Aalto University

OPEN ACCESS

This is an electronic reprint of the original article.
This reprint may differ from the original in pagination and typographic detail.

Nanongkai, Danupon; Saranurak, Thatchaphol; Yingchareonthawornchai, Sorrachai

Breaking quadratic time for small vertex connectivity and an approximation scheme

Published in:
STOC 2019 - Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing

DOI:
10.1145/3313276.3316394

Published: 23/06/2019

Document Version
Peer-reviewed accepted author manuscript, also known as Final accepted manuscript or Post-print

Please cite the original version:
Nanongkai, D., Saranurak, T., & Yingchareonthawornchai, S. (2019). Breaking quadratic time for small vertex connectivity and an approximation scheme. In M. Charikar, & E. Cohen (Eds.), *STOC 2019 - Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing* (pp. 241-252). (Proceedings of the Annual ACM Symposium on Theory of Computing). ACM. https://doi.org/10.1145/3313276.3316394

This material is protected by copyright and other intellectual property rights, and duplication or sale of all or part of any of the repository collections is not permitted, except that material may be duplicated by you for your research use or educational purposes in electronic or print form. You must obtain permission for any other use. Electronic or print copies may not be offered, whether for sale or otherwise to anyone who is not an authorised user.

Breaking Quadratic Time for Small Vertex Connectivity and an Approximation Scheme*

Danupon Nanongkai
danupon@gmail.com
KTH Royal Institute of Technology
Stockholm, Sweden

Thatchaphol Saranurak
saranurak@ttic.edu
Toyota Technological Institute at Chicago
IL, USA

Sorrachai Yingchareonthawornchai
sorrachai.cp@gmail.com
Michigan State University and Aalto University
MI and Espoo, USA and Finland

ABSTRACT

Vertex connectivity a classic extensively-studied problem. Given an integer k, its goal is to decide if an n-node m-edge graph can be disconnected by removing k vertices. Although a linear-time algorithm was postulated since 1974 [Aho, Hopcroft and Ullman], and despite its sibling problem of edge connectivity being resolved over two decades ago [Karger STOC'96], so far no vertex connectivity algorithms are faster than $O(n^2)$ time even for $k = 4$ and $m = O(n)$. In the simplest case where $m = O(n)$ and $k = O(1)$, the $O(n^2)$ bound dates five decades back to [Kleitman IEEE Trans. Circuit Theory'69]. For higher m, $O(m)$ time is known for $k \leq 3$ [Tarjan FOCS'71; Hopcroft, Tarjan SICOMP'73], the first $O(n^2)$ time is from [Kanevsky, Ramachandran, FOCS'87] for $k = 4$ and from [Nagamochi, Ibaraki, Algorithmica'92] for $k = O(1)$. For general k and m, the best bound is $\tilde{O}(\min(kn^2, n^\omega + nk^\omega))$ [Henzinger, Rao, Gabow FOCS'96; Linial, Lovász, Wigderson FOCS'86] where $\tilde{O}$ hides polylogarithmic terms and $\omega < 2.38$ is the matrix multiplication exponent.

In this paper, we present a randomized Monte Carlo algorithm with $\tilde{O}(m + k^{7/3}n^{4/3})$ time for any $k = O(\sqrt{n})$. This gives the *first subquadratic time* bound for any $4 \leq k \leq o(n^{2/7})$ (subquadratic time refers to $O(m) + o(n^2)$ time.) and improves all above classic bounds for all $k \leq n^{0.44}$. We also present a new randomized Monte Carlo $(1+\epsilon)$-approximation algorithm that is strictly faster than the previous Henzinger's 2-approximation algorithm [J. Algorithms'97] and all previous exact algorithms. The story is the same for the *directed* case, where our exact $\tilde{O}(\min\{km^{2/3}n, km^{4/3}\})$-time for any $k = O(\sqrt{n})$ and $(1+\epsilon)$-approximation algorithms improve all previous exact bounds. Additionally, our algorithm is the first approximation algorithm on directed graphs.

The key to our results is to avoid computing single-source connectivity, which was needed by all previous exact algorithms and is not known to admit $o(n^2)$ time. Instead, we design the first *local algorithm* for computing vertex connectivity; without reading the whole graph, our algorithm can find a separator of size at most k or certify that there is no separator of size at most k "near" a given seed node.

CCS CONCEPTS

• **Theory of computation → Graph algorithms analysis**.

KEYWORDS

graph algorithms, vertex connectivity, local flow algorithms

ACM Reference Format:
Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. 2019. Breaking Quadratic Time for Small Vertex Connectivity and an Approximation Scheme. In *Proceedings of the 51st Annual ACM SIGACT Symposium on the Theory of Computing (STOC '19), June 23–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3313276.3316394

1 INTRODUCTION

Vertex connectivity is a central concept in graph theory. The vertex connectivity κ_G of a graph G is the minimum number of the nodes needed to be removed to disconnect some remaining node from another remaining node. (When G is directed, this means that there is no directed path from some node u to some node v in the remaining graph.)

Since 1969, there has been a long line of research on efficient algorithms [5–8, 10–12, 14, 15, 21, 22, 28, 30–32, 36] for *deciding k-connectivity* (i.e. deciding if $\kappa_G \geq k$) or *computing the connectivity* κ_G. For the undirected case, Aho, Hopcroft and Ullman [1, Problem 5.30] conjecture in 1974 that there exists an $O(m)$-time algorithm for computing κ_G on a graph with n nodes and m edges. However, no algorithms to date are faster than $O(n^2)$ time even for $k = 4$.

On undirected graphs, the first $O(n^2)$ bound for the simplest case, where $m = O(n)$ and $k = O(1)$, dates back to five decades ago: Kleitman [28] in 1969 presented an algorithm for deciding k-connectivity with running time $O(kn \cdot \mathrm{VC}_k(n, m))$ where $\mathrm{VC}_k(n, m)$ is the time needed for deciding if the minimum size s-t vertex-cut is of size at least κ, for fixed s, t. Although the running time bound was not explicitly stated, it was known that $\mathrm{VC}_k(n, m) = O(mk)$ by Ford-Fulkerson algorithm [13]. This gives $O(k^2nm)$ which is $O(n^2)$ when $m = O(n)$ and $k = O(1)$, when we plug in the 1992 result of Nagamochi and Ibaraki [32]. Subsequently, Tarjan [40] and Hopcroft and Tarjan [24] presented $O(m)$-time algorithms when k is 2 and 3 respectively.

*The full version of this paper is at https://arxiv.org/abs/1904.04453.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
STOC '19, June 23–26, 2019, Phoenix, AZ, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6705-9/19/06…$15.00
https://doi.org/10.1145/3313276.3316394

All subsequent works improved Kleitman's bound for larger k and m, but none could break beyond $O(n^2)$ time. For $k = 4$ and any m, the first $O(n^2)$ bound was by Kanevsky and Ramachandran [25]. The first $O(n^2)$ for any $k = O(1)$ (and any m) was by Nagamochi and Ibaraki [32]. For general k and m, the fastest running times are $\tilde{O}(n^\omega + nk^\omega)$ by Linial, Lovász and Wigderson [30] and $\tilde{O}(kn^2)$ by Henzinger, Rao and Gabow [22]. Here, $\tilde{O}$ hides polylog(n) terms, and ω is the matrix multiplication exponent. Currently, $\omega < 2.37287$ [16].

For directed graphs, an $O(m)$-time algorithm is known only for $k \leq 2$ by Georgiadis [17]. For general k and m, the fastest running times are $\tilde{O}(n^\omega + nk^\omega)$ by Cheriyan and Reif [7] and $\tilde{O}(mn)$ by Henzinger et al. [22]. All mentioned state-of-the-art algorithms for general k and m, for both directed and undirected cases [7, 22, 30], are randomized and correct with high probability. The fastest deterministic algorithm is by Gabow [14] and has slower running time. Some *approximation algorithms* have also been developed. The first is the deterministic 2-approximation $O(\min\{\sqrt{n}, k\}n^2)$-time algorithm by Henzinger [21]. The second is the recent randomized $O(\log n)$-approximation $\tilde{O}(m)$-time algorithm by Censor-Hillel, Ghaffari, and Kuhn [6]. Both algorithms work only on undirected graphs.

Besides a few $O(m)$-time algorithms for $k \leq 3$, all previous exact algorithms could not go beyond $O(n^2)$ for a common reason: As a subroutine, they have to solve the following problem. For a pair of nodes s and t, let $\kappa(s,t)$ denote the minimum number of nodes (excluding s and t) required to be removed so that there is no path from s to t in the remaining graph. In all previous algorithms, there is always some node s such that these algorithms decide if $\kappa(s,t) \geq k$ for all other nodes t (and some algorithms in fact computes $\kappa(s,t)$ for all t). We call this problem *single-source k-connectivity*. Until now, there is no $o(n^2)$-time algorithm for this problem even when $k = O(1)$ and $m = O(n)$.

1.1 Our Results

In this paper, we present first algorithms that break the $O(n^2)$ bound on both undirected and undirected graphs, when k is small. More precisely:

Theorem 1.1. *There are randomized (Monte Carlo) algorithms that take as inputs an n-node m-edge graph $G = (V, E)$ and an integer $k = O(\sqrt{n})$, and can decide w.h.p.[1] if $\kappa_G \geq k$. If $\kappa_G < k$, then the algorithms also return corresponding separator $S \subset V$, i.e. a set S where $|S| = \kappa_G$ and $G[V - S]$ is not connected if G is undirected and not strongly connected if G is directed. The algorithm takes $\tilde{O}(m + k^{7/3}n^{4/3})$ and $\tilde{O}(\min(km^{2/3}n, km^{4/3}))$ time on undirected and directed graphs, respectively.*

Our bounds are the *first* $o(n^2)$ for the range $4 \leq k \leq o(n^{2/7})$ on undirected graphs and range $3 \leq k \leq o(n/m^{2/3})$ on directed graphs. Our algorithms are combinatorial, meaning that they do not rely on fast matrix multiplication. For all range of k that our algorithms support, i.e. $k = O(\sqrt{n})$, our algorithms improve upon the previous best combinatorial algorithms by Henzinger et al. [22], which take time $\tilde{O}(kn^2)$ on undirected graphs and $\tilde{O}(mn)$ on directed graphs[2]. Comparing with the $\tilde{O}(n^\omega + nk^\omega)$ bound based on algebraic techniques by Linial et al. [30] and Cheriyan and Reif [7], our algorithms are faster on undirected graphs when $k \leq n^{3\omega/7-4/7} \approx n^{0.44}$. For directed graph, our algorithm is faster where the range k depends on graph density. For example, consider the interesting case the graph is sparse but can still be k-connected which is when $m = O(nk)$. Then ours is faster than [7] for any $k \leq n^{0.44}$ like the undirected case. However, in the dense case when $m = \Omega(n^2)$, ours is faster than [7] for any $k \leq n^{\omega-7/3} \approx n^{0.039}$.

To conclude, our bounds are lower than all previous bounds when $4 \leq k \leq n^{0.44}$ for undirected graphs and $3 \leq k \leq n^{0.44}$ for directed sparse graphs (i.e. when $m = O(nk)$). All these bounds [7, 22, 30] have not been broken for over 20 years. In the simplest case where $m = O(n)$ and, hence $k = O(1)$, we break the 50-year-old $O(n^2)$ bound [28] down to $\tilde{O}(n^{4/3})$ for both undirected and directed graphs, respectively.

