arXiv:1408.0557v1 [cs.DS] 4 Aug 2014

Almost-Tight Distributed Minimum Cut Algorithms*

Danupon Nanongkai†
University of Vienna, Austria

Hsin-Hao Su‡
University of Michigan, USA

Abstract

We study the problem of computing the minimum cut in a weighted distributed message-passing networks (the CONGEST model). Let λ be the minimum cut, n be the number of nodes (processors) in the network, and D be the network diameter. Our algorithm can compute λ exactly in $O((\sqrt{n}\log^* n+D)\lambda^4\log^2 n)$ time. To the best of our knowledge, this is the first paper that explicitly studies computing the exact minimum cut in the distributed setting. Previously, non-trivial sublinear time algorithms for this problem are known only for unweighted graphs when $\lambda \leq 3$ due to Pritchard and Thurimella's $O(D)$-time and $O(D+n^{1/2}\log^* n)$-time algorithms for computing 2-edge-connected and 3-edge-connected components [ACM Transactions on Algorithms 2011].

By using the edge sampling technique of Karger [STOC 1994], we can convert this algorithm into a $(1+\epsilon)$-approximation $O((\sqrt{n}\log^* n+D)\epsilon^{-5}\log^3 n)$-time algorithm for any $\epsilon > 0$. This improves over the previous $(2+\epsilon)$-approximation $O((\sqrt{n}\log^* n+D)\epsilon^{-5}\log^2 n\log\log n)$-time algorithm and $O(\epsilon^{-1})$-approximation $O(D+n^{\frac{1}{2}+\epsilon}\operatorname{poly}\log n)$-time algorithm of Ghaffari and Kuhn [DISC 2013]. Due to the lower bound of $\Omega(D+n^{1/2}/\log n)$ by Das Sarma et al. [SICOMP 2013] which holds for any approximation algorithm, this running time is *tight* up to a $\operatorname{poly}\log n$ factor.

To get the stated running time, we developed an approximation algorithm which combines the ideas of Thorup's algorithm [Combinatorica 2007] and Matula's contraction algorithm [SODA 1993]. It saves an $\epsilon^{-9}\log^7 n$ factor as compared to applying Thorup's tree packing theorem directly. Then, we combine Kutten and Peleg's tree partitioning algorithm [J. Algorithms 1998] and Karger's dynamic programming [JACM 2000] to achieve an efficient distributed algorithm that finds the minimum cut when we are given a spanning tree that crosses the minimum cut exactly once.

*The preliminary versions of this paper appeared as brief announcement papers at PODC 2014 and SPAA 2014 [16, 22].

†Faculty of Computer Science, University of Vienna, Währinger Straße 29, A-1090 Vienna, Austria. Email: danupon@gmail.com. This work was partially done while at ICERM, Brown University USA and Nanyang Technological University, Singapore.

‡2260 Hayward St., Department of EECS, University of Michigan, Ann Arbor, MI 48109. Email: hsinhao@umich.edu. This work is supported by NSF grants CCF-1217338 and CNS-1318294. This work was done while visiting MADALGO at Aarhus University, supported by Danish National Research Foundation grant DNRF84.

1 Introduction

Minimum cut is an important measure of networks. It determines, e.g., the network vulnerability and the limits to the speed at which information can be transmitted. While this problem has been well-studied in the centralized setting (e.g. [5, 10, 6, 7, 15, 14, 2, 21, 8]), very little is known in the distributed setting, especially in the relevant context where communication links are constrained by a small *bandwidth* – the so-called CONGEST model (cf. Section 2).

Consider, for example, a simple variation of this problem, called λ-*edge-connectivity*: given an *unweighted* undirected graph G and a *constant* λ, we want to determine whether G is λ-edge-connected or not. In the centralized setting, this problem can be solved in $O(m + n\lambda^2 \log n)$ time [2], thus near-linear time when λ is a constant. (Throughout, n, m, and D denotes the number of nodes, number of edges, and the network diameter, respectively.) In the distributed setting, however, non-trivial solutions exist only when $\lambda \leq 3$; this is due to algorithms of Pritchard and Thurimella [20] which can compute 2-edge-connected and 3-edge-connected components in $O(D)$ and $O(D + n^{1/2} \log^* n)$ time, respectively, with high probability[1]. This implies that the λ-edge-connectivity problem can be solved in $O(D)$ time when $\lambda = 2$ and $O(D + n^{1/2} \log^* n)$ time when $\lambda = 3$.

For the general version where input graphs could be weighted, the problem can be solved in near-linear time [8, 14, 6, 7] in the centralized setting. In the distributed setting, the first non-trivial upper bounds are due to Ghaffari and Kuhn [4], who presented $(2 + \epsilon)$-approximation $O((\sqrt{n} \log^* n + D)\epsilon^{-5} \log^2 n \log \log n)$-time and $O(\epsilon^{-1})$-approximation $O(D + n^{\frac{1}{2}+\epsilon} \operatorname{poly} \log n)$-time algorithms. These upper bounds are complemented by a lower bound of $\Omega(D + n^{1/2}/\log n)$ for any approximation algorithm which was earlier proved by Das Sarma et al. [1] for the weighted case and later extended by [4] to the unweighted case. This means that the running times of the algorithms in [4] are tight up to a polylog n factor. Yet, it is still open whether we can achieve an approximation factor less than two in the same running time, or in fact, in any sublinear (i.e. $O(D + o(n))$) time.

Results. In this paper, we present improved distributed algorithms for computing the minimum cut both exactly and approximately. Our exact deterministic algorithm for finding the minimum cut takes $O((\sqrt{n} \log^* n + D)\lambda^4 \log^2 n)$ time, where λ is the value of the minimum cut. Our approximation algorithm finds a $(1 + \epsilon)$-approximate minimum cut in $O((D + \sqrt{n} \log^* n)\epsilon^{-5} \log^3 n)$ time with high probability. (If we only want to compute the $(1 + \epsilon)$-approximate *value* of the minimum cut, then the running time can be slightly reduced to $O((\sqrt{n} \log^* n + D)\epsilon^{-5} \log^2 n \log \log n)$.) As noted earlier, prior to this paper there was no sublinear-time exact algorithm even when λ is a constant greater than three, nor sublinear-time algorithm with approximation ratio less than two. Table 1 summarizes the results.

Techniques. The starting point of our algorithm is Thorup's tree packing theorem [23, Theorem 9], which shows that if we generate $\Theta(\lambda^7 \log^3 n)$ trees $T_1, T_2, \ldots$, where tree T_i is the minimum spanning tree with respect to the loads induced by $\{T_1, \ldots, T_{i-1}\}$, then one of these trees will contain exactly one edge in the minimum cut (see Section 4 for the definition of load). Since we can use the $O(\sqrt{n} \log^* n + D)$-time algorithm of Kutten and Peleg [12] to compute the minimum spanning tree (MST), the problem of finding a minimum cut is reduced to finding the minimum cut that 1-*respects a tree*; i.e., finding which edge in a given spanning tree defines a smallest cut (see the formal definition in Section 3). Solving this problem in $O(D + \sqrt{n} \log^* n)$ time is the first key technical contribution of this paper. We do this by using a simple observation of Karger [8] which reduces the problem to computing the sum of degree and the number of edges contained in

[1] We say that an event holds *with high probability* (w.h.p.) if it holds with probability at least $1 - 1/n^c$, where c is an arbitrarily large constant.

