An LCA for Approximated MST in General Bounded-Degree Graphs

Reut Levi
Efi Arazi School of Computer Science, Reichman University, Herzliya, Israel

Moti Medina
Faculty of Engineering, Bar-Ilan University, Ramat Gan, Israel

Daniel Prigan
Efi Arazi School of Computer Science, Reichman University, Herzliya, Israel

Abstract

We present a *local computation algorithm* (LCA) for constructing a connected spanning subgraph whose total weight is at most a $(1+\epsilon)$-factor larger than that of a minimum spanning tree, in general bounded-degree graphs. Prior to our work, nontrivial LCAs for this problem, namely, algorithms with sublinear query complexity, were known only for the restricted graph family of minor-free graphs by Levi, Ron, and Rubinfeld (Algorithmica 2020).

The query complexity of our algorithm in terms of the number of vertices, n, is $\tilde{O}(n^{2/3})$. The best known lower bound for this problem is $\Omega(n^{1/2})$.

Our approach consists of three conceptual layers. The first is a localized variant of Prim's algorithm, which reconstructs, using only local queries, a large fraction of the edges of the minimum spanning tree. The resulting subgraph at this stage is disconnected. To address this, in the second layer, we partition the partially constructed forest into *clusters* of size $\tilde{O}(n^{1/3})$. To this end, we present a *partition oracle*, as introduced by Hassidim et al. (FOCS 2009), for trees whose query complexity is nearly optimal in terms of ϵ, the parameter that controls the number of edges in the boundary. In particular, its query complexity is $\tilde{O}(d/\epsilon)$, where d denotes the degree bound. In the third and last layer, we adapt the technique from Lenzen-Levi (ICALP 2018) for locally computing a sparse spanning subgraph and obtain an algorithm that locally identifies and adds a small number of carefully chosen edges in order to restore global connectivity. We show that the number of such additional edges is small, and consequently, their total weight contributes only a small amount to the overall cost, preserving the $(1+\epsilon)$-approximation guarantee.

2012 ACM Subject Classification Theory of computation → Streaming, sublinear and near linear time algorithms; Mathematics of computing → Graph algorithms; Theory of computation → Graph algorithms analysis

Keywords and phrases Locally Computable Algorithms, Sublinear Algorithms, Minimum Spanning Trees, Partition Oracles

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.154

Funding *Reut Levi*: The author was supported by the Israel Science Foundation under Grant 1867/20.
Moti Medina: The author was supported by the Israel Science Foundation under Grant 554/23.
Daniel Prigan: The author was supported by the Israel Science Foundation under Grants 1867/20 and 843/23.

Acknowledgements We thank the anonymous reviewers for their helpful comments, and in particular for simplifying the proof of Claim 4.

1 Introduction

The minimum spanning tree (MST) problem is a classical and fundamental problem in graph algorithms, with numerous applications and close connections to other basic graph primitives. Given a connected weighted graph $G = (V, E, w)$, an MST is a spanning tree of minimum total weight.

Motivated by settings in which the input graph is too large to be processed in full, a line of work has studied how to provide *local access* to global graph structures, where the algorithm must answer queries while probing only a small portion of the input. Within this framework, Levi, Ron, and Rubinfeld [14, 15] initiated the study of providing local access to sparse connected spanning subgraphs in the local computation algorithms (LCA) model.

In their formulation, the goal is to provide access to a connected spanning subgraph with at most $(1+\epsilon)n$ edges. They showed that constructing a spanning tree locally with sublinear query complexity is impossible, and that the allowance of an additional ϵn edges is therefore necessary. A natural and significantly more challenging variant of this problem is to require the locally accessible subgraph to have near-minimum total weight. Here, the objective is to provide local access to a connected spanning subgraph whose total weight is within a factor of $(1+\epsilon)$ of the minimum spanning tree. Levi, Ron, and Rubinfeld considered this weighted variant as well, but obtained such an algorithm only for the special case of minor-free graphs.

In a survey on local graph algorithms, Rubinfeld [18] highlighted this gap and posed the following open question:

> *Are there (other) families of weighted graphs for which there exist $o(n)$-time LCAs for low-weight sparse spanning subgraphs?*

At the time, such algorithms were known only for minor-free graphs and, slightly more generally, hyperfinite graph families [12]. Since then, the only progress has been the extension to unbounded-degree minor-free graphs [16].

In this paper, we answer this question affirmatively for general bounded-degree graphs. We present a local computation algorithm that provides access to a connected spanning subgraph whose total weight is within a factor of $(1+\epsilon)$ of the minimum spanning tree, while using sublinear query complexity.

As a by-product of our approach, we also obtain a nearly optimal partition oracle for trees, supporting local reconstruction of parts of a decomposition that underlies our local treatment of connectivity and edge selection.

1.1 Our Results

Before stating our result, we first state the formulation of the problem. Our algorithm is formulated in the Local Computation Algorithms (LCA) model as introduced by Rubinfeld et al.[19] (see Definition 23 for more details about the model).

We begin by stating the LSSG problem, as introduced by Reut Levi, Dana Ron, and Ronitt Rubinfeld, and then, based on this formulation, define the approximate MST problem in the LCA model.

▶ **Definition 1** ([15]). *An algorithm $\mathcal{A}$ is a local sparse spanning graph (LSSG) algorithm if, given $n \geq 1$, $\epsilon > 0$, and query access to the incidence-lists representation of a connected graph $G = (V, E)$ over n vertices, it provides oracle access to a subgraph $G' = (V, E')$ of G such that:*

1. *G' is connected.*
2. *$|E'| \leq (1+\epsilon) \cdot n$ with high constant probability* [1]*, where E' is determined by G and the internal randomness of $\mathcal{A}$.*

More specifically, on query $u, v \in E$, $\mathcal{A}$ returns whether $(u,v) \in E'$, and for any sequence of edges, $\mathcal{A}$ answers consistently with the same G'.

An algorithm $\mathcal{A}$ is an LSSG algorithm for a family of graphs $\mathcal{C}$ *if the above conditions hold, provided that the input graph G belongs to $\mathcal{C}$.*

▶ **Definition 2** ([14])**.** *A local algorithm for $(1+\epsilon)$-approximating the minimum weight spanning graph of a graph $G = (V, E, w)$ with positive weights, is a local algorithm for $(1+\epsilon)$-sparse spanning graph of $G = (V, E, w)$ for which the following holds with high constant probability: $\sum_{e \in E'} w(e) \leq (1+\epsilon)OPT(G)$, where $OPT(G)$ is the weight of a minimum weight spanning tree of G.*

For a graph $G = (V, E, w)$ we define $w_{\max} = \max_{e \in E} w(e)$ and $w_{\min} = \min_{e \in E} w(e)$.

We provide a local algorithm for computing a $(1+\epsilon)$-approximation of the minimum-weight spanning subgraph of a graph, using $\tilde{O}\left(n^{2/3} \cdot d^2 \cdot \left(\frac{\epsilon w_{\min}}{w_{\max}}\right)^{-4/3}\right)$ probes per query, where d denotes an upper bound on the degree. As shown in [15], even for the simpler task of providing access to a sparse spanning graph, $\Omega(n^{1/2})$ queries are required. As for the dependence on $\frac{w_{\max}}{\epsilon w_{\min}}$, we note that a simple argument shows it must be at least linear [2].

In the course of designing our algorithm, we also obtain a nearly optimal *partition oracle* for trees that supports local reconstruction of parts. In particular, its query complexity is $\tilde{O}(d/\epsilon)$, which is nearly optimal in terms of ϵ, the parameter controlling the boundary size. [3]

1.2 Related Work

Local algorithms for constructing sparse spanning subgraphs. The model of *local computation algorithms* (LCAs) considered in this work was introduced by Rubinfeld et al. [19] (see also Alon et al. [1] and the survey [10]). The problem of constructing sparse spanning subgraphs in this model has been studied in a number of works [14, 12, 13, 9, 17, 15]. This problem can be viewed as a special case of constructing an ϵ-almost MST in which all edge weights are identical.

