Near-Optimal Distributed DFS in Planar Graphs

Mohsen Ghaffari[1] and Merav Parter[2]

1 **ETH Zurich.** `ghaffari@inf.ethz.ch`
2 **CSAIL, MIT.** `parter@mit.edu`

Abstract

We present a randomized distributed algorithm that computes a Depth-First Search (DFS) tree in $\widetilde{O}(D)$ rounds, in any planar network $G = (V, E)$ with diameter D, with high probability. This is the first sublinear-time distributed DFS algorithm, improving on a three decades-old $O(n)$ algorithm of Awerbuch (1985), which remains the best known for general graphs. Furthermore, this $\widetilde{O}(D)$ round complexity is nearly-optimal as $\Omega(D)$ is a trivial lower bound.

A key technical ingredient in our results is the development of a distributed method for (recursively) computing a *separator path*, which is a path whose removal from the graph leaves connected components that are all a constant factor smaller. We believe that the general method we develop for computing path separators recursively might be of broader interest, and may provide the first step towards solving many other problems.

1998 ACM Subject Classification G.2.2 Graph Algorithms

Keywords and phrases DFS, planar graphs, CONGEST

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction and Related Work

Depth First Search (DFS) is "*one of the most versatile sequential algorithm techniques known for solving graph problems*" [42]. Along with its cousin BFS, these two have a long history: DFS dates back to the 19th century [11], and BFS dates back to the 1950s [44]. Both were first used for solving different kinds of mazes, but are nowadays among basic building blocks in graph algorithms, covered in elementary courses, and with a wide range of applications.

In the centralized setting, computing BFS and DFS are straightforward. However, in the distributed setting, there is a stark contrast, and DFS turns out to be much harder. Let us first recall the definition of the distributed model.

Throughout, we use a standard message passing model of distributed computing called CONGEST [40]. The network is abstracted as an n-node graph $G = (V, E)$, with one processor on each network node. Initially, these processors do not know the graph. They solve the given graph problems via communicating with their neighbors. Communications happen in synchronous rounds. Per round, each processor can send one $O(\log n)$-bit message to each of its neighboring processors.

Distributedly computing both BFS and DFS need $\Omega(D)$ rounds, in graphs of diameter D. BFS can be computed easily in $O(D)$ rounds, in any graph with diameter D. In contrast, the best known distributed algorithm for DFS takes $O(n)$ rounds, regardless of how small diameter D is; see e.g., [40, Section 5.4] and [4]. We note that designing algorithms with complexity $o(n)$, when $D = o(n)$, and ideally close to $O(D)$, has become the target of essentially all the distributed graph algorithms for global optimization problems, since the pioneering work of Garay, Kutten, and Peleg [14, 30] which gave an $O(D + n^{0.61})$ round algorithm for minimum spanning tree. See, e.g., [6, 8, 9, 13, 15, 16, 19, 20, 27, 28, 32, 33, 36–38].

Despite this, there has been no progress on the problem over the last three decades, and no sublinear-time distributed algorithm for DFS is known. This, and especially the

lack of any sub-linear time distributed DFS, is certainly not for the lack of trying. It is widely understood that DFS is not easy to parallelize/decentralize; it has even been called *"inherently sequential"* [42] and "*a nightmare for parallel processing*" [35].

1.1 Our Contribution

In this paper, making a first step of progress on the distributed complexity of this classical problem, we present a randomized distributed algorithm that computes a DFS in $O(D \cdot \text{poly}\log n)$ rounds in planar graphs, with high probability. This time complexity is nearly optimal as it matches the trivial $\Omega(D)$ lower bound up to poly-logarithmic factors.

▶ **Theorem 1.** *There is a randomized distributed algorithm that computes a DFS in any n-node planar network with diameter D in $O(D \cdot \text{poly}\log n)$ rounds, with high probability.*

Turning to general graphs, we note that the parallel algorithms by Aggarwal and Anderson [1] and Goldberg, Plotkin and Vaidya [22] can be adapted to give an $\widetilde{O}(\sqrt{Dn} + n^{3/4})$-round DFS algorithm for graphs with diameters D. Therefore, a DFS can be computed in sublinear number of rounds for graphs with sublinear diameter $D = o(n)$. This simple corollary of [1] and [22] is shown in Appendix D. Improving the bound for general graphs remains an important open problem.

1.2 A High-Level Discussion of Our Method

Our method relies on *separator paths*. Generally, *separators* have been a key tool in working with planar graphs, starting with the seminal work of Lipton and Tarjan [34]. In a rough sense, separators are subgraphs whose removal from the graph leaves connected components that are all a constant factor smaller than the initial graph. Typically, one desires the separator to be small. However, unlike [34], we do not insist on a small separator, but instead it is crucial for us that the separator is a simple *path*. This allows us to use the separator path, with some iterations of massaging and modifications in the style of [1], as a part of a partial DFS. See Section 3 for the explanations. Now that this separator is put in the partial DFS, the left over graph is made of a number of connected components, each a constant factor smaller than the initial graph. Hence, we would have the hope to be able to solve each of these subproblems recursively, and moreover, to do that simultaneously for all the subproblems.

But two key issues remain: (1) How do we compute the separator path distributedly? This itself is the main technical contribution of our paper, and is explained in Section 4. But a crucial part of the challenge of that lies in the next point. (2) How do we recurse and most importantly, how do we compute the separator path throughout the recursions? Once we remove the first separator, the left over components are smaller in size, but they may have considerably larger diameter, even up to $\Theta(n)$. This large diameter can be a major obstacle for distributed algorithms. For instance, we cannot even assume that we can compute a BFS of each component. More generally, if we are to have a fast separator path algorithm, we cannot confine the algorithm for each subproblem to stay within the connected component of that subproblem. On the other hand, allowing the algorithm to use the other parts creates the possibility of *congestion* as now many subproblems may need to use the same edge, perhaps many times each.

Our solution uses a number of novel algorithmic ideas. It would be hard to summarize these ideas out of context, and thus we refer the interested reader to the technical sections. One key tool from prior work, which is worth pointing out and makes our life significantly

easier, is *low-congestion shortcucts* for planar graphs, developed by Ghaffari and Haeupler [18]. In a very rough sense, this tool opens the road for working on many disjoint potentially large-diameter subgraphs of the base graph G at the same time, and still enjoying the small diameter of the base graph G. Though, this is possible only in certain conditions and only for a very limited class of problems. We usually need quite some work to break our problems into modules that fit these conditions and classifications.

Separators have a wide range of applications, in centralized algorithms for planar graphs. Though, computing the separators distributedly and especially computing them recursively in the distributed setting when the remaining components have large diameter is highly non-trivial. We thus hope that the methods developed in this paper may open the road for recursive computation of separators in the distributed setting, and thus be a first step towards solving many other problems.

1.3 Related Work

Distributed Graph Algorithms in Sublinear Time: Over the past two decades, starting with the seminal work of Garay, Kutten, and Peleg [14, 30], there has a been a big body of work presenting sublinear-time distributed algorithms for various graph problems (for graphs with diameter $D = o(n)$, as otherwise that is impossible). See for instance[1] [6, 8, 9, 13, 15, 16, 19, 20, 27, 28, 32, 33, 36–38]. There are also lower bounds [7, 10, 41] which for instance show that in general graphs, computing minimum spaning trees requires $\tilde{\Omega}(D + \sqrt{n})$ rounds, hence ruling out the possibility of $\tilde{O}(D)$ round MST algorithms. A similar lower bound holds for many other problems, even when approximating, e.g., min-cut, shortest paths, min-cost connected dominated set etc. See [7]. By now, most of the classical graph problems are known to have sublinear-time distributed algorithms, at least when relaxing the problem to allow some approximation. A prominent exception is DFS!

Distributed Graph Algorithms in Planar Networks: Starting with the work of Ghaffari and Haeupler [17, 18], some attention has been paid to developing more efficient distributed algorithms for (global) network optimizations on planar or near-planar networks. This was in part motivated by trying to circumvent the aforementioned $\tilde{\Omega}(D+\sqrt{n})$ general-graph lower bound. Another motivation was also to bring in the vast array of the techniques and methodologies developed for efficient centralized algorithms for planar networks to the distributed domain.

In [18], the aformentioned lower bound was ruled out for planar networks by showing an $\tilde{O}(D)$ algorithm for MST in planar networks. A key tool in this MST algorithm was the concept of *low-congestion shortcuts*, which was introduced in [18]. An algorithm for computing this structure was also given in [18], which as one of its subroutines made use of the planar embedding algorithm of [17]. It was shown later by Haeupler, Izumi and Zuzic [23] that even without having the embedding, one can compute an approximate version of the low-congestion shortcuts, which is good enough for many applications. Furthermore, low congestion shortcuts were later extended to some near-planar graphs, concretely graph families with fixed tree-width or genus [24].

Parallel DFS Algorithms: DFS has received vast attention in the parallel literature. It is known that computing the lexicographically-first DFS —where the smallest ID unvisited neighbor should be visited first — is P-complete [42], and is thus unlikely to admit an efficient parallel algorithm. This was the reason that DFS was deemed *"inherently sequential"* [42]. However, over the years, several sophisticated but efficient parallel algorithms

[1] This is merely a sample, and is by no means exhaustive.

were developed for DFS, which compute some depth first search tree (not necessarily the lexicographically-first one). We here review the related work on only undirected graphs. Smith [45] gave an $O(\log^3 n)$-time parallel DFS algorithm for planar graphs. Shannon [43] improved this to an $O(\log^2 n)$-time parallel DFS algorithm for planar graphs, while also using only linear number of processors. Anderson gave a $\tilde{O}(\sqrt{n})$-time [2] and then a $2^{O(\sqrt{\log n})}$-time [3] parallel DFS algorithm for general graphs. Aggarwal and Anderson [1] gave the first poly-logarithmic time parallel algorithm for DFS in general undirected graphs. Kao [29] gave the first deterministic NC algorithm for DFS in planar networks. Then Hagerup [25] gave an $O(\log n)$-time randomized parallel DFS algorithms for planar networks. Finding a deterministic NC algorithm for DFS in general graphs remains open, though a quasi-NC algorithm was given very recently in [12].

We note that our distributed DFS algorithm for planar graphs is quite different than the parallel DFS algorithms for planar graphs (e.g., [43, 45]), mainly because we do not compute a separator *cycle* distributedly. Our algorithm is morally closer to the methodology of Aggarwal and Anderson (for general graphs) [1] which can work with (collections of) separator *paths*.

2 Preliminaries

Basic Notions: Let $G = (V, E)$ be a simple undirected planar graph. Given a tree $T \subseteq G$ and a non-tree edge $e = (v, u) \notin T$, the cycle formed by e and the tree path connecting v to u is called the *fundamental cycle* defined by e. Let $\mathcal{F}(G) = \{F_1, \ldots, F_k\}$ be the faces of the planar graph G. Let $G' = (V', E')$ be the *dual graph* of G, defined by including one node $v'_i \in V'$ for each face $F_i \in \mathcal{F}$ and connecting two nodes $v'_i, v'_j \in V'$ if their corresponding faces share an edge[2]. We may interchange between the dual-nodes v'_i and the faces F_i. A superface $\mathcal{F}$ is a collection of faces whose boundary is a simple cycle.

Dual Tree and its Distributed Representation: Given a spanning tree T of G, we define its dual-tree in the dual-graph G' as follows: Let $\phi_F : \mathcal{F}(G) \to V'$ be the bijection between the faces of G and the dual-nodes of G'. The nodes of dual tree T' are the faces of G, and two dual-nodes v'_i and v'_j are connected iff the two faces $\phi_F^{-1}(v'_i)$ and $\phi_F^{-1}(v'_j)$ share an non-tree edge $e \notin T$. We define a bijection $\phi_E : E \setminus E(T) \to E(T')$ between the non-tree edges $G \setminus T$ and the dual-tree edges of T', where in the aforementioned example, we have $\phi_E(e) = \{v'_i, v'_j\}$.

In the distributed representation of this dual-tree, the leader $\ell(e')$ of a dual-edge $e' \in T'$ is the higher-ID endpoint of the edge $\phi_E^{-1}(e') = \{u, v\}$. The leader $\ell(v')$ of the dual-node v' is the node in the face $\phi_F(v')$ of maximum ID. The dual-tree is known in a distributed manner where for every edge $e' \in T'$, its leader $\ell(e')$ knows that this edge belongs to T'. The nodes $v' \in V(T')$ and dual edges $e' \in E(T')$ will be simulated by their leader nodes $\ell(v')$ and $\ell(e')$ respectively.

Planar Embeddings: The geometric planar embedding of graph G is a drawing of G on a plane so that no two edges intersect. A combinatorial planar embedding of G determines the clockwise ordering of the edges of each node $v \in G$ around that node v such that all these orderings are consistent with a plane drawing (i.e., geometric planar embedding) of G. Ghaffari and Haeupler [17] gave a deterministic distributed algorithm that computes a combinatorial planar embedding in $O(D \min\{\log n, D\})$ rounds, where each node learns the clockwise order of its own edges.