Reference	Time	Approximation
Pritchard&Thurimella [20]	$O(D)$ for $\lambda \leq 2$	exact
Pritchard&Thurimella [20]	$O(\sqrt{n}\log^* n + D)$ for $\lambda \leq 3$	exact
This paper	$O((\sqrt{n}\log^* n + D)\lambda^4 \log^2 n)$	exact
Das Sarma et al. [1]	$\Omega(\frac{\sqrt{n}}{\log n} + D)$	any
Ghaffari&Kuhn [4]	$O((\sqrt{n}\log^* n + D)\epsilon^{-5} \log^2 n \log\log n)$	$2+\epsilon$
This paper	$O((\sqrt{n}\log^* n + D)\epsilon^{-5} \log^3 n)$	$1+\epsilon$

Table 1: Summary of Results

a subtree rooted at each node. We use this observation along with Garay, Kutten and Peleg's *tree partitioning* [12, 3] to quickly compute these quantities. This requires several (elementary) steps, which we will discuss in more detail in Section 3.

The above result together with Thorup's tree packing theorem immediately imply that we can find a minimum cut exactly in $O((D+\sqrt{n}\log^* n)\lambda^7 \log^3 n)$ time. By using Karger's random sampling result [7] to bring λ down to $O(\log n/\epsilon^2)$, we can find an $(1+\epsilon)$-approximate minimum cut in $O((D+\sqrt{n}\log^* n)\epsilon^{-14}\log^{10} n)$ time. These time bounds unfortunately depend on large factors of λ, $\log n$ and $1/\epsilon$, which make their practicality dubious. Our second key technical contribution is a new algorithm which significantly reduces these factors by combining Thorup's greedy tree packing approach with Matula's contraction algorithm [14]. In Matula's $(2+\epsilon)$-approximation algorithm for the minimum cut problem, he partitioned the graph into *components* according to the *spanning forest decomposition* by Nagamochi and Ibaraki [15]. He showed that either a component induces a $(2+\epsilon)$-approximate minimum cut, or the minimum cut does not intersect with the components. In the latter case, it is safe to contract the components. Our algorithm used a similar approach, but we partitioned the graph according to Thorup's greedy tree packing approach instead of the spanning forest decomposition. We will show that either (i) a component induces a $(1+\epsilon)$-approximate minimum cut, (ii) the minimum cut does not intersect with the components, or (iii) the minimum cut 1-respect a tree in the tree packing. This algorithm and analysis will be discussed in detail in Section 4. We note that our algorithm can also be implemented in the centralized setting in $O(m + n\epsilon^{-7}\log^3 n)$ time. It is slightly worse than the current best $O(m + n\epsilon^{-3}\log^3 n)$ by Karger [6].

2 Preliminaries

Communication Model. We use a standard message passing network model called CONGEST [19]. A network of processors is modeled by an undirected unweighted n-node graph G, where nodes model the processors and edges model $O(\log n)$-*bandwidth* links between the processors. The processors (henceforth, nodes) are assumed to have unique IDs in the range of $\{1, \ldots, \text{poly}(n)\}$ and infinite computational power. We denote the ID of node v by $\mathsf{id}(v)$. Each node has limited topological knowledge; in particular, it only knows the IDs of its neighbors and knows *no* other topological information (e.g., whether its neighbors are linked by an edge or not). Additionally, we let $w : E(G) \to \{1, 2, \ldots, \text{poly}(n)\}$ be the edge weight assignment. The weight $w(uv)$ of each edge uv is known only to u and v. As commonly done in the literature (e.g., [4, 11, 13, 12, 3, 17]), we will assume that the maximum weight is poly(n) so that each edge weight can be sent through an edge (link) in one round.

There are several measures to analyze the performance of distributed algorithms. One fundamental measure is the *running time* defined as the worst-case number of *rounds* of distributed

communication. At the beginning of each round, all nodes wake up simultaneously. Each node u then sends an arbitrary message of $B = \log n$ bits through each edge uv, and the message will arrive at node v at the end of the round. (See [19] for detail.) The running time is analyzed in terms of number of nodes and the diameter of the network, denoted by n and D respectively. Since we can compute n and 2-approximate D in $O(D)$ time, we will assume that every node knows n and the 2-approximate value of D.

Minimum Cut Problem. Given a weighted undirected graph $G = (V, E)$, a *cut* $C = (S, V \setminus S)$ where $\emptyset \subsetneq S \subsetneq V$, is a partition of vertices into two non-empty sets. The *weight* of a cut, denoted by $w(C)$, is defined to be the sum of the edge weights crossing C; i.e., $w(C) = \sum_{u \in S, v \notin S} w(uv)$. Throughout the paper, we use λ to denote the weight of the minimum cut. A $(1 + \epsilon)$-approximate minimum cut is a cut C whose weight $w(C)$ is such that $\lambda \leq w(C) \leq (1 + \epsilon)\lambda$. The (approximate) minimum cut problem is to find a cut $C = (S, V \setminus S)$ with the minimum or approximately minimum weight. In the distributed setting, this means that nodes in S should output 1 while other nodes output 0.

Graph-Theoretic Notations. For $G = (V, E)$, we define $V(G) = V$ and $E(G) = E$. When we analyze the correctness of our algorithms, we will always treat G as an *unweighted multi-graph* by replacing each edge e with $w(e)$ by $w(e)$ copies of e with weight one. We note that this assumption is used only in the analysis, and in particular we still allow only $O(\log n)$ bits to be communicated through edge e in each round of the algorithm (regardless of $w(e)$). For any cut $C = (S, V \setminus S)$, let $E(C)$ denote the set of edges crossing between S and $V \setminus S$ in the multi-graph; thus $w(C) = |E(C)|$. Given an edge set $F \subseteq E$, we use G/F to denote the graph obtained by contracting every edge in F. Given a partition $\mathcal{P}$ of nodes in G, we use $G/\mathcal{P}$ to denote the graph obtained by contracting each set in $\mathcal{P}$ into one node. Note that $E(G/\mathcal{P})$ may be viewed as the set of edges in G that cross between different sets in $\mathcal{P}$. For any $U \subseteq V$, we use $G \mid U$ to denote the subgraph of G induced by nodes in U. For convenience, we use the subscript $*_H$ to denote the quantity $*$ of H; for example, λ_H denote the value of the minimum cut of the graph H. A quantity without a subscript refer to the quantity of G, the input graph.

