Breaking the $O(mn)$-Time Barrier for Vertex-Weighted Global Minimum Cut

Julia Chuzhoy*
cjulia@ttic.edu
Toyota Technological Institute at Chicago
Illinois, USA

Ohad Trabelsi
ohadt@ttic.edu
Toyota Technological Institute at Chicago
Illinois, USA

Abstract

We consider the Global Minimum Vertex-Cut problem: given an undirected vertex-weighted graph G, compute a minimum-weight subset of its vertices whose removal disconnects G. The problem is closely related to Global Minimum Edge-Cut, where the weights are on the graph edges instead of vertices, and the goal is to compute a minimum-weight subset of edges whose removal disconnects the graph. Global Minimum Cut is one of the most basic and extensively studied problems in combinatorial optimization and graph theory. While an almost-linear time algorithm was known for the edge version of the problem for awhile (Karger, STOC 1996 and J. ACM 2000), the fastest previous algorithm for the vertex version (Henzinger, Rao and Gabow, FOCS 1996 and J. Algorithms 2000) achieves a running time of $\tilde{O}(mn)$, where m and n denote the number of edges and vertices in the input graph, respectively. For the special case of unit vertex weights, this bound was broken only recently (Li et al., STOC 2021); their result, combined with the recent breakthrough almost-linear time algorithm for Maximum s-t Flow (Chen et al., FOCS 2022, van den Brand et al., FOCS 2023), yields an almost-linear time algorithm for Global Minimum Vertex-Cut with unit vertex weights.

In this paper we break the 28 years old bound of Henzinger et al. for the general weighted Global Minimum Vertex-Cut, by providing a randomized algorithm for the problem with running time $O(\min\{mn^{0.99+o(1)}, m^{1.5+o(1)}\})$.

CCS Concepts

• **Theory of computation → Network flows**.

Keywords

Maximum-Flow, Global Minimum-Cut

*Supported in part by NSF grant CCF-2402283 and NSF HDR TRIPODS award 2216899.

STOC '25, Prague, Czechia

ACM ISBN 979-8-4007-1510-5/25/06
https://doi.org/10.1145/3717823.3718185

ACM Reference Format:

Julia Chuzhoy and Ohad Trabelsi. 2025. Breaking the $O(mn)$-Time Barrier for Vertex-Weighted Global Minimum Cut. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing (STOC '25), June 23–27, 2025, Prague, Czechia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3717823.3718185

1 Introduction

We consider the Global Minimum Vertex-Cut problem: given an undirected n-vertex and m-edge graph $G = (V, E)$ with integral weights $0 \leq w(v) \leq W$ on its vertices $v \in V$, compute a subset $S \subseteq V$ of vertices, such that the graph $G \setminus S$ is not connected. A closely related and extensively studied problem is Global Minimum Edge-Cut: here, the input graph has weights on its edges instead of vertices, and the goal is to compute a minimum-weight subset of edges, whose removal disconnects the graph. The Global Minimum Cut problem is among the most fundamental and extensively studied in computer science, with algorithms dating back to the 1960's [18, 33, 40, 46]. The famous edge contraction-based algorithm of Karger and Stein [31] for the edge version of the problem is one of the gems of Theoretical Computer Science and is routinely taught in basic Algorithms classes. The notion of a Global Minimum Cut is closely related to the central Graph-Theoretic notions of edge- and vertex-connectivity. The edge-connectivity of a graph G is the smallest value λ, such that, for every pair u, v of vertices of G, there is a collection λ of edge-disjoint paths connecting u to v. Vertex-connectivity is defined similarly, except that the paths are required to be internally vertex-disjoint. By Menger's Theorem [41], if G is a graph with unit vertex weights, then the value of the Global Minimum Vertex-Cut in G is equal to its vertex-connectivity. One can naturally extend the notion of vertex-connectivity to the vertex-weighted setting, where a collection of λ internally disjoint u-v paths is replaced with a u-v flow of value λ, that obeys the vertex capacities defined by their weights. Similarly, the value of the Global Minimum Edge-Cut in a graph with unit edge weights is equal to its edge-connectivity, and the value of the Global Minimum Edge-Cut in a graph with arbitrary edge weights can be thought as corresponding to the "weighted" notion of edge-connectivity, where edge-disjoint paths are replaced by flows.

The edge version of the Global Minimum Cut problem has been studied extensively, and is now reasonably well understood in undirected graphs. The classical algorithm of Karger and Stein [31] mentioned above achieves running time $\tilde{O}(n^2)$, improving upon the previous fastest $\tilde{O}(mn)$-time algorithm by Hao and Orlin [24].

Later, Karger [30] obtained a randomized algorithm with near optimal running time $\tilde{O}(m)$, using tree-packing and dynamic programming. This running time was recently matched by a deterministic algorithm [26]. The same problem was also studied in the directed setting: Cen et al. [9], building on some of the ideas from [30], recently presented a randomized algorithm for the directed edge version with running time $O(\sqrt{n} \cdot m^{1+o(1)})$.

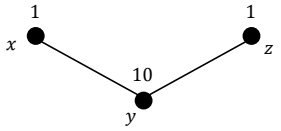
Despite this, progress on the vertex-cut version of the problem has been much slower. In 1996, Henzinger et al. [28] presented an algorithm for vertex-weighted Global Minimum Cut with running time $\tilde{O}(mn)$. In the 28 years since then, no significant progress has been made on the general vertex-weighted version. This lack of progress is likely due to several existing technical barriers to adapting the approaches used for the edge-cut version of the problem. For example, while there are at most $\binom{n}{2}$ minimum cuts in the undirected edge-cut setting, the number of minimum vertex-cuts can be as large as $\Theta(2^k \cdot (n/k)^2)$ even in the unweighted setting, where k is the value of the optimal solution [29]. Furthermore, the tree-packing method, such as the one used in [9, 30], is not known to provide similar guarantees in the vertex capacitated setting. Only recently, the $O(mn)$ barrier was broken for the *unweighted* version of Global Minimum Vertex-Cut by Li et al. [36]. Their algorithm, combined with the recent breakthrough almost-linear time algorithm for Maximum s-t Flow [14, 49], achieves a running time of $O(m^{1+o(1)})$. They also obtain an $O(mn^{1-1/12+o(1)})$-time algorithm for the directed version of the problem with unit vertex weights.

We note that the recent progress on fast algorithms for Maximum s-t Flow mentioned above does not appear to directly imply algorithms for Global Minimum Cut with general vertex weights whose running time is below $\Theta(mn)$. Moreover, while, in the context of Maximum s-t Flow, it is well known that a vertex-capacitated version of the problem in undirected graphs can be cast as a special case of the directed edge-capacitated version, this connection is not known in the context of Global Minimum Vertex-Cut. Unfortunately, the standard reduction, where an undirected vertex-weighted graph is replaced by the corresponding edge-weighted directed *split graph*[1], does not work for the Global Minimum Vertex-Cut problem (see Figure 1), and so the recent algorithms for the directed Global Minimum Edge-Cut problem of [9] cannot be directly leveraged to obtain a faster algorithm for undirected Global Minimum Vertex-Cut. This raises the following fundamental question:

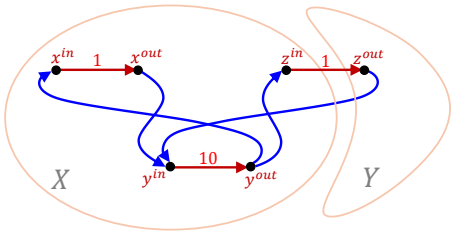
Q1: *Can the Global Minimum Vertex-Cut problem be solved in time faster than $\Theta(mn)$?*

In this paper, we answer this question affirmatively, by presenting a randomized algorithm for weighted Global Minimum Vertex-Cut with running time $O(\min\{mn^{0.99+o(1)}), m^{3/2+o(1)}\})$. Our result also resolves an open question posed by [44] regarding the existence of an $o(n^2)$-time algorithm for the problem, for the setting where $m = O(n)$. We introduce several new technical tools and expand the scope of some existing techniques; we hope that these tools and techniques will lead to even faster algorithms for the problem, and will be useful in other graph-based problems. We now provide a more detailed overview of previous work, followed by the formal statement of our results and a high-level overview of our techniques.

[1]A split graph of a vertex-weighted undirected graph G is obtained by replacing every vertex $v \in V(G)$ by a directed edge $(v^{\text{in}}, v^{\text{out}})$ of capacity $w(v)$, and every edge (x, y) of G by a pair of edges $(x^{\text{out}}, y^{\text{in}})$ and $(y^{\text{out}}, x^{\text{in}})$ of infinite capacity.



(a) A vertex-weighted graph G. The unique global minimum vertex-cut in G is $(\{x\}, \{y\}, \{z\})$, of value OPT = 10.



(b) The corresponding split graph G'; the blue edges have infinite capacity. Consider the cut (X, Y), where $Y = \{z^{\text{out}}\}$ and $X = V(G') \setminus Y$. The cut value is 1, but it does not define a vertex cut in G of the same value.

Figure 1: An illustration of the classic split-graph reduction from undirected vertex-weighted graphs to directed edge-weighted graphs, in the context of Global Minimum Vertex-Cut.

1.1 Overview of Previous Related Work

In this subsection we provide a high-level overview of previous results for central variants of the Global Minimum Cut problem, and highlight the main challenges in adapting the techniques used in these results to the weighted Vertex-Cut setting.

Undirected edge-cut version. The famous algorithm of Karger and Stein [31] mentioned above achieves an $\tilde{O}(n^2)$ running time for the problem. The algorithm is iterative. In every iteration, an edge (u, v) is sampled with a probability proportional to its weight, and then u and v are contracted into a single vertex. This process continues until only two vertices remain in the graph, that naturally define an edge-cut, which is then returned by the algorithm. Intuitively, the algorithm succeeds in computing a Global Minimum Edge-Cut with a reasonably high probability, because, if we fix some optimal cut (X, Y), then the edges crossing the cut have a relatively small weight, making it not very likely that an edge crossing the cut is chosen for contraction. This method is clearly tailored to the (undirected) edge-cut version of the problem.

In a later paper, Karger [30] presented an $\tilde{O}(m)$-time algorithm, that achives a near-optimal running time for the same setting. This algorithm is based on the seminal result by Tutte and Nash-Williams [45, 48], which showed that any unweighted c-edge-connected graph contains a collection of $c/2$ edge-disjoint spanning trees. Karger [30] made a crucial observation that the edges of a minimum cut must be partitioned across these $c/2$ spanning trees, and so there must be some tree T in the packing, such that the number of edges of T crossing the cut is at most 2; if the number of edges of a tree T crossing a cut (X, Y) is at most i, then we say that the cut *i-respects* the tree. To find the global minimum cut, the algorithm identifies, for each tree in the packing, the minimum cut among those that 2-respect the tree, and returns the smallest resulting cut. To make this approach efficient, the following strategy was used. First, in a preprocessing step, the input graph is sparsified by randomly subsampling edges in proportion to their capacities, and treating the resulting "skeleton" graph H as unweighted. This reduces the number of graph edges to $\tilde{O}(n)$ and the minimum cut value to $O(\log n)$, while ensuring that, if (X, Y) is a minimum cut in G, then its value in H is close to the value of the optimal cut in H. Next, $O(\log n)$ trees are packed into H using the algorithm of Gabow [21]. Finally, dynamic programming is used to examine all trees in $\tilde{O}(m)$ time, in order to compute a minimum cut in the original graph. A sequence of works aimed at achieving fast deterministic algorithms for undirected Global Minimum Edge-Cut [27, 32, 34] recently culminated with an $\tilde{O}(m)$-time algorithm [26].