For restricted families of graphs, it was shown that the complexity of the problem can be independent of n. In particular, Levi et al. [12] showed that for graph families that are, roughly speaking, sufficiently non-expanding, one can design an algorithm whose query complexity is independent of n, albeit super-exponential in $1/\epsilon$. Their approach is based on simulating a localized version of Kruskal's algorithm. On the negative side, the same work [12] shows that for graphs with slightly stronger expansion properties, there is *no local algorithm* whose query complexity is independent of n.

The first LCA for sparse spanning subgraphs in general bounded-degree graphs was given in [9], achieving query complexity $\tilde{O}\left(n^{2/3} \cdot \mathrm{poly}(d, 1/\epsilon)\right)$ and stretch $O(\log^2 n \cdot \mathrm{poly}(d/\epsilon))$. More recently, the algorithm of [11] introduces a parameter ϕ representing a lower bound on

[1] In some papers, the required success probability is high, i.e., at least $1 - 1/\Omega(n)$.

[2] Consider a graph consisting of components that are either paths or cycles of length $\Theta(w_{\max}/(\epsilon w_{\min}))$, where all edges have weight $w_{\min}$ except possibly one edge of weight $w_{\max}$. In a path component, all edges must be included in any spanning forest, while in a cycle component the unique heavy edge should be excluded from the MST. Thus, in order to decide whether to include a queried heavy edge, the algorithm must determine whether it lies on a cycle or merely on a path, which requires exploring a distance $\Omega(w_{\max}/(\epsilon w_{\min}))$ in the worst case.

[3] Since the total number of edges crossing between parts is at most ϵdn, the average part must have size $\Omega(1/(d\epsilon))$, implying a trivial lower bound of $\Omega(1/(d\epsilon))$ on the query complexity.

the conductance of the input graph, and achieves query complexity $\tilde{O}(\sqrt{n}/\phi^2)$. This bound is nearly optimal for constant ϕ, and improves upon [9] for $\phi = \Omega(n^{-1/12})$. Their result is further extended to $(k, \phi_{\text{in}}, \phi_{\text{out}})$-clusterable graphs [4].

Local algorithms for $(1+\epsilon)$-approximating the minimum-weight spanning subgraph. Levi, Ron, and Rubinfeld [15] gave a local algorithm for constructing sparse spanning subgraphs in minor-free graphs of bounded degree, with query and time complexity polynomial in d and $1/\epsilon$. This result was later extended to the weighted setting in [16].

Approximating the weight of the MST. In their seminal work, Chazelle, Rubinfeld, and Trevisan [3] showed that the weight of the minimum spanning tree can be approximated in time independent of n via local exploration; however, their approach does not yield a way to reconstruct the edges of the MST (or even a sparse approximation). Intuitively, while estimating the total weight can be done by sampling and aggregating statistics, reconstructing edges requires making consistent global decisions. Indeed, the $\Omega(\sqrt{n})$ lower bound for the latter task shows that such reconstruction cannot be achieved with n-independent complexity.

Partition oracles. Partition oracles were introduced by Hassidim et al. [6] as a tool for approximating graph parameters and testing properties of minor-free bounded-degree graphs. The partition oracle of [6] has query complexity exponential in ϵ^{-1}. This dependence was later improved by Levi and Ron [13] to quasi-polynomial in ϵ^{-1}, and eventually Kumar, Seshadhri, and Stolman [8] obtained a partition oracle with query complexity polynomial in ϵ^{-1}. For graphs of bounded treewidth, Edelman et al. [5] obtained a partition oracle whose query complexity is polynomial in ϵ^{-1}. However, the degree of this polynomial is very large: even for trees (which have treewidth 1), the dependence is at least $O(\epsilon^{-33})$.

1.3 Overview of our algorithm

We begin by describing our algorithm from a global perspective, and then explain how this global procedure can be simulated locally using $\tilde{O}(n^{2/3})$ queries. For the purpose of the introduction, we assume that d, ϵ, and $w_{\min}/w_{\max}$ are constants; we note that the dependence on these parameters is polynomial.

Our algorithm consists of three layers. In the first layer, we partition the minimum spanning tree into $O(n^{2/3})$ disjoint connected subtrees. All edges belonging to these subtrees are included in E', the edge set of the constructed subgraph. In the second layer, we further partition each of these subtrees into smaller subtrees, which we refer to as *clusters*. The goal of this partitioning is to obtain clusters of size $\tilde{O}(n^{1/3})$, which can be reconstructed locally using only $\tilde{O}(n^{2/3})$ queries. In the third and final layer, we connect the resulting clusters by closely following the framework of Lenzen and Levi [9].

1.3.1 First Layer

In the first layer of our algorithm, each vertex is independently selected to be a *center* with probability $\Theta(1/n^{1/3})$. The vertices of the graph are then partitioned into connected components, which we call *Prim's cells*.

Informally, the Prim's cell of a center c consists of all vertices v for which Prim's algorithm, when initiated at v, reaches c before reaching any other center. More precisely, starting from a vertex v, we grow a connected subgraph by repeatedly adding the minimum-weight edge crossing the current cut. The vertex v is assigned to the Prim's cell of the first center encountered by this process; if this center is c, then v belongs to the cell of c.

The main technical challenge in the analysis of this part of the algorithm is to show that, under this definition, each Prim's cell induces a connected subgraph.

The main difficulty is that Prim's explorations initiated from adjacent vertices may evolve very differently, and even if the edge between them is eventually added, the explored subgraphs need not coincide; nevertheless, we show that if an edge lies on the minimum-spanning-tree path from a vertex to the first center added to its explored vertex set, then this edge enforces a consistency condition that guarantees both endpoints are assigned to the same center.

1.3.2 Second Layer

At this layer, our approach diverges from the partitioning technique of Lenzen and Levi. In their work, the initial cells are Voronoi cells of small diameter, which enables a relatively simple recursive partitioning scheme. This property was later exploited and reused by Parter et al. [17] and Arviv et al. [2]. The small diameter of the cells in these works is achieved via a local simulation of a distributed algorithm, relying on the fact that the underlying sparse spanning subgraph has locally bounded neighborhoods.

In our setting, however, the initial cells are induced by the minimum spanning tree. Unlike the sparse spanning graphs considered in prior work, an MST may have an arbitrarily large diameter, and therefore, the Voronoi-based partitioning techniques are no longer applicable. As a result, we cannot guarantee the formation of small-diameter cells and must adopt an alternative approach.

To overcome this difficulty, we design a nearly optimal partition oracle for trees, whose query complexity is roughly linear in the size of the parts it produces. Our algorithm applies this oracle to the Prim cells obtained in the previous layer, ensuring that the resulting parts are small and can be reconstructed efficiently in a local manner.

This result is of independent interest, as partition oracles have numerous applications. Moreover, our use of a partition oracle for trees provides evidence that such oracles can be useful even when the underlying graph is not itself a tree. We next briefly outline the approach underlying our partition oracle.

Our partition oracle for trees. Our partitioning technique builds on a classification procedure that partitions the vertex set into three main categories. For the sake of exposition, we describe the procedure assuming the graph is a rooted tree. We classify vertices whose subtree size is below a given threshold as *light*, and all remaining vertices as *heavy*. We further partition the heavy vertices into *dominant* and *recessive*, according to the number of heavy children: a heavy vertex with more than one heavy child is called dominant, and otherwise it is recessive. We show that the number of dominant vertices is small, and hence we can afford to make each of them a singleton in the final partition. The main technical challenge is therefore to partition the intervals formed by recessive heavy vertices together with their attached light subtrees into small subintervals. We show that this can be achieved efficiently in a local manner via a combination of sampling and a careful exploration paradigm. Finally, we emphasize that our algorithm does not rely on the presence of a rooted tree. Extending the classification and partitioning scheme to this setting introduces additional subtleties, which are handled by our approach.

