Near-Optimal Distributed Computation of Small Vertex Cuts

Merav Parter *
Weizmann Institute
merav.parter@weizmann.ac.il

Asaf Petruschka
Weizmann Institute
asaf.petruschka@weizmann.ac.il

Abstract

We present near-optimal algorithms for detecting small vertex cuts in the CONGEST model of distributed computing. Despite extensive research in this area, our understanding of the *vertex* connectivity of a graph is still incomplete, especially in the distributed setting. To this date, all distributed algorithms for detecting cut vertices suffer from an inherent dependency in the maximum degree of the graph, Δ. Hence, in particular, there is no truly sub-linear time algorithm for this problem, not even for detecting a *single* cut vertex. We take a new algorithmic approach for vertex connectivity which allows us to bypass the existing Δ barrier.

- As a warm-up to our approach, we show a simple $\widetilde{O}(D)$-round[1] randomized algorithm for computing all cut vertices in a D-diameter n-vertex graph. This improves upon the $O(D + \Delta/\log n)$-round algorithm of [Pritchard and Thurimella, ICALP 2008].
- Our key technical contribution is an $\widetilde{O}(D)$-round randomized algorithm for computing all cut *pairs* in the graph, improving upon the state-of-the-art $O(\Delta \cdot D)^4$-round algorithm by [Parter, DISC '19]. Note that even for the considerably simpler setting of *edge* cuts, currently $\widetilde{O}(D)$-round algorithms are currently known *only* for detecting pairs of cut edges.

Our approach is based on employing the well-known linear graph sketching technique [Ahn, Guha and McGregor, SODA 2012] along with the heavy-light tree decomposition of [Sleator and Tarjan, STOC 1981] . Combining this with a careful characterization of the survivable subgraphs, allows us to determine the connectivity of $G \setminus \{x, y\}$ for every pair $x, y \in V$, using $\widetilde{O}(D)$-rounds. We believe that the tools provided in this paper are useful for omitting the Δ-dependency even for larger cut values.

*This project is funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 949083), and by the Israeli Science Foundation (ISF), grant No. 2084/18.

[1]Throughout the paper, we use the notation $\widetilde{O}$ to hide poly-logarithmic in n terms.

Contents

1 Introduction and Our Contribution

The vertex connectivity of the graph is a central concept in graph theory and extensive attention has been paid to developing algorithms to compute it in various computational models. Recent years have witnessed an enormous progress in our understanding of vertex cuts, from a pure graph theoretic perspective [PY21] to many algorithmic applications [NSY19, LNP+21, PY21, HLW21]. Despite this exciting movement, our algorithmic toolkit for handling vertex cuts is still somewhat limited. A large volume of the work, in the centralized setting, has focused on fast algorithms for detecting minimum vertex cuts of size at most k, for some small number k. Until recently, near-linear time algorithms where known only for $k \leq 2$ [Tar72, HT73]. A sequence of recent breakthrough results [CKL+22, LNP+21] provides an almost-linear time sequential algorithm for computing the vertex connectivity (even for large connectivity values).

As we see soon, the situation is considerably worse in distributed settings, where the problem is still fairly open already for $k = 1$. Throughout, we consider the CONGEST model [Pel00]. In this model, each node holds a processor with a unique and arbitrary ID of $O(\log n)$ bits, and initially only knows the IDs of its neighbors in the graph. The execution proceeds in synchronous rounds and in each round, each node can send a message of size $O(\log n)$ to each of its neighbors. The primary complexity measure is the number of communication rounds. For n-vertex D-diameter graphs, Pritchard and Thurimella [PT11] presented a randomized algorithm for detecting a (single) cut vertex (a.k.a articulation point) within $O(D + \Delta/\log n)$ CONGEST rounds, where Δ is the maximum degree of the graph. [PT11] conclude their paper by noting:

> [PT11] *It would be interesting to know if our distributed cut vertex algorithm could be synthesized with the cut vertex algorithm of [Thu97] to yield further improvement. Alternatively, a lower bound showing that no $O(D)$-time algorithm is possible for finding cut vertices would be very interesting.*

No progress on the complexity of this problem has been done since then. For small cut values k, Parter [Par19] employed the well-known fault-tolerant sampling technique [WY13, KP21] for detecting k vertex cuts in $(\Delta \cdot D)^{\Theta(k)}$ deterministic rounds. Turning to approximation algorithms, for $k = \Omega(\log n)$, Censor-Hillel, Ghaffari and Kuhn [CHGK14] provided a $O(\log n)$ approximation for computing the *value* of the vertex connectivity of the graph within $\widetilde{O}(D + \sqrt{n})$ rounds. [CHGK14] also presented a lower bound of $\widetilde{\Omega}(D + \sqrt{n/k})$ V-CONGEST rounds. In the V-CONGEST model, each *node* (rather than an edge) is restricted to send only $O(\log n)$ bits, in total, in every round. As shown in this paper, this lower bound does not hold in the standard CONGEST model.

We follow the terminology of [PT11]: a *cut vertex* is a vertex x such that $G \setminus \{x\}$ is not connected. A *cut pair* is a pair of vertices x, y such that $G \setminus \{x, y\}$ is not connected. For brevity we call these objects, small cuts. Our main results in this paper are near-optimal algorithms for detecting these small cuts, in the sense that for every small cut, there is at least one vertex in the graph that learns it. Our first contribution is in presenting a (perhaps surprisingly) simple randomized algorithm[2] that can detect all cut vertices in the graph in $\widetilde{O}(D)$ rounds. The edge-congestion[3] of the algorithm is $\widetilde{O}(1)$ bits[4].

[2]As usual, all presented randomized algorithms in this paper have success guarantee of $1 - 1/n^c$, for any given constant $c > 1$.

[3]The edge congestion of a given algorithm is the worst-case bound on the total number of messages exchanged through a given edge e in the graph.

[4]We exploit this bounded congestion for detecting cut pairs.

Theorem 1.1. *There is a randomized algorithm that w.h.p. identifies all single cut-vertices in G within $\widetilde{O}(D)$ rounds. The edge congestion is $\widetilde{O}(1)$. In the output, each vertex $x \in V$ learns if it is a cut vertex.*

This settles the question raised by [PT11]. Our algorithm is based on the well-known *graph-sketching* technique of Ahn et al. [AGM12]. This technique has admitted numerous applications in the context of connectivity computation under various computational settings, e.g., [KKM13, KW14, GKKT15, KKT15, MK18, GP16, DP17, DP21]. Yet, to the best of our knowledge, it has not been employed before in the context of CONGEST algorithms for minimum vertex-cut computation.

We then turn to consider the problem of detecting cut pairs. It has been noted widely in the literature that there is a sharp qualitative difference between a single failure and two failures. This one-to-two jump has been accomplished by now for a wide-variety of fault-tolerant settings, e.g., reachability oracles [Cho16], distance oracles [DP09], distance preservers [Par15, GK17, Par20] and vertex-cuts [HT73, BT89, BT96, GILP15]. While it is relatively easy to extend our algorithm of Theorem 1.1 to detect cut pairs in $\widetilde{O}(D^2)$ rounds, providing a near-optimal complexity of $\widetilde{O}(D)$ rounds, turns out to be quite involved. Our key technical contribution is:

Theorem 1.2. *There is a randomized algorithm that w.h.p. identifies all cut pairs in G within $\widetilde{O}(D)$ rounds. For each cut pair x, y, either x or y learns that fact.*

We observe that even for the simpler problem of edge-connectivity (see Remark below), an $\widetilde{O}(D)$-round algorithm is currently only known for edge cuts of size at most *two* due to [PT11]. Hence, we are now able to match the complexity of these two problems for small cut values. Our algorithm is based on distinguishing between two structural cases depending on the locations of the cut pair x, y in a BFS tree of G. The first case which we call *dependent* handles the setting where the x and y have ancestry/descendant relations. The second *independent* case assumes that x and y are not dependent, i.e., $\mathrm{LCA}(x,y) \notin \{x,y\}$, where $\mathrm{LCA}(x,y)$ is the lowest/least common ancestor of x and y in the BFS tree. Each of these cases call for a different approach. We believe that the tools provided in this paper should hopefully pave the way towards detecting larger vertex-cuts with no dependency in the maximum degree Δ (as it is the case for the state-of-the-art algorithm by [Par19]). For a more in-depth technical overview, see Sec. 1.1.

Remark on the Edge-Connectivity. It is widely known that in undirected graphs, vertex connectivity and vertex cuts are significantly more complex than edge connectivity and edge cuts, for which now the following result are known: an $\widetilde{O}(m)$-time centralized exact algorithm [Kar99, GNT20, GMW20] and an $\widetilde{O}(D+\sqrt{n})$ exact distributed algorithms [DEMN21]. For constant values of edge-connectivity a $\mathrm{poly}(D)$-round algorithm is given in [Par19].

1.1 Our Approach, in a Nutshell

We provide the key ideas of our algorithms. Our end goal is to simulate a connectivity algorithm in the graph $G \setminus \{x,y\}$ for *every* pair of vertices $x, y \in V$. Note that this is not trivial already for a single x, y pair as the diameter of the subgraph $G \setminus \{x,y\}$ might be as large as $\Omega(\Delta D)$, hence using on-shelf connectivity algorithms lead to a round complexity of $O(\min\{D+\sqrt{n}, \Delta D\})$. We bypass this Δ dependency by using the edges incident to the vertices x, y as *shortcuts*. Then, to minimize the congestion imposed by running possibly n^2 connectivity algorithms in parallel, we employ a preprocessing phase in which we collect *graph-sketch* information (explained next) at each vertex x. This information allows each vertex x to pinpoint at a bounded number of cut-mate suspects. In addition, it allows x, in certain cases, to locally simulate connectivity queries without using further communication. Throughout, let T be a BFS tree rooted at some source s, and denote the T-paths by $\pi(\cdot,\cdot)$.

We start by employing the well-known *heavy-light tree decomposition* technique by Sleator and Tarjan [ST83]. This classifies the edges of T into light and heavy edges. The useful properties are that each vertex v has $O(\log n)$ light edges on its tree path $\pi(s,v)$, and in addition, each v is the parent of *one* heavy edge, connecting v to its unique heavy child. It is easy to compute this decomposition on T in $\widetilde{O}(D)$ rounds. For a vertex x, let T_x be the subtree of T rooted at x.

Basic Tools: Graph Sketches and Borůvka Algorithm. A *graph sketch* of a vertex v is a randomized string of $\widetilde{O}(1)$ bits that compresses v's edges [AGM12]. The linearity of these sketches allows one to infer, given the sketches of subset of vertices S, an outgoing cut edge $(S, V\setminus S)$ with constant probability. A common approach for deducing the graph connectivity merely from the sketches of the vertices is based on the well-known Borůvka algorithm [NMN01]. This algorithm works in $O(\log n)$ phases, where in each phase, from each growable component an outgoing edge is selected. All these outgoing edges are added to the forest, while ignoring cycles. Each such phase reduces the number of growable components by constant factor, thus within $O(\log n)$ phases a maximal forest is computed. Since this algorithm only requires the computation of outgoing edges it can simulated using $O(\log n)$ independent sketches for each of the vertices. In our algorithms, we aggregate graph sketches over the BFS tree T which allows the vertices x to locally simulate Borůvka in the graph $G\setminus\{x\}$. This is illustrated in our algorithm for detecting a single cut vertex, described next.

Warm Up: Detecting Single Cut Vertices. Our algorithm starts by letting each vertex v locally compute its individual $\mathsf{Sketch}_G(v)$. Then, by aggregating the sketches (using their linearity) from the leaf vertices to the root s over the BFS tree T, each vertex v learns its subtree-sketch $\mathsf{Sketch}_G(V(T_v))$. Once this is completed, it is easy to let each vertex $x\in V$ learn the G-sketch information of all the connected components in $T\setminus\{x\}$. We then show that x can locally modify these G-sketches into $(G\setminus\{x\})$ sketches. At this point, the vertex x can locally apply the Borůvka algorithm in $G\setminus\{x\}$ and deduce if $G\setminus\{x\}$ is connected.

We now turn to consider the considerably more challenging task of detecting cut-pairs. We classify these pairs into dependent and independent.

Detecting Dependent Cut Pairs. Our approach for the dependent case is based on designing algorithms $\{\mathcal{A}_y\}_{y\in V}$, where $\mathcal{A}_y$ detects all xy cut pairs of the form $x\in T_y$. We show that each such an algorithm $\mathcal{A}_y$ can be designed in a way that sends a total of $\widetilde{O}(1)$ messages only along edges incident to $V(T_y)$, and runs in $\widetilde{O}(D)$ rounds. The standard random delay technique allows us then to schedule the execution of all n algorithms $\{\mathcal{A}_y\}_{y\in V}$ within $\widetilde{O}(D)$ rounds. At a high level, each algorithm $\mathcal{A}_y$ is based on employing the single-vertex cut algorithm in the graph $G\setminus\{y\}$. Our challenge is then twofold: first, the diameter of the graph $G\setminus\{y\}$ might be as large as $\Omega(\Delta D)$, and second, communication is restricted to use only edges incident to $V(T_y)$. We overcome these challenges by using y as a coordinator, providing global computation services and communication shortcuts that essentially enables efficient simulation (in both dilation and congestion) of the vertex cut algorithm in $G \setminus \{y\}$.

Detecting Independent Cut Pairs. The most technically involved case is where x, y are independent, namely, *do not* have ancestry relations in T. A-priori, the number of such potential cut-mates y for a given vertex x might be even linear in n. To filter out irrelevant options, the algorithm starts by computing at each vertex x a tree $\widehat{T}_x$ that encodes the connectivity between s and the vertices in $V_x = V(T_x) \setminus \{x\}$ in the graph $G \setminus \{x\}$. Let $\mathcal{C}_x = \{C_1, \ldots, C_k\}$ denote the collection of maximal connected components in the graph $G[V_x]$. The tree $\widehat{T}_x$ consists of k paths of the form $\pi(s, u_C) \circ (u_C, v_C)$ for every component $C \in \mathcal{C}_x$, where v_C is some representative vertex in C. It is then easy to observe that the potential cut mates y must appear on the paths $\{\pi(s, u_C) \mid C \in \mathcal{C}_x\}$. For a given suspect y, we call the $\mathcal{C}_x$-components C for which $y \in \pi(s, u_C)$, *y-sensitive*. Our argument has the following structure.

Multiple xy-Connectivity Algorithms, Under a Promise. For a fixed xy pair, we design an algorithm $A^P_{x,y}$ that determines the connectivity in $G \setminus \{x, y\}$ *given* an x-y path $\Pi_{x,y}$ (on which x, y can exchange messages). The algorithm $A^P_{x,y}$ has the special property that it sends messages either along $\Pi_{x,y}$, or else along edges incident to a restricted subset of vertices in T_x, T_y, defined as follows. Let $\mathsf{LDS}(x, y) \subset V(T_x)$ be the set of all vertices which are descendants of the *light* children of x, and belong to a y-sensitive component in $\mathcal{C}_x$. The set $\mathsf{LDS}(y, x)$ is defined in an analogous manner. The algorithm $A^P_{x,y}$ is then guaranteed to send $\widetilde{O}(1)$ messages only along $\Pi_{x,y}$ and along edges incident to the vertices of $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. This restriction is crucial in order to run multiple $A^P_{x,y}$ algorithms, for distinct x, y, in parallel. Using the properties of the heavy-light tree decomposition and our sensitivity definition, one can show that each vertex $w \in V$ belongs to the $\mathsf{LDS}(x, y)$ sets of at most $\widetilde{O}(D)$ pairs xy. The main challenge is in bounding the overlap between the $\Pi_{x,y}$ paths, cross distinct xy pairs. We show that given a subset $Q \in V \times V$, the collection of $\{A^P_{x,y} \mid (x, y) \in Q\}$ algorithms can be scheduled in parallel in $\widetilde{O}(D)$ rounds, given that following promise holds for Q:

[**Promise**:] *There is a path collection $\mathcal{P}_Q = \{\Pi_{x,y} \mid (x, y) \in Q\}$ such that each path has length $O(D)$, and each edge appears on $\widetilde{O}(D)$ paths in $\mathcal{P}_Q$.*

One can show, using the properties of heavy-light decomposition, that each vertex belongs to the $\mathsf{LDS}(x, y)$ sets of at most $\widetilde{O}(D)$ pairs x, y. Hence, by combining this fact with the promise, the algorithms for all the Q pairs can be run in parallel, using the random delay approach [LMR94, Gha15].

On a high level, each algorithm $A^P_{x,y}$ works by letting x and y jointly simulating the Borůvka algorithm in $G \setminus \{x, y\}$. The main challenge is that the communication is restricted to the edges incident to $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$, despite the fact that one should also take into account the remaining vertices in T_x, T_y, e.g., descendants of the *heavy* children of x, y. In each Borůvka phase, we maintain the invariant that x, y jointly hold the sketches of connected-subsets (denoted as *parts*) in $G \setminus \{x, y\}$, where we split the responsibility between x, y in a careful manner. We mainly distinguish between parts that contain a heavy child of x, y and the remaining *light* parts that are contained in $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. The merges of the light parts are implemented by using communication between vertices in $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. The merges concerning the heavy parts are implemented by using the direct xy communication over the $\Pi_{x,y}$ path. Each such Borůvka phase is implemented in $\widetilde{O}(D)$ rounds. At the end of the simulation, x, y both learn whether $G \setminus \{x, y\}$ is connected.

