Finding a Small Vertex Cut on Distributed Networks

Yonggang Jiang
MPI-INF and Saarland University
Germany
yjiang@mpi-inf.mpg.de

Sagnik Mukhopadhyay
University of Sheffield
United Kingdom
s.mukhopadhyay@sheffield.ac.uk

ABSTRACT

We present an algorithm for distributed networks to efficiently find a small vertex cut in the CONGEST model. Given a positive integer κ, our algorithm can, with high probability, either find κ vertices whose removal disconnects the network or return that such κ vertices do not exist. Our algorithm takes $\kappa^3 \cdot \tilde{O}(D+\sqrt{n})$ rounds, where n is the number of vertices in the network and D denotes the network's diameter. This implies $\tilde{O}(D+\sqrt{n})$ round complexity whenever $\kappa = \text{polylog}(n)$.

Prior to our result, a bound of $\tilde{O}(D)$ is known only when $\kappa = 1, 2$ [Parter, Petruschka DISC'22]. For $\kappa \geq 3$, this bound can be obtained only by an $O(\log n)$-approximation algorithm [Censor-Hillel, Ghaffari, Kuhn PODC'14], and the only known exact algorithm takes $O\left((\kappa\Delta D)^{O(\kappa)}\right)$ rounds, where Δ is the maximum degree [Parter DISC'19]. Our result answers an open problem by Nanongkai, Saranurak, and Yingchareonthawornchai [STOC'19].

CCS CONCEPTS

• **Theory of computation** → **Distributed algorithms**.

KEYWORDS

vertex connectivity, congest model, vertex cut

ACM Reference Format:
Yonggang Jiang and Sagnik Mukhopadhyay. 2023. Finding a Small Vertex Cut on Distributed Networks. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC '23), June 20–23, 2023, Orlando, FL, USA*. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3564246.3585201

1 INTRODUCTION

For any undirected non-complete[1] graph $G = (V, E)$, a set $S \subseteq V$ is called a *vertex cut* if $G \setminus S$ contains at least two connected components, where $G \setminus S$ is obtained by removing vertices in S from G. In the *vertex cut* or *vertex connectivity* problem, we are given a positive integer κ and want to either find a vertex cut of size at most κ or to answer that such vertex cut does not exist. Vertex cut is a fundamental graph property and computing it is one of the most basic problems in graph algorithms. For example, it quantifies the vulnerability of a communication network in terms of the minimum number of vertices whose failures can disconnect the network. In the *sequential* model, this problem has been extensively studied over many decades (e.g. [2, 9, 18, 19, 21, 22, 26, 37–40, 43, 48, 50, 54, 58, 67]). For $\kappa = 1$, a linear-time algorithm via depth-first search was long known due to Tarjan [67]. For $\kappa = 2$, the linear-time algorithm was due to Hopcroft and Tarjan [39]. For $\kappa = \text{polylog}(n)$, an $\tilde{O}(m\kappa^2)$-time algorithm was recently discovered by [21, 58]. For other values of κ, a reduction to maxflow by [48] together with the very recent fast maxflow algorithm of [8] led to an almost-linear time algorithm.

To conclude, the vertex cut/connectivity problem is almost solved in the sequential setting. However, when it comes to *distributed networks computing their own vertex cut*, much less is known. This is the case even when it wants to find a few (say, $\kappa = 2$) vertices whose failures might destroy its communication. A distributed algorithm for finding a small vertex cut is the focus of this paper.

Distributed Vertex Cut. We study computing the vertex cut problem in the CONGEST model of distributed networks. In this model, an undirected graph $G = (V, E)$ is given as the communication network. Two important parameters are $n := |V|$ and D, the diameter of G. Time is divided into discrete rounds. In each round, each vertex can send an $O(\log n)$ bits message to each of its neighbors. After each round, each vertex can locally perform arbitrary computation and decide what to send in the next round. Initially, each vertex is given a specified input indicating some local information of the network (e.g. neighbors and weights of its incident edges). For the vertex cut problem, the input of each vertex is simply the set of its neighbors and integer κ. After several rounds, all vertices are expected to terminate and generate the desired output. For the case of the vertex cut problem, we expect at most κ vertices to identify themselves as being in a vertex cut if such a cut exists; otherwise, every vertex knows that such a cut does not exist. The goal is to minimize the number of rounds before all vertices terminate.

The CONGEST model is a standard model for studying basic graph algorithms in the message-passing distributed networks, e.g. minimum spanning tree (MST), shortest paths, min-cut, and approximate maxflow [12, 14, 16, 17, 20, 23, 28, 30, 31, 44, 59, 62]. These problems typically admit a trivial lower bound of $\Omega(D)$; thus, the focus is usually on the dependency on n. A large number of graph problems were shown to require $\tilde{\Theta}(D + \sqrt{n})$ rounds, and this bound has become a gold standard.[2] Examples of such problems include MST [12, 16, 23, 44, 62], approximate shortest paths [36, 47, 59], approximate 2-edge connected spanning subgraph (2-ECSS) [13, 15], tree packing [3] and approximate maxflow [29]. For

[1]For complete graphs, the problem is trivial so we ignore this case.

[2]Throughout, $\tilde{O}$, $\tilde{\Omega}$, and $\tilde{\Theta}$ hide polylog(n).

cut-related problems, a line of work (e.g. [11, 12, 14, 30, 32, 59]) led to an $\tilde{O}(D+\sqrt{n})$ bound for computing *edge cut* λ that holds even for weighted graphs [14, 52]. The bound matches the lower bounds from [12, 30] (the lower bounds hold when λ is large enough).[3] Moreover, when the edge cut λ is small, better algorithms exist: For $\lambda \in \{1,2\}$, the problem can be solved in $O(D)$ time [63]. For other values of λ, there is a $O((\lambda D)^{O(\lambda)})$ rounds algorithm[60]. (The last bound is small under a typical assumption that $D \ll n$.)

In sharp contrast with the above, our understanding of distributed vertex cut is much less complete. To the best of our knowledge, existing algorithms consist of

(1) an $O(D+\sqrt{n}\log^*(n))$-round algorithm that works only when $\kappa = 1$ [68],
(2) an $O(D+\Delta/\log n)$-round algorithm that works only when $\kappa = 1$ [63] (Δ denotes the maximum degree),
(3) an $O(\log n)$-approximation $\tilde{O}(\sqrt{n}+D)$-round algorithm [4], and
(4) an $O\left((\kappa\Delta D)^{O(\kappa)}\right)$-round algorithm [60],
(5) an $\widetilde{O}(D)$-round algorithm that works only when $\kappa = 1, 2$ [61].

Thus, even to find $\kappa = 3$ vertices that can disconnect the network, the available solutions are to either settle with a much bigger approximate solution of size $\Theta(\log n)$ [5] or find an exact solution in $O\left((\kappa\Delta D)^{O(\kappa)}\right)$ time [60] which can be prohibitively slow for typical networks with large-degree "hubs" (e.g. the star networks). In other words, even for $\kappa = 3$ we are already very far from the typical $\tilde{O}(\sqrt{n}+D)$-time exact algorithms!

Challenges. A fundamental difficulty in solving the vertex cut problem is its tight connection to *maxflow computation.* For example, while edge cut algorithms that are faster than solving maxflow were known in the sequential model for many decades (e.g. [24, 25, 41, 52, 53], it was only very recently that a vertex cut algorithm that is as fast as solving maxflow (and not faster) was found [48]. The situation is even worse in the distributed setting. For example, consider the case where we know two vertices s and t such that removing κ vertices in G would disconnect s from t (this is a basic case that all the state-of-the-art sequential algorithms have to solve [21, 48, 58]). When $\kappa = O(1)$, one can solve vertex cut in linear time in the sequential model using the Ford-Fulkerson algorithm. In contrast, in the distributed setting we cannot even solve this case in the typical $\tilde{O}(\sqrt{n}+D)$ rounds because it generalizes the *distributed reachability problem*, whose best-known round complexities are $\tilde{O}(D+\sqrt{n}D^{1/4})$ [33] and $\widetilde{O}(\sqrt{n}+n^{1/3+o(1)}\cdot D^{2/3})$ [51]. More generally, the distributed setting poses an additional challenge for computing vertex cut because there is no non-trivial maxflow algorithm available.[4] Thus, to design distributed vertex cut algorithms, one needs to overcome fundamental questions of whether one could avoid maxflow computations or develop maxflow algorithms specialized for solving vertex cut. Since efficient maxflow algorithms are not available in many models of computation (e.g. graph streaming and parallel computing), answering these questions may lead to efficient vertex cut algorithms in other models as well.

[3][12] proved a lower bound of $\tilde{\Omega}(\sqrt{n})$ for computing weighted mincut on some graphs of diameter $D = \Theta(\log n)$. For the unweighted case, it follows from [30, Theorem 6.4] that for any $\epsilon > 0$, there is a lower bound of $\tilde{\Omega}(n^{1/2-\epsilon})$ some graphs with diameter $D = \tilde{O}(n^{1/2-3\epsilon})$ and edge cut $n^{2\epsilon}$.

