with no additional queries.*

Proof. Once the algorithm has a sparsifier H, it can just try all possible cuts and output the cut U maximizing $F(U;H)$. Indeed, for any other cut S, we would have $F(U;G) \geq \frac{F(U;H)}{1+\epsilon} \geq \frac{F(S;H)}{1+\epsilon} \geq \frac{(1-\epsilon)F(S;G)}{1+\epsilon} \geq (1-2\epsilon)F(S;G)$, so U is indeed a $(1-\epsilon)$-approximate max cut. ◀

Throughout this section, let $d > 5$ be constant, and let $\delta \in (1/2, 1)$ be a constant sufficiently close to 1. Then let c_0 and c_1 be sufficiently large positive constants. Also, let W_{tot} be the total weight of all edges in the graph, and W the ratio between the largest and smallest nonzero edge weights. Finally, for a vertex set $S \subseteq V(G)$, let $G[S]$ denote the vertex-induced subgraph of G on S. Note that we do not require that G be connected. We organize the remainder of this section as follows:

1. In Section 4.1, we set up and analyze the algorithmic tools necessary to adapt [31]'s algorithm to weighted graphs.
2. In Section 4.2, we present the direct adaptation of [31]'s algorithm to weighted graphs and show that it constructs a sparsifier in $\widetilde{O}(n \log W)$ queries.
3. In Section 4.3, we use ideas introduced by [7] relating edge strengths to maximum spanning trees in order to construct a sparsifier for weighted graphs in $\widetilde{O}(n)$ queries, thus eliminating the dependence of the query complexity on W.
4. Additionally, in Appendix D of the full version, we show that we can achieve a $(1-\epsilon)$-approximation for max-cut in $\widetilde{O}(n)$ queries without needing the optimizations in Section 4.3. This is achieved by essentially stopping our adaptation of [31]'s algorithm early. This will not construct a sparsifier, but it will construct something "close enough" to suffice for max-cut. Intuitively, we do this by discarding edges of weight $< W_{\text{tot}}/\text{poly}(n)$ since these will not have much effect on the max cut, thereby reducing the problem to one where $W = \text{poly}(n)$.
   Given this result, our sparsifier for weighted graphs in Section 4.3 is not necessary for max-cut specifically, but it makes for a conceptually cleaner algorithm for max-cut and may be applicable to other problems.

4.1 Setup and Algorithmic Tools

The key idea is the notion of edge strength introduced by [7].

▶ **Definition 17** (([7], as stated in [31]). *The* strong connectivity *of G, denoted $K(G)$, is the value of G's min cut.*

▶ **Definition 18** ([7], as stated in [31]). *Given an edge e in G, the strong connectivity or* edge strength *k_e of e is the maximum min cut over all vertex-induced subgraphs of G containing e:*

$$k_e = \max_{S \subseteq V: u,v \in S} K(G[S]).$$

The idea introduced by [7] and used by [31] is that subsampling edges of G with probabilities inversely proportional to their strength will give a sparsifier. We cannot do exactly this in the cut query model, but we can subsample in a way that is "close enough" to independent. We need some basic primitives to support our algorithm, and we capture all of them in the following lemma:

▶ **Lemma 19.** *There exists a data structure supporting the following operations.*

1. *InitializeDS(H): Initialize the structure's state and carry out any preprocessing needed with the starting graph H.*
2. *Contract(S): Contract a given supernode set S.*
3. *GetEdge(): Find and return an edge from H (that has not been contracted) with weight at least $\frac{2W(H)}{n^2}$. Here $W(H)$ is the total weight of not-yet-contracted edges.*
4. *GetTotalWeight(S): Return the total weight of all edges with both endpoints in S (that have not yet been contracted). S must once again be a set of supernodes.*
5. *Sample(S): Sample a random edge with both endpoints in S (that has not yet been contracted), with probability proportional to its weight. S must be a set of supernodes here as well.*

It takes $O(n)$ queries for each call to InitializeDS, $O(1)$ queries for each call to Contract, $O(\log n)$ queries for each call to GetEdge, $O(1)$ queries for each call to GetTotalWeight, and $O(\log n)$ queries for each call to Sample.