Approximation algorithms. We can adjust the same techniques to get $(1+\epsilon)$-approximate κ_G with faster running time. In addition, we give another algorithm using a different technique that can $(1+\epsilon)$-approximate κ_G in $\tilde{O}(n^\omega/\epsilon^2)$ time.

We define the function $T_{\text{flow}}(k, m, n)$ as

$$T_{\text{flow}}(k,m,n) = \begin{cases} \min(m^{4/3}, nm^{2/3}k^{1/2}, \\ \quad mn^{2/3+o(1)}/k^{1/3}, \\ \quad n^{7/3+o(1)}/k^{1/6}) & \text{if } k \leq n^{4/5}, \\ n^{3+o(1)}/k & \text{if } k > n^{4/5}. \end{cases} \quad (1)$$

Theorem 1.2 (Approximation Algorithm). *There is a randomized (Monte Carlo) algorithm that takes as input an n-node m-edge graph $G = (V, E)$ and w.h.p. outputs $\tilde{\kappa}$, where $\kappa_G \leq \tilde{\kappa} \leq (1+\epsilon)\kappa_G$, in $\tilde{O}(m + \text{poly}(1/\epsilon)\min(k^{4/3}n^{4/3}, k^{2/3}n^{5/3+o(1)}, n^{3+o(1)}/k, n^\omega)) = \tilde{O}(\min\{n^{2.2}, n^\omega\})$ time for undirected graph, and in $\tilde{O}(\text{poly}(1/\epsilon)$ $\min(T_{\text{flow}}(k,m,n), n^\omega)) = \tilde{O}(\min\{n^{2.2}, n^\omega\})$ time for directed graph where $T_{\text{flow}}(k,m,n)$ is defined in Equation (1). The algorithm also returns a pair of nodes x and y where $\kappa(x,y) = \tilde{\kappa}$. Hence, with additional $O(m\min\{\sqrt{n}, \tilde{\kappa}\})$ time, the algorithm can compute the corresponding separator.*

As noted earlier, previous algorithms achieve 2-approximation in $O(\min\{\sqrt{n}, k\}n^2)$-time [21] and $O(\log n)$-approximation in $\tilde{O}(m)$ time [6]. For all possible values of k, our algorithms are strictly faster than the 2-approximation algorithm of [21].

Our approximation algorithms are also strictly faster than all previous exact algorithms with current matrix multiplication time (and are never slower even if $\omega < 2.2$). In particular, even when $\epsilon = 1/n^\gamma$ for small constant $\gamma > 0$, our algorithms are always polynomially faster than the exact algorithms by [22] with running time $\tilde{O}(mn)$ and $\tilde{O}(kn^2)$ on directed and undirected graphs, respectively. Compared with the bound $\tilde{O}(n^\omega + nk^\omega)$ by [30] and [7], our bound for undirected and directed graphs are $\tilde{O}(\min\{n^{2.2}, n^\omega\})$ for any density, which are less than current matrix multiplication time. Finally, note that the previous approximation algorithms [6, 21] only work on undirected graphs, while we also show algorithms on directed graphs.

1.2 The Key Technique

At the heart of our main result in Theorem 1.1 is a new *local algorithm* for finding minimum vertex cuts. In general, we say that an

[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.

[2] As $k \leq \sqrt{n}$ and $m \geq nk$, we have $k \leq m^{1/3}$. So $km^{2/3}n \leq mn$.

algorithm is *local* if its running time does not depend on the size of the whole input.

More concretely, let $G = (V, E)$ be a directed graph where each node u has out-degree $\deg^{\text{out}}(u)$. Let $\deg^{\text{out}}_{\min} = \min_u \deg^{\text{out}}(u)$ be the minimum out-degree. For any set $S \subset V$, the *out-volume* of S is $\text{vol}^{\text{out}}(S) = \sum_{u \in S} \deg^{\text{out}}(u)$ and the set of *out-neighbors* of S is $N^{\text{out}}(S) = \{v \notin S \mid (u, v) \in E\}$. We show the following algorithm (see Theorem 4.1 for a more details):

Theorem 1.3 (Local vertex connectivity (informal)). *There is a deterministic algorithm that takes as inputs a node x in a graph G and parameters ν and k where ν, k are not too large, and in $\tilde{O}(\nu^{1.5}k)$ time either*

(1) returns a set $S \ni x$ where $|N^{\text{out}}(S)| \le k$, or
(2) certifies that there is no set $S \ni x$ such that $\text{vol}^{\text{out}}(S) \le \nu$ and $|N^{\text{out}}(S)| \le k$.

Our algorithm is the first local algorithm for finding small vertex cuts (i.e. finding small separator $N^{\text{out}}(S)$). The algorithm either finds a separator of size at most k, or certifies that no separator of size at most k exists "near" some node x. Our algorithm is exact in the sense that there is no gap on the cut size k in the two cases.

Previously, there was a rich literature on local algorithms for finding *low conductance cuts*[3], which is a different problem from ours. The study was initiated by Spielman and Teng [38] in 2004. Since then, deep techniques have been further developed, such as *spectral-based* techniques[4] (e.g. [2–4, 18, 39]) and newer *flow-based* techniques [20, 35, 41, 42]). Applications of these techniques for finding low conductance cuts are found in various contexts (e.g. balanced cuts [37, 39]), edge connectivity [20, 27], and dynamically maintaining expanders [33, 34, 37, 43]).

It is not clear a priori that these previous techniques can be used for proving Theorem 1.3. First of all, they were invented to solve a different problem, and there are several small differences about technical input-output constraints. More importantly is the following conceptual difference. In most previous algorithms, there is a "gap" between the two cases of the guarantees. That is, if in one case the algorithm can return a cut $S \ni x$ whose conductance is at most $\phi \in (0, 1)$, then in the other case the algorithm can only guarantees that there is no cut "near" x with conductance $\alpha\phi$, for some $\alpha = o(1)$ (e.g. $\alpha = O(\phi)$ or $O(1/\log n)$)[5].

Because of these differences, not many existing techniques can be adapted to design a local algorithm for vertex connectivity. In fact, we are not aware of any spectral-based algorithms that can solve this problem, even when we can read the whole graph. Fortunately, it turns out that Theorem 1.3 can be proved by adapting some recent flow-based techniques. In general, a challenge in designing flow-based algorithms is to achieve the following goals simultaneously.

(1) Design some well-structured graph so that finding flows on this graph is useful for our application (proving Theorem 1.3 in this case). We call such graph an *augmented graph*.
(2) At the same time, design a local flow-based algorithm which is fast when running of the augmented graph.

[3]The conductance of a cut $(S, V - S)$ is defined as $\Phi(S) = \frac{|E(S, V-S)|}{\min\{\text{vol}(S), \text{vol}(V-S)\}}$.
[4]They are algorithms based on some random-walk or diffusion process.
[5]The algorithms from [20, 27] in fact do not guarantee non-existence of some low conductance cuts in the second case, but the guarantee is about min-cuts.

For the first task, the design of the augmented graph require some careful choices (see Section 2.2 for the high-level ideas and Section 4.1 for details). For the second task, it turns out that previous flow-based local algorithms [20, 35, 41, 42] can be adjusted to give useful answers for our applications when run on our augmented graph. However, these previous algorithms only give slower running time of at least $\tilde{O}((\nu k)^{1.5})$. To obtain the $\tilde{O}(\nu^{1.5}k)$ bound, we first speed up Goldberg-Rao max flow algorithm [19] from running time $\tilde{O}(m \min\{\sqrt{m}, n^{2/3}\})$ to $\tilde{O}(m\sqrt{n})$ when running on a graph with certain structure. Then, we "localize" this algorithm in a similar manner as in [35], which completes our second task (see Section 2.2 for more discussion).

As a byproduct, our modification of Goldberg-Rao algorithm in fact gives the fastest weakly-polynomial algorithm for computing s-t vertex connectivity in node-weighted graphs:

Theorem 1.4 (Weighted s-t vertex connectivity). *Let $G = (V, E)$ be a directed graph with n nodes and m edges where each node has integer weight from $[1, U]$. For any $s, t \in V$, in time $O(m\sqrt{n}\log n \log U))$, we can compute deterministically the minimum weight s-t separator $S \subset V$, i.e., $s, t \notin S$ and there is no path from s to t in $G[V - S]$.*

The previous fastest algorithm is by using the general max flow algorithm by Lee and Sidford [29], giving an $O(m\sqrt{n}\text{polylog}(nU))$ running time. This algorithm is randomized. Our algorithm is deterministic and slightly faster.

Given the key local algorithm in Theorem 1.3, we obtain Theorems 1.1 and 1.2 by combining our local algorithms with other known techniques including random sampling, Ford-Fulkerson algorithm, Nagamochi Irabaki's connectivity certificate [32] and convex embedding [7, 30]. We sketch how everything fits together in Section 2.

2 OVERVIEW

2.1 Exact Algorithm

To illustrate the main idea, let us sketch our algorithm with running time $\tilde{O}(m + n^{4/3})$ only on an *undirected graph* with $m = O(n)$ and $k = O(1)$. This regime is already very interesting, because the best bound has been $\tilde{O}(n^2)$ for nearly 50 years [28]. Throughout this section, $N(C)$ is a set of neighbors of nodes in $C \subseteq V$ that are not in C, and $E_G(S, T)$ is the set of edges between (not necessarily disjoint) vertex sets S and T in G (the subscript is omitted when the context is clear). A vertex partition (A, S, B) is called a *separation triple* if $A, B \neq \emptyset$ and there is no edge between A and B, i.e., $N(A) = S = N(B)$.

Given a graph $G = (V, E)$ and a parameter k, our goal is to either return a set $C \subset V$ where $|N(C)| < k$ or certify that $\kappa_G \ge k$. Our first step is to find a sparse subgraph H of G where $\kappa_H = \min\{\kappa_G, k\}$ using the algorithm by Nagamochi and Ibaraki [32]. The nice property of H is that it is formed by a union of k disjoint forests, i.e. H has *arboricity* k. In particular, for any set of nodes C, we have $|E_H(C, C)| \le k|C|$. As the algorithm only takes linear time, from now, we treat H as our input graph G.

The next step has three cases. First, suppose there is a separation triple (A, S, B) where $|S| < k$ and $|A|, |B| \ge n^{2/3}$. Here, we sample $\tilde{O}(n^{1/3})$ many pairs (x, y) of nodes uniformly at random. With high

probability, one of these pairs is such that $x \in A$ and $y \in B$. In this case, it is well known (e.g. [11]) that one can modify the graph and run a max xy-flow algorithm. Thus, for each pair (x, y), we run Ford-Fulkerson max-flow algorithm in time $O(km) = O(n)$ to decide whether $\kappa(x, y) < k$ and if so, return the corresponding cut. So w.h.p. the algorithm returns set C where $|N(C)| < k$ in total time $\tilde{O}(n^{1+1/3})$.

