Near-Optimal Distributed DFS in Planar Graphs

Mohsen Ghaffari[1] and Merav P. Parter[2]

1 **ETH Zürich, Switzerland**
`ghaffari@inf.ethz.ch`
2 **CSAIL, MIT, Cambridge, USA**
`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, separator

Digital Object Identifier 10.4230/LIPIcs.DISC.2017.21

1 Introduction and Related Work

Depth First Search (DFS) is "*one of the most versatile sequential algorithm techniques known for solving graph problems*" [38]. Along with its cousin BFS, these two have a long history: DFS dates back to the 19th century [10], and BFS dates back to the 1950s [40]. 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 [36]. 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., [36, 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 [13, 27] which gave an $O(D + n^{0.61})$ round algorithm for minimum spanning tree. See, e.g., [5, 7, 8, 12, 14, 15, 18, 19, 24, 25, 28, 29, 32, 33, 34].

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*" [38] and "*a nightmare for parallel processing*" [31].

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 [20] 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 [20] is shown in the full version. 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 [30]. 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 [30], 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 Sec. 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 Sec. 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 [17]. 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 [13, 27], 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] [5, 7, 8, 12, 14, 15, 18, 19, 24, 25, 28, 29, 32, 33, 34]. There are also lower bounds [6, 9, 37] 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 [6]. 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 [16, 17], 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 [17], 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 [17]. An algorithm for computing this structure was also given in [17], which as one of its subroutines made use of the planar embedding algorithm of [16]. It was shown later by Haeupler, Izumi and Zuzic [21] 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

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

congestion shortcuts were later extended to other special graph families such as those with fixed tree-width or genus [22].

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 [38], and is thus unlikely to admit an efficient parallel algorithm. This was the reason that DFS was deemed *"inherently sequential"* [38]. However, over the years, several sophisticated but efficient parallel algorithms 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 [41] gave an $O(\log^3 n)$-time parallel DFS algorithm for planar graphs. Shannon [39] 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 [26] gave the first deterministic NC algorithm for DFS in planar networks. Then Hagerup [23] 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 [11].

We note that our distributed DFS algorithm for planar graphs is quite different than the parallel DFS algorithms for planar graphs (e.g., [39, 41]), 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

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

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 [16] 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.

Low-Congestion Shortcuts. In a subsequent paper [17], 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 [17] 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 Thm. 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 sub-problems 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 $\mathcal{C}'_i$ of $\mathcal{C} \setminus \mathcal{T}$, all neighbors of $\mathcal{C}'_i$ 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. Fig. 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 [17], 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 [17] 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

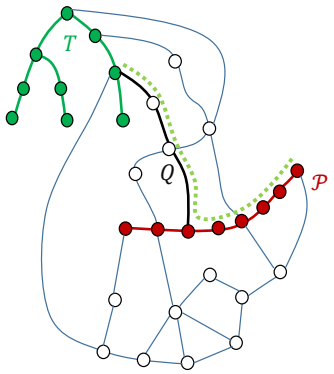

Figure 1 Growing the partial DFS tree. The green tree shows the current partial DFS $\mathcal{T}$, and the rest of the nodes are those of $\mathcal{C} \setminus \mathcal{T}$. The red path shows the current separator path $\mathcal{P}$. The black path is $\mathcal{Q}$, which connects the deepest point 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 $\mathcal{T}$.

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 [17]. 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. 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 [17] 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 the full version, 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 Fig. 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 Fig. 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

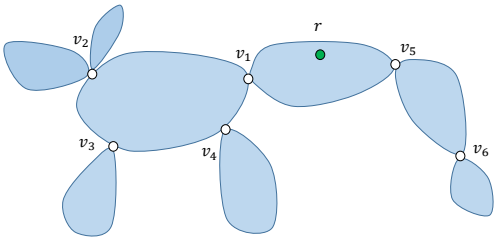

Figure 2 An induced connected subgraph $\mathcal{C}$ of G, depicted with its DFS root r, as well as its biconnected components and the corresponding cut-nodes v_1 to v_6.

$C_1, C_2, \ldots, C_\ell$ 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 [17] 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 [17] 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. Fig. 2 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 the full version.

4 Computing A Separator Path

4.1 Method Outline and Challenges

A celebrated result of Lipton and Tarjan [30] 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 $\mathcal{C}_1$, $\mathcal{C}_2$, $\ldots$, $\mathcal{C}_N$ 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. We start by computing a spanning tree T in $\mathcal{C}$. This is done using the MST algorithm of [17], 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 Sec. 2. See Fig. 3. 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 $\mathcal{F}_i$.

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

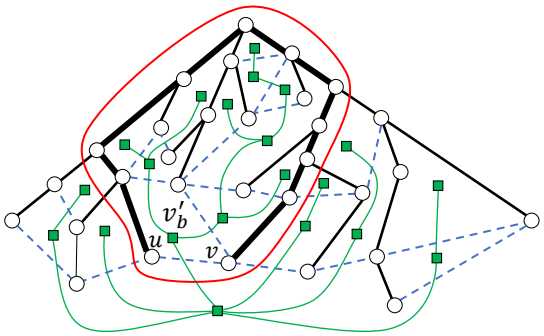

Figure 3 Shown is a planar graph and its path separator as computed by our algorithm. Solid edges are T-edges and the dashed blue edges are the non T-edges. These non-tree edges define the edges of the dual-tree T'. The dual-nodes are depicted as squares and the dual-edges of T' are the curved green 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.

