Distributed Algorithms for Planar Networks II: Low-Congestion Shortcuts, MST, and Min-Cut*

Mohsen Ghaffari
MIT
ghaffari@mit.edu

Bernhard Haeupler
CMU
haeupler@cs.cmu.edu

Abstract

This paper introduces the concept of *low-congestion shortcuts* for (near-)planar networks, and demonstrates their power by using them to obtain near-optimal distributed algorithms for problems such as Minimum Spanning Tree (MST) or Minimum Cut, in planar networks.

Consider a graph $G = (V, E)$ and a partitioning of V into subsets of nodes $S_1, \ldots, S_N$, each inducing a connected subgraph $G[S_i]$. We define an *α-congestion shortcut with dilation β* to be a set of subgraphs $H_1, \ldots, H_N \subseteq G$, one for each subset S_i, such that

1. For each $i \in [1, N]$, the diameter of the subgraph $G[S_i] + H_i$ is at most β.
2. For each edge $e \in E$, the number of subgraphs $G[S_i] + H_i$ containing e is at most α.

We prove that any partition of a D-diameter planar graph into individually-connected parts admits an $O(D \log D)$-congestion shortcut with dilation $O(D \log D)$, and we also present a distributed construction of it in $\tilde{O}(D)$ rounds. We moreover prove these parameters to be near-optimal; i.e., there are instances in which, unavoidably, $\max\{\alpha, \beta\} = \Omega(D \frac{\log D}{\log \log D})$.

Finally, we use low-congestion shortcuts, and their efficient distributed construction, to derive $\tilde{O}(D)$-round distributed algorithms for MST and Min-Cut, in planar networks. This complexity nearly matches the trivial lower bound of $\Omega(D)$. We remark that this is the first result bypassing the well-known $\tilde{\Omega}(D + \sqrt{n})$ existential lower bound of general graphs (see Peleg and Rubinovich [FOCS'99]; Elkin [STOC'04]; and Das Sarma et al. [STOC'11]) in a family of graphs of interest.

1 Introduction

This paper introduces *low-congestion shortcuts* and exhibits their utility by using them to derive near-optimal distributed algorithms for a number of fundamental network optimization problems.

Throughout, we use the standard distributed message passing model called CONGEST [Pel00]: the network is abstracted as a graph $G = (V, E)$, with n nodes and diameter D; communications occur in synchronous rounds, and per round, $O(\log n)$ bits can be sent along each edge.

1.1 The Motivation for, and Definition of, Low-Congestion Shortcuts

Consider the following scenario, which is a recurring theme throughout distributed approaches for many network optimization problems:

> *The graph is partitioned into a number of disjoint individually-connected parts, and we need to compute a (typically simple) function for each part, e.g., the minimum of the values held by the nodes in the part.*

See Figure 1 for a pictorial illustration of this partition. This scenario typically appears when the algorithmic approach works via, often iterations of, merging solutions of smaller subproblems. This includes various methods (loosely) based on *divide and conquer*. There are many examples, e.g., [GHS83, GKP93, KP95, Elk04a, DSHK+11, GK13, NS14, Gha14, GKK+15], but perhaps the most prominent is the 1926 algorithm of Boruvka[1] [NMN01] for Minimum Spanning Tree: Starting with the trivial partition of each node being its own part, in each iteration, each part computes the *minimum-weight outgoing edge*, and adds it to the current partial solution, making the parts incident on this edge merge. After $O(\log n)$ iterations, we arrive at the MST.

Typically, since the number of parts can be large, we need to solve the problems of all parts in parallel. Most naturally, this would be by solving each part's problem using communication only inside the part. However, this would take a long time as the diameter of these

*This research was supported in part by the NSF award *Distributed Algorithms for Near Planar Networks* (CCF-1527110).

[1] We note that, within the distributed literature, this method is more often called the *GHS approach*, after Gallagher, Humblet and Spira who rediscovered it in 1983 [GHS83], and extended it to asynchronous settings.

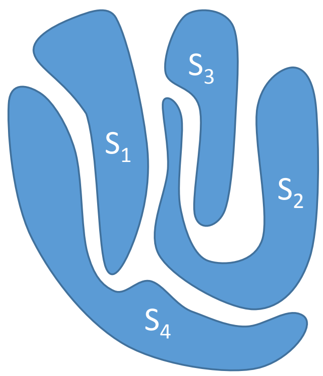


Figure 1: An illustration of the recurring scenario.

parts can be large, much larger than the diameter D of the graph. It is often hard to enforce small part-wise diameters as the structure of the parts (i.e., subproblems) is usually dictated by the problem itself; see, e.g., MST.

To overcome this generic issue, we introduce the notion of *low-congestion shortcuts*: Intuitively, we want to augment each part with some extra edges, taken from the network G, so that we effectively reduce the diameter of the part. These shortcutting edges are to be used for communication purposes. To be able to use these shortcuts without creating communication bottlenecks, we need to ensure that each edge is not used in too many shortcuts. We next formalize this intuition:

DEFINITION 1. *Given a graph $G = (V, E)$ and a partition of V into disjoint subsets $S_1, \ldots, S_N \subset V$, each inducing a connected subgraph $G[S_i]$, we define an α-congestion shortcut with dilation β to be a set of subgraphs $H_1, \ldots, H_N \subset G$, one for each set S_i, such that:*

(1) For each i, the diameter of the subgraph $G[S_i] + H_i$ is at most β.

(2) For each edge $e \in E$, the number of subgraphs $G[S_i] + H_i$ containing e is at most α.

As we will see later, given such a low-congestion shortcut, one can solve the common scenario stated above in $\tilde{O}(\max\{\alpha, \beta\})$ rounds, using standard *random delay* techniques for pipelining messages.

1.2 Technical Results In general graphs, there are instances in which, unavoidably, $\max\{\mathsf{congestion}, \mathsf{dilation}\} = \Omega(D + \sqrt{n})$; see Section 1.3. Our key technical contribution is to show the following result for planar graphs:

THEOREM 2. *For any planar graph $G = (V, E)$ and any partition of V into disjoint individually-connected parts $S_1, \ldots, S_N$, there exists an $O(D \log D)$-congestion $O(D \log D)$-dilation shortcut, where D denotes the diameter of G. Moreover, there is a distributed algorithm that computes such a shortcut in $\tilde{O}(D)$ rounds.*

We note that the above theorem (partially) extends to near-planar graphs, to *bounded-genus* graphs particularly. To be precise, for graphs that can be drawn on a genus g surface with no crossing, we get $O((g+1)D \log D)$-congestion $O(D \log D)$-dilation shortcuts. However, currently, we do not have the distributed construction of this extension, solely because we do not have the respective efficient distributed embedding algorithm.

As we will see, when using shortcuts, the complexity depends on $\max\{\mathsf{congestion}, \mathsf{dilation}\}$, and this is the key quality measure. In this regard, we show that Theorem 2 is nearly optimal:

LEMMA 3. *There is a planar graph $G = (V, E)$ with diameter D and a partition of V into disjoint individually-connected parts $S_1, \ldots, S_N$ such that, regardless of the choice of the shortcuts, we will have* $\max\{\mathsf{congestion}, \mathsf{dilation}\} = \Omega(\frac{D \log D}{\log \log D})$.

Finally, we use low-congestion shortcuts to prove the following two end-results:

THEOREM 4. *For any weighted planar graph $G = (V, E)$, there is an $\tilde{O}(D)$ round distributed MST algorithm.*

THEOREM 5. *For any weighted planar graph $G = (V, E)$, there is an $\tilde{O}(D)$ round distributed algorithm that computes a $(1 + \varepsilon)$ approximation of the minimum cut. Here, ε is an arbitrarily small positive constant*[2].

Note that, both of the above algorithms have a near-optimal complexity as $\Omega(D)$ is a trivial and folklore lower bound for each of the problems.

1.3 The Motivation for (Near-)Planar Networks, and Related Work Distributed algorithms for network optimization problems have a long and rich history. A first-order summary of the state of the art is that, for many of the basic problems, including MST and $(1+\varepsilon)$-approximations of Min-Cut, Max-Flow, and Shortest-Paths, the best-known upper bound is $\tilde{O}(D +$

[2]In reality, we do not need ε to be a constant; in general, the bound has a $O(\mathrm{poly}(1/\varepsilon))$ dependency on ε. For simplicity, in our notations, we imagine ε being a constant, thus allowing us to absorb $\mathrm{poly}(1/\varepsilon)$ into the $O()$ notation. Furthermore, note that, e.g., in unweighted planar graphs, the min-cut size is an integer and at most 6 and thus, by choosing $\varepsilon \leq 1/7$, we get an exact min-cut algorithm.

$\sqrt{n}$), or close to it. See e.g. [KP95, Elk04a, LPS13, GK13, Nan14, LPS14, CHGK14, NS14, GL14, Gha14, GKK$^+$15]. Furthermore, this round complexity is essentially the best-possible in general graphs, i.e., there are graphs in which one cannot do better [PR99, Elk04b, DSHK$^+$11].

In hindsight, one simplistic way of explaining the $\tilde{O}(D + \sqrt{n})$ bound is via shortcuts. In any graph, one can always achieve low-congestion shortcuts with $\max\{\mathsf{congestion}, \mathsf{dilation}\} = \Theta(D + \sqrt{n})$. However unfortunately, this is the best-possible for general graphs, as can be easily inferred from the lower bounds of [PR99, Elk04b, DSHK$^+$11]. To obtain $\max\{\mathsf{congestion}, \mathsf{dilation}\} = O(D + \sqrt{n})$, simply set the shortcut $H_i = G$ for each part S_i where $|S_i| \geq \sqrt{n}$, and let it be empty for any other part. This ensures that each shortcutted part has diameter at most $\max\{D, \sqrt{n}\} = O(D + \sqrt{n})$ and each edge is used in at most $n/\sqrt{n} + 1 = O(\sqrt{n})$ shortcutted parts. Of course, this is not all that it takes to get to the algorithms listed above, but it demonstrates the key tradeoff in quite a few of them, e.g., [KP95, GK13, NS14, Gha14, GKK$^+$15]. In fact, this simple idea can be turned into a very short and clean $O((D + \sqrt{n}) \log n)$ round MST algorithm for general graphs.

We now go back to the discussion about the stated far-reaching $\tilde{\Omega}(D + \sqrt{n})$-round lower bound of [PR99, Elk04b, DSHK$^+$11] for general graphs. We believe that the fact that there is a network in which one cannot obtain better than $\tilde{\Omega}(D+\sqrt{n})$ algorithms is not a strong enough justification for being satisfied with this complexity in *all networks* and not looking further. This point becomes more crucial considering that this barrier is not only for one problem, but rather for most of the fundamental network optimization tasks. Hence, we think that an important step forward is to examine what can be done when these lower bound graphs are ruled out, i.e., in special graph families. We focus on (near-)planar networks, mainly because of two reasons:

(A) Planar graphs have always been a primary graph family of interest. This is due to their frequent appearance in practice and because of the richness of the theoretical/algorithmic aspects found in them. This has been true since 1735 when Euler introduced graphs as a mathematical representation to discuss (Eulerian) paths in the plane, and continues to this day, where each year about a dozen papers present new, typically algorithmic, results on planar or near-planar graphs. Thanks to this richness and theoretical depth, (near-)planar graphs even have (algorithmic) courses [KM, Kle, DMST] and textbooks [KMft, NC88] of their own.