The next case is when all separation triples (A, S, B) where $|S| < k$ are such that either $|A| < n^{2/3}$ or $|B| < n^{2/3}$. Suppose w.l.o.g. that $|A| < n^{2/3}$. By a binary search trick, we can assume to know the size $|A|$ up to a factor of 2. Here, we sample $\tilde{O}(n/|A|)$ many nodes uniformly at random. For each node x, we run the local vertex connectivity subroutine from Theorem 1.3 where the parameter k in Theorem 1.3 is set to be $k - 1$. Note that the volume of A is

$$\text{vol}(A) = 2|E(A, A)| + |E(A, S)| = O(k|A|) = O(|A|)$$

where the second equality is because G has arboricity k and $|S| < k$ (also recall that we only consider $m = O(n)$ and $k = O(1)$ in this subsection). We set the parameter $\nu = \Theta(|A|)$. With high probability, we have that one of the samples x must be inside A. Here, the local-max-flow cannot be in the second case, and will return a set C where $|N(C)| < k$, which implies that $\kappa_G < k$. The total running time is $\tilde{O}(n/|A|) \times \tilde{O}(|A|^{1.5}) = \tilde{O}(n^{1+1/3})$ because $|A| < n^{2/3}$.

The last case is when $\kappa_G \geq k$. Here, both of Ford-Fulkerson algorithm and local max flow algorithm will never return any set C where $|N(C)| < k$. So we can correctly report that $\kappa_G \geq k$. All of our techniques generalize to the case when κ_G is not constant.

2.2 Local Vertex Connectivity

In this section, we give a high-level idea how to obtain our local vertex connectivity algorithm in Theorem 1.3. Recall from the introduction that there are two tasks which are to design an *augmented graph* and to devise a *local flow-based algorithm* running on such augmented graph. We have two goals: 1) the running time of our algorithm is *local*; i.e., it does not depend on the size of the whole graph and 2) the local flow-based algorithm's output should be useful for our application.

The local time principles. We first describe high-level principles on how to design the augmented graph and the local flow-based algorithm so that the running time is local[6].

(1) *Augmented graph is absorbing*: Each node u of the augmented graph is a *sink* that can "absorb" flow proportional to its degree $\deg(u)$. More formally, each node u is connected to a *super-sink* t with an edge (u, t) of capacity $\alpha \deg(u)$ for some constant α. In our case, $\alpha = 1$.

(2) *Flow algorithm tries to absorb before forward*: Suppose that a node u does not fully absorb the flow yet, i.e. (u, t) is not saturated. When a flow is routed to u, the local flow-based algorithm must first send a flow from u to t so that the sink at u is fully absorbed, before forwarding to other neighbors of u. Moreover, the absorbed flow at u will stay at u forever.

We give some intuition behind these principles. The second principle resembles the following physical process. Imagine pouring water on a compartment of an ice tray. There cannot be water flowing out of an unsaturated compartment until that compartment is saturated. So if the amount of initial water is small, the process will stop way before the water reaches the whole ice tray. This explains in principle why the algorithm needs not read the whole graph.

The first principle allows us to argue why the cost of the algorithm is proportional to the part of the graph that is read. Very roughly, the total cost for forwarding the flow from a node u to its neighbors depends on $\deg(u)$, but at the same time we forward the flow only after it is already fully absorbed at u. This allows us to charge the total cost to the total amount of absorbed flow, which in turn is small if the initial amount of flow is small.

Augmented graph. Let us show how to design the augmented graph in the context of *edge connectivity* in undirected graphs first. The construction is simpler than the case of vertex connectivity, but already captures the main idea. We then sketch how to extend this idea to vertex connectivity.

Let $G = (V, E)$ be an undirected graph with m edges and $x \in V$ be a node. Consider any numbers $\nu, k > 0$ such that

$$2\nu k + \nu + 1 \leq 2m. \tag{2}$$

We construct an undirected graph G' as follows. The node set of G' is $V(G') = \{s\} \cup V \cup \{t\}$ where s and t is a super-source and a super-sink respectively. For each node u, add (u, t) with capacity $\deg_G(u)$. (So, this satisfied the first local time principle.) For each edge $(u, v) \in E$, set the capacity to be 2ν. Finally, add an edge (s, x) with capacity $2\nu k + \nu + 1$.

Theorem 2.1. *Let F^* be the value of the s-t max flow in G'. We have the following:*

(1) If $F^* = 2\nu k + \nu + 1$, then there is no vertex partition (S, T) in G where $S \ni x$, $\text{vol}(S) \leq \nu$ and $|E(S, V - S)| \leq k$.

(2) If $F^* \leq 2\nu k + \nu$, then there is a vertex partition (S, T) in G where $S \ni x$ and $|E(S, V - S)| \leq k$.

Proof. To see (1), suppose for a contradiction that there is such a partition (S, T) where $S \ni x$. Let $(S', T') = (\{s\} \cup S, T \cup \{t\})$. The edges between S' and T' has total capacity

$$c(E_{G'}(S', T')) = 2\nu|E_G(S, V - S)| + \text{vol}_G(S) \leq 2\nu k + \nu.$$

So $F^* \leq 2\nu k + \nu$, a contradiction. To see (2), let $(S', T') = (\{s\} \cup S, T \cup \{t\})$ be a min st-cut in G' corresponding to the max flow, i.e. by the min-cut max-flow theorem, the edges between S' and T' has total capacity

$$c(E_{G'}(S', T')) \leq 2\nu k + \nu. \tag{3}$$

Observe that $S' \neq \{s\}$ and $S \ni x$ because the edge (s, x) has capacity strictly more than $2\nu k + \nu$. Also, $T' \neq \{t\}$ because edges between $\{s\} \cup V$ and $\{t\}$ has total capacity $\text{vol}(V) = 2m > 2\nu k + \nu$ (the inequality is because of Equation (2)). So (S, T) gives a cut in G where $S \ni x$. Suppose that $|E_G(S, T)| \geq k + 1$, then $c(E_{G'}(S', T')) \geq 2\nu(k + 1) = 2\nu k + 2\nu > 2\nu k + \nu$ which contradicts Equation (3). □

Observe that the above theorem is similar to Theorem 1.3 except that it is about edge connectivity. To extend this idea to vertex connectivity, we use a standard transformation as used in [12, 22] by constructing a so-called *split graph*. In our split graph, for each

[6] In fact, these are also principles behind all previous local flow-based algorithms. To the best of our knowledge, these general principles have not been stated. We hope that they explain previous seemingly ad-hoc results.

node v, we create two nodes v_{in} and v_{out}. For each edge (u, v), we create an edge $(u_{\text{out}}, v_{\text{in}})$ with infinite capacity. There is an edge $(v_{\text{in}}, v_{\text{out}})$ for each node v as well. Observe that a cut set with finite capacity in the split graph corresponds to a set of nodes in the original graph. Then, we create the augmented graph of the split graph in a similar manner as above, e.g. by adding nodes s and t and an edge (s, x) with $2\nu k + \nu + 1$. The important point is that we set the capacity of each $(v_{\text{in}}, v_{\text{out}})$ to be 2ν. The proof of Theorem 1.3 (except the statement about the running time) is similar as above (see Section 4.1 for details).

Local flow-based algorithm. As discussed in the introduction, we can in fact adapt previous local flow-based algorithms to run on our augmented graph and they can decide the two cases in Theorem 1.3 (i.e. whether there is a small vertex cut "near" a seed node x). Theorem 2.1 in fact already allows us to achieve this with slower running time than the desired $\tilde{O}(\nu^{1.5}k)$ by implementing existing local flow-based algorithms. For example, the algorithm by [35], which is a "localized" version of Goldberg-Rao algorithm [19], can give a slower running time of $\tilde{O}((\nu k)^{1.5})$. Other previous local flow-based algorithms that we are aware of (e.g. [20, 35, 41, 42]) give even slower running time (even after appropriate adaptations).

We can speed up the time to $\tilde{O}(\nu^{1.5}k)$ by exploiting the fact that our augmented graph is created from a split graph sketched above. To begin with, we first observe that, when running Goldberg-Rao algorithm on split graphs (which are weighted), the running time can be sped up from $\tilde{O}(m \min\{\sqrt{m}, n^{2/3}\})$ to $\tilde{O}(m\sqrt{n})$. This already gives us the new fastest algorithm for computing *s-t weighted* vertex connectivity as stated in Theorem 1.4. This improvement resembles the idea by Hopcroft and Karp [23] (see also [12, 26]) which yields an $O(m\sqrt{n})$-time algorithm for computing *s-t unweighted* vertex connectivity. The idea is to show that Dinic's algorithm with running time $O(m \min\{\sqrt{m}, n^{2/3}\})$ on a general unit-capacity graph can be sped up to $\tilde{O}(m\sqrt{n})$ when run on a special graph called "unit network". It turns out that unit networks share some structures with our split graphs, allowing us to apply a similar idea. Although our improvement is based on a similar idea, it is more complicated to implement this idea on our split graph since it is weighted.

Finally, we "localize" our improved algorithm by enforcing the second local time principle. Our way to localize the algorithm goes hand in hand with the way Orecchia and Zhu [35] did to the standard Goldberg-Rao algorithm (see Section 4.3 for details).

3 PRELIMINARIES

3.1 Directed Graph

Let $G = (V, E)$ be a *directed* graph where $|V| = n$ and $|E| = m$. For any edge (u, v), we denote $e^R = (v, u)$. For any directed graph $G = (V, E)$, the *reverse graph* G^R is $G^R = (V, E^R)$ where $E^R = \{e^R : e \in E\}$.

Definition 3.1 (δ, deg, vol, N)**.** Definitions below are defined for any vertex v on graph G and subset of vertex $U \subseteq V$.

- $\text{vol}^{\text{out}}_G(U) = \sum_{v \in U} \deg^{\text{out}}_G(v)$ and $\text{vol}^{\text{in}}_G(U) = \sum_{v \in U} \deg^{\text{in}}_G(v)$.
- $N^{\text{in}}_G(v) = \{u : (u, v) \in E\}$ and $N^{\text{out}}_G(v) = \{u : (v, u) \in E\}$.
- $N^{\text{in}}_G(U) = \bigcup_{v \in U} N^{\text{in}}_G(v) \setminus U$ and $N^{\text{out}}_G(U) = \bigcup_{v \in U} N^{\text{out}}_G(v) \setminus U$.

Definition 3.2 (Paths)**.** For $s, t \in V$, we say a path P is an (s, t)-*path* if P is a directed path starting from s and ending at t. For any $S, T \subseteq V$, we say P is an (S, T)-*path* if P starts with some vertex in S and ends at some vertex in T.

Definition 3.3 (Edge- and Vertex-cuts)**.** Let s and t be any distinct vertices. Let $S, T \subset V$ be any disjoint non-empty subsets of vertices. We call any subset of edges $C \subseteq E$ (respectively any subset of vertices $U \subseteq V$):

- an (S, T)-*edge-cut* (respectively an (S, T)-*vertex-cut*) if there is no (S, T)-path in $G \setminus C$ (respectively if there is no (S, T)-path in $G \setminus U$ *and* $S \cap U = \emptyset, T \cap U = \emptyset$),
- an (s, t)-*edge-cut* (respectively an (s, t)-*vertex-cut*) if there is no (s, t)-path in $G \setminus C$ (respectively if there is no (s, t)-path in $G \setminus U$ *and* $s, t \notin U$),
- an *edge-cut* (respectively *vertex-cut*) if it is an (s, t)-edge-cut (respectively (s, t)-vertex-cut) for some distinct vertices s and t. In other words, $G \setminus C$ (respectively $G \setminus U$) is not strongly connected.

