Distributed Maximum Flow in Planar Graphs

Yaseen Abd-Elhaleem[*]
yaseenuniacc@gmail.com

Michal Dory[*]
mdory@ds.haifa.ac.il

Merav Parter[†]
merav.parter@weizmann.ac.il

Oren Weimann[*]
oren@cs.haifa.ac.il

Abstract

The dual of a planar graph G is a planar graph G^* that has a vertex for each face of G and an edge for each pair of adjacent faces of G. The profound relationship between a planar graph and its dual has been the algorithmic basis for solving numerous (centralized) classical problems on planar graphs involving distances, flows, and cuts. In the distributed setting however, the only use of planar duality is for finding a recursive decomposition of G [DISC 2017, STOC 2019].

In this paper, we initiate the study of distributed algorithms on dual planar graphs. Namely, we extend the distributed algorithmic toolkit (such as recursive decomposition and minor-aggregation) to work on the dual graph G^*. These tools can then facilitate various algorithms on G by solving a suitable dual problem on G^*.

Given a directed planar graph G with positive and negative edge-lengths and hop-diameter D, our key result is an $\tilde{O}(D^2)$-round algorithm[1] for Single Source Shortest Paths on G^*. This algorithm implies an $\tilde{O}(D^2)$-round algorithm for Maximum st-Flow in G, and an $\tilde{O}(D^2)$-round algorithm for directed Global Minimum Cut in G. Prior to our work, no $\tilde{O}(\text{poly}(D))$-round algorithms were known for these problems. When G is undirected, we further obtain a near optimal $\tilde{O}(D)$-round algorithm for computing the weighted girth of G, and a $D \cdot n^{o(1)}$-rounds $(1 - \epsilon)$-approximation algorithm for Maximum st-Flow in G when s and t lie on the same face.

The main challenges in our work are that G^* is not the communication graph (e.g., a vertex of G is mapped to multiple vertices of G^*), and that the diameter of G^* can be much larger than D (i.e., possibly by a linear factor). We overcome these challenges by carefully defining and maintaining subgraphs of the dual graph G^* while applying the recursive decomposition on the primal graph G. The main technical difficulty, is that along the recursive decomposition, a face of G gets shattered into (disconnected) components yet we still need to treat it as a dual node.

We believe that the toolkit developed in this paper for exploiting planar duality will be used in future distributed algorithms for various other classical problems on planar graphs (as happened in the centralized setting).

---

[*] *Department of Computer Science, University of Haifa.*

[†] *Faculty of Mathematics and Computer Science, Weizmann Institute of Science.*

[1] The $\tilde{O}(\cdot)$ notation is used to omit poly $\log n$ factors.

Contents

1 Introduction

Distributed algorithms for network optimization problems have a long and rich history. These problems are commonly studied under the CONGEST model [36] where the network is abstracted as an n-vertex graph $G = (V, E)$ with hop-diameter D; communications occur in synchronous rounds, and per round, $O(\log n)$ bits can be sent along each edge. A sequence of breakthrough results provided $\tilde{O}(D + \sqrt{n})$-round algorithms for fundamental graph problems, such as minimum spanning tree (MST) [9], approximate shortest-paths [33], minimum cuts [5], and approximate flow [16]. For general graphs, $\tilde{O}(D + \sqrt{n})$ rounds for solving the above mentioned problems is known to be near optimal, existentially [39].

A major and concentrated effort has been invested in designing improved solutions for special graph families that escape the topology of the worst-case lower bound graphs of [39]. The lower bound graph is sparse, and of arboricity two, so it belongs to many graph families. Arguably, one of the most interesting non-trivial families that escapes it, is the family of planar graphs. Thus, a significant focus has been given to the family of planar graphs, due to their frequent appearance in practice and because of their rich structural properties. In their seminal work, Ghaffari and Haeupler [13, 14] initiated the line of distributed planar graph algorithms based on the notion of *low-congestion shortcuts*. The latter serves the communication backbone for obtaining $\tilde{O}(D)$-round algorithms for MST [14], minimum cut [14, 18] and approximate shortest paths [41, 42] in planar graphs.

An additional key tool in working with planar graphs, starting with the seminal work of Lipton and Tarjan [27], is that of a planar *separator path*: a path whose removal from the graph leaves connected components that are a constant factor smaller. Ghaffari and Parter [17] presented a $\tilde{O}(D)$-round randomized algorithm for computing a cycle separator of size $O(D)$ which consists of a separator path plus one additional edge (that is possibly a *virtual* edge that is not in G). By now, planar separators are a key ingredient in a collection of $\tilde{O}(\text{poly}(D))$-round solutions for problems such as DFS [17], distance computation [26], and reachability [35]. An important aspect of the planar separator algorithm of [17] is that it employs a computation on the dual graph, by communicating over the primal graph.

Primal maximum flow via dual SSSP. Our goal in this paper is to expand the algorithmic toolkit for performing computation on the dual graph. This allows us to exploit the profound algorithmic duality in planar graphs, in which solving a problem A in the dual graph provides a solution for problem B in the primal graph. Within this context, our focus is on the *Maximum st-Flow* problem (in directed planar graphs with edge capacities). The Maximum st-Flow problem is arguably one of the most classical problems in theoretical computer science, extensively studied since the 50's, and still admitting breakthrough results in the sequential setting, such as the recent almost linear time algorithm by Chen, Kyng, Liu, Peng, Gutenberg and Sachdeva [2]. Despite persistent attempts over the years, our understanding of the distributed complexity of this problem is still quite lacking. For general *undirected n-vertex graphs*, there is a $(1 + o(1))$-approximation algorithm that runs in $(\sqrt{n} + D)n^{o(1)}$ rounds, by Ghaffari, Karrenbauer, Kuhn, Lenzen and Patt-Shamir [16]. For directed n-vertex *planar* graphs, a $D \cdot n^{1/2+o(1)}$-round exact algorithm has been given by de Vos [4]. No better tradeoffs are known for undirected planar graphs. In lack of any $\tilde{O}(\text{poly}(D))$-round maximum st-flow algorithm for directed planar graphs (not even when allowing approximation) we ask:

Question 1.1. Is it possible to compute the maximum st-flow in directed planar graphs within $\tilde{O}(\text{poly}(D))$ rounds?

In directed planar graphs with integral edge-capacities, it is known from the 80's [45] that the maximum st-flow can be found by solving $\log \lambda$ instances of *Single Source Shortest Paths* (SSSP) with positive and negative edge-lengths on the *dual* graph G^*, where λ is the maximum st-flow value. We answer Question 1.1 in the affirmative by designing a $\tilde{O}(D^2)$-round SSSP algorithm on the dual graph G^*. Our

1

algorithm works w.h.p.[2], it works in the most general setting (i.e. when G^* is directed and has positive and negative integral edge-lengths) and it matches the fastest known exact SSSP algorithm in the primal graph. We show:

Theorem 1.2 (Exact Maximum st-Flow in Directed Planar Graphs). *There is a randomized distributed algorithm that given an n-vertex directed planar network G with hop-diameter D and integral edge-capacities, and two vertices s, t, computes the maximum st-flow value and assignment w.h.p. in $\tilde{O}(D^2)$ rounds.*

No prior $\tilde{O}(\text{poly}(D))$ algorithm has been known for this problem, not even when allowing a constant approximation. We further improve the running time to $D \cdot n^{o(1)}$ rounds while introducing a $(1 - \epsilon)$ approximation, provided that G is undirected and that s and t both lie on the same face with respect to the given planar embedding:

Theorem 1.3 (Approximate Maximum st-Flow in Undirected st-Planar Graphs). *There is a randomized distributed algorithm that given an n-vertex undirected planar network G with hop-diameter D and integral edge-capacities, and two vertices s, t lying on the same face, computes a $(1 - \epsilon)$-approximation of the maximum st-flow value and a corresponding assignment in $D \cdot n^{o(1)}$ rounds w.h.p.*

This latter algorithm is based on an approximate SSSP algorithm that runs in $D \cdot n^{o(1)}$ rounds in planar graphs [42]. Our implementation of the algorithm on the dual graph matches its round complexity on the primal graph. The obtained almost-optimal round complexity improves significantly over the current algorithm for general graphs that runs in $(\sqrt{n} + D)n^{o(1)}$ rounds [16].

Minimum st-cut. By the well-known Max Flow Min Cut theorem of [7], our flow algorithms immediately give the value (or approximate value) of the minimum st-cut. We show that they can be extended to compute a corresponding bisection and the cut edges without any overhead in the round complexity. Moreover, since our exact flow algorithm works with directed planar graphs, it admits a solution to the *directed* minimum st-cut problem. To the best of our knowledge, prior to our work, $\tilde{O}(\text{poly}(D))$-round CONGEST algorithms for the minimum st-cut problem were known only for general graphs with constant cut values by [34]. This is in contrast to the global (undirected) minimum cut problem that can be solved in $\tilde{O}(D)$ rounds in planar graphs [14, 18].

Directed Global Minimum Cut. The directed global minimum weight cut problem is given a weighted directed graph and asks to find a bisection $(S, V \setminus S)$ of the vertex set, such that, the set of edges leaving S to $V \setminus S$ is of minimal total weight. To our knowledge, this problem was not studied in the CONGEST model and is open even for approximations, as opposed to the (directed or undirected) minimum st-cut problem [4, 34] and the undirected global minimum cut problem [5, 14, 18]. In light of efficient planar centralized results [31, 46] and motivated by the importance of the problem, we ask:

Question 1.4. Is it possible to compute the directed global minimum cut in planar graphs within $\tilde{O}(\text{poly}(D))$ rounds?

We answer Question 1.4 in the affirmative by showing an extension to our SSSP algorithm in the dual, allowing us to compute the minimum weight directed cycle in the dual. By duality, the same set of edges corresponds to the desired cut and determines a corresponding bisection in the primal, all within the same $\tilde{O}(D^2)$ bound on the round complexity:

Theorem 1.5 (Planar Directed Global Minimum Cut). *Given a directed planar graph G with non-negative integral edge weights, there is an $\tilde{O}(D^2)$ round randomized algorithm that finds a global directed minimum cut w.h.p..*

---

[2]W.h.p. stands for a probability of $1 - 1/n^c$ for an arbitrary fixed constant $c > 0$.

Primal weighted girth via dual cuts. A distance parameter of considerable interest is the network *girth*. For unweighted graphs, the girth is the length of the shortest cycle in the graph. For weighted graphs, the girth is the cycle of minimal total edge weight. Distributed girth computation has been studied over the years mainly for general *n*-vertex unweighted graphs. Frischknecht, Holzer and Wattenhofer [8] provided an $\Omega(\sqrt{n})$-round lower bound for computing a $(2-\epsilon)$ approximation of the unweighted girth. The state-of-the-art upper bound for the unweighted girth problem is a $(2-\epsilon)$ approximation in $\tilde{O}(n^{2/3}+D)$ rounds, obtained by combining the works of Peleg, Roditty and Tal [37] and Holzer and Wattenhofer [21]. The weighted girth problem has been shown to admit a near-optimal lower bound of $\tilde{\Omega}(n)$ rounds in general graphs [22, 28]. Turning to planar graphs, Parter [35] devised a $\tilde{O}(D^2)$ round algorithm for computing the weighted girth in directed planar graphs via SSSP computations. For undirected and unweighted planar graphs, the (unweighted) girth can be computed in $\tilde{O}(D)$ rounds by replacing the $\tilde{O}(D^2)$-round SSSP algorithm by a $O(D)$-round BFS algorithm. In light of this gap, we ask:

Question 1.6. Is it possible to compute the weighted girth of an undirected weighted planar graph within (near-optimal) $\tilde{O}(D)$ rounds?

We answer this question in the affirmative by taking a different, non distance-related, approach than that taken in prior work. Our $\tilde{O}(D)$ round algorithm exploits the useful duality between cuts and cycles. By formulating the dual framework of the *minor-aggregation* model, we show how to simulate the primal exact minimum cut algorithm of Ghaffari and Zuzic [18] on the dual graph. This dual simulation matches the primal round complexity. The solution to the dual cut problem immediately yields a solution to the primal weighted girth problem. We show:

Theorem 1.7 (Planar Weighted Girth). *There is a randomized distributed algorithm that given an n-vertex undirected weighted planar network G with hop-diameter D, computes the weighted girth and finds the edges of a shortest cycle w.h.p. in $\tilde{O}(D)$ rounds.*

2 Technical Overview

The dual of a planar graph G is a planar graph G^* that has a node[3] for each face of G. For every edge e in G there is an edge e^* in G^* that connects the nodes corresponding to the two faces of G that contain e. Our results are based on two main primal tools that we extend to work on the dual graph: *Minor Aggregation* and *Bounded Diameter Decomposition*. We highlight the key ideas of these techniques and the challenges encountered in their dual implementation. For all the algorithms that we implement in the dual graph, we match the primal round complexity.

2.1 Minor-Aggregation in the Dual

An important recent development in the field of distributed computing was a new model of computation, called the *minor-aggregation* model introduced by Zuzic Ⓡ[4] Goranci Ⓡ Ye Ⓡ Haeupler Ⓡ Sun [42], then extended by Ghaffari and Zuzic [18] to support working with *virtual nodes* added to the input graph. Recent state-of-art algorithms for various classical problems can be formulated in the minor-aggregation model (e.g., the exact min-cut algorithm of [18], and the undirected shortest paths approximation algorithms of [41, 42]). Motivated by the algorithmic power of this model, we provide an implementation of the minor aggregation model on the dual graph. As noted by [42], minor aggregations can be implemented by solving the (simpler) part-wise aggregation task, where one needs to compute an aggregate

---

[3] Throughout, we refer to faces of the primal graph G as *nodes* (rather than vertices) of the dual graph G^*.
[4] Ⓡ is used to denote that the authors' ordering is randomized.

function in a collection of vertex-disjoint connected parts of the graph. The distributed planar separator algorithm of [17] implicitly implements a part-wise aggregation algorithm on the dual graph. Our contribution is in providing an explicit and generalized implementation of the dual part-wise aggregation algorithm and using it to implement the minor-aggregation model on the dual graph, see Section 4 for the details. We then use this algorithm for computing the exact minimum weighted cut in the dual graph G^*, which by duality provides a solution to the *weighted girth* problem in the primal graph G. We also use it to simulate the recent approximate SSSP by [42] on the dual graph, leading to our approximate max st-flow algorithm on the primal graph when s and t are on the same face. Since currently there are fast SSSP minor-aggregation algorithms only for undirected graphs, this approach leads to an approximate max st-flow algorithm in undirected planar graphs. To solve the more general version of the max st-flow problem, we need additional tools described next.

2.2 SSSP in the Dual via Distance Labels

SSSP via distance labels. In order to compute single source shortest paths (SSSP) in the dual graph we compute *distance labels*. That is, we assign each face of G (node of G^*) a short $\tilde{O}(D)$-bit string known by all the vertices of the face s.t. the distance between any two nodes of G^* can be deduced from their labels alone. This actually allows computing all pairs shortest paths (APSP) by broadcasting the label of any face to the entire graph (thus learning a shortest paths tree in G^* from that face).

Theorem 2.1 (Dual Distance Labeling). *There is a randomized distributed $\tilde{O}(D^2)$-round algorithm that computes $\tilde{O}(D)$-bit distance labels for G^*, or reports that G^* contains a negative cycle w.h.p., upon termination, each vertex of G that lies on a face f, knows the label of the node f in G^*.*

The above theorem with aggregations on G^* allows learning an SSSP tree in the dual, i.e.,

Lemma 2.2 (Dual Single Source Shortest Paths). *There is a randomized $\tilde{O}(D^2)$ round algorithm that w.h.p. computes a shortest paths tree from any given source $s \in G^*$, or reports that G^* contains a negative cycle. Upon termination, each vertex of G knows for each incident edge whether its dual is in the shortest paths tree or not.*

Our labeling scheme follows the approach of [10] who gave a labeling scheme suitable for graph families that admit a small *separator*. We show the general idea on the primal graph and then our adaptation to the dual graph.

As known from the 70's [27, 29], the family of planar graphs admits cycle separators (a cycle whose removal disconnects the graph) of small size $O(\sqrt{n})$ or $O(D)$. Since $\Omega(D)$ is a lower bound for all the problems we discuss in the distributed setting, we will focus on separators of size $\tilde{O}(D)$. The centralized divide-and-conquer approach repetitively removes the separator vertices S_G from the graph G and recurses on the two remaining subgraphs that are a constant factor smaller, constituting a hierarchical decomposition of the graph with $O(\log n)$ levels. The labeling scheme is defined recursively, where the label of a vertex v in G stores distances between v and S_G and recursively the label of v in the subgraph that contains v (either the interior or the exterior of S_G). Due to S_G being a cycle, any shortest u-to-v path P for any two vertices u, v in G either crosses S_G (and the u-to-v distance is known by the distances between u, v and S_G stored in their labels), or P is enclosed in the interior or the exterior of S_G (and the u-to-v distance is decoded from the recursive labels of u and v in the subgraph that contains them). Moreover, since the separator is of size $\tilde{O}(D)$ and since there are $O(\log n)$ levels in the decomposition, the labels are of size $\tilde{O}(D)$.

Bounded diameter decomposition (BDD). The BDD is a distributed hierarchical decomposition for planar graphs, which plays an analogous role to the centralized recursive separator decomposition. It was

devised by Li and Parter [26] by carefully using and extending a distributed planar separator algorithm of Ghaffari and Parter [17]. The BDD has some useful properties that are specific to the distributed setting. In particular, all the separators and subgraphs obtained in each recursive level have small diameter of $\tilde{O}(D)$ and are nearly disjoint, which allows to broadcast information in all of them efficiently in CONGEST. An immediate application of the BDD is a distributed distance labeling algorithm for primal planar graphs that follows the intuition above. In addition, BDDs have found other applications in CONGEST algorithms on the primal graph G (e.g. diameter approximation, routing schemes and reachability [6, 26, 35]).

Challenges in constructing distance labels for the dual graph. Ideally, we would like to apply the same idea on the dual graph. However, there are several challenges. First, if we apply it directly, the size of the separator and the running time of the algorithm will depend on the diameter of the dual graph, that can be much larger than the diameter D of the primal graph (possibly by a linear factor). So it is unclear how to obtain a running time that depends on D. Second, even though there are efficient distributed algorithms for constructing the BDD in the primal graph, it is unclear how to simulate an algorithm on the dual network, as this is not our communication network.

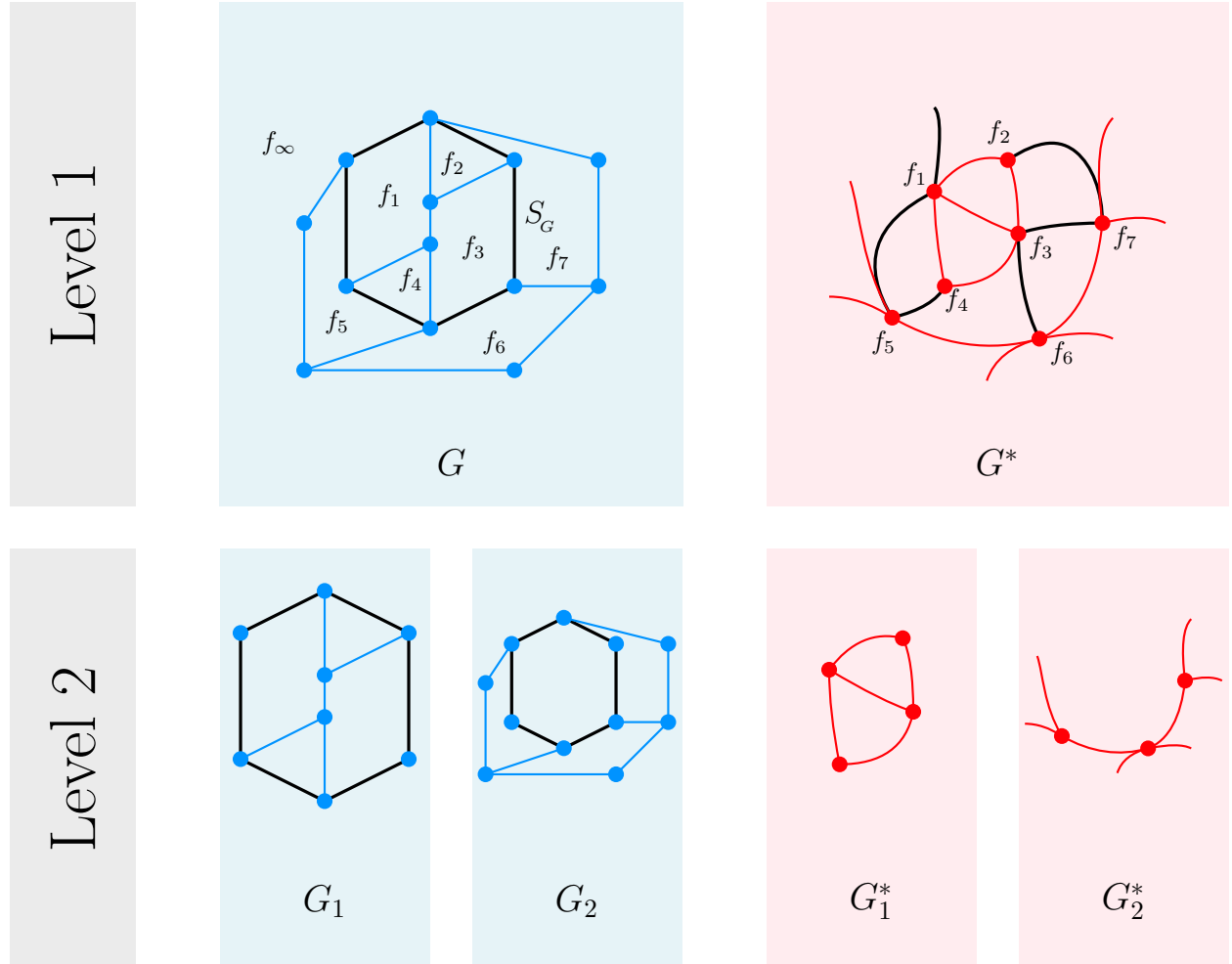
Figure 1: Two levels of the recursive separator decomposition of G and their correspondent levels in the desired dual decomposition of G^*. The black edges are the separating cycle S_G in G, the same edges map to a cut in G^*. For simplicity, the dual node corresponding to the face f_∞ of G is not illustrated in G^* (its incident edges are the ones missing an endpoint).

5

Our approach: recurse on primal, solve on dual. To overcome this, we suggest a different approach. We use the primal BDD, and infer from it a decomposition of the dual graph. To get an intuition, note that by the cycle-cut duality, a cycle in G constitute a cut in G^*. This means that the separator of G can be used as a separator of G^*, in the sense that removing the cut edges (or their endpoints) disconnects G^*. I.e, a hierarchical decomposition of G can (conceptually) be thought of as a hierarchical decomposition of G^*. See Figure 1.

Then, we would hope that using [26]'s algorithm for such a hierarchical decomposition for planar graphs would allow us to apply the same labeling scheme on G^*, where we consider the dual endpoints of S_G edges as our separator in G^*. However, taking a dual lens on the primal decomposition introduces several challenges that arise when one needs to define the dual subgraphs from the given primal subgraphs. This primal to dual translation is rather non-trivial due to critical gaps that arise when one needs to maintain information w.r.t. faces of G, rather than vertices of G. To add insult to injury, the nice structural property of the separator being a cycle is simply not true any more. This is because the separator of [17] that the BDD of [26] uses is composed of two paths plus an additional edge that exists only when the graph is two-connected. Otherwise, that edge is not in G and cannot be used for communication. This *virtual* edge, if added, would split a face into two parts, introducing several challenges, including:
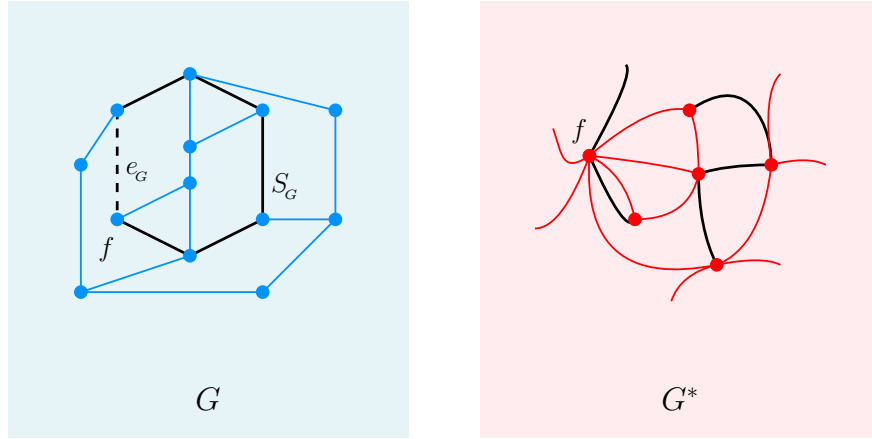

Figure 2: The edge e_G in the interior of the face f is the virtual edge added to close a cycle, i.e., $P = S_G \setminus \{e_G\}$. Note, since e_G is not an edge of G, it does not exist in G^*, and f corresponds to a single node in G^* instead of two. Thus, removing the separator vertices in G disconnects G, however, removing its real (non-virtual) edges in G^* does not disconnect it.

Challenge I: dual subgraphs. There are three complications in this regard: (1) Taking the natural approach and defining the dual decomposition to contain the dual subgraph of each subgraph in the primal decomposition, does not work. This is because in the child subgraphs there are faces that do not exist in their parent subgraph (e.g. in Figure 1, the "external" face of black edges in G_1). More severely, the cycle S_G may not even be a cycle of G, rather, a path P accompanied with a virtual edge. Thus, P is not necessarily a cut in G^*. (2) The second complication arises from the distributed hierarchical decomposition of [26] which may decompose each subgraph into as many as $\tilde{O}(D)$ subgraphs rather than two. This is problematic in the case where a given face (dual node) is split between these subgraphs. (3) Finally, the useful property that a parent subgraph in the primal decomposition is given by the union of its child subgraphs is simply not true in the dual, since removing P does not necessarily disconnect G^* and may leave it connected by the face that contains the virtual edge (see Figure 2). In order to disconnect G^*, that face must be split, which solves one problem but introduces another as explained next.

Challenge II: face-parts. In the primal graph, a vertex is an atomic unit, which keeps its identity through-

out the computation. The situation in the dual graph is considerably more involved. Consider a primal graph G with a large face f containing $\Theta(n)$ edges. Throughout the recursive decomposition, the vertices of the face f are split among multiple faces, denoted hereafter as *face-parts*, and eventually f is shattered among possibly a linear number of subgraphs. This means that a node in a dual subgraph does no longer correspond to a face f of the primal subgraph, but rather to a subset of edges of f. This creates a challenge in the divide-and-conquer computation, where one needs to assemble fragments of information from multiple subgraphs.

New Structural Properties of the Primal BDD. We mitigate these technical difficulties by characterizing the way that faces are partitioned during the primal BDD algorithm of [26]. While we run the primal BDD almost as is, our arguments analyze the primal procedure from a dual lens. Denote each subgraph in the BDD as a *bag*. We use the primal BDD to define a suitable decomposition of G^* that is more convenient to work with when performing computation on the dual graph. Interestingly, for various delicate reasons, in our dual BDD , the dual bag X^* of a primal bag X in the BDD is not necessarily the dual graph of X. Finally, we show that each vertex in the primal graph can acquire the local distributed knowledge of this dual decomposition.

Our dual perspective on the primal BDD allows us to provide a suitable labeling scheme for G^*. Next we give a more concrete description of our dual-based analysis of the primal BDD. The full details are provided in Section 5.1.

Few face-parts. We show that in each bag X of the BDD there is at most one face of G that can be partitioned between different child bags of X and was not partitioned in previous levels. This face is exactly the *critical* face f that contains the virtual edge of the bag. We call the different parts of f that appear in different child bags *face-parts*. Since the decomposition has $O(\log n)$ levels, overall we have at most $O(\log n)$ face-parts in each bag. Note that we do not count face-parts that were obtained by splitting the critical face in ancestor bags (i.e., by splitting existing face-parts). For full details, the reader is referred to Section 5.1.1.
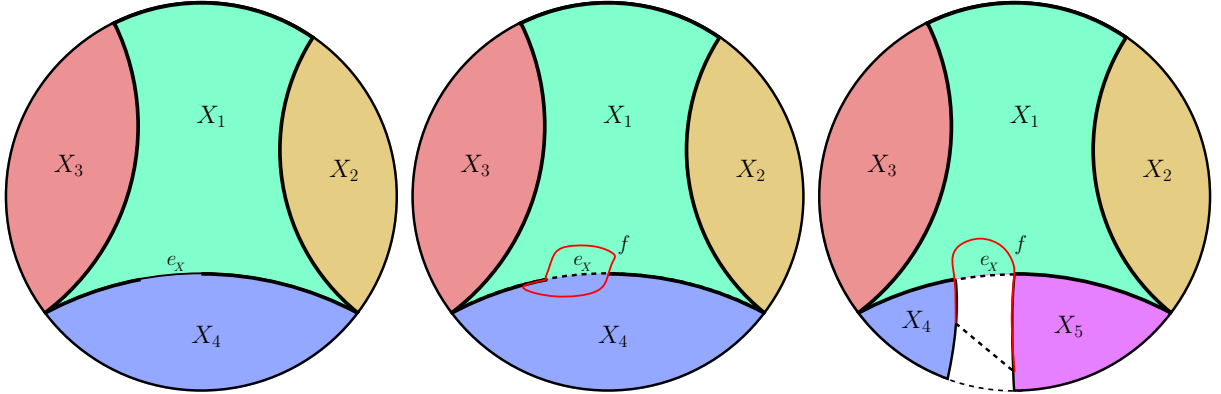
Figure 3: Three examples of a bag X and its child bags X_i (of distinct colors). The bold black cycle is the separator S_x. The virtual edge e_x is dashed. The green child bag X_1 is the interior of S_x. When $e_x \in E(G)$ (left image), we show that no face is partitioned between the child bags X_i. When $e_x \notin E(G)$ (middle and right images), edges of the critical face f are red. Notice, the face f (middle) is partitioned into two child bags (X_1 and X_4) and the face-part f (right) is partitioned into three child bags (X_1, X_4, X_5).