[4]The exception is the approximate maxflow algorithm of [28]. However, approximate maxflow was not known to be useful for solving vertex cut.

1.1 Our Result

We show that, in $\tilde{O}(D+\sqrt{n})$ rounds, a distributed network can find up to $O(\text{polylog}(n))$ vertices that can disconnect itself. More generally, our result is the following.

Theorem 1.1 (Informal. See Theorem 2.11 for a formal version.). *There is a randomized algorithm in the CONGEST model that, with input $\kappa < n^{1/4}$ and undirected graph G, takes $\kappa^3 \cdot \tilde{O}(D+\sqrt{n})$ rounds and determine whether G is κ-vertex-connected or not; if not, output the minimum vertex cut.*[5]

Our bound can be thought of as generalizing the $\tilde{O}(D+\sqrt{n})$ bound of [68] that works only when $\kappa = 1$ to any $\kappa = O(\log n)$; however, the techniques we use are very different. It is sublinear in n as long as $\kappa \ll n^{1/6}$. Our result answers an open problem from [58].

1.2 Techniques

We provide a detailed overview of this framework and our algorithm in the next section. Here, we discuss some challenges and techniques to overcome them that might be of independent interest. Our algorithm follows the framework used by the algorithms of [21, 58] for solving vertex cut in $\tilde{O}(m\kappa^2)$ time in the sequential model, where m denotes the number of edges. These algorithms consider two types of vertex cuts of size κ (assuming that they exist): a vertex cut that leads to a small connected component C is called *unbalanced* and otherwise it is called *balanced*.

To find these cuts, we have to execute some *maxflow* algorithms which keep finding *augmenting paths*. For an intuition, suppose that there are κ internally vertex-disjoint (s,t)-paths between two vertices s and t. An augmenting path is an st-path that, together with the existing paths, let us create $\kappa+1$ internally vertex-disjoint (s,t)-paths. (See Fig. 1 for an example and Section 2 for a more detailed definition.) Finding an augmenting path is useful because we can show that it exists if and only if there is no vertex cut of size κ that disconnects s and t. We now consider finding two types of cuts. Note that below we use 'Lemma' for lemmas that are used to provide intuition and are not actually proven.

Finding Unbalanced Cuts: Local Flows and Resolving Congestions ('Lemma' 2.6). To find unbalanced cuts, [21, 58] use *local flow* algorithms. Like many maxflow algorithms, a local flow algorithm keeps finding augmenting paths to increase the flow size; however, under some conditions, it can find augmenting paths *without reading the whole input graph.* For example, for vertex cut, [21, 58] use local flow algorithms to solve a problem where, given a vertex s in the above small connected component C, the algorithms can find the cut vertices in time roughly the size of the connected component C defined above (more precisely, the *volume* of C), which can

[5]When $\kappa \geq n^{1/4}$, our running time guarantee becomes at least $\Omega(kn)$, which is quite bad, so we do not consider this case here. Also notice that by running Ford–Fulkerson algorithm from a sampled node to any other nodes in parallel, one can easily get a $O(kn)$ algorithm.

be much less than the size of the whole input graph. By not reading the whole graph, we can execute multiple local flow algorithms in near-linear time in total. This feature plays a key role in designing many efficient sequential algorithms, e.g. finding balanced cuts [65, 66]), edge cut [37, 42], and dynamically maintaining expanders [10, 56, 57, 65, 69].

Applying the above idea in the CONGEST model, however, requires solving the *congestion issue*: many augmenting paths from different executions may go through the same edge. For example, the sequential vertex cut algorithms of [21, 58] need to compute $\Omega(n)$ local flows at some point, and we cannot rule out the case where all these executions require augmenting paths that share the same edge, which would cause $\Omega(n)$ rounds to modify all $\Omega(n)$ flows along these augmenting paths.

Congestion is a fundamental issue in the CONGEST model (thus the name). It is typically avoided by not executing too many algorithms in parallel. However, for vertex cut, we do not know how to avoid this. As far as we know, the same issue also arose in the *distributed expander decomposition computation* [6, 7], where the authors use *PageRank* algorithms instead of local flow algorithms (both algorithms can be used to compute the expander decomposition in the sequential model). Then, they exploit the property of PageRank to show that there is not much congestion, thus the congestion issue can be avoided.

In this paper, we solve the congestion issue differently. Essentially, we show that even when there are huge congestions, $\Omega(1)$ fraction of the executions can still proceed. To show this, we prove the following (see 'Lemma' 2.6 for detail). We have up to $\Omega(n)$ executions of the local flow algorithm of [21] running in parallel. Consider two augmenting paths p_1 and p_2 from two executions with sources s_1 and s_2. If p_1 and p_2 meet at some vertex t, then there is a path p either from s_1 to s_2 or from s_2 to s_1 that uses only edges explored by the two executions so far such that *p can be used as an augmenting path by one of the two executions.*[6] In other words, if the augmenting paths from two executions meet at the same vertex, then one of them can augment to another one.

This argument can be extended to show that if many augmenting paths meet at a vertex, then they can stop and only use what they have explored to finish the augmentation for half of them. This property helps reduce congestion when finding augmenting paths from different vertices.

To conclude, the above property allows us to find a vertex cut of size κ in $\tilde{O}(\kappa^3\alpha)$ where $\alpha := |C|$, the number of vertices in one of the connected components in the cut (see Lemma 2.1 for detail). Finally, note that given the prevalence of local flow algorithms in designing efficient graph algorithms, similar issues to the above may arise for other problems, and it is interesting to see if our technique can be applied elsewhere.

Finding Balanced Cuts: Specialized Fast Reachability Algorithm ('Lemma' 2.7). Before discussing this case, note that the above algorithm with round complexity $\tilde{O}(\kappa^3\alpha)$ already lends itself to a sublinear time algorithm for vertex cut with $\kappa = O(1)$—one can use this algorithm for small α, and Ford-Fulkerson and reachability algorithm when α is large. In order to improve the round complexity to $\tilde{O}(D + \sqrt{n})$ even when $\kappa = O(1)$, there is another fundamental barrier: the need to solve the *distributed reachability* problem.

For concreteness, assume that removing $\kappa = O(1)$ vertices leaves us with two connected components A and B each of $\Omega(n)$ vertices. This case cannot be solved efficiently by the local flow algorithm since $\alpha = \Omega(n)$. In the sequential setting, this case can be easily solved by sampling two vertices s and t and computing a (s,t)-maxflow of size $\Theta(\kappa)$ in a graph. To do this, simply find augmenting paths for $\Theta(\kappa)$ rounds (i.e. the Ford-Fulkerson algorithm). This takes $O(m\kappa)$ time and succeeds with constant probability (since $Pr[s \in A \text{ and } t \in B] = \Omega(1)$). In the CONGEST model, however, even answering a simpler question of whether there is *one* augmenting path from s to t (i.e., solving the (s,t)-*reachability*) requires larger than $\tilde{O}(D + \sqrt{n})$ rounds: The best distributed algorithms for reachability require $\tilde{O}(D + \sqrt{n}D^{1/4})$ rounds [33] and $\widetilde{O}(\sqrt{n} + n^{1/3+o(n)} \cdot D^{2/3})$ rounds [51] .

In this paper, we develop an algorithm specialized for our case: when we want to find an augmenting path, we are solving a reachability problem where most edges in the graph are *undirected.* A result implied by our technique when $\alpha = \Omega(n)$ is as follows. (See 'Lemma' 2.7 for the full statement.)

'Lemma' 1.2. *There exists a randomized CONGEST algorithm that, given two vertices $s,t \in V$ and a set of ℓ internally vertex-disjoint (s,t)-paths P, either returns an augmenting path or declares that such path does not exist. The algorithm takes $\ell^2 \cdot \widetilde{O}(D + \sqrt{n})$ rounds.*

So, to find κ internally vertex-disjoint (s,t)-paths, we use the above algorithm κ times, taking $\kappa^3 \cdot \widetilde{O}(D+\sqrt{n})$ rounds in total. This partially explains the round complexity of our final algorithm.

The main technique for proving the above 'lemma' is to modify the framework in the reachability algorithms [33, 51, 55]: As usual, we sample hubs and grow BFS trees from each hub and build a virtual graph on the hubs. Our novelty is to use a clustering technique to create a small number of *strongly connected components* (or clusters) and give them some ordering with the following guarantee: Any vertex in a cluster can reach any vertex in another cluster which is ordered lower than the former cluster. This clustering lets us reduce the number of vertices and edges in the virtual graph without affecting reachability as well as makes it possible to broadcast the whole virtual graph. See Section 2.2 for an overview of this algorithm.

1.3 Open Problems

This paper presents a study on the computational complexity of the vertex connectivity problem for small κ in the CONGEST model. There are several avenues for future research that may further improve upon the findings presented in this study.

Vertex connectivity in CONGEST model.