[2] In-fact, the dual-graph is a multigraph as there might be many edges between two dual-nodes

Low-Congestion Shortcuts: In a subsequent paper [18], Ghaffari and Haeupler introduced the notion of *low-congestion shortcuts* which plays a key role in several algorithms for planar graphs (e.g., MST, min-cut). We will also make frequent use of this tool. The definition is as follows.

▶ **Definition 2.** (**α-congestion β-dilation shortcut**) Given a graph $G = (V, E)$ and a partition of V into disjoint subsets $S_1, \ldots, S_N \subseteq V$, each inducing a connected subgraph $G[S_i]$, we call a set of subgraphs $H_1, \ldots, H_N \subseteq G$, where H_i is a supergraph of $G[S_i]$, an α-congestion β-dilation shortcut if we have the following two properties: (1) For each i, the diameter of the subgraph H_i is at most β, and (2) for each edge $e \in E$, the number of subgraphs H_i containing e is at most α.

Ghaffari and Haeupler [18] proved that any partition of a D-diameter planar graph into disjoint subsets $S_1, \ldots, S_N \subseteq V$, each inducing a connected subgraph $G[S_i]$, admits an α-congestion β-dilation shortcut where $\alpha = O(D \log D)$ and $\beta = O(D \log D)$. They also gave a randomized distributed algorithm that computes such a shortcut in $\tilde{O}(D)$ rounds, with high probability. We will make black-box use of this result, frequently.

3 Outline of the Depth First Search Tree Construction

Towards proving Theorem 1, in this section, we explain the outline of our $\widetilde{O}(D)$-round algorithm for computing a *Depth-First Search* (DFS) tree. Detailed steps are explained in later sections.

We compute a DFS tree of a graph $G = (V, E)$ rooted in a given node $s \in V$. The algorithm is based on a *divide-and-conquer* style approach. A key technical ingredient is a *separator path* algorithm, which we use for *dividing* the problem into independent subproblems of constant factor smaller size. We describe this separator algorithm in the next section. In this section, we explain how via recursive (black-box) applications of a separator path subroutine, we compute a DFS.

We note that our approach is inspired by an idea of Aggarwal and Anderson [1]. However, the overall method is quite different. On one hand, we have an easier case here because we need to deal with only a *single path* instead of a large collection of them, thanks to the nice structure of planar graphs. On the other hand, computing this single path, and especially being able to do it recursively, has its own challenges, as we discuss in the next section. We will have to deal with a number of difficulties that are unique to the distributed setting, as we will point out.

High-Level Outline: The high-level outline of the approach is as follows. The method is recursive. In each (independent) branch of the recursion, we have a connected induced subgraph $\mathcal{C} \subseteq G$ and a root $r \in \mathcal{C}$ and we need to compute a DFS of $\mathcal{C}$ rooted in r. In the beginning, we simply have $\mathcal{C} = G$ and $r = s$. Furthermore, in each step of recursion, we will assume that $\mathcal{C}$ is *biconnected*, that is, removing any single node $v \in \mathcal{C}$ from $\mathcal{C}$ leaves a connected subgraph $\mathcal{C} \setminus \{v\}$. Notice that this may not hold at the beginning, that is, G may have some cut-nodes. We will later discuss how to deal with cut nodes, by dividing the problem further into a number of independent subproblems, one for each biconnected component. For now, we assume that $\mathcal{C}$ is biconnected.

The Framework of One Recursion Level: We wish to compute a partial DFS $\mathcal{T}$ of $\mathcal{C}$ rooted at r such that each connected component of $\mathcal{C} \setminus \mathcal{T}$ has size at most $2|\mathcal{C}|/3$. This partial DFS $\mathcal{T}$ of $\mathcal{C}$ is such that it can be completed to a full DFS of $\mathcal{C}$ rooted at r. In particular, it has the following *validity* property: there are no two branches of $\mathcal{T}$ which are connected to each other via a path with all its internal nodes in $\mathcal{C} \setminus \mathcal{T}$. In other words, for each connected component C_i of $\mathcal{C} \setminus \mathcal{T}$, all neighbors of C_i in $\mathcal{T}$ are in one branch (i.e., rooted path) of

$\mathcal{T}$. Once we compute this partial DFS $\mathcal{T}$, we can then recurse on each of those remaining connected components of $\mathcal{C} \setminus \mathcal{T}$, all in parallel. As the component size decreases by a 2/3 factor per level of recursion, the recursion has depth $O(\log n)$.

The Procedure for One Recursion Level: Thus, the key is to grow a partial DFS $\mathcal{T}$ of $\mathcal{C}$ in $\widetilde{O}(D)$ rounds, in a way that each connected component of $\mathcal{C} \setminus \mathcal{T}$ has size at most $2|\mathcal{C}|/3$. We will do this in $\tilde{O}(D)$ rounds. For that purpose, we compute a *separator path* $\mathcal{P} \subseteq \mathcal{C}$ of $\mathcal{C}$. That is, each connected component of the graph $\mathcal{C} \setminus \mathcal{P}$ has size at most $2|\mathcal{C}|/3$. We explain this subroutine in the next section. For now, let us assume that such a path $\mathcal{P}$ is computed.

Let $\mathcal{Q}$ be a simple path that connects the root r to some node in $\mathcal{P}$ (and is otherwise disjoint from $\mathcal{P}$). Let v be the endpoint of $\mathcal{Q}$ in path $\mathcal{P}$ and suppose that $\mathcal{P} = u_1, u_2 \ldots, u_\ell, v, w_1, w_2, \ldots, w_{\ell'}$. Let $\mathcal{P}_1 = u_1, u_2 \ldots, u_\ell, v$ and $\mathcal{P}_2 = v, w_1, w_2, \ldots, w_{\ell'}$. We will use the longer one of $\mathcal{P}_1$ and $\mathcal{P}_2$ and append it to the path $\mathcal{Q}$ connecting the root r to v. Without loss of generality, suppose that the longer subpath is $\mathcal{P}_2$. We add the path $Q \cup \mathcal{P}_2$ as the first branch of the DFS $\mathcal{T}$. Moreover, we update the separator $\mathcal{P}$ to be the remaining part of the separator path, concretely $\mathcal{P}_1$ in the assumed case. We note that this is the idea that we borrow from Aggarwal and Anderson [1]. We have two important properties: (1) the new path $\mathcal{P}$ is still a separator of $\mathcal{C} \setminus \mathcal{T}$, and (2) the length of the new separator path $\mathcal{P}$ is at most half of the length of the previous separator.

Thanks to these two properties, we have the means to continue and exhaust the separator path in $O(\log n)$ repetitions. Each time we grow the partial DFS $\mathcal{T}$ further. Let us explain one step of this repetition. Figure 1 illustrates an example for this step. We find the deepest node r' in the current partial DFS $\mathcal{T}$ rooted at r that is directly or indirectly connected to a node in the current separator $\mathcal{P}$, in the graph $\mathcal{C} \setminus \mathcal{T}$. Notice that this deepest node is unique, due to the validity of the current partial DFS, as all neighbors of the connected component of $\mathcal{C} \setminus \mathcal{T}$ containing $\mathcal{P}$ are in one branch of $\mathcal{T}$. We then find a path Q as above starting from r' and connecting to some node v in $\mathcal{P}$. This is done with the help of an $\widetilde{O}(D)$ round *minimum spanning tree* (MST) algorithm of Ghaffari and Haeupler [18], as we outline next.

In particular, let each node of $\mathcal{T}$ send its DFS depth to its neighbors in $\mathcal{C} \setminus \mathcal{T}$. Then, we run a connected component identification algorithm of [18] on the subgraph $\mathcal{C}' = \mathcal{C} \setminus \mathcal{T}$ of G. In identifying the connected components of the graph $\mathcal{C}' = \mathcal{C} \setminus \mathcal{T}$, the component leader is chosen according to having a $\mathcal{T}$-neighbor with deepest DFS depth. This finds the deepest node r' of $\mathcal{T}$ that is in the connected component of $\mathcal{P}$ and thus has a path to $\mathcal{P}$. Furthermore, we can find a path $\mathcal{Q} \subset \mathcal{C} \setminus \mathcal{T}$ connecting r' to $\mathcal{P}$ in a similar manner.

Let us explain this step for finding path $\mathcal{Q}$, in $\tilde{O}(D)$ rounds. On the graph $\mathcal{C}'$, give an edge weight of 0 to each $\mathcal{P}$ edge and edge weight of 1 to each $\mathcal{C}' \setminus \mathcal{P}$ edge. Then, compute an MST of $\mathcal{C}'$ according to these weights using the algorithm of [18]. The unique path of weight-1 edges in the MST that connects node r' to a node $v \in \mathcal{P}$ is our desired path $\mathcal{Q}$. This path $\mathcal{Q}$ can be identified in $\widetilde{O}(D)$ rounds.

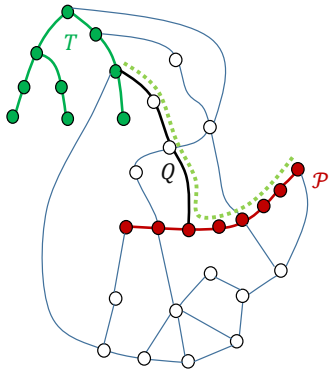

Figure 1 Growing the partial DFS tree. The green tree shows the current partial DFS $\mathcal{T}$, and the rest of the nodes are those of $\mathcal{C} \setminus \mathcal{T}$. The red path shows the current separator path $\mathcal{P}$. The black path is $\mathcal{Q}$, which connects the deepest point of $\mathcal{T}$ to some node in $\mathcal{P}$. The green dotted line indicated the path that will be added to the DFS, which is composed of $\mathcal{Q}$ and the longer half of $\mathcal{P}$ from the point of intersection with $\mathcal{Q}$. After this DFS growing, the leftover separator $\mathcal{P}$ will be the single edge at the left endpoint of $\mathcal{P}$. This is still a separator path of $\mathcal{C} \setminus \mathcal{T}$, for the new $\mathcal{T}$.

One endpoint of it r' is clear by now. We first identify the other endpoint v, as follows: Discard all the zero-weight edges of the MST. Then, with another iteration of [18] on the subgraph edge-induced by weight-1 edges of the MST, we can identify the node $v \in \mathcal{P}$ who is the endpoint of the path $\mathcal{Q}$ connecting r' to $\mathcal{P}$. This is the only $\mathcal{P}$-node in the same component with r'. Now that we have the two endpoints r' and v of our path $\mathcal{Q}$, which is a part of the computed MST, we can fully mark this path $\mathcal{Q}$ in $\widetilde{O}(D)$ rounds, easily. We defer the details of that step to Appendix C.2, where we explain a routine for marking a tree-path connecting two nodes.

Now that we have found a path $\mathcal{Q}$ connecting the deepest possible node r' of $\mathcal{T}$ to a node $v \in \mathcal{P}$, we work as before. We break $\mathcal{P}$ at v, as depicted in Figure 1, and append the longer half to Q, and then add the resulting path to the DFS $\mathcal{T}$, essentially hanging it from node $r' \in \mathcal{T}$. This is the dotted green path in Figure 1. One can see that, as we chose v to be the deepest $\mathcal{T}$-node with a connection to $\mathcal{P}$, the resulting new tree $\mathcal{T}$ preserves the validity property. That is, each remaining connected component of $\mathcal{C} \setminus \mathcal{T}$ has neighbors in only one branch of the this new DFS tree $\mathcal{T}$. This is because each newly added node is connected to the deepest possible point in the DFS. After each such repetition, the length of the remaining separator path P decreases by a $1/2$ factor. Hence, after $O(\log n)$ iterations, we exhaust $\mathcal{P}$. At that point, we have a valid partial DFS $\mathcal{T}$ rooted at r and furthermore, $\mathcal{C} \setminus \mathcal{T}$ is made of connected components, each of which has size at most $2|\mathcal{C}|/3$.

Preparation for Next Recursions: At this point, we are almost ready for recursing on the connected components of $\mathcal{C} \setminus \mathcal{T}$, each as a subproblem of its own. Though, we should do a preparation step so that each subproblem is in the format that we assumed above, while describing the recursive step. In particular, we should identify the connected components $C_1, C_2, \ldots, C_\ell$ of $\mathcal{C} \setminus \mathcal{T}$, by giving a connected component identifier to each of them, and more importantly, we should declare a DFS root for each of them. Let each node in $\mathcal{T}$ sends its DFS depth to each of its neighbors in $\mathcal{C} \setminus \mathcal{T}$. Then, for each component C_i, we define the component leader and also the DFS root r_i to be a node $v \in C_i$ that received the greatest DFS depth from its $\mathcal{T}$ neighbors (breaking ties based on the id of v). Notice that for each component, this greatest depth $\mathcal{T}$-node is uniquely defined, because of the validity of the partial DFS. Moreover, this is a valid DFS root, in the sense that adding a DFS of C_i rooted at r_i to the current partial DFS $\mathcal{T}$ would be a correct partial DFS. These component leaders (i.e., component-wise DFS roots r_i) can be identified for all the connected components in parallel in $\widetilde{O}(D)$ rounds, using the connected component identification algorithm of Ghaffari and Haeupler [18] for planar graphs. It is crucial to note that here D is the diameter of the very base graph G and not just $\mathcal{C}$. See [18] for details.