Lemma 2.3 (Few face-parts, Informal). *Any bag X of the BDD contains at most $O(\log n)$ face-parts.*

To prove Lemma 2.3, we consider the separator S_x of X computed in the construction of the BDD by [26]. S_x constitutes a separating cycle in X containing at most one virtual edge e_x that might not be in

$E(G)$. The interior of S_x defines one child bag X_1 of X, and its exterior may define as many as $\tilde{O}(D)$ child bags $X_2, X_3, \ldots$ (see Figure 3). We prove that the face containing the endpoints of e_x is the only face that might get partitioned in the bag X. First, we show that if $e_x \in E(G)$ then no face is partitioned, and all faces are entirely contained in one of the child bags. If $e_x \notin E(G)$, then we show that the critical face is the only face of G that can be partitioned in X. We remark that face-parts can also be further partitioned, as shown in Figure 3.

Dual bags and separators. Since we are interested in faces of G, we trace them down the decomposition and take face-parts into consideration when defining the dual bags. That is, both the faces and the face-parts that belong to a bag X have corresponding dual nodes in X^*. We connect two nodes of X^* by a dual edge only if their corresponding faces share a primal (non-virtual) edge in X. If $X = G$, then there are no face-parts and X^* is the standard dual graph G^*. For full details, the reader is referred to Section 5.1.

As mentioned earlier, we would like the separator of a bag X to constitute a separator for X^*. We show that this is almost the case. In particular, we define a set F_x of size $\tilde{O}(D)$ that constitutes a separator in X^*. The set F_x consists of (1) nodes incident to dual edges of S_x in X^*, and (2) nodes corresponding to faces or face-parts that are partitioned between child bags of X^*. See Figure 4.
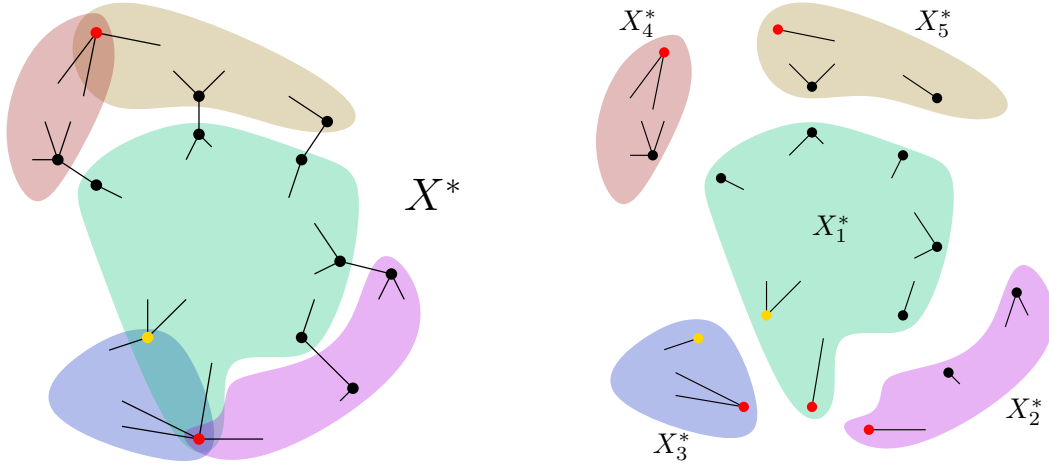

Figure 4: A bag X^* on the left and its child bags on the right. Each shadow color demonstrates a distinct child bag of X^*. Each edge that is shadowed by a color, is entirely contained in the child bag corresponding to that color. The nodes shown in X^* are the nodes of F_x. Some of these nodes are partitioned into several nodes in different child bags of X^*. The black nodes are nodes of X^* that are incident to a (dual) edge of S_x, the yellow node corresponds to the critical face of X, and the red nodes correspond to face-parts of X. Notice how for the face-parts and the critical face of X there are incident edges, each is entirely contained in a distinct child bag of X^*. Edges that are not contained in any child bag of X^* are (dual) edges of S_x.

Following the high-level idea of the labeling scheme, the set F_x is crucial. The main structural property it provides is the following.

Lemma 2.4 (Dual Separator - Informal). *Any path in X^* that is not entirely contained in a single child bag of X^* must intersect F_x.*

The above lemma is proved in Section 5.1.2. Intuitively, when X^* is split into its child bags, some of its nodes are "shattered" (those that correspond to its critical face or its $O(\log n)$ face-parts), and some of its edges are removed (dual S_x edges). We show how to assemble X^* from its child bags by considering faces and face-parts of child bags that correspond to nodes of F_x.

Distributed knowledge. A node f in X^* is simulated by the vertices of the corresponding face or face-part f of X, and each dual edge adjacent to f will be known by one of its endpoints. In particular, we assign each face and face-part in a bag X a unique $\tilde{O}(1)$-bit identifier and learn for each vertex $v \in X$, the set of faces and face-parts containing it and its adjacent edges. This is done by keeping track of G's edges along the decomposition. At first we learn G^* and then extend this recursively to its child bags. The implementation relies on the fact that we have only a few face-parts in each bag (Lemma 2.3) preventing high congestion when we broadcast information about face-parts to learn the dual nodes. For details see Section 5.1.3.

The labeling scheme. Our next goal is to use the decomposition in order to compute distances in the dual graph. More concretely, we (distributively) compute, for every node in every bag X^*, a short label of size $\tilde{O}(D)$, such that given the labels alone of any two nodes in X^* we can deduce their distance. The labeling scheme is a refinement of the intuition given earlier. For full details, the reader is referred to Section 5.2.

Recall, the set F_X plays the role of a node separator of X^* (Lemma 2.4). A node $g \notin F_X$ of X^* corresponds to a real face of G (i.e. not a face-part) contained in X. The distance label $\text{Label}_{X^*}(g)$ of $g \notin F_X$ is defined recursively. If X^* is a leaf in the decomposition then $\text{Label}_{X^*}(g)$ stores $\text{ID}(g)$ and the distances between g and all other nodes $h \in X^*$. Otherwise, the label consists of the ID of X, the distances in X^* between g and all nodes of F_X, and (recursively) the label of g in the child bag of X^* that entirely contains g. In case $g \in F_X$, we simply store the distances between g and all other nodes in F_X. The labels are of size $\tilde{O}(D)$-bits since $|F_X| = \tilde{O}(D)$ and since the BDD is of a logarithmic height.

The correctness follows from Lemma 2.4. I.e. a g-to-h shortest path P in X^* is either: (1) Entirely contained in X_i^*, and $\text{dist}_{X^*}(g, h)$ can be deduced from $\text{Label}_{X_i^*}(g)$ and $\text{Label}_{X_i^*}(h)$, or (2) P intersects F_X with an edge, then $\text{dist}_{X^*}(g, h) = \min_{f \in F_X}\{\text{dist}_{X^*}(g, f) + \text{dist}_{X^*}(f, h)\}$. If one of g or h is a node of F_X then distances are retrieved instantly, as each node in X^* stores its distances to F_X.

The algorithm. We compute the distance labels bottom-up on the decomposition. In the leaf bags, due to their small size ($\tilde{O}(D)$-bits) we allow ourselves to gather the entire bag in each of its vertices. Then, vertices locally compute the labels of faces and face-parts that contain them. If a negative cycle is detected, it gets reported via a global broadcast on G and the algorithm terminates.
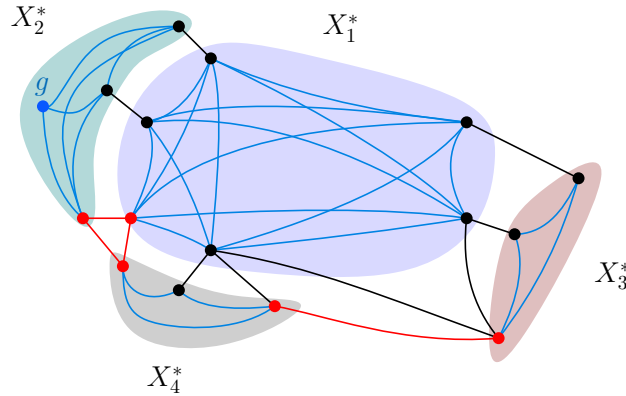
Figure 5: The graph $DDG(g)$. The blue node is g, other nodes are nodes of F_X, in particular, red nodes correspond to face-parts and black nodes to endpoints of S_X. The blue edges represent distances between nodes that belong to the same child bag of X^*, red edges connect face-parts of the same face in X^*, and black edges are dual edges to S_X edges. For simplicity, the figure is undirected.

In non-leaf bags, we broadcast $\tilde{O}(D^2)$ information that includes labels of nodes in F_X in the child bags ($\tilde{O}(D)$ labels, each of size $\tilde{O}(D)$), and the edges of the separator. Since all the bags in the same level have

9

$\tilde{O}(D)$ diameter and are nearly disjoint, we can broadcast all the information in $\tilde{O}(D^2)$ rounds in all bags. We prove that based on this information nodes can locally deduce their distance label in X^*.

In particular, each vertex $v \in X$ that belongs to a face or a face-part g of X constructs locally a *Dense Distance Graph* denoted $DDG(g)$. $DDG(g)$ is a small (non-planar) graph (with $\tilde{O}(D)$ vertices and $\tilde{O}(D^2)$ edges) that preserves the distances in X^* between pairs of nodes in $F_X \cup \{g\}$. See Figure 5. Via (local) APSP computations on the DDGs, v learns the distances between g and F_X nodes in X^* (i.e., constructs Label$_{X^*}(g)$). Again, vertices check for negative cycles locally (in the DDG). To prove correctness, we show that each shortest path P in X^* can be decomposed into a set of subpaths and edges whose endpoints are in $F_X \cup \{g\}$. Each such subpath is entirely contained in a child bag of X^* (so its weight can be retrieved from the already computed labels) and each such edge is a (dual) edge of S_X (whose identity and weight were broadcast). For full details, the reader is referred to Section 5.3.

Finally, in order to compute distances from any given node $s \in G^*$, it suffices to broadcast its label to the entire graph. Thus, vertices can learn locally for each face of G that contains them the distance from s. To learn an SSSP tree from s in G^*, for each node g in G^* we mark its incident edge $e = (f, g)$ that minimizes dist$(s, f) + w(e)$. For this task, we use our Minor-Aggregation implementation on G^*. Concluding our main result, a labeling scheme that gives dual SSSP and hence primal maximum st-flow.

2.3   Flow Assignments and Cuts

Maximum st-flow. It was shown in the 90's by Miller and Naor [30], that an exact st-flow algorithm on a directed planar graph G with positive edge-capacities can be obtained by $\log \lambda$ applications of a SSSP algorithm on the dual graph G^* with positive and negative edge-weighs (each application with possibly different edge weights). We use this result as a black-box. At the end, we obtain the flow value and the flow assignment to the edges of G. This works in $\tilde{O}(D^2)$ rounds, since λ (the value of the maximum st-flow) is polynomial in n and since our dual SSSP algorithm terminates within $\tilde{O}(D^2)$ rounds.

Approximate maximum st-flow. For the case of undirected planar graphs where s and t lie on the same face with respect to the given embedding, we further obtain an improved round complexity at the cost of solving the problem approximately. Namely, it was shown in the 80's by Hassin [20] that in this setting it suffices to use an SSSP algorithm on the dual graph after augmenting it with a single edge, dual to the edge (s, t), that is limited to positive edge-weights and to undirected graphs. Since s and t are on the same face it is possible to add such an edge while preserving planarity. Note that adding this edge splits a node in the dual graph. We show that we can still simulate efficently minor-aggregation algorithms on the augmented dual graph. To compute SSSP on that graph we use an SSSP algorithm by [42] that works in the minor-aggregation model. In particular, it computes a $(1 + \epsilon)$-approximate SSSP tree within $D \cdot n^{o(1)}$ rounds, which allows us to find a $(1 - \epsilon)$-approximate flow value in this time.

Finding the flow assignment. As mentioned above, our exact maximum st-flow algorithm also gives a flow assignment. However, in the approximate case, our algorithm easily gives the approximate flow value, but it is not guaranteed to give a corresponding flow assignment due to the fact that the reduction we use by [20] is meant to be used with an exact algorithm. In particular, to have an assignment, the outputted SSSP tree needs to (approximately) satisfy the triangle inequality. However, most distance approximation algorithms do not satisfy that quality. To overcome this, we apply a method of [40], that given any approximate SSSP algorithm produces an SSSP tree that (approximately) satisfies the triangle inequality. To obtain an assignment, by [40], we have to apply the approximate SSSP algorithm $\tilde{O}(1)$ times each on different virtual graphs related to the dual graph. This is not immediate, and raises several challenges. Most importantly, their algorithm was not implemented in the minor-aggregation model so it is not immediate to simulate it on the dual graph. We exploit the features of the minor-aggregation model excessively in order to resolve this complication.

Minimum st-cut. By the celebrated min-max theorem of Ford and Fulkerson [7], the exact (or approximate) value of the max st-flow is equivalent to that of the min st-cut. We show that our algorithms can also be extended to compute a corresponding bisection and mark the cut edges of the min st-cut. In the exact case, we use classic textbook methods (e.g, residual graphs), and since this flow algorithm works for directed planar graphs, we obtain a *directed* minimum st-cut. In the approximate case, we have to work harder and exploit the minor-aggregation model on the dual graph one last time, in which we apply a special argument on the cycle-cut duality that was given in the 80's by Reif [38].

Directed global minimum cut. We exploit our techniques to find also the directed global minimum cut. The general idea is simple, we use the cycle-cut duality: finding the (directed) minimum cut in G amounts to finding the minimum weight (directed) simple cycle C in G^*. Thus, given an extended BDD of G and having distance labels already computed in $\tilde{O}(D^2)$ rounds for all dual bags in the BDD, we have that any shortest cycle of G^* either intersects a dual separator node or is entirely contained in one of G^*'s child bags. Thus, in order to find C, we take the minimum over all: (i) directed cycles that intersect separator nodes of G^* (deduced by pairwise decoding labels of the separator nodes), and (ii) minimum directed cycles in G^*'s child bags (found recursively). However, we note that the duality does not hold in general directed planar graphs, which complicates the above approach. I.e., in order for the duality to hold, we have to work with a graph where for each edge, its *reversal dart* (an edge in the opposite direction) is present, so we add those darts. Then, we should ensure that the cycles we consider are simple darts cycles (i.e. a cycle where no edge and its reversal dart appear together), which we do by investigating two options related to the dual separator. Finding the cut bisection is done simply by detecting connected components (in the undirected sense) after the removal of the cut edges and their reversal darts.

Roadmap. The rest of the paper is organized as follows. Section 3 discusses preliminaries. Section 4 gives an overview of our minor-aggregation simulation on the dual graph and its applications for the weighted girth, where full details and proofs appear in Appendix B. Our labeling scheme for the dual graph appears in Section 5, and our applications for max st-flow, min st-cut appear in Section 6, while the directed global min-cut appears in Section 7. See also Figure 6.
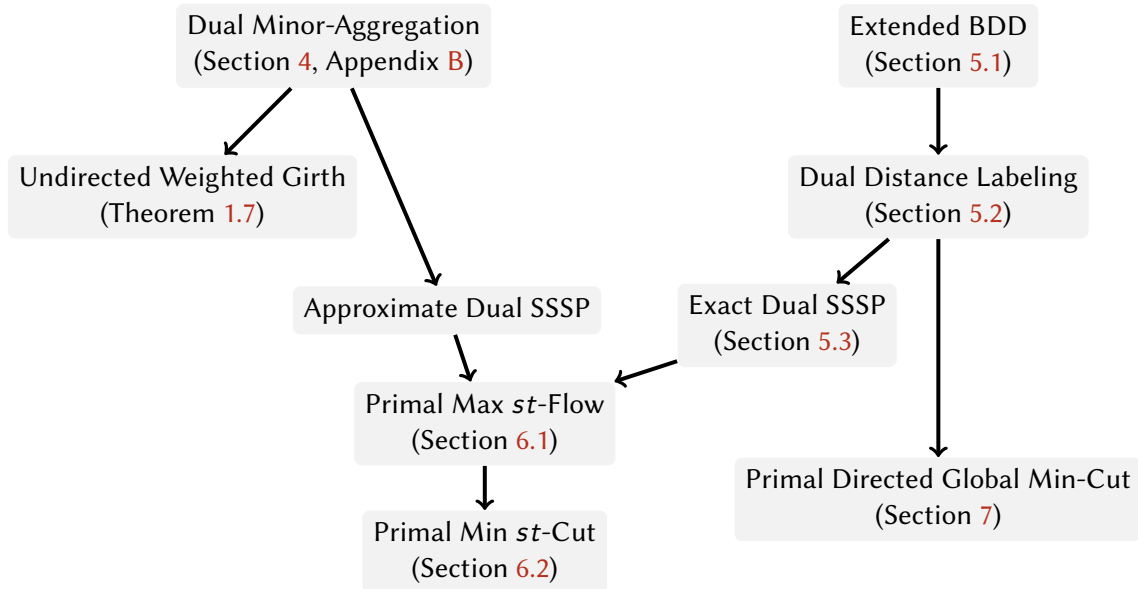

Figure 6: A roadmap of the paper.

11

3 Preliminaries

We denote by $G = (V, E)$ the directed (possibly weighted) simple planar network of communication, and by D the network's undirected and unweighted (hop) diameter. Let S be a set of vertices or edges, we denote by $G[S]$ the subgraph of G induced by S.

The CONGEST **model.** We work in the standard distributed CONGEST model [36]. Initially, each vertex knows only its unique $O(\log n)$-bit identifier and the identifiers of its neighbors. Communication occurs in synchronous rounds. In each round, each vertex can send (and receive) an $O(\log n)$-bit message on each of its incident edges (different edges can transmit different messages). When the edges of G are weighted we assume that the weights are polynomially bounded integers. Thus, the weight of an edge can be transmitted in $O(1)$ rounds. This is a standard assumption in the CONGEST model.

Distributed storage. When referring to a distributed algorithm that solves some problem on a network G, the input (and later the output) is stored distributively. That is, each vertex knows only a small (local) part of the information. E.g. when computing a shortest paths tree from a single source vertex s, we assume that all vertices know (as an input) the ID of s and the weights and direction of their incident edges. When the algorithm halts, each vertex knows its distance from s and which of its incident edges are in the shortest paths tree.

Planar embedding. Let $G = (V, E)$ be a directed planar graph. The *geometric* planar embedding of G is a drawing of G on a plane so that edges intersect only in vertices. A *combinatorial* planar embedding of G provides for each $v \in G$, the local clockwise order of its incident edges, such that, the ordering of edges is consistent with some geometric planar embedding of G. Throughout, we assume that a combinatorial embedding of G is known locally for each vertex. This is done in $\tilde{O}(D)$ rounds using the planar embedding algorithm of Ghaffari and Haeupler [13].

The dual graph G^*. The *dual* of the *primal* planar graph G is a planar graph, denoted G^*. The nodes of G^* correspond to the faces of G. The dual graph has an edge e^* for each pair of faces in G that share an edge $e \in G$ (and a self-loop when the same face appears on both sides of an edge). If e is directed then the direction of e^* is from the face on the left of e to the face on the right of e where left and right are defined with respect to the direction of e. Observe that if two faces of G share several edges then G^* has several parallel edges between these faces. I.e., G^* might be a multi-graph even when the primal graph G is a simple graph. See Figure 7. Since the mapping between edges of G and G^* is a bijection, we sometimes abuse notation and refer to both e and e^* as e. We refer to the vertices of G as *vertices* and to the vertices of G^* as *nodes*.
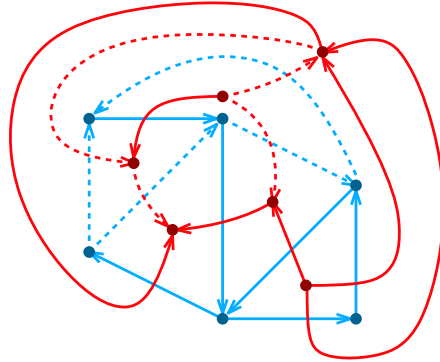

Figure 7: The graph G in blue and its dual graph G^* in red. A primal cycle and its dual cut are dashed.

We use the well-known duality between primal cycles and dual cuts:

Fact 3.1 (Undirected Cycle-Cut Duality [25]). A set of edges C is a simple cycle in an embedded connected planar graph G if and only if C is a simple cut in G^* (a cut C is said to be simple if when removed, the resulting graph has exactly two connected components).

The face-disjoint graph $\hat{G}$. In the distributed setting, the computational entities are vertices and not faces. Since a vertex can belong to many (possibly a linear number of) faces of G, this raises several challenges for simulating computations over the dual graph G^*: (1) We want to communicate on G^* but the communication network is G. (2) G^* might be of a large diameter (up to $\Omega(n)$ even when the diameter of G is $D = O(1)$). (3) We wish to compute aggregate functions on sets of faces of G. To do so efficiently we want faces to be vertex-disjoint.

To overcome these problems, the *face disjoint graph* $\hat{G}$ was presented in [17] as a way for simulating aggregations on the dual graph G^* in the distributed setting, which we slightly modify here. As it is mainly a communication tool, we think of $\hat{G}$ as an undirected unweighted graph, however, it will still allow working with weighted and directed input graphs G, as each endpoint of an edge in $\hat{G}$ that represents an edge of G^*, shall know the weight and direction of that edge. Intuitively, $\hat{G}$ can be thought of as the result of duplicating all edges of G so that faces of G map to distinct faces of $\hat{G}$ that are both vertex and edge disjoint. See Figure 8.
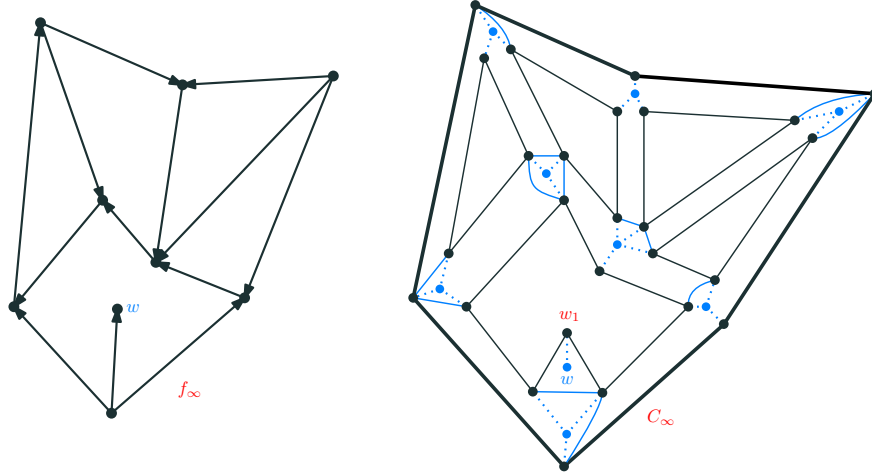
Figure 8: On the left, the primal graph G. On the right, the face-disjoint graph $\hat{G}$. Each vertex in G has a star-center (blue) copy in $\hat{G}$, in addition to $\deg(v)$ (black) additional copies. Black edges in $\hat{G}$ are duplicates of G edges, such that, connected components induced by them map to faces of G, e.g., the face f_∞ of G maps to the component C_∞ in $\hat{G}$ (thick black). Dashed blue edges connect each star-center to other copies of the same vertex and solid blue edges connect two components if and only if there corresponding faces should be connected in G^*. For example, there are two such edges connecting C_∞ to the component that maps to the face which contains w.

The following are the properties of $\hat{G}$ that we use. For the full definition of $\hat{G}$ and a proof of these properties see Appendix A.

1. $\hat{G}$ is planar and can be constructed in $O(1)$ rounds (after which every vertex in G knows the information of all its copies in $\hat{G}$ and their adjacent edges).

2. $\hat{G}$ has diameter at most $3D$.

3. Any r-round algorithm on $\hat{G}$ can be simulated by a $2r$-round algorithm on G.

4. There is an $\tilde{O}(D)$-round algorithm that identifies G's faces by detecting their corresponding faces in $\hat{G}$. When the algorithm terminates, every such face of $\hat{G}$ is assigned a face leader that knows the face's ID. Finally, the vertices of G know the IDs of all faces that contain them, and for each of their incident edges the two IDs of the faces that contain them. Thus, each vertex of G knows for each pair of consecutive edges adjacent to it (using its clockwise ordering of edges) the ID of the face that contains them.

5. There is a 1-1 mapping between edges of G^* and a subset of edges E_C of $\hat{G}$. Both endpoints of an edge in E_C know the weight and direction of its corresponding edge in G^* (if any).

4   Weighted Girth via Minor Aggregations

We provide an implementation of the minor-aggregation model in the dual graph, which immediately yields a collection of algorithmic results in the primal graph. In this section we provide an overview of the approach, where the full details and proofs appear in Appendix B.

In the Minor-Aggregation model, introduced in [42], there is a given graph G, where the vertices and edges of G are computational units, and an algorithm works in synchronous rounds, where in each round we can either contract some of the edges, or compute an aggregate function on certain sets of disjoint vertices or edges. For the full details of the model see Appendix B. As shown in [42], the minor-aggregation model can be simulated in CONGEST by solving the following part-wise aggregation problem.

Definition 4.1 (Part-Wise Aggregation (PA)). Consider a partition $\{S_1, \ldots, S_N\}$ of V where every $G[S_i]$ is connected. Assume each $v \in S_i$ has an $\tilde{O}(1)$-bit string x_v. The PA problem asks that each vertex of S_i knows the aggregate function $\bigoplus_{v \in S_i} x_v$.

An aggregate operator is a function that allows to replace $\tilde{O}(1)$-bit strings by one $\tilde{O}(1)$-bit string. This is usually a simple commutative function such as taking a minimum or a sum.

Minor-Aggregation for the dual graph. Our goal is to use the face disjoint graph $\hat{G}$ to simulate a minor-aggregation algorithm on the dual graph, where the vertices of a face simulate the corresponding dual node, and each dual edge is simulated by the endpoints of the corresponding primal edge. By the above discussion, our main goal is to show how to solve the PA problem on the dual graph.

We remark that Ghaffari and Parter [17] showed how to solve specific aggregate functions on G^* using the graph $\hat{G}$ (aggregations on each face of G and sub-tree sums on G^*). For our purposes, we need something more general, so we show how to perform general part-wise aggregations on G^*. In particular, we need to perform aggregations that take into consideration the outgoing edges of each part S_i, something that was not done in [17]. This specific task required our small modification to $\hat{G}$ compared to the one defined in [17].

To solve the PA problem on G^* we exploit the structure of $\hat{G}$. We prove that a PA problem in the dual graph G^* can be translated to a corresponding PA problem in $\hat{G}$. In the graph $\hat{G}$ there is a disjoint cycle representing each face of G, where two such cycles are connected if there is a dual edge between the corresponding faces. Hence, if we take a partition $\{S_1, \ldots, S_N\}$ of the dual nodes, such that each $G[S_i]$ is connected in the dual graph, we can convert it to a partition of the vertices of $\hat{G}$, where each dual node $f \in S_i$ is replaced by the vertices of the cycle corresponding to f in $\hat{G}$. Since cycles corresponding to different faces are vertex-disjoint in $\hat{G}$ we indeed get a partition.

In addition, by the properties of $\hat{G}$, it is a planar graph with diameter $O(D)$, and we can simulate CONGEST algorithms efficiently on $\hat{G}$. Based on this we can solve the PA problem in $\tilde{O}(D)$ rounds in $\hat{G}$, which leads to solving the PA problem on the dual graph G^* in $\tilde{O}(D)$ rounds. For full details see

Appendix B. We also prove that we can simulate an extended version of the minor-aggregation model defined in [18] that allows adding $\tilde{O}(1)$ virtual nodes to the network, which is useful for our applications.

Dealing with parallel edges in G^*. Recall the the dual graph G^* can be a multi-graph, where for some of our applications it is useful to think about it as a simple graph. For example, when computing shortest paths we would like to keep only the edge of minimum weight connecting two dual nodes, and when computing a minimum cut we can replace multiple parallel edges by one edge with the sum of weights. Note that if G^* was the network of communication it was trivial to replace multiple parallel edges by one edge locally, but in our case each edge adjacent to a face f can be known by a different vertex of f. While we can compute an aggregate operator of all the edges adjacent to f efficently, if a face f has many different neighboring faces g, we will need to compute an aggregate operator for each pair of neighboring faces which is too expensive. To overcome it, we use the low arboricity of planar graphs to compute a low out-degree orientation of the edges of the dual graph. This orientation guarantees that from each face f there are outgoing edges only to $O(1)$ other faces g, hence we can allow to compute an aggregate function on the edges between each such pair efficiently. The algorithm for computing the orientation is based on solving a series of $\tilde{O}(1)$ part-wise aggregation problems. For details see Appendix B.

Applications. By simulating minor-aggregation algorithms on the dual graph we can compute the weighted girth in $\tilde{O}(D)$ rounds by simulating a minimum weight cut algorithm on the dual graph and exploiting the cycle-cut duality of planar graphs (Fact 3.1). This approach allows us to find the weight of the minimum weight cycle, but we show that we can extend it to find also the edges of the cycle, for details see Appendix B. Another application is a $(1 - \epsilon)$-approximation for maximum st-flow when the graph is undirected and s, t lie on the same face with respect to the given planar embedding. Here, we simulate an approximate shortest path algorithm on a virtual graph obtained from the dual graph G^* by slight changes, for details see Section 6.

5 Dual Distance Labeling

In this section we show how to compute single source shortest paths (SSSP) in the dual graph G^* using a labeling scheme. I.e., we assign each dual node a label, such that, using the labels alone of any two dual nodes s and t, one can deduce the s-to-t distance in G^*. Notice that such labeling actually allows computation of all pairs shortest paths (APSP), i.e, we can solve SSSP from any source node by broadcasting its label to the entire graph. In this section, our algorithms are stated to be randomized, however, the only randomized component is an algorithm of [17] that we implicitly apply when using the BDD of [26] in a black-box manner. A derandomization for that algorithm would instantly imply a derandomization of our results in this section.

