Brief Announcement: Almost-Tight Approximation Distributed Algorithm for Minimum Cut

Danupon Nanongkai*

Abstract

In this short paper, we present an improved algorithm for approximating the minimum cut on distributed (CONGEST) networks. Let $\lambda$ be the minimum cut. Our algorithm can compute $\lambda$ exactly in $\tilde{O}((\sqrt{n} + D) \text{poly}(\lambda))$ time, where $n$ is the number of nodes (processors) in the network, $D$ is the network diameter, and $\tilde{O}$ hides poly log $n$. By a standard reduction, we can convert this algorithm into a $(1 + \epsilon)$-approximation $\tilde{O}((\sqrt{n} + D)/\text{poly}(\epsilon))$-time algorithm. The latter result improves over the previous $(2 + \epsilon)$-approximation $\tilde{O}((\sqrt{n} + D)/\text{poly}(\epsilon))$-time algorithm of Ghaffari and Kuhn (DISC 2013). Due to the lower bound of $\tilde{\Omega}((\sqrt{n} + D))$ by Das Sarma et al. (SICOMP 2013), this running time is tight up to a poly log $n$ factor. Our algorithm is an extremely simple combination of Thorup’s tree packing theorem [Combinatorica 2007], Kutten and Peleg’s tree partitioning algorithm [J. Algorithms 1998], and Karger’s dynamic programming [JACM 2000].
1 Introduction

Problem In this paper, we study the time complexity of the fundamental problem of computing minimum cut on distributed network. Given a graph $G$, edge weight assignment $w$, and any set $X$ of nodes in $G$, the cut $\mathcal{E}(X)$ is defined as $\mathcal{E}(X) = \sum_{(x,y) \in E(G)} w(x,y)$. Our goal is to find $\lambda(G) = \min_{\emptyset \neq X \subseteq V} \mathcal{E}(X)$.

Communication Model We use a standard message passing network (the CONGEST model [7]). Throughout the paper, we let $n$ be the number of nodes and $D$ be the diameter of the network. Every node is assumed to have a unique ID, and initially knows the weights of edges incident to it. The execution in this network proceeds in synchronous rounds and in each round, each node can send a message of size $O(\log n)$ bits to each of its neighbors. The goal of the problem is find the minimum or approximately minimum cut $X$. (Every node outputs whether it is in $X$ in the end of the process.) The time complexity is the number of rounds needed to compute this. (For more detail, see [3].)

Previous Work The current best algorithm is by Ghaffari and Kuhn [3] which takes $\tilde{O}(\sqrt{n} + D)$ time with an approximation ratio of $(2 + \epsilon)$ (due to the space limit, we refer the readers to [9, Lemma 7] for the statement of Karger’s sampling result). The running time of this algorithm matches the lower bound of Das Sarma et al. [1] which showed that this problem cannot be computed faster than $\tilde{O}(\sqrt{n} + D)$ even when we allow a large approximation ratio. (This lower bound was also shown to hold even when a quantum communication is allowed [2], and when a capacity of an edge is proportional to its weight [3].) For a more comprehensive literature review, see [3].

Our Results Our main result is a distributed algorithm that can compute the minimum cut exactly in $\tilde{O}(\sqrt{n} + D) \ poly(\lambda)$ time. For the case where the minimum cut is small (i.e. $\tilde{O}(1)$), the running time of our algorithm matches the lower bound [1, 3]. When the minimum cut is large, Karger’s edge sampling technique [4] can be used to reduce the minimum cut to $\tilde{O}(1)$ with the cost of $(1 + \epsilon)$ approximation factor (due to the space limit, we refer the readers to [9, Lemma 7] for the statement of Karger’s sampling result). This makes our algorithm a $(1 + \epsilon)$-approximation $\tilde{O}(\sqrt{n} + D)$-time one, improving the previous algorithm of Ghaffari and Kuhn [3].

Techniques Our algorithm is a simple combination of techniques from [9, 6, 5]. The starting point of our algorithm is Thorup’s tree packing theorem, 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. (Due to the space limit, we refer the readers to [9, Theorem 9] for the full statement.) Since we can use the $\tilde{O}(\sqrt{n} + D)$-time algorithm of Kutten and Peleg [6] 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 2). Solving this problem is our main result.

To solve this problem, we use simple observation of Karger [5] which reduces the problem to computing the sum of degree and the number of edges contained in a subtree rooted at each node. We use this observation along with Kutten and Peleg’s tree partitioning [6] to quickly compute these quantities. This requires several (elementary) steps, which we will discuss in more detail in Section 2.

Concurrent Result Independent from our work, Su [8] also achieved a $(1 + \epsilon)$-approximation $\tilde{O}(\sqrt{n} + D)$-time algorithm for this problem. His starting point is, like ours, Thorup’s theorem [9]. The way he finds the minimum cut that $1$-respects a tree is, however, very different. In particular, he uses edge sampling to make the minimum cut of a certain graph be one and use Thurimella’s algorithm [10] to find a bridge. (See Algorithm 2 in [8] for details.) This gives a nice and simple way to achieve essentially the same approximation result as ours, with a small drawback that minimum cut cannot be computed exactly, even when it is small.
2 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 define 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 \( v \). The problem is then to compute \( c^* = \min_{v \in V(G)} \mathcal{C}(v^\downarrow) \).

**Theorem 2.1 (Main Result).** There is an \( \tilde{O}(n^{1/2} + D) \)-time distributed algorithm that can compute \( c^* \) as well as find a node \( v \) such that \( c^* = \mathcal{C}(v^\downarrow) \).