If the graph has capacity function $c : E \to \mathbb{R}_{\geq 0}$ on edges, then $c(C) = \sum_{e \in C} c_e$ is the total capacity of the cut C.

Definition 3.4 (Edge set)**.** We define $E(S, T)$ as the set of edges $\{(u, v) : u \in S, v \in T\}$.

Definition 3.5 (Vertex partition)**.** Let $S, T \subset V$. We say that (S, T) is a *vertex partition* if S and T are not empty, and $S \sqcup T = V$. In particular, $E(S, T)$ is an (x, y)-edge-cut for some $x \in S, y \in T$.

Definition 3.6 (Separation triple)**.** We call (L, S, R) a *separation triple* if L, S, and R partition the vertex V in G where L and R are non-empty, and there is no edge from L to R.

Note that, from the above definition, S is an (x, y)-vertex-cut for any $x \in L$ and $y \in R$.

Definition 3.7 (Shore)**.** We call a set of vertices $S \subseteq V$ an *out-vertex shore* (respectively *in-vertex shore*) if $N^{\text{out}}_G(S)$ (respectively $N^{\text{in}}_G(S)$) is a vertex-cut.

Definition 3.8 (Vertex connectivity κ)**.** We define vertex connectivity κ_G as the minimum cardinality vertex-cut or $n-1$ if no vertex cut exists. More precisely, for distinct $x, y \in V$, define $\kappa_G(x, y)$ as the smallest cardinality of (x, y)-vertex-cut if exists. Otherwise, we define $\kappa_G(x, y) = n-1$. Then, $\kappa_G = \min\{\kappa_G(x, y) \mid x, y \in V, x \neq y\}$. We drop the subscript when G is clear from the context.

3.2 Undirected Graph

Let $G = (V, E)$ be an undirected graph. We assume that G is simple, and connected.

Theorem 3.9 ([32]). *There exists an algorithm that takes as input undirected graph $G = (V, E)$, and in $O(m)$ time outputs a sequence of forests $F_1, F_2, \ldots, F_n$ such that each forest subgraph $H_k = (V, \bigcup_{i=1}^{k} F_i)$ is k-connected if G is k-connected. H_k has aboricity k. For any set of vertices S, we have $E_{H_k}(S, S) \leq k|S|$. In particular, the number of edges in H_k is at most kn.*

To compute vertex connectivity in an undirected graph, we turn it into a directed graph by adding edges in forward and backward directions and run the directed vertex connectivity algorithm.

4 LOCAL VERTEX CONNECTIVITY

Recall that a directed graph $G = (V, E)$ is strongly connected where $|V| = n$ and $|E| = m$.

Theorem 4.1. *There is an algorithm that takes as input a pointer to any vertex $x \in V$ in an adjacency list representing a strongly-connected directed graph $G = (V, E)$, positive integer ν ("target volume"), positive integer k ("target x-vertex-cut size"), and positive real ϵ satisfying*

$$\nu/\epsilon + \nu < m, \quad (1+\epsilon)(\frac{2\nu}{\epsilon k} + k) < n \quad \text{and} \quad \deg^{\text{out}}_{\min} \geq k \tag{4}$$

or,

$$\nu/\epsilon + (1+\epsilon)nk < m, \quad \text{and} \quad \deg^{\text{out}}_{\min} \geq k \tag{5}$$

and in $\tilde{O}(\frac{\nu^{3/2}}{\epsilon^{3/2}k^{1/2}})$ time outputs either

- *a vertex-cut S corresponding to the separation triple $(L, S, R), x \in L$ such that*

$$|S| \leq (1+\epsilon)k \quad \text{and} \quad \text{vol}^{out}_G(L) \leq \nu/\epsilon + \nu + 1, \text{or} \tag{6}$$

- *the "$\perp$" symbol indicating that there is no separation triple $(L, S, R), x \in L$ such that*

$$|S| \leq k \quad \text{and} \quad \text{vol}^{out}_G(L) \leq \nu. \tag{7}$$

By setting $\epsilon = 1/(2k)$, we get the exact version for the size of vertex-cut. Observe that Equation (6) is changed to $|S| \leq (1 + 1/(2k))k = k + 1/2$. So $|S| \leq k$ since $|S|$ and k are integers.

Corollary 4.2. *There is an algorithm that takes as input a pointer to any vertex $x \in V$ in an adjacency list representing a strongly-connected directed graph $G = (V, E)$, positive integer ν ("target volume"), and positive integer k("target x-vertex-cut size") satisfying Equation (4), or Equation (5) where $\epsilon = 1/(2k)$, and in $\tilde{O}(\nu^{3/2}k)$ time outputs either*

- *a vertex cut S corresponding to the separation triple $(L, S, R), x \in L$ such that*

$$|S| \leq k \quad \text{and} \quad \text{vol}^{out}_G(L) \leq 2\nu k + \nu + 1, \text{or} \tag{8}$$

- *the "$\perp$" symbol indicating that there is no separation triple $(L, S, R), x \in L$ such that*

$$|S| \leq k \quad \text{and} \quad \text{vol}^{out}_G(L) \leq \nu. \tag{9}$$

The rest of this section is devoted to proving the above theorem. For the rest of this section, fix x, ν, k and ϵ as in the theorem statement.

4.1 Augmented Graph and Properties

Definition 4.3 (Augmented Graph G')**.** Given a directed uncapacitated graph $G = (V, E)$, we define a directed capacitated graph $(G', c_{G'}) = ((V', E'), c_{G'})$ where

$$V' = V_{\text{in}} \sqcup V_{\text{out}} \sqcup \{s, t\} \quad \text{and} \quad E' = E_\nu \sqcup E_\infty \sqcup E_{\deg} \sqcup \{(s, x_{\text{out}})\}, \tag{10}$$

where $\sqcup$ denotes disjoint union of sets, s and t are additional vertices not in G, and sets in Equation (10) are defined as follows.

- For each vertex $v \in V \setminus \{x\}$, we create vertex v_{in} in set V_{in} and v_{out} in set V_{out}. For the vertex x, we add only x_{out} to V_{out}.
- $E_\nu = \{(v_{\text{in}}, v_{\text{out}}) \colon v \in V \setminus \{x\}\}$.
- $E_\infty = \{(v_{\text{out}}, w_{\text{in}}) \colon (v, w) \in E\}$.
- $E_{\deg} = \{(v_{\text{out}}, t) \colon v \in V_{\text{out}}\}$.

Finally, we define the capacity function $c_{G'} : E' \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ as:

$$c_{G'}(e) = \begin{cases} \nu/(\epsilon k) & \text{if } e = (v_{\text{in}}, v_{\text{out}}) \in E_\nu \\ \deg^{\text{out}}_G(v) & \text{if } e = (v_{\text{out}}, t) \in E_{\deg} \\ \nu/\epsilon + \nu + 1 & \text{if } e = (s, x_{\text{out}}) \\ \infty & \text{otherwise} \end{cases}$$

Lemma 4.4. *Let C^* be the minimum-capacity (s, t)-cut in G'. Recall that $c_{G'}(C^*)$ is its capacity and ν and k satisfy Equation (4) or Equation (5) .*

(I) If there exists a separation triple $(L, S, R), x \in L$ in G satisfying Equation (7), then $c_{G'}(C^*) \leq \nu/\epsilon + \nu$.

(II) If $c_{G'}(C^*) \leq \nu/\epsilon + \nu$, then there exists a separation triple $(L, S, R), x \in L$ in G satisfying Equation (6).

We prove Lemma 4.4 in the rest of this subsection.

We define useful notations. For $U \subseteq V$ in G, define $V_{\text{out}}(U) = \{v_{\text{out}} \mid v \in U\} \subseteq V_{\text{out}}$ in G'. Similarly, we define $V_{\text{in}}(U) = \{v_{\text{in}} \mid v \in U\} \subseteq V_{\text{in}}$ in G' .

We first introduce a standard *split graph SG* from G'.

Definition 4.5 (Split graph SG)**.** Given G', a *split graph SG* is an induced graph $SG = G'[W]$ where

$$W = V_{\text{in}} \sqcup V_{\text{out}} \sqcup \{x\},$$

with capacity function $c'_G(e)$ restricted to edges in $G'[W]$ where the edge set of $G'[W]$ is $E_\nu \sqcup E_\infty$.

Proof of Lemma 4.4(I). We fix a separation triple (L, S, R) given in the statement. Since $x \in L$, S is an (x, y)-vertex-cut for some $y \in R$ by Definition 3.6.

Let $C = \{(u_{\text{inn}}, u_{\text{out}}) \colon u \in S\}$. It is easy to see that C is an $(x_{\text{out}}, y_{\text{in}})$-edge-cut in the split graph SG. In G', we define an edge-set $C' = C \sqcup \{(v, t) \mid v \in V_{\text{out}}(L)\}$. It is easy to see that C' is an (s, t)-edge-cut in G'.

We now compute the capacity of the cut C'.

$$\begin{aligned} c_{G'}(C') &= c_{G'}(C \sqcup \{(v, t) \colon v \in V_{\text{out}}(L)\}) \\ &= c_{G'}(C) + c_{G'}(\{(v, t) \colon v \in V_{\text{out}}(L)\}) \\ &= \nu|S|/(\epsilon k) + \sum_{v \in S} \deg^{\text{out}}_G(v) \\ &= \nu|S|/(\epsilon k) + \text{vol}^{\text{out}}_G(S) \\ &\leq \nu/\epsilon + \nu \end{aligned}$$

The last inequality follows from $|S| \leq k$ and $\text{vol}^{\text{out}}_G(S) \leq \nu$.

Hence, the capacity of the minimum (s, t)-cut C^* is $c_{G'}(C^*) \leq c_{G'}(C') \leq \nu/\epsilon + \nu$. □

Before proving Lemma 4.4(II), we observe structural properties of an (s, t)-edge-cut in G'.

Definition 4.6. Let $\mathcal{C}$ be the set of (s, t)-cuts of finite capacities in G'. We define three subsets of $\mathcal{C}$ as,

- $\mathcal{C}_1 = \{C : C \in \mathcal{C}$, and one side of vertices in $G' \setminus C$ contains s or t as a singleton $\}$.
- $\mathcal{C}_2 = \{C : C \in \mathcal{C} \setminus \mathcal{C}_1$, and C is an $(\{s\} \sqcup V_{\text{in}}, \{t\})$-edge-cut$\}$.
- $\mathcal{C}_3 = \{C : C \in \mathcal{C} \setminus \mathcal{C}_1$, and C is an $(\{s\}, \{v_{\text{in}}, t\})$-edge-cut for some $v_{\text{in}} \in V_{\text{in}}\}$.

Observe that three partitions in Definition 4.6 formed a complete set $\mathcal{C}$ and are pairwise disjoint by Definition 3.3, and by the construction of G'.