Section organization. First, in Section 5.1 we extend the analysis of the BDD, divided into three subsections, in each we provide the proofs of some new properties of the BDD. In particular, Section 5.1.1 focuses on analyzing the way faces get split, Section 5.1.2 focuses on the dual decomposition, and Section 5.1.3 shows how to learn that decomposition distributively. After which, we would be ready to formally define the labeling scheme in Section 5.2. Finally, in Section 5.3 we show an algorithm that constructs the labels and deduces an SSSP tree.

5.1 Extending the BDD

The Bounded Diameter Decomposition (BDD), introduced by Li and Parter [26], is a hierarchical decomposition of the planar graph G using cycle separators. It is a rooted tree $\mathcal{T}$ of depth $O(\log n)$ whose nodes (called *bags*) correspond to connected subgraphs of G with small diameter $\tilde{O}(D)$. The root of $\mathcal{T}$ corresponds to the entire graph G, and the leaves of $\mathcal{T}$ correspond to subgraphs of small size $\tilde{O}(D)$. In [26],

each bag X was defined as a subset of vertices. For our purposes, it is convenient to define X as a subset of edges (a subgraph). Given that we want to work with the dual graph G^*, the bijection between the edges of G^* and the edges of G allows us to move smoothly between working on G and G^*.

In this section, our goal is to extend the BDD of [26] to the dual network G^*. We stress that the extension to the BDD is not obtained by a simple black-box application of the BDD on G^* (for the two reasons mentioned earlier: G^* is not the network of communication, and G^*'s diameter might be much larger than the diameter of G). Instead, we use the same BDD $\mathcal{T}$ and carefully define the duals X^* of bags X. We begin with the following lemma summarizing the properties of the BDD of [26].

Lemma 5.1 (Bounded Diameter Decomposition [26])**.** *Let G be an embedded planar graph with hop diameter D. There is a distributed randomized $\tilde{O}(D)$-rounds algorithm that w.h.p. computes a bounded diameter decomposition $\mathcal{T}$ of G, satisfying:*

1. *$\mathcal{T}$ is of depth $O(\log n)$.*

2. *The root of $\mathcal{T}$ corresponds to the graph G.*

3. *The leaves of $\mathcal{T}$ correspond to graphs of size $O(D \log n)$.*

4. *For a non-leaf bag X, let S_x be the set of vertices of X which are present in more than one child bag of X. Then, $|S_x| = O(D \log n)$.*

5. *For every bag X, the diameter of X is at most $O(D \log n)$.*

6. *For every bag X with child bags $X_1, X_2, \ldots$, we have $X = \cup_i X_i$.*

7. *Each edge e of G is in at most two distinct bags of the same level of $\mathcal{T}$.*

8. *Every bag has a unique $\tilde{O}(1)$-bit ID. Every vertex knows the IDs of all bags containing it.*

By the construction of the BDD, the set S_x plays the role of a cycle separator of X.

Since our focus is on the dual graph, we need to understand how faces of G are affected by the BDD. Namely, decomposing a bag X into child bags $X_1, X_2, \ldots$ might partition some faces of X into multiple parts, where each such *face-part* is contained in a distinct child bag of X. We next discuss this in detail. Since each edge belongs to two faces of G, it is convenient to view an edge as two copies (called *darts*) in opposite directions. This is fairly common in the centralized literature on planar graphs.
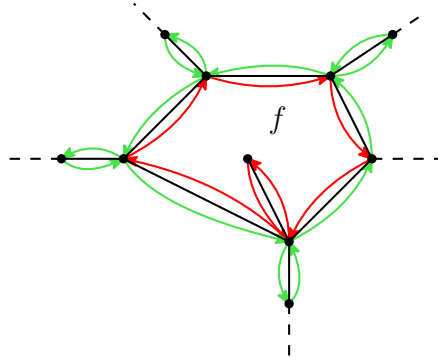

Figure 9: Edges are black, each edge has two darts (green or red). The red clockwise cycle of darts corresponds to the face f.

Darts. Every edge $e \in E(X)$ is represented by two darts d^- and d^+ (if the graph is directed, d^+ has the same direction as e, and d^- is in the opposite direction). The reverse dart is defined as $\text{rev}(d^+) = d^-$

16

and $\text{rev}(d^-) = d^+$. We think of the two darts as embedded one on top of the other (i.e., replacing the edges with darts does not create new faces). When we mention a path or a cycle C of darts, we denote by $\text{rev}(C)$ the path or cycle obtained by reversing all darts of C. The faces of G define a partition over the set of darts (i.e., each dart is in exactly one face) such that each face of G is a cycle of clockwise directed darts. See Figure 9.

Face-parts. Ideally, we would like each face of a bag X to be entirely contained in one of its child bags. However, it is possible that a face f is split into multiple child bags, each containing a subset of f's darts (which we call a *face-part*). This issue was not discussed in [26] as they work with the primal graph G.

We keep track of the faces of G and their partition into face-parts. In G, each face is a cycle of directed darts. A face-part in a bag X is identified by a collection of directed paths of darts, all belonging to the same face of G. Notice that a face-part might be disconnected. A bag X therefore contains darts that form faces of G that are entirely contained in X, and darts that form face-parts. If a dart d belongs to X but $\text{rev}(d)$ does not belong to X, we say that $\text{rev}(d)$ lies on a *hole* (rather than a face or a face-part) of X. This happens when d is a dart of S_H for some ancestor bag H of X.

We think of obtaining face-parts as a recursive process that starts from the root of $\mathcal{T}$. In each level of $\mathcal{T}$, each bag X has its faces of G and face-parts of G. Both of these may get partitioned in the next level between child bags of X, resulting with new (smaller) face-parts each contained in one child bag of X.

Dual bags. We define the dual bag X^* of a bag X to be the dual graph of the graph X when treating face-parts (and not only faces) of G in X as nodes of X^*. Formally, each face or face-part f of G in X has a corresponding dual node f^* in X^*. The dual edges e^* incident to f^* in X^* are the edges e (collection of paths) of X that define f and have both their darts in X. Recall that each dart of e is part of exactly one face or face-part, and the dual edge e^* connects the corresponding dual nodes (if only one dart of e is in X then e lies on a hole of X and we do not consider holes as dual nodes). The weight of e^* is the same as that of e and its direction is defined as in Section 3. For an illustration see Figure 10. When discussing dual bags, we refer to a node that corresponds to a face-part as a *node-part*.

Ideally, we would like the child bags X_i^* to be subgraphs of X^*. If that were true, then X^* would be the union of all X_i^* plus the edges dual to S_X (since they are the only edges in more than a bag of the same level). This is almost the case, except that X^* may also have as many as $O(\log n)$ (we prove this next) faces and face-parts that are further partitioned into smaller face-parts in the child bags X_i^*. I.e., the union of all X_i^* has a different node set than that of X^*. Thus, in X^* we have $O(\log n)$ nodes that might be divided to multiple nodes, each in some child bag X_i^*.

We are now ready to prove the following new properties of the BDD.

Theorem 5.2 (BDD Additional Properties). *Let $\mathcal{T}$ be a Bounded Diameter Decomposition of an embedded planar graph G with hop-diameter D. Within additional $\tilde{O}(D)$ rounds, $\mathcal{T}$ satisfies the following additional properties.*

Few face-parts property:

9. *Each bag X contains at most $O(\log n)$ face-parts. Moreover, the number of face-parts resulting from partitioning faces and face-parts of X across its child bags is $O(D \log^2 n)$.*

Dual bags' properties:

10. *The dual bags of leaves of $\mathcal{T}$ are of size $O(D \log n)$.*

11. *For a non-leaf bag X^*, let F_X be the set of nodes whose incident edges are not contained in a single child bag of X^*. Then, F_X is a node-cut (separator) of X^* of size $|F_X| = O(D \log n)$. Concretely, F_X is the set of (a) nodes incident to dual edges of S_X, and (b) nodes corresponding to faces or face-parts that are partitioned between child bags of X^*.*

17
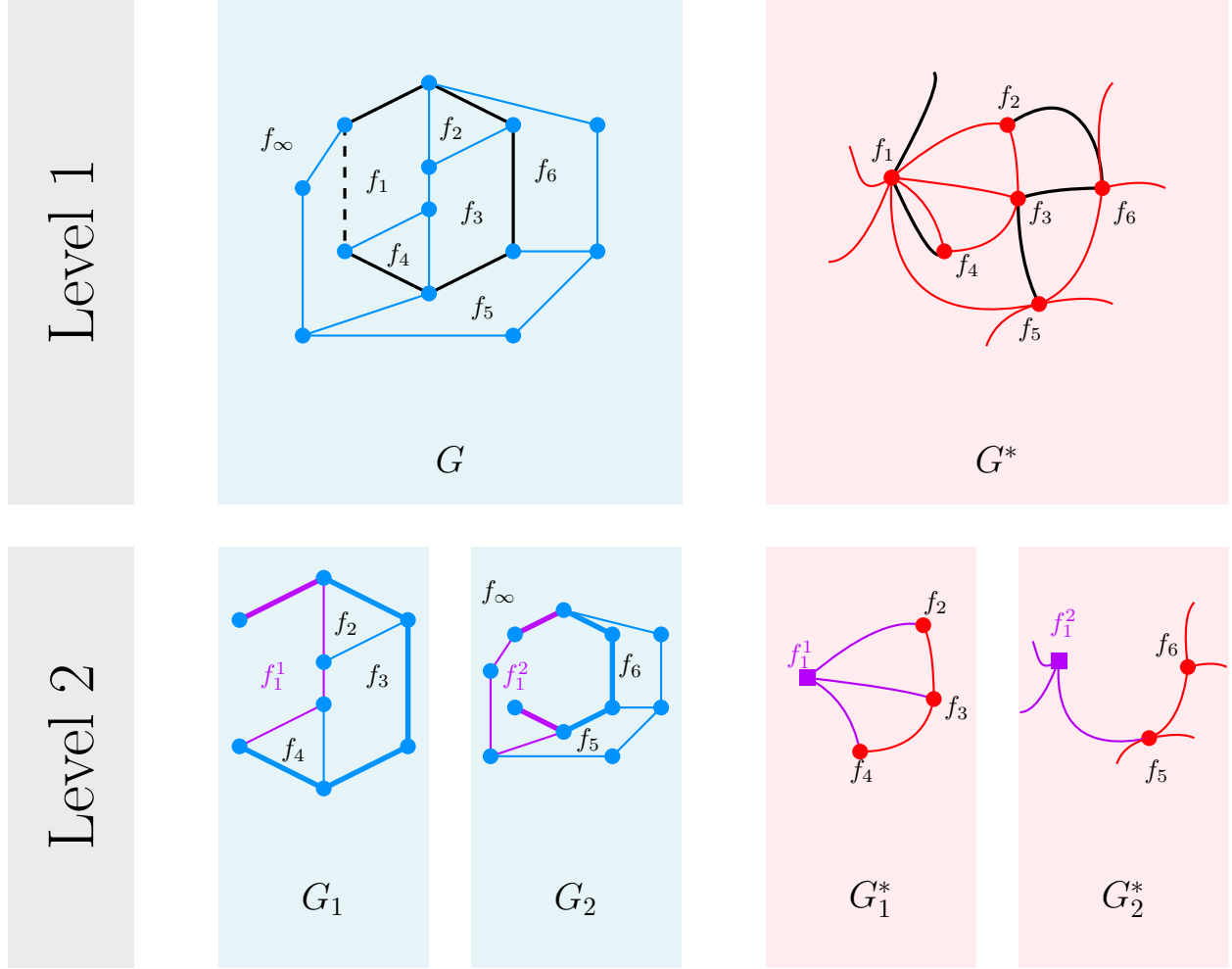
Figure 10: Two levels of the BDD (blue) and their corresponding levels in the dual decomposition (red). For simplicity, the dual node corresponding to the face f_∞ of G is not illustrated in G^* (its incident edges are the ones missing an endpoint).

First level: the graph G and its dual G^*. Black edges are the separator S_G edges, the dashed one is the virtual edge e_G. Note, e_G splits the critical face, f_1.

Second level: the child bags G_1, G_2 of G and their corresponding dual bags G_1^*, G_2^* (respectively). Purple edges in G_1 and G_2 are edges of the critical face f_1 of G. In G_1, those edges define the face-part f_1^1 and in G_2 they define the face-part f_1^2. Both face-parts are represented by dual nodes (squared purple). Bold edges in G_1, G_2 are edges of S_G and non of them has a dual edge in G_1^*, G_2^* as they lie on a hole. Each other edge has a dual edge, in particular, edges of f_1^1 (f_1^2) connect them to their neighbors from G^* in G_1^* (G_2^*).

12. *Let X^* be a non-leaf bag with child bags $X_1^*, X_2^*, \ldots$; Then, X^* is equivalent to $\cup_i X_i^* \cup X^*[S_x]$ after connecting all node-parts $f_1, f_2, \ldots$ in $X_1^*, X_2^*, \ldots$ (resp.) corresponding to the same $f \in F_x$ with a clique and contracting it.* [5]

Distributed knowledge:

---
[5]For an intuition, see Figure 10: merging f_1^1 and f_1^2 to one node would result with G^* after adding the separator edges.

13. *Each node f of X* has an $\tilde{O}(1)$ ID, s.t. the vertices v of X lying on f know its ID. Moreover, v knows whether f corresponds to a face, a face-part or the critical face of X. The IDs are such that the ID of a face (face-part) f of X is contained in the ID of face-parts f' resulting from partitioning f in child bags of X.*

14. *For all bags X, its vertices know the dual edges in X* (if exist) corresponding to each of their incident edges.*

5.1.1 The Few Face-parts Property

We start by proving Property 9.

Lemma 5.3 (Few face-parts). *Any bag $X \in \mathcal{T}$ contains at most $O(\log n)$ face-parts. Moreover, there is at most one face of G that is entirely contained in X and is partitioned in between child bags of X.*

Proof. Since the height of $\mathcal{T}$ is $O(\log n)$, it is sufficient to prove that in each bag X of $\mathcal{T}$ there is at most one face of G that is entirely contained in X and is partitioned into face-parts in X's child bags.[6] This would imply that each face-part of X can be associated with the unique leafmost ancestor of X in which this face was whole (and thus there are only $O(\log n)$ face-parts in X). Note, in bags X of deeper recursion levels, a face-part might consist of multiple disconnected paths, we still count it as one face-part of X since all of those paths belong to the same face of G.

To show that indeed there is only one such face, we need to understand the cases in which faces are partitioned in the BDD of [26]. The set S_X in a bag $X \in \mathcal{T}$ is a simple (unoriented) cycle consisting of two paths in a spanning tree T of X and an additional edge e_X. The edge e_X might not be an edge of $E(G)$, in which case, we refer to it as the *virtual* edge of X. We think of e_X as if it is temporarily added to X, in order to define its child bags, and then removed. The cycle S_X is a *balanced* cycle separator, i.e. a cycle that contains a constant factor of $V(G)$ in each of its interior and exterior.

The edge e_X is found in [17] by carefully identifying two vertices u, v, then setting $e_X = (u, v)$ closing a cycle with T. e_X is embedded without violating planarity. Given the two endpoints and the embedding of e_X, one can identify the *critical* face (face-part) f that contains e_X by simply considering the face (face-part) that contains the two edges in the local ordering of v which e_X is embedded in between.

The cycle S_X defines the child bags of X as follows (see Figure 11): The interior of the cycle S_X defines one child bag (denoted X_1) and the exterior may define several child bags. Each external child bag X_i is identified with a unique (internally disjoint) subpath S_i of S_X. That is, the only edges that belong to more than one X_i are the edges of S_X. For each such edge, one of its darts d belongs to X_1 and $\text{rev}(d)$ belongs to some $X_i \neq X_1$ (note that $\text{rev}(d)$ belongs to a hole in X_1 and that d belongs to a hole in X_i). Note that every path between a vertex of X_i and a vertex of $X_j \neq X_i$ must intersect a vertex of S_i (Property 4 of the BDD).

The critical face f is partitioned (by e_X) into two face-parts. One part goes to X_1 and the other to an external child bag $X_i \neq X_1$. If f is a face of G, then we need to count the two new face-parts that it generates. If f is a face-part, then we don't need to count them (since we only care about faces of G that get partitioned). However, in both cases we need to show that, except for f, no other faces can be partitioned between the child bags of X. For that we dive a bit more into the way the child bags are defined in [26].

Case I: When $e_X \in E(G)$. In this case we claim that no face of G gets partitioned among X's children. Assume for the contrary that some face g of G gets split between some X_i and X_j. The face g must

---

[6]In [26], a single hole of X (the union of some S_H edges of ancestors H of X, referred to in [26] as the *boundary* of X) might be partitioned between X's child bags. In our case, since we do not care about holes, we do not count this as a face (or a face-part) that gets partitioned.
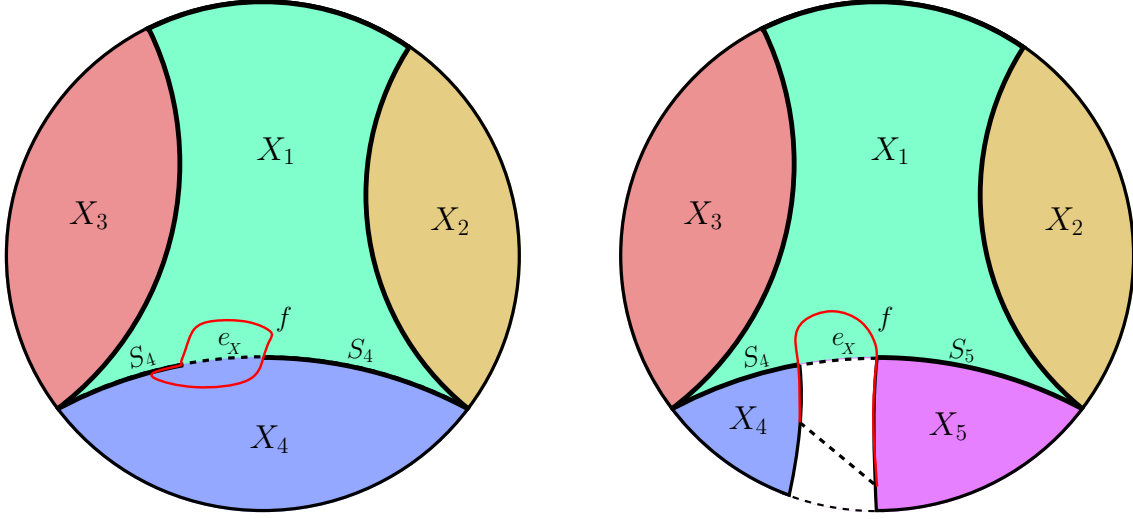
Figure 11: Two examples of a bag X. The bold cycle is S_x, and the edge e_x is dashed. Distinct child bags of X are of distinct colors. The child X_1 (green) is the interior of S_x. The external child bags X_i are each defined by a single subpath S_i of S_x. One such subpath may contain e_x (S_4 in the left image) if its external part (bag X_4 in the left image) is connected. Otherwise, the external part is broken into two (connected) bags (X_4, X_5 in the right image), each with its own S_i that does not include e_x (the additional dashed edges are some ancestor virtual edges). In the left (resp. right) image, X_1 and X_4 (resp. X_1, X_4, X_5) contain a face-part of the critical face (face-part) f (in red).

contain a dart (a, b) in X_i and a dart (c, d) in X_j such that neither (a, b) nor (c, d) is a dart of S_i. Since g is a cycle of darts, it is composed of $(a, b), (c, d)$, and two dart-disjoint b-to-c and d-to-a paths. Each of these two paths must intersect the vertices of S_i, and so g contains two vertices x, y of S_i.

I.a. If g has some dart internal to S_x (w.l.o.g. (a, b) in $X_j = X_1$) then, g is a cycle that consists of two x-to-y paths, one is internal to S_x (in X_1) and the other is external to S_x, thus, g must enclose the x-to-y subpath of S_i constituting a chord embedded inside the cycle g, contradicting that it is a face.

I.b. If g has no dart internal to S_x, then all of its darts are in external child bags (g has at least two darts each in a distinct external child bag). If g encloses a subpath of S_i then we have our contradiction. Otherwise, g does not enclose anything and its reversal $\text{rev}(g)$ encloses the entire graph. In that case, g has to contain the first and last vertex in all the subpaths S_i for all external bags X_i (since from the construction of the BDD, these are the only vertices of X_i that can be shared with another external child bag, and there is no edge connecting two different child bags). Hence, the only way to close a cycle g that contains darts from different child bags is to contain all these vertices. There is only one face of X that can do that, which by [26] is a hole. In fact, the endpoints of all S_i are detected in the algorithm of [26] by looking for intersections between g (referred to as the *boundary* in their paper) and S_x. g is a hole because in [26] the boundary is defined as the set of S_H edges contained in X, where $H \neq X$ is any ancestor of X in $\mathcal{T}$.

Case II: When $e_x \notin E(G)$. In this case, there might be one child bag $X_i \neq X_1$ whose corresponding subpath S_i contains e_x (in which case, the above proof of Case I fails as the x-to-y subpath might be missing an edge), meaning there is a face that gets partitioned. We argue that there is only one such face,

the critical face, and that this can only happen (see Figure 11 left) if the endpoints of e_x are connected in some $X_i \neq X_1$.

II.a. When the endpoints of e_x are connected in some $X_i \neq X_1$, then consider what happens if we add e_x as a real edge of $E(G)$ and then remove it. By the proof of Case I, when e_x is in $E(G)$, no face gets partitioned. Then, when we remove e_x, the only face that can be affected is the one enclosing both endpoints of e_x (i.e. the critical face). Again there might be a hole that gets partitioned (exactly as in Case I) that we do not care about.

II.b. When the endpoints of e_x are not connected in any external bag, then no S_i contains both endpoints of e_x. This can only happen if the critical face f is a face-part (see Figure 11 right), for otherwise the endpoints of e_x would be connected by f. In this case, the critical face-part f is partitioned between three (and not two) child bags (X_1 and two external child bags). However, apart from f, no other face gets partitioned. As in Case I, there is a dart of g in X_1, and then we can find a subpath of S_x that crosses g, a contradiction. The proof follows the same proof as in Case I (all S_i consist of G edges); Otherwise, if no dart of g is in X_1, then g is not a cycle, a contradiction. That is, g has to use the endpoints of all S_i and somehow connect them to form a cycle, but the endpoints of e_x are only connected in X_1. ◻

The following completes the proof of Property 9.

Corollary 5.4. *The total number of face-parts resulting from partitioning faces and face-parts of a bag $X \in \mathcal{T}$ between its child bags is $O(D \log^2 n)$.*

Proof. By Lemma 5.3, in a bag X, there are at most $O(\log n)$ face-parts and one critical face that get partitioned between child bags of X. We claim that any bag X has at most $O(D \log n)$ child bags (and the corollary follows). To see why, recall that S_x is a cycle separator of X s.t its interior defines a child bag, and each external child bag is identified with a distinct (edge-disjoint) subpath of S_x. By Property 4 of the BDD we have $|S_x| = O(D \log n)$. ◻

We prove the following, which will be useful in the labeling scheme and in proving properties 11 and 12 of the BDD.

Lemma 5.5. *Each dart d of G belongs to exactly one bag X in each level of $\mathcal{T}$. Moreover, if a dart d is in X and $\mathrm{rev}(d)$ is not in X then d belongs to S_H for some ancestor H of X.*

Proof. In the root of $\mathcal{T}$, we have the graph G and the claim trivially holds. Assume the claim holds for level $\ell - 1$ of the BDD, we prove for level ℓ. Recall that for a bag X, every edge e of X either belongs to one child bag of X or to two child bags of X (if $e \in S_x$). If e is not an edge of S_x, then e belongs to one child bag X_i and its (one or two) darts in X belong to X_i (as both of e's endpoints are in X_i). If e is an edge of S_x that is contained only in one child bag, then by the BDD algorithm e must be an edge of S_H for some ancestor $H \neq X$ of X. Thus, e's only dart is also contained in the same child bag that contains e. Finally, if e is an edge of S_x that is contained in two child bags of S_x, then e is in X and does not participate in S_H of any ancestor H, then, e has both of its darts in X. Thus, e belongs to two child bags of X, in particular, one child bag is the internal (to S_x) and the other is external, each would contain one dart of e as each dart of e is in a face (not a hole), and in this case, S_x separates those two faces (face-parts), each to a child bag. That way, all darts that correspond to the same face go to the same child bag of X. ◻

Corollary 5.6. *Each face of G in a bag $X \in \mathcal{T}$, except for the critical face (if exists), is entirely contained in one child bag of X. Face-parts of X may get further partitioned between its child bags.*

5.1.2 Dual Bags and Their Properties

We are finally prepared to prove the additional properties 10, 11, and 12 of the BDD, starting with a proof for Property 10.

Lemma 5.7. *The size of a dual leaf bag X^* is $O(D \log n)$.*

Proof. By Property 3 of the BDD, we know that X has $O(D \log n)$ vertices. Since X is a simple planar graph, it therefore has at most $O(D \log n)$ edges (Euler formula). For each edge e in X there is at most one dual edge e^* in X^* as we define it. Then, $|E(X^*)| = O(D \log n)$, thus, X^* has at most $O(D \log n)$ nodes as well (as each edge has two endpoints). Notice, there are no isolated nodes (i.e, we are not under-counting), as if there were, then they are not connected to any edge, and the only case that a primal edge has no dual is when it lies on a hole, thus, if there were isolated nodes, they would correspond to holes, contradicting the definition of X^*. $\qquad\square$

Next, we prove Property 11.

Lemma 5.8 (Dual Separator). *For a non-leaf bag X^*, let F_x be the set of nodes whose incident edges are not contained in a single child bag of X^*. Then, F_x is a node-cut (separator) of X^* of size $|F_x| = O(D \log n)$. Concretely, F_x is the set of (a) nodes incident to dual edges of S_x, and (b) nodes corresponding to faces or face-parts that are partitioned between child bags of X^*.*

Proof. First, we show that both definitions of F_x are equivalent. I.e., a node (node-part) of X^* does not have all of its edges in a single child bag if and only if it is contained in: (a) The set of X^*'s nodes (node-parts) which get partitioned further into node-parts between its child bags, or (b) The endpoints of edges dual to S_x contained in X^*.

(if) Note, the dual edges of S_x contained in X^* are not contained in any child bag of it as they lie on holes in them (Lemma 5.5), thus, their endpoints are in F_x. In addition, the nodes of X^* that get divided correspond to the critical face and a subset of face-parts in X (Corollary 5.6), thus, they have at least two distinct edges, each in a distinct child bag of X^*, hence, are in F_x.

(only if) Consider a node $f \in F_x$, by definition, some of its incident edges either: (1) Not contained in any child bag of X^*, i.e, f is incident to an S_x edge. Or (2) there are at least two edges incident to f, each in a distinct child bag of X^*. Since a face of G in X is either fully contained in a child bag or is divided across child bags (Corollary 5.6), we get that f corresponds to the critical face or to one of the face-parts in X.

From the first definition of F_x, it is clear that it constitutes a separating set, as each path that crosses between child bags shall use at least two edges that share a node of X^* which are not contained in a single child bag. I.e, that node is in F_x.

Finally, since there are at most $O(D \log n)$ edges of S_x (Property 4 of the BDD), there are at most twice the number of their endpoints in X^*. Moreover, there are at most $O(\log n)$ nodes (node-parts) in X^* that correspond to the faces and face-parts that get partitioned between child bags of X^* (Property 9 of the BDD). $\qquad\square$

Finally, we provide a proof of Property 12.

Lemma 5.9 (Assembling X^* from child bags). *Let X^* be a non-leaf bag with child bags $X_1^*, X_2^*, \ldots$; Then, X^* is equivalent to $\cup_i X_i^* \cup X^*[S_x]$ after connecting all node-parts $f_1, f_2, \ldots$ in $X_1^*, X_2^*, \ldots$ (resp.) corresponding to the same $f \in F_x$ with a clique and contracting it.*

Proof. Let K_f be the clique on the set of node-parts across child bags of X^* corresponding to $f \in F_x$. Denote $M := (\cup_i X_i^*) \bigcup (X^*[S_x]) \bigcup \left(\cup_{f \in F_x} K_f \right)$. We start by showing that M is very close to X^*, then show that the adjustment (contraction) suggested in the property on M indeed yields X^*.

22

First, note that all edges of X^* are contained in M, as any non-S_X edge is contained in a child bag (Lemma 5.5) and S_X edges that are contained in X^* are added to M by definition. In addition, each node of X^*, except for some nodes of F_X, is present in M, as each node that is in some X_i^* is contained in M by definition, and any node of X^* that is not in any X_i^* must be an F_X node (by Property 11).

Now, we show that the contraction produces F_X nodes, connects them correctly and removes only non-X^* edges from M (the lemma follows). Consider the graph M', given by M after contracting K_f for all $f \in F_X$.

1. The contraction gets rid of K_f edges, and only them, as distinct node-parts across child bags of X^* that correspond to the same $f \in F_X$ share no other edges. Thus, M' has the exact same set of edges as X^*.

2. Consider a node f that was partitioned into node-parts $f_1, f_2, \ldots$, such that each f_i is in a distinct child bag of X^*. We want to show that contracting all f_i into one node, results with one node that is incident to all f's X^* neighbors in M', this node is identified with f. That follows from the following claims (1) Each edge of f in X goes to one face-part of f in the child bag that contains it (by definition of face-parts). (2) The only edges without a dual in X^* are edges of S_H for some ancestor H of X (Lemma 5.5). Thus, if a primal edge does not have a dual in any of X^*'s child bags, it is either an edge of $S_H \neq S_X$, in which case it is not in X^* either (the claim does not fail), or it is in S_X, thus, it is added to M by definition. Thus, f's incident edges in X^* are partitioned among f_i's in M. Moreover, by the previous item, the contraction affects none of these edges and leaves only them incident to the resulting node, which we identify with f. □

5.1.3 Distributed Knowledge Properties

To implement our algorithm, each vertex should know the IDs of all faces (face-parts) that contain it in every bag $X \in \mathcal{T}$ (i.e., the list of nodes $f \in X^*$ it participates in). Here we show that the distributed knowledge properties (13 and 14) of the BDD hold, which concludes the proof of Theorem 5.2. As before, it will be convenient to think of an edge as having two *copies* (one for each face containing it). To simplify, we refer to these as edge copies (rather than darts) since there is no need to maintain the dart orientation. Every vertex incident to an edge $e = (u, v)$ considers itself incident to two copies e_1, e_2 of e, such that, e_1 (e_2) is the *first* (*second*) copy of e relative to v according to a clockwise order (for the other endpoint u, e_2 is its first copy).