Dealing with Cut Nodes: Finally, we come back to the assumption of the connected component $\mathcal{C}$ being biconnected, and we address the possibility of having cut nodes. Figure 6 illustrates an example for this case, where a connected component $\mathcal{C}$ is drawn which has several cut nodes. In this case, we break the problem into several independent DFS problems that can be solved independently. In particular, we will partition the graph into edge-disjoint parts, each being one of the biconnected components of $\mathcal{C}$, and we solve a rooted DFS problem in each of these biconnected components. The root of the biconnected component containing root r is node r itself. For each other biconnected component C, the DFS root is the cut node of C that lies on the shortest path to the root r. It is easy to see that if we compute these rooted DFSs and glue them together in the natural way—hanging the DFS of each biconnected component C from its root as a subtree of DFS of the neighboring biconnected component closer to the node r — we get a DFS of $\mathcal{C}$. Computing a rooted DFS in each of these biconnected component is performed using the recursive method explained above. So, what remains to be explained is identifying two things (1) the biconnected components of $\mathcal{C}$, and (2) the corresponding DFS roots. We describe these components in Appendix C.1.

4 Computing A Separator Path

4.1 Method Outline and Challenges

A celebrated result of Lipton and Tarjan [34] demonstrates the existence of a *separator path* in planar graphs. Their proof shows that

> Any spanning tree T in a planar graph $G = (V, E)$ contains a tree path $P \subseteq T$ which is a *separator path*. That is, each connected component of $G \setminus P$ contains at most $2|V|/3$ nodes.

If one takes T to be a BFS (i.e., shortest path tree) of G, then the separator consists of at most two shortest paths. Hence, in this case, the separator path also has a small length of $O(D)$. For our purposes in this section, we do not need a small separator. Moreover, for reasons that shall become clear during the recursive steps, we will not be able to pick our separators based on BFS trees (of the remaining components). We will work with more general trees, and thus will not insist on the separator path being small. As a side remark, we note that if we did not need the separator to be a path, then there would be ways for having it be also small (even throughout the recursions).

In most applications of separators, we need to compute the separators not *once* but rather many times, recursively. That is, after computing a separator path in G, the separator is removed and the graph breaks into connected components; then in each component, we compute a separator and recurse. The first recursion level where we compute the separator in G may be delusively simple. This is because, whereas the diameter of G is D, in later levels, we need to compute the separator in connected induced subgraphs $\mathcal{C}$, which potentially may have much larger diameter than D.

Throughout this section, we describe how to compute a separator for a given induced subgraph $\mathcal{C} \subseteq G$, which is biconnected, but may have diameter much larger than D. We note that in reality, there are potentially many subgraphs $\mathcal{C}_1$, $\mathcal{C}_2$, ..., $\mathcal{C}_N$ for which we are computing separators, at the same time. Our description focuses on just one of these. Dealing with all these disconnected subgraphs in parallel will follow by standard usage of low-congestion shortcuts.

To avoid cumbersome notation, let us abuse notation and use n as the size of the subgraph $\mathcal{C}$. Our algorithm will compute a path P that breaks $\mathcal{C} \setminus P$ into components of size $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$, for a small constant $\epsilon > 0$, say $\epsilon = 0.01$.

Algorithm Outline: Here, we describe a high-level outline of the algorithm for finding a separator path. A more concise description of the algorithm is given in Appendix B.2. We start by computing a spanning tree T in $\mathcal{C}$. This is done using the MST algorithm of [18], in $\widetilde{O}(D)$ rounds, where D is the diameter of the base graph G rather then the diameter of the subgraph $\mathcal{C}$. Our separator path will correspond to a fundamental cycle of the MST tree T in $\mathcal{C}$. Picking this primal tree T also leads to defining a *dual tree* T', containing the *dual edges* of the non T-edges, as described in Section 2. See Figure 2. In this dual-tree T', each two faces who share a non-tree edge $e \notin T$ are adjacent. We will use this dual tree T' to find a collection of faces, i.e., dual nodes, that can be merged into a superface whose boundary can be used as a separator.

To choose a separator path on the tree T, we introduce the notion of *weight* for the dual-tree T'. We define the *weight* of a superface to be the number of nodes on the superface boundary plus the number of nodes inside the superface. Let $\mathcal{F}_i$ denote the superface corresponding to the dual-node v'_i, obtained by merging the faces of all dual-nodes in the subtree $T'(v'_i)$, i.e., the subtree of the dual tree T' rooted in v'_i. The weight of the subtree $T'(v'_i)$ is the weight of the superface $\mathcal{F}_i$.

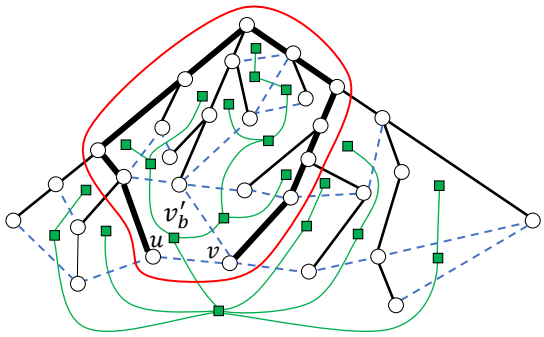

Figure 2 Shown is a planar graph and its path separator as computed by our algorithm. Solid edges are T-edges and the dashed blue edges are the non T-edges. These non-tree edges define the edges of the dual-tree T'. The dual-nodes are depicted as squares and the dual-edges of T' are the curved green edges in the figure. The dual-node v'_b is a balanced dual-node as the total weight of its superface (shown in the figure) is in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. The boundary of the superface of v'_b—i.e., the subtree of dual rooted in v'_b—consists of one non T-edge $e = \{u, v\}$ and a T-path. The path-separator, indicated via thick black edges, is the T-path between u and v.

Our algorithm would not be able to compute the exact weights and instead it would compute a $(1+\epsilon)$-approximation of these weights. Using these approximated weights, we explain how the algorithm chooses a separator path. First, the algorithm attempts to find an (approximate) *balanced* dual-node v'_b such that the weight of its subtree $T'(v'_b)$ is in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. If such a dual-node exists, then the boundary of the corresponding superface—obtained by merging all the faces in the dual subtree of v'_b—is a cycle separator. It is indeed a fundamental cycle of T. See Fig. 2 for an illustration. Otherwise, if no balanced dual-node exists, there must be a dual-node v'_c such that the weight of its subtree $T'(v'_c)$ is larger than $2(1+\epsilon)n/3$ but the weight of each of its descendants sub-trees is less than $n/(3(1+\epsilon))$. We call v'_c a *critical* dual-node.

In the case that we have a *critical* dual-node, we will compute a separator path slightly differently. This will be essentially by mimicking the separator computation of Lipton and Tarjan in the triangulated version of G. In fact, it will suffice to triangulate only the face corresponding to the dual-node v'_c. We note that generally, it is unclear how to efficiently simulate triangulation in distributed manner as this requires simulating many virtual edges. Our construction, however, only uses triangulation implicitly in the analysis. That is, we compute a separator and then show that it is the same as computed by the algorithm of Lipton and Tarjan on the triangulated version of $\mathcal{C}$.

Challenges and Our Approach for Overcoming Them: Our goal is to implement the above algorithm in $\widetilde{O}(D)$ rounds, where D is the diameter of the base graph G. Note that the diameter of $\mathcal{C}$ might be as large as $\Theta(n)$. We face two key challenges: (CI) we need to simulate each dual-node in a distributed manner. Note that a dual node is made of a face, which can be long, and it may interact with other faces through far apart parts of this face. (CII) More severely, we need to implement communications on the dual tree. The nodes and edges of this tree are not real nodes and edges of the graph. Even simulating each node of it is not straightforward, and is the challenge mentioned before. To add insult to injury, the diameter of the dual tree (even in terms of dual-nodes) can be much larger than D. For instance, it is possible that the primal graph has diameter $D = O(1)$ and yet, the diameter of the dual graph is $\Theta(n)$. We next briefly outline the methods we use for overcoming these two challenges.

To deal with challenge (CI), we use the low-congestion shortcuts of [18], as defined in Definition 2, one shortcut for each of the faces. This application is not straightforward

because an important requirement for low-congestion shortcuts is not met in our setting. To use the low-congestion shortcuts of [18], the collection of subsets $S_1, \ldots, S_k$ must be node-disjoint. In our case, however, the S_i sets are the nodes of faces. Hence, these sets are *not node-disjoint*; in fact, a node may belong to several different faces. We bypass this obstacle by transforming the graph G into an auxiliary graph $\widehat{G}$, in which the sets S_i, that correspond to the faces of $\mathcal{C}$, are mapped to node-disjoint connected sets. We then show that the auxiliary graph $\widehat{G}$ can be simulated efficiently in the original graph G.

To deal with challenge (CII), our approach is inspired by a method of [18, Section 5] for aggregating information on a tree with large diameter in planar graphs with low diameter. They used this method for aggregating information on the MST. Though, we need to adjust this method to suit our case. A straightforward combination would suggest a round complexity of $\tilde{O}(D^2)$. This is because, our method for communication inside faces (i.e., dual-nodes) itself takes $\tilde{O}(D)$ rounds, and on top of that, the method of [18, Section 5] for dealing with large-diameter trees needs $\tilde{O}(D)$ iterations of communicating on the dual-nodes. Thus, the naive combination would be $\tilde{O}(D^2)$. We will however be able to put the ideas together in a way that leads to a round complexity of $\tilde{O}(D)$.

Roadmap: In Section 4.2, we present the basic computational tools for efficient distributed communication inside a dual-node and on a dual tree, i.e., dealing with challenges (CI) and (CII) respectively. Then, in Appendix B.2, we present our algorithm for computing a path-separator in an arbitrary (biconnected) induced subgraph $\mathcal{C} \subseteq G$, using the tools explained in Section 4.2. The related analysis appears in Appendix B.3. Some smaller subroutines are deferred to Appendix C.

4.2 Key Tools

We begin by explaining how to preform communication inside nodes of each face, and later how to perform communication on the dual tree.

4.2.1 Tool (I): Communication Inside Dual-Nodes

To simulate communication inside the dual-nodes, we consider two basic tasks.

- **(T1) Face identification**: Assign each face F_i in $\mathcal{C}$ a unique ID, $ID(F_i)$, such that each node knows the IDs of the faces to which it belongs. In addition, for each edge $\{u, v\} \in \mathcal{C}$, the endpoints of this edge should know the two face IDs, $(ID(F_i), ID(F_j))$, to which the edge $\{u, v\}$ belongs.
- **(T2) Low-Congestion Shortcuts for all Faces**: Let S_i denote the nodes of face F_i. Compute an (α, β) low-congestion shortcuts H_i for the S_i sets, for $\alpha, \beta = \widetilde{O}(D)$.

To tackle both of these tasks, we transform the original planar graph G into a virtual planar graph $\widehat{G}$ in which the subsets of nodes belonging to the faces of $\mathcal{C}$ are mapped to *node-disjoint* subsets $\widehat{S}_i$ for which low-congestion shortcuts can be computed. We then show that any r-round algorithm for $\widehat{G}$ can be simulated in G using $2r$ rounds.

The virtual graph $\widehat{G}$ is defined as follows. See Figure 3. First, it contains all edges of $G \setminus \mathcal{C}$. The edges of $\mathcal{C}$ are transformed in the following manner. Consider a node v that belongs to y faces $F_{i_1}, \ldots, F_{i_y}$ in $\mathcal{C}$, ordered in a clockwise manner. Then, v creates y many virtual copies of itself named $v^1, \ldots, v^y$. In $\widehat{G}$, the identifier of the ℓ^{th} copy of v is (ID_v, ℓ). By computing the embedding of the original graph G, each node v knows the clockwise

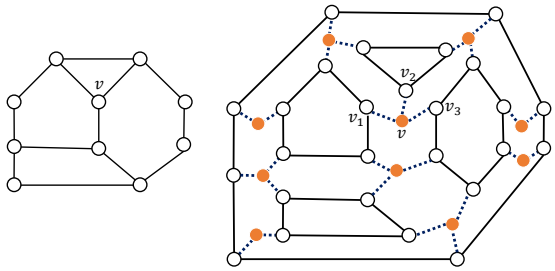

Figure 3 The transformation from G to $\widehat{G}$, which maps faces to node-disjoint connected subsets. The left figure depicts the graph before the transformation, and the right one depicts it after the transformation. The dotted links show the star-edge E_S. Notice that in the graph $\widehat{G}$ after the transformation, if we remove the star-edges, we get a collection of connected components, each corresponding to a face of $\mathcal{C}$.

ordering of all its edges in G. This can be used to deduce the clockwise ordering[3] of its edges in $\mathcal{C}$. The clockwise ordering of the edges of v in $\mathcal{C}$ imposes a local numbering of its faces in $\mathcal{C}$, each two consecutive edges in the clock-wise order define one new face. On each edge $\{u, v\}$, the nodes u and v exchange their local face numberings for that edge. Since a given edge appears in at most two faces, this can be done in 2 rounds. In the graph $\widehat{G}$, we connect v to y copies $v^1, \ldots, v^y$, one per face in $\mathcal{C}$. In addition, for each edge $\{v, u\} \in \mathcal{C}$ belonging to the i^{th} face of v and the j^{th} face of u, we connect v^i to u^j. We use E_S to denote the set of star-edges $\{v, v_i\}$ in $\widehat{G}$.

