Subcubic Algorithms for Gomory–Hu Tree in Unweighted Graphs*

Amir Abboud†
Weizmann Institute of Science, Israel
amir.abboud@weizmann.ac.il

Robert Krauthgamer‡
Weizmann Institute of Science, Israel
robert.krauthgamer@weizmann.ac.il

Ohad Trabelsi
Weizmann Institute of Science, Israel
ohad.trabelsi@weizmann.ac.il

ABSTRACT

Every undirected graph G has a (weighted) *cut-equivalent tree* T, commonly named after Gomory and Hu who discovered it in 1961. Both T and G have the same node set, and for every node pair s, t, the minimum (s, t)-cut in T is also an exact minimum (s, t)-cut in G.

We give the first subcubic-time algorithm that constructs such a tree for a simple graph G (unweighted with no parallel edges). Its time complexity is $\tilde{O}(n^{2.5})$, for $n = |V(G)|$; previously, only $\tilde{O}(n^3)$ was known, except for restricted cases like sparse graphs. Consequently, we obtain the first algorithm for All-Pairs Max-Flow in simple graphs that breaks the cubic-time barrier.

Gomory and Hu compute this tree using $n - 1$ queries to (single-pair) Max-Flow; the new algorithm can be viewed as a fine-grained reduction to $\tilde{O}(\sqrt{n})$ Max-Flow computations on n-node graphs.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**; **Graph algorithms analysis**; **Network flows**.

KEYWORDS

Gomory–Hu tree, graph cuts, maximum flow, expander decomposition, fine-grained complexity

ACM Reference Format:
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 (STOC '21), June 21–25, 2021, Virtual, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3406325.3451073

1 INTRODUCTION

A fundamental discovery of Gomory and Hu in 1961, now a staple of textbooks on Algorithms, says that every undirected graph can be compressed into a single tree while exactly preserving all pairwise minimum cuts (and maximum flows).

Theorem 1.1 (Gomory and Hu [38])**.** *Every undirected graph G (even with edge weights) has an edge-weighted tree T on the same set of vertices $V(G)$ such that:*

- *for all pairs $s, t \in V(G)$ the minimum (s, t)-cut in T is also a minimum (s, t)-cut in G, and their values are the same.*

Such a tree is called a cut-equivalent tree*, aka* Gomory–Hu tree*. Moreover, the tree can be constructed in the time of $n - 1$ queries to a Max-Flow algorithm,[1] where throughout $n = |V(G)|$.*

In the interim sixty years, the Gomory–Hu tree has been investigated thoroughly from various angles (see Section 1.4). In spite of ingenious new approaches and much progress in simpler or harder settings, the chief open question remains elusive.

Open Question 1.2. *Can one construct a Gomory–Hu tree faster than the time of $O(n)$ Max-Flow computations?*

In the most basic setting of *simple graphs* (undirected, unweighted, no parallel edges), each Max-Flow query can be answered in $\tilde{O}(n^2)$ time[2] using the algorithm of Karger and Levine [53], and the question becomes whether the natural cubic barrier can be broken.

Open Question 1.3. *Can one construct a Gomory–Hu tree of a simple graph G in $o(n^3)$ time?*

Before the current work, subcubic[3] algorithms were known only for sparse graphs [4, 15, 53], planar graphs [17], surface-embedded graphs [16], and bounded-treewidth graphs [3, 10].

1.1 Results

We resolve Open Question 1.3 in the affirmative by giving the first subcubic algorithm for computing a Gomory–Hu tree for unweighted graphs.

Theorem 1.4. *There is a randomized algorithm, with success probability $1 - 1/\mathrm{poly}(n)$, that constructs a Gomory–Hu tree of a simple graph G in $\tilde{O}(n^{2.5})$ time.*

Like the Gomory–Hu algorithm (and others), our new algorithm relies on queries to Max-Flow on contracted graphs. While the number of queries is still $\Omega(n)$, the gains come from reducing the

*Full version available at arXiv:2012.10281.
†Work partially done at the IBM Almaden Research Center, US.
‡Work partially supported by ONR Award N00014-18-1-2364, the Israel Science Foundation grant #1086/18, and a Minerva Foundation grant.

STOC '21, June 21–25, 2021, Virtual, Italy

ACM ISBN 978-1-4503-8053-9/21/06…$15.00
https://doi.org/10.1145/3406325.3451073

[1] The notation Max-Flow refers to the maximum (s, t)-flow problem, which clearly has the same value as minimum (s, t)-cut. In fact, we often need algorithms that find an optimal cut (not only its value), which is clearly different (and usually harder) than Global-Min-Cut.
[2] The notation $\tilde{O}(\cdot)$ hides a factor that is polylogarithmic in n.
[3] An algorithm is called subcubic if its running time is $o(n^3)$ where n is the number of nodes. It is common to distinguish between mildly-subcubic $O(\frac{n^3}{\mathrm{poly}\log n})$ and truly-subcubic $O(n^{3-\varepsilon})$. In this paper we refer to the latter.

sizes of the contracted graphs more massively and rapidly, making the total time proportional to only $\tilde{O}(\sqrt{n})$ queries. The key ingredient is a new Expanders-Guided Querying procedure that leads the algorithm to make Max-Flow queries on "the right" pairs, guaranteeing that large portions of the graph get contracted. In fact, since these $\Omega(n)$ queries can be done in parallel inside each of $O(\log n)$ levels of recursion, they can be simulated using only $\tilde{O}(\sqrt{n})$ queries on graphs with n nodes (by simply taking disjoint unions of the smaller graphs and connecting all sources/sinks into one source/sink, see e.g. [61]). However, due to the contractions, these queries need to be performed on weighted graphs.

All-Pairs Max-Flow. Once a Gomory–Hu tree is computed many problems become easy. Notably, the first algorithm for computing the Edge-Connectivity of a graph, i.e. Global-Min-Cut, computes the Gomory–Hu tree and returns the smallest edge [38]. Faster near-linear time algorithms are known by now [32, 33, 35, 48, 52, 54, 55, 65], but state-of-the-art algorithms for computing the maximum-flow value between all $\binom{n}{2}$ pairs of nodes, called the All-Pairs Max-Flow problem,[4] still rely on the reduction to a Gomory–Hu tree. Indeed, to compute the maximum flow, simply identify the lightest edge on the path between each pair in the tree, which takes only $\tilde{O}(1)$ time per pair; the bottleneck is computing the tree.

The modern tools of fine-grained complexity have been surprisingly unhelpful in establishing a conditional lower bound for All-Pairs Max-Flow. The Strong ETH[5] rules out fast algorithms in *directed* graphs (where the Gomory–Hu tree does not exist [47]) [2, 6, 58], but it probably cannot give an $n^{2+\varepsilon}$ lower bound for simple graphs [4].[6] Moreover, all natural attempts for placing it in the All-Pairs Shortest-Path subcubic-hardness class [83] had failed, despite the fact that Max-Flow feels harder than Shortest-Path, e.g., single-pair in $\tilde{O}(n^2)$ time is an easy coding-interview question for Shortest-Path but certainly not for Max-Flow. It turns out that subcubic time is indeed possible, at least in unweighted graphs.

Corollary 1.5. *There is a randomized algorithm, with success probability $1-1/\text{poly}(n)$, that solves All-Pairs Max-Flow in simple graphs in $\tilde{O}(n^{2.5})$ time.*

For both All-Pairs Max-Flow and All-Pairs Shortest-Path, cubic time is a natural barrier — shouldn't each answer take $\Omega(n)$ time on average? About thirty years ago, Seidel's algorithm [79] broke this barrier for All-Pairs Shortest-Path in unweighted graphs using fast matrix multiplication with an $O(n^\omega)$ upper bound; back then ω was 2.375477 [27] and today it is 2.37286 [8]. Our new $\tilde{O}(n^{2.5})$ algorithm for All-Pairs Max-Flow breaks the barrier using a different set of techniques, mainly expander decompositions, randomized hitting sets, and fast Max-Flow algorithms (that in turn rely on other methods such as continuous optimization). Interestingly, while designing an $O(n^{3-\varepsilon})$ algorithm with "combinatorial" methods has been elusive for All-Pairs Shortest-Path (see e.g. [1, 5, 11, 21, 22, 83, 85]), it can already be accomplished for All-Pairs Max-Flow. Our upper bound when using only combinatorial methods is $\tilde{O}(n^{2\frac{5}{6}})$.

More Bounds. Since the first version of this paper came out, a new algorithm has been published for solving a single Max-Flow query in weighted graphs in time $\tilde{O}(m+n^{1.5})$ [82], improving over the previous bound $\tilde{O}(m\sqrt{n})$ [60], where throughout $m = |E(G)|$. This development directly has led to the improvement of our bound for Gomory–Hu tree from $\tilde{O}(n^{2.75})$ in the previous version to $\tilde{O}(n^{2.5})$ in the current version. In fact, our previous version had already included an $\tilde{O}(n^{2.5})$-time algorithm assuming a hypothetical $\tilde{O}(m)$-time Max-Flow algorithm; it turns out that the additive term $+n^{1.5}$ in the new algorithm [82] is inconsequential for our result.

The running time of our new algorithm improves beyond $n^{2.5}$ if the number of edges is below n^2. The precise bound is $\tilde{O}(n^{3/2}m^{1/2})$. However, for density below $n^{1.5}$, a previously known (and very different) algorithm [4] is faster; its time bound is $\tilde{O}(m^{3/2})$ under the hypothesis that Max-Flow is solved in near-linear time, and using the new algorithm [82] instead gives a slightly worse bound. The previous bounds and the state of the art for all density regimes are summarized in Figure 1. It is also summarized in the following theorem together with the best combinatorial algorithms. In each item, the rightmost term is new, and the other terms (which are better for sparse graphs) are by previous work [4] (when plugging in [82]).

Theorem 1.6. *There is a randomized algorithm, with success probability $1-1/\text{poly}(n)$, that constructs a Gomory–Hu tree of a simple graph with n nodes and m edges in time:*

(1) $\tilde{O}\left(\min\{m^{3/2}n^{1/6}, mn^{3/4}, n^{3/2}m^{1/2}\}\right)$ *using existing Max-Flow algorithms,*

(2) $\tilde{O}\left(\min\{m^{3/2}n^{1/3}, n^{11/6}m^{1/2}\}\right)$ *using existing combinatorial Max-Flow algorithms.*

1.2 Previous Algorithms

Over the years, the time complexity of constructing a Gomory–Hu tree has decreased several times due to improvements in Max-Flow algorithms, but there have also been a few conceptually new algorithms. Gusfield [40] presented a modification of the Gomory–Hu algorithm in which all the $n-1$ queries are made on the original graph G (instead of on contracted graphs). Bhalgat, Hariharan, Kavitha, and Panigrahi [15] designed an $\tilde{O}(mn)$-time algorithm utilizing a tree packing approach [28, 32] that has also been used in other algorithms for cut-equivalent trees [4, 26, 41]. In particular, they designed an $O(mk)$-time algorithm for constructing a k-partial Gomory–Hu tree, which preserves the minimum cuts if their size is up to k (see [73] and the full version [14]). A simple high-degree/low-degree strategy is helpful in sparse graphs: only $\sqrt{m}$ nodes can have degree (and therefore outgoing flow) above $\sqrt{m}$, thus a $\sqrt{m}$-partial tree plus $\sqrt{m}$ Max-Flow queries are sufficient, which takes $O(m^{3/2})$ time if Max-Flow is solved in linear time. Using the current Max-Flow algorithms, this strategy results in the time bound $\tilde{O}(\min\{m^{3/2}n^{1/6}, \max\{mn^{3/4}, m^{3/2}\}\})$ [4]. Additionally, two recent algorithms accelerate the Gomory–Hu algorithm

[4] Since we are in the setting of simple graphs, a possible name for the problem is All-Pairs Edge Connectivity. However, Edge-Connectivity is often used for the size of the *global minimum cut* in simple graphs, not the st-version.