We aim to assign each face (face-part) in any bag X a unique $\tilde{O}(1)$ identifier and learn for each vertex $v \in X$, the set of faces (face-parts) containing it and its adjacent edge copies. This is done by keeping track of G's edge copies along the decomposition. We start with (1) a local procedure that allows the endpoints of each edge copy e to learn all the bags X that contain e, then, (2) the endpoints of each edge copy e learn the face of G that e participates in, and (3) we show how to extend this for the next level of $\mathcal{T}$. For a proof see Appendix C. This allows us to (distributively) learn the nodes of X^*, after which, we have the necessary information to learn the edges of X^*.

Lemma 5.10. *In a single round, each vertex $v \in G$ can learn, for each incident edge copy, a list of the bags $X \in \mathcal{T}$ that contain it.*

Lemma 5.11. *There is a $\tilde{O}(D)$-round distributed algorithm that assigns unique IDs to all faces of G. By the end of the algorithm, each vertex $v \in G$ learns a list (of IDs) of faces that contain it. Moreover, v learns for each incident copy e' of an edge e in G the face ID of the face of G that contains e'.*

The IDs assigned to faces and face parts are defined such that the ID of a face (face-part) f of X is contained in the ID of face-parts resulting from partitioning f in child bags of X. Thus, we get the following lemma and corollary proving property 13 of the BDD:

Lemma 5.12 (Distributed Knowledge of Faces). *There is a $\tilde{O}(D)$-round distributed algorithm that assigns unique $\tilde{O}(1)$-bit IDs to all faces and face-parts in all bags $X \in \mathcal{T}$. By the end of the algorithm, each vertex $v \in G$ learns a list (of IDs) of the faces and face-parts it lies on for each bag X that contains it. Moreover, v learns for each incident copy of an edge e' the (ID of) the face or face-part that e' participates in for each bag X that contains e'. The ID of a face (face-part) f of X is contained in the ID of face-parts f' resulting from partitioning f in child bags of X.*

Corollary 5.13. *In $\tilde{O}(D)$ rounds, for each bag X, all vertices know for each of their incident copies of edges whether they participate in the critical face (face-part) and whether they participate in any face-part of X.*

After each vertex v knows the necessary information for each copy of an incident edge e, in an additional round, v knows if e exists in X^*, its weight in X^* and its direction. Proving property 14 of the BDD.

Lemma 5.14. *Let $e = (u, v)$ be an edge in a bag X. In one round after applying Lemma 5.12, the endpoints of e learn the corresponding dual edge e^* in X^* (if exists).*

5.2 The Labeling Scheme

Here we describe our distance labeling scheme. Recall that, for every dual bag X^*, we wish to label the nodes of X^*, such that, from the labels alone of any two nodes we can deduce their distance in X^*. Recall, we refer to the dual node f' in a child bag X_i^* as a *node-part* of a node $f \in X^*$, if f' corresponds to a face-part in X_i resulting from partitioning the face or face-part f of X.

We already know that this set would play the role of a node-cut (separator) of X^*, and is crucial for the labeling scheme and algorithm. The main structural property that F_X provides is: Any path in X^* either intersects F_X or is entirely contained in a child bag of X^*.

Note, a node $g \notin F_X$ of X^* corresponds to real faces of G (i.e. not face-parts) contained in X. The distance label $\mathsf{Label}_{X^*}(g)$ of $g \notin F_X$ is defined recursively. If X^* is a leaf-bag then $\mathsf{Label}_{X^*}(g)$ stores $ID(g)$ and the distances between g and all other nodes $h \in X^*$. Otherwise, the label consists of the ID of X, the distances in X^* between g and all nodes of F_X, and (recursively) the label of g in the unique (by Corollary 5.6) child bag X_i^* of X^* that entirely contains g. Formally:

$$\mathsf{Label}_{X^*}(g) = (ID(X), ID(g), \{ID(f), \mathsf{dist}_{X^*}(f, g), \mathsf{dist}_{X^*}(g, f) : f \in F_X\}, \mathsf{Label}_{X_i^*}(g))$$

For the set of nodes $g \in F_X$, we remove the recursive part, and define

$$\mathsf{Label}_{X^*}(g) = (ID(X), ID(g), \{ID(f), \mathsf{dist}_{X^*}(f, g), \mathsf{dist}_{X^*}(g, f) : f \in F_X\})$$

As we showed in property 11, faces and face-parts that correspond to nodes in F_X are the interface of the child bags of X^* with each other.

Lemma 5.15 (Property 11 of the BDD, rephrased). *Let g, h be two nodes in a non-leaf bag X^*, the (edges of the) shortest g-to-h path in X^* is either entirely contained in some X_i^* or intersects F_X.*

Thus, a path in X^* from $g \in X_i^*$ to $h \in X_{j \neq i}^*$ must pass through some (part of a) node $f \in F_X$. I.e., a g-to-h shortest path P in X^* is either: (1) Entirely contained in X_i^*, then $\mathsf{dist}_{X^*}(g, h)$ can be deduced from $\mathsf{Label}_{X_i^*}(g)$ and $\mathsf{Label}_{X_i^*}(h)$. Or (2), P intersects F_X, then $\mathsf{dist}_{X^*}(g, h) = \min_{f \in F_X}\{\mathsf{dist}_{X^*}(g, f) + \mathsf{dist}_{X^*}(f, h)\}$. If one of g or h is a node of F_X then distances are retrieved instantly, as each node in X^* stores its distances to F_X.

By this reasoning, obtaining the distance between two nodes g, h in a leaf bag is trivial, as all pairwise distances are saved in each label. For a non-leaf bag X^*, by Lemma 5.15, the information saved in the labels of g, h in X^* is sufficient in order to obtain the g-to-h distances in X^*.

Lemma 5.16. *Let g, h be two nodes in X^*, then we can decode their distance in X^* from $\mathsf{Label}_{X^*}(g)$ and $\mathsf{Label}_{X^*}(h)$ alone.*

Note, we define the labels in $O(\log n)$ steps from the leaf to the root of $\mathcal{T}$, in each step, we append to the label at most $\tilde{O}(D)$ bits for distances from F_x nodes. I.e., we have $|F_x| = \tilde{O}(D)$ (Property 11 of the BDD), the weights are polynomial, the size of a leaf bag is $O(D \log n)$ (Property 10 of the BDD) and IDs are of $\tilde{O}(1)$-bits (Property 8 and Property 13). Hence,

Lemma 5.17. *The label of any node g in a bag X^* of $X \in \mathcal{T}$ is of size $\tilde{O}(D)$.*

5.3 The Labeling Algorithm

Here we show how to compute the labels of all nodes in all bags X^* of $\mathcal{T}$. At the end of the algorithm, each vertex in a face g of X shall know the label of g in X^*. The planar network of communication G is assumed to be directed and weighted, let $w(e)$ denote the weight of an edge e. The BDD tree $\mathcal{T}$ of G is computed in $\tilde{O}(D)$ rounds, by Lemma 5.1 and Theorem 5.2, such that, each (primal) vertex v knows all bags X that contain it, the IDs of all faces (face-parts) of X that contain it, and the dual edges corresponding to each of its incident edges in X (if any).

We compute the label of every dual node $g \in X^*$ for all $X^* \in \mathcal{T}$ in a bottom-up fashion, starting with the leaf-bags. For a non-leaf bag X^*, we have two main steps: (i) broadcasting labels of (parts of the dual separator) F_x nodes contained in the child bags of X^*, broadcasting the edges dual to (the primal separator) S_x and (ii) using the received information locally in each vertex of X for computing labels in X^*. For a high-level description, see Algorithm 1.

Algorithm 1: Dual Distance Labeling (high-level description)

Input: A BDD of a D-diameter network G satisfying the properties of Theorem 5.2.
Result: Each vertex v of G learns $\mathsf{Label}(g)$ in G^* for each face g of G that contains it.

1 **forall** *bags X of the same level, bottom-up, in parallel,* **do**
2   **if** *X is a leaf-bag* **then**
3   Collect the entire graph X in each vertex $v \in X$;
4   **end**
5   **else**   `/* Non-leaf bags */`
                                             `/* Broadcast step */`
6   Broadcast the dual edges of S_x edges that exist in X^* to the entire bag X;
7   Broadcast, in two steps, the labels computed in the previous level (in child bags of X^*) of faces (face-parts) that correspond to F_x to the entire bag X.
                                             `/* Local Step */`
8   Each vertex $v \in X$ uses the received information in the above broadcast in order to locally construct the Dense Distance Graph for each face (face-part) g that contains it in X;
9   **end**
10   Using the previously received information, each vertex $v \in X$ computes locally $\mathsf{Label}_{X^*}(g)$ for each face (face-part) g of X that contains it or reports a negative cycle aborting the algorithm;
11 **end**

Leaf bags. For leaf-bags X^*, we show that we can broadcast the entire bag X^* in $\tilde{O}(D)$ rounds. First, by Property 13, each vertex of X knows all the IDs of faces of X (both real faces of G and face-parts) that contain it. These faces are exactly the nodes of the graph X^*, by Property 10, $|X^*| = \tilde{O}(D)$. That is, we

can broadcast them to the whole bag in $\tilde{O}(D)$ rounds. Note that it may be the case that several different vertices broadcast the same ID, but since overall we have $\tilde{O}(D)$ distinct messages, we can use pipelined broadcast to broadcast all the IDs in $\tilde{O}(D)$ rounds. We can work simultaneously in all leaf bags, as each edge of G is contained in at most two bags of the same level (Property 7 of the BDD). Next, we broadcast the edges of X^*. From Property 14, for each edge e in X the endpoints of the edge know if this edge exists in X^*, and if so they know the corresponding dual edge in the dual bag X^*. We let the endpoint of e with smaller ID broadcast a tuple $(ID(e^*), w(e))$ to the bag, where $ID(e^*) = (ID(f), ID(g))$ if e^* is directed towards g in X^*. Notice, X^* might be a multi-graph, thus, there might be multiple edges with the same ID of e^*. This does not require any special attention, we broadcast all of them as the total number of edges in X^* is conveniently bounded. Since for each edge we send $\tilde{O}(1)$-bits and in total we have $\tilde{O}(D)$ edges in X^* (Property 10), the broadcast terminates within $\tilde{O}(D)$ rounds. Afterwards, all the vertices of X know the complete structure of X^*. Then, all vertices of X can locally compute the labels of nodes of X^* by locally computing the distances in the graph X^*. Finally, since vertices have full knowledge of X^*, they can check for negative cycles, if found, a special message is sent over a BFS tree of G, informing all vertices to abort.

Non-leaf bags. We assume we have computed all labels of bags in level ℓ of $\mathcal{T}$, and show how to compute the labels of all bags in level $\ell - 1$ simultaneously in $\tilde{O}(D^2)$ rounds. We describe the procedure for a single bag X^* with child bags $X_1^*, X_2^*, \ldots$, by Property 7 of the BDD we can apply the same procedure on all bags of the same level without incurring more than a constant overhead in the total round complexity. The computation is done in two steps, a broadcast step and a local step.

Broadcast step. We first broadcast the following information regarding F_X: (1) The edges of S_X contained in X^*, (2) The labels of nodes (node-parts) of F_X in the child bags of X^*. I.e, if $f \in F_X$ is entirely contained in a child bag X_i^* of X^*, we broadcast $\text{Label}_{X_i^*}(f)$. Otherwise, f is the critical face of X or a face-part of X, f may get partitioned to $f_1, f_2, \ldots, f_k$ in between the child bags of X. Thus, for each such $f \in F_X$, we broadcast the labels $\text{Label}_{X_j^*}(f_i)$, where X_j^* is the child bag that contains the node-part f_i. Using this information, we shall compute the distances from any node g to all nodes of F_X. By induction, for each face (face-part) f of a child bag X_i, every vertex v that lies on f knows the label $\text{Label}_{X_i^*}(f)$ of f in X_i^*.

1. Since each (primal) vertex v knows its incident S_X edges (by the BDD construction), v also knows for each such incident edge whether its dual is in X^* or not (By Property 14). Thus, we can broadcast edges in S_X contained in X^* to the whole bag X. For each edge, one of its endpoints (say the one with smaller ID) broadcasts the edge (i.e., an $\tilde{O}(1)$-bit tuple $(ID(e^*), w(e))$). The broadcast terminates in $\tilde{O}(D)$ rounds as $|S_X| = \tilde{O}(D)$ (Property 4 of the BDD).

Next, we explain how to broadcast labels of nodes (node-parts) that correspond to F_X in child bags of X^*. We do that in two steps, a step for nodes of F_X that get partitioned further in child bags, and a step for the remaining nodes, which by definition of F_X, have to be endpoints of edges in S_X.

2. Consider a face or a face-part f of X that gets partitioned into parts $f_1, \ldots, f_k$ in between child bags of X. For every f_i, vertices v lying on f_i broadcast the label $\text{Label}_{X_j^*}(f_i)$ to the entire graph X (where X_j^* is the child bag containing f_i). By Property 13, each vertex v in X on a face or a face-part f knows f as such, if it is partitioned, v also knows the IDs of all parts f_i that contain it. Thus, v can broadcast the relevant labels. The broadcast is pipelined, s.t. if a vertex receives the same message more than once, it passes that message once only. A label is of $\tilde{O}(D)$-bits (Lemma 5.17), the diameter of X is $\tilde{O}(D)$ (Property 5 of the BDD) and by Property 9 there are at most $\tilde{O}(D)$ face-parts in the child bags of X, i.e., at most $\tilde{O}(D)$ distinct labels get broadcast, thus, the broadcast terminates within $\tilde{O}(D^2)$ rounds.

3. By the first item, vertices incident to an edge $e \in S_x$ s.t. $e^* \in X^*$ know the IDs of the endpoints f, g of e^*. Thus, if f is not the critical face nor a face-part (i.e., f is entirely contained in X_i^*), any vertex v contained in f knows this about f (Property 13), then v broadcasts the label $\text{Label}_{X_i^*}(f)$ of f. Notice that we send the labels of all (parts of) endpoints of S_x in X^* (if they are face-parts they get handled in the previous broadcast, if not, they get handled in this one).

Again, the broadcast is pipelined, and each vertex passes the same message only once. There are at most $|S_x| = \tilde{O}(D)$ distinct labels being broadcast at any given moment. The size of a label is $\tilde{O}(D)$, the diameter of X is $\tilde{O}(D)$. Thus, the broadcast terminates in $\tilde{O}(D^2)$ rounds.

Local step. For every face or face-part g in X, the vertices lying on g compute locally $\text{Label}_{X^*}(g)$. Intuitively, this is done by decoding all pairwise distances of labels that were received in the previous step, then, constructing a graph known as the *Dense Distance Graph (DDG)* and computing distances in it. The structure of this graph is described next, after that we shall prove that it indeed preserves distances in X^*.

The DDG (see Figure 12) is a (non-planar) graph that preserves the distances in X^* between a subset of nodes of X^*. Traditionally, the DDG is a union of cliques on a (primal) separator, each clique representing distances inside a unique child bag, and the cliques overlap in nodes. In our setting, the DDG is slightly more complicated: (1) the nodes of the DDG are not exactly the separator nodes F_x, but nodes corresponding to faces and face-parts of F_x in child bags of X. (2) different cliques do not overlap in nodes, instead, their nodes are connected by (dual) edges of S_x that are also added to the DDG. (3) edges connecting face-parts of the same face but in different child bags are added to the DDG with weight zero.
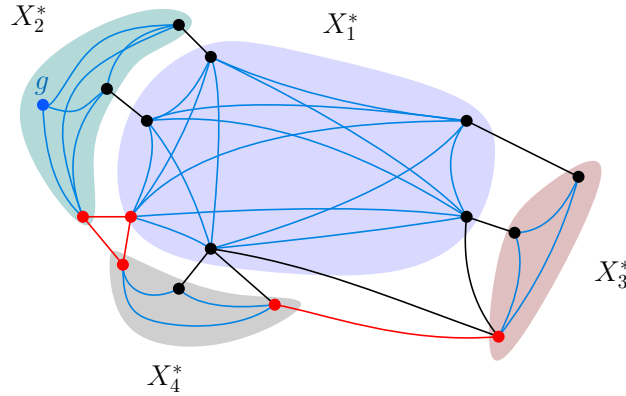

Figure 12: The graph $DDG(g)$. The blue node is g, incident red nodes correspond to face-parts of the same face of X^* across its different child bags, and black nodes correspond to endpoints of S_x edges. The blue edges represent the clique on nodes that belong to the same child bag of X^*, red edges are bidirectional edges of zero weight connecting face-parts of the same face in X^*, and black edges are dual edges to S_x edges. For simplicity, the figure is undirected.

Formally, the graph $DDG(g)$ is a weighted directed graph. The node set of $DDG(g)$ consists of (1) g, (2) the nodes in child bags of X that correspond to nodes (node-parts) of F_x. Denote by V_i the set $V(DDG(g)) \cap V(X_i^*)$ for a child bag X_i^* of X^*, then, the edge set of $DDG(g)$ is defined as:

- An edge between every two nodes of $DDG(g)$ that belong to the same child bag of X^*. The weight of such an edge $(f, h) \in V_i \times V_i$ is the f-to-h distance in the graph X_i^*. By Lemma 5.16, each such distance can be obtained by pairwise decoding two labels of nodes (received in the broadcast step) that belong to the same child bag X_i^*.

- The (dual) edges of S_x which have both endpoints in X^* (i.e., not edges incident to a hole). For each such edge $e = (f, h)$, it is either the case that f exists in X^* and in some X_j^* or f is divided into

27

parts $f_1, f_2, \ldots$ in the child bags of X^*. In the first case, e is incident to f in the DDG. In the second case, e is incident to the part f_i of f that contains e. Notice, in case f is not partitioned then it is in the DDG as it is in F_x for being an endpoint of $e \in S_x$. If it does get partitioned, then all its parts are in the DDG as they are face-parts corresponding to a node of F_x. The weight and direction of e is as in X^*. In either case, the label of those nodes (f or its parts) got broadcast in the broadcasting step. In addition, the information about e (ID, direction and weight) is known in vertices of X also by the broadcast. Thus, vertices can know locally what nodes shall be incident to edges $e \in S_x$ that are contained in X^*.

- A bidirectional zero weighted edge between each two nodes $f_i \in V_i$, $f_j \in V_j$ that correspond to node-parts in X_i, X_j (respectively) of the same node f in F_x. Again, this can be known locally in any vertex v of X since v received the labels (which contain the IDs) of f_i, f_j in the broadcast step, and since the correspondence of f_i, f_j to f can be known from the information received in the broadcast (specifically, the IDs of f_i, f_j contain the ID of f. For more details, see the proof of Lemma 5.12 in Appendix C).

Notice, the description of $DDG(g)$ does not directly address the case where g is not contained in exactly one child bag of X^* (i.e., g gets partitioned). However, if that happens, we have a special case where (by definition) $g \in F_x$, and all of its parts are represented in the DDG. Meaning, the definition of $DDG(g)$ for a node $g \in X^*$ is general enough for both cases ($g \in F_x$ and $g \notin F_x$).

After each vertex v on the face or face-part corresponding to $g \in X^*$ locally constructs $DDG(g)$ and computes distances in it, v checks for a negative cycle. If found, v broadcasts a special message informing all vertices of G (via a global BFS tree of G in $O(D)$ rounds) to abort their algorithm. Otherwise, v locally constructs $\text{Label}_{X^*}(g)$ by appending distances computed in $DDG(g)$ to the recursive label of g (if exists). That is, v sets $\text{dist}_{X^*}(g, f)$ (resp. $\text{dist}_{X^*}(f, g)$), where $f \in F_x$, to be $\text{dist}_{DDG(g)}(g, f')$ (resp. $\text{dist}_{DDG(g)}(f', g)$) where f' is f (in case $f \in V(DDG(g))$) or is an arbitrary part of f in some child bag of X. Due to the fact that all parts f' of f are connected by zero weight edges, then the distance to any of them is exactly the distance to all others.

Lemma 5.18. *For all bags $X^* \in \mathcal{T}$, the above algorithm computes correct label as described in Section 5.2 for all nodes in X^*.*

In order to see why does the above hold, we need to show that $DDG(g)$ of a node $g \in X^*$ is distance preserving w.r.t. the shortest paths metric of X^*.

Proof. For leaf bags, there is no DDG and the labels of all nodes get computed in the first step of the algorithm. For non-leaf bags, it remains to show that the distances computed between $g \in X^*$ and all $f \in F_x$ via $DDG(g)$ are correct in X^*. If $f \in F_x$ is contained in some child bag of X^* then it has a node in $DDG(g)$. If not, then each of its parts has a node. These nodes are all connected by zero weight edges, thus, contracting this clique to a single node in the $DDG(g)$ does not affect distances in the DDG. By Property 12 of the BDD, the resulting graph is a graph on a subset of nodes of X^*, which by construction of $DDG(g)$ are connected with edges that represent paths in X^* (the weights and direction were obtained by decoding labels in child bags of X^*, which are correct by Lemma 5.16). So the distances in the DDG are at most their real distances in X^*. Clearly they are also at least the real distances, as all edges represent paths in X^*. $\qquad\square$

Note, if there is a negative cycle in X^* it gets detected in the leaf-most descendant of X^*.

Lemma 5.19. *A negative cycle in X^* gets detected if and only if it exists.*

Proof. If X^* has no negative cycle then we never report a negative cycle because by Lemma 5.18 all the distances we consider are correct distances in X^*. If X^* does have a negative cycle, let X^* be a leafmost such bag. If X^* is a leaf-bag, then we detect the negative cycle since each vertex of X collects the whole graph X^* and checks for it. For a non-leaf bag X, by Lemma 5.15, all paths (and thus cycles) in X^* are either contained in child bags of X^* or cross the set F_x of nodes. Since X^* is a leafmost bag containing a negative cycle, the negative cycle must contain edges of at least two child bags. Therefore, it can be divided into subpaths that cross F_x (Lemma 5.15). For any node g of the negative cycle, the graph $DDG(g)$ represents all these subpaths correctly (Lemma 5.18), and so the cycle is detected by some vertex v that lies on the face (face-part) g. □

To conclude, a BDD $\mathcal{T}$ is built in $\tilde{O}(D)$ rounds with each vertex $v \in G$ knowing all relevant information that is locally needed about X^* (i.e., X^* is distributively stored). In each level of $\mathcal{T}$ we have at most $\tilde{O}(D^2)$ rounds. Thus, by Lemmas 5.18 and 5.19 the above gives a correct distance labeling algorithm of the dual network. Formally,

Theorem 2.1 (Dual Distance Labeling). *There is a randomized distributed $\tilde{O}(D^2)$-round algorithm that computes $\tilde{O}(D)$-bit distance labels for G^*, or reports that G^* contains a negative cycle w.h.p., upon termination, each vertex of G that lies on a face f, knows the label of the node f in G^*.*

5.3.1 SSSP in the Dual

After we compute the labeling of Theorem 2.1, we want to learn an SSSP tree in G^* from a given source $s \in G^*$. That is, (1) all vertices of a face f in G shall know the distance from s to f in G^*, and (2) each vertex of G shall know its incident edges whose dual counterparts are in the shortest paths tree.

We first choose the vertex v of minimum ID in the face s, which can be done in $\tilde{O}(D)$ rounds by communication over a BFS tree. Then, v broadcasts Label(s) to all vertices of the bag G, this can be done in $\tilde{O}(D)$ rounds, as we can broadcast the label in $\tilde{O}(D)$ messages of size $O(\log n)$ over a BFS tree in $\tilde{O}(D)$ rounds. Then, all vertices of a face f can decode the distance from s to f in G^* from Label(s), Label(f) by Lemma 5.16.

To mark the SSSP tree edges, an aggregate operator is computed for each node v of G^* over its neighbors (using Lemma B.9, we perform a single part-wise aggregation task). In particular, for a dual node, we mark its incident edge that minimizes the distance product with its neighbors, i.e, we take the operator computed is $\min_{u \in N(v)} \{\text{dist}(s, u) + w(u, v)\}$ (where $N(v)$ is v's neighborhood). Finally, G^* might be a multi-graph, so there might be many edges connecting v with a neighbor. We mark the minimal weight edge connecting v to the neighbor that minimizes the above (breaking ties by IDs for edges and for neighbors). Thus,

Lemma 2.2 (Dual Single Source Shortest Paths). *There is a randomized $\tilde{O}(D^2)$ round algorithm that w.h.p. computes a shortest paths tree from any given source $s \in G^*$, or reports that G^* contains a negative cycle. Upon termination, each vertex of G knows for each incident edge whether its dual is in the shortest paths tree or not.*

The above theorem in addition to Section 4 imply distributed (exact and approximate) algorithms for Maximum st-Flow in G (i.e., proves Theorems 1.2 and 1.3) by an adaptation of the centralized algorithms of Miller-Naor [30] and of Hassin [20]. The details are given in Section 6.

6 Primal Maximum st-Flow and Minimum st-Cut

In this section we show how to use the labeling algorithm of Section 5 to compute the maximum st-flow in G, in addition, how to use the minor-aggregation model simulation on G^* of Section 4 to approximate

the maximum st-flow in G in the special case when G is undirected and both s and t lie on the same face in the given planar embedding. I.e, when G is st-planar. The maximum st-flow results naturally extend to minimum st-cut algorithms.

6.1 Maximum st-Flow

Theorem 1.2 (Exact Maximum st-Flow in Directed Planar Graphs). *There is a randomized distributed algorithm that given an n-vertex directed planar network G with hop-diameter D and integral edge-capacities, and two vertices s, t, computes the maximum st-flow value and assignment w.h.p. in $\tilde{O}(D^2)$ rounds.*

The proof mostly follows a centralized algorithm of Miller and Naor [30], we provide an overview of the general idea. For the full proof see Appendix D.1. The algorithm uses a logarithmic number of SSSP computations in the dual, after which one can deduce the value of the max st-flow and a flow assignment. In particular, it performs a binary search on the value λ of the maximum st-flow. In each iteration, λ units of flow are pushed along a (not necessarily shortest nor directed) s-to-t path. If this does not violate the capacity of any dart in P then it is a feasible flow and we conclude that the maximum st-flow is at least λ. Otherwise, a residual graph is defined, possibly having negative capacities. To fix that, an SSSP computation from an arbitrary node f in the dual is performed, such that, the flow can be fixed to not violate capacities if and only if the dual does not contain a negative cycle (if a negative cycle was found, then, the max st-flow value is less than λ). This gives the flow value. In the former case, Miller and Naor show that the following is a feasible st-flow assignment of value λ. For every edge $d \in G$ that maps to $d^* = (g, h) \in G^*$: the flow on d is set to $\text{dist}(f, h) - \text{dist}(f, g) + \lambda$ if d is directed towards t in P, to $\text{dist}(f, h) - \text{dist}(f, g) - \lambda$ if d is directed towards s in P, and to $\text{dist}(f, h) - \text{dist}(f, g)$ if it is not in P.

The *undirected* maximum st-flow problem asks to find the maximum flow that can be pushed from s to t in an undirected network. Alternatively, where two neighbors are connected by two opposite direction edges with the same capacity. Although the s-to-t flow value equals t-to-s flow value, the s-to-t flow itself is directed (e.g. the flow entering s is non-positive, the flow entering t is non-negative).

Theorem 1.3 (Approximate Maximum st-Flow in Undirected st-Planar Graphs). *There is a randomized distributed algorithm that given an n-vertex undirected planar network G with hop-diameter D and integral edge-capacities, and two vertices s, t lying on the same face, computes a $(1 - \epsilon)$-approximation of the maximum st-flow value and a corresponding assignment in $D \cdot n^{o(1)}$ rounds w.h.p.*

To prove this, we simulate a centralized algorithm of Hassin [20], where he adds an edge $e = (t, s)$ and then computes SSSP in the dual of the resulting graph. s and t are required to be on the same face in order to ensure that e does not violate planarity (it is embedded in the interior of the face containing s and t). We use our minor-aggregation simulation on the dual (Section 4) in order to apply an approximate SSSP algorithm of [42] on $(G \cup \{e\})^*$, the dual of $G \cup \{e\}$. We use an approximate algorithm since there is no exact one in the minor-aggregation model. This simulation instantly gives the value of the approximate maximum st-flow, however, in order to get an assignment we need to work a bit harder because Hassin's algorithm is an exact algorithm whilst we use it approximately. Hassin defines the flow assignment on an edge e' whose dual is $e'^* = (f, g)$ to be $\text{dist}(f_1, g) - \text{dist}(f_1, f)$, where f_1 is the source of the SSSP tree computed in $(G \cup \{e\})^*$ (note, we need the exact distances for the assignment). Thus, approximately, if we want the flow assignment to be capacity respecting (amongst other requirements), for each edge, we need to bound the difference of its endpoints' (approximate) distances, i.e, we need the approximate distances to obey the triangle inequality. Most (distributed) distance approximation algorithms do not have this feature. Meaning, the approximation can be very far off the true answer. E.g, let u, v be two neighbors of distance $\Omega(n)$ from the source, connected by a zero weight edge, we want the difference of their (approximate) distances from the source to be zero, however, the difference of the approximate

distances might be at least as large as ϵn. To overcome that, we show how to apply a recent algorithm of [40] on the dual. This algorithm was not implemented in the minor-aggregation model before.

Proof. The algorithm follows the centralized algorithm of Hassin [20]: Let f be a face of G containing both s and t. Firstly, we add an artificial edge $e = (t, s)$ to G embedded inside the face f (preserving planarity) with infinite capacity. This edge is assigned a weight of nW, where W is the largest weight in the network, representing infinite capacity in $\tilde{O}(1)$-bits.
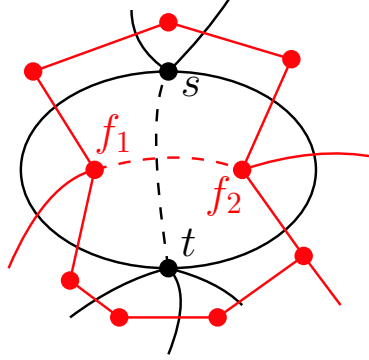

Figure 13: $G \cup \{e\}$ in black and its dual in red. e (e^*) is the dashed edge. f is the face in which e is embedded, note, f is split into two faces f_1, f_2. Each of f_1 and f_2 has a corresponding node in the dual.

