Using MSTs to Compute All-Pairs Mincuts and Beyond ...

Surender Baswana * Anupam Roy †

Abstract

Maximum spanning trees (MSTs) and mincuts are fundamental graph structures. In the past, MSTs have been used to obtain an upper bound on the distinct values of all-pairs mincuts [GH61] as well as all-pairs second mincuts [BR26]. In this article, we establish a deeper connection between these problems. We design an algorithmic framework for computing all-pairs mincuts that is similar to the computation of an MST in a special model. The main strength of this framework lies in its flexibility, simplicity, and generality, which we use to arrive at the following two results for undirected graphs on n vertices.

(1) **First Parallel Algorithm for All-Pairs Mincuts:** We design a parallel algorithm which shows that the depth of computing all-pairs mincuts differs from that of a single pair mincut by only a logarithmic factor, while achieving work done equal to $\mathcal{O}(n \log n)$ (u, v)-mincut computations. Previous algorithms either provide an approximate solution [LNPS23] or achieve $\Omega(n)$ depth [ALPS23].

(2) **First Algorithm for All-Pairs Second Mincuts:** A 2^{nd} (u, v)-mincut is an (u, v)-cut of second minimum value. Recently, [BR26] have shown that there are at most $2n-4$ distinct values of second mincuts for all pairs of vertices. However, the only way to compute all-pairs second mincuts is to trivially make $\mathcal{O}(n^2)$ 2^{nd} (u, v)-mincut computations. We design the first efficient algorithm for computing all-pairs second mincuts, that requires $\mathcal{O}(n \log n)$ 2^{nd} (u, v)-mincut computations. The number of 2^{nd} mincut computations in our algorithm is optimal upto $\mathcal{O}(\log n)$ factor.

All our results are for arbitrary cut functions, that is, the value of each cut $(A, V \setminus A)$ is given by an arbitrary real valued function F such that $F(A) = F(V \setminus A)$. In this setting, the algorithm of Cheng and Hu [CH91] has remained the best known algorithm for computing all-pairs mincuts for the last 35 years. Quite surprisingly, this algorithm turns out to be identical to a special execution of our framework. Moreover, the algorithms derived from our framework closely resemble the classical algorithms for computing MST – Prim's, Borůvka's and Kruskal's.

*Indian Institute of Technology Kanpur, India. Email: sbaswana@cse.iitk.ac.in. Partially supported by Tapas Mishra Memorial Chair at IIT Kanpur, India.

†Indian Institute of Technology Kanpur, India. Email: anupam@cse.iitk.ac.in.

Contents

1 Introduction **1**
- 1.1 New Framework for All-Pairs Mincuts . . . 2
- 1.2 First parallel algorithm for All-Pairs Mincuts . . . 3
- 1.3 First Algorithm for All-Pairs Second Mincuts . . . 4

2 Preliminaries **5**

3 Algorithmic Framework for All-Pairs Mincuts **5**
- 3.1 Components of our Framework . . . 6
- 3.2 Framework . . . 8
- 3.3 Analysis . . . 8

4 Applications of Our Framework for All-Pairs Mincuts **9**
- 4.1 Algorithm of Cheng and Hu [CH91] as a special case of our framework . . . 9
- 4.2 First Parallel Algorithm for All-Pairs Mincuts . . . 10

5 First algorithm for All-Pairs Second Mincuts **11**
- 5.1 Eliminating Triangles in $\mathcal{G}_2$ . . . 11
- 5.2 Generalizing Subroutines of the Framework . . . 13
- 5.3 Satisfying the Premise of Lemma 9 . . . 13
- 5.4 Algorithm . . . 15
- 5.5 Analysis . . . 15

A Impact on Directed Graphs **19**

B When 2^{nd} Mincut Does not Exist for a pair **19**

C Algorithm of [CH91] as a special case of our framework **20**

D Recursive Approach to PruneEdges and AssignMincut **21**
- D.1 AssignMincut for Algorithm 3 . . . 23
- D.2 Designing Assign2^{nd}Mincut for Algorithm 7 . . . 23

E Parallel Implementation of Algorithm 4 **24**

1 Introduction

Let $G = (V, E)$ be an undirected weighted graph on $n = |V|$ vertices and $m = |E|$ edges where the *capacity* $c(e)$ of each edge $e \in E$ is a non-negative real value. A cut of G is defined as a non-empty subset proper of V. We can associate a value with each cut C in G as follows. Let F be any real-valued function defined on the set of all cuts such that for each cut C, $F(C)$ is a finite value and $F(C) = F(\overline{C})$, where $\overline{C} = V \setminus C$. While F is often defined to be the capacity[1] $c(C)$ of the cut, there are many applications where $F \neq c$. For any pair of vertices (u, v), a cut is said to be a (u, v)-cut if it separates[2] (u, v). A (u, v)-cut of the least value is called a (u, v)-mincut. A second (u, v)-mincut is a (u, v)-cut of second minimum value.

The problem of all-pairs mincuts is to compute a (u, v)-mincut for all $u, v \in V$. In 1961, Gomory and Hu [GH61] designed the first non-trivial algorithm for this problem in the classical setting where $F = c$. To arrive at it, [GH61] showed that although there are $\binom{n}{2}$ pairs of vertices, there exists a set of $n - 1$ cuts such that at least one mincut for each pair is present in this set. We call this set a *mincut cover*. They further showed that this set can be stored in $\mathcal{O}(n)$ space using a spanning tree called the *Gomory-Hu tree*. Finally, they designed an algorithm to compute the Gomory-Hu tree using $n - 1$ (u, v)-mincut computations. After an extensive research for the last six decades, the problem of computing a Gomory-Hu tree has now culminated due to the recent seminal result of [ALPS23] which shows that all-pairs mincuts can be computed in $m^{1+o(1)}$ time.

The existence of the Gomory-Hu tree crucially relies on the fact that $F = c$ is a submodular function [Lov79]. However, there are applications [CH92, Has88, LR89], which require a (u, v)-cut that has *less* capacity and also divides the graph more *evenly*. In order to compute such a cut, one would like to define F as $F(C) = \frac{c(C)}{|C|.|\overline{C}|}$ for every cut C instead of $F = c$. Unfortunately, with this change, F ceases to remain submodular. There are many other well-known problems such as *symmetric all-pairs mincuts* in directed graphs [Ben95a], *vital edges* for (u, v)-mincut [AFLR19, BB24], where F needs to be defined in a way such that F is not submodular. As a result, none of the existing algorithms that focus on computing Gomory-Hu tree can be used to solve these problems. Quite surprisingly, even though in such situations, the Gomory-Hu tree might not exist, the number of distinct mincut values for all pairs continues to remain at most $n - 1$. In fact, it is true even if we consider F to be any arbitrary real valued function. Utilizing this fact, many efficient algorithms [Has88, Has90, CH91] have been designed for computing all-pairs mincuts in this setting. However, the best known bound was obtained by Cheng and Hu [CH91] who showed that we can compute all-pairs mincuts using $n - 1$ (u, v)-mincut computations. To arrive at it, [CH91] showed that even for the setting of arbitrary cut functions, there exists a mincut cover of size $n - 1$ and can be represented by a structure called the *ancestor tree*.

Unlike the Gomory-Hu tree, storing the ancestor tree requires $\mathcal{O}(n^2)$ space. Moreover, for computing the ancestor tree, the bound of $n - 1$ mincut computations has remained unbroken for 35 years since the work of [CH91]. The primary reason behind these differences is that the sub-modularity of cuts provides the following *uncrossing property of mincuts*. If A is a (u, v)-mincut, then for any $x, y \in A$, there exists a (x, y)-mincut, say B, such that $B \subseteq A$. This property is important both combinatorially and algorithmically as it provides a natural divide and conquer strategy. However, this fails to hold for arbitrary cut functions.

In the past, many interesting relations have been established between two seemingly unrelated problems such as the duality between (s, t)-maxflow and (s, t)-mincut [FF56], the farthest (s, t)-mincut and fault tolerant (s, t)-reachability [BCR16], the semi-duality between global mincut and

[1] For a cut C, its capacity $c(C)$ is the sum of capacity of the edges separated by C

[2] A cut C is said to *separate* a pair (u, v) if $u \in C$ and $v \in \overline{C}$

number of edge disjoint spanning trees [NW61, Kar00]. Not only these relations have led to an efficient way to solve a given problem, but have also provided new insights into the problem. Maximum spanning tree (MST) is a fundamental graph structure. Many algorithms for computing an MST appear in basic textbooks and are well-known to the algorithm's community. Interestingly, more than 60 years ago, Gomory and Hu [GH61] established the following relation between MST and the distinct values of all-pairs mincuts. Quite recently, [BR26] have shown that this relationship can be generalized even for cuts beyond mincuts for all pairs of vertices.

Theorem 1 (MST and Mincut [GH61])**.** *Let $\mathcal{G}_1$ be a complete graph on V where capacity of each edge is its mincut value, and let $\mathcal{T}_1$ be any MST for $\mathcal{G}_1$. For any pair of vertices (u, v), the value of (u, v)-mincut is equal to the capacity of minimum capacity edge on the (u, v)-path in $\mathcal{T}_1$.*

Unfortunately, for the last 6 decades, Theorem 1 is the only relationship that exists between all-pairs mincuts and MSTs. In this article, we demonstrate a deeper relationship between MST and all-pairs mincuts. We design an algorithmic framework for computing all-pairs mincuts that is similar to the computation of an MST in a *specific* model. The framework consists of three components: a tool EliminateTriangle and two subroutines PruneEdges and MergeTrees.

1. EliminateTriangle(u, v, x) : Given any three vertices u, v, x, it efficiently eliminates the least capacity edge in the triangle $\langle u, v, x\rangle$ in $\mathcal{G}_1$.

Let $V_1, V_2 \subsetneq V$ be any two disjoint subsets of vertices and T_1, T_2 be the two MSTs formed on V_1, V_2 in $\mathcal{G}_1$. The two subroutines perform the following tasks.

2. PruneEdges(T_1, T_2): Prunes the inter-tree edges between T_1 and T_2 which need not be in an MST of $\mathcal{G}_1$, using EliminateTriangle.
3. MergeTrees(T_1, T_2) : Computes a MST for $V_1 \cup V_2$ in $\mathcal{G}_1$ using T_1, T_2, and the inter-tree edges that remain after PruneEdges(T_1, T_2).

This framework outputs an MST for $\mathcal{G}_1$ and as a secondary result, it also outputs a mincut cover for all pairs of vertices. The main strength of this framework lies in its flexibility and simplicity. We exploit these attributes to arrive at the following three results. We design a new divide-and-conquer algorithm for computing all-pairs mincuts, which can be implemented seamlessly in the parallel environment. Secondly, we show that the algorithm of Cheng and Hu [CH91] is just a special execution of this framework. Finally, we show that this framework is much more generic, and surprisingly, it extends to computing all-pairs second mincuts as well. All the results we obtain are for the setting of arbitrary cut functions.

All previous approaches focus on computing the compact structures (Gomory-Hu tree [GH61], ancestor tree [CH91], $\mathcal{O}(\log n)$ gen-ancestor trees [BR26]). Despite their advantages, they overlook the shared structures underlying the three problems: (1) an MST and (2) a *small set* that contains a (second) mincut for all pairs. The generic nature of our framework stems from its focus on computing these two shared structures.

1.1 New Framework for All-Pairs Mincuts

For any subset of vertices V', a Gomory-Hu tree of V' is a spanning tree on V' such that for any $x, y \in V'$, an (x, y)-mincut in the tree is an (x, y)-mincut in the graph. Hence, a Gomory-Hu tree of V' stores a mincut cover for all $V' \times V'$ pairs; it is also an MST for V' in $\mathcal{G}_1$. The following property is implicitly used in various algorithms for computing Gomory-Hu tree. Let (V_1, V_2) be a (x, y)-mincut. The Gomory-Hu tree of $V_1 \cup V_2$ can be constructed by joining the Gomory-Hu tree of V_1 and the Gomory-Hu tree of V_2 using a suitably computed edge (a, b) with $a \in V_1$ and $b \in V_2$.

However, this property is critically dependent on the submodularity of the cuts and therefore fails when F is an arbitrary cut function. We show in the following theorem that not everything is lost for the setting of arbitrary cut function. In fact, even if we consider V_1, V_2 to be any arbitrary subsets of vertices, we can efficiently compute an MST for $V_1 \cup V_2$ in $\mathcal{G}_1$ and a mincut cover for the missing pairs $V_1 \times V_2$ using just $|V_1|$ instead of $|V_1| \times |V_2|$ mincut computations.