[5] The Strong Exponential Time Hypothesis (SETH) of Impagliazzo and Paturi [19, 50] states that k-SAT cannot be solved in $(2-\varepsilon)^n$ time for an $\varepsilon > 0$ independent of k. It has been a popular hardness assumption for proving polynomial lower bounds like $n^{3-o(1)}$ in recent years.

[6] Reducing k-SAT to All-Pairs Max-Flow would give a faster co-nondeterministic algorithm for k-SAT and refute the so-called Nondeterministic SETH [20].

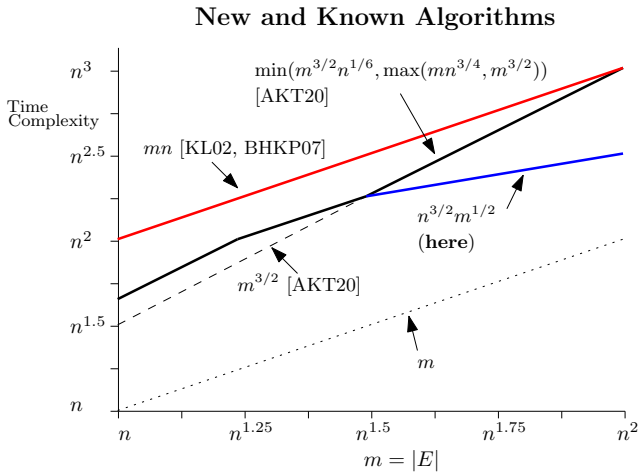

Figure 1: State-of-the-art time bounds for constructing a Gomory–Hu tree of simple graphs. The dotted line is the best one can hope for (as no super-linear lower bound is known). The dashed line represents the $m^{3/2+o(1)}$-time algorithm [4] that assumes a near-linear time Max-Flow algorithm.

by making multiple steps at once to achieve $\tilde{O}(m)$ time, one requires nondeterminism [4] and the other requires a (currently nonexistent) fast minimum-cut data structure [3]. The new algorithm uses a similar approach.

In weighted graphs, the bound for Gomory–Hu tree is still n times Max-Flow and therefore cubic using the very recent $\tilde{O}(m + n^{1.5})$ algorithm for Max-Flow [82]. If the graph is sparse enough and the largest weight U is small, one can use algorithms [62–64] that run in time $\tilde{O}(\min\{m^{10/7}U^{1/7}, m^{11/8}U^{1/4}, m^{4/3}U^{1/3}\})$ that might give a better time bound. It is likely that the new techniques will lead to faster algorithms for this case as well, although new ideas are required.

Further discussion of previous algorithms can be found at the Encyclopedia of Algorithms [73].

1.3 Technical Overview

This section attempts to explain the new algorithm in its entirety in a coherent and concise way. This is challenging because the algorithm is complicated both conceptually, relying on several different tools, and technically, involving further complications in each ingredient. The main text gives the overall algorithm while making nonobvious simplifying assumptions, and meanwhile the text inside the boxes discusses how these assumptions are lifted or how to handle other technical issues that arise. The reader is advised to skip the boxes in the first read.

Given a graph $G = (V, E)$ the goal is to compute its cut-equivalent tree T. The new algorithm, like the Gomory–Hu algorithm builds T recursively, by maintaining an intermediate (partial) tree T' that gets *refined* with each recursive call. The nodes of T' are subsets of V and are called super-nodes. Each recursion step considers a super-node of T' and splits-off parts of it. In the beginning, there is a single super-node and $T' = \{V\}$. If the minimum (s, t)-cut $(S, V \setminus S)$ for a pair $s, t \in V$ is found, then T' is refined by splitting V into two super-nodes S and $V \setminus S$ and by connecting them with an edge of weight $\lambda_{s,t}$ the connectivity between s and t. Then, the main observation is that when refining S the entire super-node $V \setminus S$ can be contracted into a single node; this does not distort the connectivities for pairs in S and it ensures that the refinements of S and of $V \setminus S$ are consistent with one another (and can therefore be combined into the final tree T). More generally, consider an intermediate tree T' whose super-nodes $V_1, \ldots, V_l$ form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. The algorithm then refines each super-node V_i by operating on an *auxiliary graph* G_i that is obtained from G by contracting nodes outside V_i in a manner informed by T': each connected component of T' after the removal of V_i is contracted into a single node.

Now, consider such super-node V_i in an intermediate tree T' along with its auxiliary graph G_i,[7] and let T_i be its (unknown) cut-equivalent tree, i.e., the subtree of T induced on V_i. The goal is to refine V_i. Gomory and Hu took an arbitrary pair $s, t \in V_i$ and used an s, t-Max-Flow query to get a cut. (For a more detailed exposition of the Gomory–Hu algorithm, see Section 2.3 and Figure 3). Another natural approach is to take the global minimum cut of V_i. Either way, the depth of the recursion could be $n - 1$ if each refinement only splits-off one node from V_i.[8] The new algorithm aims to finish within recursion depth $O(\log n)$, and the strategy is to refine V_i into multiple disjoint minimum cuts $V = V_{i,1} \sqcup \cdots \sqcup V_{i,k}$ at once, where $|V_{i,j}| \leq |V_i|/2$ ensuring that the depth of the recursion is logarithmic.

The idea is to pick a pivot node $p \in V_i$ (can be thought of as a root for T_i) and to compute Max-Flow between p and every node in V_i. That is, for each node $v \in V_i$, the algorithm computes a minimum (p, v)-cut $(C_v, V_i \setminus C_v)$ where $v \in C_v$ and $p \in V \setminus C_v$. While this gives a lot of information, it is not necessarily sufficient for computing T_i because these cuts may not determine an optimal cut for other pairs in V_i.[9] Still, it allows us to make progress by splitting-off cuts $V_{i,j}$ from V_i with $|V_{i,j}| \leq |V_i|/2$ such that $V_{i,j} = C_v$ for some $v \in V_i$ and can therefore be safely used to refine V_i. This approach indeed works if p happens to be a *good* pivot, which means that it is centroid-like in T_i in the sense that most cuts C_v have less than half the nodes. Moreover, for the correctness of the algorithm, a good pivot must satisfy that the "depth" of T_i from p is $O(\sqrt{n})$. It turns out that both requirements can be accomplished.

[7] For most of this overview, it is safe to think of the very first iteration where $V_i = V$ and $G_i = G$. Later on we will point out the complications that arise due to the existence of contracted nodes $V(G_i) \setminus V_i$.

[8] Another natural approach is to look for a *balanced* minimum cut. It is not clear that such a cut exists or that it can be computed efficiently. However, it is fair to say that part of the new algorithm uses a similar approach.

[9] An extreme scenario is if for every $v \in V_i$ the minimum (p, v)-cut is $(\{p\}, V_i \setminus \{p\})$; clearly, this information is not sufficient for knowing the whole tree. But this will not happen if we choose a good pivot p.

Complication 1: How do we get a good pivot?

A recent nondeterministic algorithm [4] chooses a pivot by guessing the centroid (there are other, more crucial uses for guessing in that algorithm). An NC algorithm of Anari and Vazirani [9] for planar graphs tries all nodes as pivots in parallel. Bhalgat et al. [15] and another recent work [3] choose a pivot at random in each iteration and argue that the recursion depth is bounded by $O(\log n)$ with high probability. The new algorithm takes a similar approach but requires more care, because now a bad pivot not only slows down progress but it could lead to wrong cuts. For the correctness of what follows, a good pivot must also satisfy that a certain notion of depth of T_i when rooted at p is bounded by $O(\sqrt{n})$. And it must hold, with high probability, for every single iteration. Thus, to decrease the failure probability from constant to $1/\text{poly}(n)$, the algorithm chooses a random set S of $\tilde{O}(\sqrt{n})$ candidate pivots, computes a refinement of T' by following the standard Gomory–Hu algorithm while only picking pairs from S using $|S|-1$ queries to Max-Flow, and then picks the pivot p to be the node in S whose component in the refinement is largest. It follows that p is both centroid-like ($|C_v| \leq |V_i|/2$ for most $v \in V_i$) and its component (after this refinement) has a small depth. Notably, derandomizing this part would lead to a deterministic algorithm with $n^{1-\varepsilon}$ queries to Max-Flow.

At this point, the algorithm has a good pivot p for G_i and the goal is to compute the minimum (p,v)-cut for all nodes $v \in V_i$. To focus this exposition on the novel aspects of the new algorithm, let us make a few simplifying assumptions. First, assume that there is a unique Gomory–Hu tree T for G (and therefore there is a unique T_i for G_i).

Complication 2: Operating as if there is a unique tree.

A standard way to make the Gomory–Hu tree unique is to perturb edge weights in G by small $1/\text{poly}(m)$ additive terms to ensure that all minimum cuts are unique. However, this would prevent the algorithm from using known Max-Flow algorithm for unweighted graphs and hurt the running time, so it cannot be done in our real algorithm. The uniqueness of the tree is important both for the analysis, as one can safely talk about *the* minimum cut C_v, and for the correctness – what guarantees that cuts C_v for different $v \in V_i$ are all consistent with a single Gomory–Hu tree? Our real algorithm escapes these issues by working with *latest* minimum cuts, i.e., a minimum (p,v)-cut $(C_v, V \setminus C_v)$ that has the smallest possible $|C_v|$. A similar approach was used by Bhalgat et al. [15] and others [13]. See Section 2.5 for more background on latest cuts.

The second simplifying assumption is that the weights in T_i are decreasing along any path from the root p. As a result, the minimum (p,v)-cut for every node $v \in V_i$ is exactly its subtree; this greatly simplifies describing and analyzing the cuts with respect to the tree (see Figure 2).

Complication 3: Cut-membership trees.

In general, the lightest edge on the path from a node $v \in V_i$ to p in T_i is not necessarily the first edge (right "above" v). In other words, the minimum (p,v)-cut C_v, whose nodes we call *cut-members* of v, could be a strict superset of v's subtree in T_i, which complicates the analysis of subsequent ingredients, as they rely on bounding the number of cut-members of nodes. A convenient tool for reasoning about this is the cut-membership tree $\mathcal{T}_p^*$ with respect to p [3, Section 3]. This is a coarsening of T_i where all nodes $B \subseteq V_i$ whose minimum cut to p is the same are merged into one bag . Importantly, it is still a tree and it satisfies the assumption of decreasing weights, meaning that the minimum cuts are indeed always subtrees. All of the analysis is carried on with $\mathcal{T}_p^*$ rather than T_i.