- (Small κ) for small values of κ, it would be interesting to investigate whether it is possible to surpass the $O(D+\sqrt{n})$ running time with an algorithm given that there is no $\Omega(D+\sqrt{n})$ lower bound for unweighted vertex connectivity. Although algorithms have been developed that run in $\tilde{O}(D)$ rounds for $\kappa = 1, 2$, the true complexity for larger κ remains unknown.
- (Large κ) the current best algorithms for the general vertex connectivity problem in the CONGEST model do not

[6]Here, we also exploit the fact that a source of one execution can be a sink for other executions.

have sub-linear time complexity when κ is as large as $\Theta(n)$. It would be interesting to explore the development of sub-linear algorithms for cases where κ is large.

- (Universally optimal) In recent years, there have been many papers seeking universally optimal algorithms, starting from the work by Haeupler, Wajc and Zuzic [34]. Since our algorithm meet the $\tilde{O}(D + \sqrt{n})$ upper bound for $\kappa = \text{polylog}(n)$, it would be interesting to explore the development of an algorithm that is universally optimal.

Parallel vertex connectivity. By combining the current best sequential algorithm for small κ with the current best parallel algorithm for reachability with depth $n^{1/2}$, it is possible to develop an almost linear work parallel algorithm with depth $n^{3/4}$. It would be interesting to investigate whether it is possible to further reduce the depth of the algorithm to the best reachability algorithm depth of $n^{1/2}$ or better. As this paper provides an example of surpassing the reachability running time for small κ in the CONGEST model, it is reasonable to expect that similar improvements may be possible in the parallel model as well.

Other models of computation. In addition to advancements in the CONGEST and parallel models of computation, we would like to see further advancements in cut-query and two-party communication models, both in classical and quantum settings, for the problem of vertex connectivity (and minimum vertex cut). Notably, the edge connectivity (and minimum edge cut) has nearly been resolved within the classical setting [45, 52, 64] and considerable progress has been achieved within the quantum setting [1, 46]. However, no substantial progress is made for vertex connectivity.

Other graph cut problems. Ultimately, significant advancements have yet to be made in addressing alternative variants of graph cut problems, including directed edge connectivity and minimum weighted vertex cut, in CONGEST and other distributed models of computation. Consequently, any headway achieved in these domains within any of the distributed models of computation would be of considerable interest.

2 OVERVIEW

In this section, we sketch the proof of our main result, i.e. Theorem 1.1. For notations, we use the following: for $S \subseteq V$, $\mathsf{N}(S) = \{v \mid \exists(u,v) \in E, u \in S, v \notin S\}$ denotes the neighbors of S in graph $G = (V, E)$, and $\mathsf{N}^+(S) = \mathsf{N}(S) \cup S$.

The crux of our algorithm is the subroutines called IsolatingSmallCut and SingleSourceLocalCut, which give guarantees as in Lemmas 2.1 and 2.2 below. We sketch the proofs of Lemmas 2.1 and 2.2 in Section 2.1 and Section 2.2 respectively. Then, in Section 2.3, we show how to combine them together by following the framework of [21].

We also denote the vertex cut of the graph G by (L, S, R), where $|L| \le |R|$ are the two sides of the cut, and S is the set of vertices whose removal disconnects L from R. Lemma 2.1 roughly guarantees that if we have a set of vertices $A \subseteq V$ such that, for some κ-cut (L, S, R), exactly one vertex in A is in $\mathsf{N}^+(L)$ (i.e. L and its neighbors), then we will be able to find such a cut or a similar cut in $\tilde{O}(\kappa^3|L|)$ rounds. So, to find a small vertex cut (L, S, R) when $|L|$ is small (the "unbalanced case" mentioned earlier), this algorithm will be fast assuming that we can find such an A. For intuition, note the following related sequential algorithms. **(i)** In [48], the same statement to ours is proved in the sequential setting with an algorithm that takes max-flow time (which is currently almost linear [8]). This is done via the *isolating cut technique* [49], thus the word "Isolating" in the name of our algorithm. Unfortunately, we cannot use the same technique since we do not have an efficient exact max-flow algorithm in the distributed setting. **(ii)** In [21], a similar statement can be guaranteed in $O(m\kappa^2)$ time in the sequential setting. Compared to our requirement that $|A \cap \mathsf{N}^+(L)| = 1$, the statement of [21] requires a weaker condition that $|A \cap L| \ge 1$ (A that satisfies this condition can be easily found, e.g. $A = V$). As we will show in Section 2.1, our algorithm follows the idea of [21], but our stricter condition gives us some leverage to avoid the congestion issue that we would face if we simply followed the ideas of [21] (discussed in the previous section).

Lemma 2.1 (IsolatingSmallCut($G = (V, E), A \subseteq V, \kappa, \alpha$)). *There exists a CONGEST algorithm that given an undirected graph $G = (V, E)$, a set of vertices $A \subseteq V$, and $\kappa, \alpha \in \mathbb{N}$,*[7] *either outputs a valid κ-cut*[8] *(L, S, R) with one side L such that $|A \cap \mathsf{N}^+(L)| = 1$, or outputs $\perp$. The output satisfies*

- *if there exists a vertex set $L \subseteq V$ such that $|\mathsf{N}(L)| < \kappa$, $|A \cap \mathsf{N}^+(L)| = 1$, and $|L| \le \alpha$, then the algorithm outputs $\perp$ with at most constant probability*[9]*, and*
- *the algorithm runs in $\tilde{O}(\kappa^3\alpha)$ rounds.*

Lemma 2.2 roughly guarantees that if we know two vertices s and t that are on the opposite sides of a κ-cut, i.e. for some κ-cut (L, S, R) we have $s \in L$ and $t \in R$, then we can find a κ-cut efficiently; here, "efficiently" means the dilation of $\tilde{O}(\kappa^{2.5}\sqrt{n} + \kappa^3 D)$ and congestion of $\tilde{O}(\kappa^{2.5}|L|/\sqrt{n})$. We need the congestion to be $\tilde{O}(\kappa^{2.5}|L|/\sqrt{n})$ so that we can run $O(n/|L|)$ algorithms with different s, t simultaneously, while still keeping the running time $\tilde{O}(\kappa^{2.5}\sqrt{n} + \kappa^3 D)$. It is necessary to run $\Theta(n/|L|)$ algorithms since we need to sample $\Theta(n/|L|)$ vertices to guarantee at least one vertex is inside L. Each algorithm will take one sampled vertex as s.

Note that a similar statement was achieved in the sequential setting in $O(m\kappa)$ time, which is the time to compute a max-flow of size κ using Ford-Fulkerson algorithm. As discussed earlier, computing a max-flow will not allow us to beat the time to solve reachability. For this reason, we need some clustering ideas which we show in Section 2.2.

Lemma 2.2 (SingleSourceLocalCut($G = (V, E), s, t, \kappa, \alpha$)). *There exists a CONGEST algorithm that given an undirected graph $G = (V, E)$, two vertices $s, t \in V$ and $\kappa, \alpha \in \mathbb{N}$, where $\kappa \le \alpha$, either outputs a valid κ-cut, or outputs $\perp$, such that*

- *if there exists $L \subseteq V$ such that $|\mathsf{N}(L)| < \kappa$, $\{s, t\} \cap \mathsf{N}^+(L) = \{s\}$, $|L| \le \alpha$, then the algorithm outputs $\perp$ with constant probability,*
- *the algorithm has dilation $\tilde{O}(\kappa^{2.5}\sqrt{n} + \kappa^3 D)$ and congestion $\tilde{O}(\kappa^{2.5}\alpha/\sqrt{n})$.*

[7]Every vertex knows of their membership in A and κ, α.
[8]Every vertex knows of their membership in S.
[9]When we say "with constant probability" in this paper, we mean a constant less than 1.

Remark 2.3. Throughout this paper, it is important for the reader to keep in mind that our algorithm is a Monte Carlo algorithm with one-sided error. Specifically, when the output is a cut, it must be a valid cut with a size less than κ. However, when the output is $\perp$, it is possible that the graph has a cut with a size less than κ, and the algorithm **cannot** distinguish whether the output is correct or not. Nevertheless, since the algorithm has one-sided error, as long as the error probability is bounded by a constant between 0 and 1, it can be reduced to as small as $\frac{1}{n^c}$ by repeating the algorithm $O(\log n)$ times.

2.1 IsolatingSmallCut (Proof Sketch of Lemma 2.1)

The starting point is to run the algorithm of [21] for every vertex in A simultaneously, i.e, run κ rounds of DFS to find *augmenting path* on residual graphs, defined below. For a path $p = (v_0, v_1, v_2, ..., v_\ell)$, we define $pre_p(v_i) = v_{i-1}$ for any $0 < i \leq \ell$ and $suc_p(v_i) = v_{i+1}$ for any $0 \leq i < \ell$. $v_1, v_2, ..., v_{\ell-1}$ are called the *internal vertices* of p. A set of paths are called *internally vertex disjoint* if any two of them do not share the same internal vertex. We define $V(p)$ as the vertex set consisting of all vertices in p. For a set of paths P, we define $V(P) = \cup_{p\in P} V(p)$.