Proof. The pre-processing and queries when contracting are due to having to keep track of (super)node degrees. GetEdge and Sample can be handled using recursive bisection procedures similar to that used to prove Corollary 2.2 in [31]. We provide details in Appendix G.1 of the full version. ◀

Before we go any further, we make an important comment about how we regard contraction in our algorithms (including Lemma 19). When we contract a set of vertices, we regard that set of vertices as one supernode as usual, but we do not merge any edges that have now become parallel. So the set of edges will always be a subset of the edges from the original graph.

It will be convenient to regard our algorithms as having two separate stages, although the two stages share some ingredients. In the first stage, the algorithm iteratively subsamples and contracts the graph to estimate the strengths of all edges within a constant factor. In the second stage, the algorithm uses these edge strength estimates to construct the sparsifier, following the ideas of [7]. We address these two stages in the next two subsections respectively.

We note that these algorithms are very similar to those presented by [31]; the key difference is that whenever the sparsification algorithm in [31] subsamples the graph, it does so independently for each edge. This works because [31] focuses on unweighted graphs. With weighted graphs, we would like to sample edges proportionately to their weight, and this cannot be done independently without knowledge of the graph. We modify their subsampling procedures to obtain algorithms that do not sample completely independently, but still have the concentration properties that we need.

4.1.1 Constant-Factor Edge Strength Estimates

We next describe the main tool of our edge strength estimation, which we call EstimateAndContract. It takes in the input graph G where some vertex sets S_i have already been contracted and a strength parameter κ, and further contracts G while also estimating the strength of any edges that get contracted. For any graph H, let $W(H)$ denote the total edge weight of H.

We state some key properties of Algorithm 4.1 and defer their proofs to Appendix G.4 of the full version:

▶ **Lemma 20** (Analogous to claim 3.6 from [31]). *With probability $1 - O(n^{1-d})$, for all e such that $k_e \geq \kappa$ and e is not contracted by any of the sets in $\mathcal{C}$, it will be assigned $k'_e = \kappa/2$ and contracted.*

Algorithm 4.1 EstimateAndContract$(G, \mathcal{C}, \kappa, X)$.

```
Data: Initial graph $G$ with vertex set $V(G) \subseteq [n]$, disjoint collection $\mathcal{C}$ of contracted
      sets, strength parameter $\kappa$, partial list $X$ of strength estimates
Result: Updated collection of contracted sets $\mathcal{C}$, updated list $X$ of strength estimates
Let $G'$ be $G$ with all sets from $\mathcal{C}$ contracted and $\lambda = \frac{c_0 \log^2 n}{\kappa} W(G')$. (We can find
 $W(G')$ using GetTotalWeight from Lemma 19.)
1. Sample $\lambda$ edges from $G'$, proportionally to their weights, with replacement.
   If an edge $e$ is sampled at least once, assign weight $\frac{w(e)}{p(e)}$ to it, where
   $p(e) = 1 - (1 - \frac{w_e}{W(G')})^\lambda$. These newly weighted edges form a new graph $G''$.
2. While there exists a connected component of $G''$ with a cut of size $\leq (1-\delta)\kappa$,
   delete all edges of that cut from $G''$. Let the resulting connected components of
   $G''$ be $C_1, C_2, \ldots, C_r$.
3. For each $i \in [r]$, append the tuple $(C_i, \kappa/2)$ to $X$. (Here, what we are saying is
   "assign a strength estimate of $\kappa/2$ to any edge with both endpoints in $C_i$ that
   has not already been contracted", but we phrase it differently to account for the
   fact that the algorithm may not actually know these edges.)
4. Add $C_i$ to $\mathcal{C}$ (and remove any subsets of $C_i$ to maintain disjointness) for all $i \in [r]$.
```

▶ **Lemma 21** (Analogous to claim 3.7 from [31]). *Assume that any edge contracted by $\mathcal{C}$ has strength $\geq \kappa/2$. Then with probability $1 - O(n^{1-d})$, no edge e such that $k_e < \kappa/2$ is assigned a strength estimate or contracted.*

4.1.2 Sparsifier Construction

Next, we describe our algorithm that will construct a sparsifier if provided constant-factor strength estimates for all edges in the graph, which we call ConstructSparsifier.