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. 3 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 [17], as defined in Def. 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 [17], 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 [17, 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 [17, 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 Sec. 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, we present our algorithm for computing a path-separator in an arbitrary (biconnected) induced subgraph $\mathcal{C} \subseteq G$, using the tools explained in Sec. 4.2. The related analysis and smaller subroutines appear in the full version.

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 Fig. 4. 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 ordering of all its edges in G. This can be used to deduce the clockwise ordering[3] of its edges in $\mathcal{C}$.

[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 $\mathcal{C}$.

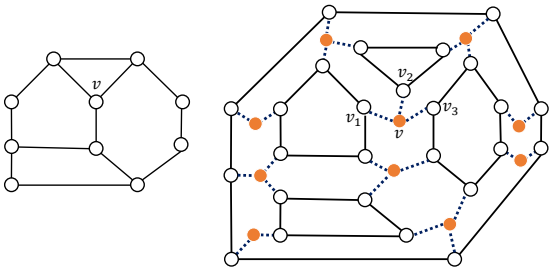

Figure 4 The transformation from G to $\widehat{G}$, which maps faces to node-disjoint connected subsets. The left figure depicts the graph before the transformation, and the right one depicts it after the transformation. The dotted links show the star-edge E_S. Notice that in the graph $\widehat{G}$ after the transformation, if we remove the star-edges, we get a collection of connected components, each corresponding to a face of $\mathcal{C}$.

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.*

Proof. 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. ◀

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.*

Proof. 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 [17] 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. ◀

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 $\alpha, \beta = O(D \log D)$.*

Proof. Consider the algorithm $\mathcal{A}$ of [17] 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 $\mathcal{C}$. ◀

▶ **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.*

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 Subsec. 4.2.2, we then describe how to use this recursive merging to solve the two tasks (D1) and (D2).

Recursive Face-Merging Process. To avoid computation in time $O(Diam(T'))$, we employ an idea of [17, 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 [35].

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. 5 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 Subsec. 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 Subsec. 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

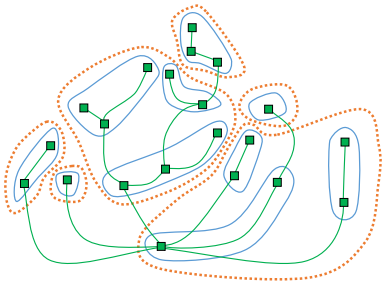

Figure 5 The fragmentation of the dual-tree from Fig. 3. Shown are the first three levels of merging. As each dual-node corresponds to a face in G, the merged fragment of the dual-tree is formed by a merging of faces.

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 the full version. We conclude by presenting a concise description of the entire algorithm, its detailed description and analysis is deferred to the full version.

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 Subset-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.

References

1. A. Aggarwal and R. Anderson. A random nc algorithm for depth first search. In *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. Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *PODC*, 2014.
6. 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 *STOC*, pages 363–372, 2011.
7. Atish Das Sarma, Danupon Nanongkai, and Gopal Pandurangan. Fast distributed random walks. In *PODC*, pages 161–170, 2009.
8. Atish Das Sarma, Danupon Nanongkai, Gopal Pandurangan, and Prasad Tetali. Efficient distributed random walks with applications. In *PODC*, pages 201–210, 2010.
9. Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *STOC*, pages 331–340, 2004.
10. Shimon Even. *Graph algorithms*. Cambridge University Press, 2011.
11. Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-nc. In *STOC*, pages 754–763. ACM, 2016.
12. Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *SODA*, pages 1150–1162, 2012.
13. J.A. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *FOCS*, 1993.
14. M. Ghaffari and F. Kuhn. Distributed minimum cut approximation. In *DISC*, pages 1–15, 2013.
15. 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.
16. Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks i: Planar embedding. In *PODC*, pages 29–38, 2016.
17. Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks ii: Low-congestion shortcuts, mst, and min-cut. In *SODA*, pages 202–219, 2016.
18. Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *PODC*, pages 81–90. ACM, 2015.
19. Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *International Symposium on Distributed Computing*, pages 197–211. Springer, 2014.
20. 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.
21. Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. In *PODC*, pages 451–460. ACM, 2016.
22. 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.
23. Torben Hagerup. Planar depth-first search in o(\logn) parallel time. *SIAM Journal on Computing*, 19(4):678–704, 1990.

24 Monika Henzinger, Sebastian Krinninger, and Danupon Nanongkai. A deterministic almost-tight distributed algorithm for approximating single-source shortest paths. In *STOC*, pages 489–498, 2016.

25 Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *PODC*, pages 355–364, 2012.

26 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.

27 Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *PODC*, pages 238–251, 1995.

28 Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *STOC*, pages 381–390, 2013.

29 Christoph Lenzen and Boaz Patt-Shamir. Improved distributed steiner forest construction. In *PODC*, 2014.

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

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

32 Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *STOC*, 2014.

33 Danupon Nanongkai, Atish Das Sarma, and Gopal Pandurangan. A tight unconditional lower bound on distributed randomwalk computation. In *PODC*, pages 257–266, 2011.

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

35 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.

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

37 David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed MST construction. In *FOCS*, pages 253–, 1999.

38 John H Reif. Depth-first search is inherently sequential. *IPL*, 20(5):229–234, 1985.

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

40 Steven S Skiena. *The algorithm design manual: Text*, volume 1. Springer Science & Business Media, 1998.

41 Justin R Smith. Parallel algorithms for depth-first searches i. planar graphs. *SIAM Journal on Computing*, 15(3):814–830, 1986.