Local-on-Average Distributed Tasks.

Merav Parter * David Peleg† Shay Solomon‡

Abstract

A distributed task is *local* if its time complexity is (nearly) constant, otherwise it is *global*. Unfortunately, local tasks are relatively scarce, and most distributed tasks require time at least logarithmic in the network size (and often higher than that). In a dynamic setting, i.e., when the network undergoes repeated and frequent topological changes, such as vertex and edge insertions and deletions, it is desirable to be able to perform a *local* update procedure around the modified part of the network, rather than running a static *global* algorithm from scratch following each change.

This paper makes a step towards establishing the hypothesis that many (statically) *non-local* distributed tasks are *local-on-average* in the dynamic setting, namely, their *amortized* time complexity is $O(\log^* n)$.

Towards establishing the plausibility of this hypothesis, we propose a strategy for transforming static $O(\text{polylog}(n))$ time algorithms into dynamic $O(\log^* n)$ amortized time update procedures. We then demonstrate the usefulness of our strategy by applying it to several fundamental problems whose static time complexity is logarithmic, including forest-decomposition, edge-orientation and coloring sparse graphs, and show that their amortized time complexity in the dynamic setting is indeed $O(\log^* n)$.

1 Introduction

Background and motivation. What can be computed locally *on average* in dynamic networks? Two decades ago, in their seminal paper [45], Naor and Stockmayer raised a similar question in the static setting, and since then the locality of many tasks have been studied in static networks. A local algorithm allows the nodes to communicate with their direct neighbors in the network for a small (typically constant) number of rounds, after which they need to produce their outputs, which are required to form a valid global solution. The notion of locality usually means a running time that is independent of the network size n. In this paper we slightly extend this notion to include a running time of $O(\log^* n)$; indeed, $\log^* n \leq 5$ for all practical purposes.

Since communication networks were traditionally viewed as mostly static (with changes being treated as faults or as exceptional events that have to be tolerated), the notion of locality was usually studied in the static setting, where the goal is to compute *from scratch* a (global) solution to a given distributed task. However, recent advances in networking and distributed systems drive the need for a theoretical understanding of locality issues in *dynamic* networks. Indeed, in many modern settings, e.g., in mobile and vehicular ad-hoc networks (VANETs), the topology constantly changes. The rapidly-growing presence of mobile communication in all realms of modern technology creates a pressing need for the development of distributed algorithms capable of dealing with constantly changing topologies and maintaining their operation in such settings.

The distinction between static and dynamic distributed settings emerges acutely when dealing with continuous maintenance tasks where the network is required to constantly preserve a certain (existing) structural property or an efficient data-structure with some desirable properties. The dynamic system is characterized by bursts of structural changes, such as vertex and edge insertions and deletions, which may violate the validity of the current solution maintained by the system and hence require a correction (or a re-balancing) procedure. The efficiency of those update procedures is a key parameter in evaluating the quality of a maintenance algorithm in the dynamic setting.

In the presence of frequent topological changes, it is only natural to place an emphasis on bounding the *average* (rather than worst-case) running time (in terms of communication rounds) of such update procedures, hence distributed maintenance tasks can be classified according to the average time required for their update operations. Hereafter, we refer to a distributed task of this type as *local-on-average* if the number of rounds

*CSAIL, MIT. E-mail: parter@mit.edu. Part of this work was done while the author was affiliated with the Weizmann Institute of Science, supported by the Google European Fellowship.

†The Weizmann Institute of Science, Rehovot, Israel. Email: david.peleg@weizmann.ac.il. Supported in part by the Israel Science Foundation (grant 1549/13), the I-CORE program of the Israel PBC and ISF (grant 4/11), and the United States-Israel Binational Science Foundation (grant 2008348).

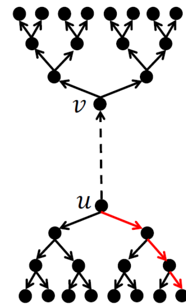
‡School of Computer Science, Tel Aviv University, Tel Aviv, Israel. E-mail: solo.shay@gmail.com. Part of this work was done while the author was affiliated with the Weizmann Institute of Science, under support of the Koshland Center for basic Research.

required by the update procedure for repairing the global solution following a topological change in a dynamic network is constant or nearly-constant on average (over the topological changes). Ideally, just the updated vertices should wake up as a result of the topological change and the ensuing update operation – such a limitation (hereafter, the "local wake-up" limitation) reflects most real-life scenarios. So whereas in the standard setting one measures the maximum explored radius among all vertices of the network, in the dynamic setting the measure of interest is the *average* (over a sequence of update operations) explored radius among the *updated vertices*.

The challenge in developing algorithms for *distributed dynamic* networks that are *local-on-average* stems from the need to cope with the two types of uncertainties arising from the concurrent presence of distribution and dynamics, namely, (1) the possibly unstable local neighborhoods due to topological changes, and (2) the limited local view and lack of information about the global topology at any time.

The locality of various tasks have been studied thoroughly for static networks, yielding both positive and negative results. The key observation emerging from the long line of research efforts on static locality is the unfortunate fact that local tasks are rather scarce, and most distributed tasks (e.g., MIS, coloring, MST construction, etc.) are *non*-local. For example, in an n-vertex graph G with maximum degree Δ and diameter D, computing a maximal independent set (MIS) requires $\Omega(\max\{\sqrt{\log n}, \log \Delta\})$ rounds [40] and constructing an MST (as well as other types of spanning trees) requires $\Omega(D)$ rounds.

In contrast, very little is known about the locality-on-average of distributed tasks. Clearly, any (statically) local problem is also local-on-average (as well as in the worst-case) in the dynamic setting. Yet given the scarcity of natural local distributed tasks, it is important to understand which (statically) non-local distributed tasks are *dynamically local*. To understand this fundamental question better, let us consider two examples of non-local (and significant) tasks: MIS selection and δ-edge-orientation.[1] In a dynamic setting, the goal is to maintain the required structure (MIS in the former example, δ-edge-orientation in the latter) *while keeping a tab on the update time*. It is easy to see that dynamically maintaining an MIS requires $O(1)$ rounds in the worst-case, as for any addition or removal of a vertex v or an edge (u, v), it is sufficient to explore the 2-neighborhoods around u and v. Hence MIS selection is a dynamically local (even in the worst-case) task. Dynamically maintaining a δ-orientation seems, at first glance, to be non-local, even in *trees* (or *forests*) – as an update operation might require $\Omega(\log_\delta n)$ rounds in the worst-case (as shown in the figure for $\delta = 2$). Yet, as will be shown in this paper, a more careful exploration reveals that the task of maintaining edge-orientations with near-optimal outdegree is local-on-average (over a sequence of vertex and edge updates) in general graphs.



Towards the ultimate goal of understanding which non-local distributed task are local-on-average, we provide a novel amortization technique, referred to hereafter as *log-starization*. Our log-starization technique is then used to convert a number of $O(\text{polylog}(n))$-round static distributed algorithms for several variants of load-balancing tasks, such as forest-decomposition, edge-orientation, and coloring, into $O(\log^* n)$-round dynamic distributed algorithms, i.e., ones that maintain the solution for the task within $O(\log^* n)$ rounds on average in the presence of topological changes such as edge and vertex insertions and deletions).

While our log-starization technique can be applied to any of the aforementioned tasks (as well as others), in this extended abstract we chose to demonstrate it via the edge-orientation problem. Low outdegree orientations have found various applications, e.g., for finding simple cycles and paths via the "color-coding" scheme [4], or for maintaining data-structures for answering adjacency and short-path queries [16, 12, 37, 38]. Such orientations have also proven to be useful in the contexts of load-balancing, maximal matchings, prize-collecting TSP's and more. (See [14, 23, 21, 19, 47, 30], and the references therein.) A closely related problem is *forest-decomposition*, where one aims to decompose the edges of the graph G into a small number of edge-disjoint forests. Obviously, a decomposition of a graph into ℓ edge-disjoint forests immediately yields an ℓ-orientation. In this paper we show the other direction: An ℓ-orientation yields a decomposition into at most 2ℓ edge disjoint forests. Moreover, we observe that a dynamic maintenance of the former can be translated into a dynamic maintenance of the latter with a constant overhead in the update time (even in the worst-case), in both centralized and distributed networks; this resolves a question raised by Erickson [24]; see Sec. 3.1.

[1] A δ-edge-orientation (shortly, δ-orientation) is an orientation of the edges of an undirected graph (yielding a digraph) so that the outdegree of all vertices is bounded by some parameter $\delta \geq 1$.

Common to all aforementioned tasks is the need to maintain a "balanced" state of the network in which the "load" of each vertex is near-optimal, where the definitions of load and optimal are context-dependent. While the amortized time complexity of all our algorithms is $O(\log^* n)$ for any graph, the load guarantee inevitably depends on the *density* of the graph. More concretely, the *arboricity* $a(G)$ of a graph G is the minimum number of edge-disjoint forests into which G can be decomposed, thus measuring how *uniformly sparse* the graph is; see Sec. 1.6 for more details. For edge-orientation, the "load" of a vertex is the number of its outgoing neighbors, i.e., its outdegree, and it is easy to see that the overall load (i.e., the maximum outdegree of a vertex) must be at least $a(G) - 1$; thus in a "balanced" state the outdegrees of all vertices should be close to $a(G)$. For forest-decomposition, the load of a vertex v can be viewed as the number of edge-disjoint forests that contain an edge incident on v, and the overall load (i.e., the number of edge-disjoint forests into which G is decomposed) cannot be smaller than $a(G)$ by definition. As for coloring, the vertex load can be viewed as the number of distinct colors occupied by its neighbors, and the overall load is the number of colors used. While there are graphs with low chromatic number and high arboricity (e.g., a complete bipartite graph), there exist (infinitely many) graphs with arboricity $\Theta(a)$ and chromatic number $\Omega(a)$ (e.g., a clique on a vertices, with the remaining $n - a$ vertices being isolated), for any parameter a. Consequently, one cannot hope to color *every* graph G with $o(a(G))$ colors. (See Theorems 1.2 and 1.3.) We stress that, although the family of constant arboricity graphs is important by itself (including all planar and excluded minor graphs), our results apply to *general graphs*.

Our work builds upon two lines of work. The first is the study of dynamic edge-orientations initiated by Brodal and Fagerberg [12] (and followed by [37, 27]), who provided *dynamic centralized* algorithms for maintaining low outdegree orientations for general graphs, with the bound on the outdegree depending on the graph's arboricity; as mentioned, such a dependency is inevitable. In particular, for graphs with arboricity a, the algorithm of [12] maintains an $O(a)$-orientation with amortized update time $O(\log n + a)$.[2] The second line involves the work of Barenboim and Elkin [9], who studied the forest-decomposition problem in the distributed static setting. Their main results are summarized in the following theorem.

[2]The algorithm of [37] maintains an $(\log n + a)$-orientation with amortized time $O(a)$, and that of [27] extends these results to a complete tradeoff: $O(\beta \cdot a)$-orientation with amortized update time $O(\log(n/(\beta \cdot a))/\beta) \approx O(\log n/\beta)$.

THEOREM 1.1. *[9] For a graph G with arboricity $a(G)$ and any $q > 2$, there exists a distributed algorithm that computes a decomposition of G into at most $((2 + q) \cdot a(G))$ forests (and hence also a $((2 + q) \cdot a(G))$-orientation) in $O(\frac{\log n}{\log q})$ rounds; an $O(q \cdot a(G)^2)$-coloring for G can be computed in $O(\log^* n)$ more rounds.*

A fundamental result of Linial [41] states that any $O(\log_d n)$-round algorithm for coloring d-regular n-vertex *trees* uses $\Omega(\sqrt{d})$ colors. This lower bound was generalized in [9] for graphs with higher arboricity: Any distributed algorithm for computing $O(q \cdot a(G)^2)$-coloring requires $\Omega(\frac{\log n}{\log q + \log a(G)})$ rounds, where $a(G) = O(n^{1/4})$ and $q = O(n^{1/4}/a(G))$. Moreover, the number of rounds required to compute a decomposition into at most $O(q \cdot a(G))$ forests (or an $O(q \cdot a(G))$-orientation) is $\Omega(\frac{\log n}{\log q + \log a(G)}) - O(\log^* n)$. In particular, none of these problems can be solved statically in sub-logarithmic time, even in trees! [3]

None of the above results consider the dynamic distributed setting: The former deals with the dynamic setting but provides a centralized solution, whereas the latter provides a distributed solution but concerns only the static setting (and it is unclear how these distributed algorithms can be extended to maintain their solution in a dynamic environment, without applying them from scratch following each update).

Our contributions. The main result of this paper is summarized in the following theorem.

THEOREM 1.2. *For every n-vertex dynamic graph G with arboricity $a(G)$, there is a distributed algorithm for maintaining an $O(a(G) + \log^* n)$-orientation in $O(\log^* n)$ amortized time (per update operation).*

In Sec. 3, we present some applications of our amortization scheme. Some of these applications (e.g., adjacency-labeling scheme) follow immediately from the maintenance algorithm for edge-orientation, whereas others (e.g., coloring) require a more delicate adaptation of the log-starization technique.

THEOREM 1.3. *(a) Dynamic forest-decomposition: A decomposition into $O(a(G) + \log^* n)$ forests can be maintained with $O(\log^* n)$ amortized time (per update operation).*

(b) Dynamic coloring: An $O(a(G) \cdot \log^* n)$-coloring can be maintained with $O(\log^* n)$ amortized time.

[3]Note that the (distributed) forest-decomposition problem is not trivial even if the underlying network is a forest. To see why, note that at the end of the execution of a distributed algorithm that computes a decomposition into q forests (even if $q = 1$), any vertex should know which one among its neighbors is its parent, for each of the q underlying forests.

(c) *Dynamic adjacency labeling: an adjacency labeling scheme with label size of $O(a(G) + \log^* n)$ bits can be maintained with $O(\log^* n)$ amortized time.*

To appreciate this contribution, note that while all these problems have a static logarithmic time-complexity even in trees, our results show that these problems are local-on-average in *general graphs*. Moreover, while a low amortized time does not imply a-priori any non-trivial worst-case guarantee, all our algorithms achieve $O(\log^* n)$ amortized time together with a worst-case update time of $O(\log n)$.