The graph $\widehat{G}$ is planar. Furthermore, it has the additional benefit that the nodes corresponding to the faces of the $\mathcal{C}$ are now node-disjoint subsets, while still each face induces a connected subgraph. Hence, one can construct low-congestion shortcuts for these node sets in the graph $\widehat{G}$. Notice that $\widehat{G}$ has diameter at most $3D$. This is because every edge $\{u, v\} \in \mathcal{C}$ becomes a path $(u - u^i - v^j - v)$ in $\widehat{G}$, and every edge not in $\mathcal{C}$ is unchanged. Since each edge belongs to two faces, we have:

▶ **Lemma 3** (Simulation of $\widehat{G}$ in G). *Any r-round algorithm $\mathcal{A}$ in $\widehat{G}$ can be implemented in G within at most $2r$ rounds.*

Missing proofs of this section are in Appendix A. From now on, it suffices to consider algorithms in $\widehat{G}$. Since the node faces are the connected components of $\widehat{G} \setminus E_S$, we have:

▶ **Lemma 4.** *The Faces Identification task can be solved in $\widetilde{O}(D)$ rounds.*

Turning to the second task of computing low congestion shortcuts for each face F_i, we have:

▶ **Lemma 5.** *Let $S_1, \ldots, S_N$ be the nodes on faces $F_1, \ldots, F_N$ of the graph $\mathcal{C}$. W.h.p., one can construct in $\widetilde{O}(D)$ rounds, an (α, β) low-congestion shortcut graphs $H_1, \ldots, H_N$ for $\alpha, \beta = O(D \log D)$.*

▶ **Corollary 6.** *One can compute any aggregate function, which has $O(\log n)$-bit size values, in all faces of $\mathcal{C}$ in parallel in $\widetilde{O}(D)$ rounds.*

[3] Note that since the diameter of $\mathcal{C}$ can be larger than D, we cannot afford computing the embedding for $\mathcal{C}$ from scratch, via communicating only inside $\mathcal{C}$.

4.2.2 Tool (II): Communication on the Dual Tree

In tool (I), we described how to perform efficient communication within each face, that is, inside each node of the dual tree. We now explain how to perform efficient communication on the dual tree T' of a spanning tree T of the subgraph $\mathcal{C}$. We mainly need to solve the following two computational tasks in the dual tree T': (D1) Edge Orientation: orienting the dual-edges towards a given dual root, and (D2) Subset-OR: given a rooted dual tree T', and initial binary input values $x(v')$ for each dual-node v', the leader node $\ell(v')$ of the dual node v' should learn the OR of its subtree, that is, the value $y(v') = \vee_{u' \in T'(v')} x(u')$.

An important tool for both of these tasks is a recursive fragment merging process, which we describe next. In Appendix B.1, we then describe how to use this recursive merging to solve the two tasks (D1) and (D2). Our final path-separator algorithm is presented in Appendix B.2.

Recursive Face-Merging Process: To avoid computation in time $O(Diam(T'))$, we employ an idea of [18, Section 5]. It is worth noting that this idea itself is inspired by merges in the style of Boruvka's classical minimum spanning tree algorithm [39].

We have $O(\log n)$ levels of merging faces, where each merge happens along some edge of the dual-tree node T'. The faces involved in each merge correspond to a connected subgraph of the dual tree, which we will call a *fragment* or a *face-fragment*, stressing that it is a merge of some faces. The dual-tree gets partitioned into fragments in a hierarchical fashion, where the fragments of level i are formed by merging fragments of level $i-1$. See Figure 7 for an illustration. Considering that the dual-tree nodes are faces of the primal graph, the fragments of the i^{th}-level are obtained by merging the (sets of) faces corresponding to the fragments of level $i-1$.

The $O(\log n)$ levels of face-fragment merging of the dual tree T' are implemented by using low-congestion shortcuts in G, as described next. For every fragment j in level i, let $S_{i,j}$ be the set of nodes appearing on the faces of fragment j. By using the tools provided in Section 4.2.1 and mainly Lemma 5, we construct low-congestion shortcut subgraphs for each set $S_{i,j}$ (i.e., despite the fact that these sets are not disjoint). Here, we slightly change the definition of the auxiliary graph $\widehat{G}$ that was defined in Section 4.2.1. For simplicity, consider the first level of the face merging process where two faces of $\mathcal{C}$, say F_j and F_k, are merged. Let $e = \{u, v\}$ be a common edge of F_j and F_k. The endpoint u indicates the merging of these faces in the auxiliary graph $\widehat{G}$, by adding an edge between its copies u^j and u^k corresponding to the merged faces F_j and F_k. As a result, the nodes on the faces F_j and F_k now belong to the same connected component in the graph $\widehat{G} \setminus E_S$ (where E_S are the star-edges $\{u, u^j\}$). Since the face identification is done by identifying the connected components of $\widehat{G} \setminus E_S$, this step ensures that F_j and F_k would be identified as one merged face.

Equipped with the low-congestion shortcut subgraphs for each face-fragment (i.e., the node sets $S_{i,j}$), all nodes inside each fragment can communicate in their fragment in parallel, for all fragments in level i, in $\widetilde{O}(D)$ rounds. Hence, the $O(\log n)$ face merging process can be done in $\widetilde{O}(D)$ rounds. A detailed description of the face merging process is described in Appendix B.1.

References

1 A. Aggarwal and R. Anderson. A random nc algorithm for depth first search. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 325–334, 1987.

2 Richard Anderson. A parallel algorithm for the maximal path problem. *Combinatorica*, 7(4):315–326, 1987.

3 RJ Anderson. A parallel algorithm for depth-first search. 1986. In *Extended abstract*, 1986.

4 Baruch Awerbuch. A new distributed depth-first-search algorithm. *Information Processing Letters*, 20(3):147–150, 1985.

5 Philipp Brandes, Marcin Kardas, Marek Klonowski, Dominik Pajak, and Roger Wattenhofer. Approximating the size of a radio network in beeping model. In *International Colloquium on Structural Information and Communication Complexity*, pages 358–373. Springer, 2016.

6 Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, 2014.

7 Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 363–372, 2011.

8 Atish Das Sarma, Danupon Nanongkai, and Gopal Pandurangan. Fast distributed random walks. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 161–170, 2009.

9 Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. Efficient distributed random walks with applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 201–210, 2010.

10 Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 331–340, 2004.

11 Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.

12 Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-nc. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 754–763. ACM, 2016.

13 Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *Proc. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, pages 1150–1162, 2012.

14 J.A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, 1993.

15 M. Ghaffari and F. Kuhn. Distributed minimum cut approximation. In *Proc. of the Int'l Symp. on Dist. Comp. (DISC)*, pages 1–15, 2013.

16 Mohsen Ghaffari. Near-optimal distributed approximation of minimum-weight connected dominating set. In *International Colloquium on Automata, Languages, and Programming*, pages 483–494. Springer, 2014.

17 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks i: Planar embedding. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 29–38, 2016.

18 Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *Proc. of ACM-SIAM Symp. on Disc. Alg. (SODA)*, pages 202–219, 2016.

19 Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 81–90. ACM, 2015.

20 Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Distributed Computing*, pages 197–211. Springer, 2014.

21 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. URL: http://doi.acm.org/10.1145/2933057.2933103, doi:10.1145/2933057.2933103.

22 Andrew V Goldberg, Serge A Plotkin, and Pravin M Vaidya. Sublinear-time parallel algorithms for matching and related problems. In *Foundations of Computer Science, 1988., 29th Annual Symposium on*, pages 174–185. IEEE, 1988.

23 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In *Proceedings of the 2016 ACM Symposium on Principles of Distributed Computing*, pages 451–460. ACM, 2016.

24 Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Near-optimal low-congestion shortcuts on bounded parameter graphs. In *International Symposium on Distributed Computing*, pages 158–172. Springer, 2016.

25 Torben Hagerup. Planar depth-first search in o(\logn) parallel time. *SIAM Journal on Computing*, 19(4):678–704, 1990.

26 Michal Hanckowiak, Michal Karonski, and Alessandro Panconesi. On the distributed complexity of computing maximal matchings. *SIAM Journal on Discrete Mathematics*, 15(1):41–57, 2001.

27 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 489–498, 2016.

28 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 355–364, 2012.

29 Ming-Yang Kao. All graphs have cycle separators and planar directed depth-first search is in dnc. In *Aegean Workshop on Computing*, pages 53–63. Springer, 1988.

30 Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 238–251, 1995.

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

32 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 381–390, 2013.

33 Christoph Lenzen and Boaz Patt-Shamir. Improved distributed steiner forest construction. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, 2014.

34 Richard J Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

35 Kurt Mehlhorn and Peter Sanders. Graph traversal. *Algorithms and Data Structures: The Basic Toolbox*, pages 175–189, 2008.

36 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. of the Symp. on Theory of Comp. (STOC)*, 2014.

37 Danupon Nanongkai, Atish Das Sarma, and Gopal Pandurangan. A tight unconditional lower bound on distributed randomwalk computation. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 257–266, 2011.

38 Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *International Symposium on Distributed Computing*, pages 439–453. Springer, 2014.

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

40 David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

41 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed MST construction. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 253–, 1999.

42 John H Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

43 Gregory E Shannon. A linear-processor algorithm for depth-first search in planar graphs. *Information Processing Letters*, 29(3):119–123, 1988.

44 Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.
45 Justin R Smith. Parallel algorithms for depth-first searches i. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, 1986.

Appendix

A Missing Proofs for Section 4.2.1

Proof of Lemma 3. The edges of $\mathcal{C}$ are transformed into two types of edges in $\widehat{G}$: star-edges between v and its copies, whose simulation requires no real communication in G, and face-edges $\{v^j, u^i\}$. Since each edge $\{u, v\}$ in G simulates the communication of two edges, namely, $\{u^{i_1}, v^{j_1}\}$ and $\{u^{i_2}, v^{j_2}\}$ in $\widehat{G}$, every round r of $\mathcal{A}$ in $\widehat{G}$ can be implemented in G using two rounds. ◀

Proof of Lemma 4. Let $\mathcal{C}$ be the induced subgraph of G, which is biconnected, and for which we are computing a separator path. Let $\widehat{\mathcal{C}}$ be the subgraph of $\widehat{G}$. We employ the $\widetilde{O}(D)$-round connectivity algorithm of [18] in the graph $\widehat{G} \setminus E_S$ but only for the nodes of $\mathcal{C}$. Recall that E_S denotes the star edges in $\widehat{G}$. By using Lemma 3, this algorithm can be simulated in G in $\widetilde{O}(D)$ rounds. Let the ID of each connected component of $\widehat{\mathcal{C}} \setminus E_S$ be the node with maximum ID in the component. Since each connected component of $\widehat{\mathcal{C}} \setminus E_S$ corresponds to a face of $\mathcal{C}$, each node now knows the IDs of its faces, in particular, it knows the face IDs of each of its copies in $\widehat{\mathcal{C}}$. In addition, each node $v \in \mathcal{C}$ also learns the IDs of the two faces $ID(F_i)$ and $ID(F_j)$ of each of its edges $\{u, v\}$ in $\mathcal{C}$. The lemma follows. ◀

Proof of Lemma 5. Consider the algorithm $\mathcal{A}$ of [18] for constructing the low-congestion shortcuts in $\widehat{G}$. By Lemma 3, Algorithm $\mathcal{A}$ can be simulated in G in $\widetilde{O}(D)$ rounds. Let $\widehat{H}_i$ be the $(\alpha, \beta/2)$ low-congestion shortcuts computed for the sets $\widehat{S}_i$ in $\widehat{G}$. Let H_i be obtained from $\widehat{H}_i$ by omitting star-edges $\{v, v^j\}$ and replacing $\{u^i, v^j\}$ edges with $\{u, v\}$ edges. The subgraphs H_i are (α, β) low-congestion shortcuts for the sets S_i in $\mathcal{C}$. ◀

B Missing Details for the Path Separator Algorithm

B.1 Tool (II): Communication on the Dual Tree

So-far, we described how to perform efficient communication within each face, that is, inside each node of the dual tree. We now explain how to perform efficient communication on the dual tree T' of a spanning tree T of the subgraph $\mathcal{C}$. We mainly need to solve the following two computational tasks in the dual tree T':

- **(D1) Edge Orientation Towards the Root**: For a given dual-root, the leader $\ell(e')$ of each dual-edge e' should know the orientation of the edge such that all edges of T' are oriented towards the root.
- **(D2) Subset-OR**: Given a rooted dual tree T', and initial binary input values $x(v')$ to each dual-node v', the leader node $\ell(v')$ of the dual node v' computes or of its subtree as $y(v') = \vee_{u' \in T'(v')} x(u')$.