3 Distributed Algorithm for Finding a Cut that 1-Respects a Tree

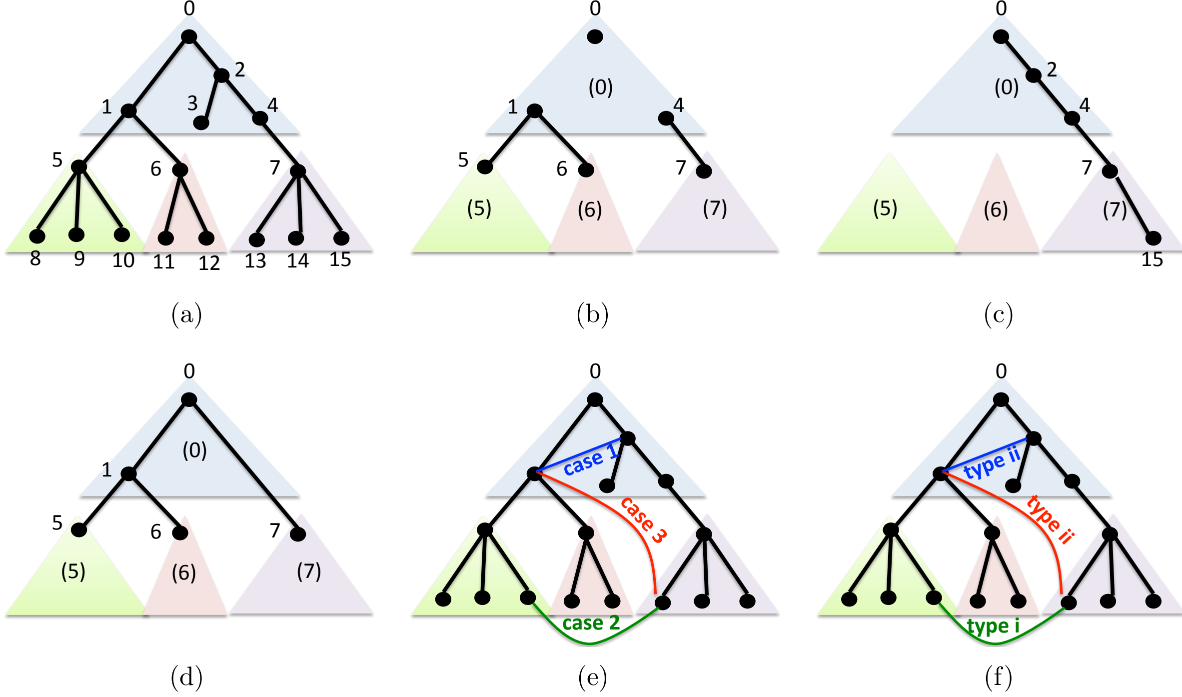
In this section, we solve the following problem: Given a spanning tree T on a network G rooted at some node r, we want to find an edge in T such that when we cut it, the cut defined by edges connecting the two connected component of T is smallest. To be precise, for any node v, define $v^\downarrow$ to be the set of nodes that are descendants of v in T, including v. Let $C_v = (v^\downarrow, V \setminus v^\downarrow)$. The problem is then to compute $c^* = \min_{v \in V(G)} w(C_v)$. The main result of this section is the following.

Theorem 3.1. *There is an $O(D + n^{1/2} \log^* n)$-time distributed algorithm that can compute c^* as well as find a node v such that $c^* = w(C_v)$.*

In fact, at the end of our algorithm every node v knows $w(C_v)$. Our algorithm is inspired by the following observation used in Karger's dynamic programming [8]. For any node v, let $\delta(v)$ be the weighted degree of v, i.e. $\delta(v) = \sum_{u \in V(G)} w(u, v)$. Let $\rho(v)$ denote the total weight of edges whose end-points' least common ancestor in T is v. Let $\delta^\downarrow(v) = \sum_{u \in v^\downarrow} \delta(u)$ and $\rho^\downarrow(v) = \sum_{u \in v^\downarrow} \rho(u)$.

Lemma 3.2 (Karger [8] (Lemma 5.9))**.** *$w(C_v) = \delta^\downarrow(v) - 2\rho^\downarrow(v)$.*

Our algorithm will make sure that every node v knows $\delta^\downarrow(v)$ and $\rho^\downarrow(v)$. By Lemma 3.2, this will be sufficient for every node v to compute $w(C_v)$. The algorithm is divided in several steps, as follows.



(a) (b) (c) (d) (e) (f)

Figure 1

Step 1: Partition T into Fragments and Compute "Fragment Tree" T_F. We use the algorithm of Kutten and Peleg [12, Section 3.2] to partition nodes in tree T into $O(\sqrt{n})$ subtrees, where each subtree has $O(\sqrt{n})$ diameter[2] (every node knows which edges incident to it are in the subtree containing it). This algorithm takes $O(n^{1/2}\log^* n + D)$ time. We call these subtrees *fragments* and denote them by $F_1, \ldots, F_k$, where $k = O(\sqrt{n})$. For any i, let $\mathsf{id}(F_i) = \min_{u \in F_i} \mathsf{id}(u)$ be the *ID of* F_i. We can assume that every node in F_i knows $\mathsf{id}(F_i)$. This can be achieved in $O(\sqrt{n})$ time (the running time is independent of D) by a communication within each fragment. Figure 1a illustrates the tree T (marked by black lines) with fragments (defined by triangular regions).

Let T_F be a rooted tree obtained by contracting nodes in the same fragment into one node. This naturally defines the child-parent relationship between fragments (e.g. the fragments labeled (5), (6), and (7) in Figure 1b are children of the fragment labeled (0)). Let the *root* of any fragment F_i, denoted by r_i, be the node in F_i that is nearest to the root r in T. We now make every node know T_F: Every "inter-fragment" edge, i.e. every edge (u, v) such that u and v are in different fragments, either node u or v broadcasts this edge and the IDs of fragments containing u and v to the whole network. This step takes $O(\sqrt{n} + D)$ time since there are $O(\sqrt{n})$ edges in T that link between different fragments and so they can be collected by pipelining. Note that this process also makes every node know the roots of all fragments since, for every inter-fragment edge (u, v), every node knows the child-parent relationship between two fragments that contain u and v.

Step 2: Compute Fragments in Subtrees of Ancestors. For any node v let $F(v)$ be the set of fragments $F_i \subseteq v^{\downarrow}$. For any node v in any fragment F_i, let $A(v)$ be the set of ancestors of v in T that are in F_i or the *parent* fragment of F_i (also let $A(v)$ contain v). (For example, Figure 1c shows $A(15)$.) We emphasize that $A(v)$ does not contain ancestors of v in the fragments that are neither

[2] To be precise, we compute a $(\sqrt{n}+1, O(\sqrt{n}))$ *spanning forest*. Also note that we in fact do not need this algorithm since we obtain T by using Kutten and Peleg's MST algorithm, which already computes the $(\sqrt{n}+1, O(\sqrt{n}))$ spanning forest as a subroutine. See [12] for details.

F_i nor the parent of F_i. The goal of this step is to make every node v knows (i) $A(v)$ and (ii) $F(u)$ for all $u \in A(v)$.

First, we make every node v know $F(v)$: for every fragment F_i we aggregate from the leaves to the root of F_i (i.e. upcast) the list of child fragments of F_i. This takes $O(\sqrt{n} + D)$ time since there are $O(\sqrt{n})$ fragments to aggregate and each fragment has diameter $O(\sqrt{n})$. In this process every node v receives a list of child fragments of F_i that are contained in $v^{\downarrow}$. It can then use T_F to compute fragments that are descendants of these child fragments, and thus compute *all* fragments contained in $v^{\downarrow}$.

Next, we make every node v in every fragment F_i know $A(v)$: every node u sends a message containing its ID down the tree T until this message reaches the leaves of the child fragments of F_i. Since each fragment has diameter $O(\sqrt{n})$ and the total number of messages sent inside each fragment is $O(\sqrt{n})$, this process takes $O(\sqrt{n})$ time (the running time is independent of D). With the following minor modifications, we can also make every node v know $F(u)$ (the fragment that u is in) for all $u \in A(v)$: Initially every node u sends a message (u, F'), for every $F' \in F(u)$, to its children. Every node u that receives a message (u', F') from its parent sends this message further to its children *if* $F' \notin F(u)$. (A message (u', F') that a node u sends to its children should be interpreted as "u' is the lowest ancestor of u such that $F' \in F(u')$".)