We also provide a load-balancing scheme for expander graphs. Consider a dynamic model allowing edge, vertex and job insertions and deletions. The system evolution is described by a sequence of graphs $\sigma = \{G_t\}$, where G_t is the graph after the t'th operation, containing n_t vertices and J_t jobs. Assume that throughout the execution, the graphs preserve two properties: (1) they are vertex-expanders (defined later); (2) the load ratio J_t/n_t is bounded by $\widehat{\delta}$. Such a network is called *load-bounded expander network*.[4]

THEOREM 1.4. *Given a load-bounded expander network $\sigma = \{G_t\}$, there is a distributed algorithm for maintaining $O(\widehat{\delta} + \log^* n)$ jobs at each vertex $v \in G_t$ with $O(\log^* n)$ amortized time (per update operation).*

A general hypothesis. For our log-starization technique to work, it is critical to allow some *slack* in the load guarantee. However, this relaxation is justified for two reasons. First, allowing no slack whatsoever implies strong lower bounds on the amortized time complexity of many distributed tasks, under the local wake-up limitation. Indeed, a single update operation in one part of the network may trigger a change in a far away part of the network. Consider for example the maximum matching problem. Following a single edge deletion/insertion on one side of an n-vertex simple path, the matching edges must be swapped with the non-matching edges. Since most of these edges are at distance $\Omega(n)$ from the two updated vertices, we obtain a lower bound of $\Omega(n)$ on the amortized update time. A similar scenario happens with the edge-orientation and forest decomposition problems, assuming we aim for an $a(G)$-orientation or a decomposition into $a(G)$ forests. There the lower bound on the amortized update time is $\Omega(n/a(G)^2)$.[5] Second, as mentioned in the paragraph following Theorem 1.1, the *static* time complexity of edge-orientation and forest-decomposition allowing a *multiplicative slack of* q is at least $\Omega(\frac{\log n}{\log q})$ *even in trees*, which gives a lower bound of $\Omega(\log^{1-\epsilon} n)$ in the regime $q = O(2^{\log^\epsilon n})$; for coloring (with slack) there is a similar lower bound. In other words, all these statically non-local tasks (with slack) are local-on-average. Maybe this phenomenon holds in general? This paper makes an initial step towards the following hypothesis: For any distributed task (that allow for a small slack) with static time complexity $t(n)$, its amortized time complexity is $t^*(n)$ (defined as the number of times the function t must be iteratively applied before the result is less than or equal to 1). Since the static time complexity of many distributed tasks is $O(\text{poly}(\log n))$, our hypothesis suggests that *all* such tasks are local-on-average.

Our technique. Our log-starization technique is based on the accounting method (cf. Chapter 17 in [17]).[6] The underlying idea behind our technique applied to all problems studied is as follows. Each vertex will store tokens in its account, in proportion to its load. In a perfectly balanced state, all vertex loads are low (bounded by say ℓ_{min}), and all accounts are empty. More concretely, a vertex of load ℓ should have at least $f(\ell)$ tokens in its account, with f being a monotone increasing function satisfying $f(0) = f(1) = \ldots = f(\ell_{min}) = 0$. A vertex whose load increases as a result of an update operation should be compensated by depositing sufficiently many tokens in its account. Consider an update operation that "defects" some vertex u, i.e., increases its load beyond the required threshold ℓ_{max}. Our update scheme starts by exploring the local neighborhood $\Gamma_{h_0}(u)$ of radius $h_0 = O(1)$ around u. If a balanced state can be restored by modifying only the local neighborhood $\Gamma_{h_0}(u)$, then we are done. Otherwise, a "reset" operation is applied on $\Gamma_{h_0}(u)$ in order to make it perfectly balanced. This reset may damage vertices on the boundaries of $\Gamma_{h_0}(u)$ (i.e., the vertices connecting it to the rest of the graph), thus turning these vertices to be *imbalanced*. Yet, our analysis shows that the subgraph induced on $\Gamma_{h_0}(u)$ has high expansion (i.e., number of vertices is exponential in the depth h_0), and that most vertices in $\Gamma_{h_0}(u)$ have high

[4]Note that the load guarantee (provided by Theorem 1.4) does not depend on the arboricity of the network.

[5]This lower bound is given in [12] for centralized networks, but it also applies to (local wake-up) distributed networks.

[6]The *accounting method* is a standard approach for amortized analysis, adhering to the following guiding principle. When a topological change occurs, we issue a number $\mathcal{T}$ of new *tokens* and store these tokens at *accounts* reserved for vertices of the graph. Each token can be used by the algorithm to "pay" for $O(1)$ communication rounds. The goal is to show that at any stage, the total number of tokens accumulated so far is at least as large as the total runtime of the algorithm. Following an update operation whose actual cost is smaller than $\mathcal{T}$, the system acquires a surplus of tokens that will be stored at vertices of our choice. This surplus may be used later for supporting update operations whose actual cost is larger than $\mathcal{T}$.

load. Hence, exponentially many tokens (in h_0) can be withdrawn from the accounts of the internal vertices of $\Gamma_{h_0}(u)$ due to the reset operation. This procedure can therefore be applied recursively for every imbalanced vertex w on the boundary of $\Gamma_{h_0}(u)$ (essentially, this can be done *in parallel* for all these vertices), with an exponentially increasing exploration radius $h_1 = 2^{h_0}$. Repeating this process results with a balancing procedure of "exponential speedup", which terminates within $\log^* n$ phases. Although the balancing procedure consists of $\log^* n$ phases, it is a-priori unclear why the amortized cost should exceed some constant. Indeed, assuming all neighborhood explorations at phase i of the procedure can be "paid for" by tokens released due to reset operations of previous phases, the entire procedure is carried out for free in the amortized sense.

The crux of the problem lies in the "termination cost" of the procedure. Consider the first phase of the procedure, and assume the subgraph induced on $\Gamma_{h_0}(u)$ does not have high expansion. In this case a balanced state can be restored by modifying the local neighborhood $\Gamma_{h_0}(u)$; that is, the load of u is decreased at the expense of increasing the load of a carefully chosen vertex w in $\Gamma_{h_0}(u)$ of significantly lower load. While w remains balanced, sufficiently many tokens should be deposited in its account (to compensate for its load increase). Since each update operation is followed by a grant of $O(\log^* n)$ fresh tokens, these tokens can be used to cover the termination cost of this subgraph. The interesting case arises when the vertices of $\Gamma_{h_0}(u)$ are of high load, and we cannot terminate this subgraph. (We terminate a graph if it contains a low load vertex.) In general, the number of explored subgraphs (and thus the number of terminating subgraphs) may grow exponentially in each phase. Assuming the balancing procedure does not terminate its execution (at all explored subgraphs) within $O(1)$ phases, the $O(\log^* n)$ fresh tokens cannot cover the total termination costs of the procedure. Therefore, the tokens withdrawn due to a reset operation should cover the "termination costs" of the next phase, in addition to the aforementioned "exploration costs".

Let us take a deeper look at a terminating subgraph "rooted" at some vertex x that becames imbalanced, either by the update operation (which initiated the balancing procedure) or throughout the execution of the procedure. Suppose that the load of x increased from ℓ_x to $\ell_x + 1$. Our analysis shows that, in addition to the $f(\ell_x)$ tokens that are guaranteed to be stored at x's account, $h(\ell_x)$ more tokens can be deposited in x's account, where $h(\ell_x) < f(\ell_x) - f(\ell_x - 1)$, either due to the fresh tokens of the update operation or due to reset operations of previous phases. To terminate this subgraph we decrease the load of x from $\ell_x + 1$ to say ℓ_x, but this is done at the expense of increasing the load of some other vertex y from ℓ_y to say $\ell_y + 1$. While the number of tokens that can be released from x's account (due to its load decrease) is $h(\ell_x)$, the number of tokens that we need to deposit in y's account (to compensate for its load increase) is $f(\ell_y + 1) - f(\ell_y)$. In order to terminate the subgraph, it is crucial that the total, i.e., $h(\ell_x) - (f(\ell_y + 1) - f(\ell_y))$, will be non-negative. As $h(\ell_x) < f(\ell_x) - f(\ell_x - 1)$, the load of y must be smaller than that of x by at least two units, i.e., $\ell_y \leq \ell_x - 2$. Thus we need a "2-unit decay" to terminate a subgraph, which means that the load of the explored vertices may decrease by one unit at each phase of the procedure. Indeed, this is the worst-case scenario with respect to all the problems studied: A gradual "1-unit decay" in the vertex loads (thus precluding termination), starting with the updated vertex u whose load exceeds ℓ_{max}, where in each phase of the procedure all subgraphs but one terminate, and the non-terminating subgraph has vertex loads smaller by 1 than those of the previous phase, but exponentially larger radius. A gradual decay in vertex loads introduces a contradictory constraint: The loads of all vertices in a non-terminating subgraph must exceed ℓ_{min}, otherwise no tokens can be withdrawn from their accounts due to the reset operations. Thus for this scheme to work, the required load threshold ℓ_{max} must be larger than ℓ_{min} by a factor of $\log^* n$ (either additive or multiplicative, depending on the problem). As will be shown by a careful examination of this amortization scheme – the optimizing amortized cost due to the above constraints is $O(\log^* n)$.

Related work. The design of distributed protocols for dynamic networks has received increasing attention in recent years. The first results [2, 7, 6, 8] employed simulation techniques to convert a distributed algorithm running in a static network to one running in a dynamic network (which undergoes topological changes). These results concern algorithms for one-time *computations* (from scratch). Since then, many other distributed algorithms have been designed for *specific* distributed tasks assuming some underlying dynamic model. These works can be roughly classified into two categories: (1) distributed dynamic *computation*, where one has to compute (not maintain) a one-time solution (from scratch) in the presence of topological changes, cf. [39, 25, 18, 29, 13, 42]; the simulation results of [2, 7, 6] fall into this category; (2) distributed dynamic *maintenance*, where some target structure or function has to be maintained; our results fall into this category. This "dynamic maintenance" setting was studied for *dynamic routing schemes* [3, 32, 35], *informative labeling*

scheme [31, 36, 33, 34], maximal independent set [15], BFS trees [28] and sparse spanners [22, 10].

Another variant of distributed maintenance is *self-healing* for reconfigurable networks. In this setting, the distributed *local* algorithm is allowed to *add* edges to the structure following adversarial removal of vertices or edges, in order to preserve certain graph properties (e.g., diameter, maximum degree, expansion) of the original network (i.e., before the attack), cf. [26, 50, 51]. Note that this approach requires the network to be reconfigurable, in the sense that it should be possible to change the network topology. In contrast, in the current work we follow the standard dynamic model where the input physical network is given (i.e., determined by the original network and the adversarial modifications over time). Here, the goal is to maintain a *subnetwork* that satisfies some desired properties. Finally, a long line of research focused on *dynamic load balancing* [20, 43, 44, 5, 11]. Related problems are *token distribution* (or *job distribution*) [49] and *token aggregation* [1].

The dynamic distributed model. The dynamic distributed model is defined as follows. Starting with the empty graph $G_0 = (V, E_0 = \{\emptyset\})$, in every round $t > 0$, the adversary chooses a vertex or an edge to be either inserted or deleted from G_{t-1}, resulting in G_t. (As a result of a vertex deletion, all its incident edges are deleted. A vertex insertion does not entail any insertions of edges incident on it.) The communication model is the $\mathcal{LOCAL}$ model (cf. [48]), which is a standard distributed computing model capturing the essence of spatial locality. In this model, computation proceeds in fault-free synchronous rounds during which every processor exchanges messages of unbounded size. Upon the insertion or deletion of a vertex v or an edge $e = (u, v)$, only the affected vertices are woken up (v in the former case, u and v in the latter). We assume that the topological changes occur serially and are sufficiently spaced so that the protocol has enough time to complete its operation before the occurrence of the next change. since all our algorithms achieve a worst-case update time of $O(\log n)$, this assumption should be acceptable in many practical scenarios; also, it can be removed at the cost of increasing our time bounds by a factor of k, where k is a bound on the number of concurrent updates. In the distributed dynamic setting, an *amortized update time* bounds the *average* number of communication rounds in the $\mathcal{LOCAL}$ model, needed for repairing the solution per update operation, over a worst-case sequence of update operations.

2 Dynamic distributed orientation

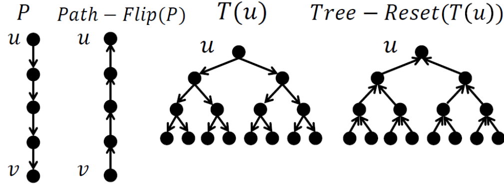
2.1 Preliminaries The arboricity of a graph $G = (V, E)$ is defined by $a(G) = \max\{\lceil |E(U)|/(|U| - 1)\rceil \mid U \subseteq V, |U| \geq 2\}$ where $E(U) = \{(u, v) \in E \mid u, v \in U\}$. An *$f$-forest-decomposition* of a graph G is a partition of its edge set, $E = E_1 \cup E_2 \cup \ldots \cup E_f$, such that the subgraph induced by each E_i is a forest. The well-known theorem of Nash-Williams [46] states that a graph G has arboricity at most a iff G has an a-forest-decomposition. An immediate (yet inefficient) approach to maintaining forest-decomposition in a dynamic distributed setting is to apply, following each update operation, the *static distributed algorithm* of [9]. This approach results in $O(\log n)$ rounds per update. Note that the lower bound of [41, 9] does not hold for the dynamic case. Hence, better bounds can be obtained, as we show in this paper.

In the context of edge-orientations, another approach is to distribute the *dynamic centralized algorithm* of [12]. However, this approach will also result in $O(\log n)$ rounds on average (per update operation).

In this section we demonstrate our log-starization technique via the edge-orientation problem.

