Optimal Short Cycle Decomposition in Almost Linear Time

Merav Parter
Weizmann IS, Rehovot, Israel
http://www.weizmann.ac.il/math/parter/home

Eylon Yogev
Technion, Haifa, Israel
https://www.eylonyogev.com/about

Abstract

Short cycle decomposition is an edge partitioning of an unweighted graph into edge-disjoint short cycles, plus a small number of extra edges not in any cycle. This notion was introduced by Chu et al. [FOCS'18] as a fundamental tool for graph sparsification and sketching. Clearly, it is most desirable to have a *fast* algorithm for partitioning the edges into as *short* as possible cycles, while omitting *few* edges.

The most naïve procedure for such decomposition runs in time $O(m \cdot n)$ and partitions the edges into $O(\log n)$-length edge-disjoint cycles plus at most $2n$ edges. Chu et al. improved the running time considerably to $m^{1+o(1)}$, while increasing both the length of the cycles and the number of omitted edges by a factor of $n^{o(1)}$. Even more recently, Liu-Sachdeva-Yu [SODA'19] showed that for every constant $\delta \in (0, 1]$ there is an $O(m \cdot n^{\delta})$-time algorithm that provides, w.h.p., cycles of length $O(\log n)^{1/\delta}$ and $O(n)$ extra edges.

In this paper, we significantly improve upon these bounds. We first show an $m^{1+o(1)}$-time *deterministic* algorithm for computing nearly optimal cycle decomposition, i.e., with cycle length $O(\log^2 n)$ and an extra subset of $O(n \log n)$ edges not in any cycle. This algorithm is based on a reduction to *low-congestion cycle covers*, introduced by the authors in [SODA'19].

We also provide a simple deterministic algorithm that computes edge-disjoint cycles of length $2^{1/\epsilon}$ with $n^{1+\epsilon} \cdot 2^{1/\epsilon}$ extra edges, for every $\epsilon \in (0, 1]$. Combining this with Liu-Sachdeva-Yu [SODA'19] gives a *linear* time randomized algorithm for computing cycles of length $\mathsf{poly}(\log n)$ and $O(n)$ extra edges, for every n-vertex graphs with $n^{1+1/\delta}$ edges for some constant δ.

These decomposition algorithms lead to improvements in all the algorithmic applications of Chu et al. as well as to new distributed constructions.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Cycle decomposition, low-congestion cycle cover, graph sparsification

Digital Object Identifier 10.4230/LIPIcs.ICALP.2019.89

Category Track A: Algorithms, Complexity and Games

Funding *Merav Parter*: The Israel Science Foundation grant no. 2084/18
Eylon Yogev: The European Union's Horizon 2020 research and innovation program under grant agreement No. 742754.

1 Introduction

Short cycle decomposition introduced by Chu et al. [3] is a partitioning of the graph edges into edge-disjoint short cycles and a small subset of extra edges that are not in any cycle.

▶ **Definition 1** (Short Cycle Decomposition). *An $(\widehat{m}, L)$-short cycle decomposition of an unweighted undirected graph G is a collection of edge-disjoint cycles in G, each of length at most L, such that at most $\widehat{m}$ edges of G are not covered by these cycles.*

46th International Colloquium on Automata, Languages, and Programming (ICALP 2019).
Editors: Christel Baier, Ioannis Chatzigiannakis, Paola Flocchini, and Stefano Leonardi;
Article No. 89; pp. 89:1–89:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In their recent paper, Chu et al. [3] demonstrated the power of short cycle decomposition as a fundamental tool for a number of problems in graph sparsification. This includes the construction of degree preserving sparsifiers, resistance sparsifiers, graphical spectral sketches, approximation of the Laplacian's determinant and more. Chu et al. [3] showed that the efficiency of this long list of problems is determined by the time complexity, the cycle length, and the number of uncovered edges of the short cycle decomposition routine in hand. Clearly, it is most desirable to compute fast a decomposition into the shortest possible edge-disjoint cycles, while omitting as few as possible edges.

As they have observed, there is a naïve short cycle decomposition which runs in time $O(mn)$ (where n is the number of vertices in the graph, and m is the number of edges), and outputs an optimal[1] decomposition: cycle length of $O(\log n)$ and $O(n)$ extra edges. In their key algorithmic result, [3] significantly improved this time complexity by presenting an almost-linear time algorithm[2] which decomposes G into edge-disjoint cycles of length $n^{o(1)}$, and an extra number of $n^{1+o(1)}$ edges. Thus, the improvement in the time complexity came with the cost of increasing both the cycle length as well as the number of left-over edges by a multiplicative factor of $n^{o(1)}$. Improving the efficiency of the short cycle decomposition was stated in [3] as a highly motivated target, as it immediately leads to a large sequence of algorithmic consequences:

> *"Critically, any improvement to our short-cycle decomposition algorithm will achieve an improvement in all of our results."*

Very recently, Liu, Sachdeva and Yu [8] provided the first improvement for the problem, by presenting an $O(m \cdot n^{\delta})$-time algorithm that decomposes G into edge-disjoint cycles of length $O(\log n)^{1/\delta}$ and an extra subset of $O(n)$ edges, for any *constant* $\delta \in (0, 1]$. This simplifies and improves the decomposition algorithm of [3] in terms of all parameters, but still leaves the following fundamental question open:

Is there an ***optimal*** *cycle decompositios in almost* ***linear*** *time?*

In this paper, we answer this question in the affirmative and present an $m^{1+o(1)}$-time algorithm for decomposing the graph into edge-disjoint cycles of length $O(\log^2 n)$, and an extra number of $O(n \log n)$ edges.

▶ **Theorem 2** (Nearly Optimal Decomposition in Almost-Linear Time). *There is an almost-linear time algorithm for computing a cycle decomposition with cycle length of $O(\log^2 n)$, and $O(n \log n)$ extra edges.*

We also have a much simpler algorithm that achieves the same quality of cycle decomposition[3] as by [3], only in $\widetilde{O}(m)$ time. An additional benefit of this algorithm is that it is deterministic.

▶ **Theorem 3** (Longer Cycles in Linear Time). *For every n-vertex graph $G = (V, E)$ with m edges, one can compute in $\widetilde{O}(m)$ time, a decomposition with cycle length $n^{o(1)}$ and $n^{1+o(1)}$ extra edges.*

---

[1] Optimality in this context is up to poly-logarithmic terms.

[2] A graph algorithm is almost-linear if runs in time $m^{1+o(1)}$.