Step 3: Compute $\delta^{\downarrow}(v)$. For every fragment F_i, we let $\delta(F_i) = \sum_{v \in F_i} \delta(v)$ (i.e. the sum of degree of nodes in F_i). For every node v in every fragment F_i, we will compute $\delta^{\downarrow}(v)$ by separately computing (i) $\sum_{u \in F_i \cap v^{\downarrow}} \delta(u)$ and (ii) $\sum_{F_j \in F(v)} \delta(F_j)$. The first quantity can be computed in $O(\sqrt{n})$ time (regardless of D) by computing the sum within F_i (every node v sends the sum $\sum_{u \in F_i \cap v^{\downarrow}} \delta(u)$ to its parent). To compute the second quantity, it suffices to make every node know $\delta(F_i)$ for all i since every node v already knows $F(v)$. To do this, we make every root r_i know $\delta(F_i)$ in $O(\sqrt{n})$ time by computing the sum of degree of nodes within each F_i. Then, we can make every node know $\delta(F_i)$ for all i by letting r_i broadcast $\delta(F_i)$ to the whole network.

Step 4: Compute Merging Nodes and T'_F. We say that a node v is a *merging node* if there are two distinct children x and y of v such that both $x^{\downarrow}$ and $y^{\downarrow}$ contain some fragments. In other words, it is a point where two fragments "merge". For example, nodes 0 and 1 in Figure 1a are merging nodes since the subtree rooted at node 0 (respectively node 1) contains fragments (5), (6), and (7) (respectively (5) and (6)).

Let T'_F be the following tree: Nodes in T'_F are both roots of fragments (r_i's) and merging nodes. The parent of each node v in T'_F is its lowest ancestor in T that appears in T'_F (see Figure 1d for an example). Note that every merging node has at least two children in T'_F. This shows that there are $O(\sqrt{n})$ merging nodes. The goal of this step is to let every node know T'_F.

First, note that every node v can easily know whether it is a merging node or not in one round by checking, for each child u, whether $u^{\downarrow}$ contains any fragment (i.e. whether $F(u) = \emptyset$). The merging nodes then broadcast their IDs to the whole network. (This takes $O(\sqrt{n})$ time since there are $O(\sqrt{n})$ merging nodes.) Note further that every node v in T'_F knows its parent in T'_F because its parent in T'_F is one of its ancestors in $A(v)$. So, we can make every node know T'_F in $O(\sqrt{n}+D)$ rounds by letting every node in T'_F broadcast the edge between itself and its parent in T'_F to the whole network.

Step 5: Compute $\rho^{\downarrow}(v)$. We now count, for every node v, the number of edges whose least common ancestors (LCA) of their end-nodes are v. For every edge (x, y) in G, we claim that x and y can compute the LCA of (x, y) by exchanging $O(\sqrt{n})$ messages through edge (x, y). Let z denote the LCA of (x, y). Consider three cases (see Figure 1e).

Case 1: First, consider when x and y are in the same fragment, say F_i. In this case we know that z must be in F_i. Since x and y have the lists of their ancestors in F_i, they can find z by exchanging these lists. There are $O(\sqrt{n})$ nodes in such list so this takes $O(\sqrt{n})$ time. In the next two cases we assume that x and y are in different fragments, say F_i and F_j, respectively.

Case 2: z is *not* in F_i and F_j. In this case, z is a merging node such that $z^{\downarrow}$ contains F_i and F_j. Since both x and y knows T'_F and their ancestors in T'_F, they can find z by exchanging the list of their ancestors in T'_F. There are $O(\sqrt{n})$ nodes in such list so this takes $O(\sqrt{n})$ time.

Case 3: z is in F_i (the case where z is in F_j can be handled in a similar way). In this case $z^{\downarrow}$ contains F_j. Since x knows $F(x')$ for all its ancestors x' in F_i, it can compute its lowest ancestor x'' such that $F(x'')$ contains F_j. Such ancestor is the LCA of (x, y).

Now we compute $\rho^{\downarrow}(v)$ for every node v by splitting edges (x, y) whose LCA is v into two types (see Figure 1f): (i) those that x and y are in different fragments from v, and (ii) the rest. For (i), note that v must be a merging node. In this case one of x and y creates a message $\langle v \rangle$. We then count the number of messages of the form $\langle v \rangle$ for every merging node v by computing the sum along the breadth-first search tree of G. This takes $O(\sqrt{n} + D)$ time since there are $O(\sqrt{n})$ merging nodes. For (ii), the node among x and y that is in the same fragment as v creates and keeps a message $\langle v \rangle$. Now every node v in every fragment F_i counts the number of messages of the form $\langle v \rangle$ in $v^{\downarrow} \cap F_i$ by computing the sum through the tree F_i. Note that, to do this, every node u has to send the number of messages of the form $\langle v \rangle$ to its parent, for all v that is an ancestor of u in the same fragment. There are $O(\sqrt{n})$ such ancestors, so we can compute the number of messages of the form $\langle v \rangle$ for every node v *concurrently* in $O(\sqrt{n})$ time by pipelining.

4 Minimum Cut Algorithms

This section is organized as follows. In Section 4.1, we review properties of the greedy tree packing as analyzed by Thorup [23]. We use these properties to develop a $(1+\epsilon)$-approximation algorithm in Section 4.2. We show how to efficiently implement this algorithm in the distributed setting in Section 4.3 and in the sequential setting in Section 4.4.

4.1 A Review of Thorup's Work on Tree Packings

In this section, we review the duality connection between the tree packing and the partition of a graph as well as their properties from Thorup's work [23].

A *tree packing* $\mathcal{T}$ is a multiset of spanning trees. The *load* of an edge e with respect to $\mathcal{T}$, denoted by $\mathcal{L}^{\mathcal{T}}(e)$, is the number of trees in $\mathcal{T}$ containing e. Define the *relative load* to be $\ell^{T}(e) = \mathcal{L}^{\mathcal{T}}(e)/|\mathcal{T}|$. A tree packing $\mathcal{T} = \{T_1, \ldots, T_k\}$ is *greedy* if each T_i is a minimum spanning tree with respect to the loads induced by $\{T_1, \ldots, T_{i-1}\}$.

Given a tree packing $\mathcal{T}$, define its *packing value* $\texttt{pack_val}(\mathcal{T}) = 1/\max_{e \in E} \ell^{\mathcal{T}}(e)$. The packing value can be viewed as the total weight of a fractional tree packing, where each tree has weight $1/\max_{e \in E} \mathcal{L}^{\mathcal{T}}(e)$. Thus, the sum of the weight over the trees is $|T|/\max_{e \in E} \mathcal{L}^{\mathcal{T}}(e)$, which is $\texttt{pack_val}(\mathcal{T})$. Given a partition $\mathcal{P}$, define its *partition value* $\texttt{part_val}(\mathcal{P}) = \frac{|E(G/\mathcal{P})|}{|\mathcal{P}|-1}$. For any

tree packing $\mathcal{T}$ and partition $\mathcal{P}$, we have the weak duality:

$$\begin{aligned}
\texttt{pack_val}(\mathcal{T}) &= \frac{1}{\max_{e\in E} \ell^{\mathcal{T}}(e)} \\
&\leq \frac{1}{\max_{e\in E(G/\mathcal{P})} \ell^{\mathcal{T}}(e)} \\
&\leq \frac{|E(G/\mathcal{P})|}{\sum_{e\in E(G/\mathcal{P})} \ell^{\mathcal{T}}(e)} && \text{(since max} \geq \text{avg)} \\
&\leq \frac{|E(G/\mathcal{P})|}{|\mathcal{P}|-1} && \text{(since each } T\in\mathcal{T} \text{ contains at least } |\mathcal{P}|-1 \text{ edges crossing } \mathcal{P}) \\
&= \texttt{part_val}(\mathcal{P})
\end{aligned}$$

