Breaking the All Subsets Barrier for Min k-Cut

Daniel Lokshtanov
University of California Santa Barbara, CA, USA

Saket Saurabh
The Institute of Mathematical Sciences, HBNI, Chennai, India
University of Bergen, Norway

Vaishali Surianarayanan
University of California Santa Barbara, CA, USA

Abstract

In the Min k-Cut problem, the input is a graph G and an integer k. The task is to find a partition of the vertex set of G into k parts, while minimizing the number of edges that go between different parts of the partition. The problem is NP-complete, and admits a simple $3^n \cdot n^{\mathcal{O}(1)}$ time dynamic programming algorithm, which can be improved to a $2^n \cdot n^{\mathcal{O}(1)}$ time algorithm using the fast subset convolution framework by Björklund et al. [STOC'07]. In this paper we give an algorithm for Min k-Cut with running time $\mathcal{O}((2-\varepsilon)^n)$, for $\varepsilon > 10^{-50}$. This is the first algorithm for Min k-Cut with running time $\mathcal{O}(c^n)$ for $c < 2$.

2012 ACM Subject Classification Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Exact algorithms, min k-cut, exponential algorithms, graph algorithms, k-way cut

Digital Object Identifier 10.4230/LIPIcs.ICALP.2023.90

Category Track A: Algorithms, Complexity and Games

Funding *Daniel Lokshtanov*: NSF award CCF-2008838.
Saket Saurabh: European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant no. 819416), and Swarnajayanti Fellowship grant DST/SJF/MSA01/2017-18.
Vaishali Surianarayanan: NSF award CCF-2008838.

1 Introduction

A *k-cut* of a graph G is a partition of the vertex set $V(G)$ into k non-empty parts. The *weight* of a k-cut is the total number of edges with endpoints in different parts of the partition. In the Min k-Cut problem the input is a graph G on n vertices and m edges, and an integer k. The task is to find a k-cut of G of minimum weight.

The problem is known to be NP-complete [23] and is extremely well studied from the perspective of approximation algorithms [42, 47, 48], parameterized algorithms [35, 13, 15], extremal combinatorics [33, 27, 28], and more recently parameterized approximation [26, 25, 34, 40]. For all of the above perspectives the best known algorithmic results come quite close to asymtotically matching existing combinatorial or complexity theoretic lower bounds: On one hand, there are several $2(1-\frac{1}{k})$-approximation algorithms that run in time polynomial in n and k [42, 47, 48]. On the other hand, this approximation ratio cannot be improved assuming the Small Set Expansion Hypothesis (SSE) [41]. A $(1+\epsilon)$-approximation algorithm with running time $(k/\epsilon)^{\mathcal{O}(k)} n^{\mathcal{O}(1)}$ was recently obtained by Lokshtanov et al. [40], the running time of this algorithm cannot be substantially improved without violating the Exponential Time Hypothesis (ETH).

The fastest known algorithm with parameter k by Gupta et al. [25] runs in time $\mathcal{O}(f(k)n^{k+o(1)})$. At the same time, an algorithm with running time $\mathcal{O}(f(k)n^{(1-\epsilon)\frac{\omega}{3}+o(1)})$ for $\epsilon > 0$, where $\omega < 2.373$ [2] is the matrix multiplication exponent, would imply an improved algorithm for k-Clique. An $\mathcal{O}(f(k)n^{(1-\epsilon)k+o(1)})$ time algorithm for $\epsilon > 0$ for the edge weighted version of Min k-Cut would imply an improved algorithm for Max-Weight k-Clique. This has been conjectured not to exist [1, 3]. The algorithm of Gupta et al. [27, 28] also proves that the number of minimum weight k-cuts in a graph is upper bounded by $\mathcal{O}(f(k)n^{k+o(1)})$, coming very close to matching the $(n/k)^k$ lower bound obtained from the complete k-partite graph.

While exact algorithms for Min k-Cut for small values of k have received considerable attention since the early 1990's [23], exact exponential time algorithms whose running time is measured in terms of the number n of vertices only remain largely unexplored.

Naive Algorithm. A simple $3^n n^{\mathcal{O}(1)}$ time algorithm can be obtained by reduction to finding a minimum weight path on exactly k edges in the following directed acyclic graph S_G: The graph S_G has a vertex for every subset X of $V(G)$. For every pair X, Y of vertex subsets, S_G has an edge from X to Y if X is a subset of Y. The weight of this edge is the number of edges in G with one endpoint in X and the other in $Y \setminus X$. It is easy to see that S_G is a directed acyclic graph, and that there is a one to one correspondence between k-cuts in G and paths from $\emptyset$ to $V(G)$ in S_G using exactly k edges. The graph S_G has 2^n vertices and 3^n edges because every edge XY in S_G corresponds to a partition of $V(G)$ into 3 parts, namely $(X, Y \setminus X, V(G) \setminus Y)$. Finding a minimum weight path using exactly k edges in a DAG can be done in time $\mathcal{O}(k(|V(S_G)| + |E(S_G)|)) = 3^n n^{\mathcal{O}(1)}$. Note that this beats $\mathcal{O}(n^k)$ time algorithms for instances where $k \geq \frac{2n}{\log n}$.

In 2009 Björklund et al. [9] gave a general purpose approach, based on the inclusion-exclusion principle, to solve partitioning problems in $2^n n^{\mathcal{O}(1)}$ time. Their applications (see [9, Proposition 6]) include a $2^n n^{O(1)}$ time algorithm for Min k-Cut. Prior to this work the algorithm of Björklund et al. [9] has remained the state of the art.

When the best algorithm for a well-studied problem is stuck at 2^n for a long time it is prudent to ask whether it is possible to do better at all, or whether 2^n really is the best possible. On one hand, better than 2^n time algorithms have been found for a number of problems, including, among many others k-SAT [49, 46] and the satisfiability problem for a number of circuit classes [10, 39, 38], Hamiltonian Cycle [4], k-Coloring for $k \in \{3, 4, 5, 6\}$ [19, 57], Bin Packing [45], Max-Cut [51], Chordal Vertex Deletion [11], Treewidth [21] and Scheduling Partially Ordered Jobs [17]. On the other hand the failure to find better than 2^n time algorithms for the satisfiability of CNF formulas and Set Cover has led to conjectures [14, 31] that no substantially better algorithms for these problems are possible, driving the field of fine-grained complexity [53, 54, 52]. For problems such as Directed Hamiltonicity, Travelling Salesman and Chromatic Number, obtaining better than 2^n time algorithms are outstanding open problems [20, 44, 55, 56]. In this paper we give the first algorithm for Min k-Cut with running time $\mathcal{O}((2-\varepsilon)^n)$ for $\varepsilon > 0$.

▶ **Theorem 1.** *There exists an algorithm that takes as input an unweighted simple graph G on n vertices, an integer $k > 0$ and returns the min k-cut of G in time $\mathcal{O}(2-\varepsilon)^n$ for some $\varepsilon > 0$.*

Observe that Theorem 1 only considers *unweighted simple graphs.* For Min k-Cut on *weighted* graphs, previous to our work, the best algorithm was the $\mathcal{O}(2^n W^{O(1)})$ time algorithm of Björklund et al. [9]. Our naive $\mathcal{O}(3^n)$ time algorithm described above can easily be made to work for weighted graphs with running time $3^n(\log W)^{\mathcal{O}(1)}$. As part of the proof of Theorem 1 we make an alternative algorithm for weighted graphs that runs in time $2^{n+o(n)}(\log W)^{\mathcal{O}(1)}$.

▶ **Theorem 2.** *There exists an algorithm that takes as input an edge-weighted graph G on n vertices having weights on edges from $[0, W]$, an integer $k > 0$ and returns the min k-cut of G in time $2^{n+o(n)}(\log W)^{\mathcal{O}(1)}$.*

Theorem 2 is the current best algorithm for Min k-Cut for the weighted case. A natural question that arises here is whether one can improve the running time to $(2-\epsilon)^n(\log W)^{\mathcal{O}(1)}$. The answer seems to be leaning towards a no, or at the very least towards a "that's a pretty difficult question". From the perspective of $(2-\epsilon)^n$ time algorithms it is folklore that (Edge Weighted) Densest Subgraph is at least as hard as Edge Weighted Clique. In the latter problem the goal is to find a maximum weight clique in an edge weighted graph having edge weights that may be positive, negative, or even exponential in n. The best known algorithms for Edge Weighted Clique are $2^n \cdot (\log W)^{\mathcal{O}(1)}$ and $2^{\frac{\omega}{3}n}W^{\mathcal{O}(1)}$, where W is the maximum edge weight and ω is the matrix multiplication exponent. Edge Weighted Clique is a basic problem - n^k hardness of the problem is a conjecture that is sometimes used as a basis for hardness in fine grained complexity [1, 3]. From here it is not too big of a leap to conjecture that it is hard to get $(2-\epsilon)^n \cdot (\log W)^{O(1)}$ time algorithms as well. So it makes a lot more sense to focus on a $(2-\epsilon)^n \cdot (\log W)^{O(1)}$ time algorithm for Edge Weighted Clique before attempting such an algorithm for (Edge Weighted) Min k-Cut.