1.3.3 Third Layer

In the third layer, we closely follow the paradigm presented in [9]. Each cluster is independently marked with probability $\Theta(1/n^{1/3})$, yielding $\tilde{O}(n^{1/3})$ marked clusters with high probability. Unmarked clusters are then grouped around adjacent marked clusters, forming larger structures that we refer to as *super-clusters*.

Finally, we exploit the available budget to add at most one, carefully chosen edge, between each cluster and an adjacent super-cluster. This guarantees that all clusters become connected while adding only a small number of additional edges, and therefore only a small fraction of total weight to the constructed subgraph.

Since the super-clusters may be too large we cannot locally construct them. Instead, we only construct the marked cluster in the center of the super-cluster and reveal the identity of the Prims' cells that are adjacent to it. For each adjacent cluster A, we add an edge connecting A and an adjacent marked cluster only if this edge minimizes some local conditions.

1.3.4 The local implementation

At a high level, the local implementation mimics the global construction by reconstructing only the relevant portion of the structure around the queried edge. Given a query (i.e., whether an edge $e = u, v$ belongs to the output), the algorithm treats each endpoint separately and locally simulates the decisions that the global procedure would make around both u and v, by exploring bounded neighborhoods and using shared randomness to ensure consistency across queries. In the first layer, the algorithm locally emulates the behavior of Prim's algorithm from each endpoint until reaching a designated center, thereby determining the local Prim cells of u and v. In the second layer, it invokes the partition oracle to determine the clusters containing u and v, again by exploring only small portions of the graph. Finally, in the third layer, it identifies the relevant inter-cluster connections by examining a limited set of candidate edges; in particular, this requires reconstructing only a constant number of clusters. The key point is that each of these steps depends only on a sublinear number of probes, and the use of a fixed random seed ensures that all local views are consistent with a single global solution.

2 Preliminaries

Throughout this paper, we consider a bounded degree (undirected) simple graph $G = (V, E, w)$, where $V = [n]$ and its maximum degree is $d = \max_{v \in V} d_G(v)$, where $d_G(v)$ denotes the degree of v w.r.t. the graph G. The identifier (ID in short) of a vertex $v \in V$ is simply v.

The total order over the vertices induces a total order (ranking) ρ over the edges of the graph in the following straightforward manner: for any $\{u, v\}, \{u', v'\} \in E$, $\rho(\{u, v\}) < \rho(\{u', v'\})$ if and only if $\min\{u, v\} < \min\{u', v'\}$ or $\min\{u, v\} = \min\{u', v'\}$ and $\max\{u, v\} < \max\{u', v'\}$.

We assume, without loss of generality, that all edge weights in the graph are distinct [4]. This ensures that the minimum spanning tree (MST) of the graph is unique. Thus, whenever we refer to *the* MST, we mean this unique spanning tree.

[4] We break ties by the ranks of the edges.

We recall that Prim's algorithm is a greedy algorithm that constructs a minimum spanning tree by starting from an arbitrary vertex and repeatedly adding the minimum-weight edge crossing the cut between the already constructed tree and the remaining vertices.

3 Prim's Cells – Localizing Prim's Algorithm

In this section, we describe the algorithm Local-Prim's-Algorithm, a localized variant of Prim's algorithm. Given a starting vertex, Local-Prim's-Algorithm locally simulates Prim's MST exploration from that vertex, revealing only a small portion of the tree and stopping well before the entire MST is constructed. Thereafter, we prove that these local simulations suffice to consistently recover most MST edges.

Algorithm 1 Local-Prim's-Algorithm.

Input: a vertex $v \in V$, a set of centers $\mathcal{S}$.
Output: the unique path in the MST between v and $\text{Center}_{\mathcal{S}}(v)$

1. Set $T = \{v\}$ and $H = \emptyset$.
2. Start with a tree containing a single vertex, v.
3. Expand the tree by adding one edge: Add the edge $e = \{u, w\}$, of minimum weight from which $u \in T$ and $w \in V \setminus T$. Add w to T and e to H.
4. Repeat Step 3 until an edge incident to a vertex in $\mathcal{S}$ is added to the tree. Let $\text{Center}_{\mathcal{S}}(v)$ denote this vertex.

The algorithm Local-Prim's-Algorithm (Algorithm 1), on input $v \in V$ and a set of centers $\mathcal{S}$, expands the MST starting from v until it reaches one of the vertices in $\mathcal{S}$. By construction, Local-Prim's-Algorithm on input $(v, \mathcal{S})$ finds $\text{Center}_{\mathcal{S}}(v)$. Moreover, it also identifies the unique path in the MST between v and $\text{Center}_{\mathcal{S}}(v)$.

▶ **Definition 3.** *We define the* Prim-cell *of a vertex v with respect to a set of centers $\mathcal{S}$, denoted by $\text{Prim}_{\mathcal{S}}(v)$, to be the set of all vertices u such that $\text{Center}_{\mathcal{S}}(u) = \text{Center}_{\mathcal{S}}(v)$.*

In the next claim, we prove that the subgraph induced by each Prim cell is connected and, moreover, is spanned by edges of the MST. Due to space constraints, the proof of this claim, as well as some additional proofs and pseudocode, are deferred to the appendix.

▷ Claim 4. For every $v \in V$ and $\mathcal{S} \subset V$, the subgraph induced on $\text{Prim}_{\mathcal{S}}(v)$ is connected. Moreover, it is spanned by a subgraph of the MST. We denote this subgraph by $\text{Tree}_{\mathcal{S}}(v)$.

Proof. Let v be a vertex with center x, and let $v = v_0, v_1, \ldots, v_\ell = x$ be the path from v to x in the MST. It suffices to show that x is also the center of v_1, since the general statement then follows by induction along the path.

Consider an execution of Prim's algorithm starting from v. For $i = 0, 1, \ldots, \ell - 1$, let T_i denote the tree just before the edge $e_i = \{v_i, v_{i+1}\}$ is added. Thus e_i is the minimum-weight outgoing edge of T_i.

Now consider an execution of Prim's algorithm starting from $u := v_1$. For $i = 1, \ldots, \ell - 1$, let U_i denote the analogous tree (these are well-defined since the MST is unique).

We claim that $U_i \subseteq T_i$ for all i. Suppose for contradiction that this fails, and let i be the smallest index such that $U_i \nsubseteq T_i$.

Let $e = \{w, y\}$ be the first edge added in the execution from u that is not in T_i, and let U_i' be the tree just before adding e. By the choice of e, all edges added prior to this point lie in T_i, and hence $U_i' \subseteq T_i$.

We now argue that $v_i \in U'_i$. If $i = 1$ then clearly $v_1 \in U'_1$, because the execution starts from v_1. For $i \geq 2$, observe that the edge $e_{i-1} = \{v_{i-1}, v_i\}$ is the first edge added to U_{i-1}, hence, it must have been added before reaching U'_i, implying that $v_i \in U'_i$.

Since $v_i \in U'_i \subseteq T_i$ and $v_{i+1} \notin T_i$, it follows that the edge $e_i = \{v_i, v_{i+1}\}$ is an outgoing edge of U'_i. Moreover, e is also an outgoing edge of U'_i by construction.

However, e_i is the minimum-weight outgoing edge of T_i, and hence also of $U'_i \subseteq T_i$, contradicting the fact that the execution from u selects e before e_i.