In fact, at the end of our algorithm every node \( v \) knows \( \mathcal{C}(v^\downarrow) \). Our algorithm is inspired by the following observation used in Karger’s dynamic programming [5]. 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 endpoints’ least common ancestor is \( v \). Let \( \delta^i(v) = \sum_{u \in v^i} \delta(u) \) and \( \rho^i(v) = \sum_{u \in v^i} \rho(u) \).

**Lemma 2.2 (Karger [5] (Lemma 5.9)).** \( \mathcal{C}(v^\downarrow) = \delta^i(v) - 2\rho^i(v) \).

Our algorithm will make sure that every node \( v \) knows \( \delta^i(v) \) and \( \rho^i(v) \). By Lemma 2.2, this will be sufficient for every node \( v \) to compute \( c^* \). The algorithm is divided in several steps, as follows.

**Step 1: Partition \( T \) into Fragments and Compute “Fragment Tree” \( T_F \)** We use the algorithm of Kutten and Peleg [6] Section 3.2] to partition nodes in tree \( T \) into \( O(\sqrt{n}) \) subtrees, where each subtree has \( O(\sqrt{n}) \) diameter. (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 \( \text{id}(F_i) = \min_{u \in F_i} \text{id}(u) \) be the ID of \( F_i \). We can assume that every node in \( F_i \) knows \( \text{id}(F_i) \). This can be achieved in \( O(\sqrt{n}) \) time by a communication within each fragment.

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 \( O(\sqrt{n}) \) time since there are \( O(\sqrt{n}) \) edges in \( T \) that links between different fragments. Note that this process also makes every node knows 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 \( A(15) \).) The goal of this step is to make every node \( v \) know (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 \( O(\sqrt{n}) \) time since there are \( O(\sqrt{n}) \) fragments to aggregate. 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 \( v \) 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 \( O(\sqrt{n}) \), this process takes \( O(\sqrt{n}) \) time. With some

\[1\text{To be precise, we compute a } (\sqrt{n} + 1, O(\sqrt{n})) \text{ spanning forest. Also note that we in fact do not need this algorithm since we obtain } T \text{ by using Kutten and Peleg's MST algorithm, which already computes the } (\sqrt{n} + 1, O(\sqrt{n})) \text{ spanning forest as a subroutine. See [6] for details.}\]
minor modifications, we can also make every node $v$ know $F(u)$ 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 parents 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 $\delta^i(v)$** For every fragment $F_i$, we let $\delta(F_i) = \sum_{v \in F_i \cap v^i} \delta(v)$. For every node $v$ in every fragment $F_i$, we will compute $\delta^i(v)$ by separately computing (i) $\sum_{u \in F_i \cap u^i} \delta(u)$ and (ii) $\sum_{F_i \in F(u)} \delta(F_i)$. The first quantity can be computed in $O(\sqrt{n})$ time by computing the sum within $F_i$ (every node $v$ sends the sum $\sum_{u \in F_i \cap u^i} \delta(u)$ to its parent). To compute the second quantity, it suffices to make every node know $\delta(F_i)$ for all $i$ by every node $v$ already knows $F(v)$. To do this, we make every root $v_j$ 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 $v_i$ 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 $x^i$ and $y^i$ contain some fragments (e.g. nodes 0 and 1 in Figure 1a). In other words, it is a point where two fragments “merge”. Let $T'_F$ be the following tree: Nodes in $T'_F$ are roots of fragments ($r_j$’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 $u^i$ 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 the ancestors in $A(v)$. So, we can make every node knows $T'_F$ in $O(\sqrt{n})$ 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 $\rho^i(v)$** We now count, for every node $v$, the number of edges whose least common ancestor (LCA) of its 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 10). 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. 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 $z^i$ contains $F_i$ and $F_j$. Since both $x$ and $y$ knows $F_i$ and their ancestors in $T'_F$, they can find $z$ by exchanging the list of their ancestors in $T'_F$. 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 $z^i$ 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 $\rho^i(v)$ for every node $v$ by splitting edges $(x, y)$ whose LCA is $v$ into two types (see Figure 11): (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 $v^i \cap F_i$ by computing the sum through the tree $F_i$. Note that, to do this, every node $v$ 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).

**Acknowledgment:** The author would like to thank Thatchaphol Saranurak for bringing Thorup’s tree packing theorem [9] to his attention.
References

[1] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In STOC, pages 363–372, 2011.

[2] Michael Elkin, Hartmut Klauck, Danupon Nanongkai, and Gopal Pandurangan. Quantum distributed network computing: Lower bounds and techniques. CoRR, abs/1207.5211, 2012.

[3] Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In DISC, pages 1–15, 2013.

[4] David R. Karger. Random sampling in cut, flow, and network design problems. In STOC, pages 648–657, 1994.

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

[6] Shay Kutten and David Peleg. Fast distributed construction of small $k$-dominating sets and applications. J. Algorithms, 28(1):40–66, 1998. Announced at PODC 1995.

[7] David Peleg. Distributed Computing: A Locality-Sensitive Approach. SIAM Society for Industrial and Applied Mathematics Monographs on Discrete Mathematics and Applications, Philadelphia, 2000.

[8] Hsin-Hao Su. A distributed minimum cut approximation scheme. CoRR, abs/1401.5316, 2014.

[9] Mikkel Thorup. Fully-dynamic min-cut. Combinatorica, 27(1):91–127, 2007. Announced at STOC 2001.

[10] Ramakrishna Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. J. Algorithms, 23(1):160–179, 1997. Announced at PODC 1995.
Figure 1