At a very high level our algorithm for Min k-Cut is based on a dynamic programming algorithm operating on a pruned state space having less than 2^n states. Based on how the input instance and optimal solution look, we use different methods to prune the state space. We show that the cases for which such pruning strategies don't work are "Densest t-Subgraph instances in disguise" with the optimal solution having only one non-singleton part. Here we apply the $\mathcal{O}(2^{\frac{\omega n}{3}}n^{\mathcal{O}(1)})$ time matrix multiplication based algorithm for Densest t-Subgraph by Chang et.al. [12]. The final running time is just the maximum over the running time of the algorithms we design for the various cases we consider. Our algorithm carefully balances various tools and techniques from past work on Min k-Cut, graph theory and the exact algorithms world. This includes DP table sparsification, Thorup tree packing, split and list, graph sparsification, as well as the $\binom{p+q}{p}$ combinatorial bound for the number of connected vertex sets of size p with q neighbors.

Admittedly our algorithm chops the input space into various pieces based on the properties of the input instance and the optimal solution and thus uses quite a handful of cases. One might ask to which degree this amount of case work is necessary. At the very least, the same reduction[1] that showed that *weighted* Min k-Cut is at least as hard as *weighted* Densest t-Subgraph shows that *unweighted* Min k-Cut is at least as hard as *unweighted* Densest t-Subgraph. The only known $(2-\epsilon)^n$ time algorithm [12] for Densest t-Subgraph is based on the Split and List method of Williams [51]. Therefore, the set of inputs to Min k-Cut contains both "Densest t-Subgraph instances in disguise", as well as instances that behave very differently. For an example instances with a few large cliques with few edges between each other behave much more like instances of classic "cut and separation" or clustering problems, and do not appear to be solvable by Split and List based approaches. Hence, a $(2-\epsilon)^n$ algorithm for unweighted Min k-Cut either has to invent a new method for Densest t-Subgraph (which is a very interesting research goal in and of itself) or somehow

[1] To reduce from Densest t-Subgraph to Min k-Cut, add a universal vertex and set $k = n + 2 - t$.

separate the instances of Min k-Cut into ones that are "Densest t-Subgraph instances in disguise" and the ones that can be handled by other means. Of course this only justifies the split into two cases, as opposed to our rather large case tree. It is a nice open problem whether it is possible to handle all of the non-Densest t-Subgraph instances in a more uniform way. This would likely also result in a better running time bound.

We remark that the improvement over 2^n obtained in Theorem 1 is small and holds only for unweighted simple graphs. But importantly it shows that 2^n is not the boundary for Min k-Cut. We hope that this work will initiate a line of research on exact algorithms for Min k-Cut just as the $(2-\epsilon)$-approximation by Gupta et.al [26] kick started a series of work [25, 34, 40] on FPT approximation for Min k-Cut, or the $\mathcal{O}(k^{\mathcal{O}(k)}n^{(\frac{2\omega}{3}+o(1))k})$ time algorithm of Gupta et al. [25] led to a chain of improvements [37, 27, 29, 30] for $n^{\mathcal{O}(k)}$ time algorithms for the problem.

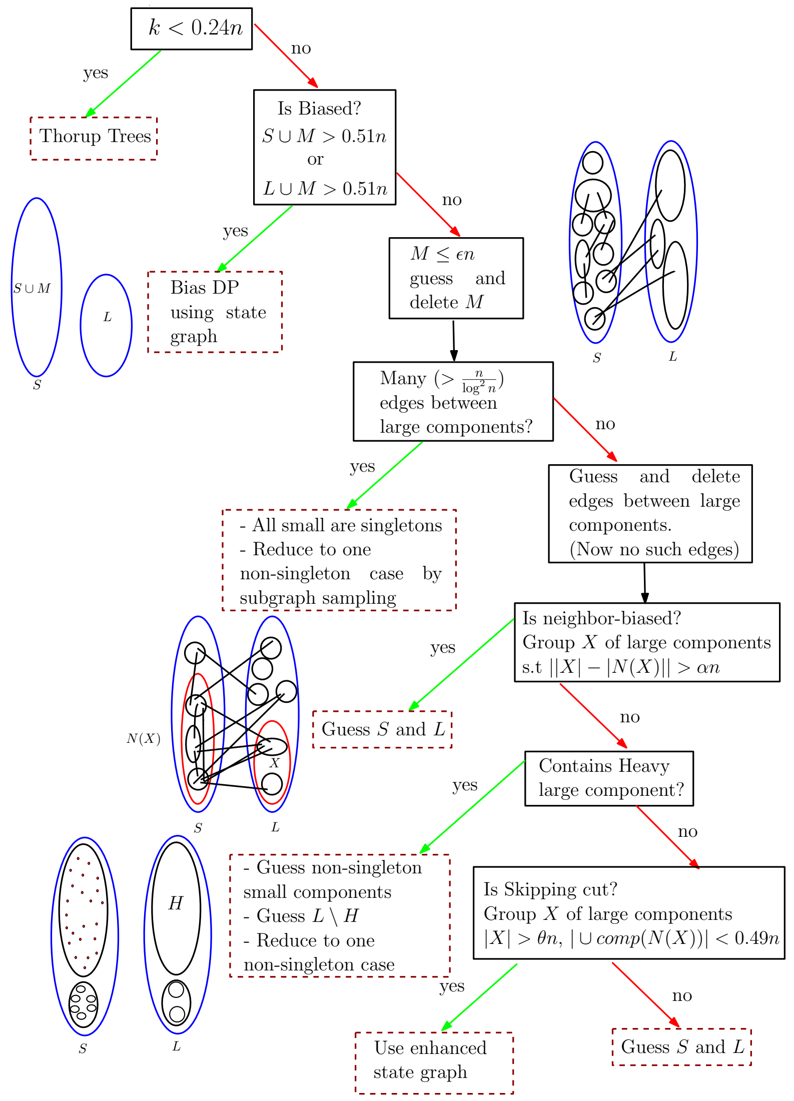
1.1 Algorithm Overview

The first substantial hurdle in designing an algorithm for Min k-Cut that beats the bound of 2^n, is that even getting an algorithm with running time 2^n is non-trivial. The Min k-Cut problem is a graph partitioning problem, and thus the first approach to design an algorithm for Min k-Cut is to explore methods developed for such problems. All known general methods for set partitioning problems, such as those developed by Björklund et al. [9, 5], rely on enumerating all vertex subsets. Known approaches to speed up such methods rely on the input graph being sufficiently sparse, such as having bounded degree or bounded average degree [6, 7, 8, 16, 24]. As we cannot make any such assumptions here we first design an alternative stand-alone $2^{n+o(n)}$ time algorithm that does not make use of the methods for set partitioning problems of Björklund et al. [9].

Our new $2^{n+o(n)}$ time algorithm is based on the fact that Min k-Cut is solvable in time $n^{\mathcal{O}(k)}$. Note that the running time of the naive algorithm, based on a reduction to finding a minimum weight path on exactly k edges in the state graph S_G, is 3^n because S_G has many edges. If we restrict ourselves to only using edges of S_G that correspond to parts of size at most $\mathcal{O}(\frac{n}{\log n})$, then the out-degree of every vertex in the state graph drops to $2^{o(n)}$, leading to an algorithm of running time $2^{n+o(n)}$, that computes for every subset $S \subseteq V(G)$ and integer $k \leq n$ the best way of partitioning S into k parts, each of size at most $\frac{n}{\log n}$. After doing this, in order to find the best partition of $V(G)$ that may or may not use large parts (i.e parts bigger than $\frac{n}{\log n}$), we go over all choices of S and find the best partition of $V(G) \setminus S$ using the $n^{\mathcal{O}(k)}$ time algorithm of Thorup [50]. Since, all the parts in $V(G) \setminus S$ are large we only need to invoke the $n^{\mathcal{O}(k)}$ time algorithm with $k = \mathcal{O}(\log n)$. In the remainder of this outline, when we talk about the state graph S_G we will refer to the *sparsified* state graph with only $2^{n+o(n)}$ edges whose paths correspond to cuts with parts of size at most $\frac{n}{\log n}$

To design an algorithm for Min k-Cut with running time $\mathcal{O}((2-\varepsilon)^n)$, for some fixed $\varepsilon > 0$, we consider many special cases and design faster algorithms for each individual case. Some of the cases are handled by standard methods, while some require interesting new insights. In Figure 1 we provide a case tree depicting the various cases our algorithm branches into - we suggest the reader to refer to the case tree while reading the overview to get a complete picture. The first case we consider is when $k < 0.24n$. Here, k is so small that one would expect ideas from the $n^{\mathcal{O}(k)}$ time algorithm to apply, but too large to use them as black boxes because this leads to a running time of $n^{\mathcal{O}(n)}$. We use the main object behind the deterministic $n^{\mathcal{O}(k)}$ time algorithm, namely Thorup trees [50].

Thorup's algorithm is based on a tree-packing theorem which allows to efficiently find a tree T that crosses the optimal k-cut at most $2k-2$ times. Using T one can design a $\binom{n}{2k}3^{2k}$ time algorithm [50]. If $k < 0.24n$ then we can pick a random edge e from the tree and declare that it is not part of the cut. We know that we fail (i.e. that we picked an edge from the optimal solution) with probability at most $\frac{2k}{n} \leq 0.48$. However, if we succeed we can contract the edge e, leading to an instance with $n-1$ vertices. Standard running-time/success probability trade-offs now yield that the running time of our algorithm is upper bounded by the recurrence $T(n) \leq \frac{T(n-1)}{0.52}$, which solves to $\mathcal{O}(1.93^n)$. Since we aim for a deterministic algorithm, instead of picking edges at random we directly use a pseudo-random construction from [18] instead.