The Nash-Williams-Tutte Theorem [18, 25] states that a graph G contains $\min_{\mathcal{P}}\lfloor\frac{|E(G/\mathcal{P})|}{|\mathcal{P}|-1}\rfloor$ edge-disjoint spanning trees. Construct the graph G' by duplicating $|\mathcal{P}|-1$ edges for every edge in G. It follows from the Nash-Williams-Tutte Theorem that G' has exactly $|E(G/\mathcal{P})|$ edge-disjoint spanning trees. By assigning each spanning tree a weight of $1/(|\mathcal{P}|-1)$, we get a tree packing in G whose packing value equals to $\frac{|E(G/\mathcal{P})|}{|\mathcal{P}|-1}$. Therefore,

$$\max_{\mathcal{T}} \texttt{pack_val}(\mathcal{T}) = \min_{\mathcal{P}} \texttt{part_val}(\mathcal{P}).$$

We will denote this value by Φ. Let $\mathcal{T}^*$ and $\mathcal{P}^*$ denote a tree packing and a partition with $\texttt{pack_val}(\mathcal{T}^*) = \Phi$ and $\texttt{part_val}(\mathcal{P}^*) = \Phi$. Karger [8] showed the following relationship between Φ and λ (recall that λ is the value of the minimum cut).

Lemma 4.1. $\lambda/2 < \Phi \leq \lambda$

Proof. $\Phi \leq \lambda$ is obvious because a minimum cut is a partition with partition value exactly λ. Consider an optimal partition $\mathcal{P}^*$. Let $C_{\min}$ be the smallest cut induced by the components in $\mathcal{P}^*$. We have

$$\lambda \leq w(C_{\min}) \leq \frac{\sum_{S\in\mathcal{P}^*} |E(S, V\setminus S)|}{|\mathcal{P}^*|} \leq \frac{2|E(G/\mathcal{P}^*)|}{|\mathcal{P}^*|} < 2\Phi. \quad \square$$

Thorup [23] defined the *ideal relative loads* $\ell^*(e)$ on the edges of G by the following.

1. Let $\mathcal{P}^*$ be an optimal partition with $\texttt{part_val}(\mathcal{P}^*) = \Phi$.
2. For all $e \in G/\mathcal{P}^*$, let $\ell^*(e) = 1/\Phi$.
3. For each $S \in \mathcal{P}^*$, recurse the procedure on the subgraph $G|S$.

Define the following notations:

$$E^X_{\circ\delta} = \{e \in E \mid \ell^X(e) \ \circ \ \delta\}$$

where X can be $\mathcal{T}$ or $*$, and $\circ$ can be $<$, $>$, $\leq$, $\geq$, or $=$. For example, $E^*_{<\delta}$ denote the set of edges with ideal relative loads smaller than δ.

Lemma 4.2 ([23], Lemma 14)**.** *The values of Φ are non-decreasing in the sense that for each $S \in P^*, \Phi_{G|S} \geq \Phi$*

Corollary 4.3. *Let $0 \leq l \leq 1/\Phi$. Each component H of the graph $(V, E^*_{\leq l})$ must have edge-connectivity of at least Φ.*

Proof. Accroding to how the ideal relative load was defined and Lemma 4.2, we must have $\Phi_H \geq \Phi$. By Lemma 4.1, $\lambda_H \geq \Phi_H \geq \Phi$. ☐

Thorup showed that the relative loads of a greedy tree packing with a sufficient number of trees approximate the ideal relative loads, due to the fact that greedily packing the trees simulates the multiplicative weight update method. He showed the following lemma.

Lemma 4.4 ([23], Proposition 16)**.** *A greedy tree packing $\mathcal{T}$ with at least $(6\lambda \ln m)/\epsilon^2$ trees, $\epsilon < 2$ has $|\ell^{\mathcal{T}}(e) - \ell^*(e)| \leq \epsilon/\lambda$ for all $e \in E$.*

4.2 Algorithms

In this section, we show how to approximate the value of the minimum cut as well as how to find an approximate minimum cut.

Algorithm for computing minimum cut value. The main idea is that if we have a nearly optimal tree packing, then either λ is close to 2Φ or all the minimum cuts are crossed exactly once by some trees in the tree packing.

Lemma 4.5. *Suppose that $\mathcal{T}$ is a greedy tree packing with at least $6\lambda \ln m/\epsilon^2$ trees, then $\lambda \leq (2+\epsilon) \cdot \texttt{pack_val}(\mathcal{T})$. Furthermore, if there is a minimum cut C such that it is crossed at least twice by every tree in $\mathcal{T}$, then $(2+\epsilon) \cdot \texttt{pack_val}(\mathcal{T}) \leq (1+\epsilon/2)\lambda$.*

Proof. By Lemmas 4.1 and 4.4, $1/\texttt{pack_val}(\mathcal{T}) \leq 1/\texttt{pack_val}(\mathcal{T}^*) + \epsilon/\lambda \leq 2/\lambda + \epsilon/\lambda$. Therefore, $\lambda \leq (2+\epsilon) \cdot \texttt{pack_val}(\mathcal{T})$.

If each tree in $\mathcal{T}$ crosses C at least twice, we have $\sum_{e \in C} \ell^{\mathcal{T}}(e) \geq 2$. Therefore,

$$2/\lambda \leq \sum_{e \in C} \ell^{\mathcal{T}}(e)/w(C) \leq \max_{e \in C} \ell^{\mathcal{T}}(e) \leq 1/\texttt{pack_val}(\mathcal{T})\,. \tag{1}$$

This implies that $(2+\epsilon) \cdot \texttt{pack_val}(\mathcal{T}) \leq (1+\epsilon/2)\lambda$. ☐

Using Lemma 4.5, we can obtain a simple algorithm for $(1+\epsilon)$-approximating the minimum cut *value*. First, greedily pack $\Theta(\lambda \log n/\epsilon^2)$ trees and compute the minimum cut that 1-respects the trees (using our algorithm in Section 3). Then, output the smaller value between the minimum cut found and $(2+\epsilon) \cdot \texttt{pack_val}(\mathcal{T})$. The running time is discussed in Section 4.3.

Algorithm for finding a minimum cut. More work is needed to be done if we want to *find* the $(1+\epsilon)$-approximate minimum cut (i.e. each node wants to know which side of the cut it is on). Let $\epsilon' = \Theta(\epsilon)$ be such that $(1-2\epsilon') \cdot (1-\epsilon') = 1/(1+\epsilon)$. Let $l_a = (1-2\epsilon')/\texttt{pack_val}(\mathcal{T})$. We describe our algorithm in Algorithm 4.1.

1: Find a greedy tree packing $\mathcal{T}$ with $(6\lambda \ln m)/\epsilon'^2$ trees in G.
2: Let C^* be the minimum cut among cuts that 1-respect a tree in $\mathcal{T}$.
3: Let $l_a = (1-2\epsilon')/\texttt{pack_val}(\mathcal{T})$.
4: **if** $(V, E^{\mathcal{T}}_{<l_a})$ has more than $(1-\epsilon')|V|$ components **then**
5: Let $C_{\min}$ be the smallest cut induced by the components in $(V, E^{\mathcal{T}}_{<l_a})$.
6: **else**
7: Let $C_{\min}$ be the cut returned by APPROX-MIN-CUT$(G/E^{\mathcal{T}}_{<l_a})$.
8: Return the smaller cut between C^* and $C_{\min}$.