Theorem 2. *Let V_1 and V_2 be any two disjoint subsets of V. Let C_1 and C_2 be the mincut cover for all $V_1 \times V_1$ pairs and all $V_2 \times V_2$ pairs respectively. Let T_1 and T_2 be the MSTs of $\mathcal{G}_1[V_1]$ and $\mathcal{G}_1[V_2]$ respectively. There exists a subset $P \subsetneq V_1 \times V_2$ with $|P| = \min(|V_1|, |V_2|)$ such that the following properties hold.*

- *Mincut for each pair in P along with $\mathcal{C}_1 \cup \mathcal{C}_2$ is a mincut cover for all-pairs from $V_1 \times V_2$.*
- *an MST on edges $(T_1 \cup T_2 \cup P)$ is an MST for $\mathcal{G}_1[V_1 \cup V_2]$.*
- *There exists an algorithm that computes P in $\mathcal{O}(|V_1|.|V_2|)$ time.*

This result forms the backbone of our framework for computing all-pairs mincuts. The fact that Theorem 2 allows V_1, V_2 to be any arbitrary subsets is what makes the framework so flexible. Despite this flexibility, the number of mincut computations performed during any execution of the framework never exceeds $\mathcal{O}(n \log n)$. Moreover, a special execution of our algorithm, that requires $n-1$ mincut computations, turns out to be identical to the algorithm of Cheng and Hu [CH91], albeit viewed from the perspective of MST computation. Finally, we show that this special execution of the algorithm has a close resemblance to the Prim's algorithm for MST [Pri57].

1.2 First parallel algorithm for All-Pairs Mincuts

The parallel model of computation has been well studied for many decades. Let P be a parallel algorithm that given any designated pair of vertices (s, t), can compute an (s, t)-mincut. Let $D_P(s, t)$ be its depth and $W_P(s, t)$ denote its work done. A trivial way to compute all-pairs mincuts in the parallel environment is to compute a mincut for each pair in parallel using $\binom{n}{2}$ processors. This requires $\mathcal{O}(D_P(s, t))$ depth and $\mathcal{O}(n^2 W_P(s, t))$ work. On the other extreme, the algorithm of Cheng and Hu [CH91], being inherently sequential, requires $\mathcal{O}(nD_P(s, t))$ depth and $\mathcal{O}(nW_P(s, t))$ work. We present the first parallel algorithm for computing all-pairs mincuts that matches the best, of both these extremes, up to a logarithmic factor, as follows.

Theorem 3 (Parallel All-Pairs Mincuts)**.** *Let $G = (V, E)$ be a graph on n vertices where the value of each cut C is given by any arbitrary real valued function F such that $F(C) = F(V \setminus C)$. There exists a parallel algorithm that computes all-pairs mincuts in G achieving $\mathcal{O}(\log n D_P(s, t) + \log^2 n)$ depth and $\mathcal{O}(n \log n W_P(s, t) + n^2)$ work.*

Interestingly, our parallel algorithm is obtained by utilizing the flexibility of our framework. In particular, we pick V_1 and V_2 to be of equal size, since Theorem 2 allows us to pick them arbitrarily. This provides a divide-and-conquer algorithm for all-pairs mincuts with $\mathcal{O}(\log n)$ recursion depth. Moreover, the subroutines of the framework are suitably modified, so that the algorithm can be easily parallelized. This algorithm has resemblance to the Borůvka's algorithm for MST [Bor26].

In the classical setting ($F = c$), the problem of computing a mincut for a single pair of vertices has been very well addressed in the parallel setting [OS93, LS14, VDBGJDV25]. Many algorithms also exist for computing all-pairs approximate mincuts in the parallel setting (see [AKL+, LNPS23]). However, no non-trivial algorithms are known for computing all-pairs mincuts (exactly) in the parallel setting, as stated in [LNPS23]. Although the best known sequential algorithm for all-pairs mincuts [ALPS23] achieves $m^{1+o(1)}$ work, one of its main components has $\Omega(n)$ depth [PS]. Using

the parallel algorithm of [VDBGJDV25] for computing (s,t)-mincut (Theorem 1.2 and Corollary 1.5), Theorem 3 provides the first parallel algorithm for computing all-pairs mincuts that achieves $\tilde{\mathcal{O}}(\sqrt{n})$ depth and $\tilde{\mathcal{O}}(n(m+n^{1.5}))$ work. This also provides the first parallel algorithm for *symmetric* all-pairs mincuts in directed graphs (refer to Section A).

1.3 First Algorithm for All-Pairs Second Mincuts

A lot of research has also been done on cuts beyond minimum capacity [VY92, DN95, Ben95b, GV95, NNI97, Kar00, BBR25]. Quite expectedly, they often lead to efficient algorithms for fundamental problems [Kar95, KYN07, Ben95b] as well as efficient data structures in the *fault tolerant model* [DN95, BBP23, Bha25]. In 1992, [VY92] designed an algorithm that computes a second (s,t)-mincut, using $\mathcal{O}(n^2)$ (s,t)-maxflow computations. Recently, [BBR25] have improved this bound by establishing an interesting relation between a second (s,t)-mincut and a global mincut (the least valued mincut among all pairs). Given these relations with (s,t)-maxflows and global mincuts, second (s,t)-mincuts are combinatorially quite interesting. Within a decade of the introduction of (s,t)-mincut [FF56], [GH61] addressed the problem of all-pairs mincuts, which has been settled recently. It is now natural to address the problem of computing second mincuts for all pairs of vertices, given that second (s,t)-mincuts were introduced more than three decades ago. This also acts as a natural generalization to the all-pairs mincuts problem. However, no non-trivial algorithm exists for computing all-pairs second mincuts even for $F = c$.

Recently, in the setting of arbitrary cut functions, a combinatorial bound has been established on all-pairs second mincuts [BR26], stated as follows. There exists a set of $\mathcal{O}(n \log n)$ cuts such that for any pair (u,v), there exists a second (u,v)-mincut that belongs to the set. We call this set a *second mincut cover*. [BR26] show this bound to be optimal upto a $\mathcal{O}(\log n)$ factor and also provide a randomized construction to build this set. However, this construction requires $\binom{n}{2}$ second mincut computations, which is trivial. Generalizing our algorithmic framework for all-pairs mincuts, we present the first efficient algorithm for computing all-pairs second mincuts. The algorithm is deterministic and the number of second mincut computations it performs matches exactly the size of the second mincut cover, stated in [BR26].

Theorem 4 (All-Pairs Second Mincuts)**.** *Let $G = (V,E)$ be a graph on n vertices where the value of each cut C is given by any arbitrary real valued function F such that $F(C) = F(V \setminus C)$. Let $T_2(s,t)$ denote the time complexity of computing a second (s,t)-mincut in G. There exists an algorithm that given a mincut cover for all pairs, computes all-pairs second mincuts in $\mathcal{O}(n \log n T_2(s,t))$ time.*

Although our framework acts as the building block for Theorem 4, its components require significant generalization, especially, the tool EliminateTriangle. Overall, the tool aims at computing (second) mincut for three pairs in a triangle using at most two (second) mincut computations. For mincuts, this is achieved quite easily since there can be at most two distinct mincut values for the three pairs in a triangle. However, the second mincut values can be distinct for all the three pairs [BR26]. Moreover, Theorem 2, which plays the key role in extending EliminateTriangle from 3 to all n vertices, fails to hold for second mincuts. Interestingly, we overcome these challenges by *strongly* using the all-pairs mincuts. Firstly, we provide a novel classification of all triangles into two types based on mincut values. Secondly, we generalize Theorem 2 for second mincuts, by picking the two subsets V_1, V_2 in a *specific* order, instead of picking them arbitrarily. Moreover, due to this order, this algorithm closely resembles the Kruskal's [Kru56] algorithm for MST.

In the classical setting ($F = c$), a second (u,v)-mincut can be computed in $\sqrt{n}m^{1+o(1)}$ time [BBR25]. Hence, using this algorithm and Theorem 4, all-pairs second mincuts can be computed in $n^{1.5}m^{1+o(1)}$ time for undirected graphs. Given that Gomory-Hu tree can be now computed in $m^{1+o(1)}$

time [ALPS23], one may wonder whether a better running time can be achieved for computing all-pairs second mincuts. However, the main property which led to so many efficient algorithms for computing all-pairs mincuts in undirected graphs, that is, the *uncrossing* property of mincuts, fails to hold for second mincuts. More precisely, in a second (u,v)-mincut $C^2_{u,v}$, although $x,u \in C^2_{u,v}$, there might not exist any second (x,u)-mincut $C^2_{x,u}$ such that $C^2_{x,u} \subseteq C^2_{u,v}$, as shown in Figure 1$(i)$. The reason is $C^2_{u,v} \cap C^2_{x,u}$ and $C^2_{u,v} \cup C^2_{x,u}$ are not second mincuts (but mincuts) for the pairs (x,u) and (u,v), even though, submodularity of cuts hold.

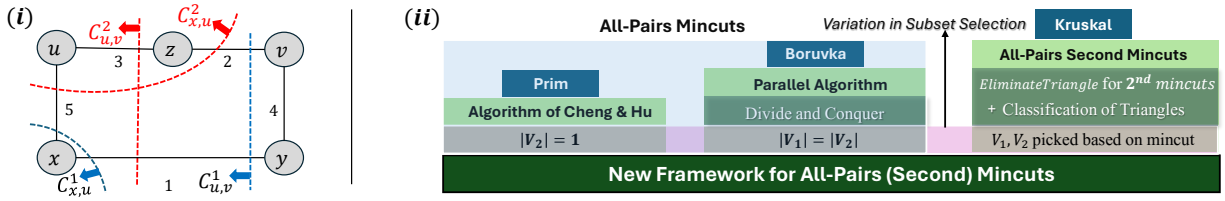


Figure 1: (i) Second mincuts $C^2_{u,v}, C^2_{x,u}$ for the pairs $(u,v),(x,u)$ are unique. (ii) Our framework.

2 Preliminaries

- A cut C is said to be a second (s,t)-mincut if C is an (s,t)-cut of minimum value among the (s,t)-cuts, whose value is strictly greater than that of (s,t)-mincut.
- $\lambda_1(u,v)$ denotes the value of (u,v)-mincut. Likewise, $\lambda_2(u,v)$ denotes the value of second (u,v)-mincut.
- $\mathcal{G}_1$ (likewise $\mathcal{G}_2$) denotes the complete graph on the set of vertices V where the capacity of each edge (u,v) is equal to $\lambda_1(u,v)$ (likewise $\lambda_2(u,v)$).
- $V(T)$ denotes the set of vertices in a tree T.
- For any graph H and subset of vertices $X \subset V$, $H[X]$ denotes the subgraph induced by X in H, that is the subgraph with vertex set X and edges from H with both endpoints in X.

Lemma 1 (Cycle Property)**.** *Let H be any graph and O be any cycle in H. There exists an MST of H that does not contain the least capacity edge in O.*

Lemma 2 (Upper Bound on Second Mincut [BR26])**.** *Let (u,v) be any pair of vertices and C be any cut that separates (u,v). Then $\lambda_1(u,v) = F(C)$ or $\lambda_2(u,v) \leq F(C)$.*

Lemma 3 (3-pairs Mincuts [GH61, CH91])**.** *Let x,y,z be any three vertices. If (x,z) has the least mincut value among the three pairs $(x,y),(y,z)$ and (x,z), then $\lambda_1(x,z) = \min\{\lambda_1(x,y), \lambda_1(y,z)\}$.*

Observe that although a mincut exists for every pair of vertices, there exist graphs where a second mincut might not exist for some pairs. However, this issue can be addressed easily (refer to Appendix B). Henceforth, we assume that a second mincut exists for all pairs of vertices.

3 Algorithmic Framework for All-Pairs Mincuts

We design a framework for computing all-pairs mincuts in the model where one can make the following *mincut query – report a (u,v)-mincut for any given $u,v \in V$*. The aim is to minimize the number of these queries. Our framework focuses on computing an MST for $\mathcal{G}_1$, since by Theorem 1, it is the same as computing the mincut value for each pair of vertices. However, we cannot use classic MST algorithms on $\mathcal{G}_1$ as they require that all the edge capacities of $\mathcal{G}_1$ are known. The main

insight underlying our framework is that we can afford to prune away most of the edges in $\mathcal{G}_1$ using the following property.

Property 1. *Let $C^1_{u,v}$ be an (u,v)-mincut in G. In $\mathcal{G}_1$, (u,v) is the maximum capacity edge in $C^1_{u,v}$.*

This insight allows us to obtain a *small* set of edges, called the *proxy edges*, which have the following property. Computing an MST on the proxy edges serves as an MST of $\mathcal{G}_1$. Moreover, a mincut cover for the proxy edges serves as the mincut cover for all pairs. Quite surprisingly, we later show that such a *small* set of proxy edges exists for all-pairs second mincuts as well. Our goal is to thus show their existence and how to compute them efficiently.

3.1 Components of our Framework

The *cycle property* (Lemma 1) is one of the well-known properties of MST. Consider any cycle of length 3 in $\mathcal{G}_1$. We show how to eliminate the least capacity edge in this cycle by making a single mincut query. Moreover, the mincut of the least capacity edge coincides with the mincut of one of the other two edges. To achieve these goals, we begin with following lemma.

