Optimal Short Cycle Decomposition in Almost Linear Time *

Merav Parter Eylon Yogev†

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 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 runs in nearly linear time of $\widetilde{O}(m)$, and computes edge-disjoint cycles of length $O(\log n)$ and a leftover of $n^{1+o(1)}$ edges. For dense graphs with $n^{1+\delta}$ edges for some constant $\delta \in (0, 1]$, we provide a nearly optimal decomposition in nearly linear time.

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

*Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Emails: {merav.parter,eylon.yogev}@weizmann.ac.il.

†Supported in part by grants from the Israel Science Foundation grant no. 950/16.

Contents

1 Introduction **2**
- 1.1 Low-Congestion Cycle Covers . . . 3
- 1.2 Improved Graph Sparsification Algorithms via Short Cycles . . . 4
- 1.3 Distributed Implementation . . . 6

2 Preliminaries **7**

3 Almost Linear Leftover in Nearly Linear Time **7**

4 Nearly Optimal Cycles in Almost Linear Time **11**
- 4.1 A Reduction to Small Diameter Graphs . . . 12
- 4.2 Solving the Key Task . . . 14
- 4.3 Analyzing the Algorithm . . . 17

5 Distributed Computation **22**
- 5.1 Cycle Cover . . . 23

A Missing Details for Algorithm DistCycleCover **28**

1 Introduction

Short cycle decomposition introduced by Chu et al. [6] 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.*

In their recent paper, Chu et al. [6] 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. [6] 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, [6] 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 [6] 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 [12] 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 [6] 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 1 (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.*

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

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

We also have a much simpler algorithm that achieves the same quality of cycle decomposition[3] as by [6], only in $\widetilde{O}(m)$ time. An additional benefit of this algorithm is that it is deterministic.

Theorem 2 (Larger Leftover 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 $O(\log n)$ and $n^{1+o(1)}$ extra edges.*

The same algorithm of Thm. 2 can be used to obtain cycles of length $2^{O(\sqrt{\log n})}$ and $n^{1+O(1/\sqrt{\log n})}$ leftover edges in time $\widetilde{O}(m)$. The latter provides an improvement for approximating the determinant of the Laplacian. We can also combine with the algorithm of [12] 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 3 (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 [6, 12].

	Cycle Length	**#Uncovered Edges**	**Time**	**Type**
Chu et al. [6]	$n^{o(1)}$	$n^{1+o(1)}$	$m^{1+o(1)}$	Randomized
Liu-Sachdeva-Yu [12]	$O(\log n)^{1/\delta-1}$ 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	$O(\log n)$	$n^{1+o(1)}$	$\widetilde{O}(m)$	Deterministic
This Work	$O(\log n)^{1/\delta}$ for constant $\delta \leq 1$	$O(n)$	$m + n^{1+1.1\delta}$	Randomized

Our algorithms are based on independent ideas compared to [6, 12], which are related to the concept of *low-congestion cycle covers*.

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 [10] 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, [16] 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 2 (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.*

[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 [6] et al. it is $2^{(\log n)^{3/4}}$.

[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.

Low-congestion cycle covers provide the basic communication backbone in different settings of resilient distributed computation such as Byzantine fault model and secure computation [16, 15]. Whereas a-priori it is not clear that cycles of short length and small overlap exist, our main result in [16] 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, 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 [16] 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 [18] 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 [5] 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. [6] 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 [6].

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 [11] showed that when requiring the high probability guarantee for a *fixed* unknown vector, the size of the data structure can be made $\tilde{O}(n/\epsilon)$. 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 [7] who conjectured that resistance sparsifiers with $\widetilde{O}(n/\epsilon)$ edges always exist. The conjecture was indeed resolved by [6] using the tool of short cycle decomposition. By combining Theorem 1 and Theorem 2 with [6, Theorem 6.1] we get:

Theorem 4 (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 [12] that gives sparsifiers with $\widetilde{O}(n/\epsilon)$ edges; and (ii) the $m^{1+o(1)}$-time algorithms of [6, 12] 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. [6] with Theorem 1, we get:

Theorem 5 (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. [6] to construct a sparsification of Eulerian directed graphs. By plugging Theorem 1 in Theorem 5.1 of [6], we get:

Theorem 6 (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 $\widetilde{O}(m)$.*

[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.

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

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 4, 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 [6, Thm. 3.8] on H, we get:

Theorem 7 (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 $\widetilde{O}(m) + (n+t)n^{o(1)}\epsilon^{-1.5}$.*

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 [6], the running time bottleneck of the determinant estimation algorithm for Laplacians by Durfee et al. [8] 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 7 in Lemma B.1 of [6] yields:

Corollary 1 (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 1 (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 $\mathtt{d}, \mathtt{c} = 2^{O(\sqrt{\log n})}$.*

This improves upon the linear time algorithm of [16, 15]. By combining this algorithm with Luby-MIS algorithm [13], we get:

Theorem 8 (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.

[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.

Comparison to the work of Chu et al. [6] and Liu-Sachdeva-Yu [12]. We first observe that in [12], the number of leftover edges is $O(n)$, whereas in our case it is $O(n \log n)$. By applying the algorithm of [12] 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 [12]. 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 [12] 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 $O(\text{poly} \log n)$ 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 [12] use low-diameter decomposition[10]. This allows one to restrict attention to $O(\log n)$-diameter graphs. The approach of [12] 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 Preliminaries

Graph Notations. For a tree $T \subseteq G$, let $T(z)$ be the subtree of T rooted at z, and let $\pi(u, v, T)$ be the tree path between u and v, when T is clear from the context, we may omit it and simply write $\pi(u, v)$. Let P_1 be a u-v path (possibly $u = v$) and P_2 be a v-z path, we denote by $P_1 \circ P_2$ to be the concatenation of the two paths. The fundamental cycle $C_{e,T}$ of an edge $e = (u, v) \notin T$ is the cycle formed by taking e and the tree path between u and v in T_0, i.e., $C_{e,T} = e \circ \pi(u, v, T)$.

For $u, v \in G$, let $\text{dist}(u, v, G)$ be the length (in edges) of the shortest $u - v$ path in G. For every integer $i \geq 1$, let $\Gamma_i(u, G) = \{v \mid \text{dist}_G(u, v) \leq i\}$. When $i = 1$, we simply write $\Gamma(u, G)$. Let $\deg(u, G) = |\Gamma(u, G)|$ be the degree of u in G. For a subset of edges $E' \subseteq E(G)$, let $\deg(u, E') = |\{v : (u, v) \in E'\}|$ be the number of edges incident to u in E'. For a subset of nodes U, let $\deg(U, E') = \sum_{u \in U} \deg(u, E')$. For a subset of vertices $S_i \subseteq V(G)$, let $G[S_i]$ be the induced subgraph on S_i.

3 Almost Linear Leftover 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. [6], 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 [12] achieves such cycles in almost linear time

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

$m \cdot n^{O(\log\log n/\sqrt{\log n})} \cdot 2^{\sqrt{\log n}/\log\log n}$. We can also reduce the number of leftover edges to $O(n)$, by running the algorithm of [12] 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 9. *For every $\epsilon \in (0,1]$, there exists an $\widetilde{O}(m/\epsilon)$-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. 3 follows immediately: Set $\epsilon = 1/\log\log n$ in Theorem 9, yeilding cycles of length $O(\log n)$ and $n^{1+o(1)}$ leftover edges. Then, using the randomized algorithm of [12] 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. 2 follows by plugging $\epsilon = 1/\sqrt{\log n}$ in Theorem 9.

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 1 (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 2. *[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 1, 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 3 (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.

4 Nearly Optimal Cycles in Almost Linear Time

In this section, we present the main algorithm for computing nearly optimal short cycle decomposition[11]. For didactic reasons, our short cycle algorithm uses a low-congestion cover algorithm as a black-box. We note that one could instead directly modify the low-congestion cover algorithm to yield edge-disjoint cycles from first place, rather then cycles with small overlap, which are later modified into edge-disjoint. This direct modification of the low-congestion cycle cover algorithm improves the running time by a factor of $n^{o(1)}$, which is negligible in our context. For this reason, we prefer to take the modular approach, which highlights the connection between these two decomposition notions. We will then provide a key subroutine that is mutual for both of these notions.

Theorem 10. *There exists an algorithm, such that for every $\epsilon \in (0, 1)$, and an n-vertex graph G with m edges, computes an $(\widehat{m}, L)$-cycle decomposition in time $1/\epsilon \cdot m \cdot 2^{1/\epsilon} \cdot n^{O(\epsilon)}$, where $\widehat{m} = O(n \log n)$ and $L = O(2^{1/\epsilon} \cdot \log n)$. Setting $\epsilon = 1/\log\log n$ yields an $m^{1+o(1)}$-time algorithm which computes a $(\widehat{m}, L)$-cycle decomposition with $\widehat{m} = \widetilde{O}(n)$ and $L = \widetilde{O}(1)$.*

[11] In this work, log-n factors are negligible and hence for brevity, we may call short cycle optimal if length of cycles is $\widetilde{O}(1)$ and number of leftover edges is $\widetilde{O}(n)$. The reason is that these log-factors do not effect the size of the resulting spectral sparsifiers, which have log-n factors due to other steps.

In the next section, we will also present an alternative linear construction (up to $\text{polylog}(n)$-terms) that computes cycles of length $n^{o(1)}$ (same quality as in [6]).

From Low-Congestion Cycle Covers to Short Cycle Decomposition The core of our short cycle decomposition is an algorithm for constructing low-congestion cycle covers. Low-congestion cycle covers are relatively new notion introduced by us in [16] as a fundamental communication backbone for secure distributed computation [15]. This notion was introduced independently to the cycle decomposition. Whereas in [15], our focus was in showing the existence of optimal cycle covers – that enjoy short cycles and low-congestion, here our focus is in computation time. For the propose of short cycle decomposition, we will consider a relaxed version of cycle covers which covers all but $O(n \log n)$ edges in the graph.