We capture the desired sparsification properties in the following lemma:

▶ **Lemma 22.** *Fix $X = [(C_1, \beta_1), (C_2, \beta_2), \ldots, (C_r, \beta_r)]$ as in ConstructSparsifier (X might be random, but we condition on a particular list of values for now). Then for each i, define:*

$$E(C_i) = \{e \in G : \text{ both endpoints of } e \text{ are in } C_i\}$$
$$\widetilde{E}(C_i) = E(C_i) \setminus \bigcup_{j : C_j \subset C_i} E(C_j)$$

(Thus $\widetilde{E}(C_i)$ is the set of edges that will be contracted at the time that C_i is contracted.)

Assume each of the following conditions:

1. *Each connected component of G is contained in at least one C_i (every edge in G gets contracted), and*
2. *For all i and edges $e \in \widetilde{E}(C_i)$, we have $\beta_i \in [k_e/4, k_e]$ (edge strength estimates are correct within a constant factor).*

Then with probability $1 - O(n^{-d})$, ConstructSparsifier will output a sparsifier H that approximates the cuts of G within a factor of $1 \pm 2\epsilon$. (Thus this probability is only considering the randomness of ConstructSparsifier.)

Proof. This follows using similar ideas to [7], but we need to take some extra care because the subsampling we use to construct the sparsifier is not independent. We provide details in Appendix G.2.2 of the full version. ◀

Algorithm 4.2 ConstructSparsifier(G, X).

```
Data: Graph $G$ with vertex set $[n]$, list $X = [(C_1, \beta_1), (C_2, \beta_2), \ldots, (C_r, \beta_r)]$ of
      strength estimates. We assume that $\{C_1, \ldots, C_r\}$ is a laminar family of
      vertex sets and that if $C_i \subset C_j$ then $i < j$. (This is because we will be
      contracting $C_1, \ldots, C_r$ in that order.) Note that we do not assume here that
      $X$ includes a strength estimate for every edge.
Result: Potential sparsifier $H$
Initialize $G' = G$ and $H$ to be empty.
InitializeDS($G'$). (This is just to reset and ignore any previous contractions we may
  have done in EstimateAndContract.)
for $i \leftarrow 1$ to $r$ do
    1. Let $\mu_i = \frac{c_1 \log^2 n}{\epsilon^2 \beta_i} W(C_i)$. (Here $W()$ is with respect to $G'$, and once again can
       be found using GetTotalWeight from Lemma 19.)
    2. Sample $\mu_i$ edges from $C_i$ proportionally to their weights, with replacement.
       If an edge $e$ is sampled at least once, add it to $H$ with weight $\frac{w_e}{p_e}$, where
       $p_e = 1 - (1 - \frac{w_e}{W(C_i)})^{\mu_i}$ is the probability that $e$ is sampled at least once.
       (Note that this sampling would be done by calling Sample($C_i$) from Lemma 19.)
    3. Contract $C_i$ in $G'$.
end
```

4.2 Sparsification with $\widetilde{O}(n \log W)$ Queries

Here we analyze the naive generalization of [31]'s sparsifier to weighted graphs. The procedure is described in Algorithm 4.3.

Algorithm 4.3 NaiveWeightedSubsample(G, T).

```
Data: Graph $G$ on $n$ vertices, positive real parameter $T$
Result: A potential $(2\epsilon)$-sparsifier $H$ of $G$.
Initialize $\mathcal{C} = \emptyset$ and $X$ as an empty list.
InitializeDS($G$).
Find $W_{\text{tot}}$ by running GetTotalWeight($G$).
Initialize $\kappa$ to be the smallest power of 2 that is at least $W_{\text{tot}}$.
while $GetTotalWeight(G) > 0$ and $\kappa > W_{tot}/T$ do
    EstimateAndContract($G, \mathcal{C}, \kappa, X$)
    $\kappa \leftarrow \kappa/2$
end
$H \leftarrow$ ConstructSparsifier($G, X$).
```