Lemma 4. *Let u, v, and x be any three vertices, and $C^1_{u,v}$ be any (u,v)-mincut. If $C^1_{u,v}$ keeps x on the side of u, exactly one of the following assertions hold.*

1. *$\lambda_1(x,v) = \lambda_1(u,v)$ and $C^1_{u,v}$ is a (x,v)-mincut.*
2. *$\lambda_1(x,v) < \lambda_1(u,v)$, and any cut is a (x,v)-mincut if and only if it is a (x,u)-mincut.*

Proof. Observe that $C^1_{u,v}$ is a (x,v)-cut. So $\lambda_1(x,v) \leq \lambda_1(u,v)$. If $\lambda_1(x,v) = \lambda_1(u,v)$, Assertion 1 holds. Let us now consider the case when $\lambda_1(x,v) < \lambda_1(u,v)$. Let $C^1_{x,v}$ be an (x,v)-mincut. It follows that $C^1_{x,v}$ cannot separate u, v and hence $C^1_{x,v}$ keeps (u,v) on the same side. So, $C^1_{x,v}$ is a (x,u)-cut and $\lambda_1(x,u) \leq \lambda_1(x,v) < \lambda_1(u,v)$. Since $\lambda_1(x,u) < \lambda_1(u,v)$, any (x,u)-mincut cannot separate (u,v) as well and hence is a (x,v)-cut. So, $\lambda_1(x,v) \leq \lambda_1(x,u)$. Therefore, $\lambda_1(x,v) = \lambda_1(x,u)$. Moreover, since $\lambda_1(x,v) = \lambda_1(x,u) < \lambda_1(u,v)$, neither any (x,u)-mincut nor any (x,v)-mincut can separate (u,v). This completes the proof of Assertion (2). □

Note 1. *A weaker version of Lemma 4 appeared as Lemma 2 in [CH91], which is stated only for the mincut values of the three pairs.*

Suppose we have computed an (u,v)-mincut $C^1_{u,v}$. Now only using this cut, we have to eliminate the least capacity edge in the triangle $\langle u,v,x\rangle$ in $\mathcal{G}_1$. We accomplish this task using Algorithm 1, which is an algorithmic version of Lemma 4. It also plays a crucial role in each of our subsequent algorithms for computing all-pairs mincuts.

Algorithm 1 Computing least capacity edge in a triangle in $\mathcal{G}_1$

```
1: procedure EliminateTriangle(u, v, x, C^1_{u,v})
2:     if x ∈ C^1_{u,v} then
3:         return (x, v);          ▷ (x, v) is the least weight edge in triangle ⟨u, v, x⟩
4:     else
5:         return (x, u);          ▷ (x, u) is the least weight edge in triangle ⟨u, v, x⟩
6:     end if
7: end procedure
```

The following lemma, which can be viewed as a corollary of Lemma 4, states an important property of Algorithm 1. This also acts as the bridge between MSTs and mincuts.

Lemma 5. *Given any pair of vertices (u, v), mincut $C^1_{u,v}$, and any vertex $x \notin \{u, v\}$, Algorithm 1 reports in $\mathcal{O}(1)$ time a pair $(x, a), a \in \{u, v\}$, satisfying the following properties.*

1. *Either $C^1_{u,v}$ is (x, a)-mincut or every (x, b)-mincut is a (x, a)-mincut, where $b \in \{u, v\} \setminus \{a\}$.*
2. *(x, a) is the least capacity edge in the triangle formed by u, v, x in $\mathcal{G}_1$.*

It follows from Lemma 5(1) that we can compute a mincut for the three pairs in a triangle using only two mincut computations. We now extend Lemma 5 from a pair of vertices to any set of vertices. This extension is used to arrive at Theorem 2.

Lemma 6 (Pruning Edges)**.** *Given any subset of vertices V_2, let T_2 be any MST of $\mathcal{G}_1[V_2]$ and $\mathcal{C}_2$ be a mincut cover for all pairs $V_2 \times V_2$. Let $x \notin V_2$ be any vertex. There exists an algorithm that reports in $\mathcal{O}(|V_2|)$ time, an edge from x to V_2, denoted by $e_p(x, T_2)$, satisfying the following properties.*

1. *For any vertex $v \in V_2$, either (x, v)-mincut belongs to $\mathcal{C}_2$ or every $e_p(x, T_2)$-mincut is a (x, v)-mincut.*
2. *$e_p(x, T_2)$ is the maximum capacity edge among edges in $\{(x, v)|v \in V_2\}$ in $\mathcal{G}_1$.*

Proof. The algorithm starts with all the edges $\{(x, v)|v \in V_2\}$. Pick any two edges arbitrarily, say (x, u) and (x, v). Since $u, v \in V_2$, retrieve an (u, v)-mincut $C^1_{u,v}$ from $\mathcal{C}_2$. Invoke EliminateTriangle$(u, v, x, C^1_{u,v})$ (Algorithm 1) and remove one the edges. Without loss of generality, assume (x, v) is removed. It follows from Lemma 5 that (x, v)-mincut is either in $\mathcal{C}_2$ or every (x, u)-mincut is a (x, v)-mincut and $\lambda_1(x, u) \geq \lambda_1(x, v)$. Repeat this procedure $|V_2| - 1$ times till a single edge is left. This edge is $e_p(x, T_2)$ and satisfies the properties in Lemma 5. □

Suppose we are given MSTs T_1, T_2 of two disjoint vertex sets V_1, V_2 in $\mathcal{G}_1$. Alongside, we are given a mincut cover $\mathcal{C}$ for all the pairs in $V_1 \times V_1$ and $V_2 \times V_2$. We now want to compute (1) a mincut cover for all the inter-tree edges, that is, (x, v), where $x \in V_1, v \in V_2$ and (2) an MST for $V_1 \cup V_2$ in $\mathcal{G}_1$. To achieve the first objective, it follows from Lemma 6(1) that computing mincut for each edge in $\{e_p(x, T_2)|x \in V_1\}$ suffices. This leads to Theorem 2(1). Next, we want to compute an MST for $\mathcal{G}_1[V_1 \cup V_2]$. Suppose $(x, u) = e_p(x, T_2)$, where $u \in V_2$. It follows from Lemma 6(2) that $e_p(x, T_2) \cup T_2$ is an MST of $\mathcal{G}_1[x \cup V_2]$. Moreover, for any other $v \in V_2$, where $v \neq u$, (x, v) is the least weight edge in the triangle u, v, x. This observation along with the Cycle Property (Lemma 1) implies the following. Computing an MST on the set of edges $T_1 \cup T_2 \cup \{e_p(x, T_2)|x \in V_1\}$ serves as an MST for $\mathcal{G}_1[V_1 \cup V_2]$; hence, leading to Theorem 2(2). Thus, we can achieve both our objectives using these steps. We provide a pseudocode for these steps in Algorithm 2, which can be seen as an algorithmic version of Theorem 2.

Algorithm 2 Merging two trees and computing Mincut Cover for inter-tree edges

```
1: procedure MergeTrees(T_1, T_2, C, E_p)        ▷ E_p is the set of proxy edges from T_1 to T_2
2:     If |V(T_1)| > |V(T_2)|, swap T_2 and T_1;
3:     for each u ∈ V(T_1) do                    ▷ Computing mincut for the edges from T_1 to T_2
4:         Let (u, v) ← e_p(u, T_2); Compute a (u, v)-mincut C^1_{u,v};   ▷ Proxy edge from u to T_2
5:         C ← C ∪ {C^1_{u,v}};
6:     end for
7:     Compute an MST T' on T_1 ∪ T_2 ∪ {e_p(u, T_2)|u ∈ V(T_1)};   ▷ Use any MST algorithm
8:     return {T', C};
9: end procedure
```

3.2 Framework

The framework computes an MST for $\mathcal{G}_1$ and also a mincut cover $\mathcal{C}$ in $n-1$ iterations. At the end of ith iteration, there is a forest $\mathcal{F}(i)$ of spanning trees satisfying the following properties.

1. Each tree $T \in \mathcal{F}(i)$ is an MST on the graph $\mathcal{G}_1[V(T)]$.
2. For each pair (u, v) of vertices in the same tree in $\mathcal{F}(i)$, a mincut is stored in set $\mathcal{C}$.
3. For a vertex x and a tree $T \in \mathcal{F}(i)$ such that $x \notin V(T)$, we store a proxy edge $e_p(x, T)$ (Lemma 6). Refer to Figure 2(i). The set of proxy edges in iteration i is denoted by $E_p(i) = \{e_p(x, T) | x \in V, T \in \mathcal{F}(i) \text{ s.t. } x \notin V(T)\}$.

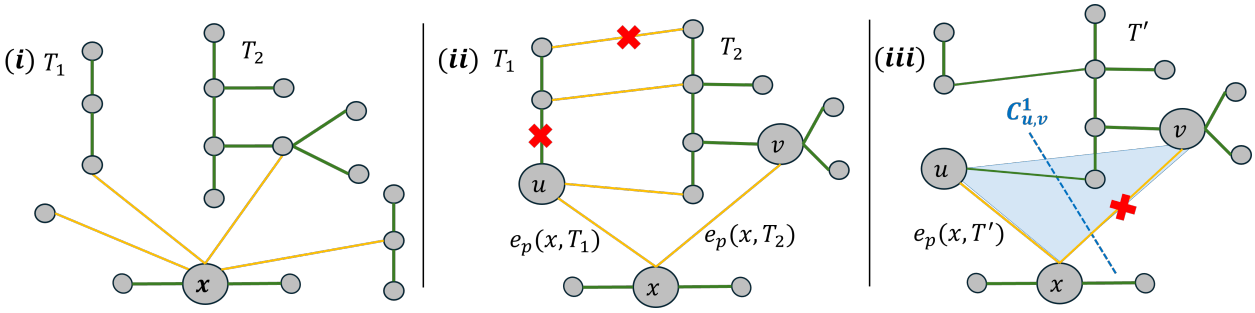


Figure 2: (i) A vertex x has exactly one edge to every other tree (ii) Trees T_1, T_2 are joined into T' (iii) $e_p(x, T')$ is computed using EliminateTriangle$(u, v, x, C^1_{u,v})$.

At the beginning of the framework, $\mathcal{F}(0)$ is a forest on n singleton vertices, $\mathcal{C}$ is empty and $E_p(0)$ is set of all pairs of vertices. While at the end of the framework, $\mathcal{F}(n-1)$ is an MST for $\mathcal{G}_1$, $\mathcal{C}$ is a mincut cover for all pairs and $E_p(n-1)$ is empty. We now describe the ith iteration. Pick any two trees, say T_1, T_2, arbitrarily from $\mathcal{F}(i-1)$ and perform the following two tasks to maintain the above three invariants.

(1)Merging two trees: Use MergeTree$(T_1, T_2, \mathcal{C})$ to obtain an MST, say T', for $\mathcal{G}_1[V(T_1) \cup V(T_2)]$ and an updated mincut cover $\mathcal{C}$ such that for each $u, v \in V(T')$, an (u, v)-mincut is stored in $\mathcal{C}$. Figure 2(ii) and (iii) shows the transformation of T_1, T_2 into T'. Observe that this maintains the first two invariants.

(2)Pruning Edges: Any vertex x outside T' has two proxy edges (instead of one) to the new tree T': $e_p(x, T_1)$ and $e_p(x, T_2)$. We use EliminateTriangle (Lemma 5) to prune the proxy edge with least capacity in $\mathcal{G}_1$ and assign the other edge as $e_p(x, T')$. Figure 2(iii) illustrates this step. This maintains the third invariant.

While merging two trees T_1, T_2, the second invariant ensures that an (u, v)-mincut $C^1_{u,v} \in \mathcal{C}$, where $u \in V(T_1), v \in V(T_2)$. However, we need to retrieve $C^1_{u,v}$ efficiently for pruning one of $(x, u), (x, v)$ for every $x \notin V(T')$. Hence, we need to assign a pointer from (u, v) to $C^1_{u,v}$ in $\mathcal{C}$ after MergeTree(T_1, T_2). This task is accomplished by AssignMincut$(T_1, T_2, \mathcal{C})$ (Line 12 in Algorithm 3). Its implementation takes $\mathcal{O}(n^2)$ time during the entire algorithm (refer to Appendix D.1).

3.3 Analysis

The three invariants of Algorithm 3 ensure that $\mathcal{F}(n-1)$ is an MST for $\mathcal{G}_1$ and $\mathcal{C}$ is a mincut cover for all pairs. We now provide a bound on the running time of the algorithm.

Bound on the number of mincut computations: We provide an upper bound on the number of mincut computations in Algorithm 3 using a charging argument. Observe that mincut computations are performed only when two trees T_1, T_2 get merged. Without loss of generality, assume $|V(T_1)| \leq |V(T_2)|$.

