Finding Most-Shattering Minimum Vertex Cuts of Polylogarithmic Size in Near-Linear Time

Kevin Hua
University of Michigan, Ann Arbor, MI, USA

Daniel Li
University of Michigan, Ann Arbor, MI, USA

Jaewoo Park
University of Michigan, Ann Arbor, MI, USA

Thatchaphol Saranurak
University of Michigan, Ann Arbor, MI, USA

Abstract

We show the first near-linear time randomized algorithms for listing all minimum vertex cuts of polylogarithmic size that separate the graph into at least *three* connected components (also known as *shredders*) and for finding the *most shattering* one, i.e., the one maximizing the number of connected components. Our algorithms break the quadratic time bound by Cheriyan and Thurimella (STOC'96) for both problems that has been unimproved for more than two decades. Our work also removes an important bottleneck to near-linear time algorithms for the vertex connectivity augmentation problem (Jordan '95) and finding an even-length directed cycle in a graph, a problem shown to be equivalent to many other fundamental problems (Vazirani and Yannakakis '90, Robertson et al. '99). Note that it is necessary to list only minimum vertex cuts that separate the graph into at least three components because there can be an exponential number of minimum vertex cuts in general.

To obtain a near-linear time algorithm, we have extended techniques in local flow algorithms developed by Forster et al. (SODA'20) to list shredders on a local scale. We also exploit fast queries to a pairwise vertex connectivity oracle subject to vertex failures (Long and Saranurak FOCS'22, Kosinas ESA'23). This is the first application of using connectivity oracles subject to vertex failures to speed up a static graph algorithm.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis

Keywords and phrases Graphs, Flows, Randomized Algorithms, Vertex Connectivity

Digital Object Identifier 10.4230/LIPIcs.ICALP.2024.87

Category Track A: Algorithms, Complexity and Games

Related Version *Full Version*: `https://arxiv.org/abs/2405.03801`

Funding *Thatchaphol Saranurak*: Supported by NSF grant CCF-2238138.

1 Introduction

Given an undirected graph G with n vertices and m edges, a *minimum vertex cut* is a smallest set of vertices whose removal disconnects G. The *vertex connectivity* of G is the size of any minimum vertex. The problem of efficiently computing vertex connectivity and finding a corresponding minimum vertex cut has been extensively studied for more than half a century [16, 25, 9, 8, 12, 7, 21, 1, 19, 5, 22, 4, 13, 14, 11, 2]. Let k denote the vertex connectivity of G. Recently, a $\tilde{\mathcal{O}}(m + nk^3)$-time algorithm was shown in [10], which is near-linear when $k = \mathcal{O}(\text{polylog}(n))$. Then, an almost-linear $\mathcal{O}(m^{1+o(1)})$-time algorithm for the general case was finally discovered [18, 3]. In this paper, we show new algorithms for two closely related problems:

1. Find a *most-shattering* minimum vertex cut, i.e., a minimum vertex cut S such that the number of (connected) components of $G \setminus S$ is maximized over all minimum vertex cuts.
2. List all minimum vertex cuts S such that $G \setminus S$ has at least three components.

In the latter problem, the restriction to at least three components is natural for polynomial-time algorithms. This is because there are at most n many minimum vertex cuts whose removal results in at least three components [15], but the total number of minimum vertex cuts can be exponential or, more specifically, at least $2^k(n/k)^2$ [24][1]. We say that a vertex set S is a *separator* if $G \setminus S$ is not connected and a *shredder* if $G \setminus S$ has at least three components. An s-separator (s-shredder) is a separator (shredder) of size s. In other words, the second problem is to list all k-shredders[2].

The state-of-the-art algorithm for both Problems 1 and 2 was discovered by Cheriyan and Thurimella [6] and runs in $\mathcal{O}(k^2n^2 + k^3n\sqrt{n})$ time. This bound has remained unimproved for over two decades. Their approach inherently requires quadratic time because their algorithm makes $\Omega(n + k^2)$ max flow calls. Naturally, one may ask whether a subquadratic algorithm exists.

Our Contribution

We answer the above question in the affirmative by showing a randomized algorithm for listing all k-shredders and computing a most shattering min-cut in near-linear time for all $k = \mathcal{O}(\text{polylog}(n))$. Our main results are stated below.

▶ **Theorem 1.1.** *Let $G = (V, E)$ be an n-vertex m-edge undirected graph with vertex connectivity k. There exists an algorithm that takes G as input and correctly lists all k-shredders of G with probability $1 - n^{-97}$ in $\mathcal{O}(m + k^5 n \log^4 n)$ time.*

▶ **Theorem 1.2.** *Let $G = (V, E)$ be an n-vertex m-edge undirected graph with vertex connectivity k. There exists a randomized algorithm that takes G as input and returns a most shattering minimum-cut (if one exists) with probability $1 - n^{-97}$. The algorithm runs in $\mathcal{O}(m + k^5 n \log^4 n)$ time.*

▶ Remark 1.3. The $k^5 n \log^4 n$ term in both theorems can be improved to $k^3 n^{1+o(1)}$ using a pairwise vertex connectivity oracle developed by Long and Saranurak in [20], which is faster for large values of k.

Given recent developments in fast algorithms for computing vertex connectivity, one might expect that some of these modern techniques (e.g. local flow [23, 10], sketching [18], expander decomposition [26]) will be useful for listing shredders and finding most shattering min-cuts. It turns out that they are indeed useful, but *not enough*.

We have extended the techniques developed for local flow algorithms [10] to list k-shredders and compute the number of components that they separate. Specifically, our local algorithm lists k-shredders that separate the graph in an unbalanced way in time *proportional to the smaller side* of the cut. To this end, we generalize the structural results related to shredders from [6] to the local setting. To carry out this approach, we bring a new tool into the area – our algorithm queries a pairwise connectivity oracle subject to vertex failures [20, 17]. Surprisingly, this is the first application of using connectivity oracles subject to vertex failures to speed up a static graph algorithm.

---

[1] Both problems are specific to vertex cuts; recall that every minimum *edge* cut always separates a graph into two components.

[2] E.g. in a tree, vertices with degree at least three are 1-shredders. In the complete bipartite graph $K_{k,k}$ with $k \geq 3$, both the left and right halves of the bipartition are k-shredders.

2 Technical Overview

Let $G = (V, E)$ be an n-vertex m-edge undirected graph with vertex connectivity k. Cheriyan and Thurimella developed a deterministic algorithm called `All-`k`-shredders`$(\cdot)$ that takes G as input and lists all k-shredders of G in $\mathcal{O}(knm + k^2\sqrt{n}m)$ time. They improved this bound by using the sparsification routine developed in [22] as a preprocessing step. Specifically, there exists an algorithm that takes G as input and produces an edge subgraph G' on $\mathcal{O}(kn)$ edges such that all k-shredders of G are k-shredders of G' and vice versa. The algorithm runs in $\mathcal{O}(m)$ time. Using this preprocessing step, they obtained the bound for listing all k-shredders in $\mathcal{O}(m) + \mathcal{O}(k(kn)n + k^2\sqrt{n}(kn)) = \mathcal{O}(k^2n^2 + k^3n\sqrt{n})$ time.

In this paper, we resolve a bottleneck of `All-`k`-shredders`$(\cdot)$, improving the time complexity of listing all k-shredders from $\mathcal{O}(knm + k^2\sqrt{n}m)$ to $\mathcal{O}(k^4m\log^4 n)$. Using the same sparsification routine, our algorithm runs in $\mathcal{O}(m + k^5n\log^4 n)$ time. The proofs of the theorems and lemmas presented here have been omitted to the full version of the paper.

2.1 The Bottleneck

A key subroutine of `All-`k`-shredders`$(\cdot)$ is a subroutine called `Shredders`$(\cdot,\cdot)$ that takes a pair of vertices (x, y) as input and lists all k-shredders that separate x and y. This subroutine takes $\mathcal{O}(m)$ time plus the time to compute a flow of size k, which is at most $\mathcal{O}(mk)$ time. The idea behind `All-`k`-shredders`$(\cdot)$ is to call `Shredders`$(\cdot,\cdot)$ multiple times to list all k-shredders. `All-`k`-shredders`$(\cdot)$ works as follows. Let Y be an arbitrary set of $k + 1$ vertices. Any k-shredder S will either separate a pair of vertices in Y, or separate a component containing $Y \setminus S$ from the rest of the graph.