The algorithm now creates for each node $v \in V_i$ an estimate $c'(v)$ for the connectivity $\lambda_{p,v}$ between p and v; it is initialized to $c'(v) = \deg(v)$, which is always an upper bound. Then, the algorithm repeats procedure Expanders-Guided Querying described below $O(\log n)$ times; each iteration finds new (p,v)-cuts and updates these estimates by keeping for each v the minimum value seen so far. Since each cut is an upper bound on the connectivity, the estimates $c'(v)$ never decrease below $\lambda_{p,v}$, and with high probability they are eventually tight for all nodes. A node v is called *done* if $c'(v) = \lambda_{p,v}$ and *undone* otherwise.

In the beginning a node is done if and only if it is a leaf in T_i. (Imagine a tree with green leaves and red internal nodes.) Each iteration of the Expanders-Guided Querying procedure is makes a node done if its subtree contains at most $\sqrt{n}$ undone nodes, with high probability. (At each step, a red node becomes green if its subtree contains at most $\sqrt{n}$ red nodes.) How many iterations until all nodes are done (green)? Using the fact that p is a good pivot and the depth of T_i is $O(\sqrt{n})$, it can be shown that with high probability $O(\log n)$ iterations suffice.

Procedure Expanders-Guided Querying. For each value $w = 2^{i'}$ for $i' = 0, 1, \ldots, \log n$ there is a sub-procedure that aims to find C_v for all nodes $v \in V_i$ such that $w \leq \lambda_{p,v} < 2w$. Fix such a node v that is undone; then $\deg(v) \geq c'(v) > \lambda_{p,v} \geq w$. Thus, only nodes of degree $> w$ are targeted by the sub-procedure so we call them *relevant*.

In a preprocessing phase, for each $w = 2^{i'}$, the algorithm prepares an expander decomposition $V = H_1^w \sqcup \cdots \sqcup H_h^w$ of the entire graph G with expansion parameter $\phi_w = 1/\sqrt{w}$. Each component H_j^w is a ϕ_w-expander, meaning that there is no sparse cut inside it, and the total number of edges outside the expanders is bounded $\frac{1}{2}\sum_j \delta(H_j^w) = O(|E(G)| \cdot \phi_w \cdot \log^3 n)$. Efficient and simple algorithms for computing this decomposition exist, e.g. [78].

Complication 4: Lower bounding w.

The expander decomposition algorithm requires the parameter ϕ to be $O(1/\log n)$, and furthermore it must be $O(1/\log^3 n)$ for the outside edges bound to be meaningful.
When $\phi_w = 1/\sqrt{w}$ this is only an issue for very small w (and can be resolved by decreasing ϕ_w by a $\log^3 n$ factor). Our combinatorial $\tilde{O}(n^{2\frac{5}{6}})$ time algorithm, however, uses $\phi_w = n^{1/3}/w$, thus w must be at least $\Omega(n^{1/3}\log^3 n)$. For this reason, the very first "refinement" step of the algorithm is to compute a k-partial tree with $k = O(n^{1/3}\log^3 n)$ using the $O(nk^2)$-time algorithm of Bhalgat et al. [15] (based on tree packings). This gives a partial tree T' in which two nodes u, v are in separate super-nodes if and only if their connectivity is at most k. Afterwards, values $w < n^{1/3}\log^3 n$ can be ignored as pairs in the same super-node $u, v \in V_i$ are guaranteed to have $\lambda_{u,v} > k$.

Let H_v be the expander containing v; it could be of one of three kinds, each kind is solved by a different method (even though neither v nor H_v is known to the algorithm). Let $L = C_v \cap H_v$ be the piece of H_v that falls inside v's subtree C_v and let $R = (V \setminus C_v) \cap H_v$ be the remainder of H_v; we call them the left and right parts of H, respectively, see Figure 2.

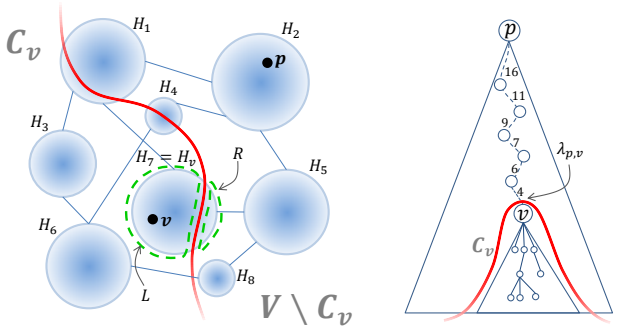

Figure 2: An expander decomposition of the graph G (on left), and the (unknown) cut-equivalent tree T rooted at pivot p (on right). The unique minimum (p, v)-cut $(C_v, V \setminus C_v)$ for an arbitrary node v is depicted by a red curve in both figures. The algorithm must find C_v. The edges in T are assumed to be decreasing, thus C_v is exactly v's subtree with an edge of weight $\lambda_{p,v}$ above it. The expander decomposition of G is $\{H_i\}_{i=1}^{8}$, and the expander containing v is $H_v = H_7$. The sets L and R are depicted by dashed green lines; at least one of them must be small. The three kinds of expanders can be seen: small (H_4, H_8), large lefty (H_7, H_3, H_6), and large righty (H_1, H_2, H_5). Since H_v is a large lefty expander, the cut C_v should be found when handling Case 3.

(1) Small: H_v contains $|H_v| < w/8$ nodes.
(2) Large righty: $|H_v| \geq w/8$ but contains only $|L| \leq 2\sqrt{w}$ relevant nodes from C_v.
(3) Large lefty: $|H_v| \geq w/8$ but contains only $|R| \leq 2\sqrt{w}$ relevant nodes from $V \setminus C_v$.

A key observation is that H_v cannot have both $|L|, |R| > 2\sqrt{w}$ due to the fact that H_v is a $1/\sqrt{w}$-expander: the cut (L, R) in H_v has at most $\lambda_{p,v} < 2w$ edges and thus its volume $\min(\text{vol}(L), \text{vol}(R)) \geq \min(|L|w, |R|w)$ cannot be more than $2w\sqrt{w}$. Therefore, a large expander must either be lefty or righty; it cannot be balanced. This analysis assumes that all nodes in L and R are relevant, but in general they could contain many low-degree nodes. Handling this gap is perhaps the most interesting complication of this paper, distinguishing it from recent applications of expander-based methods for Global-Min-Cut where low-degree nodes are not an issue: if the global minimum cut has value w then all nodes must have degree $\geq w$.

Complication 5: Low-degree nodes in expanders.

Our use of expander-decompositions is nonstandard but is reminiscent of the way they are used in two recent algorithms for Global-Min-Cut (which is equivalent to finding the smallest edge in the Gomory–Hu Tree). The Kawarabayashi-Thorup [55] technique (as described by Saranurak [77]) takes each expander, *trims* and *shaves* it to make sure it is exclusively on one side of the cut, and then contracts it. It is shown that most edges remain inside the expanders (and are therefore contracted away) despite the trimmings. The algorithm of Li and Panigrahy [61] maintains a set of candidates U containing nodes on each side of the cut and then uses the expanders to iteratively reduce its size by half. Some nodes of each expander are kept in U, and it is argued that there must be expanders that are mostly on the left and expanders that are mostly on the right.
Unfortunately, neither of these approaches seems to work when searching for all minimum (p, v)-cuts rather than for a single global minimum cut. At their bottom, both rely on the following observation: if L, the left side of H, contains $\geq \ell$ nodes that are committed to H, i.e., most of their edges stay inside H, then the volume of L is $\Omega(\ell w)$. This is because when the global minimum cut has value w, all nodes must have degree (or capacitated degree in the weighted case) at least w. However in our setting, the minimum (p, v)-cut C_v does not have to be minimal for any $u \in C_v$ except v, and nodes on the left could have arbitrarily small degrees (see Section 3.3 for an extreme example).
Consequently, the arguments in this paper are a little different and only lead to $O(\sqrt{w})$ savings rather than $\Omega(w)$. If there was a magical way to get rid of nodes of degree $< w$, then a near-linear time algorithm could follow. The k-partial tree (from Complication 4) indeed only leaves high-degree nodes in V_i but the contracted nodes $V(G_i) \setminus V_i$ could have arbitrary degrees. Instead, the algorithm and the analysis reason about the subsets $\hat{H}, \hat{L}, \hat{R}$ of H, L, R containing high degree nodes, as well as H, L, R themselves. With some care, the arguments presented in this section can indeed be made to work despite the gap.

To handle **Case 1** the algorithm simply asks a Max-Flow query between p and each *relevant* node u in each small expander H, i.e., $|H| < w/8$. Clearly, if v happens to be in a small expander it gets queried and its optimal cut is found with probability 1. The slightly trickier part is arguing that only $t = O(\sqrt{n})$ queries are performed: any queried node u has degree $\geq w$ (since it is relevant)

while a small expander only has $< w/8$ nodes, thus overall $\Omega(tw)$ edges are outside the expanders. But the expander decomposition guarantees that only $O(|E|/\sqrt{w})$ edges are outside the expanders, and thus $t = O(|E|/w^{1.5})$; in the hardest case where $w = \Omega(n)$ this is $t = O(\sqrt{n})$. The fact that the auxiliary graph G_i is a multigraph does not matter, because the expander decomposition is for G. This is the only case in the algorithm that requires G to be simple;[10] the argument is similar to [55, Observation 5].

Complication 6: Saving when $w = o(n)$ using Nagamochi–Ibaraki sparsification.
In the hardest case where $w = \Omega(n)$ the number of queries $t = O(|E|/w^{1.5})$ in Case 1 is already $O(\sqrt{n})$, but to get the $n^{2.5+o(1)}$ bound in general, the algorithm utilizes two sparsifications. First, it computes the expander-decomposition for each $w = 2^i$ on a sparsifier G_w with $|E_w| = O(nw)$ rather than on G. The sparsifier of Nagamochi and Ibaraki [67] with parameter k has $O(nk)$ edges and ensures that all cuts of value up to k are preserved. This gives a better upper bound on the number of edges outside expanders and leads to $t = O(|E_w|/w^{1.5}) = O(n/\sqrt{w})$. Second, while the number of queries exceeds $\sqrt{n}$ when $w = o(n)$, the algorithm saves by computing each of these queries in $\tilde{O}(nw)$ time rather than $O(m)$ by operating on a sparsifier of the same kind, but now for the auxiliary graph G_i, not G. This does not introduce error as the algorithm is only interested in cuts of value $\leq 2w$ and the total time becomes $\tilde{O}(n^2\sqrt{w}) = \tilde{O}(n^{2.5})$.