[3] In fact, the quality here is slightly better, as the $n^{o(1)}$ factor (in the cycle length and number of uncovered edges) is $2^{\sqrt{\log n}}$ and in [3] et al. it is $2^{(\log n)^{3/4}}$.

The latter provides an improvement for approximating the determinant of the Laplacian. We can also combine with the algorithm of [8] to obtain a randomized decomposition that is optimal (up to log-n factors) in all three complexity measures: time, length and number of leftover edges provided that the graph is sufficiently dense.

▶ **Theorem 4** (Optimal Decomposition for Dense Graphs). *For every constant $\delta \in (0,1]$, there exists a randomized algorithm that computes in time $\widetilde{O}(m) + n^{1+1.1\delta}$ a collection of edges disjoint cycles of length $O(\log n)^{1/\delta}$ and at most $O(n)$ leftover edges.*

Table 1 provides a summary of our results in comparison to [3, 8]. Our algorithms are based on independent ideas compared to [3, 8], which are related to the concept of *low-congestion cycle covers*.

Table 1 Summary of our results.

	Cycle Length	#Uncovered Edges	Time	Type
Chu et al. [3]	$n^{o(1)}$	$n^{1+o(1)}$	$m^{1+o(1)}$	Randomized
Liu-Sachdeva-Yu [8]	$O(\log n)^{1/\delta-1}$ for constant $\delta \leq 1$	$O(n)$	$O(mn^{\delta})$	Randomized
This Work	$O(\log^2 n)$	$O(n \log n)$	$m^{1+o(1)}$	Deterministic
This Work	$O(\log n)^{1/\delta}$ for constant $\delta \leq 1$	$O(n)$	$m + n^{1+1.1\delta}$	Randomized
This Work	$n^{o(1)}$	$n^{1+o(1)}$	$m + n^{1+o(1)}$	Deterministic

1.1 Low-Congestion Cycle Covers

A cycle cover of a graph G is a collection of cycles such that each edge of G appears in at least one of the cycles. Cycle covers were introduced by Itai and Rodeh [6] in 1978 with the objective to cover all edges of a bridgeless[4] graph with cycles of minimum *total* length. Motivated by applications to distributed computation, [11] recently introduced[5] the notion of *low-congestion* cycle covers: a collection of cycles that are both *short*, nearly *edge-disjoint* and covering all edges.

▶ **Definition 5** (Low-Congestion Cycle Cover). *A $(\mathtt{d}, \mathtt{c})$-cycle cover of a graph G is a cycle collection that covers all edges in G. Each cycle has length at most $\mathtt{d}$, and each edge participates in at least one cycle and at most $\mathtt{c}$ cycles.*

Low-congestion cycle covers provide the basic communication backbone in different settings of resilient distributed computation such as Byzantine fault model and secure computation [11, 10]. Whereas a-priori it is not clear that cycles of short length and small overlap exist, our main result in [11] shows that one can enjoy a dilation of $O(D \log n)$ while incurring only a poly-logarithmic congestion, where D is the diameter of the graph.

Comparison to Short Cycle Decomposition. Low-congestion covers bare some similarity to short cycle decomposition but differs from it in two main aspects. The first aspect follows from the definition: Low-congestion covers allow a small *overlap* between the cycles,

[4] A graph G is bridgeless, if any single edge removal keeps the graph connected.
[5] In an independent manner to the notion of short cycle decomposition.

but require covering *all* edges. On the other hand, short cycle decomposition insists on edge-disjoint cycles, i.e., with no-overlap, but allows omitting a subset of leftover edges that are not in any cycle. The second difference concerns the algorithmic focus. In low-congestion covers, the main goal was in showing that optimal covers which are both short and with small overlap *exist*. Computation time was not the primary concern, and in fact, the first step in the construction in [11] used the naïve decomposition algorithm (that runs in time $O(m \cdot n)$) to reduce the number of uncovered edges into $2n$. In contrast, for short cycle decomposition, it is easy to obtain the optimal decomposition in $O(m \cdot n)$ time, and hence the primary algorithmic focus is computation time.

1.2 Improved Graph Sparsification Algorithms via Short Cycles

Spectral sparsifiers introduced by Spielman and Teng [13] are sparse (weighted) subgraphs that approximately preserve the Laplacian quadratic form of the graph. Recall that the *Laplacian*, $\mathbf{L}_G$, of an undirected weighted graph $G = (V, E_G, w_G)$ is the unique symmetric $V \times V$ matrix such that for all $x \in \mathbb{R}^{|V|}$, it holds that

$$x^T \mathbf{L}_G x = \sum_{(u,v)\in E_G} w_G(u,v) \cdot (x_u - x_v)^2 \ .$$

For $\epsilon < 1$, a graph $H = (V, E_H, w_H)$ is an *ϵ-sparsifier* for G if

$$\forall x \in \mathbb{R}^n,\ x^T \mathbf{L}_G x \in (1 \pm \epsilon) x^T \mathbf{L}_H x \ .$$

Batson, Spielman and Srivastava [2] presented a construction of spectral sparsifiers with $O(n/\epsilon^2)$ edges, which is tight. In the last years, related graph structures have been defined, which are weaker than spectral sparsifiers, and thus potentially sparser. The recent work of Chu et al. [3] used short cycle decomposition to derive new existential results on the sparsity of sparsifiers and spectral sketches. As the time complexity and the quality of their algorithms depend on the efficiency of the decomposition, our decomposition algorithm leads to immediate improvements for all the algorithmic results from [3].

Graphic ϵ-Spectral Sketch and Resistance Sparsifiers. A spectral sketch [1] is a data structure for a graph G that given a query vector $x \in \mathbb{R}^n$ returns w.h.p. a $(1+\epsilon)$ approximation for the quadratic form $x^T \mathbf{L}_G x$. A data structure that works w.h.p. for *all* $x \in \{\pm 1\}^n$ requires n/ϵ^2 space. However, Jambulapati and Sidford [7] showed that when requiring the high probability guarantee for a *fixed* unknown vector, the size of the data structure can be made $\widetilde{O}(n/\epsilon)$. In the same manner, one can define the *graphic spectral sketch* of G to be a sparse graph H satisfying $x^T \mathbf{L}_G x \in (1 \pm \epsilon) x^T \mathbf{L}_H x$ for a fixed unknown vector x with high probability. Chu et al. showed that graphical spectral sketches with $\widetilde{O}(n/\epsilon)$ edges exist.