Directed edge-cut version. The fastest current algorithm for this version, due to [9], solves the problem in the time of $O(\sqrt{n})$ applications of Maximum s-t Flow, that, combined with the recent almost-linear time algorithm for the latter problem by [14, 49], yields an $O(\sqrt{n} \cdot m^{1+o(1)})$-time algorithm for directed edge-weighted Global Minimum Cut. At a high level, the algorithm uses the ideas from [30] described above. However, one significant barrier in this setting is that, in the context of directed graphs, a sparsifier that (approximately) preserves all cuts may not exist (see, e.g., [8]). In order to overcome this difficulty, the authors only relied on this method if the global minimum cut is unbalanced; since the number of unbalanced cuts can be suitably bounded even in directed graphs, arguments similar to those from [30] can be used. The case of a balanced global minimum cut is handled by sampling vertices and computing Maximum s-t Flow between a subset of pairs of vertices from that sampled set; it is then shown that, with high probability, this subset contains at least one pair of vertices from opposite sides of the cut.

We note that the tree-packing based techniques used in the above algorithms appear to be unhelpful in the vertex-cut setting, as tree packings have no known analogues in the vertex-connectivity regime that achieve similar guarantees (see [11]).

Unweighted vertex-cut version. This version of the problem received a considerable amount of attention over the years [18, 33, 40, 46]. Recently, a series of results [19, 44] culminated in an almost linear $O(m^{1+o(1)})$-time algorithm for this setting in undirected graphs, along with a somewhat similar $O(mn^{1-1/12+o(1)})$-time algorithm for directed graphs [36]; both of the above bounds follow from the recent breakthrough almost-linear time algorithm for directed Maximum s-t Flow of [14, 49]. Our algorithm for weighted Global Minimum Vertex-Cut builds on several high-level ideas of [36], but requires a significant additional amount of work and new ideas in order to handle the weighted version of the problem. We provide a more detailed description of the algorithm of [36], the barriers to adapting it to the weighted version of the problem, and describe how our algorithm overcomes them in Section 1.2.

Weighted vertex-cut version. Henzinger et al. [28] presented an algorithm with running time $\tilde{O}(mn)$ for this version of the problem, by generalizing the algorithm of Hao and Orlin [24] for the edge-cut version. At a high level, the algorithm of [24] identifies $n-1$ instances of the Maximum s-t Flow problem, such that one of the instances is guaranteed to correspond to the global minimum cut. However, instead of performing $n-1$ applications of the algorithm for Maximum s-t Flow, which would be too time consuming, Hao and Orlin showed that one can solve all these $n-1$ instances in time that is amortized to match the time of a single Maximum s-t Flow computation. However, this approach only works provided that the Maximum s-t Flow algorithm follows the Push-Relabel scheme [23]. Unfortunately, the best current such algorithm has running time $O(mn)$, and the recent almost linear-time algorithms for maximum s-t flow [14, 49] do not follow the Push-Relabel paradigm and hence cannot be leveraged. Henzinger et al. [28] refined this method of [24] and adapted it to the vertex-weighted version of the problem. Their algorithm uses the ideas from [5] to identify a collection of $\tilde{O}(n)$ instances of Maximum s-t Flow, whose solution is sufficient in order to compute global minimum vertex-cut. A combination of these two approaches then yields their $\tilde{O}(mn)$-time algorithm for Global Minimum Vertex-Cut. Another relevant work in this context is the recent combinatorial $O\left(n^{2+o(1)}\right)$-time algorithm for Maximum s-t Flow of [6]. The algorithm proceeds by setting the initial s-t flow f to 0, and then iteratively augmenting it. In every iteration, an approximate Maximum s-t Flow f' in the residual flow network G_f is computed, via an algorithm that can be viewed as following the Push-Relabel scheme. The flow f is then augmented via this flow f', and the algorithm continues until an optimal flow f is obtained. Since this adaptation of the Push-Relabel scheme only computes an approximate flow in the residual flow network, to the best of our understanding, it cannot be directly used to obtain faster algorithms for Global Minimum Vertex-Cut using the approach of [24, 28] described above. Other recent developments on vertex-weighted Global Minimum Cut include a $(1+\varepsilon)$-approximation algorithm with running time $\tilde{O}(n^2/\varepsilon^2)$ by [9], and an exact algorithm with pseudo-polynomial running time $\tilde{O}(\mathsf{OPT} \cdot n \cdot \sum_{u \in V} w(u))$ by [13].

Other Related Work. As mentioned already, there is a long and extensive line of work on Global Minimum Cut and its many variations. We summarize here some additional previous results that are most relevant to us, namely algorithms for the unweighted version of Global Minimum Vertex-Cut. We denote by k the value of the optimal solution (that may be as high as $\Omega(n)$), and by $T(m)$ the running time of the Maximum s-t Flow algorithm on an m-edge graph. One of the earliest algorithms for this problem by [33]

achieved a running time of $O(nk \cdot T(m))$ for undirected graphs. Subsequently, [18, 46] developed algorithms for directed graphs, with a similar running time $O(nk \cdot T(m))$. Even [17] provided an algorithm with a running time of $(n + k^2) \cdot T(m)$, and later [5] designed an algorithm with a running time of $\tilde{O}(n) \cdot T(m)$ for undirected graphs. Linial, Lovász, and Wigderson [40] discovered a surprising connection between the problem and convex hulls, that led to an algorithm with running time $\tilde{O}(n^\omega + nk^\omega)$ for undirected graphs, where ω is the matrix multiplication exponent, that simplifies to $\tilde{O}(n^{1+\omega})$ in the worst case when $k = \Theta(n)$. Their proof mimics a physical process, where an embedding of a graph is computed by viewing its edges as ideal springs, and then letting its vertices settle into an equilibrium. In a follow-up work, [16] extended their result to directed graphs. An algorithm with running time $O(k^3n^{1.5} + k^2n^2)$ was designed for the undirected version of the problem by [15, 43]. Finally, [22] provided an algorithm with running time $\tilde{O}(n+k\sqrt{n})\cdot T(m) = \tilde{O}(n^{1.5})\cdot T(m)$ by using expanders in order to identify a small number of vertex pairs for which an algorithm for the Maximum s-t flow is then executed. More recently, [19] gave an algorithm with $\tilde{O}(m + nk^3)$ running time for undirected graphs and $\tilde{O}(mk^2)$ time for directed graphs, and [47] provided an algorithm with running time $\tilde{O}\left(m^{1+o(1)} \cdot 2^{O(k^2)}\right)$ for undirected graphs. We note that in the worst-case setting where $k = \Omega(n)$, the running times of all these algorithms are at least as high as $\Theta(mn)$.

1.2 Our Results and Techniques

Our main result is summarized in the following theorem.

Theorem 1.1. *There is a randomized algorithm, that, given a simple undirected n-vertex and m-edge graph G with integral weights $0 \leq w(v) \leq W$ on vertices $v \in V(G)$, returns a vertex-cut (A, B, C) in G, such that, with probability at least $1 - 1/n$, cut (A, B, C) is a global minimum vertex-cut in G. The running time of the algorithm is $O\left(\min\{m^{3/2+o(1)}, mn^{0.99+o(1)}\} \cdot (\log W)^{O(1)}\right)$.*

We now provide a high-level overview of the techniques used in the proof of Theorem 1.1, where we mostly focus on the algorithm with running time $O\left(mn^{0.99+o(1)} \cdot (\log W)^{O(1)}\right)$, the most technically challenging part of our result. The high-level intuitive description provided here is somewhat oversimplified and omits some technical details, since we prioritize clarity of exposition over precision in this high-level overview.

Let G be the input simple undirected n-vertex and m-edge graph with integral weights $0 \leq w(v) \leq W$ on its vertices. In order to simplify the discussion in this section, we assume that $W \leq \text{poly}(n)$. For simplicity, we assume that all vertex weights are non-zero; we later show a simple transformation that allows us to achieve this property without changing the problem. Throughout, we use a parameter $0 < \epsilon < 1$. We fix a global minimum vertex-cut (L, S, R), and we assume w.l.o.g. that $w(L) \leq w(R)$. We denote by $\mathsf{OPT} = w(S)$ the value of the optimal solution. It appears that the most difficult special case of the problem is where $|L| \leq n^\epsilon$ and the graph G is dense (e.g., $m \geq n^{1.98}$). We start by providing simple algorithms for other special cases, that essentially reduce the problem to this hard special case.

Algorithm Alg_1: when $|S|$ is small. Our first algorithm, Alg_1, is designed to work in the regime where $|S| \leq |L| \cdot n^{1-\epsilon}$. The algorithm for this special case follows the paradigm that was introduced by [36] in the context of computing global minimum vertex-cut in unweighted graphs. We start by carefully subsampling a subset T of vertices of G in a way that ensures that, with probability at least $\Omega(1/n^{1-\epsilon})$, $|T \cap L| = 1$ and $T \cap S = \emptyset$ holds. If the sampled set T of vertices has these properties, then we say that the sampling algorithm is *successful*. We then use the Isolating Cuts Lemma of [3, 37], combined with the recent almost linear-time algorithm for directed Maximum s-t Flow of [14, 49] (see also Theorem 2.2) to compute, in time $O(m^{1+o(1)})$, for every vertex $v \in T$, a minimum vertex-cut separating v from $T \setminus \{v\}$ in G. Lastly, we return the smallest of the resulting cuts. It is immediate to verify that, if the sampling algorithm is successful, then the cut returned by the algorithm is indeed a global minimum vertex-cut. By repeating the algorithm $\tilde{O}(n^{1-\epsilon})$ times, we ensure that with high probability, in at least one of its executions, the sampling algorithm is successful. The total running time of the algorithm for this special case is $O(mn^{1-\epsilon+o(1)})$.