Handling **Case 2** involves a surprising Isolating-Cuts procedure that can *almost* compute single-source all-sinks Max-Flow with $O(\log n)$ queries to Max-Flow. It is a tricky but lightweight reduction that only involves contracting subsets of nodes.

Lemma 1.7 ([61], see also Lemma 3.4)**.** [11] *Given a pivot p and a set of terminals C, procedure Isolating-Cuts uses $O(\log n)$ Max-Flow queries and returns an estimate (p, v)-cut S_v for each terminal $v \in C$ such that: for all $v \in C$, if the minimum (p, v)-cut C_v satisfies $C_v \cap C = \emptyset$ (i.e., this cut isolates terminal v) then $S_v = C_v$.*

This procedure is quite strong in the sense that it solves hard cases with $\tilde{O}(1)$ Max-Flow queries when other approaches require $\Omega(n)$ queries, for instance in cases where all subtrees are small (see Section 3). But it is also weak in the sense that there is no way to distinguish the correct answers from fake ones. Nodes with large subtrees may never be isolated yet the procedure returns a (fake) small cut that, if used by the algorithm, could lead to a wrong tree. The canonical such example is discussed in Section 3 as it motivates the use of an expanders-based approach.

The Isolating-Cuts procedure is handy in **Case 2** where the algorithm takes each expander H with $|H| \geq w/8$ (there are only $O(n/w) = O(1)$ such expanders) and guesses that $H = H_v$ and that it is a large righty expander, i.e., $|L| \leq 2\sqrt{w}$. In this case, if indeed $v \in H$, it is possible to pick $t = \tilde{O}(\sqrt{w})$ sets of terminals $C_1, \ldots, C_t$ such that at least one of them isolates v: for each $i' \in [t]$ include each node $u \in H$ into $C_{i'}$ with probability $1/\sqrt{w}$. For a set $C_{i'}$ to isolate v, it must (1) pick v and (2) not pick any other node from L, which happens with probability $\Omega(1/\sqrt{w})$; therefore, with high probability at least one of the sets $C_1, \ldots, C_t$ isolates v. The Isolating-Cuts procedure is called for each $C_{i'}$, getting new estimates for all $v \in V_i$; a total of $\tilde{O}(\sqrt{w})$ queries for all large expanders. The intuition is that a large righty expander helps the algorithm pick a good set of terminals that isolates v, a highly nontrivial task if v's subtree is large, by focusing the attention on a component that (presumably) only contains few nodes from the subtree. Indeed, the Isolating-Cuts procedure is not as helpful in **Case 3** where H_v is a large lefty expander and $|L|$ can be up to n, since randomly chosen terminals from H are unlikely to leave v isolated.

To handle the final **Case 3**, the algorithm identifies the set $\text{Top}_k(H)$ of the top $k = 3\sqrt{n} + 1$ nodes in terms of their $c'(u)$ estimate for each large expander H, and performs a Max-Flow query for each one. Since there are only $O(n/w) = O(1)$ large expanders, only $O(\sqrt{n})$ queries are performed. Suppose that v is indeed in a large lefty expander H_v with $|R| \leq 2\sqrt{w}$. The argument is that v must be among the queried nodes unless there are $> \sqrt{n}$ undone nodes $u \in C_v$ in its subtree.[12] This is because any done node $u \in C_v$ has $c'(u) = \lambda_{p,u} \leq \lambda_{p,v}$, since C_v is a valid (p, u)-cut of value $\lambda_{p,v}$, while $\lambda_{p,v} < c'(v)$ (otherwise v is already done). Therefore, any node $u \in H_v$ with $c'(u) \geq c'(v)$ can either be in $R \subseteq V \setminus C_v$ or it could be an undone node in $L \subseteq C_v$. There are at most $2\sqrt{w} + \sqrt{n}$ such nodes in total; thus $v \in \text{Top}_k(H_v)$. Observe that this argument does not work in **Case 2** where H_v is a large righty expander and $|R|$ can be up to n, since the nodes in R might have larger connectivity than v.

Finally, if v happens to be in a large lefty expander when its subtree contains $> \sqrt{n}$ undone nodes, then the algorithm is not guaranteed to find C_v in this iteration of procedure Expanders-Guided Querying. (v is a red node with many red nodes in its subtree.) Soon enough, within $O(\log n)$ iterations, the algorithm gets to a point where most of v's subtree has become done and then C_v is found. (The green from the leaves quickly "infects" the entire tree.)

1.4 Related Work

Harder settings. On the hardness side, the only related lower bounds are for All-Pairs Max-Flow in the harder settings of directed graphs [2, 6, 58] or undirected graphs with node weights [4], where Gomory–Hu trees cannot even exist, because the $\Omega(n^2)$ minimum cuts might all be different [47] (see therein also an interesting exposition of certain false claims made earlier). In particular, SETH gives an $n^{3-o(1)}$ lower bound for weighted sparse directed graphs [58] and the 4-Clique conjecture gives an $n^{\omega+1-o(1)}$ lower bound for unweighted dense directed graphs [2]. Nontrivial algorithms are known for unweighted directed graphs, with time $O(m^\omega)$ [23] (fast matrix multiplication techniques have only been helpful in directed graphs so far), and also for special graph classes such as planar

[10] Besides, of course, to speed up the Max-Flow queries: since all $\lambda_{s,t} \leq n$, the time bound of Karger and Levine [53] is $\tilde{O}(n^2)$.

[11] In the late stages of writing this paper, we have encountered the very recent work of Li and Panigrahy (FOCS 2020) and found out that they have also discovered this simple yet powerful procedure, naming it *Isolating Cuts Lemma* [61, Theorem II.2]. We have kept our proof in the full version, as it has a different flavor. The usage is similar but different: there, it deterministically finds the global minimum cut when it is unbalanced (has a small side), while here it finds many small subtrees at once.

[12] Recall that a node u is undone if $c'(u) > \lambda_{p,u}$ and done if $c'(u) = \lambda_{p,u}$.

[59] and bounded-treewidth [3, 10]. Moreover, algorithms exist for the case we only care about pairs of nodes whose Max-Flow value is bounded by small k [2, 15, 34]. Generalizations of the Gomory–Hu tree to other cut requirements such as multiway cuts or cuts between groups of nodes have also been studied [24, 29, 43–46].

Approximations. Coming up with faster constructions of a Gomory–Hu tree at the cost of approximations has also been of interest, see e.g. [73], with only few successful approaches so far. One approach is to sparsify the graph into $m' = \tilde{O}(\varepsilon^{-2}n)$ edges in randomized $\tilde{O}(m)$ time using the algorithm of Benczur and Karger [18] (or its generalizations), and then apply an exact Gomory–Hu tree algorithm on the sparse (but weighted) graph. Unfortunately, even when aiming for an approximate tree, each query throughout the Gomory–Hu (or the new) algorithm must be exact (see [3, 73]). Therefore, with current Max-Flow algorithms, a $(1+\varepsilon)$-approximate Gomory–Hu tree of unweighted graphs can be constructed in $\tilde{O}(\varepsilon^{-2}n^{2.5})$ time. Using a different approach that produces a flow-equivalent tree (rather than a cut-equivalent tree, meaning that the minimum cuts in the tree, viewed as node bipartitions, might not correspond to minimum cuts in the graph), one can design a $(1+\varepsilon)$-All-Pairs Max-Flow algorithm that runs in time $\tilde{O}(n^2)$ [3]. Finally, one can use Räcke's approach to compute a cut-sparsifier tree [75], which has a stronger requirement (it approximates all cuts of G) but can only give polylogarithmic approximation factors. Its fastest version runs in almost-linear time $m^{1+o(1)}$ and achieves $O(\log^4 n)$-approximation [76].

Applications and experimental studies. Cut-equivalent trees have appeared in countless application domains. One example is the pioneering work of Wu and Leahy [84] in 1993 on image segmentation using Gomory–Hu tree that has evolved into the *graph cuts* paradigm in computer vision. Another example is in telecommunications where Hu [49] showed that the Gomory–Hu tree is the optimal solution to the *minimum communication spanning tree* problem; consequently there is interest in characterizing which graphs have a Gomory–Hu tree that is a *subgraph* [57, 69]. In mathematical optimization, a seminal paper of Padberg and Rao [72] uses the Gomory–Hu tree to find odd cuts that are useful for the b-matching problem (and that have been used in a breakthrough NC algorithm for perfect matching in planar graphs [9]). The question of how the Gomory–Hu tree changes with the graph has arisen in applications such as energy and finance and has been investigated, e.g. [12, 42, 74], starting with Elmaghraby in 1964 [30] and up until very recently [13]. Motivated by the need of a scalable algorithm for Gomory–Hu Tree, Akiba et al. [7] have recently introduced a few heuristic ideas for getting a subcubic complexity in social networks and web graphs. Earlier, Goldberg and Tsioutsiouliklis [37] conducted an experimental comparison of the Gomory–Hu and Gusfield's algorithms.

Expander Decompositions. A key ingredient of our Gomory–Hu tree algorithm is an expander-decomposition of the graph [25, 51, 70, 71, 78, 80]. Such decompositions have led to several breakthroughs in algorithms for basic problems in the past decade, e.g. [56, 68, 81]. A typical application solves the problem on each expander and then somehow combines the answers, treating each expander as a node. The clique-like nature of each expander and the sparsity of the outer graph lead to gains in efficiency. This approach does not seem helpful for Gomory–Hu tree since there may not be any connection between the tree and the decomposition. The application in this paper is more reminiscent of recent deterministic Global-Min-Cut algorithms [55, 61, 77], but is also different from those (see Complication 5 in Section 1.3).

2 PRELIMINARIES

2.1 General Notations

We will mostly work with unweighted graphs $G = (V, E)$, but throughout our algorithms we might contract vertices and end up with auxiliary graphs $G = (V, E, c)$ that are weighted $c : E \to [U]$, i.e. with capacities in $[U] = \{1, \ldots, U\}$ on the edges. All graphs in this paper will be undirected. We denote by $\deg(v)$ and $\text{cdeg}(v)$ the number of edges and the total capacity on edges incident to $v \in V$, respectively. We treat *cuts* as subsets $S \subset V$, or partitions $(S, V \setminus S)$. The *value* of a cut S is defined as $\delta(S) = |\{\{u, v\} \in E : u \in S, v \in V \setminus S\}|$, and if two subsets $S, T \subseteq V$ are given, then $\delta(S, T) = |\{\{u, v\} \in E : u \in S, v \in T\}|$. When the graph is weighted we define the values as $\delta(S) = \sum_{\{u,v\} \in E, u \in S, v \in V \setminus S} c(u, v)$ and $\delta(S, T) = \sum_{\{u,v\} \in E, u \in S, v \in T} c(u, v)$. For a pair $u, v \in V(G)$, we denote by $\lambda_{u,v}$ or Max-Flow(u, v) the value of a minimum (u, v)-cut, $\delta(S)$.