Resistance sparsifiers are sparse subgraphs that preserve the effective resistance[6] of all vertex pairs up to a multiplicative factor of $(1 + \epsilon)$. This notion was introduced by Dinitz-Krauthgamer-Wagner [4] who conjectured that resistance sparsifiers with $\widetilde{O}(n/\epsilon)$ edges always exist. The conjecture was indeed resolved by [3] using the tool of short cycle decomposition. By combining Theorem 2 with [3, Theorem 6.1] we get:

[6] The effective resistance between a pair u, v is the difference in the voltage between u, v when the graph is an electrical network, with every edge e of weight w_e has a resistor of resistance $1/w_e$ and 1 unit of current is sent from u to v.

▶ **Theorem 6** (Graphic Spectral Sketches and Resistance Sparsifiers). *One can compute an ϵ-resistance sparsifier H and a graphical spectral sketch H' with $\widetilde{O}(n/\epsilon)$ edges in time $m^{1+o(1)}$. Alternatively, these algorithms can be tuned to run in $\widetilde{O}(m)$ time while producing such graphs H, H' with $n^{1+o(1)}/\epsilon$ edges.*

These two results should be compared with (i) the $O(m \cdot n^{\Theta(1)})$-time algorithm of [8] that gives sparsifiers with $\widetilde{O}(n/\epsilon)$ edges; and (ii) the $m^{1+o(1)}$-time algorithms of [3, 8] that give sparsifiers with $n^{1+o(1)}/\epsilon$ edges. Hence, we provide the first almost-linear time algorithm for optimal size sparsifiers with $\widetilde{O}(n/\epsilon)$ edges, and the first near linear time algorithm[7] for almost-linear size sparsifiers with $\widetilde{O}(n^{1+o(1)}/\epsilon)$ edges.

Degree Preserving Sparsifiers and Sparsifiers for Eulerian Directed Graphs. Short cycle decompositions are useful for providing spectral sparsifiers that preserve additional key properties in the original graphs. *Degree preserving sparsifier* is a spectral sparsifier H for a graph G which preserves (exactly) the *weighted degree* of each vertex $v \in V$. To get an intuition for the usefulness of edge-disjoint cycles in this context, imagine G to be an unweighted union of edge-disjoint cycles of even length. A degree preserving sparsifier H that contains half of the edges in G, can be obtained by the following correlated sampling: For each cycle, with probability $1/2$ add to H the odd edges with weight 2, and with probability $1/2$ add to H the even edges with weight 2. It is easy to see that every vertex v has exactly the same weighted degree in H as in G, and the number of edges in G was cut by half. By combining Theorem 3.3 of Chu et al. [3] with Theorem 2, we get:

▶ **Theorem 7** (Optimal Degree Preserving Sparsifiers in Almost Linear Time). *There exists an algorithm that runs in time $m^{1+o(1)}$ and constructs a degree-preserving ϵ-sparsifier of G with $\widetilde{O}(n/\epsilon^2)$ edges, with high probability. Alternatively, an $n^{1+o(1)}/\epsilon^2$-size degree-preserving sparsifier can be computed in $\widetilde{O}(m)$ time.*

A similar approach has been taken in Chu et al. [3] to construct a sparsification of Eulerian directed graphs. By plugging Theorem 2 in Theorem 5.1 of [3], we get:

▶ **Theorem 8** (Sparsification of Eulerian Directed Graphs). *There exists an algorithm $\mathcal{A}$ that given an Eulerian directed graph $\overrightarrow{G}$ with polynomial bounded edge weights returns an Eulerian directed graph $\overrightarrow{H}$ such that either: (i) $\overrightarrow{H}$ has $\widetilde{O}(n/\epsilon^2)$ edges and $\mathcal{A}$ has time complexity $m^{1+o(1)}$, or (ii) $\overrightarrow{H}$ has $n^{1+o(1)}/\epsilon^2$ edges and $\mathcal{A}$ has time complexity $\widetilde{O}(m)$.*

Estimation of the Effective Resistance and the Determinant of the Laplacian. Finally, we show how incorporating our improved construction of resistance sparsifiers can yield a faster approximation of the determinant of the graph Laplacian with the last row and column removed. In particular, this yields the first linear time algorithm for sufficiently dense graphs. Recall that by Theorem 6, given a graph G with m edges, we can compute in time $\widetilde{O}(m)$, a resistance sparsifier H with $n^{1+o(1)}/\epsilon$ edges, with high probability. By applying [3, Thm. 3.8] on H, we get:

▶ **Theorem 9** (Faster Approximation of Effective Resistance). *Given an undirected graph G with m edges, one can compute with high probability an ϵ-approximations to the effective resistances between a given set of t vertex pairs in time $\widetilde{O}(m) + (n+t)n^{o(1)}\epsilon^{-1.5}$.*

[7] A graph algorithm is *near linear* if it runs in $O(m \cdot \mathsf{poly}(\log n))$ time.

Hence, for $t = o(m^{1-o(1)} \cdot \epsilon^{1.5})$, we obtain a linear time algorithm for approximating the effective resistance. As observed in [3], the running time bottleneck of the determinant estimation algorithm for Laplacians by Durfee et al. [5] is the estimation of the effective resistance of $O(n^{1.5})$ pairs with an error of $\epsilon = n^{-0.25}$. Plugging our improved Theorem 9 in Lemma B.1 of [3] yields:

▶ **Corollary 10** (Faster Approx. of Laplacian's Determinant). *Given a graph Laplacian* $\mathbf{L}$ *and any error* $0 < \epsilon < 1/2$*, one can compute an* $1 \pm \epsilon$ *estimate to* $\det(\mathbf{L}_{-n})$[8] *in time* $\widetilde{O}(m) + n^{15/8+o(1)} \cdot \epsilon^{-7/4}$*, thus in linear time for sufficiently dense graphs.*

1.3 Distributed Implementation

Our centralized construction has the benefit of naturally being implemented in the standard CONGEST model of distributed computing. As in the centralized setting, the decomposition is based on constructing low-congestion cycle covers. We show:

▶ **Lemma 11** (Distributed Low-Congestion Covers). *There exists a distributed algorithm that given n-vertex graph* $G = (V, E)$ *constructs a* $(\mathtt{d}, \mathtt{c})$ *cycle cover within* $O(\mathtt{d} \cdot \mathtt{c})$ *rounds for* $\mathtt{d}, \mathtt{c} = 2^{O(\sqrt{\log n})}$.