Algorithm Alg_2: when $|S|$ is large and G is not dense. Our second algorithm, Alg_2, is designed for the regime where $|S| \geq n^{1-\epsilon} \cdot |L|$ and $m \leq n^{2-2\epsilon}$. In this case, we let T be the set of all vertices of G whose degree is at least $n^{1-\epsilon}$. Since $m \leq n^{2-2\epsilon}$, it is easy to verify that $|T| \leq 2n^{1-\epsilon}$. Our main observation is that at least one vertex of T must lie in L. Indeed, it is easy to verify that every vertex of S must have an edge connecting it to some vertex of L, and so the total number of edges incident to the vertices of L is at least $|S|$. Since $|S| \geq n^{1-\epsilon} \cdot |L|$, at least one vertex in L must have degree at least $n^{1-\epsilon}$, so it must lie in T. Next, we provide a random procedure that receives as input a vertex $x \in V(G)$, and returns another vertex $y_x \in V(G)$, such that, if $x \in L$, then with a sufficiently high probability $y_x \in R$ holds. The procedure uses the fact that, if $x \in L$, then the total weight of all vertices of S that are not neighbors of x is bounded by $w(L) \leq w(R)$. Vertex y_x is then selected uniformly at random from $V(G) \setminus (\{x\} \cup N_G(x))$, and, if $x \in L$, then it is guaranteed to lie in R with a constant probability. Lastly, for every vertex $x \in T$, we compute a minimum vertex-cut separating x from y_x in G using the almost-linear algorithm of [14, 49] for directed Maximum s-t Flow (see also Corollary 2.1), and return the cut of the smallest weight from among all the resulting cuts. Since at least one vertex $x \in T$ must lie in L, with a sufficiently high probability $y_x \in R$ must hold, and in this case, the cut that the algorithm returns is indeed a minimum vertex-cut.

By combining Algorithms Alg_1 and Alg_2, we obtain an algorithm for global Minimum Vertex-Cut in non-dense graphs: specifically, if $m \leq n^{2-2\epsilon}$, then the running time of the resulting algorithm is $O\left(mn^{1-\epsilon+o(1)}\right)$. We note that this algorithm can be used directly to obtain a randomized algorithm for global minimum vertex-cut with running time $O\left(m^{3/2+o(1)}\right)$; see details in the full version. We use this algorithm with $\epsilon = 1/100$ to obtain an algorithm with running time $O\left(mn^{0.99+o(1)}\right)$ for graphs in which $m \leq n^{1.98}$ holds. From now

on it is sufficient to design an algorithm for dense graphs, where $m \geq n^{1.98}$. In the remainder of this exposition, we focus on such graphs, and we set $\epsilon = 1/45$ for this part of the algorithm. Note that Algorithm Alg_1 works well even in dense graphs, provided that $|S| \leq |L| \cdot n^{1-\epsilon}$ holds. From now on we assume that this is not the case, so in particular, $|L| \leq \frac{|S|}{n^{1-\epsilon}} \leq n^{\epsilon}$.

A good set of terminals. At a very high level, we employ the notion of *a good set of terminals*, that was (implicitly) introduced by [36], in their algorithm for global Minimum Vertex-Cut in unweighted graphs. Recall that we have fixed a global minimum vertex-cut (L, S, R), where $|L| \leq n^{\epsilon}$. For a vertex set $T \subseteq V(G)$, we say that it is a *good set of terminals* if $|T \cap L| = 1$ holds, and, additionally, if we denote by x the unique vertex in $T \cap L$, then $T \cap S \subseteq N_G(x)$. For every vertex $x \in T$, we denote by $T_x = T \setminus (\{x\} \cup N_G(x))$. We say that (x, T) is a *good pair*, if T is a good set of terminals, and x is the unique vertex in $L \cap T$. Notice that, if (x, T) is a good pair, then $T_x \subseteq R$, and so any minimum vertex-cut separating x from T_x in G is a global minimum cut.

The algorithm of [36] starts by selecting a set T of vertices at random, where every vertex is added to T with probability roughly $1/|L|$. It is not hard to show that, if the graph G is unweighted, then with a reasonably high probability, the resulting set T is a good set of terminals, and moreover, $|T| \leq \tilde{O}(n/|L|)$. For every vertex $x \in T$, let G_x be the graph obtained from G by unifying the vertices of T_x into a destination vertex t_x. For every vertex $x \in T$, the algorithm of [36] computes the value of the minimum x–t_x vertex cut in G_x, and then returns the smallest of the resulting cuts. In order to do so efficiently, they first *sparsify* each such graph G_x, by cleverly employing sparse recovery sketches. With a sufficiently high probability[2], each resulting sparsified graph $\tilde{G}_x$ contains at most $\tilde{O}(\mathsf{OPT} \cdot |L|)$ edges. Applying the almost-linear time algorithm for maximum s-t flow of [14, 49] to each such graph $\tilde{G}_x$ for $x \in T$ then yields an algorithm for computing global minimum cut in G, whose running time, with high probability, is bounded by $O\left((\mathsf{OPT} \cdot |L|)^{1+o(1)} \cdot |T|\right) \leq O\left(\mathsf{OPT} \cdot n^{1+o(1)}\right)$. Finally, it is easy to see that, in the unweighted setting, the degree of every vertex in G must be at least OPT, so $m \geq \Omega(\mathsf{OPT} \cdot n)$ must hold, leading to the bound of $O\left(m^{1+o(1)}\right)$ on the running time of their algorithm.

Our algorithm follows this high-level scheme: we use a randomized procedure, that is adapted to vertex-weighted graphs, in order to select a subset $T \subseteq V(G)$ of vertices, so that, with a reasonably high probability, T is a good set of terminals. Then for every terminal $x \in T$, we compute the value of the minimum x-t_x cut in the corresponding graph G_x, and output the smallest such value. But unfortunately, the algorithm of [36] can no longer be used in order to sparsify each such graph G_x in the weighted setting, and it is not clear how to extend it to this setting. Among other problems, in the weighted setting, the value OPT of the optimal solution may be much higher than n, and we can no longer claim that the degree of every vertex in G is at least OPT. Additionally, in order to ensure that the set T of vertices is a good set of terminals with a sufficiently high probability, we need to sample vertices into T with probability that is proportional to their weight, and other parts of the analysis of the algorithm of [36] do not work in this setting. We now proceed to describe our algorithm in more detail.

We start by computing the *split graph* G' corresponding to graph G, that is a standard step in transforming cut and flow problems in undirected vertex-capacitated graphs to directed edge-capacitated graphs. The set of vertices of G' is $V(G') = \left\{v^{\text{in}}, v^{\text{out}} \mid v \in V(G)\right\}$; we refer to v^{in} and v^{out} as the *in-copy* and the *out-copy* of v. Given a vertex set $X \subseteq V(G)$, we will also denote by $X^{\text{in}} = \left\{x^{\text{in}} \mid x \in X\right\}$, $X^{\text{out}} = \left\{x^{\text{out}} \mid x \in X\right\}$, and $X^* = X^{\text{in}} \cup X^{\text{out}}$. We denote by $V^{\text{in}} = \left\{v^{\text{in}} \mid v \in V(G)\right\}$ and by $V^{\text{out}} = \left\{v^{\text{out}} \mid v \in V(G)\right\}$. The set $E(G')$ of edges of G' is partitioned into two subsets: the set of *special edges*, that contains, for every vertex $v \in V(G)$, the special edge $e_v = (v^{\text{in}}, v^{\text{out}})$ of capacity $c(e_v) = w(v)$, and the set of *regular edges*, that contains, for every edge $(u, v) \in E(G)$, the regular edges $(u^{\text{out}}, v^{\text{in}})$ and $(v^{\text{out}}, u^{\text{in}})$, whose capacities are set to a value $W_{\max} > 4n \cdot W$, that is an integral power of 2.

For every vertex $x \in T$, let G'_x be the graph that is obtained from G' by adding a destination vertex t, and, for every vertex $y \in T_x$, adding a regular edge (y^{in}, t) of capacity $W_{\max}$ to the graph. It is immediate to verify that the value of the minimum x-t_x vertex-cut in G_x is equal to the value of the minimum x^{out}-t edge-cut in G'_x, which, in turn, from the Max-Flow / Min-Cut theorem, is equal to the value of the maximum x^{out}-t flow in G'_x.

Our algorithm computes, for every vertex $x \in T$, a value c_x, that is at least as high as the value of the maximum x^{out}-t flow in G'_x. Additionally, if (x, T) is a good pair, then our algorithm guarantees that, with a sufficiently high probability, c_x is equal to the value of the maximum x^{out}-t flow in G'_x (which, in turn, must be equal to OPT). Let $x \in T$ be the vertex for which c_x is the smallest, and let y be any vertex in T_x. If T is a good set of terminals, then the value of the global minimum cut in G is then guaranteed, with a sufficiently high probability, to be equal to the value of the minimum edge-cut separating x from y in G, which, in turn, is equal to c_x. Computing the minimum x-y vertex-cut in G then yields the optimal global minimum vertex-cut with a sufficiently high probability.

At the heart of the algorithm is our main technical subroutine, that receives as input a set $T \subseteq V(G)$ of vertices and a vertex $x \in T$, together with two copies of the graph G'_x in the adjacency-list representation. The algorithm outputs a value c_x, that is at least as high as the value of the maximum x^{out}-t flow in G'_x. Moreover, if (x, T) is a good pair, then with probability at least $1/2$, c_x is guaranteed to be equal to the value of the maximum x^{out}-t flow in G'_x. In the remainder of this exposition, we focus on the algorithm for this subroutine. For convenience, we denote G'_x by G' and $x^{\text{out}} \in V(G')$ by s. We also denote by OPT_x the value of the maximum s-t flow in G'. We note that, since $|T|$ may be as large as $\Omega(n)$, we need to ensure that the running time of the subroutine is significantly lower than n^2. In particular, if G is sufficiently dense, this running time may need to be lower than $|E(G')|$.

Recall that we have set the parameter $\epsilon = 1/45$. We also use another parameter $n^{7/9} \leq \gamma \leq 2n^{7/9}$, that is an integral power of 2. This setting ensures that $n^{14\epsilon} \leq \gamma \leq 2n^{1-10\epsilon}$. The algorithm for the

[2] In fact they provide a different simple algorithm for the special case where S contains few low-degree vertices, and the above bound only holds for the remaining case; we ignore these technical details in this informal exposition.

subroutine consists of $z = O(\text{poly}\log n)$ *phases*. For all $0 \leq i \leq z$, we also use the parameter $M_i = \frac{W'_{\max}}{2^i \cdot \gamma}$, where $W'_{\max}$ is an integral power of 2 that is greater than the largest weight of any vertex in G.

Intuitively, our algorithm maintains an s-t flow f in G', starting with $f = 0$, and then gradually augments it. The *deficit* of the flow f is $\Delta(f) = \text{OPT}_x - \text{val}(f)$. Specifically, for all $1 \leq i \leq z$, we denote by f_i the flow f at the beginning of the ith phase. We ensure that, for all $1 \leq i \leq z$, flow f_i is M_i-integral, and its deficit is bounded by $4nM_i$. Additionally, for technical reasons, we require that, for every vertex $v \in V(G)$ with $w(v) \geq M_i$, if $e = (v^{\text{in}}, v^{\text{out}})$ is the corresponding special edge, then $f_i(e) \leq w(v) - M_i$; in other words, the residual capacity of the edge e remains at least M_i. Assume first that we can indeed ensure these properties. If z is sufficiently large, then the flow f_z has deficit at most $4nM_z < 1$. We can then construct a graph $\hat{G} \subseteq G'$, that only contains edges $e \in E(G')$ with $f(e) > 0$; the number of such edges is bounded by the total running time of our subroutine so far, which, in turn, is significantly less than $\Theta(n^2)$. Since all edge capacities in $\hat{G}$ are integral, the value of the maximum s-t flow in $\hat{G}$ must be $\lceil \text{val}(f_z) \rceil = \text{OPT}_x$. We then compute the value of the maximum s-t flow in $\hat{G}$ and output it as c_x. Before we describe our algorithm for the preprocessing step and for each phase, we need to introduce the notion of *shortcut operations*.