(B) The graph of [PR99, Elk04b, DSHK$^+$11], which exhibits the powerful $\tilde{\Omega}(D+\sqrt{n})$ round lower bound, is extremely simple and is contained in most basic graph families. The graph is made of a full binary tree with $\sqrt{n}$ leaves, and $\Theta(\sqrt{n})$ paths of length $\sqrt{n}$, where the i^{th} leaf is connected to the i^{th} nodes of all the paths. See Figure 2. This graph has a very low sparsity, in fact an arboricity of 2. One can even make the max degree 3, by substituting each leaf's star-connections with a binary tree, while keeping the lower bound at $\tilde{\Omega}(D+\sqrt{n})$. One can see that this graph, or close variants of it, fits within many other basic graph families as well. The main (non-trivial[3]) exception we found that rules out this graph is the (near-)planar family, and our results show that indeed one can bypass the lower bound in such networks.

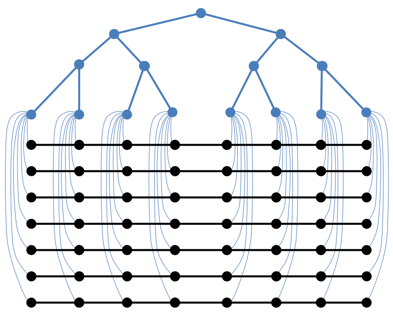


Figure 2: The graph of the $\tilde{\Omega}(D + \sqrt{n})$ lower bound.

2 Preliminaries

Basic Definitions: Consider a Breadth First Search (BFS) tree T of the graph G rooted in an arbitrary node r. Also consider a planar embedding of G, or more generally, an embedding of G on a genus-g surface, where g is the genus of graph G. This embedding determines a clockwise order of the edges incident on each node v around it, and particularly, it dictates a clockwise order for the BFS-edges of each node v around v. With respect to this, we define a *left-order* and a *right-order* among the nodes, as follows: We use a left-first (or *right-first*) Depth First Search traversal in the BFS-tree, labeling each node with a unique *left-ID* (resp., *right-ID*) from $\{1, 2, \ldots, n\}$. For two nodes u, v, we say u is to the left (or right) of v if u's left-ID (resp., right-ID) is smaller than that of v. It is easy to see that u is an ancestor (or descendant) of v if and only if both the left-ID and the right-ID of u are smaller (resp., greater) than those of v. We say a node v is *strictly-right* (or *strictly-left*) of a node u if it is to the right (resp., left) of it but not an ancestor or descendant of it. For each part S_i, we define the *leftmost* node (and the *rightmost* node) of S_i to be the node in S_i that has

[3]The lower bounds do not hold in, e.g., trees, but in trees the problems under consideration are trivial anyways.

the smallest left-ID (resp., the smallest right-ID). The *lowest common ancestor* (LCA) of a part S_i is the lowest common ancestor of its leftmost and rightmost nodes on the BFS-tree T. Moreover, we define the *boundary paths* of S_i to be the two paths connecting the BFS-root r to the leftmost and rightmost nodes of S_i.

3 Existence of Shortcuts

In this section, we show that the shortcuts as claimed in Theorem 2 exist. In the next section, we explain how to turn this existence proof into an efficient distributed construction.

We first present a weaker version of our result that proves the existence of shortcuts with $O(D)$-congestion and $O(D^2)$-dilation. Later, by adding some more edges to these shortcuts, we strengthen this to prove the existence of shortcuts with $O(D \log D)$-congestion and $O(D \log D)$-dilation. At the end of this section, we show in Theorem 12 that these parameters are essentially optimal.

3.1 Take 1: Shortcuts with Congestion $O(D)$ and Dilation $O(D^2)$: Consider the given partition of V to disjoint individually-connected parts $S_1, \ldots, S_N$. We first define our choice of the shortcut subgraphs H_i, i.e., we explain what is the set of the shortcut edges that will be given to each part S_i. Then, we prove that this choice of subgraphs H_i is indeed a shortcut with congestion $O(D)$ and dilation $O(D^2)$.

Designating Shortcuts, Version 1: For each subgraph $G[S_i]$, define the designated shortcut subgraph H_i to be the subset of the BFS-tree T edges e that satisfy one of the following two conditions: (1) e has a BFS-ancestor in S_i, (2) e has a BFS-descendant in S_i but e is not on the *boundary paths* of S_i. We call the latter kind of BFS-edges *encapsulated* by S_i.

Figure 3 shows some examples of a BFS-edge e encapsulated by a part S_i. Regarding the conditions stated in the above rule, notice that we have particularly ruled out using the *boundary path edges*. It is easy to see that if one adds (all) the boundary path edges to the shortcut, then the congestion can grow in an uncontrolled way, even up to N, the total number of parts. In the next subsection, we take a closer look at this issue and see how by adding a carefully chosen subset of these boundary-path edges, we can reduce the diameter without increasing the congestion too much.

Analysis: We now prove that these shortcuts have congestion $O(D)$ and dilation $O(D^2)$. We in fact present a stronger result, showing that if G is embeddable in a genus-g surface, then congestion is at most $O((g+1)D)$ and dilation is $O(D^2)$. Note that a planar graph is simply a graph embeddable on a genus $g = 0$ surface, e.g., a plane or a sphere.

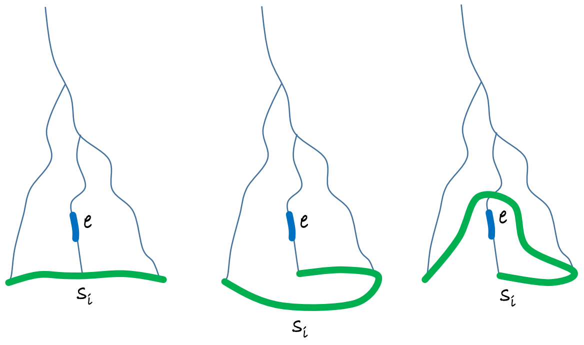


Figure 3: Examples of encapsulation. The BFS- tree is depicted with thin blue lines and the thicker green line indicates the part S_i. In the figure on the right side, e satisfies both of the conditions; it has an ancestor in S_i and it is also encapsulated in S_i.

We note that a somewhat different proof of this fact for the $g = 0$ case, as well as the related choice of the shortcuts, was found in collaboration with Shay Mozes.

LEMMA 6. *Assuming G has genus g, each BFS-edge e is in at most $\alpha = O((g+1)D)$ of the designated shortcut sub-graphs H_i.*

Proof. Note that a shortcut H_i includes e either because S_i includes a BFS-ancestor of e, or because S_i encapsulates e. Since S_i-sets are disjoint and the BFS tree has depth at most D, the number of parts S_i that include an ancestor of e is at most D. Let k be the number of parts S_i that encapsulate e and do not include any ancestor of e. To complete the proof, we show that $k \le O((g+1)D)$.

Let us focus on parts S_i that encapsulate e but do not contain an ancestor of e. For each of these parts S_i, define the *three e-encapsulations* paths of S_i as follows: Considering Figure 3 might be helpful here. The path going from the root through e to its descendant node $c \in S_i$ is the *central path* of this encapsulation, while the paths going from the root to the leftmost and rightmost nodes of S_i are called respectively the *left and right paths* of the encapsulation.

Imagine a virtual graph $\mathcal{G}_e$ with one node representing each part S_i—each part that encapsulates e but does not contain an ancestor of it—and put a directed edge from the node representing S_i to the node representing $S_{i'}$ if set S_i intersects at least one of the three e-encapsulation paths of $S_{i'}$. Note that for each part S_i, its three e-encapsulation paths in total include at most $3D$ nodes. Hence, as the parts are node-disjoint, the in-degree of each node in $\mathcal{G}_e$ is at most $3D$. This means, in the undirected version $\overline{\mathcal{G}_e}$ of $\mathcal{G}_e$, the average degree is at most $6D$. Turan's theorem (see e.g., [AS04, pages 95-96] or [BM08, exercise 13.2.10]) shows that an η-node undirected graph with average-degree x has an independent set of size at least $\frac{\eta}{x+1}$. Thus, if $k > (6D+1)(4g+1)$, we would get that $\overline{\mathcal{G}_e}$ has an independent set of size at least $4g+2$. In the next claim, essentially by exhibiting a $K_{3,4g+3}$ minor—a minor that is a complete bipartite graph with the size of the two sides being 3 and $4g+3$—we show that this would be in contradiction with graph G being embeddable on a genus g surface, thus completing the proof.

Claim 7. *For each BFS-edge e, there cannot be $4g+2$ parts S_i, each encapsulating e but not having an ancestor of it, such that none of them intersects the e-encapsulation paths of any other.*

For the case of planar graphs ($g = 0$), one can give a simple intuitive sketch, which is almost a proof: just pictorially consider the encapsulation possibilities for one part S_i, see the examples given in Figure 3. If we have another part S_j that encapsulates e but does not intersect the encapsulation paths of S_i, then by Jordan curve theorem, S_j will have to be in one of the faces defined by the three paths of S_i and part S_i itself. One can see that in each of the cases, at least one of the encapsulation paths of S_j would cross/intersect S_i.

Proof. [Proof of Claim 7] Consider a BFS edge $e = (v, u)$, where v is the BFS-parent of u. For the sake of contradiction, suppose that there are $4g+2$ parts, each encapsulating e but not having an ancestor of it, such that none of them intersects the e-encapsulation paths of any other. We show that, since we have assumed G to have genus g, this implies $K_{3,4g+3}$—that is, the complete bipartite graph with 3 nodes on one side and $4g+3$ on the other—can be embedded on a surface with genus g. This would be in contradiction with known results (see e.g., [Rin65a, Rin65b]) that show that $K_{a,b}$ has genus exactly $\lceil\frac{(a-2)(b-2)}{4}\rceil$.

First, contract each part S_i into a single node. Note that since each S_i induces a connected subgraph, this operation does not increase the genus. Moreover, since none of these parts intersects the e-encapsulation paths of any other, during these contractions, the e-encapsulation paths do not get contracted (except for the endpoints in each part S_i being united). Now contract the BFS-path connecting v to the BFS-root r, and let us call the resulting node v'. Note that again, this contraction does not increase the genus as we contracted a connected induced subgraph. The middle drawing in Figure 4 shows the result of these contractions, for one part S_i.

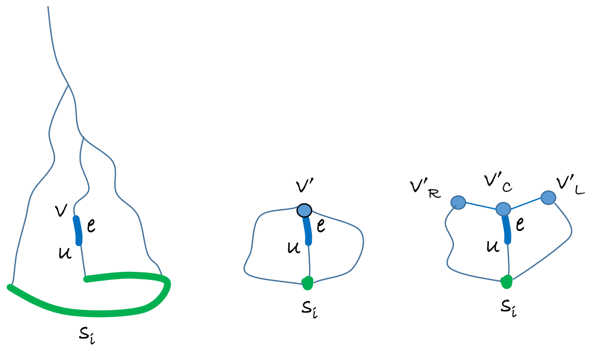


Figure 4: The result of the contractions, and then the local splitting, depicted for one part.