Observation 4.7.

$$\mathcal{C} = \mathcal{C}_1 \sqcup \mathcal{C}_2 \sqcup \mathcal{C}_3$$

Proposition 4.8. *We have the following lower bounds on cut capacity for cuts in $\mathcal{C}_1 \sqcup \mathcal{C}_2$.*

- *For all $C \in \mathcal{C}_1$, $c_{G'}(C) \geq \min(\nu/\epsilon + \nu + 1, m)$*
- *For all $C \in \mathcal{C}_2$, $c_{G'}(C) \geq \min(\nu/\epsilon + \nu + 1, \max((n - (1+\epsilon)k)k, m - (1+\epsilon)nk))$*

Corollary 4.9. *For all $C \in \mathcal{C}$, if $c_{G'}(C) \leq \nu/\epsilon + \nu$, then $C \in \mathcal{C}_3$*

We now ready to prove Lemma 4.4(II).

Proof of Lemma 4.4(II). In G, we show the existence of a separation triple (L, S, R) where $x \in L, |S| \leq (1+\epsilon)k$.

The minimum (s,t)-cut in G', C^*, is an $(s, \{v_{\text{in}}, t\})$-edge-cut (with finite capacity) for some $v_{\text{in}} \in V_{\text{in}}$. Since $c_{G'}(C^*) \leq \nu/\epsilon + \nu$, by Corollary 4.9, $C^* \in \mathcal{C}_3$.

We can write $C^* = E^*_{\text{deg}} \sqcup E^*_v$ where $\emptyset \neq E^*_{\text{deg}} \subsetneq E_{\text{deg}}$ and $\emptyset \neq E^*_v \subsetneq E_v$ in G'. To see that $E^*_v \neq \emptyset$, suppose otherwise, then C^* must be in $\mathcal{C}_1$, a contradiction.

It is easy to see that E^*_v is an $(x_{\text{out}}, v_{\text{in}})$-edge-cut in SG.

To show a separation triple (L, S, R), it is enough to define S, and show that S is an (x, y)-vertex-cut where $x \in L$ and $y \in R$. This is because L and R can be found trivially when we remove S from G.

Let $S = \{u \in V : (u_{\text{in}}, u_{\text{out}}) \in E^*_v\}$. It is easy to see that S is an (x, y)-vertex-cut in G for some $y \in V$.

Next, $|S| \leq (1+\epsilon)k$ since otherwise $c_{G'}(C^*) > (1+\epsilon)k(\nu/(\epsilon k)) = \nu/\epsilon + \nu$, a contradiction to the capacity of C^*.

It is easy to see that $\text{vol}^{\text{out}}_G(L) \leq \nu/\epsilon + \nu + 1$. This follows from the in-flow is at most $\nu/\epsilon + \nu + 1$.

□

4.2 Preliminaries for Flow Network and Binary Blocking Flow

We define notations related flows on a capacitated directed graph $G = (V, E, c)$. We fix vertices s as source and t as sink.

Definition 4.10 (Blocking flow)**.** Given a capacitated graph $G = (V, E, c)$, a *blocking flow* is a flow that saturates at least one edge on every (s,t)-path in G.

We will use Definition 4.10 mostly on the residual graph G_f.

Given a binary length function ℓ on (G, c, f), we define a natural distance function to each vertex in (G, c, f) under ℓ. $d(v)$ is the length of the shortest (s, v)-path in G_f under the binary length function ℓ.

For any $(v, w) \in E_f$, $d_\ell(v) + \ell(v, w) \geq d_\ell(w)$. If $d_\ell(v) + \ell(v, w) = d_\ell(w)$, then we call (v, w) *admissible edge* under length function ℓ.

We denote E_a to be the set of admissible edges of E_f in (G, c, f) under length function ℓ.

Definition 4.11 (Admissible graph)**.** Given a residual graph (G, c, f), and a length function function ℓ, we define an *admissible graph* $A(G, c, f, \ell) = (G[E_a], c, f)$ to be an induced subgraph of (G, c, f) that contains only admissible edges under length function ℓ.

Definition 4.12 (Δ'-or-blocking flow)**.** For any $\Delta' > 0$, a flow is called a *Δ'-or-blocking flow* if it is a flow of value exactly Δ', or a blocking flow.

Definition 4.13 (Binary length function $\tilde{\ell}$)**.** Given $\Delta > 0$, a capacitated graph (G, c) and a flow f, we define binary length functions $\hat{\ell}$ and $\tilde{\ell}$ for any edge (u, v) in a residual graph (G, c, f) as follows.

$$\hat{\ell}(u, v) = \begin{cases} 0 & \text{if residual capacity } c(u,v) - f(u,v) \geq \Delta \\ 1 & \text{otherwise} \end{cases}$$

Let $\hat{d}(v)$ be the shortest path distance between s and v under the length function $\hat{\ell}$. We define *special edge* (u, v) to be an edge (u, v) such that $\hat{d}(u) = \hat{d}(v), \Delta/2 \leq c(u, v) - f(u, v) < \Delta$, and $c(v, u) - f(v, u) \geq \Delta$. We define the next length function $\tilde{\ell}$.

$$\tilde{\ell}(u, v) = \begin{cases} 0 & \text{if } (u, v) \text{ is special} \\ \hat{\ell}(u, v) & \text{otherwise} \end{cases}$$

Lemma 4.14 ([19])**.** *Let $A(G, c, f, \ell)$ be an admissible graph and m_A be its number of edges. Then, there exists an algorithm that takes as input A and $\Delta > 0$, and in $O(m_A \log(m_A))$ time, outputs a $\Delta/4$-or-blocking flow. We refer the algorithm as* BinaryBlockingFlow$(A(G, c, f, \ell), \Delta)$.

We now define the notion of *shortest-path flow*. Intuitively, it is a union of shortest paths on admissible graphs. This is the flow resulting from, e.g., the Binary Blocking Flow algorithm [19].

Definition 4.15 (Shortest-path flow)**.** Given a graph (G, c) with a flow f, and length function ℓ, and let G_f be the residual graph. A flow f^* in G_f is called *shortest-path flow* if it can be decomposed into a set of shortest paths under length function ℓ, i.e., $f^* = \sum_{i=1}^{b} f^*_i$ for some integer $b > 0$ where support(f^*_i) is a shortest-path in G_f under length function ℓ.

Observe that BinaryBlockingFlow$(A(G, c, f, \ell), \Delta)$ always produces a shortest-path flow.

From the rest of this section, we fix an augmented graph $(G', c_{G'})$ (Definition 4.3), and also a flow f.

Given residual graph G'_f, and d_ℓ, we can use BinaryBlockingFlow$(A(G', c_{G'}, f, \tilde{\ell}), \Delta)$ to compute a $\Delta/4$-or-binary blocking flow in $(G', c_{G'}, f)$ in $\tilde{O}(m)$ time.

[35] provide a slightly different binary length function such that the algorithm in [19] has local running time.

Our goal in next section is to output the same $\Delta/4$-or-binary blocking flow in G'_f in $\tilde{O}(\nu k)$ time using a slight adjustment from [35].

4.3 Local Augmented Graph and Binary Blocking Flow in Local Time

The goal in this section is to compute binary blocking flow on the residual graph of the augmented graph $(G', c_{G'})$ with a flow f in "local" time. To ensure local running time, we cannot construct the augmented graph G' explicitly. Instead, we compute binary blocking flow from a subgraph of G' based on "absorbed" vertices.

Definition 4.16 (Split-node-saturated set)**.** Given a residual graph $(G', c_{G'}, f)$, let B_{out} be the set of vertices $v \in V_{\text{out}} \sqcup \{x\}$ in the residual graph $(G', c_{G'}, f)$ whose edge to t is saturated. The *split-node-saturated set* B is defined as:

$$B = B_{\text{out}} \sqcup N^{\text{out}}_{G'}(B_{\text{out}}) \setminus \{t\}$$

Note that x is a fixed vertex as in Definition 4.3.

Definition 4.17 (Local binary length function)**.** Fix a parameter $\Delta > 0$ to be selected, let $\tilde{\ell}$ be the length function in Definition 4.13 for the residual graph $(G', c_{G'}, f)$. For vertex u, v in the residual graph, if $u, v \in B$, we call residual edge (u, v) *modern*. Otherwise, we call residual edge (u, v) *classical*.

We define *local binary length* function ℓ:

$$\ell(u,v) = \begin{cases} 1 & \text{if } (u,v) \text{ is classical} \\ \tilde{\ell}(u,v) & \text{otherwise} \end{cases}$$

Definition 4.18 (Distance under local binary length ℓ)**.** Define distance function $d(v)$ as the shortest path distance between the source vertex s and vertex v in the residual graph $(G', c_{G'}, f)$ under the local length function ℓ.

The following obsevations about structural properties of the residual graph G'_f follows immediately from the definition of local length function ℓ.

Observation 4.19. *For a given residual graph $(G', c_{G'}, f)$,*

- *for any residual edge $(u, v) \in E_{\infty,f}$ that is modern, $\ell(u, v) = 0$.*
- *for any residual edge $(u, v) \in E_{\deg,f} \sqcup (s, x)$, (u, v) is classical.*
- *any residual edge with length zero is modern.*

Definition 4.20 (Layers)**.** Given distance function d on residual graph $(G', c_{G'}, f)$, define $L_j = \{v \in G' : d(v) = j\}$ to be the set of j^{th}–layer with respect to distance d. Define $d_{\max} = d(t)$ to be distance between s and t in $(G', c_{G'}, f)$.

The proof of the following Lemma is similar to that from [35], but we focus on the augmented graph $(G', c_{G'}, f)$. Recall split-node-saturated set B from Definition 4.16. The proof is in the full version.

Lemma 4.21. *If $d_{max} < \infty$ and (x, t) is saturated, then we have:*

(I) $d_{max} \geq 3$.
(II) $L_0 = \{s\}$.
(III) $L_j \subseteq B$ *for* $1 \leq j \leq d_{max} - 2$.
(IV) $L_j \subseteq B \cup N^{\text{out}}_{G'}(B)$ *for* $j = d_{max} - 1$.

Definition 4.22 (Local graph, LG)**.** Given the augmented graph $G' = (V', E')$ and split-node-saturated set B, we define the *local graph* $LG(G', B) = G'[V''] = (V'', E'')$ as an induced subgraph of G' where

$$V'' = B \sqcup N^{\text{out}}_{G'}(B) \sqcup \{s, t\} \quad \text{and} \quad E'' = E''_v \sqcup E''_\infty \sqcup E''_{\deg} \sqcup \{(s, x)\} \tag{11}$$

where the sets in Equation (11) are defined as follows.

- $E''_v = \{(v_{\text{in}}, v_{\text{out}}) : v_{\text{out}} \in B_{\text{out}} \sqcup N^{\text{out}}_{G'}(B), (v_{\text{in}}, v_{\text{out}}) \in E_v\}$.
- $E''_\infty = \{(v_{\text{out}}, w_{\text{in}}) : v_{\text{out}} \in B_{\text{out}}, w_{\text{in}} \in V', (v_{\text{out}}, w_{\text{in}}) \in E_\infty\}$.
- $E''_{\deg} = \{(v_{\text{out}}, t) : v_{\text{out}} \in B_{\text{out}} \sqcup N^{\text{out}}_{G'}(B)\}$.

Using the same capacity and flow as in G', the *residual local graph* is $(LG(G', B), c_{LG}, f_{LG})$ where c_{LG} and f_{LG} are the same as $c_{G'}$ and $f_{G'}$, but restricted to the edges in $LG(G', B)$. The local length function ℓ also applies to $LG(G', B)$.