Lemma 2. *Assume that there exists an algorithm $\mathcal{A}$ that runs in time τ and computes a $(\mathtt{d}, \mathtt{c})$-cycle cover $\mathcal{C}$ that covers all G edges except for at most $\widehat{m}$ edges. Then, there exists an algorithm $\mathcal{A}'$ that runs in time $\tau' = \widetilde{O}(\mathtt{d} \cdot \mathtt{c} \cdot \tau)$ and computes an $(O(\widehat{m}), \mathtt{d})$-cycle decomposition.*

Proof. The collection $\mathcal{D}$ of edge-disjoint (short) cycles are computed by applying algorithm $\mathcal{A}$ iteratively for $O(\mathtt{d} \cdot \mathtt{c})$ many iterations. In each iteration, the algorithm is given the current uncovered subgraph G' (possibly not connected), containing all edges that are not yet covered by the current collection of edge-disjoint cycles $\mathcal{D}$. Initially, $G' = G$ and $\mathcal{D} = \emptyset$. The algorithm then applies Alg. $\mathcal{A}$ in G' resulting in a $(\mathtt{d}, \mathtt{c})$-cycle cover collection $\mathcal{C}$ that covers all but $\widehat{m}$ edges in G'. It then greedily computes a maximal collection of edge-disjoint cycles from $\mathcal{C}$. These edge-disjoint cycles are added to the collection $\mathcal{D}$ and their edges are omitted from G'. This procedure repeats until $|G'| \leq \widehat{m}$.

We claim that the algorithm has $O(\mathtt{d} \cdot \mathtt{c} \cdot \log n)$ iterations by showing that the collection of edge-disjoint cycles computed in each iteration covers an $\Omega(1/(\mathtt{d} \cdot \mathtt{c}))$ fraction of the yet uncovered edges (i.e., in G'). Each cycle $C \in \mathcal{C}$ intersects with at most $|C| \cdot \mathtt{c}$ many cycles in $\mathcal{C}$. Thus, when we add the cycle C into the collection $\mathcal{D}$ we cover $|C|$ many edges. When we remove the $|C| \cdot \mathtt{c}$ cycles that intersect with C, we remove the cover of at $|C| \cdot \mathtt{c} \cdot \mathtt{d}$ many edges in G' that appear on these cycles. That is, at each iteration, the ratio between the number of edges we cover and the number of edges that the covering cycle removed is $1/\mathtt{dc}$.

Let x_i be the number of yet uncovered edges in iteration i. If $x_i \leq \widehat{m}$ then we can just add x_i to the set of uncovered edges and we are done. Otherwise, in the next iteration, we cover at least $(x_i - \widehat{m})/\mathtt{cd}$ edges. Thus,

$$x_{i+1} \leq x_i - (x_i - \widehat{m})/\mathtt{cd} = (1 - 1/(\mathtt{cd})) \cdot (x_i - \widehat{m}) + \widehat{m}.$$

Hence, after $\ell = O(\mathtt{c} \cdot \mathtt{d} \log n)$ iterations, it holds that $x_\ell \leq O(\widehat{m})$, and the claim follows. □

4.1 A Reduction to Small Diameter Graphs

We next show that the problem of computing a short cycle decomposition boils down into the computation of low-congestion cycle covers in graphs of diameter $O(\log n)$, namely, we focus on the following key problem.

Our reduction is formulated by the next Lemma.

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:** Construct a $(\mathtt{d},\mathtt{c})$-cycle cover for the non-tree edges $G \setminus E(T)$ in time $O(1/\epsilon(\mathtt{c}n + m))$ where $\mathtt{c} = 1/\epsilon \cdot n^{O(\epsilon)}$ and $\mathtt{d} = O(2^{1/\epsilon} \cdot \log n)$.

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

Lemma 3. *Let $\mathcal{A}$ be an algorithm that solves the Key Task above (Figure 2) in time τ. Then, there exists an algorithm $\mathcal{A}'$ that for every n-vertex graph G', computes a $(\mathtt{d},\mathtt{c})$-cycle cover for all but $\widehat{m} = O(n \log n)$ edges in G' in time $\tau' = \widetilde{O}(\tau)$.*

Proof. The *t-neighborhood cover* [2] of the graph G is a collection of connected subgraphs $G_1, \ldots, G_k$ each of diameter $O(t \cdot \log n)$ such that (i) for every vertex v, there exists a subgraph G_i that contains its entire t-neighborhood, and (ii) each vertex appears on $O(\log n)$ subgraphs.

The algorithm $\mathcal{A}'$ begins by computing 1-neighborhood cover using the (nearly) linear time deterministic sequential algorithm of Awerbuch et al. [3]. This decomposes G into a collection of subgraphs $G_1, \ldots, G_\ell$, each with diameter $O(\log n)$ and overlap $O(\log n)$. We process these subgraphs one by one. Consider the subgraph G_i and let $T_i \subseteq G_i$ be a BFS tree. Apply algorithm $\mathcal{A}$ on G_i, T_i and let $\mathcal{C}_i$ be the output $(\mathtt{d},\mathtt{c})$-cycle cover for the non-tree edges of $G_i \setminus T_i$. Note that G_i, T_i is a valid input for algorithm $\mathcal{A}$ since $\mathsf{Depth}(T_i) = O(\log n)$. Let $\mathcal{C}' = \bigcup_{i=1}^{\ell} \mathcal{C}_i$.

We now claim that except for $O(n \log n)$ edges, all edges in G are covered by the cycles of $\mathcal{C}'$. First, notice that from the definition of t-neighborhood cover it holds that $\bigcup G_i = G$. That is, every edge in G is contained in at least one of the G_i's. For each such G_i we cover all edges except the edges of T_i. Since each vertex v appears in $O(\log n)$ subgraphs G_i, we have that $|\bigcup_{i=1}^{k} T_i| = O(n \log n)$. Thus, we cover all edges expect at most $O(n \log n)$.

Finally, we analyze the running time. The construction of the neighborhood cover by Awerbuch et al. [3] takes $O(m \log n + \log^2 n)$ time. Let n_i and m_i be the number of vertices (resp., edges) in G_i. Applying algorithm $\mathcal{A}$ in each subgraph G_i takes time $\tau_i = O(\mathtt{c} \cdot n_i + m_i)$. Since each vertex appears in $O(\log n)$ clusters we have that $\sum n_i = O(n \log n)$ and that $\sum_i m_i = O(m \log n)$.

Thus, the total time complexity is bounded by

$$\begin{aligned}
\tau' &= O(m \log n + \log^2 n + \sum_{i=1}^{\ell} (\mathtt{c} \cdot n_i + m_i)) \\
&\leq O(m \log n + \log^2 n + \mathtt{c} \cdot n \log n + m \log n) \\
&= \widetilde{O}(m + \mathtt{c}n) = \widetilde{O}(\tau).
\end{aligned}$$

□

Putting Lemmas 2 and 3 together we get an algorithm, ImprovedShortCycleDecomp, that constructs a short cycle decomposition based on a procedure PartialCycleCover that computes a low-congestion cycle covers for small depth graphs. The pseudocode for this algorithm is given in Figure 5.

Algorithm ImprovedShortCycleDecomp$(G = (V, E), \epsilon)$

1. Set $\mathsf{c} = 2^{1/\epsilon} \cdot n^{O(\epsilon)}, \mathsf{d} = 2^{O(1/\epsilon)} \cdot \log n$, $G' \leftarrow G$, $\mathcal{D} \leftarrow \emptyset$.
2. Repeat for $\widetilde{O}(\mathsf{c} \cdot \mathsf{d})$ times:
   (a) Construct 1-neighborhood cover on G' resulting in subgraphs $G_1, \ldots, G_k$.
   (b) Let $T_1, \ldots, T_k$ be $O(\log n)$-depth spanning trees in $G_1, \ldots, G_k$.
   (c) Apply Alg. PartialCycleCover(G_i, T_i) to construct an (c, d)-cycle cover $\mathcal{C}_i$ for the non-tree edges $G_i \setminus T_i$ for every $i \in \{1, \ldots, k\}$.
   (d) Greedily pick a maximal set of edge-disjoint cycles in $\bigcup_{i=1}^{k} \mathcal{C}_i$ and add it to $\mathcal{D}$.
   (e) Remove all the covered edges (i.e., edges of the cycles in $\mathcal{D}$) from G'.

Figure 3: Procedure for short cycle decomposition

Due to Lemma 3, we restrict our attention to a graph G with diameter $O(\log n)$, a BFS tree $T \subseteq G$ and show how to cover its non-tree edges fast, using short cycle and low-congestion.

4.2 Solving the Key Task

In this subsection, we show how to solve the Key Task (Figure 2).

Lemma 4 (Covering non-tree edges)**.** *For every $\epsilon \in (0, 1)$, given an n-vertex (connected) graph G and a spanning tree $T \subseteq G$, there exists an algorithm, namely, Algorithm* PartialCycleCover(G, T, ϵ)*, that computes a (d, c) cycle cover that covers all non-tree edges with $\mathsf{d} = 2^{O(1/\epsilon)} \cdot \mathsf{Diam}(T)$ and $\mathsf{c} = n^{O(\epsilon)}$. The time complexity of the algorithm is $O(1/\epsilon \cdot (\mathsf{c} \cdot n + m))$.*

Algorithm PartialCycleCover is based on the following two core subroutines that can be computed in linear time.