The edge e divides f into two faces, f_1, f_2, where f_1 is the face on the left [7] of e if e were to be directed from t to s (this would be important for the flow direction later). We compute SSSP from f_1 (this time all edge-lengths are non-negative) in $(G \cup \{e\})^*$. See Figure 13. Hassin showed that the value of the maximum st-flow in G is equal to $\text{dist}(f_1, f_2)$ in $(G \cup \{e\})^*$. We note that Hassin's algorithm works for directed and undirected graphs. For us however, it is crucial that the graph is undirected since the (distributed) SSSP algorithm that we use works only for undirected graphs.

Distributively, all vertices need to know the IDs of s, t, and f. We detect the face f by a simple part-wise aggregation in $\hat{G}$ (Properties 2, 3 and 4 of $\hat{G}$), where the input of each copy of s (resp. t) is $ID(s)$ ($ID(t)$) and an identity item for any other vertex (e.g. minus one, as no ID is negative). The aggregate operator simply counts the distinct values of input. Then, the face f is the face that has the minimum ID of the (at most two) faces which got a count of two (s, t know these IDs and can choose the minimum locally). Note, this aggregation terminates in $\tilde{O}(D)$ since the values aggregated are of size $\tilde{O}(1)$-bits and there are at most two distinct values aggregated on each face of $\hat{G}$ that maps to a face of G. Then, both s and t add e to the clockwise ordering of their incident edges in G, and embed it to be in between two consecutive edges of their incident edges in f. We use the minor aggregation model implementation (Section 4) for the dual graph to simulate an SSSP minor-aggregation algorithm on $(G \cup \{e\})^*$. We use the algorithm of [42] that computes $(1 + \epsilon)$-approximate SSSP in $2^{O(\log n \cdot \log \log n)^{3/4}} / \epsilon^2$ minor-aggregation rounds, yielding an $D \cdot n^{o(1)}$ CONGEST algorithm in G for the undirected st flow problem in st-planar networks. However, we want to simulate the algorithm on $(G \cup \{e\})^*$ rather than G^*, meaning the simulation result of Section 4 cannot be used straightforwardly. Moreover, [42]'s algorithm assumes weights in $[1, n^{O(1)}]$ but not zero (not allowing zero capacities). So, we assume that we do not have zero weight edges and first dive into the details of the simulation, then we show how to get an assignment and finally show a simple fix for dealing with zero weight edges.

---

[7] Left and right are defined w.r.t. the direction of e and can be deduced from the combinatorial embedding of e's endpoints.

Simulation. Here we desire to simulate [42]'s approximate SSSP algorithm on $(G \cup \{e\})^* \neq G^*$ by applying Section 4 (Theorem B.14) to simulate the extended minor aggregation model on G^*. We exploit that the extended model can simulate any basic or extended minor aggregation algorithm on any virtual graph G^*_{virt} that contains a small (poly-logarithmic) number of arbitrarily connected virtual nodes (see Appendix B for exact definitions). Thus, the only thing we need to do, is to use that (Theorem B.14) in order to apply the approximate SSSP algorithm on $G^*_{virt} = (G \cup \{e\})^*$. To do so, we only need to distributively store $(G \cup \{e\})^*$ as needed (matching the input format of the algorithm stated in Theorem B.14, for full details, see Appendix B). We explain; Recall, the face-disjoint graph $\hat{G}$ is used as the base graph for representing G^*, i.e., $\hat{G}$ is the distributed method for storing G^*; To store any virtual graph G^*_{virt} that is obtained by replacing nodes or adding virtual nodes to G^*, we need that all vertices on a face $\hat{g}$ of $\hat{G}$ (maps to node $g \in G^*$) to know a list of (IDs of) g's incident virtual edges and all virtual edges that connect any two virtual nodes (and their weight if exists) in G^*_{virt}. So, we need to (1) show how to delete f from G^*, (2) add two nodes that play the roles of f_1, f_2, and (3) connect them to neighbors of f and to themselves properly, represented in $\hat{G}$ as described; Such that, the obtained G^*_{virt} has the exact same structure as $(G \cup \{e\})^*$. Note, changes need only to be made on $\hat{f}$'s vertices and vertices lying on (faces $\hat{g}$ in $\hat{G}$ corresponding to) neighbors g of f in G^*.

We start by: (1) Enumerating $\hat{f}$'s vertices in $\hat{G}$ in ascending order and deactivating them (deleting f). This numbering would be useful to define f_1, f_2 (i.e, split $\hat{f}$). (2) Learning (locally) in $\hat{G}$ for each edge of E_C [8] that is incident to $\hat{f}$, whether its corresponding edge in G^* shall be incident to f_1 or to f_2. Then, (3) Each (face $\hat{g}$ in $\hat{G}$ that maps to a) neighbor g of f needs to know its minimal weight edge connecting it to f_1 and f_2, as we are interested in shortest paths and cannot afford to broadcast all edges of f, due to their possibly large quantity. To do so, we compute a minimum aggregate over weights of $\hat{g}$'s edges that are mapped to edges that shall be incident to f_1 (f_2), so that all vertices in $\hat{g}$ know the edge that connects g to f_1 (f_2). Finally, (4) A global maximum operator is computed over edge weights of G, such that, all vertices of $\hat{G}$ learn that f_1, f_2 are connected by an edge whose weight is nW, where W is the maximum weight found. We describe how each step is implemented; Assume $\hat{G}$ is already constructed and its faces that correspond to G faces are identified (Properties 1, 4 of $\hat{G}$).

1. For enumerating $\hat{f}$'s vertices in $\hat{G}$, we use a simple classic procedure of a subtree sum using part-wise aggregations on $\hat{G}$. Specifically, the input tree is $\hat{f}$ minus the E_R [9] edge copy of the edge preceding e in s's local ordering of its incident edges (this edge is known to s locally), the root is the copy of s that is incident to the removed edge, and all vertices get an input value of one initially. The procedure we use is similar to those described in [14, 17]. We can do so in $\tilde{O}(D)$ rounds by Section 4 (Lemma B.6).

   Denote by s', the copy of s in $\hat{f}$ which is incident to the two consecutive edges e'_1, e'_2 (of E_R in $\hat{G}$), that map to the edges e_1, e_2 incident to s, s.t. e is embedded in between e_1, e_2 (in G), we define t' analogously. Intuitively, let e_a, e_b be the two edges incident to s s.t. all edges between e_a, e_b (in the local ordering of s) are in f, then all edges in between e_a, e_1 have to go to f_1 and all edges in between e_2, e_b have to go to f_2, as e is embedded right after e_1 and right before e_2. We define s' to capture this in $\hat{G}$ (where edges have a lot of copies, sometimes in the same face [10]), in order to define f_1, f_2 correctly.

   Now, s' (resp. t') broadcasts its ID, assigned number, and $ID(f)$. The ID of f_1, f_2 is defined to

---

[8] Those edges map 1-1 to edges of G^*, see Section 3 and Appendix A for more details.

[9] For each edge e of G, there are two edge copies in $E_R \subset E(\hat{G})$. Intuitively, there is an edge copy for each face of G that contains e.

[10] E.g, if an edge e is internal to a face f, then it has its two E_R copies in the face $\hat{f}$ of $\hat{G}$ that maps to f. More generally, if there is a tree attached to a vertex in the interior of a face f, then each of those tree edges has two copies in $\hat{f}$, as $\hat{f}$ is basically an Euler tour on f. See Section 3 and Appendix A for more details.

be $(ID(f), 0)$ and $(ID(f), 1)$, respectively, all vertices now know these as well. The broadcast is pipelined and terminates in $\tilde{O}(D)$ rounds as there is a constant amount of distinct messages sent. From now on, all vertices in $\hat{f}$ consider themselves as deactivated - they might still be used for communication in the simulation, but they do not participate as an input.

2. Each vertex on a face $\hat{g}$ of $\hat{G}$ already knows (by construction of $\hat{G}$) for each of its incident E_C edge e^*, the (IDs of) faces it participates in and the ID of the edge e of G that e^* corresponds to (recall, E_C edges are meant to simulate G^* edges in $\hat{G}$). Thus, in one round of communication, each vertex in $\hat{G}$ can know for each incident edge in E_C whether it shall be in f_1 or f_2, by firstly receiving the numbering of its neighbors in $\hat{f}$ and then, by deciding on each edge e^* with both endpoints of e having a number in between s' and t' to be in f_1, otherwise, to be in f_2. Note, this is consistent with f_1 being on the left of e if directed from t to s due to the choice of s' and t'.

3. Now that edges incident to f_1 and f_2 are known to their endpoints in $\hat{G}$, two minimum aggregations in $\tilde{O}(D)$ rounds are then computed and broadcast in each neighbor $\hat{g}$ of $\hat{f}$ in $\hat{G}$ over E_C edges that connect f_1 to g and f_2 to g, allowing all vertices on $\hat{g}$ to learn the two edges connecting g and each of f_1, f_2 in $(G \cup \{e\})^*$, if any.

4. Finally, an aggregate over G edge weights is computed in $O(D)$ rounds for finding W, then, since each vertex in $\hat{G}$ knows the ID of f_1, f_2 and W, it learns that f_1, f_2 are connected by an edge with weight nW in $(G \cup \{e\})^*$.

Notice, all the above terminates within $\tilde{O}(D)$ rounds on G by Section 4 (Lemma B.6). Finally, since $G^*_{virt} = (G \cup \{e\})^*$ now is properly distributively stored, then, we can simulate the extended minor-aggregation model on it (see Theorem B.14 for details); So, we first use it in order to deactivate parallel edges (Lemma B.15), keeping only the minimal-weighted edge between its endpoints (we aim to compute SSSP). Then, since G^*_{virt} is simple and we can simulate any minor aggregation algorithm on it, it satisfies the requirements for applying the mentioned theorem with [42]'s SSSP algorithm, obtaining an approximate SSSP tree rooted at the source f_1.

Distances (flow value). Assuming we have computed an approximate SSSP tree T (with zero weighted edges included) as promised, we can compute the distances of any node from the source by a simple procedure. That is, root the computed tree with the source, then compute the tree ancestor sum of every node $g \in T$, i.e., the weight of the f_1-to-g T-path. For that, we use a procedure of [18] that runs in $\tilde{O}(1)$ minor aggregation rounds (Lemma 16 in their paper), thus, $\tilde{O}(D)$ rounds in G. After that process terminates, each vertex lying on f_2 (in G) knows the approximate distance from f_1 to f_2 in G^*_{virt}, thus, it can broadcast it to the whole graph. Note, that distance is a $(1 + \epsilon)$ approximation of the maximum st-flow value (by Hassin [20]). Since the algorithm guarantees to report $\alpha \cdot \text{dist}(f_1, f_2)$ instead of $\text{dist}(f_1, f_2)$, for $\alpha \in [1, 1 + \epsilon]$, multiplying by $(1 - \epsilon)$, we get $(1 - \epsilon)\alpha \in [1 - \epsilon, 1 - \epsilon^2]$, that is, a $(1 - \epsilon)$ approximation of the maximum st-flow value.

Approximate flow assignment. Note that in Theorem 1.2 we obtained both the value of the maximum flow and a corresponding assignment of flow to the edges. Here however, we approximate the flow value but do not immediately obtain an assignment (or an approximate assignment) of the flow. The reason is as follows: Hassin's algorithm was designed as an exact algorithm (running an exact SSSP computation) while we use an approximate SSSP. For the flow value this is fine, instead of $\text{dist}(f_1, f_2)$ we report a $(1 - \epsilon)$-approximation of $\text{dist}(f_1, f_2)$. However, for the corresponding flow assignment, recall that the flow on edge $e' \in G$ with $e'^* = (f, g) \in (G \cup \{e\})^*$ is $\text{dist}(f_1, g) - \text{dist}(f_1, f)$. This means we need the approximation algorithm to bound $\text{dist}(f_1, g) - \text{dist}(f_1, f)$ by the edge's capacity (and not just approximate

dist(f_1, g) and dist(f_1, f)). I.e., we want the approximate distances to satisfy the triangle inequality. This indeed is possible using (a dual implementation of) a recent algorithm of Rozhon, Haeupler, Martinsson, Grunau and Zuzic [40], however, their algorithm was not implemented in the minor-aggregation model. Given access to a $(1 + \epsilon)$-approximate shortest paths oracle, [40]'s algorithm produces *smooth* (defined next) approximate distances within a logarithmic number of calls to the oracle with a parameter $\epsilon' = O(\epsilon/\log n)$. Before we dive into the simulation, we define what a smooth distance approximation is, suggest a flow assignment and prove it is a feasible $(1 - \epsilon)$ approximate assignment using the smoothness property.

Smooth distances and a feasible flow assignment. In order to prove that the approximate SSSP computation gives a *feasible* $(1 - \epsilon)$ approximate flow assignment, we need to show an assignment that is: (a) Capacity respecting: the flow being pushed on each edge is non-negative and is at most its capacity, (b) Obeys conservation: for all vertices except for s and t, the flow pushed into a vertex equals the flow pushed by it. Finally, the assignment should be (c) $(1 - \epsilon)$ approximate: the flow pushed from s to t is between $(1 - \epsilon)$ and once the maximum st-flow. Note, even thought the graph of input is undericted, the flow itself is directed, so we address that as well. We first define some notions and define the flow assignment the, then proceed to prove that the above holds.

In [40], a $(1 + \epsilon)$ distance approximation $d(\cdot)$ from a given (single) source in a graph $\mathcal{G}$ is said to be $(1 + \epsilon)$-*smooth* (definition 4.2 in their paper) if it satisfies the following:

$$\forall u, v \in V(\mathcal{G}) : d(v) - d(u) \leq (1 + \epsilon) \cdot \text{dist}(u, v)$$

We assume we have computed a $(1 + \epsilon)$ smooth approximate distances from f_1 in $(G \cup \{e\})^*$ by combining [42] and [40], denoted $d(\cdot)$, and we define $\delta(\cdot) := (1 - \epsilon)d(\cdot)$. Then, for $e' = \{u, v\} \in G \cup \{e\}$ s.t. $e'^* = \{f, g\}$ in $(G \cup \{e\})^*$, we define the flow value pushed on e' to be $|\delta(g) - \delta(f)|$. When $\delta(g) \geq \delta(f)$, the flow is being pushed on e from u to v, otherwise, it is pushed from v to u. We denote this flow assignment by $\phi(\cdot)$, taking the induced assignment on G gives the final assignment.

(a) *Capacity respecting.* Obviously, the flow pushed on each edge is non-negative (by definition). Given a $(1 + \epsilon)$-smooth distance approximation from the source f_1 in $G^*_{virt} = (G \cup \{e\})^*$, we get a capacity respecting flow assignment as follows. Consider any edge $e' = (u, v)$ where $e'^* = (f, g)$ and the approximate distances $d(\cdot)$, s.t. $d(g) \geq d(f)$, then, from the smoothness property:

$$d(g) - d(f) \leq (1 + \epsilon) \cdot \text{dist}(f, g) \leq (1 + \epsilon) \cdot c(e') \tag{1}$$

multiplying by $(1 - \epsilon)$, we get,

$$\delta(g) - \delta(f) \leq (1 - \epsilon^2) \cdot \text{dist}(f, g) \leq (1 - \epsilon^2) \cdot c(e') \tag{2}$$

Note, the left-hand side of Equation (2) is exactly $\phi(e')$, the flow pushed on e'; $c(e')$ is its capacity and the property follows. The left inequality in Equation (1) follows simply from the smoothness property, whilst the right one follows since the the edge weights of G^*_{virt} are the edge capacities of the corresponding primal edges and that the exact distances in G^*_{virt} is a metric that obeys the triangle inequality, i.e, dist(f, g) is at most the weight of the edge connecting f, g. Equation (2) follows from Equation (1) since, by definition, $\delta(\cdot) = (1 - \epsilon)d(\cdot)$.

(b) *Conservation.* By [25], assuming a planar directed graph with edge capacities and an assignment of *potentials* (weights) $\rho(\cdot)$ on its faces, then, assigning for each edge a flow of $\rho(g) - \rho(f)$, where g (resp. f) is the face on the right (left) of that edge, gives a flow assignment that obeys conservation in all vertices [11]. Considering $G \cup \{e\}$, the assignment we suggest, is deliberately defined so that the

---

[11]This kind of a flow assignment is referred to as a *circulation* in the centralized literature on planar graphs.

directions of the flow would be consistent with the above. I.e, directing an edge in its defined flow direction and pushing $|\delta(g) - \delta(f)|$ units of flow would be consistent with the definition above. Thus, the flow assignment we suggest obeys conservation in every vertex of $G \cup \{e\}$. Note, G is equivalent to $G \cup \{e\}$ after removing $e = \{s, t\}$, thus, the induced flow assignment on G obeys conservation except in the endpoints of e, which are s and t.

(c) *Flow value of the assignment.* f_1 was picked to be the source of the approximate SSSP computation ($\delta(f_1) = 0$), thus, $\delta(f_2) \geq \delta(f_1)$, meaning, according to our definition of $\phi(\cdot)$, the flow on e shall be directed from t to s, as f_1 shall be the face on left of e w.r.t. the flow direction on e. As a result, the flow pushed on e from t to s is $\delta(f_2) - \delta(f_1)$. Note, $\phi(\cdot)$ obeys conservation in all vertices of $G \cup \{e\}$, thus, removing e from $G \cup \{e\}$ would induce the flow assignment in G, directed from s to t, of value:

$$\phi(e) = \delta(f_2) - \delta(f_1) = (1 - \epsilon) \cdot d(f_2) - 0 = (1 - \epsilon) \cdot \alpha \cdot \text{dist}(f_2, f_1) \tag{3}$$

where $\alpha \in [1, 1 + \epsilon]$. We already have shown that the right-hand side of Equation (3) is a $(1 - \epsilon)$ approximation of the maximum st-flow value, thus, the induced flow assignment by $\phi(\cdot)$ on G of value $\phi(e)$ is $(1 - \epsilon)$ approximate.

It remains only to show that we can apply [40]'s algorithm (Theorem 5.1, Algorithms 3 and 4 in their paper), in order to obtain smooth $(1 + \epsilon)$ approximate single source shortest paths from f_1 in $G^*_{virt} = (G \cup \{e\})^*$ by invoking the algorithm of [42] that we use for approximating distances, which we refer to as the *oracle*. We note that [40]'s algorithm assumes real weights in $[1, n^{O(1)}]$, whereas we work with integral weights, however, they suggest a fix (by scaling) for this that appears in Appendix C of their paper. Note, scaling can be implemented efficiently in one minor-aggregation round, henceforth, we consider this issue resolved.

[40]'s algorithm in a high-level. We describe [40]'s algorithm for smooth distance approximation in a high-level, highlighting only the computational tasks necessary to implement it in the minor aggregation model. For the algorithm in full resolution, the reader is referred to Section 5 of [40]. Initially, nodes of the input graph shall know the source s, the approximation parameter ϵ and a trivial estimate Δ of the maximum distance in the graph ($\Delta := 2nW$ in our case), then, each node computes additional parameters that depend only on the former. Note, all of the former may be implemented in only one minor-aggregation round.

Next, the algorithm computes a distance approximation, based on a call to the approximation oracle (any SSSP approximation algorithm, without any additional special guarantees) with s and $\epsilon' = O(\epsilon / \log n)$. Then, sequentially, $O(\log n)$ phases are performed, each defining a distance approximation, such that, the final output of the algorithm is the approximation computed in the last phase. In each phase, two calls to the approximate oracle are performed with s and $\epsilon'/100$. Each call to the oracle is on a *level graph*. Afterwards, each level graph defines a distance approximation of the input graph, and the phase's distance approximation is computed by taking the element- (node-) wise minimum over: (1) the approximations computed on the level graphs, and (2) the approximation from the previous phase. Again, taking the minimum requires only one minor-aggregation round.

Level graphs. The only vagueness in the implementation lies in the level graphs, which we now define, then, show that they can constructed and simulated efficiently in the minor-aggregation model. The level graph H (Definition 5.7 in [40]) in a specific phase of an input graph $\mathcal{G}$ where each node of $\mathcal{G}$ knows its (approximate) distance from a source s, is defined as follows. The node and edge sets of H are the same as those of $\mathcal{G}$, then: (1) Each node assigns itself a level (a number), depending only on its distance from s and some parameter ω, (2) each edge that connects neighbors of different levels is removed, the weights

of edges kept are altered depending only on previously computed parameters that all nodes of $\mathcal{G}$ know. Finally, (3) An edge between s and each $u \in \mathcal{G}$ is added, with weight depending only on ω and the s-to-u distance approximated in the previous phase.

Implementation of level graphs. First, we start by computing ω in one minor-aggregation round on G^*_{virt} as it depends only on parameters known to all nodes of the input graph (as Δ, ϵ and others). Then, we compute the level of each node, also in one round, as it only depends on ω and the s-to-u distance approximated in the previous phase. The remaining things to explain are: (1) How to "delete" edges that cross levels, and (2) How to connect s to all nodes with the appropriate edge weights.

1. Deactivating edges that cross levels is simple. Each edge $e = (u, v)$ in the input graph knows the level of both its endpoints, thus, it knows if it should participate in the level graph or not. In more detail, recall, e in the minor aggregation model is simulated by its endpoints u', v' in the base communication graph ($\hat{G}$ in our case). In one round on $\hat{G}$, u' knows the level of the super-node that contains v' (and vise-versa). Thus, u', v' know that e is prevented from participating in computations on H. In case v' is virtual then all nodes of the graph simulate it (in particular u'), and know its information (in particular, its level).

2. Luckily, we work in the extended minor-aggregation model, where adding virtual nodes and connecting them arbitrarily to any other virtual or non-virtual node, is a feature of the model. In the same manner that we described earlier (where we obtained G^*_{virt}), we alter H and replace s with a virtual node, all nodes in H now know they are connected to the source. Notice, there are no problems regarding weights of edges, as the weight of an added edge (s, u) depends only on ω and the approximated s-to-u distance in the previous phase (already known to u). Thus, when nodes connect themselves to the replaced s, they connect themselves with the needed weights. Again, virtual nodes are simulated by all vertices, so for each virtual v, all real nodes know its info and locally consider it to be connected to s with the right weight.

Note, each phase is of $\tilde{O}(1) + T_{SSSP(\epsilon')}$ minor-aggregation rounds, where $T_{SSSP(\epsilon')}$ denotes the minor-aggregation round complexity of a $(1 + \epsilon')$-approximate SSSP oracle. Thus, the overall minor-aggregate round complexity of the $(1 + \epsilon)$ smooth approximate distance computation is $O(\log n) \cdot \tilde{O}(T_{SSSP(\epsilon')})$. Implementing the approximate SSSP oracle with the algorithm of [42], results with a total minor-aggregate round complexity of $O(\log n) \cdot (\frac{1}{\epsilon'})^2 \cdot 2^{O(\log n \cdot \log \log n)^{3/4}}$, i.e., $\frac{\log^3 n}{\epsilon^2} \cdot 2^{O(\log n \cdot \log \log n)^{3/4}}$, which translates to $\frac{D}{\epsilon^2} \cdot n^{o(1)}$ CONGEST rounds on the network of communication G. For conclusion, since each vertex v in G simulates its copies in $\hat{G}$, which in turn simulate the faces v participates in in G^*_{virt}, then, v knows for each face it participates in, its distance from the source. Thus, after computing smooth distances in the relevant graph G^*_{virt}, in one round of communication, v knows for each incident edge e', such that $e'^* = (f, g)$, the values $\delta(f_1, g)$ and $\delta(f_1, f)$ and can locally compute $\phi(e')$. Finally, we are ready to go through the last modification required to keep our promise, that is, we show next how to allow zero capacities (zero-weight edges in the dual shortest paths approximation).

Zero weighted edges. The algorithm of [42] that we use for the approximate SSSP and of [40] used to obtain the smoothness property work for weights in $[1, n^{O(1)}]$, indeed all weights in G^*_{virt}, even that of the infinite capacity edge are in $[1, n^{O(1)}]$ except the zero-weighted ones. To deal with that issue, we simulate the obtained smooth approximate SSSP algorithm above while considering zero weighted edges as follows: We run the algorithm on maximal connected components of $G^*_{virt} \setminus \{e : w(e) = 0\}$, obtaining a partial SSSP tree of G^*_{virt}, then we augment it with some zero weighted edges without closing a cycle, obtaining an SSSP tree T of G^*_{virt}, all done via the extended minor aggregation model.

Concretely, we work in two phases. In the first phase, we compute an approximate SSSP tree T_1 on the minor G_1^* of G_{virt}^* where all zero-weighted edges are contracted, where the source is the super-node that contains f_1. Now, we need to choose what zero-weighted edges shall be added to complete the approximate SSSP tree T_1 of G_1^* into an approximate SSSP tree T of G_{virt}^*. T_1 defines a set of connected components in G_{virt}^*, contracting those components in G_{virt}^* (after we undo the zero-weight edge contraction) results with another minor G_2, that contains only zero-weight edges. I.e., we are looking for some spanning tree in G_2. So, we compute a minimum spanning tree T_2 of G_2^* by a minor aggregation procedure of [42] (Example 4.4 in their paper). Hence, the tree $T = T_1 \cup T_2$ (considering the endpoints of edges of T_1, T_2 in G_{virt}^* rather than G_1^*, G_2^* when taking the union) constitutes an approximate SSSP tree in G_{virt}^*.

Note, T obeys the triangle-inequality (which is crucial for a flow assignment) for the following reasoning. Distances in T_1 represent distances in G_{virt}^*, by construction any pair of nodes in T_1 satisfies the triangle inequality. Thus, map each node $f \in G_{virt}^*$ to the super-node a of T_1 that contains it, the property follows since f and a have the same distances from the source because T_1 was obtained by contracting zero-weight edges only. I.e, $T = T_1 \cup T_2$ satisfies the triangle inequality as any pair of (super-)nodes in T_1 does. $\qquad\square$

6.2 Minimum st-Cut

By the well known Max-Flow Min-Cut Theorem [7], having the exact (approximate) value of the maximum st-flow in G immediately gives the exact (approximate) value of the minimum st-cut in G. So, here we focus on finding the cut edges and bisection. We use our flow algorithms to do so.

Theorem 6.1 (Distributed Exact Planar Directed Minimum st-Cut). *Given a directed planar graph G with non-negative edge weights and two vertices s, t, there is a $\tilde{O}(D^2)$ round randomized algorithm that finds the minimum st-cut w.h.p.*

To prove this, we use Theorem 1.2 to compute a maximum st-flow f in G in $\tilde{O}(D^2)$ rounds. By [7], a corresponding minimum st-cut is the cut defined by the saturated edges (an edge is saturated if the flow it pushes equals its capacity) that precede any other saturated edge on any path that contains them and starts at s. To find those edges we set the weight of each saturated edge to one and all other edges to zero. Then, we use an exact distance algorithm that runs in $\tilde{O}(D^2)$ rounds for planar graphs by [26] on the graph with modified weights. Each edge that has an endpoint with distance zero from s and another endpoint of distance one from s is a cut edge. The bisection can be deduced from the distances as well, each vertex whose distance from s is zero on s's side of the cut, others are on t's side. For a full proof, the reader is referred to Appendix D.2.

Next, we show an almost optimal algorithm for $(1 + \epsilon)$ approximation of the undirected minimum st-cut, where s and t lie on the same face.

Theorem 6.2 (Distributed Approximate st-Planar Undirected Minimum st-Cut). *Given an undirected planar graph G with non-negative edge weights and two vertices s, t that lie on the same face of G, there is a randomized algorithm that finds an approximate $(1 + \epsilon)$ minimum st-cut and runs in $D \cdot n^{o(1)}$ rounds w.h.p.*

Proof. Note, the method used in Theorem 6.1 does not apply instantly here, for two reasons, (1) the round complexity of the primal SSSP computation is very high compared to our promise, and (2) we cannot compute the residual graph, as we only have an approximation of the flow, meaning, we may not distinguish between saturated edges and non-saturated edges, a crucial step in the previous method. Eventually, the cut edges are going to be the path P found by the SSSP computation we perform when we approximate the max st-flow (see the proof of Theorem 1.3 for more details).

Concretely, we take a primal-dual detour, where we consider faces in the dual graph that map to the vertices s and t in the primal[12]. See Figure 14. We refer to faces of the dual as *dual faces*. A special case of the cycle-cut duality (Fact 3.1) is given by Reif [38], stating: An st-separating cycle in the dual graph is the dual of an st-cut in the primal, where a cycle in the dual graph is said to be st-separating, iff it contains the the dual face that is mapped to s in its interior, and the dual face mapped to t in its exterior (or vice versa). Thus, we find a $(1 + \epsilon)$-approximation of the minimum st-separating cycle in G^* in order to find its dual, the desired $(1 + \epsilon)$-approximate minimum st-cut in G.
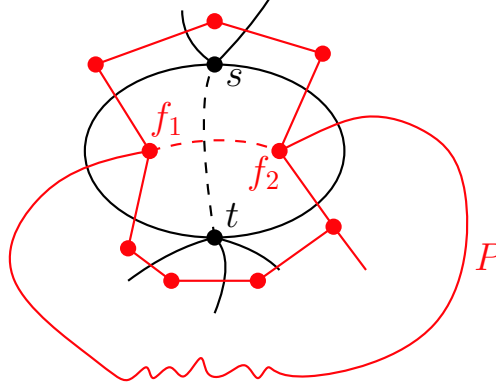

Figure 14: $G \cup \{e\}$ in black and its dual in red. e (e^*) is the dashed edge. Each of s and t correspond to a face in the dual. Both of those dual faces contain e. Note, the cycle C is demonstrated by the f_1-to-f_2 path P and e^*, where the dual face t is in its interior and the dual face s in its exterior.