Notation. For an undirected graph $G = (V, E)$ and vertex $v \in V$, let $\texttt{deg}(v) = |\{u \mid (u, v) \in E\}|$ be the degree of vertex v. For a digraph $\overrightarrow{G} = (V, \overrightarrow{E})$, let $\Gamma_{in}(v) = \{u \mid (u, v) \in \overrightarrow{E}\}$ and $\Gamma_{out}(v) = \{u \mid (v, u) \in \overrightarrow{E}\}$ denote the set of incoming and outgoing neighbors of v. Let $\texttt{deg}_{in}(v) = |\Gamma_{in}(v)|$ (resp., $\texttt{deg}_{out}(v) = |\Gamma_{out}(v)|$) denote the in-degree (resp., outdegree) of v in $\overrightarrow{G}$. Let $dist(v, u, G)$ (resp., $dist(v, u, \overrightarrow{G})$) denote the length of the shortest-path (resp., shortest directed path) from v to u. The directed $u \to v$ distance in $\overrightarrow{G}$ is then $dist(v, u, \overrightarrow{G})$. The *out-neighborhood* $\Gamma_d(v, \overrightarrow{G}) = \{u \mid dist(v, u, \overrightarrow{G}) \leq d\}$ is the set of all vertices at directed distance at most d from v. A directed BFS tree $T(v)$ of depth $\max_u dist(v, u, \overrightarrow{G})$ containing all vertices of $\Gamma_{d^*}(v, \overrightarrow{G})$ is called a *full depth* directed BFS tree. For a directed BFS tree $T(v)$ of depth d, any level i for $i \in \{0, \ldots, d-1\}$ is called an *internal* level. The set of vertices at the *last* internal level (level $d-1$) is denoted by $\widehat{L}(v)$; the set of leaves (at level d) is denoted by $L(v)$.

Reset operations. An undirected (resp., directed) edge between u and v is denoted by $\{u, v\}$ (resp., (u, v)). We define two types of operations. A *flip* operation of a directed edge $e = (u, v)$, denoted by $\texttt{Edge-Flip}(e)$, reverses the direction of the edge into (v, u), i.e., $\texttt{Edge-Flip}(e) = (v, u)$. The *reset* operation, $\texttt{BE}(\overrightarrow{G}')$, on a directed subgraph $\overrightarrow{G}'$ with bounded arboricity a, employs the Barenboim and Elkin algo-



rithm of Thm. 1.1, resulting with an $O(a)$-orientation in $O(\log n)$ communication rounds.

Besides the BE algorithm, we refer throughout to two additional types of re-orientation operations applied on trees and paths. Let $T(u)$ be a directed tree all whose edges are directed *away* from the root u. Denote by `Tree-Reset`$(T(u), u)$ the operation of reversing the directions of all edges and directing them *towards* the root u (making the root's outdegree zero). For a directed $u \to v$ path P, let `Path-Flip`(P) denote the operation of reversing the orientation of all path edges, i.e., transforming P into a directed $v \to u$ path. For an illustration of these operations see the above figure.

Observation 1. *Every reset operation on a directed $w \to w'$ path P increases (resp., decreases) the outdegree of w' (resp., w) by 1 and does not change the outdegrees of the internal path vertices.*

Our balancing procedure is applied only following edge insertions (all other update operations do not increase the outdegree of vertices). We assume there is an integer a such that throughout the execution, $a(G_t) \leq a$, for every $t \geq 0$. Our goal is to maintain a δ-orientation with amortized time $O(\log^* n)$, where δ is close to the arboricity bound a of the dynamic graph $G = G_t$.

The key technique: from $O(\log n)$-static to $O(\log^* n)$-amortized. We introduce a new amortized log-starization technique, allowing one to obtain better bounds compared with the results of either Brodal and Fagerberg (in the *dynamic centralized* setting) or Barenboim and Elkin (in the *static distributed* setting). The crux of the problem is to collect $\Omega(\log n)$ tokens, using which the system can be reset via a *worst-case* $O(\log n)$-time algorithm. We start (Sect. 2.2) with graphs of arboricity 1, i.e., cycle-free graphs (or forests). First, we describe a simple amortization scheme that maintains a 3-orientation with amortized time $O(\log \log n)$. Note, however, that the resulting algorithm has a bounded *amortized* (rather than worst-case) time, and hence one cannot simply repeat this algorithm to improve the amortized cost. Breaking the simple $O(\log \log n)$-time barrier is quite challenging; we focus on maintaining an $O(\log^* n)$-orientation with amortized time $O((\log^* n)!)$, and then sketch the ideas for improving the amortized time further to $O(\log^* n)$. This algorithm (for forests) already captures most of the high-level ideas that are needed for obtaining the ultimate result. In Sect. 2.3, we generalize this algorithm for arbitrary arboricity.

2.2 Handling forests. Consider a sequence of *cycle-free* graphs $\{G_t\}$. Initially, all edges are directed towards some arbitrary root (so that the maximum outdegree is 1) and the accounts of all vertices are empty. Every edge $\{u, v\}$ insertion is followed by a grant of x tokens, where x equals the amortized cost of the update operation. Newly introduced edges are directed arbitrarily. The outdegrees of the vertices increase, and the number of tokens stored in their accounts increases accordingly. This token aggregation continues up to a predefined outdegree threshold δ.

Consider an update operation which increases the outdegree of a vertex u above the outdegree threshold δ. In such a case, u initially examines its *local* neighborhood $\Gamma_h(u)$ of radius h (where h is a small parameter to be determined later), aiming towards finding a directed path P to a nearby vertex v with outdegree at most $\delta - 1$. Let $T(u)$ be the h-depth directed BFS tree rooted at u. If $T(u)$ has a low outdegree vertex v, then P is the unique $u \to v$ path from u to v in $T(u)$. In this case we invoke the `Path-Flip`(P) operation, thus reducing the outdegree of u back to δ at the expense of increasing the outdegree of v by 1, while the outdegrees of the internal vertices of P remain intact. In this case we say that $T(u)$ is *terminating*. To compensate for the increase in v's outdegree, we need to store sufficiently many tokens in v's account. We refer to this quantity as the *termination cost of u*.

We henceforth assume that no such vertex exists, thus all vertices in $T(u)$ have outdegree δ. In this case we invoke the `Tree-Reset`$(T(u), u)$ operation, which orients all edges of $T(u)$ towards the root u. As a result, the outdegree of all internal vertices of $T(u)$ decreases from δ to (at most) 1, yielding a dramatic improvement in their outdegrees. On the negative side, the outdegree of the leaves of $T(u)$ is *increased* from δ to $\delta + 1$, thus violating the outdegree threshold. We say that these leaves are *imbalanced*.

Since all vertices of $T(u)$ have outdegree δ before the reset and as the graph has no cycles, there is a good vertex expansion; in particular, it is easy to see that $T(u)$ contains $\Omega(\delta^{h-1})$ internal vertices. As the outdegree of each of these internal vertices decreased, the tokens aggregated in their accounts can be withdrawn, and used for balancing the newly imbalanced leaf set $L(u) = \{u_1, \ldots, u_k\}$ of $T(u)$.

The power of this reset operation stems from its ability to perfectly balance *exponentially* (in h) many vertices, hence releasing exponentially many tokens for

balancing the imbalanced leaf set $L(u)$ of $T(u)$. Next, we show how to fix the imbalanced vertices. We present a simple solution whose amortized cost is $O(\log\log n)$, and then carefully build on top of it to obtain the required $\log^* n$-style bounds.

Warm Up: 3-orientation in $O(\log\log n)$ amortized cost. Let the initial radius h be $\Theta(\log\log n)$ and set $\delta = 3$. As all internal vertices in $T(u)$ have outdegree 3, their number is $\Omega(\delta^{h-1}) = \Omega(\log n)$. To obtain $\Omega(\log n)$ tokens, all we need is to be able to withdraw a single token from each of these vertices' accounts. To this end, we store a single token at the account of a vertex whose outdegree increases from 2 to 3. Having the termination cost as small as 1 already suffices for maintaining the invariant that each vertex with outdegree 3 has a token at its account. (Note that the invariant does not guarantee any tokens for vertices with outdegree 2; we may henceforth assume that the vertices' outdegrees in the initial state are at most 2 rather than 1.)

Equipped with $\Omega(\log n)$ tokens, the next phase starts. The idea is to handle each of the imbalanced leaves $u_1, \ldots, u_k$ in the same way we handled u, but *in parallel*, with the only difference being that the local neighborhoods examined by these leaves are of (exponentially larger) radius $\Theta(\log n)$. Since the exploration radius is $\Theta(\log n)$, there must exist a path P_j from each leaf u_j to a vertex v_j of outdegree at most 1. We therefore invoke the `Path-Flip`(P_j) operation, for each $1 \le j \le k$, thus reducing the outdegree of u_j back to δ at the expense of increasing the outdegree of v_j by 1; since the outdegree of v_j increases to at most 2, this is done "for free", i.e., we do not need to deposit any tokens in v_j's account. It is crucial that the outdegree of v_j will be at most 1 (prior to increasing its outdegree), otherwise we would have to deposit a token in its account to compensate for increasing its outdegree; making such a deposit for all vertices v_j would be too costly. Thus, in order to terminate u_j, we must have a "2-unit decay" between the outdegrees of u_j and v_j. We will return to this subtle point in the sequel.

Towards $O(\log^* n)$ amortized cost. The simple argument above is based on the observation that $O(\log n)$ communication rounds suffice for finding a directed path from any imbalanced leaf to a vertex of outdegree at most 1. After flipping all edges along such a path (and doing so in parallel for all imbalanced leaves), all vertices will have low outdegree again, which completes the balancing procedure.

This approach leads to an amortized cost of $O(\log\log n)$. To achieve an amortized cost of $O(\log^* n)$, we must bound the initial radius $h_0 = h$ by $O(\log^* n)$. As a result, the number of vertices in the h_0-depth directed BFS tree $T(u)$ will be at most exponential in $O(\log^* n)$, so the overall number of tokens that we can withdraw from the accounts of these vertices cannot come close to $\log n$. Hence, balancing the imbalanced leaf set $L(u)$ of $T(u)$ requires a more intricate approach.

The improved balancing procedure consists of at most δ (rather than 2) phases, where each phase i requires $\Theta(h_i)$ communication rounds, for $h_0 = O(1)$ and $h_i = 2^{h_{i-1}}$ for every $i \ge 1$.

Phase 0 starts when u's outdegree increases to $\delta+1$ due to an edge insertion, and it becomes imbalanced. This insertion is granted with x tokens that are used for covering (1) the *communication cost* `CommCost`(0) of computing the h_0-depth directed BFS tree $T(u)$ as well as manipulating on it as described next, and (2) the *terminating cost* `TermCost`(0) of terminating the tree $T(u)$ (if needed). Specifically, if a low outdegree vertex v is found in $T(u)$, then `CommCost`(0) should include the cost of flipping all edges along the $u \to v$ path (via the `Path-Flip` operation), whereas `TermCost`(0) is just the cost of storing enough tokens at v's account to compensate for the increase in its outdegree; in this case the balancing procedure terminates. Otherwise, `CommCost`(0) should include the cost of resetting the tree $T(u)$ (via the `Tree-Reset` operation), which perfectly balances the outdegree of all $\Omega(\delta^{h_0-1})$ internal vertices (including u), thus releasing tokens from their accounts. The total number of released tokens, denoted `Gain`(0), depends on δ and h_0. These `Gain`(0) tokens are then used for balancing the resulting imbalanced leaves $u_1, \ldots, u_k$ of $T(u)$ in phase 1. (See Fig. 1.)

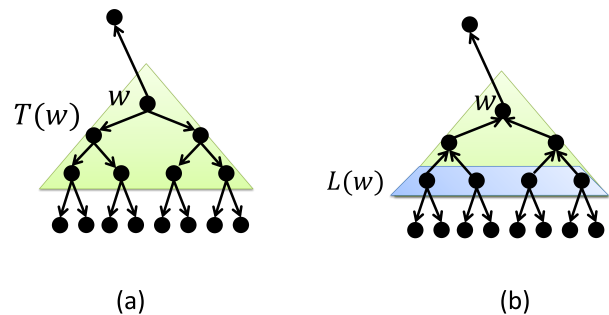


Figure 1: The Tree-Reset operation on $T(w)$. (a) The directed tree $T(w)$, with edges directed away from the root w. The edge connecting vertex w to its parent is not in $T(w)$, and hence is unaffected by the reset on $T(w)$. (b) The output of the reset operation on $T(w)$. The outdegree of all vertices in the leaf set $L(w)$ has increased by 1.

In phase 1, each of the imbalanced leaves u_j is handled in the same way we handled u, but *in parallel*.

The local neighborhoods examined by these leaves are of *exponentially larger* radius $h_1 = 2^{h_0}$. Consequently, the communication cost $\texttt{CommCost}(1)$ of phase 1 is set as $\Theta(h_1) = \Theta(2^{h_0})$. Let $T(u_1), \ldots, T(u_k)$ be the h_1-depth directed BFS trees rooted at the k imbalanced leaves $u_1, \ldots, u_k$. We terminate every tree $T(u_j)$ that contains a low outdegree vertex v_j by flipping the corresponding $u_j \to v_j$ path. The termination cost of this tree is the number of tokens needed to compensate for the increase in v_j's outdegree, and the sum of all such costs is the termination cost $\texttt{TermCost}(1)$ of phase 1.

If all trees $T(u_j)$ are terminating, then the balancing procedure terminates. So we henceforth assume that at least one of these trees $T(u_j)$ has no vertices of low outdegree, and reset every such non-terminating tree $T(u_j)$ (orienting all edges towards the root u_j). Such a reset will release exponentially (in $h_1 = 2^{h_0}$) many tokens, thus allowing us to explore neighborhoods of radius $h_2 = 2^{2^{h_0}}$ in the next phase. More specifically, let $\texttt{Gain}(1)$ be the total number of released tokens from all the non-terminating trees of phase 1. These $\texttt{Gain}(1) = \Omega(2^{2^{h_0}})$ tokens are next used for balancing the resulting imbalanced leaves of all the non-terminating trees in phase 2.

This balancing procedure is applied repeatedly, ending up with an exponentially increasing sequence of radius explorations $h_0, h_1 = 2^{h_0}, h_2 = 2^{2^{h_0}}, \ldots$. After at most $\log^* n - 1$ phases, we are guaranteed to either terminate at *all* explored trees or collect $\Omega(\log n)$ tokens that suffice to terminate all imbalanced leaves as described in the "Warm Up" section. Intuitively, this is why we get $\log^* n$-style bounds. Note also that the worst-case number of communication rounds required by the procedure is $O(\log n)$.