Algorithm 4.1: APPROX-MIN-CUT(G)

The main result of this subsection is the following theorem.

Theorem 4.6. *Algorithm 4.1 gives a $(1+\epsilon)$-approximate minimum cut.*

The rest of this subsection is devoted to proving Theorem 4.6. First, observe that if a minimum cut is crossed exactly once by a tree in $\mathcal{T}$, then C^* must be a minimum cut. Otherwise, C is crossed at least twice by every tree in $\mathcal{T}$. In this case, we will show that the edges of every minimum cut will be included in $E^{\mathcal{T}}_{\geq l_a}$. As a result, we can contract each connected component in the partition $(V, E^{\mathcal{T}}_{<l_a})$ without contracting any edges of the minimum cuts.

If $(V, E^{\mathcal{T}}_{<l_a})$ has at most $(1-\epsilon')|V|$ components, then we contract each component and then recurse. The recursion can only happen at most $O(\log n/\epsilon)$ times, since the number of nodes reduces by a $(1-\epsilon')$ factor in each level. On the other hand, if $(V, E^{\mathcal{T}}_{<l_a})$ has more than $(1-\epsilon')|V|$ components, then we will show that one of the components induces an approximate minimum cut.

Lemma 4.7. *Let C be a minimum cut such that C is crossed at least twice by every tree in $\mathcal{T}$. For all $e \in C$, $\ell^{\mathcal{T}}(e) \geq (1-2\epsilon')/\texttt{pack_val}(\mathcal{T})$.*

Proof. The idea is to show that if an edge in $E(C)$ has a small relative load, then the average relative load over the edges in $E(C)$ will also be small. However, since each tree cross $E(C)$ twice, the average relative load should not be too small. Otherwise, a contradiction will occur.

Let $l_0 = \min_{e \in C} \ell^*(e)$ be the minimum ideal relative load over the edges in $E(C)$. Consider the induced subgraph $(V, E^*_{\leq l_0})$. $E(C)$ must contain some edges in a component of $(V, E^*_{\leq l_0})$, say component H. Notice that two endpoints of an edge in a minimum cut must lie on different sides of the cut. Therefore, $C \cap H$ must be a cut of H. By Corollary 4.3, $w(C \cap H) \geq \Phi$. Therefore, more than Φ edges in C have ideal relative loads equal to l_0. Since the maximum relative load of an edge is at most $\frac{1}{\Phi}$, $\sum_{e \in C} \ell^{\mathcal{T}^*}(e) \leq \Phi \cdot l_0 + (\lambda - \Phi) \cdot \frac{1}{\Phi} = \Phi \cdot l_0 + \frac{\lambda}{\Phi} - 1 < \Phi \cdot l_0 + 1$, where the last inequality follows by Lemma 4.1 that $\lambda < 2\Phi$.

On the other hand, since each tree in $\mathcal{T}$ crosses C at least twice, $\sum_{e \in C} \ell^{\mathcal{T}}(e) \geq 2$. By Lemma 4.4, $\sum_{e \in C} \ell^*(e) \geq 2 - \epsilon'$. Therefore, $\Phi \cdot l_0 + 1 > 2 - \epsilon'$, which implies

$$\begin{aligned} l_0 &\geq (1-\epsilon') \cdot \frac{1}{\Phi} > \frac{1}{\Phi} - \frac{2\epsilon'}{\lambda} && \lambda < 2\Phi \\ &\geq 1/\texttt{pack_val}(\mathcal{T}) - \frac{3\epsilon'}{\lambda} && \text{By Lemma 4.4} \end{aligned}$$

Therefore, by Lemma 4.4 again, for any $e \in E(C)$, $\ell^{\mathcal{T}}(e) \geq l_0 - \epsilon'/\lambda > 1/\texttt{pack_val}(\mathcal{T}) - 4\epsilon'/\lambda \geq (1-2\epsilon')/\texttt{pack_val}(\mathcal{T})$, where the last inequality follows from equation (1). □

Lemma 4.8. *Let $C_{\min}$ be the smallest cut induced by the components in $(V, E^{\mathcal{T}}_{<l_a})$. If $(V, E^{\mathcal{T}}_{<l_a})$ contains at least $(1-\epsilon')|V|$ components, then $w(C_{\min}) \leq (1+\epsilon)\lambda$.*

Proof. Let $comp(V, E^{\mathcal{T}}_{<l_a})$ denote the collection of connected components in $(V, E^{\mathcal{T}}_{<l_a})$, and n', the number of connected components in $(V, E^{\mathcal{T}}_{<l_a})$. By an averaging argument, we have

$$w(C_{\min}) \leq \frac{\sum_{S \in comp(V, E^{\mathcal{T}}_{<l_a})} |E(S, V \setminus S)|}{n'} = \frac{2|E(G/E^{\mathcal{T}}_{<l_a})|}{n'} \leq \frac{2|E(G/E^{\mathcal{T}}_{<l_a})|}{(1-\epsilon') \cdot |V|} \tag{2}$$

Next we will bound $|E(G/E^{\mathcal{T}}_{<l_a})|$. Note that for each $e \in E(G/E^{\mathcal{T}}_{<l_a})$, $\ell^{\mathcal{T}}(e) \geq (1-2\epsilon')/\texttt{pack_val}(\mathcal{T})$.

$$\begin{aligned} \sum_{e \in E(G/E^{\mathcal{T}}_{<l_a})} \ell^{\mathcal{T}}(e) &\geq |E(G/E^{\mathcal{T}}_{<l_a})| \cdot (1-2\epsilon') \cdot \left(\frac{1}{\texttt{pack_val}(\mathcal{T})}\right) \\ &\geq |E(G/E^{\mathcal{T}}_{<l_a})| \cdot \left(1-2\epsilon'\right) \cdot \frac{2}{\lambda}. && \text{(by Equation (1))} \end{aligned} \tag{3}$$

On the other hand,

$$\sum_{e \in E(G/E^{\mathcal{T}}_{<l_a})} \ell^T(e) \leq |V| - 1, \tag{4}$$

since each tree in $\mathcal{T}$ contains $|V|-1$ edges. Equations (3) and (4) together imply that

$$|E(G/E^{\mathcal{T}}_{<l_a})| \leq \frac{\lambda \cdot |V|}{2(1-2\epsilon')}.$$

By plugging in this into (Equation (2)), we get that

$$w(C_{\min}) \leq \frac{\lambda}{(1-2\epsilon')(1-\epsilon')} \leq (1+\epsilon)\lambda\,. \quad \square$$

4.3 Distributed Implementation

In this section, we describe how to implement Algorithm 4.1 in the distributed setting. To compute the tree packing $\mathcal{T}$, it is straightforward to apply $|\mathcal{T}|$ minimum spanning tree computations with edge weights equal to their current loads. This can be done in $O(|\mathcal{T}|(D + \sqrt{n}\log^* n))$ rounds by using the algorithm of Kutten and Peleg [12].

