Cut-Equivalent Trees are Optimal for Min-Cut Queries

Amir Abboud
IBM Almaden Research Center
Almaden, U.S.A.
amir.abboud@gmail.com

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*—Min-Cut queries are fundamental: Preprocess an undirected edge-weighted graph, to quickly report a minimum-weight cut that separates a query pair of nodes s, t. The best data structure known for this problem simply builds a *cut-equivalent tree*, discovered 60 years ago by Gomory and Hu, who also showed how to construct it using $n-1$ minimum st-cut computations. Using state-of-the-art algorithms for minimum st-cut (Lee and Sidford, FOCS 2014), one can construct the tree in time $\tilde{O}(mn^{3/2})$, which is also the preprocessing time of the data structure. (Throughout, we focus on polynomially-bounded edge weights, noting that faster algorithms are known for small/unit edge weights, and use n and m for the number of nodes and edges in the graph.)**

Our main result shows the following equivalence: Cut-equivalent trees can be constructed in near-linear time if and only if there is a data structure for Min-Cut queries with near-linear preprocessing time and polylogarithmic (amortized) query time, and even if the queries are restricted to a fixed source. That is, equivalent trees are an essentially optimal solution for Min-Cut queries. This equivalence holds even for every minor-closed family of graphs, such as bounded-treewidth graphs, for which a two-decade old data structure (Arikati, Chaudhuri, and Zaroliagis, J. Algorithms 1998) implies the first near-linear time construction of cut-equivalent trees.

Moreover, unlike all previous techniques for constructing cut-equivalent trees, ours is robust to relying on *approximation algorithms*. In particular, using the almost-linear time algorithm for $(1+\varepsilon)$-approximate minimum st-cut (Kelner, Lee, Orecchia, and Sidford, SODA 2014), we can construct a $(1+\varepsilon)$-approximate flow-equivalent tree (which is a slightly weaker notion) in time $n^{2+o(1)}$. This leads to the first $(1+\varepsilon)$-approximation for All-Pairs Max-Flow that runs in time $n^{2+o(1)}$, and matches the output size almost-optimally.

***Keywords*-Gomory-Hu; all-pairs max-flow; cut-equivalent tree; flow-equivalent tree; ultrametrics.**

I. Introduction

Minimum st-cut queries, or Min-Cut queries for short, are ubiquitous: Given a pair of nodes s, t in a graph G we ask for the minimum cut that separates them. Countless papers study their algorithmic complexity from various angles and in multiple contexts. Unless stated otherwise, we are in the standard setting of an undirected graph $G = (V, E, c)$ with $n = |V|$ nodes and $m = |E|$ weighted edges, where the weights (aka capacities) are polynomially bounded, While a *Min-Cut query* asks for the set of edges of the minimum cut, a *Max-Flow query* only asks for its weight.[1] A single Min-Cut or Max-Flow query can be answered in time $\tilde{O}(m\sqrt{n})$ [3],[2] and there is optimism among the experts that near-linear time, meaning $\tilde{O}(m)$, can be achieved.

In the *data structure* (or *online*) setting, we would like to preprocess the graph once and then quickly answer queries. There are two naive strategies for this. We can either skip the preprocessing and use an offline algorithm for each query, making the query time at least $\Omega(m)$. Or we can precompute the answers to all possible $O(n^2)$ queries, making the query time $O(1)$, at the cost of increasing the time and space complexity to $\Omega(n^3)$ or worse.

Half a century ago, Gomory and Hu gave a remarkable solution [4]. By using an algorithm for a single Min-Cut query $n-1$ times, they can compute a *cut-equivalent tree* (aka Gomory-Hu tree) of the original graph G. This is a tree on the same set of nodes as G, with the strong property that for every pair of nodes $s, t \in V$, their minimum cut in the tree is also their minimum cut in the graph.[3] This essentially reduces the problem from arbitrary graphs to trees, for which queries are much easier — the minimum st-cut is attained by cutting a single edge, the edge of minimum weight along the unique st-path, which can be reported in logarithmic time.[4] Cut-equivalent trees have other attractive properties beyond making queries faster, as they also provide a deep structural understanding of the graph by compressing all its minimum cut information into $O(n)$ machine words, and in particular they give a data structure which is space-optimal, as $\Omega(n)$ words are clearly necessary. Let us clarify that a cut-equivalent tree guarantees that for all $s, t \in V$, every edge e_{st} that has minimum weight along the tree's unique st-path, not only has the same weight as a minimum st-cut in G, but this edge also bipartitions the nodes into $V = S \sqcup T$ (the two connected components when e_{st} is removed from the tree), such that (S, T) is a minimum cut in the graph

[1]This terminology is common in the literature, although some recent papers [1], [2] use other names.

[2]The notation $\tilde{O}(\cdot)$ hides $\operatorname{poly}\log n$ factors (and also $\operatorname{poly}\log U$ factors in our case of $U = \operatorname{poly}(n)$).

[3]If G has a unique minimum st-cut then the reverse direction clearly holds as well.

[4]This immediately answers Max-Flow queries in logarithmic time. For Min-Cut queries extra work is required to output the edges in amortized logarithmic time; one simple way for doing it is shown in the full version.

2575-8454/20/$31.00 ©2020 IEEE
DOI 10.1109/FOCS46700.2020.00019

G. Without this additional property we would only have a weaker notion called a *flow-equivalent tree*.

Gomory and Hu's solution ticks all the boxes, except for the preprocessing time. Using current offline algorithms for each query [3], the total time for computing the tree is $\tilde{O}(mn^{3/2})$, and no matter how much the offline upper bound is improved, this strategy has a barrier of $\Omega(mn)$. While this barrier was not attained (let alone broken) for general inputs, there has been substantial progress on special cases of the problem. If the largest weight U is small, one can use offline algorithms [5], [6] that run in time $\tilde{O}(\min\{m^{10/7}U^{1/7}, m^{11/8}U^{1/4}\})$ to get even closer to the barrier. In the unweighted case (i.e., unit-capacity $U = 1$), Bhalgat, Hariharan, Kavitha, and Panigrahi [7] (see also [8]) achieved the bound $\tilde{O}(mn)$ without relying on a fast offline algorithm, and this barrier was partially broken recently with a time bound of $\tilde{O}(m^{3/2}n^{1/6})$ [9]. Near-linear time algorithms were successfully designed for planar graphs [1] and surface-embedded graphs [2]. See also [10] for an experimental study, and the Encyclopedia of Algorithms [11] for more background.

Meanwhile, on the hardness side, the only related lower bounds are for the online problem in the harder settings of directed graphs [12], [13], [14] or undirected graphs with node weights [9], where Gomory-Hu trees cannot even exist, because the $\Omega(n^2)$ minimum cuts might all be different [15]. However, no nontrivial lower bound, i.e., of time $\Omega(m^{1+\varepsilon})$, is known for computing cut-equivalent trees, and there is even a barrier for proving such a lower bound under the popular Strong Exponential-Time Hypothesis (SETH) at least in the case of unweighted graphs, due to the existence of a near-linear time *nondeterministic* algorithm [9]. Thus, the following central question remains open.

Open Question 1. *Can one compute a cut-equivalent tree of a graph in near-linear time?*

A seemingly easier question is to design a data structure with near-linear time preprocessing that can answer queries in near-constant (which means $\tilde{O}(1)$, i.e., polylogarithmic) time. We should clarify that we are interested in near-constant *amortized* time; that is, if the output minimum st-cut has $k_{s,t}$ edges then it is reported in time $\tilde{O}(k_{s,t})$. Building cut-equivalent trees is one approach, but since they are so structured they might be limiting the space of algorithms severely.

Open Question 2. *Can one preprocess a graph in near-linear time to answer Min-Cut queries in near-constant amortized time?*

An even simpler question is the *single-source* version, where the data structure answers only queries $s, t \in V$ where s is a fixed source (i.e., known at preprocessing stage) and t can be any target node. This restriction seems substantial, as the number of possible queries goes down from $O(n^2)$ to $O(n)$, and in several contexts the known single-source algorithms are much faster than the all-pairs ones. One such context is shortest-path queries, where single-source is solved in near-linear time via Dijkstra's algorithm, while the all-pairs problem is conjectured to be cubic. Another context is Max-Flow queries in *directed* graphs(digraphs), where single-source is trivially solved by $n-1$ applications of Max-Flow, while based on some conjectures, all-pairs requires at least $\Omega(n^{3/2})$ such applications [13], [14]. Single-source Max-Flow queries is currently faster than all-pairs also in the special case of unit-capacity DAGs [16]. However, this is still open for undirected Min-Cut queries.

Open Question 3. *Can one preprocess a graph in near-linear time to answer Min-Cut queries from a single source s to any target $t \in V$ in near-constant amortized time?*

It is natural to suspect that each of these questions is strictly easier than the preceding one. The case of bounded-treewidth graphs gives one point of evidence since a positive solution to Question 2 (and thus 3) was found over two decades ago [17], but Question 1 remained open to this day.

A. Our Results

Our first main contribution is to prove that all three open questions above are *equivalent*. We can extract a cut-equivalent tree from any data structure, even if it only answers single-source queries, without increasing the construction time by more than logarithmic factors. Thus, the appealingly simple trees are near-optimal as data structures for Min-Cut queries in all efficiency parameters; we find this conclusion quite remarkable.