2.2 Max Flow Algorithms: Unweighted, Weighted, and Combinatorial

We use as a black box known algorithms for Max-Flow to get a minimum (u, v)-cut for a given pair u, v. Throughout the paper, three existing algorithms are used. First, if the flow size is bounded, the Karger-Levine algorithm [53] that runs in time $\tilde{O}(m + nF)$ where F is the size of the maximum flow is particularly fast. Second, for larger flows we use the very recent algorithm [82] that runs in time $\tilde{O}(m + n^{1.5})$. All tools used in this paper are considered combinatorial with [82] being the only exception, as it uses interior-point methods from continuous optimization. Unlike other non-combinatorial methods such as fast matrix multiplication, these techniques tend to be fast in practice. Still, if one is interested in a purely combinatorial algorithm one can replace this bound for weighted graphs with the Goldberg-Rao algorithm [36] that has running time $\tilde{O}(min(n^{2/3}, m^{1/2})m \log U)$. The resulting algorithm is slower but still subcubic.

2.3 Gomory–Hu's Algorithm and Partial Trees

First, we give some general definitions that many algorithms that are related to Gomory–Hu trees use.

Partition Trees. A *partition tree* T of a graph $G = (V, E)$ is a tree whose nodes $1, \ldots, l$ are *super-nodes*, which means that each node i is associated with a subset $V_i \subseteq V$; and these super-nodes form a disjoint partition $V = V_1 \sqcup \cdots \sqcup V_l$. An *auxiliary graph* G_i is constructed from G by merging nodes that lie in the same connected component of $T \setminus \{i\}$. For example, if the current tree is a path on super-nodes $1, \ldots, l$, then G_i is obtained from G by merging $V_1 \cup \cdots \cup V_{i-1}$ into one *contracted node* and $V_{i+1} \cup \cdots \cup V_l$ into another contracted node. We will use the notations $n_i := |V_i|$, $m_i := |E(G_i)|$, and $n'_i := |V(G_i)|$. Note that $n'_i \geq n_i$ since $V(G_i)$

contains V_i as well as some other contracted nodes, with $n'_i = n_i$ only if the tree is a single node. The following is a brief description of the classical Gomory–Hu algorithm [38] (see Figure 3).

The Gomory–Hu algorithm. This algorithm constructs a cut-equivalent tree $\mathcal{T}$ in iterations. Initially, $\mathcal{T}$ is a single node associated with V (the node set of G), and the execution maintains the invariant that $\mathcal{T}$ is a partition tree of V. At each iteration, the algorithm picks arbitrarily two graph nodes s, t that lie in the same tree super-node i, i.e., $s, t \in V_i$. The algorithm then constructs from G the auxiliary graph G_i and invokes a Max-Flow algorithm to compute in this G_i a minimum (s,t)-cut, denoted C'. The submodularity of cuts ensures that this cut is also a minimum (s,t)-cut in the original graph G, and it clearly induces a disjoint partition $V_i = S \sqcup T$ with $s \in S$ and $t \in T$. The algorithm then modifies $\mathcal{T}$ by splitting super-node i into two super-nodes, one associated with S and one with T, that are connected by an edge whose weight is the value of the cut C', and further connecting each neighbor of i in $\mathcal{T}$ to either S or T (viewed as super-nodes), depending on its side in the minimum (s,t)-cut C' (more precisely, neighbor j is connected to the side containing V_j).

The algorithm performs these iterations until all super-nodes are singletons, which happens after $n-1$ iteration. Then, $\mathcal{T}$ is a weighted tree with effectively the same node set as G. It can be shown [38] that for every $s, t \in V$, the minimum (s,t)-cut in $\mathcal{T}$, viewed as a bipartition of V, is also a minimum (s,t)-cut in G, and of the same cut value. We stress that this property holds regardless of the choice made at each step of two nodes $s \neq t \in V_i$. A GH-Equivalent Partition Tree is a partition tree that can be obtained by a truncated execution of the Gomory–Hu algorithm, in the sense that there is a sequence of choices for the pairs $s \neq t \in V_i$ that can lead to such a tree. The following simple lemma describes the flexibility in designing cut-equivalent tree algorithms based on the Gomory–Hu framework.

Lemma 2.1. *Given a GH-Equivalent Partition Tree T' of an input graph G, and a cut-equivalent tree T_i of each auxiliary graph G_i for the super-nodes V_i of T', it is possible to construct a full cut-equivalent tree T of G in linear time.*

Proof. In a preprocessing step, for every super node V_i in T' and every contracted node $q \in V(G_i) \setminus V_i$, save a pointer to the super node $V_j \subseteq q$ that is adjacent to V_i in T'. Now, for every super-node V_i, identify each contracted node $q \in V(G_i) \setminus V_i$ with the super-node it contains that is adjacent to V_i in T', and connect the nodes of V_i to the super-nodes they are adjacent to (if any). Finally, if a node $u \in V_i$ is connected to a super-node V_j, and a node $v \in V_j$ is connected to V_i, remove these connections and connect u to v directly, and call the result T. Observe that T must be a tree.

To see why T is a correct cut-equivalent tree of G, observe that there exists a simulated Gomory–Hu execution that results in T. Given the GH-Equivalent Partition Tree T', pick pairs of nodes from V_i and cuts according to T_i. This is guaranteed to produce a tree $\tilde{T}_i$ whose projection on V_i is identical to T_i, while the subtrees adjacent to V_i in $\tilde{T}$ are connected to the same nodes of V_i as their contracted counterparts in T_i. Applying this simulated execution to all super-nodes concludes the proof. □

Next, we discuss a few kinds of (GH-Equivalent) partition trees.

2.3.1 Partial Trees for Subsets. A *partial tree* for a subset $Q \subseteq V(G)$ is a partition tree T of G such that each super-node V_i in T contains exactly one node from Q, and for every two nodes $a, b \in Q$, the minimum cut in T between the super-nodes containing them A, B is a minimum (a,b)-cut in G. It is folklore that for every subset $Q \subseteq V$, a partial tree for Q can be computed by $|Q|-1$ applications of Max-Flow.

Lemma 2.2 (see e.g. [39])**.** *Given a graph $G = (V, E, c)$ and a subset $Q \subseteq V$, it is possible to compute a partial tree for Q in time $O(|Q|T(m))$, where $T(m)$ is the time for solving (single pair) Max-Flow.*

This lemma follows by running a Gomory–Hu execution, but always picking pairs from Q (thus making T a GH-Equivalent Partition Tree).

2.3.2 k-Partial Trees. A k-partial tree, formally defined below, can also be thought of as the result of contracting all edges of weight greater than k in a cut-equivalent tree of G. Such a tree can obviously be constructed using the Gomory–Hu algorithm, but as stated below (in Lemma 2.4), faster algorithms were designed in [15, 41], see also [73, Theorem 3]. It is known (see [4, Lemma 2.3]) that such a tree is a GH-Equivalent Partition Tree.

Definition 2.3 (k-Partial Tree [41])**.** *A k-*partial tree *of a graph $G = (V, E)$ is a weighted tree on $l \leq |V|$ super-nodes constituting a partition $V = V_1 \sqcup \cdots \sqcup V_l$, with the following property: For every two nodes $s, t \in V$ whose minimum-cut value in G is at most k, s and t lie in different super-nodes $s \in S$ and $t \in T$, such that the minimum (S,T)-cut in the tree defines a bipartition of V which is a minimum (s,t)-cut in G and has the same value.*

Lemma 2.4 ([15])**.** *There is an algorithm that given an undirected graph with n nodes and m edges with unit edge-capacities and an integer $k \in [n]$, constructs a k-partial tree in time $\min\{\tilde{O}(nk^2), \tilde{O}(mk)\}$.*

2.3.3 A Basic Property of the Gomory–Hu Tree.

Lemma 2.5. *Given a graph $G = (V, E)$ and a tree T on the same set of nodes, if for every edge $uv \in T$ the cut (S_u, S_v) resulting from removing uv in T is a minimum cut in G, then T is a cut-equivalent tree.*

Proof. The proof follows by simulating a Gomory–Hu tree execution with node pairs and minimum cuts taken according to the edges in T. □

2.4 Nagamochi–Ibaraki Sparsification

We use the sparsification method by Nagamochi and Ibaraki [67], who showed that for any graph G it is possible to find a subgraph H with at most $k(n-1)$ edges, such that H contains all edges crossing cuts of value k or less. It follows that if a cut has value at most $k-1$ in G then it has the same value in H, and if a cut has value at least k in G then it also has value at least k in H. The authors [67] gave an algorithm that performs this sparsification in $O(m)$ time on unweighted graphs, independent of k. They also gave a sparsification algorithm for weighted graphs, with an $O(m + n \log n)$ running time [66]. In weighted graphs, the sparsification is

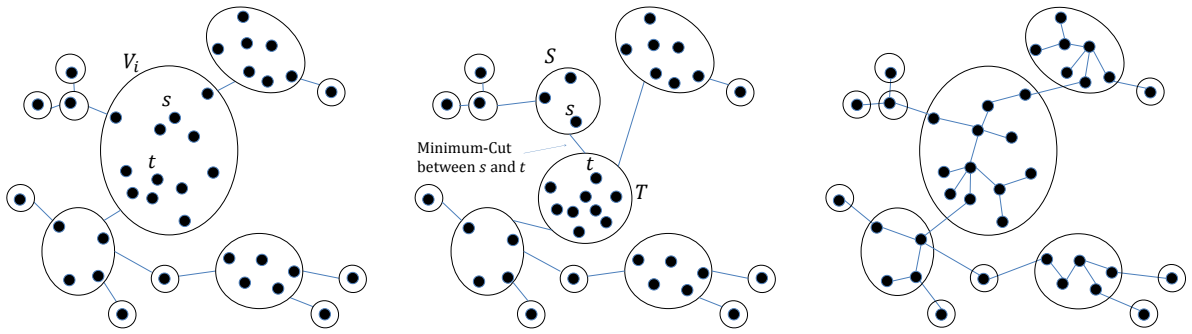

Figure 3: Illustration of the construction of $\mathcal{T}$. Left: $\mathcal{T}$ right before the partition of the super-node V_i. Middle: after the partitioning of V_i. Right: $\mathcal{T}$ as it unfolds after the Gomory–Hu algorithm finishes.

defined by equating an edge of weight w with a set of w unweighted (parallel) edges with the same endpoints.

2.5 Latest Cuts

Introduced by Gabow [31], a *latest* minimum (s,t)-cut (or a *minimal* minimum (s,t)-cut, in some literature) is a minimum (s,t)-cut $(S_s, S_t = V \setminus S_s)$ such that no strict subset of S_t is a minimum (s,t)-cut. It is known that latest minimum cuts are unique, and can be found in the same running time of any algorithm that outputs the maximum network flow between the pair, by finding all nodes that can reach t in the residual graph. In particular, all upper bound stated in Section 2.2 above for Max-Flow also hold for finding the latest minimum (s,t)-cut.

We will use the following properties of cuts.