For the first case, they call `Shredders`(u, v) for all pairs of vertices $(u, v) \in Y \times Y$. This takes $\mathcal{O}(k^2)$ calls to `Shredders`$(\cdot,\cdot)$, which in total runs in $\mathcal{O}(k^3m)$ time. For the second case, we can add a dummy vertex z and connect z to all vertices in Y. Notice that any k-shredder S separating a component containing $Y \setminus S$ from the rest of the graph must separate z and a vertex $v \in V \setminus (Y \cup S)$. We can find these k-shredders by calling `Shredders`(z, v) for all $v \in V \setminus Y$. The bottleneck of Cheriyan's algorithm is listing k-shredders that fall into the second case. They call `Shredders`$(\cdot,\cdot)$ $\Omega(n)$ times as $|V \setminus Y| = n - \mathcal{O}(k) = \Theta(n)$ for small values of k. This step is precisely where the quadratic $\mathcal{O}(knm)$ factor comes from. To bypass this bottleneck, we categorize the problem into different cases and employ randomization.

2.2 Quantifying a Notion of Balance

Let S be a k-shredder of G. A useful observation is that if we obtain vertices x and y in different components of $G \setminus S$, then `Shredders`(x, y) will list S. To build on this observation, one can reason about the relative *sizes* of the components of $G \setminus S$ and employ a random sampling approach. Let C denote the "largest" component of $G \setminus S$. If C does not greatly outsize the remaining components, we can obtain two vertices in different components of $G \setminus S$ without too much difficulty. To capture the idea of relative *size* between components of $G \setminus S$, we define a quantity called *volume*. This definition is used commonly throughout the literature, but we have slightly altered it here.

▶ **Definition 2.1** (Volume). *For a vertex set Q, we refer to the* volume *of Q, denoted by* $\mathrm{vol}(Q)$*, to denote the quantity*

$$\mathrm{vol}(Q) = |\{(u, v) \in E \mid u \in Q\}|.$$

If $\mathcal{Q}$ is a collection of disjoint vertex sets, then we define the volume of $\mathcal{Q}$ as the quantity

$$\mathrm{vol}(\mathcal{Q}) = \mathrm{vol}\left(\bigcup_{Q\in\mathcal{Q}} Q\right).$$

For convenience, we will say that a k-shredder S admits *partition* $(C, \mathcal{R})$ to signify that C is the largest component of $G \setminus S$ by volume and $\mathcal{R}$ is the set of remaining components. We will also say $x \in \mathcal{R}$ to denote a vertex in a component of $\mathcal{R}$. We can categorize a k-shredder S by comparing the volume ratio between the largest component of $G \setminus S$ and the union of remaining components (the ratio between $\mathrm{vol}(C)$ and $\mathrm{vol}(\mathcal{R})$).

▶ **Definition 2.2** (Balanced/Unbalanced k-Shredders). *Let S be a k-shredder of G with partition $(C, \mathcal{R})$. We say S is* balanced *if $\mathrm{vol}(\mathcal{R}) \geq m/k$. Conversely, we say S is* unbalanced *if $\mathrm{vol}(\mathcal{R}) < m/k$.*

Suppose that S is a k-shredder with partition $(C, \mathcal{R})$. Building on our discussion above, if S is balanced, then C does not greatly outsize the rest of the graph (by a factor of k at most). Conversely, if S is unbalanced, then C greatly outsizes the rest of the graph.

2.3 Listing Balanced k-Shredders via Edge Sampling

Listing all balanced k-shredders turns out to be straightforward via random edge sampling. Let S be a k-shredder that admits partition $(C, \mathcal{R})$. Suppose that S is balanced. The idea is to sample an edges (x, x') and (y, y') such that x and y are in different components of $G \setminus S$. Then, `Shredders`(x, y) will list S. Intuitively, this approach works because we know that each component of $G \setminus S$ cannot be too large, as S is a balanced k-shredder. Hence, if we sample two edges (x, x') and (y, y') at random, the probability that x and y live in different components of $G \setminus S$ is some constant. We can then boost the success probability by repeating the procedure for a polylogarithmic number of times. The formal statement is given below.

▶ **Lemma 2.3.** *Let $G = (V, E)$ be an n-vertex m-edge undirected graph with vertex connectivity k. There exists a randomized algorithm that takes G as input and returns a list $\mathcal{L}$ that satisfies the following. If S is a balanced k-shredder of G, then $S \in \mathcal{L}$ with probability $1 - n^{-100}$. Additionally, every set in $\mathcal{L}$ is a k-shredder of G. The algorithm runs in $\mathcal{O}(k^2 m \log n)$ time.*

Because handling balanced k-shredders is simple, we omit the algorithm for listing balanced k-shredders and its analysis. We will primarily focus on our methods of listing unbalanced k-shredders.

2.4 Listing Unbalanced k-Shredders via Local Flow

Unbalanced k-shredders are more difficult to list than balanced k-shredders. For balanced k-shredders S with partition $(C, \mathcal{R})$, we can obtain two vertices in different components of $G \setminus S$ without much difficulty. This is no longer the case for unbalanced k-shredders, mainly because $\mathrm{vol}(\mathcal{R})$ may be arbitrarily small. Hence, most sampled edges will be incident to C. To handle unbalanced k-shredders, we introduce a new structural definition.

▶ **Definition 2.4** (Capture). *Let S be an unbalanced k-shredder of a graph $G = (V, E)$ with partition $(C, \mathcal{R})$. Consider an arbitrary tuple (x, ν, Π), where x is a vertex in V, ν is a positive integer, and Π is a set of paths. We say that the tuple (x, ν, Π)* captures *S if the following holds.*

1. *x is in a component of $\mathcal{R}$.*
2. *$\frac{1}{2}\nu < \mathrm{vol}(\mathcal{R}) \leq \nu$.*
3. *Π is a set of k openly-disjoint simple paths, each starting from x and ending at a vertex in C, such that the sum of lengths over all paths is at most $k^2\nu$.*

At a high level, we will spend some time constructing random tuples (x, ν, Π) in the hopes that one of the tuples captures S. The main result is stated below.

▶ **Lemma 2.5.** *Let G be a graph with vertex connectivity k. Let (x, ν, Π) be a tuple where x is a vertex, ν is a positive integer, and Π is a set of paths. There exists a deterministic algorithm that takes (x, ν, Π) as input and outputs a list $\mathcal{L}$ of k-shredders and one set U such that the following holds. If S is a k-shredder that is captured by (x, ν, Π), then $S \in \mathcal{L}$ or $S = U$. The algorithm runs in $\mathcal{O}(k^2\nu \log \nu)$ time.*

What is most important about this result is that we have constructed an algorithm that can identify k-shredders on a *local* scale. This means it spends time proportional to an input volume parameter ν instead of a global quantity like n or m. The idea behind Lemma 2.5 is to modify `Shredders`$(\cdot, \cdot)$ using recent advancements in local flow algorithms.

2.5 Verification via Pairwise Connectivity Oracles

In our algorithm, we will obtain a list of k-shredders and a list of unverified sets. The union of these two sets will include all k-shredders of G with high probability. However, within the list of unverified sets, there may be some false k-shredders. To filter out the false k-shredders, we utilize a pairwise connectivity oracle subject to vertex failures developed by Kosinas in [17]. To determine whether an unverified set S is a k-shredder, we will make some pairwise connectivity queries between vertex pairs (u, v) to determine whether u, v are disconnected in $G \setminus S$. These local queries, along with some more structural observations, will help us determine whether $G \setminus S$ contains at least three components.

3 Preliminaries

This paper concerns finite, undirected, and unweighted graphs with vertex connectivity k. We use standard graph-theoretic definitions. Let $G = (V, E)$ be a graph with vertex connectivity k. For a vertex subset $S \subseteq V$, we use $G \setminus S$ to denote the subgraph of G induced by removing all vertices in S. A *connected component* (component for short) of a graph refers to any maximally-connected subgraph, or the vertex set of such a subgraph. Suppose that S is a k-shredder of G. The *largest component* of $G \setminus S$ is the component with the greatest *volume*, where volume is defined in Definition 2.1. We break ties arbitrarily. For convenience, we will say that a k-shredder S admits partition $(C, \mathcal{R})$ to signify that C is the largest component of $G \setminus S$ and $\mathcal{R}$ is the set of remaining components. We will also write $x \in \mathcal{R}$ to indicate a vertex in a component of $\mathcal{R}$.