Omitting the Promise Based on Classification Into Light and Heavy Independent Pairs. While the promise clearly holds for $\widetilde{O}(D)$ pairs, it clearly does not hold for all n^2 pairs, in general. Our approach is based on classifying the collection of the xy pairs into two classes: *light* and *heavy*. This classification is based on the trees $\widehat{T}_x, \widehat{T}_y$, as well as, on the heavy-light decomposition of T. Informally, for a light pair xy, one can define a $\Pi_{x,y}$ that intersects a light subtree of either x or y. These paths can be shown to have a bounded overlap, hence satisfying the promise. Handling the heavy pairs is more involved. Here we take a mixed approach. We define a special subset of the heavy pairs for which the promise can be satisfied (denoted as *mutual pairs*). This subset is chosen in a careful way that guarantees the following, perhaps surprising, property: the remaining (not mutual) heavy pairs x, y can be decided locally, at either x or y. Our key observation is that for a xy heavy pair, the graph $G \setminus \{x, y\}$ is connected iff one of the heavy children of x, y is connected to s in $G \setminus \{x, y\}$. Hence, it is mainly essential for x, y to collect a sketch information on the components of these heavy children in $\mathcal{C}_x, \mathcal{C}_y$. This information can be then aggregated over T.

1.2 Preliminaries

Throughout the paper, we fix a connected n-vertex graph $G = (V, E)$, and a BFS tree T for G rooted at some arbitrary source vertex $s \in V$. We denote the unique tree path from u to v by $\pi(u, v, T)$. When the tree T is clear from context, we may omit it and simply write $\pi(u, v)$. We use the $\circ$ operator for path-concatenation. An (undirected) edge between vertices u,v is denoted by (u, v). For $x, y \in V$, a vertex subset $S \subseteq V$ is said to be xy-connected if all the vertices of S belong to the same connected component of $G \setminus \{x, y\}$.

Heavy-Light Tree Decomposition. We now present our heavy-light terminology, the notion of *compressed paths*, and their distributed computation.

Definition 1.1 (Heavy-light decomposition)**.** For a non-leaf vertex $v \in V(T)$, its *heavy child*, denoted v_h, is the child v' of v maximizing[5] the number of vertices in its subtree $T_{v'}$. Any other v-child of v is a *light child*. A tree vertex is *heavy* if it is the heavy child of its parent, and *light* otherwise (so the root s is light). A tree edge is *heavy* if it connects a vertex to its heavy child, and *light* otherwise. If (u, u') is a heavy (resp., light) edge in the path $\pi(s, v)$, then u is a *heavy ancestor* (resp., *light ancestor*) of v, and v is a *heavy descendant* (resp, *light descendant*) of u. (Note that e.g. a 'heavy ancestor' need not be a heavy vertex itself.) We denote by $\mathsf{LA}(v)$ (resp., $\mathsf{LD}(v)$) the set of v's light ancestors (resp., descendants). It is easy to show that $\pi(s, v, T)$ contains $O(\log n)$ *light* vertices/edges, hence also $|\mathsf{LA}(v)| = O(\log n)$.

Definition 1.2 (Compressed paths)**.** Let $v \in V(T)$. Let $L = [s = v_0, v_1, \ldots, v_k]$ be the ordered list of the light vertices on the root-to-v path $\pi(s, v, T)$. The *compressed path* of v with respect to T, denoted $\pi^*(s, v, T)$ consists of the list L, along with a table mapping each v_i to the number of heavy vertices appearing between v_i and v_{i+1} in $\pi(s, v, T)$ (where we define $v_{k+1} = v$). Note that the compressed path $\pi^*(s, v, T)$ has bit-length $O(\log^2 n)$.

Observe that the compressed paths can be used as *ancestry labels* in T: Given the compressed path $\pi^*(s, u, T)$ and $\pi^*(s, v, T)$, one can check whether $\pi(s, u, T)$ is a prefix of $\pi(s, v, T)$, and hence determine whether u is an ancestor of v.

[5]Ties are broken arbitrarily and consistently.

Lemma 1.3. *For every tree T, there is an $\widetilde{O}(D(T))$-rounds $\widetilde{O}(1)$-congestion algorithm letting each vertex v of T learn its heavy/light classification and its compressed path $\pi^*(s,v,T)$.*

Missing proofs in this section are deferred to Appendix A.1.

Graph Sketches. We now give a formal but brief definition of graph sketches. We follow [DP21], and refer the reader to Section 3.2.1 therein for a detailed presentation of the subject. Throughout, let $\oplus$ denote the bitwise-XOR operator. The first required ingredients are randomized *unique edge identifiers*:

Lemma 1.4 (Modification of Lemma 3.8 in [DP21])**.** *Using a random seed $\mathcal{S}_{ID}$ of $O(\log^2 n)$ random bits, one can compute a collection of $M = \binom{n}{2}$ $O(\log n)$-bit identifiers for the pairs in $\binom{V}{2}$, denoted $\mathcal{I} = \{\mathrm{UID}(e_1), \ldots, \mathrm{UID}(e_M)\}$, with the following property: For any nonempty subset $E' \subseteq E$, $\Pr[\oplus_{e\in E'} \mathrm{UID}(e) \in \mathcal{I}] \leq 1/n^{10}$. Furthermore, for any $e = (u,v)$, the identifier $\mathrm{UID}(e)$ can be computed from $\mathrm{ID}(u)$, $\mathrm{ID}(v)$ and the random seed $\mathcal{S}_{ID}$.*

Next, we define the notion of *extended edge identifiers*, formed by augmenting the $\mathrm{UID}(e)$ with the IDs and the T-ancestry labels of the endpoints based on compressed paths, namely $\mathsf{ANC}_T(v) = \pi^*(s,v,T)$. Formally, an edge $e = (u,v)$ we have

$$\mathrm{EID}_T(e) = [\mathrm{UID}(e), \mathrm{ID}(u), \mathrm{ID}(v), \mathsf{ANC}_T(u), \mathsf{ANC}_T(v)] \ . \tag{1.1}$$

Equipped with these definitions, we are ready to define the sketches. We now follow [DP16, DP17, DP21] and use pairwise independent hash functions for this purpose. Choose $L = c\log n$ pairwise independent hash functions $h_1, \ldots, h_L : \{0,1\}^{\Theta(\log n)} \to \{0, \ldots, 2^{\log M} - 1\}$, and for each $i \in \{1, \ldots, L\}$ and $j \in [0, \log M]$ define the edge set $E_{i,j} = \{e \in E \mid h_i(e) \in [0, 2^{\log M - j})\}$. Each of these hash functions can be defined using a random seed of logarithmic length [Vad12]. Thus, a random seed $\mathcal{S}_h$ of length $O(L\log n)$ can be used to determine the collection of all these L functions. For each vertex v and indices i,j, let $E_{i,j}(v)$ be the edges incident to v in $E_{i,j}$. The i^{th} *basic sketch unit* of each vertex v is then given by:

$$\mathsf{Sketch}_{G,i}(v) = [\oplus_{e\in E_{i,0}(v)} \mathrm{EID}_T(e), \ldots, \oplus_{e\in E_{i,\log M}(v)} \mathrm{EID}_T(e)].$$

We extend the sketches to be defined on vertex subsets by XORing. Namely, for every subset of vertices S, we define $\mathsf{Sketch}_{G,i}(S) = \oplus_{v\in S}\mathsf{Sketch}_{G,i}(v)$. The *sketch* of each vertex v is defined by a concatenation of $L = \Theta(\log n)$ basic sketch units:

$$\mathsf{Sketch}_G(v) = [\mathsf{Sketch}_{G,1}(v), \mathsf{Sketch}_{G,2}(v), \ldots \mathsf{Sketch}_{G,L}(v)] \ .$$

Again, we extend this definition to vertex subsets $S \subseteq V$ by $\mathsf{Sketch}_G(S) = \oplus_{v\in S}\mathsf{Sketch}_G(v)$. The main use of graph sketches is in finding outgoing edges:

Lemma 1.5. *[Modification of Lemma 3.11 in [DP21]] For any subset S, given a basic sketch unit $\mathsf{Sketch}_{G,i}(S)$ and the seed $\mathcal{S}_{ID}$ one can compute, with constant probability*[6] $\mathrm{EID}_T(e)$ *for an outgoing edge e from S in G, if such exists.*

Lemma 1.6. *Let $S \subseteq V$, and let $E' \subseteq E$ be a set of outgoing edges from S. Then, given $\mathsf{Sketch}_G(S)$, the random seed $\mathcal{S}_h$, and the extended identifiers $\mathrm{EID}_T(e)$ of all $e \in E'$, one can compute the $\mathsf{Sketch}_{G\setminus E'}(S)$.*

[6]Over the choice of the random seeds $\mathcal{S}_{ID}$ and $\mathcal{S}_h$.

Distributed Scheduling. The congestion of an algorithm $\mathcal{A}$ is defined by the worst-case upper bound on the number of messages exchanged through a given graph edge when simulating $\mathcal{A}$. Throughout, we make an extensive use of the following random delay approach of [LMR94], adapted to the $\mathsf{CONGEST}$ model.

Theorem 1.7 ([Gha15, Theorem 1.3])**.** *Let G be a graph and let $\mathcal{A}_1, \ldots, \mathcal{A}_m$ be m distributed algorithms, each algorithm takes at most d rounds, and where for each edge of G, at most c messages need to go through it, in total* over all *these algorithms. Then, there is a randomized distributed algorithm that w.h.p. runs all the algorithms in $\widetilde{O}(\mathsf{c}+\mathsf{d})$ rounds.*

2 Single Cut Vertices

In this section we describe the distributed algorithm for detecting single vertex cuts of Theorem 1.1. This serves both as a warm-up to our approach in the subsequent sections devoted to dual vertex cuts detection, as well as for a detailed presentation of basic tools used in these next sections. We assume each vertex v is equipped with its heavy/light classification in T and with its ancestry label which is its compressed path, $\mathsf{ANC}_T(v) = \pi^*(s, v, T)$. This can be achieved in $\widetilde{O}(D)$ rounds by Lemma 1.3.

Step 0: Computing Extended Edge IDs. The source s samples a random seed $\mathcal{S}_{ID}$ of $\widetilde{O}(1)$ bits and shares it with all vertices. Then, using Lemma 1.4, each vertex v can then locally compute the unique edge-ID $\mathrm{UID}(e)$ for each of its incident edges. By letting all neighbors in G exchange their ANC_T-labels, each $\mathrm{UID}(e)$ can be concatenated with the required information to create $\mathrm{EID}(e)$.

Step 1: Computing Subtree Sketches. The source s locally samples the random seed $\mathcal{S}_h$ of $\widetilde{O}(1)$ bits and sends it to all the vertices. Along with the extended edge IDs, this provides all the required information for the computation of $\mathsf{Sketch}_G(v)$ locally in each vertex v. By XOR-aggregation of the individual sketches from the leaves of T up to the root s, each vertex v obtains its subtree sketch, given by $\mathsf{Sketch}_G(V(T_v)) = \oplus_{u \in T_v} \mathsf{Sketch}_G(v)$. Next, within $\widetilde{O}(1)$ rounds, each vertex passes its subtree sketch to its parent, so that each vertex now holds the subtree sketch for each of its children. Finally, the source s also broadcasts its subtree sketch, which is $\mathsf{Sketch}_G(V)$, to all the other vertices.

Step 2: Local Connectivity Computation. This step is locally applied at every vertex x, and requires no additional communication. We show that each vertex x, given the received sketch information in Step 1, can locally simulate the Borůvka's algorithm [NMN01] in the graph $G \setminus \{x\}$, and consequently determine if $G \setminus \{x\}$ is connected. Let $x_1, \ldots, x_k$ be the children of x in T. We assume that $x \neq s$; the case $x = s$ is easier and requires only slight modifications. The connected components in $T \setminus \{x\}$ are denoted by $\mathcal{C}_x = \{V(T_{x_j}) \mid j = 1, \ldots, k\} \cup \{V \setminus V(T_x)\}$. By Step 1, x holds the G-sketch of each component in $\mathcal{C}_x$: It has explicitly received $\mathsf{Sketch}_G(V(T_{x_j}))$ from each child x_j. In addition, it can locally infer $\mathsf{Sketch}(V \setminus V(T_x)) = \mathsf{Sketch}(V) \oplus \mathsf{Sketch}(V(T_x))$. To implement Borůvka's algorithm on these components, we first need to update these G-sketches into $(G \setminus \{x\})$-sketches.

2.1: Obtaining Sketch Information in $G \setminus \{x\}$. Recall x knows the random seed $\mathcal{S}_h$ as well as the extended identifiers of its incident edges (from Step 0). For each such edge (x, u), it first uses the ancestry label of u and of its T-children (found in the EID_T's) to determine the component C

of u in $\mathcal{C}_x$. It then cancel this edges from the sketch of the component C using Lemma 1.6. This allows x to obtain $\mathsf{Sketch}_{G\setminus\{x\}}(C)$ for every $C \in \mathcal{C}_x$.

2.2: Simulating Borůvka in $G\setminus\{x\}$. The input to this step is the $(G\setminus\{x\})$-sketch information of the components in $\mathcal{C}_{x,0} = \mathcal{C}_x$. The desired output is determining the connectivity of $G \setminus \{x\}$. The algorithm consists of $O(\log n)$ phases of the Borůvka algorithm, and is very similar to the (centralized) decoding algorithm of [DP21]. Each phase i will be given as input a partitioning $\mathcal{C}_{x,i} = \{C_{i,1}, \dots, C_{i,k_i}\}$ of (not necessarily maximal) connected components in $G \setminus \{x\}$ along with their sketch information $\mathsf{Sketch}_{G\setminus\{x\}}(C_{i,j})$. The output of the phase is a coarser partitioning $\mathcal{C}_{x,i+1}$, along with the sketch information of the new parts. A component $C_{i,j} \in \mathcal{C}_{x,i}$ is said to be *growable* if it has at least one outgoing edge to a vertex in $V\setminus(C_{i,j}\cup\{x\})$. To obtain outgoings edges from the growable components in $\mathcal{C}_{x,i}$, the algorithm uses the i^{th} basic-unit sketch $\mathsf{Sketch}_{G\setminus\{x\},i}(C_{i,j})$ of each $C_{i,j} \in \mathcal{C}_{x,i}$. By Lemma 1.5, from every growable component $C_{i,j} \in \mathcal{C}_{x,i}$, we get one outgoing edge $e = (u, v)$ with constant probability. To find the component $C_{i,j'}$ containing the other endpoint of e (to be merged with $C_{i,j}$), we use the T-ancestry labels found in $\mathrm{EID}_T(e)$. Say this endpoint is v. We determine the component of v in $T \setminus \{x\}$, i.e. the component $C_{0,q}$ containing v in $\mathcal{C}_{x,0}$, by querying the ancestry relation between v and each child of x using $\mathsf{ANC}_T(v)$ and the labels of x's children. Then v belongs to the unique component $C_{i,j'} \in \mathcal{C}_{x,i}$ containing $C_{0,q}$. The sketch information for the next phase $i+1$ is given by XORing over the sketches of the components in $\mathcal{C}_{x,i}$ that got merged into a single component in $\mathcal{C}_{x,i+1}$. Note that it is important to use fresh randomness (i.e., independent sketch information) in each of the Borůvka phases [AGM12, KKM13, DP16]. Since each growable component gets merged with constant probability, the expected number of growable components is reduced by a constant factor in each phase. Thus after $O(\log n)$ phases, the expected number of growable components is at most $1/n^5$, and by Markov's inequality we conclude that w.h.p. there are no growable components. The partitioning at this point corresponds to the maximal connected components in $G \setminus \{x\}$, so its connectivity can be inferred. This concludes the proof of Theorem 1.1.

Finally, we note that by tracking the merges throughout the Borůvka simulation, x can also find a subset $\widetilde{E}$ of the outgoing edges received throughout the simulation such $(T \setminus \{x\}) \cup \widetilde{E}$ is a maximal spanning forest of $G \setminus \{x\}$. This becomes useful in next sections.

3 Dependent Cut Pairs

In this section we present an $\widetilde{O}(D)$-rounds distributed algorithm for detecting *dependent* cut pairs in G, i.e. pairs xy where x is a descendant of y in the BFS tree T rooted at s. Recall that our approach is based on scheduling the execution of algorithms $\{\mathcal{A}_y\}_{y\in V}$, where $\mathcal{A}_y$ detects all cut pairs xy such that $x \in T_y$ (see Section 1.1). By employing the single-vertex cut algorithm from Section 2 as a common preprocessing phase prior to the execution of the $\{\mathcal{A}_y\}_{y\in V}$ algorithms, we may assume that there are no 1-vertex cuts in G. Furthermore, by carefully examining the properties of this algorithm, we may assume that every $v \in V$ holds the following preprocessing information:

- The random seeds $\mathcal{S}_{ID}$ and $\mathcal{S}_h$.
- $\mathrm{EID}_T(e)$ for every edge e incident to v.
- $|V(T_v)|$ and $|V(T_{v'})|$ for every T-child v' of v.

- $\mathsf{Sketch}_G(v)$, $\mathsf{Sketch}_G(V)$, $\mathsf{Sketch}_G(V(T_v))$ and $\mathsf{Sketch}_G(V(T_{v_i}))$ for every T-child v' of v.
- An edge set $\widetilde{E}(v) \subseteq E \setminus E(T)$ such that $\widetilde{T}(v) = (T \setminus \{v\}) \cup \widetilde{E}(v)$ is a spanning tree of $G \setminus \{v\}$. For each $e \in \widetilde{E}(v)$, its extended identifier $\mathrm{EID}_T(e)$ is known.