We already described how to computes the minimum cut that 1-respects a tree in $O(D + \sqrt{n}\log^* n)$ rounds in Section 3. To compute l_a, it suffices to compute `pack_val`$(\mathcal{T})$. To do this, each node first computes the largest relative load among the edges incident to it. By using the upcast and downcast techniques, the maximum relative load over all edges can be aggregated and boardcast to every node in $O(D)$ time. Therefore, we can assume that every node knows l_a now. Now we have to determine whether $(V, E^{\mathcal{T}}_{<l_a})$ has more than $(1-\epsilon')|V|$ components or not. This can be done by first removing the edges incident to each node with relative load at least l_a. Then label each node with the smallest ID of its reachable nodes by using Thurimella's connected component identification algorithm [24] in $O(D + \sqrt{n}\log^* n)$ rounds. The number of nodes whose label equals to its ID is exactly the number of connected component of the subgraph. This number can be aggregated along the BFS tree in $O(D)$ rounds after every node is labeled.

If $(V, E^{\mathcal{T}}_{<l_a})$ has more than $(1-\epsilon')|V|$ components, then we will compute the cut values induced by each component of $(V, E^{\mathcal{T}}_{<l_a})$. We show that it can be done in $O(D+\sqrt{n})$ rounds in Appendix A. On the contrary, if $(V, E^{\mathcal{T}}_{<l_a})$ has less than $(1-\epsilon')|V|$ components, then we will contract the edges with load less than l_a and then recurse. The contraction can be easily implemented by setting the weights of the edges inside contracted components to be -1, which is strictly less than the load of any edges. The MST computation will automatically treat them as contracted edges, since an MST must contain exactly $n'-1$ edges with weights larger than -1, where n' is the number of connected components. [3]

Time analysis. Suppose that we have packed t spanning trees throughout the entire algorithm, the running time will be $O(t(D+\sqrt{n}\log^* n))$. Note that $t = O(\epsilon^{-3}\lambda\log^2 n)$, because we pack at most $O(\epsilon^{-2}\lambda\log n)$ spanning trees in each level of the recursion and there can be at most $O(\epsilon^{-1}\log n)$ levels, since the number of nodes reduces by a $(1-\epsilon')$ factor in each level. The total running time is $O(\epsilon^{-3}\lambda\log^2 n \cdot (D + \sqrt{n}\log^* n))$.

[3] We note that the MST algorithm of [12] allows negative-weight edges.

Dealing with graphs with high edge connectivity. For graphs with $\lambda = \omega(\epsilon^{-2}\log n)$, we can use the well-known sampling result from Karger's [7] to construct a subgraph H that perserves the values of all the cuts within a $(1 \pm \epsilon)$ factor (up to a scaling) and has $\lambda_H = O(\epsilon^{-2}\log n)$. Then we run our algorithm on H.

Lemma 4.9 ([6], Corollary 2.4)**.** *Let G be any graph with minimum cut λ and let $p = 2(d+2)(\ln n)/(\epsilon^2\lambda)$. Let $G(p)$ be a subgraph of G with the same vertex set, obtained by including each edge of G with probability p independently. Then the probability that the value of some cut in $G(p)$ has value more than $(1+\epsilon)$ or less than $(1-\epsilon)$ times its expected value is $O(1/n^d)$.*

In particular, let $\epsilon' = \Theta(\epsilon)$ such that $(1+\epsilon) = (1+\epsilon')^2/(1-\epsilon')$. First we will compute λ', a 3-approximation of λ, by using Ghaffari and Kuhn's algorithm. Let $p = 6(d+2)\ln n/(\epsilon'^2\lambda')$ and $H = G(p)$. Since p is at least $2(d+2)\ln n/(\epsilon'^2\lambda)$, by Lemma 4.9, for any cut C, w.h.p. $(1-\epsilon')p \cdot w_G(C) \le w_{H_i}(C) \le (1+\epsilon')p \cdot w_G(C)$. Let C^* be the $(1+\epsilon')$-approximate minimum cut we found in H. We have that w.h.p. for any other cut C',

$$w_G(C^*) \le \frac{1}{p} \cdot \frac{w_{H_i}(C^*)}{1-\epsilon'} \le \frac{1}{p} \cdot \frac{(1+\epsilon')\lambda_H}{1-\epsilon'} \le \frac{1}{p} \cdot \frac{(1+\epsilon')w_{H_i}(C')}{1-\epsilon'} \le \frac{(1+\epsilon')^2}{1-\epsilon'} \cdot w_G(C') = (1+\epsilon)w_G(C')$$

Thus, we will find an $(1+\epsilon)$-approximate minimum cut in $O(\epsilon^{-5}\log^3 n(D+\sqrt{n}\log^* n))$ rounds.

Computing the exact minimum cut. To find the exact minimum cut, first we will compute a 3-approximation of λ, λ', by using Ghaffari and Kuhn's algorithm [4] in $O(\lambda \log n \log\log n(D + \sqrt{n}\log^* n))$ rounds.[4] Now since $\lambda \le \lambda' \le 3\lambda$, by applying our algorithm with $\epsilon = 1/(\lambda'+1)$, we can compute the exact minimum cut in $O(\lambda^4 \log^2 n(D+\sqrt{n}\log^* n))$ rounds.

Estimating the value of λ. As described in Section 4.2, we can avoid the recursion if we just want to compute an approximation of λ without actually finding the cut. This gives an algorithm that runs in $O(\epsilon^{-2}\lambda\log n \cdot (D+\sqrt{n}\log^* n))$ time. Also, the exact value of λ can be computed in $O((\lambda^3 + \lambda\log\log n)\log n(D+\sqrt{n}\log^* n))$ rounds. Notice that the $\lambda\log\log n$ factor comes from Ghaffari and Kuhn's algorithm for approximating λ within a constant factor. Similarly, using Karger's sampling result, we can $(1+\epsilon)$-approximate the value of λ in $O(\epsilon^{-5}\log^2 n\log\log n(D+\sqrt{n}\log^* n))$ rounds.

4.4 Sequential Implementation

We show that Algorithm 4.1 can be implemented in the sequential setting in $O(\epsilon^{-3}\lambda(m+n\log n)\log n)$ time. To get the stated bound, we will show that the number of edges decreases geometrically each time we contract the graph.

Lemma 4.10. *If $(V, E^{\mathcal{T}}_{<l_a})$ has less than $(1-\epsilon')|V|$ components, then $|E(G/E^{\mathcal{T}}_{<l_a})| \le |E(G)|/(1+\epsilon')$.*

Proof. Consider a component S of $(V, E^{\mathcal{T}}_{<l_a})$. Since $E(S) \subseteq E^{\mathcal{T}}_{<l_a}$ and $|T \cap E(S)| \ge |S|-1$, we have $|S|-1 \le \sum_{e\in S}\ell^{\mathcal{T}}(e) < l_a|E(S)|$. By summing this inequality over all components of $(V, E^{\mathcal{T}}_{<l_a})$, we have

$$l_a|E^{\mathcal{T}}_{<l_a}| \ge |V| - |V(G/E^{\mathcal{T}}_{<l_a})| > |V| - (1-\epsilon')|V| = \epsilon'|V| \tag{5}$$

If we sum up the relative load over each $e \in E(G/E^{\mathcal{T}}_{<l_a})$, we have

$$l_a|E(G/E^{\mathcal{T}}_{<l_a})| \le \sum_{e\in E(G/E^{\mathcal{T}}_{<l_a})} \ell^{\mathcal{T}}(e) \le |V| \tag{6}$$

[4] Ghaffari and Kuhn's result runs in $O(\log^2 n\log\log n(D+\sqrt{n}\log^* n))$ rounds. However, without using Karger's random sampling beforehand, it runs in $O(\lambda\log n\log\log n(D+\sqrt{n}\log^* n))$ rounds, which will be absorbed by the running time of our algorithm for the exact minimum cut.