For a vertex subset $Q \subseteq V$, we define the set of neighbors of Q as the set $N(Q) = \{v \in V \setminus Q \mid (u, v) \in E, u \in Q\}$. For vertex subsets $Q_1, Q_2 \subseteq V$, we define $E(Q_1, Q_2) = \{(u, v) \in E \mid u \in Q_1, v \in Q_2\}$.

Let π be a simple path in G. We refer to the *length* of π as the number of edges in π. Suppose π starts at a vertex $x \in V$. We say the *far-most endpoint* of π to denote the other endpoint of π. Although all paths we refer to are undirected, our usage of the far-most endpoint will be unambiguous. Let x and y be two arbitrary vertices in π. We denote $\pi[x \rightsquigarrow y]$ as the subpath of π from x to y. We denote $\pi(x \rightsquigarrow y)$ as the subpath $\pi[x \rightsquigarrow y]$ *excluding* the

vertices x and y. We say two paths π, π' are *openly-disjoint* if they share no vertices except their endpoints. Let Π be a set of paths. Then, Π is openly disjoint if all pairs of paths in Π are openly disjoint. We say $v \in \Pi$ to denote a vertex among one of the paths in Π and $(u, v) \in \Pi$ to denote an edge used by one of the paths in Π.

4 Cheriyan and Thurimella's Algorithm

We will use $\texttt{Shredders}(\cdot, \cdot)$ as a subroutine and extend it to a *localized setting*. To do this, we need to review the terminology and ideas presented in [6].

▶ **Theorem 4.1** ([6, Algorithm 2]). *Let G be an n-vertex m-edge undirected graph with vertex connectivity k. Let x and y be two distinct vertices. There exists a deterministic algorithm $\texttt{Shredders}(\cdot, \cdot)$ that takes (x, y) as input and returns all k-shredders of G separating x and y in $\mathcal{O}(km)$ time.*

At a high level, $\texttt{Shredders}(\cdot, \cdot)$ works as follows. Let x and y be two vertices in a k-vertex-connected graph G. Firstly, we use a flow algorithm to obtain a set Π of k openly-disjoint simple paths from x to y. Observe that any k-shredder S of G separating x and y must contain exactly one vertex from each path of Π. More crucially, at least one component of $G \setminus S$ must also be a component of $G \setminus \Pi$. That is, for some component Q of $G \setminus \Pi$, we have $N(Q) = S$. This is the main property that $\texttt{Shredders}(\cdot, \cdot)$ exploits. It lists potential k-shredders by finding components Q of $G \setminus \Pi$ such that $|N(Q)| = k$ and $N(Q)$ consists of exactly one vertex from each path in Π (see Figure 1).

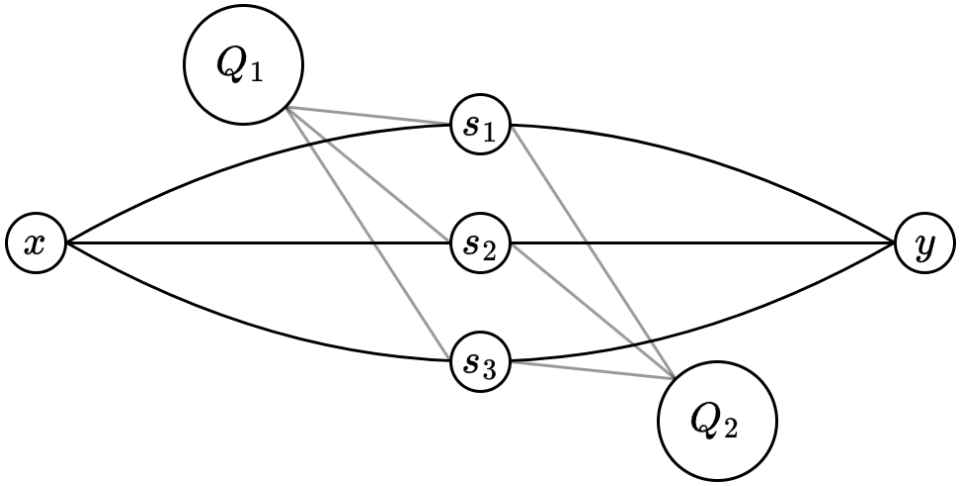

Figure 1 Let Π be the set of three openly-disjoint simple paths from x to y. A call to $\texttt{Shredders}(x, y)$ will identify $N(Q_1) = N(Q_2)$ as potential 3-shredders.

We can state these facts formally with the following definitions.

▶ **Definition 4.2** (Bridge). *A* bridge Γ *of* Π *is a component of $G \setminus \Pi$ or an edge $(u, v) \in E$ such that $(u, v) \notin \Pi$, but $u \in \Pi$ and $v \in \Pi$.*

If Γ is an edge, we define $\text{vol}(\Gamma) = 1$. Otherwise, $\text{vol}(\Gamma)$ is defined according to Definition 2.1.

▶ **Definition 4.3** (Attachments). *Let Γ be a bridge of Π. If Γ is a component of $G \setminus \Pi$, then the set of* attachments *of Γ is the vertex set $N(\Gamma)$. Otherwise, if Γ is an edge $(u, v) \in E$, the set of* attachments *of Γ is the vertex set $\{u, v\}$.*

For convenience, we refer to a single vertex among the attachments of Γ as an attachment of Γ. For a path $\pi \in \Pi$, we denote $\pi(\Gamma)$ as the set of attachments of Γ that are in π. If Q is an arbitrary vertex set, then we use $\pi(Q)$ to denote the set $Q \cap \pi$. If $\pi(Q)$ is a singleton set, then we may use $\pi(Q)$ to represent the unique vertex in $Q \cap \pi$. In all contexts, it will be clear what object the notation is referring to.

▶ **Definition 4.4** (k-Tuple). *A set S is called a k-tuple with respect to Π if $|S| = k$ and for all $\pi \in \Pi$, we have $|S \cap \pi| = 1$.*

We adopt the following notation as in $\texttt{Shredders}(\cdot,\cdot)$. If a component Γ of $G \setminus \Pi$ is such that $N(\Gamma)$ forms a k-tuple with respect to Π, then $N(\Gamma)$ is called a *candidate k-shredder* or *candidate* for short. In general, not all candidates are true k-shredders. To handle this, $\texttt{Shredders}(\cdot,\cdot)$ performs a *pruning* phase by identifying fundamental characteristics of false candidates. To identify false candidates, we formalize some of the key definitions in [6].

▶ **Definition 4.5** (δ_π). *Let π be a path starting from a vertex x. For a vertex $v \in \pi$, we define $\delta_\pi(v)$ as the distance between x and v along path π.*

▶ **Definition 4.6** (Straddle). *Let Γ_1, Γ_2 be two bridges of Π. We say Γ_1* straddles *Γ_2 (or Γ_2 straddles Γ_1) if there exist paths $\pi, \pi' \in \Pi$ such that there exist vertex pairs $(v_1, v_2) \in \pi(\Gamma_1) \times \pi(\Gamma_2)$ satisfying $\delta_\pi(v_1) < \delta_\pi(v_2)$ and $(u_1, u_2) \in \pi'(\Gamma_1) \times \pi'(\Gamma_2)$ satisfying $\delta_{\pi'}(u_1) > \delta_{\pi'}(u_2)$.*

See figure Figure 2 for an example.

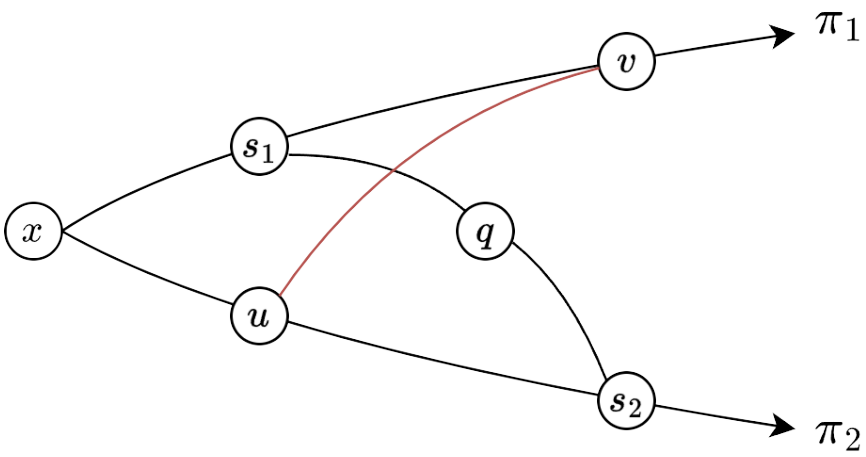

Figure 2 Here we have $\Pi = \{\pi_1, \pi_2\}$. The candidate k-shredder $\{s_1, s_2\}$ is straddled by the edge (u, v) because $\delta_{\pi_2}(u) < \delta_{\pi_2}(s_2)$ and $\delta_{\pi_1}(v) > \delta_{\pi_1}(s_1)$.