We next describe the algorithms $\mathcal{A}_y$:

Lemma 3.1. *Assuming all vertices know their preprocessing information, there is an $\widetilde{O}(D)$-rounds $\widetilde{O}(1)$-congestion algorithm $\mathcal{A}_y$ that detects all cut pairs xy where $x \in V(T_y)$. The algorithm $\mathcal{A}_y$ sends messages only on edges incident to $V(T_y)$.*

Step 0: Local Computation of Component Tree for $\widetilde{T}(y)$ in y. Throughout, let $\widetilde{E} = \widetilde{E}(y)$ and $\widetilde{T} = \widetilde{T}(y)$, and denote the T-children of y by $y_1, \ldots, y_k$. This preliminary step is executed by local computation in y. It constructs the *component tree* $\widetilde{CT}$ in which every connected component of $\widetilde{T} \setminus \widetilde{E}$ is contracted into a single node. Note that $\widetilde{T} \setminus \widetilde{E} = T \setminus \{y\}$, namely the nodes in $\widetilde{CT}$ correspond to connected components of $T \setminus \{y\}$. More concretely, for every $i = 1, \ldots, k$ the component $C_i = V(T_{y_i})$ is a node of $\widetilde{CT}$, and (unless $y = s$) there is another node for the component $C_0 = V(T) \setminus V(T_y)$. Each edge (C_i, C_j) in $\widetilde{CT}$ correspond to the unique $\widetilde{E}$-edge incident to both C_i and C_j. Observe that the extended edge identifiers known to y by preprocessing contain the T-ancestry labels of all endpoints of $\widetilde{E}$-edges, as well as those of the y_i's. Using these ancestry labels, y can determine the components incident to each edge $e \in \widetilde{E}$, and therefore construct $\widetilde{CT}$.

For clarity of presentation we assume $y \neq s$; the special case $y = s$ is easier, and requires only slight modifications. We set s as the root of $\widetilde{T}$, and accordingly C_0 is the root of $\widetilde{CT}$. For each $i = 1, \ldots, k$, denote by $e_i = (r_i, p_i)$ the unique edge in $\widetilde{E}$ connecting C_i to its parent in $\widetilde{CT}$, where r_i is the endpoint of e_i inside C_i, and p_i is the endpoint lying in the parent component. See Fig. 1 for an illustration.

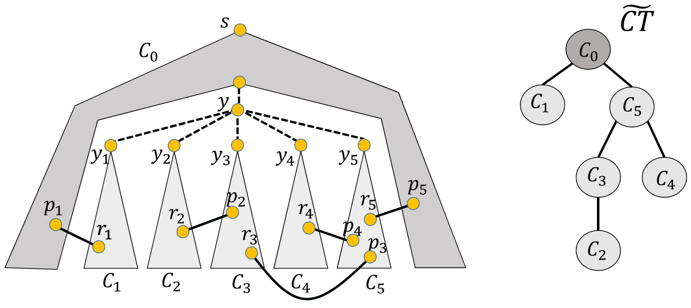

Figure 1: Left: Illustration of the trees T and $\widetilde{T}$. The dashed edges are T-edges adjacent to y, and the solid edges are $\widetilde{E}$-edges. The components $C_0, C_1, \ldots, C_5$ are each internally connected via original T-edges. The tree $\widetilde{T}$ is obtained by removing y and its incident edges from the T and adding the $\widetilde{E}$ edges. Right: The component tree $\widetilde{CT}$.

Step 1: Construction of $\widetilde{T}$. The goal of this step is for each vertex in $V(T_y) \setminus \{y\}$ to learn its parent in $\widetilde{T}$. First, y sends its children their corresponding edges from $\widetilde{E}$, namely each y_i learns

$\text{EID}_T(e_i)$. The y_i's then propagate (in parallel) their received edges down their T-subtrees, so that for all $i = 1, \dots, k$, all the vertices of component C_i know $\text{EID}_T(y_i)$. Then, a BFS procedure initilized in r_i is executed inside each tree T_{y_i} (in parallel). This completes the step, since the $\widetilde{T}$-parent of each vertex in C_i is its BFS-parent from this last procedure, except for r_i whose $\widetilde{T}$-parent is p_i.

Step 2: Computing $\widetilde{T}$-Ancestry labels. In later steps, we will locally simulate Borůvka's algorithm similarly to Section 2, but with the initial components being parts of $\widetilde{T}$. In order to identify which components get merged by the outgoing edges, we will need ancestry labels with respect to the tree $\widetilde{T}$ rather than T. As we are restricted to send messages only on $V(T_y)$-incident edges, we would like the T- and $\widetilde{T}$-labels to coincide for vertices in C_0 (as some of them cannot be informed of new labels). Note that the compressed paths of $v \in C_0$ w.r.t. T and $\widetilde{T}$ are generally different, even though $\pi(s, v, T) = \pi(s, v, \widetilde{T})$, as the these trees have different heavy-light notions. Hence, instead of relying solely on compressed paths in $\widetilde{T}$, we take a hybrid approach and define new labels based on breaking each $\widetilde{T}$-path to a T-part and a strictly $\widetilde{T}$-part, and compressing them accordingly. We still have the challenge of computing (at least part of) the heavy-light decomposition of $\widetilde{T}$. As the diameter of $\widetilde{T}$ might be $\Omega(\Delta D)$, we cannot use simple bottom-up or top-down computations on $\widetilde{T}$. The key for overcoming this is utilizing y as a coordinator, enabling the parts C_i to work in parallel. The full details appear in the proof of the next lemma, found in Appendix A.2.

Claim 3.2. *In $\widetilde{O}(D)$-rounds of computation with $\widetilde{O}(1)$ congestion, in which messages are sent only on $V(T_y)$-incident edges, one can compute $\widetilde{T}$-ancestry labels $\mathsf{ANC}_{\widetilde{T}}(\cdot)$ of $\widetilde{O}(1)$ bits, such that* every *vertex v of $\widetilde{T}$ learns $\mathsf{ANC}_{\widetilde{T}}(v)$.*

Step 3: Computing Sketches w.r.t. $G \setminus \{y\}$ and $\widetilde{T}$. First, we define new extended edge identifiers for the edges of $G \setminus \{y\}$ based on the spanning tree $\widetilde{T}$. Namely, for an edge $e = (u, v)$ of $G \setminus \{y\}$, let

$$\text{EID}_{\widetilde{T}}(e) = [\text{UID}(e), \text{ID}(u), \text{ID}(v), \mathsf{ANC}_{\widetilde{T}}(u), \mathsf{ANC}_{\widetilde{T}}(v)].$$

Now, for every vertex $v \in V \setminus \{y\}$ we define its sketch $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v)$ similarly to $\mathsf{Sketch}_G(v)$, only ignoring edges incident to y in the sampling, and using the $\text{EID}_{\widetilde{T}}$ identifiers for the edges. By this point of the algorithm, computing these new sketches requires $\widetilde{O}(1)$ rounds of communication, in which every $v \in C_1 \cup \cdots \cup C_k$ sends $\mathsf{ANC}_{\widetilde{T}}(v)$ to all its $(G \setminus \{y\})$-neighbors. As the T- and $\widetilde{T}$-ancestry labels coincide on the vertices of C_0, every vertex $v \in V \setminus \{y\}$ can now determine $\text{EID}_{\widetilde{T}}(e)$ for every edge e incident to it in $G \setminus \{y\}$, and use the random seed $\mathcal{S}_h$ to compute $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v)$.

3.1: Computing $\widetilde{T}$-Subtree Sketches. Our next goal is for every $x \in C_1 \cup \cdots \cup C_k$ to learn the $(G \setminus \{y\})$-sketch of its $\widetilde{T}$-subtree (not T-subtree), namely $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V(\widetilde{T}_x)) = \oplus_{v\in\widetilde{T}_x}\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v)$. This is done by using y as a coordinator similarly to the $\widetilde{T}$-subtree sum computation of Step 2.1. We start by bottom-up XOR-aggregation of the sketches on each T_{y_i} (in parallel), which produces the component sketches $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(C_i)$. Next, within $\widetilde{O}(1)$ rounds, the component sketches are all passed to y from its children. Observe that now y can locally compute the $\widetilde{T}$-subtree sketch of each r_i as follows: $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V(\widetilde{T}_{r_i})) = \oplus_{j\in J(i)}\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(C_j)$ where $J(i)$ is the set of all indices j such that C_j is the subtree of C_i in the component tree $\widetilde{CT}$. Then

y sends each of its children y_i the $\widetilde{T}$-subtree sketch of r_i, and this information is then propagated down on each T_{y_i} (in parallel), so that each r_i learns its $\widetilde{T}$-subtree sketch. The r_i's then send their $\widetilde{T}$-subtree sketches to their $\widetilde{T}$-parent, which are the p_i's. For each vertex v of $\widetilde{T}$, let

$$\beta_v = \begin{cases} \text{if } v = p_j\text{:} & \mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v) + \mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V(\widetilde{T}_{r_j})) \\ \text{otherwise:} & \mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(v) \end{cases}$$

Then by this point of the algorithm, every $v \in C_1 \cup \cdots \cup C_k$ know its β_v value. For $i = 1, \ldots, k$, let $\widetilde{T}^{(i)}$ be the tree induced on C_i by $\widetilde{T}$, where the parents in $\widetilde{T}^{(i)}$ are the same as in $\widetilde{T}$. Equivalently, $\widetilde{T}^{(i)}$ is the tree obtained by rerooting T_{y_i} at the vertex r_i. Each of its leaves is either an original $\widetilde{T}$-leaf or a p_j vertex for some j. The crux is that for each $x \in C_i$ it holds that $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(x) = \oplus_{v\in\widetilde{T}^{(i)}_x}\beta_v$. That is, the $\widetilde{T}$-subtree sketch of x is equal to the sum-of-β's in its $\widetilde{T}^{(i)}$-subtree. Hence, we complete the computation in this step by executing bottom-up XOR-aggregation of the β_v values in each of the trees $\widetilde{T}^{(i)}$ in parallel.

3.2: Computing the Sketch of $V \setminus \{y\}$. The last required sketch ingredient for the local simulation of Borůvka in the next step is letting all vertices $x \in C_1 \cup \cdots \cup C_k$ to learn the global sum-of-sketches in $G \setminus \{y\}$, i.e. $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V \setminus \{y\})$. To this end, we carefully examine the contribution of the vertices in C_0 to this sum, as some of them are not $V(T_y)$-adjacent and cannot participate in the computation. This enables us to transform the global sketch $\mathsf{Sketch}_G(V)$ (known from preprocessing) to the desired global sketch in $G \setminus \{y\}$. The details appear in the appendix, in the proof of the following claim:

Claim 3.3. *In $\widetilde{O}(D)$-rounds of computation with $\widetilde{O}(1)$ congestion, in which messages are sent only on $V(T_y)$-incident messages, each vertex $x \in C_1 \cup \cdots \cup C_k$ can learn $\mathsf{Sketch}^{\widetilde{T}}_{G\setminus\{y\}}(V \setminus \{y\})$.*

Step 4: Local Borůvka Simulation In $G \setminus \{x, y\}$. This entire step is executed by local computation in which each $x \in C_1 \cup \cdots \cup C_k$ determines whether it is a cut vertex in $G \setminus \{y\}$, or equivalently if xy is a cut pair in G. This is done by locally simulating Borůvka's algorithms using the sketches of the components of $\widetilde{T} \setminus \{x\}$ (which are known to x by Step 3) in an identical manner to the last step of the (single) cut vertex detection algorithm of Section 2, replacing G and T there with $G \setminus \{y\}$ and $\widetilde{T}$. We note that the new ancestry labels, extended identifiers and sketches, *computed with respect to* $\widetilde{T}$, are important for this simulation to follow through exactly as in Section 2. This completes the proof of Lemma 3.1.

We conclude this section by describing the scheduling of the algorithms $\{A_y\}_{y\in V}$:

Lemma 3.4. *The collection of algorithms $\{\mathcal{A}_y\}_{y\in V}$ can be executed simultaneously within $\widetilde{O}(D)$ rounds, w.h.p.*

Proof. The key observation is that every edge e participates in $O(D)$ algorithms. Specifically, since each algorithm $\mathcal{A}_y$ exchanges messages only on edges incident to $V(T_y)$, we get that the algorithms using $e = (u, v)$ are exactly $\{\mathcal{A}_y \mid y \in \pi(s, u, T) \cup \pi(s, v, T)\}$. Therefore, the total number of messages sent through $e = (u, v)$ in the collection of n algorithms $\{A_y\}_{y\in V}$ is at most $\widetilde{O}(1) \cdot (|\pi(s, u, T)| + |\pi(s, v, T)|) = \widetilde{O}(D)$. The proof follows by employing Theorem 1.7 with congestion and dilation bounds of $\widetilde{O}(D)$. ■

4 Independent Cut Pairs

We now turn to consider the case where the cut pair xy is independent, i.e., x, y have no ancestor-descendant relations. Throughout this section, for every vertex $x \in V$, let $V_x = V(T_x) \setminus \{x\}$. Recall that we assume that there is no single cut vertex in the graph. Our algorithm is based on the introduced notion of *x-connectivity trees*, $\widehat{T}_x$, computed locally at each vertex x. Let $\mathcal{C}_x = \{C_1, \ldots, C_k\}$ denote the maximal connected components in the induced graph $G[V_x]$. For each $C \in \mathcal{C}_x$, the tree $\widehat{T}_x$ contains a path $\pi_x(s, C) = \pi(s, u_C) \circ (u_C, v_C)$, where (u_C, v_C) is a G-edge such that $v_C \in C$, and $x \notin \pi_x(s, C)$. Therefore, $\widehat{T}_x$ encodes the connectivity of s to V_x in the graph $G \setminus \{x\}$. See Fig. 3 for illustrations of these trees. We next describe the computation of these $\widehat{T}_x$ trees, and later on show how they guide the identification of independent cut pairs. Throughout, we assume that the ID of each vertex v contains also its compressed-path information $\pi^*(s, v)$. For every $v \in V_x$, let $C_{x,v}$ denote the component containing v in $\mathcal{C}_x$. When $v = x_h$, we let $H_x = C_{x,x_h}$ and denote it as the *heavy component* of x.

4.1 Computing x-Connectivity Trees

The computation has two main steps, both are based on the bottom-up aggregation of certain graph sketches over the BFS tree T. The purpose of first step is to allow every $x \in V$ to determine the connected components $\mathcal{C}_x$ in $G[V_x]$ where each such component C is identified by the vertex of largest ID among all the T-children of x in C. In addition, in the output of this step each vertex $u \in V_x$ learns the ID of its component $C_{x,u} \in \mathcal{C}_x$. The second step aggregates a special form of graph sketches that provide x with the required path information in order locally compute $\widehat{T}_x$.

Step 1: Computing Connectivity in $G[V_x]$. For ease of notation, let $D = \text{depth}(T)$ and $d_x = \text{depth}(x)$ denote the depth of x in T. We say that an edge $e = (u, v)$ has depth d if $\text{depth}(\text{LCA}(u, v)) = d$. To locally simulate the connectivity Borůvka algorithm in $G[V_x]$ at every x, it is required for x to learn $\mathsf{Sketch}_{G[V_x]}(V(T_w))$ for each T-child w of x. Observe that the edges of $G[V_x]$ can be identified as G-edges in $V_x \times (V \setminus \{x\})$ of depth at least d_x. For this purpose, the algorithm is based on aggregating the information of D graph sketches, for every depth $d \in \{1, \ldots, D\}$. The computation of d^{th} sketch $\mathsf{Sketch}^d_G(\cdot)$ will be restricted to sampling only edges of depth *at least* d.

In Appendix A.3, we show:

Lemma 4.1. *There is a randomized $\widetilde{O}(D)$-round algorithm that w.h.p. computes connectivity in each $G[V_x]$ for every $x \in V$ simultaneously. At the end of the execution, each u holds a component-ID in the graph $G[V_x]$ for every $x \in \pi(s, u)$. Moreover, within additional $\widetilde{O}(D)$ rounds, each u can send its entire component-ID information (for every $x \in \pi(s, u)$) to all its neighbors.*

Step 2: Computing x-Connectivity Trees $\widehat{T}_x$ via Path-Sketches. Our next goal is to provide each vertex x with the path information $\pi_x(s, C)$, for every component $C \in \mathcal{C}_x$. Such a path connects a vertex $v_C \in C$ to the source s in $G \setminus \{x\}$. As we assume that x is not a cut vertex, such a path indeed exists. Towards that goal, we augment the identifier of each edge (u, v) with the tree paths $\pi(s, u), \pi(s, v)$. Formally,

$$\text{EID}^P_T(e) = [\text{UID}(e), \text{ID}(u), \text{ID}(v), \mathsf{ANC}_T(u), \mathsf{ANC}_T(v), \pi(s, u), \pi(s, v)] \ . \quad (4.1)$$

In contrast to the extended-ID of Eq. (1.1) which have $\widetilde{O}(1)$ bits, the latter $\mathrm{EID}^P_T(e)$ identifiers have $\widetilde{O}(D)$ bits. The sketches obtained with these $\mathrm{EID}^P_T(e)$ IDs are called *path-sketches*, denoted as $\mathsf{Sketch}^P_G(S)$ for $S \subseteq V$. The advantage of these path-sketches is that any detected outgoing edge (u,v) obtained from $\mathsf{Sketch}^P_G(Q)$ includes the path information $\pi(s,u)$ and $\pi(s,v)$. Note that the path-sketches $\mathsf{Sketch}^P_G(S)$ have $\widetilde{O}(D)$ bits, since the edge IDs have now $\widetilde{O}(D)$ bits.

Our goal is to let each x learn the path-sketches $\mathsf{Sketch}^P_G(C)$ for each component $C \in \mathcal{C}_x$. Since each path-sketch has $\widetilde{O}(D)$ bits, we cannot allow to compute D sketches for each depth $d \in \{1,\ldots,D\}$. Instead we only aggregate the $\mathsf{Sketch}^P_G(u)$ information in a bottom-up manner on T, which allows every vertex x to learn $\mathsf{Sketch}^P_G(V(T_w))$ for each of its T-children w. By combining with the output of the first step, x can then determine $\mathsf{Sketch}^P_G(C)$ for every $C \in \mathcal{C}_x$. The proof of the following lemma, which appears in the appendix, explains this process in further detail.