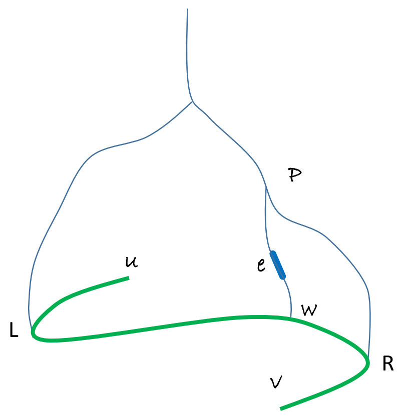
For each part S_i, there are three paths that connect node v' to set S_i and we have the restriction that, in the already fixed clockwise ordering of the embedding, one path should be to the left of the edge e (now going from v' to u), and one path should be to the right of it, while the third goes through e. Because of this restriction, we can locally split node v' into three copies, v'_L, v'_C, and v'_R, respectively for left, central, and right copies, with connections as follows: we have connected each of v'_L and v'_R to v'_C, and moreover, the left encapsulation paths are attached to v'_L, the right encapsulation paths are attached to v'_R, and the central encapsulation paths go through e which is attached to v'_C. The right drawing in Figure 4 shows an example. Note that this operation does not increase the genus because for any previous embedding on a genus-g surface (which has the left and right encapsulation paths respectively on the left and right of e), after this local splitting process, the old embedding with this split is an embedding on the same surface, with no crossing.

Now, consider the complete bipartite minor defined by side A being the set of contracted nodes representing parts S_i and the node v'_C, and side B being three nodes of V'_L, V'_R and u. Each of the A-side nodes is connected to all three of the B-side nodes. Hence, this is a $K_{3,4g+3}$ minor of a graph embedded on a g-genus surface. This is a contradiction as the genus of $K_{3,4g+3}$ is $\lceil\frac{(3-2)(4g+3-2)}{4}\rceil = \lceil\frac{4g+1}{4}\rceil = g+1$.

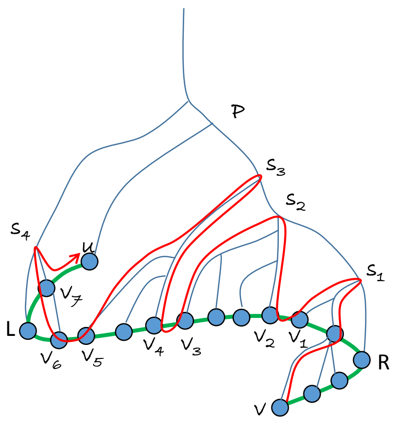
Lemma 8. *For each i, the diameter of the sub-graph $G[S_i] + H_i$ is at most $\beta = O(D^2)$.*

Proof. Consider two arbitrary nodes $v, u \in S_i$. We show that there is a path in $G[S_i] + H_i$ connecting v to u that has length at most at most $\beta = O(D^2)$.

Consider the union $\mathcal{P}$ of boundary paths of S_i and partition S_i into *classes* such that the nodes in each



(a)



(b)

Figure 5: (a) An edge e on the tree-path Q_w connecting w to the tree-path $\mathcal{P}$. Edge e must be encapsulated by S_i, which means its is added to H_i. (b) A shortcutted walk going from v to u is shown in red, which has length at most $O(D^2)$ hops and uses only edges of $G[S_i] + H_i$.

class have the same lowest BFS-ancestor on $\mathcal{P}$. Since $\mathcal{P}$ has at most $2D$ nodes, we have at most $2D$ classes.

We first argue that for each node $w \in S_i$, each BFS-edge e on BFS-path $\mathcal{Q}_w$ that connects w to its lowest tree-ancestor on $\mathcal{P}$ is added to H_i. See Figure 5(a). This is true because edge e is a BFS-edge that is not on the boundary paths of S_i and it has a descendant $w \in S_i$. Hence, e is encapsulated by S_i, which means e is included in H_i. This implies that each two nodes in the same class have distance at most $2D$ in $G[S_i] + H_i$. Particularly, we can go from one to the other by traversing up from one of them through the H_i edges to their common lowest boundary ancestor and then going down again via H_i edges to the other, in at most $2D$ hops.

Now imagine contracting all the S_i-nodes of each class into a single *big-node*, along with the tree-paths connecting them to their common boundary ancestor. Since S_i is connected, in this contracted graph, there is a path traversing $G[S_i]$-edges which goes from the big-node containing v to the big-node containing u. This path contains at most $2D$ big-nodes, simply because the total number of the big-nodes is at most $2D$. Now, we can take this path back to $G[S_i] + H_i$ by expanding the contractions, where crossing each big-node costs us at most $2D$ hops. Overall this is $O(D^2)$ hops. Figure 5(b) shows an example of such a shortcutted path between $v, u \in S_i$. Hence, in $G[S_i] + H_i$, there is a path of length at most $\beta = O(D^2)$ between any two nodes $v, u \in S_i$.

3.2 Take 2: Shortcuts with Congestion $O(D \log D)$ and Dilation $O(D \log D)$: Recall that in the previous section, we did not add the edges on the BFS boundary paths to the shortcut graphs. This was particularly because, if one adds all of these, then the congestion can grow in an uncontrolled way, even growing up to N, the total number of parts. In this section, we explain how by adding a carefully chosen subset of these boundary path edges to the shortcut subgraphs H_i, we can reduce the dilation to $O(D \log D)$, while sacrificing only a little bit in the congestion, i.e., increasing it to $O(D \log D)$. In the next subsection, we show that this trade-off is in fact essentially optimal; that is, there are instances in which, regardless of the choice of the shortcuts, achieving a dilation of $O(D \log D)$ requires congestion of $\Omega(D \frac{\log D}{\log \log D})$.

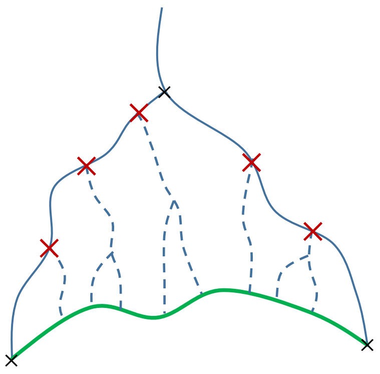
For each part S_i, consider the BFS-path connecting its leftmost and rightmost nodes. We call a node v on this path a *junction* if v is incident on a BFS-edge encapsulated by S_i. See Figure 6(a).

Designating Shortcuts, Version 2: Add to the shortcut H_i of version 1, stated in Section 3.1, the set of BFS-edges satisfying the following third property: (3) Consider an edge e on the BFS-path $\mathcal{P}$ connecting left-most and rightmost nodes of S_i and let u be the first/lowest junction above e in this path. Let ℓ_1 be the tree-distance from e to u and ℓ_2 be the tree-distance from u to the LCA of S_i. See Figure 6(b). Add e to H_i if $\ell_2 \geq \ell_1/2 > 0$.

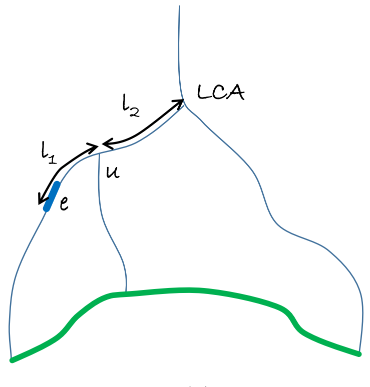
We next show that the addition of these edges does not increase the congestion beyond $O(D \log D)$, while it does reduce the dilation to $O(D \log D)$, as promised.

LEMMA 9. *Assuming G has genus g, each edge e is added to $O((g + 1)D \log D)$ shortcuts H_i.*

Proof. From Lemma 6, we know that the number of shortcut subgraphs that use e following conditions (1) or



(a)



(b)

Figure 6: (a) Red markings on the boundary indicate the junction nodes, and the smaller black ones indicate the leftmost, the rightmost, and the LCA nodes. (b) The parameters ℓ_1 and ℓ_2 used in the rule for determining whether to include a boundary edge e or not.

(2) stated in Section 3.1 is at most $O(D)$. To complete the proof, we show that $O(D \log D)$ shortcut subgraphs H_i use e as an edge on the boundary path of $G[s_i]$.

For the sake of simplicity, let us divide the subgraphs H_i that use e as an edge on their boundary path into four categories, depending on the (clockwise) ordering of the connections around the related first junction: For each subgraph H_i, consider the related first junction node u. There are edges going from u to e, from u to its S_i-descendant that is not a descendant of e, and from u to the LCA. The latter two edges can each be on the left or right of the edge going to e, in the clockwise ordering dictated by the fixed embedding. These two possibilities for each of these two edges form four possibilities in total. We bound the number of subgraphs in each of these categories separately. The arguments for the four cases are essentially the same, so we will explain only one of the cases, say when both connections are to the right of the edge going to e, as depicted in Figure 6(b).

Consider all the shortcut subgraphs that use e as their boundary edge, with right-right connections around the junction, as explained above. Walking from e upwards on the BFS-path P_e connecting e to the BFS-root, let w_1 be the first LCA, related to these subgraphs. Define the *first segment* to be portion of path P_e from e to w_1 (and excluding w itself). Let A_1 be the set of all parts S_j (or equivalently their shortcut subgraphs H_i) that have their first junction in this segment and use e as a boundary edge (with right-right connections). Using an argument similar to Lemma 6, in Lemma 10, we show that there can be at most $O(gD)$ many such parts. Now remove the parts in A_1. Besides these $O(gD)$ parts, any other part that uses e as a boundary edge (with right-right connections) must have its first junction on P_e on or above w. Define w_2 to be the lowest LCA among the LCAs of the remaining parts, and then define the second segment be the portion of P_e between w_1 and w_2, but excluding w_2. Let A_2 be the set of parts that have their first junction in the second segment (and right-right junction connections). By Lemma 10, we again see that there can be at most $O(gD)$ such parts. We can now remove the parts in A_2 and repeat. The key fact is that, the distance from w_2 to e must be at least a $3/2$ factor larger than the distance from w_1 to e. This is because, the first junction of the part with LCA at w_2 is on or above w_1, and the rule for using boundary edges is that the distance from LCA to first junction must be at least half of the distance from the first junction to e. Now, we repeat the same process of defining segments on the remaining parts. In every repetition, the distance from e to the new lowest LCA w_j grows by a $3/2$ factor. Thus, we have at most $O(\log D)$ segments. In each segment, there are at most $O(gD)$ many parts with their first junction in that segment. Hence, in total, at most $O(gD \log D)$ parts use edge e as a boundary edge.

Claim 10. *For each BFS-edge e and each segment defined as above on the BFS-path P_e connecting e to the BFS-root, there are at most $O(gD)$ parts that have their first junction in this segment and right-right connections around the first junction.*

Proof. The proof is to a large extent similar to that of Lemma 6. Consider the parts that have the BFS-edge e added to their shortcut as a boundary edge, and particularly parts that have right-right connections around their first junction. Define the three *special paths* of each of these parts as follows: (p1) the BFS-path going from the BFS-root through e to S_i, (p2) the BFS-path going from the BFS-root through the related first junction (and an encapsulated edge incident to it) to S_i, (p3) the BFS path going from the BFS-root through the LCA to other side of the boundary to

S_i. We show that there cannot be more than $4g+2$ parts such that none of them intersects any of the three special paths of any other. This is an analogue of Claim 7. From this, by repeating the arguments of Lemma 6, one can show that the total number of parts using the BFS-edge e on their boundary, and with first junctions on the given segment, is at most $O((g+1)D)$.