Definition 2.4 ((G, s, P)-Augmenting Path). *Let $G = (V, E)$ be an undirected graph, $s \in V$ and P is a set of k internally vertex disjoint paths starting from s. (We call P a* ***flow-path set*** *of s.) A path p_{aug} in G is called (G, s, P)-augmenting if*

(i) ***Starting vertex:*** *p_{aug} starts at s and,*
(ii) ***Forced retreat:*** *for any consecutive vertices u_1, u_2 in p_{aug} where u_2 is not the end of p_{aug} and any $p \in P$, if $u_2 \in V(p) \setminus \{s\}$ and $u_1 \neq suc_p(u_2)$, then $suc_{p_{aug}}(u_2) = pre_p(u_2)$.*

Figure 1 provides an example of such an (G, s, P)-augmenting path. Intuitively speaking, if an augmenting path enters a vertex in path $p \in P$ that is not from its successor, then it is forced to go backward (or *retreat*).

For a minimum vertex cut (L, S, R), our goal is to find the maximum number of vertex disjoint paths from $s \in L$ to R (from which we can infer the vertex cut), and we use augmenting paths to this end as follows. 'Lemma' 2.5 shows (i) if an augmenting path ending at R can be found, then we can increase the number of vertex disjoint path, (2) if no augmenting path ending at R can be found, then we can find a vertex cut.

'Lemma' 2.5. *Suppose $G = (V, E)$ is an undirected graph, if P is a set of k internally vertex disjoint paths starting from $s \in V$, ending at a vertex set T, then*

(i) (Augmentation.) Suppose p is a (G, s, P)-augmenting path, ending at t, then there exists a set of $k + 1$ internally vertex disjoint paths P' ending at $T \cup \{t\}$. See Figure 1 as an example.
(ii) (Find a cut.) Let S' contain all the nodes that s can reach through a (G, s, P)-augmenting path. If $S' \neq V$, then the following nodes form a vertex cut: for any $p \in P$, the node in $S' \cap V(p)$ that has the largest distance to s on p. See Figure 2 as an example.

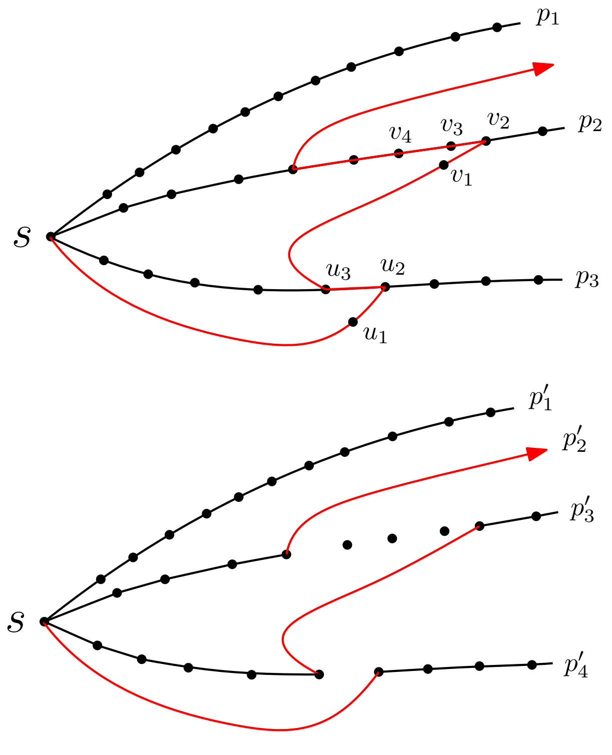

Figure 1: The left figure is an example of Definition 2.4 with $P = \{p_1, p_2, p_3\}$. The red line starts at s. The consecutive three vertices (u_1, u_2, u_3) satisfy $u_2 \in V(p_3), u_1 \neq suc_{p_3}(u_2), u_2 \neq s$, so $u_3 = pre_{p_3}(u_2)$. The same holds for (v_1, v_2, v_3). Thus, the red line is a (G, s, P)-augmenting path. The right figure is the resulting $P' = \{p'_1, p'_2, p'_3, p'_4\}$ according to 'Lemma' 2.5.

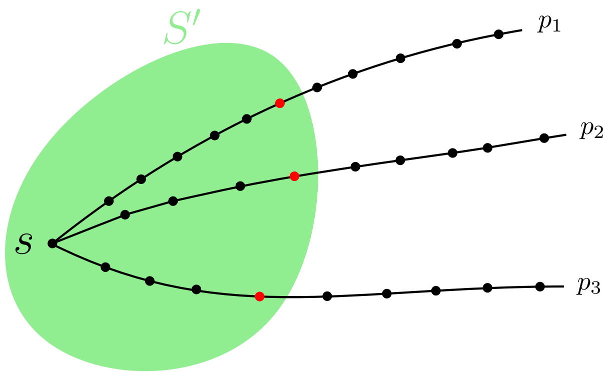

Figure 2: S' contains all the nodes that s can reach through a (G, s, P)-augmenting path. The set of red vertices are vertices in S' farthest from s in each of their paths and form a vertex cut.

It is not hard to see that, in the *Augmentation* case above, the minimum vertex cut separating s and $T \cup \{t\}$ has a size at least $k+1$ if s does not have an edge to $T \cup \{t\}$—this follows from Menger's theorem.

Our algorithm IsolatingSmallCut for Lemma 2.1 works as follows. Initially, each node $s \in A$ has an empty flow-path set P_s. We run κ iterations where, in each iteration, we increase the size of P_s by 1 for each vertex $s \in A$: In each iteration, very informally, each vertex s sends a DFS token to explore G in a DFS manner for $\Theta(\kappa\alpha)$ rounds in order to find a (G, s, P_s)-augmenting path. If the DFS gets *stuck* (This is explained shortly.), then we use 'Lemma' 2.5 to find a cut. Indeed, our main challenge is to reduce congestion caused by all of these DFS traversals running in parallel. To this end, we exploit the following property of augmenting paths which is the main technical lemma of this subsection. We start with some definitions which provide the necessary context.

A (G, s, P)-augmenting path $p_{aug} = (s, v_1, ..., v_{\ell-1}, v_\ell)$ is called *retreating* if there exists $p \in P$, such that $v_\ell \in V(p)\backslash\{s\}, v_{\ell-1} \neq suc_p(v_\ell)$, i.e., the only way to extend p_{aug} to a (G, s, P)-augmenting path $(s, v_1, ..., v_\ell, v_{\ell+1})$ is to set $v_{\ell+1} = pre_p(v_\ell)$. For example, in Figure 1, the red (G, s, P)-augmenting path from s to u_2 is retreating. A (G, s, P)-augmenting path is called *non-retreating* if it is not *retreating*.

'Lemma' 2.6. *For any undirected graph $G = (V, E)$, consider two vertices $u, v \in V$, and let P_u and P_v be flow-path sets of u and v, respectively. Let p_u and p_v be non-retreating (G, u, P_u)- and (G, v, P_v)-augmenting paths, respectively. If p_u and p_v end at the same vertex, then there exists a path p on the subgraph of G resulting from combining all edges of P_u, P_v, p_u, p_v such that p is either (G, u, P_u)-augmenting ending at v, or (G, v, P_v)-augmenting ending at u.*

With 'Lemma' 2.6, the algorithm becomes the following. Denote the *flow-path set* of $s \in A$ as P_s. Initially $P_s = \emptyset$. Run the following procedure for κ iterations: In each iteration, we make sure that the size of P_s increases by 1 for all $s \in A$.

(i) **Whole-graph DFS:** In parallel, every vertex $s \in A$ sends a token (denoted by the s-token) to explore new vertices in G in a DFS manner: Each vertex u (including s), once receiving the token, finds out which of its neighbors is not explored yet by the s-token, and sends the s-token to one such unexplored neighbor. The DFS follows the forced retreat property described in Definition 2.4, i.e., when an s-token arrives at a vertex u on a flow-path $p \in P_s$ not from $suc_p(u)$, then the token must be sent to $pre_p(u)$.The DFS traversal ends in either of the following three ways:
   - If s explores $\Theta(\kappa\alpha)$ vertices or s reaches another vertex $t \in A$, it stops.
   - If two tokens from $u, v \in A$ meet at a vertex t, then they stop, form a pair (u, v), and report this fact back to u and v through DFS trees. Denote the path from u and v to t in the DFS trees by p_u and p_v respectively. Define subgraph $H_{(u,v)}$ as the subgraph formed by the union of edges in p_u, p_v, P_u, P_v. This graph will be used in the next step.
     If many tokens $u_1, u_2, \ldots, u_\ell$ meet at t, we pair them up $(u_1, u_2), \ldots$ to get subgraphs $H_{(u_1,u_2)}, \ldots$. In the case where ℓ is odd, u_ℓ is allowed to continue its DFS t onward.
   - If s finishes DFS (i.e., has explored all vertices it can reach) without exploring $\Theta(\kappa\alpha)$ vertices and without reaching another vertex $t \in A$, output the small cut using 'Lemma' 2.5 (ii)(If several vertices finish DFS, we just need to pick an arbitrary one.)