Informal Theorem 1. *Cut-equivalent trees can be constructed in near-linear time* ***if and only if*** *there is a data structure with near-linear time preprocessing and $\tilde{O}(1)$ amortized time for Min-Cut queries,* ***and even if*** *the queries are restricted to a fixed source.*

The main new link that we establish in this paper is to reduce Question 1 to Question 3, by essentially designing an entirely new algorithm for constructing cut-equivalent trees. The precise statement is given in Theorem III.1. The two other links required for the equivalence are from Question 3 to Question 2, which holds by definition, and from Question 2 to Question 1. The latter link is to be expected, and was shown before in specific settings; for completeness, we give a simple proof via 2D range-reporting in the full version. Thus, we get the reduction from all-pairs to single-source indirectly by going through the trees, and we are not aware of another way to prove this counter-intuitive link.

Notably, our result holds not only for general graphs but also for every graph family closed under minors. It is particularly useful for bounded-treewidth graphs, for which the two-decades-old results of Arikati, Chaudhuri, and Zaroliagis [17] now imply the construction of a cut-equivalent tree in near-linear time, as stated below. We do not see an alternative way to compute a cut-equivalent tree, e.g., using directly the techniques of [17], where parts of the graph G are replaced by constant-size mimicking networks [18].

Corollary I.1 (see Corollary III.2)**.** *A cut-equivalent tree for a bounded-treewidth graph G can be constructed in randomized time $\tilde{O}(m)$.*

In planar graphs, combining our reduction with the single-source algorithm of [19] gives an alternative to the all-pairs algorithm of [1] that used a very different technique.[5]

To evaluate our results, consider how much other existing techniques for constructing cut-equivalent trees would benefit from a (hypothetical) data structure for Min-Cut queries. The classical Gomory-Hu algorithm would have two main issues. First, it modifies the graph (merging some nodes) after each Min-Cut query, hence preprocessing a single graph (or a few ones) cannot answer all the $n-1$ queries. This issue was alleviated by Gusfield [20], who modified the Gomory-Hu algorithm so that all the $n-1$ queries are made on the original graph G. A second issue is that the answer to each query might have $\Omega(m)$ edges, hence the total time $\Omega(mn)$ would far exceed $\tilde{O}(m)$. Optimistically, a more careful analysis could give an upper bound of $O(\phi)$, where ϕ is the total number of edges (in the original graph) in the $n-1$ cuts corresponding to the final tree's edges. Clearly, any such algorithm that does not merge edges must take $\Omega(\phi)$ time. Still, in weighted graphs ϕ could be $\Omega(mn)$, and even bounded-treewidth graphs could have $\phi = \Omega(n^2)$ even though $m = O(n)$ (e.g., a path with an extra node connected to all others). Therefore, our approach, which is very different from Gusfield's, shaves a factor of n. Notably, our result does not apply if the data structure is available only for unweighted graphs, because we need to perturb the edge weights to make all minimum cuts unique; but in this unweighted setting $\phi = O(m)$ [7, Lemma 5], hence it is plausible that other techniques, e.g. [20], [8], would be capable of showing the equivalence.

It is worth mentioning in this context a somewhat restricted form of the equivalence in unweighted graphs. In this case, the known $\tilde{O}(mn)$ time algorithm [7] for constructing a cut-equivalent tree actually runs in time $\tilde{O}(\phi \cdot c)$ where $c = \max_{u,v\in V} \textsf{Max-Flow}(u,v)$ is at most n in unweighted graphs, utilizes a tree-packing approach [21], [22] to find *minimal* Min-Cuts between a single source and multiple targets, meaning that the side not containing the source is minimal with respect to containment. Their method crucially relies on this minimality property to bypass the well-known barrier of uncrossing multiple cuts found in the same graph (which could be an auxiliary graph or the input G). This tree-packing approach is the basis of a few algorithms for cut-equivalent trees [23], [24], [9], and it does not seem useful for weighted graphs.

While the equivalence for flows is incomparable to that for cuts, our techniques are robust enough to prove it. In particular, we show that $\tilde{O}(n)$ Max-Flow queries are sufficient to construct a flow-equivalent tree. Currently, this relaxation (flow-equivalent instead of cut-equivalent tree) is not known to make the problem easier in any setting, although Max-Flow queries could potentially be computed faster than Min-Cut queries. Our proof follows from a lemma that an n-point ultrametric can be reconstructed from $\tilde{O}(n)$ distance queries, under the assumption that it contains at least (and thus exactly) $n-1$ distinct distances (a discussion is deferred to the full version). Interestingly, it is easy to show that without this extra assumption, $\Omega(n^2)$ queries are needed. To our knowledge, this is the first efficient construction of flow-equivalent trees only from Max-Flow queries (without looking at the cuts themselves). A well-known non-efficient construction (see [4]) is to make Max-Flow queries for all $O(n^2)$ pairs, view it as a complete graph with edge weights, and take a maximum-weight spanning tree.

Informal Theorem 2. *Flow-equivalent trees can be constructed in near-linear time **if and only if** there is a data structure with near-linear time preprocessing and $\tilde{O}(1)$ time for Max-Flow queries.*

$(1+\varepsilon)$*-Approximations:* Our first result offers a quantitative improvement over the Gomory-Hu reduction from cut-equivalent trees to Min-Cut queries. It turns out that our technique also gives a qualitative improvement. A well-known open question among the experts, see e.g. [11], is to utilize *approximate* Min-Cut queries (to construct an approximate cut-equivalent tree). An obvious candidate is an algorithm of Kelner et al. [25] for the offline setting (i.e., a single query), that achieves $(1+\varepsilon)$-approximation and runs in near-linear time. It beats the time-bound of all known exact algorithms, however no one has managed to utilize it for the online setting, or for constructing equivalent trees. It is not difficult to come up with counter-examples (see Section II-A) that show that following the Gomory-Hu algorithm but using at each iteration a $(1+\varepsilon)$-approximate (instead of exact) minimum cut, results with a tree whose quality (approximation of the graph's cut values) is arbitrarily large. Our second main contribution is an efficient reduction from *approximate* equivalent trees to *approximate* Min-Cut queries. Previously, no such reductions were known (the aforementioned maximum-weight spanning tree would again give a non-efficient solution).

Informal Theorem 3 (see Theorem II.1)**.** *Assume there is an oracle that can answer Min-Cut queries within $(1+\varepsilon)$-approximation. Then one can compute, using $\tilde{O}(n)$ queries*

[5]The conference paper of [1] appeared in FOCS 2010, before [19] appeared in FOCS 2012. While the latter solves an easier task (single-source), it does so for the harder setting of *directed* planar graphs.

to the oracle and an additional processing in time $\tilde{O}(n^2)$:

1) *a $(1+\varepsilon)$-approximate flow-equivalent tree; and*
2) *a tree-like data structure that stores $\tilde{O}(n)$ cuts and can answer a Min-Cut query in time $\tilde{O}(1)$ and with approximation $1+\varepsilon$ by reporting (a pointer to) one of these stored cuts.*

For unweighted graphs, we can improve the $\tilde{O}(n^2)$ term to $\tilde{O}(m)$ which could be significant. While it may not be obvious why our new data structure is better than the oracle we start with, there are a few benefits (see Section II-B). Most importantly, since it only uses $\tilde{O}(n)$ queries, we can combine our reduction with the algorithm of Kelner et al. [25] (even though it is for the offline problem, we essentially plug it into our reduction), and obtain three new approximate algorithms that are faster than state-of-the-art exact algorithms! We discuss these results next.

Corollary I.2 (Section II-B)**.** *Given a capacitated graph G on n nodes, one can construct a $(1+\varepsilon)$-approximate flow equivalent tree of G in randomized time $\varepsilon^{-4} \cdot n^{2+o(1)}$.*

It follows that the All-Pairs Max-Flow problem in undirected graphs can be solved within $(1+\varepsilon)$-approximation in time $n^{2+o(1)}$, which is optimal up to sub-polynomial factors since the output size is $\Omega(n^2)$. This problem is also well-studied in directed graphs [26], [27], [15], [19], [16], [28], where it is known that exact solution in sub-cubic time is conditionally impossible [13], [14], but it is open for approximated solutions.

Corollary I.3 (Section II-B)**.** *Given a capacitated graph G on n nodes, one can construct in $\varepsilon^{-4} \cdot n^{2+o(1)}$ randomized time, a data structure of size $\tilde{O}(n^2)$, that stores a set $\mathcal{C}$ of $\tilde{O}(n)$ cuts, and can answer a Min-Cut query in time $\tilde{O}(1)$ and with approximation $1+\varepsilon$ by reporting a cut from $\mathcal{C}$.*

Altogether, we provide for all three problems above (flow-equivalent tree, All-Pairs Max-Flow, and data structure for Max-Flow) randomized algorithms that run in time $n^{2+o(1)}$. Previously, the best approximation algorithm known for these three problems was to sparsify G into $m' = \tilde{O}(\varepsilon^{-2}n)$ edges in randomized time $\tilde{O}(m)$ using [29] (or its generalizations), and then execute on the sparsifier the Gomory-Hu algorithm, which takes time $\tilde{O}(n \cdot m'\sqrt{n}) = \tilde{O}(\varepsilon^{-2}n^{2.5})$. The best exact algorithms previously known for these problems was essentially to compute a cut-equivalent tree runs in time $O(mn^{1.5})$. An alternative way to approximate Max-Flow queries without the Gomory-Hu algorithm is to use Räcke's approach of a cut-sparsifier tree [30]. This is a much stronger requirement (it approximates all cuts of G) and can only give polylogarithmic approximation factors. Its fastest version runs in near-linear time $m^{1+o(1)}$ and achieves approximation factor $O(\log^4 n)$ [31].