This part of the proof is very much similar to Claim 7: Contract each of these parts S_i into a single node. Moreover, let $e=(v,u)$, where v is the parent, and contract the BFS-path connecting v to the BFS root into a new node v'. Since the set of parts under consideration have their first-junction in this segment but (strictly) before the lowest LCA, and since the junction connections are right-right, we get that after the contraction, all the (p3)-paths going through LCA and the other side of boundary are to the right of the (p2)-paths going through the first junction, which themselves are to the right of the (p1)-paths going through e. Hence, we can again do a local splitting of v' into copies v'_L, v'_C, and v'_R, putting connections from V'_L and v'_R to V'_C, and attaching (p2)-paths to V'_L and (p3)-paths connections to v'_R while the edge e, which goes to u and thus (p1)-paths, is attached to V'_c. This local split again preserves the embedding on a genus g surface. Therefore, we have found our $k_{3,4g+3}$ complete bipartite minor, where here v'_C and parts S_i are on one side and v'_R, v'_L and u are on the other side. This completes the proof as it is contradiction with genus g of the graph G.

Lemma 11. *For each part S_i, the sub-graph $G[S_i]+H_i$ has diameter at most $O(D\log D)$.*

Proof. The proof is quite close to Lemma 8, with the exception that here we will have only $O(\log D)$ classes, hence proving dilation $O(D\log D)$.

Consider a part S_i and the BFS-path $\mathcal{P}$ connecting its leftmost node to its rightmost node. Walking on $\mathcal{P}$ from the leftmost to the LCA, mark the junction nodes. Figure 6(a) shows an illustration of the markings. Call the interval between each two consequent marks a *piece*. We show that except for at most $O(\log D)$ pieces, all the edges in the rest of the pieces are added to H_i. This would mean, if we define classes based on the highest node on the boundary reachable solely via H_i edges, there will be at most $O(\log D)$ classes. Clearly, the H_i-distance between each two nodes of the same class is at most $2D$ and hence, a repetition of the argument of Lemma 8 would prove a dilation $O(D\log D)$.

To show that there are at most $O(\log D)$ boundary pieces not added to H_i, let us walk on one of the boundary paths from the LCA towards the leftmost node. The argument for the right side is similar. Walking down from LCA, skip the first piece and count it for free as one of the pieces not added to H_i. For the j^{th} piece for $j\geq 2$, let e_j be the lowest edge on this piece. Note that if e_j is added to the shortcut H_i as a boundary edge, then also all the edges in the j^{th} piece are added to H_i, as the ℓ_1 distance of them is smaller but their ℓ_2 is the same. If edge e_j is not used in the boundary, we get that the length of the j^{th} piece must have been more than 2 times larger than the summations of the lengths of pieces 1 to $j-1$. Hence, walking down from LCA, every piece not added to H_i grows the tree-distance from LCA by at least a 2 factor. Therefore, there are at most $O(\log D)$ pieces that are not added to H_i.

3.3 Lower Bound, Proving Near-Optimality of Take 2

In this subsection, we show that the parameters obtained in the previous subsection are almost optimal. Particularly, we present a simple planar graph and a partition of its vertices into individually-connected parts such that, regardless of the choice of the shortcuts, we will have $\max\{\mathsf{congestion},\mathsf{dilation}\}=\Omega(D\frac{\log D}{\log\log D})$. More formally, we show the following:

Theorem 12. *(Rephrased Version of Lemma 3) There is an instance of a planar graph $G=(V,E)$ and a partition of its vertices into parts $S_1,S_2,\ldots,S_N$, each inducing a connected subgraph of G, such that regardless of how the shortcut subgraphs H_i are chosen, if each $G[S_i]+H_i$ has diameter at most $\frac{D\log D}{10}$, then there is at least one edge that suffers a congestion of $\Omega(\frac{D\log D}{\log\log D})$.*

Proof. Throughout the construction of this hard instance, we make frequent use of a simple *gadget*, which we explain next.

See Figure 7 for a pictorial illustration of the gadget. The gadget contains $D/5$ *rows*, each being one part S_i. Each part is simply a path; it is made of $\log D$ segments, each containing $2D/5$ nodes. The boundaries of the segments in different rows are connected to each other in a vertical path formation, which we call *column*, and the top node in each of these columns has an edge going out of the gadget box. We later explain where these connections attach to.

Having defined the gadget, we are now ready for explaining the whole structure of the hard instance. We have one path of length $D/5$ hops at the top. We will connect many copies of the above gadget to this path. In fact, the gadgets are divided into $\Theta(\frac{\log D}{\log\log D})$ levels, regarding the span of the path that they connect to; that is, the connections of each level-i gadget are in a sub-interval of length $\Theta(\log^i D)$. More concretely, for each level-1 gadget, the related $\log D$ connections are attached to $\log D$ consecutive nodes on the path, and we have about $\Theta(\frac{D}{\log D})$ such level-1 gadgets. In between the level-1 gadgets, we have the connection points of the level-2 gadgets, thus each of them having its $\log D$ connections span over a subinterval of length $O(\log^2 D)$ of the top path, which also means we have $\Theta(\frac{D}{\log D})$ level-2 gadgets. The construction continues in a similar manner to higher levels, up to level $\Theta(\log D/\log\log D)$, at which point the connections of a single gadget cover the whole top path. See Figure 8 which shows a partial view of the lower bound graph.

First notice that the graph has diameter D, because the farthest node-pairs are those inside the gadgets and for each node v in any of the gadgets, we can reach to its closest column in $D/5$ hops, and from there we can get to the top path in another $D/5$ hops. If we do this for any pair of nodes v and u, possibly in different gadgets, we get to the top path after a total of $4D/5$ hops, from two sides, and the top path itself has length $D/5$ hops.

Notice that each part S_i, i.e., each row in a gadget, has diameter $\frac{D\log D}{5}$. Hence, if $G[S_i] + H_i$ is to have diameter at most $\frac{D\log D}{10}$, H_i should have some shortcut edges. Particularly, one can easily see that going through the other gadgets does not help and essentially the only way to shortcut a row S_i is to add to H_i some of column edges of the related gadget and more importantly, some of the edges of the top path. For a row S_i in a level-j gadget, let $v_1, v_2, \ldots, v_{\log D}$ be the nodes on the top path that the gadget of S_i connects to. Let us call the interval between each v_k and v_{k+1} a *segment*. We can see that, for each segment such that at least one of its edges is not included in H_i, we lose (at least) an additive $D/5$ in diameter and in total, we can afford to not include at most $\log D/2$ segments in H_i, as the diameter of $G[S_i] + H_i$ must be at most $\frac{D\log D}{10}$. From this, we get that the shortcut H_i of each level-1 row S_i must use about half of the edges of the corresponding sub-interval of the top path. Since all but $o(1)$ fraction of the top path edges are each in sub-intervals of level-1 gadgets, and thus $D/5$ rows, we get that overall the level-1 rows add an average load at least about $\frac{D}{10}(1-o(1))$ to the top path edges. Using a similar argument, we can see that the average load added to the top path by each level is at least about $\frac{D}{10}(1-o(1))$. Having $\Theta(\frac{\log D}{\log\log D})$ levels, we get that the average load on the top path is at least $\Theta(\frac{D\log D}{\log\log D})$, which means at least one of the top path edges has a congestion of $\Omega(\frac{D\log D}{\log\log D})$, thus completing the proof.

4 The Algorithmic Aspects of Low-Congestion Shortcuts

In this section, we explain how to distributedly compute the low-congestion shortcuts proven to exist in the previous section, and we also explain the generic method of using these shortcuts, i.e., how to efficiently route messages on the shortcutted parts.

Basic Algorithmics: Throughout, we will assume that the following basics are already computed: a planar embedding, a BFS-tree T, and a left-first-DFS ordering and also a right-first-DFS ordering of it, as described in Section 2. Note that the planar embedding can be computed in $O(D\log n)$ rounds, using the distributed algorithm we presented in the first part of this project [GH15]. Moreover, the BFS can be easily computed in $O(D)$ rounds and even computing the related left and right numberings takes $O(D)$ rounds, using standard techniques: e.g., by computing the number of descendants of all nodes using an upcast and then by allocating the related numbering intervals while performing a downcast. Once these are done, let each node use the tuple (left-ID, right-ID) as its identifier, and in $O(D)$ additional rounds of a pipelined downcast, make each node know the identifiers of all its BFS-ancestors.

4.1 Constructing Low-Congestion Shortcuts

We use part-IDs defined as follows: For each part S_i, we define the part-ID of S_i to be the tuple (min-left-ID, min-right-ID), where min-left-ID and min-right-ID are respectively the smallest left-ID and the smallest right-ID, among the nodes in part S_i. For the purpose of this subsection, suppose that these part-IDs are already computed and each node knows the part-ID of its part. We note that a time-efficient distributed computation of these part-IDs is a highly non-trivial task itself and for it, one has to use ideas (implicitly or explicitly) similar to low-congestion shortcuts. We will compute them using a recursive approach, via the help of shortcuts (of recursively defined smaller parts). This recursive approach for computing part-IDs is explained later in Section 4.3. For the sake of this subsection, suppose that the part-IDs are already computed.

We present some rules for forwarding the part-IDs such that running these forwarding processes exactly identifies the shortcut edges of each part. In Section 4.1.1, we explain these forwarding rules and show that they identify the shortcut edges correctly. In Section 4.1.2, we explain how to run these forwarding processes for all the parts concurrently, in $O(D\log n)$ rounds.

4.1.1 Forwarding Rules Recall that an edge gets added to shortcut H_i because of satisfying one of the three properties described in Section 3.1 and Section 3.2. We explain three forwarding processes, corresponding respectively to these three properties. These three forwarding processes will be performed one by one, and in each of these processes, if a part-ID passes through

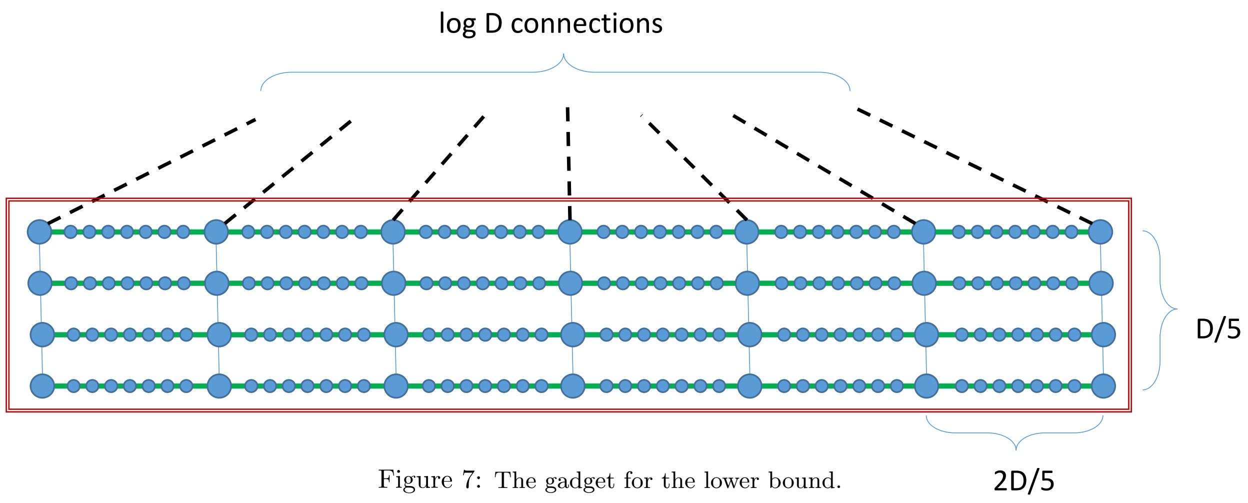


Figure 7: The gadget for the lower bound.