The communication costs of the balancing procedure are set in the obvious way, with $\texttt{CommCost}(0) = \Theta(h_0) = O(1)$, and $\texttt{CommCost}(i) = \Theta(h_i) = \Theta(2^{h_{i-1}})$, for $i \geq 1$. Determining the termination costs $\texttt{TermCost}(i)$, for $i \geq 0$, as well as the initial grant of tokens x (which dominates $\texttt{TermCost}(0)$) is more involved. The crux of the problem is to show that these parameters can be set so as to guarantee that the sum of costs $\sum_{i \geq 0} \texttt{CommCost}(i) + \texttt{TermCost}(i)$ of the balancing procedure does not exceed the initial grant of tokens x (that we can define as $\texttt{Gain}(-1)$) plus the overall number $\sum_{i \geq 0} \texttt{Gain}(i)$ of tokens released throughout all phases $i \geq 0$ of the procedure. To this end, it suffices to guarantee that for each $i \geq 0$,

$$\texttt{Gain}(i-1) \geq \texttt{CommCost}(i) + \texttt{TermCost}(i). \qquad (2.1)$$

Almost there: $O(\log^* n)$-orientation in amortized cost $O((\log^* n)!)$. In our amortization scheme, every vertex maintains an account of tokens in proportion to its current outdegree. For a vertex $w \in V$, let $B_i(w)$ be the number of tokens in w's account at the beginning of phase i, and let $\texttt{deg}_{out}(w, i)$ be its outdegree at the time. We show that the following invariant can be maintained throughout the balancing procedure. Consider the integer function $f(k) = k!$ if $k \geq 2$ and $f(k) = 0$ if $k \in \{0, 1\}$.

INVARIANT 1. *$B_i(w) \geq f(\texttt{deg}_{out}(w, i))$, for every vertex $w \in V$ and for every phase $i \geq 0$.*

To obtain outdegree $\delta = O(\log^* n)$ with amortized cost $O((\log^* n)!)$, the algorithm initially issues $x = \Theta(\delta!) = \Theta((\log^* n)!)$ new tokens, which can easily cover the communication and termination costs of phase 0. This proves the validity of Equation (2.1) for $i = 0$.

We henceforth assume that the initial tree $T(u)$ is non-terminating, and turn to proving the validity of Equation (2.1) for $i = 1$. Recall that upon a reset operation on $T(u)$, the outdegree of the internal vertices is reduced from δ to at most 1. Since $f(0) = f(1) = 0$, all the tokens in the accounts of these vertices can be released. Thus, each internal vertex of $T(u)$ releases $f(\delta) = \delta!$ tokens. We need to show that the overall number of released tokens, namely $\texttt{Gain}(0)$, is at least as large as $\texttt{CommCost}(1) + \texttt{TermCost}(1)$.

Consider the parents of the imbalanced leaves in $T(u)$. Each such parent distributes the $f(\delta)$ tokens it released uniformly among its δ imbalanced children. Thus, each leaf u_j will receive at least $f(\delta)/\delta = f(\delta - 1)$ tokens for terminating its subtree $T(u_j)$ in phase 1. (We later show that the total number of such tokens is at least $\texttt{TermCost}(1)$.) Note also that the number of tokens released by the remaining internal vertices of $T(u)$ is at least $\delta^{h_0 - 2} \cdot f(\delta)$, and obviously suffices to cover $\texttt{CommCost}(1) = \Theta(2^{h_0})$. Next, consider a terminating tree $T(u_j)$ with a low outdegree vertex v_j, and recall that the root u_j of $T(u_j)$ has a surplus of $f(\delta - 1)$ tokens. Upon flipping the edges of the $u_j \to v_j$ path, the outdegree of v_j increases by 1, say from δ' to $\delta' + 1$. To maintain the invariant, the number of tokens needed to be transferred to v_j's account, or the *termination cost* of $T(u_j)$, is thus $f(\delta' + 1) - f(\delta')$. Note that this term increases with the outdegree δ' of v_j. For our amortization scheme to work, we require the outdegree δ' of v_j to be smaller than the outdegree δ of the root u_j by at least two. This "2-unit decay" guarantees that the termination cost is bounded by $f(\delta' + 1) \leq f(\delta - 1)$, which can be covered by the surplus of u_j.

This proves the validity of Equation (2.1) for $i = 1$. The proof for larger values of i is similar, with a twist: As a by-product of the "2-unit decay" requirement, the outdegree threshold is prone to decrease by 1 at each phase of the procedure. Consequently, in a non-

terminating tree of phase i, all vertices should have outdegree at least $\delta_i - 1 = \delta - i - 1$, whereas a terminating tree of phase i should contain a vertex with outdegree at most $\delta_i - 2$. Let w be the root of such a terminating tree $T(w)$, and let y be a low outdegree vertex in $T(w)$. As before, w will receive a surplus of at least $f(\delta - i)/(\delta - i) = f(\delta - i - 1)$ from its parent in the tree, while the termination cost of $T(w)$ is at most $f(\delta - i - 1) - f(\delta - i - 2) \leq f(\delta - i - 1)$. This shows that $\texttt{Gain}(i-1)$ will cover $\texttt{TermCost}(i)$ in exactly the same way as before. In order for $\texttt{Gain}(i-1)$ to cover the cost of $\texttt{CommCost}(i) = \Theta(h_i)$, we must have a good vertex expansion in the trees explored at phase $i-1$. To this end, the outdegree threshold $\delta - i$ of phase $i-1$ should be at least 2. Since the procedure consists of at most $\log^* n$ phases, it suffices to set the initial outdegree threshold δ as $\log^* n + 2$. (See Fig. 2 for an illustration.)

$O(\log^* n)$-orientation in $O(\log^* n)$ amortized cost. In the above balancing procedure, when a reset is applied on a non-terminating tree $T(w)$, all the δ_y outgoing edges of any internal vertex y (except for the one leading to its original parent in $T(w)$) flipped their orientation and became incoming to y, thus its outdegree became 1. Although this operation leads to the release of $f(\delta_y) - f(1)$ tokens, these tokens had to be distributed evenly among the δ_y children of y. The key observation is that in order to improve the amortized cost to $O(\log^* n)$, it is preferable to flip the edges towards *two* (arbitrary) children and not more. In this way only $f(\delta_y) - f(\delta_y - 1)$ tokens are released, but distributing them among two children of y enables us to work with the much-slower growing function $f(k) = k^2 - 1$ (for $k \geq 1$). While the outdegree bound δ remains $O(\log^* n)$, the amortized cost will be linear in $f(\delta) - f(\delta - 1) = 2\delta - 1 = O(\log^* n)$ (details omitted).

2.3 General arboricity.

For simplicity, we assume that the vertices know the bound δ of the maximum allowed outdegree (i.e., for preserving δ-orientations).

The balancing procedure described in Sect. 2.2 is based on basic expansion properties of trees. In Sect. 2.3.1 we show that these properties generalize naturally for arbitrary arboricity. We extend the balancing procedure to general graphs in Sect. 2.3.2. This extension requires a graph-theoretic result (Thm. 2.1), whose proof is deferred to Sect. 2.3.3. Finally, in Sect. 2.3.4 we derive stronger results for the easy case of large arboricity.

2.3.1 Expansion in general graphs.

Let $G = (V, E)$ be a directed graph with arboricity $a \geq 1$, and let s be some vertex in V. Denote by V_i (resp., $\hat{V}_i$) the set of vertices reachable from s by directed paths of length at most (resp., exactly) i. Note that $\hat{V}_i = V_i \setminus V_{i-1}$.

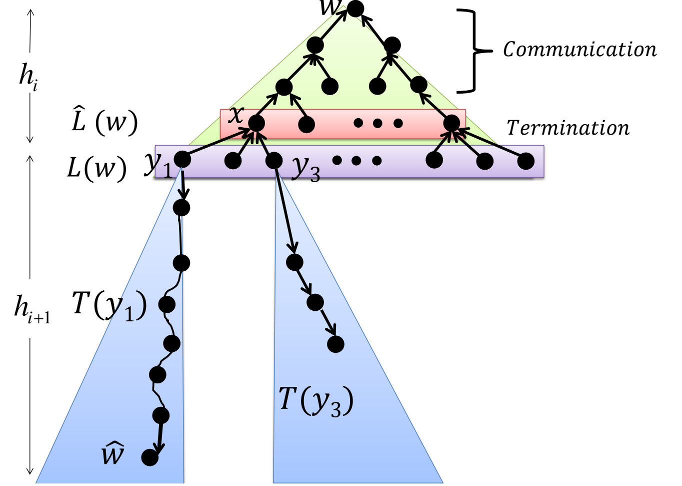


Figure 2: An illustration of a non-terminating tree $T(w)$ of an i-active vertex w. The tokens of the top $h_i - 2$ levels are used to cover the communication round complexity of phase $i+1$. The tokens of the last internal level $\widehat{L}(w)$ are used to cover the termination cost of every leaf vertex tree $T(w')$, $w' \in L(w)$. E.g., the vertex $x \in \widehat{L}(w)$ covers the termination cost of its of its children y_1, y_2, y_3.

CLAIM 1. *Assume all vertices of V_i have outdegree at least $d \geq 2a$. Then for $i \geq 1$, $|V_i| > \left(\frac{d}{a}\right) \cdot |V_{i-1}|$.*

Proof. Note that the total number of edges incident on vertices in V_{i-1} is at least $d \cdot |V_{i-1}|$, and all these edges are incident only on vertices of V_i. Since the graph's arboricity is at most a, it follows that $|V_i| - 1 \geq d \cdot |V_{i-1}|/a$. Hence $|V_i| > \frac{d}{a} \cdot |V_{i-1}|$. ∎

The next corollary shows that the expansion is good whenever the outdegree is large w.r.t. the arboricity.

COROLLARY 2.1. *If all vertices of V_i have outdegree at least $d \geq 2a$, then for $i \geq 1$: (1) $|V_i| > \left(\frac{d}{a}\right)^i \geq 2^i$. (2) $|\hat{V}_i| \geq (\frac{d}{a} - 1) \cdot |\hat{V}_{i-1}|$. (3) $|\hat{V}_i| > \left(\frac{d}{a} - 1\right)^i$.*

Proof. (1) Apply Claim 1 inductively, with $|V_0| = 1$. (2) Since $|\hat{V}_i| \geq |V_i| - |V_{i-1}| \geq (\frac{d}{a} - 1) \cdot |V_{i-1}| \geq (\frac{d}{a} - 1) \cdot |\hat{V}_{i-1}|$. (3) Apply item (2) inductively. ∎

Let T be a directed BFS tree of G rooted at s of depth h. By definition, V_i denotes the set of vertices at directed distance i from s in T (not just in G). Hence the vertex set of T is V_h; Corollary 2.1(1) implies that there are at least $\left(\frac{d}{a}\right)^i$ vertices in T. An internal vertex v in T is called *lonely* if it is the only child of its parent in the tree. (The root is non-lonely by definition.)

We next argue that the expansion remains good even if we disregard the lonely vertices of the tree. Denote by n_i the number of non-lonely vertices in V_i, for $i = 0, 1, \ldots, h$. By definition, we have $n_0 = 1$.

LEMMA 2.1. *For* $1 \leq i \leq h$, $n_i \geq (\frac{d}{a} - 2) \cdot (\frac{d}{a} - 1)^{i-1}$.

Proof. Fix any $i = 1, 2, \ldots, h$. By Corollary 2.1(2), $|\hat{V}_i| \geq (\frac{d}{a} - 1) \cdot |\hat{V}_{i-1}|$. By definition, any lonely vertex in $\hat{V}_i$ is the only child of some vertex in $\hat{V}_{i-1}$. Thus the number of lonely vertices in $\hat{V}_i$ is at most $|\hat{V}_{i-1}|$, and so the number n_i of non-lonely vertices in $\hat{V}_i$ is at least $|\hat{V}_i| - |\hat{V}_{i-1}| \geq (\frac{d}{a} - 2) \cdot |\hat{V}_{i-1}| \geq (\frac{d}{a} - 2) \cdot (\frac{d}{a} - 1)^{i-1}$. (The last inequality follows from Corollary 2.1(3).) ∎

Lem. 2.1 implies that

$$\sum_{i=0}^{h} n_i \geq 1 + \sum_{i=1}^{h} \left(\frac{d}{a} - 2\right) \cdot \left(\frac{d}{a} - 1\right)^{i-1} = \left(\frac{d}{a} - 1\right)^{h}. \quad (2.2)$$

Consider the tree T^* obtained from T by contracting all single-child paths. Note that no vertex in T^* is lonely, i.e., every internal vertex in T^* has at least two children. Moreover, there is a 1-1 correspondence between the vertices of T^* and the non-lonely vertices of T. By Eq. (2.2), T^* contains at least $(\frac{d}{a} - 1)^h$ vertices.

2.3.2 A generalized balancing procedure.

By setting the initial outdegree threshold to $\delta = \log^* n + 3a + 1$, the outdegree threshold in all phases of the procedure will be at least $3a + 1$. By Corollary 2.1, we should continue to have a good expansion. Nevertheless, in order to generalize our scheme for general graphs, there are a few delicate issues that need to be addressed.

Issue 1: The outdegrees in a BFS tree may be much smaller than the respective outdegrees in the graph. Phase 0 starts as before, by computing the h_0-depth directed BFS tree $T(u)$ rooted at u. Note that in order to terminate such a tree, we need to find a vertex whose outdegree in the graph (rather than the tree) is small. If such a vertex v is found, we terminate the tree as before. Otherwise, there is a good expansion. In this case we would like to reset the tree $T(u)$ as before (orienting all the edges towards the root u), in order to release exponentially many tokens for covering the communication and termination costs of the next phase. However, lower bounding the number of released tokens becomes more involved.

Consider some internal vertex y in $T(u)$, whose outdegree in G (resp., $T(u)$) is d (resp., d'). Upon the reset of $T(u)$ the outdegree of y is reduced to $d - d' + 1$, thus releasing $f(d) - f(d - d' + 1)$ tokens. These tokens are distributed uniformly among the d' children of y in $T(u)$, giving a surplus of $\frac{f(d) - f(d-d'+1)}{d'}$ tokens to each of them. However, if $d' = 1$, this surplus is zero. Thus, for our scheme to extend to general graphs, we must have $d' \geq 2$. To this end, we contract all single-child paths in $T(u)$, resulting in a smaller tree $T^*(u)$ where all internal vertices have outdegree at least 2. By Eq. (2.2), $T^*(u)$ has a good expansion. (The contracted vertices are disregarded only when *analyzing the number of tokens released* due to the reset of $T(u)$. These vertices participate in the reset operation as all other vertices, i.e., the former parent in $T(u)$ of such a vertex becomes its only outgoing neighbor after the reset.)