Fact 2.6 (Submodularity of cuts)**.** *For every two subsets of nodes $A, B \subseteq V$, it holds that $\delta(A) + \delta(B) \geq \delta(A \cup B) + \delta(A \cap B)$.*

Fact 2.7 (Posimodularity of cuts)**.** *For every two subsets of nodes $A, B \subseteq V$, it holds that $\delta(A) + \delta(B) \geq \delta(A \setminus B) + \delta(B \setminus A)$.*

Lemma 2.8. *Let G be any graph and p be any node. There is a cut-equivalent tree that contains all the latest minimum cuts with respect to p.*

Proof. This follows because all latest minimum cuts with respect to p form a laminar family, by Fact 2.6. □

Lemma 2.9. *If A, B are minimum (p,a)-cut and minimum (p,b)-cut, respectively, and $b \in A$ then $A \cup B$ is a minimum cut for a.*

Proof. Considering Fact 2.6, it only remains to show that $\delta(B) - \delta(A \cap B) \leq 0$. But this is immediate, as b is in both sets, and B is a minimum cut for b. □

Lemma 2.10. *If A, B are minimum (p,a)-cut and minimum (p,b)-cut, respectively, and $a \notin B, b \notin A$ then $A \setminus B$ is a minimum cut for a.*

Proof. Considering Fact 2.7, it only remains to show that $\delta(B) - \delta(B \setminus A) \leq 0$. But this is immediate, as b is in both sets, and B is a minimum cut for b. □

2.6 Expander Decomposition

We mostly follow notations and definition from [78]. Let $\mathrm{vol}_G(C) = \sum_{v \in C} \mathrm{cdeg}_G(v)$ be the *volume* of $C \subseteq V$, where subscripts indicate what graph we are using, and are omitted if it is clear from the context. The *conductance* of a cut S in G is $\Phi_G(S) = \frac{\delta(S)}{\min(\mathrm{vol}_G(S), \mathrm{vol}_G(V \setminus S))}$. The expansion of a graph G is $\Phi_G = \min_{S \subset V} \Phi_G(S)$. If G is a singleton, we define $\Phi_G = 1$. Let $G[S]$ be the subgraph induced by $S \subset V$, and we denote $G\{S\}$ as the induced subgraph $G[S]$ but with added self-loops $e = (v,v)$ for each edge $e' = (v,u)$ where $v \in S, u \notin S$ (where each self-loop contributes 1 to the degree of a node), so that any node in S has the same degree as its degree in G. Observe that for any $S \subset V$, $\Phi_{G[S]} \geq \Phi_{G\{S\}}$, because the self-loops increase the volumes but not the values of cuts. We say a graph G is a ϕ *expander* if $\Phi_G \geq \phi$, and we call a partition $H_1, \ldots, H_h$ of V a ϕ expander decomposition if $\min_i \Phi_{G[V_i]} \geq \phi$.

Theorem 2.11 (Theorem 1.2 in [78])**.** *Given a graph $G = (V,E)$ of m edges and a parameter ϕ, there is a randomized algorithm that with high probability finds a partitioning of V into $V_1, \ldots, V_k$ such that $\forall i : \Phi_{G[V_i]} \geq \phi$ and $\sum_i \delta(V_i) = O(\phi m \log^3 m)$. In fact, the algorithm has a stronger guarantee that $\forall i : \Phi_{G\{V_i\}} \geq \phi$. The running time of the algorithm is $O(m \log^4 m/\phi)$.*

3 MOTIVATING EXAMPLES AND THE ISOLATING-CUTS PROCEDURE

This section attempts to explain the thought process that has led to the introduction of two new tools into the context of Gomory–Hu tree algorithms: the Isolating-Cuts and the Expanders-Guided Querying procedures, as well as to give more details about the former while deferring the latter to the full version. Let G be a graph and T be its cut-equivalent tree. As described in Section 1.3, previous techniques (Gomory–Hu recursion, partial trees, and randomized pivot selection) can reach the point that T, when rooted at a designated pivot p, has small sublinear depth. This suggests that a path-like T is not the hardest to construct since its size must be sublinear. It is natural to wonder if the other extreme of star-like T is also easy, or perhaps it can lead to a hardness reduction.

The most extreme case is when T is precisely a star with p at the center and all other nodes as leaves. Constructing this tree is easy because the minimum (p, v)-cut for all $v \in V$ is the singleton $(\{v\}, V \setminus \{v\})$ and its value is $\deg(v)$. (However, verifying that the star is the correct tree is perhaps as hard as the general problem.)

The following two examples show that by changing the star slightly one reaches challenging scenarios that require new tools.

3.1 Many Small Subtrees

Consider a graph whose cut-equivalent tree T is a "star of triples graphs" as depicted in Figure 4. In this example, T is simply a star on n nodes where each leaf is connected to additional two leaves, and so altogether the number of nodes is $3n + 1$. Call the center node p, the inner nodes $L_1 = \{u_1, \ldots, u_n\}$, and the leaves L_2. Assuming that p and L_1 are given in advance, how fast can we identify which pair of leaves belongs to which inner node in the cut-equivalent tree (thus constructing T)? Note that it can be done by simply asking a Max-Flow query between p and each node $u \in L_1$, but can it be done faster than $\Omega(n)$ applications of Max-Flow?[13]

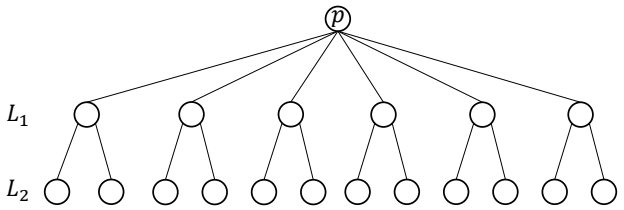

Figure 4: The cut-equivalent tree T.

It turns out that the answer is yes! A simple but surprising algorithm computes T using only $\tilde{O}(1)$ queries to a Max-Flow algorithm (on a weighted graph).

The Algorithm.

(1) Add edges of very large capacity $U := |E|^2$ from p to every node in L_1. Denote the new graph by G_m and its cut-equivalent tree by T_m.
(2) Repeat $10 \log^2 n$ times:
   (a) In G_m split p to p_1 and p_2, connecting the (previously connected to p) newly added edges to p_1 or p_2 with probability $1/2$, and the rest of the edges arbitrarily. Then add an edge of weight U^2 between p_1 and p_2. Denote the new graph G_h and its cut-equivalent by T_h.
   (b) Find the minimum (p_1, p_2)-cut in G_h.
(3) Construct the new tree T' as follows. For every node $u \in L_1$ connect it in T' to the leaves in L_2 that went to the same side as u in all repetitions.
(4) If any node in L_1 has a number of leaves that is different than two, output "failure", otherwise report T'.

Why does it work? Consider G_m, T_m, G_h, G_h in one iteration.

Observation 3.1. *The tree T_m is identical to T, except for the weights of edges adjacent to p.*

[13] Readers familiar with All-Pairs Shortest-Path-hardness results may recall that triangle identification is at the core of most reductions, making this example appealing.

For every edge $\{u, v\}$ in T, consider two cases. If u or v is p, then the new edge increases all (u, v)-cuts by at least U, and since the old cut increases by exactly U, it is a minimum cut in T_m as well. Otherwise, if neither of u nor v is p, then the minimum (u, v)-cut is still the same because the new edges did not affect this cut, and can only increase the weights of other cuts. Thus, the minimum (u, v)-cut in each case is the same in T and T_m.

Observation 3.2. *There is an edge between p_1 and p_2 in T_h. If it is contracted T_h becomes identical to T_m.*

This is true because p_1 and p_2 are on the same side of every minimum (a, b)-cut for any pair $\{a, b\} \neq \{p_1, p_2\}$ because the weight of the edge between them is larger than the sum of all others.

The following is the key claim in this section.

Claim 3.3. *The minimum (p_1, p_2)-cut in G_h sends each pair of siblings in L_2 to the same side as their parent in L_1.*

Proof. Let $u_1, u_2 \in L_2$ be a pair of siblings whose father is $u \in L_1$. By Observations 3.1 and 3.2, u_1 and u_2 are u's neighbors in T_h. Consider T_h after the removal of the edge $\{p_1, p_2\}$: the two resulting subtrees are the two sides of the minimum (p_1, p_2)-cut in G_h. Since u, u_1, u_2 are connected in T_h, it follows that they must be on the same side of the cut. □

Hence, every node $u \in L_1$ can know its children with high probability by seeing which two nodes in L_2 are always sent with it to the same side of the minimum (p_1, p_2)-cuts throughout all iterations.

3.2 The Isolating-Cuts Procedure

The above algorithm can be generalized to obtain the Isolating-Cuts procedure that is a key ingredient of the new algorithm. Notably, the added weights are not necessary (can be replaced with contractions) nor is the randomness (can be replaced with deterministic separating choices). Let $MF(N, M, F)$ be an upper bound on Max-Flow in graphs with N nodes, M edges, and where the flow size is bounded by F. We will utilize the following statement that essentially follows from the very recent work of Li and Panigrahy [61, Theorem II.2] for Global-Min-Cut. We provide another proof in the full version both for completeness and because it could be of interest as it exploits the structure of the Gomory–Hu tree (as we did above) instead of using the submodularity of cuts directly. In particular, we use ideas similar to Lemma 2.5.

Lemma 3.4 (The Isolating-Cuts Procedure)**.** *Given an undirected graph $G = (V, E, c)$ on n nodes and m total edges, a* pivot *node $p \in V$, and a set of* connected *vertices $C \subseteq V$, let $(C_v, V \setminus C_v)$ where $v \in C_v, p \in V \setminus C_v$ be the latest minimum (p, v)-cut for each $v \in C$. There is a deterministic $O(MF(n, m, c(E)) \cdot \log n)$-time algorithm that returns $|C|$ disjoint sets $\{C'_v\}_{v \in C}$ such that for all $v \in C$: if $C_v \cap C = \{v\}$ then $C'_v = C_v$.*

3.3 A Single Large Subtree

To solve the previous example, we have exploited the fact that the identity of the nodes in L_1 is known to us in advance. However, by picking each node to be "connected" (and added to the set C) with probability $1/2$ there is a $1/8$ chance that a parent $u \in L_1$

is in C while its two children $u_1, u_2 \in L_2$ are not. Then, applying Lemma 3.4 $O(\log n)$ times is sufficient for constructing T. More generally, if all nodes in T have a subtree that is small, e.g. with n^ϵ descendants, then n^ϵ repetitions of the Isolating-Cuts procedure with random choices for C ought to suffice.

This leads to the second hard case where there is a single (unknown) node with a subtree of size $\Omega(n)$. Consider the example in Figure 5 where T is composed of a left star centered at c_ℓ and a right star centered at c_r, and suppose that c_r is the (known) pivot p while the challenge is to identify c_ℓ. The minimum (c_ℓ, c_r)-cut is large and balanced (C_ℓ, C_r) while nearly all other minimum (u, v)-cuts in the graph are the trivial singleton cuts. The weights on T can be set up so that c_ℓ is almost like a needle in a haystack: asking almost any (v, c_r) Max-Flow query for any $v \neq c_\ell$ may not reveal any information about c_ℓ. Suppose that λ_{c_ℓ, c_r} is w, about half of the edges between c_r and its children are $> w$ and half are $< w$, while nearly all nodes on the left have an edge of weight $< w$ to c_ℓ. Thus, even if a randomly chosen pair u, v will have one node on each side, the minimum (u, v)-cut will be either $(\{u\}, V \setminus \{u\})$ or $(\{v\}, V \setminus \{v\})$ with high probability, which does not help discover c_ℓ nor any of its cut members C_ℓ.