$k < 0.24n$
yes
no
Thorup Trees
Is Biased? $S \cup M > 0.51n$ or $L \cup M > 0.51n$
yes
no
$S \cup M$
L
S
Bias DP using state graph
$M \leq \epsilon n$ guess and delete M
S
L
Many ($> \frac{n}{\log^2 n}$) edges between large components?
yes
no
- All small are singletons - Reduce to one non-singleton case by subgraph sampling
Guess and delete edges between large components. (Now no such edges)
Is neighbor-biased? Group X of large components s.t $||X| - |N(X)|| > \alpha n$
yes
no
$N(X)$
X
S
L
Guess S and L
Contains Heavy large component?
yes
no
H
S
L
- Guess non-singleton small components - Guess $L \setminus H$ - Reduce to one non-singleton case
Is Skipping cut? Group X of large components $|X| > \theta n$, $|\cup comp(N(X))| < 0.49n$
yes
no
Use enhanced state graph
Guess S and L

Figure 1 Case Tree depicting the various cases our algorithm branches into. Leaf cases are denoted by dotted brown boxes. Supporting figures are inserted to visualize the cases. The notation $\cup comp(N(X))$ denotes the set of all vertices in components having vertices from $N(X)$.

From now onwards we assume that $k \geq 0.24n$. Let C be a hypothetical solution, that is, an optimum k-cut. We split the parts of the solution into small, medium, and large, as follows. A component is said to be *small* if its size is at most $\log n$, *medium* if its size is greater than $\log n$ and less than $\frac{n}{\log n}$, and *large* if its size is at least $\frac{n}{\log n}$. Given a cut C

of G, we denote the set of all vertices in small, medium, and large components in C by S, M and L, respectively. The next case we deal with is that $|S|$ is at most $n(\frac{1}{2} - \epsilon)$. Here, our main observation is that we can make the $2^{n+o(n)}$ time algorithm already run in time $\mathcal{O}((2-\varepsilon)^n)$; only build the state graph, S_G, for vertex sets of size at most $n(\frac{1}{2} - \epsilon)$. For each choice of S use the $\binom{n}{2k}3^{2k}$ time algorithm to find the best partition of $V(G) \setminus S$ into medium and large size parts. Since, $k \leq \frac{n}{\log n}$, the $\binom{n}{2k}3^{2k}$ time algorithm runs in time $(\frac{en}{n/\log n})^{n/\log n} 3^{\mathcal{O}(n/\log n)} = 2^{o(n)}$.

The next case we handle is when $|S \cup M| \geq n(\frac{1}{2} + \epsilon)$. This is the first interesting case. We use the fact that $k > 0.24n$ to infer that there are many small parts. Indeed, observe that at most $0.2n$ parts of C can have size at least 5. Thus, at least $0.04n$ parts of C have size at most 4. To handle this case we extend the idea of Koivisto [36] for speeding up the algorithm for Set Cover when all sets are small (also see the recent generalization by Nederlof [43]). This case is similar to the Set Cover case handled by Nederlof in [43] and our algorithm is based on ideas similar to those used in [36, 43, 45]. Nevertheless we are not able to directly apply these results (it would not be surprising to us if the *methods* used in [43] do apply in this setting).

Specifically, our algorithm proceeds as follows, we pick uniformly at random a set Q, which has the following properties with probability close to 1.

- $|Q| = \frac{n}{2} \pm o(n)$,
- $|Q \cap (S \cup M)| = \frac{|S \cup M|}{2} \pm o(n)$
- At least $\frac{n}{100} - o(n)$ parts P of C of size at most 10 satisfy that $|P \setminus Q| > |P \cap Q|$.
- At least $\frac{n}{100} - o(n)$ parts P of C of size at most 10 satisfy that $|P \setminus Q| < |P \cap Q|$.

Consider now the *ordered* partition **of parts in** C in which: all parts with $|P \cap Q| > |P \setminus Q|$ come first, ordered by size (smaller parts first). Then come all parts P with $|P \cap Q| = |P \setminus Q|$ in any order, followed by all parts with $|P \cap Q| < |P \setminus Q|$ ordered by size (this time smaller parts last). A simple counting argument shows that the path C' in the state graph that corresponds to this ordered partition only visits sets X with the following property:

- $|X| \leq n(\frac{1}{2} - \frac{\epsilon}{10})$ or,
- $|X| \geq n(\frac{1}{2} + \frac{\epsilon}{10})$ or,
- $|X \cap L| \geq \frac{n}{2}(\frac{1}{2} + \frac{\epsilon}{100})$

Thus, to find the path C' we only need to build the subgraph of the state graph with vertices of the three types above. The total number of vertices of this types is upper bounded by $\mathcal{O}((2-\varepsilon)^n)$, leading to an improved algorithm.

The instances not handled by any of the previously discussed cases must satisfy: $k > 0.24n$, $|S|$ and $|L|$ are both between $n(\frac{1}{2} - \epsilon)$ and $n(\frac{1}{2} + \epsilon)$, and $|M|$ is at most $2\epsilon n$. Note that we (the algorithm designers) have control over ϵ and we can choose it as small as we like. The price we pay is that as ϵ becomes smaller the running time of our algorithm comes closer and closer to 2^n. This prevents us from choosing $\epsilon = o(1)$. We design an algorithm with running time $\mathcal{O}((2-\delta)^n)$ for instances where $|S|$ and $|L|$ are both between $n(\frac{1}{2} - \tau)$ and $n(\frac{1}{2} + \tau)$ and $M = \emptyset$. This implies a $\mathcal{O}((2-\epsilon)^n)$ time algorithm for the case where $|M| \leq 2\epsilon n$, whenever ϵ is small enough compared to δ. We can just guess M, run the $n^{\mathcal{O}(k)}$ algorithm on $G[M]$ and run the $\mathcal{O}((2-\delta)^n)$ time algorithm on $G - M$. The total running time is at most $\mathcal{O}(\binom{n}{2\epsilon n}(2^{o(n)} + (2-\delta)^n))$, which beats 2^n whenever ϵ is small enough.

We now turn our attention to the case when $k > 0.24n$, $|S|$ and $|L|$ are both between $n(\frac{1}{2} - \epsilon)$ and $n(\frac{1}{2} + \epsilon)$, and $M = \emptyset$. Our initial intuition was that it should not be too hard to extend the biasing argument from the case that $|S \cup M| \geq n(\frac{1}{2} + \epsilon)$ to also handle these cases, however we were unable to do this. In fact, for a long time we were only able to handle a very special subcase: when all small parts of the optimal solution have size precisely 1, and there is one large part that covers all of L. We call this special case the *at most one non-singleton case*. Observe that solving this case is equivalent to solving the Densest $(n-k+1)$-subgraph problem. Thus we can solve this case in $\mathcal{O}(2^{\frac{\omega n}{3}} n^{\mathcal{O}(1)})$ time using a known algorithm for Densest ℓ-subgraph by Chang et al. [12]. Their algorithm is an adaptation of the celebrated split and list algorithm of Williams [51] for the MAX-CUT problem. At a glance it would appear that this very restricted special case is not useful at all for handling the general case. However, this special case turns out to be pivotal! Indeed, the (rather long and technical) remainder of our argument is to reduce most of the remaining instances of the problem to precisely this special case.

Hard Case. We now provide a high level overview of the tools and techniques used to handle the hard case where $k > 0.24n$, $|S|$ and $|L|$ are both between $n(\frac{1}{2}-\epsilon)$ and $n(\frac{1}{2}+\epsilon)$, and $M = \emptyset$.

First we divide further into two cases based on whether the number of cut edges between the large components in C is greater than $\frac{n}{\log^2 n}$ or not. The idea behind such a split is that there are only $\binom{n^2}{n/\log^2 n} \leq n^{2n/\log^2 n} = 2^{\log n \cdot \frac{2n}{\log^2 n}} = 2^{o(n)}$ subsets of edges of size at most $\frac{n}{\log^2 n}$. So if there are only a few($\leq \frac{n}{\log^2 n}$) cut edges between the large components, we guess and remove them with a $2^{o(n)}$ overhead. This reduces the few cut edges between large components case to one that has no such cut edges. But the no cut edges between large components case turns out to be harder than the many($> \frac{n}{\log^2 n}$) cut edges between large components case. We next show how to handle the latter case followed by the harder one.

1.) Many cut edges between large components. To handle this case, we first observe that all small components are singletons. Indeed, there are at most $\log n$ large parts but there are at least $\frac{n}{\log^2 n}$ cut edges between them. So there exists two large parts P_1 and P_2 having more than $\log n$ cut edges between them. If there exists a non-singleton small part, then combining P_1 and P_2 into a single part and removing a vertex from a non-singleton small part, which by definition has size at most $\log n$, as a singleton gives a strictly better cut.

We use this property to reduce to disjoint instances of the at most one non-singleton case. Here, (a) except for at most $\log n$ many vertices in L, all vertices in L have at least $\frac{n}{\log^4 n}$ neighbors within their large component in C – Suppose a set X of $\log n$ vertices in L have at most $\frac{n}{log^4 n}$ neighbors within their large component. Then, making $L - X$ one big component and all other vertices in X a singleton yields a better cut than C. Also, (b) the number of cut edges between large parts can be upper bounded by $n \log n$ – If not, we can remove $\log n$ vertices in L as singletons and merge all other vertices in L into one big component to obtain a strictly better cut. This is because there are at most $\log n$ large parts.

Next we sample a subgraph G' of G having $V(G') = V(G)$. We keep every edge in G in G' with probability $\frac{1}{\log^3 n}$. We use (a) and (b) to show that G' with high probability satisfies: (A) contains only few ($\leq \frac{n}{\log^2 n}$) cut edges between large components in C and (B) except for $(\log n)^{\mathcal{O}(1)}$ many vertices in L, all vertices in L still have at least $\frac{n}{\log^8 n}$ neighbors in G' within their large component in C.