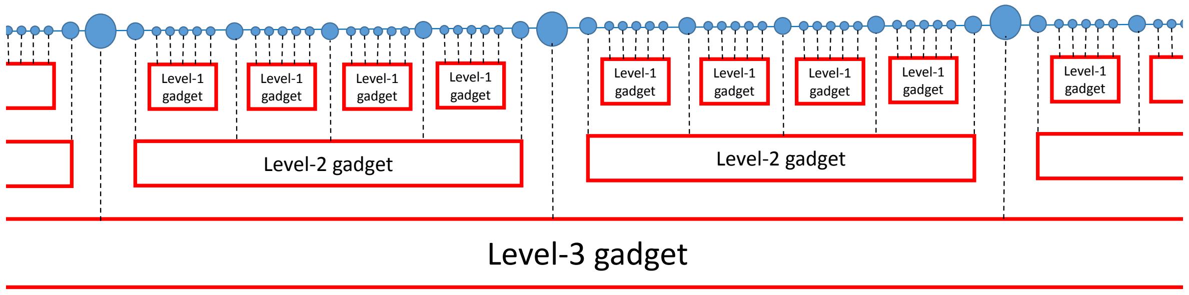


Figure 8: A cropped view of three levels of the lower bound construction.

a BFS-edge e, then as we will prove edge e actually satisfies the respective property and thus e gets added to the shortcut of this part. The forwarding rules are as follows:

(F1) Downcast part-IDs to all BFS-descendants. This identifies edges with ancestors in the part.

(F2) Upcast part-IDs as follows: each node v with ID (l_v, r_v) forwards the part-ID (l_P, r_P) upwards to its parent iff $l_v > l_P$ and $r_v > r_P$. This identifies the encapsulated edges.

(F3) Lastly, we identify the boundary edges that are added to the shortcuts. Each node v that received a part-ID (l_P, r_P) but did not send it up is a boundary node for this part. As such, it checks its ancestor-ID vector from itself upwards for the lowest node u whose ID (l_u, r_u) satisfies $l_u < l_P$ and $r_u < r_P$. Node u is the LCA of that part. Then, v forwards the part-ID downwards the boundary of this part for $\lfloor l_2/2 \rfloor$ steps, where l_2 is the hop distance to the LCA u. To forward along the boundary, each node v forwards to its child w that satisfies one of the following two conditions: (1) $r_v > r_P$ and w has the largest left-ID l_w among children of v such that $l_w \leq l_P$, (2) $l_v > l_P$ and w has the largest right-ID r_w among children of v such that $r_w \leq r_P$. Note that each node knows the IDs of its children and thus can determine which child satisfies this property.

Lemma 13. *The above forwarding rule is correct. That is, the part-ID of a part S_i goes through a BFS-edge e if and only if edge e should be added to the shortcut subgraph H_i of that part.*

Proof. The first forwarding process clearly identifies the BFS-edges that have ancestors in the part. Regarding the second forwarding process, which is supposed to identify encapsulated edges, consider a BFS edge $e = (v, u)$ where u is the parent and v has ID (l_v, r_v). For each part S_i with part-ID (l_P, r_P), edge e is encasuplated by S_i if and only if $l_v > l_P$ and $r_v > r_P$ and v has a descendant in S_i. Note that $l_v > l_P$ and $r_v > r_P$ exactly means that v is not on the boundary path. Suppose that w is the descendant of v in S_i. Then, along the path from w to v, each node has left- and right-IDs respectively greater than l_v and r_v, which also means greater than l_P and r_P. Hence, the part-ID (l_P, r_P) will be sent up by each of the nodes on this path and it will reach v and get sent to its parent u. This means e gets added to the shortcut.

Next, we discuss the case of boundary edges. Notice that, if in the previous forwarding process, a node v receives a part-ID but it does not send it up to its parent, then v is a junction node, and conversely this happens only for junction nodes. Consider an edge e that should be in the shortcut of a part S_i. Recall that the rule for including e in the shortcut as a boundary edge is that the distance from e to the related first junction u is at most twice the distance from u to the LCA. Since u is a junction, it will send this part-ID down on the boundary for $\lfloor l_2/2 \rfloor$ steps, where l_2 is the hop distance from u to the LCA. Hence, this part-ID will go through e implying that e gets added to the shortcut. On the other hand, since only junction nodes start this down-forwarding, a BFS-edge e' gets added to the shortcut (as a boundary) only if there is a junction above it such that the distance from that junction to LCA is at least half of the distance from e to that junction. In this case, even the first junction above e' satisfies this condition, which means e' is added to the shortcut correctly. This means, in the last forwarding process, a part-ID goes through a BFS-edge e if and only if e is one of the boundary shortcut edges of the related part, completing the proof.

4.1.2 Forwarding Identifiers of All Parts Concurrently

We now explain how to perform these forwarding processes concurrently for all the parts, in $O(D \log n)$ rounds overall. We perform the three processes one by one:

(F1): The forwarding process (F1) can be done easily for all the parts in $O(D)$ rounds, using a standard synchronous downcast. Here's a brief sketch: start the forwarding of the part-ID of each node that is at depth i of BFS at round $i + 1$. Since each node has only one part-ID, all these downcasts continue down the BFS in a pipelined fashion and we get the following properties: the part-ID of the root starts at round 1 and gets to each leaf after at most D rounds. After that, every one round, another part-ID arrives. Overall, these mean the whole process finishes after at most $O(D)$ rounds.

(F2): Here we need to do a bit more work. Notice that for each edge e, during (F2), each part-ID needs to go through edge e at most once. Since each edge is in shortcuts of at most $O(D \log D)$ parts, we get that overall, at most $O(D \log D)$ part-IDs need to go through e. Moreover, the whole (F2) process is an upcast on a BFS-tree which means, for each part, the process would take at most D rounds, if the forwarding process of the part was performed alone. Thus, the question is to schedule a number of distributed algorithms, each of which takes at most $O(D)$ rounds if it was alone, and where $O(D \log D)$ messages need to go through each edge. To perform these upcasts together, we use the standard *random delays* technique, first presented by Leighton, Maggs, Rao [LMR94]. Here, we present a brief but sufficient sketch. For more details and a more general treatment of scheduling of distributed algorithms, see [Gha15]. Break time into phases, each made of $O(\log n)$ contiguous rounds. We will perform the upcasts in a synchronous fashion at the speed of one hop per phase. For each part S_i, pick a uniformly random delay $\delta_i \in [1, D]$, and delay the start of the upcast of S_i-nodes by δ_i phases[4]. As a result, for each phase and each BFS-edge, the expected number of part-IDs that are scheduled to go through this edge in this phase is $\Theta(\log D) = O(\log n)$, which means w.h.p., no more than $\Theta(\log n)$ part-IDs are scheduled to go through this edge in this phase. Hence, all of them can be delivered to the endpoint of the edge within the phase, which has room for $\Theta(\log n)$ part-IDs. Therefore, all the upcasts of all parts get performed at a speed of one hop per phase, once they are started. Since the maximum delay is D phases and each upcast takes at most D phases, this means the whole (F2) process finishes after at most $O(D)$ phases, that is, after $O(D \log n)$ rounds.

(F3): The downcast process of (F3) can be done using the same random delay idea, as we did for (F2). The only point to note is that, again, for each BFS-edge e and each part-ID, this part-ID needs to go through e at most once: particularly the copy of the part-ID that starts at the lowest junction above e is the first one that reaches e and it also has the longest allowed hop-count for traversing downwards after passing e. Hence, there is no reason to forward the later copies of the same part-ID that might reach e. Therefore, again, overall at most $O(D \log D)$ messages need to go through each edge, and each downcast would take $O(D)$ rounds if it was alone. This means, repeating the same random delays idea, we can perform the (F3) process in $O(D \log n)$ rounds.

4.2 Routing on the Low-Congestion Shortcuts

Here, we explain the generic method we take for using the low-congestion shortcuts. Consider the scenario that each node v has an $O(\log n)$-bit value x_v and we

[4]Notice that the same random delays δ_i is used by all nodes of the part S_i. For this, we need to share randomness between the nodes. However, as explained in [Gha15], sharing $O(\log^2 n)$ bits of randomness is sufficient for all the parts, as then one can feed these into a psuedo-randomness generator, which produces poly(n) bits of $\Theta(\log n)$-wise independent random bits. This much of independence is enough for standard Chernoff bounds, as shown by [SSS95]. Moreover, sharing $O(\log^2 n)$ bits of randomness can be done by having the root node sample these bits and broadcast them, in $O(D + \log n)$ rounds.

want to compute the part-wise min. That is, we want to have each node u know $x'_u = \min_{v \in S(u)} x_v$ where $S(u)$ indicates the part that contains u. We explain how to solve this problem in $O(D \log D \frac{\log n}{\log \log n})$ rounds, using the shortcuts. We note that the same approach we describe here can be used for other aggregate functions, such as max or sum. We will later use this both in the recursive computation of the part-IDs, and also in the applications including MST and Min-Cut.

For each part S_i, we compute a BFS tree of the subgraph $G[S_i] + H_i$ rooted at the leftmost node of S_i. Note that each of these BFSs will have diameter $O(D \log D)$, because of Theorem 11. To perform all these BFSs for all the subgraphs $G[S_i]+H_i$ concurrently, we again use the random delays trick. Again, the BFSs grow at a synchronous speed of one hop per phase, where each phase now consists of $O(\frac{\log n}{\log \log n})$ consecutive rounds. We delay the start of the BFS of each part S_i by δ_i phases, where δ_i is a uniformly random value in $[1, D \log D]$, sampled by the respective BFS-root. Since by Lemma 9 each edge e is in at most $O(D \log D)$ of subgraphs $G[S_i] + H_i$, the expected number of BFS-tokens scheduled to go through an edge per phase is $O(1)$. Using a Chernoff bound, this means, w.h.p, no more than $\Theta(\frac{\log n}{\log \log n})$ BFS-tokens need to go through the edge per phase. Thus, the phase has enough room for delivering all the scheduled tokens. Therefore, all the BFSs get performed in parallel, growing at a speed of one hop per phase. Since each of the BFSs has diameter $O(D \log D)$, and the initial random delay is at most $D \log D$ phases, all finish after at most $O(D \log D \frac{\log n}{\log \log n})$ rounds.

Now, to compute the part-wise min, we perform a convergecast on each of these BFSs, and then a broadcast/upcast on them. These convergecasts can be pipelined/scheduled similar to what we did above in $O(D \log D \frac{\log n}{\log \log n})$ rounds, and the broadcast afterward also takes a similar time, using the same approach. Hence, overall, after $O(D \log D \frac{\log n}{\log \log n})$ rounds, each node u knows $x'_u = \min_{v \in S(u)} x_v$ where $S(u)$ indicates the part that contains u.

4.3 Computing Part-IDs via Recusive Applications of Shortcuts

Recall that in the construction of shortcuts explained above, we assumed that each part has a part-ID, defined as the tuple (min-left-ID, min-right-ID). We noted that a time-efficient distributed computation of these IDs itself takes an idea similar to shortcuts. Here, we explain how a recursive approach can be used to ratify this problem.