The proof appears in the full version of the paper. This improves upon the linear time algorithm of [11, 10]. By combining this algorithm with Luby-MIS algorithm [9], we get:

▶ **Theorem 12** (Distributed Short Cycle Decomposition). *There exists an* $2^{O(\sqrt{\log n})}$*-round distributed algorithm that given n-vertex graph* $G = (V, E)$ *decomposes* G *into edge-disjoint cycles of length* $2^{O(\sqrt{\log n})}$ *plus* $O(n \log n)$ *extra edges.*

One might hope that using this distributed construction of cycle decomposition we would get a distributed algorithm for all the above application. Unfortunately, in the distributed setting, to this point, we do not have an efficient algorithm that approximates (even up to a constant factor) the effective resistance of all edges[9] in G. This is the only missing piece for obtaining the above mentioned algorithmic applications of the short cycle decomposition in a distributed setting.

Comparison to the work of Chu et al. [3] and Liu-Sachdeva-Yu [8]. We first observe that in [8], the number of leftover edges is $O(n)$, whereas in our case it is $O(n \log n)$. By applying the algorithm of [8] on these last $O(n \log n)$ edges, for any constant $\delta \in (0, 1)$, we can compute an $O(n, (\log n)^{1/\delta-1})$-decomposition in time $O(n^{1+\delta} \log n + m^{1+1/\log\log n})$, which considerably improves upon the time complexity of $O(m \cdot n^{\delta})$ of [8]. Since we consider log-n factors to be negligible in this work (i.e., the size of the sparsifiers is $\Omega(n \log n)$ in any case), we omit this step.

Fixing the number of leftover edges to $\widetilde{O}(n)$, then [8] computes cycles of length $O(\log n)^{1/\delta-1}$ in time $2^{O(1/\delta)} \cdot n^{\delta} \cdot m$. In comparison, our algorithm computes cycles of length $2^{1/\delta} \cdot O(\log n)$ in roughly the same time $2^{O(1/\delta)} \cdot n^{\delta} \cdot m$. For example, when taking $\delta = 1/\log\log n$, both algorithms have roughly the same time complexity, but our algorithm produces cycles of length $O(\mathsf{poly}\log n)$ and their algorithm has cycle length $O(\log n)^{\log\log n}$.

From an algorithmic point of view, both our algorithm and the algorithm of [8] use low-diameter decomposition[10]. This allows one to restrict attention to $O(\log n)$-diameter graphs.

---

[8] the determinant of $\mathbf{L}$ with the last row and column removed

[9] In the output of such an algorithm we want each edge (u, v) to obtain a constant approximation for the effective resistance between u and v in G.

[10] We use a neighborhood covers which are close variant of low-diameter decomposition.

The approach of [8] contracts each of the components of the low-diameter decomposition and recursively computes vertex-disjoint short cycles on the contracted graph. The diameter of each super-node is $O(\log n)$, when lifting the contracted cycles back to edges in G the length of the cycles increases exponentially with the number of recursive layers. In particular, halving within $1/\delta$ recursive calls the length of the cycles becomes $O(\log n)^{1/\delta-1}$. Our approach is quite different. We also decompose trees into smaller components, but instead of contracting these components we use their internal edges carefully in our cycles. By enjoying the internal edges inside each cycle, the length of the cycles increases by a factor of at most 2 in each level of the recursion, thus after $1/\delta$ recursion levels, the length of the cycle is $2^{1/\delta}$.

2 Longer Edge-Disjoint Cycles in Nearly Linear Time

We begin by describing a deterministic algorithm for computing cycle decomposition of the same quality as that of Chu et al. [3], but in nearly linear time $\widetilde{O}(m)$. In particular, the cycle length will be bounded by $2^{\sqrt{\log n}}$ and at most $2^{\sqrt{\log n}} \cdot n$ edges will be left uncovered. Note that the recent randomized algorithm of [8] achieves such cycles in almost linear time $m \cdot n^{O(\log\log n/\sqrt{\log n})} \cdot 2^{\sqrt{\log n}/\log\log n}$. We can also reduce the number of leftover edges to $O(n)$, by running the algorithm of [8] on the remaining subset of $n^{1+o(1)}$ leftover edges. However, note that in any case, the efficiency of the algorithmic applications for these cycles depends on $\widehat{m} + nL$ where $\widehat{m}$ is the number of leftover edges and L is the largest cycle length.

▶ **Theorem 13.** *For every $\epsilon \in (0,1]$, there exists an $\widetilde{O}(m)$-time algorithm that computes an $(\widehat{m}, L)$ short cycle decomposition with $\widehat{m} = 1/\epsilon \cdot 2^{1/\epsilon} \cdot n^{1+O(\epsilon)}$ and $L = 2^{O(1/\epsilon)}$. Setting $\epsilon = 1/\sqrt{\log n}$, gives $\widehat{m} = 2^{O(\sqrt{\log n})} \cdot n$ and $L = 2^{O(\sqrt{\log n})}$.*

Thm. 4 follows immediately: Set $\epsilon = 1/\log\log n$ in Theorem 13, yeilding cycles of length $O(\log n)$ and $n^{1+o(1)}$ leftover edges. Then, using the randomized algorithm of [8] on this remaining subgraph with some constant $\delta \in (0,1]$, covers the remaining edges with cycles of length $O(\log n)^{1/\delta}$ with time complexity of $n^{1+1.1\delta}$. In addition, Thm. 3 follows by plugging $\epsilon = 1/\sqrt{\log n}$ in Theorem 13.

Throughout, a *block* is a subset of vertices with a bounded size. The algorithm is recursive and has $\ell = \lceil 1/\epsilon \rceil$ levels of recursion. During each recursion level, some virtual edges $\widetilde{E}$ will be added to the set of edges E' that we wish to cover by cycles (initially $E' = E$). Informally speaking, whenever the algorithm adds a virtual edge between two nodes u and v, it implies that the algorithm has already computed a u-v walk denoted by $W((u,v))$, and the virtual edge (u,v) indicates the need for computing another u-v walk so that we will end up with a cycle. In other words, adding a virtual edge means that we defer the closure of the cycle to future iterations. We also maintain a leftover subgraph H, and in certain cases, we give up on completing the cycles of the virtual edges, and add their walk edges to this subgraph.

We now describe Alg. FasterLongerCycles. The algorithm is recursive and has $O(1/\epsilon)$ levels of recursion. Initially, let $E' = E(G)$. The preliminary walk collection is $\mathcal{W} = \{e \mid e \in G\}$, the cycle collection $\mathcal{C}'$ is empty. In addition, we have a subgraph $H \leftarrow \emptyset$ that will contain the edges that are not covered by cycles.