   Let (L, S, R) be the vertex cut as claimed in Lemma 2.1, i.e., $|S| < \kappa$, $A \cap (L \cup S) = \{s\}$ and $|L| \leq \alpha$. Note that s succeeds in finding a (G, s, P_s)-augmenting path that terminates in R in the first case with a constant probability: (i) If s explores $\Omega(\kappa\alpha)$ vertices, then a random vertex among the explored vertices is in R with probability at least $1 - \frac{1}{\Omega(\kappa)}$. So we can choose this random vertex as the terminating vertex of the augmenting path[10]. (ii) If s reaches $t \in A$ that $t \neq s$, then t is the terminating vertex and $t \in R$.
   Once the DFS traversals stop for every $s \in A$, we move to the next step.

(ii) **Subgraphs DFS:** For each pair (u, v), u and v run DFS traversal on $H_{(u,v)}$. These DFS traversals in all $H_{(u,v)}$'s are run simultaneously using the random delay technique [27] to avoid congestion[11]. If u find a (G, u, P_u)-augmenting path p to v, it uses p to increase the size of P_u by 1. Do the same for v. 'Lemma' 2.6 guarantees that one of u and v will succeed in finding an augmenting path.

Note that executing Step (i) and (ii) will increase $|P_s|$ for a constant fraction of $s \in A$ by 'Lemma' 2.6. We repeat these two steps $O(\log n)$ times to make sure $|P_s|$ increases for every $s \in A$.

Round complexity. We first bound the round complexity for the two steps. One can see that Step (i) runs in $O(\kappa\alpha)$ rounds. The round complexity of Step (ii) depends on the dilation (i.e., the diameter of subgraph $H_{(u,v)}$) and congestion (i.e., the maximum number of $H_{(u,v)}$ for different pairs (u, v) that shares the same edge) which we bound below. We crucially use the following fact: A (G, s, P)-augmenting path p of length ℓ w.r.t. a flow-path set P can increase the number of path edges in the new flow-path set by at most an additive factor of ℓ. [12]

Dilation. Note that each p_s^i, $i \in [\kappa]$, is of size $O(\kappa\alpha)$. From the fact stated above, it is straightforward to bound the size of $H_{(u,v)}$ (which is composed of p_u, p_v, P_u, P_v) by $\widetilde{O}(\kappa^2\alpha)$.

Congestion. The number of $H_{(u,v)}$ that contain an edge e is bounded by the number of times e is visited by DFS traversals in Step (i), as e can be included in some $H_{(u,v)}$ only after it is visited in any DFS traversal in Step (i) by u or v. Every edge e is included in at most one DFS traversal in each round of Step (i). Since Step (i) lasts for $\widetilde{O}(\kappa\alpha)$ rounds in each of the κ iterations, an upper bound on the number of times e is visited by DFS traversals in Step (i) is $\widetilde{O}(\kappa^2\alpha)$.

The total round complexity is $\kappa \times O(\log n) \times \widetilde{O}(\kappa^2\alpha) = \tilde{O}(\kappa^3\alpha)$: The first κ is the number of iterations, $O(\log n)$ is the number of times Step (i) and (ii) are repeated in each iteration.

[10]A-priori we do not know if our chosen vertex is in R or not. However, we show that, if the algorithm outputs a valid vertex cut in the end, it will be a cut of size at most κ. See Remark 2.3.

[11]According to [27], running independent CONGEST algorithms simultaneously can be done using random delay in $\widetilde{O}$(dilation + congestion) rounds.

[12]This observation follows directly from the following fact which is easy to see. Suppose $s \in A$ has flow-path set P_s^i at the end of each iteration i (We assume $P_s^0 = \emptyset$), and consider the (G, s, P_s^i)-augmenting paths $p_s^1, p_s^2, ..., p_s^i$ that are used to generate different P_s^i: Each p_s^i is a (G, s, P_s^{i-1})-augmenting path. We claim that the edges in P_s^i is a subset of edges in $p_s^1, p_s^2, ..., p_s^i$. Note that it might not be true that the set of edges in P_s^{i-1} is a subset of the set of edges in P_s^i.

2.2 SingleSourceLocalCut (Proof Sketch of Lemma 2.2)

For intuition, note that a statement similar to Lemma 2.2 can be shown in the sequential setting [21, 58] by running the Ford-Fulkerson algorithm. This algorithm runs for κ iterations where in each iteration it increases the amount of st-flow by one via an augmenting path. We follow this basic idea but need some modifications. First, in each of the k iterations, we randomly select some *terminals*, where each vertex has probability $O(1/(\kappa\alpha))$ to be the terminal. We allow the augmenting path to end at a terminal instead of at t. This suffices because if there exists a vertex cut (L, S, R) such that $\{s,t\} \cap (L \cup S) = \{s\}, |L \cup S| \leq \alpha$ (thus L satisfies the condition in the first bullet of Lemma 2.2), a simple union bound shows that the random terminals on all κ rounds are in R with constant probability. The algorithm for finding the augmenting path is stated as the following lemma. We will use this algorithm with $x = \kappa\alpha$. Recall from Definition 2.4 the notion of *flow-paths* and (G, s, P)-augmenting path.

'Lemma' 2.7 (RandomAugmenting$(G = (V, E), s, P, x)$). *There exists a CONGEST algorithm called RandomAugmenting that takes an undirected graph $G = (V, E)$, two vertices $s, t \in V$, integer x and a set P of flow-paths of s where each path in P has length bounded by $O(x)$, as input and the algorithm either*

- *outputs a vertex cut of size $|P|$, or*
- *outputs a (G, s, P)-augmenting path with length bounded by $\widetilde{O}(x)$, either ending at t, or ending at a random vertex $\tilde{t}$, where* $\Pr[\tilde{t} = v] = O(1/x)$ *for any $v \in V$.*

The algorithm has dilation $\widetilde{O}(|P|^{1.5}\sqrt{n} + |P|^2 D)$ and congestion $\widetilde{O}(|P|^{0.5}x/\sqrt{n})$.

To prove Lemma 2.2 using 'Lemma' 2.7, our algorithm starts with $P = \emptyset$. It proceeds in κ iterations, where in each iteration we find a (G, s, P)-augmenting path using 'Lemma' 2.7 with $x = \Theta(\kappa\alpha)$ to increase the size of P by 1. Since $|P| < \kappa$, one can see that the dilation is $\kappa \cdot \widetilde{O}(|P|^{1.5}\sqrt{n} + |P|^2 D) = \widetilde{O}(\kappa^{2.5}\sqrt{n} + \kappa^3 D)$ and the congestion is $\kappa \cdot \widetilde{O}(|P|^{0.5}x/\sqrt{n}) = \widetilde{O}(\kappa^{2.5}\alpha/\sqrt{n})$, which is what we want in Lemma 2.2. The rest of this section is devoted to showing the proof idea of 'Lemma' 2.7.

Proof idea of 'Lemma' 2.7. We first review the framework for distributed reachability algorithms used in [33, 51, 55]. (We will modify this framework to find a (G, s, P)-augmenting path as guaranteed in 'Lemma' 2.7.) This framework consists of two phases, where the first phase is identical in all algorithms in [33, 51, 55], and these algorithms differ in the second phase. Suppose we want to find a path from s to t. The two phases are:

(i) **Build a virtual graph.** Pick appropriate parameter d (we will pick $d = |P|^{1.5}\sqrt{n}$ to prove 'Lemma' 2.7). Construct a virtual[13] graph $G_{vir} = (V_{vir}, E_{vir})$ where V_{vir} (also called set of *hubs*) includes every vertex of V with probability $1/d$ as well as s, and an edge $e = (h_1, h_2)$ is included in E_{vir} if the distance from h_1 to h_2 in G is at most d. E_{vir} can be constructed by constructing a BFS tree T_h of depth d from each vertex $h \in V_{vir}$ in G.

[13]By "virtual" it means that edges in the virtual graph might not be edges in the input network.

(ii) **Reachability in the virtual graph.** Find all the hubs that s can reach in G_{vir}, denoted by H_r (the way to efficiently find H_r differs by different algorithms). Now we claim that $\cup_{h \in H_r} T_h$ are all the vertices s can reach in the original graph G.

The correctness is guaranteed by the following arguments: since we sample hubs with probability $\frac{1}{d}$, the path from s to a vertex v contains hubs with distance $\widetilde{O}(d)$ one after another along the path, with high probability. Therefore, the hubs in the path form a directed path in the virtual graph, where the last hub in the path has distance d to v in G.

Using reachability algorithm to find an augmenting path. Our definition for augmenting path in Definition 2.4 can be reformulated as a directed path in a directed graph, by the standard way of duplicating each vertex into in-vertex and out-vertex. Thus, we can use a directed graph reachability algorithm to find an augmenting path.

However, directly applying this framework to prove 'Lemma' 2.7 is not efficient as there can be $\Omega(n/d)$ BFS tree constructions that can lead to dilation $\Omega(d)$ and congestion $\Omega(n/d)$. Recall that in 'Lemma' 2.7 we want dilation $\widetilde{O}(|P|^{1.5}\sqrt{n} + |P|^2 D)$ and congestion $\widetilde{O}(|P|^{0.5}x/\sqrt{n})$, where x can be much smaller than n. There is no way to set appropriate d to satisfy both the dilation and congestion. To achieve a better dilation and congestion trade-off, we will only grow a BFS tree on fewer carefully chosen hubs instead of all $\Omega(n/d)$ hubs.