Lemma 4.23. *Let m' be the number of edges in $LG(G', B)$, and $n' = |V''|$ be the number of vertices in $LG(G', B)$. We have*

$$m' \leq 4v/\epsilon \quad \text{and} \quad n' \leq 8v/(\epsilon k).$$

The proof of the following Lemma is a straightforward modification from [35].

Lemma 4.24. *Given the local length function ℓ on both residual augmented graph $(G', c_{G'}, f)$ and residual local graph $(LG, c_{LG}, f_{LG}) = (V'', E''_f, c_{LG,f})$ (Recall f_{LG} from Definition 4.22). Let f_1 be the output of* BinaryBlockingFlow$(A(G', c_{G'}, f, \ell), \Delta)$. *Let f_2 be the output of*
BinaryBlockingFlow$(A(LG, c_{LG}, f_{LG}, \ell), \Delta)$. *Then,*

- $f_1 = z(f_2)$ *where*

$$z(f_2)(e) = \begin{cases} 0 & \text{if } e \notin E''_f. \\ f_2(e) & \text{otherwise} \end{cases}$$

  i.e., f_1 and f_2 coincide.
- BinaryBlockingFlow$(A(LG, c_{LG}, f_{LG}, \ell), \Delta)$ *takes $\tilde{O}(v/\epsilon)$ time.*

4.4 Local Goldberg-Rao's Algorithm for Augmented Graph

Theorem 4.25. *Given graph G, we can compute the (s, t) max-flow in G' in $\tilde{O}(v^{3/2}/(\epsilon^{3/2}\sqrt{k}))$ time.*

Algorithm 1: LocalFlow(G, x, v, k)

Input: $x \in V, v, k$
Output: maximum (s, t)-flow and its corresponding minimum (s, t)-edge-cut in G'

Let G' be an *implicit* augmented graph on G. // No need to construct explicitly.
$\Lambda \leftarrow \sqrt{8v/(\epsilon k)}$
$F \leftarrow 2vk + v + 1 - \deg^{\text{out}}_G(x)$ // F is an upper bound on (s, t)-flow value in G'.
if $F \leq 0$ **then** the minimum (s, t)-edge-cut is (s, x), and return.
$f \leftarrow$ a flow of value $\deg^{\text{out}}_G(x)$ through $s - x - t$ path.
$B \leftarrow \{x\} \sqcup N^{\text{out}}_{G'}(x)$ // a set of saturated vertices and out-neighbors.
while $F \geq 1$ **do**
  $\Delta \leftarrow F/(2\Lambda)$
  for $i \leftarrow 1$ **to** 5Λ **do**
    $LG \leftarrow$ local subgraph of G' given B. // see Definition 4.22
    $\ell \leftarrow$ local length function on current flow f.
    $f \leftarrow f +$ BinaryBlockingFlow$(A(LG, c_{LG}, f, \ell), \Delta)$.
    $C \leftarrow$ vertices in $N^{\text{out}}_{G'}(B)$ whose edges to sink are saturated in the new flow.
    $B \leftarrow B \sqcup C \sqcup N^{\text{out}}_{G'}(C)$
  $F \leftarrow F/2$
return *maximum (s, t)-flow f and its corresponding minimum (s, t)-edge-cut A in G'.*

Correctness. We show that F is the upper bound on the maximum flow value in G'_f. We use induction on inner loop. Before entering the inner loop for the first time, F is set to be the value of (s,t) edge minus $\deg^{\text{out}}_G(x)$. Since F is positive, then G_f has valid maximum flow upper bound F. Now, we consider the inner loop. After 5Λ times, either

- we find a flow of value $\Delta/4$ at least 4Λ times, or
- we find a blocking flow at least Λ times.

If the first case holds, then we increase the flow by at least $\geq (\Delta/4)(4\Lambda) = F/2$. Hence, the flow $F/2$ is the valid upper bound. For the second case, we need the following Lemma whose proof is essentially the same as the original proof of Goldberg-Rao's algorithm [19]:

Lemma 4.26. *A flow augmentation does not decrease the distance $d(t)$. On the other hand, a blocking flow augmentation strictly increases $d(t)$.*

If the second case holds, we claim:

Claim 4.27. *If we find a blocking flow at least Λ times, then there exists an (s,t)-edge cut of capacity at most $\Delta\Lambda = F/2$, which is an upper bound of the remaining flow to be augmented.*

The correctness follows since at the end of the loop we have $F < 1$.

Running Time. By Lemma 4.24, we can compute Δ-blocking flow in LG with local binary length function ℓ in $\tilde{O}(\nu/\epsilon)$ time. The time already includes the time to read LG. The number of such computations is $O(\Lambda \log(\nu/\epsilon)) = O(\sqrt{\nu/(\epsilon k)}\log(m)) = \tilde{O}(\sqrt{\nu/(\epsilon k)})$. So the total running time is $\tilde{O}(\nu^{3/2}/(\epsilon^{3/2}k^{1/2}))$. This completes the proof of Theorem 4.25.

4.5 Proof of Theorem 4.1

Proof of Theorem 4.1. Given G, x, ν, k, ϵ, by Theorem 4.25, we compute the minimum (s,t)-edge-cut C^* in G' in $\tilde{O}(\nu^{3/2}/(\epsilon^{3/2}k^{1/2})$ time. If the edge-cut C^* has capacity $> \nu/\epsilon + \nu$, then by Lemma 4.4(I), we can output $\perp$. Otherwise, C^* has capacity at most $\nu/\epsilon + \nu$, by Lemma 4.4(II), we can output the separation triple (L,S,R) with the properties in Lemma 4.4(II). □

5 VERTEX CONNECTIVITY VIA LOCAL VERTEX CONNECTIVITY

Theorem 5.1 (Exact vertex connectivity). *There exist randomized (Monte Carlo) algorithms that take as inputs a graph G, integer $0 < k < O(\sqrt{n})$, and in $\tilde{O}(m + k^{7/3}n^{4/3})$ time for undirected graph (and in $\tilde{O}(\min(km^{2/3}n, km^{4/3}))$ time for directed graph) can decide w.h.p. if $\kappa_G \geq k$. If $\kappa_G < k$, then the algorithms also return the corresponding vertex-cut.*

We define the function $T(k,m,n)$ as

$$T(k,m,n) = \begin{cases} \min(m^{4/3}, nm^{2/3}k^{1/2}, \\ \quad mn^{2/3+o(1)}/k^{1/3}, \\ \quad n^{7/3+o(1)}/k^{1/6}) & \text{if } k \leq n^{4/5}, \\ \quad n^{3+o(1)}/k & \text{if } k > n^{4/5}. \end{cases} \tag{12}$$

Theorem 5.2 (Approximate vertex connectivity). *There exist randomized (Monte Carlo) algorithms that take as inputs a graph G, an positive integer k, and positive real $\epsilon < 1$, and in $\tilde{O}(m + \text{poly}(1/\epsilon)\min(k^{4/3}n^{4/3}, k^{2/3}n^{5/3+o(1)}, n^{3+o(1)}/k))$ time for undirected graph (and in $\tilde{O}(\text{poly}(1/\epsilon)T(k,m,n))$ time for directed graph where $T(k,m,n)$ is defined [7] as in Equation (12)) w.h.p. return a vertex-cut with size at most $(1+O(\epsilon))\kappa_G$ or cerify that $\kappa_G \geq k$.*

This section is devoted to proving Theorem 5.1. and Theorem 5.2.

5.1 Vertex Connectivity Algorithms

We describe the algorithm in the generic form as in Algorithm 2.

5.2 Correctness

We can compute approximate vertex connectivity by standard binary search on k with the decision problem. We focus on correctness of Algorithm 2 for approximate version. For exact version, the same proof goes through when we use $\epsilon = 1/(2k)$, and $\kappa_G \leq \sqrt{n}/2$. Let $\Delta = \min(n/(1+\epsilon), (m/(1+\epsilon))^{1/2})$. For the purpose of analysis of the decision problem, we assume the followings.

Assumption 5.3. *If k is specified in Algorithm 2, then*

(I) $\deg^{\text{out}}_{\min} \geq k$.

(II) $k \leq \Delta$. *We use $k \leq \sqrt{n}/2$ for exact vertex connectivity.*

(III) Local conditions in Theorem 4.1 are satisfied. We use exact version of local conditions for exact vertex connectivity.

5.2.1 High Vertex Connectivity.

Proposition 5.4. *If $\kappa_G \geq \Delta$, then $|\deg^{\text{out}}_{\min}| \leq (1+\epsilon)\kappa_G$.*

5.2.2 Edge-Sampling with LocalVC.

Lemma 5.5. *Algorithm 2 with edge-sampling, and* LocalVC *outputs correctly w.h.p. a vertex-cut of size $\leq (1+\epsilon)k$ if $\kappa_G \leq k$, and a symbol $\perp$ if $\kappa_G > k$.*

We describe notations regarding edge-sets from a separation triple (L,S,R) in G. Let $E^*(L,S) = E(L,L) \sqcup E(L,S) \sqcup E(S,L)$, and $E^*(S,R) = E(R,R) \sqcup E(S,R) \sqcup E(R,S)$.

Definition 5.6 (L-volume, and R-volume of the separation triple)**.** For a separation triple (L,S,R), we denote $\text{vol}^*_G(L) = \sum_{v\in L} \deg^{\text{out}}_G(v) + |E(S,L)|$ and $\text{vol}^*_G(R) = \sum_{v\in R} \deg^{\text{out}}_G(v) + |E(S,R)|$.

It is easy to see that $\text{vol}^*_G(L) = |E^*(L,S)|$ and $\text{vol}^*_G(R) = |E^*(S,R)|$.

The following observations follow immediately from the definition of $E^*(L,S)$ and $E^*(S,R)$, and a separation triple (L,S,R).

Observation 5.7. *We can partition edges in G according to (L,S,R) separation triple as*

$$E = E^*(L,S) \sqcup E(S,S) \sqcup E^*(S,R)$$

And,

- *For any edge $(x,y) \in E^*(L,S)$, $x \in L$ or $y \in L$.*
- *For any edge $(x,y) \in E^*(S,R)$, $x \in R$ or $y \in R$.*

Furthermore,

$$m = \text{vol}^*_G(L) + |E(S,S)| + \text{vol}^*_G(R)$$

We proceed the proof. There are three cases for the set of all separation triples in G. The first case is there exists a separation triple (L,S,R) such that $|S| \leq k, \text{vol}^*_G(L) \geq a, \text{vol}^*_G(R) \geq a$ We show that w.h.p. Algorithm 2 outputs a vertex-cut of size at most $(1+\epsilon)k$.

[7] The term $m^{4/3}$ appears when $m \geq k^3$.

Algorithm 2: VC(Sampling method, LocalVC, $\kappa(x,y)$; G, k, a, ϵ)

Input: Sampling method, LocalVC, G = (V,E), k, a, ϵ
Output: a vertex-cut U such that $|U| \le k$ or a symbol $\perp$.