In each independent level $i \geq 1$ of the recursion, we are given a block B, and a subset of edges E' with both endpoints in B. In addition, we are given a set of current walks $\mathcal{W}$ for the edges in E', a current cycle collection $\mathcal{C}'$, and a leftover subgraph H.

Block Partitioning. First the algorithm partitions B into $k = n^{\epsilon}$ balanced blocks $B_1, \ldots, B_k$ each with $\Theta(|B|/n^{\epsilon})$ vertices.

Taking care of edges between blocks. Our goal is to replace edges between blocks, to edges inside blocks. For block B_a, we do as follows for every $v \in B_a$ and every $b \in \{a+1, \ldots, k\}$. Define by $N_{a,b}(v) = \{u \in B_b \mid (u,v) \in E'\}$ to be the E'-neighbors of v in B_b. If $N_{a,b}(v)$ is odd, we will omit from it at most one vertex u, in order to make it even. The edge (u,v) of the omitted vertex u is omitted from E', and its walk $W((u,v))$ is added to the leftover subgraph H.

From now on, we can assume that the set $N_{a,b}(v)$ is even. We then (arbitrarily) match the vertices in $N_{a,b}(v)$ into pairs $\langle x,y \rangle$. Each matched pair $\langle x,y \rangle$ is handled as follows:

Case (1): The set E' already contains an (x,y) edge. If E' already contains an edge (x,y) (this edge might be virtual), we define a cycle $C = W((v,x)) \circ W((x,y)) \circ W((y,v))$ and add it to the cycle collection $\mathcal{C}'$. In addition, we omit the edges $(v,x),(x,y)$ and (y,v) from E', and omit their walks from $\mathcal{W}$.

Case (2): The set E' does not contain an (x,y) edge. In this case, we add a virtual edge (x,y) to E', as well as a walk $W((x,y)) = W((x,v)) \circ W((v,y))$ to $\mathcal{W}$. This completes the description of the i^{th} recursion level. The algorithm then recurses on each of the blocks $B_1, \ldots, B_k$. See Fig. 1 for pseudocode. Finally, as in Alg. PartialCycleCover, the cycles of $\mathcal{C}'$ might re-visit the same vertex, and hence in the final cleanup phase, the algorithm traverses each of the cycles in $\mathcal{C}'$ and simplify them.

Algorithm FasterLongerCycles$(B, E', \mathcal{W}, \mathcal{C}', H)$**.**
Level i of the Recursion (for non-singleton block B):

1. Decompose B into $k = n^{\epsilon}$ blocks $B_1, \ldots, B_k$ each with $|B|/k$ vertices.
2. For every block B_a and every vertex $v \in B_a$, do the following for every $b > a$:
   - **a.** Let $N_{a,b}(v) = \{u \in B_b \mid (u,v) \in E'\}$
   - **b.** If $|N_{a,b}(v)|$ is odd:
     - Omit an arbitrary u from $N_{a,b}(v)$.
     - Omit the walk $W((u,v))$ from $\mathcal{W}$ and the edge (u,v) from E'.
     - Add $W((u,v))$ to H.
   - **c.** Match the vertices in $N_{a,b}(v)$ into pairs $\langle x,y \rangle$ (in an arbitrary manner).
   - **d.** For each matched pair $\langle x,y \rangle$ do:
     - If $(x,y) \in E'$:
       - Add the cycle $W((v,x)) \circ W((x,y)) \circ W((y,v))$ to $\mathcal{C}'$.
       - Remove the edges $(v,x),(x,y),(y,v)$ from E', and their walks from $\mathcal{W}$.
     - Otherwise:
       - Add a virtual edge (x,y) to E', and add to $\mathcal{W}$ the x-y walk:

$$W((x,y)) = W((x,v)) \circ W((v,y)).$$

3. For every $a \in \{1, \ldots, k\}$ do:
   - Let E'_a be the edges in E' with both endpoints in B_a.
   - Let $\mathcal{W}_a = \{W(e) \in \mathcal{W} \mid e \in E'_a\}$.
   - Apply FasterLongerCycles$(B_a, E'_a, \mathcal{W}_a, \mathcal{C}', H)$.

Figure 1 Description of $n^{o(1)}$-length cycle decomposition in $\widetilde{O}(m)$ time.

Analysis. Let $E_i, \mathcal{W}_i$ be the union of the $E', \mathcal{W}$ sets over all the recursion calls in level i.

▷ Claim 14 (Cycle Length). (a) All walks added in level i have length $\leq 2^i$; (b) All cycles added in level i have length $\leq 2^{i+1}$.

Proof. Consider the first level for the base of the induction. Let $B_1, \ldots, B_k$ be the first level blocks. Fix a pair of blocks B_a, B_b for $a < b$, and $v \in B_a$. Let $\langle x, y \rangle$ be a matched pair in $N_{a,b}(v)$. First, assume that when considering $\langle x, y \rangle$, the current edge set E' does not contain (x, y). In such a case, we add an x-y walk $W((x, y)) = (x, v) \circ (v, x)$ of length 2 as required. Otherwise, E' already contains the edge (x, y) and by the explanation above, $|W((x, y))| \leq 2$. In such a case, we add a cycle $C = (v, x) \circ W((x, y)) \circ (y, v)$ which has length at most 4 as required.

Assume that the claim holds up to level $i-1$, and consider level i. Using the induction assumption, we can apply the same argument for the induction base and get that either: (1) we add an x-y walk $W((x, y)) = W((x, v)) \circ W((v, y))$. Note that by definition (x, v) and (y, v) are edges between blocks in level $i+1$. Thus if these edges are virtual, they must have been added in level $i-1$ (since all virtual edges added in level i connect vertices in the same $(i+1)$-level blocks). We have by induction assumption that $|W((x, v))|, |W((v, y))| \leq 2^{i-1}$. Thus $|(W((x, y))| \leq 2^i$. (2) Otherwise, if E' already contained the edge (x, y) when considering this matched pair, the algorithm adds a cycle $C = W((v, x)) \circ W((x, y)) \circ W((y, v))$. Note that the edge (x, y) could potentially be added in level-i (since it is inside an $(i+1)$-level block). Thus $|W((x, y))| \leq 2^i$ (by the previous case), and $|W((x, v))|, |W((v, y))| \leq 2^{i-1}$. Overall, $|C| \leq 2^{i+1}$. The claim follows. ◁

Since the algorithm has $\ell = O(1/\epsilon)$ levels, overall all cycles have length $2^{O(1/\epsilon)}$ as required.