Piece 1: Routing Disjoint Matching. In the *routing disjoint matching* problem (see Lemma 4.3.2 [17]), given is a rooted tree T and a set of $2k$ marked nodes $M \subseteq V(T)$ for $k \leq n/2$. The goal is to compute a matching of these vertices such that the tree paths connecting each matched pair are *edge-disjoint*. Missing proofs in this section are deferred to **??**.

Lemma 5. *There exists an algorithm,* DisjointMatching*, that given an instance $\langle T, M \rangle$ solves the Routing Disjoint Matching problem in time $O(|T|)$.*

Piece 2: Balanced Block Decomposition. Our second tool is a decomposition of the tree T into edge-disjoint sub-trees that are, roughly speaking, balanced in terms of the number of edges that they are incident to. Let $E' \subseteq E(G)$ be a subset of edges and let $T' \subseteq T$ be subtree with root r. We define the *density* of T' with respect to E', denoted by density(V', E'), by the number of edges in E' that have at least one endpoint in $T' \setminus \{r\}$. That is,

$$\mathsf{density}(T, E') = |\{e \in E' : e \text{ has an endpoint in } V(T') \setminus \{r'\}\}| \,.$$

The reason why we omit the root will become clear later.

Definition 3 (Balanced Tree Decomposition)**.** *Given a tree T, a subset of non-tree edges E' and a density threshold Δ, a balanced block decomposition requires to decompose T into k edge-disjoint subtrees $T_1, \ldots, T_k$, such that*

1. $\bigcup_i V(T_i) = V$[12].
2. $\forall i \in [k]$: *if* $|T_i| > 1$ *then* $\mathsf{density}(T_i, E') \in [\Delta/2, \Delta]$.
3. *The number of blocks is* $k = O(|E'|/\Delta)$.

We refer to these bounded density subtrees as blocks.

Lemma 6. *Given a tree T, subset of non-tree edges E' and density threshold Δ, there is an algorithm that computes the block partitioning in time $O(|E'| + |T|)$.*

Note that we can immediately define each heavy node (with E'-degree larger than Δ) as a singleton block. There could be at most $O(|E'|/\Delta)$ such blocks. In addition, in our partitioning algorithm, the root of each block T will also become a singleton block, and it will appear in T only to provide connectivity to this block. To see why we do that, consider a heavy node with many light children. To keep the number of blocks small we want all light children to be in the same block, and thus we need to add their parent to this block. This root in such a case is important only to provide connectivity for the block. For this reason, the measure $\mathsf{density}(T, E')$ does not include the root of T, as these roots appear as singleton blocks.

The Main Algorithm Recall that the input for the algorithm is a parameter $\epsilon \in (0, 1)$, and a graph G with a spanning tree $T \subseteq G$ of diameter $O(\log n)$. Our goal is to compute a $(\mathtt{d}, \mathtt{c})$-cycle cover $\mathcal{C}$ for covering the non-tree edges $E' = E(G) \setminus T$ with dilation $\mathtt{d} = O(2^{1/\epsilon} \cdot \log n)$ and congestion $\mathtt{c} = O(1/\epsilon \cdot n^{2\epsilon})$. The desired running-time is $O(1/\epsilon \cdot (\mathtt{c} \cdot n + m))$ where $m = |G|$. Algorithm PartialCycleCover is recursive and has $\ell = \Theta(1/\epsilon)$ levels of recursion.

In every independent level $1 \leq i \leq \ell - 1$ of the recursion, the input of the algorithm is as follows: a subtree T', the current collection of non-tree edges E' to be covered (with both endpoints in $V(T')$), a set of walks $\mathcal{W}$, and the current cycle collection $\mathcal{C}'$. The set of edges E' to be covered might contain virtual edges that are not in G. In addition, the given set of walks $\mathcal{W}$ contains an x-y walk in G for every edge $e = (x, y) \in E'$. Initially, $T' = T$, $E' = E(G) \setminus T$, $\mathcal{W} = \{\{e\} \in E'\}$, and $\mathcal{C}' = \emptyset$.

Step (1): Partitioning into Balanced Blocks. First, the algorithm partitions T' into balanced blocks using the algorithm of Lemma 6 with parameter $\Delta_i = 16\Delta_{i-1}/n^{\epsilon}$ where $\Delta_0 = |E(G) \setminus T|$. The output of this decomposition is a collection of k edge-disjoint blocks, $T_1, \ldots, T_k$, such that

$$\forall a \in [k] : \mathsf{density}(T_a, E') \in [\Delta_i/2, \Delta_i] \ .$$

In the analysis section, we will show that $k \leq n^{\epsilon}$. Note that the blocks are not vertex-disjoint, and two blocks T_a, T_b might share a common root. This root forms its own block. Throughout, when considering edges incident to a block T_a with root r_a, we refer to the edges that are incident to $V(T_a) \setminus \{r_a\}$. Since blocks can only share a common root, the internal vertices of the blocks are disjoint. The goal of the recursion call is to convert the inter-block edges between T_a, T_b for every $a \neq b \in \{1, \ldots, k\}$ to virtual edges inside these blocks.

[12]Note that we do not require that $\bigcup_i E(T_i) = E(T)$.

Step (2): Handling Edges Between Blocks via Routing Disjoint Matching. For a fixed pair of blocks T_a, T_b, we will show how to replace the edges between T_a and T_b with edges inside either T_a or T_b. The same procedure will be repeated for all pairs of blocks. Let $E_{a,b}$ be all the edges in E' with one endpoint in $V(T_a) \setminus \{r_a\}$ and the other in $V(T_b) \setminus \{r_b\}$ where r_a, r_b are the roots of $T_{a,b}$ respectively. When referring to an edge between blocks T_a and T_b we mean an edge between a vertex $u \in V(T_a) \setminus \{r_a\}$ and $v \in V(T_b) \setminus \{r_b\}$.

Assume, without loss of generality, that $|E_{a,b}|$ is even, as otherwise we simply cover a single edge in $E_{a,b}$ by taking its fundamental cycle and add it $\mathcal{C}'$.

Our goal now is the following: we want to find a matching of the edges[13] in $E_{a,b}$ in a way such that if two edges $e = (x, y)$ and $e' = (x', y')$ are matched together, where $x, x' \in T_a$ and $y, y' \in T_b$, then there is a path $\pi(x, x')$ in T_a such that these paths, for all matched pairs, are edge-disjoint. Then, we add the virtual edge $(y, y') \in T_b \times T_b$ to E', and remove e and e' from E'. For simplicity now, assume that we add a virtual edge (y, y') only once, in fact, it might be the case there is also a real G-edge (y, y'). This assumption is only for simplifying the explanation and we soon explain how to handle the case where same edge (y, y') is added several times. Furthermore, we will add to the walk collection $\mathcal{W}$ an $y - y'$ walk $W(\widehat{e})$ for every virtual edge $\widehat{e} = (y, y')$, where $W(\widehat{e}) = W(e) \circ \pi(x, x') \circ W(e')$. Notice that the inter-block edges e, e' might be virtual (unless it is the first recursive call). In such a case, as will be shown in the analysis, the input walk collection $\mathcal{W}$ already contains the walks $W(e), W(e')$ connecting the endpoints of these edges. See Fig. 4 for an illustration.

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 in $E_{a,b}$, 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. See Fig. 4. Thus, we can now assume that each vertex in T_a and T_b is adjacent to at most one edge in $E_{a,b}$.

Let $M_{a,b} \subset T_a$ be all the vertices in T_a that are adjacent to an endpoint of an edge in $E_{a,b}$. Assume, without loss of generality, that $|M_{a,b}|$ is even, as otherwise we cam omit a single vertex from $M_{a,b}$, and simply cover its unique edge in $E_{a,b}$ by taking its fundamental cycle and add it $\mathcal{C}'$. We then apply Algorithm DisjointMatching (see Lemma 5) 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 define the matching between the remaining edges, along with edge-disjoint paths.

Finally, we consider the case where when the edges $e = (x, y)$ and $e' = (x', y')$ are matched and the current edge set E' already contains an edge $\widehat{e} = (y, y')$. This case is actually simpler: we define a cycle $C = W(e) \circ W(\widehat{e}) \circ W(e')$ and add it to the cycle collection $\mathcal{C}'$. We then omit the edges e, e' and $\widehat{e}$ from E' and omit their walks from $\mathcal{W}$. This completes the description of level i in the recursion. After $\ell = \Theta(1/\epsilon)$ levels, all leaf blocks in the recursion tree are singletons.

Final Cleanup. In Corollary 2 we show that every edge appears at most once on cycle collection $\mathcal{C}'$, but it is still possible that a given vertex appears multiple times on a given cycle. The algorithm iterates over the cycles $\mathcal{C}'$ one by one. For each cycle $C \in \mathcal{C}'$, it traverses the vertices in order and once it arrives at a vertex that appears twice, a new cycle can be cut from C and be added to $\mathcal{C}$. This completes the description of the algorithm. See Figure 5 for pseudocode. We next turn to prove Lemma 4 by analyzing the algorithm.

[13] Here by matching we mean partitioning the edges into disjoint pairs, i.e., this is not a graphical matching.