Unfortunately, we could not prove the same results for $(1+\varepsilon)$-*cut*-equivalent trees and more new ideas are required; in Section II-A we show an example where our approach fails. Interestingly, this is the first setting where we see different time bounds showing that the extra requirements of cuts indeed make the equivalent trees harder to construct.

Besides the inherent interest in the equivalence result and its applications, we believe that our results make progress towards the longstanding goal of designing optimal algorithms for cut-equivalent trees. It is likely that such algorithms will be achieved via a fast algorithm for online queries, as was the case for bounded-treewidth graphs.

B. Preliminaries

A *Min-Cut data structure* for a graph family $\mathcal{F}$ is a data structure that after preprocessing of a capacitated graph $G \in \mathcal{F}$ in time $t_p(m)$, can answer Min-Cut queries for any two nodes $s,t \in V$ in amortized query time (or output sensitive time) $t_{mc}(k_{st})$, where k_{st} denotes the output size (number of edges in this cut). This means that the actual query time is $O(k_{st} \cdot t_{mc}(k_{st}))$. A $(1+\varepsilon)$-approximate Min-Cut data structure is defined similarly but for $(1+\varepsilon)$-approximate minimum st-cut whose total capacity is at most $(1+\varepsilon)$ times that of the minimum st-cut in G. We denote by $\mathsf{Max\text{-}Flow}_G(s,t)$ the value of the minimum-cut between s and t, and we might omit the graph G subscript when it is clear from the context. Throughout, we restrict our attention to connected graphs and thus assume that $m \geq n-1$, and additionally we assume that the edge-capacities are integers (by scaling).

II. Our Approximation Algorithms

In this section we present the high level ideas in our approximation algorithms, where the complete details are in the full version.

A. Overview

Here we discuss the obstacles to speeding up Gomory-Hu's approach, and why plugging in *approximate* Min-Cut queries fails to produce an *approximate* cut-equivalent tree. To explain how our approach overcomes these issues, we present the key ingredients in our approximation algorithm from Section II-B. This overview also prepares the reader for Section III, which is the most complicated part of the paper and proves our main result (Theorem III.1).

Overview of the Gomory-Hu method: Start with all nodes forming one super-node V. Then, pick an arbitrary pair of nodes s,t from the super-node, find a minimum st-cut $(S, V \setminus S)$, and split the super-node into two super-nodes S and $V \setminus S$. Then connect the two new super-nodes by an edge of weight $w(S, V \setminus S)$, and recurse on each of them. In each recursive call (which we also view as an iteration), say on a super-node V', the Min-Cut query is performed on an auxiliary graph $G_{V'}$ that is obtained from G by contracting every super-node other than V'. These contractions prevent the other super-nodes from being split by the cut, which is crucial for the consistency of the constructed tree, and by a key lemma about uncrossing cuts (proved using submodularity of cuts), these contractions (viewed as imposing restrictions on the feasible cuts in $G_{V'}$) do not increase the value of the minimum st-cut. The cut found in $G_{V'}$ is then used to split V' into two new super-nodes, and every edge that was incident to V' is "rewired" to exactly one of the new super-nodes. The process stops when every super-node contains a single node, which takes exactly $n-1$ iterations and results in a tree on n super-nodes, giving us a tree on V.

Why Gomory-Hu fails when using approximations: There are two well-known issues (see [11]) for employing this approach using *approximate* (rather than exact) Min-Cut queries, even if the approximation factor is as good as $1+\varepsilon$. The first issue is that errors of this sort multiply, and thus a $(1+\varepsilon)$-factor at each iteration accumulates in the final tree to $(1+\varepsilon)^d$, where d is the depth of the recursion. The second issue is even more dramatic; without the uncrossing-cuts property, the error could increase faster than multiplying and might be unbounded even after a single iteration. The reason is that when we find in super-node V' a cut $(S, V' \setminus S)$ that is (approximately) optimal for a pair $s, t \in V'$, we essentially assume that for all pairs $s' \in S, t' \in V \setminus S$ there is an (approximately) optimal cut that splits at most one of S and $V' \setminus S$ (not both). While true for exact optimality, it completely fails in the approximate case, and there are simple examples, see e.g. Figure 1, where allowing $(1+\varepsilon)$-approximation in the very first iteration makes the error of the final tree unboundedly large. We will refer to this issue as the main issue.

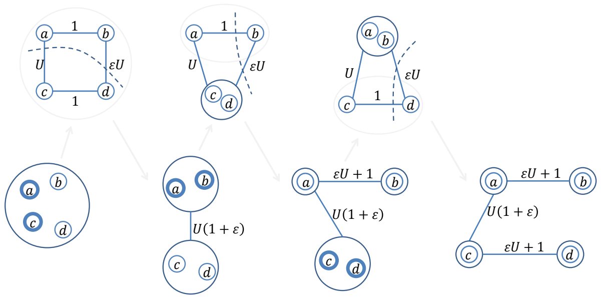

Figure 1: An example of the main issue with using $(1+\varepsilon)$-approximate minimum cuts in the Gomory-Hu algorithm. The input graph G is at the top left; the intermediate trees are at the bottom, from left to right; and the auxiliary graphs $G_{V'}$ are at the top. Each iteration uses a $(1+\varepsilon)$ Min-Cut for the node pair shown in bold. In the input graph $\mathsf{Max\text{-}Flow}(b, c) = 2$ but in the tree it is $\Omega(U)$; thus the error can be as bad as $\text{poly}(n)$.

Our strategy: Our approach is different and simultaneously resolves both issues for flow-equivalent trees; for cut-equivalent trees, as we show below, the first issue remains (but not the second).

Our main insight is to identify a property of the cut $(S, V' \setminus S)$, that is sufficient to resolve the main issue: This property is stronger than being a minimum st-cut, and requires that for all pairs $s' \in S, t' \in V' \setminus S$, this same cut is an (approximate) minimum $s't'$-cut, i.e., it works for them as well. Thus, the error for every pair s', t' from this split of V' is bounded by $(1+\varepsilon)$-factor, and we can recursively deal with pairs inside the same super-node. While this property may seem too strong, notice that it holds whenever $(S, V' \setminus S)$ is an (approximate) global minimum cut (i.e., achieves the minimum over all pairs $s', t' \in V'$). While our algorithm builds on this intuition, it does not compute a global minimum cut at each iteration, but rather employs a more complicated strategy that it is substantially more efficient. For example, its recursion depth is bounded by $O(\log n)$, which is important to bound the overall running time, and also to control the approximation factor.

Bounding the depth of the recursion: The foremost idea is that the recursion depth should be bounded by $O(\log n)$. This does not happen in the Gomory-Hu algorithm, nor in the aforementioned strategy of using an (approximate) global minimum cut, where splits could be unbalanced and recursion depth might be $\Omega(n)$. Assuming – by way of wishful thinking – that the total time spent in all recursive calls of the same level is $\tilde{O}(m)$,[6] the challenge is to dictate how to (quickly) choose cuts so that the recursion depth is small.

Instead of insisting on a balanced cut, we partition the super-node V' into *multiple* sets at once, which can be viewed as performing a batch of consecutive Gomory-Hu iterations at the cost of one iteration (up to logarithmic factors). This approach was previously used in a few other algorithmic settings, however, none of their methods is applicable in our context.[7] Before explaining how our algorithm computes a partition, let us explain which properties it needs to satisfy. A partition of super-node V' into r sets $S_1, \ldots, S_r$ (that will be processed recursively) should satisfy the following strong property:

(*) For every pair $s' \in S_i, t' \in S_j$ for $i \neq j$, at least one of $(S_i, V' \setminus S_i)$ or $(S_j, V' \setminus S_j)$ corresponds in $G_{V'}$ to a $(1+\varepsilon)$-approximate minimum $s't'$-cut.

[6]One moral justification is that super-nodes V' of the same recursion level are disjoint, as they form a partition of V. However, the real challenge is to process their auxiliary graphs $G_{V'}$. This may be possible in the special case where G is unweighted, becuase the total size (number of edges) of these auxiliary graphs (from one level) is $O(m)$ [7], [32], [8], [9], but for a general graph G the total size of these auxiliary graphs might easily exceed $\tilde{O}(m)$.

[7]This approach was used in three different algorithmic settings: (1) in the special case of an unweighted graph G [7], [32]; (2) in parallel algorithms [33], which can compute in parallel polynomially-many cuts (e.g., for all $s', t' \in V'$) to find a partition; or (3) in non-deterministic algorithms [9], which can "guess" a good partition but have to verify it quickly (achieved in [9] for an unweighted graph G).

(We will actually allow an exception of one set S_0 that does not satisfy this property, and must be handled in a special way; this is the set V''_{big} in Section II-C.) In addition, the sizes of these sets should be bounded by $|V'|/2$ (with the exception of the set S_0, which is bounded by $\frac{3}{4}|V'|$) which guarantees recursion depth $O(\log n)$, unlike a global minimum cut.

Our algorithm to partition V' picks a pivot node $p \in V'$ and queries a data structure built for $G_{V'}$ for an (approximate) minimum cut between p and every other node $u \in V'$; let $S_u \subset V'$ be the side of u in the returned cut. To form a partition out of these $|V'|-1$ sets S_u, reassign each node u to a set $S_{u'}$ that contains u, which naturally defines a partition (by grouping nodes reassigned to the same $S_{u'}$). The reassignment process is elaborate and subtle, aiming to preserve property (*) while reassigning nodes only to sets $S_{u'}$ of size at most $|V'|/2$.