In G', because of (A), with just $2^{o(n)}$ overhead we guess and remove all remaining cut edges between large components in C. Next, using (B) we show that for each large component X in C we have a set of $(\log n)^{\mathcal{O}(1)}$ vertices $V_X \subset X$ such that $X \subseteq N_{G'}(V_X)$. For every large component X, we guess V_X – There are at most $\log n$ large components and for each, this guessing incurs a $n^{(\log n)^{\mathcal{O}(1)}}$ overhead. Recall that all small components in C are singletons. Next because we removed all cut edges between large components in C in G', V_X has no neighbors in any large component apart from X in G'. Thus for every distinct pair X_1, X_2

of large components in C, every vertex in $N_{G'}(V_{X_1}) \cap N_{G'}(V_{X_2})$ is a singleton in C and can be removed. Finally, in the resulting graph each $V_X \cup N(V_X)$ is a disjoint instance of the at most one non-singleton case.

2.) No cut edges between large components. We handle this case by either reducing to the at most one non-singleton case or by identifying S and L. In the latter case, we use S and L to obtain a cut as good as C. We now show how to efficiently compute a cut as good as C given S and L. To find the part of the cut induced by L, we just use the n^k exact algorithm on $G[L]$ as there are at most $\log n$ large parts. So we shift our focus to finding the part of the cut induced by S. For this, we use the state graph with some guessing. Recall that the vertices of the state graph S_G correspond to subsets of vertices and the edges correspond to components. A path from $\emptyset$ to any subset S in S_G can be mapped to the cut containing all parts corresponding to the edges in P and vice versa. Thus using less than 2^n time, we precompute and store for each state X having $|X| \leq n(\frac{1}{2} - \delta)$ the best ℓ-cut of X containing only small components - $small(X, \ell)$. For this we only build S_G for vertex sets of size at most $n(\frac{1}{2} - \epsilon)$ and edges corresponding to small parts. For bounding our run times, we use that for $\tau \in [0, 1]$, $\binom{n}{\tau n} \leq 2^{h(\tau)n}$, where h is the binary entropy function [32]. So precomputing $small$ takes time at most $\mathcal{O}(2^{h(\frac{1}{2}-\delta)n+o(n)})$. Given S, we guess a union X of small parts in C such that $|S \setminus S'| \leq n(\frac{1}{2} - \delta)$. Then we do a lookup of $small(S', \ell) \cup small(S \setminus S', k - \ell)$ to obtain the part of the cut induced by S. Thus given S and L, we can compute C in time $\binom{\frac{n}{2}+\epsilon n}{(\epsilon+\delta)n} \leq 2^{h(\epsilon+\delta)n}$ since $|S - \frac{n}{2}| \leq \epsilon n$.

Next, to explain the subcases we divide into, we introduce some properties of the cut. Why we need these properties will become clear when we use them to solve the subcases. We say that C is *α-neighbor-biased* if there is a union X of large components whose neighborhood size substantially differs from its size. More precisely if $||X| - |N(X)|| > \alpha n$. If there is no such union of large components, we say C is *neighbor-balanced.* We call a large component *θ-heavy* if it contains all vertices of L except at most θn of them. Look at the order: $\epsilon, \delta, \alpha, \theta$, in which we defined our parameters. We set these parameters such that for each parameter, the ones before it in the order are small enough for our ideas to work. So we start by setting θ which is the highest, followed by α, then δ and finally ϵ. Another nice tool that we use to recover some sets and their neighborhoods is: all p-sized subsets of vertices having a q-sized neighborhood in a graph can be enumerated in time $\mathcal{O}(n\binom{p+q}{q})$ [22, Lemma 3.2].

With that, we divide into three subcases - (i) neighbor-biased (ii) neighbor-balanced with a heavy large component and (iii) neighbor-balanced with no heavy large component.

2.1) α-Neighbor-biased. The core idea in this case is to use the bias in size between X and $N(X)$ to efficiently guess L. We guess X in time $\binom{|X|+|N(X)|}{|X|}$ and recover $N(X)$. Next we guess $L \setminus X$ in time $2^{n-|X|-|N(X)|}$. This lets us recover L and S in time $\binom{|X|+|N(X)|}{|X|} \cdot 2^{n-|X|-|N(X)|}$. Given S and L we do some guessing and lookup of $small$ to compute C in time $2^{h(\epsilon+\delta)n}$ as discussed above. Our final runtime ends up being $\binom{|X|+|N(X)|}{|X|} \cdot 2^{n-|X|-|N(X)|} \cdot 2^{h(\epsilon+\delta)n}$. A simple calculation simplifying $\binom{|X|+|N(X)|}{|X|}$ in terms of h shows that the final runtime is substantially less than 2^n as long as α is sufficiently large as compared to ϵ and δ.

2.2) α-Neighbor-balanced with a θ-heavy large component. Set $\theta = 10^{-5}$. Here we reduce to the at most one non-singleton case. Recall that the case we reduce to can be solved in $\mathcal{O}(2^{\frac{\omega}{3}n})$ time. Thus we can afford an overhead of $2^{(0.999-\frac{\omega}{3})n}$ to get to that case.

First we guess the set $L \setminus H$ of vertices in L not in the heavy component H. This incurs a cost of $\binom{n}{\theta n} \leq 2^{h(\theta)n}$. Then we guess the set X of vertices in non-singleton small components. We show that $|X|$ is actually very small and that the $\binom{n}{|X|}$ overhead to guess X is acceptable.

Observe that we can compute the cut induced by $L \setminus H$ in $2^{o(n)}$ time using the n^k algorithm and the cut induced by X in $\mathcal{O}(1)$ time by a lookup of *small*. Thus after guessing $L \setminus H$ and X we reduce to the one non-singleton case. All that remains to do now is to prove $|X|$ is small enough to be able to guess X. We bound $|X|$ by $0.01n$. Then $\binom{n}{|X|} \leq \binom{n}{0.01n} \leq 2^{h(0.01)n}$, which is an easily affordable overhead.

Suppose $|X| > 0.01n$, we show that C can be improved. We call a non-singleton component *good* if the heavy component is adjacent to 99% of the vertices in it. Since C is neighbor-balanced and $0.01 >> (\theta = 10^{-5})$, we can show that the heavy component is adjacent to almost all vertices in many non-singleton components; that there are at least $n/(\log n)^{\mathcal{O}(1)}$ good components. Because there are $n/(\log n)^{\mathcal{O}(1)}$ good components and good components are all small, we can find an $i (\leq \log n)$ such that there are i good components of size i. Merging $i-1$ good components of size i with the heavy component and breaking one good component of size i into singletons improves C; the extra cost to break is only $0.05\binom{i}{2}$ while the savings from merging is $0.99(i-1)i$.

2.3) Neighbor-balanced with no heavy large component. As a first step, we make our state graph more robust to handle special cases. Recall that the edges of the state graph correspond to components. We build the state graph for vertex sets of size either at most $n(\frac{1}{2} - \epsilon)$ or at least $n(\frac{1}{2} + \epsilon)$ and edges corresponding to small parts. We then add a few additional edges that correspond to groups of components. For each pair X, $X \cup Y$ of vertex sets in the new state graph, we draw an edge from X to $X \cup Y$ if Y is a union of a set of at most $\log n$ connected components in $G[V \setminus X]$. We call the constructed state graph the *enhanced* state graph. Observe that the enhanced state graph has size $\mathcal{O}((2-\varepsilon)^n)$. Now we demonstrate how the new edges help.

Skipping Cut. Suppose C has a union of small components S' and union of large components L' such that (i) $|S'| \leq n(\frac{1}{2} - \theta)$ (ii) $|S' \cup L'| \geq n(\frac{1}{2} + \theta)$ and (iii) $N(L') \subseteq S'$. Also recall that C does not contain edges between large components. In this case, there is a path P in the new state graph from $\emptyset$ to V corresponding to C. We decompose P into 4 portions: (1) a path from $\emptyset$ to S' using the edges corresponding to parts in S', (2) a edge from S' to $S' \cup L'$ corresponding to parts in L', (3) a path from $S' \cup L'$ to $S \cup L'$ using the edges corresponding to parts in $S \setminus S'$ and (4) an edge from $S \cup L'$ to V corresponding to parts in $L \setminus L'$. We say C is a *skipping* cut if it satisfies (i), (ii) and (iii) as we can find C in the modified state graph by skipping between states of size at most $n(\frac{1}{2} - \theta)$ and states of size at least $n(\frac{1}{2} + \theta)$ but by avoiding states of size $\frac{n}{2}$. In conclusion, if C is neighbor balanced with no heavy large component but C is a skipping cut then we can find C using the enhanced state graph in time $\mathcal{O}(2-\epsilon)^n$.

We now turn our attention to the final case: neighbor-balanced with no heavy large component and not skipping. To solve this case we reduce to a special case in which S and L can be identified efficiently using $2^{\frac{n}{2}}$ time. Thus we can afford an overhead of $2^{0.499n}$ to get to that case; but we incur a much smaller overhead. First we highlight the properties of C required for the special case and show how to use them to obtain S and L. Then we show how to guess a few vertices in S to reduce to the case we want.