Algorithm 3 Algorithmic Framework for computing all-pairs mincuts

1: **procedure** AllPairsMincut(G)
2: Forest $\mathcal{F}(0)$ consists of n trees: each tree is a single vertex from V, $\mathcal{C} \leftarrow \emptyset$;
3: **for** each $v \in V$ and $T \in \mathcal{F}(0)$ where $v \notin V(T)$ **do**
4: $e_p(v, T) \leftarrow$ edge between v and the vertex in T;
5: **end for**
6: **for** $i = 1$ to $n - 1$ **do**
7: *Merging two trees:*
8: Let (a, b) be any edge in $E_p(i-1)$; ▷ Picking any two arbitrary trees
9: Let $a \in V(T_1), b \in V(T_2)$, where $T_1, T_2 \in \mathcal{F}(i-1)$;
10: $\{T', \mathcal{C}\} \leftarrow$ MergeTree$(T_1, T_2, \mathcal{C})$; ▷ Joining Trees and updating $\mathcal{C}$
11: $\mathcal{F}(i) \leftarrow (\mathcal{F}(i-1) \setminus \{T_1, T_2\}) \cup T'$; ▷ Updating $\mathcal{F}$
12: AssignMincut$(T_1, T_2, \mathcal{C})$; ▷ Assigns a mincut to $(u, v), \forall u \in V(T_1), v \in V(T_2)$
13: *Pruning Edges:*
14: **for** each $x \notin V(T')$ **do** ▷ Removing $e_p(x, T_1)$ and $e_p(x, T_2)$ from E_p
15: Let $(x, u) \leftarrow e_p(x, T_1)$ and $(x, v) \leftarrow e_p(x, T_2)$;
16: $e_p(x, T') \leftarrow \{(x, u), (x, v)\} \setminus$ EliminateTriangle$(u, v, x, C^1_{u,v})$;
17: **end for**
18: **end for**
19: **end procedure**

We charge a vertex x in a tree T_1 one unit since exactly one mincut computation is done for x, which is for the proxy edge $e_p(x, T_2)$. Observe that, when T_1 is merged with T_2, the size of the new tree is $|V(T_1)| + |V(T_2)| \geq 2|V(T_1)|$, that is at least double the size of the smaller tree. Hence, the total charge any vertex accumulates during the algorithm is $\mathcal{O}(\log n)$. Therefore, a total of $\mathcal{O}(n \log n)$ mincut computations are performed in Algorithm 3.

Running Time of Algorithm 3: Let us analyze the running time of MergeTree over all the $n - 1$ iterations. Each MergeTree computes mincut for a set of proxy edge and updates the MST using these edges. As shown above, the number of such proxy edges is $\mathcal{O}(n \log n)$ over all iterations. Hence, using the dynamic ST-trees of Sleator and Tarjan [ST81], the time spent in updating the MST during the entire algorithm is $\mathcal{O}(n \log^2 n)$. Since AssignMincut can be implemented in $\mathcal{O}(n^2)$ time, let us now analyze the second part of the algorithm: *Pruning Edges*. It follows from Lemma 5 that EliminateTriangle can be implemented in $\mathcal{O}(1)$ time for each vertex. Hence, during each iteration, *Pruning Edges* requires $\mathcal{O}(n)$ time. Therefore, the total running time of Algorithm 3 is equal to $\mathcal{O}(n \log n)$ mincut computations with additional $\mathcal{O}(n^2)$ time.

4 Applications of Our Framework for All-Pairs Mincuts

In this section, we use the flexibility of our framework to obtain two new algorithms for all-pairs mincuts. We also show their resemblance to the well-known MST algorithms.

4.1 Algorithm of Cheng and Hu [CH91] as a special case of our framework

Faster Algorithm for All-Pairs Mincuts: Refer to Line 8,9 in Algorithm 3. Instead of picking any two trees T_1, T_2 arbitrarily, choose T_1 to be the tree with the maximum number of vertices. As

a result, T_2 would always be a singleton tree during the algorithm. Moreover, we would need to compute mincut for exactly one proxy edge in each iteration. Hence, this algorithm requires $n-1$ mincut computations which matches the bound obtained by [CH91]. Interestingly, we now show that this algorithm is in fact same as the algorithm of Cheng and Hu [CH91], albeit when viewed from the perspective of an MST computation.

The algorithm for all-pairs mincuts by Cheng and Hu [CH91] computes an ancestor tree incrementally over $n-1$ iterations. It is a rooted binary tree whose leaf nodes are the vertices in G and for any pair of vertices (u, v), the lowest common ancestor (LCA) of u and v stores a (u, v)-mincut. The algorithm classifies all vertices into *seed* and *unseed* vertices. At the beginning of ith iteration, the ancestor tree has $i-1$ internal nodes and i leaf nodes. (1) Each internal node stores a cut and a pair of seed vertices. These pairs form a spanning tree on i seed vertices. This tree is identical to the non-singleton tree in our algorithm while the $n-i$ unseed vertices correspond to the singleton trees. (2) The i leaf nodes store a partition of V such that the following hold. Each leaf node contains exactly one seed vertex and each unseed vertex is mapped to a single seed vertex. This mapping is captured by the proxy edges in our algorithm: each singleton tree has exactly one proxy edge to a vertex in the non-singleton tree. We provide an overview of the algorithm of [CH91] and an elaborate description of these similarities in Appendix C.

Resemblance to the Prim's Algorithm [Pri57]: At any iteration, the Prim's algorithm keeps only the maximum capacity edge from the partially grown MST T to each external vertex, say x. This edge is identical to the proxy edge $e_p(x, T)$ that our algorithm keeps for vertex x (Lemma 6(2)). Moreover, similar to the Prim's algorithm, our faster algorithm grows the MST for $\mathcal{G}_1$ by adding one external vertex in each of the $n-1$ iterations. However, unlike Prim's algorithm where we might have a unique external vertex included in the tree, our algorithm allows inclusion of *any* external vertex.

4.2 First Parallel Algorithm for All-Pairs Mincuts

At first sight, Algorithm 3 appears to be sequential due to the following reason. During the ith iteration, while invoking MergeTree on two trees, say T_1, T_2, we need the proxy edge from each vertex in T_1 to T_2. Each such edge is computed in the jth iteration, for some $j < i$. However, observe that we can afford to compute the proxy edges between the two trees in a lazy manner. This flexibility in computing proxy edges, along with the flexibility of picking any arbitrary subsets of vertices in Algorithm 3 allows us to design a divide-and-conquer algorithm for all-pairs mincuts. Refer to Algorithm 4 for the pseudo-code.

Recall that there are three procedures in our framework: MergeTree, AssignMincut and PruneEdges (computing proxy edges between two given trees). In MergeTree (Algorithm 2), observe that the mincut computations for the proxy edges can be performed concurrently. For the other two procedures PruneEdges and AssignMincut, we provide a divide and conquer implementation in Appendix D that achieves logarithmic depth and linear work. They are stated as DACPruneEdges and DACAssignMincut in Algorithm 4. Since, the recursion depth of Algorithm 4 is $\mathcal{O}(\log n)$, we can seamlessly implement Algorithm 4 in parallel CREW model so that it achieves the bounds stated in Theorem 3. A detailed implementation along with analysis is provided in Appendix E.

Resemblance to the Borůvka's Algorithm [Bor26]: Borůvka's algorithm takes a bottom up approach to computing MST and terminates in at most $\mathcal{O}(\log n)$ rounds. This is similar to our algorithm, which also has $\mathcal{O}(\log n)$ recursion depth. However, at any round, the choice of the two trees that are merged in Borůvka's algorithm, is based on the edge capacities (it picks the maximum capacity edge incident on each tree). Whereas, our algorithm pairs the trees arbitrarily. This offers

Algorithm 4 Divide and Conquer Algorithm for computing All-Pairs Mincuts

procedure DACforAllPairsMincut(G)
  AllPairsMincutforSubset(G, V);
end procedure
procedure AllPairsMincutforSubset(G, S)
  if $|S| \leq 2$ **then** ▷ Base Case
    if $|S| = 2$ **then** Let $S = u, v$;
    Compute a (u, v) mincut $C^1_{u,v}$ and T be the edge (u, v) with capacity $\lambda_1(u, v)$;
    return $\{T, C^1_{u,v}\}$; **else return** $\{S, \emptyset\}$;
  else
    Partition S into 2 equal sized sets: A, B; ▷ Bounds recursion depth by $\mathcal{O}(\log n)$
    $\{T_A, \mathcal{C}_A\} \leftarrow$ AllPairsMincutforSubset(G, A); ▷ MST and mincut cover for set A
    $\{T_B, \mathcal{C}_B\} \leftarrow$ AllPairsMincutforSubset(G, B); ▷ MST and mincut cover for set B
    $E_p \leftarrow$ DACPruneEdges($T_A, T_B, \mathcal{C}_A, \mathcal{C}_B$); ▷ $\forall x \in V(T_A)$, compute $e_p(x, T_B)$
    $\{T, \mathcal{C}\} \leftarrow$ MergeTree($T_A, T_B, \mathcal{C}_A \cup \mathcal{C}_B, E_p$); ▷ Obtain MST for $A \cup B$
    DACAssignMincut($T_A, T_B, \mathcal{C}_A \cup \mathcal{C}_B$); ▷ Assigns a mincut to $(x, v), \forall x \in A, v \in B$
    return $\{T, \mathcal{C}\}$;
  end if
end procedure

more flexibility than Borůvka's algorithm but comes at the following cost. In Borůvka's algorithm, whenever two trees T_A, T_B are merged, T_A, T_B are treated as singleton vertices and only a single new edge is added between T_A and T_B to obtain a new tree. This is not true for our algorithm. Nevertheless, at most $\min\{|T_A|, |T_B|\}$ edges (Theorem 2) are added between T_A, T_B to obtain T.

Remark 1. *The divide and conquer strategy used in Algorithm 4 is quite standard in the algorithms community. However, the novelty lies in showing that we can efficiently merge the mincut cover of two arbitrary subsets A, B, which is possible due to Theorem 2.*

5 First algorithm for All-Pairs Second Mincuts

In this section, we design an algorithm that computes an MST of $\mathcal{G}_2$ and a second mincut cover of size $\mathcal{O}(n \log n)$. Our framework for all-pairs mincuts is centered around the pruning of edges. Notably, MergeTree depends on it to get a small number of proxy edges between trees. In turn, the pruning of edges is done using EliminateTriangle. In order to extend our framework for all-pairs second mincuts, we first generalize EliminateTriangle for second mincuts (Section 5.1). However, unlike mincuts, this generalized EliminateTriangle can process a triangle only in a certain orientation. To ensure this constraint, we investigate into the structure of mincuts lying underneath any two trees, albeit picked in a *specific* order, unlike Algorithm 3. Once we show this in Section 5.3, MergeTree is seamlessly generalized to obtain our algorithm.

5.1 Eliminating Triangles in $\mathcal{G}_2$

Let x, u, v be any three vertices. Our goal is to find a second mincut for all the three pairs, namely, $(u, v), (x, u), (x, v)$, by using at most two second mincut computations. For the case of mincuts, we achieve this goal through Algorithm 1, derived using Lemma 4. However, a natural extension of Lemma 4 for second mincuts fails to achieve our goal as follows. Suppose we pick any arbitrary pair,

say (u, v), and let $C^2_{u,v}$ be any second (u, v)-mincut. Suppose $C^2_{u,v}$ separates (x, v). Unlike Lemma 4, second (x, v)-mincut might neither be $C^2_{u,v}$ nor a second (x, u)-mincut. The reason is, among the pairs $(u, v), (x, v), (x, u)$, while the smallest two mincut values always coincide (Lemma 3), the second mincut values can be different for each of the three pairs, as shown in [BR26].

Our main insight is that we can use the mincut of the three pairs to achieve our goal. Among the three pairs, suppose (u, v) has the highest mincut value among the three pairs. It follows from Lemma 3 that $\lambda_1(u, v) \geq \lambda_1(x, u) = \lambda_1(x, v)$. We first consider the case when $\lambda_1(u, v) > \lambda_1(x, u) = \lambda_1(x, v)$. Observe that any (u, v)-mincut $C^1_{u,v}$ must separate (x, u) or (x, v). Interestingly, combining this observation with Lemma 2, we arrive at the following property.

Lemma 7 (Type 1)**.** *Suppose $\lambda_1(u, v) > \lambda_1(x, u) = \lambda_1(x, v)$. If $C^1_{u,v}$ keeps x on the side of u, exactly one of the following assertions hold.*

1. *$\lambda_2(x, v) = \lambda_1(u, v)$ and $C^1_{u,v}$ is a second (x, v)-mincut.*
2. *$\lambda_2(x, v) < \lambda_1(u, v)$, and any cut is a second (x, v)-mincut if and only if it is a second (x, u)-mincut.*

Proof. Since $C^1_{u,v}$ is a (x, v)-cut and $\lambda_1(u, v) > \lambda_1(x, v)$, it follows from Lemma 2 that $\lambda_2(x, v) \leq \lambda_1(u, v)$. If $\lambda_2(x, v) = \lambda_1(u, v)$, Assertion *1* holds. Let us now consider the case when $\lambda_2(x, v) < \lambda_1(u, v)$. Let $C^2_{x,v}$ be a second (x, v)-mincut. It follows that $C^2_{x,v}$ cannot separate (u, v). Hence, $C^2_{x,v}$ must separate (x, u). Since $\lambda_1(x, u) = \lambda_1(x, v)$, it follows from Lemma 2 that $\lambda_2(x, u) \leq \lambda_2(x, v)$. Since $\lambda_2(x, u) \leq \lambda_2(x, v) < \lambda_1(u, v)$, any second (x, u)-mincut cannot separate (u, v) as well and hence is a (x, v)-cut. So, again by Lemma 2, we have $\lambda_2(x, v) \leq \lambda_2(x, u)$. Therefore, $\lambda_2(x, v) = \lambda_2(x, u)$. Moreover, since $\lambda_2(x, u) = \lambda_2(x, v) < \lambda_1(u, v)$, neither any second (x, u)-mincut nor any second (x, v)-mincut can separate (u, v). This completes the proof of Assertion *2*. □

It follows from Lemma 7 that we can compute the second mincut for the three pairs using only two second mincut computations when the mincut values of the three pairs are not same. Now, it follows from Lemma 3 that the only remaining case is when the mincut values of the three pairs are same. We address it in the following lemma, which can be seen as a natural extension of Lemma 4 for second mincuts. The proof makes use of the fact that if the mincut value for two pairs is same, the second mincut for one pair cannot be mincut for the other.