Choosing effective pivots: The above technique is not sufficient for bounding the depth of the recursion, because a poorly chosen pivot p might result in many unbalanced cuts (sets S_u of size larger than $\frac{3}{4}|V'|$), in which case this pivot is ineffective. Our next idea is that for a randomly chosen pivot $p \in V'$ this will not happen with high probability.[8] We analyze the performance of a random pivot using a simple lemma about tournaments that works as follows. Assume for now that the Min-Cut data structure is deterministic (we show how to lift this assumption in the full version), then every query $\{x, y\}$ (described as an unordered pair) is answered with some cut (S_x, S_y), and obviously $|S_x| \leq |V'|/2$ or $|S_y| \leq |V'|/2$ (or both). It follows by symmetry that a query for $\{u, p\}$ has a chance of at least $1/2$ of having $|S_u| \leq |V'|/2$, in which case we say that node u is "good" (in Section II-B we call these V_{small}). But we need a stronger property, that at least $1/4$ of the nodes in V' are good in this sense; we thus define on the nodes V' a tournament, with an edge directed from x to y whenever $|S_x| \leq |S_y|$, and prove that most nodes have a large out-degree, and will thus be effective pivots.

With constant probability, such an effective pivot is chosen, hence the number of nodes that are not good is bounded by $\frac{3}{4}|V'|$, and we must handle them with a separate recursive call (this is the problematic set V''_{big} in Section II-C). A related but different issue that arises in Section III is that we cannot afford a Min-Cut query from p to all other $u \in V'$. To handle this we utilize the mentioned tournament properties by making Min-Cut queries from a random pivot p to only a small sample of targets.

Using dynamic-connectivity algorithms: Even if the recursion depth is bounded by $O(\log n)$, it is not clear how to execute the entire algorithm in near-linear time, as each iteration computes $|V'|-1$ cuts followed by a reassignment process. A straightforward implementation could require quadratic time $\Omega(n^2)$ even in the first iteration (on supernode V), which appears to be necessary because in some instances the total size of all good sets S_u (where $|S_u| \leq n/2$) is indeed $\Omega(n^2)$. For unweighted graphs, however, the total number of *edges* in these cuts (all minimum cuts from a fixed source to all targets) can be bounded by $O(m)$ (proof appears in the full version), and indeed in this case our entire algorithm can be executed in time $\tilde{O}(m)$. The key is to only spend time proportional to the number of edges in each cut, rather than to the number of nodes $|S_u|$. In unweighted graphs, and also in the "capacitated auxiliary graphs" that we construct in Section III, the total number of nodes and edges our algorithm observes is bounded by $\tilde{O}(m)$. The reassignment process poses an additional challenge. For example, can one decide whether $u \in S_{u'}$ in time that is proportional to the number of edges (rather than nodes) in the cut $S_{u'}$ (more precisely, the reported cut between p and u' in $G_{V'}$)? Our solution utilizes an efficient dynamic-connectivity algorithm (we use a simple modification of [34]), that preprocesses a graph in near-linear time, and support edge updates and connectivity queries in polylogarithmic time — we simply delete the edges of the cut S_v and then ask if u and u' are connected.

B. Approximate Min-Cut Queries and Flow-Equivalent Trees

In this section we expand on our results for using *approximate* Min-Cut queries that were presented in Section I and a technical overview for them was given in Section II-A, where the details are in deferred to the full version.

The following theorems formalize Informal Theorem 3 and give Corollaries I.2 and I.3 from Section I.

Theorem II.1. *There is a randomized algorithm such that given a capacitated graph $G = (V, E, c)$ on n nodes, m edges, and using $\tilde{O}(n)$ queries to a deterministic $(1+\varepsilon)$-approximate Min-Cut data structure for G with a running time t_p and amortized time t_{mc}, can with high probability:*

- *construct in time $O(t_p(n)) + \tilde{O}(n^2)$ a $(1+\varepsilon)$-approximate flow-equivalent tree T of G, and*
- *construct in time $O(t_p(n)) + \tilde{O}(n^2)$ a data structure D of size $\tilde{O}(n^2)$ that stores a set $\mathcal{C}$ of $\tilde{O}(n)$ cuts, such that given a queried pair $s, t \in V$ returns in time $\tilde{O}(1)$ a pointer to a cut in $\mathcal{C}$ that is a $(1+\varepsilon)$-approximate minimum st-cut.*

While the significance of the first item of the theorem is clear (the flow-equivalent tree) let us say a few words about why the second item is interesting compared to the assumption. The first benefit of our data structure is that it only stores $\tilde{O}(n)$ cuts and therefore it will only have $\tilde{O}(n)$ different answers to the $\binom{n}{2}$ possible queries it can receive. This makes it more similar to a cut-equivalent tree. Second,

[8]A random pivot was previously used in [32] in the special case of an unweighted graph G, and their proof relies heavily on this restriction. Moreover, the cuts S_u in their algorithm form a laminar family, hence their reassignment process is straightforward.

the space complexity of our data structure is upper bounded by $\tilde{O}(n^2)$ in weighted or $\tilde{O}(m)$ in unweighted graphs, while the oracle could have used larger space; thus we could save space without incurring loss to the preprocessing and query times by more than log factors. The third benefit is that it only uses $\tilde{O}(n)$ queries to the assumed oracle, which allows us to obtain consequences even from an oracle with larger query times and even from *offline* algorithms. If rather than a $(1+\varepsilon)$ Min-Cut data structure we have an offline $(1+\varepsilon)$-approximate minimum st-cut algorithm such as [25], by simply computing it every time there is a query, we get the following theorem.

Theorem II.2. *If in Theorem II.1 instead of a* $(1+\varepsilon)$*-approximate Min-Cut data structure we have an offline* $(1+\varepsilon)$*-approximation algorithm with running time* $t_{\mathit{offline}}(m)$*, the time bounds for constructing* P *and* D *become* $\tilde{O}(n \cdot t_{\mathit{offline}}(n))$*.*

We also remark that the above theorems only deal with deterministic data structures and algorithms. The reason will be clarified during the proof. However, this restriction can be removed, and due to space constraints, we expand on that in the full version.

To conclude Corollaries I.2 and I.3 from Section I, given a graph we begin by applying a sparsification due to Benczur and Karger [35], where a near-linear-time construction transforms any graph on n nodes into an $O(n \log n/\varepsilon^2)$-edge graph on the same set of nodes whose cuts $(1+\varepsilon)$-approximate the values in the original graph. This incurs a $(1+\varepsilon)$ approximation factor to the result. By utilizing a $(1+\varepsilon)$-approximate minimum st-cut algorithm for general capacities by [25] with $t_{\text{offline}}(m) = m^{1+o(1)}/\varepsilon^2$ we get the $n^{2+o(1)}/\varepsilon^4$ upper bound for constructing $(1+\varepsilon)$-approximate flow-equivalent trees and the tree-like data structure. The main previously known method for constructing a data structure that can answer $(1+\varepsilon)$-approximate minimum st-cuts is to construct an exact cut equivalent tree of a sparsification of the input graph using, e.g., Benczur-Karger [29]. For general capacities, this gives a total running time of $\tilde{O}(n^{5/2})$. For unit-capacities, since this sparsification introduces edge weights, it is not clear how to do anything better for the approximation version than the exact bounds.

In the unit-capacity case, using the same techniques as in Theorem II.1 (but with extra care), our bounds are better: we replace the $\tilde{O}(n^2)$ term with $\tilde{O}(m)$. While we do not currently have an application for this improved bound, it will be significant in the likely event that a $(1+\varepsilon)$-approximate Min-Cut data structure can be designed for sparse unweighted graphs that will have near-linear or even $O(n^{1.5-\delta})$ preprocessing time. Then, our improved theorem would give an approximate flow-equivalent tree construction that improves on the $n^{1.5}$ barrier that currently exists for exact [9]. We remark that, since the results of this section do not use any edge contractions and only ask queries about the original graph, they hold for *any* graph family even if it is not minor-closed. This is important since the family of sparse graphs is not minor closed. This is further discussed in the full version.

C. *Our Tree-Like Data Structure*

Here, we give a brief description of the proof of the second item in Theorem II.1.

Let G be the input graph with node set V, we show how to construct a data structure D that utilizes a tree structure T, and we will also construct a graph H which we will call *flow-emulator* on the same node set V that will only be used for our flow-equivalent tree construction. We assume we are given an arbitrary data structure for answering $(1+\varepsilon)$-approximate Min-Cut queries, and give a new data structure or flow-equivalent tree with error $(1+\varepsilon)^2$. Thus, to get the theorem we could use a data structure with parameter $\varepsilon' = \varepsilon/3$.

Preprocessing: To construct our data structure we recursively perform *expansion* operations. Each such operation takes a subset $V' \subset V$ and partitions it into a few sets $S_i \subseteq V'$ on which the operation will be applied recursively until they have size 1 (V' can be thought of as a super-node as in Gomory-Hu but here we do not have auxiliary graphs and contractions). The partition S_i will (almost) satisfy the strong property (*) that we discussed in Section II-A. In the beginning we apply the expansion on $V' := V$. It will be helpful to maintain the recursion-tree T that has a node $t_{V'}$ for each expansion operation that stores V' as well as some auxiliary information such as cuts and a mapping from each node $v \in V'$ to a cut $S_{f(v)}$. To perform a query on a pair u, v we will go to the recursion-node in T that separated them, i.e. the last V' that contains both of them, and we will return one of the cuts stored in that node.