Let C satisfy the following properties: (i) All large components in C can be grouped into two parts L_1 and L_2 each having size nearly $\frac{n}{4}$ (ii) $N(L_1) \cap N(L_2) = \emptyset$ (iii) All small components in C are of size two and contain one vertex from $N(L_1)$ and the other from $N(L_2)$. To find L and S, we first guess L_1 in time $2^{|L_1|+|N(L_1)|}$ and recover $N(L_1)$. We then recover $N(L_2)$ as $N(N(L_1)) \setminus L_1$. Then the set of remaining vertices is L_2, yielding $L = L_1 \cup L_2$. This in total only takes about $2^{|L_1|+|N(L_1)|} \leq 2^{\frac{n}{2}}$ time since $|L_1| \sim \frac{n}{4}$.

Suppose property (i) is not satisfied, let L' be a group of large components having size at least θn but less than $\frac{n}{4}$; such a group exists because there is no heavy large component. Then C is a skipping cut with witness L' and S' for some $S' \supseteq N(L')$ which exists because C is neighbor balanced. All that remains to be done is to show that we can guess some vertices in S to satisfy properties (ii) and (iii). Let S' be the set of vertices in small components of size two that contain one vertex from $N(L_1) \setminus N(L_2)$ and the other from $N(L_2) \setminus N(L_1)$. Also let $\mathcal{F}'$ be the family containing all such components. We guess $S \setminus S'$. One can easily verify that removing $S \setminus S'$ guarantees properties (ii) and (iii). To conclude that such a guess is feasible, we need to bound the size of $S \setminus S'$. We prove $|S \setminus S'| \leq 30(\epsilon + \theta + \alpha)n$.

Let $\mathcal{F}$ to be the family of all small components X in C that satisfy: (a) X has no vertex from $S \setminus N(L)$, (b) X has a vertex from $N(L_1)$ and a vertex from $N(L_2)$, and (c) X has no vertex from $N(L_1) \cap N(L_2)$.
Observe that the subfamily of all components of size two in $\mathcal{F}$ is $\mathcal{F}'$. We use this to obtain a bound on $|S \setminus S'|$. By definition, every component in $\mathcal{F}$ has at least two vertices, one in $N(L_1) \setminus N(L_2)$ and one in $N(L_2) \setminus N(L_2)$. Thus $|S| - 2|\mathcal{F}|$ will account for all but two vertices in components of size three or more in $\mathcal{F}$ and for all vertices in S not in any component in $\mathcal{F}$. This in turn implies $3 \cdot (|S| - 2|\mathcal{F}|)$ is an upper bound for the number of vertices in S not contained in any component of size two in $\mathcal{F}$, i.e bound for $|S \setminus S'|$. We will show $|S| - 2|\mathcal{F}| \leq 10(\epsilon + \alpha + \theta)n$ from which we can infer $|S \setminus S'| \leq 3(|S| - 2|\mathcal{F}|) \leq 30(\epsilon + \alpha + \theta)n$.

Next, for each of the properties $(a) - (c)$, we bound the number of small components that do not satisfy that property. Since $|S|$ and $|L|$ are both between $n(\frac{1}{2} - \epsilon)$ and $n(\frac{1}{2} + \epsilon)$ and C is α-neighbor-balanced, there are at most $(2\epsilon + \alpha)n$ vertices in S that are not in $N(L)$. Thus $|S \setminus N(L)| \leq (2\epsilon + \alpha)n$. So at most $(2\epsilon + \alpha)n$ small components do not satisfy (a). At most $(\epsilon + \theta/2)n$ small components do not have a neighbor in L_1. If not, since $|S| \leq n(\frac{1}{2} + \epsilon)$, we can show that C is a skipping cut with witness $L' = L_1$ and some $S' \supset N(L_1)$. The same argument holds with respect to L_2. Thus, at most $(2\epsilon + \theta)n$ small components do not satisfy (b). Next, since C is α-neighbor-balanced, $|N(L)| = |N(L_1)| + |N(L_2)| - |N(L_1) \cap N(L_2)| \leq |L_1| + \alpha n + |L_2| + \alpha n - |N(L_1) \cap N(L_2)|$. So $|N(L_1) \cap N(L_2)| \leq |L| - |N(L)| + 2\alpha n \leq 3\alpha n$. This bounds the number of small components not satisfying (c) by $3\alpha n$.

For an easy read, until this point we mentioned $k > 0.24n$. But in our algorithm, we set $k > \frac{n}{4} - \epsilon n$ and need it for this case to work. Since $k > \frac{n}{4} - \epsilon n$, using all the three bounds obtained, $|\mathcal{F}| \geq 0.25n - \epsilon n - (2\epsilon + \alpha)n - (2\epsilon + \theta)n - 3\alpha n \geq 0.25n - (5\epsilon + 4\alpha + \theta)n$. Thus, $|S| - 2|\mathcal{F}| \leq (0.5 + \epsilon)n - 2(0.25n - (5\epsilon + 4\alpha + \theta)n) \leq (10\epsilon + 8\alpha + 2\theta)n \leq 10(\epsilon + \alpha + \theta)n$. Note that the main proof is structured differently for tighter bounds with these being the core ideas.

2 Preliminaries

Given a graph G, we use $V(G)$ and $E(G)$ to denote the vertex sets and edge sets, respectively. For any subset $X \subseteq V(G)$, let $G[X]$ denote the induced subgraph of G on X. A subset $X \subseteq V(G)$ is said to be connected if $G[X]$ is a connected graph. We denote the number of connected components in G by $cc(G)$. Given a path P in G, let $E(P)$ denote the edges in the path. Given a subset $E' \subseteq E(G)$, we denote the subgraph G' having $V(G') = V(G)$ and $E(G') = E(G) \setminus E'$ by $G \setminus E'$. For any subset $X \subseteq V(G)$, let $N_G(X)$ denote the set of vertices in G adjacent to some vertex in X and let $N_G[X] = N_G(X) \cup X$. We will omit the subscript when the graph is clear from context.

A *k-cut* of a graph G is a partition of the vertex set $V(G)$ into k non-empty parts. We will refer to the parts of a cut as *components* of the cut. The *weight* of a k-cut is the total number of edges with endpoints in different parts of the partition. We refer to the edges of G having end points in different components in C as *cut edges* of C. We use $\mathsf{best}(X, l)$ to denote a least weight l-cut of $G[X]$. For any two disjoint subsets A, B of $V(G)$, we denote the number of edges having one end point in A and the other in B by $w(A, B)$. For every $E' \subseteq E(G)$, we let $w(E') = |E'|$. For every $X \subseteq V(G)$, we let $\delta(X)$ denote the set of edges having exactly one end point in X in G.

If G is an edge-weighted graph, then the *weight* of a k-cut is the sum of weights of edges with endpoints in different parts of the partition. For any two disjoint subsets A, B of $V(G)$, we denote the sum of weights of edges having one end point in A and the other in B by $w(A, B)$. For every $E' \subseteq E(G)$, we denote the sum of weights of all edges in E' by $w(E')$. We consider G to be a simple unweighted graph throughout the paper except in Section where we provide the $2^{n+o(n)}(\log W)^{\mathcal{O}(1)}$ time algorithm for edge-weighted graphs.

We also need the following lemma to enumerate connected sets of size $b + 1$ with at most f neighbors in our algorithms.

▶ **Lemma 3** ([22, Lemma 3.2]). *All vertex sets of size $b + 1$ with f neighbors in a graph G can be enumerated in time $\mathcal{O}(n\binom{b+f}{b})$ by making use of polynomial space.*

There exists an algorithm for Min k-Cut using Thorup trees that runs in time $\mathcal{O}(\binom{n}{2k}3^{2k})$. We express this runtime in this way because it enables us to obtain a $2^{o(n)}$ algorithm for k-cut whenever $k \leq \frac{n}{\log n}$. Given a polynomial in n, m, and $\log n$ sized family of spanning trees that contains a Thorup tree, the algorithm for each tree guesses all possible sets of edges of size at most $2k - 2$, removes them and contracts the remaining forests into vertices and solves k-cut on this contracted graph in time 3^{2k}.

▶ **Lemma 4** (Exact Algorithm [50]). *Given a graph G, positive integer k, the min k-cut of G can be found in time $\mathcal{O}(\binom{n}{2k}3^{2k})$.*

The cut induced by a cut C of X on a subset Y of X is the cut obtained by taking the intersection of each component in C with Y. We define the union of two disjoint cuts C_1 of vertex set X_1 and C_2 of vertex set X_2 as the cut C of $X_1 \cup X_2$ that induces cut C_1 on X_1 and cut C_2 on X_2. We define the union of a subfamily $\{Y_0, \cdots, Y_z\}$ of components of a cut to be $\cup_{i \leq z} Y_i$, the set of all vertices in any component in the subfamily. Given a family $\mathcal{F}$ of subsets of $V(G)$, let $U(\mathcal{F}) = \cup_{X \in \mathcal{F}} X$ denote the set of all vertices in some set in $\mathcal{F}$. We will refer to a cut having a single component as just a set of vertices for convenience.

We also need the following classical single source shortest path algorithm to traverse the state graph, S_G (formally defined in later)[2].

▶ **Lemma 5** (Dijkstra's Algorithm). *Given an edge weighted directed graph D on n vertices and m edges, with two special vertices s and t, we can find a minimum weight shortest path from s to t in time $\mathcal{O}((n + m)\log n)$.*

We also need the following upper bounds on binomial coefficients for our purposes.