Therefore, $U_i \subseteq T_i$ for all i, and in particular $U_{\ell-1} \subseteq T_{\ell-1}$. This implies that the execution from v_1 reaches x before encountering any other center, and hence x is also the center of v_1, as required. ◁

4 Nearly Optimal Partition Oracle for Trees

We begin by formally defining a partition oracle, introduce the definitions needed to describe the global partition, and finally present our partition oracle, which locally reconstructs the partition.

▶ **Definition 5** (Partition Oracle for Trees). *Let $T = (V, E)$ be a tree of maximum degree d and let $\epsilon \in (0, 1)$. A* partition oracle *for trees at proximity parameter ϵ is a possibly randomized local algorithm that, given query access to the adjacency-list of T and a vertex $v \in V$, returns the part $P(v) \subseteq V$ containing v, where the collection $\mathcal{P} = \{P_1, \ldots, P_m\}$ satisfies:*

1. ***(Connectivity)*** *Each P_i induces a connected subgraph of T.*
2. ***(Small boundary)*** *With high constant probability (e.g., 2/3), the number of inter-part edges is at most $\epsilon d|V|$, i.e., $\left|\{\{x, y\} \in E : x \in P_i,\ y \in P_j,\ i \neq j\}\right| \leq \epsilon d|V|$.*
3. ***(Size bound)*** $\max_i |P_i| \leq f(\epsilon)$ *for some function f independent of $|V|$ (typically $f(\epsilon) = \mathrm{poly}(1/\epsilon)$).*

The oracle must answer each query without materializing the entire partition, using only local exploration of T, and with running time and query complexity sublinear in $|V|$. The oracle is consistent*: fixing its random seed determines a single partition $\mathcal{P}$, and repeated queries return labels consistent with that same partition.*

▶ **Definition 6** (Subtree w.r.t. an incident edge). *Let $T = (V, E)$ be a tree. For adjacent vertices $u, v \in V$, write $T(u \mid v)$ for the unique connected component of $T - \{(u, v)\}$ that contains u (equivalently, the component containing u in $T \setminus \{v\}$). Let $s(u \mid v) = |V(T(u \mid v))|$ denote the size of this component.*

We next define our classification method, which is a crucial part of our algorithm; see Figure 1 for an illustration.

▶ **Definition 7** (Light, heavy–recessive, heavy–dominant (threshold k)). *Fix a threshold $k \in \mathbb{N}$ where $k < (|V| - 1)/d$. For a vertex $v \in V$, let $N(v) = \{u \in V : (u, v) \in E\}$ denote the set of neighbors of v, and define $N_{>k}(v) = \{u \in N(v) : s(u \mid v) > k\}$, $c(v) = |N_{>k}(v)|$. We classify v as follows:*

- ***Light:*** *$c(v) = 1$ (exactly one neighbor's component exceeds k).*
- ***Heavy–recessive:*** *$c(v) = 2$ (exactly two neighbors' components exceed k).*
- ***Heavy–dominant:*** *$c(v) \geq 3$ (more than two neighbors' components exceed k).*

We also define a vertex v to be **special** if it is light and $s(v \mid p(v)) > k$, where $p(v)$ denotes the unique vertex in $N_{>k}(v)$.

4.1 The Global partition $\mathcal{P}$

The global partition is determined by the vertices that are selected to be *singletons*. All *heavy–dominant* vertices are singletons by definition. After removing all heavy–dominant vertices, the remaining heavy vertices are all *recessive*.

Heavy intervals (paths of recessive vertices). Consider the subgraph induced by the heavy–recessive vertices. Since every heavy–recessive vertex has at most two heavy–recessive neighbors [5], it follows that each connected component of this induced subgraph is a simple path. We call each such path a *heavy interval*, that is, a maximal chain of recessive vertices.

Weights of recessive vertices and intervals. For a recessive vertex v, let S_v be the set of vertices consisting of v together with all vertices in the light components attached to v, namely

$$S_v = \{v\} \cup \bigcup_{\substack{u \in N(v) \\ u \text{ is light}}} V(T(u \mid v)).$$

Define the weight of v to be $w(v) = |S_v|$.

Let $I = (v_0, \ldots, v_\ell)$ be a sub-interval of a heavy interval. Define its *interval weight* as $w(I) = \sum_{i=0}^{\ell} w(v_i)$.

Partition of heavy intervals. Every vertex independently tosses a coin and becomes *sampled* with probability $\epsilon/2$. A *heavy–recessive* or a *special* vertex v becomes a singleton in either of the following cases:

1. v is sampled itself; or
2. v has at least one sampled vertex in one of its light subtrees.

We refer to such vertices as *induced singletons*.

Set $k \stackrel{\text{def}}{=} \lceil 2\epsilon^{-1} \ln(2/\epsilon) \rceil$ and let $I = (v_0, \ldots, v_\ell)$ be a maximal sub-interval that contains v and such that none of the vertices v_i is an induced singleton. If $w(I) > k$, then v becomes a singleton. We refer to such vertices as *remote singletons*.

The part of a vertex that is not a singleton is defined as its connected component in the graph obtained after removing all singleton vertices. This concludes the description of the partition. We next prove that the partition has small boundary. Since the input graph is a tree, it suffices to prove the number of parts is small.

4.1.1 Bounding the number of parts

We begin by proving that every heavy vertex has a special vertex in each one of its non-light subtrees. We then prove a general claim on the relation between the number of leaves and the number of vertices of degree greater than 2 in trees.

Finally, we combine these claims with a bound on sums of partially dependent Bernoulli random variables to bound the number of parts in $\mathcal{P}$.

[5] Indeed, let v be a heavy–recessive vertex, and let $u \in N(v)$ be a neighbor such that $s(u \mid v) \leq k$. We claim that u cannot be heavy–recessive (nor heavy–dominant). To see this, suppose for contradiction that u has a neighbor $x \neq v$ such that $s(x \mid u) > k$. Then the component $T(u \mid v)$ contains the entire subtree $T(x \mid u)$, and therefore $s(u \mid v) \geq s(x \mid u) > k$, contradicting the assumption that $s(u \mid v) \leq k$.

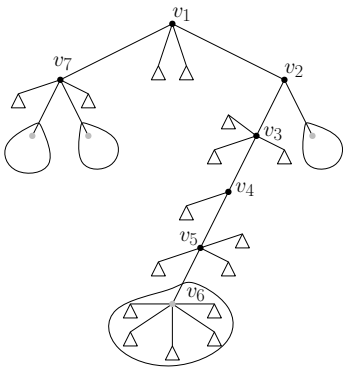

Figure 1 The triangles depict light subtrees. The black vertices v_1, v_2, and v_7 are heavy–dominant vertices, since more than two of their neighbors' components exceed the threshold. The vertex v_6 (as well as the other gray vertices) is a special vertex, since its subtree exceeds the threshold while none of its children's subtrees do. The vertices v_3, v_4, v_5 are heavy–recessive vertices.

▷ Claim 8. Let v be a heavy vertex (either dominant or recessive). Then for every neighbor $u \in N(v)$ such that $s(u \mid v) > k$, the subtree $T(u \mid v)$ contains at least one special vertex.

Proof. Fix a heavy vertex v. Fix $u \in N(v)$ such that $s(u \mid v) > k$. We prove the claim by induction on $s(u \mid v)$.

If u is light then is has to be special because $N_{>k}(u) = 1$ and v is the unique vertex in $N_{>k}(u)$. To see this, observe that for every heavy vertex, v, $s(v \mid u) > k$ for every $u \in N(v)$.

If u is heavy, then there exists $w \in N(u)$ such that $w \in T(u \mid v)$ and $s(w \mid u) > k$ (since $c(v) \geq 2$). Thus, the claim follows by an inductive argument. ◁

▷ Claim 9. Let T be a tree and let L be its set of leaves. Let $B = \{v \in V(T) : \deg_T(v) \geq 3\}$. Then $|B| \leq |L| - 2$.