Lemma 4.2. *W.h.p., all vertices x can compute the x-connectivity trees $\widehat{T}_x$ within $\widetilde{O}(D)$ randomized rounds.*

For each $x \in V$ and $C \in \mathcal{C}_x$, we define the compressed path of $\pi_x(s,C)$ as $\pi^*_x(s,C) = \pi^*(s,v_C) \circ (v_C,u_C)$ (hence, $\pi^*_x(s,C)$ has $\widetilde{O}(1)$ bits). We conclude the computation regarding the connectivity trees by letting each vertex v learn the compressed-path $\pi^*_x(s,C_{x,v})$ for each of its ancestors $x \in \pi(s,v)$. Since the compressed-path has $\widetilde{O}(1)$ bits, a vertex is required to receive $\widetilde{O}(D)$ bits of information, which can be done in $\widetilde{O}(D)$ rounds:

Lemma 4.3. *There is an $\widetilde{O}(D)$-round algorithm that allows each vertex v to learn the compressed path $\pi^*_x(s,C_{x,v})$ for each $x \in \pi(s,v)$, as well as the entire path $\pi_x(s,C_{x,v})$ for each $x \in \mathsf{LA}(v)$. In addition, each vertex v can share all of this information with neighbors.*

Proof. We let every vertex x send the full path $\pi_x(s,C_{x,x'})$ to each light child x' of x, and the compressed path $\pi^*_x(s,H_x)$ to its heavy child x_h. This information is propagated towards the leaf vertices of T_x. Since each vertex is required to receive $\widetilde{O}(D)$ bits of information from each of its *light* ancestors, as well as $\widetilde{O}(1)$ bits from each of its heavy ancestors, overall it is required to receive $\widetilde{O}(D)$ bits. This can be done in $\widetilde{O}(D)$ rounds, by standard pipeline techniques. Since each v learns $\widetilde{O}(D)$ bits of information, the learned information can be exchanged between every pair of neighbors within $\widetilde{O}(D)$ rounds, as well. ■

4.2 Component Classification Based on Sensitivity

We next use the structure of the x-connectivity tree $\widehat{T}_x$ to classify the xy pairs into several types. We also filter-out possibly many irrelevant xy pairs (for which we deduce immediately that xy is not a cut) using the notion of *sensitivity*.

Definition 4.1 (Sensitivity Notions of $\mathcal{C}_x$ Components)**.** Fix an independent pair x, y. A component $C \in \mathcal{C}_x$ is *y-sensitive* if $y \in \pi_x(s,C)$. The y-sensitive components of $\mathcal{C}_x$ are further classified into two types: *pseudo-sensitive* and *fully-sensitive*, as follows. A component $C \in \mathcal{C}_x$ is *pseudo y-sensitive* if the tree path $\pi_x(s,C)$ contains some edge (y,y') such that $x \notin \pi_y(s,C_{y,y'})$, where $C_{y,y'}$ is the component containing y' in $\mathcal{C}_y$. Finally, a y-sensitive component $C \in \mathcal{C}_x$ is *fully y-sensitive* if C is not pseudo-sensitive.

Hence, in particular a component $C \in \mathcal{C}_x$ is fully y-sensitive if either that last edge of $\pi_x(s,C)$ is incident to y, or that there is an edge $(y,y') \in \pi_x(s,C)$ such that the component $C_{y,y'} \in \mathcal{C}_y$ is

x-sensitive. Note that non-y-sensitive components are clearly connected to s in $G \setminus \{x, y\}$. We later on show that this is true also for pseudo y-sensitive components, therefore their sensitivity to y is superficial. Let $\mathcal{S}(x, y), \mathcal{PS}(x, y), \mathcal{FS}(x, y)$ denote the components in $\mathcal{C}_x$ that are y-sensitive, pseudo y-sensitive and fully y-sensitive, respectively[7]. We next show that each vertex x can determine, for every $C \in \mathcal{C}_x$, certain y vertices for which C is fully y-sensitive by running the procedure described in the following lemma (whose proof is in the appendix). Note that by having the compressed-path $\pi^*_y(s, C_{y,y'})$ and the $\pi^*(s, x)$, it is possible to determine if $x \in \pi_y(s, C_{y,y'})$, hence determining if $C_{y,y'}$ is x-sensitive.

Lemma 4.4. *There is an $\widetilde{O}(D)$-round algorithm that computes the following for every $x \in V$ (in parallel):*

- *$\pi^*_y(s, C_{y,y'})$ for every edge $(y, y') \in \pi_x(s, C)$ and every $C \in \mathcal{C}_x \setminus \{H_x\}$.*
- *$\pi^*_y(s, C_{y,y'})$ for every* light *edge $(y, y') \in \pi_x(s, H_x)$.*

4.3 xy-Connectivity Algorithms Under a Promise

Throughout, we assume that all vertices applied the pre-processing steps of computing the x-connectivity trees $\widehat{T}_x$, as well as, applied the $\widetilde{O}(D)$-round procedures of Lemma 4.3 and 4.4. From this point on, we explain how to determine the connectivity in $G \setminus \{x, y\}$, first for a single pair xy, and then for all pairs that satisfy a given promise.

Recall that $\mathsf{LD}(x)$ is the collection of light descendants of x in T. For a vertex y, let $\mathsf{LDS}(x, y)$ be the collection of light descendants of x that are sensitive to y. Formally, the **l**ight x-**d**escendants y-**s**ensitive vertices are defined by:

$$\mathsf{LDS}(x, y) = \{v \in \mathsf{LD}(x) \mid y \in \pi_x(s, C_{x,v}) \setminus V(T_x)\} \ . \tag{4.2}$$

Observation 4.1. Every vertex v belongs to a total of $O(D \log n)$ sets $\mathsf{LDS}(x, y)$ for $x, y \in V$.

Proof. A vertex $v \in V$ has $O(\log n)$ light ancestors (i.e., belongs to $O(\log n)$ sets of $\mathsf{LD}(x)$). In addition, for each light ancestor $x \in \pi(s, v)$, there are $O(D)$ vertices $y \in \pi_x(s, C_{x,v})$. Therefore, it belongs to $O(D \log n)$ sets as required. ■

Theorem 4.5 (xy-Connectivity Given an x-y Path)**.** *Fix $x, y \in V$ and assume that there is an x-y path $\Pi_{x,y} \subseteq G$ (known in a distributed manner) of length $O(D)$. Then, there is an xy-connectivity algorithm $\mathcal{A}^P_{x,y}$ (i.e., that determines the connectivity in $G \setminus \{x, y\}$) in $\widetilde{O}(D)$ and $\widetilde{O}(1)$-congestion, by sending messages only along on the edges of $\Pi_{x,y}$ or edges incident to $\mathsf{LDS}(x, y) \cup \mathsf{LDS}(y, x)$. At the end of the computation, both x and y know whether $G \setminus \{x, y\}$ is connected or not.*

Before proving Theorem 4.5, we show that given a set of pairs $Q \subseteq V \times V$, then all algorithms $\{\mathcal{A}^P_{x,y} \mid (x, y) \in Q\}$ can be scheduled simultaneously when provided a path collection $\mathcal{P}_Q = \{\Pi_{x,y} \mid (x, y) \in Q\}$ that satisfies the following promise:

[**Promise**:] *$\mathcal{P}_Q$-paths have length $O(D)$, and each edge appears on $\widetilde{O}(D)$ paths in $\mathcal{P}_Q$.*

By a straightforward application of the random delay approach, we show:

[7]Notice that these notations are not symmetric in x, y, e.g. $\mathcal{S}(x, y)$ is different than $\mathcal{S}(y, x)$.

Corollary 4.6. *[All Pairs xy-Connectivity Under a Promise] Let $Q \subseteq V \times V$ be a collection of independent pairs and let $\mathcal{P}_Q = \{P_{x,y} \mid (x,y) \in Q\}$ be a collection of x-y paths that satisfy the promise. Then, the collection of algorithms $\{\mathcal{A}^P_{x,y} \mid (x,y) \in Q\}$, where each $\mathcal{A}_{x,y}$ uses the corresponding path $\Pi_{x,y} \in \mathcal{P}_Q$, can be scheduled simultaneously within $\widetilde{O}(D)$ rounds, w.h.p.*

Proof. We use the standard random delay technique of Theorem 1.7 by showing that the total congestion of all these algorithms is bounded by $\widetilde{O}(D)$. By the properties of algorithm $\mathcal{A}^P_{x,y}$, the algorithm sends $\widetilde{O}(1)$ messages on $\Pi_{x,y} \in Q$, and along each edge incident to the vertices in $\mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$. By Obs. 4.1, each v can belong to the $\mathsf{LDS}(x,y)$ sets of at most $\widetilde{O}(D)$ pairs $x,y \in V$. In addition, each edge appears on $\widetilde{O}(D)$ paths in $\mathcal{P}_Q$. We get that the total edge congestion and dilation of each algorithm $\mathcal{A}^P_{x,y}$ is $\widetilde{O}(D)$ as needed. ■

Description of the Connectivity Algorithm $\mathcal{A}^P_{x,y}$. The algorithm is based on simulating the Borůvka algorithm using the sketch information of connected subsets in $G \setminus \{x,y\}$, held jointly by x and y. Throughout, we refer to the given x-y path $\Pi_{x,y}$ as *the xy channel*. Recall that the algorithm can send only $\widetilde{O}(1)$ bits on that channel. The input for the $i \geq 1$ phase of Borůvka is the following. There is a partitioning $\mathcal{P}_{i-1} = \{P_{i-1,1}, \ldots, P_{i-1,k_{i-1}}\}$ of the vertices in $V \setminus \{x,y\}$ into connected subsets (in $G \setminus \{x,y\}$). We call each $P \in \mathcal{P}_{i-1}$ a *part* (to avoid confusion with the term 'component' reserved for sets in $\mathcal{C}_x$ and $\mathcal{C}_y$). We mark a special vertex in each $P_{i,j} \in \mathcal{P}_i$, called the *leader* of the part. The source vertex s is the leader of its own part (called the s-part), and the leaders of the other parts are some chosen T-children of x or y in these parts. The part-ID is the ID of its leader. The part containing x_h (resp., y_h) is called *x-heavy* (resp., *y-heavy*)[8]. The parts that are free of s, x_h, y_h are called *light*. Hence every light part is contained in $\mathsf{LD}(x) \cup \mathsf{LD}(y)$. A part P is denoted as *growable* if there is an outgoing G-edge connecting P to $V \setminus (P \cup \{x,y\})$. The Borůvka algorithm has $K = O(\log n)$ forest growing phases in $G \setminus \{x,y\}$, each phase reduces the number of growable parts by a constant factor, in expectation. We maintain the following invariant for the beginning of each phase $i \in \{1, \ldots, K\}$:

(I1) x,y know $\mathsf{Sketch}_{G \setminus \{x,y\}}(P)$ of the part $P \in \mathcal{P}_{i-1}$ containing s.

(I2) $z \in \{x,y\}$ knows $\mathsf{Sketch}_{G \setminus \{x,y\}}(P)$ for every *light* part $P \in \mathcal{P}_{i-1}$ whose leader is in T_z.

(I3) x,y know $\mathsf{Sketch}_{G \setminus \{x,y\}}(P)$ as well as the part-IDs of the heavy parts P in $\mathcal{P}_{i-1}$.

(I4) $z \in \{x,y\}$ knows, for each T-child z' of z, the part-ID of the part containing z' in $\mathcal{P}_{i-1}$.

Satisfying the Invariant for the First Borůvka Phase. We start by defining the partitioning $\mathcal{P}_0$ and in particular, focus first on the definition of the part containing s. Recall Def. 4.1 and that $\mathcal{S}(x,y), \mathcal{PS}(x,y), \mathcal{FS}(x,y) \subseteq \mathcal{C}_x$ are the y-sensitive, pseudo y-sensitive and fully y-sensitive components, respectively. Let $\mathsf{NS}(x,y) = \bigcup_{C \in \mathcal{C}_x \setminus \mathcal{FS}(x,y)} C$. The set $\mathsf{NS}(y,x)$ is defined in an analogous manner. Then the s-part in $\mathcal{P}_0$ is given by $U(x,y) = (V \setminus (V(T_x) \cup (T_y))) \cup \mathsf{NS}(x,y) \cup \mathsf{NS}(y,x)$. The next observation exploits the fact that the pseudo y-sensitive components in $\mathcal{C}_x$ and the pseudo x-sensitive components in $\mathcal{C}_y$ are all connected to s in $G \setminus \{x,y\}$.

Observation 4.2. $G[U(x,y)]$ is connected.

[8]A part can be both x-heavy and y-heavy.

Proof. Let $S = V \setminus (V(T_x) \cup (T_y))$, so $s \in S$. Note that $G[S]$ is connected ($T \setminus (T_x \cup T_y)$ is a spanning tree). We fix $v \in \mathsf{NS}(x,y)$, so $C_{x,v} \notin \mathcal{FS}(x,y)$, and show that it is connected to s in $G[U(x,y)]$. First assume that $y \notin \pi_x(s, C_{x,v})$. In such a case, it indeed holds that v is connected to s in $G[S \cup \mathsf{NS}(x,y)]$, as $C_{x,v} \subseteq \mathsf{NS}(x,y)$. Otherwise, $C_{x,v}$ is pseudo y-sensitive, so there is an edge $(y,y') \in \pi_x(s, C_{x,v})$ such that $x \notin \pi_y(s, C_{y,y'})$. Therefore, y' is connected to s in $G[S \cup \mathsf{NS}(y,x)]$, as $C_{y,y'} \subseteq \mathsf{NS}(y,x)$. Since y' and v are also connected in $G[C_{y,y'} \cup C_{x,v}]$, we get that s and v are connected in $G[U(x,y)]$. The same argument holds in a symmetric manner for every $v \in \mathsf{NS}(y,x)$. ■

We partition the responsibilities on the parts in $\mathcal{P}_0$ between x and y, as follows. Let $\mathcal{P}_{0,x} = \mathcal{FS}(x,y)$ be the components in $\mathcal{C}_x$ that are fully-sensitive to y. Similarly, $\mathcal{P}_{0,y} = \mathcal{FS}(y,x)$. The 0^{th} partitioning of $V \setminus \{x,y\}$ is given by $\mathcal{P}_0 = \{U(x,y)\} \cup \mathcal{P}_{0,x} \cup \mathcal{P}_{0,y}$. For every $z \in \{x,y\}$, the leader of each $C \in \mathcal{P}_{0,z}$ is chosen as the vertex of largest ID among all the T-children of x, y in C. The leader of $U(x,y)$ is the root s. To satisfy the invariants for the beginning of phase $i \geq 1$, it is sufficient to show the following claims for x (the proofs, found in Appendix A.3, work in a symmetric manner for y):

Claim 4.7. *Within $\widetilde{O}(D)$ rounds, the vertex x can compute $\mathsf{Sketch}_{G \setminus \{x,y\}}(C)$ for every component $C \in \mathcal{S}(x,y)$. In addition, the vertex y can determine its neighbors in $\{v \in V_x \mid y \notin \pi_x(s, C_{x,v})\}$. The communication is restricted to the edges of $\mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$ and using the xy channel.*

Claim 4.8. *By exchanging $\widetilde{O}(1)$ bits of information (using the promised channel), invariants (I1-I4) hold w.r.t $\mathcal{P}_0$.*

Simulation of the i^{th} Borůvka Phase. We now describe the execution of phase $i \geq 1$ assuming that at the beginning of the phase the invariant holds w.r.t $\mathcal{P}_{i-1}$. The output of the execution will be the partitioning $\mathcal{P}_i$, for which we later show that the invariant holds as well. Our goal is to let x, y simulate a Borůvka phase in which parts of $\mathcal{P}_{i-1}$ are merged along their outgoing edges. The main objective of this phase is to reduce the number of *growable* parts by a constant factor, in expectation. Throughout, we use the following auxiliary claim which allows the vertices in every light part to exchange $\widetilde{O}(1)$ bits, in parallel.

Claim 4.9. *Let P be a light part in $\mathcal{P}_{i-1}$ such that each vertex $v \in P$ holds a $\widetilde{O}(1)$-bit value $val(v)$. Then, there is an $\widetilde{O}(D)$-round algorithm that allows all vertices in P to compute any aggregate function of the $val(v)$ values for $v \in P$, by sending messages* only *along edges incident to P. Consequently, all light parts in $\mathcal{P}_{i-1,x} \cup \mathcal{P}_{i-1,y}$ can compute their respective aggregate functions, in parallel.*

For efficiency of computation, we restrict the merge shapes to be star shapes by using random coins (see e.g., [GH16]). Such star merges are obtained by letting each part $\mathcal{P}_{i-1}$ toss a random coin, and allowing only merges centered on head-parts, each accepting incoming suggested merge-edges from tail-parts. The leader of this head-part becomes the leader of the merged part. We show that under the promise and the $(i-1)^{th}$ invariant, this merging phase can be implemented in $\widetilde{O}(D)$ rounds as follows. W.l.o.g., we make x be responsible for the s-part $P_s \in \mathcal{P}_{i-1}$.

Implementing Merges. Each vertex $z \in \{x,y\}$ tosses a (fresh) random coin for each of its parts in $\mathcal{P}_{i-1,z}$. In addition, x tosses a coin for the s-part P_s. Next, for each of the tail part $P \in \mathcal{P}_{i-1,z}$, z locally computes an outgoing edge for each of its *tail* parts in $\mathcal{P}_{i-1,z}$. In addition, x computes an outgoing edge for the s-part (in case that the coin toss of that part is tail). For each growable part $P \in \mathcal{P}_{i-1,z}$, such an edge can be detected from $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ with constant probability. The parts of $\mathcal{P}_i$ are formed by merging every head part $P^* \in \mathcal{P}_{i-1}$ with all the tail parts in $\mathcal{P}_{i-1}$ whose outgoing edges point at P^*. The leader of the merged part is the leader of the head part P^*. For every tail part $P \in \mathcal{P}_{i-1,x}$, let $e_P = (u_P, v_P)$ be the detected outgoing edge obtained by x from $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$.