An important tool for both of these tasks is a recursive fragment merging process, which we describe next. Once we have described this process, we will come back to using it to implement the above two tasks (D1) and (D2).

Recursive Face-Merging Process: To avoid computation in time $O(Diam(T'))$, we employ an idea of [18, Section 5]. It is worth noting that this idea itself is inspired by merges in the style of Boruvka's classical minimum spanning tree algorithm [39].

We have $O(\log n)$ levels of merging faces, where each merge happens along some edge of the dual-tree node T'. The faces involved in each merge correspond to a connected subgraph of the dual tree, which we will call a *fragment* or a *face-fragment*, stressing that it is a merge of some faces. The dual-tree gets partitioned into fragments in a hierarchical fashion, where the fragments of level i are formed by merging fragments of level $i-1$. See Fig. 7 for an illustration. Considering that the dual-tree nodes are faces of the primal graph, the fragments of the level-u are obtained by merging the (sets of) faces corresponding to the fragments of level $i-1$. We now describe this fragment-merging process. The identifier of each dual-node v' is the ID of its face $\phi_F(v')$, which can be computed in $\widetilde{O}(D)$ rounds by solving the Face Identification problem as described in Lemma 4.

Initially, all edges of G are unmarked. Throughout the process, some merges happen along dual edges of T'. Then, we mark the non-tree G-edges that correspond to the dual-edges of T' that have been merged. At the beginning of each level $i \geq 0$, we have the following: a partition $\mathcal{P}_i$ of the faces of $\mathcal{C}$ into face-fragments $\mathcal{F}_{i,1}, \ldots, \mathcal{F}_{i,N_i}$. Let $S_{i,j} \subseteq V$ be the nodes that belong to the faces of $\mathcal{F}_{i,j}$. The ID of the fragment $\mathcal{F}_{i,j}$, denoted by $ID(\mathcal{F}_{i,j})$, is the maximum face ID, $ID(F_k)$, that belongs to the fragment $\mathcal{F}_{i,j}$. Each node in $S_{i,j}$ knows its fragment ID. Each fragment $\mathcal{F}_{i,j}$ has a leader node $\ell_{i,j} \in S_{i,j}$, which is simply the leader $\ell(F_k)$ of the face F_k of maximum ID in $\mathcal{F}_{i,j}$.

Level $i = 0$ of the Merges: We first show that these properties can be achieved for level 0. The initial partition $\mathcal{P}_0$ is trivial, where each face fragment $\mathcal{F}_{0,j} = \{F_j\}$ consists simply of a single face. Using Lemma 4, each node knows its face ID and for each edge $e = \{u, v\}$ in $\mathcal{C}$, its endpoints u and v know the IDs of the two faces this edge belongs to. In addition, by Corollary 6, each face can compute the node of maximum ID to be the leader of that face. We use Lemma 5 to compute low-congestion shortcuts $H_{0,1}, \ldots, H_{0,N}$ for each face. Using the random delay approach of [31], a BFS tree $B_{0,j}$ in $H_{0,j}$ and rooted at the leader can be constructed for all faces $\mathcal{F}_{0,j}$ in $\widetilde{O}(D)$ rounds. This completes the information for level 0. From now on, we assume that at the beginning of level i, we are given this information, and we describe how to proceed to the next level.

Level $i \geq 1$ of the Merges: For each unmarked non-MST edge $e \in G \setminus T$, let $e' = \phi_E(e)$ be the dual edge of e, and let its ID be $(ID(\mathcal{F}_{i,j_1}), ID(\mathcal{F}_{i,j_2}))$ where $\mathcal{F}_{i,j_1}, \mathcal{F}_{i,j_2} \in \mathcal{P}_i$. The leader $\ell(e') \in V$ sends this edge to the leaders ℓ_{i,j_1} and ℓ_{i,j_2} of the face-fragments $\mathcal{F}_{i,j_1}$ and F_{i,j_2}, using the BFS trees B_{i,j_1}, B_{i,j_2} respectively. We upcast only one such dual-edge on each such tree $B_{i,j}$, breaking ties arbitrarily. Note that the leader $\ell(e')$ belongs to the sets S_{i,j_1} and S_{i,j_2} and hence belongs to both BFS trees B_{i,j_1} and B_{i,j_2}. Since these subtrees are subgraphs of the low-congestion shortcuts $H_{i,j}$'s, using the random-delay approach [31], within $\widetilde{O}(D)$ rounds, the leader $\ell_{i,j}$ of each face-fragment $\mathcal{F}_{i,j}$ selects one outgoing dual-edge in T'. This elected dual-edge is downcast on the BFS trees $B_{i,j}$, so that their leaders (who suggested these edges) would know if their edge-suggestion is accepted.

At the same time, the leader $\ell(\mathcal{F}_{i,j})$ of face-fragment $\mathcal{F}_{i,j} \in \mathcal{P}_j$ tosses a coin and downcasts the coin toss result to all its nodes on $B_{i,j}$. The leaders $\ell(e')$ of the elected dual edges $e' = (\mathcal{F}_{i,j}, \mathcal{F}_{i,j'})$—i.e., whose edges have been elected by the face leader $\ell(F_{i,j})$—acknowledge the leader $\ell(\mathcal{F}_{i,j})$ only if the following condition is satisfied: $\ell(\mathcal{F}_{i,j})$ has a head coin and $\ell(\mathcal{F}_{i,j'})$ has a tail coin. This acknowledgement means that the related edge is accepted for a merge. Notice that because of these coins, the merge are star shapes, centered at face-fragments who draw a head coin.

Finally, we want each node to know the new leader of the merged fragment along with the ID of the new fragment. To do that the leaders of the elected dual-edges $(\mathcal{F}_{i,j}, \mathcal{F}_{i,j'})$ send on the BFS tree $B_{i,j}$ of the head fragment, say $\mathcal{F}_{i,j}$. The nodes on $B_{i,j}$ upcast the information (ID, leader) of the fragment with the maximum ID. The leader and the ID of

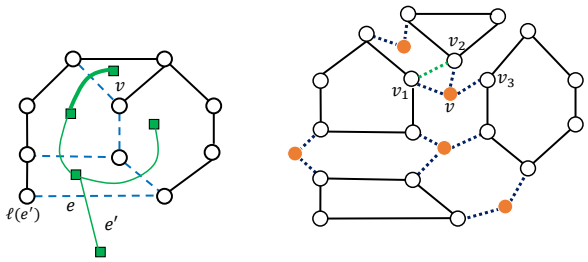

Figure 4 The transformed graph $\widehat{G}_i$ after merging of faces. In the figure, red edges are tree edges T and green edges are the dual tree edges T'. The thick dual edge has been merged at level i. In the transformed graph $\widehat{G}_i$, the node v adds an edge (shown in green) between the two v copies corresponding to the two merged faces.

the merged fragment is downcast on the tree and then passed through the leaders of the merged dual-edges on the BFS trees $B_{i,j'}$ of the merged face-fragments. Finally, the leaders of the merged dual-edges mark their corresponding primal edges in G as merged.

At that point, all nodes know their fragment-ID in the new partition $\mathcal{P}_{i+1}$. Also, for each dual-edge $\{u, v\}$, the leader of the edge knows the new ID's of the two face-fragments in $\mathcal{P}_{i+1}$ of this edge. To compute the low-congestion shortcuts for the face-fragments of partition $\mathcal{P}_{i+1}$, we again consider the virtual graph $\widehat{G}_i$ but with slight changes: we construct $\widehat{G}_i$ as before, i.e., containing all edges of $G \setminus \mathcal{C}$, and for every $v \in \mathcal{C}$, creating copies $v^1, \ldots, v^y$ of node v for each of its faces in $\mathcal{C}$, all these copies are connected to v via a star. Only this time, for each $v \in \mathcal{C}$, we also add edges between copies of v^i that belong to the same merged face. That is, $\widehat{G}_i$ consists of all edges of $\widehat{G}_{i-1}$ with additional edges connecting the copies of v that correspond to faces that have been merged in level i. See Figure 4, where on the right side, the one dashed green edge depicts such a connection between copies of the same node, effectively merging the two components.

Let E_S^i be the star edges between each v to all its copies and let $\widehat{\mathcal{C}}_i$ be the edges of $\widehat{G}_i$ restricted to the nodes of $\mathcal{C}$. By adding the internal edges between v's copies, we get that the connected components $\widehat{\mathcal{C}}_i \setminus E_S^i$ are exactly the fragments of the partition $\mathcal{P}_{i+1}$. We can then compute low-congestion subgraphs $\widehat{H}_{i+1,j}$ for each connected component in $\widehat{\mathcal{C}}_i \setminus E_S^i$ and this translates to low-congestion shortcuts in the original graph. That is because each edge belongs to at most two faces, see Lemma 5. Finally, we compute the BFS tree $B_{i+1,j} \subseteq H_{i+1,j}$ rooted at the leader of each face-fragment. This completes the description of the face-merging process.

Since in each phase i, the number of faces is reduced by constant factor, within $L = O(\log n)$ iterations, we are left with one merged face. Next, we explain how we use this $O(\log n)$ face-fragment merging process to solve tasks (D1) and (D2).

Task (D1), Orientation of the Dual Tree T': We proceed by describing how to orient the dual tree towards an arbitrary root. Let F_* be the face with the maximum ID. We then root T' at the dual-node $v'_* = \phi_F(F_*)$.

We employ the recursive procedure of merging dual-tree fragments, as explained above. Consider the very last level L in which we have one face-fragment formed by a star-merge of the level $(L-1)$ face-fragments. Using the low-congestion shortcuts within each of these $(L-1)$ face-fragments, the leader $\ell(v'_*)$ belonging to the face of maximum ID can notify that (its maximality) to its entire face-fragment. Consider a merged dual-edge which is merged in

this transition from level $(L-1)$ fragments to level L fragments. The leader of this merged dual-edge, who is in the face-fragment that contains the root, updates the other fragment to which it belongs and also orient its dual edge accordingly. The dual-node which is the other endpoint of the merged dual-edge is the root of its own $(L-1)$ fragment. With one more application of this process (as the star shape of merges has diameter ≤ 2), all nodes of the $(L-1)$ fragments know the dual-root node. Moreover, the dual-edges of T' of this last star-merge are now oriented towards the $(L-1)$ face-fragment that contains the dual-root. The other endpoints of these dual star-edges are the dual-roots of their fragments.

Then, we remove the T'-edges between these level $(L-1)$ fragments. Then, we recurse, going one level deeper, each time orienting the edges merged in that level. Since in each level i, we have low-congestion shortcuts for each of the fragments, we can identify their dual-roots and orient the dual-edges accordingly, in a total of $O(D \log D \log^2 n)$ rounds.

Task (D2), Computing the Subset-OR: Recall that in the Subset-OR problem, we are given a rooted dual tree T', and initial binary input values $x(v')$ to each dual-node v'. The goal is to have each leader node $\ell(v')$ compute the value $y(v') = \vee_{u' \in T'(v')} x(u')$. Again our approach is inspired by the solution for of the Subset-SUM problem that has been considered in [18].

The first step, which is done for simplicity, is to redefine the $O(\log n)$ face-fragmentation process such that the level-i face-fragment is made of merging a level-$(i-1)$ face-fragment and some of its level-$(i-1)$ fragment children. To do that, in iteration i, each face-fragment suggests the dual-edge to its parent fragment for the merge and then, the *head* face-fragment accepts the merge suggestions from all its children that are *tail* face-fragments. This is implemented as follows. The leader $\ell(e')$ of each dual edge e' knows the IDs $ID(F_j)$ and $ID(F_{j'})$ of the two faces to which that edge belongs. In addition, it knows the orientation of the edge, say, $F_{j'}$ its the parent of F_j in T'. It also knows the $(i-1)$ face-fragments of these two faces, let is be $F_j \in \mathcal{F}_{i-1,k}$ and $F_{j'} \in \mathcal{F}_{i-1,k'}$ where $\mathcal{F}_{i-1,k}, \mathcal{F}_{i-1,k'} \in \mathcal{P}_{i-1}$. Now it offers the edge e' only to the face-fragment $\mathcal{F}_{i-1,k}$ that contains the parent face F_j. Again, we can easily see that after $L = O(\log n)$ levels, w.h.p., we reach a single face fragments.

With these redefined fragment merges, we are ready to compute the subset-OR for each dual-node. We solve it by doing two types of recursion: an outer top-down recursion and an internal bottom-up recursion. Specifically, as we soon explain, every step of the top-down recursion is solved using a a bottom-up recursion. In level-L, we have one face-fragment made of a star merge of a level $(L-1)$ face-fragment and its level $(L-1)$ children fragment. We first want to be able to remove the dual-edges of this last merge, thus allowing the top-down recursion to go one level deeper and focus on only level $(L-1)$ fragments.