We will prove that, because of how we build the partition, the depth of the recursion will be $O(\log n)$. For each level of the recursion, the expansion operations are performed on disjoint subsets V_i'. All the work that goes into the expansion operations in one level can be done in $O(n^2)$ time in a straightforward way. In unweighted graphs, it can even be done in $\tilde{O}(m)$ time by adapting known dynamic connectivity algorithms; the full details are deferred to the full version.

III. Algorithm for a Cut-Equivalent Tree

In this section we show a new algorithm for constructing a cut-equivalent tree for graphs from a minor-closed family $\mathcal{F}$ (for example all graphs), given a Min-Cut data structure for this family $\mathcal{F}$. For ease of exposition, we first assume that the data structure supports also Max-Flow queries (reporting the value of the cut) in time $t_{mf}(m)$; we will later show that Min-Cut queries suffice.

Theorem III.1. *Given a capacitated graph $G \in \mathcal{F}$ on n nodes and m edges, and access to a deterministic Min-Cut data structure for $\mathcal{F}$ with preprocessing time $t_p(\cdot)$ and output sensitive time $t_{mc}(\cdot)$, one can construct, with high probability, a cut-equivalent tree for G in time $\tilde{O}(t_p(m) + m \cdot t_{mc}(m))$. Furthermore, it suffices that the data structure's queries are restricted to a fixed source.*

By combining our algorithm with the Min-Cut data structure of Arikati, Chaudhuri, and Zaroliagis [17] for graphs with treewidth bounded by (a parameter) t, which attains $t_p = n \log n \cdot 2^{2^{O(t)}}$ and $t_{mc} = t_{mf} = 2^{2^{O(t)}}$, we immediately get the first near-linear time construction of a cut-equivalent tree for graphs with bounded treewidth, as follows.

Corollary III.2 (Expanded Corollary I.1). *Given a graph G with n nodes and treewidth at most t, one can construct, with high probability, a cut-equivalent tree for G in time $\tilde{O}(2^{2^{O(t)}} n)$.*

The rest of this section is devoted to proving Theorem III.1. Our analysis relies on the classical Gomory-Hu algorithm [4], hence we start by briefly reviewing it (largely following [9]) with a bit more details than in Section II-A.

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 tree; each tree node i is a *super-node*, which means that it is associated with a subset $V_i \subseteq V$; and these super-nodes form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. Each iteration works as follows: pick arbitrarily two graph nodes s, t that lie in the same tree super-node i, i.e., $s \neq t \in V_i$, then construct from G an auxiliary graph G' by merging nodes that lie in the same connected component of $\mathcal{T} \setminus \{i\}$, and invoke a Max-Flow algorithm to compute in G' a minimum st-cut, denoted C'. (For example, if the current tree is a path on super-nodes $1, \ldots, l$, then G' is obtained from G by merging $V_1 \cup \cdots \cup V_{i-1}$ into one node and $V_{i+1} \cup \cdots \cup V_l$ into another node.) The submodularity of cuts ensures that this cut is also a minimum st-cut in the original graph G, and it clearly induces a 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 reconnecting each j which was a neighbor of i in $\mathcal{T}$ to either super-node S or T, depending on which side of the minimum st-cut C' contains V_j.

The algorithm performs these iterations until all super-nodes are singletons, and then $\mathcal{T}$ is a weighted tree with effectively the same node set as G. It is proved in [4] that for every $s, t \in V$, the minimum st-cut in $\mathcal{T}$, viewed as a bipartition of V, is also a minimum st-cut in G, and of the same cut value. We stress that this property holds regardless of the choices, made at each iteration, of two nodes $s \neq t \in V_i$.

A. The Algorithm for General Capacities

We turn out attention to proving Theorem III.1. Let $G = (V, E, c)$ be the input graph. We shall make the following assumption, justified by a standard random-perturbation argument.

Assumption III.3. *The input graph G has a single cut-equivalent tree $\mathcal{T}^*$, with $n-1$ distinct edge weights.*[9]

B. Overview of the Algorithm

At a very high level, our algorithm accelerates the Gomory-Hu algorithm by performing every time a batch of Gomory-Hu steps instead of only one step. Similarly to the actual Gomory-Hu algorithm, our algorithm is iterative and maintains a tree $\mathcal{T}$ of super-nodes, which means that every tree node i is associated with $V_i \subseteq V$, and these super-nodes form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. This tree $\mathcal{T}$ is initialized to have a single super-node corresponding to V, and since it is modified iteratively, we shall call $\mathcal{T}$ the *intermediate tree*. Eventually, every super-node is a singleton and the tree $\mathcal{T}$ corresponds to $\mathcal{T}^*$.

In a true Gomory-Hu execution, every iteration partitions some super-node i into exactly two super-nodes, say $V_i = S \sqcup T$, which are connected by an edge according to the minimum cut between a pair $s \in S, t \in T$ that is computed in an auxiliary graph. In contrast, our algorithm partitions a super-node i into multiple super-nodes, say $V_i = U_p \sqcup V_{i,1} \sqcup \cdots \sqcup V_{i,d}$, that are connected in a tree topology where the last edge in the path from U_p to each $V_{i,j}$, $j \in [d]$, is set according to the minimum cut between a pivot $p \in U_p$ and a corresponding $u_{i,j} \in V_{i,j}$, where all these cuts are computed in the same auxiliary graph. We call this an *expansion step* and super-node U_p is called the *expansion center*; see Figure 2 for illustration. Each iteration of our algorithm applies such an expansion step to every super-node in the intermediate tree $\mathcal{T}$. These iterations can also be viewed as recursion, and thus each expansion step occurs at a certain recursion depth, which will be bounded by our construction.

To prove that our algorithm is correct, we will show that every expansion step corresponds to a valid sequence of Gomory-Hu steps. Just like in the Gomory-Hu algorithm, our algorithm relies on minimum-cut computations in auxiliary graphs, although it will make multiple queries on the same auxiliary graph. This alone does not guarantee overall running time $\tilde{O}(m)$, because in some scenarios the total size of all auxiliary graphs at a single depth is much bigger than m. For example, if $\mathcal{T}^*$ consists of two stars of size $n/3$

[9]Even though the perturbation algorithm is Monte Carlo, our algorithm can still be made Las Vegas since if a random perturbation fails Assumption III.3, then our algorithm could encounter two crossing cuts, but it can identify this situation and restart the algorithm with another perturbation.

connected by a path of length $n/3$, and G is similar but has in addition all possible edges between the stars (with low weight), the total size of all auxiliary graphs would be $\Omega(n^3)$. We overcome this obstacle using a *capacitated auxiliary graph* (CAG), which is the same auxiliary graph as in the Gomory-Hu algorithm, but with parallel edges merged into a single edge with their total capacity. We will show (in Lemma III.13) that the total size of all CAGs at a single depth is linear in m.

Another challenge is to bound the recursion depth by $O(\log n)$. A partition in the Gomory-Hu algorithm might be unbalanced, where in our algorithm, this issue comes into play by a poor choice of a pivot; for example, in a star graph with edge-capacities $1, \dots, n-1$, if the pivot p is the leaf incident to the edge of capacity 1, then the minimum cut between p and any other node is the same $(\{p\}, V \setminus \{p\})$, giving little information on how to partition V and make significant progress. Observe however that a random pivot would work much better in this example; more precisely, a set of $O(\log n)$ random pivots contains, with high probability, at least one pivot p for which the minimum cuts between p and each of the other nodes will partition V into super-nodes that are all constant-factor smaller, thus our expansion step will decrease the super-node size by a constant factor. But notice that even if a pivot p is given, we still need to bound the time it takes to partition the super-node. Our algorithm repeatedly computes a minimum cut between p and some other node, such that the time spent on computing this minimum cut is proportional to its progress in reducing $|V_i|$, until $\Omega(|V_i|)$ nodes are separated away from V_i. Altogether, all these minimum cuts (from a single pivot p) take time that is near-linear in the size of the corresponding CAG. It will then follow that the total time of all expansion steps at a single depth is near-linear in the total size of their CAGs, which as mentioned above is linear in m, and finally since the depth is $O(\log n)$, the overall time bound is $\tilde{O}(m)$.

C. Full Algorithm

To better illustrate our main ideas, we now present our algorithm with a slight technical simplification of employing both Min-Cut and Max-Flow queries. After analyzing its correctness and running time in Section III-D, we will show (in the full version) that Max-Flow queries are not necessary.

The algorithm initializes $\mathcal{T}$ as a single super-node associated with the entire node set V, and ends when all super-nodes in $\mathcal{T}$ are singletons, supposedly corresponding to the cut-equivalent tree $\mathcal{T}^*$. At every recursion depth in between, the algorithm performs an expansion step in every non-singleton super-node. The expansion of super-node $i \in \mathcal{T}$ of size $n_i = |V_i| \geq 2$, whose CAG is denoted G_i, works as follows. Pick a pivot node $p \in V_i$ uniformly at random, and for every node $u \in V_i \setminus \{p\}$ let $(S_u, V(G_i) \setminus S_u)$ be the minimum up-cut in G_i, and let $S'_u = V_i \cap S_u$. In order to compute $|S'_u|$, create in a preprocessing step a copy $\tilde{G}_i$ of G_i, and assuming its edge-capacities are integers (by scaling), connect (in $\tilde{G}_i$) the pivot p to all other nodes $u \in V_i \setminus \{p\}$ by new edges of small capacity $\delta = 1/n^3$. Note that $\tilde{G}$ depends on p but not on u, hence it is preprocessed once per pivot p then used for multiple nodes u. Then for every node $u \in V_i \setminus \{p\}$ compute

$$h_p(u) := [\mathsf{Max\text{-}Flow}_{\tilde{G}_i}(u,p) - \mathsf{Max\text{-}Flow}_{G_i}(u,p)]/\delta,$$

which clearly satisfies $h_p(u) = |S'_u|$, and then compute the set

$$V_i^{\leq 1/2}(p) := \{u \in V_i \setminus \{p\} : h_p(u) \leq n_i/2\}.$$

Now repeat picking random pivots until finding a pivot p for which $|V_i^{\leq 1/2}(p)| \geq n_i/4$.