If undirected, replace $E = \{(u,v),(v,u) : (u,v) \in E(H_{k+1})\}$ where H_{k+1} as in Theorem 3.9.
if Sampling method = vertex **then**
  for $i \leftarrow 1$ **to** $n/(\epsilon a)$ (use n/a for exact version) **do**
    Sample a random pair of vertices $x, y \in V$.
    if k is not specified **then** compute approximate $\kappa_G(x,y)$.
    if $\kappa_G(x,y) \le (1+\epsilon)k$ **then**
      return the corresponding (x,y)-vertex-cut U.
if Sampling method = edge **then**
  for $i \leftarrow 1$ **to** $m/(\epsilon a)$ (use m/a for exact version) **do**
    Sample a random pair of edges $(x_1,y_1),(x_2,y_2) \in E$.
    if k is not specified **then**
      compute approximate $\kappa_G(x_1,y_2), \kappa_G(x_1,x_2), \kappa_G(y_1,x_2), \kappa_G(y_1,y_2)$.
    if $\min(\kappa_G(x_1,y_2), \kappa_G(x_1,x_2), \kappa_G(y_1,x_2), \kappa_G(y_1,y_2)) \le (1+\epsilon)k$ **then**
      return the corresponding (x,y)-vertex-cut U.
if LocalVC is not specified **then**
  Let x^*, y^* be vertices with minimum $\kappa_G(x^*,y^*)$ computed so far.
  Let W be the vertex-cut corresponding to $\kappa_G(x^*,y^*)$
  Let $v_{\min}, u_{\min}$ be the vertex with the minimum out-degree in G and G^R respectively.
  return The smallest set among $\{W, N^{\text{out}}_G(v_{\min}), N^{\text{out}}_{G^R}(u_{\min})\}$.
Let $\mathcal{L} = \{2^\ell : 1 \le \ell \le \lceil \log_2 a \rceil, \text{and } \ell \in \mathbb{Z}\}$.
if Sampling method = vertex **then**
  for $s \in \mathcal{L}$ **do**
    for $i \leftarrow 1$ **to** n/s **do**
      Sample a random vertex $x \in V$.
      Let $v \leftarrow O(s(s+k))$.
      if LocalVC(G,x,v,k,ϵ) or LocalVC(G^R,x,v,k,ϵ) outputs a vertex-cut U **then**
        return U.
if Sampling method = edge **then**
  for $s \in \mathcal{L}$ **do**
    for $i \leftarrow 1$ **to** m/s **do**
      Sample a random edge $(x,y) \in E$.
      Let $v \leftarrow O(s)$, and $\mathcal{G} = \{G, G^R\}$.
      for $H \in \mathcal{G}, z \in \{x,y\}$ **do**
        if LocalVC(H,z,v,k,ϵ) outputs a vertex-cut U. **then**
          return U.
return $\perp$.

Lemma 5.8. *If G has a separation triple (L,S,R) such that $|S| \le k, \text{vol}^*_G(L) \ge a, \text{vol}^*_G(R) \ge a$, then w.h.p. Algorithm 2 outputs a vertex-cut of size at most $(1+\epsilon)k$.*

The second case is there exists a separation triple (L,S,R) such that $|S| \le k$ and $\text{vol}^*_G(L) < a$ or $\text{vol}^*_G(R) < a$. We show that w.h.p. Algorithm 2 outputs a vertex-cut of size at most $(1+\epsilon)k$.

Lemma 5.9. *If G has a separation triple (L,S,R) such that $|S| \le k$ and $\text{vol}^*_G(L) < a$ or $\text{vol}^*_G(R) < a$, then w.h.p. Algorithm 2 outputs a vertex-cut of size at most $(1+\epsilon)k$.*

The final case is when every separation triple (L,S,R) in G, $|S| > k$. In other words, $\kappa_G > k$. If Algorithm 2 outputs a vertex-cut, then it is a $(1+\epsilon)$-approximate vertex-cut. Otherwise, Algorithm 2 outputs $\perp$ correctly.

5.2.3 *Vertex-Sampling with LocalVC.*

Lemma 5.10. *Algorithm 2 with vertex-sampling, and* LocalVC *outputs correctly w.h.p. a vertex-cut of size $\le (1+\epsilon)k$ if $\kappa_G \le k$, and a symbol $\perp$ if $\kappa_G > k$.*

5.3 Running Time

Let $T_1(m,n,k,\epsilon)$ be the time for deciding if $\kappa(x,y) \le (1+\epsilon)$, $T_2(v,k,\epsilon)$ be the running time for approximate LocalVC, and $T_3(m,n,\epsilon)$ be the time for computing approximate $\kappa(x,y)$. If G is undirected, we can replace m with nk with additional $O(m)$ preprocessing time. The running time for exact version is similar.

5.3.1 *Edge-Sampling with LocalVC.*

Lemma 5.11. *Algorithm 2 with edge-sampling, and* LocalVC *terminates in time*

$$\tilde{O}((m/(\epsilon a))(T_1(m,n,k,\epsilon) + T_2(a,k,\epsilon))).$$

5.3.2 *Vertex-Sampling with LocalVC.*

Lemma 5.12. *Algorithm 2 with vertex-sampling, and* LocalVC *terminates in time*

$$\tilde{O}((n/(\epsilon a))(T_1(m,n,k,\epsilon) + T_2(a^2 + ak,k,\epsilon))).$$

5.3.3 *Vertex-Sampling without LocalVC.*

Lemma 5.13. *Algorithm 2 with vertex-sampling without* LocalVC *terminates in time*

$$\tilde{O}(n/(\epsilon^2 k) T_3(m,n,\epsilon)).$$

Proof. The running time follows from the first loop where we set a such that the number of sample is $n/(\epsilon^2 k)$, and computing approximate $\kappa(x,y)$ can be done in $T_3(m,n,\epsilon)$ time. □

5.4 Proof of Theorems 5.1 and 5.2

For exact vertex connectivity, LocalVC runs in $v^{1.5}k$ time by Corollary 4.2. We can decide $\kappa(x,y) \le k$ in $O(mk)$ time.

For undirected exact vertex connectivity where $k < O(\sqrt{n})$, we first sparsifiy the graph in $O(m)$ time. Then, we use edge-sampling with LocalVC algorithm where we set $a = m'^{2/3}$, where $m' = O(nk)$ is the number of edges of sparsified graph.

For directed exact vertex connectivity where $k < O(\sqrt{n})$, we use edge-sampling with LocalVC algorithm where we set $a = m^{2/3}$

if $m < n^{3/2}$. If $m > n^{3/2}$, we use vertex-sampling with LocalVC algorithm where we set $a = m^{1/3}$.

For approximate vertex connectivity, approximate LocalVC runs in $\text{poly}(1/\epsilon)v^{1.5}/\sqrt{k}$ by Theorem 4.1. Also, we can decide $\kappa(x,y) \le (1+O(\epsilon))k$ or cerify that $\kappa \ge k$ in time $\tilde{O}(\text{poly}(1/\epsilon)\min(mk, n^{2+o(1)}))$. The running time $\text{poly}(1/\epsilon)n^{2+o(1)}$ is due to [9].

For undirected approximate vertex connectivity, we first sparsify the graph in $O(m)$ time. Let m' be the number of edges of the sparsified graph. For $k < n^{0.8}$, we use edge-sampling with approximate LocalVC algorithm where we set $a = m^{\hat{a}}$, where $\hat{a} = \frac{\min(5\hat{k}+2, \hat{k}+4)}{3\hat{k}+3}$, and $\hat{k} = \log_n k$. For $k > n^{0.8}$, we use vertex-sampling without LocalVC.

6 $(1+\epsilon)$-APPROXIMATE VERTEX CONNECTIVITY VIA CONVEX EMBEDDING

Theorem 6.1. *There exists an algorithm that takes G and $\epsilon > 0$, and in $O(n^\omega/\epsilon^2 + \min(\kappa_G, \sqrt{n})m)$ time outputs a vertex-cut U such that $|U| \le (1+\epsilon)\kappa$.*

6.1 Preliminaries

Definition 6.2 (Pointset in $\mathbb{F}^k$)**.** Let $\mathbb{F}$ be any field. For $k \ge 0$, $\mathbb{F}^k$ is k-dimensional linear space over $\mathbb{F}$. Denote $X = \{x_1, \ldots, x_n\}$ as a finite set of points in $\mathbb{F}^k$. The *affline hull* of X is $\text{aff}(X) = \{\sum_{i=1}^k c_i x_i \mid x_i \in X \text{ and } \sum_{i=1}^k c_i = 1\}$. The rank of X denoted as $\text{rank}(X)$ is one plus dimension of $\text{aff}(X)$. In particular, if $\mathbb{F} = \mathbb{R}$, then we will consider the *convex hull* of X, denoted as $\text{conv}(X)$.

For any sets V, W, any funtion $f : V \to W$, and any subset $U \subseteq V$, we denote $f(U) = \{f(u) \colon u \in U\}$.

Definition 6.3 (Convex directed X-embedding)**.** For any $X \subset V$, a convex directed X-embedding of a graph $G = (V, E)$ is a function $f : V \to \mathbb{R}^{|X|-1}$ such that for each $v \in V \setminus X$, $f(v) \in \text{conv}(f(N_G^{\text{out}}(v)))$.

Definition 6.4. For $X, Y \subseteq V$, $p(X, Y)$ is the maximum number of vertex-disjoint paths from X to Y where different paths have different end points.

Lemma 6.5. *For any non-empty subset $U \subseteq V \setminus X$, w.h.p. a random modular directed X-embedding $f : V \to \mathbb{Z}_p^{|X|-1}$ satisfies $\text{rank}(f(U)) = p(U, X)$.*

Definition 6.6 (Fixed k-neighbors)**.** For $v \in V$, let $N_{G,k}^{\text{out}}(v)$ be a fixed, but arbitrarily selected subset of $N_G^{\text{out}}(v)$ of size k. Similarly, For $v \in V$, let $N_{G,k}^{\text{in}}(v)$ be a fixed, but arbitrarily selected subset of $N_G^{\text{in}}(v)$ of size k.

Lemma 6.7. *Let ω be the exponent of the running time of the optimal matrix multiplication algorithm. Note it is known that $\omega \le 2.372$.*

- *For $y \in V$, a random modular directed $N_{G,k}^{\text{out}}(y)$-embedding f can be constructed in $O(n^\omega)$ time.*
- *Given such f, for $U \subseteq V$ with $|U| = k$, $\text{rank}(f(U))$ can be computed in $O(k^\omega)$ time.*

6.2 Algorithm

We present an approximation algorithm using convex embedding in Algorithm 3.