It is useful to note that Definition 4.6 is still well-defined even for candidates (i.e. we can use the $\preceq, \succeq$ operators to compare bridges to candidates and candidates to candidates).

Candidate Pruning

The crucial observation is that a candidate S reported by $\texttt{Shredders}(x, y)$ is a k-shredder of G if and only if no bridge of Π straddles S. Hence, to prune false candidates, $\texttt{Shredders}(x, y)$ finds all bridges of Π that are straddled using standard sorting and interval merging. These notions are formalized in the following two lemmas.

▶ **Lemma 4.7.** *Let x and y be two distinct vertices and let Π denote a set of k openly disjoint simple paths from x to y. Let S be a candidate with respect to Π. Then, S is a k-shredder separating x and y if and only if no bridge of Π straddles S.*

▶ **Lemma 4.8.** *Let x be a vertex in G and let Π denote a set of k openly disjoint simple paths starting from x whose sum of lengths over all paths is at most ℓ. Let $\mathcal{C}, \mathcal{B}$ be a set of candidates and a set of bridges of Π, respectively. There exists a deterministic algorithm that, given $(\mathcal{C}, \mathcal{B}, \Pi)$ as input, lists all candidates $S \in \mathcal{C}$ such that S is not straddled by another candidate in $\mathcal{C}$ nor by any bridge in $\mathcal{B}$. The algorithm runs in $\mathcal{O}(k|\mathcal{C}| \log |\mathcal{C}| + \ell + \mathrm{vol}(\mathcal{B}))$ time.*

Challenges

Fix an unbalanced k-shredder S with partition $(C, \mathcal{R})$. Unfortunately, the same probabilistic approach we used for listing balanced k-shredders does not work here. Specifically, if we sample two edges $(x, x'), (y, y')$, the probability that x and y are in different components of $G \setminus S$ can be arbitrarily small. What this dilemma implies is that we must spend at least a linear amount of time just to collect samples. More critically, we must spend a sublinear amount of time processing an *individual* sample to make any meaningful improvement. This time constraint rules out the possibility of calling $\texttt{Shredders}(\cdot, \cdot)$ per sample. Instead, we must develop a *localized* version of $\texttt{Shredders}(\cdot, \cdot)$ that spends time relative to a parameter of our choice, instead of a global value such as m or n.

In detail, consider an unbalanced k-shredder S with partition $(C, \mathcal{R})$. Observe that there must exist a power of two 2^i such that $2^{i-1} < \mathrm{vol}(\mathcal{R}) \leq 2^i$. If we sample $\tilde{\mathcal{O}}(m/2^i)$ edges (x, y), we will sample a vertex $x \in \mathcal{R}$ with high probability due to the classic hitting set lemmas. The goal is to spend only $\tilde{\mathcal{O}}(\mathrm{poly}(k) \cdot 2^i)$ time processing each sample to list S. Suppose that such a local algorithm exists. Although we do not know the exact power of two 2^i, we know that $\mathrm{vol}(\mathcal{R}) < \frac{m}{k}$. Hence, we can simply try all powers of two up to $\frac{m}{k}$. This gives us the near-linear runtime bound:

$$\sum_{i=0}^{\lceil \log \frac{m}{k} \rceil} \tilde{\mathcal{O}}\left(\frac{m}{2^i}\right) \cdot \tilde{\mathcal{O}}(\mathrm{poly}(k) \cdot 2^i) = \sum_{i=0}^{\lceil \log \frac{m}{k} \rceil} \tilde{\mathcal{O}}(\mathrm{poly}(k) \cdot m) = \tilde{\mathcal{O}}(\mathrm{poly}(k) \cdot m).$$

5 Local Techniques

Let S be an unbalanced k-shredder with partition $(C, \mathcal{R})$. Our challenge is to list S in time relative to a parameter of our choice rather than the size of the entire graph. To achieve this, we will implement a local search procedure from a sampled vertex. To formalize the procedure, recall Definition 2.4.

Suppose that we have obtained a tuple (x, ν, Π) that captures S. Let Q denote the component in $\mathcal{R}$ containing x. It is straightforward to see that all remaining components of $\mathcal{R} \setminus \{Q\}$ will also be components of $G \setminus \Pi$. Thus, we may attempt to explore components of $G \setminus \Pi$ and apply Lemma 5.3 (a local version of Lemma 4.7) and Lemma 4.8 to list S. An important distinction from $\texttt{Shredders}(\cdot, \cdot)$ is that in this case, Π is no longer a set of k openly-disjoint simple paths from x to a single vertex y, but rather from x to some vertices in C.

▶ **Lemma 2.5.** *Let G be a graph with vertex connectivity k. Let (x, ν, Π) be a tuple where x is a vertex, ν is a positive integer, and Π is a set of paths. There exists a deterministic algorithm that takes (x, ν, Π) as input and outputs a list $\mathcal{L}$ of k-shredders and one set U such that the following holds. If S is a k-shredder that is captured by (x, ν, Π), then $S \in \mathcal{L}$ or $S = U$. The algorithm runs in $\mathcal{O}(k^2 \nu \log \nu)$ time.*

Our goal is to show that the inner workings of `Shredders`$(\cdot, \cdot)$ can be efficiently translated to our localized setting. We show how to implement this in Algorithm 1. A major caveat concerns the set U. We can think of U as an *unverified set*. Why U exists is a consequence of locality. Essentially, because the algorithm in Lemma 2.5 is time-limited by a volume parameter ν, we may discover a set U but not have enough time to verify whether U is a k-shredder. We first state two helpful lemmas that will provide some understanding of how Algorithm 1 works. These lemmas summarize and translate a large portion of Section 4 in terms of our new setting.

▶ **Fact 5.1.** *Let S be a k-shredder with partition $(C, \mathcal{R})$ captured by (x, ν, Π). For every path $\pi \in \Pi$, we have $|S \cap \pi| = 1$.*

▶ **Lemma 5.2.** *Let S be a k-shredder with partition $(C, \mathcal{R})$ captured by (x, ν, Π). Let Q denote the component in $\mathcal{R}$ containing x. Every component in $\mathcal{R} \setminus \{Q\}$ is a component of $G \setminus \Pi$.*

▶ **Lemma 5.3.** *Let S be a k-shredder with partition $(C, \mathcal{R})$ captured by (x, ν, Π). Then, no bridge of Π straddles S.*

Algorithm Outline

Consider Algorithm 1. At a high level, Algorithm 1 works as follows. Let (x, ν, Π) be a tuple given as input to the algorithm. For each path $\pi \in \Pi$, we can traverse the path from x to the far-most endpoint of π. At a given vertex $u \in \pi$, we can explore the bridges of Π attached to u using a breadth-first search (BFS). While doing so, we maintain a list of bridges of Π. We also maintain a list of candidate k-shredders of Π by checking whether the attachment set of a bridge forms a k-tuple. Among the list of candidate k-shredders, false candidates are then pruned correctly and efficiently via Lemma 5.3 and Lemma 4.8. So far, we have not deviated from `Shredders`$(\cdot, \cdot)$.

The key modification is to impose a restriction on the number of edges we are allowed to explore via BFS. We can explore at most ν edges along each path. Suppose we are processing a vertex u along path π. If the number of edges explored for this path exceeds ν while exploring bridges of Π attached to u, we terminate early and mark u as an *unverified vertex*. Intuitively, the unverified vertex u serves as a boundary of exploration. It signifies that bridges attached to u were not fully explored, so we flag u and treat it with extra caution during a later step. While this may seem dubious, the key is that we are concerned only with k-shredders that are *captured* by (x, ν, Π). Let S be a k-shredder with partition $(C, \mathcal{R})$. For each path $\pi \in \Pi$, if we walk in increasing distance from x and explore attached bridges of Π, observe that it is impossible to visit more than $\text{vol}(\mathcal{R})$ edges prior to reaching a vertex in S. Since $\text{vol}(\mathcal{R}) \leq \nu$, this implies we must either identify S as a candidate k-shredder, or as the unverified set U.