Claim 4.10. *Using $\widetilde{O}(D)$ rounds of communication over edges incident to $\mathsf{LDS}(x,y)$ and the given xy channel, z can determine for all its tail parts $P \in \mathcal{P}_{i-1,z}$ with an outgoing edge $e_P = (u_P, v_P)$, the following information: (i) the part-ID of the second endpoint $v_P \notin P$ and, (ii) the coin-toss of the part of v_P.*

Proof. We first let x, y send the part-ID information for all vertices in $\mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$. In addition, for each *light* part $P \in \mathcal{P}_{i-1}$ we propagate the coin flip to the entire part, using Claim 4.9. We now focus on $z = x$ (the proof works in the same manner for y). Let P_s be the s-part in $\mathcal{P}_{i-1}$. We show that x can learn the part-IDs and coin-toss outcomes of the vertices in $Z_x = \{v_P \mid P \in \mathcal{P}_{i-1,x} \cup \{P_s\},\ P \text{ is tail}\}$.

First, we use the xy channel to let y sending x the part-ID and coin-toss of its heavy child y_h. In addition, y also sends, over the channel, the coin tosses of its heavy parts in $\mathcal{P}_{i-1,y}$. Note that for every vertex in Z_x, x can locally determine which of these endpoints are descendants[9] of y_h. Also note that x knows the part-IDs of all the vertices in $Z_x \cap (V \setminus V(T_y))$ (using Invariant (I4)).

Next, x considers (at most three) special tails parts: at most two tail heavy parts in $\mathcal{P}_{i-1,x}$, and the s-part. For each of these parts P, x sends (u_P, v_P) over the xy channel, and y-responds with the part-ID of the u_P, v_P vertices that belong to T_y, as well as, with the coin-tosses of these parts in case, they belong to $\mathcal{P}_{i-1,y}$.

Finally, it remains to consider the light parts. Using Claim 4.9, for every light part P, the edge-ID (u_P, v_P) is broadcast over P. This happens for all light parts in parallel. Consequently, the u_P vertex in each such part P is informed (and can also inform x that $v_P \notin P$). It remains to consider the case where v_P is not a heavy descendant of y. The interesting case is when $v_P \in \mathsf{LD}(y)$.

Since the algorithm applied, in the preprocessing step, the procedure of Lemma 4.3, u_P can determine if C_{y,v_P} is x-sensitive (as u_P holds $\pi^*_y(s, C_{y,v_P})$ and the ID of x). If it is not x-sensitive, then this information can be broadcast on P (using Claim 4.9), and x then concludes that v_P is in the s-part.

If C_{y,v_P} is x-sensitive, then by the first paragraph, v_P knows its part-ID (as $v_P \in \mathsf{LDS}(y,x)$) and it can send it to u_P. If v_P belongs to a light part in $\mathcal{P}_{i-1,y}$, then v_P also knows the coin-toss of that part and can send it to u_P. Finally, if v_P belongs to a heavy part or to part in $\mathcal{P}_{i-1,x}$, then x already knows the coin toss of this part. The information acquired by u_P can be then sent to x using Claim 4.9. Altogether x receives that part-ID of v_P for each outgoing edge (u_P, v_P), and in the case where that part is in $\mathcal{P}_{i-1,y}$, x receives in addition also the coin-toss of that part. This completes the proof. ∎

To implement the merges and satisfy the invariant, it is required for $z \in \{x, y\}$ to learn the updated sketch information of their head parts in $\mathcal{P}_{i-1,z}$. We next explain how y can compute the sketch information of each of its head parts $P^* \in \mathcal{P}_{i-1,y}$. (A similar procedure would work for x).

[9] E.g., as the ancestry labels can be provided as part of the vertex ID.

By Claim 4.10, x knows for every head part $P^* \in \mathcal{P}_{i-1,y}$, the collection of tail parts in $\mathcal{P}_{i-1,x}$ that should be merged with P^*.

Any $\rightarrow$ Non-Light Merges. There are (at most three) non-light parts in $\mathcal{P}_{i-1}$, corresponding to at most two heavy parts and the s-part[10]. For each of these non-light part P^*, x aggregates that sketch information of the corresponding tail parts $P \in \mathcal{P}_{i-1,x}$, and send it to y over the xy channel.

From the point on, x considers the transfer of information concerning the light head parts P^* in $\mathcal{P}_{i-1,y}$.

Non-Light $\rightarrow$ Light Merges. It uses the xy channel to send y the sketch information of its non-light tail parts P, along with the part-ID of their head parts (to which they should be merged).

Light $\rightarrow$ Light Merges. The sketch of all other (light) parts in $\mathcal{P}_{i-1,x}$ are communicated to y over the edges incident to the light sensitive xy descendants, $\mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$, as follows. Using Claim 4.9, each light and tail part $P \in \mathcal{P}_{i-1,x}$ can learn $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ (as x holds this information, by the invariant). Note that by definition $P, P^* \subseteq \mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$. The vertices of P then send this received information to all their neighbors. At this point, for every light head part P^* in $\mathcal{P}_{i-1,y}$, and for every tail *light* part P in $\mathcal{P}_{i-1,x}$, there is a vertex $v_P \in P^*$ that holds $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$. By applying Claim 4.9, all vertices in P^* can learn the sum of all these sketches. This provides y with all the required information from x to compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(P^*)$ for each head part $P^* \in \mathcal{P}_{i-1,y}$. In a symmetric manner, x can compute the sketch of the merged parts for all its head parts in $\mathcal{P}_{i-1,x}$. Using the xy channel, x and y can exchange the part-ID and sketch information of the heavy parts and the s-part in $\mathcal{P}_i$. This satisfies (I1,I2,I3) for the partitioning $\mathcal{P}_i$.

To satisfy (I4), note that the part-ID has changed only for tail parts in $\mathcal{P}_{i-1}$. For the tail-parts in $\mathcal{P}_{i-1,z}$, z holds their new part-ID using Claim 4.10 (i.e., this is the part-ID of the detected outgoing edges). This completes the description for phase i.

We are now ready to complete the proof of Theorem 4.5.

Proof of Theorem 4.5. By the description of the i^{th} phase, the invariant holds w.r.t $\mathcal{P}_i$. We next show that the i^{th} phase sends $\widetilde{O}(1)$ messages along edges incident to $\mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$, as well as over the xy channel. It is also easy to see that given the promised channel that running time is $\widetilde{O}(D)$ using Claim 4.9. Finally, we show that within $k = O(\log n)$ phases it holds that there are no growable components in $\mathcal{P}_k$.

Recall that a part P in $\mathcal{P}_j$ is denoted as *growable* if there is a G-edge $(u,v) \in P \times (V \setminus (\{x,y\} \cup P))$. We claim that the number of growable part reduces by a constant factor in each Borůvka phase. Given a sketch information $\mathsf{Sketch}_{G\setminus\{x,y\}}(P)$ for a growable part P, one can infer an outgoing edge (u,v) from P with constant probability. In addition, with probability $1/4$ this edge is valid (i.e., P is a tail part and v is in a head part). Therefore, overall the number of growable parts reduces by a constant factor, in expectation. By the Markov inequality, w.h.p. there is no growable part after $O(\log n)$ phases. Since x, y jointly hold the sketch information of all parts in $\mathcal{P}_k$ they can determine if there is more than one part in $\mathcal{P}_k$ by exchanging information along their channel (i.e., if $G \setminus \{x,y\}$ is not connected, then w.h.p. either x or y holds a part whose leader is not s). The theorem follows. ■

[10]The latter is held by x, so when revering the roles of x, y, y might be required to send x the sum of sketch information of the tail parts in $\mathcal{P}_{i-1,y}$ that got merged with the s-part.

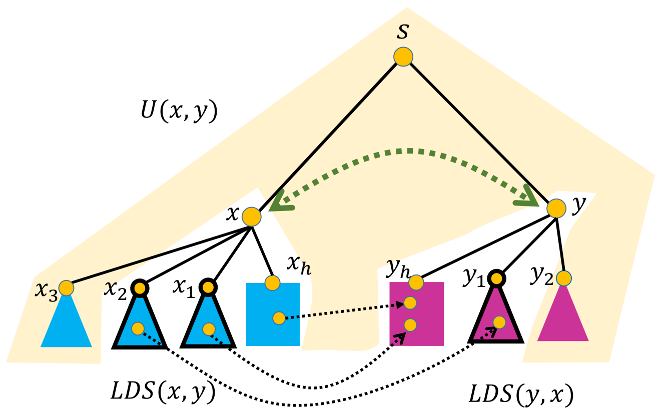

Figure 2: Simulating the first Borůvka phase in algorithm $\mathcal{A}^P_{x,y}$. Each triangle corresponds to a light component in $\mathcal{C}_x, \mathcal{C}_y$. The square boxes correspond to the heavy components H_x, H_y. The framed triangles correspond the subtrees of x, y that belong to the set $\mathsf{LDS}(x,y) \cup \mathsf{LDS}(y,x)$. The dashed green bidirectional arrow represents the xy channel given by the promise. The dashed black arrows correspond to the outgoing edges obtained by x, y from the sketch information of their components. In the example, the light subtrees T_{x_2} and T_{y_1} exchange information over their outgoing edge, which allows y to compute the sketch of the merged component $V(T_{x_2}) \cup V(T_{y_1})$. The sketch of the merged component $V(T_{y_h}) \cup V(T_{x_1}) \cup V(T_{x_h})$ is computed by y by letting x send $\mathsf{Sketch}_{G\setminus\{x,y\}}(V(T_{x_1})) \oplus \mathsf{Sketch}_{G\setminus\{x,y\}}(V(T_{x_h}))$.

Omitting the Promise by Classification to Light and Heavy Pairs. In the next subsections, we omit the promise by partitioning the xy pairs into two classes: light and heavy, defined as follows.

Definition 4.2 (Light and Heavy Independent Pairs)**.** An independent pair x, y is denoted as a *light-pair* if either (i) there is a path $\pi_x(s,C) \subseteq \widehat{T}_x$ such that either the last edge of the path (y, v_C) for $v_C \in C$ or else, there is a light edge $(y, y') \in \pi_x(s,C)$ such that $x \in \pi_y(s, C_{y,y'})$, or (ii) there is a path $\pi_y(s,C) \subseteq \widehat{T}_y$ such that either the last edge of the path (x, v_C) for $v_C \in C$ or else, there is a light edge $(x, x') \in \pi_y(s,C)$ and $y \in \pi_y(s, C_{x,x'})$. The remaining independent pairs are denoted as *heavy*-pairs. See Fig. 3 for an illustration.

The following observation provides a more explicit characterization of the heavy pairs.

Observation 4.3. For every heavy pair xy the following holds: for every fully y-sensitive component $C \in \mathcal{C}_x$ it holds that $(y, y_h) \in \pi_x(s,C)$, and similarly for every fully x-sensitive component $C \in \mathcal{C}_y$ it holds that $(x, x_h) \in \pi_y(s,C)$.

Proof. Consider a fully y-sensitive component $C \in \mathcal{C}_x$, and recall that $\pi_x(s,C) = \pi(s,u_C) \circ (u_C, v_C)$ (see Def. 4.1). Then, since xy is not light, we have that $u_C \neq y$. In addition, as C is fully y-sensitive and $u_C \neq y$, there is an edge $(y, y') \in \pi(s, u_C)$. Since xy is not light, we conclude that $y' = y_h$. The proof works similarly for $C \in \mathcal{C}_y$. ∎

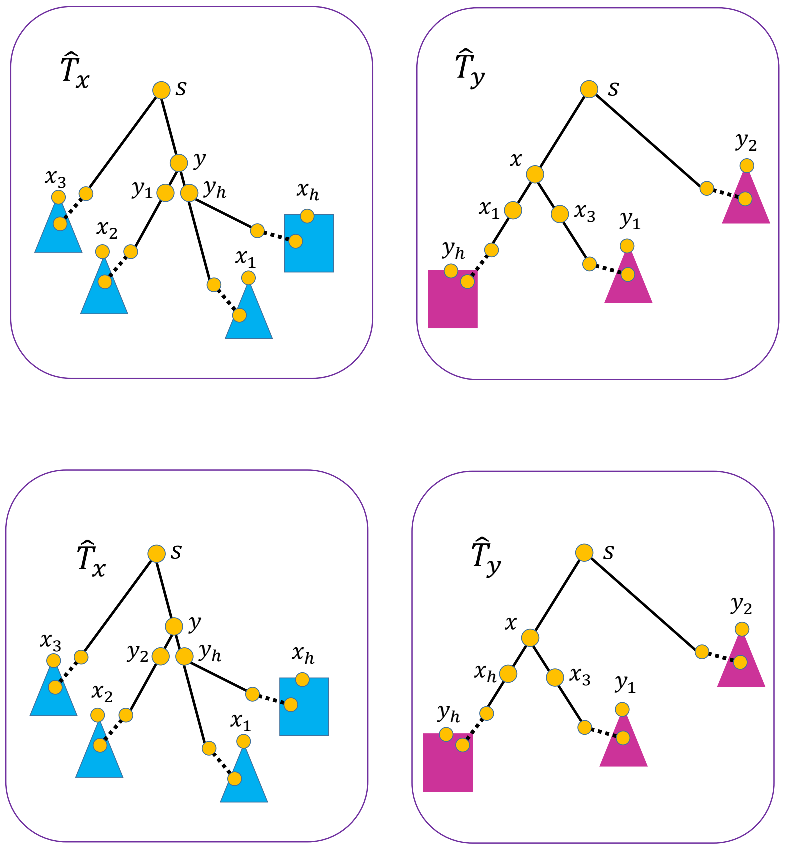

Figure 3: Top: An illustration of a light xy pair. Shown are the trees $\widehat{T}_x, \widehat{T}_y$. The pair xy is light since tree $\widehat{T}_x$ contains a light edge (y, y_1) and $x \in \pi_y(s, C_{y,y_1})$. Bottom: Illustration of a heavy pair xy. For the light child y_2 of y in $\widehat{T}_x$, we have that $x \notin \pi_y(s, C_{y,y_2})$. Similarly, for the light child x_3 of x in $\widehat{T}_y$, we have that $y \notin \pi_x(s, C_{x,x_3})$. Hence, the fully y-sensitive components in $\mathcal{C}_x$ are connected in $\widehat{T}_x$ to the heavy child y_h (and vice-versa).

4.4 Running the Light $\mathcal{A}_{x,y}$ Algorithms in Parallel

For each light pair xy, we describe an algorithm $\mathcal{A}_{x,y}$ which implements a channel of communication between x, y, by defining a path $\Pi_{x,y}$, in a way that replaces the promise of $\mathcal{A}^P_{x,y}$. Algorithm $\mathcal{A}_{x,y}$ works in the exact same manner as $\mathcal{A}^P_{x,y}$, with the only distinction that the messages that all the messages exchanged in $\mathcal{A}^P_{x,y}$ over the promised channel, will be sent instead along the path $\Pi_{x,y}$.

For every (ordered) light pair x, y, assume w.l.o.g. that there is a component $C \in \mathcal{C}_x$ such that either $u_C = y$ (i.e., the last edge of $\pi_x(s, C)$ is (y, v_C) for $v_C \in C$), or that there is a light edge $(y, y') \in \pi_x(s, C)$ such that $x \in \pi_y(s, C_{y,y'})$. Note that there might be multiple such components

$C \in \mathcal{C}_x$ that satisfies the above, and in the following x picks one such C arbitrarily. Then, define[11]:

$$\Pi_{x,y} = \pi(x, v_C) \circ (v_C, u_C) \circ \pi(u_C, y) \ . \tag{4.3}$$

Let Q_{light} be the collection of xy light pairs. We first show that the collection of $\{\Pi_{x,y} \mid (x,y) \in Q_{light}\}$ paths satisfies the promise.

Lemma 4.11. *The collection of paths $\{\Pi_{x,y} \mid (x,y) \in Q_{light}\}$ satisfies the promise: all paths are of length $O(D)$ and each edge appears on $\widetilde{O}(D)$ paths.*

Proof. The length bound on $\Pi_{x,y}$ is immediate, and so we consider the edge congestion. An edge (a,b) appears in the first segment, (i.e., $\pi(x, v_C)$ of Eq. (4.3)) of at most $O(D \log n)$ many paths: For every ancestor $x \in \pi(s,a)$ and every light edge (y,y') on the path $\pi_x(s, C_{x,a})$. In the case where the second segment (i.e., $\pi(u_C, y)$ of Eq. (4.3)) consists of a single vertex, we have that $\Pi_{x,y}$ consists of only edges incident to $V(T_x)$. In the remaining case, (a,b) appears on the second segment (i.e., $\pi(u_C, y)$ of Eq. (4.3)) of $O(D \log n)$ paths: for every light ancestor y on $\pi(s,b,T)$ and every vertex on $x \in \pi_y(s, C_{y,b})$. ■

Finally, we show in Appendix A.3 that we can perform an *handshake* between each such pair, that allows the vertices x,y to distributively define their xy-channel, and thus execute the collection of $A_{x,y}$ algorithms for all light pairs.

Lemma 4.12. *One can schedule the collection of the $\mathcal{A}_{x,y}$ algorithms for all light pairs x,y in $\widetilde{O}(D)$ rounds.*

4.5 Running the Heavy $\mathcal{A}_{x,y}$ Algorithms in Parallel

We now consider the most challenging configuration of *heavy* pairs. Our strategy is based on identifying a limited number of carefully chosen pairs for which we implement the promise using bounded-congestion xy-paths. We then show that in order to handle all remaining pairs, it is sufficient for the vertices to collect a small amount of information over the tree T. Perhaps surprisingly, this information enables x,y to run a local simulation of Borůvka in $G \setminus \{x,y\}$ with no further communication (between the subtrees of x and y). We use the following key notion.

Definition 4.3 (Doubly-Connected Sets)**.** Let $x \in V$ and $C \in \mathcal{C}_x$. Denote by $N_x(C)$ the set of vertices in $V \setminus V(T_x)$ which have a neighbor inside C. We say C is *doubly-connected* to a vertex u, if there exist two distinct vertices $a,b \in N_x(C)$ s.t. $\mathrm{LCA}(a,b) = u$.