To do that, we make the root of each child level $(L-1)$ fragment learn the value of its fragment. Then, this root passes the value to the parent fragment. Once that is done, we can remove the edges merged in level $L-1$, and recurse in the top-down recursion. Specifically, for each dual-edge that connects the parent fragment to a child fragment, the leader of this dual-edge belongs to both of the fragments. Thus, this leader can pass this information to the parent fragment. Once, the leader of the dual-edge of each child fragment passes that subset-OR of its subtree to the parent fragment, we can remove these dual-edges and recurse top-down in each of the $(L-1)$-fragments, all in parallel.

To make the root of the child $L-1$ fragment learn the value of its fragment, we now do the second recursion, this time a bottom-up recursion, from level 0 to $L-1$. We iterate over the $L-1$ levels, starting with the first level, and keep the invariant that the root of each face-fragment knows the OR value of its fragment. In level-1 fragment, each fragment is a face F_j and the leader of the dual-node $v'_j = \phi_F(F_j)$ knows the value $x(v'_j)$. In level $i \geq 1$, each level-i face-fragment is formed by merging one level-$(i-1)$ face-fragment with some of its level-$(i-1)$ fragment children. The leader of each of these dual-children knows the OR

of its own face-fragment. Using the BFS trees $B_{i-1,j}$ of each face-fragment $\mathcal{F}_{i-1,j}$ and the random-delay approach, in $\widetilde{O}(D)$ rounds all nodes in their fragment knows the OR value. This particularly includes the leader of the dual-edge between the fragment to its parent. This leader belongs to both of the fragments. (Note the fragment children might have the same leader node and in this case it suffices that this node will send the OR of all these fragment children on the BFS tree of the parent fragment.) All the values get OR-ed on the BFS tree of the parent fragment, and the leader node of the parent fragment adds also its own value (i.e., the OR of its face). After $O(\log n)$ such steps, the leader of each $(L-1)$ face-fragment knows the OR of its subtree in T'. At that point, we can ignore the dual-edges connecting the $(L-1)$ face-fragments, and recurse. Overall, we have $O(\log^2 n)$ recursion levels: $O(\log n)$ recursion levels of the top-down recursion, each level is solved by doing $O(\log n)$-levels of the bottom-up recursion. In each such recursion level, we spend $\widetilde{O}(D)$ rounds of communication within face-fragments and between neighboring face-fragments. Hence, overall solving the Subset-OR task takes $\widetilde{O}(D)$ rounds.

B.2 Algorithm for Computing the Separator Path in Biconnected Subgraph $\mathcal{C}$

In this section, we present Algorithm ComputePathSep. The analysis appears in Appendix B.3. We first present a concise description of the algorithm, in the following algorithmic box. Then, we discuss the details of each of the steps.

Algorithm ComputePathSep

Input: A n-node biconnected induced subgraph $\mathcal{C}$ of a planar graph G with diameter D, approximation parameter $\epsilon \in (0, 1/2)$.
Output: A separator path P in $\mathcal{C}$, so that each component of $C \setminus P$ has size at most $2(1+\epsilon)n/3$.

- **Step (S1): Computing the Dual Tree T'**
  - Compute an MST T in $\mathcal{C}$. Non T-edges of $\mathcal{C}$ correspond to the edges of dual-tree T'.
- **Step (S2): Orienting the Dual Tree T' Towards a Root**
  - This step is done via a recursive face-fragment merging process.
- **Step (S3): Computing the Weights of the Dual Nodes in T'**
  - For each $i \in \{1, \ldots, \log_{1+\epsilon} n\}$, we have $N_\epsilon = O_\epsilon(\log n)$ *experiments*, as follows:
    - ▷ Sample each of the nodes of $\mathcal{C}$ with probability $1/(1+\epsilon)^i$.
    - ▷ Use Subet-OR to inform each dual-node if there is a marked node in its subtree.
  - Using these experiments, dual-nodes deduce a $(1+\epsilon)$ approximation of their weight.
  - Detect a balanced dual-node, i.e., a dual-node with weight in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$.
  - If there is no balanced dual-node, detect a critical dual-node, that is, a dual-node with weight at least $2(1+\epsilon)n/3$ but each of its children has weight less than $n/(3(1+\epsilon))$.
- **Step (S4): Marking the Separator Path**
  - For balanced dual node: mark the tree path connecting the boundary of its superface.
  - For critical dual-node: mark a tree path by simulating Lipton-Tarjan on its superface.

Step (S1-S2): The algorithm begins by computing a spanning tree T in $\mathcal{C}$ using the MST algorithm of [18]. This defines a dual tree T' by taking the dual edges corresponding to the non T-edges, as described in Section 2. We then use the dual-tree orientation procedure and

orient all dual edges of T' towards the dual-node root of maximum ID.

Step (S3): Computing the Weight of Each Dual Node: For a dual-node v', we consider its superface obtained by merging all faces in its subtree $T'(v')$. The *weight* of v' is defined to be the number of nodes inside or on the boundary of this superface. The main challenge in computing these weights is to avoid double counting, which can possibly happen because of each node appearing in many faces of a subtree $T'(v')$. Our approach uses a simple and by-now standard linear *sketching* type of idea, see [5,15,18]. In particular, we next describe a procedure that lets each leader $\ell(v')$ obtain an $(1+\epsilon)$-approximation of its weight.

The procedure consists of $K_\epsilon = O(\log_{1+\epsilon} n)$ phases, each phase consists of $N_\epsilon = O_\epsilon(\log n)$ *experiments*. Set $q_1 = e^{-(1+\epsilon)}$, $q_2 = e^{-1/(1+\epsilon)}$, $q_3 = (q_2 - q_1)/(q_2 + q_1)$ and $N_\epsilon = 3e^3 \cdot c \cdot \log n/(q_3)^2$.

In the i^{th} phase for $i \in \{K_\epsilon, \ldots, 1\}$, we do the following for N_ϵ experiments. Each node v in $\mathcal{C}$ marks itself with probability $p_i = 1/(1+\epsilon)^i$ and 0 otherwise. That is, we let $z_i(v) = 1$ with probability p_i and 0 otherwise. Using Corollary 6, the leader of each dual-node $\ell(v')$ can compute the initial value $x(v') = \vee_{v \in \phi_F^{-1}(v')} z_i(v)$. This is simply the OR of all $z_i(v)$ values of the nodes v belonging to the face $\phi_F^{-1}(v')$. We then apply the Subset-OR procedure on the x_i values. This allows the leader of each dual-node v' count the number of experiments $N_i(v')$, among the N_ϵ many experiments of phase i, in which the Subset-OR of its subtree $T'(v')$ is zero (or non-zero). In each of the N_ϵ experiments, the z_i values are resampled using fresh and independent randomness, with probability p_i.

For a given dual-node v', let $i^* \in \{K_\epsilon, \ldots, 1\}$ be the first phase number for which $N_{i^*}(v') < (1+p_3) \cdot p_1 \cdot N_\epsilon$. Then, the estimate weight of v' is $(1+\epsilon)^{i^*}$. In Lemma 8 we prove that this indeed provides a $(1+\varepsilon)$-approximation of the weight of the subtree of v', with high probability.

At the end of Step (S3), the leader of each dual node knows (an approximation of) the weight of its superface and also the weights of its children in the dual-tree. A dual-node whose approximated weight is in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$ is called a *balanced dual-node*. A dual-node whose approximated weight is larger than $2(1+\epsilon)n/3$ but the approximated weight of each its children is smaller than $n/(3(1+\epsilon))$ is called a *critical dual node*. Note that since we compute only $(1+\epsilon)$-approximation for the weight of the superface of each dual-node, our path-separator might break the graph into components of size at most $2(1+\epsilon)n/3$, instead of $2n/3$, for some small constant ϵ, e.g., $\varepsilon = 0.01$. This is sufficient for our purposes as we still get a constant factor reduction in the component's size, thus the recursion has depth of $O(\log n)$.

Step (S4): Marking the Nodes on a Separator Path: First, we check whether there is a balanced dual node or not. For that, we let all balanced nodes upcast the identity of their face on the shortcut of $\mathcal{C}$, in $\widetilde{O}(D)$ rounds. In case there is at least one balanced dual-node, we select one such node v'_b. Note that this balanced dual-node v'_b cannot be the root of T'. This is because the boundary of the superface of the dual-root is the boundary of the infinite face and hence it contains all nodes inside it. Let $\ell(e')$ be the leader of the dual-edge that connects the face $\phi_F(v'_b)$ to its parent in T'. In addition, let $e = \phi_E^{-1}(e') = \{u, v\}$ be the edge in $\mathcal{C}$ corresponding to e'. We then mark the $u - v$ path in T using the procedure described in Appendix C.2.

Alternatively, suppose that there is no balanced node. In that case, there must be exactly one critical dual node, let us denote it v'_c. Again, all nodes in $\mathcal{C}$ can know the face identifier of the critical dual-node in $\widetilde{O}(D)$ rounds. If the critical node has no children in the dual-tree, then letting $e = \{x, y\}$ be the non T-edge corresponding to the dual edge that connects v'_c to its parent in T', we mark the T-path between x and y as our separator.

From now on, we assume that v'_c has at least one child in T'. Let $e = (u, v)$ be a non T-edge on the face $\phi_F^{-1}(v'_c)$, where v appears before u when traversing the boundary of this face in a clock-wise manner. We now use the procedure explained in Appendix C.3 to label the nodes $x_{i,1}, \ldots, x_{i,k}$ on the face $\phi_F^{-1}(v'_c)$ in increasing numbers from u to v (the nodes u and v are not in $x_{i,1}, \ldots, x_{i,k}$). For every $x_{i,j}$ for $j \geq 2$, consider the fundamental cycle obtained by adding the edge $\{u, x_{i,j}\}$ to T and define the weight W_j by the number of nodes inside or on the boundary of that cycle. Our goal is compute the first node $x_{i,j}$ in the ordering, for which the weight W_j becomes in the desired range of $[n/3, 2n/3]$. The path-separator in a such case is simply the T-path between u and $x_{i,j}$.

We now want the leader node of the dual-node v'_c to compute this node $x_{i,j}$. Instead of computing the exact weights W_j, we will compute approximated weights $\widehat{W}_j$. For every $j \geq 2$, define $\widehat{w}(x_{i,j})$ as follows: set $\widehat{w}(x_{i,j}) = 1$ if $e_{i,j} = \{x_{i,j-1}, x_j\}$ is a T-edge. Otherwise, let $\widehat{w}(x_{i,j})$ be the approximated weight of the dual-child $v'_{i,j}$ connected to v'_c via the dual-edge $\phi_E(e_{i,j})$ minus one.[4] The approximated weight $\widehat{W}_j = \sum_{\ell=1}^{j} \widehat{w}(x_{i,\ell})$ is the sum of the $\widehat{w}$ values. Let p be the *smallest* index satisfying that $\widehat{W}_p \in [n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. We will use the node $x_{i,p}$ to define our separator path.

Next, we explain how the leader $\ell(v'_c)$ can compute this value p, i.e., realizing that $\widehat{W}_p$ is the index for which the sum of approximated weights is in the right range. The index p is computed in a binary-search like fashion. At any given iteration, we are given an index $\ell \in \{1, \ldots, k\}$ and we want to compute $\widehat{W}_\ell$. First, using Corollary 6, the leader $\ell(v'_c)$ can compute the number of T-edges on the face path $[x_{i,1}, \ldots, x_{i,\ell}]$. By Step (S3), each dual-node knows its approximated weighted sum. We then let only the dual children in the range $[x_{i,1}, \ldots, x_{i,\ell}]$ to send this sum to their dual-node parent v'_c. More specifically, only the dual-children $v'_{i,j}$ connected to v'_c via the dual edge $\phi_E(e_{i,j})$, where $e_{i,j} = \{x_{i,j-1}, x_j\}$, for $j \in \{2, \ldots, \ell\}$, upcast their approximated weight on the BFS tree constructed in the low-congestion shortcut subgraph of the nodes of the face of v'_c. Note that the leaders of the dual-edges connecting the children to v'_c know their local ordering on the boundary of $\phi_F(v'_c)$ and hence they can know if they are active in the given iteration of this binary-search, or not. This allows the leader node to compute $\widehat{W}_\ell$ in $\widetilde{O}(D)$ rounds. Within $O(\log n)$ such binary-search iterations, the leader $\ell(v'_c)$ can find the first breaking point p for which $\widehat{W}_p$ is in the desired range. We then use the procedure described in Appendix C.2 to mark path connecting $x_{i,p}$ and u in T. This is the output separator path.

B.3 Analysis of the Separator Path Algorithm

B.3.1 Analysis of Step (S3)

We argue that in Step (S3), the leader of each dual-node learns an approximation for the weight of its face. Consider a dual-node v' and let k be the weight of its subtree in the dual-tree, i.e., the number of primal nodes inside or on the boundary of its superface. Consider the i^{th} phase were each node marks itself with probability $1/r_i$ for $r_i = (1+\epsilon)^i$. We show the following auxiliary claims.

▶ Claim 7. (1) If $r_i < k/(1+\epsilon)$, then $N_i(v') < (1+p_3)p_1 \cdot N_\epsilon$ with high probability.
(2) If $r_i > (1+\epsilon)k$, then $N_i(v') > (1+p_3)p_1 \cdot N_\epsilon$ with high probability.