Next, initialize $U_p := V_i$, pick uniformly at random a node $u \in U_p \cap V_i^{\leq 1/2}(p)$, and enumerate the edges in the cut $(S_u, V(G_i) \setminus S_u)$. Partition U_p into two super-nodes, $U_p \cap S_u$ and $U_p \setminus S_u$, connected by an edge of capacity $\mathsf{Max\text{-}Flow}(u,p)$, then reconnect every edge previously connected to U_p in $\mathcal{T}$ to either $U_p \cap S_u$ or $U_p \setminus S_u$ according to the cut $(V(G_i) \setminus S_u, S_u)$. Repeat the above, i.e., pick another node $u \in U_p \cap V_i^{\leq 1/2}(p)$ and so forth, as long as $|U_p| > 7n_i/8$ (we shall prove that such a node u always exists), calling these nodes $u_1, \dots, u_d$ in the order they are picked by the algorithm; when $|U_p| \leq 7n_i/8$ is reached, conclude the current expansion step.

Recall that the algorithm performs such an expansion step to every non-singleton super-node (i.e., $n_i \geq 2$) at the current depth, and only then proceeds to the next depth. The base case $n_i = 1$ can be viewed as returning a trivial tree on V_i.

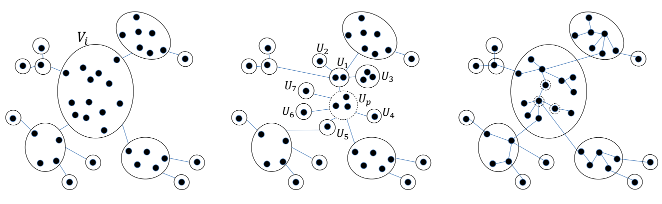

Figure 2: The changes to $\mathcal{T}$ by our algorithm. Left: before expansion step of V_i. Middle: after expansion step with expansion center U_p (dashed), and the subtree of $\mathcal{T}$ corresponds to partition $V_i = \bigsqcup_{j=1}^{7} U_j \sqcup U_p$. Right: when the algorithm terminates.

D. Analysis

We start with a lemma, whose proof is deferred to the full version, showing that whenever our algorithm reports a tree, there exists a Gomory-Hu execution that produces the same tree. Notice that super-nodes at the same depth are disjoint, hence an expansion of one of them does not affect the other super-nodes, and the result of these expansion steps is the same regardless of whether they are executed in parallel or sequentially in any order.

Lemma III.4 (Simulation by Gomory-Hu Steps)**.** *Suppose there is a sequence of Gomory-Hu steps producing tree $\mathcal{T}^{(j)}$, and that an expansion step performed to $V_i \in \mathcal{T}^{(j)}$ produces $\mathcal{T}^{(j+1)}$. Then there is a sequence of Gomory-Hu steps that simulates also this expansion step and produces $\mathcal{T}^{(j+1)}$.*

The next corollary follows from Lemma III.4 immediately by induction.

Corollary III.5. *There is a Gomory-Hu execution that outputs the same tree as our algorithm, which by the correctness of the Gomory-Hu algorithm and Assumption III.3, is the cut-equivalent tree $\mathcal{T}^*$.*

We proceed to prove the time bound stated in Theorem III.1. Our strategy is to bound the running time of a single expansion step in proportion to the size of the corresponding CAG, and then bound the total size, as well as the construction time, of all CAGs at a single depth of the recursion. Finally, we will bound the recursion depth by $O(\log n)$, to conclude the overall time bound stated in Theorem III.1.

Lemma III.6. *Assuming $t_p(m) = \tilde{O}(m)$ and $t_{mc}(m) = \tilde{O}(1)$, the (randomized) running time of a single expansion step on V_i, including constructing the children CAGs, and preprocessing it for queries, is near-linear in the size of G_i with probability at least $1 - 1/n^3$.*

Proof: We start with bounding the number of pivot choices. To do that, we use the following general corollary, whose proof is deferred to the full version, about cuts between every pair of nodes.

Corollary III.7. *Let $F = (V_F, E_F)$ be a graph where each pair of nodes $u, v \in V_F$ is associated with a cut $(S_{uv}, S_{vu} = V_F \setminus S_{uv})$ where $u \in S_{uv}, v \in S_{vu}$ (possibly more than one pair of nodes are associated with each cut), and let $V'_F \subseteq V_F$. Then there exist $|V'_F|/2$ nodes p' in V'_F such that at least $|V'_F|/4$ of the other nodes $w \in V'_F \backslash \{p'\}$ satisfy $|S_{p'w} \cap V'_F| > |S_{wp'} \cap V'_F|$.*

We apply Corollary III.7 with $V_F = V(G_i)$, $V'_F = V_i$, and $H_{G_i}(V_i)$ as the helper graph of G_i on V_i, where the corresponding cuts are the minimum cuts between pairs in V_i, to get that the probability that at least $4 \log n$ random pivots p all satisfy $|V_i^{\le 1/2}(p)| < n_i/4$, which we call an *unsuccessful* choice of pivot p, is bounded by $1/n^4$. The number of expansion steps is at most $n-1$, because the final tree $\mathcal{T}$ contains $n-1$ edges, and each expansion step creates at least one such edge. By a union bound we conclude that with probability at least $1 - 1/n^3$, every expansion step picks a successful pivot within $4 \log n$ trials. Observe that for every choice of p we compute $h_p(u)$ for all $u \in V_i$, which takes time $\tilde{O}(|V_i| + |G_i|)$ for all pivots. We can thus focus henceforth on the execution with a successful pivot p.

We now turn to bound the total time spent on queries in G_i. Let $\mathcal{T}_i^*$ be the subgraph of $\mathcal{T}^*$ induced on V_i. Observe that $\mathcal{T}_i^*$ must be connected, because V_i is a super-node in an intermediate tree of the Gomory-Hu algorithm (see Lemma III.4). Define a function $\ell : V(\mathcal{T}_i^*) \setminus \{p\} \to E(\mathcal{T}_i^*)$, where $\ell(u)$ is the lightest edge in the path between u and p in $\mathcal{T}_i^*$, and $\ell(p) = \emptyset$ (see Figure 3 for illustration); it is well-defined because Assumption III.3 guarantees there are no ties. For an edge $e \in \mathcal{T}_i^*$, we say that e is *hit* if the targets $u_{i,1}, \ldots, u_{i,d}$ picked by the expansion step include a node u such that $\ell(u) = e$. Let H_e be an indicator for the event that edge e is hit. In order to bound the total number of nodes and edges in the CAG that participate in minimum-cut queries performed by the expansion step, we first bound the number of edges that are hit along any single path.

Claim III.8. *With high probability, for every path P between a leaf and p in $\mathcal{T}_i^*$, the number of edges in P that are hit is $\sum_{e \in P} H_e \le O(\log n)$.*

Proof: Let $\mathcal{T}_{i,\ell}^*$ be the graph constructed from $\mathcal{T}_i^*$ by merging nodes whose image under ℓ is the same. Observe that nodes that are merged together, namely, $\ell^{-1}(e)$ for $e \in E(\mathcal{T}_i^*)$, are connected in $\mathcal{T}_i^*$, and therefore the resulting $\mathcal{T}_{i,\ell}^*$ is a tree. See Figure 3 for illustration. We shall refer to nodes of $\mathcal{T}_{i,\ell}^*$ as *vertices* to distinguish them from nodes in the other graphs. For example, p is not merged with any other node, and thus forms its own vertex.

For sake of analysis, fix a leaf in $\mathcal{T}_{i,\ell}^*$, which determines a path to the root p, denoted P_ℓ, and let us now bound the number of nodes picked (by the expansion step) from vertices in P_ℓ.

Claim III.9. *With high probability, the total number of nodes u picked by the algorithm from vertices in P_ℓ is at most $O(\log n)$.*

Proof: We will need the following two observations.

Observation III.10. *No vertex in $\mathcal{T}_{i,\ell}^*$ contains nodes from both $V_i^{\le 1/2}(p)$ and $V_i \setminus V_i^{\le 1/2}(p)$.*

This is true because all nodes u in the same vertex $\ell^{-1}(e)$ have the same minimum *up*-cut in G, which is a basic property of the cut-equivalent tree $\mathcal{T}^*$, and thus all these nodes will have the same S_u and the same S'_u computed in the CAG G_i.

Observation III.11. *The vertices that contain nodes in $V_i^{\le 1/2}(p)$ form a prefix of the path P_ℓ.*

This is true by monotonicity of $|S_x|$ as a function of the hop-distance of x from p in P_ℓ, denoted P'_ℓ.