At the end of processing all the paths in Π, we will have obtained a list of bridges of Π, a list of candidate k-shredders of Π, and an *unverified set* consisting of all the unverified vertices. The final step is to black box Lemma 4.8 to prune false candidates among the list of candidate k-shredders.

Algorithm 1 Finding the unbalanced k-shredders with small components containing x.

Input: x - a sample vertex
ν - volume parameter
Π - a set of k openly disjoint paths starting from x
Output: $\mathcal{L}$ - a list of k-shredders captured by (x, ν, Π)
U - an unverified set

$\mathcal{L} \leftarrow \varnothing$
$U \leftarrow \varnothing$
for each path $\pi \in \Pi$ **do**
    reset explored edges and vertices to null
    for each vertex $u \in \pi$ in increasing distance from x **do**
        if u is the far-most endpoint of π **then**
            $U \leftarrow U \cup \{u\}$
            break
        explore all unexplored bridges of Π attached to u with BFS:
            terminate early the moment more than ν edges are explored
        if BFS terminated early **then**
            $U \leftarrow U \cup \{u\}$
            break
        else
            for each bridge Γ explored by BFS **do**
                if the attachment set A of Γ forms a k-tuple of Π **then**
                    $\mathcal{L} \leftarrow \mathcal{L} \cup \{A\}$
if $x \in U$ **then**
    return $(\varnothing, \varnothing)$

prune false k-shredders in $\mathcal{L} \cup \{U\}$ using Lemma 4.8
$F \leftarrow$ the set of far-most endpoints of all paths in Π
if U was not pruned $U \cap F = \varnothing$ **then**
    return $(\mathcal{L}, U)$
else
    return $(\mathcal{L}, \varnothing)$

We state two more lemmas that will be useful in the later sections.

▶ **Lemma 5.4.** *Suppose that Algorithm 1 on input (x, ν, Π) returns an unverified set U such that $U \neq \varnothing$. Then, U is a k-separator. Specifically, let z be the far-most endpoint of an arbitrary path $\pi \in \Pi$. Then, x and z are not connected in $G \setminus U$.*

▶ **Lemma 5.5.** *Suppose that Algorithm 1 returns a nonempty unverified set U on input (x, ν, Π). Let Q denote the component of $G \setminus U$ containing x. There exists a modified version of Algorithm 1 that also computes* $\mathrm{vol}(Q)$ *in $\mathcal{O}(k\nu)$ time.*

6 Resolving Unverified Sets

We now address the case where Algorithm 1 returns an unverified set U on input (x, ν, Π). The difference between U and the list of returned k-shredders $\mathcal{L}$ is that the algorithm did not find a component Q of $G \setminus \Pi$ such that $N(Q) = U$. In some sense, this means that

we "lose" a component of $G \setminus U$. The danger in guessing that U is a k-shredder is that U might only be a k-separator. Imagine that $G \setminus U$ has exactly two components: C_1 and C_2. If $\text{vol}(C_2) \gg \text{vol}(C_1)$, then it becomes quite challenging to determine whether there exists a third component of $G \setminus U$ in $\mathcal{O}(\text{vol}(C_1))$ time. To cope with this uncertainty, we make another structural classification.

▶ **Definition 6.1** (Low-Degree, High-Degree). *Let S be a k-shredder with partition $(C, \mathcal{R})$. Let ν be the unique power of two satisfying $\frac{1}{2}\nu < \text{vol}(\mathcal{R}) \leq \nu$. We say that S has* low-degree *if there exists a vertex $s \in S$ such that $\deg(s) \leq \nu$. Otherwise, we say S has* high-degree.

One aspect of this definition may seem strange: we have specifically described ν as a power of two. This is because in the later sections, we will use geometric sampling to capture k-shredders. The sampling parameters we use will be powers of two. Hence, we have imposed this slightly arbitrary detail on Definition 6.1. For now, all that matters is that $\frac{1}{2}\nu < \text{vol}(\mathcal{R}) \leq \nu$.

Organization

There are two main lemmas for this section. Lemma 6.3 handles low-degree unverified sets and is presented in Section 6.1. Lemma 6.7 handles high-degree unverified sets and is presented in Section 6.2. The idea is that after Algorithm 1 returns an unverified set U, we will use Lemma 6.3 to test whether U is a low-degree k-shredder. If not, we will leave keep U as a potential high-degree k-shredder. After all unverified sets have been returned, we use Algorithm 3 to extract all high-degree k-shredders from the remaining unverified sets.

6.1 Low-Degree

Suppose that U is a low-degree k-shredder with partition $(C, \mathcal{R})$. Suppose that the tuple (x, ν, Π) captures U and Algorithm 1 reports U as an unverified set. Our goal is to design an algorithm that confirms U is a k-shredder in the same time complexity as Algorithm 1 up to $\text{polylog}(n)$ and $\text{poly}(k)$ factors.

Lemma 5.4 gives us two vertices in different components of $G \setminus U$: x and z (where z is the far-most endpoint of any path in Π). Since U is a k-shredder, there must exist a third component of $G \setminus U$ and every vertex in U must be adjacent to a vertex in this third component. Because U is low-degree, there must exist a vertex $u \in U$ with $\deg(u) \leq \nu$. The idea is to scan through the edges adjacent to this low-degree vertex u and find an edge (u, y) such that y is neither connected to x nor z in $G \setminus U$. To make the scanning procedure viable, we need to efficiently answer pairwise connectivity queries in $G \setminus U$. The key ingredient is a result obtained by Kosinas in [17]. Specifically, the following holds.

▶ **Theorem 6.2.** *There exists a deterministic data structure for an undirected graph G on n vertices and m edges with $\mathcal{O}(km \log n)$ preprocessing time that supports the following operations.*

1. *Given a set F of k vertices, perform a data structure update in time $\mathcal{O}(k^4 \log n)$.*
2. *Given a pair of vertices (x, y) return* `true` *if x is connected to y in $G \setminus F$ in time $\mathcal{O}(k)$.*

We defer the method of capturing k-shredders for later. For now, let us assume that the tuple (x, ν, Π) captures a k-shredder U and U is reported as an unverified set by Algorithm 1. We show an auxiliary algorithm that will be helpful in listing low-degree k-shredders.

▶ **Lemma 6.3.** *After preprocessing the input graph using the data structure from Theorem 6.2, there exists a deterministic algorithm that takes in as input (x, ν, Π, U), where U is an unverified set returned by Algorithm 1 on input (x, ν, Π). The algorithm outputs* `true` *if U is a k-shredder and there exists a vertex $u \in U$ such that $\deg(u) \leq \nu$ in $\mathcal{O}(k^4 \log n + k\nu)$ time.*

▶ Remark 6.4. In [20], Long and Saranurak showed the existence of a pairwise connectivity oracle subject to vertex failures with $\mathcal{O}(k^2 n^{o(1)})$ update time and $\mathcal{O}(k)$ query time. Our usage of Kosinas's oracle involves making one update operation with a set of size k and $\mathcal{O}(\nu)$ many connectivity queries. Using Long and Saranurak's version, we can improve the dependency on k in the time complexity of Algorithm 2 to $\mathcal{O}(k^2 n^{o(1)} + k\nu)$ time.

Algorithm Outline

As mentioned above, the idea is to scan through the edges adjacent to u in order to find a vertex y such that y is neither connected to x nor z in $G \setminus U$. We will be utilizing Theorem 6.2 to determine whether a pair of vertices (x, y) are connected in $G \setminus U$. Pseudocode is given in Algorithm 2.

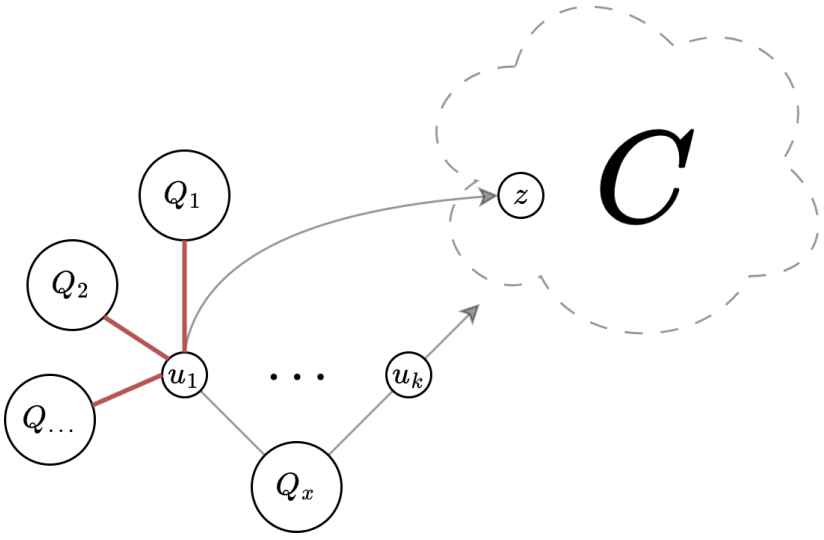

Figure 3 Here the set $U = \{u_1, \dots, u_k\}$ was reported as an unverified set. Suppose there exist a vertex v_1 without loss of generality such that $\deg(u_1) \leq \nu$. Then, we can simply scan through the neighbors of u_1 in $\mathcal{O}(\nu)$ iterations to obtain an edge incident to a component of $G \setminus U$ that is not Q_x nor C.