Lemma 8 (Type 2)**.** *Suppose $\lambda_1(u, v) = \lambda_1(x, u) = \lambda_1(x, v)$. If $C^2_{u,v}$ keeps x on the side of u, exactly one of the following assertions hold.*

1. *$\lambda_2(x, v) = \lambda_2(u, v)$ and $C^2_{u,v}$ is a second (x, v)-mincut.*
2. *$\lambda_2(x, v) < \lambda_2(u, v)$, and any cut is a second (x, v)-mincut if and only if it is a second (x, u)-mincut.*

Proof. Since $C^2_{u,v}$ is a (x, v)-cut and $\lambda_1(u, v) = \lambda_1(x, v)$, it follows from Lemma 2 that $\lambda_2(x, v) \leq \lambda_2(u, v)$. If $\lambda_2(x, v) = \lambda_2(u, v)$, Assertion *1* holds. Let us now consider the case when $\lambda_2(x, v) < \lambda_2(u, v)$. Let $C^2_{x,v}$ be a second (x, v)-mincut. It follows from Lemma 2 that $C^2_{x,v}$ cannot separate (u, v). Hence, $C^2_{x,v}$ must separate (x, u). Since $\lambda_1(x, u) = \lambda_1(x, v)$, it again follows from Lemma 2 that $\lambda_2(x, u) \leq \lambda_2(x, v)$. Since $\lambda_2(x, u) \leq \lambda_2(x, v) < \lambda_2(u, v)$ and $\lambda_1(x, u) = \lambda_1(u, v)$, it follows from Lemma 2 any second (x, u)-mincut cannot separate (u, v) as well and hence is a (x, v)-cut. So, again by Lemma 2, we have $\lambda_2(x, v) \leq \lambda_2(x, u)$. Therefore, $\lambda_2(x, v) = \lambda_2(x, u)$. Moreover, since $\lambda_2(x, u) = \lambda_2(x, v) < \lambda_2(u, v)$, neither any second (x, u)-mincut nor any second (x, v)-mincut can separate (u, v). This completes the proof of Assertion *2*. □

It follows from Lemma 7 and Lemma 8, that we can classify any triplet u, v, x as follows. We say u, v, x is of Type1 if $\lambda_1(u, v) > \lambda_1(x, u) = \lambda_1(x, v)$. Otherwise if $\lambda_1(u, v) = \lambda_1(x, u) = \lambda_1(x, v)$, we

say u, v, x is of Type2. We now provide an algorithm, stated as Algorithm 5, that combines Lemma 7 and Lemma 8 to eliminate the least capacity edge in a triangle in $\mathcal{G}_2$.

Algorithm 5 Computing Least Capacity Edge in a Triangle in $\mathcal{G}_2$

```
1: procedure EliminateTriangle(u, v, x, C^1_{u,v}, C^2_{u,v})
2:     if λ1(u, v) > λ1(x, u) then                                  ▷ Type1
3:         if x ∈ C^1_{u,v} then return (x, u); else return (x, v);
4:     else if λ1(u, v) = λ1(x, u) then                             ▷ Type2
5:         if x ∈ C^2_{u,v} then return (x, u); else return (x, v);
6:     end if
7: end procedure
```

Lemma 9. *Given any pair of vertices (u, v), $C^1_{u,v}$, $C^2_{u,v}$, and any vertex $x \notin \{u, v\}$ such that $\lambda_1(u, v) \geq \lambda_1(x, u) = \lambda_1(x, v)$, Algorithm 5 reports in $\mathcal{O}(1)$ time, a pair $(x, a), a \in \{u, v\}$, that is the least capacity edge in the triangle formed by u, v, x in $\mathcal{G}_2$.*

5.2 Generalizing Subroutines of the Framework

We now show the existence of proxy edges for second mincuts, thus generalizing Lemma 6. Its proof is immediate from Lemma 7, Lemma 8, Lemma 9 and the proof of Lemma 6.

Lemma 10 (Proxy Edge for 2^{nd} Mincut)**.** *Let V_2 be any subset of vertices and $x \notin V_2$ be any vertex such that for any pair of vertices $(u, v) \in V_2$, $\lambda_1(u, v) \geq \lambda_1(x, u) = \lambda_1(x, v)$. Let T_2 be any MST of $\mathcal{G}_2[V_2]$ and $\mathcal{C}_2$ be a set consisting of a second mincut cover and a mincut cover for all pairs $V_2 \times V_2$. There exists an algorithm that reports in $\mathcal{O}(|V_2|)$ time, an edge from x to V_2, denoted by $e_p(x, T_2)$, satisfying the following properties.*

1. *For any vertex $v \in V_2$, either second (x, v)-mincut belongs to $\mathcal{C}_2$ or every second $e_p(x, T_2)$-mincut is a second (x, v)-mincut.*
2. *$e_p(x, T_2)$ is the maximum capacity edge among edges in $\{(x, v) | v \in V_2\}$ in $\mathcal{G}_2$.*

Suppose we are given two MSTs T_1, T_2 for two disjoint vertex sets V_1, V_2 in $\mathcal{G}_2$ such that the following holds. The mincut value of all pairs in $V_1 \times V_1$ and $V_2 \times V_2$ is greater than the mincut value of pairs in $V_1 \times V_2$. Since this satisfies the premise of Lemma 9, we can easily show using Cycle Property (Lemma 1) that $T_1 \cup T_2 \cup \{e_p(x, T_2) | x \in V_1\}$ serves as an MST for $\mathcal{G}_2[V_1 \cup V_2]$. Moreover, suppose $\mathcal{C}$ is a set containing the mincut cover and second mincut cover for all $V_1 \times V_1$ pairs and all $V_2 \times V_2$ pairs. Then, to compute a second mincut for all the inter-tree edges (x, v) where $x \in V_1, v \in V_2$, it follows from Lemma 9 that computing a second mincut for each edge in $\{e_p(x, T_2) | x \in V_1\}$ suffices. Therefore, we can easily extend MergeTree seamlessly for second mincuts by just replacing mincut with second mincut in Lines 4 and 5 of Algorithm 2. The pseudocode is provided in Algorithm 6, for completeness.

5.3 Satisfying the Premise of Lemma 9

If the premise of Lemma 9 holds, then all the three components of our framework are extended for second mincuts (Lemma 10, Algorithm 6). Now to generalize our framework for all-pairs second mincuts, we only need to maintain the following invariants. At the end of ith iteration, there is a forest $\mathcal{F}(i)$ of spanning trees and a set of cuts $\mathcal{C}$ satisfying the following properties.

1. Each tree $T \in \mathcal{F}(i)$ is an MST on the graph $\mathcal{G}_2[V(T)]$.

Algorithm 6 Merging two trees and computing Second Mincut Cover for Inter-tree edges

procedure MERGETREE($T_1, T_2, \mathcal{C}, E_p$) ▷ E_p is the set of proxy edges from T_1 to T_2
  If $|V(T_1)| > |V(T_2)|$, swap T_2 and T_1;
  for each $u \in V(T_1)$ **do** ▷ Computing 2^{nd} mincut for proxy edges from T_1 to T_2
    Let $(u, v) \leftarrow e_p(u, T_2)$; Compute a second (u, v)-mincut $C^2_{u,v}$; ▷ *Proxy edge* from u to T_2
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{C^2_{u,v}\}$;
  end for
  Compute an MST T' on $T_1 \cup T_2 \cup \{e_p(u, T_2) | u \in V(T_1)\}$; ▷ Use any MST algorithm
  return $\{T', \mathcal{C}\}$;
end procedure

2. For each pair (u, v) of vertices in the same tree in $\mathcal{F}(i)$, a second mincut is stored in set $\mathcal{C}$.
3. For a vertex x and a tree $T \in \mathcal{F}(i)$ such that $x \notin V(T)$, we store a proxy edge $e_p(x, T)$ (Lemma 10). Moreover, define $E_p(i) = \{e_p(x, T) | x \in V, T \in \mathcal{F}(i) \text{ s.t. } x \notin V(T)\}$.

Suppose the above invariants hold at the end of $(i-1)$th iteration. During ith iteration, suppose we pick two trees T_1, T_2 from $\mathcal{F}(i-1)$ and merge them into a single tree T', using MERGETREE (Algorithm 6). For each vertex $x \notin V(T')$, we now have two proxy edges, say, $(x, u) = e_p(x, T_1)$ and $(x, v) = e_p(x, T_2)$. To maintain invariant 3, we now need to prune one of them. However, to use ELIMINATETRIANGLE, $\langle u, v, x \rangle$ must satisfy the premise of Lemma 9.

Our aim is to thus choose T_1 and T_2 so that for every (u', v') with $u' \in V(T_1), v' \in V(T_2)$ and any $x' \notin V(T')$, with $(x', u'), (x', v') \in E_p(i-1)$, $\lambda_1(u', v') \geq \lambda_1(x', u') = \lambda_1(x', v')$. However, this seems totally unclear since we do not have any visible relation among the mincut values of the edges $\{(u', v') | u' \in V(T_1), v' \in V(T_2)\}$, and we need to satisfy the premise for all of them at once. Clearly, we can no longer pick T_1, T_2 arbitrarily. However, if T_1, T_2 are chosen carefully, then the following insight into mincuts reveals an unifying property for the edges $\{(u', v') | u' \in V(T_1), v' \in V(T_2)\}$.

Lemma 11. *Let V_1 and V_2 be any two disjoint subsets of V such that the mincut value of each pair in $V_1 \times V_1$ and $V_2 \times V_2$ is greater than or equal to the mincut value of each pair in $V_1 \times V_2$. Then, the mincut values for all the pairs in $V_1 \times V_2$ are the same.*

Proof. Let α_1 (likewise α_2) be the smallest mincut value among all the pairs $V_1 \times V_1$ (likewise $V_2 \times V_2$). Now, suppose (a, b) has the largest mincut value, say α, among all the pairs $V_1 \times V_2$. Let (u, v) be any pair other than (a, b) such that $u \in V_1, v \in V_2$. It follows from the premise that $\lambda_1(u, v) \leq \alpha \leq \alpha_1$ and α_2. Let $C^1_{u,v}$ be any (u, v)-mincut. Observe that there are only two possibilities. $C^1_{u,v}$ separates a pair (x, y) where $x, y \in V_1$ or $x, y \in V_2$. Or, $C^1_{u,v}$ separates (a, b) since it does not separate any pair in $V_1 \times V_1$ or $V_2 \times V_2$. As $C^1_{u,v}$ is a cut for the pairs, we get the following upper bounds on their mincut values. In the first case, $\lambda_1(u, v) \geq \alpha_1$ or $\alpha_2 \geq \alpha$. While in second case, $\lambda_1(u, v) \geq \alpha$. Hence, we get $\lambda_1(u, v) = \alpha$. □

To make use of Lemma 11, we introduce a fourth invariant for our algorithm.

4. Mincut value of each pair with endpoints in the same tree is greater than or equal to the mincut value of each pair with endpoints in different trees in $\mathcal{F}(i-1)$.

We now show how we maintain this and the third invariant at the end of ith iteration. To ensure the premise of Lemma 9, we simply pick the edge, say (a, b), in $E_p(i-1)$ which has the highest mincut value. Since any edge in $E_p(i-1)$ is always between two different trees, we then pick the trees T_1, T_2 such that $a \in V(T_1)$ and $b \in V(T_2)$. Although, we pick only one edge between T_1, T_2, Lemma 11 ensures that the premise of Lemma 9 holds for every edge between T_1, T_2 (not just the

proxy edges). Hence, after merging T_1, T_2 into a new tree T' (using MERGETREE), we can now use ELIMINATETRIANGLE to prune one of $e_p(x, T_1)$ and $e_p(x, T_2)$, for each $x \in V(T')$. This ensures the invariant (3).

It follows from invariant (4) and Lemma 11 that for any two trees, say $T_A, T_B \in \mathcal{F}(i-1)$, all the edges between T_A, T_B have the same mincut value. In other words, the pruned edges between T_A, T_B will have the same mincut value as the proxy edges between T_A, T_B. Since we picked (a, b) to be the edge in $E_p(i-1)$ with highest mincut value, observe that Lemma 11 ensures that every edge between T_1, T_2 has mincut value greater than or equal to the mincut value of every edge between T_A, T_B for any $T_A, T_B \in \mathcal{F}(i-1)$. This ensures invariant (4) for the new tree T' in ith iteration.

Notice that although our aim is to compute an MST in $\mathcal{G}_2$, we process edges in decreasing order of their capacities in $\mathcal{G}_1$, which is quite surprising. We now conclude with a generalization of Theorem 2 for second mincuts. Its proof is immediate from Lemma 11 and the discussion in Section 5.2.