Path centered clustering. The key idea to reduce the number of BFS tree constructions is a structure called *path-centered clustering*. Here we give a simplified version of the structure. Note that the following definition is different from the full version of the paper due to the page limit. However, it shows the general idea of the more complicated definition, so we use it for ease of explanation. For a given network $G = (V, E)$ of diameter D, a *path centered clustering* is a tuple $C = (P, \{S_u\}_{u \in V(P)})$ where P is a flow-path set, and $\{S_u\}_{u \in V(P)}$ is a partition of V (i.e. V is a disjoint union of all S_u's), called *clusters* with the following guarantees: Each cluster S_u contains $u \in V(P)$, and each induced subgraph $G[S_u]$ has a diameter at most D. We call u the *center* of every vertex $v \in S_u$ and denote it by $\mathsf{Center}_C(v)$. See Figure 3 for an example.

Definition of Before *and active hubs.* We need a few definitions to show the properties of path centered clustering. For a path $p = (v_0, v_1, ..., v_\ell)$ and v_i, v_j on the path, we say $v_i \preceq_p v_j$ if $i \leq j$ and we say $v_i \prec v_j$ on path p if $i < j$. For any two vertices $h, h' \in V$ and a path centered clustering $C = (P, \{S_u\}_{u \in V(P)})$, we say $h' \preceq_C h$, if $\mathsf{Center}_C(h')$ and $\mathsf{Center}_C(h)$ belong to some path $p \in P$ and $\mathsf{Center}_C(h') \preceq_p \mathsf{Center}_C(h)$. The relationship $\preceq_C$ is not total as not every two vertices in G are comparable by $\preceq_C$. For each hub h (recall that hubs are sampled vertices in G with sample probability $\frac{1}{d}$), we use $\mathsf{Before}_C[h]$ to denote the number of hubs h' with $h' \preceq_C h$. We will assume the following assumption.

Assumption 2.8. If $h' \preceq_C h$, then h can reach h' through an augmenting path.

Remark 2.9. It is to be noted that our actual clustering is more fine-grained than what is described above to tackle the following technical problem: Assumption 2.8 is true if $\mathsf{Center}_C(h') \prec_p \mathsf{Center}_C(h)$

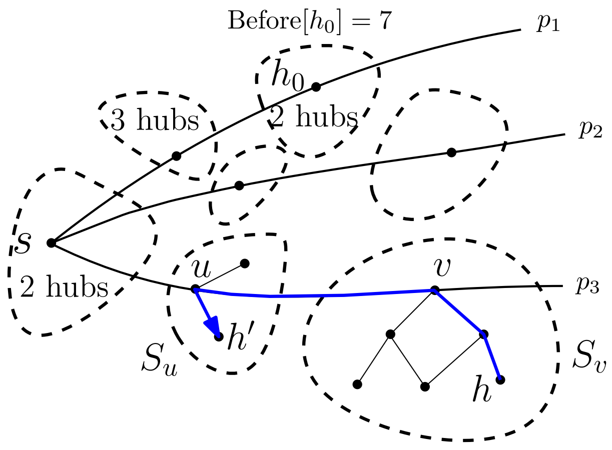

Figure 3: Each dashed circle is a cluster. For simplicity, we only draw the inside structure of cluster S_u and S_v. We can see that there are 7 hubs x with $x \preceq_C h_0$, so $\mathsf{Before}_C[h_0] = 7$. We also have $h' \preceq_C h$. The blue line shows how h can reach h' through an augmenting path.

(In Figure 3, the blue line shows an augmenting path from h to h'.) and may not be true if $\mathsf{Center}_C(h') = \mathsf{Center}_C(h)$. This is solved by making the clustering more fine-grained—more details are provided in the full version of this paper. In this section, we assume Assumption 2.8 holds for ease of explanation.

Build a virtual graph with fewer BFS tree constructions. In this part we will show how to build a virtual graph G_{vir} on hubs with $O(x/d) \cdot |P|$ BFS tree constructions, such that either

- G_{vir} preserves the s-reachability (in the sense that all the vertices reachability by s in G can be reached from a vertex u in G_{vir} with distance d, such that s can reach u in G_{vir}), or
- s can reach a random vertex $\tilde{t}$ such that each vertex in V becomes $\tilde{t}$ with probability $O(\frac{1}{x})$.

Now we give our algorithm. We first compute a path centered clustering C. We call a hub h *active hub* if $\mathsf{Before}_C[h] = O(x/d)$. Other hubs are called *non-active hubs*. Denote the set of all active hubs as V_{act}. One can argue that $|V_{act}| = O(x/d) \cdot |P|$. We only grow BFS trees on active hubs. By setting $d = |P|^{1.5}\sqrt{n}$, the dilation and congestion of constructing all the BFS trees satisfy the requirement in 'Lemma' 2.7. By doing that, we can get a virtual graph $G_{vir} = (V_{vir}, E_{vir})$ where E_{vir} includes an edge $e = (h_1, h_2)$ if $h_1 \in V_{act}$ and h_1 has distance at most d to h_2 in G.

Now we argue the property of G_{vir}. If in G_{vir}, s can reach a non-active hub h through active hubs, then we can pick a uniform random hub h' among all hubs $h' \preceq_C h$ as the destination. Notice that a non-active node h satisfies $\mathsf{Before}_C[h] = \Omega(n/d)$, thus, each node has probability at most $O(1/d) \cdot O(d/x) = O(1/x)$ to be the destination. On the other hand, if s cannot reach any non-active hub, then by growing BFS trees on all active hubs, we can find all vertices that s can reach in G exactly.

Find reachability in virtual graph. Let H_r contain all the active hubs that s can reach in the virtual graph G_{vir}. Our goal in this part is to find H_r efficiently. Notice that if we can find H_r, the according to the argument in the previous part, either we can find all vertices in G that s can reach, or find a non-active hub such that we can choose a random destination with probability $O(1/x)$.

We first discuss the difficulty. Notice that $|H_r| = O(|P| \cdot x/d) = O\left(x/(|P|^{0.5}\sqrt{n})\right)$. Possible values of $x, |P|$ are $x = \Theta(n)$ and $|P| = O(1)$. In this case, $|H_r| = O(\sqrt{n})$. All the existing algorithms fail to find reachability with round complexity $\tilde{O}(\sqrt{n} + D)$ on a virtual graph with $\sqrt{n}$ vertices. However, our virtual graph is not an arbitrary directed graph. We will exploit some properties of our virtual graph to come up with an efficient algorithm.

The idea is to sparsify the transitive closure of G_{vir} and broadcast the whole sparsified graph. We will make sure that the sparsified graph has the same reachability relationship as the original graph, and it is possible to broadcast the sparsified graph using $O(|P| \cdot |V_{act}|)$ messages. There are two types of edges in the sparsified graph.

Backward edges. These are edges (h, h') where $h' \preceq_C h$. To learn this type of edge, we give each flow-path $p \in P$ an id. Each vertex v on p can learn p's id and its position on p (the number of vertices x with $x \preceq_p v$) efficiently by existing results. After that, each active hub h broadcasts the flow-path id where $\mathsf{Center}_C[h]$ is on, as well as the position on the flow-path.

Forward edges. For each active hub h, recall that T_h is the directed tree with depth d rooted at h. Instead of keeping all edges from h to all hubs in T_h, we preserve the "highest hub" for each path $p \in P$: let T_h^p contain all hubs x in T_h with $\mathsf{Center}_C[x]$ on p. Let h_p^* be an arbitrary hub in T_h^p such that for every other hub $h' \in T_h^p$, we have $h' \preceq_C h_p^*$. (h, h_p^*) is added to the virtual graph for any $p \in P$.

One can see that the number of messages broadcast by every active hub is bounded by $|P|$. Thus, the congestion is $\widetilde{O}(|P|^{0.5}x/\sqrt{n})$, which fits our goal. To see that the reachability relationship does not change, suppose $h' \in T_h$ where $\mathsf{Center}_C[h']$ is on p, then h can reach h' in the virtual graph by first using the upward edge (h, h_p^*), then using the downward edge (h_p^*, h').

Remark 2.10. We skip the mapping of each edge in the virtual graph to a path in the original graph efficiently in the technical overview, see the full version of this paper for more details. Actually, to recover the path in the original graph efficiently, the sparsified virtual graph defined in the full version of this paper is different from here and more complicated, while the high-level ideas are the same.

2.3 Putting Everything Together

We first restate Theorem 1.1 formally.

Theorem 2.11. *There is a randomized vertex cut algorithm in the CONGEST model that, with input $\kappa < n^{1/4}$ and undirected graph G, takes $\kappa^3 \cdot \tilde{O}(D + \sqrt{n})$ rounds, either outputs a minimum vertex cut of G, or outputs $\perp$, satisfying*

(1) If the output is a vertex cut, then it must be a minimum vertex cut of G.
(2) If G is not κ-connected, then $\perp$ is output with at most constant probability.