To do so, we compute a $(1 + \epsilon)$-approximate max st-flow by Theorem 1.3, not $(1 - \epsilon)$, see the proof of that theorem where we first obtain a $(1 + \epsilon)$ approximation (and then transform it to a $(1 - \epsilon)$ one). Then we consider $G^*_{virt} = (G \cup \{e\})^*$, where $e = (s, t)$ splits the face f in G on which both s, t lie into two faces, f_1 and f_2. Recall, the algorithm from Theorem 1.3, computes a shortest path P from f_1 to f_2 in G^*_{virt} (see the proof of that theorem, its correctness is used in this proof). We claim that P corresponds exactly to the approximate minimum st-cut in G. See Figure 14.

Note, any st-cut in $G \cup \{e\}$ shall contain e, and removing it gives G. Thus, to have the min st-cut in G, we can look for the min st-cut in $G \cup \{e\}$. In particular, there is a mapping between st-cuts of G and $G \cup \{e\}$, where the value of the cut in $G \cup \{e\}$ is the same value of the corresponding cut in G plus the weight (nW) of e. Thus, when applying the duality of Reif we consider G^*_{virt}. That is, we look for the approximate minimum separating cycle in G^*_{virt}. By duality, this cycle should contain $e^* = (f_1, f_2)$.

Note, the edge $e^* = (f_1, f_2)$ closes a cycle with P, denoted $C = P \cup \{e^*\}$, that would be the desired approximate min st separating cycle. C is separating as any cycle in a planar graph is separating (Jordan curve theorem [44]), in particular it is st separating, for: (1) s and t are faces in the dual, each has to be either in the interior or the exterior of C, (2) one dart of e is in C's interior and the other in its exterior, each such dart corresponds to a distinct dual face, one of which maps to s and the other maps to t because e connects s and t in the primal. Note, C is $(1 + \epsilon)$-approximate st-separating minimum cycle in G^*_{virt}. Otherwise, P is not a $(1 + \epsilon)$-approximation to the shortest f_1-to-f_2 path. I.e, since $C = P \cup \{e\}$, then the length of C is that of P plus nW. I.e, if C is not a $(1 + \epsilon)$ approximation of the minimum st separating cycle, then, the length of C minus nW is not a $(1 + \epsilon)$ approximation to the shortest f_1-to-f_2 path, contradicting the fact that P is.

Next, we aim to mark those st-cut edges in G. By the above, it is sufficient to mark edges in G who are

---

[12]The dual of the dual is the primal, i.e, primal vertices map to faces of the dual graph. Moreover, the dual face that is mapped to any vertex v is the (only) face that consists of all darts entering v. See [25] for more details.

the duals of edges P in G^*_{virt}. Assuming we already applied Theorem 1.3, in particular, we have computed an approximate SSSP tree T from f_1, then, we want to mark P. We already know that we can simulate any minor-aggregation algorithm on G^*_{virt} (for details, one is referred to the proof of Theorem 1.3), thus, we can simulate any minor-aggregation algorithm on any subgraph of G^*_{virt} (Corollary 11 of [18]). I.e., we can invoke any minor-aggregation algorithm on T. To mark P edges, a minor-aggregation tree-rooting and subtree-sum procedures of [18] are invoked, where we root T with f_1 and compute subtree-sum on the rooted tree, such that, the input value of all nodes is zero, except for f_2 that would have an input of one. The edges whose both endpoints have subtree-sum value of one are marked, those exactly are the P-edges. Those procedures are of $\tilde{O}(1)$ minor-aggregation rounds, which are simulated within $\tilde{O}(D)$ rounds in G by applying Theorem B.14, the exact details of this simulation on G^*_{virt} are provided in the proof of Theorem 1.3.

Next, we show how to find the cut vertices, in particular, we show how to find a set S such that $P = \delta(S, V \setminus S)$. To do that, we take an approach similar to that described in the proof of Theorem 6.1, where we define a graph R', in which, each P-edge is assigned a weight of one, and all other edges a weight of zero, we then again use the minor-aggregation SSSP approximation algorithm of [42] with an arbitrary constant approximation parameter ϵ in order to compute a $(1 + \epsilon)$-approximate SSSP tree T from s in R'. In particular, we modify the algorithm to work with edges with weights in $[0, n^{O(1)}]$ rather in $[1, n^{O(1)}]$, as shown in the proof of Theorem 1.3. Finally, vertices whose distance is zero from s are in S (other vertices are in $V \setminus S$).

The round complexity of simulating the SSSP approximation algorithm is $D \cdot n^{o(1)}$, since the part-wise aggregation problem can be solved on G in $\tilde{O}(D)$ CONGEST rounds (Corollary B.6), and the approximate SSSP algorithm of [42] runs in $n^{o(1)}$ minor-aggregation rounds. Thus, by Lemma B.8 simulating the algorithm incurs a total of $D \cdot n^{o(1)}$ CONGEST rounds in G. Note, after applying the distance approximation algorithm, vertices of G know locally to what side of the cut do they belong. □

7 Directed Global Minimum Cut

In this section, we show that our labeling algorithm from Section 5 for the dual graph G^* can be used to compute the directed global minimum cut in the primal graph G. The problem asks to find a bisection $(S, V \setminus S)$ of the vertex set, such that, the set of edges C leaving S to $V \setminus S$ is of minimal total weight.

Theorem 1.5 (Planar Directed Global Minimum Cut). *Given a directed planar graph G with non-negative integral edge weights, there is an $\tilde{O}(D^2)$ round randomized algorithm that finds a global directed minimum cut w.h.p..*

In order to prove Theorem 1.5, we work with *darts*: each directed edge is represented by two darts in opposite directions. For a definition, see Section 5.1. Specifically, we rely on a dart version of the cycle-cut duality (Theorem 4.6.2 in [25]).

Fact 7.1 (Dart Cycle-Cut Duality). Let G be a connected embedded planar graph (in the undirected sense). A set of darts C is a *simple dart cycle* in G^* if and only if C is a simple cut in G. Where a cycle is a simple dart cycle iff it is a simple cycle that uses no dart and its reversal, and, a cut is simple iff when removed it disconnects the graph into two connected components (in the undirected sense).

The reason we chose to work with darts is simply that the duality does *not* hold for general directed planar graphs, i.e., when an edge may be present without its reversal dart. For example, the minimum cut in G, when G is a directed cycle, is simply a single directed edge (i.e., there are two edges in the cut but only one of them is directed from S to $V \setminus S$). Hence, in the dual it corresponds to the edge $e = (f, f_\infty)$, where f (resp. f_∞) is the dual node that corresponds to the interior (exterior) of the cycle, since e is a single edge that connects two distinct nodes, it is not a cycle. See Figure 15.
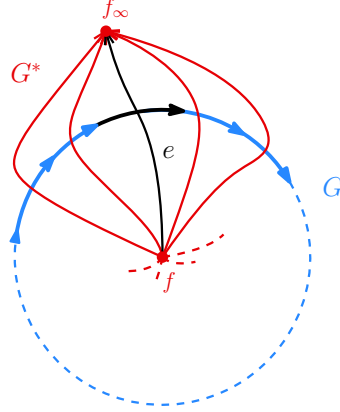
Figure 15: In blue, the primal graph G (a clockwise directed cycle). In red, the dual G^* (two nodes with parallel edges in only one direction). The black edge e is the minimal weight edge in G, constituting the directed global minimum cut in G, whilst its dual is the (single) black edge in G^*. e is not a cycle in G^*.

Thus, we augment G with the reversal dart of all of its edges. Since the problem of interest is the minimum cut, all of those darts are assigned a weight of zero. This guarantees that the value of the minimum cut remains the same. Henceforth, by Fact 7.1, it is enough to compute the directed minimum weight simple dart cycle in G^*.

Proof. By Fact 7.1, assuming G is connected in the undirected sense, the minimum weight directed global cut C of G is given by the minimum weight simple cycle of darts in G^*. We compute this cycle in G^* in a bottom-up fashion on the extended BDD of G. Some terms are borrowed from the proof of Theorem 2.1 (the distance labeling algorithm of G^*), e.g. the dual separator F_x and the dense distance graph $DDG(g)$. Thus, the reader is referred to Section 5.3 to recall the details of Theorem 2.1's proof.

The general idea is simple, given an extended BDD of G, by Lemma 5.15 , any shortest path (or cycle) of G^* either intersects a dual node in F_G (the separator of G^*) or is entirely contained in one of G^*'s child bags. Thus, in order to find C, we take the minimum over: (i) directed cycles that intersect F_G, and (ii) minimum directed cycles in G^*'s child bags (found recursively). Note, we need to ensure that all cycles found are simple dart cycles, which introduces a further complication to the general approach. The following describes in detail how to find the desired cycle of G^* and then infer the cut in G.

Preprocessing: We work with the network obtained from G by adding the reversal dart of each edge to the graph, such that, the two darts of an edge are ordered one after another in the embedding. We abuse notation and denote this network by G. The reversal dart gets assigned a weight of zero. It is not hard to see that the new network can be constructed and embedded in the plane without any need for communication (assuming an embedding of the original network). Note that any CONGEST round on the augmented network is simulated within two rounds on the communication graph.

Next, in $\tilde{O}(D^2)$-rounds, by Theorem 2.1, we construct an extended BDD $\mathcal{T}$ and compute distance labels for each node in each bag $X^* \in \mathcal{T}$. The algorithm of Theorem 2.1 succeeds to compute distance labels in G^* and does not abort since all weights are non-negative (i.e., there are no negative cycles in G^*).

Leaf bag case: Each leaf bag X^* of $\mathcal{T}$ is gathered in all vertices of X in $\tilde{O}(D)$ rounds (Property 10 of $\mathcal{T}$). Thus, the minimum weight simple dart cycle of each such X^* is computed locally and is known to all vertices in X, this cycle is denoted by C_X and is assigned a unique ID, that of the minimal ID node on it concatenated with its weight.

Non-leaf bag case: Let X^* be a non-leaf bag of $\mathcal{T}$, we first find the minimum weight cycle that crosses between X^*'s child bags, i.e., a cycle that did not appear in any child bag of X^*, (hence, crosses F_x), by distance computations in $DDG(f)$ for an arbitrary $f \in F_x$. Then, we take the minimum between this cycle and cycles found in the previous recursion on the child bags of X^* (cycles entirely contained in a child bag of X^*). Note that: (1) $DDG(f)$ is the same graph for all $f \in F_x$, and (2) all (primal) vertices on each F_x node know the distance labels of all nodes in F_x.

1. $ID(C_{X_i})$ for each child bag X_i of X is aggregated on X in $\tilde{O}(D)$ rounds so that the ID of the minimum weight cycle C_X (that does not cross F_x) is known to all vertices in X. Note, vertices in X_is know $ID(C_i)$, thus they can broadcast it (this is obviously true for leaf bags, for non-leaf bags, this is recursively maintained in the next two steps). Since X is of $\tilde{O}(D)$ diameter, this aggregation terminates in $\tilde{O}(D)$ rounds. Next, we consider cycles that cross F_x.

2. The $f \in F_x$ with maximum ID is found first. Due to the fact that X is of diameter $\tilde{O}(D)$ and since all vertices of X know the IDs of faces and face-parts they participate in (Properties 5 and 13 of $\mathcal{T}$), this aggregation of IDs terminates in $\tilde{O}(D)$ rounds.

3. Finally, all vertices of X locally construct $DDG(f)$ and compute the minimum weight simple dart cycle C by investigating the following two options related to S_x (the edges of the primal separator, see Section 5.1 for exact definitions).

   (a) *C uses an S_x dart:* In this case, one can deduce C by iterating over S_x darts present in $DDG(f)$. For each such dart $d = (g, h)$, we remove its reversal from $DDG(f)$ and compute SSSP from h. Note, the shortest h-to-g path found maps to a shortest simple h-to-g path P in X^* that
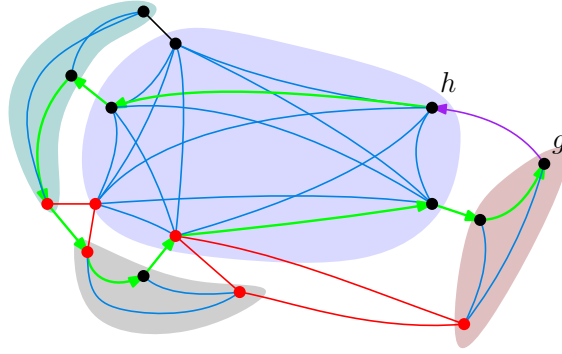
Figure 16: $DDG(f)$ in the iteration of the dart (g, h) (purple), where each shadow color demonstrates a distinct child bag of X^*. Black vertices are endpoints of S_x edges and red vertices are node parts in X^*'s child bags. The reversal dart of (g, h) is removed from the DDG. All other edges have both their darts present (for simplicity, some are undirected in the picture). The green directed path demonstrates a shortest h-to-g path that closes a cycle with the dart (g, h).

does not use rev(d). P is simple because there are no negative cycles in G^*. The following claim is a corollary of Lemma 5.18.

Claim. *$DDG(f) \setminus \{D_X\}$ preserves distances in $X^* \setminus \{D_X\}$, where D_X is a subset of S_x darts that are present in $DDG(f)$.*

Due to the above claim, we get that P combined with d gives the minimum weight simple dart cycle that uses d, denoted C_d. We define C to be the minimum over C_d for all S_x darts

d. If C is of weight strictly less than C_X, then C_X is set to C and is identified with the ID of f concatenated with the cycle's weight.

(b) *C does not use S_x:* In this case, we iterate over node-parts g_i in F_X, in each iteration we remove S_x darts from $DDG(f)$ and all clique edges between all node-parts $g_2, g_3, \ldots$ that correspond to the same face g of G as g_i. Then, we compute the shortest path in $DDG(f)$ from g_i. We

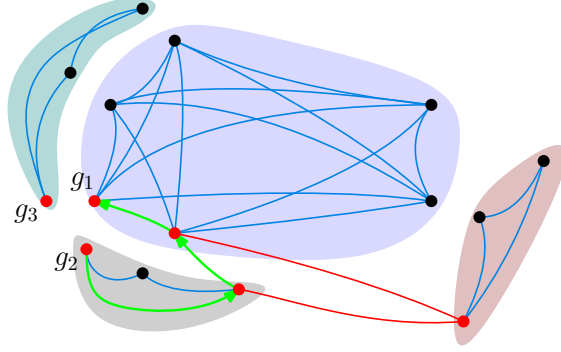Figure 17: $DDG(f)$ in the iteration of node-part g_2. g_1, g_2, g_3 correspond to the face g of G, where each shadow color demonstrates a distinct child bag of X^*. Black vertices are endpoints of S_x edges and red vertices are node parts in X^*'s child bags. All S_x darts are removed from the DDG, as well as clique edges on g_1, g_2, g_3. All other edges have both their darts present (for simplicity, not some are undirected in the picture). The green directed path demonstrates a shortest g_2-to-g_1 path, representing a cycle that contains g in G^*.

define C_g to be the shortest path found from any g_i to any $g_j \neq g_i$. If C_g is of weight strictly less than C_X, then C_X is set to C_g and is identified with the ID of g concatenated with the cycle's weight.

C_g corresponds to a simple dart cycle in G^*. A cycle since g_i, g_j are parts of the same node g of G^*; And simple since C_g uses a dart incident to g_i and a dart incident to g_j, those darts are not the reversal of each other because each of g_i, g_j is in a distinct child-bag of X^*. I.e., the cycle C_g in G^* leaves g with a dart incident to g_i and ends entering g with a dart incident to g_j and those darts are not the reversal of each other. The removal of the dart incident to g_j from C_g in G^* gives a path P, P is simple due to the fact that it is a shortest path, hence P with a dart that its reversal does not appear in P constitutes a simple dart cycle, C_g. Note, we removed the clique edges in the DDG to have that the shortest g_i-to-g_j path in the DDG is not both darts of a clique edge (which is also not an edge of G^*).

The correctness of the cycle's weight is again due to the claim above, where D_X is set to contain all S_x darts. Note, removal of clique edges may only increase distances, however, all distances computed still map to paths in G^*.

Cut edges and bisection: After the recursive process terminates, each $v \in G$ knows the weight of C_G and an ID of a node f of G^* that participates in C_G. We aim to extend this information to find the edges and a corresponding bisection of the cut in G. Note, even if the node identified with the cycle was a face-part f' in the bag X where the cycle was found, we can locally identify the face f of G that f' is a part of, as the ID of f is contained in the ID of f' (Property 13 of the extended BDD).

Claim. *The minimal weight simple dart cycle in G^* can be decomposed into a shortest f-to-g path P and a dart $d = (g, f)$, where g is the node that minimizes the total weight of $P \cup \{d\}$ s.t. $P \neq \mathrm{rev}(d)$.*

To prove the above claim, note, the desired cycle contains f, thus, f has two incident darts in the cycle, specifically, there is a dart d that enters it, denote d's other endpoint by g, we get that the cycle minus d is a shortest f-to-g path P (otherwise, a shorter cycle exists). Since the cycle is a simple dart cycle, we have that P must not be the reversal of d. g obviously minimizes the weight of $P \cup \{d\}$ (otherwise, a shorter cycle exists).

Henceforth, to find the edges of the cycle in G^*, we compute an SSSP tree T rooted at f in G^*. Then, each non-T dart entering f determines a single cycle. We need to find such a simple dart cycle $P \cup \{d\}$ of total minimal weight where $P \neq \text{rev}(d)$. To do that, we first need to discard all reversals $\text{rev}(d)$ of the darts d that leave f in T, because all such $\text{rev}(d)$s close non-simple dart cycles with T. Then, we can indeed look for the dart $d = (g, f)$ entering f that minimizes the (only) cycle of $T \cup \{d\}$. We do that by aggregations on G^* using the minor-aggregation model (Section 4).

1. By the proof of Lemma 2.2, within additional $\tilde{O}(D)$ rounds after computing distance labels, using aggregations on G^*, we can obtain the tree T.

    Note that T is represented by $\hat{G}$, our tool for aggregations on G^*. Thus, we next work with it and with minor-aggregations on G^* and T, which by Section 4 is possible since (1) we can do that on G^* or any minor of it, and (2) T is a subgraph (minor) of G^*. For an extended discussion of $\hat{G}$ and minor-aggregations on G^*, see Sections 3 and 4, and Appendices A and B.

2. f and each node g of its children in T both discard the dart $\text{rev}(d)$, where d connects f and g in T, from participating in the algorithm, ensuring that no dart closes a non-simple dart cycle with T. This is done locally in $\hat{G}$: the endpoints u, v of the edge simulating d in $\hat{G}$, know that d connects f and a child of it g in T, thus, u, v stop considering their other common edge that simulates $\text{rev}(d)$ as an input for the aggregations on $\hat{G}$ that simulate aggregations on G^* (and T).

3. Now we work with minor-aggregations on G^*. Since each node g of G^* knows its distance from f (the weight of the f-to-g T-path P), for the next aggregation, all active darts (g, f) choose an input value of their weight plus the weight of P, other darts (that f is not their head) chose an identity value. Thus, within one minor-aggregation round, f computes the minimum over the input values of its incident active darts. This allows f to learn the dart d that closes the minimal weight simple dart cycle with T, determining C_G as the only cycle in $T \cup \{d\}$. d marks itself as a dart of C_G.

4. Next we want to mark the rest of the cycle (i.e. the path P). In order to do so, f broadcasts the ID of d to the entire graph G^* in one minor-aggregation round. Allowing the other endpoint g of d to learn that $C_G \setminus \{d\}$ is the T-path P from f to it. Then, the node g chooses an input value of one, all other nodes of G^* chose zero, and a subtree sum is computed over T in $\tilde{O}(1)$ minor-aggregation rounds as shown in [18]. The only path of T whose nodes got a sum of one is P. In one minor-aggregation round, darts on P mark themselves (those darts are the only darts with sum of one for each of their endpoints).

    Turning back to the primal graph G, we have that vertices incident to darts of C_G know those darts. I.e., the cut edges are known.

5. To obtain a bisection, by Fact 7.1, it is sufficient to compute connected components in the underlying undirected graph after the removal of C_G darts and their reversal $\text{rev}(C_G)$ from G. I.e., each component is a side of the cut, and S is the component which the darts C_G leave.

    We do that by a Boruvka-like algorithm using aggregations (low-congestion shortcuts) on G (e.g. the connectivity algorithm of [14]), where we discard C_G and $\text{rev}(C_G)$ corresponding edges [13] from

---

[13]Edges and not darts, as we talk about an undirected graph.

participating as an input. Thus, each vertex of G learns the ID of the connected component containing it out of two.

Finally, we identify the component S: any vertex v incident to a dart of C_G leaving it, broadcasts its component ID to the entire graph G over a BFS tree. Since there are only two distinct IDs passed, the broadcast terminates in $O(D)$ rounds. Hence, vertices with the same component ID learn they are in S, other vertices learn they are in $V \setminus S$.

Note, this phase requires $\tilde{O}(D)$ CONGEST rounds, as each step either performs $\tilde{O}(D)$ CONGEST rounds on G, or $\tilde{O}(1)$ minor aggregation rounds on G^*, which also boil down to $\tilde{O}(D)$ CONGEST rounds by Section 4. □

References

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

[2] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. Maximum flow and minimum-cost flow in almost-linear time. In *63rd FOCS*, pages 612–623, 2022.

[3] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Information and Control*, 70(1):32–53, 1986.

[4] Tijn de Vos. Minimum cost flow in the congest model. In *30th SIROCCO*, pages 406–426, 2023.

[5] Michal Dory, Yuval Efron, Sagnik Mukhopadhyay, and Danupon Nanongkai. Distributed weighted min-cut in nearly-optimal time. In *53rd STOC*, pages 1144–1153, 2021.

[6] Jinfeng Dou, Thorsten Götte, Henning Hillebrandt, Christian Scheideler, and Julian Werthmann. Brief announcement: Distributed construction of near-optimal compact routing schemes for planar graphs. In *41tst PODC*, pages 67–70, 2023.

[7] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[8] Silvio Frischknecht, Stephan Holzer, and Roger Wattenhofer. Networks cannot compute their diameter in sublinear time. In *23rd SODA*, pages 1150–1162, 2012.

[9] Juan A. Garay, Shay Kutten, and David Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. *SIAM J. Comput.*, 27(1):302–316, 1998.

[10] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. Distance labeling in graphs. In *12th SODA*, pages 210–219, 2001.

[11] Pawel Gawrychowski, Shay Mozes, and Oren Weimann. Minimum cut in $O(m \log^2 n)$ time. In *47th ICALP*, pages 57:1–57:15, 2020.

[12] Barbara Geissmann and Lukas Gianinazzi. Parallel minimum cuts in near-linear work and low depth. In *30th SPAA*, pages 1–11, 2018.

[13] Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks I: planar embedding. In *34th PODC*, pages 29–38, 2016.

[14] Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks II: low-congestion shortcuts, mst, and min-cut. In *27th SODA*, pages 202–219, 2016.

[15] Mohsen Ghaffari and Bernhard Haeupler. Low-congestion shortcuts for graphs excluding dense minors. In *39th PODC*, pages 213–221, 2021.

[16] Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *33rd PODC*, pages 81–90, 2015.

[17] Mohsen Ghaffari and Merav Parter. Near-Optimal Distributed DFS in Planar Graphs. In *31st DISC*, pages 21:1–21:16, 2017.

[18] Mohsen Ghaffari and Goran Zuzic. Universally-optimal distributed exact min-cut. In *40th PODC*, pages 281–291, 2022.

[19] Bernhard Haeupler, Taisuke Izumi, and Goran Zuzic. Low-congestion shortcuts without embedding. *Distributed Comput.*, 34(1):79–90, 2021.

[20] Refael Hassin. Maximum flow in (s, t) planar networks. *Inf. Process. Lett.*, 13(3):107, 1981.

[21] Stephan Holzer and Roger Wattenhofer. Optimal distributed all pairs shortest paths and applications. In *30th PODC*, pages 355–364, 2012.

[22] Qiang-Sheng Hua, Lixiang Qian, Dongxiao Yu, Xuanhua Shi, and Hai Jin. A nearly optimal distributed algorithm for computing the weighted girth. *Sci. China Inf. Sci.*, 64(11), 2021.

[23] David R. Karger. Using randomized sparsification to approximate minimum cuts. In *5th SODA*, pages 424–432, 1994.

[24] David R. Karger. Minimum cuts in near-linear time. *J. ACM*, 47(1):46–76, 2000.

[25] Philip Klein and Shay Mozes. *Optimization Algorithms for Planar Graphs*. https://planarity.org. Book draft.

[26] Jason Li and Merav Parter. Planar diameter via metric compression. In *51st STOC*, pages 152–163, 2019.

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

[28] Vignesh Manoharan and Vijaya Ramachandran. Computing minimum weight cycle in the CONGEST model. In *43rd PODC*, pages 182–193, 2024.

[29] Gary L. Miller. Finding small simple cycle separators for 2-connected planar graphs. In *16th STOC*, pages 376–382, 1984.

[30] Gary L. Miller and Joseph Naor. Flow in planar graphs with multiple sources and sinks. *SIAM Journal on Computing*, 24:1002–1017, 1995.

[31] Shay Mozes, Kirill Nikolaev, Yahav Nussbaum, and Oren Weimann. Minimum cut of directed planar graphs in $O(n \log \log n)$ time. In *29th SODA*, pages 477–494, 2018.

[32] Sagnik Mukhopadhyay and Danupon Nanongkai. Weighted min-cut: sequential, cut-query, and streaming algorithms. In *52nd STOC*, pages 496–509, 2020.

[33] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *46th STOC*, pages 565–573, 2014.

[34] Merav Parter. Small cuts and connectivity certificates: A fault tolerant approach. In *33rd DISC*, pages 30:1–30:16, 2019.

[35] Merav Parter. Distributed planar reachability in nearly optimal time. In *34th DISC*, volume 179, pages 38:1–38:17, 2020.

[36] David Peleg. *Distributed Computing: A Locality-Sensitive Approach*. Society for Industrial and Applied Mathematics, 2000.

[37] David Peleg, Liam Roditty, and Elad Tal. Distributed algorithms for network diameter and girth. In *39th ICALP*, pages 660–672, 2012.

[38] John H. Reif. Minimum s-t cut of a planar undirected network in $O(n \log^2 n)$ time. *SIAM J. Comput.*, 12(1):71–81, 1983.

[39] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Panduran-gan, David Peleg, and Roger Wattenhofer. Distributed verification and hardness of distributed approximation. In *43rd STOC*, pages 363–372, 2011.

[40] Václav Rozhon Ⓡ Bernhard Haeupler Ⓡ Anders Martinsson Ⓡ Christoph Grunau Ⓡ Goran Zuzic. Parallel breadth-first search and exact shortest paths and stronger notions for approximate distances. In *55th STOC*, pages 321–334, 2023.

[41] Václav Rozhon Ⓡ Christoph Grunau Ⓡ Bernhard Haeupler Ⓡ Goran Zuzic Ⓡ Jason Li. Undirected (1+ϵ)-shortest paths via minor-aggregates: near-optimal deterministic parallel and distributed algorithms. In *54th STOC*, pages 478–487, 2022.

[42] Goran Zuzic Ⓡ Gramoz Goranci Ⓡ Mingquan Ye Ⓡ Bernhard Haeupler Ⓡ Xiaorui Sun. Universally-optimal distributed shortest paths and transshipment via graph-based l_1-oblivious routing. In *33rd SODA*, pages 2549–2579, 2022.

[43] Mikkel Thorup. Fully-dynamic min-cut. In *33rd STOC*, pages 224–230, 2001.

[44] Helge A. Tverberg. A proof of the jordan curve theorem. *Bulletin of The London Mathematical Society*, 12:34–38, 1980.

[45] Shankar M. Venkatesan. *Algorithms for network flows*. Ph.D. thesis, Pennsylvania State University, 1983.

[46] Christian Wulff-Nilsen. Girth of a planar digraph with real edge weights in $O(n \log^3 n)$ time. *CoRR*, arXiv:0908.0697, 2009.

A The Graph $\hat{G}$

We next give the full definition of the graph $\hat{G}$, and prove its properties. Let $\deg(v)$ denote the undirected degree of a vertex v in G. Then, a vertex v in G has $\deg(v) + 1$ copies in $\hat{G}$: A copy v referred to as a *star center* (we denote by $V_S = V(G)$ the set of all star centers) and a copy v_i for each *local region* of v. A local region of v is defined by two consecutive edges in v's local embedding. Notice that v has $\deg(v)$ local regions, and that each local region belongs to a face of G (it is possible that several local regions of v belong

to the same face). Eventually, we will have that the faces of G correspond to (vertex and edge disjoint) faces of $\hat{G}$. If i local regions of v belong to the same face f of G, then v will have i copies in the face of $\hat{G}$ that corresponds to f. The vertices of $\hat{G}$ are therefore $V_S \cup_{v \in G} N(v)$ where $N(v) = \{v_1, \ldots, v_{\deg(v)}\}$.