Proof. We start with claim (1) In a given step of phase i, the probability that no node among the k is marked is at most $(1-(1+\epsilon)/k)^k \leq e^{-(1+\epsilon)} = q_1$. Hence, the expected

[4] This minus one is important to avoid double counting due to nodes appearing on the edges $e_{i,j}$.

number of trials in which no node is marked is at least $N_\epsilon \cdot q_1$. Item (1) follows by using the Chernoff bound.

We now consider claim (2). In a given step of phase i, the probability that no node among the k is marked is at least $(1-1/((1+\epsilon)k))^k \geq q_2 \cdot (1-1/((1+\epsilon)k)) \geq q_2/2$. Hence, in expectation, there are at least $N_\epsilon \cdot q_2/2$ steps in which no node is marked in the superface of v'. The claim again follows by applying the Chernoff bound. ◀

Now, we use this auxiliary claim to prove the approximation guarantee, as follows.

▶ **Lemma 8.** *An $(1+\epsilon')$-approximation of the weight of the superface of $T'(v')$ can be computed in $\widetilde{O}(D)$ rounds for $\epsilon' = \sqrt{1+\epsilon} - 1$.*

Proof. Recall that the estimate for the weight of v' is given by the first index i^* for which the number of steps in which no node is marked on its superface is at most $N_{i^*}(v') < (1+p_3)p_1 \cdot N_\epsilon$. Thus, $N_{i^*+1}(v') > (1+p_3)p_1 \cdot N_\epsilon$. By Claim 7(1), $r_{i^*} < k/(1+\epsilon)$ and by Claim 7(2), $r_{i^*+1} < (1+\epsilon)k$. We get that r_{i^*} is an $(1+\epsilon)^2 = (1+\epsilon')$ approximation for k. The lemma follows. ◀

B.3.2 Analysis of Step (S4)

We proceed by showing that the path computed in Step (S4) is indeed a separator path. The analysis for the two cases with a separator based on a balanced-node and with a separator based on a critical dual-node are presented in Lemma 9 and Lemma 10, respectively.

By Lemma 8, each dual node computes in Step (S3) a $(1+\epsilon)$-approximation of its weight. Hence, if there is a dual node with weight in $[n/3, 2n/3]$, there exists a dual-node with approximated weight in $[n/(3(1+\epsilon)), 2n(1+\epsilon)/3]$. We call this node a balanced dual node. We now show that the boundary of this dual-node's superface is a fundamental cycle, that is, with the exception of one edge, all edges on the boundary cycle of its superface are T-edges.

▶ **Lemma 9.** *Assume that there is a balanced dual-node v' in T'. Then, the boundary of the superface obtained by merging all faces in $T'(v')$ consists of only one non T-edge.*

Proof. We prove the claim by induction on the the depth of the dual-node in T'. For the base case consider a leaf dual-node ℓ' in T'. This dual-node is connected to T' via one dual-edge to its parent in T', hence all remaining edges on the boundary of its face $\phi_F(\ell')$ are BFS edges.

Assume that the claim holds to subtrees up to depth d, and consider a dual-node v' whose subtree $T'(v')$ is of depth $d+1$. Let $v'_1, \ldots, v'_k$ be the children of v' in T'. By induction assumption, the boundary of the superface obtained by merging the faces in $T'(v'_i)$ consists of only one non-BFS edge. Hence, this is exactly the edge e such that the dual edge $\phi_E(e)$ connects v'_i to its parent v' in T' (recall that the balanced dual-node cannot be the root of T'). When creating the merged face of $T'(v')$ these non-BFS edges of the children v'_i become internal and do not appear on the boundary of the merged superface. All the edges of the parent face v' that are not in common with its children are BFS edges except for the edge that corresponds to the dual-edge which connects v' to its parent in T' (unless v' is the root). The claim follows. ◀

Since the boundary of the superface of the balanced dual-node is a cycle separator, by Lemma 9, it follows that the dual-edge that connects v'_b to its parent in the dual-tree defines a fundamental cycle separator in T. As our algorithm marks the tree path of the edge corresponding to this dual-edge, the correctness follows.

We now turn to the complimentary case that there is no balanced dual-node with approximated weight in $[n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. In such a case, there is a unique *critical* dual-node whose approximated weight is larger than $2(1+\epsilon)n/3$ and the weight of each of its children is smaller than $n/(3(1+\epsilon))$, this is exactly our definition for a critical dual-node. We show:

▶ **Lemma 10.** *The path computed for a critical dual-node v'_c is a separator for $\mathcal{C}$.*

Proof. First, consider the case where v'_c has no children in the dual tree T'. In such a case, the boundary of v'_c is a separating cycle. Since v'_c has one parent and not children, there is only one non T-edge on this cycle. Thus, taking the tree path between the only non T-edge on this boundary is a separating path.

From now on, consider the case where v'_c has at least one child in the dual tree T'. Consider the non T-edge $\{u, v\}$ on the boundary of v'_c that we use as a reference. Recall that v is appearing right before u in a clockwise traversal on the boundary of v'_c. See Figure 5 for an illustration. Recall that $x_{i_1}, \ldots, x_{i_k}$ are the nodes on the boundary of v'_c ordered in clockwise ordering from u to v. For $j \geq 2$, let the weight (resp., approximated weight) of $x_{i,j}$ be $w(x_{i,j}) = 1$ (resp., $\widehat{w}(x_{i,j}) = 1$) if $e_{i,j} = (x_{i,j-1}, x_j)$ is a T-edge. Otherwise, let $w(x_{i,j})$ (resp., $\widehat{w}(x_{i,j})$) be the weight (resp., approximated weight computed in Step (S3)) of the dual-child $v'_{i,j}$ connected to v'_c via the dual-edge $\phi_E(e_{i,j})$ minus one.

Let $W_j = \sum_{\ell=1}^{j} w(x_{i,\ell})$ and $\widehat{W}_j = \sum_{\ell=1}^{j} \widehat{w}(x_{i_\ell})$ and let p be the first index satisfying that $\widehat{W}_p \in [n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. The separator given by the algorithm is then the T-path between u and $x_{i,p}$.

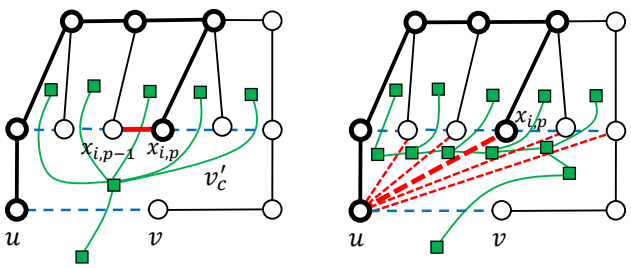

Figure 5 Illustration of the critical node and the selection of the path separator (shown as nodes with thick boundary). Solid edges are T-edges and dashed edges are non T-edges. The curved edges are the dual edges of T'. Left: the original $\mathcal{C}$ graph. The node $x_{i,p}$ is the first node in the ordering for which $\widehat{W}_p \in [n/(3(1+\epsilon)), 2(1+\epsilon)n/3]$. Right: the transformed triangulated graph. The tick triangulated edge $\{u, x_{i,p}\}$ defines a fundamental separator cycle in T. The path separator is the the T-path between u and $x_{i,p}$.

Now consider a triangulated version of $\mathcal{C}$ where the node u is used to triangulate the face of v'_c by adding the edges $\widehat{e}_j = \{u, x_{i_j}\}$ for every j.

We will claim that the path separator that we compute is defined by a fundamental cycle in the triangulated graph, in a similar way to the one computed by Lipton and Tarjan for the triangulated version of $\mathcal{C}$. By triangulating the graph $\mathcal{C}$, the maximum degree of the dual tree of the triangulated graph is 3 and hence it has a triangulated edge that breaks the $\mathcal{C}$ into parts of weights in $[n/3, 2n/3]$. Since there is exactly one critical dual-node, this triangulated edge

must be one of the u-edges, say, $\widehat{e}_j = \{u, x_{i_j}\}$ for $j \in \{1, \ldots, k\}$ that we added to triangulate the face of dual-node v'_c. Thus, $W_j \in [n/3, 2n/3]$ and $\widehat{W}_j \in [n/(3(1+\epsilon)), 2n(1+\epsilon)/3]$. Since p is the first index satisfying that $\widehat{W}_p \in [n/(3(1+\epsilon)), 2n(1+\epsilon)/3]$, we get that the triangulated edge $\widehat{e}_p$ defines the fundamental separator cycle in the tree T. Yielding that the tree path connecting the triangulated point u and the endpoint x_{i_p} is a path separator. See Figure 5 for an illustration. ◀

C Auxiliary Procedures

C.1 Identifying Cut Nodes and the DFS Skeleton of Biconnected Components

Here, we explain an $\tilde{O}(D)$-round procedure that we use for identifying the cut nodes and bi-connected components. In particular, let us consider an induced connected subgraph $\mathcal{C}$ of the base graph G, and furthermore, suppose we are given a node $r \in \mathcal{C}$ who shall be the DFS root of the DFS of $\mathcal{C}$. See Figure 6 for an illustration. We have two goals:

(1) We would like to identify *cut nodes* of $\mathcal{C}$, i.e., nodes $v \in \mathcal{C}$ whose removal disconnects the graph $\mathcal{C}$, leaving a disconnected subgraph $\mathcal{C} \setminus \{v\}$. This also identifies the bi-connected components $\mathcal{C}'_1$, $\mathcal{C}'_2$, …, $\mathcal{C}'_N$ of $\mathcal{C}$, that are the maximal subgraphs of $\mathcal{C}$ that are bi-connected, meaning that removing any single node leaves them connected.

(2) We would like that for each bi-connected component $\mathcal{C}'_i$ of $\mathcal{C}$, we identify the DFS root r_i of $\mathcal{C}'_i$. For the bi-connected component $\mathcal{C}'_i$ that includes r, this is node r itself. For any other biconnected component $\mathcal{C}'_j$, this root r_j is the first cut node on the shortest path to the overall root r of $\mathcal{C}$.

Identifying Cut Nodes: To identify the cut-nodes, we use the transformation from G to the virtual graph $\widehat{G}$, which we explained in Section 4.2.1. Using explained Lemma 4, in $\widetilde{O}(D)$ rounds, we get a component identification of $\widehat{G} \setminus E_S$. Recall that E_S denotes the star edges between each real node v and its copy nodes v^1, v^2, …, v^y. Once we have this connected component identification, a node $v \in \mathcal{C}$ is a cut node of $\mathcal{C}$ if and only if at least two of its copy nodes v^i and v^j for $j \neq i$ receive the same component identifier. See Figure 8, which depicts an example of a cut node and the faces it is involved in. If that happens, it means that these two copy nodes v^i and v^j are in the same connected component in $\widehat{G} \setminus E_S$. This implies that v is a cut node of $\mathcal{C}$. That is because of the following: consider the cycle defined by traversing from v to v^i, then taking the $\widehat{G} \setminus E_S$ path from v^i to v^j, and then going from v^j back to v. Notice that inside and outside of this cycle are non-empty and each of them includes some nodes of $\mathcal{C}$, as otherwise we would have $v^i = v^j$. This cycle separates a non-empty part of the graph $\mathcal{C}$ from the rest of it. Hence, removing v would disconnect $\mathcal{C}$, which means node v is a cut node.

Identifying DFS roots of Biconnected Components: First, we use the identification of cut nodes as done above to determine the bi-connected components $\mathcal{C}'_1$, $\mathcal{C}'_2$, …, $\mathcal{C}'_N$ of $\mathcal{C}$. For each cut node node v, consider two copies of it v^i and v_j who receive the same connected component identifier. This indicates that in the clockwise traversal around v, the $\mathcal{C}$-edges incident on v between v^i and v^j would become disconnected from the other $\mathcal{C}$-edges incident on v, if we remove v. Hence, these two different categories of $\mathcal{C}$-edges belong to different biconnected components. See Figure 8 for an example, where three biconnected components around a cut node are depicted. By considering all of its copies who receive the same component identifiers, and examining them in a clockwise order, each cut node node v can determine which of its edges belong to the same biconnected component. Once this is done, the cut node v splits itself into many independent versions, one for each of the

biconnected component, with no connection between the versions. This effectively separates different biconnected components from each other. Then, we use the connected component identification procedure of [18] to assign identifiers to the biconnected components. This allows each node v to learn the identifiers of each of the biconnected components that includes v.

Once we have identified the biconnected components $\mathcal{C}'_1, \mathcal{C}'_2, \ldots, \mathcal{C}'_N$ of $\mathcal{C}$, we can identify the respective DFS roots. In doing so, we consider each biconnected component $\mathcal{C}'_i$ as an initial fragment, and we employ the recursive *fragment-merging* process explained in Appendix B.1. In a manner virtually the same as what we did for task (D1) in Appendix B.1, we can identify the root of each fragment, with regard to the overall root r. This means each biconnected $\mathcal{C}'_i$ component learns its own DFS root r_i.