▶ **Lemma 6** ([32]). *Let n be a positive integer and $\alpha \in [0, 1]$, then $\binom{n}{\alpha n} \leq 2^{h(\alpha)n}$, where $h(\alpha)$ represents the entropy of a Bernoulli random variable with probability of success α, satisfying $h(\alpha) = -\alpha \log \alpha - (1 - \alpha)\log(1 - \alpha)$. Further, for a positive integer $k \leq n$, $\binom{n}{k} \leq \left(\frac{en}{k}\right)^k$, where e is the base of natural logarithms.*

[2] S_G will be a DAG, so one can also resort to other means to find single source shortest paths.

By truncating the Taylor series for h, given by

$$h(p) = 1 - \frac{1}{2\ln 2}\sum_{n=1}^{\infty}\frac{(1-2p)^{2n}}{n(2n-1)},$$

we get the following useful upper bound for h, when η is close to $\frac{1}{2}$.

▶ **Lemma 7.** *For $\eta \leq 1/10$, we have that $h(\frac{1}{2}-\eta) \leq 1 - \frac{\eta^2}{\ln 2}$.*

We also need the following definition of "contraction and uncontraction of edges".

▶ **Definition 8** ((Un)Contraction of Cuts and Graphs). *Given a graph G and an edge $e = uv$, the graph G/uv obtained by* contracting *the edge uv, is the one where we delete the vertices $\{u, v\}$, and introduce a new vertex v_e and add edges from v_e to every vertex in $N(u)$ and $N(v)$. If $w \in N(u) \cap N(v)$, then the process will make multi-edges.*

Given a cut C and an edge uv such that, u, v belongs to the same part in C, then by contraction of cut*, denoted by C/uv, we mean the cut constructed from C, where we replace the part P in C, that contains u and v with $(P \setminus \{u, v\}) \cup \{v_e\}$. Similarly, given a vertex set X, we define* uncontraction of a cut C*, denoted by $C/^{-1}X$, as a cut where we replace each vertex $x \in X$, with the set of vertices that has been contracted into x.*

3 Proof of Main Theorem and Case Breakdown

In this section we provide an overview of our algorithm with a case breakdown showing how we organize our cases throughout the paper. We first introduce some basic terminology used in the algorithm. Then we provide a guide to the different cases with pointers to various subroutines called in the algorithm. Finally we define a data structure that is used by most of our subroutines and state a known result for the at most one non-singleton case.

3.1 Properties of Cuts and Special Cases

In this section we define multiple properties of cuts that will be helpful for designing our algorithm. Let G be a graph on n vertices. We will design algorithms to find a cut with different combinations of these properties in G and then combine them to obtain our final algorithm. We divide the components of any cut of G into three types depending on their size.

▶ **Definition 9** (Component Types). *A component is said to be* small *if its size is at most $\log n$. A component is said to be* medium *if its size is greater than $\log n$ and less than $\frac{n}{\log n}$. A component is said to be* large *if its size is at least $\frac{n}{\log n}$. Given a cut C of G, we denote the set of all vertices in small, medium and large components in C by S_C, M_C and L_C respectively.*

This notion is useful because the number of subsets of $V(G)$ that can be small or medium components is at most $n\binom{n}{\frac{n}{\log n}} \leq n(\frac{en}{n/\log n})^{n/\log n} = 2^{o(n)}$. Also, for any subset L of $V(G)$ and $\ell \leq \log n$, we can find a ℓ-cut of $G[L]$ having weight no more than a minimum weight an ℓ-cut of $G[L]$ containing only large components in time $2^{o(n)}$ using Lemma 4. This is because there can be at most $\log n$ large components in any cut. We will use both these facts crucially while designing our algorithms. The above discussion leads us to define the following definitions.

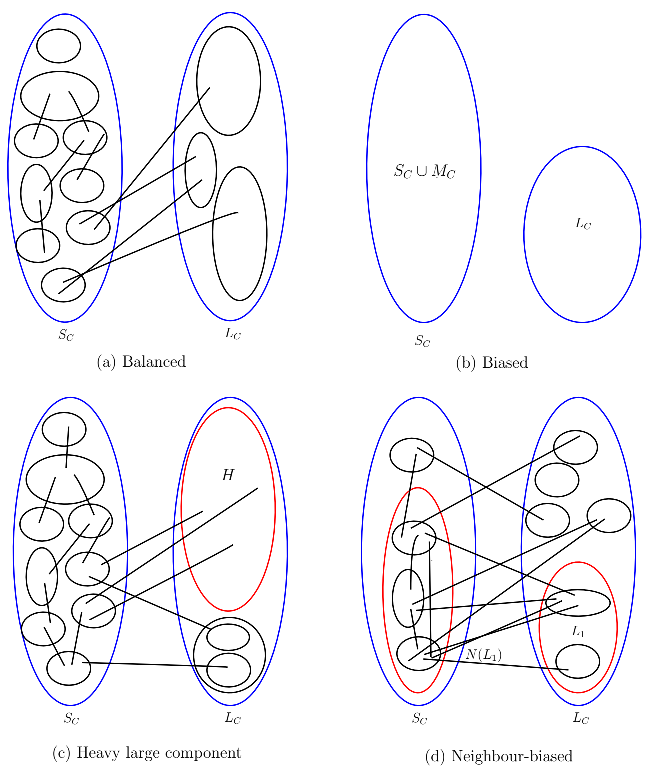


Figure 2 Visualizing cut properties (a) Balanced cut has $|S_C|$ nearly equal to $|L_C|$ and no medium components. (b) Biased cut has one of $|S_C \cup M_C|$ or $|L_C \cup M_C|$ substantially higher. (c) Heavy large component H contains nearly all vertices in L_C. (d) Neighbor-biased cut has a union of heavy components L_1 such that $||L_1| - |N(L_1)||$ is high.

▶ **Definition 10.** *For a subset $X \subseteq V(G)$ and $\ell \leq n$,* small(X, ℓ) *represents an ℓ-cut of $G[X]$ having weight no more than the weight of a least weight ℓ-cut of $G[X]$ containing only small and medium components.*

▶ **Definition 11** (Biased and Balanced Cuts)**.** *For any $0 < \delta < \frac{1}{2}$, a cut C is said to be δ-*biased *if either $|M_C \cup S_C| \geq n(\frac{1}{2} + \delta)$ or $|M_C \cup L_C| \geq n(\frac{1}{2} + \delta)$ and it is said to be δ-*balanced *if it contains no medium components, $|S_C| \leq n(\frac{1}{2} + \delta)$ and $|L_C| \leq n(\frac{1}{2} + \delta)$.*

▶ **Definition 12** (Properties of Large Components)**.** *We say that a cut C of G has* many edges between large components *if the number of cut edges of C having both end points in L_C is greater than $\frac{n}{\log^2 n}$. We say that the cut has* few edges between large components *if the number of such edges is at most $\frac{n}{\log^2 n}$.*

Observe that we can reduce the case of few edges between large components to no edges by guessing the set of edges between the large components with just $2^{o(n)}$ guesses. This again will prove helpful for us to get more structure.

▶ **Definition 13.** *For any $0 \leq \alpha < 1$, we say that a large component X in a cut C of G is α-*heavy *if it has all vertices in L_C except for at most αn vertices, i.e $|L_C \setminus X| \leq \alpha n$.*

▶ **Definition 14.** *For any $0 \leq \alpha < \frac{1}{2}$ and $0 < \rho < \frac{1}{2}$, we say that a α-balanced cut C is ρ-neighbor-biased if there is a union X of large components in C having p vertices such that $N(X) \subseteq S_C$, $|N(X)| = q$ and $|p - q| > \rho n$. Otherwise the cut is said to be ρ-neighbor-balanced.*

We note that we will use the last two properties only on balanced cuts having no edges between large components. Figure 2 captures all the properties defined above.

3.2 Steps of Our Algorithm

Armed with all the technical definitions, we now provide a guide to the different cases our algorithm for Min k-Cut handles. First, we summarize the different parameters that we use in our algorithm and the final value we set them to obtain our main result.

- $\alpha_1 = 10^{-20}$: $k < \frac{n}{4} - \alpha_1 n$ and $k \geq \frac{n}{4} - \alpha_1 n$
- $\alpha_2 = 10^{-20}$: α_2-balanced or $\frac{\alpha_2}{2}$-biased cut
- $\alpha_3 = 10^{-5}$: α_3-heavy large component
- $\alpha_4 = 10^{-2}$: number of non singletons
- $\alpha_5 = 10^{-5}$: α_5-neighbor biased or balanced
- $\delta = 10^{-20}$: δ-small cut data structure

Note that our results will work for a range of these parameters but for convenience we fix these parameters to a particular value and obtain our final result. Given an input graph G on n vertices and integer k, let C be a *hypothetical* min k-cut of G. Our goal is to compute C. We divide into cases based on the value of k and the properties of C. We use Lemmas 6 and 7 to upper bound the running time for each case by $(2-\varepsilon)^n$ for some $\varepsilon > 10^{-50}$. In the summary, for each case we mention which one of the two Lemmas is used to bound the running time in parenthesis.

(Case 1: $k < \frac{n}{4} - \alpha_1 n$.) In this case we apply Lemma 15 and directly solve the problem in time $\frac{2^{n+o(n)}}{(1+4\alpha_1)^{\alpha_1 n}}$ (Lemma 6).