We now define the edges of $\hat{G}$ in three parts $E(\hat{G}) = E_S \cup E_R \cup E_C$. The *star edges* E_S are of the form (v, v_i), the edges E_R are of the form (v_i, u_j) (for $u \neq v$) and the edges E_C are of the form (v_i, v_j) (for $i \neq j$). For every edge $e = (u, v)$ we need endpoints of e to figure out which edges (v_i, u_j) belong to $\hat{G}$ so that $\hat{G}$ is planar and faces of G map to faces of $\hat{G}$. While a vertex of G does not know any global information about the faces of G, adjacent vertices know that there is some face that contains both of them.
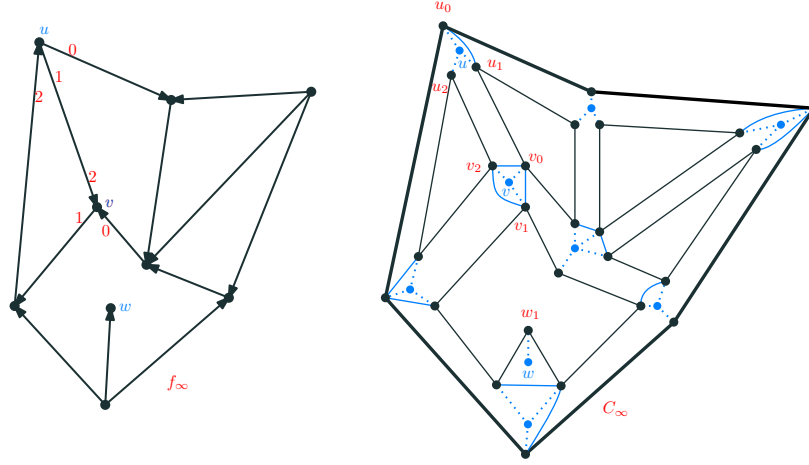

Figure 18: On the left, the primal graph G. The clockwise local numbering of u and v's edges is in red. f_∞ denotes the infinite face of G. On the right, the face-disjoint graph $\hat{G}$. The face f_∞ of G maps to the face C_∞ (thick black) of $\hat{G}[E_R]$. Vertex v in G has a star-center copy v in $V_s \subset V(\hat{G})$, and $\deg(v)$ additional duplicates v_0, v_1, v_2 in $V(\hat{G})$ connected by star edges $E_S \subset E(\hat{G})$ (dashed blue). The edge (u, v) of G maps to three edges in $\hat{G}$: an edge (v_0, v_2) in $E_C \subset E(\hat{G})$ (solid blue) and two edges (u_2, v_2) and (u_1, v_0) in $E_R \subset E(\hat{G})$ (black edges). The vertex $w \in G$ is of degree one, thus, it has a star-center copy and one additional copy in $\hat{G}$ connected by a dashed blue edge. Its only incident edge in G has two duplicates, each connected to a distinct non star-center copy of its only neighbor. The non star-center copies of w's neighbor are connected by a solid blue edge (corresponding to the edge incident to w), this edge corresponds to a self loop in G^* as it is connects the same face to itself (the face containing w).

E_S: These edges connect every star center v and its copies $v_i \in N(v)$.

E_R: For each edge $(u, v) \in G$, there are two edges in E_R, one copy for each face of G that contains (u, v). Let (u, v) be the i'th and j'th edge in the clockwise local embedding of u and v, respectively. Then the edges added to E_R are (u_{i+1}, v_j) and (u_i, v_{j+1}). This is because the face containing the i'th and $(i+1)$'th edges of u is the face containing the j'th and $(j-1)$'th edges of v (and the face containing the i'th and $(i-1)$'th edges of u is the face containing the j'th and $(j+1)$'th edges of v).

E_C: In order to simulate computations on G^*, we make sure that if two faces of G share an edge e (i.e. are adjacent nodes in G^*) then their corresponding faces in $\hat{G}$ are connected by a corresponding edge in E_C, between two vertices v_j, v_{j+1} of the same star, since v_j, v_{j+1} each corresponds to a face that contains e. For an edge (u, v) we can add an edge in E_C between copies of u or copies of v, the choice of which such edge to add is arbitrary: For every edge (u, v) in G we add the edge (v_j, v_{j+1}) to E_C if v's ID is larger than u's, and (u, v) is the j'th edge in the clockwise local embedding of v.

In [17], $\hat{G}$ was defined without the edges E_C, as their purposes did not require it. However, the following properties (except for Property 5) of the face-disjoint graph were proven in [17] and are yet valid.

1. $\hat{G}$ is planar and can be constructed in $O(1)$ rounds (after which every vertex in G knows the information of all its copies in $\hat{G}$ and their adjacent edges).

2. $\hat{G}$ has diameter at most $3D$.

3. Any r-round algorithm on $\hat{G}$ can be simulated by a $2r$-round algorithm on G.

4. There is an $\tilde{O}(D)$-round algorithm that identifies G's faces by detecting their corresponding faces in $\hat{G}$. When the algorithm terminates, every such face of $\hat{G}$ is assigned a face leader that knows the face's ID. Finally, the vertices of G know the IDs of all faces that contain them, and for each of their incident edges the two IDs of the faces that contain them. Thus, each vertex of G knows for each pair of consecutive edges adjacent to it (using its clockwise ordering of edges) the ID of the face that contains them.

5. There is a 1-1 mapping between edges of G^* and E_C. Both endpoints of an edge in E_C know the weight and direction of its corresponding edge in G^* (if G^* is directed and/or weighted).

Proof.

(1) $\hat{G}$ can be constructed in $O(1)$ rounds, as the only information transmitted on an edge is its local clockwise numbering in its endpoints' combinatorial embedding.

For proving that $\hat{G}$ is planar, we show an explicit planar embedding, which can be deduced from the planar embedding of G. Consider first the star-center copy of v in $\hat{G}$. Recall that v has $\deg(v)$ copies $v_1, v_2, \dots$ that are incident to the star-center, the star center local clockwise embedding is $(v, v_1), (v, v_2), \dots$. Now consider a non star-center copy v_i of v, such that u and w are neighbors of v in G and the edge (u, v) precedes the edge (w, v) in v's local clockwise embedding. v_i is possibly incident to the following edges $(v_i, u_j), (v_i, v_{i+1}), (v_i, v), (v_i, v_{i-1}), (v_i, w_k)$ which we embed in that order.

(2) $\hat{G}$ has diameter at most $3D$ as each edge in G becomes a path $(u - u_i - v_j - v)$ in $\hat{G} \setminus E_C$, and adding E_C can only make the diameter smaller.

(3) Any r-round CONGEST algorithm on $\hat{G}$ can be simulated in $2r$ CONGEST rounds in G, because each edge in G has exactly two copies in E_R. These are the only edges that require simulation as edges $(v_i, v_j), (v, v_i)$ in $E_C \cup E_S$ are simulated locally by v.

(4) There is an $\tilde{O}(D)$ round algorithm that identifies G's faces by detecting their corresponding (vertex and edge disjoint) connected components of $\hat{G}[E_R]$ while considering $\hat{G}$ as the communication network. The algorithm is a variant of [13]'s connectivity algorithm, working in a Boruvka-like fashion performing $O(\log n)$ phases of merges.[14] When the algorithm terminates, every non star-center vertex of $\hat{G}$ knows its face ID and each face is assigned a leader. Hence, the vertices of G know the IDs of all faces that contain them, and for each of their incident edges the IDs of the two faces that contain them.

---

[14]The mentioned algorithm is randomized, however, the only randomized step in it is performing star-shaped merges, a primitive which was derandomized by [18] and can be used as a black box in [13]'s algorithm to derandomize it.

(5) There is a 1-1 mapping between edges of G^* and E_C. Every edge e^* of G^* corresponds to exactly one edge of e of G so we show the 1-1 mapping between edges of G and E_C. Each edge e in G maps to one edge $\hat{e}$ of E_C. Each edge $\hat{e}$ in E_C is of the form (v_i, v_{i+1}). v_{i+1} is connected to some u_j by an edge and v_i is connected to u_{j+1} by an edge. These two edge belong to E_R and correspond to one edge $e = (u, v)$ in G. We map the edge $\hat{e}$ to e.

Consider $e^* \in G^*$ that maps to $\hat{e} = (v_i, v_{i+1})$ in E_C. Since $\hat{e}$ connects copies of $v \in G$ then it is mapped to an incident edge e of v, meaning, v knows e's weight and direction. Thus, $\hat{e}$ is assigned the same weight as e, and the direction of $\hat{e}$ is defined to be (v_i, v_{i+1}) (resp. (v_{i+1}, v_i)) if and only if $e = (v, u)$ (resp. $e = (u, v)$). The directions are defined this way for the following reason. Consider an edge $e = (v, u) \in G$, consider the clockwise local numbering of v of e, denote the face to the left of e by v's numbering by f and the face to the right of e by v's numbering by g, then, $e^* := (f, g)$. And indeed, when $e = (v, u)$, then, v_i is in the face that represents f in $\hat{G}$ (the face to the left of e) and v_{i+1} is in the face that represents g (the face to the right of e), and vise versa in case $e = (u, v)$. Meaning, when e^* is directed from f to g, $\hat{e}$ is directed from the face that represents f to the face that represents g.

The above can be achieved locally after construction, as the only information needed to one endpoint of an edge is the other endpoint's local numbering of that edge.

$\square$

B Deferred Details and Proofs from Section 4

B.1 Preliminaries

Low-congestion shortcuts and part-wise aggregation. In recent years, several breakthroughs in distributed computing were based on the notion of *low-congestion shortcuts* introduced by Ghaffari and Haeupler [14] (first for planar graphs, then for other graph families [15,19]). Consider a partition $\{S_1, \dots, S_N\}$ of V where every $G[S_i]$ is connected.

Definition B.1 (Low-Congestion Shortcuts). An α-congestion β-dilation shortcut of G, denoted (α, β)-shortcut, is a set of subgraphs $\{H_1, \dots, H_N\}$ of G such that the diameter of each $G[S_i] \cup H_i$ is at most $O(\beta)$ and each edge of G participates in at most $O(\alpha)$ subgraphs H_i.

Definition B.2 (Shortcut Quality). The shortcut quality $SQ(G)$ of a graph G is the smallest α for which G admits an (α, α) low-congestion shortcut.

Definition B.3 (Aggregation Operators). Given a set of b-bit strings $\{x_1, \dots, x_m\}$ and an aggregation operator $\oplus$ (AND /OR /SUM etc.), their aggregate $\bigoplus_i x_i$ is defined as the result of repeatedly replacing any two strings x_i, x_j with the (b-bit) string $x_i \oplus x_j$, until a single string remains.

Definition B.4 (Part-Wise Aggregation). Assume each $v \in S_i$ has an $\tilde{O}(1)$-bit string x_v. The PA problem asks that each vertex of S_i knows the aggregate function $\bigoplus_{v \in S_i} x_v$.

The following lemma captures the relation between the PA problem and shortcuts.

Lemma B.5 (Part-wise aggregation via low-congestion shortcuts [14, 15, 19]). *If G admits an (α, β)-shortcut that can be constructed in r rounds, then the PA problem can be solved on G in $\tilde{O}(r + \alpha + \beta)$ rounds. In particular, the PA problem can be solved on G in $\tilde{O}(r + SQ(G))$ rounds, where r is the number of rounds for constructing shortcuts of quality $SQ(G)$.*

Since a planar graph G of hop-diameter D admits shortcuts of quality $SQ(G) = \tilde{O}(D)$ [14, 15, 19] that can be constructed in $\tilde{O}(D)$ rounds we get:

Corollary B.6. *The PA problem can be solved on a planar graph G of hop-diameter D in $\tilde{O}(D)$ rounds.*

The Minor-Aggregation model.

Definition B.7 (Basic Minor-Aggregation Model [42])**.** Given an undirected graph $G = (V, E)$, both vertices and edges are computational entities (i.e. have their own processor and their own private memory). Each vertex has a unique $\tilde{O}(1)$-bit ID. Communication occurs in synchronous rounds. All entities wake up at the same time, each vertex knows its ID, and each edge knows its endpoints' IDs. Each round of communication consists of the following three steps (in that order):

1. *Contraction step.* Each edge e chooses a value in $\{0, 1\}$. Contracting all edges that chose 1 defines a minor $G' = (V', E')$ of G. The nodes V' of G' are subsets of V called *super-nodes*.

2. *Consensus step.* Each node $v \in V$ chooses an $\tilde{O}(1)$-bit value x_v. For each super-node $s \in V'$, let $y_s = \bigoplus_{v \in s} x_v$, where $\oplus$ is a pre-defined aggregation operator. All $v \in s$ learn y_s.

3. *Aggregation step.* Each edge $e = (a, b) \in E'$ learns y_a and y_b, then chooses two $\tilde{O}(1)$-bit values, $z_{e,a}$ and $z_{e,b}$. For each super-node $s \in V'$, let $I(s)$ be the set of incident edges of s, then, $z_s = \bigotimes_{e \in I(s)} z_{e,s}$, where $\otimes$ is a pre-defined aggregation operator. All $v \in s$ learn the same z_s.

It was shown by [42] that using low-congestion shortcuts, any minor-aggregation algorithm can be simulated in the standard CONGEST model. It is easy to check that the consensus and aggregation steps of the minor-aggregation model can indeed be phrased as PA problems, and it was shown in [42] that the contraction step can be broken to a series of $O(\log n)$ PA problems. More specifically, it was proved that any efficient ($\tilde{O}(1)$-round) minor-aggregation algorithm implies an efficient CONGEST algorithm on G if one can solve the PA problem efficiently on G. Our goal is to show a similar result for the dual graph G^*.

Lemma B.8 (Minor-Aggregation Model Simulation in CONGEST via PA [18, 42])**.** *Any τ-round minor-aggregation algorithm can be simulated in the CONGEST model on G in $O(\tau \cdot N)$ rounds where N is the CONGEST round complexity for solving the PA problem on G.*

Recall (Lemma B.5) that the PA problem on G can be solved in $N = \tilde{O}(r + SQ(G))$ CONGEST rounds where r is the round complexity for constructing shortcuts of quality $SQ(G)$. Therefore, the above lemma implies that any τ-round minor-aggregation algorithm can be simulated in the CONGEST model in $\tilde{O}(\tau \cdot (r + SQ(G)))$ rounds, which is $\tilde{O}(\tau \cdot D)$ for planar graphs (by Corollary B.6).

B.2 Minor-Aggregation for the Dual Graph

We prove that any minor-aggregation algorithm A can be compiled into a CONGEST algorithm that runs in the planar network of communication G, and simulates A on the dual graph G^*. An application of this, is a CONGEST algorithm for finding the minimum-weight cycle of an undirected weighted planar graph G. In this case, the algorithm A is a min-cut algorithm, and so running A on G^* gives the minimum-weight cycle of G by Fact 3.1 (to be shown later in this section). Another application is an approximation for maximum st-flow when the graph is undirected and s, t lie on the same face. In that case, the algorithm A is an approximate single source shortest paths algorithm (see Section 6 for details and proof).

Since G^* is not in hand, a straightforward CONGEST simulation on G^* might suffer a linear blow-up in the round complexity. Moreover, we cannot directly follow the method of [42] of compiling the

Minor-Aggregation model to CONGEST, since it is unclear how to do so on G^*. Instead, we combine their method with part-wise aggregations on G^* via $\hat{G}$.

Using aggregations on $\hat{G}$, aggregations on G^* were previously done by Ghaffari and Parter [17] to solve the specific problem of computing a primal *balanced separator*, thus, their aggregation tasks were problem-specific (aggregations on faces of G and sub-tree sums on G^*). For our purposes, we need something more general, so we extend their method to perform general part-wise aggregations on G^*. In particular, we need to perform aggregations that take into consideration the outgoing edges of each part S_i, something that was not done in [17]. This specific task required our small modification to $\hat{G}$ compared to the one defined by Ghaffari and Parter.

Lemma B.9 (Part-Wise Aggregations on G^*). *Let $P = \{S_1, S_2, \ldots, S_N\}$ be a partition of $V(G^*)$, such that, (1) all $G^*[S_i]$ are connected, (2) each vertex on a face $\hat{f}$ of $\hat{G}$ that correspond to a node $f \in S_i$, knows the same ID for S_i, (3) each input of a node f is known to the leader vertex of $\hat{f}$ in $\hat{G}$ and (4) each input of an edge e^* is known to the two endpoints in $\hat{G}$ of the edge $\hat{e} \in E_C$ that maps to e^*. Then, any aggregate operator over all S_i can be computed simultaneously in $\tilde{O}(D)$ CONGEST rounds, such that, the output value of each S_i is known to all vertices of G that lie on the faces in S_i.*

Proof. The computation is done via aggregations inside $\hat{G}$, in particular, a partition of G^* induces a partition on $\hat{G}$. Notice that $\hat{G}$ is planar, is built in $O(1)$ rounds, of diameter $O(D)$ and allows CONGEST simulation in G with a constant blow-up factor in the round complexity (Properties 1, 2 and 3 of $\hat{G}$). Thus, $\hat{G}$ admits $\tilde{O}(D)$ quality low-congestion shortcuts in $\tilde{O}(D)$ rounds and the PA problem is solved in $\tilde{O}(D)$ rounds on it (Lemma B.5 and Corollary B.6).

A partition P of G^* induces a partition $\hat{P}$ on $\hat{G}$ as follows. A node f of G^* maps to a face $\hat{f}$ of $\hat{G}$, the part $\hat{S}_i \in \hat{P}$ is defined to contain all vertices on faces $\hat{f}$ of $\hat{G}$ that map to nodes f of G^* in S_i. All $\hat{S}_i$ are connected and vertex-disjoint, as each two distinct $f, g \in G^*$ map to distinct faces $\hat{f}, \hat{g}$ of $\hat{G}$, in addition, $\hat{f}, \hat{g}$ are connected by an E_C edge for each edge in G^* connecting f, g. Notice that star-center vertices (V_S) are not in any part $\hat{S}_i$, so we define each to be its own part. Hence, one can see that $\hat{P} = (\cup_{i \in [N]} \{\hat{S}_i\}) \cup (\cup_{v \in V_S} \{v\})$ is a partition of $\hat{G}$ into connected vertex-disjoint subgraphs. Thus, to compute part-wise aggregation on P, we compute part-wise aggregations on $\hat{P}$, in which, vertices of V_S do not participate as an input to the part-wise aggregation task and are used exclusively for communication, henceforth, we do not refer to these vertices.

Initially, all vertices in a part $\hat{S}_i$ know the same ID of $\hat{S}_i$ (equivalently, ID of S_i). Thus, low-congestion shortcuts can be constructed. Recall that in the PA task, each vertex v in a part of the partition has an $\tilde{O}(1)$-bit input, and the aggregate operator is computed over all inputs of vertices which are in the same part. When computing a PA task over nodes of G^*, the input of $f \in G^*$ is required to be known to some vertex $\hat{v} \in \hat{G}$ (w.l.o.g. the leader) on the face of $\hat{G}$ that maps to f (Property 4 of $\hat{G}$). Then, aggregations on all $\hat{S}_i$ shall simulate aggregations on S_i, where only the input of the leader of $\hat{f}$ is considered as an input of the dual node f. That is, the aggregation task considers input from leaders only, and all other vertices of $\hat{G}$ participate in the algorithm just for the sake of communication (their input is an identity element).

If the PA task is over inside-part edges or over outgoing edges, then, each vertex v of $\hat{G}$ that is incident to an edge $\hat{e} = \{u, v\} \in E_C$ (recall, edges of E_C map 1-1 to $E(G^*)$ by Property 5 of $\hat{G}$)), knows if $\hat{e}$'s input should or should not be considered in the aggregation - as v knows the ID of the part $\hat{S}_i$ that contains it, and know the ID of $\hat{S}_j$ that contains u in one communication round. Then, aggregations on all $\hat{S}_i$ simulate aggregations on S_i, where any $v \in \hat{G}$ that participates in the aggregation considers some of its edges as an input depending on the task and on whether they are inside-part edges or not. Vertices that have no incident input edge, participate in the algorithm just for the sake of communication (their input is an identity element).

Finally, after the aggregation task is done, all vertices in $\hat{G} \setminus V_S$ know the output value of their part, which directly translates from $\hat{G}$ to G (as in Property 1 of $\hat{G}$). Notice, even though for any specific node

$f \in G^*$ there is no entity in $\hat{G}$ that knows all its "local" information (i.e. a list of its incident edges), we still can compute part-wise aggregations over G^*. $\qquad\square$

Now, having part-wise aggregations on G^*, we show a sketch of simulating of the minor-aggregation model in CONGEST via PA (as in [42]), where the input graph is the same as the network of communication. Then describe the differences with the similar proof for simulation on G^*. A vertex of G simulates itself, and the two endpoints of an edge e simulate e. The simulation consists of three main steps: (1) A few ($\tilde{O}(1)$) part-wise aggregation tasks are executed to elect a leader for each super node, defining the nodes of the obtained minor, (2) A part-wise aggregation task is executed to simulate consensus, where a part is $G[s]$ (s is some super-node), and finally (3) The endpoints of each edge exchange their parts' consensus values and one more part-wise aggregation task is performed to compute the aggregation values. A corollary of Lemmas B.8 and B.9 is the following theorem.

Theorem B.10 (Basic Minor-Aggregation Model for G^*). *Any τ-round minor aggregation algorithm A can be simulated on G^* within $\tilde{O}(\tau \cdot D)$ CONGEST rounds on G, such that, the output of each node $f \in G^*$ is known to all vertices of the corresponding face f of G, and the output of an edge $e^* \in G^*$ is known to the endpoints of $e \in G$.*

Proof. Let $s = \{f_1, f_2, \dots\}$ be a super-node of G^* ($f_i \in V(G^*)$), the vertices of $\hat{G}$ that map to s are the vertices that lie on the faces $\hat{f}_i$ of $\hat{G}$, such that, $\hat{f}_i$ maps to the node f_i. Then, the computational power of a super-node of G^* is simulated by all vertices of $\hat{G}$ that map to it. The computational power of an edge in (a minor of) G^* shall be simulated by the endpoints of its corresponding E_C edge in $\hat{G}$ (Property 5). Often, we will use Lemma B.9 as a black-box for PA on G^* in order to implement the simulation.

There are three steps in a minor-aggregate round: contraction, consensus and aggregation. In fact, there is an implicit initial step at the start of each round for electing a leader of each super-node. We follow the method of [42] and show how all steps compile down to PA in G^* and a few simple operations in $\hat{G}$ as follows.

1. *Electing a leader (and contraction).* Recall, in the contraction step, each edge has a value of zero or one, and considered to be contracted if and only if its value is one. To implement that, we have mainly two computational tasks. First, merge *clusters* of nodes of G^* over their outgoing edges of value one (clusters eventually define super-nodes of G^*). Second, assign each cluster an ID that is known to all vertices inside it.

   Initially, a cluster is a single node of G^*, then, clusters are grown to match the set of super-nodes of (the minor of) G^*.

   Implementing that, (1) connected components of $\hat{G}[E_R]$ are detected (those are faces of $\hat{G}$ that map to a node $v \in G^*$), where the ID of a node of G^* is the minimal ID of a vertex in $\hat{G}$ that maps to it, this is done in $\tilde{O}(D)$ rounds by property 4 of $\hat{G}$. Now each vertex of $\hat{G}$ knows in which of these connected components it is (defines a partition on G^* where each part is a single node) defining each node as a cluster. Notice that now Lemma B.9 (PA in G^*) is applicable, due to the fact that all vertices of $\hat{G}$ that correspond to a face f in a cluster, know the same ID for that cluster. Then, (2) we grow clusters. The merging process follows Boruvka's classic scheme, where the process breaks down into $O(\log n)$ star-shaped merges, the classic implementation is randomized and similar to that of [14, 17], however, one can implement this merging process deterministically with the derandomization of [18]. Namely, the PA task on G^* here is when each cluster defines a part in a partition of $V(G^*)$, and the aggregate operator is computed over the outgoing edges of each cluster, deciding the merges; the operator computed is over labels of neighboring clusters, where each cluster is labeled as a *joiner* or a *receiver*, such that, joiners suggest merges and receivers accept them (i.e., receivers are the star centers in the star-shaped merges). Randomly, the labels are decided by a fair

coin flip that the leader of the cluster tosses and broadcasts to the whole cluster, deterministically, the classic algorithm for 3-coloring by Cole and Vishkin [3] is used (for more details see [18]). At most $\tilde{O}(1)$ merges are repeatedly done until clusters match super-nodes of G^*. In each phase of merges, all nodes maintain the minimum ID of a node v (of G^*) in the cluster. The vertex of $\hat{G}$ that has that exact ID is considered to be the leader of vertices in $\hat{G}$ that map to the super-nodes represented by the cluster. Notice, there is such a vertex in $\hat{G}$ that has that exact ID since IDs of nodes of G^* are IDs of vertices in $\hat{G}$ that map to them. Notice that at the end of each phase Lemma B.9 (PA on G^*) is applicable, thus, applying it once in each of the $\tilde{O}(1)$ phases results with a round complexity of $\tilde{O}(D)$ for the entire process.

The above implements the contraction step, as now all vertices of $\hat{G}$ that simulate the same super node of G^* have a leader and a shared cluster-unique ID.

2. *Consensus.* After assigning IDs to clusters, as said, each vertex of $\hat{G}$ knows the ID of the cluster that it maps to, which is used as the part ID in the part-wise aggregation tasks on G^*. Regarding input, the input is (perhaps unique) per node of G^*, which elected leaders in $\hat{G}$ know, hence, Lemma B.9 is applicable and is used to implement the consensus step.

3. *Aggregation.* After performing the consensus step, each endpoint in $\hat{G}$ of an edge in E_C informs the other endpoint with the consensus value of the cluster that it maps to. Then, an endpoint $v \in \hat{G}$ of an edge $e \in E_C$ that maps to the edge $e^* = (a_1, a_2)$ (where a_1, a_2 are distinct super nodes of G^*), does the following. If v is in the cluster (that maps to) a_i, v thinks of the edge e as if it was assigned the value/weight of z_{e,a_i}, and an identity element otherwise. Then, the aggregate operator is computed over the outgoing edges of super-nodes, where the weight z_{e,a_i} is considered for the edge e in the cluster (that maps to) a_i. Again, the aggregation is done by Lemma B.9. Finally, the two endpoints of each edge know the output of the aggregate operator, thus, they can choose zero or one for that edge, and we are ready for the next round.

Notice that after the simulation finishes, each vertex in a cluster knows the output value of its cluster's corresponding super-node. Hence, by definition of $\hat{G}$ (see Property 1 of $\hat{G}$), for each $f^* \in G^*$ its output is known to all vertices on the corresponding face f of G, and the value of an edge e^* that maps to $e \in \hat{G}$ is known to e's endpoints in $\hat{G}$ as specified in the statement of Lemma B.9. □

The extended model. We now define the extended Minor-Aggregation model as presented in [18]. Then, we show that it can be simulated in G^*.

Definition B.11 (Extended Minor-Aggregation Model [18])**.** This model extends the above model to work on virtual graphs that have a small number of virtual nodes without strongly affecting the round complexity. In particular, (1) arbitrarily connected $\tilde{O}(1)$ virtual nodes can be added to the network, and (2) $\tilde{O}(1)$ nodes of the network, each can be replaced with an arbitrarily connected virtual node.

A virtual node is a node that does not exist in the input network (has no computational power) and must be simulated by nodes that do actually exist. Adding a virtual node v_{virt} means to decide to which other (virtual and real) nodes it is connected. Replacing a node v of the network with a virtual node v_{virt}, means that v does not participate in computations, instead, v_{virt} (that might be connected to different neighbors than v) is simulated by other (than v) non-virtual nodes of the network. An edge that is connected to a virtual node is a *virtual edge*. The obtained graph from the input network G by adding virtual nodes is called a *virtual graph* and denoted G_{virt}. All non-virtual nodes of G_{virt} are required to know the list of all (IDs of) virtual nodes in G_{virt}. A virtual edge connecting a non-virtual $u \in G_{virt}$ and a virtual $v \in G_{virt}$ is only known to u. A virtual edge between two virtual nodes in G_{virt} is required to be known by all non-virtual nodes of G_{virt}.

In our context, a node of the graph is a face of $\hat{G}$, thus, all vertices on a face $\hat{f}$ of $\hat{G}$ that maps to a node f of G^* shall know the (IDs of) all virtual nodes, virtual edges connecting f to virtual nodes of G^*_{virt} and virtual edges connecting two virtual nodes. Notice, this is feasible, as there are at most $\tilde{O}(1)$ virtual nodes, thus, there is a total of $\tilde{O}(1)$-bits that shall be known to vertices of $\hat{f}$. We now want to simulate a minor-aggregation algorithm A on G^*_{virt} that has at most $O(\beta)$ virtual nodes. That is, we want to (1) represent (store) the obtained network G^*_{virt} distributively as defined above, and (2) run any minor aggregation algorithm A on G^*_{virt} with $O(\beta)$ blow up factor in the minor-aggregate round complexity of A.

Given the additional $O(\beta)$ virtual nodes and virtual nodes that replace real nodes as input - that is, real nodes that are incident to a virtual node know its ID, and a real node that shall be replaced with a virtual node, knows itself as such. Ghaffari and Zuzic [18] showed the following.

Lemma B.12 (Lemma 15 and Theorem 14 of [18]). *There is an $O(\beta)$ minor-aggregation rounds deterministic algorithm A_1 that stores a virtual network G_{virt}. If multiple edges connect some virtual node v_{virt} with some neighbor, G_{virt} will contain a single edge with a weight equal to the sum (or any aggregate operator) of such edges in G.*

Lemma B.13 (Lemma 15 and Theorem 14 of [18]). *Let G_{virt} be a virtual graph obtained from a connected graph G, and A be an $O(\tau)$-round minor-aggregation algorithm. Then, there is a minor aggregation algorithm A_2 on G that simulates A on G_{virt} within $O(\tau\beta)$ minor-aggregate rounds. Upon termination, each non-virtual node v learns all information learned by v and all virtual nodes. The output of a virtual edge is known to all nodes if both of its endpoints are virtual, otherwise, its output is only known to its non-virtual endpoint.*

Our proof would make a black-box use of Lemma B.13, however, we provide a sketch of the proof of [18] just to make things a bit clearer (for the full proof see Theorem 14 of [18]). (1 - Contraction) Contract non-virtual edges that shall be contracted, then, in $O(\beta)$ rounds, each super node shall learn the IDs of virtual-nodes it is connected to, via computing an aggregate over its incident virtual edges. Then, the super-nodes' IDs get computed, (2 - Consensus) At first, all super-nodes that do not contain any virtual node perform their consensus step, then, we iterate over all virtual nodes and compute the consensus of the super-node they are contained in, such that, all nodes of the graph know the output value of each super node. Finally (3 - Aggregation) A real edge simulates itself, a virtual edge that has one non-virtual endpoint node is simulated by that endpoint and a virtual edge that the two of its endpoints are virtual is simulated by all nodes. The z value of each edge is computed and an aggregation is performed as in the consensus step.