Shortcut Operations. Over the course of the algorithm, we may slightly modify the graph G' by performing *shortcut operations*. In a shortcut operation, we insert an edge (v, t) into G', for some vertex $v \in V(G')$. If the pair (x, T) is not good, then we say that such an operation is always *valid*. Notice that the insertion of the edge may only increase the value of the maximum s-t flow in G'. Otherwise, if the pair (x, T) is good, then we say that the operation is valid only if $v \in S^{\text{out}} \cup R^*$, that is, v is either an out-copy of a vertex of S, or it is an in-copy or an out-copy of a vertex of R. By inspecting the minimum s-t edge-cut in G' (whose corresponding edge set is $\left\{(u^{\text{in}}, u^{\text{out}}) \mid u \in S\right\}$), it is easy to verify that a valid shortcut operation may not change the value of the maximum s-t flow in G' in this case. We ensure that, with a sufficiently high probability, all shortcut operations that the algorithm performs are valid. For clarity, we continue to denote the original graph G' by G', and we use G'' to denote the graph obtained from G' after our algorithm possibly performed a number of shortcut operations. The flow f that our algorithm maintains is in graph G''. Recall that our algorithm is given as input 2 copies of the graph G' in the adjacency-list representation. We use one of these copies to maintain the graph G'', and we use the second copy in order to maintain the residual graph of G'' with respect to the current flow f, that we denote by H.

The Preprocessing Step. The purpose of the preprocessing step is to compute a flow f_1 that can serve as the input to the first phase. Recall that we require that f_1 is M_1-integral, where $M_1 = \frac{W'_{\max}}{2\gamma}$, and that its deficit is at most $4nM_1$. We also require that, for every vertex $v \in V(G)$ with $w(v) \geq M_1$, if $e = (v^{\text{in}}, v^{\text{out}})$ denotes the corresponding edge of G'', then $f_1(e) \leq w(v) - M_1$. Consider the graph $\hat{G} \subseteq G''$, that is obtained from G'' as follows. First, we delete, for every vertex $v \in V(G)$ with $w(v) < M_1$, both copies of v from the graph. Next, for every remaining special edge e, we decrease the capacity of e by at least M_1 and at most $2M_1$ units, so that the new capacity is an integral multiple of M_1. It is easy to verify that, by computing a maximum s-t flow in the resulting flow network $\hat{G}$, we can obtain the flow f_1 with all desired properties. Unfortunately, it is possible that the graph $\hat{G}$ is very dense, so we cannot compute the flow directly. Instead, we *sparsify* $\hat{G}$ first, using the following two observations, that essentially generalize Equation (2) and Observation 2 of [36] to arbitrary vertex capacities. Let $N = \left\{v \in V(\hat{G}) \mid (s, v) \in E(\hat{G})\right\}$, and let $N' \subseteq V(G)$ contain all vertices u such that a copy of u lies in N. The first observation is that there is an optimal flow in $\hat{G}$ that does not use any edges that enter the vertices of N, except for the edges that start at s, so all such edges can be deleted. Second, it is not hard to prove that the total number of vertices $v \in S \setminus N'$ of weight at least M_1 is bounded by $2|L|\gamma$. It then follows that, if $u \in V(G)$ has more than $3|L|\gamma$ neighbors in $V(G) \setminus N'$ that have weight at least M_1, then at least one such neighbor must lie in R, and so u may not belong to L. We can then safely add a shortcut edge (v^{out}, t) to both G'' and $\hat{G}$, and we can ignore all other edges leaving v^{out}. These two observations allow us to sparsify the graph $\hat{G}$, obtaining a smaller graph $\hat{G}' \subseteq \hat{G}$, such that, on the one hand, $|E(\hat{G}')| \leq \tilde{O}(n^{2-4\epsilon})$, while on the other hand, all shortcut operations performed so far are valid with a high enough probability, so the value of the maximum s-t flow in $\hat{G}'$ is at least as high as that in $\hat{G}$, and the two flow values are equal if (x, T) is a good pair.

Executing a Single Phase. We now provide the description of the ith phase, for $1 \leq i \leq z$. Recall that we receive as input a flow $f = f_i$ in the current graph G'', that is M_i-integral, and whose deficit is bounded by $4nM_i$. Additionally, for every vertex $v \in V(G)$, if $e = (v^{\text{in}}, v^{\text{out}})$ is its corresponding edge in G'', then $f(e) \leq w(v) - M_i$. Therefore, if we denote by $U_i = \{v \in V(G) \mid w(v) \geq M_i\}$, then, for every vertex $v \in U_i$, the residual capacity of the edge $e = (v^{\text{in}}, v^{\text{out}})$ is at least M_i. Throughout, we denote by H the residual flow network of G'' with respect to the current flow f. By the properties of the residual flow network, the value of the maximum s-t flow in H is at least $\text{OPT}_x - \text{val}(f)$ (and it is equal to $\text{OPT}_x - \text{val}(f)$ if (x, T) is a good pair and all shortcut operations so far have been valid). As before, we can consider the graph $\hat{H}$, that is obtained from H by first deleting, for every vertex $u \in V(G) \setminus U_i$, both copies of u, and then reducing, for each remaining forward special edge $(v^{\text{in}}, v^{\text{out}})$, its residual capacity by at least $M_{i+1} = M_i/2$ and at most $2M_{i+1}$, so that it becomes an integral multiple of M_{i+1}. As before, computing a maximum s-t flow in the resulting flow network $\hat{H}$ would yield the flow f_{i+1} with all desired properties. But unfortunately graph $\hat{H}$ may be very dense, and we may not be able to afford to compute the flow directly. As before, we will sparsify this graph first, though the sparsification procedure is much more challenging now. In fact our algorithm may first gradually augment the flow f and add some new shortcut edges, before sparsifying the resulting new graph $\hat{H} \subseteq H$.

The key step in our sparsification procedure is to compute a partition (Q, Q') of the set U_i of vertices of G, so that $|Q' \cap S| < n^{1-4\epsilon}$. Additionally, if $|Q| > n^{1-4\epsilon}$, then we also compute an s-t edge-cut

(X^*, Y^*) in the current residual flow network H, that has the following property: the total residual capacity of the edges $E_H(X^*, Y^*)$ is close to the value of the maximum s-t flow in H. In other words, in any maximum s-t flow in H, the edges of $E_H(X^*, Y^*)$ are close to being saturated. Lastly, we ensure that $|E_H(X^*, Y^*)|$ is relatively small, and, for every vertex $v \in Q$, $v^{\text{in}} \in X^*$ and $v^{\text{out}} \in Y^*$ holds. Assume for now that we have computed the partition (Q, Q') of U_i and the cut (X^*, Y^*) as above. Observe that, if some vertex $v \in V(G)$ has more than $2n^{1-4\epsilon}$ neighbors in Q', then, since $|L| \leq n^{\epsilon}$ and $|Q' \cap S| \leq n^{1-4\epsilon}$, at least one of these neighbors must lie in R, and so v may not lie in L. We can then safely add a shortcut edge (v^{out}, t), and ignore all other edges leaving v^{out} in $\hat{H}$. Assume now that $|Q| > n^{1-4\epsilon}$, and let $E' = \{e = (y, x) \in E(H) \mid y \in Y^*, x \in X^*\}$. Consider any maximum s-t flow f^* in H, and denote its value by F. Since the total capacity of all edges in $E_H(X^*, Y^*)$ is close to F, it is easy to see that the total flow that f^* sends on the edges of E' is relatively low. Indeed, if $\mathcal{P}$ is a flow-path decomposition of f^*, and $\mathcal{P}' \subseteq \mathcal{P}$ is the subset of paths containing the edges of E', then every path in $\mathcal{P}'$ must use at least two edges of $E_H(X^*, Y^*)$, so these paths may only carry a relatively small amount of flow. We can then delete the edges of E' from the sparsified graph without decreasing the value of the maximum s-t flow by too much. We use these and other observation in order to sparsify the graph $\hat{H}$, to obtain a graph $\hat{H}'$ with $|E(\hat{H}')| \leq O(n^{2-4\epsilon+o(1)})$, such that the value of the maximum s-t flow in $\hat{H}'$ is close to that in $\hat{H}$, and all edge capacities in $\hat{H}'$ are integral multiples of M_{i+1}. Computing the maximum s-t flow in $\hat{H}'$ then yields the desired flow f_{i+1}, that serves as the output of the current phase.

Computing the Partition (Q, Q'). The most technically challenging part of our algorithm is computing the partition (Q, Q') of U_i, and, if needed, the cut (X^*, Y^*) in H with the properties described above. Our first step towards this goal is to compute a subgraph $J \subseteq H$ with $s \in J$ and $t \notin J$, such that $|V^{\text{out}} \cap V(J)|$ is small, and additionally, for every vertex $u \notin J$, the total number of edges in set $\{(x, u) \mid x \in V(J)\}$ is relatively small, as is their total capacity in H. Assume first that we have computed such a graph J. We start with an initial partition (Q, Q') of U_i, where Q contains all vertices $v \in U_i$ for which $v^{\text{in}} \in V(J)$ and $v^{\text{out}} \notin V(J)$, and $Q' = U_i \setminus Q$. We show that $|Q' \cap S|$ must be relatively small. Consider now a graph $\tilde{H}$, that is obtained from H, by unifying all vertices of $V(H) \setminus V(J)$ into a new destination vertex t'; we keep parallel edges and discard self-loops. Note that, for every edge $e = (x, y) \in E(H)$ with $x \in V(J)$ and $y \notin V(J)$, there is an edge (x, t') corresponding to e in $\tilde{H}$; we do not distinguish between these two edges. Let F denote the value of the maximum s-t flow in H, and let F' denote the value of the maximum s-t' flow in $\tilde{H}$. Our key observation is that, on the one hand, F' is quite close to F, while, on the other hand, if f' is any maximum s-t' flow in $\tilde{H}$, then the total flow on the edges $\left\{(v^{\text{in}}, v^{\text{out}}) \mid v \in Q \cap S\right\}$ must be close to their capacity. In other words, if we compute any maximum s-t' flow f' in $\tilde{H}$, and a subset $B \subseteq Q$ of vertices, such that the total amount of flow on the edges of $\left\{(v^{\text{in}}, v^{\text{out}}) \mid v \in B\right\}$ is significantly lower than their capacity, then only a small fraction of vertices of B may lie in S; we call such a vertex set $B \subseteq Q$ a *bad batch*. Our algorithm iteratively computes bad batches $B \subseteq Q$ of a sufficiently large cardinality, and then moves all vertices of B from Q to Q'. In order to compute a bad batch, we set up an instance of Min-Cost Maximum s-t' Flow in $\tilde{H}$, by setting the *cost* of every edge (v^{in}, t') corresponding to vertices $v \in Q$ to 1, and setting the costs of all other edges to 0. Once the algorithm terminates, we are guaranteed that Q may no longer contain large bad batches of vertices, or, equivalently, in any maximum s-t' flow in $\tilde{H}$, the vast majority of the edges in $\left\{(v^{\text{in}}, v^{\text{out}}) \mid v \in Q\right\}$ are close to being saturated. If $|Q| \leq n^{1-4\epsilon}$, then we terminate the algorithm with the resulting partition (Q, Q') of U_i. Otherwise, we perform one additional step that computes a cut (X, Y) in the graph $\tilde{H}$, that in turn defines an s-t cut (X^*, Y^*) in H with the desired properties. Note that this step crucially relies on the fact that the graph $\tilde{H}$ is relatively small, which, in turn, follows from the fact that $|V(J) \cap V^{\text{out}}|$ is relatively small, and so is the number of edges $(x, y) \in E(H)$ with $x \in V(J)$ and $y \notin V(J)$.