▶ **Theorem 23.** *NaiveWeightedSubsample(G, ∞) runs in $O(n \log^3 n(\log n + \log W + \frac{1}{\epsilon^2}))$ queries and outputs a (2ϵ)-sparsifier H with probability $1 - O(n^{1-d}(\log n + \log W))$.*

Proof. We outline the proof here and provide details in Appendices G.5 (correctness) and G.8 (efficiency) of the full version. The proof proceeds in two parts.

First, we address the calls to EstimateAndContract. Given Lemmas 4.6 and 4.7, EstimateAndContract can be thought of as estimating the strengths of and then contracting edges whose strengths are within a window that is a factor of 4 wide. The lemmas tell us that these strength estimates are accurate within a factor of 4. Then NaiveWeightedSubsample essentially "slides" this window across all possible edge strengths so that all edges of G are assigned a strength estimate. This part has query complexity $O(n \log^3 n(\log n + \log W))$; the $\log W$ term is because the outer loop could run for $O(\log n + \log W)$ iterations. This is also why the success probability depends on W, as this is obtained by taking a union bound over all iterations.

Secondly, because all edges are assigned an accurate strength estimate (within a constant factor), Lemma 22 tells us that ConstructSparsifier will output a (2ϵ)-sparsifier with high probability. This part has query complexity $O(n \log^3 n/\epsilon^2)$. ◀

4.3 Sparsification with $\widetilde{O}(n)$ Queries

We now show how to eliminate the dependence of the query complexity and success probability on W, thus constructing sparsifiers in $\widetilde{O}(n)$ queries. We begin by setting up the necessary ideas.

4.3.1 Crude Edge Strength Estimates

Recall that the key problem with Algorithm 4.3 was that our "sliding window" for edge strength estimation could potentially repeat $O(\log n + \log W)$ times. The key idea is to mitigate this by finding very crude (within a factor of n^4) estimates for the edge strengths for all edges in G, before refining these estimates using EstimateAndContract. We do this using the idea of [7] to estimate edge strengths from the maximum spanning forest (MSF) of G. Fix an MSF $\mathcal{T}$ of G. Then for any edge e with endpoints i, j, define $d_e = d_{i,j}$ to be the minimum weight of an edge on the MSF path between the endpoints of e. We first state a lemma shown in [7]:

▶ **Lemma 24** ([7]). *For all edges e, we have $d_e \leq k_e \leq n^2 d_e$.*

This would immediately give us sufficient edge strength estimates but we do not know of a way to find the MSF efficiently in the cut query model. In fact, we can adapt the max cut dimension argument from Lemma A.3 in the full version to a "max tree dimension" argument to show that deterministically finding the MSF requires $\Omega(n^2)$ queries; see Appendix A.1 of the full version for details. So instead, we run what we call "approximate Kruskal" using the primitives we have available from Lemma 19.

Algorithm 4.4 ApproximateKruskal(G).

```
Data: Graph $G$ on $n$ vertices
Result: A forest $\widetilde{\mathcal{T}}$ using the edges of $G$.
Initialize $\widetilde{\mathcal{T}}$ to be the empty graph on $n$ vertices.
InitializeDS($G$).
while $GetTotalWeight(G) > 0$ do
    $e$ = GetEdge()
    Contract($e$)
    Add $e$ to $\widetilde{\mathcal{T}}$.
end
```

It follows from Lemma 19 that ApproximateKruskal can be run in $O(n \log n)$ queries. Once we run ApproximateKruskal, we will have a spanning forest of G so we will also know its connected components. Moreover, the following lemma tells us that even this crude approximation to the MSF suffices to give us edge strength estimates. We defer the proofs of this lemma and its straightforward corollary to Appendix G.6.1 of the full version.