Observation 4.4. Fix x and $C \in \mathcal{C}_x$ and let y' be a vertex on $\pi_x(s,C) = \pi(s,u_C) \circ (u_C, v_C)$ such that C is doubly connected to y'. Then, s and all vertices in C are connected in $G \setminus \{x,y\}$ for every vertex $y \neq y'$ in $\pi(y', u_C)$.

Proof. Since C is doubly-connected to y', there are two distinct vertices $a,b \in N_x(C)$ with $\mathrm{LCA}(a,b) = y'$. Therefore, as y is below y' in T, y cannot be a common ancestor of a,b. Without loss of generality, assume that $y \notin \pi(s,a)$. Now let a' be a neighbor of a in C, which exists as $a \in N_x(C)$. Then $\pi(s,a) \circ (a,a')$ is a path connecting s to C in $G \setminus \{x,y\}$. ■

[11] In the below $\Pi_{x,y}$ definition, u_C might be equal y.

For every vertex x, recall that $H_x \in \mathcal{C}_x$ is the component that contains the heavy child of x, namely, x_h. We then focus on the collection of vertices on $\pi_x(s, H_x)$ that are doubly connected to H_x. Let $\mathsf{TD}(x)$ be the topmost vertex on $\pi_x(s, H_x)$ that is doubly connected to H_x. If there are no such vertex, define $\mathsf{TD}(x) = u_{H_x}$. By Observation 4.4, for every vertex y that lies strictly below $\mathsf{TD}(x)$ on $\pi_x(s, H_x)$ it holds that H_x is connected to s in the graph $G \setminus \{x, y\}$. See Fig. 4.

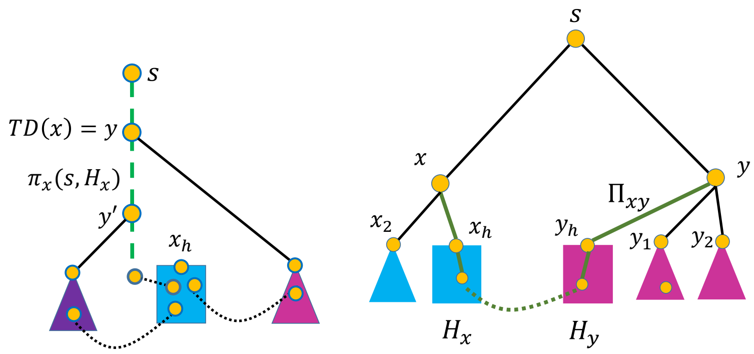

Figure 4: Left: An failure of a vertex y' that lies below $\mathsf{TD}(x)$ on $\pi_x(s, H_x)$ does not disconnect H_x from s. Right: Illustration of the $\Pi_{x,y}$ path for a heavy xy pair.

Claim 4.13. *There is an $\widetilde{O}(D)$-round algorithm that allows each vertex x compute $\mathsf{TD}(x)$.*

Definition 4.4. An ordered heavy pair x, y is *mutual* if $\mathsf{TD}(x) = y$ and $\mathsf{TD}(y) = x$.

To handle the mutual and non-mutual heavy pairs, our approach is based on having preliminary $\widetilde{O}(D)$-round procedure that equipped each vertex x with useful information on each vertex $y \in \widehat{T}_x$. In particular, we apply the following:

Lemma 4.14. *There is an $\widetilde{O}(D)$-round algorithm that allows every vertex x learn the following for each $y \in V(\widehat{T}_x) \setminus V(T_x)$: (i) $\mathsf{TD}(y)$, (ii) $\pi^*_y(s, H_y)$ and (iii) a fresh sketch information $\mathsf{Sketch}'_{G\setminus\{y\}}(H_y)$.*

Handling the Mutual Heavy Pairs. Our goal is to show that one can implement the promised channel for all the mutual pairs $\langle x, \mathsf{TD}(x)\rangle$ using $\widetilde{O}(D)$ rounds. First observe that every x can locally recognize its mutual mate y (if exists). This holds as by Claim 4.13, x knows $y = \mathsf{TD}(x)$, and by Lemma 4.14 it also knows $\mathsf{TD}(y)$. Similarly to the light pairs, we handle the mutual pairs xy by defining a collection of path $\Pi_{x,y}$ as follows. Let $u = u_{H_x}, v = v_{H_x}$, so (u, v) is the last edge of $\pi_x(s, H_x)$. Then $\Pi_{x,y} = \pi(x, v) \circ (v, u) \circ \pi(u, y)$, see Fig. 4. For every mutual pair xy, the algorithm $\mathcal{A}_{x,y}$ is $\mathcal{A}^P_{x,y}$ with the xy-channel being $\Pi_{x,y}$.

Lemma 4.15. *The collection of algorithms $\mathcal{A}_{x,y}$, for all mutual xy pairs, can be run in $\widetilde{O}(D)$ rounds, w.h.p.*

Proof. Let Q_{mutual} be the collection of mutual pairs. We show first that the collection of paths $\mathcal{P}_{mutual} = \{\Pi_{x,y} \mid (x, y) \in Q_{mutual}\}$ satisfies the promise of Cor. 4.6. Clearly, the length of each path in $\mathcal{P}_{mutual}$ is $O(D)$. We now show that each edge appears on $O(D)$ many paths. Every vertex

a can appear only on paths of the form $\Pi_{x,y}$ for every $x \in \pi(s,a)$. Hence, each vertex a appears on $O(D)$ many paths, and consequently, each edge appears on $O(D)$ paths. Since the length of each $\Pi_{x,y}$ path is $O(D)$, using the random delay approach of Theorem 1.7, we can exchange $\widetilde{O}(1)$ bits between all mutual pairs x, y within $\widetilde{O}(D)$ rounds, w.h.p. This establishes the paths $\mathcal{P}_{mutual}$ in a distributed manner. The proof then follows by Cor. 4.6. ■

Handling the Remaining Heavy Pairs. We next show that the connectivity in $G \setminus \{x,y\}$ of the remaining pairs can be now determined locally, w.h.p., by either x or y. We start by noting the following property for every heavy pair x, y:

Observation 4.5. For every heavy pair xy it holds that $G \setminus \{x,y\}$ is connected iff there exists $z_h \in \{x_h, y_h\}$, such that s is connected to z_h in $G \setminus \{x,y\}$.

Proof. It is sufficient to show that if s is connected to z_h in $G \setminus \{x,y\}$, then the following holds: (i) all the vertices in the fully y-sensitive components in $\mathcal{C}_x$ are connected to s in $G \setminus \{x,y\}$, and (ii) all the vertices in the fully x-sensitive components in $\mathcal{C}_y$ are connected to s in $G \setminus \{x,y\}$. We next assume that $z_h = y_h$, but the same proof works in a symmetric manner for the case where s is connected to x_h in $G \setminus \{x,y\}$.

By Obs. 4.3, for every fully y-sensitive component C in $\mathcal{C}_x$, it holds that $(y, y_h) \in \pi_x(s, C)$. Therefore, y_h is connected to every fully y-sensitive component $C \in \mathcal{C}_x$ in $G \setminus \{x,y\}$. Since y_h is connected to s in $G \setminus \{x,y\}$, (i) holds. We now turn to show (ii).

From (i), we know that x_h is connected to s in $G \setminus \{x,y\}$. By Obs. 4.3, for every fully x-sensitive component C in $\mathcal{C}_y$, it holds that $(x, x_h) \in \pi_x(s, C)$. Therefore, x_h is connected to every fully x-sensitive component of $\mathcal{C}_y$ in $G \setminus \{x,y\}$, concluding that (ii) holds. ■

From that point on, we fix a non-mutual heavy pair xy. We break the symmetry between x and y by assuming w.l.o.g. that $\mathsf{TD}(y) \neq x$ (but possibly, $\mathsf{TD}(x) = y$). We show that y can locally determine in this case the connectivity in $G \setminus \{x,y\}$, by distinguishing between the following cases.

Case 1: $\mathsf{TD}(y)$ is not below x on $\pi_y(s, H_y)$. We claim that y can safely deduce that $G \setminus \{x,y\}$ is connected. By Obs. 4.5, it is sufficient to show that y_h is connected to s in $G \setminus \{x,y\}$. This indeed holds by Obs. 4.4.

Case 2: $\mathsf{TD}(y)$ is below x on $\pi_y(s, H_y)$. We first claim that if either $(x, x_h) \notin \pi_y(s, H_y)$ or that $(y, y_h) \notin \pi_x(s, H_x)$, then $G \setminus \{x,y\}$ is connected. Assume that $(x, x_h) \notin \pi_y(s, H_y)$. Since xy is a heavy pair, for every fully x-sensitive component $C \in \mathcal{C}_y$, it holds that $(x, x_h) \in \pi_y(s, C)$. Hence, by the assumption, H_y is not fully x-sensitive component, and therefore y_h is connected to s in $G \setminus \{x,y\}$, and the claim holds by Obs. 4.5.

Assume that $(y, y_h) \notin \pi_x(s, H_x)$. Since xy is a heavy pair, by the assumption, H_x is not fully y-sensitive component, and therefore x_h is connected to s in $G \setminus \{x,y\}$, and the claim holds by Obs. 4.5.

Assume from now on that $(x, x_h) \in \pi_y(s, H_y)$ and that $(y, y_h) \in \pi_x(s, H_x)$. Let $\mathcal{C}_{y,x} = \{C \in \mathcal{C}_y \mid (x, x_h) \in \pi_y(s, C)\}$ be the collection of y-components in the subtree of $\widehat{T}_y$ rooted at x_h. In addition, let $\widehat{C} = \bigcup_{C \in \mathcal{C}_{y,x}} C$. See Fig. 5 for an illustration. We will show that y can locally determine the connectivity in $G \setminus \{x,y\}$ by noting the following:

Observation 4.6. $G \setminus \{x,y\}$ is connected iff $H_x \cup \widehat{C}$ has an outgoing edge to $(V \setminus (H_x \cup \widehat{C} \cup \{x,y\}))$.

Proof. Since x is above $\mathsf{TD}(y)$ on $\pi_y(s, H_y)$, we know that H_y is connected only to H_x among all other components in $\mathcal{C}_x$. Also, we know that for every fully y-sensitive component $C \in \mathcal{C}_x$, it holds that $y_h \in \pi_x(s, C)$. Therefore, we conclude that there is exactly *one* fully y-sensitive component in $\mathcal{C}_x$, namely, H_x. In other words, all components of $\mathcal{C}_x \setminus \{H_x\}$ are connected to s in $G \setminus \{x, y\}$. In addition, we know that all the vertices in $\widehat{C}$ are connected to x_h in $G \setminus \{x, y\}$, and thus also to the heavy component H_x. As xy is a heavy pair, every $C' \in \mathcal{C}_y \setminus \mathcal{C}_{y,x}$ is connected to s in $G \setminus \{x, y\}$. Therefore, any outgoing edge of $H_x \cup \widehat{C}$ must be to a vertex that connected to s in $G \setminus \{x, y\}$. ■

It remains to show that y can compute $\mathsf{Sketch}'_{G\setminus\{x,y\}}(H_x \cup \widehat{C})$. This would conclude the claim as using this sketch information, y can determine w.h.p. if $H_x \cup \widehat{C}$ has an outgoing edges (i.e., using $O(\log n)$ basic-sketch units). We first show that y can compute the sketches $\mathsf{Sketch}'_{G\setminus\{x,y\}}(Q)$ for $Q \in \{H_x, H_y\}$. From the extended-ID of x, y knows $\mathsf{Sketch}'_{G\setminus\{x\}}(H_x)$. In addition, y can determine its edges in H_x and cancel them (using Lemma 1.6). This holds since every vertex v knows its component-ID of $C_{x,v} \in \mathcal{C}_x$ for every $x \in \pi(s, v)$. This total amount of $\widetilde{O}(D)$-bit information can be sent from each v to all the neighbors of v. Therefore, y knows its edges in the x-heavy component H_x. The sketch information of these edges can be omitted from $\mathsf{Sketch}'_{G\setminus\{x\}}(H_x)$ and obtain $\mathsf{Sketch}'_{G\setminus\{x,y\}}(H_x)$.

We now show that y can also compute $\mathsf{Sketch}'_{G\setminus\{x,y\}}(H_y)$. Since H_y is *not* doubly-connected to x, and H_x is connected to H_y, we get that x has *no* neighbors in H_y. Therefore, $\mathsf{Sketch}'_{G\setminus\{x,y\}}(H_y) = \mathsf{Sketch}'_{G\setminus\{y\}}(H_y)$. Finally, we show that y can determine $\mathsf{Sketch}'_{G\setminus\{x,y\}}(C)$ for every light component $C \in \mathcal{C}_{y,x}$. We need the following claim:

Claim 4.16. *All vertices $v \in V$ can compute $\mathsf{Sketch}'_{G\setminus\{x\}}(V(T_v))$, for every $x \in \bigcup_{y\in\mathsf{LA}(v)} \pi_y(s, C_{y,v})$, within $\widetilde{O}(D)$ rounds.*

By Claim 4.16, every light child $y' \in C$ of y knows $\mathsf{Sketch}'_{G\setminus\{x\}}(V(T_{y'}))$. Therefore, by sending this information to y, y can compute $\mathsf{Sketch}'_{G\setminus\{x\}}(C)$ (as C is a union of $T_{y'}$ subtrees). In addition, y can locally cancel-out its incident edges in C, resulting in $\mathsf{Sketch}'_{G\setminus\{x,y\}}(C)$ (using Lemma 1.6). We have that y knows $\mathsf{Sketch}'_{G\setminus\{x,y\}}(H_x \cup \widehat{C})$ and can therefore determine connectivity w.h.p.

This concludes the $\widetilde{O}(D)$-round algorithm for detecting independent cut pairs. By Sec. 3, within another $\widetilde{O}(D)$ rounds we can also detect all dependent cut pairs. Theorem 1.2 follows.

References

[AGM12] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms*, pages 459–467. SIAM, 2012.

[BT89] Giuseppe Di Battista and Roberto Tamassia. Incremental planarity testing (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 436–441. IEEE Computer Society, 1989.

[BT96] Giuseppe Di Battista and Roberto Tamassia. On-line maintenance of triconnected components with spqr-trees. *Algorithmica*, 15(4):302–318, 1996.

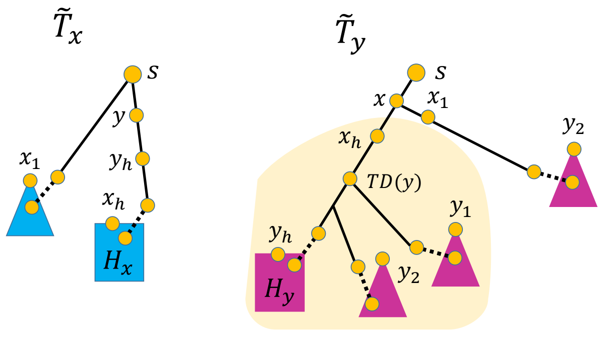

Figure 5: Illustration for Obs. 4.6 where x appears above $\mathsf{TD}(y)$ on $\pi_y(s, H_y)$. Since xy is a heavy pair and as H_y is *not* doubly-connected to x, it must hold that H_x is the only component in $\mathcal{C}_x$ connected to x_h (see $\widehat{T}_x$). Consequently, any outgoing edge of the component $H_x \cup \widehat{C}$ must connect to a vertex that is xy-connected to s.

[CHGK14] Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 156–165. ACM, 2014.

[Cho16] Keerti Choudhary. An optimal dual fault tolerant reachability oracle. In Ioannis Chatzigiannakis, Michael Mitzenmacher, Yuval Rabani, and Davide Sangiorgi, editors, *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, volume 55 of *LIPIcs*, pages 130:1–130:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[CKL$^+$22] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. *CoRR*, abs/2203.00671, 2022.

[DEMN21] Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Distributed weighted min-cut in nearly-optimal time. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1144–1153. ACM, 2021.

[DP09] Ran Duan and Seth Pettie. Dual-failure distance and connectivity oracles. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 506–515. SIAM, 2009.

[DP16] Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. *CoRR*, abs/1607.06865, 2016.

[DP17] Ran Duan and Seth Pettie. Connectivity oracles for graphs subject to vertex failures. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16-19*, pages 490–509, 2017.

[DP21] Michal Dory and Merav Parter. Fault-tolerant labeling and compact routing schemes. In Avery Miller, Keren Censor-Hillel, and Janne H. Korhonen, editors, *PODC '21: ACM Symposium on Principles of Distributed Computing, Virtual Event, Italy, July 26-30, 2021*, pages 445–455. ACM, 2021.

[GH16] Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, MST, and min-cut. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 202–219. SIAM, 2016.

[Gha15] Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 3–12, 2015.

[GILP15] Loukas Georgiadis, Giuseppe F. Italiano, Luigi Laura, and Nikos Parotsidis. 2-vertex connectivity in directed graphs. In Magnús M. Halldórsson, Kazuo Iwama, Naoki Kobayashi, and Bettina Speckmann, editors, *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part I*, volume 9134 of *Lecture Notes in Computer Science*, pages 605–616. Springer, 2015.

[GK17] Manoj Gupta and Shahbaz Khan. Multiple source dual fault tolerant BFS trees. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 127:1–127:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.

[GKKT15] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

[GMW20] Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in o(m log^2 n) time. In Artur Czumaj, Anuj Dawar, and Emanuela Merelli, editors, *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020, July 8-11, 2020, Saarbrücken, Germany (Virtual Conference)*, volume 168 of *LIPIcs*, pages 57:1–57:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[GNT20] Mohsen Ghaffari, Krzysztof Nowicki, and Mikkel Thorup. Faster algorithms for edge connectivity via random 2-out contractions. In Shuchi Chawla, editor, *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms, SODA 2020, Salt Lake City, UT, USA, January 5-8, 2020*, pages 1260–1279. SIAM, 2020.