Now we move to simulating the extended minor-aggregation model to run on G^*, which is a direct corollary of Lemmas B.12, B.13 and Theorem B.10. Let G^*_{virt} be a virtual graph that is obtained from G^* by adding and replacing nodes with up to β arbitrarily connected virtual nodes, then,

Theorem B.14. *Any τ-round minor-aggregation algorithm A can be simulated on G^*_{virt} within $\tilde{O}(\tau\beta D)$ CONGEST rounds of communication on G, such that, the output of each non-virtual $f \in G^*_{virt}$ is known to all vertices on the corresponding face f of G, and the output of each virtual $f_{virt} \in G^*_{virt}$ is known to all $v \in G$. The output of a virtual edge e is known to all vertices of G if both of its endpoints are virtual in G^*_{virt}, otherwise, e's output is only known to vertices of G that lie on the faces that correspond to e's non-virtual endpoint.*

Notice that adding virtual nodes and edges to G^* might violate planarity, but that is not a problem, as these virtual nodes and edges are simulated by (on top of) the same underlying network $\hat{G}$ that is used to simulate the basic model, without making any changes on it. In particular, this extended model, is simulated by the basic model unconditionally, i.e., Lemma B.13 and B.12 constitute a universal algorithm

that runs in the basic model for simulating the extended model, so we simply apply that algorithm on G^* using Theorem B.10.

Dealing with parallel edges in G^*. We now address the problem of G^* being a non-simple graph, i.e. has self-loops and parallel edges. This might be a problem due to the fact that many algorithms assume their input graph is simple. We provide a procedure for deactivating parallel edges of G^*: Consider two nodes u, v of G^* that are connected by multiple edges $e_1, e_2, \ldots, e_m$. We deactivate all these edges except for one (called the *active* edge). If edges are assigned a weight $w(\cdot)$, the active edge gets assigned a new weight $\otimes_i w(e_i)$, where $\otimes$ is a predefined aggregation operator. E.g., $\otimes$ is the summation operator if one is interested in computing a minimum cut, and the minimum operator if one is interested in computing shortest paths.

One cannot straightforwardly use part-wise aggregations on G^* in order to do the above, since a node of G^* might be connected to many nodes, and the required task is an aggregation for each neighbor rather than one aggregation over all incident edges, this may lead to high congestion if all nodes do those aggregations naively and simultaneously. We use the the fact that G^* has a small *arboricity* of $\alpha = 3$ (the arboricity of a graph is the minimal number of forests into which its edges can be partitioned) to obtain a minor-aggregation procedure that requires $\tilde{O}(\alpha)$ minor-aggregate rounds for orienting G^* edges, so that each node has at most a constant out-degree (we count all parallel edges between two nodes as one), which we utilize for load balancing the aggregation tasks. We then apply that procedure on G^* in a black-box manner using Theorem B.14, resulting with an $\tilde{O}(D)$ CONGEST algorithm on G simulating the procedure on G^*. We henceforth assume that G^* is a simple graph. Formally, we obtain the following lemma.

Lemma B.15 (Deactivating parallel edges and self-loops). *There is a minor-aggregation algorithm that runs in $\tilde{O}(\alpha)$-rounds and deactivates parallel edges and self-loops in an input graph of arboricity α, assigning each active edge (u, v) a weight equivalent to a predefined aggregate operator over the weights of all edges with the same endpoints.*

Let α be the arboricity of the input graph, we implement an algorithm of [1] in the minor-aggregation model that runs in $\tilde{O}(\alpha)$ minor-aggregation rounds and produces such an orientation. Since the arboricity of G^* is three (counting all parallel edges between two nodes as one), we get our constant out-degree orientation.

Proof. First, we omit self loops, each edge whose endpoints have the same ID, deactivates itself, this is done in only one minor-aggregate round. Next, we describe [1]'s algorithm and then describe how to implement each step in the minor-aggregation model. At the start of the algorithm, all nodes are *white*. For each node, there is a time where it becomes *black*, the algorithm terminates when all nodes turn black. The algorithm produces a partition of the node's set into $\ell = \lceil 2 \log n \rceil$ parts, $H_1, \ldots, H_\ell$. Upon termination, each node knows the part H_i that contains it. Moreover, the following orientation of edges, results with an orientation with at most $O(\alpha)$ outgoing edges for each nodes (counting parallel edges between two nodes as one). The orientation is as follows. An edge $e = (u, v)$ where $u \in H_i$ and $v \in H_j$ s.t. $i < j$, gets oriented towards v. Otherwise, if both u, v belong to the same part, e is oriented towards the endpoint with a greater ID.

The algorithm for computing such a partition is simple. Let v be a (super-)node, then, for a phase $i \in [\ell]$: If v is white with at most 3α white neighbors, (1) make v black, (2) add v to H_i and (3) notify v's neighbors that v has turned black and has joined H_i. This is implemented this in the minor aggregation model is as follows.

1. We want to know for each node v if it has at most 3α white neighbors. We do that with at most 3α consensus and aggregation steps. That is, a node v computes an aggregate over IDs and la-

bels (black/white) of its neighboring nodes, iteratively counting white nodes that were not already counted by v in the current phase, then the ID of that node that just got counted is broadcast (inside the super node v). Thus, given the previously broadcast IDs one can count a new node that was not already counted and avoid counting nodes that were counted already. If v counts more than 3α values it stops counting.

2. If the super-node v has counted 3α neighbors, via a consensus step, this information (v shall turn black) is broadcasted to all inside v. I.e., edges incident to v learn this information, and v's neighbors learn that v is black and belongs to H_i.

After the algorithm terminates, each edge knows to orient itself as was described earlier. Each node now has at most α outgoing edges (counting parallel edges as one). This procedure terminates in $\tilde{O}(\alpha)$ rounds. Note, the orientation of an edge can be decided locally by the two vertices in the communication graph which simulate that edge.

Deactivate parallel edges. Assume edges are assigned unique $\tilde{O}(1)$-bit IDs in $\tilde{O}(1)$ rounds. Having self-loops deactivated, we now use the orientation to deactivate parallel edges. Each node deactivates its outgoing parallel edges in $O(\alpha)$ rounds. First, enumerate outgoing edges from 1 to 3α, where all edges that connect v to the same neighbor have the same number, this is done in $O(\alpha)$ rounds, in an iterative way similar to that of counting neighbors. Then, for all $i \in [3\alpha]$, for edges with number i (1) An aggregation step over their weights is computed, (2) An edge is chosen to be the active edge by another aggregation step (say the minimum ID one) and other edges deactivate themselves. This process terminates in $O(\alpha)$ rounds. □

Note, each edge in G^* can be identified by its primal endpoints IDs (as the primal graph is simple), we consider this to be the edge's ID and is known locally by the edge's endpoints in the communication graph $\hat{G}$.

B.3 Weighted Girth

In this subsection we show how to compute the *weighted girth* (i.e. the minimum-weight cycle) of an undirected weighted planar graph G. We do this by exploiting the result of the previous subsection for simulating a minor-aggregate weighted minimum-cut algorithm on G^*, which (by Fact 3.1) gives the weighted girth of G.

Recently, a *universally optimal* [15] algorithm for the exact weighted minimum cut problem was discovered by Ghaffari and Zuzic [18], generalizing a breakthrough of Dory, Efron, Mukhopadhyay and Nanongkai [5]. Their algorithm is formulated in the extended minor-aggregation model.

Theorem B.16 (Minor-aggregate exact min-cut, Theorems 12, 18 and 40 of [18]). *Let G be an $\exp(\tilde{O}(1))$ edge-weighted input graph. There is a randomized $\tilde{O}(1)$-round minor aggregation algorithm that computes the weight of the minimum weight cut C of G w.h.p. Upon termination, the value of C is known to all vertices.*

Notice, applying Theorem B.16 above on G^* already allows us to find the weighted girth of our planar network. In [18]'s work, the focus was on finding the weight of the minimum cut, but it can be easily extended to find also the edges of the minimum cut. Let T be a spanning tree of G and $C = (S, V(G) \setminus S)$ be a cut of G, then, C is, respectively, *(one-) two-respecting* of T if and only if there are exactly (one) two edges of T that cross the cut. Given such a tree, where its edges that cross the cut are known, in $O(1)$ minor aggregation rounds, one can mark all edges of the cut.

---

[15] An algorithm that has an optimal round complexity (up to polylog n factor) on each specific network topology, conditioned on an efficient construction of low-congestion shortcuts in that topology. For an extended discussion, see [18, 41, 42].

Lemma B.17 (Mark Cut). *Let G be any network, T a spanning tree of G and C any cut that one- (two-) respects T, defined by edges $e_1, e_2 \in T$ (possibly $e_1 = e_2$). Given G, assuming that edges of T and the edges e_1, e_2 know themselves, there is an $O(1)$ deterministic minor-aggregation algorithm that marks all C edges in G.*

Proof. Consider the tree T, we start by (a contraction step) contracting all T edges except for e_1, e_2, which results in a minor G' of G with at most three super nodes. Next, each super-node shall compute its cost - defined as the number of edges from e_1, e_2 that are incident to it. That is done by an aggregation step, each edge in G' chooses a weight of zero, except for e_1, e_2, each chooses a weight of one, and an aggregation step is performed with the sum operator, resulting with the sum of exiting edges which is the cost of that super node. Now, the ID of the maximum-cost node is computed and broadcast to all the graph - by contracting all the graph into a single super-node s, initiating the input of each $v \in s$ to the ID and cost of $s' \in G'$ s.t. $v \in s'$, then computing the maximum by a consensus step (breaking ties arbitrarily). Finally, we return to consider G' as our graph (we undo the last contraction that we did). Notice, edges of G' know the ID of the maximum-cost node, so, they mark themselves if only if they are incident to it.

The round complexity is clear. The correctness is as follows. G' is a graph that is composed of two or three super-nodes, each super-node corresponds to a connected component of $T \setminus C$, each induces a connected subgraph of G, such that, one of which is incident to all edges e_1, e_2 in G, that subgraph defines one side of the cut (denoted S) and the other (at most two) subgraphs define the other side of the cut. Since S is incident to e_1, e_2 it has the highest cost, thus, all edges exiting S in G get marked, these are exactly the edges of G that cross the cut, hence, all edges of C. $\square$

The algorithm of [18] (Theorem B.16), like many min-cut algorithms (both centralized and distributed, see [5, 11, 12, 14, 23, 24, 32, 43]) consists of three main steps: (1) pack spanning trees of G, (2) compute a minimum 1-respecting cut of each packed tree and (3) compute a minimum 2-respecting cut of each packed tree, the final output is the overall minimum. As said, upon termination the value of C is known to all nodes, in addition, it is implied in [18] that the edges of a spanning tree T of G that C (1-) 2-respects know themselves, in particular, edges in $T \cap C$ know themselves as such. Using Theorem B.14 on G^* in order to apply the exact min-cut algorithm (Theorem B.16) and to apply the procedure for marking cut edges (Lemma B.17), we get an undirected planar weighted girth algorithm:

Theorem 1.7 (Planar Weighted Girth). *There is a randomized distributed algorithm that given an n-vertex undirected weighted planar network G with hop-diameter D, computes the weighted girth and finds the edges of a shortest cycle w.h.p. in $\tilde{O}(D)$ rounds.*

Proof. First we make G^* simple by applying Lemma B.15 for deactivating parallel edges, the (only) active edge left between any two nodes u, v, gets assigned a weight equal to the sum of weights of edges with the same endpoints in G^*. Then, we compute the minimum cut C of the dual network G^* by applying Theorem B.14 to simulate [18]'s minor-aggregate exact min-cut algorithm (Theorem B.16) on G^*. After-which, we can assume that we have the min-cut C value, that C is (1-) 2-respecting to some tree T, and that each edge in T along with T's edges e_1, e_2 that cross the cut know themselves ($e_1 = e_2$ in case C is 1-respecting). Now we use Theorem B.14 for applying Lemma B.17 in order to mark the edges of C in G^*. By the cycle-cut duality (Fact 3.1), the weight of C is the weight of G's minimum cycle and the edges of C constitute a minimum weight cycle in G. $\square$

We clarify a low-level detail regarding marking the edges of C. Recall, the minor-aggregation model simulation on G^* is done via the face-disjoint graph $\hat{G}$. Since the edges of C are edges of G^*, each edge e^* of them maps to exactly one edge $\hat{e}$ of $\hat{G}$ (Property 5), which itself maps to the edge $e \in G$ (the dual of e^*), that is, indeed, marking edges of G^* marks the corresponding edges of G. Notice, the marked edges

$\hat{e} \in \hat{G}$ might not form a cut nor a cycle in $\hat{G}$ but that is fine as it is only a communication tool and not our input graph for the problem.

C   Deferred Details and Proofs from Section 5.1.3

To implement our labeling algorithm, each vertex should know the IDs of all faces (face-parts) that contain it in every bag $X \in \mathcal{T}$ (i.e., the list of nodes $f \in X^*$ it participates in). After (distributively) learning the nodes of X^*, we also learn its edges. That is, for each edge (u, v), its endpoints should know the corresponding edge in the dual bag X^* (if exists). I.e., the IDs of the two faces (face-parts) in X containing the edge, as well as the weight and direction (if any) of the edge in X^*. We next explain how all vertices learn this information by keeping track of the edges of G along the decomposition. As before, it will be convenient to think of an edge as having two *copies* (one for each face containing it). To simplify, in this section we refer to these as edge copies (rather than darts) since there is no need to maintain the dart orientation (every vertex that stores an incident edge $e = (u, v)$ stores both its copies e_1, e_2). We identify e_1 as the *first* copy of e relative to v, and e_2 as the *second* copy of e relative to v (for the other endpoint u, e_2 is its first copy).

Lemma 5.10. *In a single round, each vertex $v \in G$ can learn, for each incident edge copy, a list of the bags $X \in \mathcal{T}$ that contain it.*

Proof. We wish to learn, for each copy of an edge the ID of the single bag in each level of $\mathcal{T}$ that contains it. For the root G of $\mathcal{T}$, all vertices know that all their incident edges belong to G. For a general bag X, we assume each vertex $v \in X$ knows which of its incident copies of edges belong to X, and we show how to learn this for every child bag of X that contains v. If v is in only one child bag X_i of X (v knows it is in X_i), then v's incident edges in X_i are the same as in X. Otherwise, v must be a vertex of S_X (and by the BDD properties, v knows this). For each child bag X_i that contains v, v knows two incident edges a, b, such that, its incident edges in X_i are exactly a, b and all edges $E_v(X_i)$ strictly between a and b in v's clockwise order. Thus, each child bag that contains v is identified with a pair of ordered edges incident to v. Observe that one of those edges might be the virtual edge e_X (if v is an endpoint of e_X). Though e_X is not an actual edge of X, it (and its embedding) is known to its endpoints by the BDD construction. To conclude, for any copy e' of an edge e incident to v, v can learn if that copy is in X_i by checking if $e' = a_2$ (second copy of a), or $e' = b_1$ (first copy of b), or $e' \in E_v(X_i)$. $\square$

We next show how for all bags X, every vertex $v \in X$ learns the IDs of faces (face-parts) that contain it and for each of its edge copies e' in X the ID of the face (face-part) in X containing e'. We begin by showing this for $X = G$ and then for a general bag X.

Lemma 5.11. *There is a $\tilde{O}(D)$-round distributed algorithm that assigns unique IDs to all faces of G. By the end of the algorithm, each vertex $v \in G$ learns a list (of IDs) of faces that contain it. Moreover, v learns for each incident copy e' of an edge e in G the face ID of the face of G that contains e'.*

Proof. We first assign unique $O(\log n)$-bit IDs to the faces of the graph G and let all vertices learn the IDs of all the faces of G that contain them. We can do so in $\tilde{O}(D)$ rounds by using Property 4 of the face-disjoint graph (Section 3). Once a vertex v knows all faces containing it, it remains only to assign those faces to the copies of edges incident to v (which is done locally).

Recall, each edge e of G has two copies e_1, e_2 in the set of edges E_R of $\hat{G}$, and each vertex v of G has $\deg(v)$ many copies v_i in $\hat{G} \setminus V_S$. Moreover, $\hat{G}[E_R]$ is a collection of disjoint cycles, each corresponding to a face of G. Let $e = (u, v) \in E(G)$ be the i'th and j'th edge in u's and v's local embedding (in G), respectively. Then, the copies u_{i+1}, v_j are connected by an edge in E_R identified with the copy e_1 of e,

and the copies u_i, v_{j+1} are connected an edge in E_R identified with the copy e_2 of e (relative to v). It is important that e_1 is the first copy of e relative to v (breaking symmetry between the two copies of an edge), as copies v_i of v are ordered according to v's local embedding, this way, the first copy of e and the second copy of its predecessor edge in the ordering of v participate in the same face. This is the same order we keep in other algorithms (lemmas) for consistency. We assign e_1 (e_2) the component (face) ID that contains them in $\hat{G}[E_R]$. Since u, v in G simulate all their copies in $\hat{G}$, each of them knows for each of e_1, e_2 the connected component ID in $\hat{G}[E_R]$ it participates in. This is the ID of the face that corresponds to that component. Thus, for each of e_1, e_2, both u and v learn the correspondence to faces of G. □

Lemma 5.12 (Distributed Knowledge of Faces). *There is a $\tilde{O}(D)$-round distributed algorithm that assigns unique $\tilde{O}(1)$-bit IDs to all faces and face-parts in all bags $X \in \mathcal{T}$. By the end of the algorithm, each vertex $v \in G$ learns a list (of IDs) of the faces and face-parts it lies on for each bag X that contains it. Moreover, v learns for each incident copy of an edge e' the (ID of) the face or face-part that e' participates in for each bag X that contains e'. The ID of a face (face-part) f of X is contained in the ID of face-parts f' resulting from partitioning f in child bags of X.*

Algorithm 2: Distributed Knowledge of Faces

Input: A BDD $\mathcal{T}$ of a D-diameter network G and a BFS tree T_x for each $X \in \mathcal{T}$.
Result: Each vertex v of G learns for each incident edge copy the list of faces and face-parts that contains it in each bag $X \in \mathcal{T}$ that contains it.

1 Each vertex learns for each of its incident edge copies the list of bags that contains it (Lemma 5.10);
2 Each vertex learns for each of its incident edge copies the list of faces that contain it in bag $X = G$ (Lemma 5.11);
3 **forall** *bags X of level $\ell \in [2, \mathrm{Depth}(\mathcal{T})]$, in parallel,* **do**

                                                           /* Global step */
4   Detect the critical face (face-part) of X if exists and broadcast a message on T notifying all vertices that it shall be partitioned;
5   For each edge that participates in a face-part of X, broadcast the ID of the face-part and the child bag that contains it to the root of T;
6   The root of T broadcasts to all vertices of X the IDs of face(-parts) that shall be partitioned in between child bags of X^*;

                                                            /* Local Step */
7   Each vertex of X learns locally for each incident edge copy the new ID of each face(-part) that contains it in each child bag of X^* that contains it (if any);
8 **end**

Proof. Throughout, we assume that for each $X \in \mathcal{T}$ we already computed a BFS tree T_x of X rooted at an arbitrary vertex of X. The subscript is dropped when X is not ambiguous. This can be done in $\tilde{O}(D)$ rounds using the properties of the BDD. The algorithm described in the proof is given in a high-level pseudocode by Algorithm 2.

First, we apply Lemma 5.10 in order for each vertex to learn for each incident edge copy the bags that contain it in the decomposition. Then, for $X = G$ we apply Lemma 5.11 assigning each copy of an edge a face in G. For a general bag X, we assume we have the required information and show how to maintain it for child bags of X. For a face or face-part f that is contained in X and is entirely contained in one of its child bags X_i, endpoints of edge copies in f inherent the ID of f from X. However, in the case where f

gets partitioned between child bags of X, we need to identify each new face-part and assign it a unique ID.

We work in two steps. First, a global broadcast over X is performed in order to detect the faces and face-parts of X that shall be partitioned. Then, a local step of assigning IDs to the new resulting face-parts in the child bags of X.

1. Global step. In this step we detect the face of G and all face-parts f in X to be partitioned (if any). We start with the unique (by Lemma 5.3) critical face (or face-part) of G in X. The endpoints of e_x lie on f, are on S_x and know if e_x is virtual or not (if it is, then f gets partitioned). Therefore, f can be detected by the endpoints u, v of e_x by the BDD construction (i.e., using the algorithm of [17] that [26] use for computing a separator). More concretely, the edge e_x is embedded between two consecutive incident edges a, b in the clockwise embedding of v. Copies a_2, b_1 (in that order) of those edges define the face f s.t. a, e_x, b appear in that order in the embedding of v (analogously for u). Additionally, an arbitrary endpoint of e_x broadcasts the ID of the critical face (face-part) on T saying it shall be partitioned as well (if it does).

After identifying the critical face, we move on to identifying the other face-parts that shall be partitioned. We exploit the small (logarithmic) number of face-parts in X (Lemma 5.3) for performing the following broadcast. Each vertex v on a face-part f of X knows (its incident edge copies in) f. Then, v upcasts $ID(f), ID(X_i)$ to the root of the BFS tree T of X, where X_i is a child bag that contains edge copies of f incident to v (the correspondence between copies of edges and child bags is known from Lemma 5.10). We need to be careful in this upcast not to suffer unnecessary congestion, as a face-part might contain many vertices and more than one connected component. So, if some vertex of X receives more than one child bag ID for the same face-part f, it stops passing messages of f, and upcasts only one special message saying that f shall be partitioned. The upcast is performed in a pipelined method. Finally, after the root stops receiving messages, it broadcasts the IDs of all face-parts that shall be partitioned.

2. Local step. For each face or a face-part f that is not partitioned, the vertices lying on f consider the ID of f to be the same as from the parent level and remember that it also exists as a whole in the child bag it is contained in. For a face or a face-part f that is partitioned, the vertices v of f know for each incident copy e' of an edge that belongs to f in which child bag X_i is it contained (Lemma 5.10). Thus, v locally learns $(ID(f), ID(X_i))$ as the ID of the face-part that contains e' in X_i. We get that all vertices on a part of f in some child bag of X know the same ID for that part.

We claim that this terminates within $\tilde{O}(D)$ rounds. First, the IDs of face-parts are of size $\tilde{O}(1)$ since (1) by Lemma 5.11 the faces in G are assigned IDs of size $O(\log n)$, (2) by the construction of the BDD in [26], the ID of a child bag is of size $\tilde{O}(1)$ (and there are $O(\log n)$ levels of T), and (3) a face-part in a level ℓ is assigned the ID of a face (or face-part) of level $\ell - 1$ after appending $O(\log n)$ additional bits. Second, by Lemma 5.3 the number of face-parts in X is at most $O(\log n)$ and there is at most one critical face (or face-part). Meaning, any vertex shall pass at most $O(\log n)$ messages of size $\tilde{O}(1)$ throughout the course of the procedure (one message for the critical face and one per face-part of X). Finally, the BFS tree used for broadcast has a diameter of $O(D \log n)$ (Property 5 of the BDD), and all bags X can run the procedure in parallel with a constant factor overhead (Property 7 of the BDD). □

Lemma 5.14. *Let $e = (u, v)$ be an edge in a bag X. In one round after applying Lemma 5.12, the endpoints of e learn the corresponding dual edge e^* in X^* (if exists).*

Proof. Recall that by the definition of the dual bag, there is a dual edge e^* between two faces or face-parts f_1, f_2 of X (on both sides of e) iff the two copies of e are contained in X (i.e., none of them is on a hole). By invoking Lemma 5.12 and Lemma 5.10 v can know which of its incident edges e has both copies in X, and the ID of the face (face-part) each copy participates in. The endpoints of such edges e can compute the correspondence locally as follows. Consider such an edge e incident to a vertex $v \in X$ and let f_1, f_2 be the

faces containing the first and second copies of e (relative to v), respectively. Then, the corresponding dual edge is directed from f_1 to f_2 when e's direction in G is outwards v. Notice, this definition is consistent (independent from v) and relies only on the direction of e and on faces it participate in. I.e., the first copy of e relative to v is the second copy of e relative to u (and vice versa). If G is undirected, then dual edges are undirected as well. If G is weighted, then, e^* has the same weight as e. $\qquad\square$

D   Deferred Details and Proofs from Section 6

D.1   Deferred Details and Proofs from Section 6.1

Theorem 1.2 (Exact Maximum st-Flow in Directed Planar Graphs). *There is a randomized distributed algorithm that given an n-vertex directed planar network G with hop-diameter D and integral edge-capacities, and two vertices s, t, computes the maximum st-flow value and assignment w.h.p. in $\tilde{O}(D^2)$ rounds.*

Proof. The algorithm follows the centralized algorithm of Miller and Naor [30]. It works by performing a binary search on the value λ of the maximum st-flow. In each iteration, λ units of flow are pushed along a (not necessarily shortest) s-to-t path of darts P. If this does not violate the capacity of any dart in P then it is a feasible flow and we conclude that the maximum st-flow is at least λ. Otherwise, a residual graph is defined by subtracting λ from the capacity of every dart in P and adding λ to their reverse dart (in the beginning every edge e of capacity $c(e)$ corresponds to two darts - one in the direction of e with capacity $c(e)$, and its reverse with capacity zero). Notice that residual capacities may be negative for the darts of P. To fix this, an arbitrary face f of G is chosen and SSSP (with positive and negative edge lengths) in G^* is computed from f. Miller and Naor [30] showed that if a negative cycle is detected then we can conclude that the maximum st-flow in G is smaller than λ, and otherwise it is at least λ. Moreover, in the latter case, the following is a feasible st-flow assignment of value λ. For every dart $d \in G$ that maps to $d^* = (g, h) \in G^*$: the flow on d is set to $\text{dist}(f, h) - \text{dist}(f, g) + \lambda$ if $d \in P$, to $\text{dist}(f, h) - \text{dist}(f, g) - \lambda$ if $\text{rev}(d) \in P$, and to $\text{dist}(f, h) - \text{dist}(f, g)$ otherwise.

The path P can be found in $O(D)$ rounds (notice that P is a directed path of darts but does not need to be a directed path of edges in G), and the dual SSSP computation is done in $\tilde{O}(D^2)$ rounds as we showed in Section 5 (with a slight modification that we show next). Since λ is assumed to be polynomial, all iterations of the binary search are done in $\tilde{O}(D^2)$ rounds. The flow assignment on the edges of G is given by the assignment to its darts, that can be deduced locally from its endpoint vertices by Property 14 of the BDD (since for a dart $d = (u, v) \in G$ with $d^* = (f, g) \in G^*$ the endpoints u, v belong to both faces f and g in G).

Notice however, the SSSP algorithm on G^* from Section 5 uses darts for gaining the necessary distributed knowledge for it to work, however, its output is an SSSP tree of edges and not darts. Thus, we need to show that it can output an SSSP of darts, i.e., to consider the two darts of an edge (with weights as described earlier) when computing the labeling and the shortest paths tree. To do that, we slightly modify the input network and use the algorithm as a black-box on it. We work with the network G' obtained from G by adding the reversal of each edge to the graph, such that, the two copies of an edge are ordered one after another in the embedding. The reversal of an edge gets assigned a capacity of zero. It is not hard to see that G' can be constructed and embedded in the plane (assuming G is embedded) without any need for communication, and that each CONGEST round on G' is simulated within two rounds on G.

The algorithm runs as follows, G' is constructed, an s-to-t path P in G' is found in $O(D)$ rounds via a breadth first search starting at s, then, $\lambda = 1$ (at first) units of flow are pushed on P, capacities of P edges and their reversals are (locally) changed as explained earlier, then the algorithm for an SSSP tree of G'^* from Lemma 2.2 is activated with the source f being the minimum ID face of G' (f can be known in $O(D)$ rounds by aggregating on G' the IDs of its faces, which are known by Property 13 of the BDD),

thus, each node v in G' knows the distance from f to each of the (dual nodes mapped to) faces it lies on, thus, v can learn the flow value for each of its incident edges. Notice, in each iteration, each vertex in G' knows locally the current value of λ, depending on whether or not it has received a message reporting a dual negative cycle from the Dual SSSP algorithm. Finally, indeed by the end of the algorithm, each vertex v of G knows the st-flow value in G and (locally) a corresponding flow assignment to its incident edges (the assignment induced by G'). □

D.2 Deferred Details and Proofs from Section 6.2

Theorem 6.1 (Distributed Exact Planar Directed Minimum st-Cut). *Given a directed planar graph G with non-negative edge weights and two vertices s, t, there is a $\tilde{O}(D^2)$ round randomized algorithm that finds the minimum st-cut w.h.p.*

Proof. At first, we use Theorem 1.2 to compute a maximum st-flow f in G in $\tilde{O}(D^2)$ rounds. To mark the minimum st-cut edges we work in the *residual graph* of G with respect to f. The residual graph R w.r.t. a flow assignment f of a network G, is the network where each edge has a weight equivalent to its capacity minus the flow sent on it by f. By [7], a minimum st-cut is a cut consisting of the s-most saturated edges, i.e, the saturated edges which have no other saturated edge that precedes them in any s-to-t path that contains them. In R, those are zero weighted edges. We denote this cut by $\overrightarrow{\delta}(S, V \setminus S)$, where S (respectively, $V \setminus S$) is the side of the cut containing s (t). To find those edges, we compute an SSSP tree from s in R', where we modify the edge weights of R' as follows: A saturated edge gets assigned a weight of one and all other edges, a weight of zero. Note, finding the saturated edges in G is equivalent to finding the s-most edges whose weight is zero in R, which is equivalent to finding the s-most edges whose weight is one in R', thus, we work in R'. After computing an SSSP tree from s in R', we decide that an edge (u, v) is in the cut, iff its weight is one in R', dist$(s, u) = 0$ and dist$(s, v) = 1$. Those are exactly the saturated s-most edges, for the following reasons: (1) (u, v) is saturated since its weight is one in R', (2) it is the s-most saturated edges on any path starting at s that contains it, for otherwise dist$(s, u) > 0$. Notice, the same method allows to compute the vertices of S (equivalently, $V \setminus S$). Namely, a vertex u is in S, if and only if u has dist$(s, u) = 0$. For otherwise, dist$(s, u) > 0$ which means it is further from a saturated edge that lies on some path starting at s, meaning it is on the other side of the cut (otherwise, it would contradict that the cut computed is composed of s-most saturated edges).

The endpoints of an edge know if it is saturated or not, since they know its weight and flow value, thus, R' can be constructed in one round. To compute a directed SSSP tree from s, we borrow an exact SSSP algorithm of Li and Parter's [26] for planar graphs, which runs in $\tilde{O}(D^2)$ rounds (Theorem 1.6, Lemma 5.3 in their paper). Although it is stated for the undirected case in their paper, it can be easily adapted for the weighted case as noted by Parter in [35]. □