The algorithm only picks nodes from $V_i^{\le 1/2}(p)$, thus it suffices to bound the nodes picked from (the vertices along)

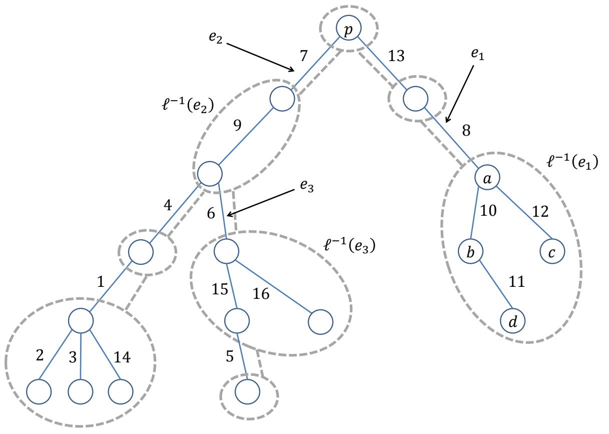

Figure 3: An illustration showing $\mathcal{T}_i^*$ with solid blue lines, while the corresponding graph $\mathcal{T}_{i,\ell}^*$ with dashed gray lines. For example, $e_1 = \ell(a) = \ell(b) = \ell(c) = \ell(d)$. The nodes in $\ell^{-1}(e_2)$ are not in $V_i^{\le 1/2}(p)$, and so the expansion step never picks any of them as a sink. After picking any node from $\ell^{-1}(e_3)$, a new super-node containing $\ell(e_3)$ (and possibly the vertex below as well) is formed.

the prefix P'_ℓ. Fix a list π of the nodes in (vertices in) P'_ℓ in increasing order of their hop-distance from p in P_ℓ, Now recall that the targets $u_{i,1}, \dots, u_{i,d}$ are chosen sequentially, each time uniformly at random from $U_p \cap V_i^{\le 1/2}(p)$ for the current U_p. Initially, U_p contains all the nodes in π (but may contain also nodes outside the path P_ℓ). Now each time a target u is chosen, some nodes are separated away from U_p. Define the list π' to be the restriction of π to nodes currently in U_p; notice that U_p and π' change during the random target choices, but π is fixed. We can classify the randomly chosen target u into three types.

1. u is not from the current list π': In this case π' does not change. We call this a "don't care" event, because we shall ignore this choice.
2. u is from the current list π': In this case π' is shortened into a prefix of π' that *does not* contain u. We now have two subcases:
   - 2.a. u is from the first half of π': Then π' is shortened by factor at least 2. We call this event "big progress".
   - 2.b. u is from the second half of π': We call this event "small progress".

Now to complete the proof of Claim III.9, consider the random process of choosing the targets u. To count the number of targets u from P_ℓ, we can ignore targets of type 1 and focus on targets of type 2, in which case type 2a occurs with probability at least $1/2$. As the initial list π has length at most n, with high probability the random process terminates within $16 \log n$ steps (counting only targets of type 2).[10] ■

Proceeding with the proof of Claim III.8, suppose the path P consists of nodes $v_1, \dots, v_k = p$ where v_1 is the leaf. Then the path P_ℓ consists of $\ell^{-1}(\ell(v_1)), \dots, \ell^{-1}(\ell(v_k))$ restricted to distinct vertices. Note that whenever an edge e in P that is hit, some target u is picked from $\ell^{-1}(e)$ and in particular from P_ℓ. By Claim III.9, with high probability the number of target nodes picked from P_ℓ is bounded by $O(\log n)$, implying that also the number of hit edges in P is bounded by $O(\log n)$. Finally, Claim III.8 follows by applying a union bound over all (at most n) leaves. ■

Next, we use Claim III.8 to bound the total running time of an expansion step (proof is deferred to the full version).

Claim III.12. *An internal iteration in the expansion step, that partitions a super-node U_p into $U_p \setminus S_u$ and $U_p \cap S_u$, takes time $\tilde{O}(|S_u| + k_{up}^i)$, where k_{up}^i is the number of edges in the minimum up-cut $(V(G_i) \setminus S_u, S_u)$.*

We continue with the proof of Lemma III.6, that the total time for an expansion step is bounded. We may assume henceforth that the $O(\log n)$ bound in Claim III.8 holds, as it occurs with high probability. The number of times a node $u \in V(G_i)$ is queried (when it belongs to some S_v) is equal to the number of hit edges in its path to the pivot p in $\mathcal{T}_i^*$, which we just assumed to be bounded by $O(\log n)$. The number of times an edge $e \in E(G_i)$ is queried is equal to the number of hit edges in $\mathcal{T}_i^*$ along the two paths from e's ends to the pivot p, which we just assumed to be bounded by $O(\log n)$. Altogether, the time it takes to scan the cuts $S_{u_{i,1}}, \dots, S_{u_{i,d}}$ and the corresponding super-nodes $V_{i,1}, \dots, V_{i,d}$ that are separated away from V_i is bounded, by Claim III.12, by

$$\tilde{O}\Big(\sum_{j=1}^{d} |S_{u_{i,j}}| + k^i_{u_{i,j}p}\Big) \le \tilde{O}\Big(|V(G_i)| + |E(G_i)|\Big).$$

Finally, observe that the total time it takes to construct the CAGs of any super-node V_i's children in a single expansion step is linear in the size of V_i's CAG. This completes the proof of Lemma III.6. ■

Next, we show that the total size of all CAGs at a certain depth is bounded by $O(m)$. In fact, we show it for partition trees, which generalize the intermediate trees produced by our algorithm. A *partition tree* T of a graph $G = (V, E)$ is a tree whose nodes $V_1, \dots, V_l$ are super-nodes of G and form a partition $V = V_1 \sqcup \cdots \sqcup V_l$. Clearly, our intermediate tree $\mathcal{T}$ is a partition tree, and so we are left with proving the following lemma, whose proof is in the full version.

[10]The similar but different idea that the minimum cuts from a uniformly random node p partition the auxiliary graph in a balanced way with high probability, which allows bounding the recursion depth by analyzing the maximal length of paths in the recursion tree, appears in Lemma 35 and Theorem 11 in [32].

Lemma III.13. *Let $G = (V, E)$ be an input graph, and let T be a partition tree on super-nodes $V_1, \dots, V_l$. Then the total size of the corresponding* CAG*s $G_1, \dots, G_l$ is at most $2n + 3m = O(m)$.*

We are now ready to prove the main Theorem.

Proof of Theorem III.1 under the assumption on Max-Flow *queries:* To simplify matters, let us assume henceforth that $t_p(m) = \tilde{O}(m)$ and $t_{mc}(m) = \tilde{O}(1)$. The general case is analyzed similarly and results in the time bound $\tilde{O}(t_p(m) + m \cdot t_{mc}(m))$ stated in Theorem III.1 for the following reasons. The preprocessing time is performed $\tilde{O}(1)$ times per CAG, hence the total preprocessing time over all CAGs that the algorithm constructs is at most $\tilde{O}(t_p(m))$, the first summand above. The total size of all answers to all queries at a single depth is near-linear in the total size of all CAGs at this depth; hence over all depths it is bounded by $\tilde{O}(m \cdot t_{mc}(m))$, the second summand above.

First, assume the perturbation attempt is successful. By Lemma III.6 the total time spent at each super-node V_i is near-linear in the size of G_i, and thus by Lemma III.13, the total time spent at each recursion depth is bounded by $O(m)$. By the definition of the algorithm, at each super-node V_i during the recursion, $\Theta(|V_i|)$ nodes are partitioned away from V_i, and so by Lemma III.13, $\Theta(n)$ nodes are partitioned away from all CAGs at this depth, thus after the $O(\log n)$ depth, each super-node V_i is a singleton, concluding Theorem III.1 in this case.

Second, if the perturbation attempt is unsuccessful, which happens with probability at most $1/n^3$, and two cuts are crossing each other, then we would identify that and restart the algorithm. By Lemma III.6, with probability at most $1/n^3$ the number of incorrect pivots exceeds $O(\log n)$, and by a union bound with the probability of a failed perturbation attempt, the running time of the algorithm is bounded by $\tilde{O}(m)$ with high probability. ■

E. Lifting the Assumption on Max-Flow Queries

Recall that our goal is to construct a cut-equivalent tree using access to Min-Cut queries. So far we have assumed that we also have access to Max-Flow queries. In this subsection we show how to lift this additional assumption. We will change the algorithm and the analysis slightly.

First, at each expansion step, run the algorithm on $4 \log n$ preprocessed copies of G_i, each on one of the randomly picked pivots. Similar to our calculation from the original proof, with high probability, for every expansion step throughout the execution, at least one of the corresponding graphs will have a successful pivot. We will make sure that an unsuccessful pivot will never output a wrong tree; it may only keep running indefinitely (until we halt it). Since with high probability at least one of the graphs is of a successful pivot, this only incurs a factor of $\tilde{O}(1)$ to the running time.

Second, instead of picking a node $u \in U_p \cap V_i^{\leq 1/2}(p)$ at random as in the original algorithm, pick $4 \log_{8/7} n$ nodes from U_p and using arguments from dynamic connectivity (which are deferred to the full version), check for $4 \log_{8/7} n$ copies of G_i, simultaneously, each for one of the chosen nodes u, if $|S'_u| \leq n_i/2$. If all nodes were unsuccessful choices, draw another set of $4 \log_{8/7} n$ nodes. Continue to draw batches until at least one node is successful. Then, for an arbitrary successful node u, use dynamic connectivity arguments to find the k^i_{up} edges in the minimum up-cut, and the nodes in S'_u.

Since the probability for a single node u chosen at random to satisfy $|S'_u| \leq n_i/2$ is always at least $1/8$, and as we pick $4 \log_{8/7} n$ nodes uniformly at random each time, we get that: with probability at least $1 - (7/8)^{4 \log_{8/7} n} = 1 - 1/n^4$, at least one of the $4 \log_{8/7} n$ chosen nodes is successful. By a union bound over the maximal number of partitions in expansion steps throughout the execution, i.e. internal iterations of expansion steps (at most n), we get that with probability at least $1 - 1/n^3$ each one of the batches results in at least one of the $4 \log_{8/7} n$ nodes in the batch is successful. Hence, the only part of the proof that needs to be further addressed is Claim III.8. In particular, we prove the following variant of the claim.