[GP16] Mohsen Ghaffari and Merav Parter. MST in log-star rounds of congested clique. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing, PODC 2016, Chicago, IL, USA, July 25-28, 2016*, pages 19–28, 2016.

[HLW21] Zhiyang He, Jason Li, and Magnus Wahlström. Near-linear-time, optimal vertex cut sparsifiers in directed acyclic graphs. In Petra Mutzel, Rasmus Pagh, and Grzegorz Herman, editors, *29th Annual European Symposium on Algorithms, ESA 2021, September*

6-8, 2021, Lisbon, Portugal (Virtual Conference), volume 204 of *LIPIcs*, pages 52:1–52:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[HT73] John E. Hopcroft and Robert Endre Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.

[Kar99] David R Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.

[KKM13] Bruce M Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1131–1142. SIAM, 2013.

[KKT15] Valerie King, Shay Kutten, and Mikkel Thorup. Construction and impromptu repair of an MST in a distributed network with o(m) communication. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC 2015, Donostia-San Sebastián, Spain, July 21 - 23, 2015*, pages 71–80, 2015.

[KP21] Karthik C. S. and Merav Parter. Deterministic replacement path covering. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 704–723. SIAM, 2021.

[KW14] Michael Kapralov and David Woodruff. Spanners and sparsifiers in dynamic streams. In *Proceedings of the 2014 ACM symposium on Principles of distributed computing*, pages 272–281, 2014.

[LMR94] Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

[LNP$^+$21] Jason Li, Danupon Nanongkai, Debmalya Panigrahi, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Vertex connectivity in poly-logarithmic max-flows. In Samir Khuller and Virginia Vassilevska Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 317–329. ACM, 2021.

[MK18] Ali Mashreghi and Valerie King. Broadcast and minimum spanning tree with o(m) messages in the asynchronous CONGEST model. In *32nd International Symposium on Distributed Computing, DISC 2018, New Orleans, LA, USA, October 15-19, 2018*, pages 37:1–37:17, 2018.

[NMN01] Jaroslav Nešetřil, Eva Milková, and Helena Nešetřilová. Otakar Borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1):3–36, 2001.

[NSY19] Danupon Nanongkai, Thatchaphol Saranurak, and Sorrachai Yingchareonthawornchai. Breaking quadratic time for small vertex connectivity and an approximation scheme. In Moses Charikar and Edith Cohen, editors, *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019, Phoenix, AZ, USA, June 23-26, 2019*, pages 241–252. ACM, 2019.

[Par15] Merav Parter. Dual failure resilient BFS structure. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*, pages 481–490, 2015.

[Par19] Merav Parter. Small cuts and connectivity certificates: A fault tolerant approach. In *33rd International Symposium on Distributed Computing*, 2019.

[Par20] Merav Parter. Distributed constructions of dual-failure fault-tolerant distance preservers. In Hagit Attiya, editor, *34th International Symposium on Distributed Computing, DISC 2020, October 12-16, 2020, Virtual Conference*, volume 179 of *LIPIcs*, pages 21:1–21:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach.* SIAM, 2000.

[PT11] David Pritchard and Ramakrishna Thurimella. Fast computation of small cuts via cycle space sampling. *ACM Transactions on Algorithms (TALG)*, 7(4):46, 2011.

[PY21] Seth Pettie and Longhui Yin. The structure of minimum vertex cuts. In Nikhil Bansal, Emanuela Merelli, and James Worrell, editors, *48th International Colloquium on Automata, Languages, and Programming, ICALP 2021, July 12-16, 2021, Glasgow, Scotland (Virtual Conference)*, volume 198 of *LIPIcs*, pages 105:1–105:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[ST83] Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[Thu97] Ramakrishna Thurimella. Sub-linear distributed algorithms for sparse certificates and biconnected components. *Journal of Algorithms*, 23(1):160–179, 1997.

[Vad12] Salil P. Vadhan. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1–3):1–336, 2012.

[WY13] Oren Weimann and Raphael Yuster. Replacement paths and distance sensitivity oracles via fast matrix multiplication. *ACM Transactions on Algorithms (TALG)*, 9(2):14, 2013.

A Missing Proofs

A.1 Missing Proofs for Section 1.2

Proof of Lemma 1.3. First, each vertex v learns its subtree size $|V(T_v)|$ by bottom-up aggregation on T. By passing these sizes to the parents, within another round each vertex can classify its children as heavy/light. Within one more round, each vertex is informed on its classification by its parent. Computing the compressed paths can now be executed in a top-down fashion, as a vertex can deduce its compressed path from the compressed path of its father and its own heavy/light classification. ∎

Proof of Lemma 1.6. Let $Out(S)$ be the set of outgoing edges from S in G, and $Out_{i,j}(S) = Out(S) \cap E_{i,j}$. Observe that for each $i \in \{1, \dots, L\}$:

$$\begin{aligned}\mathsf{Sketch}_{G,i}(S) &= [\oplus_{v\in S, e\in E_{i,0}(v)} \mathrm{EID}_T(e), \dots, \oplus_{v\in S, e\in E_{i,\log M}(v)} \mathrm{EID}_T(e)] \\ &= [\oplus_{e\in Out_{i,0}(S)} \mathrm{EID}_T(e), \dots, \oplus_{e\in Out_{i,\log M}(S)} \mathrm{EID}_T(e)]\end{aligned}$$

where the last equality is true as each edge e with both endpoints inside S appears either 0 or 2 times in each XOR. Now let $Out'(S)$ and $Out'_{i,j}(S)$ be defined exactly as $Out(S)$ and $Out_{i,j}(S)$, but with respect to $G \setminus E'$ instead of G. Then as $Out(S)$ is the disjoint union of $Out'(S)$ and E', we obtain

$$\mathsf{Sketch}_{G,i}(S) \oplus \mathsf{Sketch}_{G\setminus E',i}(S) = [\oplus_{e\in E'\cap E_{i,0}} \mathrm{EID}_T(e), \dots, \oplus_{e\in E'\cap E_{i,\log M}} \mathrm{EID}_T(e)].$$

The right-hand side of the above equation can be computed from the given extended IDs of E' and the random seed $\mathcal{S}_h$, and by XORing it with $\mathsf{Sketch}_{G,i}(S)$ we obtain $\mathsf{Sketch}_{G\setminus E',i}(S)$. Concatenating these basic sketch units yields the required. ■

A.2 Missing Proofs for Section 3

Proof of Claim 3.2. We define the labels $\mathsf{ANC}_{\widetilde{T}}$ as follows. If $v \in C_0$, we simply take its T-ancestry label, i.e. $\mathsf{ANC}_{\widetilde{T}}(v) = \mathsf{ANC}_T(v) = \pi^*(s, v, T)$. We now define $\mathsf{ANC}_{\widetilde{T}}(v)$ for the case $v \in C_i$, $i \neq 0$. Observe that the s-v path in $\widetilde{T}$ decomposes as $\pi(s, v, \widetilde{T}) = \pi(s, p_j, T) \circ (p_j, r_j) \circ \pi(r_j, v, \widetilde{T}_{r_j})$ for the unique $p_j \in C_0$ which is an ancestor of v in $\widetilde{T}$. The $\widetilde{T}$-ancestry label of v is obtained by heavy-light compression of the paths as $\mathsf{ANC}_{\widetilde{T}}(v) = \pi^*(s, p_j, T) \circ (p_j, r_j) \circ \pi^*(r_j, v, \widetilde{T}_{r_j})$. Given the $\mathsf{ANC}_{\widetilde{T}}$-labels of any $u, v \in V(\widetilde{T})$, one can easily determine if $\pi(s, u, \widetilde{T})$ is a prefix of $\pi(s, v, \widetilde{T})$, and thus if u is an $\widetilde{T}$-ancestor of v. As compressed paths require $O(\log^2 n)$ bits, each $\mathsf{ANC}_{\widetilde{T}}$-label consists of $O(\log^2 n) = \widetilde{O}(1)$ bits.

We now present the computation of these labels. The vertices of C_0 already hold their labels, as they are equal to their ANC_T labels, so it remains to compute them for the $C_1 \cup \cdots \cup C_k$ vertices. The algorithm proceeds by the following 3 steps.

Step 1: Heavy-Light Decomposition of $\widetilde{T}$. Our first task is for every vertex in $C_1 \cup \cdots \cup C_k$ to learn its heavy/light classification in the tree $\widetilde{T}$, which essentially involves computing subtree sizes in $\widetilde{T}$. At first glance, one might consider computing these by simple aggregation on $\widetilde{T}$. This approach fails as it requires $\Omega(D(\widetilde{T}))$ rounds, and the diameter of $\widetilde{T}$ might be $\Omega(\Delta \cdot D)$. To overcome this, we use y as a coordinator to jump-start the aggregation. Recall that all component sizes $|C_i|$ are known to y by the preprocessing. Therefore, y can locally compute the $\widetilde{T}$-subtree sizes of all the r_i's: $|V(\widetilde{T}_{r_i})|$ is the sum-of-sizes of components lying in the subtree of C_i in the component tree $\widetilde{CT}$. Then, y sends each child y_i the $\widetilde{T}$-subtree size of r_i, and this information is propagated down on each T_{y_i} (in parallel), so that each r_i learns its $\widetilde{T}$-subtree size. Upon receiving this information, each r_i passes it also to its $\widetilde{T}$-parent p_i. Next, for any vertex v of $\widetilde{T}$, denote

$$\alpha_v = \begin{cases} \text{if } v = p_i\text{:} & |V(\widetilde{T}_{r_i})| + 1, \\ \text{otherwise:} & 1. \end{cases}$$

Then by this point, every vertex $v \in C_1 \cup \cdots \cup C_k$ knows its corresponding value α_v. For $i = 1, \dots, k$, let $\widetilde{T}^{(i)}$ be the tree induced on C_i by $\widetilde{T}$, where the parents in $\widetilde{T}^{(i)}$ are the same as in $\widetilde{T}$. Equivalently,

$\widetilde{T}^{(i)}$ is the tree obtained by rerooting T_{y_i} at the vertex r_i. Each of its leaves is either an original $\widetilde{T}$-leaf or a p_j vertex for some j. The crux is that for each $v \in C_i$ it holds that $|V(\widetilde{T}_v)| = \sum_{u \in \widetilde{T}^{(i)}_v} \alpha_u$. That is, the $\widetilde{T}$-subtree size of v is equal to the sum-of-α's in its $\widetilde{T}^{(i)}$-subtree. By executing bottom-up sum-aggregation of the α_v's in each of the trees $\widetilde{T}^{(i)}$ in parallel, each vertex $v \in C_1 \cup \cdots \cup C_k$ learns its $\widetilde{T}$-subtree size. In another communication round, each such vertex passes its $\widetilde{T}$-subtree size to its parent, enabling each vertex to classify each of its children into light or heavy. Within another round, all vertices in $C_1 \cup \cdots \cup C_k$ are informed of their heavy-light classification by their parent.

Step 2: Computing Compressed $\widetilde{T}$-Paths Inside the C_i's. In this substep, each $v \in C_i$, $i \neq 0$, learns the compressed path $\pi^*(r_i, v, \widetilde{T}_{r_i})$. The main observation is that if a vertex is given the compressed path of its parent, it can easily deduce its own compressed path (as it know its own heavy/light classification). Therefore, the required compressed paths can be computed in a top-down fashion on each $\widetilde{T}^{(i)}$ (in parallel).

Step 3: Obtaining The $\mathsf{ANC}_{\widetilde{T}}$-labels. For $j = 1, \ldots, k$ define

$$\pi_j = \begin{cases} \text{if } p_j \in C_i \text{ with } i \neq 0: & \pi^*(r_i, p_j, \widetilde{T}_{r_i}), \\ \text{if } p_j \in C_0: & \pi^*(s, p_j, T) = \mathsf{ANC}_T(p_j). \end{cases}$$

Observe that by Step 2.2, the p_j's know their corresponding π_j's. To send this information to y, each p_j sends π_j to r_j, and the messages are then forwarded upwards on each T_{y_j} (in parallel), along with the heavy/light classification of the r_j's. Using the information of $\{\pi_j\}_j$, the component tree $\widetilde{CT}$ and the heavy/light classifications of the r_j's, y can locally compute $\mathsf{ANC}_{\widetilde{T}}(r_i)$ for each r_i, and send this label to the corresponding child y_i. Then, $\mathsf{ANC}_{\widetilde{T}}(r_i)$ is broadcasted on each T_{y_i} (in parallel), so that each vertex $v \in C_i$ learns this label. Finally, $\mathsf{ANC}_{\widetilde{T}}(v)$ can be locally deduced in v from the information in $\mathsf{ANC}_{\widetilde{T}}(r_i)$ and $\pi^*(v, \widetilde{T}_{r_i})$, where the latter is known to v by Step 2.2. ■

Proof of Claim 3.3. We focus on letting y learn $\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(V \setminus \{y\})$, as it can then broadcast it down T_y. First, by bottom-up aggregation of the sketch information on T_y, y can learn $\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(C_1 \cup \cdots \cup C_k)$. So, it remain to let y learn $\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(C_0)$, as the XOR of these two sketches yields the required. To this end, we partition C_0 to interior and boundry. The interior C_0° consists of all C_0-vertices with all G-neighbors inside C_0. The boundary ∂C_0 consists of the remaining C_0-vertices, namely those that have some G-neighbor inside T_y. Each boundary vertex $v \in \partial C_0$ chooses an arbitrary T_y-neighbor and sends to it its old sketch $\mathsf{Sketch}_G(v)$ and its new sketch $\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(v)$. Then, by bottom-up aggregation of these (old and new) sketches on T_y, y learns $\mathsf{Sketch}_G(\partial C_0)$ and $\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(\partial C_0)$. We assert that by this point, y has all the required information to complete the computation. First, observe that

$$\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(V \setminus \{y\}) = \mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(C_1 \cup \cdots \cup C_k) \oplus \mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(\partial C_0) \oplus \mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(C_0^\circ).$$

The first two terms in the expression are already known to y. We now show how to deduce the last term $\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(C_0^\circ)$. Observe that by our definitions of the $\widetilde{T}$-ancestry labels and the $\mathrm{EID}_{\widetilde{T}}$-identifiers, the old and new sketches of vertices in C_0° coincide. That is, for all $v \in C_0^\circ$ we have $\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(v) = \mathsf{Sketch}_G(v)$. We therefore obtain that

$$\mathsf{Sketch}^{\widetilde{T}}_{G \setminus \{y\}}(C_0^\circ) = \mathsf{Sketch}_G(C_0^\circ) = \mathsf{Sketch}_G(V) \oplus \mathsf{Sketch}_G(V(T_y)) \oplus \mathsf{Sketch}_G(\partial C_0).$$

The first two terms in the latter expression are known to y by the prepossessing, and the last has been previously computed in this step. ■

A.3 Missing Proofs for Section 4

Proof of Lemma 4.1. This is implemented as follows. The source s locally samples a random seed, and broadcast this information to all the vertices. For every $d \in \{1, \ldots, d_u\}$, a vertex u computes its d-depth sketch $\mathsf{Sketch}_G^d(u)$ by restricting the edge sampling only to edges of depth $\geq d$:

$$E_d(u) = \{(u, v) \in E \ \mid \ \text{depth}(\text{LCA}(u, v)) \geq d\} \ .$$

This edge set $E_d(u)$ can be computed locally by u, by letting each vertex learn its tree path $\pi(s, u)$ and exchanging this path information with its neighbors. Using the seed information, u computes d_u sketches $\mathsf{Sketch}_G^1(u), \ldots, \mathsf{Sketch}_G^{d_u}(u)$, where the edges in the d'th sketch $\mathsf{Sketch}_G^d(u)$ are based on using the seed to implement the sampling of the edges in $E_d(u)$.

Next, the algorithm aggregates these D sketches. This can be done in a pipeline manner from the leaf vertices up to the root, in increasing ordering of the depth d of the sketches. At the end of this computation, each vertex x of depth d_x holds the the sketch $\mathsf{Sketch}_G^{d_x}(V(T_w))$ for each of its T-child w. The final sketch $\mathsf{Sketch}_{G[V_x]}(V(T_w))$ is obtained locally at x by canceling-out the sampled x's edges from $\mathsf{Sketch}^{d_x}(V(T_w))$.

At this point, x has all the required sketch information to locally simulate the Borůvka algorithm in $G[V_x]$. As a result of this computation, x holds a component-ID of $C_{x,w} \in \mathcal{C}_x$ for each T-child w of x. This information is then propagated down the tree T in a pipeline manner, where each vertex u eventually learns the component-ID of $C_{x,u}$ for each of its ancestors $x \in \pi(s, u)$. It is easy to see that this entire computation takes $\widetilde{O}(D)$ rounds. ■

Proof of Lemma 4.2. The path-sketches $\mathsf{Sketch}_G^P(V(T_x))$ for every vertex x are computed by a standard aggregation of D-length vectors, which can be done in $\widetilde{O}(D)$ rounds via standard pipeline. At the end of this computation, each vertex x holds that path-sketch $\mathsf{Sketch}_G^P(V(T_w))$ for each of its T-children w. Again, x can locally cancel-out[12] its edges to obtain $\mathsf{Sketch}_{G\setminus\{x\}}^P(V(T_w))$. Then, by combining with connectivity information obtained in Step 1, it can locally computes the path-sketch $\mathsf{Sketch}_{G\setminus\{x\}}^P(C)$ for every $C \in \mathcal{C}_x$. Specifically, letting $N(x, C)$ be the children of x in the component C, then

$$\mathsf{Sketch}_{G\setminus\{x\}}^P(C) = \oplus_{w \in N(x,C)} \mathsf{Sketch}_{G\setminus\{x\}}^P(V(T_w)) \ .$$

Note that any outgoing edge (u_C, v_C) of a component $C \in \mathcal{C}_x$ connects $u_C \in C$ to $v_C \in V \setminus V(T_x)$. In addition, a detected outgoing edge $e = (u_C, v_C)$ from $\mathsf{Sketch}_{G\setminus\{x\}}^P(C)$ includes as part of $\text{EID}_T(e)$ the path edges $\pi(s, u_C) \cup \pi(s, v_C)$. This allows x to compute, w.h.p., the path $\pi_x(s, C) = \pi(s, v_C) \cup (v_C, u_C)$ for each C. The final tree is given by $\widehat{T}_x = \bigcup_{C \in \mathcal{C}_x} \pi_x(s, C)$. The running time is dominated by the time to compute the path-sketches. ■

Proof of Lemma 4.4. First, we let each vertex v learn the compressed-path $\pi_x^*(s, C_{y,y'})$ for every edge $(y, y') \in \pi(s, v)$. This information can be downcasted on T within $\widetilde{O}(D)$ rounds (in the same manner that u learns its tree path edges $\pi(s, v)$. This total amount of $\widetilde{O}(D)$-bits held by each v can be then sent to all v's neighbors, using $\widetilde{O}(D)$.