C.2 Marking a Tree Path

Given a tree T, which potentially has large diameter, and two nodes u and v we describe how to mark the $u-v$ path in T within $O(D)$ rounds. For simplicity we root the tree T at u (the node of higher ID, w.l.o.g.). We consider the recursive tree fragments merging where each parent merges subset of its children in T. In our case, we do the merging only till we get to the first level i^* where u and v are in the same fragment. This can be done by letting u and v upcast their ID to the leader of their fragment using the low-congestion shortcuts. Note at this last level i^*, there is an edge $\{w, z\}$ connecting the fragment of v with the parent fragment that contains the root u. We mark the edge $\{w, z\}$ and recurse – by computing the $u-w$ path in the $(i^*-1)^{th}$ fragment of u and the $z-v$ path in the $(i^*-1)^{th}$ fragment of v, going one level lower in the recursion. Hence, after $O(\log n)$ steps, we marked all nodes of the path.

C.3 Clockwise Labeling of Nodes on the Boundary of a Face

Consider a face F_i and a given node u. We now describe how to label the nodes of the face in clockwise ordering with respect to u.

First, we use the root orientation procedure of [18] to orient the boundary edges of the face away from u. By Corollary 6, the root u can compute the number of nodes on the boundary in $\widetilde{O}(D)$ rounds.

We then employ the fragment merging procedure of [18] on the oriented face boundary minus the edge u, v. In this procedure, every parent fragment merges subset of its child fragments. Since the boundary of the face is a path (cycle without the $\{u, v\}$ edge), we are merging subpaths. Consider the one before last level of recursive merging, $L-1$, where we have two fragments: one containing u and one containing v. At the point, u and v can compute the size of their fragments N_u, N_v using low-congestion shortcuts and the root u can learn also the number of nodes in the fragment of v. At the point, we will assign that first N_u slots to the nodes in its fragment and the last N_v slots to the nodes in v's fragment. We can then remove the last merged edge and recurse in each of the two fragments. For an illustration, see Figure 9.

D Sublinear Time DFS Algorithm for General Graphs

We show that the method of the parallel algorithm of Aggarwal and Anderson [1] and Goldberg, Plotkin and Vaidya [22] can be adapted to produce a sublinear-time distributed algorithm in the CONGEST model. This is for graphs with diameter $D = o(n)$; of course no sublinear algorithm exists for graphs with $D = \Theta(n)$.

▶ **Lemma 11.** *For every n-node graph G with diameter D, a DFS can be constructed in $\widetilde{O}(\sqrt{Dn} + n^{3/4})$ rounds.*

Proof. Inspired by Aggarwal and Anderson [1], our DFS algorithm for general graphs consists of two phases. In the first phase, we compute a collection of $\widetilde{O}(p)$ paths which are separators for the remaining graph (i.e., removing all nodes on these $\widetilde{O}(p)$ paths breaks the graph into components of size in $[n/3, 2n/3]$). In the second phase, we attach these paths to the DFS tree, exactly as we do for the planar graphs, only this time we have $\widetilde{O}(p)$ many paths instead of one . Using the algorithm of Goldbreg, Plotkin and Vaidya [22], we will show that the first phase can be implemented by doing $\widetilde{O}(n/p)$ applications of a maximal matching algorithm. Recall that maximal matching can be solved in general graphs distributedly within $O(\log^4 n)$ rounds, even deterministically [26]. The second phase of attaching the paths to the initial DFS segment, consists of $\widetilde{O}(p)$ applications of the MST algorithm (or just connected components identification algorithm) which takes $\widetilde{O}(D + \sqrt{n})$ rounds using the algorithm of [14]. Hence, overall for a parameter p, the time complexity of the DFS algorithm is $\widetilde{O}(n/p + p(D + \sqrt{n})) = O(n^{3/4} + \sqrt{D \cdot n})$ by taking $p = O(n^{1/4})$ if $D \leq \sqrt{n}$, and set $p = O(\sqrt{n/D})$ otherwise.

We now describe the first phase in more details, as the second phase is exactly the same as our algorithm for planar graphs. The main procedure of the first phase of is a modified version of the Reduce routine of Aggarwal and Anderson [1]. We first explain the high level idea of this routine as given by [1] and then explain the modifications which come by the algorithm of Goldbreg, Plotkin and Vaidya [22]. The routine Reduce receives as input a collection Q of k node disjoint paths which are separators. The output of the procedure is smaller set Q' of node disjoint paths which is also a separator (i.e, the largest component of $V - Q'$ has size at most $n/2$) but in addition contains $|Q'| \leq |Q|/c$ paths for some constant $c > 1$. Initially, Q contains of all nodes in the graph and by doing $O(\log n)$ applications of this Reduce routine, one is left with a constant number of paths which are separators for the remaining graphs. Our implementation will be a bit more involved as we integrate the algorithm of [22]. Specifically, in our implementation of the Reduce procedure, in addition to the output set Q', we will also have some $O(p)$ node-disjoint paths. As we apply the Reduce procedure for $O(\log n)$ many times, we will have at the end $O(p \log n)$ node-disjoint paths which will be added to the DFS segment one by one.

The routine Reduce reduces the number of separator paths of Q while persevering the separation property, in the following manner. The paths of Q are divided into two sets: a set of $|Q|/4$ paths L and the remaining set of paths S (corresponding to *long* and *short*). The algorithm then computes a maximum cardinality set of disjoint paths $\mathcal{P} = \{P_1, \ldots, P_\alpha\}$ between L and S where there is a preference for these paths to start at the lowest possible point on the paths of L. This preference is important in order to maintain the separation property of these paths and to guarantee a progress in the reduction process. Aggarwal and Anderson [1] implement that step by considering a weighted version of the maximum cardinality set of node disjoint paths. They showed that this problem can be reduced to finding the minimum weight perfect matching in some related graph G''.

We note that there is no efficient distributed algorithm for perfect matching. Thus, we turn to an idea of [22], originally used for devising a sublinear-time deterministic parallel DFS algorithm. Their idea is to replace the prefect matching component of [1] with a more relaxed version of maximal matching algorithm. They noted that it is sufficient to consider a generalization of the *Maximal* Node-Disjoint Paths problem. This generalization outputs some extra node disjoint paths which will be added to the DFS segment one after the other. Specifically, the input for the generalized maximal node disjoint problem of [22] is a collection of sinks T and a set of $\mathcal{P}_{in}$ node disjoint paths connecting sources to intermediate nodes. The goal is to compute a set $\mathcal{P}_{out}$ of paths going from sources to sinks, such that for any

node that is on a path in $\mathcal{P}_{in}$ but not on a path in $\mathcal{P}_{out}$, every path from this node to a sink intersects a path in $\mathcal{P}_{out}$. We call the paths of $\mathcal{P}_{out}$, *completed* paths. The algorithm might also output a collection of at most p active paths $\mathcal{P}_{active}$ that start with the source nodes (but failed reach the nodes of T within the time given). These paths would be added to the initial DFS segment, one after the other. [22] chose the final number of active paths to be $p = \sqrt{n}$ and showed how to solve this generalization and end with $O(\sqrt{n})$ active paths, by applying $O(\sqrt{n})$ iterations of the maximal matching algorithm.

The high level idea of their algorithm is as follows. At the beginning of each iteration, there is a current collection of active paths starting at the source nodes. The goal of a single iteration is to extend these active paths by one hop (towards the sinks). The nodes belonging to the active paths are called *active*. The remaining nodes in the graph are either *idle* or *dead*, where initially all nodes not on the input active paths $\mathcal{P}_{in}$ are idle. The algorithm attempts to extend active paths by one hop, by computing a maximal matching between the end points of these paths and the remaining set of idle nodes in the graph. The paths whose endpoints are matched get extended by one hop. The remaining paths are chopped (or clipped) by their last node which then marked as dead. This continues for $\sqrt{n}$ iterations, at the end of which it is shown that there are at most $O(\sqrt{n})$ paths (see Lemma 3.2). One can see that in order to have at most p remaining active paths at the end of this phase, we need $O(n/p)$ iterations of the maximal matching algorithm. This is because in each iteration, each path is either extended by 1 or marks one node as dead. That is, each path "consumes" one node and since the consumed node-sets by the active paths are disjoint, we get that after n/p iterations, we have at most p paths.

Turning back to our DFS setting of Aggarwal and Anderson and its main routine Reduce, recall that there we have a collection of paths L and S and it is desired to find node disjoint paths between S and L that depart from the paths L as late as possible. This fits the generalized version of [22] in the following manner: contract the paths of S to be a single sink node[5] and let the paths of L be the initial paths $\mathcal{P}_{in}$. The algorithm will then attempt to extend the paths of $\mathcal{P}_{in}$ from their endpoints and if only when it is not possible those paths will be clipped. We will apply this modified Reduce routine (i.e., where we compute maximal matchings instead of minimum weight perfect matching) by setting p as described at the beginning of the proof. This is repeated for $O(\log n)$ times as in the algorithm of Aggarwal and Anderson. In each application of the Reduce routine, we solve the generalized version of maximal node disjoint problem, at the end of which we will have some completed paths connecting S to L but also a left over of p node-disjoint paths which are added to the DFS one after the other. This completes the DFS algorithm for general graphs. ◀

We now turn to the congested-clique model. In this model, the communication graph is a clique and every pair of nodes can exchange $O(\log n)$ bits, in each rounds. In this model, each path can be added to the DFS in $O(\log^* n)$ rounds, using the MST algorithm of [21]. Plugging it in the above, we get:

▶ **Corollary 12.** *There is an $\widetilde{O}(\sqrt{n})$ algorithm for computing a DFS in the congested clique model.*

E Figures

[5] We do not really need to contract them but we can label all nodes on a path in S by the same label since it is sufficient to hit any node in the path of S.

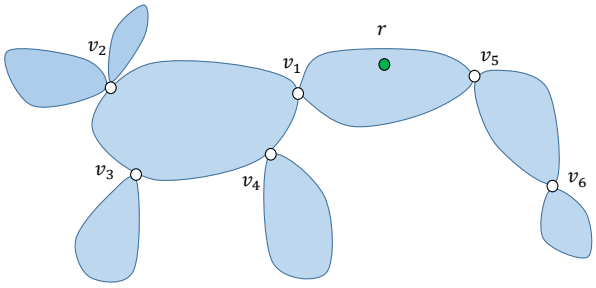

Figure 6 An induced connected subgraph $\mathcal{C}$ of G, depicted with its DFS root r, as well as its biconnected components and the corresponding cut-nodes v_1 to v_6.

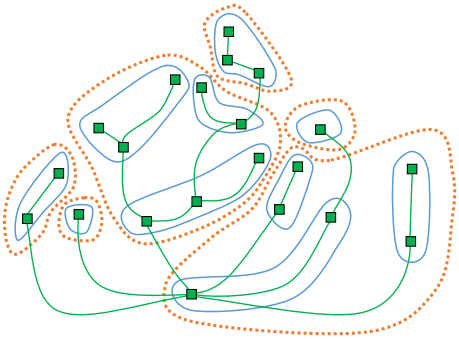

Figure 7 The fragmentation of the dual-tree from Fig. 2. Shown are the first three levels of merging. As each dual-node corresponds to a face in G, the merged fragment of the dual-tree is formed by a merging of faces.

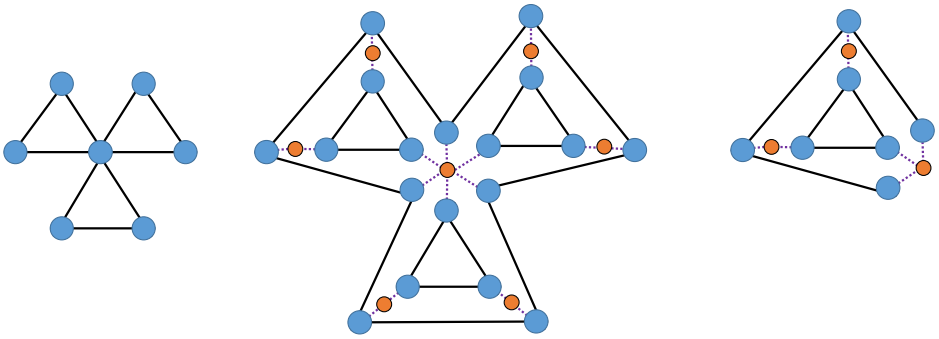

Figure 8 The graph on the left depicts the biconnected components, around a cut node. The middle graph depicts the result after the transition to the virtual graph $\widehat{G}$, which separates faces. Here, the star-edges E_S are depicted as purple dotted lines. Notice that for the cut node v at the center, after the transition to the virtual graph $\widehat{G}$ on the right, three copies of v are in the same connected component. Each two consecutive ones of these copies, in the clockwise traversal of the copies around v, indicate one biconnected component. In particular, the graph on the right depicts the portion of $\widehat{G}$ corresponding to one of the bi-connected components.

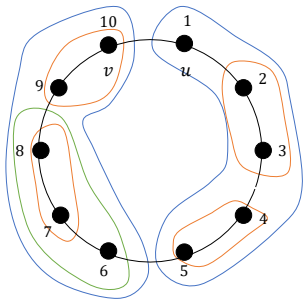

Figure 9 Clockwise Labeling of Nodes on the Boundary of the Face.