Theorem 5. *Let V_1 and V_2 be any two disjoint subsets of V such that the mincut value of all pairs $V_1 \times V_1$ and $V_2 \times V_2$ is greater than the mincut value of pairs $V_1 \times V_2$. Let $\mathcal{C}$ contain the mincut cover and second mincut cover for all $V_1 \times V_1$ pairs and all $V_2 \times V_2$ pairs. Let T_1 and T_2 be the MSTs of $\mathcal{G}_2[V_1]$ and $\mathcal{G}_2[V_2]$ respectively. Without loss of generality, assume, $|V_1| \leq |V_2|$. There exists a subset $P \subset V_1 \times V_2$ with $|P| = |V_1|$ such that the following properties hold.*

1. *Second mincut for each pair in P plus $\mathcal{C}$ is a second mincut cover for all-pairs from $V_1 \times V_2$.*
2. *an MST on edges $(T_1 \cup T_2 \cup P)$ is an MST for $\mathcal{G}_2[V_1 \cup V_2]$.*
3. *The mincut value is same for all pairs in $V_1 \times V_2$.*

5.4 Algorithm

We first compute a mincut cover for all pairs of vertices using Algorithm 3 or the algorithm of [CH91]. At the beginning of the algorithm, we store this cover in $\mathcal{C}$. Then, the algorithm proceeds as the pseudocode stated in Algorithm 7. Note that Algorithm 7 differs from Algorithm 3 only in Line 8.

5.5 Analysis

The invariants ensure that $\mathcal{F}(n-1)$ is an MST of $\mathcal{G}_2$ and $\mathcal{C}$ stores a mincut cover as well as a second mincut cover for all pairs of vertices. We now provide a bound on the running time of Algorithm 7.

The mincut values of all pairs of vertices can be sorted in non-increasing order using $\mathcal{O}(n^2 \log n)$ time. Let us analyze the running time of MERGETREE over all the $n-1$ iterations. While merging two trees T_A, T_B, where $T_A \leq T_B$, Algorithm 6 computes a second mincut for exactly one proxy edge from each vertex in A, similar to Algorithm 3. Hence, using the same charging argument used in the analysis of Algorithm 3, we can bound the number of second mincut computations to be $\mathcal{O}(n \log n)$. In Section D.2, we provide an implementation of ASSIGN2ndMINCUT in $\mathcal{O}(n^2)$ time. Let us now analyze the second part of the algorithm: *Pruning Edges*. It follows form Lemma 9 that ELIMINATETRIANGLE can be implemented in $\mathcal{O}(1)$ time for each vertex in each iteration. Therefore, the total running time of the algorithm is $\mathcal{O}(n \log n)$ second mincut computations with additional $\mathcal{O}(n^2 \log n)$ time. This completes the proof of Theorem 4.

Resemblance to the Kruskal's Algorithm [Kru56]: Our algorithm can be seen as a suitable augmentation of the Kruskal's algorithm, albeit executed on $\mathcal{G}_1$. Similar to the Kruskal's algorithm, our algorithm processes edges in non-increasing order of their capacities in $\mathcal{G}_1$, although it computes an MST in $\mathcal{G}_2$. In each iteration, it also picks two trees and merges them. However, in Kruskal's algorithm, adding only a single edge between the two trees is sufficient to obtain a new tree. As a

Algorithm 7 Algorithm for computing all-pairs second mincuts

procedure AllPairsSecondMincut(G)
  Forest $\mathcal{F}(0)$ consists of n trees: each tree is a single vertex from V, $\mathcal{C} \leftarrow \emptyset$;
  for each $v \in V$ and $T \in \mathcal{F}(0)$ where $v \notin V(T)$ **do**
    $e_p(v, T) \leftarrow$ edge between v and the vertex in T;
  end for
  for $i = 1$ to $n - 1$ **do**
    Merging two trees:
    Let (a, b) have the maximum mincut value among edges in $E_p(i-1)$;
    Let $a \in V(T_1), b \in V(T_2)$, where $T_1, T_2 \in \mathcal{F}(i-1)$;
    $\{T', \mathcal{C}\} \leftarrow$ MergeTree$(T_1, T_2, \mathcal{C}, E_p(i))$; ▷ Joining Trees and updating $\mathcal{C}$
    $\mathcal{F}(i) \leftarrow (\mathcal{F}(i-1) \setminus \{T_1, T_2\}) \cup T'$; ▷ Updating $\mathcal{F}$
    Assign2ndMincut$(T_1, T_2, \mathcal{C})$; ▷ Assigns a 2^{nd} mincut to $(u, v), \forall u \in V(T_1), v \in V(T_2)$
    Pruning Edges:
    for each $x \notin V(T')$ **do** ▷ Removing $e_p(x, T_1)$ and $e_p(x, T_2)$ from E_p
      Let $(x, u) \leftarrow e_p(x, T_1)$ and $(x, v) \leftarrow e_p(x, T_2)$;
      $e_p(x, T') \leftarrow \{(x, u), (x, v)\} \setminus$ EliminateTriangle$(u, v, x, \mathcal{C})$;
    end for
  end for
end procedure

result, the two trees are always subtrees of the newly formed tree, which is not true for our algorithm. However, even though we add multiple new edges between T_1, T_2 to obtain T', the total number of newly added edges remains bounded by $\mathcal{O}(n \log n)$ during the entire algorithm.

References

[AFLR19] Giorgio Ausiello, Paolo Giulio Franciosa, Isabella Lari, and Andrea Ribichini. Max flow vitality in general and st-planar graphs. *Networks*, 74(1):70–78, 2019. doi:10.1002/net.21878.

[AKL+] Arpit Agarwal, Sanjeev Khanna, Huan Li, Prathamesh Patil, Chen Wang, Nathan White, and Peilin Zhong. *Parallel Approximate Maximum Flows in Near-Linear Work and Polylogarithmic Depth*, pages 3997–4061. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611977912.140, arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611977912.140, doi:10.1137/1.9781611977912.140.

[ALPS23] Amir Abboud, Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. All-pairs max-flow is no harder than single-pair max-flow: Gomory-hu trees in almost-linear time. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2204–2212. IEEE, 2023.

[BB24] Surender Baswana and Koustav Bhanja. Vital edges for (s, t)-mincut: Efficient algorithms, compact structures, & optimal sensitivity oracles. In *51st International Colloquium on Automata, Languages, and Programming (ICALP 2024)*, pages 17–1. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.ICALP.2024.17.

[BBP23] Surender Baswana, Koustav Bhanja, and Abhyuday Pandey. Minimum+ 1 (s, t)-cuts and dual-edge sensitivity oracle. *ACM Transactions on Algorithms*, 19(4):1–41, 2023.

[BBR25] Surender Baswana, Koustav Bhanja, and Anupam Roy. Faster Algorithm for Second (s,t)-Mincut and Breaking Quadratic Barrier for Dual Edge Sensitivity for (s,t)-Mincut. In Anne

Benoit, Haim Kaplan, Sebastian Wild, and Grzegorz Herman, editors, *33rd Annual European Symposium on Algorithms (ESA 2025)*, volume 351 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 68:1–68:19, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2025.68, doi:10.4230/LIPIcs.ESA.2025.68.

[BCR16] Surender Baswana, Keerti Choudhary, and Liam Roditty. Fault tolerant subgraph for single source reachability: generic and optimal. In Daniel Wichs and Yishay Mansour, editors, *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 509–518. ACM, 2016. doi:10.1145/2897518.2897648.

[Ben95a] András A. Benczúr. Counterexamples for directed and node capacitated cut-trees. *SIAM J. Comput.*, 24(3):505–510, 1995. doi:10.1137/S0097539792236730.

[Ben95b] András A. Benczúr. A representation of cuts within 6/5 times the edge connectivity with applications. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*, pages 92–102. IEEE Computer Society, 1995. doi:10.1109/SFCS.1995.492466.

[Bha25] Koustav Bhanja. Minimum+1 steiner cut and dual edge sensitivity oracle: Bridging gap between global and (s, t)-cut. In Keren Censor-Hillel, Fabrizio Grandoni, Joël Ouaknine, and Gabriele Puppis, editors, *52nd International Colloquium on Automata, Languages, and Programming, ICALP 2025, July 8-11, 2025, Aarhus, Denmark*, volume 334 of *LIPIcs*, pages 27:1–27:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. URL: https://doi.org/10.4230/LIPIcs.ICALP.2025.27, doi:10.4230/LIPICS.ICALP.2025.27.

[Bor26] Otakar Borůvka. O jistém problému minimálním. *Práce Moravské Přírodovědecké Společnosti*, 3:37–58, 1926.

[BR26] Surender Baswana and Anupam Roy. All-Pairs kth Mincuts: Combinatorial and Structural Results. In Philip Bille, Seth Pettie, and Sabine Storandt, editors, *34th Annual European Symposium on Algorithms (ESA 2026)*, volume 388 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 119:1–119:23, Dagstuhl, Germany, 2026. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ESA.2026.119, doi:10.4230/LIPIcs.ESA.2026.119.

[CH91] Chung-Kuan Cheng and T. C. Hu. Ancestor tree for arbitrary multi-terminal cut functions. *Ann. Oper. Res.*, 33(3):199–213, 1991. doi:10.1007/BF02115755.

[CH92] Chung-Kuan Cheng and TC Hu. Maximum concurrent flows and minimum cuts. *Algorithmica*, 8(1):233–249, 1992.

[CKT96] Richard Cole, Philip N. Klein, and Robert E. Tarjan. Finding minimum spanning forests in logarithmic time and linear work. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing (STOC)*, pages 243–251. ACM, 1996. doi:10.1145/237502.237563.

[DN95] Yefim Dinitz and Zeev Nutov. A 2-level cactus model for the system of minimum and minimum+ 1 edge-cuts in a graph and its incremental maintenance. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 509–518, 1995.

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.

[GH61] Ralph E Gomory and Tien Chung Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961.

[GV95] Naveen Garg and Vijay V. Vazirani. A polyhedron with all$s$ - t cuts as vertices, and adjacency of cuts. *Math. Program.*, 70:17–25, 1995. doi:10.1007/BF01585926.

[Has88] Refael Hassin. Solution bases of multiterminal cut problems. *Mathematics of operations research*, 13(4):535–542, 1988.

[Has90] Rafael Hassin. An algorithm for computing maximum solution bases. *Operations research letters*, 9(5):315–318, 1990.

[Kar95] David R Karger. A randomized fully polynomial time approximation scheme for the all terminal network reliability problem. In *Proceedings of the twenty-seventh annual ACM Symposium on Theory of Computing*, pages 11–17, 1995.

[Kar00] David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, jan 2000. doi:10.1145/331605.331608.

[Kru56] Joseph B Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KT18] Robert Krauthgamer and Ohad Trabelsi. Conditional lower bounds for all-pairs max-flow. *ACM Transactions on Algorithms (TALG)*, 14(4):1–15, 2018.

[KYN07] Yoko Kamidoi, Noriyoshi Yoshida, and Hiroshi Nagamochi. A deterministic algorithm for finding all minimum k-way cuts. *SIAM Journal on Computing*, 36(5):1329–1341, 2007.

[LNPS23] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, and Thatchaphol Saranurak. *Near-Linear Time Approximations for Cut Problems via Fair Cuts*, pages 240–275. 01 2023. doi:10.1137/1.9781611977554.ch10.

[Lov79] L. Lovász. *Combinatorial problems and exercises*. North-Holland Publishing Co., Amsterdam-New York, 1979.

[LR89] Tom Leighton and Satish Rao. An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms. Technical report, 1989.

[LS14] Yin Tat Lee and Aaron Sidford. Path finding methods for linear programming: Solving linear programs in o (vrank) iterations and faster algorithms for maximum flow. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 424–433. IEEE, 2014.

[NNI97] Hiroshi Nagamochi, Kazuhiro Nishimura, and Toshihide Ibaraki. Computing all small cuts in an undirected network. *SIAM Journal on Discrete Mathematics*, 10(3):469–481, 1997. doi:10.1137/S0895480194271323.

[NW61] C St JA Nash-Williams. Edge-disjoint spanning trees of finite graphs. *Journal of the London Mathematical Society*, 1(1):445–450, 1961.

[OS93] James B Orlin and Clifford Stein. Parallel algorithms for the assignment and minimum-cost flow problems. *Operations research letters*, 14(4):181–186, 1993.

[Pri57] Robert Clay Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[PS] Debmalya Panigrahi and Thatchapol Saranurakh. personal communication.

[Sch79] Claus-Peter Schnorr. Bottlenecks and edge connectivity in unsymmetrical networks. *SIAM Journal on Computing*, 8(2):265–274, 1979.

[ST81] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 114–122, 1981.

[VDBCP+23] Jan Van Den Brand, Li Chen, Richard Peng, Rasmus Kyng, Yang P Liu, Maximilian Probst Gutenberg, Sushant Sachdeva, and Aaron Sidford. A deterministic almost-linear time algorithm for minimum-cost flow. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 503–514. IEEE, 2023.

[VDBGJDV25] Jan Van Den Brand, Hossein Gholizadeh, Yonggang Jiang, and Tjin De Vos. Parallel minimum cost flow in near-linear work and square root depth for dense instances. In *Proceedings of the 37th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 1–16, 2025.

[VY92] Vijay V Vazirani and Mihalis Yannakakis. Suboptimal cuts: Their enumeration, weight and number. In *International Colloquium on Automata, Languages, and Programming*, pages 366–377. Springer, 1992.