6.2 High-Degree

It is quite difficult to determine locally whether U is a high-degree k-shredder. We can no longer hope for a low-degree vertex $u \in U$ and scan through its edges to find a third component of $G \setminus U$. To tackle high-degree k-shredders reported as unverified sets, our idea is to ignore them as they are reported and filter them later using one subroutine. Let U be a high-degree k-shredder with partition $(C, \mathcal{R})$. Let ν be the power of two satisfying $\frac{1}{2}\nu < \mathrm{vol}(\mathcal{R}) \leq \nu$. Every vertex $x \in \mathcal{R}$ has $\deg(x) \leq \nu$ because $\mathrm{vol}(\mathcal{R}) \leq \nu$. Furthermore, every vertex $u \in U$ has $\deg(u) > \nu$ because U is high-degree. We can exploit this structure by noticing that U forms a "wall" of high-degree vertices. That is, if we obtain a vertex $x \in \mathcal{R}$, we can use BFS seeded at x to explore a small area of vertices with degree at most ν. The high degree vertices of U would prevent the graph traversal from escaping the component of $G \setminus U$ containing x. At the end of the traversal, the set of explored vertices would compose a component Q of $G \setminus U$ and we can report that $N(Q) = U$ might be a high-degree k-shredder. To obtain a vertex $x \in \mathcal{R}$, we can use the classic hitting set lemmas via random edge sampling.

Algorithm 2 Auxiliary algorithm for low-degree k-shredders.

Input: x - a sample vertex
ν - volume parameter
Π - a set of k openly-disjoint paths starting from x
U - unverified set returned by Algorithm 1 on (x, ν, Π)
Output: `true` if (1) and (2) are satisfied, `false` otherwise
(1) U is a k-shredder
(2) there exists a vertex $u \in U$ such that $\deg(u) \leq \nu$

if all vertices in U have degree greater than ν **then**
  return `false`
$u \leftarrow$ a vertex in U with $\deg(u) \leq \nu$
$\pi \leftarrow$ the path in Π containing u
$z \leftarrow$ the far-most endpoint of π

$\Phi \leftarrow$ the pairwise connectivity oracle from Theorem 6.2 (already preprocessed)
update Φ on vertex failure set U
for each edge (u, y) adjacent to u such that $y \notin U$ **do**
  $q_{x,y} \leftarrow$ connectivity query between x and y in $G \setminus U$
  $q_{y,z} \leftarrow$ connectivity query between y and z in $G \setminus U$
  if $q_{x,y}$ is `false` **and** $q_{y,z}$ is `false` **then**
    return `true`
return `false`

▶ **Lemma 6.5** (Hitting Set Lemma). *Let Q be a set of vertices. Let ν be a positive integer satisfying $\frac{1}{2}\nu < \mathrm{vol}(Q) \leq \nu$. If we independently sample $400\frac{m}{\nu}\log n$ edges (u, v) uniformly at random, we will obtain an edge (u, v) such that $u \in Q$ with probability $1 - n^{-100}$.*

We are not finished yet, as we need to connect this idea with the unverified sets reported by Algorithm 1. Suppose that U was reported by Algorithm 1 on input (x, ν, Π). We have that x is in some component Q of $G \setminus U$. Our goal is to sample a vertex y in a *different* component Q' of $G \setminus U$. As described above, we can exploit the fact that U is a high-degree k-shredder by exploring low-degree vertices in the neighborhood of y to recover Q'. At this point, we have essentially recovered two components of $G \setminus U$: Q and Q'. To make further progress, we present a short, intuitive lemma.

▶ **Lemma 6.6.** *Let U be a k-separator. Let Q, Q' be two distinct components of $G \setminus U$. Then U is a k-shredder if and only if $\mathrm{vol}(Q) + \mathrm{vol}(Q') + |E(U, U)| < m$.*

We are finally ready to handle high-degree k-shredders reported as unverified sets. We present the main result.

▶ **Lemma 6.7.** *There exists a randomized Monte Carlo algorithm that takes as input a list $\mathcal{U}$ of tuples of the form (x, ν, Π, U), where U is an unverified set returned by Algorithm 1 on (x, ν, Π). The algorithm returns a list of k-shredders $\mathcal{L}$ that satisfies the following. If $(x, \nu, \Pi, U) \in \mathcal{U}$ is a k-tuple such that U is a high-degree k-shredder, then $U \in \mathcal{L}$ with probability $1 - n^{-100}$. The algorithm runs in $\mathcal{O}(k^2 m \log^2 n)$ time.*

Algorithm Outline

We summarize the argument made above. Fix a tuple (x, ν, Π, U) in $\mathcal{U}$ such that U is a high-degree k-shredder with partition $(C, \mathcal{R})$. Let ν denote the unique power of two satisfying $\frac{1}{2}\nu < \mathrm{vol}(\mathcal{R}) \leq \nu$. Because U is high-degree, we have for all $u \in U$, $\deg(u) > \nu$. Let Q

denote the component of $G \setminus U$ containing x. Suppose we have obtained a vertex y in a component $Q' \in \mathcal{R} \setminus \{Q\}$. Since all vertices in Q' have degree at most ν, the idea is to explore all vertices connected to y that have degree at most ν. Because $N(Q') = U$ and each vertex in U has degree greater than ν, we will compute Q' as the set of explored vertices. After computing Q', we will have obtained two components of $G \setminus U$: Q and Q'. In order to confirm that U is a k-shredder, all that is left to do is to make sure that Q and Q' are not the only components of $G \setminus U$. Lemma 6.6 implies this can be done by verifying that $\text{vol}(Q) + \text{vol}(Q') + |E(U, U)| < m$. Now, the final step is to find such a vertex y. We use a random sampling procedure for this task. Consider Algorithm 3. We omit the proofs of correctness and time complexity.

Algorithm 3 Extracting all high-degree k-shredders from unverified sets.

Input: $\mathcal{U}$ - a list of tuples of the form (x, ν, Π, U)
Output: $\mathcal{L}$ - a list of k-shredders that satisfies the following:
if $(x, \nu, \Pi, U) \in \mathcal{U}$ is such that U is a high-degree k-shredder,
then $U \in \mathcal{L}$ with probability $1 - n^{-100}$

$\mathcal{L} \leftarrow \varnothing$
for $i \leftarrow 0$ to $\lceil \log\left(\frac{m}{k}\right) \rceil$ **do**
  $\nu' \leftarrow 2^i$
  for $400\frac{m}{\nu'} \log n$ times **do**
    independently sample an edge (y, z) uniformly at random
    $Q' \leftarrow$ set of vertices explored by BFS seeded at y:
      ignore vertices v such that $\deg(v) > \nu'$
      terminate early the moment more than ν' edges are explored
    if BFS terminated early **then**
      skip to next sample
    else if there exists a tuple $(x, \nu, \Pi, U) \in \mathcal{U}$ such that $N(Q') = U$ **then**
      $Q \leftarrow$ component of $G \setminus U$ containing x
      if $x \notin Q'$ **and** $\text{vol}(Q) + \text{vol}(Q') + |E(U, U)| < m$ **then**
        $\mathcal{L} \leftarrow \mathcal{L} \cup \{U\}$
return $\mathcal{L}$

7 Capturing and Listing Unbalanced k-Shredders

In the previous sections, we showed an algorithm that takes as input a tuple (x, ν, Π), and lists all k-shredders that are captured by (x, ν, Π) as well as an unverified set. We then showed how to resolve unverified sets using casework on the structural properties of k-shredders. There is one piece of the puzzle left: the method for capturing unbalanced k-shredders. For this, we will leverage random sampling and recent developments in local flow algorithms as in [10].