Missing proofs appear in the full version of the paper.

▷ Claim 15. [Number of Uncovered Edges] $|E(H)| = 1/\epsilon \cdot 2^{1/\epsilon} \cdot n^{1+O(\epsilon)}$.

Proof. We bound the number of edges added to the leftover subgraph H due to a fixed vertex v. Since the blocks are vertex-disjoint at every recursion level, a vertex belongs to at most $O(1/\epsilon)$ blocks: at most one block, in each level in the recursion tree (once a vertex becomes a singleton block, we stop sub-dividing it). Consider level i, and let B_v be the unique block containing v. Recall that in this level, the input edge set E_i contains only edges whose both endpoints are in the same i-level block.

Now, the algorithm subdivides B_v into n^ϵ disjoint blocks: $B_1, \ldots, B_k$. W.l.o.g., let B_1 be the block containing the vertex v. For every other block B_j for $j \in \{2, \ldots, k\}$, we omit at most one edge $e_j \in E_i$ and add a walk $W(e_j)$ to H. By Claim 14, every walk $W(e_j)$ is of length at most $2^{O(1/\epsilon)}$. Therefore, there is a total of $k \cdot 2^{O(1/\epsilon)}$ edges on the walks $W(e_1), \ldots, W(e_k)$ that are added to H when considering v. Summing over all the vertices, and over all $O(1/\epsilon)$ recursion levels, we get that $|H| = 1/\epsilon \cdot 2^{1/\epsilon} \cdot n^{1+O(\epsilon)}$. ◁

Finally, we show that all the edges that are not in H are covered by the cycles in $\mathcal{C}$.

▷ Claim 16 (Cover). Every edge is either in H or covered by the cycles in $\mathcal{C}$.

Proof. We claim by induction on i, that every edge $e \in G \setminus H$, either has a walk $W(e')$ such that $e' \in E_i$ and $e \in W(e')$, or that e is covered by a cycle in $\mathcal{C}$. The base of the induction holds vacuously. Assume it holds for $i-1$ and consider level i. It remains to take care for edges $e \in G$ such that (i) there exists $e' \in E_{i-1}$ satisfying that $e \in W(e')$ and (ii) $e' \notin E_i$. To see this observe that if (i) does not hold, then the statement holds by induction assumption. If (i) holds but (ii) does not hold, then $e \in W(e')$ for an $e' \in E_i$ and the statement holds.

Consider then such an edge e that satisfies the above two conditions. Since e' was omitted from E_i in phase $i-1$, it implies that $e' = (u, v)$ was an edge between two i-level (brother) blocks B_a and B_b. W.l.o.g., $u \in B_a$ and $v \in B_b$. We first observe that if u has an odd

number of edges in E_i with a second endpoint of B_b, then the unique edges e'' omitted from consideration cannot be e'. This holds since when omitting e'', we add $W(e'')$ to H. Since $e \in W(e')$ but $e \notin H$, it must hold that $e' \neq e''$. From now on, we know that e' was matched to another u-edge (u, v'). In this case either an v-v' walk $W(\widehat{e})$ for $\widehat{e} = (v, v')$ is added to the walk collection, or that a cycle containing $W(e')$ is added to $\mathcal{C}$. In either case, since $e \in W(e')$, it is indeed covered by either the walks or the cycles in level-i. ◁

Setting $\epsilon = 1/\sqrt{\log n}$, yields cycles of length $2^{\sqrt{\log n}}$ and at most $|H| = 2^{O(\sqrt{\log n})} \cdot n$ edges. All other edges not in H are covered by a cycle.

Edge-Disjoint Cycles. We prove (1) every edge belongs to at most one walk in $\mathcal{W}_i$ for every i, and (2) cycles are made by gluing a disjoint set of walks. Claim (1) can be shown by induction on the set of walks $\mathcal{W}_i$. The base of the induction holds vacuously. Assume that it holds up to i and consider the walks added in level i. The walks are formed one by one, where walks of level i formed by gluing together walks in $\mathcal{W}_i$. Whenever a walk $W(e) = W(e') \circ W(e'')$ is formed, the walks $W(e'), W(e'')$ are omitted from the walk collection and would not be considered again. The claim follows by combining with the induction assumption. To see (2) observe that whenever we form a cycle, all its walks are omitted from the walk collection. The proof follows from the fact that all walks are edge-disjoint.

Time Complexity. We claim that each all recursion calls of level i can be implemented in $O(m)$ time, for every $i = 1, \ldots, \ell$. We claim that all operations are linear in m. We keep the block ID of each vertex v (the maximum vertex ID in its block) in each level $i \geq 1$. Then by traversing over the edges in E_i, we can compute the edges $E_{a,b}$ between each pair of bothering blocks B_a, B_b in level i. We traverse the edges in $E_{a,b}$ for each vertex v in B_a. All operations of gluing walks due to an addition of virtual edges are linear in the length of the walks. Since all walks are edge-disjoint, we touch each edge $e \in E$ at most $O(1)$ many times in each phase.

3 Shorter Cycles in Almost Linear Time

We next turn to consider the high-level idea of our main algorithm which computes a decomposition with almost optimal quality in almost-linear time. Thus establishing Theorem 2. Specifically, here the cycle length will be bounded by $O(\log^2 n)$, and we will omit at most $O(n \log n)$ edges. Note that the simple algorithm described above omits $2^{1/\epsilon} \cdot n^{1+\epsilon}$ edges which is at least $2^{\sqrt{\log n}} n$ for any value of ϵ.

One option to improve this bound is by setting $\epsilon = 1/\log\log n$ to get cycles of length $O(\log n)$, while omitting $n^{1+o(1)}$ edges. Then by applying the algorithm of [8] on the remaining edges with $\delta = 1/c$ for some constant c we get an algorithm for covering all but $O(n)$ edges with $\mathsf{polylog}(n)$ length cycles in total time $m + n^{1+\delta}$. Since δ is constant, this algorithm is not an almost linear algorithm for graphs with $m = n^{1+o(1)}$ edges. Therefore, in order to obtain a truely almost linear algorithm that that omits $\widetilde{O}(n)$ edges, runs in time $m^{1+o(1)}$ and produces cycles of length $\mathsf{polylog}(n)$, we must come up with some new ideas.

Our alternative algorithm is recursive and has $O(1/\epsilon)$ recursion levels. It is also based on a balanced partitioning into blocks, only that the blocks in this context are more involved. Instead of computing edge-disjoint cycles directly, we compute short cycles that have a small amount of overlap. This notion is captured by low-congestion cycle cover defined as follows.