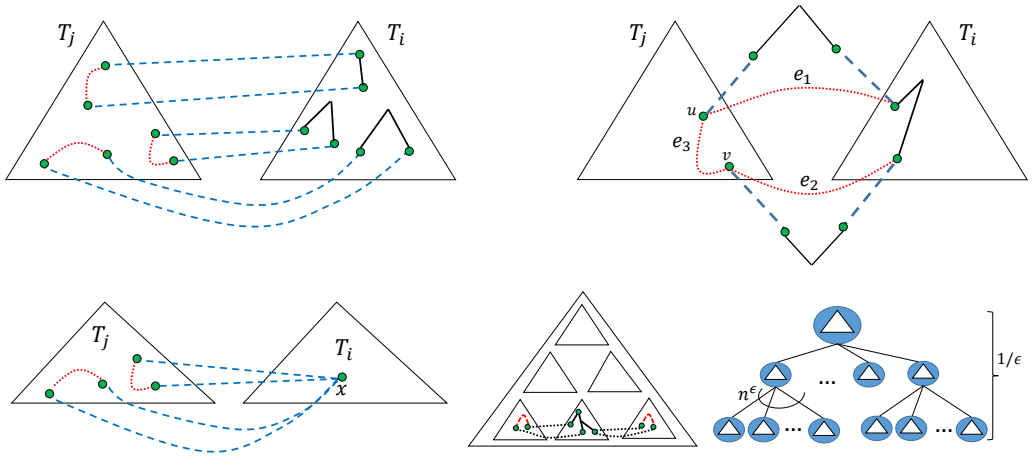

Figure 4: **Top Left**: Converting inter-block edges to edges inside blocks (first recursion level). Virtual edges are denoted in red and real non-tree edges in G are denoted in blue. By computing route disjoint matching in T_a, the inter-block edges are matched into pairs. The black paths are the edge-disjoint paths inside T_a. The red dashed edges are the newly introduced virtual edges in T_b. When covering these virtual edges by cycles, the original inter-block edges get covered by cycle as well. **Top Right**: In the subsequent recursion levels (for instance the second level), the edges between blocks might be virtual, in such a case, the algorithm has already computed a walk for these edges. **Bottom Left**: Shown is a vertex that has several inter-block edges in the same block. In such a case, those edges are matched in any arbitrary manner, leaving at most one edge that will be handled separately. **Bottom Right**: Schematic view of the recursive process. The root vertex corresponds to the original tree T, each level partitions T into n^ϵ balanced blocks, thus the depth of the tree is $O(1/\epsilon)$. In the bottom layer, the density of each block is sufficiently small, thus all intra-block edges are covered by taking their fundamental cycles in T.

4.3 Analyzing the Algorithm

Let $E_i, \mathcal{W}_i$ be the union of the *input* sets $E', \mathcal{W}$ over all level i recursion calls. We start with some auxiliary claims to be shown on induction on the recursive level.

Claim 4. *(1)* $\deg(v, E_i) \leq \deg(v, E_{i-1})$ *for every* $v \in V$ *and* $i \geq 2$.
(2) For every level-i block T'*: (a)* $\deg(T', E_i) \leq \Delta_{i-1}$ *and (b) by applying Lemma 6 with parameter* Δ_i*, it is partitioned into at most* $k = O(n^\epsilon)$ *blocks* $T_1, \ldots, T_k$ *such that* $\deg(T_a, E_i) \leq \Delta_i$ *for every* $a \in \{1, \ldots, k\}$.

Proof. (1) Recall that in level $i-1$, given the edges E_{i-1}, the algorithm adds virtual edges to the new set E_i and in addition, removes all the edges between the blocks. We will show that the degrees are not increased upon any addition of a virtual edge. When adding a virtual edge (x, y) in level-$(i-1)$ to the set E_i, it implies that there are two matched edges (x, x') and (y, y') in E_{i-1} which are omitted from E_i. In such a case, $\deg(v, E_i) \leq \deg(v, E_{i-1})$ for $v \in \{x, x', y, y'\}$ with equality holds for x and y. Since each virtual edge added to E_i corresponds to a *distinct* pair of edges in E_{i-1}, Claim (i) holds.

Algorithm PartialCycleCover$(T', E', \mathcal{W}, \mathcal{C}')$
Level i of the Recursion (for non-singleton T'):

1. Decompose T' (Lemma 6) into Δ_i-balanced blocks $T_1, \ldots, T_k$ with respect to E', where
$$\Delta_i = \Delta_{i-1}/n^\epsilon, \text{ and } k = \Theta(n^\epsilon).$$
2. For each pair of blocks T_a, T_b with roots r_a, r_b, where $a < b$ do:
   (a) $E_{a,b} \leftarrow E' \cap ((T_a \setminus \{r_a\}) \times (T_b \setminus \{r_b\}))$.
   (b) While $\exists (x,y),(x,y') \in E_{a,b}$ such that $y \neq y'$ then add a virtual edge (y,y') to E' and remove $e = (x,y)$ and $e' = (x,y')$. Remove $W(e), W(e')$ from $\mathcal{W}$.
   (c) Let $M_{a,b}$ be the vertices in T_a with one endpoint in T_b (with respect to E').
   (d) If $|M_{a,b}|$ is odd, omit from it one vertex, say x', and for its unique edge $(x,' y) \in E'$ add the fundamental cycle $\pi(x',y') \circ (x,' y)$ to $\mathcal{C}'$.
   (e) Apply Algorithm DisjointMatching (Lemma 5) with the input $T_a, M_{a,b}$.
   (f) For every matched pair x, x' do:
      i. Let $e = (x,y), e' = (x',y') \in E_{a,b}$ be the unique edges incident to x, x' in $E_{a,b}$.
      ii. If there exists an edge (y,y') in E':
         - Add the cycle $W(e) \circ W((y,y')) \circ W(e')$ to $\mathcal{C}'$.
         - Omit $e, e', (y,y')$ from E' and omit the walks $W(e), W(e'), W((y,y'))$ from $\mathcal{W}$.
      iii. Otherwise:
         - Add the virtual edge $\widehat{e} = (y,y')$ to E'.
         - Add the walk $W(\widehat{e}) = (y,x) \circ \pi(x,x') \circ (x',y')$ to $\mathcal{W}$.
         - Omit e, e' from E' and $W(e), W(e')$ from $\mathcal{W}$.
3. For every T_a, $a \in \{1, \ldots, k\}$:
   - Let E'_a be the edges in E' with both endpoints in $V(T_a)$.
   - Let $\mathcal{W}_a = \{W(e) \in \mathcal{W} \mid e \in E'_a\}$.
   - PartialCycleCover$(T_a, E'_a, \mathcal{W}, \mathcal{C}')$

Figure 5: Description of the partial covering algorithm

Claim (2): Recall that $\Delta_i = 16\Delta_{i-1}/n^\epsilon$. For every $i \geq 1$, we will show by induction that these properties hold for a level-i tree T'. For $i = 1$, the claim holds trivially by Lemma 6. Assume that both claims hold up to level $i-1$ and consider a level i.

Let T' be an i-level block. By induction assumption for $i-1$, $\deg(T', E_{i-1}) \leq \Delta_{i-1}$. By Claim (1), it also holds that $\deg(T', E_i) \leq \Delta_{i-1}$. In level i, we apply the partitioning lemma 6 with the inputs T', E_i and Δ_i. Note that we only care for the E_i edges that have at least one endpoint in T' and there are $\deg(T', E_i)$ such edges. As a result, we get $T_1, \ldots, T_k$ blocks such

that $\deg(T_a, E_i) \leq \Delta_i$ for every $a \in \{1, \ldots, k\}$, and in addition

$$k = O(\deg(T', E_i)/\Delta_i) \leq O(\Delta_{i-1}/\Delta_i) \leq O(n^\epsilon).$$

The claim follows. □

Cycle Length. We show by induction that for every virtual edge $e = (y, y')$ added in level-i, there is an y-y' walk $W(e) \in \mathcal{W}_{i+1}$ of length at most $\mathtt{d}_i = 2\mathtt{d}_{i-1} + \mathsf{Diam}(T)$, where $\mathtt{d}_0 = 1$.

For $i = 1$, when adding a virtual edge $\widehat{e} = (y, y')$ due to matching of $e = (x, y)$ and $e' = (x', y')$, there is an y-y' path $W(\widehat{e}) = (y, x) \circ \pi(x, x') \circ (x', y')$ of length $|W(\widehat{e})| = 2 + |\pi(x, x')| \leq 2 + \mathsf{Diam}(T)$ as required (see the Top-Right figure of 4).

Assume that the claim holds up to the $(i-1)^{th}$ level, and consider level i. Let $e = (x, y)$ and $e' = (x', y')$ be a matched pair. For the virtual edge $\widehat{e} = (y, y')$, there is an y-y' walk $W(\widehat{e}) = W(e) \circ \pi(x, x') \circ W(e')$. Either e is a real edge in G, or e is a virtual edge introduced in level $j \leq i-1$. In the latter case, by induction assumption, there is an x-y walk $W(e) \in \mathcal{W}_j$ of length at most $\mathtt{d}_j$. The same argument holds for e'. Thus, we get that $|W(\widehat{e})| \leq 2\mathtt{d}_{i-1} + \mathsf{Diam}(T) = \mathtt{d}_i$ as required. Since there are $\ell = O(1/\epsilon)$ iterations, we get that length of the walks is bounded by $\mathtt{d}_\ell = 2^{O(1/\epsilon)} \cdot \log n$ as $\mathsf{Diam}(T) = O(\log n)$. The walks $W(\widehat{e})$ are closed into cycles in either of the two ways: either by taking their fundamental cycle $\pi(y, y') \circ W(\widehat{e})$ or by gluing three walks together $W(e) \circ W(\widehat{e}) \circ W(e')$. Finally, note that making the cycles simple can only shorten them, thus keeping the final length bound to $2^{O(1/\epsilon)} \cdot \log n$, as required.

Congestion. Throughout, for a given collection of walks $\mathcal{W}$, the congestion of an edge e w.r.t. $\mathcal{W}$ is the total number that e appears on each of the walks in $\mathcal{W}$ (i.e., counting multiplicities on a given walk).