7.1 Leveraging Local Flow Algorithms

Let S be an unbalanced k-shredder with partition $(C, \mathcal{R})$. At a high level, we use geometric sampling to obtain a seed vertex $x \in \mathcal{R}$ and a volume parameter ν satisfying $\frac{1}{2}\nu < \text{vol}(\mathcal{R}) \leq \nu$. This obtains two items necessary for capturing S. We are still missing a set of k openly-disjoint paths Π, each starting from x and ending at a vertex in C such that the sum of lengths over all paths is at most $k^2\nu$. This is precisely a core tool developed in [10].

▶ **Definition 7.1** (Vertex Cut). *A vertex cut (L, S, R) of a graph $G = (V, E)$ is a partition of V such that for all vertex pairs $(u, v) \in L \times R$, u is not connected to v in $G \setminus S$.*

▶ **Theorem 7.2** ([10, implicit in Theorem 4.1]). *Let $G = (V, E)$ be an undirected n-vertex m-edge graph with vertex connectivity k. Let (L, S, R) be a vertex cut of G such that $\mathrm{vol}(R) < \frac{m}{k}$. There exists a randomized algorithm `LocalVC`$(\cdot, \cdot)$ that takes as input a pair (x, ν) where x is a vertex in R and ν is a positive integer satisfying $\frac{1}{2}\nu < \mathrm{vol}(R) \leq \nu$. The algorithm outputs a set Π of k openly-disjoint paths such that each path satisfies the following.*
1. *The sum of lengths over all paths in Π is at most $k^2\nu$.*
2. *The path starts from x and ends at a vertex in L.*

The algorithm outputs Π with probability $1/4$ in $\mathcal{O}(k^2\nu)$ time.

Theorem 4.1 of [10] only states that their algorithm returns a vertex cut. But they also construct the set of paths Π. Their algorithm is simple, and we will briefly explain it here. First, we perform the standard vertex-splitting reduction and reduce the problem to finding *directed edge-disjoint* paths instead. To find the first path, we perform any graph search, says DFS, from a vertex x to explore $k\nu$ volume and sample a random endpoint y_1 among all explored edges. Note that y_1 is in L with probability at least $1 - 1/k$ since $\mathrm{vol}(R) \leq \nu$. Then, we reverse the direction of edges on the path from x to y_1 in the DFS tree and obtain a "residual" graph. Then, we repeat the process in the residual graph to construct the next path from x to y_2. After k iterations, the endpoints $y_1, \ldots, y_k$ of these k paths are in L with probability $(1 - 1/k)^k \geq 1/4$. These paths in the residual graphs can be decomposed (like the flow-decomposition) into k directed edge-disjoint paths in the original graph whose total length is $k^2\nu$. These, in turn, correspond k openly vertex-disjoint paths by the standard reduction in the beginning.

Notice how the output of the algorithm described in Theorem 7.2 directly corresponds to Definition 2.4. Let R denote the union of all components in $\mathcal{R}$. Notice that (C, S, R) forms a vertex cut such that $\mathrm{vol}(R) < \frac{m}{k}$. The idea is to obtain a seed vertex $x \in R$ using a linear amount of random samples. For each sample, we can directly apply `LocalVC`$(\cdot, \cdot)$ to obtain the desired set of paths. Furthermore, we can boost the success rate of the algorithm by repeating it a polylogarithmic number of times. In the following section, we formalize this idea.

7.2 The Algorithm for Unbalanced k-Shredders

The main result is stated below.

▶ **Lemma 7.3.** *There exists a randomized algorithm that takes as input $G = (V, E)$, an n-vertex m-edge undirected graph with vertex connectivity k. The algorithm correctly lists all unbalanced k-shredders of G with probability $1 - n^{-98}$ in $\mathcal{O}(k^4 m \log^4 n)$ time.*

We first give a high level outline for listing unbalanced k-shredders. Let S be an unbalanced k-shredder with partition $(C, \mathcal{R})$. With geometric sampling, by the hitting set lemma we will sample a vertex $x \in \mathcal{R}$ with a volume parameter ν that satisfies $\frac{1}{2}\nu < \mathrm{vol}(\mathcal{R}) \leq \nu$. Then, we use Theorem 7.2 to obtain a set of k openly-disjoint paths Π, each starting from x and ending at a vertex in C such that the sum of lengths over all paths is at most $k^2\nu$. We now have a tuple (x, ν, Π) that captures S. After capturing S, we call Algorithm 1 to list S as a k-shredder or an unverified set. In the latter case, we can verify whether S is a low-degree k-shredder using Algorithm 2. If this verification step fails, S must be a high-degree k-shredder. In this case, we can add S to a global list of unverified sets. This list will be processed after all unbalanced k-shredders have been captured. Lastly, we can extract all high-degree k-shredders from the list of unverified sets using Algorithm 3. Pseudocode describing this process is given in Algorithm 4.

Algorithm 4 Listing all unbalanced k-shredders of a graph.

Input: $G = (V, E)$ - an undirected k-vertex-connected graph
Output: $\mathcal{L}$ - a list containing all unbalanced k-shredders of G

$\mathcal{L} \leftarrow \varnothing$ ▷ *list of k-shredders*
$\mathcal{U} \leftarrow \varnothing$ ▷ *unverified sets*
$\Phi \leftarrow$ initialize a pairwise connectivity oracle as in [17]

for $i \leftarrow 0$ to $\lceil \log \left(\frac{m}{k}\right) \rceil$ **do**
  $\nu \leftarrow 2^i$
  for $400 \frac{m}{\nu} \log n$ times **do**
    independently sample an edge (x, y) uniformly at random
    for $100 \log n$ times **do**
      $\Pi \leftarrow \texttt{LocalVC}(x, \nu)$ as in [10]
      $(\mathcal{L}_{local}, U) \leftarrow$ call Algorithm 1 on input (x, ν, Π)
      $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_{local}$
      if $U \neq \varnothing$ **then**
        call Algorithm 2 on input (x, ν, Π, U)
        if Algorithm 2 returned `true` **then**
          $\mathcal{L} \leftarrow \mathcal{L} \cup \{U\}$
        else
          $\mathcal{U} \leftarrow \mathcal{U} \cup \{(x, \nu, \Pi, U)\}$

$\mathcal{L}_{high-degree} \leftarrow$ call Algorithm 3 on input $\mathcal{U}$
$\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_{high-degree}$
return $\mathcal{L}$

8 Listing All k-Shredders

At last, we are ready to present the algorithm for listing all k-shredders. See Algorithm 5.

▶ **Lemma 8.1.** *Let $G = (V, E)$ be an n-vertex m-edge undirected graph with vertex connectivity k. There exists a randomized algorithm that takes G as input and correctly lists all k-shredders of G with probability $1 - n^{-97}$ in $\mathcal{O}(k^4 m \log^4 n)$ time.*

The idea is that we can classify any k-shredder as either balanced or unbalanced. We have described how to use random edge sampling to handle the former case, and Algorithm 4 handles the latter case. Because the algorithm for listing balanced k-shredders is quite trivial, we have omitted its pseudocode. Given Lemma 8.1, Theorem 1.1 follows by the standard sparsification [22].

9 Most Shattering Min-Cut

We have presented a randomized near-linear time algorithm that lists all k-shredders with high probability, but we have not yet shown how to count the number of components each k-shredder separates. Counting components proves to be a bit trickier than it was in [6]. What we will do is modify our algorithms such that whenever we list a k-shredder S, we also return the number of components of $G \setminus S$. We will do this for Algorithm 1, Algorithm 2, and Algorithm 3. Note that if there are no k-shredders of G, we can simply return a k-separator of G. This can be done in $\tilde{\mathcal{O}}(m + nk^3)$ time as shown in [10].

Algorithm 5 Listing all k-shredders of a graph.