Issue 2: The BFS trees of phase i may overlap, $i \geq 1$. Consider the leaves $u_1, \ldots, u_k$ that become imbalanced at phase $i - 1$ of the balancing procedure, for $i \geq 1$. As before, we can compute at phase i of the procedure the h_i-depth directed BFS trees $T(u_1), \ldots, T(u_k)$ rooted at $u_1, \ldots, u_k$. In the cycle-free case, these trees are pairwise vertex-disjoint, so we could handle them *in parallel*. Indeed, terminating such a tree (flipping all edges along some $u_j \to v_j$ path) or resetting it (flipping all edges towards the root u_j) has no effect whatsoever on the other trees. If all these trees are terminating, all leaves $u_1, \ldots, u_k$ become balanced again and the balancing procedure terminates; otherwise, at least one of these trees has a good expansion, which provides us with enough tokens to cover the costs of the next phase.

In general graphs, these BFS trees may overlap, which complicates the parallel computation. The key observation is that in order to balance all k imbalanced leaves, it suffices to find k *edge-disjoint* (rather than vertex-disjoint) paths $u_1 \to v_1, \ldots, u_k \to v_k$ leading to k distinct low outdegree vertices $v_1, \ldots, v_k$. Indeed, flipping all edges along these paths will balance all vertices in the graph. But what if no such k edge-disjoint paths exist – must there be a good expansion then? We answer this question in the affirmative by proving the following graph-theoretic result.

THEOREM 2.1. (PROOF IN SECT. 2.3.3) *Let $G = (V, E)$ be any digraph with arboricity $a \geq 1$. Let $u_1, \ldots, u_k$ be arbitrary vertices in G with outdegree at least $3a + 1$, let $H = (V_H, E_H)$ be the union of the h-depth directed BFS trees $T(u_1), \ldots, T(u_k)$ rooted at these vertices, and let $\tilde{H} = G[V_H] = (V_H, \tilde{E})$ be the subgraph of G induced on the vertex set V_H. If there do not exist k edge-disjoint paths in $\tilde{H}$ from $u_1, \ldots, u_k$ to k distinct vertices $v_1, \ldots, v_k$ whose outdegree in G is at most $3a - 1$, then $|V_H \setminus Q| \geq 2^h$, where Q is the set of vertices whose outdegree in G is at most $3a - 1$.*

Since the initial outdegree threshold δ is $\log^* n + 3a + 1$, the outdegree threshold in all phases of the balancing procedure is at least $3a + 1$. By Theorem 2.1, if at some phase we cannot terminate at all the k imbalanced leaves (since there do not exist k edge-disjoint paths from them to k distinct low outdegree vertices), then we must have a good expansion, which should provide sufficiently many tokens to carry out the next phase. (Proving that $|V_H| \geq 2^h$ is insufficient, as only vertices of sufficiently large outdegree are guaranteed to have enough tokens in their accounts.) In this case, resetting just the largest among these BFS trees should release enough tokens for covering the costs of the next phase.

But how can we efficiently determine if at some phase there exist k edge-disjoint paths as above, and if such paths do exist, how can we compute them? The simplest way around this hurdle is to simply upcast the entire information regarding these BFS trees (more accurately, regarding $\tilde{H}$) to the global root u, let the root solve the problem locally (local computation is free), and finally broadcast the "solution" (i.e., the edges that need to be flipped) to the appropriate descendants (those needing to flip some of their incident edges). Since we consider the $\mathcal{LOCAL}$ model, we may send messages of unbounded size. The distance between the global root u to any vertex in these BFS trees is at most $\sum_{\ell=0}^{i} h_\ell < 2h_i$, where the inequality holds as the depth of the BFS trees grows exponentially at each phase. Consequently, these additional upcast and broadcast operations will increase the total communication costs by at most a constant factor.

Finally, we would like the worst-case update time of the balancing procedure to be bounded by $O(\log n)$. The key observation is that there is no need to consider BFS trees of depth larger than $O(\log n)$. Indeed, we cannot have good expansion with respect to such depth (as $2^{O(\log n)} > n$), hence Theorem 2.1 implies that we are guaranteed to be able to terminate the k imbalanced leaves using BFS trees of depth $O(\log n)$. Consequently, once $O(\log n)$ tokens are collected, we compute directed BFS trees of depth $O(\log n)$, handle the imbalanced leaves by terminating these trees as explained above, and terminate the execution of the balancing procedure.

This completes the high level description of our scheme for any arboricity, thus proving Theorem 1.2.

2.3.3 Proof of Theorem 2.1.

Let $\tilde{H} = G[V_H] = (V_H, \tilde{E})$ be the subgraph of G induced on the vertex set V_H, with $E_H \subseteq \tilde{E}$. We add a dummy vertex s and connect it via outgoing edges to the k vertices $u_1, \ldots, u_k$, and connect all vertices in Q (i.e., those of outdegree at most $3a-1$ in G) via outgoing edges to another dummy vertex t. Denote the resulting directed graph by $H' = (V'_H, E'_H)$, where $V'_H = V_H \cup \{s,t\}$ and $E'_H = \tilde{E} \cup \{(s,u_1), \ldots, (s,u_k)\} \cup \{(q,t) \mid q \in Q\}$.

The following observation is immediate.

Observation 2. *There exist k edge-disjoint paths in $\tilde{H}$ from $u_1, \ldots, u_k$ to k distinct vertices $v_1, \ldots, v_k$ whose outdegree in G is at most $3a-1$ iff there exist k edge-disjoint paths in H' from s to t.*

To prove Theorem 2.1, we assume there do not exist k edge-disjoint paths in H' from s to t, and show that $|V_H \setminus Q| \geq 2^h$. By Menger's Theorem, there exists an s-t cut C in H' of size at most $k-1$.

Consider the graph $H'_C = H' \setminus C$ obtained from H' after removing the cut edges. By definition, there is no $s \to t$ path in H'_C. By construction, s is connected in H' via outgoing edges to the k vertices $u_1, \ldots, u_k$. Denote by $z_1, \ldots, z_{k'}$ the outgoing neighbors of s in H'_C, and note that $\{z_1, \ldots, z_{k'}\} \subseteq \{u_1, \ldots, u_k\}$. Observe also that the remaining $k-k'$ vertices from $\{u_1, \ldots, u_k\}$ are incident on cut edges, implying that $k-k' \leq |C| \leq k-1$, and so $k' \geq 1$. Besides the $k-k'$ cut edges incident on s, there are $\tilde{k} = |C| - (k-k') \leq k'-1$ cut edges; it is possible to have $\tilde{k} = 0$. Denote by $\tilde{C}$ the set of cut edges that are not incident on s, and let $\tilde{U}$ denote the set of vertices in H' that have at least one outgoing edge in $\tilde{C}$; note that $|\tilde{U}| \leq \tilde{k} \leq k'-1$.

For $j = 0, 1 \ldots, h$, denote by V_j the set of vertices reachable from s in H'_C by directed paths of length at most $j+1$, disregarding s itself. We have $V_0 = \{z_1, \ldots, z_{k'}\}$, and so $|V_0| = k'$.

Observation 3. *All vertices of $V_j \setminus \tilde{U}$ have outdegree at least $3a$, for any $j \geq 0$.*

Proof. By definition, there is a directed path from s to all the vertices of V_j in H'_C. Since C is an s-t cut, no vertex in V_j may have an outgoing edge to t in H'_C. By construction, all vertices in V_j with outdegree at most $3a-1$ have an outgoing edge to t in H', hence they all must belong to $\tilde{U}$. ∎

Claim 2. $|V_1| > 3 \cdot k'$.

Proof. Recall that $V_0 = \{z_1, \ldots, z_{k'}\} \subseteq \{u_1, \ldots, u_k\}$, hence each z_i has outdegree at least $3a+1$. Thus the total number of outgoing edges leaving the vertices of V_0 is at least $k' \cdot (3a+1)$. Let E_1 denote the set of these edges, disregarding the at most $\tilde{k} \leq k'-1$ cut edges among them. We thus have $|E_1| \geq k' \cdot (3a+1) - (k'-1) > 3k' \cdot a$. Note that all edges of E_1 are incident on vertices of V_1. Moreover, neither s nor t belongs to V_1, implying that the graph $G_1 = (V_1, E_1)$ is a subgraph of the original graph. Hence its arboricity is at most a, and so $\frac{|E_1|}{|V_1|-1} \leq a$. It follows that $|V_1| > \frac{|E_1|}{a} > 3 \cdot k'$. ∎

LEMMA 2.2. *For each* $j = 1, \ldots, h$, $|V_j| > 3 \cdot 2^{j-1} \cdot k'$.

Proof. The proof is by induction on j. The basis $j = 1$ follows from Claim 2.
Induction Step: We assume the correctness of the statement for j, with $j \geq 1$, and prove it for $j+1$.
By the induction hypothesis,

$$\begin{aligned} |V_j \setminus \tilde{U}| &\geq |V_j| - |\tilde{U}| > 3 \cdot 2^{j-1} \cdot k' - \tilde{k} \\ &\geq 3 \cdot 2^{j-1} \cdot k' - (k'-1) > 2^j \cdot k'. \end{aligned}$$

By Observation 3, all vertices of $V_j \setminus \tilde{U}$ have outdegree at least $3a$. Let E_{j+1} denote the set of all outgoing edges leaving the vertices of $V_j \setminus \tilde{U}$, and note that no edge of E_{j+1} belongs to C. We have $|E_{j+1}| \geq |V_j \setminus \tilde{U}| \cdot 3a > (3 \cdot 2^j \cdot k') \cdot a$. By definition, all edges of E_{j+1} are incident on vertices of V_{j+1}. Moreover, neither s nor t belongs to V_{j+1}, implying that the graph $G_{j+1} = (V_{j+1}, E_{j+1})$ is a subgraph of the original graph. Hence its arboricity is at most a, and we have $\frac{|E_{j+1}|}{|V_{j+1}|-1} \leq a$. It follows that

$$|V_{j+1}| > \frac{|E_{j+1}|}{a} > \frac{(3 \cdot 2^j \cdot k') \cdot a}{a} = 3 \cdot 2^j \cdot k'. \quad \blacksquare$$

Lemma 2.2 implies that $V_h > 3 \cdot 2^{h-1} \cdot k'$. By Observation 3 and the fact that $k' \geq 1$, we have

$$\begin{aligned} |V_h \setminus Q| &\geq |V_h \setminus \tilde{U}| \geq |V_h| - |\tilde{U}| \\ &> 3 \cdot 2^{h-1} \cdot k' - (k'-1) > 2^h \cdot k' \geq 2^h. \end{aligned}$$

By construction, each vertex in V_h is within directed distance at most $h+1$ from s in H'_C, and so within directed distance at most h from some imbalanced vertex u_j in H. It follows that $V_h \subseteq V_H$. We thus have $|V_H \setminus Q| \geq |V_h \setminus Q| \geq 2^h$. Theorem 2.1 follows.

2.3.4 Easy case: Arboricity $a = \Omega(\log n)$.

Note that the static lower bound of [41, 9] becomes weaker as the arboricity grows, suggesting that the static problem might be easier for larger arboricity. We next argue that this is indeed the case for the dynamic problem: When $a = \Omega(\log n)$, one can maintain a δ-orientation with $O(1)$ amortized time, for $\delta = O(a) + \log n = O(a)$.

To see this, recall that every token pays for $O(1)$ communication rounds. Following every edge $\{u, v\}$ insertion, $O(1)$ new tokens are issued at u and v. Hence when the outdegree of a vertex u exceeds δ, its account should have at least $\Omega(\log n)$ tokens. We would like to use these tokens for applying the reset algorithm `BE`.

More concretely, let us set $\delta = 3a + \log n$, and show that the following invariant can be maintained: A vertex with outdegree $3a + i$ has $\Omega(i)$ tokens in its account.

When the outdegree of a vertex u exceeds δ, its account has at least $\Omega(\log n)$ tokens. As mentioned, in order to apply the `BE` reset algorithm we need to wake up all vertices in the graph. If the $O(\log n)$-depth BFS tree rooted at u contains all vertices in the graph, then we can wake up all of them within $O(\log n)$ communication rounds, and then apply the `BE` algorithm on the entire graph within $O(\log n)$ more rounds. Following this reset algorithm, all vertices in the graph will have outdegree at most $O(a)$, hence no tokens need to be stored in their accounts. In particular, all $\Omega(\log n)$ tokens that are stored in u's account can be withdrawn from its account and used to cover the cost of this update procedure.

However, it is possible that this $O(\log n)$-depth BFS tree does not contain all vertices in the graph, in which case we cannot perform a "complete reset" of the entire graph. The key observation is that there is no need for a complete reset – a "partial reset" will suffice. Consider the $O(\log n)$-depth directed BFS tree $T(u)$ rooted at u, and let $H(u)$ be the subgraph of G induced by the vertices of $T(u)$. We set $k = \log n$, and argue that there must exist k edge-disjoint paths in $H(u)$ from u to k distinct vertices $v_1, \ldots, v_k$ whose outdegree in G is at most $3a - 1$. Indeed, otherwise we can show that $|T(u)| \geq 2^{O(\log n)} > n$. The proof for this assertion follows similar lines as those in the proof of Theorem 2.1, and is thus omitted. By flipping the edges along all these paths, the outdegree of u is reduced by $k = \log n$ units, and so $O(\log n)$ tokens can be withdrawn from u's account to cover the cost of this update procedure. On the other hand, the outdegree of all vertices $v_1, \ldots, v_k$ will increase (by 1 unit) to at most $3a$, hence no tokens need to be stored in the accounts of these vertices.