A Impact on Directed Graphs

Here, we describe the impact of Theorem 3 and Theorem 4 in the classical setting ($F = c$). In directed graphs, the problem of computing mincut for a single pair of vertices has been resolved due to the $m^{1+o(1)}$ time algorithm by [VDBCP+23]. However, there is a trivial lower bound of $\Omega(mn^2)$ [KT18] for all-pairs mincuts in directed graphs. As a result, the problem of computing *symmetric all-pairs mincuts* in directed graphs has acquired interest from many well-known researchers [Sch79, Ben95b]. Here, one has to find the minimum between the values of (u, v)-mincut and (v, u)-mincut, that is $\min\{\lambda_1(u, v), \lambda_1(v, u)\}$, for each pair (u, v). Given the settlement of all-pairs mincuts in undirected graphs, this is of more interest, as mentioned in [ALPS23]. However, the best known bound for computing symmetric all-pairs mincuts in directed graphs is obtained by the algorithm of Cheng and Hu [CH91]. Although our results do not break this bound, Theorem 3 provides the first parallel algorithm for solving the problem with $\tilde{\mathcal{O}}(\sqrt{n})$ depth while achieving work done almost equal to that of the algorithm of [CH91]. Moreover, Theorem 4 shows that the number of second mincut computations for symmetric all-pairs second mincuts (compute $\min\{\lambda_2(u, v), \lambda_2(u, v)\}$) differs from the number of mincut computations for symmetric all-pairs mincuts by only a $\mathcal{O}(\log n)$ factor, in directed graphs.

B When 2^{nd} Mincut Does not Exist for a pair

We first show a graph where there does not exist a second mincut for a pair (u, v) of vertices. An undirected unweighted graph with $n - 2$ vertex disjoint paths from u to v, each of length 2, observe that each of the 2^{n-2} (u, v)-cuts is a (u, v)-mincut. We now show that for any graph G, we can obtain a graph G' in which the second mincuts and mincuts of G are preserved and additionally a second mincut exists for all pairs of vertices.

For each vertex $v \in V$, add a new vertex v' and add an edge (v, v') with infinite capacity. Let V' denote the set of n newly added vertices and G' be the corresponding graph. Observe that for any pair $u, v \in V$, if $\lambda_2(u, v)$ exists, then $\lambda_2(u', v') = \lambda_2(u, v)$, else $\lambda_2(u', v') = \infty$. Therefore, for each pair (u', v'), a second (u', v')-mincut always exists. Moreover, computing all-pairs second mincut for pairs of vertices $V' \times V'$ in G' is equivalent to the problem of computing all-pairs second mincut for pairs $V \times V$ in G. Moreover, we can also determine the pairs in G whose second mincut does not exist.

Now, we state how the capacities of edges (v, v') can be realized. Observe that we can set these to be any arbitrarily large number greater the value of maximum value cut in G. Recall that although F is a real valued function, the maximum value taken by F is bounded by some finite value, say $a \geq 1$. Hence, we can set the capacity of edge (v, v') to be $a + 1$, for all $v' \in V'$. In the classical setting ($F = c$), we can simply set them to be the sum of capacity of all edges in G.

C Algorithm of [CH91] as a special case of our framework

We first provide a brief description of our faster algorithm for its ease of comparison with the algorithm of Cheng and Hu [CH91].

Initially, all the vertices belong to singleton trees. The algorithm arbitrarily chooses a tree and then grows it into an MST for $\mathcal{G}_1$ over the $n-1$ iterations, by adding one vertex to the tree in each iteration. At termination, all the vertices belong to an MST for $\mathcal{G}_1$, the non-singleton tree. At the beginning of ith iteration, there are i vertices in the non-singleton tree, say T, and $n-i$ singleton trees and following invariants are maintained. Each vertex, say u, in any singleton tree has exactly one proxy edge $e_p(u,T)$ to the tree T. Hence, a vertex in any singleton tree can be viewed as being mapped to a single vertex in T. For any pair of vertices in T, the algorithm stores a mincut.

During the $(i+1)$th iteration, the algorithm arbitrarily picks a singleton tree with a vertex, say u. Suppose $(u,v) = e_p(u,T)$, that is, u is mapped to the vertex v in the tree T. The algorithm computes a (u,v)-mincut and joins vertex u to T using the edge $e_p(u,T)$. Based on the cut C, Algorithm 1 is used to update the mapping of each vertex in the singleton trees. This ensures that the invariants of the algorithm are maintained. Let us now bound the number of mincut computations. In each of the $n-1$ iterations, we compute mincut for exactly one proxy edge. Therefore, the total number of mincut computations in the algorithm is $n-1$.

Overview of the algorithm of Cheng and Hu [CH91]: The algorithm of Cheng and Hu computes the ancestor tree incrementally and performs $n-1$ mincut computations. The algorithm marks each vertex as either a *seed* or an *unseed* vertex. Initially, each vertex is an *unseed* vertex and the ancestor tree is a single node containing all vertices. Before the first iteration, any arbitrary vertex is picked and marked as a *seed vertex*. At termination, all the vertices are marked as seed vertices. At the beginning of ith iteration, where $1 \leq i \leq n-1$, there are i seed vertices and $n-i$ unseed vertices. Moreover, the ancestor tree has $i-1$ internal nodes, each storing a cut; and i leaf nodes defining a partition of V, such that the following invariants are maintained. Each leaf node contains exactly one seed vertex and hence, each unseed vertex is mapped to a single seed vertex. Each internal node stores a pair of seed vertices (a,b) and an (a,b)-mincut such that the following hold. For any pair (u,v) of seed vertices, the lowest common ancestor (LCA) of (u,v) in the ancestor tree stores a (u,v)-mincut.

During the $(i+1)$th iteration, the algorithm chooses a leaf node with a seed vertex, say v and at least one unseed vertex. It arbitrarily picks an unseed vertex u from the leaf node, computes a (u,v)-mincut C, and marks u as a seed vertex. It creates an internal node storing C and a leaf node containing u. Based on the cut C, it updates the mapping of unseed vertices and also the ancestor tree such that the invariants are maintained.

Similarities with the algorithm by Cheng and Hu [CH91]: In the algorithm by Cheng and Hu [CH91], suppose each mincut computation between a unseed vertex u and a seed vertex v is represented by the edge (u,v) with capacity $\lambda_1(u,v)$. In other words, the edges of this tree are the same as the pair of seed vertices stored at each internal node in the ancestor tree. It is easy to see that at any iteration i, these set of edges form a spanning tree on the set of all i seed vertices[3]. Now, refer to our faster algorithm and simply consider each vertex of the singleton trees to be unseed vertices and the vertices in the non-singleton tree to be seed vertices. Observe that at any iteration i, the non-singleton tree is nothing but the spanning tree described above. The mapping of unseed vertices to the seed vertices is captured by the proxy edges in our algorithm. Hence, the only

[3]We note that this tree was defined in [CH91] as well but only used for analysis

difference between the two algorithms is that our modified algorithm does not use the ancestor tree, while still achieving the same running time.

As the reader may verify, given one's familiarity with various algorithms for computing MST, our algorithm is arguably simpler to describe and easier to analyze than the algorithm of [CH91]. Moreover, the mincut cover computed by our algorithm can be used to build the ancestor tree of [CH91] in $\mathcal{O}(n^2)$ time. Hence, the ancestor tree of Cheng and Hu [CH91] turns out to be merely an auxiliary tree built on the MST that our algorithm computes.

The procedure AssignMincut in our framework essentially serves the same purpose as the ancestor tree of [CH91]. However, it has the following advantages. The construction of ancestor tree is sequential. Moreover, the ancestor tree does not exist for all-pairs second mincuts [BR26]. Interestingly, as shown in Appendix D, the procedure AssignMincut can be easily extended for both our parallel algorithm and our algorithm for all-pairs second mincuts.

D Recursive Approach to PruneEdges and AssignMincut

We first present a divide and conquer approach to compute a proxy edge $e_p(x, T_B)$ from each vertex $x \in A = V(T_A)$ to the tree T_B on set of vertices $B = V(T_B)$. Since the algorithm focuses on set of vertices B rather than the tree T_B, let us denote the proxy edge $e_p(x, T_B)$ by $e_p(x, B)$ instead.

Divide and Conquer Approach to Pruning Edges: For each vertex $x \in A$, proceed as follows. Divide the set of edges $\{(x, y)|y \in B\}$ into $\lceil |B|/2 \rceil$ sets of edges, each containing a pair of at most two edges. For each set with exactly two edges, say (x, u) and (x, v), use EliminateTriangle (Algorithm 1) to prune one of the two edges (x, u) and (x, v). Hence, only $\lceil |B|/2 \rceil$ edges are left. On these remaining edges, we repeat the same procedure till there is exactly one edge left, which we assign as $e_p(x, B)$ (or $e_p(x, T_B)$). The pseudocode is provided in Algorithm 8.

Algorithm 8 Divide and Conquer Approach to Pruning Edges

```
1:  procedure DACPruneEdges(T_A, T_B, C_B)
2:      for each x ∈ V(T_A), E_p(x) ← {(x, y)|y ∈ B};
3:      for each x ∈ V(T_A) do                                   ▷ Computing e_p(x, T_B)
4:          while |E_p(x)| ≠ 1 do
5:              Partition set E_p(x) into ⌈|E_p(x)|/2⌉ sets;
6:              for each set with two pairs, say (x, u) and (x, v) do   ▷ halves the size of set E_p(x)
7:                  E_p(x) ← E_p(x) \ EliminateTriangle(u, v, x, C^1_{u,v});   ▷ C^1_{u,v} ∈ C_B
8:              end for
9:          end while
10:     end for
11:     E_p ← ∪_{x∈A} E_p(x), return E_p;          ▷ Returns the proxy edges between T_A, T_B
12: end procedure
```

Observe that the following lemma immediately follows from Algorithm 8.

Lemma 12. *Let B be any set of vertices and $x \notin B$ be any vertex. There exists a recursive algorithm with $\mathcal{O}(\log |B|)$ recursive depth that given the mincut cover of $B \times B$ pairs, can compute a proxy edge $e_p(x, B)$ from vertex x to set B in $\mathcal{O}(|B|)$ time.*

Array $\mathcal{M}$: Throughout the algorithm (Algorithm 4), we maintain a matrix $\mathcal{M}$ for all pairs of vertices. Initially the array is empty. At the end of the algorithm, for each pair (u, v), $\mathcal{M}(u, v)$ stores $\lambda_1(u, v)$ a pointer to a (u, v)-mincut in the mincut cover $\mathcal{C}$. At any intermediate subproblem, the algorithm maintains the following invariant.

4. Let X be the subset of vertices associated with the subproblem. Then for each pair (u, v) of vertices with $u, v \in X$, $\mathcal{M}(u, v)$ stores a pointer to an (u, v)-mincut in $\mathcal{C}$.

Suppose Algorithm 4 has been executed till Line 14, that is MergeTree$(T_A, T_B, \mathcal{C}_A \cup \mathcal{C}_B, E_p)$. Hence, the following facts are true for each vertex $x \in A$. The proxy edge $e_p(x, B)$ has been computed. It implies from the above invariant that for each $u, v \in B$, $\mathcal{M}(u, v)$ now stores a pointer to a (u, v)-mincut in $\mathcal{C}_B$. Moreover, for each proxy edge $e_p(x, B)$, an $e_p(x, B)$-mincut has been computed and a pointer to it is stored at $\mathcal{M}(e_p(x, B))$. We now show how to efficiently assign a pointer to $\mathcal{M}(x, y)$ for rest of the pruned edges, that is $\{(x, y)|y \in B\} \setminus e_p(x, B)$, in $\mathcal{O}(|B|)$ time. This step is stated as the subroutine DACAssignMincut in Line 15 of Algorithm 4. Its task is to ensure that above invariant is maintained for $A \cup B$.

We now provide a description of the subroutine DACAssignMincut. Observe that there is a implicit recursion tree associated with DACPruneEdges for each vertex $x \in A$. Let us denote it by $\mathcal{T}(x)$. Notice that $\mathcal{T}(x)$ has the following properties.

Recursion Tree $\mathcal{T}(x)$: The tree is a binary tree with $\mathcal{O}(\log |B|)$ recursion depth. It has $|B|$ leaf nodes storing the edges $\{(x, v)|v \in B\}$ and its root node stores the proxy edge $e_p(x, B)$. Any internal node μ stores an edge from the set $\{(x, v)|v \in B\}$ with the following property. If the two child nodes of μ store edges (x, u) and (x, v), then μ stores the edge $\{(x, u), (x, v)\} \setminus$ EliminateTriangle$(x, u, v, C^1_{u,v})$.

DACAssignMincut takes a top down approach on this tree starting from the root node, as described below.

DACAssignMincut: Consider the root node of $\mathcal{T}(x)$ containing $e_p(x, B)$. Let its two child nodes, contain (x, u) and (x, v), and suppose $e_p(x, B) = (x, u)$. Observe that for (x, u), a pointer is already stored. The same is true for (u, v), since $u, v \in B$. Hence, let us now assign a pointer to $\mathcal{M}(x, v)$. It follows from Lemma 5 that the (x, v)-mincut is either $C^1_{u,v}$ or every (x, u)-mincut is a (x, v)-mincut. Hence, if $\lambda_1(x, u) \geq \lambda_1(u, v)$, $\mathcal{M}(x, v) = \mathcal{M}(u, v)$. Else $(\lambda_1(x, u) < \lambda_1(u, v))$, $\mathcal{M}(x, v) = \mathcal{M}(x, u)$. Observe that this requires $\mathcal{O}(1)$ time. Now, we have found $\mathcal{M}(x, u)$ and $\mathcal{M}(x, v)$. So, we proceed recursively on the nodes storing (x, u) and (x, v). Hence, $\{\mathcal{M}(x, y)|y \in B)\}$ gets computed.