Outline and Intuition: Consider the partition $\mathcal{P}$ of V into parts S_1, S_2, …, S_N. We have $O(\log n)$ phases of recursion, and in each phase i, we will have a further partition $\mathcal{P}_j$, where each part of each phase $\mathcal{P}_j$ is made of merging a number of parts of $\mathcal{P}_{j-1}$. At the start, we have the trivial partition $\mathcal{P}_0$ where each node forms a part of its own. As we continue, in each phase, we perform some merges along G-edges that have both of their endpoints in the same S_i. This will be such that after $O(\log n)$ phases, w.h.p, the end result is the same partition of V into parts S_1, S_2, …, S_N that we want.

The merges are done such that each part of the partition $\mathcal{P}_j$ is formed by a *star-merge* of parts of the partition $\mathcal{P}_{j-1}$. That is, each merge is centered on one of the parts of $\mathcal{P}_{i-1}$ and this central-part just absorbs a number of its neighboring parts in the partition $\mathcal{P}_{i-1}$, some parts that are made of nodes of the same eventual part S_i.

We will maintain the guarantee that during each phase j, each node knows the min-left-ID and min-right-ID of its part in the partition $\mathcal{P}_j$. This is trivial at the start. Once we have this at the end phase, each node knows the min-left-ID and min-right-ID of its part in the partition $\mathcal{P}_{\Theta(\log n)}$. Since w.h.p. this partition is equal to the main partition $\mathcal{P}$, breaking V into S_1, S_2, …, S_N, we have then arrived at our goal.

The Algorithm for Each Phase: In each phase j, we do as follows: We first compute the shortcuts for each part. This can be done in $O(D \log n)$ rounds using the approach explained in Section 4.1.1 and Section 4.1.2 because by recursion, we know that the nodes know the min-left-ID and min-right-ID of their part in partition $\mathcal{P}_j$. Now we make each part select one of its outgoing-edges which has its other endpoint in a part of the same eventual-part S_i. The outgoing edges that are to be added can be any arbitrarily chosen such edge, say the edge that has the smallest id where the id of edge is defined by concatenating the ids of its two endpoint nodes. Computing these edges takes $O(D \log D \frac{\log n}{\log \log n})$ rounds, using the shortcuts of $\mathcal{P}_j$-parts and the approach of Section 4.2. Then, each part suggests the outgoing edge that it selected, for a merge along this edge.

To have the merges be star-shapes, we do as follows: We make the root node of each part of partition $\mathcal{P}_j$, which we define to be its leftmost node, toss a coin. We allow only merges centered at parts with a head coin, and this center-part accepts a suggested merge that is from a neighboring part that has a tail coin toss. Using the shortcuts, we make each node know whether its part has a tail or head, in $O(D \log D \frac{\log n}{\log \log n})$ rounds. Furthermore, we make each tail-part send a message to the node on the other end of its selected suggested merge-edge. On the receiving end, the node accepts this merge-edge only if its own part has a head coin, and if it did so, it sends a message back on the this merge-edge, announcing the acceptance of the merge.

Finally, we compute the new min-left-ID and min-right-ID of the new parts as follows: We make the tail-part nodes send their min-left-ID and min-right-IDs to the other endpoints of the merge-edges. On the head parts, which are the centers of merge, each node might have received the min-left-ID and min-right-ID along some merge-edges. Now using shortcuts, we make this center part compute the new min-left-ID and new min-right-ID of the new potentially bigger part. At the end, these new IDs get passed along the newly added edges, and then get broadcast on the tail parts, via another application of shortcuts of the tail-parts. We now have performed the merge and also have the guarantee that each node knows the min-left-ID and min-right-ID of its part in the new partition $\mathcal{P}_{j+1}$. Hence, we are ready to continue with the next level of recursion.

We now argue that $O(\log n)$ merge-levels suffice, w.h.p. In each phase, for each eventual part S_i such that the current partition $\mathcal{P}_j$ has S_i partitioned into $\eta \geq 2$ parts, η merge-edges get suggested, one by each of these smaller part. Since each edge can be counted at most twice, we get that at least $\eta/2$ distinct merge-edges are suggested. Furthermore, each merge suggestion actually happens with probability $1/4$, i.e., when the receiving end has a head coin and the sending end has a tail coin. This means, we expect an additive reduction of $\Theta(\eta)$ in the number of parts of the eventual part S_i, that is, an expected constant factor reduction in the number of those parts. Hence, after $O(\log n)$ phases, the expected number of remaining parts is $\frac{1}{\text{poly}(n)}$. Thus, Markov's inequality and a union bound over all parts tell us that with high probability, we have arrived at the partition S_i. Finally, since we have $O(\log n)$ levels and each takes $O(D \log D \frac{\log n}{\log\log n})$ rounds, overall this construction takes $O(D \log D \frac{\log^2 n}{\log\log n})$ rounds.

5 Applications

Here, we explain how low-congestion shortcuts allow us to obtain near-optimal $\tilde{O}(D)$ round distributed algorithms for a number of fundamental network optimization problems in planar networks.

5.1 Application 1: Minimum Spanning Tree

Here, we explain how to distributedly compute the Minimum Spanning Tree in $\tilde{O}(D)$ rounds, in weighted planar networks. To solve MST, we simply incorporate the low-congestion shortcuts into (a variant of) the classic 1926 approach of Boruvka [NMN01].

The approach is very much similar to what we did above in Section 4.3. We have $O(\log n)$ phases, initially starting with the trivial partition of each node being its own part. At each time, each part S_i suggests a merge along the edge going out of S_i that has the smallest weight among such edges. It is well-know that all such edges belong to MST. We compute these min-weight outgoing edges, one per part, using the shortcuts, in $O(D \log D \frac{\log n}{\log\log n})$ rounds.

We can again restrict the merge shapes to be star shapes, using the per-part random coin idea explained in Section 4.3, which allows only merges centered on head-parts, each accepting incoming suggested merge-edges from tail-parts. With this idea, we do not really need to perform the recursion of Section 4.3 for each level of MST: in reality, the previous levels of the MST-recursion can be used to recursively maintain that each node knows the min-left-ID and min-right-ID of its current part. Again after $O(\log n)$ phases, we reach the MST, w.h.p. Since we have $O(\log n)$ levels and each takes $O(D \log D \frac{\log n}{\log\log n})$ rounds, overall this construction takes $O(D \log D \frac{\log^2 n}{\log\log n})$ rounds.

In the journal version of this paper, we will explain how we can shave a $O(\frac{\log n}{\log^2 \log n})$ factor from this bound, using much more detailed approaches. The end bound then has the nice property that each of the logarithmic factors in it comes from a reason which is either optimal, e.g. the $\log D$ factor of the shortcuts, or seemingly hard to improve, e.g., the $O(\log n)$ factor of the MST approach of Boruvka.

5.2 Application 2: Min-Cut

Here, we explain our min-cut algorithm, which achieves the following:

THEOREM 14. *(Theorem 5 Rephrased) There is an $\tilde{O}(D)$ round distributed algorithm that in any weighted planar network, computes a $(1+\varepsilon)$ approximation of the min-cut, for any*[5] $\varepsilon > 0$.

Basics: As standard (see e.g. [GK13,NS14]), we assume that the weight $w(e)$ of each edge e is a value in $[1, \text{poly}(n)]$[6]. A cut is a bipartition of the vertex set V into two non-trivial parts $S \subset V$ and $V \setminus S$ and the size of this cut is the summation of the weights of the edges between S and $V \setminus S$. The objective is to find a $(1+\varepsilon)$ approximation of the size of the min-cut, which we denote λ, and also, more importantly, to find a cut $(S', V \setminus S')$ that has size at most $(1+\varepsilon)\lambda$.

[5]In our asymptotic notations, we assume a constant ε. In truth, the bound has a $\text{poly}(1/\varepsilon)$ dependency on ε. Also note that by setting $\varepsilon < 1/\lambda$, where λ denotes the min-cut size λ, one gets an exact algorithm.

[6]Note that any other range of weight values can be transformed to this range, by normalization and a rounding, in $O(D)$ rounds, and the rounding would cost at most a $1 + 1/\text{poly}(n)$ factor in the quality of the approximation.

5.2.1 Outline, and the High-Level Description of the Algorithm

The overall approach for achieving Theorem 14 uses a mix of a number of ingredients: the *tree-packing* approach of Thorup [Tho01], the sampling idea of Karger [Kar94], our planar MST algorithm from previous subsection, some further applications of our low-congestion shortcuts, and finally some small *sketching* type of ideas that we present. We note that the distributed min-cut $(1+\varepsilon)$-approximation of Nanongkai and Su [NS14] also uses the first two of these ingredients.

Throughout, we assume that we have a 2-approximation $\tilde{\lambda}$ of min-cut size λ. This assumption can be removed, as standard, by trying $O(\log n)$ guesses of the form $\tilde{\lambda} = 2^k$ and outputting the smallest cut found overall. Having this, we use Karger's sampling to reduce the min-cut size to $\lambda' = O(\log n)$, while keeping the min-cut sizes around their expectation. Then, we greedily pack $\tilde{O}(\lambda'^7) = \text{poly}(\log n)$ minimum spanning trees, one by one, using the MST algorithm of the previous subsection. Thorup's fascinating result shows that, there is going to be one of these trees $\mathcal{T}$, and specially one of its edges e^*, that if we remove e^* from $\mathcal{T}$, the remaining components define the two sides of a min-cut. We will check all the trees in our collection, and moreover, for each tree, we will check all the cuts each induced by removing one of the tree edges, and we report the smallest cut found. Doing this latter part in $\tilde{O}(D)$ rounds is where we use our new ideas.

Having this brief and very rough explanation, we now proceed to present the algorithm:

Karger's Sampling: Although the communications will be always be in the base graph G, for the following discussions, imagine that we replace the weighted graph G with an unweighted multi-graph G' where each edge e is replaced by $w(e)$ copies of e. Then, sample each edge with probability $\Theta(\frac{\log n}{\varepsilon^2 \tilde{\lambda}})$ and let $\mathcal{G}$ be the spanning graph with the sampled edges. By classical results of Karger [Kar94], we get that w.h.p. $\mathcal{G}$ has min-cut size $\lambda' = \Theta(\frac{\log n}{\varepsilon^2})$, which is $\Theta(\log n)$ for constant $\varepsilon > 0$, and each cut of $\mathcal{G}$ has size within $1 \pm \frac{\varepsilon}{3}$ factor of its expectation. Hence, finding a $(1+\frac{\varepsilon}{3})$ approximation of the min-cut on $\mathcal{G}$ gives an $1+\varepsilon$ approximation (at most) for the min-cut of G.

Throup's Tree-Packing: Now we use the tree-packing idea of Thorup [Tho01] on this graph $\mathcal{G}$. Initially, define the load of each edge to be 0. Then, for $\eta = \Theta(\lambda'^7 \log^3 n) = \Theta(\log^{10} n)$ iterations, do as follows: In iteration i, compute the minimum spanning tree where the weight/cost of each edge is simply its load, and remember this as tree $\mathcal{T}_i$. Increase the load of each of the edges in $\mathcal{T}_i$ by 1, and go to the next iteration. To compute each of these MSTs, we simply use the our $\tilde{O}(D)$-round MST algorithm, presented in the previous subsection.

By Throup's results [Tho01], there is one of these trees $\mathcal{T}_i$ and one edge $e^* \in \mathcal{T}_i$ such that if we remove e^* from $\mathcal{T}_i$, we get a min-cut; more precisely, each of the two connected component of $\mathcal{T}_i \setminus e^*$ is one of the sides of a min-cut. Hence, to find the min-cut, we will work on these trees one by one, each time looking for this special edge e^*, which gives us our desired small cut.