How can we compute these k edge-disjoint paths? Perhaps the simplest approach is to upcast the entire information regarding the subgraph $H(u)$ to the global root u, let the root solve the problem locally (local computation is free), and finally broadcast the "solution" (i.e., the edges that need to be flipped) to the appropriate descendants (those needing to flip some of their incident edges). Recall that we consider the $\mathcal{LOCAL}$ model, hence we may send messages of unbounded size. The distance between u and any vertex of $H(u)$ is $O(\log n)$, hence the entire update procedure requires $O(\log n)$ communication rounds.

3 Applications

3.1 Dynamic forest-decomposition.

We show how to obtain a 2δ-forest-decomposition from a δ-orientation. Let $\overrightarrow{G}$ be a δ-orientation for G, Denote by i the ID of vertex $v_i \in G$. The at most δ outgoing neighbors of vertex v_i are divided into two classes: The first (resp., second) class consists of the outgoing neighbors whose ID's are smaller (resp., larger) than v_i. The edge (v_i, v_j) is mapped to forest $(j, 1)$ (resp., $(j, 2)$) if v_j

is in the first (resp., second) class, i.e., $\mathtt{ID}(v_i) > \mathtt{ID}(v_j)$ (resp. $\mathtt{ID}(v_i) < \mathtt{ID}(v_j)$). Overall, there are at most 2δ forests. Clearly, this mapping induces no cycles and can be computed (from scratch) in a single communication round. The first assertion of Thm. 1.3 thus follows.

In the centralized static setting, the time needed to compute this forest-decomposition out of the given δ-orientation is at least *linear* in the number of edges. However, in the centralized *dynamic* setting, if we have a forest-decomposition respecting the above mapping, updating it upon a single edge flip in the underlying edge-orientation triggers a single change of that edge (moving from one forest to another). A straightforward implementation of this mapping (using doubly linked lists) yields a constant *worst-case* update time. In other words, the above mapping shows that the dynamic forest-decomposition problem reduces (up to a factor of 2) to the dynamic edge-orientation problem (both in the centralized and the distributed settings). In particular, this resolves the question raised by Erickson [24].

3.2 Distributed dynamic adjacency labeling schemes. An adjacency labeling scheme assigns an (ideally short) label to each vertex, allowing one to infer if any two vertices u and v are neighbors directly from their labels. For an adjacency representation scheme to be useful, it should be capable of reflecting online the current up-to-date picture in a dynamic setting. Moreover, the algorithm for generating and revising the labels must be distributed, in contrast with sequential and centralized label assignment algorithms. Given an f-forest-decomposition for G, the label of each vertex v can be given by $\mathtt{Label}(v) = (\mathtt{ID}(v), \mathtt{ID}(w_1), \ldots, \mathtt{ID}(w_f))$ where w_i is the parent of v in the ith forest. Our dynamic $O(a(G) + \log^* n)$-forest-decomposition immediately implies the third assertion of Thm. 1.3.

3.3 Distributed coloring in dynamic graphs. By Thm. 1.1, given an $O(q \cdot a(G))$-orientation of a graph G, one can compute an $O(q \cdot a(G)^2)$-coloring in $O(\log^* n)$ additional rounds [9]. By Thm. 1.2, this immediately yields a dynamic maintenance of $O((a(G) + \log^* n)^2)$-coloring in $O(\log^* n)$ amortized time. However, this requires all the processors in the network to simultaneously wake up upon any update, which is impossible under the local wake-up limitation. As a result, our dynamic edge-orientation machinery cannot be applied as a black-box to maintain dynamic coloring. Instead, we tackle the dynamic coloring problem directly and, as a bonus, reduce the number of colors to $O(a(G) \cdot \log^* n)$.

Consider a legal δ-coloring $\varphi(G)$ of graph G, where $\varphi(v) \in \{1, \ldots, \delta\}$ denotes the color of vertex $v \in G$.[7] Newly inserted edges whose endpoints have the same color require to re-color one of their endpoints. The number of used colors increases and so does the number of tokens stored in their accounts. Similarly to before, in our log-starization scheme, every vertex maintains an account of tokens in proportion to its color. More specifically, we would like to maintain the invariant that the number of tokens stored at v's account is at least $f(\varphi(v))$, for some slowly-growing integer function f. However, for technical reasons, the new potential function f will not be strictly increasing as before, but rather monotone non-decreasing.

Specifically, we will use the potential function $f(k) = \lceil k/a' \rceil - 1$, where $a' = \Theta(a(G))$. Intuitively, we replace each original color $i = 1, 2, \ldots, \delta$ by a *color class* $C_i = \{(i-1) \cdot a' + 1, i \cdot a' + 2, \ldots, i \cdot a'\}$ of sufficiently large size a', thus viewing all colors in the same set as equivalent. (To this end, we increase the total number of colors by a factor of a'.) Observe that the potential function f maps all colors of the ith color class C_i to $i-1$, thus f is strictly monotone when restricted to colors from different color classes.

Setting $\delta = \Theta(\log^* n)$, we will show how to get $(\delta \cdot a')$-coloring within $O(\log^* n)$ amortized time.

Following an insertion of edge $\{u, v\}$ such that $\varphi(u) = \varphi(v)$, we need to re-color one among u and v. Without loss of generality we will re-color u. The update algorithm first re-colors u in an arbitrary color $\varphi'(u)$ from the last color class $C^{(\delta)}$; the old color $\varphi(u)$ might be from another color class, thus we store enough tokens (at most $f(\delta \cdot a') - f(1) \leq \delta = O(\log^* n)$) in u's account to compensate for this change. If $C^{(\delta)}$ contains at least one free color c for u, we re-color u with c and terminate. Otherwise all colors of $C^{(\delta)}$ are already occupied by the neighbors of u, and we apply the following re-coloring procedure. Similarly to before, we would like to explore the neighborhood of radius h around u. However, now we will consider the undirected (rather than directed) neighborhood, denoted here by $\Gamma_h(u)$. Moreover, we will not consider the entire graph G, but rather restrict ourselves to the subgraph $G^{(\delta)}$ of G induced by the vertices with color from $C^{(\delta)}$; denote by $\Gamma_h^{(\delta)}(u)$ the restricted neighborhood of u. If $\Gamma_h^{(\delta)}(u)$ contains exponentially (in h) many vertices, then the algorithm re-colors them using colors from the second-to-last color class $C^{(\delta-1)}$. Even though any graph with arboricity a can be legally colored with at most $2a$ colors, this re-coloring operation may cause conflicts at the boundaries of the neighborhood $\Gamma_h^{(\delta)}(u)$. Yet,

[7] In a legal δ-coloring, each vertex v is assigned a color $\varphi(v)$ from $\{1, \ldots, \delta\}$, so that $\varphi(u) \neq \varphi(v)$ will hold for each edge (u, v).

by the same reasoning used for the edge-orientation problem, one can draw exponentially many tokens from the accounts of the newly-colored vertices to explore neighborhoods of exponentially larger radius in the next phase of the procedure that aims to correct the conflicting coloring for the boundary vertices of $\Gamma_h^{(\delta)}(u)$).

We henceforth consider the complementary case when the neighborhood $\Gamma_h^{(\delta)}(u)$ is of sub-exponential (in h) size, and so it must contain many low degree vertices. In this case we show that all the vertices in $\Gamma_h^{(\delta)}(u)$ can be legally colored with colors from the last color class $C^{(\delta)}$, without causing any *internal* conflicts (between vertices in $\Gamma_h^{(\delta)}(u)$) or *external* conflicts (between the boundary of $\Gamma_h^{(\delta)}(u)$ and vertices in $G^{(\delta)} \setminus \Gamma_h^{(\delta)}(u)$). We begin by considering cycle-free graphs, and then extend the argument to general graphs.

3.3.1 Cycle-Free Graphs. In this section we consider the case where $G^{(\delta)}$ is a tree (or a forest), denoted by $T^{(\delta)}$. Let $V_h = \Gamma_h^{(\delta)}(u)$, and consider the subtree T_h of $T^{(\delta)}$ over the vertex set V_h. Note that T_h consists of the first h levels of $T^{(\delta)}$, or in other words, it contains all vertices of distance at most h from u in $T^{(\delta)}$.

Let $B = B_{h+1}$ be the set of neighbors in $T^{(\delta)}$ of the boundary vertices of T_h, i.e., B is the set of all vertices in $T^{(\delta)}$ that are at distance precisely $h+1$ from u. All vertices of B are *locked*, in the sense that we are not allowed to re-color them. Vertices of V_h are being marked as either *locked* or *free* in the following way. We traverse the vertices of T_h bottom-up, starting from the leaves, so that any internal vertex is traversed only after all its children have been traversed. Any leaf of T_h with at least two locked neighbors (in B) becomes locked, otherwise it is free. Similarly, any internal vertex of T_h with at least two locked children becomes locked, otherwise it is free.

If the root u is locked, it has at least two locked children, each having at least two locked children of its own. We have $|V_h| \geq 2^h$ by induction, implying that the h-depth neighborhood $V_h = \Gamma_h^{(\delta)}(u)$ around u is of exponential size. In this case, as mentioned above, re-coloring the vertices of V_h in colors from the second-to-last color class $C^{(\delta-1)}$ releases exponentially (in h) many tokens to carry out the next phase.

We henceforth assume that u is free. In this case, all locked vertices retain their original colors (all of which are from the last color class $C^{(\delta)}$). The vertices of T_h are then traversed top-down, starting from the root u, assigning each free vertex an arbitrary color from $C^{(\delta)}$ that is different than the colors of its (at most one) parent and its (at most one) locked child. Assuming the size of the color classes is at least 3, we will have at least one free color in $C^{(\delta)}$ to choose from. By this color assignment to free vertices, there cannot be any color conflicts between a free vertex and another (either free or locked) vertex. As all the locked vertices retain their original colors, the only potential conflict is between the root u and its new neighbor v. Since u is free, it follows that the new coloring is legal.

The above argument concerns phase 0, which resolves a conflict of a single vertex u, due to a single new neighbor v of u. In a general phase $i \geq 1$ we need to resolve the conflicts of all the (potentially many) leaves that became illegally-colored (or imbalanced) during phase $i-1$ of the procedure.

The key observation is that at the start of phase i, each of these imbalanced leaves u' has a conflict with a single vertex, namely, its parent $\pi(u')$ in $T^{(\delta)}$. Consequently, the same argument as above can be applied for each of the imbalanced leaves u'. If one of the leaves u' is locked, then its h_i-depth subtree is of exponential size, and re-coloring its vertices with colors from the preceding color class would release enough tokens to carry out the next phase. Otherwise, each leaf u' is free, and can thus choose a color (from the same color class) different than the colors of its parent $\pi(u')$ and its (at most one) locked child.

3.3.2 General Arboricity. Extending the above argument to general arboricity graphs requires several nontrivial adjustments. As before, let $V_h = \Gamma_h^{(\delta)}(u)$, and let G_h be the subgraph of $G^{(\delta)}$ induced by the vertex set V_h. Note that G_h contains all vertices of distance at most h from u in $G^{(\delta)}$. Let $B = B_{h+1}$ be the set of all vertices in $G^{(\delta)}$ that are at distance precisely $h+1$ from u; each such vertex has at least one neighbor in G_h. We refer to the vertices of B as the *boundary vertices*.

The process of marking the vertices of V_h as either locked or free becomes more involved. While in the cycle-free case we traversed the tree bottom-up, marking vertices with at least 2 children as locked, the idea here is to extend the notions of "bottom-up" and "children" to general graphs. We start by marking all the boundary vertices (i.e., those in B) as locked, and placing them in some queue $\mathcal{Q}$. At each step, we extract an arbitrary vertex v from $\mathcal{Q}$, scan all its unmarked neighbors in V_h, and orient all the edges between v and these vertices into v. Once the out-degree of a vertex becomes at least $2a(G)$ (which is the general arboricity analogue of having at least 2 locked children), we mark it as locked and place it in $\mathcal{Q}$. We repeat this process until the queue $\mathcal{Q}$ is empty.

It is immediate that all the vertices that entered the queue at some point are marked as locked; all the remaining vertices of V_h are marked as free. We will use

the following observation in the sequel.

Observation 4.

- *Any free vertex has no incoming neighbors (as we oriented edges only towards vertices that were in the queue, and were thus locked), and at most $2a(G) - 1$ outgoing neighbors.*
- *Any locked vertex has at least $2a(G)$ outgoing neighbors, all of which are locked by the first item.*

All locked vertices will retain their original colors (all of which are from the last color class $C^{(\delta)}$). Thus, among the locked vertices, the only potential conflict is between the root u and its new neighbor v (assuming both u and v are locked).

Next, we assign colors to all the free vertices in a manner that causes no conflicts whatsoever (that is, neither between pairs of free vertices nor between pairs of free and locked vertices). While in the cycle-free case we traversed the tree top-down, assigning each free vertex a color that is different than its parent and its at most one locked child, the idea here is to extend the notions of "top-down" and "parent" to general graphs. (Although the notions "bottom-up" and "children" were already extended above in a similar context, the notions "top-down" and "parent" in the current context are not symmetric to them.)

We remove from G_h all the locked vertices and their incident edges, and consider the new graph $G_{free} = (V_{free}, E_{free})$ induced by the free vertices V_{free}. Since the arboricity of G_{free} is no greater than the arboricity $a(G)$ of the entire graph G, we can partition the vertices of G_{free} into at most $\ell = \lfloor \log |V_{free}| \rfloor$ subsets $H_1, ..., H_\ell$, so that every vertex $v \in H_i$, $i \in \{1, 2, \ldots, \ell\}$, will have at most $4a(G)$ neighbors from $\bigcup_{j=i}^{\ell} H_i$. (See [9] for more details.) Next, we traverse the ℓ subsets one after another, starting from H_ℓ and finishing at H_1, i.e., for each $i = \ell, \ldots, 1$, we traverse the vertices of H_i only after traversing the vertices of the subsets $H_{i+1}, \ldots, H_\ell$. For each vertex $v \in H_i$, $i = \ell, \ldots, 1$, we choose a color for v from $C^{(\delta)}$ that is different than the colors assigned to its at most $4a(G)$ neighbors in $\bigcup_{j=i}^{\ell} H_i$ and its at most $2a(G) - 1$ locked neighbors (all of which are outgoing of it). Assuming the size of the color classes is at least $6a(G)$, we will have at least one free color in $C^{(\delta)}$ to choose from.