Claim 5. *(1) For every $i \in \{1, \ldots, \ell\}$, each tree edge $e \in T$ has congestion at most $\mathtt{c}_i = \Theta(n^\epsilon) + \mathtt{c}_{i-1}$ with respect to the walks of $\mathcal{W}_i$, $\mathtt{c}_1 = 1$. (2) The congestion of each non-tree edge w.r.t $\mathcal{W}_i$ is at most 1.*

Proof. First observe that throughout it is sufficient to bound the congestion on the tree edges for non-singleton blocks, thus for blocks bounded density. (The singleton blocks have no tree edges, and consists of a single vertex). We prove this by induction on i. Since $\mathcal{W}_1$ contains the trivial 1-length walks, the two claims hold vacuously. Assume that both claims hold up $i-1$ and consider level i with its input walk set $\mathcal{W}_i$. Every walk $W(\widehat{e})$ added in level $i-1$ has the following form: $W(\widehat{e}) = W(e) \circ \pi(x, y) \circ W(e')$, where $\widehat{e} = (y, y')$, $e = (x, y)$ and $e' = (x', y')$. Specifically, e, e' are edges between two blocks of level i that got matched together. These walks $W(e), W(e')$ were added in some level $j \leq i-2$. Since all inter-block edges are matched into pairs[14], each inter-block edge appears on at most *one* of the walks added in level i. For every such walk $W(\widehat{e}) = W(e) \circ \pi(x, y) \circ W(e')$, it is convenient to view it as a union of two types of edges: type (i) are the edges in $W(e), W(e')$ and type (ii) are the edges on the tree segment $\pi(x, y)$. Since each of the walks added in level $j \leq i-2$ participates in at most one of the walks of level-i, each edge $e'' \in G$ appears overall $\mathtt{c}_{i-1}$ many times on the type (i) segments on the $\mathcal{W}_i$ walks. It remains to bound the number of appearances on the type (ii) segments.

These type (ii) segments are added when computing the routing disjoint matching in blocks. Since all $(i-1)$-level blocks are edge-disjoint, it is sufficient to consider one such block T'. In

[14] except for those whose fundamental cycle is added directly

level-$(i-1)$ it is partitioned into $k = \Theta(n^\epsilon)$ blocks $T_1, \ldots, T_k$. Since the algorithm applies the routing disjoint matching on each block T_a for $O(n^\epsilon)$ many times (i.e., for each block T_b for $a \neq b$), a given tree edge $e'' \in T_a$ appears in the $\pi(x,y)$ segment of at most on walk for each T_b. The reason is that by applying Alg. DisjointMatching for a fixed pair T_a, T_b, all tree paths between the matched vertices in $M_{a,b}$ are *edge-disjoint*. Thus an edge $e \in T_a$ appears at most one of the output tree paths per application the routing disjoint matching algorithm. Since there are k such applications, e appears on the type (i) segments of $O(n^\epsilon)$ many walks in level-i. Overall, e appears at most $\mathsf{c}_{i-1} + \Theta(n^\epsilon) = \mathsf{c}_i$ times on the walks added in level-i.

We next prove (2). By induction assumption, a non-tree edge has a congestion of at most 1 w.r.t. the walks $\mathcal{W}_{i-1}$ walks. When adding a walk $W(\widehat{e}) = W(e) \circ \pi(x,y) \circ W(e')$ in level $i-1$, the walks $W(e), W(e')$ are omitted from the walk collection, and since each $(i-1)$-level walk is used for defining at most one i-level walk, the claim follows. □

Since there are $(1/\epsilon)$-levels of recursion, the maximum congestion of the tree-edges by the walks bounded by $O(1/\epsilon \cdot n^\epsilon)$. We proceed with bounding the congestion for due to adding the fundamental cycles $C = \pi(x,y) \circ W(e)$ for an edge e, where e is possibly a virtual edge.

Congestion argument for the fundamental cycles. We conclude the congestion argument by bounding the congestion due to the fundamental cycles added throughout the recursive procedure. Recall that when considering the $(i+1)$-level blocks $T_1, \ldots, T_k$ children T', the algorithm adds at most the fundamental cycle for at most one (possibly a virtual) edge, between each pair of blocks T_a and T_b. Due to Cl. 5, it is sufficient to bound the congestion on the tree-path part $\pi(x,y)$ of every fundamental cycle $\pi(x,y) \circ W(e)$, for $e = (x,y)$.

Claim 6. *Each edge $e \in T$ appears on the tree-segment of at most $O(1/\epsilon \cdot n^{2\epsilon})$ fundamental cycles.*

Proof. Recall that the i-level blocks are edge-disjoint, and thus each tree edge $e' \in T$ appears on at most $\ell = O(1/\epsilon)$ blocks overall (one block in each recursion level), see Fig. 4 (Down-Right).

We now claim a tree edge e belonging to a level i tree T' can appear on the tree-path segment at most $O(n^{2\epsilon})$ fundamental cycles added in level-i for every $i \leq \ell - 1$. Let $T_1, \ldots, T_k$ be the $(i+1)$-level children blocks of T' where $k \leq n^\epsilon$. For each pair of blocks T_a, T_b, the algorithm might add at most *one* fundamental cycle to $\mathcal{C}'$. This happens when the marked set $M_{a,b}$ is even, and one vertex x is omitted from this set. Note that for every edge $(u,v) \in T_a \times T_b$, the tree path $\pi(u,v,T)$ is contained in T'. Thus there are at most k^2 edges (one per pairs of blocks T_a, T_b) with endpoints in T' whose fundamental cycle is added to $\mathcal{C}'$ in level i. This creates a congestion of $k^2 = O(n^{2\epsilon})$ on the edges of the i-level block T'. Since e appears in at most ℓ blocks, one block per level i, the total congestion due to fundamental cycles added in levels $1, \ldots, \ell - 1$, it at most $O(1/\epsilon \cdot n^{2\epsilon})$.

□

In fact, we can also make a stronger argument and show that each edge appears at most one time on each of the walks. Thus the walks are simple with respect to edges but might re-visit the same vertex several times. Let $\widetilde{E}_i$ be the union of all virtual edges added in the recursion calls for level i.

Claim 7. *(i) All virtual edges $e, e' \in \widetilde{E}_i$ whose both endpoints are in the same $(i+1)$-level block T' have edge-disjoint walks $W(e)$ and $W(e')$. (ii) In addition, $|W(e) \cap E(T')| = \emptyset$ for every $e \in \widetilde{E}_i$.*

Proof. For the base of the induction consider $i = 1$. Let $T_1, \ldots, T_k$ be the 2-level block partitioning of T. Let e, e' be two virtual edges e, e' with both endpoints in the same T_a. Let T_{b_1} (resp., T_{b_2}) be the block such that e (resp., e') was added when running the routing disjoint matching algorithm on the instance T_{b_1}, M_{a,b_1} (resp., T_{b_2}, M_{a,b_2}). There are two options. If $b_1 = b_2$, then by the properties of the routing disjoint matching algorithm, $W(e)$ and $W(e')$ are edge-disjoint. Alternatively, if $b_1 \neq b_2$, then $W(e) \subseteq T_{b_1}$ and $W(e') \subseteq T_{b_2}$ and the induction base for claim (i) follows. Since T_a, T_{b_1}, T_{b_2} are edge-disjoint we have that $W(e) \cap T_a = \emptyset$ as required.

Assume that the claim holds up to $i-1$ and consider the edges added in level-i. Let T' be an i-level block and $T_1, \ldots, T_k$ be its $(i+1)$-level children. Consider two newly added virtual edges e, e' with both endpoints in the same T_a. It then holds that $W(e) = W(e_1) \circ \pi(x_1, y_1) \circ W(e_2)$ where e_1, e_2 are virtual edges added in level $j \leq i-1$ and $\pi(x_1, y_1) \subseteq T_{b_1}$ for some $b_1 \neq a$. In the same manner, $W(e') = W(e_3) \circ \pi(x_2, y_2) \circ W(e_4)$ where e_3, e_4 are virtual edges added in level $j \leq i-1$ and $\pi(x_2, y_2) \subseteq T_{b_2}$ for some $b_2 \neq a$.

We first claim that $\pi(x_1, y_1)$ and $\pi(x_2, y_2)$ are edge-disjoint. There are two cases: Case (i): $b_1 \neq b_2$. In such a case, $\pi(x_1, y_1) \subseteq T_{b_1}$ and $\pi(x_2, y_2) \subseteq T_{b_2}$. Case (ii): $b_1 = b_2$. Since both endpoints of e, e' are in the same $(i+1)$-level block T_a, we have that these two edges were added when running the routing disjoint matching algorithm in the pair T_a, T_{b_1}. Thus the tree segments $\pi(x_1, y_1), \pi(x_2, y_2)$ are edge-disjoint.

Next, note that e_1, e_2, e_3, e_4 have both endpoints in T', thus by induction assumption for $i-1$, they are edge-disjoint. Part (i) follows. By induction assumption, $W(e_j) \cap E(T') = \emptyset$ and thus also $W(e_j) \cap E(T_a) = \emptyset$, for every $j \in \{1, 2, 3, 4\}$. Since $\pi(x_1, y_1), \pi(x_2, y_2) \subseteq T_{b_1}$ and the blocks T_a, T_{b_1} are edge-disjoint, we have that $\pi(x_1, y_1), \pi(x_2, y_2)$ are edge-disjoint with T_a. □

Corollary 2. *Each edge appears at most once on a fixed walk, and also at most once for a fixed cycle in $\mathcal{C}'$.*

Proof. Consider the walks $W(\widehat{e}) = W(e) \circ \pi(x, y) \circ W(e')$ added in level i. In such a case, there exists an i-level block T' such that both e, e' have both endpoints in T'. By Claim 7(i) we have that $W(e), W(e'), \pi(x, y)$ are edge-disjoint. By Claim 7(i), $W(e)$ and T' are edge-disjoint, and the final cycle is given by $W(e) \circ \pi(x, y)$ for $\pi(x, y) \subset T'$. □

Covering. We first prove the covering property on the preliminary collection of non-simple cycles $\mathcal{C}'$. Then, we will show that making the cycle simple preserves the covering property.