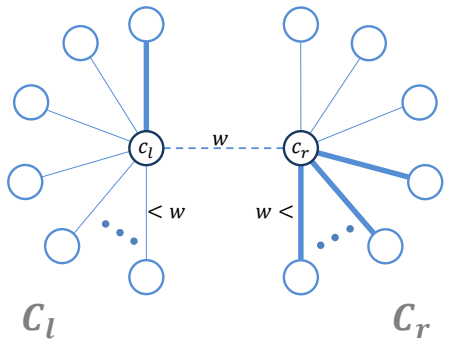

Figure 5: The cut-equivalent tree T of a hard case where c_r is given but c_ℓ, the only node with a non-trivial minimum cut to c_r, must be identified. The weight of the (C_ℓ, C_r) cut is $\lambda_{c_l, c_r} = w$, illustrated by the dashed edge at the center, and the edges of weight $> w$ are thick, while edges of weight $< w$ are thin. The minimum cut between any pair of leaves u, v is trivial unless u is attached to one of the bold edges on the left and v is attached to one of the bold edges on the right.

To apply the Isolating-Cuts procedure one must find a way to *isolate* c_ℓ, i.e. to choose a set of connected nodes C such that c_ℓ is in C but none of its $\Omega(n)$ cut-members are in C. This is an impossible task *unless* the structure of the graph G is exploited. The Expanders-Guided Querying procedure manages to do just that, as overviewed in Section 1.3 and fully described in the full version, by exploiting a decomposition of G to either reach a good C or to directly query the pair c_ℓ, c_r.

Notably, this example remains the bottleneck after the new algorithm and might lead the way to the first conditional lower bound for Gomory–Hu tree and All-Pairs Max-Flow.

4 CONCLUSION

This paper presents the first algorithm with subcubic in n running time for constructing a Gomory–Hu tree of a simple graph and, consequently, for solving the All-Pairs Max-Flow problem. It is achieved by a combination of several tools from the literature on this problem, as well as two new ingredients: the Expanders-Guided Querying and Isolating-Cuts procedures. The new ideas are reminiscent of recent algorithms [55, 61, 77] for the easier problem of Global-Min-Cut. We conclude with some remarks and open questions.

- The assumption that the graph is unweighted is only used in one specific case of the analysis for observing that: if a high degree node is in a small component then most of its edges must leave the component. A similar observation is at the heart of the breakthrough deterministic Global-Min-Cut algorithm of Kawarabayashi and Thorup [55] but can now be avoided with a moderate loss in efficiency [61]. Thus there is room for optimism that $n^{1-\epsilon} \cdot T_{\text{Max-Flow}}(n, m)$ time for *weighted graphs* is possible with the available tools.
- The new subcubic algorithm uses randomness in multiple places and succeeds with high probability. All of the ingredients can already be derandomized (with some loss) using existing methods, except for one: the randomized pivot selection (see Complication 1 in Section 1.3). It is likely that a fully deterministic algorithm making $n^{1-\epsilon}$ queries is attainable but there is an inherent challenge that has also prevented the $\tilde{O}(n^3)$ time algorithm [15] from being derandomized yet. However, matching the new $\tilde{O}(n^{2.5})$ bound deterministically seems to require more new ideas, including a deterministic $\tilde{O}(n^2)$ algorithm for Max-Flow in simple graphs that can be used instead of Karger-Levine [53].
- Perhaps the most interesting open question is whether $\tilde{O}(m)$ time can be achieved, even in simple graphs and even assuming a linear-time Max-Flow algorithm. The simplest case where breaking the $n^{2.5}$ bound is still challenging has been isolated in Section 3.3; perhaps it will lead to the first conditional lower bound for computing a Gomory–Hu tree?

ACKNOWLEDGMENTS

We thank the anonymous reviewers for many helpful comments.

REFERENCES

[1] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams. 2015. Tight Hardness Results for LCS and Other Sequence Similarity Measures. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015*. 59–78. https://doi.org/10.1109/FOCS.2015.14

[2] Amir Abboud, Loukas Georgiadis, Giuseppe F. Italiano, Robert Krauthgamer, Nikos Parotsidis, Ohad Trabelsi, Przemyslaw Uznanski, and Daniel Wolleb-Graf. 2019. Faster Algorithms for All-Pairs Bounded Min-Cuts. In *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, Vol. 132. 7:1–7:15. https://doi.org/10.4230/LIPIcs.ICALP.2019.7

[3] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. 2020. Cut-Equivalent Trees are Optimal for Min-Cut Queries. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. 105–118. https://doi.org/10.1109/FOCS46700.2020.00019

[4] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. 2020. New Algorithms and Lower Bounds for All-Pairs Max-Flow in Undirected Graphs. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '20)*. USA, 48–61. https://doi.org/10.1137/1.9781611975994.4

[5] Amir Abboud and Virginia Vassilevska Williams. 2014. Popular Conjectures Imply Strong Lower Bounds for Dynamic Problems. In *55th IEEE Annual Symposium on*

Foundations of Computer Science, FOCS 2014. 434–443. https://doi.org/10.1109/FOCS.2014.53

[6] Amir Abboud, Virginia Vassilevska Williams, and Huacheng Yu. 2018. Matching Triangles and Basing Hardness on an Extremely Popular Conjecture. *SIAM J. Comput.* 47, 3 (2018), 1098–1122. https://doi.org/10.1137/15M1050987

[7] Takuya Akiba, Yoichi Iwata, Yosuke Sameshima, Naoto Mizuno, and Yosuke Yano. 2016. Cut tree construction from massive graphs. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*. IEEE, 775–780.

[8] Josh Alman and Virginia Vassilevska Williams. 2021. A Refined Laser Method and Faster Matrix Multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*. 522–539. https://doi.org/10.1137/1.9781611976465.32

[9] Nima Anari and Vijay V Vazirani. 2020. Planar Graph Perfect Matching Is in NC. *J. ACM* 67, 4 (2020), 1–34.

[10] Srinivasa Rao Arikati, Shiva Chaudhuri, and Christos D. Zaroliagis. 1998. All-Pairs Min-Cut in Sparse Networks. *J. Algorithms* 29, 1 (1998), 82–110. https://doi.org/10.1006/jagm.1998.0961

[11] Nikhil Bansal and Ryan Williams. 2009. Regularity Lemmas and Combinatorial Algorithms. In *50th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2009*. 745–754. https://doi.org/10.1109/FOCS.2009.76

[12] Dominique Barth, Pascal Berthomé, Madiagne Diallo, and Afonso Ferreira. 2006. Revisiting parametric multi-terminal problems: Maximum flows, minimum cuts and cut-tree computations. *Discrete Optimization* 3, 3 (2006), 195–205.

[13] Surender Baswana, Shiv Gupta, and Till Knollmann. 2020. Mincut Sensitivity Data Structures for the Insertion of an Edge. In *28th Annual European Symposium on Algorithms (ESA 2020)*.

[14] Anand Bhalgat, Richard Cole, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. 2008. Efficient Algorithms for Steiner Edge Connectivity Computationand Gomory-Hu Tree Construction for Unweighted Graphs. (2008). http://hariharan-ramesh.com/papers/gohu.pdf Unpublished full version of [15].

[15] Anand Bhalgat, Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. 2007. An $O(mn)$ Gomory-Hu Tree Construction Algorithm for Unweighted Graphs. In *39th Annual ACM Symposium on Theory of Computing (STOC'07)*. 605–614. https://doi.org/10.1145/1250790.1250879

[16] Glencora Borradaile, David Eppstein, Amir Nayyeri, and Christian Wulff-Nilsen. 2016. All-Pairs Minimum Cuts in Near-Linear Time for Surface-Embedded Graphs. In *32nd International Symposium on Computational Geometry (SoCG '16, Vol. 51)*. 22:1–22:16. https://doi.org/10.4230/LIPIcs.SoCG.2016.22

[17] Glencora Borradaile, Piotr Sankowski, and Christian Wulff-Nilsen. 2015. Min st-Cut Oracle for Planar Graphs with Near-Linear Preprocessing Time. *ACM Trans. Algorithms* 11, 3 (2015), 29 pages. https://doi.org/10.1145/2684068

[18] Karl Bringmann and Marvin Kunnemann. 2015. Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. In *Proc. of 56th FOCS*. 79–97.

[19] Chris Calabro, Russell Impagliazzo, and Ramamohan Paturi. 2006. A Duality between Clause Width and Clause Density for SAT. In *21st Annual IEEE Conference on Computational Complexity, CCC 2006*. 252–260. https://doi.org/10.1109/CCC.2006.6

[20] Marco L. Carmosino, Jiawei Gao, Russell Impagliazzo, Ivan Mihajlin, Ramamohan Paturi, and Stefan Schneider. 2016. Nondeterministic Extensions of the Strong Exponential Time Hypothesis and Consequences for Non-reducibility. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science (ITCS '16)*. 261–270. https://doi.org/10.1145/2840728.2840746

[21] Timothy M. Chan. 2012. All-pairs shortest paths for unweighted undirected graphs in $o(mn)$ time. *ACM Trans. Algorithms* 8, 4 (2012), 34:1–34:17. https://doi.org/10.1145/2344422.2344424

[22] Timothy M. Chan. 2015. Speeding up the Four Russians Algorithm by About One More Logarithmic Factor. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*. 212–217. https://doi.org/10.1137/1.9781611973730.16

[23] Ho Yee Cheung, Lap Chi Lau, and Kai Man Leung. 2013. Graph connectivities, network coding, and expander graphs. *SIAM J. Comput.* 42, 3 (2013), 733–751. https://doi.org/10.1137/110844970

[24] Rajesh Chitnis, Lior Kamma, and Robert Krauthgamer. 2016. Tight Bounds for Gomory–Hu-like Cut Counting. In *42nd International Workshop on Graph-Theoretic Concepts in Computer Science (Lecture Notes in Computer Science, Vol. 9941)*. 133–144. https://doi.org/10.1007/978-3-662-53536-3_12

[25] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. 2020. A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond. In *61st IEEE Annual Symposium on Foundations of Computer Science, FOCS 2020*. 1158–1167. https://doi.org/10.1109/FOCS46700.2020.00111

[26] Richard Cole and Ramesh Hariharan. 2003. A Fast Algorithm for Computing Steiner Edge Connectivity. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC '03)*. 167–176. https://doi.org/10.1145/780542.780568

[27] Don Coppersmith and Shmuel Winograd. 1987. Matrix multiplication via arithmetic progressions. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*. 1–6.

[28] Jack Edmonds. 1970. Submodular functions, matroids, and certain polyhedra. *Combinatorial structures and their applications* (1970), 69–87.