By the description of the above coloring procedure, for a pair u, v of vertices to have the same color, both of them must be locked. However, since all locked vertices retain their original colors, the only potential conflict in colors is between the root u and its new neighbor v. In particular, in order to have an illegal coloring, the root u must be locked.

Assuming u is locked, we show that there is a good expansion. Since the root u is locked, it has at least $2a(G)$ locked outgoing neighbors. Denote by U_1 this set of locked neighbors, and generally, denote by U_i the set of locked vertices that are within directed distance i from u. Recall that the undirected distance between u and any vertex in B is $h + 1$. We thus have:

Observation 5. *For each $i = 1, 2, \ldots, h$, all vertices in U_i belong to V_h.*

Claim 3. *For each $i = 1, 2, \ldots, h$, $|U_i| \geq 2^i$.*

Proof. The proof is by induction on i, the basis $i = 1$ is trivial. We assume the validity of the statement for $i - 1$, i.e., $|U_{i-1}| \geq 2^{i-1}$, and prove it for i. Let E_i be the set of outgoing edges incident on all vertices in U_{i-1}. By definition, all these edges lead to locked vertices in V_h. Hence, the edges of E_i are incident on the vertices of U_i. Since each vertex of U_{i-1} has out-degree at least $2a(G)$, it follows that $|E_i| \geq |U_{i-1}| \cdot 2a(G) \geq 2^i a(G)$. As $\frac{|E_i|}{|U_i|-1} \leq a(G)$, we have $|U_i| - 1 \geq \frac{|E_i|}{a(G)} \geq 2^i$. ∎

It follows that $|V_h| \geq |U_h| \geq 2^h$, implying that the h-depth neighborhood $V_h = \Gamma_h^{(\delta)}(u)$ around u is of exponential size. In this case, as mentioned above, re-coloring the vertices of V_h in colors from the second-to-last color class $C^{(\delta-1)}$ releases exponentially (in h) many tokens to carry out the next phase.

The above argument concerns phase 0, which resolves a conflict of a single vertex u, due to a single new neighbor v of u. In a general phase $i \geq 1$ we need to resolve the conflicts of all the (potentially many) vertices that became illegally-colored (or imbalanced) during phase $i - 1$ of the procedure.

In contrast to the case of cycle-free graphs, we can no longer argue that each of these imbalanced vertices u' has a conflict with a single vertex. Indeed, each imbalanced vertex u' may have conflicts with *many* neighbors in $G^{(\delta)}$, due to colors assigned to these neighbors of u' during the previous phase $i - 1$. The key observation is that all these neighbors of u' belong to the neighborhood around u of radius h_i computed at phase i, and moreover, they are all at distance at least h_i from any vertex in B_{h_i+1} (where B_{h_i+1} is the set of vertices at distance precisely $h_i + 1$ from u). If u' and all its conflicting neighbors are free, they can be assigned new non-conflicting colors (from the same color class) as shown above. Otherwise, at least one of these vertices is locked, in which case we must have a good expansion; that is, in this case the neighborhood around u is of exponential size (specifically, at least 2^{h_i-1} rather than 2^{h_i}) by exactly the same argument as before. Consequently, re-coloring the vertices in this neighborhood with colors from the preceding color class would release enough tokens to carry out the next phase.

3.4 Dynamic load balancing in expander graphs. Finally, we show that our approach is efficient also in a different class of graphs, β-expanders. In the dynamic load balancing problem, we seek to keep the job load roughly evenly distributed among the processors of a given network (regardless of its arboricity). Intuitively, the number of jobs a processor has can be viewed as the outdegree of this vertex. To reduce the number of jobs a processor has, we simply find an *undirected* path to a low load processor. While in the edge-orientation problem we had to argue in some cases that there is expansion in a given subgraph, here the expansion is given by definition. The vertex expansion $\Psi(G)$ of an n-vertex graph $G = (V, E)$ is defined as $\Psi(G) = \min\{|\Gamma(S) \setminus S|/|S| \mid S \subset V, |S| \leq n/2\}$. For $\beta > 1$, a graph $G = (V, E)$ is β-expander if $\Psi(G) \geq \beta$.

The insertion and deletion of jobs as well as the topological updates of the graph in hand are modified by an adversary restricted in its power. The dynamic of the system introduced in an adversarial manner is at two levels: (1) the topology of the graph, where in each round t, a subset of edges may be added or removed. In-addition, vertices along with some incident edges may join or leave the network and (2) the number of jobs, where in each round a new job may be added or removed from one of the vertices of the network. Let G_t be the n_t-vertex graph at phase t and let J_t denote the total jobs assigned to the vertices of G_t. The adversary is restricted in two senses. First, at any time phase t, the communication graph G_t is β-expander for $\beta > 1$ and second, the average job load J_t/n_t is bounded by some $\widehat{\delta}$ for every $t \geq 1$. This latter assumption is not crucial and only added for simplification of presentation. In addition, for simplicity, also assume that the vertices know $\widehat{\delta}$ and have some bound on the number of vertices. This assumption again is not crucial and can be avoided with some extra work. Let $\delta = \Theta(\widehat{\delta} + \log^*_\beta n)$. Our work plan to maintain a load of δ jobs at each vertex is as follows. Every job insertion is followed by a grant of $I_0 = \Theta(\log^*_\beta n)$ tokens which is the amortized cost of the update operation. As jobs are introduced, the load on the vertices increases and so does the number of tokens stored in their accounts. This token aggregation continues up to a predefined level of maximum allowed load δ. When a vertex exceeds the allowed load level δ, it initiates a reset cascade applied on a exponentially growing radii of neighborhoods. A reset operation applied on a subgraph H of the network re-balances the vertices by removing exactly one job from each vertex. These surplus jobs accumulate, until sufficiently many low-load vertices are discovered and the surplus jobs are then distributed among these vertices. The benefit of the reset appears when it is applied on a subgraph containing many vertices whose load is close δ. After the reset, many tokens can now be withdrawn from the accounts of these vertices and be used to cover the further exploration for low-load vertices. The following notation is useful. Consider phase t, where a new job is assigned to vertex u and u becomes overloaded with $\delta+1$ jobs. We treat the current graph in phase t as a triple $G^t = (V_t, E_t, \mathcal{J}_t)$ where $\mathcal{J}_t : V \to \mathbb{N}$ is the load-function assigning vertex v, $\mathcal{J}_t(v)$ jobs. Let $J_t = \sum_{v \in V_t} \mathcal{J}_t(v)$ be the total number of jobs at phase t. The algorithm consists of at most $\widehat{\delta} - \delta + 1$ phases and each phase $i \in \{0, \ldots, \widehat{\delta} - \delta + 1\}$ consists of $\Theta(h_i)$ communication rounds where

$$h_0 = 4, \quad \text{and } h_i = \beta^{h_{i-1}} \quad \text{for } i \geq 1. \tag{3.3}$$

In each phase, $\imath \geq 0$ the vertex u constructs $T_i(u)$, a BFS tree rooted at u at depth h_i. Let $T_{-1}(u) = \{u\}$ and $T_{-2}(u) = \{\emptyset\}$. At the end of each phase, the root u is notified of this fact, and it then initiates the start of the next phase.

A vertex w is *i-light* if its load (i.e., number of jobs it holds) is $\mathcal{J}_t(w) \leq \delta - i - 1$, otherwise it is *$i$-heavy*. Let n_i be the number of vertices explored at the beginning of phase i, i.e., the number of vertices in $T_{i-1}(u) \setminus T_{i-2}(u)$, hence $n_0 = 1$. For $i \geq 0$, define H_i as the number of heavy vertices in $T_{i-1}(u)$ with at least $\delta - i + 1$ jobs. Let $\widehat{H}_i = \sum_{j=0}^{i} H_j$ be the total number of vertices with at least $\delta - j + 1$ tokens in $T_{j-1}(u) \setminus T_{j-2}(u)$ for $j \in \{0, \ldots, i\}$.

We now describe phase $i \in \{0, \delta - \widehat{\delta} - 1\}$ of Procedure $\mathsf{UpdateLB}$ for vertex u. Recall that initially, $n_0 = \widehat{H}_0 = H_0 = 1$. First, consider the case where $n_i \leq n/4$. The vertex u constructs a partial BFS tree $T_i(u)$ of depth h_i. It then computes (via convergcast) the number ℓ_i of i-light vertices in $T_i(u) \setminus T_{i-1}(u)$. If these i-light vertices are fewer than the total number of heavy vertices seen so far, i.e., $\ell_i < \widehat{H}_i$, then u computes the number H_{i+1} of i-heavy vertices in $\Gamma^+_{h_i}(u)$ and set $\widehat{H}_{i+1} = \widehat{H}_i + H_{i+1}$. Else, (i.e., $\ell_i \geq \widehat{H}_i$), the vertex u initiates the termination procedure for phase i described next. Each of the $\widehat{H}_i$ heavy vertices shuttles one of its surplus jobs towards u in a pipelined manner on $T_i(u)$. In addition, every vertex $z \in T_i(u)$ computes (via convergcast) the number of i-light vertices of $\Gamma^+_{h_i}(u)$ appearing in its subtree $T_i(z) \subseteq T_i(u)$. Using this counter information, the total of $\widehat{H}_i$ surplus tokens accumulated at u can be routed from u to the $\widehat{H}_i$ i-light vertices on $T_i(u)$ in a pipelined manner: each vertex that receives a job from its parent on $T_i(u)$ reduces its counter by 1. If it is i-light (currently is has less than $\delta - i - 1$ jobs), it keeps the job (and update its load-function) and otherwise it sends it to its left-most child on $T_i(u)$ with a non-zero

counter.

Finally, consider the case where $n_i \geq n/4$ or $i = \widehat{\delta} - \delta$. In this case, a BFS tree $T_i(u)$ of full depth rooted at u is constructed. In this context, a vertex is light if it has less than $\widehat{\delta}$ jobs. For every light vertex z, let $W(z) = \widehat{\delta} - \mathcal{J}_t(z)$. Let V' be the total number of light vertices. Since $\widehat{\delta}$ is the average number of jobs per vertex, it holds that $\sum_{z \in V'} W(z) \geq H_i$. Implying that the light vertices can absorb the H_i surplus jobs without reaching the capacity of $\widehat{\delta}$. Every vertex $v \in T_i(u)$ then store the sum of $W(z)$ corresponding to the number of jobs the light vertices in its subtree $T_i(z) \subseteq T_i(u)$ can absorb. Each of the $\widehat{H}_i$ heavy vertices sends one of its surplus job towards the root u in a pipelined manner. The root u then shuttles these $\widehat{H}_i$ jobs in a pipelined manner using the counter information $W(z)$ in an equivalent manner to the termination phase described above.

Analysis. The correctness is immediate, hence we proceed with the amortized analysis. Every job insertion is followed by a grant of $I_0 = \Theta(\log^*_\beta n)$ tokens. These tokens are accumulated in the bank account of the vertices. Let $B_t(u)$ be the bank account of u at the beginning of phase t and let $|B_t(u)|$ be the number of tokens in this account. Recall that $\mathcal{J}_t(u)$ is the number of jobs assigned to u at phase t. Consider the integer function

$$(3.4) \qquad f(k) = \begin{cases} c \cdot (k - \widehat{\delta})^2, & \text{if } k \geq \widehat{\delta} + 1; \\ 0, & \text{if } k \in \{0, 1, \dots, \widehat{\delta}\}, \end{cases}$$

where $c = \Theta(1)$. We will show that the following invariant is maintained for every phase t.

Invariant 2. $|B_t(z)| \geq f(\mathcal{J}_t(z))$, *for every vertex* $u \in G_t, t \geq 1$.

We assume that the invariant holds at the beginning of phase t and show that it also holds at the end of phase t. Our goal is to show that the tokens accumulated in the accounts $B_t(u)$ are sufficient to cover the communication costs of Proc. UpdateLB as well as maintaining the invariant for the new job assignment $\mathcal{J}_{t+1}$.

Let i^* be the last phase of Proc. UpdateLB. We first begin with the simplified case where $i^* = 0$. Then, the load of u was decreased by 1 and the load of some 0-light vertex z in $T_0(u)$ was increased by 1. Since the invariant holds at the beginning of phase t, u has at least $|B_t(u)| \geq f(\mathcal{J}_t(u)) + I_0 \geq f(\delta + 1)$ where I_0 is the grant added to u due to the addition of the new job. Since u has $\delta + 1$ jobs at the beginning of the phase, at least $f(\delta + 1) - f(\delta)$ tokens can be removed from its account. The cost of the phase is $\Theta(h_0) = \Theta(1)$ communication rounds and in addition adding at most $f(\delta) - f(\delta - 1)$ tokens are needed to be added to the account of z so that the invariant is maintained. By setting the constants correctly, by Eq. (3.4), we get that $f(\delta+1) - f(\delta) \geq f(\delta) - f(\delta-1) + \Theta(1)$, as desired.

So from now on, we consider the case where $i^* \geq 1$. First consider the case where the termination procedure applied since there were sufficiently many i^*-light vertices, i.e., $\ell_{i^*} \geq \widehat{H}_{i^*}$. In this context only, call a vertex z heavy if it is has at least $\delta - i^* + 1$ jobs (i.e., $\mathcal{J}_t(z) \geq \delta - i^* + 1$). By applying the termination procedure, each of the $\widehat{H}_{i^*}$ heavy vertices z releases one surplus jobs. This implies that at least $f(\delta - i^* + 1) - f(\delta - i^*)$ tokens can be released from each heavy vertex z while maintaining the invariant that $|B_{t+1}(z)| \geq f(\mathcal{J}_{t+1}(z))$. Hence, overall at least $N_{out} = \widehat{H}_{i^*} \cdot (f(\delta - i^* + 1) - f(\delta - i^*))$ tokens can be released from the heavy vertices up to phase i^*. We now show that these tokens are sufficient to balance the accounts of the i^*-light vertices that absorb the surplus tokens of the $\widehat{H}_{i^*}$ heavy vertices and in addition cover the overall communication costs of Proc. UpdateLB.