Input: $G = (V, E)$ - an undirected k-vertex-connected graph
Output: $\mathcal{L}$ - a list of all k-shredders of G

1: $\mathcal{L} \leftarrow \varnothing$

2: $\mathcal{L}_{balanced} \leftarrow$ edge sampling and `Shredders`$(\cdot, \cdot)$
3: $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_{balanced}$

4: $\mathcal{L}_{unbalanced} \leftarrow$ call Algorithm 4 on input G
5: $\mathcal{L} \leftarrow \mathcal{L} \cup \mathcal{L}_{unbalanced}$

6: **return** $\mathcal{L}$

9.1 Counting Components in the Local Algorithm

At a high level, we can modify Algorithm 1 as follows. Suppose Algorithm 1 identifies a candidate k-shredder S on line 16. By line 13, all bridges of Π attached to some vertex $s \in S$ must have been explored. The modification is as follows. We can keep a dictionary M whose keys are candidate k-shredders (in k-tuple form) mapped to nonnegative integers initialized to zero. For each bridge Γ of Π attached to s, if Γ is a component of $G \setminus \Pi$ such that $N(\Gamma) = S$, then we increment $M[S]$. We claim that if $S \in \mathcal{L}$ is a k-shredder captured by (x, ν, Π), then $M[S] + 2$ is the number of components of $G \setminus S$.

▶ **Lemma 9.1.** *Suppose that Algorithm 1 returns $(\mathcal{L}, U)$ on input (x, ν, Π). There exists a modified version of Algorithm 1 such that for each k-shredder $S \in \mathcal{L}$ that is captured by (x, ν, Π), the modified version also computes the number of components of $G \setminus S$. The modification requires $\mathcal{O}(k^2\nu)$ additional time, subsumed by the running time of Algorithm 1.*

9.2 Counting Components for Low-Degree Unverified Sets

If Algorithm 2 returned `true` on input (x, ν, Π, U), then U is a k-shredder with a vertex $u \in U$ such that $\deg(u) \leq \nu$. Notice that all components of $G \setminus U$ must contain a vertex that is adjacent to u. Immediately, we have that the number of components of $G \setminus U$ is upper bounded by ν, as $\deg(u) \leq \nu$. However, looking at two arbitrary edges $(u, v_1), (u, v_2)$ adjacent to u, it is unclear whether v_1 and v_2 are in the same component of $G \setminus U$. Although we can query whether v_1 and v_2 are connected in $G \setminus U$ in $\mathcal{O}(k)$ time, we cannot afford to make these queries for all pairs of vertices in $N(u)$. Such a procedure would require us to make $\mathcal{O}(\nu^2)$ many queries, which is too costly. To solve this issue, we use a convenient property of DFS trees to bypass making $\mathcal{O}(\nu^2)$ connectivity queries. Much of the machinery was inspired by [17], so we omit the details.

▶ **Lemma 9.2.** *Suppose that Algorithm 2 returns `true` on input (x, ν, Π, U). There exists a modified version of Algorithm 2 that also computes the number of components of $G \setminus U$. The modification requires $O(k^2\nu)$ additional time, subsumed by the running time of Algorithm 2.*

9.3 Counting Components for High-Degree Unverified Sets

Recall from Section 6.2 that the key observation is that S forms a wall of high-degree vertices. We exploited this by sampling vertices in $\mathcal{R}$ and exploring regions of vertices with low degree. The idea is that S will restrict the graph exploration within one component of $\mathcal{R}$, and we can

recover that component. We can actually extend this idea a bit further to count the number of components in $\mathcal{R}$. For all components $Q \in \mathcal{R}$, our algorithm will eventually sample a vertex $x \in Q$. By exploring low-degree vertices, we will recover Q. We can verify that $N(Q)$ is a k-shredder as by now we have listed all of them. Hence, we will eventually count all components of $G \setminus S$ throughout the sampling and exploring procedure.

▶ **Lemma 9.3.** *Suppose that Algorithm 3 returns a list $\mathcal{L}$ of k-shredders on input $\mathcal{U}$. There exists a modified version of Algorithm 3 such that for each k-shredder $S \in \mathcal{L}$, the modified version also computes the number of components of $G \setminus S$. The modification requires $\mathcal{O}(m \log^2 n)$ additional time, subsumed by the time complexity of Algorithm 3.*

Given Lemmas 9.1–9.3, we can compute a most shattering minimum-cut with high probability in $O(k^4 m \log^4 n)$ time and, hence, Theorem 1.2 follows by the standard sparsification [22].

References

1 Michael Becker, W. Degenhardt, Jürgen Doenhardt, Stefan Hertel, Gerd Kaninke, W. Kerber, Kurt Mehlhorn, Stefan Näher, Hans Rohnert, and Thomas Winter. A probabilistic algorithm for vertex connectivity of graphs. *Inf. Process. Lett.*, 15(3):135–136, 1982.

2 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *PODC*, pages 156–165. ACM, 2014.

3 Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time, 2022. arXiv:2203.00671.

4 Joseph Cheriyan and John H. Reif. Directed s-t numberings, rubber bands, and testing digraph k-vertex connectivity. *Combinatorica*, 14(4):435–451, 1994. Announced at SODA'92.

5 Joseph Cheriyan and Ramakrishna Thurimella. Algorithms for parallel k-vertex connectivity and sparse certificates (extended abstract). In *STOC*, pages 391–401. ACM, 1991.

6 Joseph Cheriyan and Ramakrishna Thurimella. Fast algorithms for k-shredders and k-node connectivity augmentation. *Journal of Algorithms*, 33(1):15–50, 1999. doi:10.1006/jagm.1999.1040.

7 Abdol-Hossein Esfahanian and S. Louis Hakimi. On computing the connectivities of graphs and digraphs. *Networks*, 14(2):355–366, 1984.

8 Shimon Even. An algorithm for determining whether the connectivity of a graph is at least k. *SIAM J. Comput.*, 4(3):393–396, 1975.

9 Shimon Even and Robert Endre Tarjan. Network flow and testing graph connectivity. *SIAM J. Comput.*, 4(4):507–518, 1975.

10 Sebastian Forster, Danupon Nanongkai, Thatchaphol Saranurak, Liu Yang, and Sorrachai Yingchareonthawornchai. 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*, pages 2046–2065. SIAM, 2020.

11 Harold N. Gabow. Using expander graphs to find vertex connectivity. *J. ACM*, 53(5):800–844, 2006. Announced at FOCS'00.

12 Zvi Galil. Finding the vertex connectivity of graphs. *SIAM J. Comput.*, 9(1):197–199, 1980.

13 Monika Rauch Henzinger. A static 2-approximation algorithm for vertex connectivity and incremental approximation algorithms for edge and vertex connectivity. *J. Algorithms*, 24(1):194–220, 1997.

14 Monika Rauch Henzinger, Satish Rao, and Harold N. Gabow. Computing vertex connectivity: New bounds from old techniques. *J. Algorithms*, 34(2):222–250, 2000. Announced at FOCS'96.

15 Tibor Jordán. On the number of shredders. *Journal of Graph Theory*, 31(3):195–200, 1999. doi:10.1002/(SICI)1097-0118(199907)31:3<195::AID-JGT4>3.0.CO;2-E.

16 D Kleitman. Methods for investigating connectivity of large graphs. *IEEE Transactions on Circuit Theory*, 16(2):232–233, 1969.

17 Evangelos Kosinas. Connectivity Queries Under Vertex Failures: Not Optimal, but Practical. In *31st Annual European Symposium on Algorithms (ESA 2023)*, volume 274 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 75:1–75:13, 2023. doi:10.4230/LIPIcs.ESA.2023.75.

18 Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. *CoRR*, abs/2104.00104, 2021. arXiv:2104.00104.

19 Nathan Linial, László Lovász, and Avi Wigderson. Rubber bands, convex embeddings and graph connectivity. *Combinatorica*, 8(1):91–102, 1988. Announced at FOCS'86.

20 Yaowei Long and Thatchaphol Saranurak. Near-optimal deterministic vertex-failure connectivity oracles. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1002–1010, 2022. doi:10.1109/FOCS54457.2022.00098.

21 David W. Matula. Determining edge connectivity in o(nm). In *FOCS*, pages 249–251. IEEE Computer Society, 1987.

22 Hiroshi Nagamochi and Toshihide Ibaraki. A linear-time algorithm for finding a sparse k-connected spanning subgraph of ak-connected graph. *Algorithmica*, 7(1-6):583–596, 1992.

23 Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Breaking quadratic time for small vertex connectivity and an approximation scheme. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2019, pages 241–252, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3313276.3316394.

24 Seth Pettie and Longhui Yin. The structure of minimum vertex cuts. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 105:1–105:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICS.ICALP.2021.105.

25 VD Podderyugin. An algorithm for finding the edge connectivity of graphs. *Vopr. Kibern*, 2:136, 1973.

26 Thatchaphol Saranurak and Sorrachai Yingchareonthawornchai. Deterministic small vertex connectivity in almost linear time. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 789–800. IEEE Computer Society, 2022. doi:10.1109/FOCS54457.2022.00080.