As stated above, the recursion tree $\mathcal{T}(x)$ has $|B|$ leaf nodes and hence $\mathcal{O}(|B|)$ total nodes. Since, DACAssignMincut requires $\mathcal{O}(1)$ time at each node, the total time taken for each vertex $x \in A$ is $\mathcal{O}(|B|)$. Moreover, since $\mathcal{T}(x)$ has $\mathcal{O}(\log |B|)$ depth, DACAssignMincut also has recursion depth $\mathcal{O}(\log |B|)$. Therefore, the following lemma is immediate.

Lemma 13. *Let B be any set of vertices and T_B be an MST for $\mathcal{G}_1[B]$. There exists an algorithm that given a proxy edge $e_p(x, T_B)$, recursion tree $\mathcal{T}(x)$ and the pointers $\{\mathcal{M}(u, v)|u, v \in B\}$, can compute $\{\mathcal{M}(x, y)|y \in B\}$ in $\mathcal{O}(|B|)$ time.*

It follows from Lemma 12 and Lemma 13, both the algorithms DACPruneEdges (Algorithm 8) and DACAssignMincut require $\mathcal{O}(|A|.|B|)$ time. Additionally, for the ease of implementation in the parallel model, we state some properties which are common to both the algorithms.

Lemma 14. *For each vertex $x \in A$, DACPruneEdges and DACAssignMincut share the following properties.*

1. *At each internal node of $\mathcal{T}(x)$, the algorithms require $\mathcal{O}(1)$ time.*
2. *The computations at each of the internal nodes at the same recursion level of $\mathcal{T}(x)$ can be performed independently.*
3. *The computations performed at recursion level i in $\mathcal{T}(x)$ are dependent only on the computations performed at recursion level $i-1$.*
4. *The algorithms require $\mathcal{O}(|B|)$ time and have $\mathcal{O}(\log|B|)$ recursion depth.*
5. *The computations at each tree $\mathcal{T}(x)$ for each vertex $x \in A$ can be performed independently.*

D.1 AssignMincut for Algorithm 3

As described above, DACAssignMincut is but a simple algorithm executed on the recursion tree $\mathcal{T}(x)$ obtained from the pruning of edges. Moreover, its running time depends only on the number of nodes in the tree. Since Algorithm 3 also prunes edges, we can also associate a similar tree with each vertex x with a single difference. Unlike Algorithm 8, pruning of edges is done in a sequential manner in Algorithm 3. In particular, exactly one edge incident on x is pruned in each iteration. The pruning comes to a pause when the tree, say T_1, containing x is merged to a tree, say T_2. As a result, the depth of the tree would be $\mathcal{O}(|V(T_2)|)$ and not necessarily logarithmic. Despite this difference, the number of nodes in the recursion tree is the same as $\mathcal{T}(x)$, that is $\mathcal{O}(|V(T_2)|)$. Hence, after merging T_1 with T_2, assigning mincut to the $|V(T_2)|-1$ edges would still require $\mathcal{O}(|V(T_2)|)$ time, similar to DACAssignMincut. Since, there are $n-1$ edges incident on x, AssignMincut would require $\mathcal{O}(n)$ time for each vertex. Therefore, during the entire execution of Algorithm 3, the total time taken by AssignMincut is $\mathcal{O}(n^2)$.

D.2 Designing Assign2ndMincut for Algorithm 7

Suppose Algorithm 7 has been executed till Line 11. Hence, the two trees T_1, T_2 have been merged. Suppose $V(T_1) \leq V(T_2)$. So, a second mincut as well as mincut for the proxy edges $\{e_p(x,T_2)|x \in V(T_1)\}$ and the pairs (u,v) for each $u,v \in V(T_2)$ is provided to us. The task of Assign2ndMincut is to assign a second mincut to the edges $\{(x,y)|y \in V(T_2)\}$, for each $x \in V(T_1)$.

It follows from the above discussions that to design Assign2ndMincut, we can again associate a recursion tree with each vertex x. Let us denote it by $\mathcal{T}_2(x)$.

Recursion Tree $\mathcal{T}_2(x)$: The tree is a binary tree with $\mathcal{O}(\log|V(T_2)|)$ recursion depth. It has $|V(T_2)|$ leaf nodes storing the edges $\{(x,y)|y \in V(T_2)\}$ and its root node stores the proxy edge $e_p(x,T_2)$. Any internal node μ stores an edge from the set $\{(x,v)|y \in V(T_2)\}$ with the following property. If the two child nodes of μ store edges (x,u) and (x,v), then μ stores the edge $\{(x,u),(x,v)\} \setminus$ EliminateTriangle$(x,u,v,C^1_{u,v},C^2_{u,v})$ (Lemma 5).

Consider any pair (u,v) of vertices. Recall that array $\mathcal{M}(u,v)$ stores a pointer to an (u,v)-mincut in $\mathcal{C}$. Let us now maintain an additional array $\mathcal{M}_2$ such that $\mathcal{M}_2(u,v)$ stores a pointer to a cut in $\mathcal{C}$, that is a second (u,v)-mincut. The only remaining task is to show that at each internal node of $\mathcal{T}_2(x)$, we can perform Assign2ndMincut in $\mathcal{O}(1)$ time. We first state a stronger version of Lemma 9, which is immediate from Lemma 7, Lemma 8 and Algorithm 5.

Lemma 15. *Given any pair of vertices (u,v), $C^1_{u,v}$, $C^2_{u,v}$, and any vertex $x \notin \{u,v\}$ such that $\lambda_1(u,v) \geq \lambda_1(x,u) = \lambda_1(x,v)$, Algorithm 5 reports in $\mathcal{O}(1)$ time a pair $(x,a), a \in \{u,v\}$, satisfying the following properties.*

1. *(x,a) is the least capacity edge in the triangle $\langle u,v,x\rangle$ in $\mathcal{G}_2$.*

2. *If* $\langle u, v, x\rangle$ *is of Type* 1*, either* $C^1_{u,v}$ *is second* (x, a)*-mincut or every second* (x, b)*-mincut is a second* (x, a)*-mincut, where* $b \in \{u, v\}\backslash\{a\}$.
3. *If* $\langle u, v, x\rangle$ *is of Type* 2*, either* $C^2_{u,v}$ *is second* (x, a)*-mincut or every second* (x, b)*-mincut is a second* (x, a)*-mincut, where* $b \in \{u, v\}\backslash\{a\}$.

Assign2ndMincut: Consider the root node of $\mathcal{T}(x)$ containing $e_p(x, T_2)$. Let its two child nodes, contain (x, u) and (x, v), and suppose $e_p(x, T_2) = (x, u)$. Observe that for (x, u) and (u, v), pointers are stored in $\mathcal{M}_2$. Hence, let us now assign a pointer to $\mathcal{M}_2(x, v)$. For this, we first check whether $\langle u, v, x\rangle$ is of Type1 or Type2 and then use Lemma 15. Suppose $\lambda_1(u, v) > \lambda_1(x, u) = \lambda_1(x, v)$ (Type1). Based on Lemma 15(2), if $\lambda_2(x, u) \geq \lambda_1(u, v)$, $\mathcal{M}_2(x, v) = \mathcal{M}(u, v)$. Else ($\lambda_2(x, u) < \lambda_1(u, v)$), $\mathcal{M}_2(x, v) = \mathcal{M}_2(x, u)$. Now, suppose $\lambda_1(u, v) = \lambda_1(x, u) = \lambda_1(x, v)$ (Type2). Based on Lemma 15(3), if $\lambda_2(x, u) \geq \lambda_2(u, v)$, $\mathcal{M}_2(x, v) = \mathcal{M}_2(u, v)$. Else, $\mathcal{M}_2(x, v) = \mathcal{M}_2(x, u)$. This procedure can be easily extended for any internal node in the tree $\mathcal{T}_2(x)$.

For every internal node, Assign2ndMincut requires $\mathcal{O}(1)$ time. Since the number of nodes in $\mathcal{T}_2(x)$ are $\mathcal{O}(|V(T_2)|)$, Assign2ndMincut requires $\mathcal{O}(|V(T_2)|)$ time for each vertex $x \in V(T_1)$. Since the number of edges incident on x is $n - 1$, Assign2ndMincut requires $\mathcal{O}(n^2)$ total time during the excution of Algorithm 7.

E Parallel Implementation of Algorithm 4

We now provide the implementation of Algorithm 4 in the CREW (*concurrent read exclusive write*) PRAM model of computation. In this model, multiple processors are allowed to read the same memory location concurrently, but only one processor can write to a given location at any moment.

The memory locations that are to be read concurrently by the processors belong to the following two structures. (1) $\mathcal{C}$ stores a set of mincuts. Each mincut $(X, \overline{X})$ is represented by the subset of vertices X, which can be represented by a Boolean array on V. (2) The matrix $\mathcal{M}$ stores the pointers from each pair of vertices to a mincut in $\mathcal{C}$.

We now provide a parallel implementation of Algorithm 4, which essentially requires a parallel implementation of its conquer step: DACPruneEdges, MergeTree and DACAssignMincut.

1. The first part of MergeTree involves computing the mincut for the $\min\{|A|, |B|\} = |B|$ proxy edges. This can be performed concurrently for each proxy edge. Hence, this requires $\mathcal{O}(D_P(u, v))$ depth and $\mathcal{O}(|B|W_P(u, v))$ work, where $D_P(u, v)$ is the depth and $W_P(u, v)$ is the work achieved by a parallel algorithm P for computing an (u, v)-mincut. The second part involves computation of an MST on $|A| + |B| + \min\{|A|, |B|\} = \mathcal{O}(|B|)$ edges. Using the parallel algorithm of [CKT96], this can be performed in $\mathcal{O}(\log |B|)$ depth and $\mathcal{O}(|B| \log |B|)$ work.
2. We now focus on the parallel implementation of the two procedures DACPruneEdges and DACAssignMincut for each vertex $x \in A$. Refer to the recursion tree $\mathcal{T}(x)$ associated with DACPruneEdges in Page 22. It follows from Lemma 14 that to implement the two procedures in parallel CREW model in $\mathcal{O}(\log |B|)$ depth, we only need to show that the computation at all nodes belonging to the same recursion level of $\mathcal{T}(x)$ can be done concurrently. Consider any internal node, say μ, of $\mathcal{T}(x)$. We allocate one processor to μ. In DACPruneEdges, this processor can execute EliminateTriangle in $\mathcal{O}(1)$ time as it only verifies whether x belongs to a cut $C^1_{u,v}$ or $\overline{C}^1_{u,v}$. This requires read access to $\mathcal{C}$ and exclusive write access to the node μ in $\mathcal{T}(x)$. The same is true for DACAssignMincut which simply requires (1) read access to the mincut values of $\{(x, u), (u, v)\}$ and the corresponding pointers $\{\mathcal{M}(x, u), \mathcal{M}(u, v)\}$, and (2) exclusive write access to $\mathcal{M}(x, v)$ (observe that once $\mathcal{M}(x, v)$ is computed, it is never

modified). Moreover, since $\mathcal{M}$ and $\mathcal{C}$ can be read concurrently, it follows from Lemma 14(5) that the two procedures can be executed in $\mathcal{O}(\log |B|)$ depth for all vertices $\{x | x \in A\}$.

We now analyze the running time of Algorithm 4 in the parallel CREW model.

Lemma 16. *The following properties hold for Algorithm 4.*

1. *The subset of vertices across all subproblems at the same recursion level are disjoint.*
2. *There are $\mathcal{O}(\log n)$ recursion levels.*

Analysis of Algorithm 4: Consider any conquer step of Algorithm 4. As shown above, DACPruneEdges and DACAssignMincut can be executed in $\mathcal{O}(\log |B|)$ depth and $\mathcal{O}(|B|)$ work for each vertex $x \in A$. Since any edge in $\mathcal{G}_1$ is processed exactly once by these procedures, the work done by them is bounded by $\mathcal{O}(n^2)$. It follows from Lemma 16(1) that the depth of these procedures is $\mathcal{O}(\log n)$ at each recursion level and as a result, by Lemma 16(2), the total depth across all levels of Algorithm 4 is bounded by $\mathcal{O}(\log^2 n)$. Consider the procedure MergeTree. At each subproblem, MergeTree requires $\mathcal{O}(D_P(u, v) + \log |B|)$ depth and $\mathcal{O}(|B| W_P(u, v) + |B| \log |B|)$ work. Hence, using Lemma 16, MergeTree achieves $\mathcal{O}(\log n D_P(u, v) + \log^2 n)$ total depth and $\mathcal{O}(n \log n W_P(u, v) + n \log^2 n)$ total work, across all levels of Algorithm 4. Therefore, Algorithm 4 achieves $\mathcal{O}(\log n D_P(u, v) + \log^2 n)$ depth and $\mathcal{O}(n \log n W_P(u, v) + n^2)$ work. This completes the proof of Theorem 3.