▶ **Lemma 25.** *For any distinct i, j, define $\tilde{d}_{i,j}$ as follows:*

- *If i, j are connected in G, then let $\tilde{d}_{i,j}$ be the minimum weight of an edge on the path in $\widetilde{\mathcal{T}}$ connecting i and j.*
- *Otherwise, let $\tilde{d}_{i,j} = 0$.*

Then for any i, j that are connected in G, we have $\frac{2}{n^2} d_{i,j} \le \tilde{d}_{i,j} \le d_{i,j}$ for all i, j.

Note that we only assume here that $\mathcal{T}$ is maximal; any assumptions we make about $\widetilde{\mathcal{T}}$ are baked into GetEdge.

▶ **Corollary 26.** *For all distinct i, j, we have $k_{i,j} \in [\tilde{d}_{i,j}, \frac{n^4}{2}\tilde{d}_{i,j}]$.*

4.3.2 Fast Weighted Subsampling

We now describe how to use our crude edge strength estimates to construct a sparsifier in $\tilde{O}(n)$ queries. The idea is that by Lemma 21, any edges with strength $< \kappa/2$ will be deleted in Step 2 of Algorithm 4.1. But we can use our crude edge strength estimates to preemptively identify some edges that will definitely be deleted, and then delete these edges to disconnect the graph a bit *before* running EstimateAndContract. We describe this procedure in Algorithm 4.5.

Algorithm 4.5 FastWeightedSubsample(G).

```
Data: Graph $G$ on $n$ vertices
Result: A potential $(2\epsilon)$-sparsifier $H$ of $G$.
Run ApproximateKruskal on a copy of $G$ and calculate $\tilde{d}_{i,j}$ for all $i, j$.
Initialize $L$ to a list of all nonzero values of $\tilde{d}_{i,j}$.
Initialize $\mathcal{C} = \emptyset$, $\kappa = \infty$, and $X$ as an empty list.
InitializeDS($G$).
while $L \neq \emptyset$ do
    Let $\tilde{D} = \max L$ and $C = \{(i,j) : \tilde{d}_{i,j} < \tilde{D}/(2n^5)\}$.
    Let $S_1, \dots, S_r$ be the connected components of $K_n$ after we remove all edges from
      $C$.
    Let $\kappa'$ be the smallest power of 2 that is at least $n^4\tilde{D}/2$.
    $\kappa \leftarrow \min(\kappa, \kappa')$
    while $\kappa \ge \tilde{D}/(2n)$ do
        for $i \leftarrow 1$ to $r$ do
            EstimateAndContract($G[S_i], \mathcal{C}, \kappa, X$)
            Remove all contracted edges from $L$.
        end
        $\kappa \leftarrow \kappa/2$
    end
end
$H \leftarrow$ ConstructSparsifier($G, X$).
```

We make one comment here about FastWeightedSubsample: for the algorithm as presented to even be well-defined, we need to check that each S_i at any stage of the algorithm is the union of some collection of supernodes. If not, it does not make sense to run EstimateAndContract on each $G[S_i]$. This condition is also necessary to ensure the applicability of Lemma 19 to each call to EstimateAndContract; Contract, GetTotalWeight, and Sample all require that their input S be a union of supernodes. We defer the verification of this and the proof of our final theorem to Appendices G.7 (correctness) and G.8 (efficiency) of the full version:

▶ **Theorem 27.** *FastWeightedSubsample runs in $O(n \log^3 n(\log n + \frac{1}{\epsilon^2}))$ queries and outputs a (2ϵ)-sparsifier H with probability $1 - O(n^{5-d})$.*

▶ **Corollary 28.** *For any $\epsilon > 0$, the query complexity for a randomized algorithm to construct an ϵ-sparsifier with $1-o(1)$ probability for a weighted undirected graph is $\widetilde{O}(n)$. This algorithm is adaptive.*

By Lemma 16, this yields our desired algorithmic result for max-cut:

▶ **Corollary 29.** *For any $c < 1$, the query complexity for a randomized algorithm to achieve a c-approximation with $1 - o(1)$ probability for global max-cut on a weighted undirected graph in the cut finding setting is $\tilde{O}(n)$. This algorithm is adaptive.*

We remind the reader that Theorem 27 extends the query-efficient sparsifier of [31] to weighted graphs (as we saw in Section 4.2, a naive generalization of [31] requires $\widetilde{O}(n \log W)$ queries).