[12]This holds as x knows the seed of the sketch as well as the $EID_T(e)$ (see Eq. (4.1)) of its incident edges e.

Consider (i) where it is required for each x to learn for every light component $C \in \mathcal{C}_x$, the collection of compressed paths $\pi^*_y(s, C_{y,y'})$ for every edge $(y, y') \in \pi_x(s, C)$. We show that for every x, there is an $\widetilde{O}(D)$-round algorithm $\mathcal{A}_x$ that has a total congestion $\widetilde{O}(D)$. The algorithm $\mathcal{A}_x$ sends messages only along edges incident to $\mathsf{LD}(x)$, namely, the light descendants of x. Since each vertex v belongs to $O(\log n)$ sets $\mathsf{LD}(x)$ (for each light ancestor x of v), we get that each vertex participates in $O(\log n)$ algorithms. Using the random delay approach of Theorem 1.7, we can then schedule all these $\mathcal{A}_x$ algorithms in $\widetilde{O}(D)$ rounds.

We next describe algorithm $\mathcal{A}_x$ for a fixed x. For component $C \in \mathcal{C}_x \setminus \{H_x\}$, let

$$Q_C = \{\pi^*_y(s, C_{y,y'}) \mid (y, y') \in \pi(s, u_C)\}.$$

Our goal is to let x learn Q_C for every $C \in \mathcal{C}_x$. This can be done as follows. By the preliminary step, v_C receives Q_C from u_C for every $C \in \mathcal{C}_x$. We then let v_C send this $\widetilde{O}(D)$-bit information to x along $\pi(x, v_C)$. Note that the collection of tree paths $\{\pi(x, v_C) \mid C \in \mathcal{C}_x\}$ are edge-disjoint, and therefore, the communication over these paths can be done in parallel.

Consider (ii). We define an algorithm $\mathcal{A}'_x$ for every x and show that its congestion is $\widetilde{O}(1)$. Moreover, $\mathcal{A}'_x$ only sends messages along $V(T_x)$. Altogether, we get that each vertex v participates in $O(D)$ algorithms, and using Theorem 1.7, all $\mathcal{A}'_x$ algorithms can be scheduled in $\widetilde{O}(D)$ rounds. Similarly to part (i) of the proof, for $C = H_x$, there is a vertex $v_C \in C$ that holds the information on $Q'_C = \{\pi^*_y(s, C_{y,y'}) \mid (y, y') \in \pi(s, u_C), (y, y') \in L(T)\}$ where $L(T)$ are the light edges of the tree T. Note that Q'_C has $\widetilde{O}(1)$ bits, and therefore, v_C can send this information to x along the tree path $\pi(v_C, x)$. The communication is indeed restricted to edges incident to H_x. ■

Proof of Claim 4.7. Clearly, x knows $\mathsf{Sketch}_{G\setminus\{x\}}(C)$ for every $C \in \mathcal{C}_x$. To compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(C)$, one needs to cancel-out the sketch information of the edges in $(C \times \{y\}) \cap E(G)$. We show that this can be done by exchanging information along edges incident to $\mathsf{LDS}(x, y)$ and by using the promised channel (e.g., to exchange information related to the heavy component in $\mathcal{C}_x$).

First, by Lemma 4.3, we can also assume that each $v \in \mathsf{LD}(x)$ knows $\pi_x(s, C_{x,v})$ and every v knows $\pi^*_x(s, C_{x,v})$ with respect to its (heavy) ancestors. By exchanging this total amounts of $\widetilde{O}(D)$-bit information with their neighbors, y can locally determine which of its neighbors v in V_x are y-sensitive and which are not.

By Lemma 4.3, y knows $\pi^*_x(s, C_{x,v})$ for each of its neighbors $v \in V_x$. Therefore, by using also its own compressed path $\pi^*(s, y)$, it can determine if it appears in $\pi_x(s, C_{x,v})$, and hence determine if $C_{x,v}$ is y-sensitive or not. Also, as y knows $\pi^*(s, v)$ (since v is its neighbor) and $\pi^*(s, x)$ (since x's ID is augmented with this information), it can determine whether $v \in \mathsf{LD}(x)$. Thus, y can determine all its neighbors in $\mathsf{LDS}(x, y)$.

Altogether, y can determine the edge set $E(y, x') = \{(y, v) \mid v \in T_{x'}\}$ for every child x' of x (where y identifies x' by the unique identity $\pi^*(s, x')$). For every light child x' such that $C_{x,x'}$ is y-sensitive, y sends to one of its neighbors in $T_{x'}$ the sketch information of $E(y, x')$ which has $\widetilde{O}(1)$ bits. This information can be then sent to x using communication inside $T_{x'} \cup \{(x', x)\}$. In addition, y sends over the promised channel sketch information of the edges $E(y, x_h)$ (which consists of $\widetilde{O}(1)$ bits). This allows x to cancel-out the edges of y from the sketch information of every y-sensitive components in $\mathcal{S}(x, y)$. ■

Proof of Claim 4.8. We first show that x and y can compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(U(x, y))$, hence satisfying (I1). Consider x and its x-connectivity tree $\widehat{T}_x$. We start by showing how x can determine the components in $\mathcal{C}_x \setminus \mathcal{FS}(x,y)$. The x-connectivity tree $\widehat{T}_x$ provides x with the knowledge on the y-sensitive components $\mathcal{S}(x,y)$. It remains to distinguish between pseudo y-sensitive and fully y-sensitive components.

By Lemma 4.4(i), for every component $C \in \mathcal{C}_x \setminus \{H_x\}$, x knows $\pi^*_y(s, C_{q,q'})$ for every edge $(q,q') \in \pi_x(s,C)$. Therefore, x can determine if C is fully y-sensitive for every $C \neq H_x$. We then let y send $\pi^*_y(s, C_{y,y_h})$ over the xy channel, and by combining with Lemma 4.4(ii), x can determine also if H_x is fully y-sensitive.

We now explain how to obtain the required sketch information. By Claim 4.7, x can compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(C)$ of every component $C \in \mathcal{S}(x,y)$. By Claim 4.7, y also know $E'_{x,y} = \{(y,v) \mid v \in V_x, y \notin \pi_x(s, C_{x,v})\}$. Using the xy channel, y can send the sketch information of $E'_{x,y}$ to x. This allows x to compute the sketch information in $G \setminus \{x,y\}$ of the union of components $C^{ns}_{x,y} = \bigcup_{C \in \mathcal{C}_x, y \notin \pi_x(s,C)} C$.

Let $C^{ps}_{x,y} = \bigcup_{C \in \mathcal{PS}(x,y)} C$ be the union of all pseudo y-sensitive components in $\mathcal{C}_x$. Then, by Claim 4.7, x can compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(C^{ps}_{x,y})$. As $\mathsf{NS}(x,y) = C^{ns}_{x,y} \cup C^{ps}_{x,y}$, x knows $\mathsf{Sketch}_{G\setminus\{x,y\}}(\mathsf{NS}(x,y))$.

It remains to compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(Q)$ for $Q = V \setminus (V(T_x) \cup V(T_y))$. We first explain how x can compute $\mathsf{Sketch}_G(Q)$. We assume that all vertices know $\mathsf{Sketch}_G(V)$. Using the channel, y can send x, $\mathsf{Sketch}_G(V(T_y))$. Therefore, x can compute $\mathsf{Sketch}_G(U(x,y)) = \mathsf{Sketch}_G(V) \oplus \mathsf{Sketch}_G(V(T_x)) \oplus \mathsf{Sketch}_G(V(T_y))$. Let $E(x,y)$ be the set of x edges incident to vertices in Q. (I.e., the edges connecting x to its non-children in T). In the same manner, y can compute, $E(y,x)$, be the set of y-edges incident to vertices in Q. By letting y send the sketch information of $E(y,x)$ over the channel, x can compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(Q)$ (by canceling the edges $E(y,x)$ and $E(y,x)$, see Lemma 1.6). Altogether, x now knows $\mathsf{Sketch}_{G\setminus\{x,y\}}(Q), \mathsf{Sketch}_{G\setminus\{x,y\}}(\mathsf{NS}(x,y))$. In a symmetric manner, y knows $\mathsf{Sketch}_{G\setminus\{x,y\}}(Q), \mathsf{Sketch}_{G\setminus\{x,y\}}(\mathsf{NS}(y,x))$. By letting x,y exchange $\mathsf{Sketch}_{G\setminus\{x,y\}}(Q')$ for $Q' \in \{\mathsf{NS}(x,y), \mathsf{NS}(y,x)\}$, both can compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(U(x,y))$.

Next consider (I2,I4). Since z knows its components in $\mathcal{P}_{0,z}$, by Claim 4.7, it can compute $\mathsf{Sketch}_{G\setminus\{x,y\}}(C)$ for every $C \in \mathcal{P}_{0,z}$ for $z \in \{x,y\}$, as well as their part-IDs. By exchanging the sketch and part-ID information of the x-heavy and y-heavy parts over the xy channel, invariant (I3) holds as well. ■

Proof of Claim 4.9. First consider the case where P is a subset of either V_x or V_y. W.l.o.g., assume that $P \subseteq V_x$, and since P is light, we have that $P \subseteq \mathsf{LD}(x)$. In such a case P is a union of T_u subtrees for a subsets of light children u of x. The aggregation of the $val(v)$ values is performed, in parallel, in each T_u tree. Each root vertex u then sends the aggregate value of the vertices in T_u to x, which allows x to compute the output for the entire part P. This output can be then propagated down the T_u trees. The same procedure can be applied when $P \subseteq V_y$.

We next turn to consider the case where P contains vertices from both V_x and V_y. Since P is xy-connected and $s \notin P$, there is some non-tree edge (u,v) connecting $u \in P \cap \mathsf{LD}(x)$ and $v \in P \cap \mathsf{LD}(y)$.

Let $P_x = P \cap \mathsf{LD}(x)$ and $P_y = P \setminus P_x$ and let q_x, q_y be the output of the aggregate function restricted to the $val(v)$ values in P_x, P_y respectively. We first let the vertices in P_x, P_y compute the values q_x, q_y, as described above. We then let P compute the non-tree edge (u,v) connecting $u \in P \cap \mathsf{LD}(x)$ and $v \in P \cap \mathsf{LD}(x)$ of maximum ID, among all possible such edges. This again can be done internally inside P_x, P_y. At the end of this step, all nodes in P know the identity of the selected (u,v) edge. By exchanging q_x, q_y over the edge (u,v), u and v can obtain the combined output value w.r.t P. This can be broadcast on P by letting u,v sending this information to x,y

respectively, and then broadcasting it in each x, y subtrees in P. Since the communication for a part P uses only edges incident to P (without using the promise channel), and since the parts in $\mathcal{P}_i$ are vertex-disjoint, this computation can be done for all the light parts in $\mathcal{P}_i$, in parallel. ■

Proof of Lemma 4.12. In order to apply Cor. 4.6, it remains to show that both x, y can learn that they are a light pair, and that their $\Pi_{x,y}$ can be established in a distributed manner.

We starting by observing that for each light pair x, y at least one of the vertices (either x or y) can learn (with no communication) that x, y is a light pair. Assume w.l.o.g. that there is a component $C \in \mathcal{C}_x$ such that either (i) $u_C = y$ (i.e., the last edge of $\pi_x(s, C)$ is (y, v_C) for $v_C \in C$), or that (ii) there is a light edge $(y, y') \in \pi_x(s, C)$ such that $x \in \pi_y(s, C_{y,y'})$. Then, by Lemma 4.4, x can determine that xy is a light pair.

We now describe algorithm $\mathcal{A}_x$ for a fixed x and show that x can notify all its light-mates y using the paths $\Pi_{x,y}$. Note that all the vertices $a \in V_x$ know $\pi^*(s, C_{x,a})$ and therefore they know the edge $(u_{C_{x,a}}, v_{C_{x,a}})$. For each light-mate y of x, let $C^{x,y} \in \mathcal{C}_x$ be the chosen component that satisfies (i) or (ii) w.r.t y. We then let x send a message to the $v_{C^{x,y}} \in C^{x,y}$ vertices. Note that the edge-congestion of this procedure is $O(\log n)$ as each vertex $v \in V_x$ receives messages regarding light edges on $\pi_x(s, C_{x,v})$, and there are only $O(\log n)$ such edges. Once $v_{C^{x,y}}$ is informed, this information can be sent to the second endpoint $u_{C^{x,y}} \in T_y$. Overall, we can schedule all $\mathcal{A}_x$ algorithms in $\widetilde{O}(D)$ rounds, using random delays.

At this point, for each y, there is a vertex $u_{C^{x,y}} \in T_y$ that is required to upcast this information to y. Since $u_{C^{x,y}}$ is a light descendant of y, and since the component of $u_{C^{x,y}}$ in $\mathcal{C}_y$ is x-sensitive, each vertex is required to send $\widetilde{O}(D)$ messages. Using random delays (or just pipeline) this can be done in $\widetilde{O}(D)$, as well. At the end of this procedure, the promised channel $\Pi_{x,y}$ is known in a distributed manner. The collection of $A_{x,y}$ algorithms can be then scheduled using Cor. 4.6. ■

Proof of Claim 4.13. By using $\widetilde{O}(D)$ rounds, each vertex v can learn the compressed-path $\pi^*_x(s, H_x)$ for each of its ancestors $x \in \pi(s, v)$. Since each v also knows the ID of $C_{x,v}$, it can tell if its belongs to H_x.

We now focus on $x \in V$ and show that $\mathsf{TD}(x)$ can be computed by an algorithm $\mathcal{A}_x$ that sends information only incident to H_x and $\widetilde{O}(1)$ congestion. Let (u_{H_x}, v_{H_x}) be the last edge of $\pi^*_x(s, H_x)$ (where $v_{H_x} \in H_x$). The goal is to compute an edge $(c, d) \in H_x \times V \setminus V(T_x)$ such that LCA((u_{H_x}, d) is of minimal possible depth. Note that this LCA must fall on $\pi_x(s, H_x)$ and that (u_{H_x}, v_{H_x}) is known to all vertices in H_x as it is part of the compressed-path $\pi^*_x(s, H_x)$. We show that by aggregating information on T_x from the leaf up to the vertex x, one can compute $\mathsf{TD}(x)$. Initially, each vertex $v \in H_x$ that has incident edges to $V \setminus V(T_x)$ keep the edge (v, d) such that the depth of LCA(u_{H_x}, d) is minimized. By sending this information along T_x the *best* edge in $(c, d) \in H_x \times V \setminus V(T_x)$ can be detected. This allows x to determine $\mathsf{TD}(x)$. The computation of all vertices x can be done in $O(D)$ rounds, using a standard pipeline technique. ■

Proof of Lemma 4.14. First, we let each vertex v define a fresh sketch information $\mathsf{Sketch}'_G(v)$. By aggregating this information from the leaf vertices towards the root s, every vertex y can compute $\mathsf{Sketch}'_{G\setminus\{y\}}(H_y)$ where $H_y = C_{y,y_h} \in \mathcal{C}_y$.

We next repeat the algorithm for computing the x-connectivity trees $\widehat{T}_x$ with the only distinction that we augment the ID of each vertex y with

$$\mathrm{EID}'(y) = \langle ID(y), \mathsf{Sketch}'_{G\setminus\{y\}}(H_y), \mathsf{TD}(y), \pi^*_y(s, H_y)\rangle.$$

The extend-ID of each edge (u, v) includes in-addition the $\mathrm{EID}'(u), \mathrm{EID}'(v)$. Note that $\mathsf{Sketch}'(.)$ is a fresh sketch information, that is independent of the sketch information $\mathsf{Sketch}(.)$ used to compute the $\widehat{T}_x$ trees (using independent seeds). Also, the extended-ID $\mathrm{EID}'(y)$ has only $\widetilde{O}(1)$ bits. Since we use the same sketch information for computing the $\widehat{T}_x$ trees, for each vertex y in $\widehat{T}_x$, x also learns its extended-ID $\mathrm{EID}'(y)$ which satisfies the claim. ■

Proof of Claim 4.16. Let $Q_v = \bigcup_{y \in \mathsf{LA}(v)} \pi_y(s, C_{y,v})$. By Lemma 4.3, v knows all edges in Q_v. In addition, we let each v send Q_v to all its neighbors which can be done within our time budget as Q_v has $\widetilde{O}(D)$ bits. Note that for every descendant u of v in T, it holds that $Q_v \subseteq Q_u$. Next, we compute these sketches by aggregating the information from the leaf vertices up to the root (and using Lemma 1.6). Each vertex v starts by computing $\mathsf{Sketch}'_{G \setminus \{x\}}(v)$ for each $x \in Q_v$. In each subtree T_v we then need to compute $|Q_v|$ aggregate functions inside T_v, since $Q_v \subseteq Q_u$ for every $u \in T_v$, this computation can be done by a standard pipeline using $\widetilde{O}(D)$ rounds. ■