Proof. Since T is a tree, $\sum_{v \in V(T)} \deg_T(v) = 2(|V(T)| - 1)$. Write $V(T) = L \cup I_2 \cup B$, where I_2 is the set of vertices of degree 2. Then $2(|V(T)| - 1) \geq |L| + 2|I_2| + 3|B|$. On the other hand, $2(|V(T)| - 1) = 2(|L| + |I_2| + |B|) - 2$. Comparing the two expressions yields $|B| \leq |L| - 2$. ◁

▶ **Corollary 10.** *Let $T = (V, E)$ be a tree. The number of heavy-dominant vertices in T is bounded from above by the number of special vertices in T.*

Proof. Remove from T every light vertex which is not special. By Claim 8, in the obtained tree all leaves are special. Thus, by Claim 9 the number of Heavy-dominant vertices in T is at most the number of special vertices, as claimed. ◀

We use the following theorem to show that with high probability, the number of remote singletons is sufficiently small.

▶ **Theorem 11** (Janson's inequality for a dependency graph [7, Theorem 2.3]). *Let $Y_1, \ldots, Y_m$ be $\{0,1\}$-valued random variables and let $Y = \sum_{i=1}^{m} Y_i$ and $\mu = \mathbb{E}[Y]$. Assume that $(Y_i)_{i=1}^{m}$ admits a dependency graph of maximum degree at most Δ, that is, there exists a graph D on $[m]$ such that for every i, the variable Y_i is independent of $\{Y_j : j \notin \{i\} \cup N_D(i)\}$. Then for every $t \geq 0$, $\Pr[Y \geq \mu + t] \leq \exp\left(-\frac{8t^2}{25(\Delta+1)(\mu+t/3)}\right)$.*

Define R_v to be the indicator variable for the event that v is remote. Denote $R = \sum_{v \in V} R_v$, $\mu = \mathbb{E}[R]$. We use the following definition to bound the degree of the dependency graph of $(R_v)_{v \in V}$.

▶ **Definition 12.** *Let v be a Heavy-recessive vertex, and let I denote the heavy interval containing v. Orient I arbitrarily, and refer to its two directions as left and right. The k-left-adjacent interval of v is the minimal interval adjacent to v on its left whose weight is at least k, or all vertices to the left of v in I if no such interval exists. The k-right-adjacent interval is defined analogously.*

Let $F(v)$ denote the set of vertices consisting of all vertices in the union of the k-left-adjacent and k-right-adjacent intervals of v, together with all vertices in the light subtrees hanging from these intervals.

Recall the partition of the heavy intervals as described in Section 4.1, and observe that R_v depends only on the random choices associated with vertices in $F(v)$. We next bound the number of remote vertices by analyzing the dependency graph of the random variables $(R_v)_{v \in V}$.

▷ Claim 13. With high probability (over the random sampling of the vertices), $R \leq \mu + O\left(\sqrt{\mu \cdot k \cdot \log n} + k \cdot \log n\right)$.

Proof. For every v, the set $F(v)$ intersects with at most $\Delta = 4k$ other sets $F(u)$.

Thus, the family $(R_v)_{v \in V}$ admits a dependency graph of maximum degree at most Δ, and therefore for any constant $c \geq 1$, with probability at least $1 - n^{-c}$,

$$R \leq \mu + C\left(\sqrt{\mu(\Delta+1)\, c \log n} + (\Delta+1)\, c \log n\right)$$

for a sufficiently large absolute constant C. This concludes the proof. ◁

▷ Claim 14 (Number of parts). The number of parts in $\mathcal{P}$ is ϵdn in expectation and at most $\epsilon dn + O\left(d(\epsilon\sqrt{n \log n \cdot \log(1/\epsilon)} + \epsilon^{-1} \log(1/\epsilon) \cdot \log n)\right)$, with high probability. Each part is of size at most $O(\epsilon^{-1} \log(1/\epsilon))$.

Proof. Start with T and iteratively remove all singleton vertices. Each removed singleton is incident to at most d edges, and therefore increases the number of connected components by at most d. Consequently, $|\mathcal{P}| \leq 1 + d \cdot (\#\text{singletons})$. It thus suffices to bound the total number of singleton vertices.

Let S denote the set of special vertices. Since for every pair of special vertices u_1 and u_2, $T(u_1 \mid p(u_1))$ and $T(u_2 \mid p(u_2))$ are vertex disjoint we obtain that $|S| \leq |V|/k$. By Corollary 10, the number of heavy–dominant vertices is at most $|S|$. Hence, it remains to bound the number of induced and remote singletons.

Induced singletons. Every induced singleton is either sampled itself or has a sampled vertex in one of its light subtrees. In particular, the number of induced singletons is at most the number of sampled vertices. Since each vertex is sampled independently with probability ϵ, a multiplicative Chernoff bound implies that, with probability at least $1 - n^{-c}$, the number of sampled vertices is concentrated around the expectation which is $\epsilon n/2$.

Remote singletons. Fix a recessive vertex v, and let $I(v)$ denote the maximal sub-interval in its heavy interval that contains v and contains no induced singleton. If v is a remote singleton, then $w(I(v)) > k$ and, in addition, $I(v)$ contains no sampled vertex in any of the light subtrees

contributing to the weight $w(I(v))$ (otherwise $I(v)$ would contain an induced singleton). This implies that either the minimal sub-interval to the right of v including v or the minimal sub-interval to the left of v of weight at least $k/2$ does not contain a sampled vertex (where right and left are defined arbitrarily). Therefore, $\Pr[v \text{ is remote}] \leq (1-\epsilon)^{(k/2)} \leq \epsilon/2$, for an appropriate setting of k. Thus, by linearity of expectation the total number of remote vertices is $\epsilon n/2$. The claim about the high probability follows by Claim 13.

The claim about the bound on the size of the parts follows by construction. ◁

4.2 Locally Reconstructing the partition $\mathcal{P}$

In this section, we describe how to locally reconstruct the partition $\mathcal{P}$. On query v, the algorithm (the listing of the algorithm appears as Algorithm 3 in the appendix) explores vertices, starting from v until the part of v in $\mathcal{P}$ is reconstructed.

The exploration is organized around a frontier of size at most two, corresponding to the directions in which the part of the vertex v may extend. The algorithm starts by classifying v (the listing of the algorithm appears as Algorithm 2 in the appendix). If v is light, then there is a unique direction in which the exploration can proceed; if v is heavy–recessive, there are exactly two such directions. These directions form the initial frontier.

Each element of the frontier is represented by a pair (u, x), where u is the next vertex to be explored on that side and x is the vertex from which u was reached. The algorithm iteratively processes the frontier while it is nonempty and the number of discovered vertices, that potentially belong to the part of v, does not exceed k.

When a vertex u is encountered, it is first classified. Crucially, the classification resumes from the previous stopping point, without re-exploring already visited vertices. If u is heavy–dominant, or if u is an induced singleton, then the exploration in the corresponding direction is terminated and that side of the frontier is set to NULL. Otherwise, u belongs to the same part as v (unless v is eventually declared remote) and is added to the discovered set. The exploration then continues through u in the unique direction that does not lead back to the caller x. Concretely, this is done by identifying the set $N_{>k}(u)$ and advancing to the unique vertex in $c(u) \setminus \{x\}$, if such a vertex exists. If no such vertex exists, the exploration in that direction is terminated.

The process continues independently on each side of the frontier. The exploration stops when both sides of the frontier are NULL or when more than k vertices have been discovered (excluding the exploration for identifying the singletons on the extreme points). In the latter case, the vertex v is declared remote and becomes a singleton; otherwise, the set of discovered vertices constitutes the part of v.

Algorithm 2 Classifying a vertex with threshold k.