Key Task:

- **Input:** Parameter $\epsilon \in (0,1)$, an n-vertex graph G of diameter $O(\log n)$ with m edges, and a BFS tree $T \subseteq G$.
- **Goal:** Cover *all* non-tree edges with cycles of length at most $\mathtt{d} = O(2^{1/\epsilon} \cdot \log n)$, such that each edge appears on at most $\mathtt{c} = 1/\epsilon \cdot n^{O(\epsilon)}$ cycles. That is, compute a $(\mathtt{d}, \mathtt{c})$ cycle cover for the non tree edges.

Figure 2 The key sub-problem for short cycle decomposition.

For a bridgeless graph $G = (V, E)$, a $(\mathtt{d}, \mathtt{c})$ cycle cover is a collection of cycles of length at most $\mathtt{d}$, such that each edge in G appears on at least one cycle and on at most $\mathtt{c}$ cycles[11]. Intuitively, if each edge appears on few cycles, then one can greedily pick a subset of edge-disjoint cycles which covers a large enough fraction of the edges, and then repeat again (after removing all edges that are currently covered). Moreover, we also show that computing the decomposition boils down into an even easier variant of the low-congestion cycle cover problem.

Note that the key task considers all graphs of diameter $O(\log n)$, whereas in our case the input graph G might have a large diameter. To add more insult to the injury, the low-congestion cover computation is computed repeatedly on the subset of yet uncovered edges (i.e., by the current subset of edge-disjoint cycles). Thus, even if the original input graph has a small diameter, already in its second application, the input graph to the algorithm might not be even connected. In the full version, we show how to settle down this mystery using the notion of neighborhood covers. From now on, we focus on the key task.

Solving the Key Task. Given a tree T and an $\epsilon \in (0,1]$, our goal now is to cover all non-tree edges $E \setminus T$ with cycles of length at most $\mathtt{d} = 2^{1/\epsilon} \cdot O(\log n)$, such that each edge appears on at most $\mathtt{c} = 1/\epsilon \cdot n^{\epsilon}$ cycles.

The algorithm is recursive with $\ell = O(1/\epsilon)$ recursion levels. In each independent level $i \in \{1, \ldots, \ell\}$ of the recursion, we are given a subtree T' and a collection of at most $m/n^{\epsilon \cdot (i-1)}$ edges E' with both endpoints in T' that should be covered. Some of the edges in E' (in level $i \geq 2$) might be virtual, and in such a case it implies that the algorithm has already computed a partial cycle (i.e., a walk) that covers them. We are also given a set of walks $\mathcal{W}$ that contains a walk $W(e)$ for each $e \in E'$. Initially, T' is simply T, E' contains all edges in $E(G) \setminus T$ that we want to cover, and $\mathcal{W} = \{e : e \in G\}$ contains the trivial walks for each edge.

Step (1): Balanced Block Partitioning. The first step of the algorithm is to partition T' into $k = \Theta(n^{\epsilon})$ edge-disjoint subtrees $T_1, \ldots, T_k$ that are *balanced* with respect to their degrees in E'. We call such balanced subtrees *blocks*. Any vertex whose degree in E' is too high defines a singleton block.

As in the previous algorithm, we will distinguish between two types of E'-edges: edges inside a block and edges between blocks. We next describe how to replace edges between blocks with virtual edges that are *inside a block*, in a way that covering the virtual edges by

[11] Our constructions do not require the graph to be bridgeless, it covers edges by cycles provided that such a cycle exists.

cycles will, later on, be translated back to a covering of the original inter-block edges. Unlike the previous algorithm, here we do not have the privilege to throw away edges to leftover subgraphs. Thus, we will have to make sure that all virtual edges are eventually completed into a cycle.

Step (2): Handling Edges Between Blocks. Let $E_{a,b}$ be all the edges in E' with one endpoint in T_a and the other in T_b. Our goal now is the following: we want to find a matching of the edges in $E_{a,b}$ in a way such that if $e = (x, y)$ and $e' = (x', y')$ where $x, x' \in T_a$ and $y, y' \in T_b$ are matched then there is a path $\pi(x, x')$ in T_a such that these paths for all pairs are edge disjoint. Then, we add the virtual edge $(y, y') \in T_b \times T_b$ and remove e and e'. Furthermore, we maintain the set of walks $\mathcal{W}$ connecting the endpoints of each virtual edge $\widehat{e} = (y, y')$ where $W(\widehat{e}) = e \circ \pi(x, x') \circ e'$.

Intuitively, the addition of a virtual edge (y, y') indicates to the algorithm that a cycle covering the edges e and e' is "under construction": there is currently an y-y' walk which will become a cycle when covering the virtual edge (y, y'). Importantly, all the virtual edges are internal to T_b and thus will be covered recursively by a path inside T_b. This path, along with $W(e)$ will complete the cycle. This procedure is applied for each pair of subtrees T_a, T_b separately, eventually converting all inter-block edges to internal block edges. Then, we apply the algorithm recursively on each block.

Notice that when the algorithm is applied recursively, then the inter-block edges e, e' might be virtual. Thus, we define the walks as follows. Initially, we set $W(e) = e$ for all edges of the graph and let $\mathcal{W} = \{W(e), e \in E'\}$. Then, we update $W(\widehat{e}) = W(e) \circ \pi(x, x') \circ W(e')$. That is, if e is a virtual edge then instead of adding it to the path we added its path $W(e)$ that contain "real" edges of the graph. This, of course, makes the walk longer and we bound their length later on.

We are left to describe how the matching is performed. As long as there is a vertex x in either T_a or T_b that is adjacent to at least two edges $E_{a,b}$, then we can match these two edges. That is, in this case, we have that $x = x'$ and thus the path $\pi(x, x') = x$ is the trivial path and is, of course, edge-disjoint from any other path. Thus, we can now assume that each vertex in T_a and T_b is adjacent to at most one edge in $E_{a,b}$. Note in the previous algorithm, when we got to a point that a vertex in a block is incident to one vertex in another block, we simply got rid of this edge by adding it to the leftover subgraph H. Here, we do not have this option, as we really need to cover *all* non-tree edges. This is exactly the point where congestion kicks in: covering all edges will come with the cost of producing cycles with some overlap, rather than edge-disjoint cycles as in the previous algorithm.