Since Theorem 2.11 states a one-side error algorithm, the success probability can be boosted efficiently. The following is the schematic of the algorithm, using the subroutine described in Lemmas 2.1 and 2.2.

Schematic algorithm for vertex cut

- **Input:** An undirected graph G with n nodes, a positive integer $\kappa < n^{1/4}$.
- **Output:** A vertex cut with size less than κ, or $\perp$.

(1) If a vertex has degree less than κ in G, output all the neighbors of this vertex. Otherwise continue the following procedures.
(2) For $1 \le i \le \log n$ do:
  (a) Let $\alpha = 2^i, A = \emptyset$. Each vertex is included in A with probability $1/\alpha$ independently.
  (b) If $\alpha \le \kappa$,
    - discard vertices in A with degree larger than κ in $G[A]$, and run a $O(\kappa)$-coloring algorithm in $G[A]$ ([35]) to get $\ell = O(\kappa)$ independent sets $A_1, A_2, ..., A_\ell$ (see Lemma 2.12);
    - run IsolatingSmallCut(G, A_i, κ, α) (see Lemma 2.1) for any $i \in [\ell]$.
  (c) If $\kappa < \alpha < \sqrt{n}$, discard all vertices in A with degree at least 1 in $G[A]$, run IsolatingSmallCut(G, A, κ, α) (see Lemma 2.1).
  (d) If $\sqrt{n} \le \alpha$, for each $s \in A$, let $t_s \in A$ be an arbitrary vertex which is distinct from s. Run SingleSourceLocalCut$(G, s, t_s, \kappa, \alpha)$ for any $s \in A$ in parallel.
(3) If any subroutine described in Lemmas 2.1 and 2.2 outputs a cut, then the algorithm outputs the cut and stop. Otherwise, output $\perp$.

Correctness. According to Lemmas 2.1 and 2.2, if a cut is output, then it must be a valid vertex cut with size less than κ. Thus, if the graph G has no valid vertex cut with size less than κ, then the algorithm will output $\perp$ with probability 1.

Suppose there is a vertex cut (L, S, R) with $|S| < \kappa$. We assume the max degree of the graph is at least κ, otherwise, a vertex cut of size less than κ can be trivially found in the first step of the algorithm. We will show that in the second step, at the first iteration when $|L| < \alpha = O(|L|)$, a cut with a size less than κ will be output with constant probability.

Case 1 ($\kappa \ge \alpha$): In this case, we get ℓ independent sets $A_1, A_2, ..., A_\ell$. We first prove the following lemma.

Lemma 2.12. *At least one of $A_1, A_2, ..., A_\ell$ (denoted by A^*) satisfies: A^* is an independent set on G, contains exactly one vertex in L, and $A^* \cap S = \emptyset$.*

Proof. Since $|L| = \Theta(\alpha)$ and we sample each vertex into A with probability $1/\alpha$, with constant probability there is exactly one vertex $u \in A \cap L$. Let $\mathsf{N}^+(u)$ contain all neighbors of u in G and u itself. Since the degree of u is at least κ and $|S| < \kappa$, we have $(L \cup S) - \mathsf{N}^+(u)$ has size at most $|L| + \kappa - \kappa = |L| = O(\alpha)$. Thus, with constant probability, $(L \cup S) - \mathsf{N}^+(u)$ contains no vertex in A. Consider the independent set A^* among $A_1, ..., A_\ell$ that contain u. We have $A^* \cap (L \cup S) = \{u\}$, which finishes the proof. □

According to Lemma 2.1, once Lemma 2.12 is proved, a cut with size less than α will be output with constant probability when IsolatingSmallCut(G, A^*, κ, α) is called.

Case 2 ($\kappa < \alpha < \sqrt{n}$): Since we sample each vertex into A with probability $1/\alpha$ and $|L| = \Theta(\alpha), S = O(\kappa) = O(\alpha)$, with constant probability, exactly one vertex is in $A \cap L$ and $A \cap S = \emptyset$. According to Lemma 2.1, a cut with size less than α will be output with constant probability.

Case 3 ($\alpha \ge \sqrt{n}$): According to the same argument, with constant probability, exactly one vertex u is in $A \cap L$ and $A \cap S = \emptyset$. Consider the instance with $s \leftarrow u$, that instance satisfies the premise of Lemma 2.2 to output a cut with size less than κ.

Round complexity. When $\alpha < \kappa$, the round complexity for the coloring algorithm is $\widetilde{O}(1)$. There are $\widetilde{O}(\kappa)$ instances of IsolatingSmallCut in Lemma 2.1, which leads to the round complexity $\widetilde{O}(\kappa^4\alpha) = \widetilde{O}(\kappa^5) = \widetilde{O}(\kappa^3\sqrt{n})$ since $\kappa = O(n^{1/4})$. When $\kappa \le \alpha < \sqrt{n}$, the round complexity is $\widetilde{O}(\kappa^3\alpha) = \widetilde{O}(\kappa^3\sqrt{n})$. When $\sqrt{n} \le \alpha$, the dilation is $\widetilde{O}(\kappa^{2.5}\sqrt{n} + \kappa^3 D)$ and the total congestion is $\widetilde{O}(n/\alpha) \cdot \widetilde{O}(\kappa^{2.5}\alpha/\sqrt{n})$, since there are $\widetilde{O}(n/\alpha)$ vertices in A w.h.p. Thus, the round complexity is $\kappa^3 \cdot \widetilde{O}(\sqrt{n} + D)$.

ACKNOWLEDGMENT

We would like to thank Danupon Nanongkai for numerous fruitful discussions through out the project, and the reviewers for their meticulous reading and comments.

REFERENCES

[1] Simon Apers, Yuval Efron, Pawel Gawrychowski, Troy Lee, Sagnik Mukhopadhyay, and Danupon Nanongkai. 2022. Cut Query Algorithms with Star Contraction. In *FOCS*. IEEE, 507–518.
[2] M. Becker, W. Degenhardt, J. Doenhardt, S. Hertel, G. Kaninke, W. Keber, K. Mehlhorn, S. Näher, H. Rohnert, and T. Winter. 1982. A probabilistic algorithm for vertex connectivity of graphs. *Inform. Process. Lett.* 15, 3 (1982), 135–136. https://doi.org/10.1016/0020-0190(82)90046-1
[3] Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. 2014. Distributed connectivity decomposition. In *PODC*. ACM, 156–165.
[4] Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. 2014. Distributed connectivity decomposition. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*. 156–165.
[5] Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. 2014. A new perspective on vertex connectivity. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 546–561.
[6] Yi-Jun Chang, Seth Pettie, and Hengjie Zhang. 2019. Distributed Triangle Detection via Expander Decomposition. In *SODA*. SIAM, 821–840.
[7] Yi-Jun Chang and Thatchaphol Saranurak. 2019. Improved Distributed Expander Decomposition and Nearly Optimal Triangle Enumeration. In *PODC*. ACM, 66–73.
[8] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2022. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In *FOCS*. IEEE, 612–623.
[9] Joseph Cheriyan and Ramakrishna Thurimella. 1991. Algorithms for Parallel k-Vertex Connectivity and Sparse Certificates (Extended Abstract). In *STOC*. ACM, 391–401.
[10] Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. 2020. A Deterministic Algorithm for Balanced Cut with Applications to Dynamic Connectivity, Flows, and Beyond. In *FOCS*. IEEE, 1158–1167.
[11] Mohit Daga, Monika Henzinger, Danupon Nanongkai, and Thatchaphol Saranurak. 2019. Distributed edge connectivity in sublinear time. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*. 343–354.
[12] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. 2012. Distributed

Verification and Hardness of Distributed Approximation. *SIAM J. Comput.* 41, 5 (2012), 1235–1265. Announced at STOC'11.

[13] Michal Dory. 2018. Distributed Approximation of Minimum k-edge-connected Spanning Subgraphs. In *PODC*. ACM, 149–158.

[14] Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. 2021. Distributed weighted min-cut in nearly-optimal time. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. 1144–1153.

[15] Michal Dory and Mohsen Ghaffari. 2019. Improved Distributed Approximations for Minimum-Weight Two-Edge-Connected Spanning Subgraph. In *PODC*. ACM, 521–530.

[16] Michael Elkin. 2006. A faster distributed protocol for constructing a minimum spanning tree. *J. Comput. Syst. Sci.* 72, 8 (2006), 1282–1308. Announced at SODA'04.

[17] Michael Elkin. 2020. Distributed Exact Shortest Paths in Sublinear Time. *J. ACM* 67, 3 (2020), 15:1–15:36. Announced at STOC'17.

[18] Shimon Even. 1975. An Algorithm for Determining Whether the Connectivity of a Graph is at Least k. *SIAM J. Comput.* 4 (1975), 393–396.

[19] Shimon Even and Robert Endre Tarjan. 1975. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.* 4, 4 (1975), 507–518.