What remains to be solved: When working on each tree $\mathcal{T}_i$, even if this is the right tree, we still do not know which of its edges is that special min-cut defining edge e^*. Hence, we will need to read/approximate the sizes of all the cuts, each defined by removing a single edge of $\mathcal{T}_i$, and we will report the smallest of these, overall, and its associated cut.

Thus, the problem that remains to be solved distributedly can be recapped as follows: given an arbitrary tree $\mathcal{T}$, we want to read the size of each of the cuts defined by removing an edge of $\mathcal{T}$, and report the smallest of these cuts (smallest up to $1+\varepsilon/3$ factor).

Reading Tree-Edge Induced Cuts: Imagine that we pick an arbitrary root for $\mathcal{T}$ and orient its edges outwards from the root, i.e., from the parents to the children. Then, the weight of the cut defined by each edge $e = (u, v)$, where u is the $\mathcal{T}$-parent of v, is equal to the total summation of the weights of $\mathcal{G}$-edges that connect the subtree $\mathcal{T}_v$ below v to the rest of the tree, i.e., $\mathcal{T} \setminus \mathcal{T}_v$. To solve this problem, a basic subroutine that we will make frequent use of it is *subset sums*, where each node u starts with a value x_u, and each node v must learn the sum of the values of itself and its descendants, $y_v = \sum_{u \in \mathcal{T}_v} x_u$. In Section 5.2.2, we explain how to solve this problem in $\tilde{O}(D)$, using our low-congestion shortcuts. Then, in Section 5.2.3, we explain how by using poly$(\log n)$ iterations of this subroutine, we can (simultaneously) compute a $(1+\varepsilon/3)$-approximation of the sizes of cuts each defined by removing one $\mathcal{T}$-edge, all in in $\tilde{O}(D)$ rounds.

5.2.2 Subtree Sums

We first explain how to orient the tree from the root outwards, in $\tilde{O}(D)$ rounds, and then explain how to use this orientation to compute the subtree sums, in $\tilde{O}(D)$ rounds. Both parts make use of our low-congestion shortcuts.

Orienting A Tree in $\tilde{O}(D)$ Rounds: Let us first see how to algorithmically orient $\mathcal{T}$ such that each node knows its parent, in $\tilde{O}(D)$ rounds. Note that the tree $\mathcal{T}$ might have an arbitrarily large diameter and hence, the standard approaches such as doing a flooding on $\mathcal{T}$ from the root outwards would not finish in $\tilde{O}(D)$ rounds. The

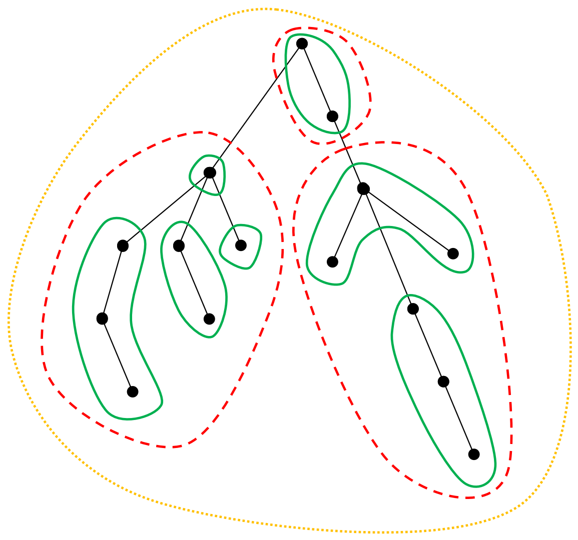


Figure 9: The fragmentation of a tree; each color shows the fragments of one level. The fragments of the first level, where each node is trivially its own fragment, are not depicted.

remedy is in using our low-congestion shortcuts.

Consider the following fragment-merging process which has $O(\log n)$ levels: in each level, the tree $\mathcal{T}$ is partitioned into a number of fragments, each being an induced subtree. In level 1, each node is its own fragment. From that point on, in each level, each fragment of level i is formed by merging some of the fragments of the level $i-1$, which are adjacent in $\mathcal{T}$. More precisely, in each iteration i, each fragment picks a $\mathcal{T}$-edge e that connects it to one of the other fragments, and suggests a merge along e. At the same time, each fragment tosses a coin. Then, each head-fragment accepts the proposed merge edges coming from tail-fragments. That is, each merge is a star-formation, centered at a fragment that has a head in its random coin toss, and with a number of sides which each had a tail coin. See Figure 9, which shows the fragments of three levels. As in the previous subsection, it is easy to see that after $O(\log n)$ such iterations, w.h.p., we have a single fragment, that is the whole tree $\mathcal{T}$. During the iterations, each of the fragments might have a large diameter. Hence, we use our low-congestion shortcuts and the routing explained in Section 4.2, which allows each fragment to pick an edge to one of the other fragments and make this edge, as well as the outcome of the random coin toss, known to all the nodes of the fragment, and this happens for all the fragments together in $O(D \log D \frac{\log n}{\log\log n})$ rounds, per level.

Now let us take a look at these fragments of the $O(\log n)$ levels, from the last level backwards. In the very last level, we have (at most) a single star merge. One of the fragments in this merge contains the root r of $\mathcal{T}$. For every other fragment, we can easily find the root of it, using a few iterations of working on the low-congestion shortcuts. First, identify the fragment that contains the root. Then, let each of the nodes in this fragment send a special message to their neighbors in $\mathcal{T}$. Nodes that received this special message but were not in that root fragment are actually root of their own fragment. Via one application of low-congestion shortcuts, we can make all of the nodes in these fragments know their fragment root. Since a star has depth at most 2, with one more repetition of the same idea, we will reach the point that we have identified the root of each fragment, in the top level. Now we remove the $\mathcal{T}$-edges between these fragments, and recurse, going one level deeper. Since always the fragments are disjoint parts of the graph, and each of them induces a connected subgraph, in each level, we can use low-congestion shortcuts to identify the root of each fragment, in $O(D \log D \frac{\log n}{\log\log n})$ rounds. After repeating this for $O(\log n)$ levels, which takes $O(D \log D \frac{\log^2 n}{\log\log n})$ rounds, we have identified the roots of the fragments of each of the levels.

Now each node v can easily identify its $\mathcal{T}$-parent as follows: v considers its lowest-level fragment, say level i, in which v is not the root of this fragment. Notice that this level-i fragment is a star-shape merge of some level $i-1$ fragments, one of which contains v as its root. During the root-identification of level-i fragments, v received the id of this level-i fragment's root along an edge between level-$(i-1)$ fragments, i.e., from one of its $\mathcal{T}$-neighbors which is in a different level-$(i-1)$ fragment. This neighbor is in fact the parent of v in $\mathcal{T}$.

Subtree Sums in $\tilde{O}(D)$ Rounds: Having the orientation defined above, here we explain an $\tilde{O}(D)$ round algorithm which computes subtree sums. More formally, suppose each node u has a $O(\log n)$-bit value x_u. The objective is to have each node v know the summation of the values in the $\mathcal{T}$-subtree below it, i.e., $y_v = \sum_{u \in \mathcal{T}_v} x_u$, where $\mathcal{T}_v$ includes v and all its $\mathcal{T}$-descendants. In the next subsubsection, we explain how this scheme helps us to find the min-cut.

The first step, which is mainly done for simplicity, is to redefine the $O(\log n)$-level fragmentation process such that each level-i fragment is made of merging a level-$(i-1)$ fragment and some of its level-$(i-1)$ fragment children. See Figure 10 for an example. This fragmentation process is quite similar to the process explained above, with the exception that here, in iteration i, each fragment suggests the edge to its parent fragment for the merge and then, the head-fragments accept all their children tail fragments. Again, we easily see that after $L = O(\log n)$ levels, w.h.p., we reach

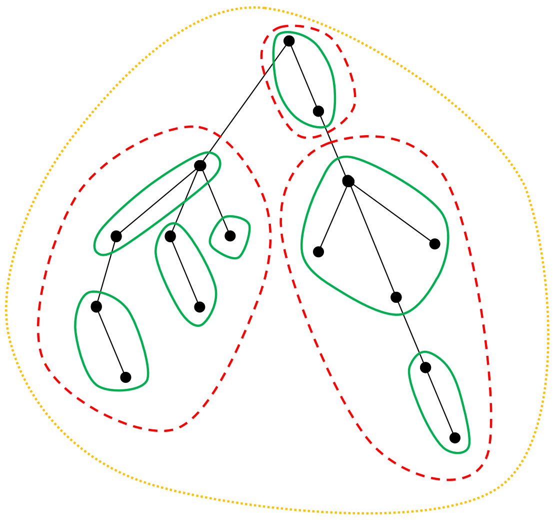


Figure 10: A fragmentation where each level-i fragment is made of merging a level-$(i-1)$ fragment and some of its level-$(i-1)$ fragment children.

a single fragment which includes the whole tree $\mathcal{T}$. As explained above, this process takes $O(D \log D \frac{\log^2 n}{\log \log n})$ rounds overall, thanks to using low-congestion shortcuts in each level.

Having these fragments, we are ready to compute the subtree sum for each node. We will solve the problem recursively. Note that in level-L, there is only one fragment, but this is (potentially) made of a level $L-1$ fragment and its level $L-1$ children. The first step is to separate the problems of these level $L-1$ fragments, by removing the edges between them. Notice that the children level $L-1$ fragments are ready to start the problem within their own fragment and do not need to learn any information from the parent level $L-1$ fragment. However, this parent fragment needs to learn some information from the children. Particularly, for each of the children fragments F_c, there is one node v in the parent fragment F_p that is the parent of the root of this child fragment F_c. We need to deliver the total sum of the values in F_c to node v. If we do this for all such nodes v in F_p, those nodes can increment their value by the received amount and afterwards drop the $\mathcal{T}$-edge to F_c. At that point, the problem would be to solve the subtree sum within each level $L-1$ fragment, which means we have progressed one recursion level, and we can continue to solve the problem recursively now, for $L = O(\log n)$ recursion levels.

To compute the sum of the values in each of the (children) level $L-1$ fragments, so that we can deliver it to $v \in F_p$, we use another recursion, now going bottom-top, i.e., from level 1 to level $L-1$: That is, we walk through these $O(\log n)$ levels, and keep the variant that at each point, the root of each fragment knows the summation of the values in that fragment. At the start, this is trivially satisfied as each level-1 fragment, which is simply a node, knows its own value. In level $i \geq 2$, each level-i fragment is formed by merging one level-$(i-1)$ fragment with some of its level-$(i-1)$ fragment children. Each of the roots of these children fragments knows the total sum of its own fragment. They send these values to their parents, in one round. At this point, we have all the values in nodes of the parent level-$(i-1)$ fragment. Using a convergecast on the BFS of the shortcutted version of these parent fragments, similar to the approach explained in Section 4.2, in $O(D \log D \frac{\log n}{\log \log n})$ rounds, we can gather the summation of these values at the root of the parent level-$(i-1)$ fragment. Then, that parent adds the value of its own level-$(i-1)$ fragment to this sum and remembers the result as the sum of its level-i fragment. After $O(\log n)$ repetitions of all the $L-1$ levels, each level $L-1$ fragment root knows the total sum of its fragment. Hence, as described above, each of these can report the value to its parent node, which is in the parent level $L-1$ fragment, and then we are ready for the higher-level top-bottom recursion to subset sum problems confined to level $L-1$ fragments.