Input: A tree $T = (V, E)$, a vertex $v \in V$, and a threshold $k \in \mathbb{N}$.
Output: The classification of v as *light*, *heavy–recessive*, or *heavy–dominant*.

1. For each neighbor $u \in N(v)$, determine the size $s(u \mid v)$ of the subtree $T(u \mid v)$ by exploring the component of $T - \{(u, v)\}$ that contains u, stopping the exploration as soon as more than k vertices are discovered.
2. Let $N_{>k}(v)$ be the set of neighbors u for which $s(u \mid v) > k$, and let $c(v) = |N_{>k}(v)|$.
3. Classify v according to the value of $c(v)$:
   - if $c(v) = 1$, declare v to be *light*;
   - if $c(v) = 2$, declare v to be *heavy–recessive*;
   - if $c(v) \geq 3$, declare v to be *heavy–dominant*.

Algorithm 3 Partition Oracle for Trees.

Input: Query access to a tree $T = (V, E)$ and a vertex $v \in V$.
Output: The part of v in $\mathcal{P}$.

1. Initialize the discovered set $X \leftarrow \{v\}$.
2. Initialize a frontier consisting of at most two *sides*. Each side is either NULL or a pair (u, x) where u is the next vertex to visit on that side, and x is the vertex from which u was reached (the "caller").
3. Set the initial frontier as follows:
   a. If v is light, let a be the unique vertex in $c(v) = N_{>k}(v)$ and set the frontier to (a, v) on one side and NULL on the other.
   b. If v is heavy–recessive, let a, b be the two vertices in $c(v)$ and set the frontier sides to (a, v) and (b, v).
4. While at least one side of the frontier is not NULL and $|X| \leq k$, repeat the following steps for each non-NULL side independently.
5. Let (u, x) be the current state of the chosen side (so u is to be processed and x is its caller). Remove (u, x) from the frontier.
6. Classify u. The classification resumes from the previous stopping point, without re-exploring already visited vertices.
   a. If u is heavy–dominant, or u is an induced singleton, then set this side of the frontier to NULL.
   b. Else, if u is light, then add the vertices in its light subtrees to X and let y be the unique vertex in $N_{>k}(u)$. If $y \neq x$, update this side of the frontier to (y, u). Otherwise, set this side of the frontier to NULL.
   c. Otherwise (u is heavy–recessive which is not an induced singleton), add u and its light subtrees to X.
      i. If x is not light or x is special (namely, x is light and $x \in N_{>k}(u)$), then let y be the unique vertex in $N_{>k}(u) \setminus \{x\}$. Update this side of the frontier to (y, u).
      ii. Otherwise, x is light (the frontier is empty). Add (y, u) to the frontier for every $y \in N_{>k}(u)$.

The procedure terminates when both frontier sides are NULL or when $|X| > k$. In the latter case, it returns $\{v\}$, otherwise it returns X.

▶ **Theorem 15.** *Algorithm 3 is a partition oracle for trees whose query complexity is* $O(d \cdot \epsilon^{-1} \log(1/\epsilon))$.

Proof. The desired properties of the partition $\mathcal{P}$ follow from the construction and Claim 14. To bound the query complexity, observe that all vertices explored by the algorithm belong to X, except for those involved in the classification of the two endpoints of the interval, which become singletons. Since the exploration resumes without revisiting any vertex, the overall query complexity is bounded by $O(dk)$, by combining the bound on $|X|$ with the $O(dk)$ query complexity of Algorithm 2. ◀

5 The Algorithm

The algorithm first picks a random set of *centers*, denoted by $\mathcal{S}$. Each $v \in V$ is selected to be a center with probability $p_1 \stackrel{\text{def}}{=} \frac{\alpha^{2/3}}{n^{1/3} \log n}$ where $\alpha \stackrel{\text{def}}{=} \frac{\epsilon w_{\min}}{w_{\max}}$. We note that it suffices to use bounded independence, using standard tools similar to those in [17]. For the sake of brevity, we assume access to truly random bits.

We consider the set $\text{Prim}_{\mathcal{S}}(v)$ and the tree $\text{Tree}_{\mathcal{S}}(v)$ rooted at $\text{Center}_{\mathcal{S}}(v)$. We partition $\text{Prim}_{\mathcal{S}}(v)$ into *clusters* using the partition oracle described in the previous section, with parameter ϵ set to $\Theta(p_1/d)$. Let $\text{Cluster}_{\mathcal{S}}(v)$ denote the cluster of v according to this partition.

5.1 Marked-cells, lonely-clusters and super-clusters

Each Prim cell is selected to be marked w.p. $p_2 \stackrel{\text{def}}{=} 1/(\alpha \cdot n)^{1/3}$. A cluster is marked if it belongs to a marked cell. A *super-cluster* of a marked cluster C is the union of C with all the clusters that are incident to it in G. Namely, all clusters, B, such that $E(C, B) \neq \emptyset$.

We say that a cluster C is *lonely* if all the clusters that are incident to C are not marked.

▶ **Definition 16** (Assigned Cluster)**.** *If a cluster A is neither marked nor lonely, then we say that A is* assigned *to one of its adjacent marked clusters. Specifically, it is assigned to*

$$C^* = \arg \min_{\substack{C \text{ marked} \\ E(A,C)\neq\emptyset}} \left\{ \min_{e\in E(A,C)} w(e) \right\},$$

that is, A is assigned to the adjacent marked cluster whose lightest connecting edge to A has the smallest weight.

Note that a cluster C may be contained in several super-clusters but is assigned to at most one marked cluster. For a cluster A let $\text{Neighbor-cells}_{\mathcal{S}}(A)$ denote the set of centers $s \in \mathcal{S}$ such that there exists $\{u, v\}$ where $u \in A$ and $\text{Center}_{\mathcal{S}}(v) = s$.

5.2 The Edge Set

We next describe the global algorithm for constructing G' (Algorithm 4).

Algorithm 4 Global construction of G'.

For two disjoint subsets of vertices C and D, we denote by $\text{Min-Weight}(C, D)$ the minimum-weight edge in $E(C, D)$. When we say that we *link C and D*, we mean that we add the edge $\text{Min-Weight}(C, D)$ to E'.

1. For every vertex v, add the edge between v and its parent $\text{Parent}_{\mathcal{S}}(v)$.
2. For every cluster A, and every marked Prim-cell $\mathcal{C}$ that is incident to A, link A and $\mathcal{C}$.
3. For every lonely cluster A, and every Prim-cell $\mathcal{C}$ that is incident to A, link A and $\mathcal{C}$.
4. For every pair of clusters A and B that are both non-lonely and unmarked, link A and B if *both* of the following conditions hold:
   a. $\text{Min-Weight}(A, \text{Prim}(B)) = \text{Min-Weight}(A, B)$, where $\text{Prim}(B)$ denotes the Prim-cell of the cluster B.
   b. The center $\text{Center}_{\mathcal{S}}(B)$ has minimum rank among the centers in $\text{Neighbor-cells}_{\mathcal{S}}(A) \cap \text{Neighbor-cells}_{\mathcal{S}}(C)$, where C denotes the marked cluster to which B is assigned (recall that B is not lonely).

▷ Claim 17. The subgraph G' is connected

5.3 Optimality

We use the following simple claims to prove that the output of the algorithm is close to optimal and for bounding the query complexity.

▷ Claim 18. For a fixed $H \subseteq V$ of size at least $cp_1^{-1} \log n$, where c is a sufficiently large constant, with high probability, it holds that $H \cap \mathcal{S} \neq \emptyset$.

▷ Claim 19. For a fixed $H \subseteq V$ of size at least $cp_2^{-1} \log n$, where c is a sufficiently large constant, with high probability, at least one of the vertices in H is marked.

▷ Claim 20. The expected contribution to the weight of E' from edges that do not belong to the MST is $O(\epsilon) \cdot OPT$.