References

1. Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 459–467. SIAM, 2012. doi:10.1137/1.9781611973099.40.
2. Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors, *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. ACM, 2012. doi:10.1145/2213556.2213560.
3. Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In Prasad Raghavendra, Sofya Raskhodnikova, Klaus Jansen, and José D. P. Rolim, editors, *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*, volume 8096 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2013. doi:10.1007/978-3-642-40328-6_1.
4. Simon Apers, Yuval Efron, Pawel Gawrychowski, Troy Lee, Sagnik Mukhopadhyay, and Danupon Nanongkai. Cut query algorithms with star contraction. *CoRR*, abs/2201.05674, 2022. arXiv:2201.05674.
5. Sepehr Assadi, Deeparnab Chakrabarty, and Sanjeev Khanna. Graph connectivity and single element recovery via linear and OR queries. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)*, volume 204 of *LIPIcs*, pages 7:1–7:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ESA.2021.7.
6. Joshua D. Batson, Daniel A. Spielman, and Nikhil Srivastava. Twice-ramanujan sparsifiers. *SIAM Rev.*, 56(2):315–334, 2014. doi:10.1137/130949117.
7. András A. Benczúr and David R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *SIAM J. Comput.*, 44(2):290–319, 2015. doi:10.1137/070705970.
8. Joakim Blikstad, Jan van den Brand, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Nearly optimal communication and query complexity of bipartite matching. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022, Denver, CO, USA, October 31 - November 3, 2022*, pages 1174–1185. IEEE, 2022. doi:10.1109/FOCS54457.2022.00113.
9. Nader H. Bshouty and Hanna Mazzawi. Toward a deterministic polynomial time algorithm with optimal additive query complexity. *Theor. Comput. Sci.*, 417:23–35, 2012. doi:10.1016/J.TCS.2011.09.005.

10 Nader H. Bshouty and Hanna Mazzawi. On parity check (0, 1)-matrix over $\mathcal{F}_p$. *SIAM J. Discret. Math.*, 29(1):631–657, 2015. `doi:10.1137/120881129`.

11 Niv Buchbinder and Moran Feldman. Deterministic algorithms for submodular maximization problems. *ACM Trans. Algorithms*, 14(3):32:1–32:20, 2018. `doi:10.1145/3184990`.

12 Niv Buchbinder, Moran Feldman, Joseph Naor, and Roy Schwartz. A tight linear time (1/2)-approximation for unconstrained submodular maximization. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS 2012, New Brunswick, NJ, USA, October 20-23, 2012*, pages 649–658. IEEE Computer Society, 2012. `doi:10.1109/FOCS.2012.73`.

13 Deeparnab Chakrabarty, Andrei Graur, Haotian Jiang, and Aaron Sidford. Improved lower bounds for submodular function minimization. *CoRR*, abs/2207.04342, 2022. `doi:10.48550/arXiv.2207.04342`.

14 Deeparnab Chakrabarty and Hang Liao. A query algorithm for learning a spanning forest in weighted undirected graphs. In Shipra Agrawal and Francesco Orabona, editors, *International Conference on Algorithmic Learning Theory, February 20-23, 2023, Singapore*, volume 201 of *Proceedings of Machine Learning Research*, pages 259–274. PMLR, 2023. URL: `https://proceedings.mlr.press/v201/chakrabarty23a.html`.

15 Sung-Soon Choi. Polynomial time optimal query algorithms for finding graphs with arbitrary real weights. In Shai Shalev-Shwartz and Ingo Steinwart, editors, *Proceedings of the 26th Annual Conference on Learning Theory*, volume 30 of *Proceedings of Machine Learning Research*, pages 797–818, Princeton, NJ, USA, 12–14 June 2013. PMLR. URL: `https://proceedings.mlr.press/v30/Choi13.html`.