Claim 8. *For each non-tree edge e and every recursion level i, either $e \in W(\widehat{e})$ for some $\widehat{e} \in E_i$ or that e is already covered by a cycle in $\mathcal{C}'$.*

Proof. For the base of the induction, the walk collection is $\mathcal{W}_1 = \{e \in E(G) \setminus T\}$ and the claim holds vacuously. Assume that it holds up to level $i-1$ and consider level i. We say that an edge e is covered by the walks of the edges E_j for some $j \in \{1, \ldots, \ell\}$ if there exists $e' \in E_j$ such that $e \in W(e')$. Using the induction assumption for $1, \ldots, i-1$, it is sufficient to show that all the edges that are covered by the walks of E_{i-1} and are not covered by the walks of E_i are already covered by a cycle in $\mathcal{C}'$.

Assume towards contradiction that there exists such an edge $e \in G \setminus T$ that is not covered by a cycle in level $i-1$. Let $e' \in E_{i-1}$ be such that $e \in W(e')$. Since e is not covered by the walks of E_i, we have that e' was omitted from E_i in level $i-1$, which in turn implies that e' was an inter-block edge in this level. There are two cases to consider: (1) the fundamental cycle of $e' = (x, y)$ is taken, and thus $C = \pi(x, y) \circ W(e')$ is added to $\mathcal{C}'$. Since $e \in W(e')$, the edge e is

indeed covered by the cycle C, contradiction. (2) The edge e' is matched to some other inter-block edge e''. Here either (2.1): a new virtual edge $\widehat{e}$ was added to E_i such that $W(e') \subseteq W(\widehat{e})$, and since $e \in W(e')$, we get a construction. Otherwise (2.2): the virtual edge $\widehat{e}$ already appears in E_i, in such a case, a cycle $W(e') \circ W(\widehat{e}) \circ W(e'')$ was added to $\mathcal{C}'$, and we get a contradiction again. The proof follows. □

We are now ready to complete the covering argument. Note that in each level i, the set E_i contains all edges with both endpoints in i-level blocks. This is because in level $i-1$, all edges between i-level blocks are omitted from E_i and replaced with edges inside the i-level blocks. Since in the last level ℓ, all blocks are singletons, we have that $E_\ell = \emptyset$ and thus all edges are covered.

Time Complexity Analysis. For every level i, all i-level blocks in the recursion tree are edge-disjoint, thus the total number of edges in all these blocks is thus at most $n-1$. The algorithm first partitions the E_i edges $e = (x, y)$ based on the identities of $(i+1)$-level blocks to which the edge endpoint x, y belong. This takes linear time in the number of edges in E_i.

By applying Lemma 6 the partitioning of all i-level blocks into balanced $(i+1)$-levels can be done in $O(m)$ times using Lemma 6.

Now fix a block T_a and one of its bothering blocks T_b in level $i+1$, and let $E_{a,b} = E_i \cap (T_a \times T_b)$. Recall that $M_{a,b}$ is the set of marked vertices in T_a with a unique edge in T_b. By Lemma 5, applying Alg. DisjointMatching on the instance $T_a, M_{a,b}$ can be done in time $O(|T_a|)$ time. Since this algorithm is applied $O(n^{2\epsilon})$ for each pair of bothering $(i+1)$-level blocks, overall this step is implemented in time

$$O(n^{\epsilon}) \cdot \sum_{a=1}^{k} |T_a| = O(n^{1+\epsilon}) \quad \text{as} \quad \sum |T_a| \leq |T| = n \ .$$

After applying Alg. DisjointMatching, a walk is defined for each pair of matched edges. This is done in time which is linear in the length of the walk (taking multiplicities into account). Since each tree edge e has congestion $O(i \cdot n^{\epsilon})$ w.r.t the walks of $\mathcal{W}_i$ forming the walks takes $O(i \cdot n^{1+\epsilon} + m)$ time. Finally, making each of the cycles $C \in \mathcal{C}'$ simple can be implemented in $O(|C| \log n)$ time, thus the total time complexity is $\sum\{|C|, C \in \mathcal{C}'\} \leq \mathsf{c} \cdot n + m = O(n^{1+2\epsilon} + m)$, as each tree edge appears on at most c many cycles. This completes the proof for Lemma 4. Theorem 10 follows by combining Lemma 4 and Lemma 2.

5 Distributed Computation

One of the benefits Algorithm PartialCycleCover is that although it is described as a centralized algorithm, it can be easily adapted to the distributed setting. We consider the standard CONGEST model of distributed computing [17]. We first describe the required adaptations to compute cycle cover in the distributed setting. Then, we show how to also compute short cycle decomposition within the same number of rounds.

5.1 Cycle Cover

As in [16], the cycle cover procedure consists of two sub-procedures: the first covers all non-tree edges for a given BFS tree T, the second covers the remaining tree edges. To cover the non-tree edges we will present Alg. DistCycleCover which will simulate all steps of Alg. PartialCycleCover. To cover the remaining tree edges, we will use the reduction of [16] to the non-tree edges. Also, this reduction can be implemented efficiently in the distributed setting. The output of the cycle cover algorithm is defined as follows: the endpoints of every edge e know the identifiers of all the edges that are covered by the cycles that go through e. We show:

Theorem 11. *For every bridgeless n-vertex graph $G = (V, E)$ with diameter D and $\epsilon \in (0, 1]$, one can compute an $(\mathtt{d}, \mathtt{c})$ cycle cover $\mathcal{C}$ with $\mathtt{d} = 2^{O(1/\epsilon)} \cdot D$ and $\mathtt{c} = O(n^{\epsilon} \log^2 n)$, within $\widetilde{O}(2^{1/\epsilon} \cdot \cdot n^{\epsilon} \cdot D)$ rounds.*

Covering Non-Tree Edges. Throughout, we are given a tree T and the goal is to cover all edges in $E(G) \setminus T$. We start by noting that the key tools of algorithm PartialCycleCover, namely, computing routing disjoint matching and block partitioning can be done in $O(D)$ rounds.

Lemma 7. *(1) There is a distributed algorithm that given an instance $\langle T, M \rangle$ solves the Routing Disjoint Matching problem in time $O(\mathsf{Depth}(T))$ (see [17]). (2) There is a distributed algorithm that computes the balanced tree partitioning (of Definition 3) in $O(\mathsf{Depth}(T))$ rounds.*

For the balanced tree partitioning, in the distributed output every vertex knows its block-ID (maximum edge ID) and its parent in its block. The root vertices are also marked. In the case where a vertex v is a root of several blocks, v knows its children in each of its blocks.

Let T be a BFS tree and let $E' = E \setminus T$ be the initial edge set to be covered. The key observation is that since the blocks in each level-i of the recursion are edge-disjoint, one can work on all these blocks *simultaneously*. In each independent level i of the recursion, we are then given a subtree T', and a marked collection of E' to be covered. The algorithm first computes a balanced partitioning of T' by dividing it into $O(n^{\epsilon})$ blocks $T_1, \ldots, T_k$. By letting all nodes exchange their information on their $(i+1)$-level blocks with their neighbors, all nodes know which of the E'-edges should be considered. We will then run k^2 algorithms for each pair of blocks T_a, T_b. For each pair of blocks, we will consider the current set of edges $E_{a,b}$. Applying the routing disjoint matching on all these subgraphs simultaneously in parallel can be done in $\widetilde{O}(k + D)$ rounds.

Note that some of the $E_{a,b}$ edges might be virtual and in such case, we will keep the guarantee that every vertex knows its virtual neighbors and the block-ID of its neighbors. To see how it can be implemented efficiently, consider the first recursion level. The virtual edges are defined based on the output of the routing disjoint matching algorithm on T_a.

Recall that every virtual edge $\widehat{e} = (y, y')$ has a path $W(\widehat{e}) = e \circ \pi(x, y) \circ e'$ where $e = (x, y), e' = (x', y') \in T_a \times T_b$. By Claim 5, each edge appears on at most $\mathtt{c} = n^{2\epsilon}$ many walks, and thus using the standard random delay approach sending $O(\log n)$-bits of information between the endpoint of all the virtual edges simultaneously can be done in $\widetilde{O}(\mathtt{c} + \mathtt{d})$ times. In the same manner, the definition of the new walks in each level is done by sending information along the low-congestion walks.