Proof. Every edge that belongs to one of the subtrees that span the Prim-cells, also belongs to the MST, thus, it suffices to bound the expected total weight of the remaining edges added to E' in Steps 2-4 of the global algorithm (Algorithm 4).

Let s denote the number of marked cells, r denote the number of marked clusters, and ℓ denote the number of clusters.

By construction, Step 2 contributes at most $\ell \cdot s$ edges to E' which bounded above by $\ell \cdot r$.

By Claim 19, with high probability, a cluster A for which Neighbor-cells$_{\mathcal{S}}(A)$ is greater than $t \stackrel{\text{def}}{=} O(p_2^{-1} \log n)$ is not lonely, thus the expected number of edges that are added due to Step 3 is at most $\ell \cdot t + 1$.

Step 4 contributes at most $\ell \cdot r$ edges. To see this, consider any cluster A and any marked cluster C. By Steps 4a and 4b, A is linked in Step 4 to at most a single cluster that is assigned to C.

Thus, the number of edges that are added in Steps 2 and 4 is bounded by $2\ell \cdot r$.

Let $\mathcal{E}$ denote the event that $\ell \leq c_\ell p_1 n$, where c_ℓ is a constant. By Claim 14, the event $\mathcal{E}$ holds with high probability for sufficiently large c_ℓ.

Conditioned on the realized partition into cells and clusters, each cluster is marked with marginal probability p_2. Therefore, by linearity of expectation, $\mathbb{E}[r] = p_2 \ell$.

Hence, $\mathbb{E}[\ell r \cdot \mathbf{1}_{\mathcal{E}}] = p_2 \mathbb{E}[\ell^2 \cdot \mathbf{1}_{\mathcal{E}}]$. Since $\ell \leq c_\ell p_1 n$ on the event $\mathcal{E}$, $\mathbb{E}[\ell^2 \cdot \mathbf{1}_{\mathcal{E}}] \leq c_\ell p_1 n \cdot \mathbb{E}[\ell] \leq 2c_\ell p_1^2 n^2$, where the last inequality follows from $\mathbb{E}[\ell] \leq 2p_1 n$ (which follows by Claim 14). Hence, $\mathbb{E}[\ell r \cdot \mathbf{1}_{\mathcal{E}}] \leq 2c_\ell p_1^2 p_2 n^2$. The contribution of $\overline{\mathcal{E}}$ to the expectation is negligible by the high-probability bound on ℓ and the trivial bounds $\ell, r \leq n$.

Overall, we obtain a bound on the expectation of the number of edges added in Steps 2-4 which is $O(\frac{w_{\min} \epsilon n}{w_{max}})$. Since OPT is bounded from below by $(n-1) \cdot w_{\min}$, and the weight of every edge is bounded by $w_{\max}$ we obtain that, in expectation, the total weight of these edges is $O(w_{\min} \epsilon n) = O(\epsilon) OPT$, as claimed. ◁

5.4 Query complexity of the local implementation

We next go over the steps of Algorithm 4 and describe how to implement them locally.

To implement Step 1, we run Local-Prim's-Algorithm from both u and v, testing membership in $\mathcal{S}$ only for vertices that are actually explored. At the end of the execution, we obtain Center$_{\mathcal{S}}(u)$ and Center$_{\mathcal{S}}(v)$, as well as the unique paths from u to Center$_{\mathcal{S}}(u)$ and from v to Center$_{\mathcal{S}}(v)$. Hence, we can determine whether u is the parent of v or vice versa. By Claim 18, this can be implemented with high probability using $O(p_1^{-1} \log n)$ queries.

To implement the remaining steps, it suffices to perform the following task a constant number of times. Given a vertex w, we first compute its cluster, Cluster$_{\mathcal{S}}(w)$. We then enumerate all vertices that are adjacent to at least one vertex in Cluster$_{\mathcal{S}}(w)$. For each such vertex, we invoke Local-Prim's-Algorithm, which allows us to identify its center and, in particular, to determine whether it belongs to a marked cluster. Moreover, by Definition 16, we can determine, for any cluster B, the cluster to which B is assigned by comparing the weights of the edges connecting B to adjacent marked clusters. In particular, this does not require reconstructing the cluster to which B is assigned.

▷ Claim 21. The query complexity of locally computing Algorithm 4 is $\tilde{O}\left(n^{2/3} \cdot d^2 \cdot (\epsilon \cdot \frac{w_{\min}}{w_{\max}})^{-4/3}\right)$.

Proof. By the description of the local implementation, up to a constant factor, the total query complexity is bounded by the query complexity of computing the cluster of a vertex plus the query complexity of Local-Prim's-Algorithm multiplied by the maximum cluster size and by d. By Claim 14, the size of a cluster is at most $O(\eta^{-1}\log(1/\eta))$ where $\eta = p_1/d$. By Claim 18 the Local-Prim's-Algorithm takes $O(p_1^{-1}\log n)$. The query complexity of finding a cluster is $O(\eta^{-1}d\log(1/\eta))$ times the query complexity of Local-Prim's-Algorithm. So overall, we obtain that the query complexity is $O((d\log n/p_1)^2)$, which is $\tilde{O}\left(n^{2/3}\cdot d^2\cdot\left(\frac{\epsilon w_{\min}}{w_{\max}}\right)^{-4/3}\right)$ as claimed. ◁

The proof of our main theorem, stated next, follows from Claims 20 and 21 by invoking the algorithm with approximation parameter $c\epsilon$ for a sufficiently small constant $c > 0$, and applying Markov's inequality to convert the expected excess weight bound of Claim 20 into a constant-probability $(1+\epsilon)$-approximation.

▶ **Theorem 22.** *There exists a local algorithm for $(1+\epsilon)$-approximating the minimum weight spanning tree of a graph $G = (V, E, w)$ with positive weights and degree bounded by d, using $\tilde{O}\left(|V|^{2/3}\cdot d^2\cdot\left(\frac{\epsilon w_{\min}}{w_{\max}}\right)^{-4/3}\right)$ probes per query where $w_{\max} = \max_{e\in E} w(e)$ and $w_{\min} = \min_{e\in E} w(e)$.*

References

1. Noga Alon, Ronitt Rubinfeld, Shai Vardi, and Ning Xie. Space-efficient local computation algorithms. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 1132–1139. SIAM, 2012. doi:10.1137/1.9781611973099.89.
2. Rubi Arviv, Lily Chung, Reut Levi, and Edward Pyne. Improved Local Computation Algorithms for Constructing Spanners. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2023)*, volume 275 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 42:1–42:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.APPROX/RANDOM.2023.42.
3. Bernard Chazelle, Ronitt Rubinfeld, and Luca Trevisan. Approximating the minimum spanning tree weight in sublinear time. *SIAM Journal on computing*, 34(6):1370–1379, 2005. doi:10.1137/S0097539702403244.
4. Artur Czumaj, Pan Peng, and Christian Sohler. Testing cluster structure of graphs. In *Proceedings of the forty-seventh annual ACM symposium on Theory of Computing*, pages 723–732, 2015. doi:10.1145/2746539.2746618.
5. Alan Edelman, Avinatan Hassidim, Huy N. Nguyen, and Krzysztof Onak. An efficient partitioning oracle for bounded-treewidth graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 530–541, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-22935-0_45.
6. Avinatan Hassidim, Jonathan A Kelner, Huy N Nguyen, and Krzysztof Onak. Local graph partitions for approximation and testing. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 22–31. IEEE, 2009. doi:10.1109/FOCS.2009.77.
7. Svante Janson. Large deviations for sums of partly dependent random variables. *Random Structures & Algorithms*, 24(3):234–248, 2004. doi:10.1002/RSA.20008.
8. Akash Kumar, C. Seshadhri, and Andrew Stolman. Random walks and forbidden minors iii: poly$\left(d\varepsilon^{-1}\right)$-time partition oracles for minor-free graph classes. In *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 257–268, 2022. doi:10.1109/FOCS52979.2021.00034.