16 Sung-Soon Choi and Jeong Han Kim. Optimal query complexity bounds for finding graphs. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 749–758. ACM, 2008. `doi:10.1145/1374376.1374484`.

17 Marcel Kenji de Carli Silva, Nicholas J. A. Harvey, and Cristiane M. Sato. Sparse sums of positive semidefinite matrices. *ACM Trans. Algorithms*, 12(1):9:1–9:17, 2016. `doi:10.1145/2746241`.

18 Uriel Feige, Vahab S. Mirrokni, and Jan Vondrák. Maximizing non-monotone submodular functions. *SIAM J. Comput.*, 40(4):1133–1153, 2011. `doi:10.1137/090779346`.

19 Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, 1995. `doi:10.1145/227683.227684`.

20 Andrei Graur, Tristan Pollner, Vikram Ramaswamy, and S. Matthew Weinberg. New query lower bounds for submodular function minimization. In Thomas Vidick, editor, *11th Innovations in Theoretical Computer Science Conference, ITCS 2020, January 12-14, 2020, Seattle, Washington, USA*, volume 151 of *LIPIcs*, pages 64:1–64:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. `doi:10.4230/LIPIcs.ITCS.2020.64`.

21 Vladimir Grebinski and Gregory Kucherov. Optimal reconstruction of graphs under the additive model. *Algorithmica*, 28(1):104–124, 2000. `doi:10.1007/s004530010033`.

22 Nicholas J. A. Harvey. *Matchings, matroids and submodular functions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2008. URL: `http://hdl.handle.net/1721.1/44416`.

23 Johan Håstad. Some optimal inapproximability results. *J. ACM*, 48(4):798–859, 2001. `doi:10.1145/502090.502098`.

24 Haotian Jiang. Minimizing convex functions with integral minimizers. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 976–985. SIAM, 2021. `doi:10.1137/1.9781611976465.61`.

25 Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 561–570. IEEE Computer Society, 2014. `doi:10.1109/FOCS.2014.66`.

26 Subhash Khot. On the power of unique 2-prover 1-round games. In John H. Reif, editor, *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 767–775. ACM, 2002. doi:10.1145/509907.510017.

27 Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O'Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable CSPs? *SIAM J. Comput.*, 37(1):319–357, 2007. doi:10.1137/S0097539705447372.

28 Troy Lee, Tongyang Li, Miklos Santha, and Shengyu Zhang. On the cut dimension of a graph. In Valentine Kabanets, editor, *36th Computational Complexity Conference, CCC 2021, July 20-23, 2021, Toronto, Ontario, Canada (Virtual Conference)*, volume 200 of *LIPIcs*, pages 15:1–15:35. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CCC.2021.15.

29 Yin Tat Lee, Aaron Sidford, and Sam Chiu-wai Wong. A faster cutting plane method and its implications for combinatorial and convex optimization. In Venkatesan Guruswami, editor, *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 1049–1065. IEEE Computer Society, 2015. doi:10.1109/FOCS.2015.68.

30 Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: sequential, cut-query, and streaming algorithms. In Konstantin Makarychev, Yury Makarychev, Madhur Tulsiani, Gautam Kamath, and Julia Chuzhoy, editors, *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020, Chicago, IL, USA, June 22-26, 2020*, pages 496–509. ACM, 2020. doi:10.1145/3357713.3384334.

31 Aviad Rubinstein, Tselil Schramm, and S. Matthew Weinberg. Computing exact minimum cuts without knowing the graph. In Anna R. Karlin, editor, *9th Innovations in Theoretical Computer Science Conference, ITCS 2018, January 11-14, 2018, Cambridge, MA, USA*, volume 94 of *LIPIcs*, pages 39:1–39:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ITCS.2018.39.

32 Xiaoming Sun, David P. Woodruff, Guang Yang, and Jialin Zhang. Querying a matrix through matrix-vector products. In Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi, editors, *46th International Colloquium on Automata, Languages, and Programming, ICALP 2019, July 9-12, 2019, Patras, Greece*, volume 132 of *LIPIcs*, pages 94:1–94:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.ICALP.2019.94.