Notice that we are able to use low-congestion shortcut throughout all of these recursions because the parameters of the shortcuts do not depend on how many parts there are, and only require that the parts are disjoint and each part induces a connected subgraph, and these two properties are clearly satisfied for the fragments of each level. Hence, after $O(\log n)$ levels of the top-bottom recursion as described above, each of which contains an $O(\log n)$ level bottom-top recursion, each node v knows its subtree sum $y_v = \sum_{u \in \mathcal{T}_v} x_u$.

5.2.3 Approximating Tree-Edge Induced Cuts

We are now ready to explain the approach we use for approximating the sizes of the cuts each defined by removing one $\mathcal{T}$-edge e from $\mathcal{T}$. This uses the subtree sum subroutine presented in the previous subsubsection, and a small sketching type of idea.

Let us focus on just one of these cuts; as we will see later, the proposed solution solves the problem for all these cuts simultaneously. Consider one $\mathcal{T}$-edge $e = (v, u)$ and suppose that u is the parent of v. Let us say we want to see if the size of the cut $(\mathcal{T}_v, V \setminus \mathcal{T}_v)$ is larger than some threshold $\tau = (1+\varepsilon')^k$ or not, where $\varepsilon' = \varepsilon/3$. Checking the cut versus the $O(1/\varepsilon)$ many thresholds of the form $(1+\varepsilon')^k$ that are within a 2-factor of our guesstimate $\tilde{\lambda}$ of λ will suffice to get a $(1+\varepsilon')$ approximation of the size of the cut.

To compare the cut $(\mathcal{T}_v, V \setminus \mathcal{T}_v)$ versus threshold

τ, what we do is based on repetitions of a simple randomized experiment. Each experiment is as follows: Mark each $\mathcal{G}$-edge as *active* with probability $1-2^{-\frac{1}{\tau}}$[7], and as *inactice* otherwise. Now for each active edge e, this edge contributes a ± 1 to the value of its endpoints, as follows: randomly select one of the endpoints of the active edge e, assign a $+1$ to this endpoint, and a -1 to the other endpoint. Define the value x_w of each node w to be the summation of all the values contributed to w by the active edges incident on w.

Now let us take a look at the subtree sum $y_v = \sum_{w \in \mathcal{T}_v} x_w$, where v is the child in the cut-defining $\mathcal{T}$-edge e under consideration. Each $\mathcal{G}$-edge that has both of its endpoints in $\mathcal{T}_v$ does not contribute anything to y_v as, either it is inactive, or the $+1$ and -1 values of its contributions are both in the subtree $\mathcal{T}_v$ and thus get canceled out. This is also clearly true for edges with both their endpoints in $\mathcal{T} \setminus \mathcal{T}_v$. Hence, the subtree sum y_v is simply the summation of the ± 1 values coming from active edges with exactly one endpoint in $\mathcal{T}_v$. Our indicator random variable for comparing the cut-size $(\mathcal{T}_v, V \setminus \mathcal{T}_v)$ versus threshold τ is whether $y_v = 0$ or not. If the number of $\mathcal{G}$-edges in the cut $(\mathcal{T}_v, V \setminus \mathcal{T}_v)$ is smaller than $\tau(1-\varepsilon')$, then we can see that, the probability that there is at least one active edge in $(\mathcal{T}_v, V \setminus \mathcal{T}_v)$ is at most $0.5-\varepsilon'/10$. On the other hand, if the cut size is at least $\tau(1+\varepsilon')$, then the same probability is at least $0.5+\varepsilon'/10$.

The above is already the *distinguisher* that we desired; but we still need to work a bit more. Note that even if the set of active edges across the cut is non-empty, it is still possible that we get unlucky and the contributions of the (single) $\mathcal{T}_v$-endpoints of these active cut-edges sum up to 0. However, this is easy to fix. For each random experiment defined as above, we repeat $b = \Theta(\log(1/\varepsilon))$ sub-experiments: Throughout each experiment, which has b sub-experiments, we keep the set of active edges the same, but in each sub-experiment, we re-sample the ± 1 contributions to the endpoints. That is, in each sub-experiment, using fresh randomness, we determine which end of each active edge gets a $+1$ and which endpoint gets a -1.

If the set of active cut-edges in an experiment is non-empty, in each of these sub-experiments, $y_v \neq 0$ with probability at least $1/2$. To see why, suppose we expose the randomness of the $\pm$ contributions one by one and just consider the last cut active edge that exposes its ± 1 contribution. Regardless of the outcome of the previous such edges, there is at least a $1/2$ chance that because of the randomness of this last edge, the sum becomes nonzero. We conclude that the probability that, even though the set of active cut-edges in an experiment is nonempty, all of its sub-experiments show $y_v = 0$ is at most $(1/2)^b \ll \varepsilon'/20$. Hence, overall, if $\tau(1+\varepsilon')$, the experiment will show $y_v \neq 0$, in at least one of its sub-experiments, with probability at least $0.5+\varepsilon'/10-\varepsilon'/20 \geq 0.5+\varepsilon'/20$.

Hence, by Hoeffding's bound, we get that $\Theta(\frac{\log n}{\varepsilon^2})$ iterations of this experiment suffice for a high probability distinguisher. More precisely, we simply repeat the above experiment for $\Theta(\frac{\log n}{\varepsilon^2})$ iterations, and check if the majority of the experiments are showing a nonzero y_v (in at least one of their sub-experiments) or not. This w.h.p distinguishes the case where the cut size is greater than $\tau(1+\varepsilon')$ from the case that the cut size is less than $\tau(1-\varepsilon')$. Doing this for the $O(1/\varepsilon)$ many thresholds of the form $\tau = (1+\varepsilon')^k$ that are within a 2-factor of our guesstimate $\tilde{\lambda}$ of λ suffices to get a $(1+\varepsilon')$ approximation of the cut.

Finally, notice that, to run the above process for all the cuts defined each by removing a single $\mathcal{T}$-edge e, all that we need to do is as follows: sample the active edges and their ± 1 endpoint contributions and then compute the subtree sums y_v for all the nodes v of $\mathcal{T}$. The former can be done locally for each edge, say by the larger-ID endpoint of the edges picking these random values, and for the latter part, we already saw how to compute subtree sums for all nodes in $\tilde{O}(D)$ rounds, using the low-congestion shortcuts. This concludes the description of our $\tilde{O}(D)$ round min-cut $(1+\varepsilon)$-approximation algorithm for planar networks.

Acknowledgment: We thank Shay Mozes for many helpful discussions regarding the existence of low-congestion shortcuts in planar graphs, which led to (a variant of) the proof of the results in Section 3.1 (for the case of planar graphs, i.e., when $g = 0$).

References

[AS04] Noga Alon and Joel H Spencer. *The probabilistic method*. John Wiley & Sons, 2004.

[BM08] John A Bondy and Uppaluri SR Murty. Graph theory, volume 244 of graduate texts in mathematics, 2008.

[CHGK14] Keren Censor-Hillel, Mohsen Ghaffari, and Fabian Kuhn. Distributed connectivity decomposition. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 156–165, 2014.

[DMST] Erik Demaine, Shay Mozes, Christian Sommer, and Siamak Tazari. Algorithms for planar graphs and beyond. http://courses.csail.mit.edu/6.889/fall11/. Accessed: July 2015.

[DSHK+11] Atish Das Sarma, Stephan Holzer, Liah Kor, Amos Korman, Danupon Nanongkai, Gopal Pandurangan, David Peleg, and Roger Wattenhofer. Distributed

[7]We note that this is roughly equal to $\frac{1}{\tau}$, but this special formula will simplify the calculations.

verification and hardness of distributed approximation. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 363–372, 2011.

[Elk04a] Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 359–368. Society for Industrial and Applied Mathematics, 2004.

[Elk04b] Michael Elkin. Unconditional lower bounds on the time-approximation tradeoffs for the distributed minimum spanning tree problem. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 331–340, 2004.

[GH15] Mohsen Ghaffari and Bernhard Haeupler. Distributed algorithms for planar networks I: Planar embedding. Manuscript, 2015.

[Gha14] Mohsen Ghaffari. Near-optimal distributed approximation of minimum-weight connected dominating set. In *the Proc. of the Int'l Colloquium on Automata, Languages and Programming (ICALP)*, 2014.

[Gha15] Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, page to appear, 2015.

[GHS83] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. on Programming Lang. and systems (TOPLAS)*, 5(1):66–77, 1983.

[GK13] Mohsen Ghaffari and Fabian Kuhn. Distributed minimum cut approximation. In *Proc. of the Int'l Symp. on Dist. Comp. (DISC)*, pages 1–15, 2013.

[GKK$^+$15] Mohsen Ghaffari, Andreas Karrenbauer, Fabian Kuhn, Christoph Lenzen, and Boaz Patt-Shamir. Near-optimal distributed maximum flow. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, 2015.

[GKP93] J.A. Garay, S. Kutten, and D. Peleg. A sublinear time distributed algorithm for minimum-weight spanning trees. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, 1993.

[GL14] Mohsen Ghaffari and Christoph Lenzen. Near-optimal distributed tree embedding. In *Proc. of the Int'l Symp. on Dist. Comp. (DISC)*, pages 197–211, 2014.

[Kar94] David R. Karger. Random sampling in cut, flow, and network design problems. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 648–657, 1994.

[Kle] Philip Klein. Topics in algorithms: Planar graph algorithms. http://cs.brown.edu/courses/csci2950-r/. Accessed: July 2015.

[KM] Philip Klein and Claire Mathieu. Optimization algorithms for planar graphs. http://cs.brown.edu/courses/cs250/. Accessed: July 2015.

[KMft] Philip Klein and Shay Mozes. *Optimization Algorithms for Planar Graphs*. http://www.planarity.org/, draft.

[KP95] Shay Kutten and David Peleg. Fast distributed construction of k-dominating sets and applications. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 238–251, 1995.

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

[LPS13] Christoph Lenzen and Boaz Patt-Shamir. Fast routing table construction using small messages: Extended abstract. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 381–390, 2013.

[LPS14] Christoph Lenzen and Boaz Patt-Shamir. Improved distributed steiner forest construction. In *the Proc. of the Int'l Symp. on Princ. of Dist. Comp. (PODC)*, pages 262–271, 2014.

[Nan14] Danupon Nanongkai. Distributed approximation algorithms for weighted shortest paths. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 565–573, 2014.

[NC88] Takao Nishizeki and Norishige Chiba. *Planar graphs: Theory and algorithms*. Elsevier, 1988.

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

[NS14] Danupon Nanongkai and Hsin-Hao Su. Almost-tight distributed minimum cut algorithms. In *Proc. of the Int'l Symp. on Dist. Comp. (DISC)*, pages 439–453, 2014.

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[PR99] David Peleg and Vitaly Rubinovich. A near-tight lower bound on the time complexity of distributed MST construction. In *Proc. of the Symp. on Found. of Comp. Sci. (FOCS)*, pages 253–, 1999.

[Rin65a] Gerhard Ringel. Das geschlecht des vollständigen paaren graphen. In *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg*, volume 28, pages 139–150. Springer, 1965.

[Rin65b] Gerhard Ringel. Der vollständige paare graph auf nichtorientierbaren flächen. *Journal für die reine und angewandte Mathematik*, 220:88–93, 1965.

[SSS95] Jeanette P Schmidt, Alan Siegel, and Aravind Srinivasan. Chernoff-hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.

[Tho01] Mikkel Thorup. Fully-dynamic min-cut. In *Proc. of the Symp. on Theory of Comp. (STOC)*, pages 224–230, 2001.