Computing the Subgraph J. Recall that, in order to compute the partition (Q, Q') as described above, we need first to compute a subgraph $J \subseteq H$ with $s \in J$ and $t \notin J$, such that $|V^{\text{out}} \cap V(J)|$ is small, and additionally, for every vertex $u \notin J$, the total number of edges in set $\{(x, u) \mid x \in V(J)\}$ is relatively small, and so is their total capacity in H. We do not compute this subgraph directly. Instead, we perform a number of iterations. In every iteration, we either add a shortcut edge and augment the current flow f by a significant amount; or we compute the subgraph J of the current residual network H with the desired properties and terminate the algorithm. We note that, as we add shortcut edges, and as the flow f is augmented, the residual flow network H is updated accordingly, and the graph J that our algorithm eventually computes is a subgraph of the current residual flow network H. Our algorithm for this step relies on the technique of *local flow augmentations*, that was first introduced by [12] in the context of the Maximal 2-Connected Subgraph problem. The technique was later refined and applied to *unweighted* and *approximation* versions of the global minimum vertex-cut problem [13, 19, 44].

We first provide a high-level intuitive description of this technique in the context of past work on unweighted global minimum vertex-cut, and then describe our approach that extends and generalizes this technique. Our description here is somewhat different from that provided in previous work, as we rephrase it to match the terminology used in this overview. Consider an instance G of the unweighted Global Minimum Vertex-Cut problem, and assume that there is an optimal solution (L, S, R) where $|L|$ is small. Suppose we are given a vertex[3] $v \in L$. We construct a split graph G' as before, and add a destination vertex t to it. Intuitively, our goal is to compute a flow of value $|S|$ from $s = v^{\text{out}}$ to t in G', while we are allowed to add valid shortcut edges to G'. We start with a flow $f = 0$, and then iterate. In every iteration, we perform a DFS search in the residual flow network H of G' corresponding to the current flow f, and, once the search discovers roughly $|L| \cdot |S|$ vertices, we terminate it. Let X be the set of vertices that the search has discovered. We note that, if G is a graph with unit vertex weights, and f is an

[3] In fact a vertex v is selected from G via some random process, and we are only guaranteed that with some reasonably high probability, $v \in L$; this is similar to our setting where we first select a random set T of vertices that is a good set of terminals with a sufficiently high probability, and then select a random vertex $v \in T$.

integral flow, then every vertex of V^{in} has exactly one edge leaving it in H, and the other endpoint of the edge lies in V^{out}. Additionally, every vertex of V^{out} has exactly one incoming edge in H. Therefore, at least half the vertices of X must lie in V^{out}. We select a vertex $u \in X \cap V^{\text{out}}$ uniformly at random, and add a shortcut edge (u, t) to G'. Since $|X| \geq |L| \cdot |S|$, with probability at least $(1 - 1/|S|)$, $u \in S^{\text{out}} \cup R^{\text{out}}$, and so the shortcut operation is valid. Let P be the path connecting s to u that the search computed. We augment f by sending 1 flow unit via the path $P \circ (u, t)$ and terminate the iteration. Lastly, if the DFS search only discovers fewer than $|L| \cdot |S|$ vertices, then the total number of vertices reachable from s in H is small, and we can compute a maximum s-t flow in H directly. Since the number of iterations is bounded by $\mathsf{OPT} = |S|$, with a reasonably high probability, all shortcut operation are valid, and so we compute the value of the Global Minimum Vertex-Cut correctly.

There are several issues with adapting this algorithm to the vertex-weighted setting. The first issue is that, if G is a graph with arbitrary vertex weights, then the residual flow network H may no longer have the property that every vertex of V^{out} has exactly one incoming edge; the number of such edges may be large. As the result, it is possible that the vast majority of the vertices in the set X discovered by the DFS lie in V^{in}. But we cannot select vertices of V^{in} for shortcut operations, since we are not allowed to add edges connecting vertices of S^{in} to t, and it is possible that $|S| = \Omega(n)$. Therefore, our DFS search needs to ensure that the number of vertices of V^{out} that it discovers is sufficiently large. To achieve this high number of vertices of V^{out}, it may be forced to explore a significantly larger number of vertices of V^{in}, leading to a much higher running time of each iteration. In order to obtain a fast overall running time, we then need to ensure that the number of iterations is sufficiently low, and this, in turn, can be ensured by sending a large amount of flow in each iteration. Therefore, when performing a DFS search in H, we only consider edges whose capacities are quite high, at least $\gamma \cdot M_i$. Once the DFS search terminates with a set X containing a relatively low number of vertices of V^{out}, we could let J be the subgraph of H induced by the vertices of X. This approach could indeed be used to guarantee that $|E(J)|$ is low, but unfortunately it does not ensure the other crucial property that we need: that every vertex $u \in V(H) \setminus V(J)$ has relatively few edges (u', u) with $u' \in V(J)$, and that all such edges have a relatively low capacity.

In order to overcome these difficulties, we significantly expand the scope of the local-search technique. In every iteration of the local-search algorithm, we explore the graph H, starting from the vertex s, as follows. Initially, we let $X = \{s\}$. Whenever we encounter an edge (u, v) with $u \in X$ and $v \notin X$, whose capacity in H is at least $\gamma \cdot M_i$, we add v to X. Additionally, whenever we discover a vertex $x \notin X$, such that, for some integer i, there are at least 2^i edges in set $\{(y, x) \in E(H) \mid y \in X\}$, whose capacity is at least $M_i \cdot \gamma / 2^i$, we add x into X. We show that, for every vertex v that is ever added to X via one of the above rules, there is an s-v flow in $H[X]$ of value $\gamma \cdot M_i$. An iteration terminates once $|X \cap V^{\text{out}}|$ is sufficiently large, or once neither of the above rules can be applied to expand X. In the latter case, we let $J = H[X]$, and we show that J has all required properties. In the former case, we select a vertex $v \in X \cap V^{\text{out}}$ uniformly at random, and connect it to t via a shortcut operation. We then compute an s-v flow f' of value $\gamma \cdot M_i$ in graph $H[X]$, which is guaranteed to contain relatively few edges, and augment the current flow f by sending $\gamma \cdot M_i$ flow units from s to t via the flow f', that is extended via the edge (v, t) to reach t. We note that, unlike previous work that used the local-flow technique, where the flow augmentation in every iteration was performed via a single path, we augment the flow f via the flow f' that may be more general.

To summarize, the algorithm for a single phase consists of three steps. In the first step, we compute the subgraph $J \subseteq H$, after possibly augmenting the initial flow and adding new shortcut edges. This step exploits the expanded local-search technique. In the second step we compute the partition (Q, Q') of U_i, and, if required, an s-t cut (X^*, Y^*) in H. The former is done by repeated applications of the algorithm for min-cost maximum flow, and the latter is achieved by computing a minimum s-t cut in an appropriately constructed graph. Lastly, in the third step, we sparsify the residual flow network H, and compute the desired flow f_{i+1}. This step relies on the properties of the partition (Q, Q') and of the cut (X^*, Y^*).

A Subgraph Oracle. The algorithm described above needs an access to a subroutine, that we refer to as the *subgraph oracle*. The oracle is given access to an undirected graph G in the adjacency-list representation, a subset Z of its vertices, and a parameter δ. The oracle must return a partition of $V(G)$ into two subsets Y^ℓ and Y^h such that, for every vertex $v \in Y^h$, the number of neighbors of v that lie in Z is at least roughly $n^{1-\delta}$, while for every vertex $u \in Y^\ell$, the number of such neighbors is at most roughly $n^{1-\delta}$. Additionally, the algorithm must return the set $E' = E_G(Y^\ell, Z)$ of edges, and its running time should be at most $n^{2-\Theta(\delta)}$. Our main technical subroutine, when processing a vertex $x \in T$, performs at most $z \leq \text{poly} \log n$ calls to the oracle, with at most one such call performed per phase. While it is not hard to design an algorithm that computes the partition (Y^ℓ, Y^h) of $V(G)$ with the desired properties by using the standard sampling technique, computing the set E' of edges within the desired running time appears much more challenging. Instead, we design an algorithm that can simultaneously process up to n such subgraph queries in bulk, in time $O(n^{3-\Theta(\delta)})$. This algorithm casts the problem of computing the sets E' of edges for all these queries simultaneously as an instance of the Triangle Listing problem in an appropriately constructed graph, and then uses the algorithm of [7] for this problem. Our final algorithm for the Global Minimum Vertex-Cut problem proceeds in z iterations. In each iteration i, for every vertex $x \in T$, we run the algorithm for processing x described above, while recording all random choices that the algorithm makes, until it performs the ith call to the subgraph oracle, and then we terminate it. Once we obtain the ith query to the subgraph oracle from each of the algorithms that process vertices of T, we ask the subgraph oracle all these queries in bulk. Then iteration $(i + 1)$ is performed similarly, except that we follow all random choices made in iteration i. Since we now have the response to the ith query to the subgraph oracle, we can continue executing the algorithm processing each vertex $x \in T$ until it performs the $(i + 1)$th such query.

In the remainder of this extended abstract we provide a high-level description of our algorithm, with many technical details deferred to the full version.

2 The Algorithm

2.1 Preliminaries

2.1.1 Parameters $W'_{\max}$, $W_{\max}$. Given an undirected graph $G = (V, E)$ with integral weights $w(v) \geq 0$ on its vertices $v \in V(G)$, we denote by $W'_{\max}(G)$ the smallest integral power of 2, such that $W'_{\max}(G) > \max_{v \in V} \{w(v)\}$ holds. We let $W_{\max}(G)$ be the smallest integral power of 2 with $W_{\max} \geq 2n \cdot W'_{\max}$. When the graph G is clear from context, we denote $W'_{\max}(G)$ and $W_{\max}(G)$ by $W'_{\max}$ and $W_{\max}$, respectively. Notice that: $W_{\max}(G) \leq 4n \cdot W'_{\max}(G) \leq 8n \cdot \max_{v \in V} \{w(v)\}$.