[29] Ori Einstein and Refael Hassin. 2005. The number of solutions sufficient for solving a family of problems. *Mathematics of Operations Research* 30, 4 (2005), 880–896.

[30] Salah E Elmaghraby. 1964. Sensitivity analysis of multiterminal flow networks. *Operations Research* 12, 5 (1964), 680–688.

[31] Harold N. Gabow. 1991. Applications of a Poset Representation to Edge Connectivity and Graph Rigidity. In *32nd Annual Symposium on Foundations of Computer Science*. 812–821. https://doi.org/10.1109/SFCS.1991.185453

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

[33] Pawel Gawrychowski, Shay Mozes, and Oren Weimann. 2020. Minimum Cut in $O(m \log^2 n)$ Time. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020 (LIPIcs, Vol. 168)*, Artur Czumaj, Anuj Dawar, and Emanuela Merelli (Eds.). 57:1–57:15. https://doi.org/10.4230/LIPIcs.ICALP.2020.57

[34] L. Georgiadis, G. F. Italiano, L. Laura, and N. Parotsidis. 2016. 2-Edge Connectivity in Directed Graphs. *ACM Transactions on Algorithms* 13, 1 (2016), 9:1–9:24. https://doi.org/10.1145/2968448

[35] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. 2020. Faster algorithms for edge connectivity via random 2-out contractions. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 1260–1279.

[36] Andrew V. Goldberg and Satish Rao. 1998. Beyond the Flow Decomposition Barrier. *J. ACM* 45, 5 (1998), 783–797. https://doi.org/10.1145/290179.290181

[37] Andrew V. Goldberg and Kostas Tsioutsiouliklis. 2001. Cut tree algorithms: an experimental study. *Journal of Algorithms* 38, 1 (2001), 51–83.

[38] Ralph E. Gomory and Te C. Hu. 1961. Multi-Terminal Network Flows. *J. Soc. Indust. Appl. Math.* 9 (1961), 551–570. http://www.jstor.org/stable/2098881

[39] Frieda Granot and Refael Hassin. 1986. Multi-terminal maximum flows in node-capacitated networks. *Discrete Applied Mathematics* 13, 2-3 (1986), 157–163.

[40] Dan Gusfield. 1990. Very simple methods for all pairs network flow analysis. *SIAM J. Comput.* 19, 1 (1990), 143–155.

[41] Ramesh Hariharan, Telikepalli Kavitha, and Debmalya Panigrahi. 2007. Efficient algorithms for computing all low $s-t$ edge connectivities and related problems. In *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 127–136. http://dl.acm.org/citation.cfm?id=1283383.1283398

[42] Tanja Hartmann and Dorothea Wagner. 2013. Dynamic Gomory–Hu Tree Construction–fast and simple. *arXiv preprint arXiv:1310.0178* (2013).

[43] David Hartvigsen. 2001. Compact representations of cuts. *SIAM Journal on Discrete Mathematics* 14, 1 (2001), 49–66.

[44] Refael Hassin. 1988. Solution bases of multiterminal cut problems. *Mathematics of Operations Research* 13, 4 (1988), 535–542. https://doi.org/10.1287/moor.13.4.535

[45] Rafael Hassin. 1990. An algorithm for computing maximum solution bases. *Operations research letters* 9, 5 (1990), 315–318.

[46] Refael Hassin. 1991. Multiterminal xcut problems. *Annals of Operations Research* 33, 3 (1991), 215–225.

[47] Refael Hassin and Asaf Levin. 2007. Flow Trees for Vertex-capacitated Networks. *Discrete Appl. Math.* 155, 4 (2007), 572–578. https://doi.org/10.1016/j.dam.2006.08.012

[48] Monika Henzinger, Satish Rao, and Di Wang. 2020. Local flow partitioning for faster edge connectivity. *SIAM J. Comput.* 49, 1 (2020), 1–36. https://doi.org/10.1137/18M1180335

[49] Te C. Hu. 1974. Optimum communication spanning trees. *SIAM J. Comput.* 3, 3 (1974), 188–195. https://doi.org/10.1137/0203015

[50] Russell Impagliazzo and Ramamohan Paturi. 2001. On the Complexity of k-SAT. *J. Comput. System Sci.* 62, 2 (March 2001), 367–375. https://doi.org/10.1006/jcss.2000.1727

[51] Ravi Kannan, Santosh Vempala, and Adrian Vetta. 2004. On clusterings: Good, bad and spectral. *J. ACM* 51, 3 (2004), 497–515. https://doi.org/10.1145/990308.990313

[52] David R. Karger. 2000. Minimum cuts in near-linear time. *J. ACM* 47, 1 (2000), 46–76.

[53] David R. Karger and Matthew S. Levine. 2015. Fast Augmenting Paths by Random Sampling from Residual Graphs. *SIAM J. Comput.* 44, 2 (2015), 320–339. https://doi.org/10.1137/070705994

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

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

[56] Jonathan A Kelner, Yin Tat Lee, Lorenzo Orecchia, and Aaron Sidford. 2014. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 217–226.

[57] Bernhard Korte and Jens Vygen. 2012. *Combinatorial optimization*. Vol. 2. Springer.

[58] Robert Krauthgamer and Ohad Trabelsi. 2018. Conditional Lower Bounds for All-Pairs Max-Flow. *ACM Trans. Algorithms* 14, 4 (2018), 42:1–42:15. https:

//doi.org/10.1145/3212510

[59] Jakub Lacki, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. 2012. Single Source - All Sinks Max Flows in Planar Digraphs. In *Proc. of the 53rd FOCS*. 599–608.

[60] Yin Tat Lee and Aaron Sidford. 2014. Path Finding Methods for Linear Programming: Solving Linear Programs in $\tilde{O}(\sqrt{rank})$ Iterations and Faster Algorithms for Maximum Flow. In *55th Annual Symposium on Foundations of Computer Science (FOCS '14)*. 424–433. https://doi.org/10.1109/FOCS.2014.52

[61] 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. https://doi.org/10.1109/FOCS46700.2020.00017

[62] Yang P. Liu and Aaron Sidford. 2019. Faster Energy Maximization for Faster Maximum Flow. *CoRR* (2019). http://arxiv.org/abs/1910.14276

[63] Yang P. Liu and Aaron Sidford. 2020. Unit Capacity Maxflow in Almost $m^{4/3}$ Time. *CoRR* abs/2003.08929 (2020). arXiv:2003.08929 https://arxiv.org/abs/2003.08929

[64] Aleksander Mądry. 2016. Computing Maximum Flow with Augmenting Electrical Flows. In *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science (FOCS '16)*. 593–602. https://doi.org/10.1109/FOCS.2016.70

[65] Sagnik Mukhopadhyay and Danupon Nanongkai. 2020. Weighted min-cut: sequential, cut-query, and streaming algorithms. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing*. 496–509.

[66] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs. *SIAM J. Discret. Math.* 5, 1 (1992), 54–66. https://doi.org/10.1137/0405004

[67] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. Linear time algorithms for finding k-edge connected and k-node connected spanning subgraphs. *Algorithmica* 7 (1992), 583–596. https://doi.org/10.1007/BF01758778

[68] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. 2017. Dynamic minimum spanning forest with subpolynomial worst-case update time. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 950–961.

[69] Guyslain Naves and F Bruce Shepherd. 2018. When Do Gomory-Hu Subtrees Exist? *CoRR* (2018). http://arxiv.org/abs/1807.07331

[70] Lorenzo Orecchia, Sushant Sachdeva, and Nisheeth K. Vishnoi. 2012. Approximating the exponential, the Lanczos method and an $\tilde{O}(m)$-time spectral algorithm for balanced separator. In *Proceedings of the 44th Annual ACM Symposium on Theory of Computing*. 1141–1160. https://doi.org/10.1145/2213977.2214080

[71] Lorenzo Orecchia and Nisheeth K Vishnoi. 2011. Towards an SDP-based approach to spectral methods: A nearly-linear-time algorithm for graph partitioning and decomposition. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 532–545.

[72] Manfred W Padberg and M Ram Rao. 1982. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research* 7, 1 (1982), 67–80.

[73] Debmalya Panigrahi. 2016. Gomory-Hu Trees. In *Encyclopedia of Algorithms*, Ming-Yang Kao (Ed.). Springer New York, 858–861. https://doi.org/10.1007/978-1-4939-2864-4_168

[74] Jean-Claude Picard and Maurice Queyranne. 1980. On the structure of all minimum cuts in a network and applications. In *Combinatorial Optimization II*. Springer, 8–16.

[75] Harald Räcke. 2002. Minimizing Congestion in General Networks. In *Proceedings of the 43rd Symposium on Foundations of Computer Science (FOCS '02)*. 43–52. https://doi.org/10.1109/SFCS.2002.1181881

[76] Harald Räcke, Chintan Shah, and Hanjo Täubig. 2014. Computing Cut-Based Hierarchical Decompositions in Almost Linear Time. In *Proceedings of the 25th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '14)*. 227–238. https://doi.org/10.1137/1.9781611973402.17

[77] Thatchaphol Saranurak. 2021. A Simple Deterministic Algorithm for Edge Connectivity. In *4th Symposium on Simplicity in Algorithms, SOSA 2021*. 80–85. https://doi.org/10.1137/1.9781611976496.9

[78] Thatchaphol Saranurak and Di Wang. 2019. Expander Decomposition and Pruning: Faster, Stronger, and Simpler. In *Proceedings of the 30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*. 2616–2635. https://doi.org/10.1137/1.9781611975482.162

[79] Raimund Seidel. 1995. On the all-pairs-shortest-path problem in unweighted undirected graphs. *Journal of computer and system sciences* 51, 3 (1995), 400–403.

[80] Daniel A Spielman and Shang-Hua Teng. 2013. A local clustering algorithm for massive graphs and its application to nearly linear time graph partitioning. *SIAM Journal on computing* 42, 1 (2013), 1–26.

[81] Daniel A Spielman and Shang-Hua Teng. 2014. Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *SIAM J. Matrix Anal. Appl.* 35, 3 (2014), 835–885.

[82] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. 2021. Minimum Cost Flows, MDPs, and ℓ_1-Regression in Nearly Linear Time for Dense Instances. *CoRR* (2021). http://arxiv.org/abs/2101.05719

[83] Virginia Vassilevska Williams and R. Ryan Williams. 2018. Subcubic Equivalences Between Path, Matrix, and Triangle Problems. *J. ACM* 65, 5 (2018), 27:1–27:38. https://doi.org/10.1145/3186893

[84] Zhenyu Wu and Richard Leahy. 1993. An optimal graph theoretic approach to data clustering: Theory and its application to image segmentation. *IEEE transactions on pattern analysis and machine intelligence* 15, 11 (1993), 1101–1113.

[85] Huacheng Yu. 2018. An improved combinatorial algorithm for Boolean matrix multiplication. *Inf. Comput.* 261 (2018), 240–247. https://doi.org/10.1016/j.ic.2018.02.006