Since the collection of all walks have congestion at most $\mathtt{c} = n^{O(\epsilon)}$ and they are of length $\mathtt{d} = 2^{O(1/\epsilon)}\mathsf{Depth}(T)$, each phase of the recursion can be implemented in $\widetilde{O}(\mathtt{c} + \mathtt{d})$ rounds. Finally, once the cycles are computed, using the random delay approach again, all edges can learn the

entire cycles that go through it in $O(\mathtt{c} \cdot \mathtt{d})$ rounds and locally make them simple. As there are $O(1/\epsilon)$ recursion levels, we have:

Lemma 8. *Given a tree $T \subseteq G$, and $\epsilon \in (0, 1]$, there is a (randomized) distributed algorithm that computes a cycle collection that covers all non-tree edges with congestion $\mathtt{c} = O(n^{\epsilon})$ and $\mathtt{d} = O(2^{1/\epsilon} \cdot \mathsf{Depth}(T))$ within $\widetilde{O}(1/\epsilon(\mathtt{c} \cdot \mathtt{d}))$ rounds.*

Covering the Tree Edges: We turn to consider the remaining tree edges $E(T)$. Algorithm DistTreeCover essentially mimics the centralized construction from [16]. We revise some notation from [16]. Throughout, when referring to a tree edge $(u, v) \in T$, the node u is closer to the root of T than v. Let $E(T) = \{e_1, \ldots, e_{n-1}\}$ be an ordering of the edges of T in non-decreasing distance from the root. For every tree edge $e \in T$, define the *swap* edge of e by $e' = \mathsf{Swap}(e)$ to be an arbitrary edge in G that restores the connectivity of $T \setminus \{e\}$. Since the graph G is 2-edge connected such an edge $\mathsf{Swap}(e)$ is guaranteed to exist for every $e \in T$. Let $e = (u, v)$ and $(u', v') = \mathsf{Swap}(e)$, we denote u by $p(v)$ (since u is the parent of v in the tree) and v' by $s(v)$, where $s(v)$ is the endpoint of $\mathsf{Swap}(e)$ that do not belong to $T(u')$ (i.e., the subtree T rooted at u). Define the v-$s(v)$ path

$$P_e = \pi(v, u') \circ \mathsf{Swap}(e).$$

By using the algorithm of Section 4.1 in [9], we can make every node v know $s(v)$ in $O(D)$ rounds. A key part in the tree covering algorithm of [14, 16] is the definition of the path $P_e = \pi(v, u') \circ (u', s(v))$ for every tree edge $e = (p(v), v)$. By computing swap edges using Section 4.1 in [9] all the edges of each P_e get marked.

Computing an independent set of tree edges. We next describe how to compute a maximal collection of tree edges $I = \{e_i\}$ whose paths P_{e_i} are edge disjoint and in addition for each edge $e_j \in E(T) \setminus I$ there exists an edge $e_i \in T'$ such that $e_j \in P_{e_i}$. To achieve this, we start working on the root towards the leaf. In every round $i \in \{1, \ldots, D\}$, we consider only *active* edges in layer i in T. Initially, all edges are active. An edge becomes inactive in a given round if it receives an inactivation message in any previous round. Each active edge in layer i, say e_j, initiates an inactivation message on its path P_{e_j}. An inactivation message of an edge e_j propagates on the path P_{e_j} round by round, making all the corresponding edges on it to become inactive.

Note that the paths P_{e_j} and $P_{e_{j'}}$ for two edges e_j and $e_{j'}$ in the same layer of the BFS tree, are edge disjoint and hence inactivation messages from different edges on the same layer do not interfere each other. We get that an edge in layer i active in round i only if it did not receive any prior inactivation message from any of its BFS ancestors. In addition, any edge that receives an inactivation message necessarily appears on a path of an active edge. It is easy to see that within D rounds, all active edges I on T satisfy the desired properties (i.e., their P_{e_i} paths cover the remaining T edges and these paths are edge disjoint).

Distributed Implementation of Algorithm TreeCover**.** First, we mark all the edges on the P_e paths for every $e \in I(T)$. As every node v with $e = (p(v), v)$ know its swap edge, it can send information along P_e and mark the edges on the path. Since each edge appears on the most two P_e paths, this can be done simultaneously for all $e \in I(T)$.

From this point on we follow the steps of Algorithm TreeCover. The balanced tree partitioning can be done in $O(D)$ rounds. We define the ID of each tree T'_1, T'_2 to be the maximum edge ID

in the tree (as the trees are edge disjoint, this is indeed an identifier for the tree). By passing information on the P_e paths, each node v can learn the tree ID of its swap endpoint $s(v)$. This allows to partition the edges of T' into $E'_{x,y}$ for $x, y \in \{1, 2\}$. Consider now the i^{th} phase in the computation of cycle cover $\mathcal{C}_{1,2}$ for the edges $E'_{1,2}$.

Applying Algorithm TreeEdgeDisjointPath can be done in $O(D)$ round. At the end, each node v_j knows its matched pair v'_j and the edges on the tree path $\pi(v_j, v'_j, T'_1)$ are marked. Let Σ be the matched pairs. We now the virtual conflict graph G_Σ. Each pair $\langle v_j, v'_j \rangle \in \Sigma$ is simulated by the node of higher ID, say, v_j. We say that v_j is the *leader* of the pair $\langle v_j, v'_j \rangle \in \Sigma$. Next, each node v that got matched with v' activates the edges on its path $P_e \cap E(T'_1)$ for $e = (p(v), v)$. Since the π edges of the matched pairs are marked as well, every edge $e' \in \pi(v_k, v'_k, T'_1)$ that belongs to an active path P_e sends the ID of the edge e to the leader of the pair $\langle v_k, v'_k \rangle$. By Claim 6 in [14], every pair σ' interferes with at most one other pair and hence there is no congestion and a single message is sent along the edge-disjoint paths $\pi(v_j, v'_j, T'_1)$ for every $\langle v_j, v'_j \rangle \in \Sigma$. Overall, we get that the construction of the virtual graph can be done in $O(D)$ rounds.

We next claim that all leaders of two neighboring pairs $\sigma, \sigma' \in G_\Sigma$ can exchange $O(\log n)$ bits of information using $O(D)$ rounds. Hence, any r-round algorithm for the graph G_Σ can be simulated in T'_1 in $O(r \cdot D)$ rounds. To see this, consider two neighbors $\sigma = \langle x, y \rangle, \sigma' = \langle x', y' \rangle$ where σ' interferes σ. Without loss of generality, assume that the leader x' of σ' wants to send a message to the leader x of σ. First, x' sends the message on the path $\pi(x', y', T'_1)$. The edge $e' \in \pi(x', y', T'_1) \cap P_e$ for $e = (p(x), x)$ that receives this message sends it to the leader x along the path P_e. Since we only send messages along edge disjoint paths, there is no congestion and can be done in $O(D)$ rounds.

Since the graph G_Σ has arboricity $O(1)$, it can be colored with $O(1)$ colors and $O(\log n)$ rounds using the algorithm of [4]. By the above, simulating this algorithm in G takes $O(D \log n)$ rounds. We then consider each color class at a time where at step j we consider $\Sigma_{i,j}$. For every $\sigma = \langle x, y \rangle$, x sends the ID of $s(y)$ to $s(x)$ along the P_e path for $e = (p(x), x)$. In the same manner, y sends the ID of $s(x)$ to $s(y)$. This allows each node in T'_2 to know its virtual edge. At that point, we run Algorithm DistNonTreeCover to cover the virtual edges. Each virtual edge is later replaced with a true path in G in a straightforward manner.

Analysis of Algorithm DistTreeCover**.**

Claim 9. *Let* $\mathtt{c} = O(n^\epsilon)$ *and* $\mathtt{d} = 2^{1/\epsilon} \cdot \mathsf{Depth}(T)$. *Algorithm* DistTreeCover *computes a* $(\mathtt{d}, \mathtt{c})$ *cycle cover for the tree edges* $E(T)$ *within* $\widetilde{O}(\mathtt{c} \cdot \mathtt{d})$ *rounds.*

Proof. Let $D = \mathsf{Depth}(T)$. The correctness follows the same line of arguments as in the centralized construction (see the Analysis in Sec. 4.1.2 in [14]), only the here we use Algorithm DistNonTreeCover. Each cycle computed by Algorithm DistNonTreeCover has length $O(2^{1/\epsilon} D)$ and the cycle covers $O(2^{1/\epsilon})$ non-tree edges. In our case, each non-tree edge is virtual and replaced by a path of length $O(D)$ hence the final cycle has still length $O(2^{1/\epsilon} D)$. With respect to congestion, we have $O(\log n)$ levels of recursion and in each level when working on the subtree T' we have $O(\log n)$ applications of Algorithm DistNonTreeCover which computes cycles with congestion $O(n^\epsilon)$. The total congestion is then bounded by $O(n^\epsilon \cdot \log^2 n)$.

We proceed with round complexity. The algorithm has $O(\log n)$ levels of recursion. In each level, we work on edge disjoint trees simultaneously. Consider a tree T'. The partitioning into T'_1, T'_2 takes $O(D)$ rounds. We now have $O(\log n)$ phases. We show that each phase takes $O(\mathtt{c} \cdot \mathtt{d})$

rounds, which is the round complexity of Algorithm DistNonTreeCover. In particular, In phase i we have the following procedures. Applying Algorithm TreeEdgeDisjointPath in T_1', T_2' takes $O(\mathsf{Depth}(T))$ rounds. The computation of the conflict graph G_Σ takes $O(D)$ rounds as well and coloring it using the coloring algorithm for low-arboricity graphs of [4] takes $O(\mathsf{Depth}(T)\log n)$ rounds. Then we apply the algorithm of Lemma 8 which takes $\widetilde{O}(\mathtt{c}\cdot\mathtt{d})$ rounds. Letting each edge know all cycles that go through it also takes the same order of the number of rounds. □

Theorem 11 follows by combining Lemma 8 and Claim 9.