▶ **Lemma 15 (*).** *There exists an algorithm that takes as input a graph G on n vertices, an integer $k < \frac{n}{4} - \alpha_1 n$ and returns the min k-cut of G in time $\frac{2^{n+o(n)}}{(1+4\alpha_1)^{\alpha_1 n}}$.*

(Case 2: $k \geq \frac{n}{4} - \alpha_1 n$) Since k is large, we have the property that the number of components of size at most 4 is at least $0.04n$. We then break this case based on how efficiently we can use this property. We use it in the case when C is $\frac{\alpha_2}{2}$-biased and handle the balanced case separately, leading to the subsequent subcases. This case is handled in Lemma 16 in time $2^{n\frac{1}{2}(1+h(\frac{1}{2}+\frac{\alpha_2}{20}))+o(n)} + 2^{(h(\alpha_2)+h(\frac{1}{2}-\alpha_2))n+o(n)}$ (Lemma 6).

▶ **Lemma 16 (*).** *There exists an algorithm that takes as input a graph G on n vertices, an integer $k \geq \frac{n}{4} - \alpha_1 n$ and returns the min k-cut of G in time $2^{n\frac{1}{2}(1+h(\frac{1}{2}+\frac{\alpha_2}{20}))+o(n)} + 2^{h(\alpha_2)+h(\frac{1}{2}-\alpha_2)n+o(n)}$, when $\alpha_1 = \alpha_2 = 10^{-20}$.*

Case 2a: ($\frac{\alpha_2}{2}$-biased case) We completely handle this case in Lemma*[3] 5.3 in time $2^{\frac{n}{2}(1+h(\frac{1}{2}+\frac{\alpha_2}{20}))+o(n)}$ (Lemma 7).

Case 2b: (α_2-balanced case) We further divide this case based on whether the number of edges between large components in C is large ($> \frac{n}{\log^2 n}$) or not. We handle this in Lemma* 5.4 in time $2^{h(\frac{1}{2}-\alpha_2)n+o(n)}$ (Lemma 7).

Case 2b (i): (Many edges between large components) We solve this case by reducing to the case of finding a cut having at most one non-singleton component. We handle this case in Lemma* 7.1 using Lemma* 3.2 as a subroutine in time $2^{\frac{\omega n}{3}+o(n)}$ (Lemma 6).

[3] We use Lemma* to refer to the Lemmas stated and proved in the full version; Lemma* 5.3 is Lemma 5.3 in the full version.

Case 2b (ii): (Few edges between large components) This is our final case and we deal with this by designing algorithms for the following cases. This case can be found in Lemma* 7.2 and the algorithm runs in time $2^{h(\frac{1}{2}-\alpha_2)n+o(n)}$ (Lemma 7).

(a) **(α_5-neighbor biased)** This case is handled in Lemma* 7.3. Here we use the fact that C is neighbor-biased to find a family of partitions of $V(G)$ containing the partition $S_C \dot\cup L_C$ of $V(G)$ and use Lemma* 3.1 to find a cut having weight no more than a α_2-balanced cut C' with $S_C = S'_C$ and $L_C = L'_C$. We handle this case in time $2^{h(\frac{1}{2}-\alpha_2)n+o(n)}$ (Lemma 7).

(b) **(α_5-neighbor balanced and α_3-heavy component)** In this case we argue that there are not too many non-singleton components and solve the case by reducing to the case of finding a cut having at most one non-singleton component. We handle this case in Lemma* 7.4 using Lemma* 3.2 as a subroutine in time $2^{(h(\alpha_3)+h(\alpha_4)+\frac{\omega}{3})n+o(n)}$ (Lemma 6).

(c) **(α_5-neighbor balanced and no α_3-heavy component)** We handle this case completely in Lemma* 7.5. Here we first run Lemma* 4.2 that involves using the state graph. Then similar to Case A, here we find a family of partitions of $V(G)$ containing the partition $S_C \dot\cup L_C$ of $V(G)$ and use Lemma* 3.1 on it. This case runs in time $2^{h(\frac{1}{2}-\alpha_2)n+o(n)}$ (Lemma 7).

A case tree depicting the various cases can be found in Figure 1. We now provide the proof of our main Theorem assuming Lemma 15($k < \frac{n}{4} - \alpha_1 n$) and Lemma 16($k \geq \frac{n}{4} - \alpha_1 n$).

Proof of Theorem 1. If $k < \frac{n}{4} - \alpha_1 n$ we run Lemma 15 with G and k else we run Lemma 16 with G and k and return the obtained k-cut. The correctness follows from the corresponding Lemmas. To bound the running time of the algorithm, we show that the running time of each subroutine mentioned above in the summary is bounded by $\mathcal{O}(2-\varepsilon)^n$, for some $\varepsilon > 10^{-50}$. To do this, we set the values of the parameters as summarized in the beginning of this subsection and use Lemma 7 and 6 as mentioned in the summary. ◀

4 Conclusion

In this paper we designed the first algorithm for Min k-Cut running in time better than 2^n. In particular, we designed an algorithm with running time $\mathcal{O}((2-\varepsilon)^n)$, for $\varepsilon > 10^{-50}$. We hope that our methods will be useful to design $\mathcal{O}(c^n)$ time algorithms for $c < 2$ for other graph partitioning problems for which progress has stopped at 2^n, including Cutwidth, Edge Multiway Cut, and more generally Edge Multicut on undirected graphs. Finally, it remains an interesting open problem to obtain a $\mathcal{O}(c^n)$ time algorithm for Min k-Cut for a constant $c < 2$ which is bounded away from 2 by more than a rounding error.

References

1 Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming – 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2014. `doi:10.1007/978-3-662-43948-7_4`.

2 Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021.

3 Arturs Backurs and Christos Tzamos. Improving viterbi is hard: Better runtimes imply faster clique algorithms. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 311–321. PMLR, 2017. URL: `http://proceedings.mlr.press/v70/backurs17a.html`.

4 Andreas Björklund. Determinant sums for undirected hamiltonicity. *SIAM J. Comput.*, 43(1):280–299, 2014. `doi:10.1137/110839229`.

5 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets möbius: fast subset convolution. In David S. Johnson and Uriel Feige, editors, *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 67–74. ACM, 2007. `doi:10.1145/1250790.1250801`.

6 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Trimmed moebius inversion and graphs of bounded degree. *Theory Comput. Syst.*, 47(3):637–654, 2010. `doi:10.1007/s00224-009-9185-7`.

7 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. The traveling salesman problem in bounded degree graphs. *ACM Trans. Algorithms*, 8(2):18:1–18:13, 2012. `doi:10.1145/2151171.2151181`.

8 Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Counting connected subgraphs with maximum-degree-aware sieving. In Wen-Lian Hsu, Der-Tsai Lee, and Chung-Shou Liao, editors, *29th International Symposium on Algorithms and Computation, ISAAC 2018, December 16-19, 2018, Jiaoxi, Yilan, Taiwan*, volume 123 of *LIPIcs*, pages 17:1–17:12. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. `doi:10.4230/LIPIcs.ISAAC.2018.17`.

9 Andreas Björklund, Thore Husfeldt, and Mikko Koivisto. Set partitioning via inclusion-exclusion. *SIAM J. Comput.*, 39(2):546–563, 2009. `doi:10.1137/070683933`.

10 Andreas Björklund, Petteri Kaski, and Ryan Williams. Solving systems of polynomial equations over GF(2) by a parity-counting self-reduction. 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 26:1–26:13. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. `doi:10.4230/LIPIcs.ICALP.2019.26`.

11 Ivan Bliznets, Fedor V. Fomin, Michal Pilipczuk, and Yngve Villanger. Largest chordal and interval subgraphs faster than 2^n. *Algorithmica*, 76(2):569–594, 2016. `doi:10.1007/s00453-015-0054-2`.

12 Maw-Shang Chang, Li-Hsuan Chen, Ling-Ju Hung, Peter Rossmanith, and Guan-Han Wu. Exact algorithms for problems related to the densest k-set problem. *Inf. Process. Lett.*, 114(9):510–513, 2014. `doi:10.1016/j.ipl.2014.04.009`.

13 Rajesh Chitnis, Marek Cygan, MohammadTaghi Hajiaghayi, Marcin Pilipczuk, and Michał Pilipczuk. Designing FPT algorithms for cut problems using randomized contractions. *SIAM J. Comput.*, 45(4):1171–1229, 2016. `doi:10.1137/15M1032077`.

14 Marek Cygan, Holger Dell, Daniel Lokshtanov, Dániel Marx, Jesper Nederlof, Yoshio Okamoto, Ramamohan Paturi, Saket Saurabh, and Magnus Wahlström. On problems as hard as CNF-SAT. *ACM Trans. Algorithms*, 12(3):41:1–41:24, 2016. `doi:10.1145/2925416`.

15 Marek Cygan, Pawel Komosa, Daniel Lokshtanov, Marcin Pilipczuk, Michal Pilipczuk, Saket Saurabh, and Magnus Wahlström. Randomized contractions meet lean decompositions. *ACM Trans. Algorithms*, 17(1):6:1–6:30, 2021. `doi:10.1145/3426738`.

16 Marek Cygan and Marcin Pilipczuk. Faster exponential-time algorithms in graphs of bounded average degree. *Inf. Comput.*, 243:75–85, 2015. `doi:10.1016/j.ic.2014.12.007`.

17 Marek Cygan, Marcin Pilipczuk, Michal Pilipczuk, and Jakub Onufry Wojtaszczyk. Scheduling partially ordered jobs faster than 2^n. *Algorithmica*, 68(3):692–714, 2014. `doi:10.1007/s00453-012-9694-7`.

18 Fedor V. Fomin, Serge Gaspers, Daniel Lokshtanov, and Saket Saurabh. Exact algorithms via monotone local search. *J. ACM*, 66(2):8:1–8:23, 2019. doi:10.1145/3284176.