[20] Sebastian Forster and Danupon Nanongkai. 2018. A Faster Distributed Single-Source Shortest Paths Algorithm. In *FOCS*. IEEE Computer Society, 686–697.

[21] Sebastian Forster, Danupon Nanongkai, Liu Yang, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2020. Computing and testing small connectivity in near-linear time and queries via fast local cut algorithms. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM, 2046–2065.

[22] Harold N. Gabow. 2006. Using Expander Graphs to Find Vertex Connectivity. *J. ACM* 53, 5 (sep 2006), 800–844. https://doi.org/10.1145/1183907.1183912

[23] Juan A. Garay, Shay Kutten, and David Peleg. 1998. A Sublinear Time Distributed Algorithm for Minimum-Weight Spanning Trees. *SIAM J. Comput.* 27, 1 (1998), 302–316.

[24] Pawel Gawrychowski, Shay Mozes, and Oren Weimann. 2020. Minimum Cut in O(m $\log^2$ n) Time. In *ICALP (LIPIcs, Vol. 168)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 57:1–57:15.

[25] Pawel Gawrychowski, Shay Mozes, and Oren Weimann. 2021. A Note on a Recent Algorithm for Minimum Cut. In *SOSA*. SIAM, 74–79.

[26] Loukas Georgiadis. 2010. Testing 2-Vertex Connectivity and Computing Pairs of Vertex-Disjoint *s*-*t* Paths in Digraphs. In *ICALP (1) (Lecture Notes in Computer Science, Vol. 6198)*. Springer, 738–749.

[27] Mohsen Ghaffari. 2015. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 3–12.

[28] Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. 2015. Near-Optimal Distributed Maximum Flow: Extended Abstract. In *PODC*. ACM, 81–90.

[29] Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. 2018. Near-Optimal Distributed Maximum Flow. *SIAM J. Comput.* 47, 6 (2018), 2078–2117.

[30] Mohsen Ghaffari and Fabian Kuhn. 2013. Distributed minimum cut approximation. In *International Symposium on Distributed Computing*. Springer, 1–15.

[31] Mohsen Ghaffari and Jason Li. 2018. Improved distributed algorithms for exact shortest paths. In *STOC*. ACM, 431–444.

[32] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. 2020. Faster Algorithms for Edge Connectivity via Random 2-Out Contractions. In *SODA*. SIAM, 1260–1279.

[33] Mohsen Ghaffari and Rajan Udwani. 2015. Brief announcement: Distributed single-source reachability. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. 163–165.

[34] Bernhard Haeupler, David Wajc, and Goran Zuzic. 2021. Universally-Optimal Distributed Algorithms for Known Topologies. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 1166–1179. https://doi.org/10.1145/3406325.3451081

[35] Magnús M. Halldórsson, Fabian Kuhn, Yannic Maus, and Tigran Tonoyan. 2021. Efficient randomized distributed coloring in CONGEST. In *STOC*. ACM, 1180–1193.

[36] Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. 2021. A Deterministic Almost-Tight Distributed Algorithm for Approximating Single-Source Shortest Paths. *SIAM J. Comput.* 50, 3 (2021).

[37] Monika Henzinger, Satish Rao, and Di Wang. 2020. Local Flow Partitioning for Faster Edge Connectivity. *SIAM J. Comput.* 49, 1 (2020), 1–36. Announced at SODA'17.

[38] Monika R. Henzinger, Satish Rao, and Harold N. Gabow. 2000. Computing Vertex Connectivity: New Bounds from Old Techniques. *Journal of Algorithms* 34, 2 (2000), 222–250. Announced at FOCS'96.

[39] John E. Hopcroft and Robert Endre Tarjan. 1973. Dividing a Graph into Triconnected Components. *SIAM J. Comput.* 2, 3 (1973), 135–158.

[40] Arkady Kanevsky and Vijaya Ramachandran. 1991. Improved Algorithms for Graph Four-Connectivity. *J. Comput. Syst. Sci.* 42, 3 (1991), 288–306. Announced at FOCS'87.

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

[42] Ken-ichi Kawarabayashi and Mikkel Thorup. 2015. Deterministic Global Minimum Cut of a Simple Graph in Near-Linear Time. In *STOC*. ACM, 665–674.

[43] D. Kleitman. 1969. Methods for Investigating Connectivity of Large Graphs. *IEEE Transactions on Circuit Theory* 16, 2 (1969), 232–233. https://doi.org/10.1109/TCT.1969.1082941

[44] Shay Kutten and David Peleg. 1998. Fast Distributed Construction of Small *k*-Dominating Sets and Applications. *J. Algorithms* 28, 1 (1998), 40–66.

[45] Troy Lee, Tongyang Li, Miklos Santha, and Shengyu Zhang. 2021. On the Cut Dimension of a Graph. In *CCC (LIPIcs, Vol. 200)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 15:1–15:35.

[46] Troy Lee, Miklos Santha, and Shengyu Zhang. 2021. Quantum algorithms for graph problems with cut queries. In *SODA*. SIAM, 939–958.

[47] Christoph Lenzen and Boaz Patt-Shamir. 2013. Fast routing table construction using small messages: extended abstract. In *STOC*. ACM, 381–390.

[48] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2021. Vertex connectivity in polylogarithmic max-flows. In *STOC*. ACM, 317–329.

[49] Jason Li and Debmalya Panigrahi. 2020. Deterministic Min-cut in Polylogarithmic Max-flows. In *FOCS*. IEEE, 85–92.

[50] Nati Linial, Lovász László, and A. Wigderson. 1988. Rubber bands, convex embeddings and graph connectivity. *Combinatorica* 8 (01 1988), 91–102. https://doi.org/10.1007/BF02122557

[51] Yang P Liu, Arun Jambulapati, and Aaron Sidford. 2019. Parallel reachability in almost linear work and square root depth. In *2019 IEEE 60th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 1664–1686.

[52] Sagnik Mukhopadhyay and Danupon Nanongkai. 2020. Weighted min-cut: sequential, cut-query, and streaming algorithms. In *STOC*. ACM, 496–509.

[53] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs. *SIAM J. Discret. Math.* 5, 1 (1992), 54–66.

[54] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. A linear-time algorithm for finding a sparsek-connected spanning subgraph of ak-connected graph. *Algorithmica* 7, 1 (1992), 583–596.

[55] Danupon Nanongkai. 2014. Distributed approximation algorithms for weighted shortest paths. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. 565–573.

[56] Danupon Nanongkai and Thatchaphol Saranurak. 2017. Dynamic spanning forest with worst-case update time: adaptive, Las Vegas, and O($n^{1/2 - \epsilon}$)-time. In *STOC*. ACM, 1122–1129.

[57] Danupon Nanongkai, Thatchaphol Saranurak, and Christian Wulff-Nilsen. 2017. Dynamic Minimum Spanning Forest with Subpolynomial Worst-Case Update Time. In *FOCS*. IEEE Computer Society, 950–961.

[58] Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2019. Breaking quadratic time for small vertex connectivity and an approximation scheme. In *STOC*. ACM, 241–252.

[59] Danupon Nanongkai and Hsin-Hao Su. 2014. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*. Springer, 439–453.

[60] Merav Parter. 2019. Small Cuts and Connectivity Certificates: A Fault Tolerant Approach. In *DISC (LIPIcs, Vol. 146)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 30:1–30:16.

[61] Merav Parter and Asaf Petruschka. 2022. Near-Optimal Distributed Computation of Small Vertex Cuts. In *36th International Symposium on Distributed Computing (DISC 2022)*.

[62] David Peleg and Vitaly Rubinovich. 2000. A Near-Tight Lower Bound on the Time Complexity of Distributed Minimum-Weight Spanning Tree Construction. *SIAM J. Comput.* 30, 5 (2000), 1427–1442. announce at FOCS'99.

[63] David Pritchard and Ramakrishna Thurimella. 2011. Fast computation of small cuts via cycle space sampling. *ACM Trans. Algorithms* 7, 4 (2011), 46:1–46:30.

[64] Aviad Rubinstein, Tselil Schramm, and S. Matthew Weinberg. 2018. Computing Exact Minimum Cuts Without Knowing the Graph. In *ITCS (LIPIcs, Vol. 94)*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 39:1–39:16.

[65] Thatchaphol Saranurak and Di Wang. 2019. Expander Decomposition and Pruning: Faster, Stronger, and Simpler. In *SODA*. SIAM, 2616–2635.

[66] Daniel A. Spielman and Shang-Hua Teng. 2013. A Local Clustering Algorithm for Massive Graphs and Its Application to Nearly Linear Time Graph Partitioning. *SIAM J. Comput.* 42, 1 (2013), 1–26.

[67] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. Announced at FOCS'71.

[68] Ramakrishna Thurimella. 1997. Sub-Linear Distributed Algorithms for Sparse Certificates and Biconnected Components. *J. Algorithms* 23, 1 (1997), 160–179. Announced at PODC'95.

[69] Christian Wulff-Nilsen. 2017. Fully-dynamic minimum spanning forest with improved worst-case update time. In *STOC*. ACM, 1130–1143.

Received 2022-11-07; accepted 2023-02-06