9 Christoph Lenzen and Reut Levi. A Centralized Local Algorithm for the Sparse Spanning Graph Problem. In *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 87:1–87:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2018. doi:10.4230/LIPIcs.ICALP.2018.87.

10 Reut Levi and Moti Medina. A (centralized) local guide. *Bulletin of the EATCS*, 122:60–92, 2017. URL: http://eatcs.org/beatcs/index.php/beatcs/article/view/495.

11 Reut Levi, Moti Medina, and Omer Tubul. Nearly Optimal Local Algorithms for Constructing Sparse Spanners of Clusterable Graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2024)*, volume 317 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 60:1–60:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.APPROX/RANDOM.2024.60.

12 Reut Levi, Guy Moshkovitz, Dana Ron, Ronitt Rubinfeld, and Asaf Shapira. Constructing near spanning trees with few local inspections. *Random Structures & Algorithms*, 50(2):183–200, 2017. doi:10.1002/RSA.20652.

13 Reut Levi and Dana Ron. A quasi-polynomial time partition oracle for graphs with an excluded minor. *ACM Transactions on Algorithms (TALG)*, 11(3):1–13, 2015. doi:10.1145/2629508.

14 Reut Levi, Dana Ron, and Ronitt Rubinfeld. Local Algorithms for Sparse Spanning Graphs. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2014)*, volume 28 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 826–842. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/LIPIcs.APPROX-RANDOM.2014.826.

15 Reut Levi, Dana Ron, and Ronitt Rubinfeld. Local algorithms for sparse spanning graphs. *Algorithmica*, 82(4):747–786, 2020. doi:10.1007/S00453-019-00612-6.

16 Reut Levi and Nadav Shoshan. Testing hamiltonicity (and other problems) in minor-free graphs. In Mary Wootters and Laura Sanità, editors, *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2021, August 16-18, 2021, University of Washington, Seattle, Washington, USA (Virtual Conference)*, volume 207 of *LIPIcs*, pages 61:1–61:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.APPROX/RANDOM.2021.61.

17 Merav Parter, Ronitt Rubinfeld, Ali Vakilian, and Anak Yodpinyanee. Local computation algorithms for spanners. *Innovations in Theoretical Computer Science (ITCS)*, 2019.

18 Ronitt Rubinfeld. Can we locally compute sparse connected subgraphs? In *International Computer Science Symposium in Russia*, pages 38–47. Springer, 2017. doi:10.1007/978-3-319-58747-9_6.

19 Ronitt Rubinfeld, Gil Tamir, Shai Vardi, and Ning Xie. Fast local computation algorithms. In Bernard Chazelle, editor, *Innovations in Computer Science - ICS 2011, Tsinghua University, Beijing, China, January 7-9, 2011. Proceedings*, pages 223–238. Tsinghua University Press, 2011. URL: http://conference.iiis.tsinghua.edu.cn/ICS2011/content/papers/36.html.

A Omitted Proofs and missing details

▶ **Definition 23** (Local Computation Algorithm (LCA) - [19]). *Let $\mathcal{P}$ be a computational problem where, given an input x of size n, the goal is to compute a valid solution $y \in F(x)$. A randomized algorithm $\mathcal{A}$ is a* ***local computation algorithm*** *for $\mathcal{P}$ if, when given random tape r and probe access to input x:*

1. ***Query Handling:*** *For any query q_i (representing a specific part of the solution), $\mathcal{A}(q_i, x, r)$ returns a partial answer y_{q_i}.*
2. ***Sublinear Complexity:*** *The probe complexity per query is sublinear in n.*

3. ***Consistency:*** *The algorithm is* consistent *with a single global valid solution $y \in F(x)$. That is, for any sequence of queries $q_1, q_2, \ldots, q_m$, the algorithm produces an output sequence $y_{q_1}, y_{q_2}, \ldots, y_{q_m}$ that matches the values in y at those locations.*
4. ***Correctness:*** *For all x, with probability at least $2/3$ (over the random tape r) [6], the algorithm answers all queries consistently with a valid solution.*

Proof of Claim 17. Let $\{u, v\} \in E$. We say that u and v are connected in G' if there is a path between u and v in G'. By Step 1 every Prim-cell is spanned by G'. Thus if $\text{Center}_{\mathcal{S}}(v) = \text{Center}_{\mathcal{S}}(u)$ then u and v are connected in G'. If either u or v is a singleton then by Step 3 they remain connected. If either u or v belong to a marked cell then by Steps 1 and 2 they remain connected. If either u or v belong to a lonely cluster then by Steps 1 and 3 they remain connected. Let A and B denote the clusters of u and v, respectively. We are left with the case that both clusters are not marked and not lonely (if one of these clusters is either lonely or marked then we are done).

Let C denote the single marked cluster to which B is assigned. By Step 4 if both conditions of Steps 4a and 4b hold then the algorithm links A and B and thus u and v are connected. Henceforth we assume at least one of the conditions does not hold.

We define an element to be either a cluster or a singleton. We define a total order on the elements that are incident to A. If a pair of elements H and F are not in the same cell then $H \prec F$ iff the rank of $\text{Prim}(H)$ is smaller than the rank of $\text{Prim}(F)$, where the rank of a cell is the rank of its center. Otherwise, $H \prec F$ iff the weight of $\text{Min-Weight}(A, H)$ is smaller than the weight of $\text{Min-Weight}(A, F)$. The rest of the proof is by induction on the order of the element that is incident to A.

For the base case consider Y which is the element that is incident to A of minimum order. If Y and A are in the same cell, or Y is either a singleton or a marked cluster or a lonely cluster then A and Y are connected (due to Steps 1-3). Otherwise, Y and A are connected due to Step 4.

Otherwise, either the condition in Step 4a or the condition in Step 4b do not hold for A and B. This implies that there is an element $H \prec B$ that is incident to A and thus, by the induction hypothesis is connected to A. If H and B are in the same Prim-cell then H is also connected to B and by transitivity A and B are connected. Otherwise, there is an element F which is incident to C and in the same Prim-cell as H. Thus H and F are connected. Since C is marked, F and C are connected and, again, by transitivity C and A are connected. Since B is connected to C, it is connected to A, implying that u and v are connected, as desired. ◁

Proof of Claim 18. Each vertex is included in $\mathcal{S}$ independently with probability p_1. Hence,

$$\Pr[H \cap \mathcal{S} = \emptyset] = (1 - p_1)^{|H|} \leq \exp(-p_1|H|).$$

For $|H| \geq cp_1^{-1} \log n$ we get $\Pr[H \cap \mathcal{S} = \emptyset] \leq n^{-c}$. Thus, $H \cap \mathcal{S} \neq \emptyset$, with high probability. ◁

Proof of Claim 19. See proof of Claim 18. ◁

[6] Alternatively, one may require high success probability, namely, $1 - 1/n^c$, for any constant c (without changing the asymptotic complexity of the algorithm).

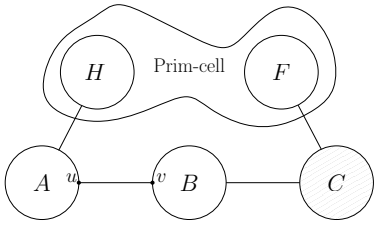

Figure 2 On query $\{u, v\}$ the algorithm recovers the clusters of u and v, denoted in the figure by A and B. The cluster C is the marked cluster that B is assigned to. If $F \prec B$, the algorithm does not add the edges $\{u, v\}$ to E' (since the rank of the cell of B is not the minimum among Neighbor-cells$_{\mathcal{S}}(A) \cap$ Neighbor-cells$_{\mathcal{S}}(C)$).