Distributed Construction of Short Cycle Decomposition

Lemma 9. *For every $\epsilon \in (0,1]$, one can compute an $(\widehat{m}, L)$ cycle decomposition with $\widehat{m} = O(n\log n)$ and $L = O(2^{1/\epsilon}\cdot\log n)$ within $O(\widehat{m}\cdot L)$ rounds.*

Proof. The algorithm consists of $O(\mathtt{c}\cdot\mathtt{d}\cdot\log n)$ applications $(\mathtt{d},\mathtt{c})$ cycle cover constructions with $\mathtt{d} = 2^{1/\epsilon}\cdot\log n$ and $\mathtt{c} = 1/\epsilon\cdot n^{\epsilon}$. Initially, set $\mathcal{D} = \emptyset$ and $E_1 = E(G)$. In phase $i \geq 1$, let E_i be set of edges in G that are not yet covered by the current edge-disjoint cycle collection $\mathcal{D}$. First, we construct a 1-neighborhood cover in E_i within $O(\log n)$ rounds. In each component $G_{i,j}$ the algorithm applies Alg. DistCycleCover. Let $\mathcal{C}_i$ be the total cycle collections computed in all these components. The next step is to compute a maximal independent set on these cycles. The vertices compute the conflict graph $\widehat{G}$ of the cycles in $\mathcal{C}_i$, where every two cycles that intersect with an edge are neighbors in $\widehat{G}$. By sending information along cycles, every round of Luby's algorithm can be implemented in the conflict graph $\widehat{G}$ using $O(\mathtt{c}\cdot\mathtt{d})$ rounds. The maximal independent collection of cycles in $\mathcal{C}_i$ are then added to $\mathcal{D}$ and their edges are omitted from E_i. This completes the description of the algorithm. By the same argument as for the centralized setting, the total number of phases is $O(\mathtt{d}\cdot\mathtt{c}\log n)$. At the end of the procedure, we are left with $O(n\log n)$ uncovered edges. Each phase takes $\widetilde{O}(\mathtt{d}\cdot\mathtt{c})$ rounds, and thus the final round complexity is $\widetilde{O}(\mathtt{d}^2\cdot\mathtt{c}^2)$. □

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] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast distributed network decompositions and covers. *Journal of Parallel and Distributed Computing*, 39(2):105–114, 1996.

[3] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998.

[4] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.

[5] Joshua Batson, Daniel A Spielman, and Nikhil Srivastava. Twice-ramanujan sparsifiers. *SIAM Journal on Computing*, 41(6):1704–1721, 2012.

[6] 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.

[7] 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.

[8] 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.

[9] Mohsen Ghaffari and Merav Parter. Near-optimal distributed algorithms for fault-tolerant tree structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 387–396. ACM, 2016.

[10] Alon Itai and Michael Rodeh. Covering a graph by circuits. In *International Colloquium on Automata, Languages, and Programming*, pages 289–299. Springer, 1978.

[11] 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.

[12] Yang P. Liu, Sushant Sachdeva, and Zejun Yu. Short cycles via low-diameter decompositions. *SODA*, 2019.

[13] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[14] Merav Parter and Eylon Yogev. Distributed computing made secure: A new cycle cover theorem. *arXiv preprint arXiv:1712.01139*, 2017.

[15] Merav Parter and Eylon Yogev. Distributed computing made secure: A graph theoreric approach. *SODA*, 2019.

[16] Merav Parter and Eylon Yogev. Low congestion cycle covers and their applications. *SODA*, 2019.

[17] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.

[18] 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.

A Missing Details for Algorithm DistCycleCover

Lemma 5. The matching and the corresponding edge-disjoint paths are computed as follows. Sort the vertices in decreasing distance from the root. Letting D be the diameter of T, the algorithm has D phases. For every vertex u in the tree, the algorithm computes in a bottom-up manner an ID of a yet unmatched vertex in the subtree of u, $T(u)$, if such exists. We also will keep a list of all the matched pairs. The invariant will be that at the end of phase $i \geq 1$, every vertex v at depth $D - i - 1$ has at most one unmatched vertex, denoted by $U(v)$, in its subtree.

Initially, $A(v) = v$ for every marked vertex v, and $A(v) = \emptyset$ for the remaining vertices. Thus the invariant clearly holds for $i = 1$. In phase $i \geq 2$, every vertex u sums up the number of unmatched vertices in its subtree (e.g., using the information of itself and of its children). It then matches arbitrarily the $U(w)$ vertices of its children w, except for at most one (if the number of these unmatched vertices is odd), into pairs and adds it to the list of the matched pairs. Set $U(v) = w'$ where w' is the remaining unmatched vertex in $T(u)$ (if such exists). This establishes the invariant for phase i.

Finally, to compute the tree paths between that matched pairs, one can use standard LCA data structure and construct these paths, edge by edge. □

Proof of Lemma 6. The *weight* of a node v with respect to E', denoted by $\omega(v, E')$ (the set E' will be omitted when clear from the context), is the density of the single set that contains v. That is,

$$\omega(v, E') = \mathsf{density}(\{v\}, E') \ .$$

The partitioning into blocks is performed as follows. First, every vertex v with at least $\Delta/2$ edges in E', i.e., $\omega(v) \geq \Delta/2$, is defined as a singleton block. Such a vertex is said to be *heavy*. We note that the blocks will be edge-disjoint but not necessarily vertex-disjoint, and that heavy vertices might also be a root for other blocks.

Then, the algorithm performs D iterations where D is the depth of the tree T, and in each iteration it assigned vertices to blocks. Define the *residual weight* of a node v, denoted by $R(v)$, as the total weight of all the vertices in its subtree $T(v)$ that are not yet assigned to a block T_i. In each iteration $i \geq 1$, the algorithm computes the residual weights of the vertices in depth $D - i$ along with the weights of their parents in the block. That is, for every vertex u in depth $D - i$, let $W'(u)$ be the total sum of the residual weights of its children and its own weight. That is,

$$\omega'(u) = \sum_{w \in \mathsf{child}(u)} R(w) + \omega(u) \ .$$

If $\omega'(u) \leq \Delta/2$, then the residual weight of u is simply $R(u) = \omega'(u)$ and we set the parent of u in its block to its parent in T.

Otherwise, if $\omega'(u) > \Delta/2$, we compute block(s) rooted at u in the following manner. We enumerate the children of u from left to right and greedily partition them into subtrees with total residual weight in the range $[\Delta/2, \Delta]$. Finally, every root forms a singleton block.

Number of blocks: Every block has density in $> \Delta/2$ when including its root edges into account. Thus letting each light root be a singleton block, increase the number of blocks by a factor of 2.

Time complexity: Since we build the blocks in a bottom-up manner, the blocks and weights and residual weights can all be computed during the traversal. Thus, this decomposes the tree T into $O(|E'|/\Delta)$ blocks, and that this decomposition can be computed in linear time. □

Postorder Numbering on a Tree. Given a BFS tree T, We now show a procedure which assigns the vertices numbers $N : V \to [1, n]$ according to a postorder traversal on T in $O(D)$ rounds. Each node u first computes the number of vertices in $T(u)$. This can be done in $O(D)$ rounds by working from the leafs up. At the end, each node also knows the number of nodes in the subtree of each of its children. For a range of integers $R = [i, j]$, let $\max(R) = j$ and $\min(R) = i$. By working from root to leaf nodes, each vertex u computes a range $R(u)$, indicating the bucket of post-order numbers given to its vertices in $T(u)$. Let $R(s) = [1, n]$. Given that a vertex v has received its range $R(v)$, it sends to its children their ranges in the following manner. Let $u_1, \ldots, u_\ell$ be the children of v ordered from left to right. Then $R(u_1) = [\min(R(v)), \min(R(v)) + |T(u_1)| - 1]$ and for every $i \geq 1$, $R(u_i) = [\max(R(u_{i-1})) + 1, \max(R(u_{i-1})) + |T(u_i)|]$. This proceeds for $O(D)$ rounds, at the end every u knows $R(u)$. Now, each node u, sets its own number $N(u) = \max(R(u_i))$. Since by this numbering, each vertex has the maximum number in its subtree, it is indeed a postorder numbering.

Sending Information on Virtual Edges. For every virtual edge (a, a') added in phase $i \geq 1$, we show in the analysis section, that there is a precomputed a-a' walk $W_{a,a'}$ in G of length $O(2^i \cdot D)$. We assume that in phase i, all the edges on the walk of virtual edge added in phase $j \leq i - 1$, are already marked and that each such edge on the walk knows the endpoint of the virtual edges and the edges in G that this walk should cover. The analysis shows that in the virtual edges added in phase $i - 1$ are important of $O(2^{i-1})$ edges in G.

Assume that it is given for all virtual edges added in phase $j \leq i - 1$, we now show how the algorithm provides these properties for the virtual edges added in phase i. Assume that both edges (a, a') and (b, b') incident to the matched pair $\langle a, b \rangle$ are virtual. Then, we send the information on the walks $\pi(a, b, T_{B_x})$, $W_{a,a'}$ and $W_{b,b'}$. This allows the endpoints of virtual edge $a', b'$to learn about each other. In addition, by passing the identifiers of the 2^{i-1} edges that are supposed to be covered by the walks $W_{a,a'}$ and $W_{b,b'}$, all the nodes on the new walk $W_{a',b'} = W_{a',a} \circ \pi(a, b, T_{B_x}) \circ W_{b,b'}$ can get this information. This is done on for all the virtual edges added in phase i simultaneously using the random delay approach. In the analysis section we show that the length of the walks $W_{a,a'}$ is bounded by $O(2^{1/\epsilon} D)$ and each edge appears on $O(n^{2\epsilon})$ many paths. Thus using the random delay approach, this can be done in $O(2^{1/\epsilon} \cdot D + n^{2\epsilon})$ rounds.

Marking Edges on Cycles and Making Them Simple. We first make every edge e (i.e., the endpoints of the edge) know the set of edges that are covered by the cycles that go through e. To do that, the endpoints of the virtual edges keep information of the covered edge endpoints. In phase 1, this is easily obtained since the endpoint of the virtual edges are also the vertices whose two edges should be covered by the cycle. Assuming that this information is kept up to phase $i-1$, in phase i, when adding the virtual edge $u-v$, the combined information traverses through the edge disjoint path $\pi(u,v)$ computed on the tree of the block.

We next make every edge e know all the edges of the cycles that go through e. Since each edge appears on n^ϵ cycles and the length of the cycles is $O(2^{1/\epsilon} \cdot D)$, using standard random delay techniques, it can be done in $\widetilde{O}(n^{2\epsilon} + 2^{1/\epsilon} \cdot D)$ rounds. Once each edge e sees the entire cycle, it can locally correct it to be simple.