2.1.2 Fast Algorithms for Flows and Cuts. Our algorithm relies on the recent breakthrough almost linear time algorithm for Minimum Cost Maximum s-t Flow [14, 49]. We use its following version (see details in the full version).

Corollary 2.1. *There is a deterministic algorithm, that receives as input an undirected m-edge graph G with integral weights $1 \leq w(v) \leq W$ on vertices $v \in V(G)$, together with two disjoint subsets $S, T \subseteq V(G)$ of vertices. The algorithm computes a minimum S-T vertex-cut (X, Y, Z) in G. The running time of the algorithm is $O\left(m^{1+o(1)} \cdot \log W\right)$.*

2.1.3 Computing Minimum Cuts via Isolating Cuts. The Isolating Cuts lemma was introduced independently in [3, 37], and it found many applications since (see, e.g. [1, 2, 4, 10, 13, 20, 25, 26, 35, 36, 38, 39, 42, 50]). We use the following theorem, that generalizes the version of the Isolating Cuts lemma from [36] that focuses on unweighted graphs to graphs with arbitrary vertex weights. The proof closely follows the arguments from [36], and is provided in the full version of the paper.

Theorem 2.2 (Isolating Cuts). *There is a deterministic algorithm, that, given as input an undirected m-edge graph G with integral weights $0 \leq w(v) \leq W$ on its vertices $v \in V(G)$, together with a subset $T \subseteq V(G)$ of vertices of G, computes, for every vertex $v \in T$, the value of the minimum vertex cut separating v from $T \setminus \{v\}$. The running time of the algorithm is $O\left(m^{1+o(1)} \cdot \log W\right)$.*

2.1.4 Properties of the Global Minimum Vertex-Cut. In this subsection we establish a useful property of a global minimum vertex-cut in undirected graphs. Consider a vertex-weighted graph G, let (L, S, R) be a global minimum vertex-cut in G, and let $x \in L$ be any vertex. The following simple claim shows that, in a sense, almost all vertices of S must be neighbors of x in G. In other words, the total weight of all vertices of S that are not neighbors of x is bounded by $w(L)$. The following claim, whose proof is deferred to the full version of the paper, can be thought of as a generalization of Proposition 3.2 in [36] from unit vertex weights to general weights.

Claim 2.3. *Let G be a simple undirected graph with integral weights $w(v) > 0$ on its vertices $v \in V(G)$, and let (L, S, R) be a global minimum vertex-cut in G. Let $x \in L$ be any vertex, and denote by $S' = S \setminus N_G(x)$. Then $w(S') \leq w(L)$ must hold.*

2.2 Algorithm Description

Next, we provide our main result, namely an algorithm for the vertex-weighted global minimum vertex cut, proving Theorem 1.1. Due to lack of space, we defer many technical details to the full version of the paper.

We assume that we are given a simple undirected n-vertex and m-edge graph G with integral weights $0 \leq w(v) \leq W$ on its vertices. It will be convenient for us to assume that all vertex weights are strictly positive. In order to achieve this, we can modify the vertex weights as follows: for every vertex $v \in V(G)$, we let $w'(v) = n^2 \cdot w(v) + 1$. Let G' be the graph that is identical to G, except that the weight of every vertex v is set to $w'(v)$, and let $W' = n^2 \cdot W + 1$, so for every vertex $v \in V(G)$, $1 \leq w'(v) \leq W'$ holds. Note that (L, S, R) is a global minimum vertex cut in G' if and only if it is also a global minimum vertex cut in G. In order to avoid clutter, we will denote G' by G, and the vertex weights $w'(v)$ by $w(v)$ for $v \in V(G)$. From now on we assume that, for every vertex $v \in V(G)$, $1 \leq w(v) \leq W'$, where $W' = n^2 \cdot W + 1$. We define the values $W'_{\max}(G)$ and $W_{\max}(G)$ as in Section 2.1.1, and denote them by $W'_{\max}$ and $W_{\max}$, respectively, throughout this section. Note that: $W_{\max} \leq O(nW') \leq O(n^3 \cdot W)$. Let $\rho = \lceil \log(W') \rceil$. For all $0 \leq i \leq \rho$, we let $U_i \subseteq V$ denote the set of all vertices v with $2^i \leq w(v) < 2^{i+1}$, so $(U_0, \ldots, U_\rho)$ is a partition of $V(G)$.

Whenever we are given any vertex cut (L, S, R) of G, we assume without loss of generality that $w(L) \leq w(R)$ holds. For all $0 \leq i \leq \rho$, we then denote $L_i = L \cap U_i$, $S_i = S \cap U_i$, and $R_i = R \cap U_i$, so that (L_i, S_i, R_i) is a partition of U_i.

We use a combination of several algorithms, that are designed to handle different edge densities and different tradeoffs between the values $|L_i|$ and $|S_i|$ for the optimal cut (L, S, R), for $0 \leq i \leq \rho$. Each such algorithm must return an estimate c on the value of the optimal solution, together with a pair s,t of vertices of G, such that c is at least as large as the value of the minimum s-t vertex-cut in G. We will ensure that, with high probability, at least one of the algorithms returns an estimate $c = \mathsf{OPT}$. Taking the smallest estimate c returned by any of these algorithms then yields the optimal solution value with high probability, and computing the minimum s-t vertex cut for the corresponding pair s, t of vertices provides an optimal global minimum cut in G. We now describe each of the algorithms in turn.

2.3 Algorithm Alg_1: when S_i is Small for Some i

Our first algorithm, Alg_1, is summarized in the following theorem.

Theorem 2.4. *There is a randomized algorithm, that we denote by Alg_1, whose input is a simple undirected n-vertex and m-edge graph G with integral weights $1 \leq w(v) \leq W'$ on vertices $v \in V(G)$, and a parameter $0 < \epsilon < 1$. The algorithm returns an estimate c on the value OPT of the global minimum vertex cut in G, and two vertices $x, y \in V(G)$, such that $c \geq \mathsf{OPT}$ holds, and the value of the minimum*

x-y vertex-cut in G is at most c. Moreover, if there is a global minimum vertex-cut (L,S,R) in G, and an integer $0 \leq i \leq \lceil \log W' \rceil$, such that $L_i \neq \emptyset$ and $|S_i| \leq |L_i| \cdot n^{1-\epsilon}$, then, with probability at least $1-1/n^4$, $c = \mathsf{OPT}$. The running time of the algorithm is $O\left(mn^{1-\epsilon+o(1)} \cdot \log^2 W'\right)$.

In the remainder of this subsection, we prove the above theorem. At a high level, Algorithm Alg_1 uses the paradigm of subsampling a set T of vertices of G and then applying the Isolating Cuts Lemma (Theorem 2.2) to T; this paradigm was introduced by [36] in the context of computing global minimum vertex-cut in unweighted graphs. The following lemma is central to the proof of the theorem.

Lemma 2.5. *There is a randomized algorithm, whose input is a simple undirected n-vertex and m-edge graph G with integral weights $1 \leq w(v) \leq W'$ on vertices $v \in V(G)$, two integral parameters $0 \leq i \leq \lceil \log W' \rceil$ and $0 < \lambda \leq n$, and another parameter $0 < \epsilon < 1$. The algorithm returns an estimate c on the value OPT of the global minimum vertex-cut in G, and two vertices $x, y \in V(G)$, such that $c \geq \mathsf{OPT}$ holds, and the value of the minimum x-y vertex-cut in G is at most c. Moreover, if there is a global minimum vertex-cut (L,S,R) in G with $\lambda \leq |L_i| + |S_i| \leq 2\lambda$, such that $L_i \neq \emptyset$ and $|S_i| \leq |L_i| \cdot n^{1-\epsilon}$, then, with probability at least $1 - 1/n^4$, $c = \mathsf{OPT}$ holds. The running time of the algorithm is $O\left(mn^{1-\epsilon+o(1)} \cdot \log W'\right)$.*

We prove Lemma 2.5 below, after we complete the proof of Theorem 2.4 using it. Let $z = \lceil \log W' \rceil$ and let $z' = \lceil \log n \rceil$. Our algorithm performs $O(z \cdot z')$ iterations, as follows. For all $0 \leq i \leq z$ and for all $0 \leq j \leq z'$, we apply the algorithm from Lemma 2.5 to graph G, with parameter i and $\lambda = 2^j$, and parameter ϵ remaining unchanged; we denote this application of the algorithm from Lemma 2.5 by $\mathcal{A}_{i,j}$, and we denote by $(c_{i,j}, x_{i,j}, y_{i,j})$ its output. We let i^*, j^* be the indices for which the value c_{i^*,j^*} that Algorithm $\mathcal{A}_{i,j}$ returned is the smallest, breaking ties arbitrarily. We then output $(c_{i^*,j^*}, x_{i^*,j^*}, y_{i^*,j^*})$. This completes the description of Algorithm Alg_1. Since the running time of the algorithm from Lemma 2.5 is $O(mn^{1-\epsilon+o(1)} \cdot \log W')$, the running time of Alg_1 is $O(mn^{1-\epsilon+o(1)}) \cdot O(z \cdot z') \leq O(mn^{1-\epsilon+o(1)} \cdot \log^2 W')$. Moreover, Lemma 2.5 ensures that $c_{i^*,j^*} \geq \mathsf{OPT}$ holds, and that the value of the minimum x_{i^*,j^*}-y_{i^*,j^*} vertex cut in G is at most c_{i^*,j^*}.

Assume now that there is a global minimum vertex cut (L,S,R) in G, and an integer $0 \leq i \leq \lceil \log W' \rceil$, such that $L_i \neq \emptyset$ and $|S_i| \leq |L_i| \cdot n^{1-\epsilon}$. Let $0 \leq j \leq z'$ be the unique integer with $2^j \leq |L_i| + |S_i| < 2^{j+1}$. We say that Algorithm $\mathcal{A}_{i,j}$ is *successful*, if its output $(c_{i,j}, x_{i,j}, y_{i,j})$ has the property that $c_{i,j} = \mathsf{OPT}$. From Lemma 2.5, with probability at least $1 - 1/n^4$, Algorithm $\mathcal{A}_{i,j}$ is successful. In this case we are guaranteed that $c_{i^*,j^*} = \mathsf{OPT}$ as well. In order to complete the proof of Theorem 2.4, it is now enough to prove Lemma 2.5.

Proof of Lemma 2.5. The proof follows the high level idea of [36], that was introduced in the context of computing global minimum vertex-cuts in unweighted graphs, of subsampling a subset T of vertices of G, and then applying the Isolating Cuts Lemma (Theorem 2.2) to it.