Dividing (5) by (6), we have $|E^{\mathcal{T}}_{<l_a}|/|E(G/E^{\mathcal{T}}_{<l_a})| > \epsilon'$ and therefore, $|E(G/E^{\mathcal{T}}_{<l_a})| < (|E^{\mathcal{T}}_{<l_a}| + |E(G/E^{\mathcal{T}}_{<l_a})|)/(1+\epsilon') = |E(G)|/(1+\epsilon')$. □

Let $\texttt{MST}(n,m)$ denote the time needed to find an MST in a graph with n-vertices and m-edges. Note that Karger [8] showed that the values of the cuts that 1-respect a tree can be computed in linear time. The total running time of Algorithm 4.1 will be

$$O\left(\epsilon'^{-2}\lambda \log n \cdot \sum_{i=0}^{\infty} \texttt{MST}(n(1-\epsilon')^i, m/(1+\epsilon')^i)\right).$$

We know that $\texttt{MST}(n,m) = O(m)$ by using the randomized linear time algorithm from [9] and notice that $\epsilon = \Theta(\epsilon')$, the running time will be at most $O(\epsilon^{-3}\lambda m \log n)$.

If the graph is dense or the cut value is large, we may want to use the sparsification results to reduce m or λ. First estimate λ up to a factor of 3 by using Matula's algorithm [14] that runs in linear time. By using Nagamochi and Ibaraki's sparse certificate algorithm [15], we can get the number of edges down to $O(n\lambda)$. By using Karger's sampling result, we can bring λ down to $O(\log n/\epsilon^2)$. The total running time is therefore $O(m + \epsilon^{-7} n \log^3 n)$ (by plugging $\lambda = \log n/\epsilon^2$ and $m = n \log n/\epsilon^2$ in the running time in the previous paragraph). [5]

Acknowledgment: D. Nanongkai would like to thank Thatchaphol Saranurak for bringing Thorup's tree packing theorem [23] to his attention.

References

[1] A. Das Sarma, S. Holzer, L. Kor, A. Korman, D. Nanongkai, G. Pandurangan, D. Peleg, and R. Wattenhofer. Distributed verification and hardness of distributed approximation. *SIAM J. Comput.*, 41(5):1235–1265, 2012. 1, 2

[2] H. N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *J. Comput. Syst. Sci.*, 50(2):259 – 273, 1995. 1

[3] J. A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998. 2

[4] M. Ghaffari and F. Kuhn. Distributed minimum cut approximation. In *DISC*, pages 1–15, 2013. 1, 2, 11

[5] D. R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *SODA*, pages 21–30, 1993. 1

[6] D. R. Karger. Random sampling in cut, flow, and network design problems. In *STOC*, pages 648–657, 1994. 1, 2, 11

[7] D. R. Karger. Using randomized sparsification to approximate minimum cuts. In *SODA*, pages 424–432, 1994. 1, 2, 11

[8] D. R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000. 1, 3, 7, 12

[9] D. R. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995. 12

[10] D. R. Karger and C. Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. In *STOC*, pages 757–765, 1993. 1

[11] M. Khan and G. Pandurangan. A fast distributed approximation algorithm for minimum spanning trees. *Distributed Computing*, 20(6):391–402, 2008. 2

[5] In this case, we can also use Prim's deterministic MST algorithm without increasing the total running time. This is because Prim's algorithm runs in $O(m + n\log n)$ time, the $n \log n$ term will be absorbed by m, as we have used $m = n\log n/\epsilon^2$.

[12] S. Kutten and D. Peleg. Fast distributed construction of small k-dominating sets and applications. *J. Algorithms*, 28(1):40–66, 1998. 1, 2, 4, 10

[13] Z. Lotker, B. Patt-Shamir, and A. Rosén. Distributed approximate matching. *SIAM J. Comput.*, 39(2):445–460, 2009. 2

[14] D. W. Matula. A linear time $2 + \epsilon$ approximation algorithm for edge connectivity. In *SODA*, pages 500–504, 1993. 1, 2, 12

[15] H. Nagamochi and T. Ibaraki. Computing edge-connectivity in multigraphs and capacitated graphs. *SIAM J. Discret. Math.*, 5(1):54–66, 1992. 1, 2, 12

[16] D. Nanongkai. Brief announcement: almost-tight approximation distributed algorithm for minimum cut. In *PODC*, pages 382–384, 2014. 1

[17] D. Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *STOC*, pages 565–573, 2014. 2

[18] C. St. J. A. Nash-Williams. Edge-disjoint spanning trees of finite graphs. *J. London Math. Soc.*, s1-36(1):445–450, 1961. 7

[19] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach.* SIAM Monographs on Discrete Mathematics ans Applications, 2000. 2, 3

[20] D. Pritchard and R. Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transactions on Algorithms*, 7(4):46, 2011. 1, 2

[21] M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997. 1

[22] H.-H. Su. Brief annoucement: a distributed minimum cut approximation scheme. In *SPAA*, pages 217–219, 2014. 1

[23] M. Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007. 1, 6, 7, 8, 12

[24] R. Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. *J. Algorithms*, 23(1):160–179, 1997. 10

[25] W. T. Tutte. On the problem of decomposing a graph into n connected factors. *J. London Math. Soc.*, s1-36(1):221–230, 1961. 7

Appendix

A Finding cuts with respect to connected components

In this section, we solve the following problem. We are given a set of connected components $\{H_1, H_2, \dots, H_k\}$ of the network G (each node knows which of its neighbors are in the same connected component), and we want to compute, for each i, the value $w(C_i)$ where C_i is the cut with respect to H_i; i.e., $C_i = (V(H_i), V(G) \setminus V(H_i))$. Every node in C_i should know $w(C_i)$ in the end. We show that this can be done in $O(n^{1/2} + D)$ rounds. The main idea is to deal with "big" and "small" components separately, where a component is big if it contains at least $n^{1/2}$ nodes and it is small otherwise. There are at most $n^{1/2}$ big components, and thus the cut value information for these components can be aggregated quickly through the BFS tree of the network. The cut value of each small component will be computed locally within the component. The detail is as follows.

First, we determine for each component H_i whether it is big or small, which can be done by simply counting the number of nodes in each component, such as the following. Initially, every node sends its ID to its neighbors in the same component. Then, for $n^{1/2}+1$ rounds, every node sends the smallest ID it has received so far to its neighbors in the same component. For each node v, let s_v be the smallest ID that v has received after $n^{1/2}+1$ rounds. If s_v is v's own ID, it construct a BFS tree T_v of depth at most $n^{1/2}+1$, and use T_v to count the number of nodes in T_v. (There will be no congestion caused by this algorithm since no other node within distance $n^{1/2}+1$ from v

will trigger another BFS tree construction.) If the number of nodes in T_v is at most $n^{1/2}$, then v broadcasts to the whole network that the component containing it is small.

Now, to compute $w(C_i)$ for a small component H_i, we simply construct a BFS tree rooted at the node with smallest ID in C_i and compute the sum $\sum_{u \in V(H_i), v \notin V(H_i)} w(u,v)$ through this tree. To compute $w(C_i)$ for a big component H_j, we compute the sum $\sum_{u \in V(H_i), v \notin V(H_i)} w(u,v)$ thorough the BFS tree of network G. Since there are at most $n^{1/2}$ big components, this takes $O(n^{1/2} + D)$ time.