Algorithm 3: ApproxConvexEmbedding(G, ϵ)

Input: $G = (V, E)$, and $\epsilon > 0$
Output: A vertex-cut U such that w.h.p. $|U| \le (1+\epsilon)\kappa_G$.

```
Let k ← max(d^out_min, d^in_min).
Let k' ← min(d^out_min, d^in_min).
repeat
    Sample two random vertices x_2, y_1 ∈ V.
    Let f be a random modular directed N^in_{G,k}(y_1)-embedding.
      Let f^R be a random modular directed
      N^in_{G^R,k}(x_2)-embedding.
    repeat
        Sample two random vertices y_2, x_1 ∈ V.
        rank(x_1, y_1) ← rank(f(N^out_{G,k}(x_1)))   // O(k^ω) time.
        rank(x_2, y_2) ← rank(f^R(N^out_{G^R,k}(y_2)))
    until Θ(n/(εk')) times
until Θ(1/ε) times
Let x*, y* be the pair of vertices with minimum rank(x, y) for
  all x, y computed so far.
Let W ← min(κ_G(x*, y*), κ_{G^R}(x*, y*))
Let v_min, u_min be the vertex with the minimum out-degree in
  G and G^R respectively.
return min(W, |N^out_G(v_min)|, |N^out_{G^R}(u_min)|)
```

6.3 Analysis

Lemma 6.8. *Algorithm 3 outputs w.h.p. a vertex-cut U such that $|U| \le (1+\epsilon)\kappa_G$.*

Proof. Let $\tilde{\kappa}$ denote the answer of our algorithm. Clearly $\tilde{\kappa} \le d_{\min}^{\text{out}}$ and $\tilde{\kappa} \le d_{\min}^{\text{in}}$ by design. Observe also that $\tilde{\kappa} \ge \kappa$ because the answer corresponds to some vertex cut. Let (A, S, B) be the optimal separation triple where A is a out-vertex shore and $|S| = \kappa$. W.l.o.g. we assume that $|A| \le |B|$, another case is symmetric.

Suppose that $|A| \le \epsilon d_{\min}^{\text{out}}$. Then $\kappa = |S| \ge d_{\min}^{\text{out}} - \epsilon d_{\min}^{\text{out}} \ge \tilde{\kappa}(1-\epsilon) \ge \kappa(1-\epsilon)$. That is, $\tilde{\kappa}$ is indeed an $(1+O(\epsilon))$-approximation of κ in this case.

Suppose now that $|A| \ge \epsilon d_{\min}^{\text{out}}$. We claim that $|B| \ge \epsilon n/4$. Indeed, if $d_{\min}^{\text{out}} \ge n/2$, then $|B| \ge \epsilon n/2$. Else if, $d_{\min}^{\text{out}} \le n/2$, then we know $|S| = \kappa \le n/2$. But $2|B| \ge |A| + |B| = n - |S| \ge n/2$. In either case, $|B| \ge \epsilon n/4$.

Now, as $|B| \ge \epsilon n/4$ and we sample $\tilde{O}(1/\epsilon)$ many y_1. There is one sample $y_1 \in B$ w.h.p. and now we assume that $y_1 \in B$. In the iteration when y_1 is sampled. As $|A| \ge \epsilon d_{\min}^{\text{out}}$ and we sample at least $\tilde{O}(n/d_{\min}^{\text{out}}\epsilon)$ many x_1. There is one sample $x_1 \in A$ w.h.p.

By Lemma 6.5, w.h.p.,

$$\text{rank}(x_1, y_1) = \text{rank}(f(N_{G,k}^{\text{out}}(x_1)))$$
$$= p(N_{G,k}^{\text{out}}(x_1), N_{G,k}^{\text{in}}(y_1)) = \kappa(x_1, y_1) = \kappa.$$

So our answer $\tilde{\kappa} = \kappa$ in this case. □

Lemma 6.9. *Algorithm 3 terminates in $O(n^\omega/\epsilon^2 + \min(\kappa_G, \sqrt{n})m)$ time.*

7 OPEN PROBLEMS

(1) Is there an $O(vk)$-time LocalVC algorithm?
(2) Can we break the $O(n^3)$ time bound when $k = \Omega(n)$? This would still be hard to break even if we had an $O(vk)$-time LocalVC algorithm.
(3) Is there an $o(n^2)$-time algorithm for vertex-weighted graphs when $m = O(n)$? Our LocalVC algorithm does not generalize to the weighted case.
(4) Is there an $o(n^2)$-time algorithm for the single-source max-flow problem when $m = O(n)$?
(5) Is there a near-linear-time $o(\log n)$-approximation algorithm?
(6) How fast can we solve the vertex connectivity problem in the dynamic setting (under edge insertions and deletions) and the distributed setting (e.g. in the CONGEST model)?

ACKNOWLEDGEMENT

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme under grant agreement No 715672 and 759557. Nanongkai was also partially supported by the Swedish Research Council (Reg. No. 2015-04659.).

REFERENCES

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. 1974. *The Design and Analysis of Computer Algorithms.* Addison-Wesley.
[2] Reid Andersen, Fan R. K. Chung, and Kevin J. Lang. 2006. Local Graph Partitioning using PageRank Vectors. In *FOCS*. IEEE Computer Society, 475–486.
[3] Reid Andersen and Kevin J. Lang. 2008. An algorithm for improving graph partitions. In *SODA*. SIAM, 651–660.
[4] Reid Andersen and Yuval Peres. 2009. Finding sparse cuts locally using evolving sets. In *STOC*. ACM, 235–244.
[5] Michael Becker, W. Degenhardt, Jürgen Doenhardt, Stefan Hertel, Gerd Kaninke, W. Kerber, Kurt Mehlhorn, Stefan Näher, Hans Rohnert, and Thomas Winter. 1982. A Probabilistic Algorithm for Vertex Connectivity of Graphs. *Inf. Process. Lett.* 15, 3 (1982), 135–136.
[6] Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. 2014. Distributed connectivity decomposition. In *PODC*. ACM, 156–165.
[7] Joseph Cheriyan and John H. Reif. 1994. Directed *s-t* Numberings, Rubber Bands, and Testing Digraph *k*-Vertex Connectivity. *Combinatorica* 14, 4 (1994), 435–451. Announced at SODA'92.
[8] Joseph Cheriyan and Ramakrishna Thurimella. 1991. Algorithms for Parallel k-Vertex Connectivity and Sparse Certificates (Extended Abstract). In *STOC*. ACM, 391–401.
[9] Julia Chuzhoy and Sanjeev Khanna. 2019. A New Algorithm for Decremental Single-Source Shortest Paths with Applications to Vertex-Capacitated Flow and Cut Problems. (2019). To appear at STOC'19.
[10] Abdol-Hossein Esfahanian and S. Louis Hakimi. 1984. On computing the connectivities of graphs and digraphs. *Networks* 14, 2 (1984), 355–366.
[11] Shimon Even. 1975. An Algorithm for Determining Whether the Connectivity of a Graph is at Least k. *SIAM J. Comput.* 4, 3 (1975), 393–396.
[12] Shimon Even and Robert Endre Tarjan. 1975. Network Flow and Testing Graph Connectivity. *SIAM J. Comput.* 4, 4 (1975), 507–518.
[13] Lester R Ford and Delbert R Fulkerson. 1956. Maximal flow through a network. *Canadian journal of Mathematics* 8, 3 (1956), 399–404.
[14] Harold N. Gabow. 2006. Using expander graphs to find vertex connectivity. *J. ACM* 53, 5 (2006), 800–844. Announced at FOCS'00.
[15] Zvi Galil. 1980. Finding the Vertex Connectivity of Graphs. *SIAM J. Comput.* 9, 1 (1980), 197–199.
[16] François Le Gall. 2014. Powers of tensors and fast matrix multiplication. In *ISSAC*. ACM, 296–303.
[17] 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.
[18] Shayan Oveis Gharan and Luca Trevisan. 2012. Approximating the Expansion Profile and Almost Optimal Local Graph Clustering. In *FOCS*. IEEE Computer Society, 187–196.
[19] Andrew V. Goldberg and Satish Rao. 1998. Beyond the Flow Decomposition Barrier. *J. ACM* 45, 5 (1998), 783–797. https://doi.org/10.1145/290179.290181
[20] Monika Henzinger, Satish Rao, and Di Wang. 2017. Local Flow Partitioning for Faster Edge Connectivity. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*. 1919–1938. https://doi.org/10.1137/1.9781611974782.125
[21] Monika Rauch Henzinger. 1997. A Static 2-Approximation Algorithm for Vertex Connectivity and Incremental Approximation Algorithms for Edge and Vertex Connectivity. *J. Algorithms* 24, 1 (1997), 194–220.
[22] Monika Rauch Henzinger, Satish Rao, and Harold N. Gabow. 2000. Computing Vertex Connectivity: New Bounds from Old Techniques. *J. Algorithms* 34, 2 (2000), 222–250. Announced at FOCS'96.
[23] John E. Hopcroft and Richard M. Karp. 1973. An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.* 2, 4 (1973), 225–231. https://doi.org/10.1137/0202019
[24] John E. Hopcroft and Robert Endre Tarjan. 1973. Dividing a Graph into Triconnected Components. *SIAM J. Comput.* 2, 3 (1973), 135–158.
[25] 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.
[26] Alexander V Karzanov. 1974. Determining the maximal flow in a network by the method of preflows. In *Soviet Math. Doklady*, Vol. 15. 434–437.
[27] Ken-ichi Kawarabayashi and Mikkel Thorup. 2015. Deterministic Global Minimum Cut of a Simple Graph in Near-Linear Time. In *STOC*. ACM, 665–674.
[28] D Kleitman. 1969. Methods for investigating connectivity of large graphs. *IEEE Transactions on Circuit Theory* 16, 2 (1969), 232–233.
[29] Yin Tat Lee and Aaron Sidford. 2014. Path Finding Methods for Linear Programming: Solving Linear Programs in Õ(vrank) Iterations and Faster Algorithms for Maximum Flow. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*. 424–433. https://doi.org/10.1109/FOCS.2014.52
[30] Nathan Linial, László Lovász, and Avi Wigderson. 1988. Rubber bands, convex embeddings and graph connectivity. *Combinatorica* 8, 1 (1988), 91–102. Announced at FOCS'86.
[31] David W. Matula. 1987. Determining Edge Connectivity in O(nm). In *FOCS*. IEEE Computer Society, 249–251.
[32] Hiroshi Nagamochi and Toshihide Ibaraki. 1992. A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph. *Algorithmica* 7, 5&6 (1992), 583–596.
[33] 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.
[34] 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.
[35] Lorenzo Orecchia and Zeyuan Allen Zhu. 2014. Flow-Based Algorithms for Local Graph Clustering. In *SODA*. SIAM, 1267–1286.
[36] VD Podderyugin. 1973. An algorithm for finding the edge connectivity of graphs. *Vopr. Kibern* 2 (1973), 136.
[37] Thatchaphol Saranurak and Di Wang. 2019. Expander Decomposition and Pruning: Faster, Stronger, and Simpler. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019, San Diego, California, USA, January 6-9, 2019*. 2616–2635. https://doi.org/10.1137/1.9781611975482.162
[38] Daniel A. Spielman and Shang-Hua Teng. 2004. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*. 81–90. https://doi.org/10.1145/1007352.1007372
[39] 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.
[40] Robert Endre Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160. Announced at FOCS'71.
[41] Nate Veldt, David F. Gleich, and Michael W. Mahoney. 2016. A Simple and Strongly-Local Flow-Based Method for Cut Improvement. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 1938–1947. http://jmlr.org/proceedings/papers/v48/veldt16.html
[42] Di Wang, Kimon Fountoulakis, Monika Henzinger, Michael W. Mahoney, and Satish Rao. 2017. Capacity Releasing Diffusion for Speed and Locality. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*. 3598–3607. http://proceedings.mlr.press/v70/wang17b.html
[43] Christian Wulff-Nilsen. 2017. Fully-dynamic minimum spanning forest with improved worst-case update time. In *STOC*. ACM, 1130–1143.