We start by describing an algorithm, that we denote by $\mathcal{A}'$. Our final algorithm will execute $\mathcal{A}'$ several times. Algorithm $\mathcal{A}'$ starts by constructing a random set T of vertices as follows: every vertex $v \in U_i$ is added to T independently with probability $\frac{1}{2\lambda}$. We then use the algorithm from Theorem 2.2 to compute, for every vertex $v \in T$, the value c_v of the minimum vertex-cut separating v from $T \setminus \{v\}$ in G. We let $x \in T$ be the vertex for which the value c_x is the smallest, breaking ties arbitrarily, and we let y be an arbitrary vertex in $T \setminus \{x\}$. We then return c_x, x and y as the outcome of the algorithm $\mathcal{A}'$. Note that the running time of Algorithm $\mathcal{A}'$ is $O(m^{1+o(1)})$. It is easy to verify that, for all $v \in T$, $c_v \geq \mathsf{OPT}$ must hold, since c_v is the value of a vertex-cut in G. In particular, $c_x \geq \mathsf{OPT}$ must hold as well. Moreover, since there is an x-y vertex-cut of value c_x in G (the cut separating x from $T \setminus \{x\}$), we get and the value of the minimum x-y vertex-cut in G is at most c_x. We say that the algorithm $\mathcal{A}$ is *successful* if $c_x = \mathsf{OPT}$; otherwise we say that it is *unsuccessful*. We will use the following claim to analyze algorithm $\mathcal{A}'$, whose proof is deferred to the full version of the paper.

Claim 2.6. *Suppose that there is a global minimum vertex-cut (L,S,R) in G with $\lambda \leq |L_i| + |S_i| \leq 2\lambda$, such that $L_i \neq \emptyset$ and $|S_i| \leq |L_i| \cdot n^{1-\epsilon}$ holds. Then the probability that Algorithm $\mathcal{A}'$ is successful is at least $\Omega\left(\frac{1}{n^{1-\epsilon}}\right)$.*

We execute Algorithm $\mathcal{A}'$ $\Theta(n^{1-\epsilon} \cdot \log n)$ times, and return the triple (c_x, x, y) with smallest value c_x that any execution of the algorithm produced. It is easy to verify that, if there is a global minimum vertex-cut (L,S,R) in G with $\lambda \leq |L_i| + |S_i| \leq 2\lambda$, such that $L_i \neq \emptyset$ and $|S_i| \leq |L_i| \cdot n^{1-\epsilon}$, then the probability that at least one of the executions of $\mathcal{A}'$ is successful is at least $1 - 1/n^4$, and in this case, if (c, x, y) is the output of the algorithm, then we are guaranteed that $c = \mathsf{OPT}$ holds. The running time of the entire algorithm is $O\left(m \cdot n^{1-\epsilon+o(1)} \cdot \log W'\right)$. □

2.4 Algorithm Alg_2: when G is not Dense and S is Large

Our second algorithm, Alg_2, will be used in the case where the input graph G is not very dense; we summarize it in the following theorem.

Theorem 2.7. *There is a randomized algorithm, that we denote by Alg_2, whose input is a simple undirected n-vertex and m-edge graph G with integral weights $1 \leq w(v) \leq W'$ on vertices $v \in V(G)$, and a parameter $0 < \epsilon < 1$, such that $m \leq n^{2-2\epsilon}$ holds. The algorithm returns an estimate c on the value OPT of the global minimum vertex-cut in G, and two vertices $x, y \in V(G)$, such that $c \geq \mathsf{OPT}$ holds, and the value of the minimum x-y vertex-cut in G is at most c. Moreover, if there is a global minimum vertex-cut (L,S,R) in G with $|S| \geq n^{1-\epsilon} \cdot |L|$, then, with probability at least $1 - 1/n^4$, $c = \mathsf{OPT}$ holds. The running time of the algorithm is $O\left(mn^{1-\epsilon+o(1)} \cdot \log W'\right)$.*

The remainder of this subsection is dedicated to the proof of Theorem 2.7.

Let T be the set of all vertices $v \in V(G)$ with $\deg_G(v) \geq n^{1-\epsilon}$. Since $\sum_{v \in V(G)} \deg_G(v) = 2m \leq 2n^{2-2\epsilon}$, we get that $|T| \leq \frac{2n^{2-2\epsilon}}{n^{1-\epsilon}} \leq 2n^{1-\epsilon}$. We use the following simple observation.

Observation 2.8. *Suppose there is a global minimum vertex-cut (L,S,R) in G with $|S| \geq n^{1-\epsilon} \cdot |L|$. Then $L \cap T \neq \emptyset$ must hold.*

Proof. Let (L, S, R) be a global minimum vertex-cut in G with $|S| \geq n^{1-\epsilon} \cdot |L|$. We claim that, for every vertex $v \in S$, there must be an edge connecting v to some vertex of L. Indeed, assume otherwise. Consider the partition (L, S', R') of vertices of G with $S' = S \setminus \{v\}$ and $R' = R \cup \{v\}$. It is easy to verify that (L, S', R') is a valid vertex-cut in G, and moreover that $w(S') < w(S)$, contradicting the assumption that (L, S, R) is a global minimum vertex-cut in G.

We conclude that the total number of edges in G connecting the vertices of L to the vertices of S is at least $|S|$, and so there must be some vertex $u \in L$ with $\deg_G(u) \geq \frac{|S|}{|L|} \geq n^{1-\epsilon}$, so $u \in T$ must hold. □

The following lemma provides a central tool for the proof of Theorem 2.7. The proof of the lemma is deferred to the full version of the paper.

Lemma 2.9. *There is a randomized algorithm, that, given a simple n-vertex and m-edge graph G with integral weights $1 \leq w(v) \leq W'$ on vertices $v \in V(G)$, and a vertex $x \in V(G)$, returns a vertex $y \in V(G)$, and the value c of the minimum x-y vertex-cut in G. Moreover, if there is a global minimum vertex-cut (L, S, R) in G with $x \in L$, then with probability at least $1 - 1/n^4$, $c = \mathsf{OPT}$ holds. The running time of the algorithm is $O\left(m^{1+o(1)} \cdot \log W'\right)$.*

We are now ready to complete the proof of Theorem 2.7. Our algorithm starts by computing the set T of vertices of G, whose degrees in G are at least $n^{1-\epsilon}$. Clearly, T can be computed in time $O(m)$, and, as observed already, $|T| \leq 2n^{1-\epsilon}$ must hold. Next, we apply the algorithm from Lemma 2.9 to every vertex $x \in T$ in turn, and we let (c_x, y_x) be the outcome of this application of the algorithm from the lemma. We then let $x' \in T$ be the vertex for which $c_{x'}$ is minimized, and return $(c_{x'}, x', y_{x'})$ as the outcome of the algorithm. Lemma 2.9 guarantees that $c_{x'}$ is the value of the minimum x'-$y_{x'}$ vertex cut in G, and so $c_{x'} \geq \mathsf{OPT}$ holds. Assume now that there is a global minimum vertex cut (L, S, R) in G with $|S| \geq n^{1-\epsilon} \cdot |L|$. Then, from Observation 2.8, $L \cap T \neq \emptyset$ must hold. Let v be any vertex in $L \cap T$. Then, when the algorithm from Lemma 2.9 was applied to vertex v, with probability at least $1 - 1/n^4$, the value c_v of the minimum v-y_v vertex cut that it returned is equal to OPT. In this case, we are guaranteed that the value c that our algorithm returns is equal to OPT as well. Since $|T| \leq n^{1-\epsilon}$, and since the running time of the algorithm from Lemma 2.9 is $O\left(m^{1+o(1)\cdot\log W'}\right)$, we get that the running time of our algorithm is $O\left(mn^{1-\epsilon+o(1)} \cdot \log W'\right)$.

2.5 Summary: an Algorithm for Non-Dense Graphs

By combining Algorithms Alg_1 and Alg_2, we obtain an algorithm for global minimum cut on graphs that are not very dense, that is summarized in the following corollary, whose proof we defer to the full version.

Corollary 2.10. *There is a randomized algorithm, whose input is a simple undirected n-vertex and m-edge graph G with integral weights $1 \leq w(v) \leq W'$ on vertices $v \in V(G)$, and a parameter $0 < \epsilon \leq 1/2$, such that $m \leq n^{2-2\epsilon}$ holds. The algorithm returns an estimate c on the value OPT of the global minimum vertex-cut in G, and two vertices $x, y \in V(G)$, such that $c \geq \mathsf{OPT}$ holds, and the value of the minimum x-y vertex-cut in G is at most c. Moreover, with probability at least $1 - 1/n^4$, $c = \mathsf{OPT}$ holds. The running time of the algorithm is $O\left(mn^{1-\epsilon+o(1)} \cdot \log W'\right)$.*

2.6 Main Technical Result: Algorithm for the Dense Case

Our final and the most involved algorithm, Alg_3, is only used in the case where the input graph G is dense; we summarize it in the following theorem. The theorem statement uses the parameter $W_{\max} = W_{\max}(G)$, defined in Section 2.1.1. The proof of the theorem is deferred to the full version of the paper.

Theorem 2.11. *There is a randomized algorithm, whose input is a simple undirected n-vertex graph G with integral weights $1 \leq w(v) \leq W'$ on vertices $v \in V(G)$. The algorithm returns an estimate c on the value OPT of the global minimum vertex-cut in G, and two vertices $x, y \in V(G)$, such that $c \geq \mathsf{OPT}$ holds, and the value of the minimum x-y vertex-cut in G is at most c. Moreover, if there is a global minimum vertex-cut (L, S, R) in G, such that, for all $0 \leq i \leq \lceil \log W' \rceil$, $|S_i| \geq n^{44/45} \cdot |L_i|$ holds, then, with probability at least $\frac{1}{2^{11} \cdot \log^3 n \cdot \log W'}$, $c = \mathsf{OPT}$ holds. The running time of the algorithm is $O\left(n^{2.96} \cdot (\log(W_{\max}))^{O(1)}\right)$.*

In order to obtain our final result – the proof of Theorem 1.1 – we use the algorithm from Corollary 2.10 if the input graph G is not very dense. When G is sufficiently dense, we use the combination of Algorithm Alg_1 from Theorem 2.4 and Alg_3 from Theorem 2.11. We provide further details in the full version of the paper.