Claim III.14. *With high probability, for every path P between a leaf and p in $\mathcal{T}_i^*$, the total number of edges in P that are hit is at most $O(\log^2 n)$.*

Proof: We mention the differences from the proof of the original Claim III.8. The classification of the choice of a random target u into three types, where only the following items are different from the ones in Claim III.8.

2.a u is from the first $1 - 1/(3 \log_{8/7} n)$ fraction of π': Then π' is shortened by factor at least $1/(3 \log_{8/7} n)$. We call this event "big progress".

2.b u is from the complement part of π': We call this event "small progress".

Here, we have a random process in which type III-E occurs with probability at least $1 - 1/(3 \log_{8/7} n)$, and therefore with high probability it terminates within $64 \log_{8/7} n \ln n$ steps (these steps count only targets of type 2). We conclude that with high probability, every such path has at most $64 \log_{8/7} n \ln n = O(\log^2 n)$ nodes chosen from its vertices. ■

We proceed to the proof of Theorem III.1, highlighting the differences.

Proof of Theorem III.1: With high probability, at each expansion step at most $O(\log n)$ unsuccessful pivots are chosen before picking a successful one. At each level, we spend at most $t_p(m)$ time for the preprocessing of the min-cut data structures for fixed sources, and so unsuccessful pivots only incur a factor $\tilde{O}(1)$ on the running time. ■

References

[1] G. Borradaile, P. Sankowski, and C. Wulff-Nilsen, "Min st-cut oracle for planar graphs with near-linear preprocessing time," *ACM Trans. Algorithms*, vol. 11, no. 3, 2015.

[2] G. Borradaile, D. Eppstein, A. Nayyeri, and C. Wulff-Nilsen, "All-pairs minimum cuts in near-linear time for surface-embedded graphs," in *32nd International Symposium on Computational Geometry*, ser. SoCG '16, vol. 51. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 22:1–22:16.

[3] Y. T. Lee and A. Sidford, "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*, ser. FOCS '14. IEEE Computer Society, 2014, pp. 424–433.

[4] R. E. Gomory and T. C. Hu, "Multi-terminal network flows," *Journal of the Society for Industrial and Applied Mathematics*, vol. 9, pp. 551–570, 1961. [Online]. Available: http://www.jstor.org/stable/2098881

[5] A. Mądry, "Computing maximum flow with augmenting electrical flows," in *Proceedings of the 57th IEEE Annual Symposium on Foundations of Computer Science*, ser. FOCS '16. IEEE Computer Society, 2016, pp. 593–602.

[6] Y. P. Liu and A. Sidford, "Faster energy maximization for faster maximum flow," *CoRR*, 2019. [Online]. Available: http://arxiv.org/abs/1910.14276

[7] A. Bhalgat, R. Hariharan, T. Kavitha, and D. Panigrahi, "An $O(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs," in *39th Annual ACM Symposium on Theory of Computing*, ser. STOC'07. ACM, 2007, pp. 605–614.

[8] D. R. Karger and M. S. Levine, "Fast augmenting paths by random sampling from residual graphs," *SIAM J. Comput.*, vol. 44, no. 2, pp. 320–339, 2015.

[9] A. Abboud, R. Krauthgamer, and O. Trabelsi, "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*, ser. SODA '20, USA, 2020, p. 48–61.

[10] A. V. Goldberg and K. Tsioutsiouliklis, "Cut tree algorithms: an experimental study," *Journal of Algorithms*, vol. 38, no. 1, pp. 51–83, 2001.

[11] D. Panigrahi, "Gomory-Hu trees," in *Encyclopedia of Algorithms*, M.-Y. Kao, Ed. Springer New York, 2016, pp. 858–861.

[12] A. Abboud, V. Vassilevska Williams, and H. Yu, "Matching triangles and basing hardness on an extremely popular conjecture," in *Proc. of 47th STOC*, 2015, pp. 41–50.

[13] R. Krauthgamer and O. Trabelsi, "Conditional lower bounds for all-pairs max-flow," *ACM Trans. Algorithms*, vol. 14, no. 4, pp. 42:1–42:15, 2018.

[14] A. Abboud, L. Georgiadis, G. F. Italiano, R. Krauthgamer, N. Parotsidis, O. Trabelsi, P. Uznanski, and D. Wolleb-Graf, "Faster Algorithms for All-Pairs Bounded Min-Cuts," in *46th International Colloquium on Automata, Languages, and Programming (ICALP 2019)*, vol. 132, 2019, pp. 7:1–7:15.

[15] R. Hassin and A. Levin, "Flow trees for vertex-capacitated networks," *Discrete Appl. Math.*, vol. 155, no. 4, pp. 572–578, 2007.

[16] H. Y. Cheung, L. C. Lau, and K. M. Leung, "Graph connectivities, network coding, and expander graphs," *SIAM Journal on Computing*, vol. 42, no. 3, pp. 733–751, 2013.

[17] S. R. Arikati, S. Chaudhuri, and C. D. Zaroliagis, "All-pairs min-cut in sparse networks," *J. Algorithms*, vol. 29, no. 1, pp. 82–110, 1998.

[18] T. Hagerup, J. Katajainen, N. Nishimura, and P. Ragde, "Characterizing multiterminal flow networks and computing flows in networks of small treewidth," *J. Comput. Syst. Sci.*, vol. 57, pp. 366–375, 1998.

[19] J. Lacki, Y. Nussbaum, P. Sankowski, and C. Wulff-Nilsen, "Single source - all sinks Max Flows in planar digraphs," in *Proc. of the 53rd FOCS*, 2012, pp. 599–608.

[20] D. Gusfield, "Very simple methods for all pairs network flow analysis," *SIAM Journal on Computing*, vol. 19, no. 1, pp. 143–155, 1990.

[21] H. N. Gabow, "A matroid approach to finding edge connectivity and packing arborescences," *J. Comput. Syst. Sci.*, vol. 50, no. 2, pp. 259–273, 1995.

[22] J. Edmonds, "Submodular functions, matroids, and certain polyhedra," *Combinatorial structures and their applications*, pp. 69–87, 1970.

[23] R. Cole and R. Hariharan, "A fast algorithm for computing steiner edge connectivity," in *Proceedings of the Thirty-fifth Annual ACM Symposium on Theory of Computing*, ser. STOC '03. ACM, 2003, pp. 167–176.

[24] R. Hariharan, T. Kavitha, and D. Panigrahi, "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, 2007, pp. 127–136. [Online]. Available: http://dl.acm.org/citation.cfm?id=1283383.1283398

[25] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford, "An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations," in *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014*, 2014, pp. 217–226.

[26] W. Mayeda, "On oriented communication nets," *IRE Transactions on Circuit Theory*, vol. 9, no. 3, pp. 261–267, 1962.

[27] F. Jelinek, "On the maximum number of different entries in the terminal capacity matrix of oriented communication nets," *IEEE Transactions on Circuit Theory*, vol. 10, no. 2, pp. 307–308, 1963.

[28] L. Georgiadis, D. Graf, G. F. Italiano, N. Parotsidis, and P. Uznanski, "All-Pairs 2-Reachability in $O(n^\omega \log n)$ Time," in *44th International Colloquium on Automata, Languages, and Programming (ICALP 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 80. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 74:1–74:14.

[29] K. Bringmann and M. Kunnemann, "Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping," in *Proc. of 56th FOCS*, 2015, pp. 79–97.

[30] H. Räcke, "Minimizing congestion in general networks," in *Proceedings of the 43rd Symposium on Foundations of Computer Science*, ser. FOCS '02. IEEE Computer Society, 2002, p. 43–52.

[31] H. Räcke, C. Shah, and H. Täubig, "Computing cut-based hierarchical decompositions in almost linear time," in *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '14. SIAM, 2014, p. 227–238.

[32] A. Bhalgat, R. Cole, R. Hariharan, T. Kavitha, and D. Panigrahi, "Efficient algorithms for Steiner edge connectivity computationand Gomory-Hu tree construction for unweighted graphs," 2008, unpublished full version of [7]. [Online]. Available: http://hariharan-ramesh.com/papers/gohu.pdf

[33] N. Anari and V. V. Vazirani, "Planar graph perfect matching is in NC," in *59th IEEE Annual Symposium on Foundations of Computer Science*, ser. FOCS '18. IEEE Computer Society, 2018, pp. 650–661.

[34] M. R. Henzinger and V. King, "Randomized dynamic graph algorithms with polylogarithmic time per operation," in *Proceedings of the Twenty-Seventh Annual ACM Symposium on Theory of Computing*, 1995, p. 519–527.

[35] A. A. Benczúr and D. R. Karger, "Randomized approximation schemes for cuts and flows in capacitated graphs," *SIAM J. Comput.*, vol. 44, no. 2, pp. 290–319, 2015.

[36] K. Mulmuley, U. V. Vazirani, and V. V. Vazirani, "Matching is as easy as matrix inversion," *Combinatorica*, vol. 7, no. 1, pp. 105–113, 1987.

[37] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*. Springer-Verlag, 1985.

[38] V. Gurvich and M. N. Vyalyi, "Characterizing (quasi-)ultrametric finite spaces in terms of (directed) graphs," *Discret. Appl. Math.*, vol. 160, no. 12, pp. 1742–1756, 2012.