19 Fedor V. Fomin, Serge Gaspers, and Saket Saurabh. Improved exact algorithms for counting 3- and 4-colorings. In Guohui Lin, editor, *Computing and Combinatorics, 13th Annual International Conference, COCOON 2007, Banff, Canada, July 16-19, 2007, Proceedings*, volume 4598 of *Lecture Notes in Computer Science*, pages 65–74. Springer, 2007. doi:10.1007/978-3-540-73545-8_9.

20 Fedor V. Fomin and Dieter Kratsch. *Exact Exponential Algorithms*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2010. doi:10.1007/978-3-642-16533-7.

21 Fedor V. Fomin, Dieter Kratsch, Ioan Todinca, and Yngve Villanger. Exact algorithms for treewidth and minimum fill-in. *SIAM J. Comput.*, 38(3):1058–1079, 2008. doi:10.1137/050643350.

22 Fedor V. Fomin and Yngve Villanger. Treewidth computation and extremal combinatorics. *Comb.*, 32(3):289–308, 2012. doi:10.1007/s00493-012-2536-z.

23 Olivier Goldschmidt and Dorit S. Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. *Math. Oper. Res.*, 19(1):24–37, 1994. doi:10.1287/moor.19.1.24.

24 Alexander Golovnev, Alexander S. Kulikov, and Ivan Mihajlin. Families with infants: Speeding up algorithms for np-hard problems using FFT. *ACM Trans. Algorithms*, 12(3):35:1–35:17, 2016. doi:10.1145/2847419.

25 Anupam Gupta, Euiwoong Lee, and Jason Li. Faster exact and approximate algorithms for k-cut. In Mikkel Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 113–123. IEEE Computer Society, 2018. doi:10.1109/FOCS.2018.00020.

26 Anupam Gupta, Euiwoong Lee, and Jason Li. An FPT algorithm beating 2-approximation for k-cut. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 2821–2837. SIAM, 2018. doi:10.1137/1.9781611975031.179.

27 Anupam Gupta, Euiwoong Lee, and Jason Li. The number of minimum k-cuts: improving the karger-stein bound. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 229–240. ACM, 2019. doi:10.1145/3313276.3316395.

28 Anupam Gupta, Euiwoong Lee, and Jason Li. The karger-stein algorithm is optimal for k-cut. 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 473–484. ACM, 2020. doi:10.1145/3357713.3384285.

29 Anupam Gupta, Euiwoong Lee, and Jason Li. The number of minimum k-cuts: Improving the karger-stein bound. *To appear in STOC 2020*, abs/1906.00417, 2020. arXiv:1906.00417.

30 Zhiyang He and Jason Li. Breaking the n^k barrier for minimum k-cut on simple graphs. In Stefano Leonardi and Anupam Gupta, editors, *STOC '22: 54th Annual ACM SIGACT Symposium on Theory of Computing, Rome, Italy, June 20 - 24, 2022*, pages 131–136. ACM, 2022.

31 Russell Impagliazzo and Ramamohan Paturi. On the complexity of k-sat. *J. Comput. Syst. Sci.*, 62(2):367–375, 2001. doi:10.1006/jcss.2000.1727.

32 Stasys Jukna. *Extremal combinatorics: with applications in computer science*. Springer Science & Business Media, 2011.

33 David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *J. ACM*, 43(4):601–640, 1996. doi:10.1145/234533.234534.

34 Ken-ichi Kawarabayashi and Bingkai Lin. A nearly 5/3-approximation FPT algorithm for min-k-cut. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 990–999. SIAM, 2020. doi:10.1137/1.9781611975994.59.

35 Ken-ichi Kawarabayashi and Mikkel Thorup. The minimum k-way cut of bounded size is fixed-parameter tractable. In *52nd Annual IEEE Symposium on Foundations of Computer Science, FOCS, 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 160–169. IEEE Computer Society, 2011. doi:10.1109/FOCS.2011.53.

36 Mikko Koivisto. Partitioning into sets of bounded cardinality. In Jianer Chen and Fedor V. Fomin, editors, *Parameterized and Exact Computation, 4th International Workshop, IWPEC 2009, Copenhagen, Denmark, September 10-11, 2009, Revised Selected Papers*, volume 5917 of *Lecture Notes in Computer Science*, pages 258–263. Springer, 2009. doi:10.1007/978-3-642-11269-0_21.

37 Jason Li. Faster minimum k-cut of a simple graph. In David Zuckerman, editor, *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019, Baltimore, Maryland, USA, November 9-12, 2019*, pages 1056–1077. IEEE Computer Society, 2019. doi:10.1109/FOCS.2019.00068.

38 Daniel Lokshtanov, Ivan Mikhailin, Ramamohan Paturi, and Pavel Pudlák. Beating brute force for (quantified) satisfiability of circuits of bounded treewidth. In Artur Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 247–261. SIAM, 2018. doi:10.1137/1.9781611975031.18.

39 Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, R. Ryan Williams, and Huacheng Yu. Beating brute force for systems of polynomial equations over finite fields. In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 2190–2202. SIAM, 2017. doi:10.1137/1.9781611974782.143.

40 Daniel Lokshtanov, Saket Saurabh, and Vaishali Surianarayanan. A parameterized approximation scheme for min k-cut. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020, Durham, NC, USA, November 16-19, 2020*, pages 798–809. IEEE, 2020. doi:10.1109/FOCS46700.2020.00079.

41 Pasin Manurangsi. Inapproximability of maximum biclique problems, minimum k-cut and densest at-least-k-subgraph from the small set expansion hypothesis. *Algorithms*, 11(1):10, 2018. doi:10.3390/a11010010.

42 Joseph Naor and Yuval Rabani. Tree packing and approximating k-cuts. In S. Rao Kosaraju, editor, *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA*, pages 26–27. ACM/SIAM, 2001. URL: http://dl.acm.org/citation.cfm?id=365411.365415.

43 Jesper Nederlof. Finding large set covers faster via the representation method. In Piotr Sankowski and Christos D. Zaroliagis, editors, *24th Annual European Symposium on Algorithms, ESA 2016, August 22-24, 2016, Aarhus, Denmark*, volume 57 of *LIPIcs*, pages 69:1–69:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.ESA.2016.69.

44 Jesper Nederlof. Bipartite TSP in $O(1.9999^n)$ time, assuming quadratic time matrix multiplication. 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 40–53. ACM, 2020. doi:10.1145/3357713.3384264.

45 Jesper Nederlof, Jakub Pawlewicz, Céline M. F. Swennenhuis, and Karol Wegrzycki. A faster exponential time algorithm for bin packing with a constant number of bins via additive combinatorics. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 1682–1701. SIAM, 2021. doi:10.1137/1.9781611976465.102.

46 Ramamohan Paturi, Pavel Pudlák, Michael E. Saks, and Francis Zane. An improved exponential-time algorithm for k-sat. *J. ACM*, 52(3):337–364, 2005. doi:10.1145/1066100.1066101.

47 R Ravi and Amitabh Sinha. Approximating k-cuts using network strength as a lagrangean relaxation. *European Journal of Operational Research*, 186(1):77–90, 2008.

48 Huzur Saran and Vijay V. Vazirani. Finding k cuts within twice the optimal. *SIAM J. Comput.*, 24(1):101–108, 1995. `doi:10.1137/S0097539792251730`.

49 Uwe Schöning. A probabilistic algorithm for k-sat and constraint satisfaction problems. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 410–414. IEEE Computer Society, 1999. `doi:10.1109/SFFCS.1999.814612`.

50 Mikkel Thorup. Minimum k-way cuts via deterministic greedy tree packing. In Cynthia Dwork, editor, *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 159–166. ACM, 2008. `doi:10.1145/1374376.1374402`.

51 Ryan Williams. A new algorithm for optimal 2-constraint satisfaction and its implications. *Theor. Comput. Sci.*, 348(2-3):357–365, 2005. `doi:10.1016/j.tcs.2005.09.023`.

52 Virginia Vassilevska Williams. Hardness of easy problems: Basing hardness on popular conjectures such as the strong exponential time hypothesis (invited talk). In Thore Husfeldt and Iyad A. Kanj, editors, *10th International Symposium on Parameterized and Exact Computation, IPEC 2015, September 16-18, 2015, Patras, Greece*, volume 43 of *LIPIcs*, pages 17–29. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. `doi:10.4230/LIPIcs.IPEC.2015.17`.

53 Virginia Vassilevska Williams. On some fine-grained questions in algorithms and complexity. In *Proceedings of the International Congress of Mathematicians: Rio de Janeiro 2018*, pages 3447–3487. World Scientific, 2018.

54 Virginia Vassilevska Williams. Some open problems in fine-grained complexity. *SIGACT News*, 49(4):29–35, 2018. `doi:10.1145/3300150.3300158`.

55 Gerhard J. Woeginger. Exact algorithms for np-hard problems: A survey. In Michael Jünger, Gerhard Reinelt, and Giovanni Rinaldi, editors, *Combinatorial Optimization – Eureka, You Shrink!, Papers Dedicated to Jack Edmonds, 5th International Workshop, Aussois, France, March 5-9, 2001, Revised Papers*, volume 2570 of *Lecture Notes in Computer Science*, pages 185–208. Springer, 2001. `doi:10.1007/3-540-36478-1_17`.

56 Gerhard J. Woeginger. Open problems around exact algorithms. *Discret. Appl. Math.*, 156(3):397–405, 2008. `doi:10.1016/j.dam.2007.03.023`.

57 Or Zamir. Breaking the 2^n barrier for 5-coloring and 6-coloring. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 113:1–113:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ICALP.2021.113`.