Every i^*-light vertex v has at most $\mathcal{J}_t(v) \leq \delta - i^* - 1$ jobs. After the termination procedure, its load is increased by 1, i.e., $\mathcal{J}_{t+1}(v) = \mathcal{J}_t(v) + 1$. Since the invariant holds at the beginning of phase t, at most $f(\delta - i^*) - f(\delta - i^* - 1)$ tokens are needed to be added to each of the H_{i^*} i^*-light vertices so that the invariant holds for these vertices. i.e., $|B_{t+1}(z)| \geq f(\mathcal{J}_{t+1}(z))$. So, overall the termination cost of phase i^* is bounded by

$$(3.5) \quad \texttt{TermCost}(i^*) = H_{i^*} \cdot (f(\delta - i^*) - f(\delta - i^* - 1)).$$

We now turn to bound the total number communication rounds. In each phase $j \in \{0, \dots, i^*\}$, a BFS tree of depth h_j is constructed and the number of j-light vertices and j-heavy vertices are computed. This can be done in $\Theta(h_j)$ rounds via standard convergcast procedure. Hence, overall it cost $\Theta(\sum_{j=0}^{i^*} h_j) = \Theta(h_{i^*})$ rounds, where the last equality follows by Eq. (3.3). In the termination procedure, H_{i^*} jobs are transferred via pipeline to the root u on $T_{i^*}(u)$ and later on transferred in a pipelined manner again from u to the appropriate i-light vertices. Since the diameter of $T_{i^*}(u)$ is h_{i^*}, the total communication costs of these operations is $\Theta(h_{i^*} + \widehat{H}_{i^*})$ rounds. So, overall the communication cost is bounded by

$$(3.6) \qquad \texttt{CommCost}(i^*) = c' \cdot (h_{i^*} + \widehat{H}_{i^*})$$

for some constant $c' \geq 1$. We now claim that $N_{out} \geq \texttt{CommCost}(i^*) + \texttt{TermCost}(i^*)$ and begin by bounding $\widehat{H}_{i^*}$.

CLAIM 4. *For every* $1 \leq i < i^*$: *(a)* $|T_i(u)| = \Theta(\beta^{h_i})$, *(b)* $|T_i(u) \setminus T_{i-1}(u)| = \Theta(\beta^{h_i})$, *(c)* $H_{i+1} = \Theta(\beta^{h_i}) = \Theta(h_{i+1})$.

Proof. Part (a) follows by the definition of β-expander and recalling that n_i (the number of vertices in $T_{i-1}(u)$) is less then $n/4$ and hence the expansion is guaranteed. By Part (a), $T_{i-1}(u) = \Theta(\beta^{h_{i-1}})$, hence Part (b) holds by Eq. (3.3). Finally to see (c), recall that since $i^* \neq i$, it implies that $|T_i(u) \setminus T_{i-1}(u)|$ contains less than $|T_{i-1}(u)|$ i-light vertices and hence by Part (a) and (b), at least constant fraction of the vertices in $|T_i(u) \setminus T_{i-1}(u)|$ are i-heavy. Part (c) follows.

By Cl. 4(c) and Equations (3.5) and (3.6),

$$\begin{aligned}
N_{out} &= \widehat{H}_{i^*} \cdot (f(\delta - i^* + 1) - f(\delta - i^*)) \\
&= \widehat{H}_{i^*} \cdot (f(\delta - i^*) - f(\delta - i^* - 1)) + 2c \cdot \widehat{H}_{i^*} \\
&= \texttt{TermCost}(i^*) + c \cdot h_{i^*} + c \cdot \widehat{H}_{i^*} \\
&= \texttt{TermCost}(i^*) + \texttt{CommCost}(i^*) \ .
\end{aligned}$$

Finally, it remains to consider the case where in phase i^* the termination is applied since there are $n_{i^*} \geq n/4$ vertices in $T_{i^*-1}(u)$. (Note that when $i^* = \delta - \widehat{\delta} - 1 = \Theta(\log^*_\beta n)$, by the definition of h_{i^*}, it holds that $h_{i^*} = \Theta(n)$ and since the diameter of G_t is $O(\log_\beta(n))$, by that time $n_{i^*} \geq n/4$ as well). Note that by the proof Cl. 4, $\widehat{H}_{i^*} = \Theta(n)$. Each of the $\widehat{H}_{i^*}$ heavy vertices z has at least $\mathcal{J}_t(z) \geq \delta - i^* > \widehat{\delta}$ jobs (since $i^* \leq \delta - \widehat{\delta} - 1$) and after the termination procedure it has one less job, $\mathcal{J}_{t+1}(z) = \mathcal{J}_t(z) - 1$. By the invariant for the beginning of phase t, it holds that $B_t(z) \geq f(\mathcal{J}_t(z))$. Hence, at least $f(\delta - i^*) - f(\delta - i^* - 1)$ tokens can be withdrawn from the account each heavy vertex while maintaining that $B_{t+1}(z) \geq f(\mathcal{J}_{t+1}(z))$. Overall, at least $N_{out} = H_{i^*} \cdot (f(\delta - i^*) - f(\delta - i^* - 1))$ tokens can be withdrawn from the H_{i^*} bank accounts. We now show that these tokens are sufficient to cover the communication costs and to maintain the invariant. The total communication costs is $\texttt{CommCost}(i^*) = O(H_{i^*} + \log_\beta n)$ since the diameter of every β-expander is $O(\log_\beta n)$ and H_{i^*} jobs are transferred via pipeline to u on the full depth BFS tree $T_{i^*}(u)$. By setting the constants correctly, we get that $N_{out} \geq \texttt{CommCost}(i^*)$. Since the load of the light vertices is increased up to the threshold $\widehat{\delta}$, by the definition of potential function, Eq. (3.4), the bank accounts of these vertices are allowed to be empty, and hence the invariant holds vacuously for this vertices. This completes the proof of Thm. 1.4.

4 Discussion and future research

This paper does not address one particular task, but rather aims at presenting a general dynamic maintenance paradigm in distributed networks, which applies to a plethora of load-balancing tasks. Whereas almost all previous works in this context focused on optimizing the *worst-case* update time, thus incurring relatively high update time bounds, our paradigm gives rise to algorithms with nearly constant *amortized* update time.

There are a few natural directions for future research. First, it would be interesting to expand the class of local-on-average tasks. Are there distributed tasks (allowing some slack) that are *not* local-on-average? What are the conditions a task should satisfy in order to be local-on average? Our framework is deterministic. It is natural to ask whether randomization helps and if so, then to what extent. Finally, a major goal of interest would be to develop a general transformation tool, *log-starization amortizer*, that receives as input a static (non-local) $O(\text{polylog}(n))$-round algorithm for *solving* a given task (from scratch), and transforms it into a dynamic algorithm of $O(\log^* n)$ rounds *on average*.

References

[1] C. N. A. Cornejo, S. Gilbert. Aggregation in dynamic networks. In *Proc. PODC*, pages 195–204, 2012.

[2] Y. Afek, B. Awerbuch, and E. Gafni. Applying static network protocols to dynamic networks. In *Proc. FOCS*, pages 358–370, 1987.

[3] Y. Afek, M. Ricklin, and E. Gafni. Upper and lower bounds for routing schemes in dynamic networks. In *Proc. FOCS*, pages 370–375, 1989.

[4] N. Alon, R. Yuster, and U. Zwick. Color-coding. *J. ACM*, 42(4):844–856, 1995.

[5] E. Anshelevich, D. Kempe, and J. Kleinberg. Stability of load balancing algorithms in dynamic adversarial systems. In *Proc. STOC*, pages 399–406, 2002.

[6] B. Awerbuch, B. Patt-Shamir, D. Peleg, and M. E. Saks. Adapting to asynchronous dynamic networks. In *Proc. STOC*, pages 557–570, 1992.

[7] B. Awerbuch and D. Peleg. Network synchronization with polylogarithmic overhead. In *Proc. FOCS*, pages 514–522, 1990.

[8] B. Awerbuch and M. Sipser. Dynamic networks are as fast as static networks. In *FOCS*, pages 206–219, 1988.

[9] L. Barenboim and M. Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.

[10] S. Baswana, K. Sumeet, and S. Soumojit. Fully dynamic randomized algorithms for graph spanners. *ACM Trans. Alg.*, 8(4), 2012.

[11] P. Berenbrink, T. Friedetzky, and R. A. Martin: Dynamic diffusion load balancing. In *Proc. ICALP*, pages 1386–1398, 2005.

[12] G. S. Brodal and R. Fagerberg. Dynamic representation of sparse graphs. In *WADS*, pages 342–351, 1999.

[13] M. L. M. R. C. Gőmez-Calzado, A. Lafuente. Fault-

tolerant leader election in mobile dynamic distributed systems. In *Proc. PRDC*, pages 78–87, 2013.

[14] J. A. Cain, P. Sanders, and N. C. Wormald. The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In *Proc. SODA*, pages 469–476, 2007.

[15] K. Censor-Hillel, E. Haramaty, and Z. Karnin. Optimal Dynamic Distributed MIS. In *CoRR abs/1507.04330*, 2015.

[16] M. Chrobak and D. Eppstein. Planar orientations with low out-degree and compaction of adjacency matrices. *Theor. Comput. Sci.*, 86(2):243–266, 1991.

[17] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, Boston, 2001.

[18] S. Dolev, R. Segala, and A. A. Shvartsman. Dynamic load balancing with group communication. *Theor. Comput. Sci.*, 369(1-3):348–360, 2006.

[19] Z. Dvorak and V. Tuma. A dynamic data structure for counting subgraphs in sparse graphs. In *Proc. WADS*, pages 304–315, 2013.

[20] D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, 12:229–243, 1986.

[21] D. Eisenstat, P. N. Klein, and C. Mathieu. An efficient polynomial-time approximation scheme for steiner forest in planar graphs. In *Proc. SODA*, pages 626–638, 2012.

[22] M. Elkin. A near-optimal distributed fully dynamic algorithm for maintaining sparse spanners. In *Proc. PODC*, pages 185–194, 2007.

[23] D. Eppstein. All maximal independent sets and dynamic dominance for sparse graphs. *ACM Trans. Alg.*, 5(4), 2009.

[24] J. Erickson. `http://www.cs.uiuc.edu/~jeffe/teaching/datastructures/2006/problems/Bill-arboricity.pdf`, 2006. Retrieved Nov. 2013.

[25] B. Haeupler and F. Kuhn. Lower bounds on information dissemination in dynamic networks. *Distributed Computing*, pages 166–180, 2012.

[26] T.P. Hayes, J. Saia and Amitabh Trehan. The forgiving graph: a distributed data structure for low stretch under adversarial attack. *Distributed Computing*, 25.4:261–278, 2012.

[27] M. He, G. Tang, and N. Zeh. Orienting dynamic graphs, with applications to maximal matchings and adjacency queries. In *ISAAC*, pages 128–140, 2014.

[28] M. Henzinger, S. Krinninger and D. Nanongkai. Sublinear-Time Maintenance of Breadth-First Spanning Tree in Partially Dynamic Networks. In *Proc. ICALP*, pages 607–619, 2013.

[29] R. Ingram, P. Shields, J. E. Walter, and J. L. Welch An asynchronous leader election algorithm for dynamic networks. In *Proc. IPDPS*, pages 1–12, 2009.

[30] T. Kopelowitz, R. Krauthgamer, E. Porat, and S. Solomon. Orienting fully dynamic graphs with worst-case time bounds. In *Proc. ICALP*, pages 532–543, 2014.

[31] A. Korman. General compact labeling schemes for dynamic trees. *Distr. Comput.*, 20(3):179–193, 2007.

[32] A. Korman. Improved compact routing schemes for dynamic trees. In *Proc. PODC*, pages 185–194, 2008.

[33] A. Korman and D. Peleg. Labeling schemes for weighted dynamic trees. In *Proc. ICALP*, pages 369–383, 2003.

[34] A. Korman and D. Peleg. Compact separator decompositions in dynamic trees and applications to labeling schemes. *ACM Trans. Alg.*, pages 313–327, 2007.

[35] A. Korman and D. Peleg. Dynamic routing schemes for graphs with low local density. *ACM Trans. Alg.*, 4(4), 2008.

[36] A. Korman, D. Peleg, and Y. Rodeh. Labeling schemes for dynamic tree networks. *Theory of Computing Systems*, 37(1):49–75, 2004.

[37] L. Kowalik. Adjacency queries in dynamic sparse graphs. *Inf. Process. Lett.*, 102(5):191–195, 2007.

[38] L. Kowalik and M. Kurowski. Oracles for bounded-length shortest paths in planar graphs. *ACM Trans. Alg.*, 2(3):335–363, 2006.

[39] F. Kuhn, N. Lynch, and R. Oshman. Distributed computation in dynamic networks. In *Proc. STOC*, pages 513–522, 2010.

[40] F. Kuhn, T. Moscibroda, and R. Wattenhofer. Local computation: Lower and upper bounds. *CoRR*, abs/1011.5470, 2010.

[41] N. Linial. Locality in distributed graph algorithms. *SIAM J. Comput.*, 21(1):193–201, 1992.

[42] O. Michail, I. Chatzigiannakis, and P. G. Spirakis. Naming and counting in anonymous unknown dynamic networks. In *Proc. SSS*, pages 281–295, 2013.

[43] M. Mitzenmacher. On the analysis of randomized load balancing schemes. *Theory of Computing Systems*, 32(3):361–386, 1999.

[44] S. Muthukrishnan and R. Rajaraman. An adversarial model for distributed dynamic load balancing. In *Proc. SPAA*, 1998.

[45] M. Naor and L. Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995.

[46] C. Nash-Williams. Decomposition of finite graphs into forests. *J. London Math. Soc.*, 39(1):12, 1964.

[47] O. Neiman and S. Solomon. Simple deterministic algorithms for fully dynamic maximal matching. In *Proc. STOC*, pages 745–754, 2013.

[48] D. Peleg. *Distributed Computing: A Locality-Sensitive Approach*. SIAM, 2000.

[49] D. Peleg and E. Upfal. The token distribution problem. *SIAM J. computing*, 18(2):229–243, 1989.

[50] G. Pandurangan and A. Trehan. Xheal: Localized Self-healing using expanders. In *Proc. PODC*, pages 301–310, 2011.

[51] G. Pandurangan, P. Robinson and A. Trehan. DEX: Self-Healing Expanders. In *Proc. IPDPS*, pages 702–711, 2014.