References

[1] Anders Aamand, Justin Y. Chen, Mina Dalirrooyfard, Slobodan Mitrović, Yuriy Nevmyvaka, Sandeep Silwal, and Yinzhan Xu. 2024. Differentially Private Gomory-Hu Trees. *CoRR* (2024). http://arxiv.org/abs/2408.01798

[2] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. 2021. APMF < APSP? Gomory-Hu Tree for Unweighted Graphs in Almost-Quadratic Time. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*. 1135–1146. doi:10.1109/FOCS52979.2021.00112

[3] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. 2021. Subcubic algorithms for Gomory-Hu tree in unweighted graphs. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 1725–1737. doi:10.1145/3406325.3451073

[4] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. 2022. Friendly Cut Sparsifiers and Faster Gomory-Hu Trees. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*. 3630–3649. doi:10.1137/1.9781611977073.143

[5] Michael Becker, W. Degenhardt, Jürgen Doenhardt, Stefan Hertel, Gerd Kaninke, W. Kerber, Kurt Mehlhorn, Stefan Näher, Hans Rohnert, and Thomas Winter. 1982. A Probabilistic Algorithm for Vertex Connectivity of Graphs. *Inf. Process. Lett.* 15, 3 (1982), 135–136. doi:10.1016/0020-0190(82)90046-1

[6] Aaron Bernstein, Joakim Blikstad, Thatchaphol Saranurak, and Ta-Wei Tu. 2024. Maximum Flow by Augmenting Paths in $n^{2+o(1)}$ Time. *CoRR* (2024). https://arxiv.org/abs/2406.03648

[7] Andreas Björklund, Rasmus Pagh, Virginia Vassilevska Williams, and Uri Zwick. 2014. Listing triangles. In *International Colloquium on Automata, Languages, and Programming*. Springer, 223–234.

[8] Ruoxu Cen, Yu Cheng, Debmalya Panigrahi, and Kevin Sun. 2021. Sparsification of Directed Graphs via Cut Balance. In *48th International Colloquium on Automata,*

Languages, and Programming (ICALP 2021), Vol. 198. 45:1–45:21. doi:10.4230/LIPIcs.ICALP.2021.45

[9] Ruoxu Cen, Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Kent Quanrud. 2021. Minimum Cuts in Directed Graphs via Partial Sparsification. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021*. 1147–1158. doi:10.1109/FOCS52979.2021.00113

[10] Ruoxu Cen, Jason Li, and Debmalya Panigrahi. 2022. Augmenting Edge Connectivity via Isolating Cuts. In *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms, SODA 2022*, Joseph (Seffi) Naor and Niv Buchbinder (Eds.). 3237–3252. doi:10.1137/1.9781611977073.127

[11] Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. 2014. A New Perspective on Vertex Connectivity. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, Chandra Chekuri (Ed.). 546–561. doi:10.1137/1.9781611973402.41

[12] Shiri Chechik, Thomas Dueholm Hansen, Giuseppe F. Italiano, Veronika Loitzenbauer, and Nikos Parotsidis. 2017. Faster Algorithms for Computing Maximal 2-Connected Subgraphs in Sparse Directed Graphs. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*. 1900–1918. doi:10.1137/1.9781611974782.124

[13] Chandra Chekuri and Kent Quanrud. 2021. Faster Algorithms for Rooted Connectivity in Directed Graphs. In *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021*, Nikhil Bansal, Emanuela Merelli, and James Worrell (Eds.), Vol. 198. 49:1–49:16. doi:10.4230/LIPICS.ICALP.2021.49

[14] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2022. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 612–623. doi:10.1109/FOCS54457.2022.00064

[15] Joseph Cheriyan, Ming-Yang Kao, and Ramakrishna Thurimella. 1993. Scan-First Search and Sparse Certificates: An Improved Parallel Algorithms for k-Vertex Connectivity. *SIAM J. Comput.* 22, 1 (1993), 157–174. doi:10.1137/0222013

[16] Joseph Cheriyan and John H. Reif. 1994. Directed *s-t* Numberings, Rubber Bands, and Testing Digraph *k*-Vertex Connectivity. *Comb.* 14, 4 (1994), 435–451. doi:10.1007/BF01302965

[17] Shimon Even. 1975. An Algorithm for Determining Whether the Connectivity of a Graph is at Least k. *SIAM J. Comput.* 4, 3 (1975), 393–396. doi:10.1137/0204034

[18] Shimon Even and Robert Endre Tarjan. 1975. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.* 4, 4 (1975), 507–518. doi:10.1137/0204043

[19] Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2020. Computing and Testing Small Connectivity in Near-Linear Time and Queries via Fast Local Cut Algorithms. In *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, Shuchi Chawla (Ed.). 2046–2065. doi:10.1137/1.9781611975994.126

[20] Kyle Fox, Debmalya Panigrahi, and Fred Zhang. 2023. Minimum Cut and Minimum *k*-Cut in Hypergraphs via Branching Contractions. *ACM Trans. Algorithms* 19, 2 (2023), 13:1–13:22. doi:10.1145/3570162

[21] Harold N. Gabow. 1995. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.* 50, 2 (1995), 259–273.

[22] Harold N. Gabow. 2006. Using expander graphs to find vertex connectivity. *J. ACM* 53, 5 (2006), 800–844. doi:10.1145/1183907.1183912

[23] Andrew V. Goldberg and Robert Endre Tarjan. 1988. A new approach to the maximum-flow problem. *J. ACM* 35, 4 (1988), 921–940. doi:10.1145/48014.61051

[24] Jianxiu Hao and James B. Orlin. 1994. A Faster Algorithm for Finding the Minimum Cut in a Directed Graph. *J. Algorithms* 17, 3 (1994), 424–446. doi:10.1006/JAGM.1994.1043 Announced at SODA 1992..

[25] Zhongtian He, Shang-En Huang, and Thatchaphol Saranurak. 2024. Cactus Representations in Polylogarithmic Max-flow via Maximal Isolating Mincuts. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024, Alexandria, VA, USA, January 7-10, 2024*. 1465–1502. doi:10.1137/1.9781611977912.60

[26] Monika Henzinger, Jason Li, Satish Rao, and Di Wang. 2024. Deterministic Near-Linear Time Minimum Cut in Weighted Graphs. In *Proceedings of the 2024 ACM-SIAM Symposium on Discrete Algorithms, SODA 2024*. 3089–3139. doi:10.1137/1.9781611977912.111

[27] Monika Henzinger, Satish Rao, and Di Wang. 2017. Local Flow Partitioning for Faster Edge Connectivity. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017*, Philip N. Klein (Ed.). 1919–1938. doi:10.1137/1.9781611974782.125

[28] Monika Rauch Henzinger, Satish Rao, and Harold N. Gabow. 2000. Computing Vertex Connectivity: New Bounds from Old Techniques. *J. Algorithms* 34, 2 (2000), 222–250. doi:10.1006/JAGM.1999.1055 Announced at FOCS 1996..

[29] Arkady Kanevsky. 1990. On the Number of Minimum Size Separating Vertex Sets in a Graph and How to Find All of Them. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1990*, David S. Johnson (Ed.). 411–421. http://dl.acm.org/citation.cfm?id=320176.320227

[30] David R. Karger. 2000. Minimum cuts in near-linear time. *J. ACM* 47, 1 (2000), 46–76. doi:10.1145/331605.331608 Announced at STOC 1996..

[31] David R. Karger and Clifford Stein. 1993. An $\tilde{O}(n^2)$ algorithm for minimum cuts. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, STOC 1993*, S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal (Eds.). 757–765. doi:10.1145/167088.167281

[32] Ken-ichi Kawarabayashi and Mikkel Thorup. 2019. Deterministic Edge Connectivity in Near-Linear Time. *J. ACM* 66, 1 (2019), 4:1–4:50. doi:10.1145/3274663

[33] Daniel J. Kleitman. 1969. Methods for Investigating Connectivity of Large Graphs. *IEEE Transactions on Circuit Theory* 16, 2 (1969), 232–233. doi:10.1109/TCT.1969.1082941

[34] Jason Li. 2021. Deterministic mincut in almost-linear time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2021*, Samir Khuller and Virginia Vassilevska Williams (Eds.). 384–395. doi:10.1145/3406325.3451114

[35] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, and Thatchaphol Saranurak. 2023. Near-Linear Time Approximations for Cut Problems via Fair Cuts. In *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms, SODA 2023*. 240–275. doi:10.1137/1.9781611977554.CH10

[36] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2021. Vertex connectivity in poly-logarithmic max-flows. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 317–329. doi:10.1145/3406325.3451088

[37] Jason Li and Debmalya Panigrahi. 2020. Deterministic Min-cut in Poly-logarithmic Max-flows. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. 85–92. doi:10.1109/FOCS46700.2020.00017

[38] Jason Li and Debmalya Panigrahi. 2021. Approximate Gomory-Hu tree is faster than $n-1$ max-flows. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 1738–1748. doi:10.1145/3406325.3451112

[39] Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. 2021. A Nearly Optimal All-Pairs Min-Cuts Algorithm in Simple Graphs. In *62nd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2021, Denver, CO, USA, February 7-10, 2022*. 1124–1134. doi:10.1109/FOCS52979.2021.00111

[40] Nathan Linial, László Lovász, and Avi Wigderson. 1988. Rubber bands, convex embeddings and graph connectivity. *Combinatorica* 8, 1 (1988), 91–102. doi:10.1007/BF02122557

[41] Karl Menger. 1927. Zur allgemeinen Kurventheorie. *Fundamenta Mathematicae* 10, 1 (1927), 96–115. http://eudml.org/doc/211191

[42] Sagnik Mukhopadhyay and Danupon Nanongkai. 2021. A Note on Isolating Cut Lemma for Submodular Function Minimization. *arXiv preprint arXiv:2103.15724* (2021).

[43] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph. *Algorithmica* 7, 5&6 (1992), 583–596. doi:10.1007/BF01758778

[44] Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2019. Breaking quadratic time for small vertex connectivity and an approximation scheme. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*. 241–252. doi:10.1145/3313276.3316394

[45] C. St.J. A. Nash-Williams. 1961. Edge-Disjoint Spanning Trees of Finite Graphs. *Journal of the London Mathematical Society* s1-36, 1 (01 1961), 445–450. doi:10.1112/jlms/s1-36.1.445

[46] V. D. Podderyugin. 1973. An algorithm for finding the edge connectity of graphs. 136 pages. https://cir.nii.ac.jp/crid/1371694371159352077

[47] Thatchaphol Saranurak and Sorrachai Yingchareonthawornchai. 2022. Deterministic Small Vertex Connectivity in Almost Linear Time. In *63rd IEEE Annual Symposium on Foundations of Computer Science, FOCS 2022*. 789–800. doi:10.1109/FOCS54457.2022.00080

[48] W. T. Tutte. 1961. On the Problem of Decomposing a Graph into n Connected Factors. *Journal of the London Mathematical Society* s1-36, 1 (01 1961), 221–230. doi:10.1112/jlms/s1-36.1.221

[49] Jan van den Brand, Li Chen, Richard Peng, Rasmus Kyng, Yang P. Liu, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. 2023. A Deterministic Almost-Linear Time Algorithm for Minimum-Cost Flow. In *64th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2023*. 503–514. doi:10.1109/FOCS57990.2023.00037

[50] Tianyi Zhang. 2022. Faster Cut-Equivalent Trees in Simple Graphs. In *49th International Colloquium on Automata, Languages, and Programming, ICALP 2022 (LIPIcs, Vol. 229)*, Mikolaj Bojanczyk, Emanuela Merelli, and David P. Woodruff (Eds.). 109:1–109:18. doi:10.4230/LIPICS.ICALP.2022.109