Let $M_{a,b} \subset T_a$ be all the vertices in T_i that are adjacent to an endpoint of an edge in $E_{a,b}$. If $|M_{a,b}|$ is odd, then we omit a single vertex y from this set, and cover the edge (x, y) by adding the "fundamental" cycle $C = W((x, y)) \circ \pi(x, y, T)$ to the cycle collection[12].

From now on, we assume that $M_{a,b}$ is even, and each $y \in M_{a,b}$ is incident to a unique edge in $E_{a,b}$ to be covered. The key tool used in this context is the following lemma:

▷ Fact 17. [12] Given a tree T and an even subset of marked vertices $M \subseteq V(T)$, one can compute a matching the vertices of M into pairs such that the collection of tree paths $\pi(x, y, T)$ over all the matched pairs are edge-disjoint.

We apply Algorithm DisjointMatching to the instance $T_a, M_{a,b}$. The output of this algorithm is a matching of the marked vertices $M_{a,b}$ into pairs $\langle x_i, y_i \rangle$, along with a collection of edge-disjoint paths in T_a connecting the matched pairs. The matched vertices naturally

[12] If e is an edge in G it is indeed a fundamental cycle.

define the matching between the remaining edges, along with edge-disjoint paths. This completes the high-level description of level i, while omitting some minor technical subtleties.

Why It Works. We give an overview of the analysis of this algorithm.

1. **Cycle lengths:** Let d_i be the maximum length of all walks at the beginning of level i of the recursion. In level i, the walks $W(\widehat{e}) = W(e) \circ \pi(x, x') \circ W(e')$ contain two $(i-1)$-level walks and a tree segment. Since the depth of the tree is $O(\log n)$, we get $d_{i+1} \leq 2d_i + O(\log n)$. Solving for $i = O(1/\epsilon)$, gives the desired cycle length of $2^{O(1/\epsilon)} \cdot \log n$.
2. **Congestion:** We first consider the congestion added when creating walks via the routing edge-disjoint matching algorithm. We claim that in every recursion level, the congestion on the tree edges is increased by (at most) an additive term of n^ϵ. The non-tree edges, in contrast, will belong to exactly one cycle (using a similar argument to the previous algorithm).
   The congestion argument works by induction as well. Assume by induction that every tree edge e appears at most c_{i-1} many times on all walks computed up to level i. In level i, every walk is of the form: $W(\widehat{e}) = W(e) \circ \pi(x, x') \circ W(e')$. That is, it has a tree segment $\pi(x, x')$ and two segments of an $(i-1)$-level walks. Since each $(i-1)$-level walk is added to at most one i-level walk (due to the matching step), the total congestion on the non-tree part of the i-level walks is kept the same. The increase in the congestion is then due to the tree segment $\pi(x, x')$. Recall that this tree segment is the outcome of applying the routing disjoint algorithm in some block T_a. For each application of this algorithm in T_a w.r.t to the edges of a fixed block T_b, the tree segments are disjoint. However, since the algorithm is applied in T_a for n^ϵ many times – per other block T_b, the total congestion on its tree edges is n^ϵ. Overall, we get that $c_i \leq c_{i-1} + n^\epsilon$. Solving for $i = O(1/\epsilon)$, gives the desired bound.
   Finally, we bound the congestion due to the fundamental cycles. Recall that whenever the number of edges between blocks T_a and T_b is odd, we cover a single edge with its fundamental cycle. Let T' be an i-level block and $T_1, \ldots, T_k$ be its children. The tree segment of the fundamental cycle of each edge between T_a and T_b is contained in T'. Since T' has $k = O(n^\epsilon)$ children, the tree edge appears on $n^{2\epsilon}$ cycles. Next, observe that since the blocks of each level are edge-disjoint, an edge appears on $O(1/\epsilon)$ many blocks over all, thus the total congestion added due to the fundamental cycles of the child components is $1/\epsilon n^{2\epsilon}$.
3. **Time Complexity:** In each phase, for each of the blocks T_a we have $k = O(n^\epsilon)$ applications of the routing disjoint matching algorithm. This algorithm can be implemented in linear time. Since the blocks are edge-disjoint, overall it takes $k \cdot \sum_a O(|T_a|) = m \cdot n^\epsilon$. Computing the walks defined by these matching outcome can be done in linear time in the total length of all i-level walks. Since each tree edge appears at most n^ϵ times on this walks (in each recursion level), the total length of the walks is $O(m + n^\epsilon \cdot n)$. Summing overall $O(1/\epsilon)$ recursion levels gives the desired bound. The pseudocode of the algorithm appears, illustrations, the complete analysis and the distributed implementation all appear in the full version of the paper.

References

1. Alexandr Andoni, Jiecao Chen, Robert Krauthgamer, Bo Qin, David P Woodruff, and Qin Zhang. On sketching quadratic forms. In *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science*, pages 311–319. ACM, 2016.
2. Joshua Batson, Daniel A Spielman, and Nikhil Srivastava. Twice-ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012.
3. Timothy Chu, Yu Gao, Richard Peng, Sushant Sachdeva, Saurabh Sawlani, and Junxing Wang. Graph Sparsification, Spectral Sketches, and Faster Resistance Computation, via Short Cycle Decompositions. In *59th Annual Symposium on Foundations of Computer Science, FOCS*. IEEE Computer Society, 2018.
4. Michael Dinitz, Robert Krauthgamer, and Tal Wagner. Towards Resistance Sparsifiers. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2015, August 24-26, 2015, Princeton, NJ, USA*, pages 738–755, 2015.
5. David Durfee, John Peebles, Richard Peng, and Anup B Rao. Determinant-preserving sparsification of SDDM matrices with applications to counting and sampling spanning trees. In *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*, pages 926–937. IEEE, 2017.
6. Alon Itai and Michael Rodeh. Covering a graph by circuits. In *International Colloquium on Automata, Languages, and Programming*, pages 289–299. Springer, 1978.
7. Arun Jambulapati and Aaron Sidford. Efficient n/ϵ Spectral Sketches for the Laplacian and its Pseudoinverse. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2487–2503. SIAM, 2018.
8. Yang P. Liu, Sushant Sachdeva, and Zejun Yu. Short Cycles via Low-Diameter Decompositions. *SODA*, 2019.
9. Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.
10. Merav Parter and Eylon Yogev. Distributed Computing Made Secure: A Graph Theoreric Approach. *SODA*, 2019.
11. Merav Parter and Eylon Yogev. Low Congestion Cycle Covers and Their Applications. *SODA*, 2019.
12. David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.
13. Daniel A Spielman and Shang-Hua Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 81–90. ACM, 2004.